Master's thesis

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

Eivind Bjørnebøle

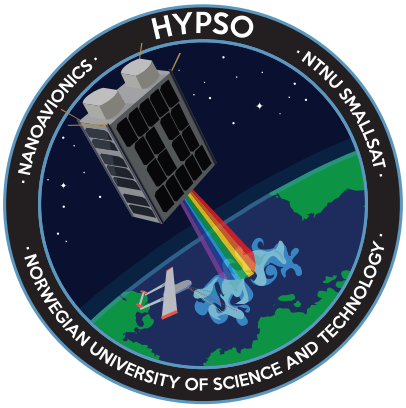
Reconstruction of Compressive Sensed Hyperspectral Images by Deep Convolutional Neural Network

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
Co-supervisor: Jon Álvarez Justo
June 2022

NTNU
Norwegian University of Science and Technology

Eivind Bjørnebøle

Reconstruction of Compressive Sensed Hyperspectral Images by Deep Convolutional Neural Network



Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
Co-supervisor: Jon Álvarez Justo
June 2022

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

NTNU
Norwegian University of
Science and Technology

Abstract

The Hyperspectral Smallsat for Ocean Observation (HYPSO) mission is an earth observational satellite for detecting algae blooms along the coast of Norway. The satellite is based on the CubeSat standard, making it small and affordable compared to conventional satellites. HYPSO-1 has a hyperspectral camera onboard, which captures hyperspectral data cubes. The imager scans the scene and sends the hyperspectral image to the ground station. The files being transmitted get large and thus take a long time to transfer from space. One way to reduce the amount of data being transferred is through the sampling technique of compressive sensing. This technique requires reconstruction, which has long been done with the help of optimization or iterative algorithms. Though these algorithms make good reconstruction results, they take a long time and often require handcrafted priors for optimality. The field of deep learning has grown in size and has found its way into compressive sensing reconstruction. Convolutional neural networks have shown state-of-the-art performance in image classification. The DeepCubeNet is a convolutional neural network, with a U-Net architecture, for reconstructing compressive sensed hyperspectral images. In this thesis, the DeepCubeNet is used to reconstruct hyperspectral images with up to 80 % discarded data. By training the U-Net with existing hyperspectral datasets like ICVL and Aviris, the model can reconstruct hyperspectral images with *PSNR* of 34 dB and *SSIM* of 0.887 taken of earth observational scenes.

Samandrag

Hyperspectral Smallsat for Oacen Observation (HYPSO) er ein satellitt for jord observasjonar, meint for å detektera algar langs kysten av Noreg. Satellitten er bygd på CubeSat standarden, noko som gjer den liten og rimeleg samanlikna med konvensjonelle satellittar. HYPSO-1 er utstyrt med eit hyperspektralt kamera, som fangar hyperspektrale data kubar. Kamerasystemet skannar området under seg og sender det hyperspektrale biletet til bakkestasjonen. Det er ofte store filer som skal sendast frå rommet, og overføringa tar tid. Ein måte å redusera mengda av data som skal sendast er igjennom ein samplingsmetode kalla compressive sensing. Denne metoden er krevjar ein rekonstruksjon av den komprimerte dataen. Rekonstruksjonen har lenge blitt utført ved hjelp av optimerings- eller iterative algoritmar. Desse algoritmane har vist gode rekonstruksjons resultat, men brukar lang tid og må ofte finjusterast for å oppnå optimalitet. Djup lærings feltet har ekspandert og funne sin veg inn i compressive sensing rekonstruksjon. Konvolusjonere nevrale nettverk har vist toppmoderande ytelse innanfor bilete klassifisering. DeepCubeNet er eit konvolusjonert nevralt nettverk, med ein U-Net arkitektur, som blir brukt til å rekonstruera hyperspektrale bileter. DeepCubeNet er brukt til å rekonstruera hyperspektrale bileter med opp til 80 % kasta data. Ved å trena nettverket med eksisterande hyperspektrale datasett, som ICVL og Aviris, klarte modellane å rekonstruera hyperspektrale bileter med *PSNR* på 34 dB og *SSIM* på 0.887 frå bileter av jord observasjonar.

Preface

This thesis concludes my five years at *Norwegian University of Science and Technology* (NTNU). The master thesis is the concluding part of the 2-year *Master of Science* (MSc) course *Electronic Systems Design*. The thesis is written in the specialization of signal processing and communication. The project involved research and implementation of deep neural networks for reconstructing hyperspectral images. The motivation is to learn more about the possibilities of deep neural networks for reconstructing compressive sensed hyperspectral images from CubeSats. The project provider is SmallSat Lab at NTNU. I am grateful for the opportunity to work in such a friendly environment as the SmallSat Lab, where there are always fellow students or professors to help or discuss issues. I see the master thesis as my final milestone before walking into the sanbox of endless opportunities called adulthood. I will now take off my academical-hat and step aside for a bit, but it will always lie here next to me. This thesis has though me a lot about neural networks, hyperspectral images, and compressive sensing, but it has also though me about the importance of having good people around you to always discuss if one hits upon an error message. If you have questions or inputs regarding the project, thesis or the codes, feel free to contact me.

Acknowledgement

I would like to thank my two supervisors Milica Orlandic and Jon Álvarez Justo for guidence and feedbacks during the project period. Further I would like to thank my fellow students and friends, especially Simen Netteland and Thomas Halvard Bolle for their guidance in Python or discussing other problems. Thanks to Edvard Birkeland, Samson Bergesen, Anders Brørvik, and Einar Avdem for support throughout the project period. I would also thank the guys at SmallSat Lab for their hospitality. Finally I would like to thank Kristin for being a supportive girlfriend.

Table of Contents

1 Introduction

Observing the surroundings has always been an essential part of the humans, watching for potential predators sneaking behind the next bush or seasonal changes to decide when to harvest the corps. In order to survive, it has been crucial to make the right decisions on the information that is currently available, and the spectre of observations has grown as technology allows it, such as sensors, imagers and satellites. Fast forward, the complex ecosystems are observed from every direction at a macro level through the lenses of satellites and at micro levels by on-ground sensors. This brings high expectations to the "observer", which must handle a significant amount of data. Computers can now process and analyze data streams at an ever-growing pace.

The demand for high-quality images and sensor data is essential for analyzing and observing abnormalities and environmental changes, such as toxification of drinking water or land soils or melting of glaciers [1]. The sensors are part of remote sensing applications, where the sensing device observes the scene at a distance. Data collected from these remote sensing applications must be transmitted to a shared database for analysis. This comes at the cost of transmitting power, which the sensing stations often lack due to their harsh and challenging environments controlled by sun cells and batteries. The transmission takes time, so the data is compressed before sending it. Data compression is a trade-off between losing information and keeping compression rates high. Thus making a need for high performing processing methods, such as compressive sensing, a sampling technique that uses much lower samples than classic sampling while keeping the same quality [2].

Compressive sensing (CS) has found its way into many image processing applications. From medicine to earth observations, all have turned their attention to the next data acquisition stage. The ever-growing demand for faster transmitting speeds and more extensive data files has long been bounded by the Nyquist theorem and compression standards for keeping quality across the data pipeline. Compressive sensing makes it possible to compress signals at a more significant rate than bounded by the Nyquist theorem while still keeping the quality high [2]. The process is possible due to the behaviour of the signals in a transform domain and by using reconstruction methods to recover the original data. This makes compressive sensing useful for extensive data demanding systems such as hyperspectral images (HSI), which are images that contain more spectral bands than regular three-banded RGB (red, green, blue) images. The extra spectral information is what makes HSIs unique when it comes to remote sensing. The bands are collections of wavelengths, often outside the human vision, representing spectral reflectance from substances and objects. Spectral reflectance is the characteristic of light when reflected from a surface [3].

Compressing sensing methods involve everything from sampling devices to reconstruction algorithms. Until recent years, the reconstruction methods have been performed with sparse optimization algorithms, which have proven high-quality reconstructions. Methods such as *Orthogonal Matching Pursuit* [4], TwIST [5] has shown promising reconstructions. Unfortunately, these methods tend to have slow reconstruction times. Nevertheless, researchers have worked on methods for faster reconstruction times and no need for fine-tuning parameters with the help of artificial intelligence (AI) or deep learning (DL). This has resulted in reconstructions achieving qualities that compete with state-of-the-art optimization algorithms [6, 7, 8].

Hyperspectral imaging has long been accomplished by scanning methods like push broom- and whisk broom- imagers. These have delivered high spectral- and spatial resolution [9]. However, they suffer from bad temporal resolutions. Temporal resolutions set limits for how fast the scene being captured can change. Snapshot imagers, on the other hand, are not as bounded by the temporal resolution. These capture the HSI in a less time consuming manner, by using liquid-crystal cells in front of the senor as a dispersive element [10, 11].

A significant candidate for earth observations is satellites. There has been a new contribution in satellite design in the last decade, where size has been the main topic. The CubeSat standard has allowed smaller teams to reach space, like SmallSat Lab at NTNU.

The HYPSO (HYPer-spectral Smallsat for ocean Observation) CubeSat is a project developed by the Small Satellite Lab at NTNU. The platform is based on the CubeSat structure, supporting the primary payload of one hyperspectral camera and one RGB camera. The mission of the HYPSO-1 is to observe ocean colour and detect harmful algal blooms [12]. The satellite is in low-earth-orbit and thus has a fast revisiting time, which is the time between the satellite passing over an area and when it passes over the same area next time,

making it suitable for Earth observation. The hyperspectral camera can capture spectral bands outside the human vision and provide information about the algae blooms that are are other ways difficult to detect. In 2019 algae bloom killed 10 000 tons of salmon along the coast of Norway [13]. Developing a system for observing and detecting harmful algae blooms in the coastal environment help the aquaculture industry protect itself against the potential extinction of its salmon population. With this system in full operation, the Norwegian aquaculture can keep exporting over 13 billion meals globally (numbers from 2020) [14].

Compared to higher-end commercial satellites, the tiny design limitations of the CubeSats set the bar for the components and payload on-board. These limitations are a sound basis for creative solutions and new research to arise. On-board processing power and power distributions are critical factors for operating the satellite, and these are resources for transmitting power and payload operation. Transferring data can be a time-consuming task for distances from low-earth-orbits back to the ground stations. Thus the amount of data should always be limited to what is necessary. This is where compressive sensing comes in. Compressive sensing limits the data being collected at the sensor, and thus data being transmitted down to the ground station is reduced. This thesis aims to explore compressive sensing for hyperspectral imagers to minimize the data being transmitted from satellites and find a CS recovery method that can be used for the next generations of CubeSats. Despite the amount of work in compressive sensing reconstruction, few studies have focused on satellite hyperspectral imagers. One potential approach is to use snapshot imagers with liquid crystal phase retarders, which can capture HSIs faster than scanning methods. Using liquid crystals as dispersive elements allows for immediate adoption of compressive sensing, as the spectral information is modulated according to the CS theory [10]. Using this sampling as a basis, the reconstruction method can be designed. A reconstruction method used in CS HSIs is to train a deep learning model called U-Net. DeepCubeNet [6] is an already developed U-Net, the initial network for exploration and testing in this thesis.

In order to accomplish the thesis's aim of reconstructing CS HSIs, the thesis gives an overview of current state-of-the-art methods. The reconstruction is done with the help of deep learning, a method within machine learning. The network structure will be an adaptation of the convolutional neural network (CNN) called U-Net. The network is trained with datasets of hyperspectral images and tested with state-of-the-art hyperspectral data such as Cuprite and Salinas. The reconstructions are also tested on other less used datasets like the ones provided by *The Interdisciplinary Computational Vision Laboratory* (ICVL) [15] and *Airborne Visible/Infrared Imaging Spectrometer* (AVIRIS) [16].

The proposed solution consists of a compressive sensing reconstruction method that utilizes deep learning for training on existing datasets. A U-Net architecture inspired by DeepCubeNet [6] is used for training and reconstruction. The compressive sensing is inspired by CS-MUSI [17], with a different approach to the spectral modulation of the HSI in the sampling process. An extra emphasis is given that the compressive sensing is done on already sampled HSIs from datasets and not applied directly in the acquisition process of the images. This is beyond the project's scope, which is focused on the reconstruction of CS HSIs.

The remainder of this work is organized as follows. Section 2 introduces the background theory used in this thesis and further gives an overview of previous work in the field of CS, HSIs and reconstruction methods. Section 3 presents the system description, explaining the network structure, training and data pre-processing. Section 4 provides the results and analysis of the thesis with different models that are trained. The results are then discussed and compared to previous works in the field. Section 5 concludes the thesis by discussing the performance of the proposed system, the advantages and disadvantages, as well as proposing further work.

2 Theoretical Background

This Section will give the reader the groundwork for understanding the theoretical process behind the implementations and decisions throughout the thesis. The goal of this section is to cover, the topics concerns sensing of the data namely compressive sensing, deep learning with convolutional nets, and hyperspectral images. The Background Section will first explain compressive sensing theory with introduction about signal sparsity. Then the optimization problem of compressive sensing will be presented followed by a gerneral introduction to deep learning and convolutional neural networks, with the U-Net architecture being the main structure. After this, hyperspectral images will be considered followed by quantitative image quality metrics for reconstruction validation. At the end, a brief introduction to the MUSI hyperspectral imager will be given.

2.1 Compressive Sensing Theory

Compressive sensing (CS) is a signal processing technique consisting of using lower samples than standard sampling theory while keeping acceptable quality of the final representation of the data. The theory of CS states that a signal can be reconstructed using a small set of samples randomly acquired if the signal is sparse in a certain transform domain [2]. In the field of signal acquisition and processing, the physical processes and parameters that are most interesting to capture are continuous while their measurements are discrete. The continuous signals are hence compressed by sampling the infinite set of points with a discrete finite set of points. Via Nyquist sampling, the continuous waveform is recovered and the Shannon-theorem allows a lossless recovery of the original analog signal. Compressive sensing in the acquisition process is a linear operation where the reconstruction of the samples can be found by solving a system of linear equations. The original signal is being sampled according to the Nyquist theorem, which states that in order to ensure reconstruction the sampling rate must be twice the highest frequency, in a given domain is $\mathbf{f}$ and contains N samples:

$$\mathbf{f}_{Nx1} = \begin{bmatrix} f_{(1)} \\ f_{(2)} \\ \vdots \\ f_{(N)} \end{bmatrix}. \tag{1}$$

A transform matrix Ψ is used to transform the signal $\mathbf{f}$ into a transform domain of choice. Matrix Ψ has dimensions $N\ x\ N$ where N is the number of samples in the original signal $\mathbf{f}$. The transform domains that are commonly used for matrix Ψ are the *Discrete Cosine Transform* (DCT), *Discrete Fourier Transform* (DFT), *Hermite Transform*, *Polynomial Fourier Transform* or the *Discrete Wavelet Transform (DWT)* [2, 18]. Matrix Ψ is given as follows:

$$\Psi_{NxN} = \begin{bmatrix} \psi_{(1,1)} & \psi_{(1,2)} & \cdots & \psi_{(1,N)} \\ \psi_{(2,1)} & \psi_{(2,2)} & \cdots & \psi_{(2,N)} \\ \vdots & \vdots & \ddots & \vdots \\ \psi_{(N,1)} & \psi_{(N,2)} & \cdots & \psi_{(N,N)} \end{bmatrix}. \tag{2}$$

The signal $\mathbf{f}$ can be represented in a transform domain as $\mathbf{x}$, then $\mathbf{f}$ can be expressed as:

$$\mathbf{f} = \Psi\mathbf{x}, \tag{3}$$

where $\mathbf{x}$ has also dimensions N x N. The random sampling in compressive sensing is performed using a measurement matrix Φ. The measurement matrix has dimensions $M\ x\ N$, where M is the number of samples of the compressed signal $\mathbf{f}$, and $M \ll N$ [4]. The matrix Φ is given as follows:

$$\Phi_{MxN} = \begin{bmatrix} \phi_{(1,1)} & \phi_{(1,2)} & \cdots & \phi_{(1,N)} \\ \phi_{(2,1)} & \phi_{(2,2)} & \cdots & \phi_{(2,N)} \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{(M,1)} & \phi_{(M,2)} & \cdots & \phi_{(M,N)} \end{bmatrix}. \tag{4}$$

The matrix Φ must be designed to give an unique representation of the signal $\mathbf{x}$ from the measurements. Several measurement matrices are available for CS applications, where the Gaussian matrix is the most used one [2]. The variables in the Gaussian matrix are randomly chosen according to a normal distribution. Other highly used matrices are the Bernoulli matrices, where the variables are between +1 and -1 and have the same probability of appearing, and partial random Fourier matrices [19].

The M measurements from the signal $\mathbf{f}$ are collected in the vector $\mathbf{y}$, which is the random projection of $\mathbf{f}$ over Φ:

$$\mathbf{y} = \Phi\mathbf{f}. \tag{5}$$

Eq. (3) can now be combined with Eq. (5) which gives:

$$\mathbf{y} = \Phi\Psi\mathbf{x}, \tag{6}$$

where the product of Φ and Ψ is also denoted as the dictionary matrix $\mathbf{A}$, which is called the sensing matrix [4]. Eq. (6) can then be rewritten as:

$$\mathbf{y} = \Phi\Psi\mathbf{x} = \mathbf{A}\mathbf{x}. \tag{7}$$

The figure below shows the concept of compressive sensing. The vector $\mathbf{x}$ is the sparse representation of the signal $\mathbf{f}$ in a transform domain. The white spots are zero values while the colors represents the non-zero coefficients. The matrix Ψ is the orthogonal basis transform matrix, thus controlling the transform domain of the linear system. Φ is the measurement matrix. The vector $\mathbf{y}$ is the measurement vector obtained after CS.

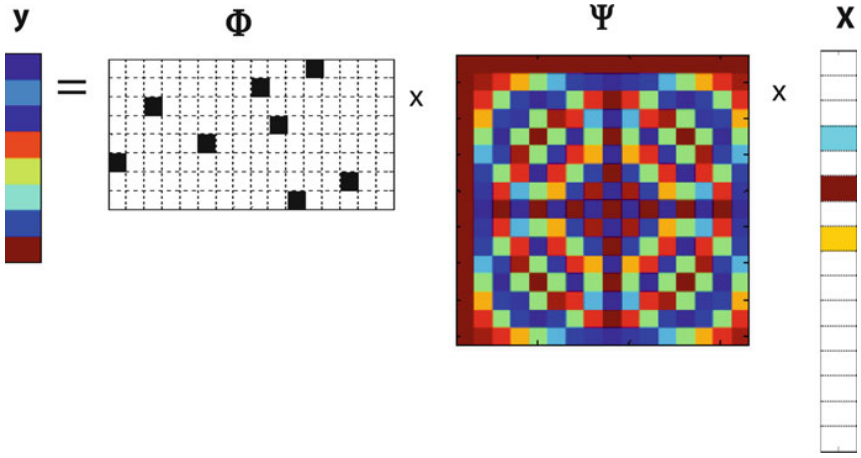


Figure 1: Compressive sensing as a concept, with the white squares being zeros [2].

In Eq. (7) the number of unknowns N is greater than the number of measurements M and the equation is an undetermined system of linear equations, meaning that there are infinitely many solutions [19]. An underdetermined system of linear equation is a mathematical expression that has to be approximately solved [20]. Compressive sensing algorithms allow for this undetermined system of linear equations to be solved, often using the sparsity constraint. With a properly chosen transform basis, the signal can be represented as sparse, which opens for accurate reconstructions. A signal is sparse if most coefficients are zero in the transform domain representation of the signal [21]. Mathematical algorithms are used in compressive sensing for error minimization. Algorithms which have been popular resolve l_1-minimization for finding the minimum l_1-norm solution to the underdetermined system of equations [22], thus finding the sparsest solution. l_1-minimization algorithms for finding the sparsest solution are *Basis Pursuit* (BP) [23] and the greedy algorithms with the *Orthogonal Matching Pursuit* (OMP) being the most popular [24, 2].

Sparsity

The sparsity of the signal $\mathbf{x}$ is defined by the number of non-zero coefficients in the transform domain. The signal is said to be κ-sparse if there are κ-non-zero coefficients in the transform domain [21]. Most real applications signals can be considered sparse when they are represented in the proper domain. Looking back at Eq. (7) with the sparsity property; most of the samples of $\mathbf{x}$ are zero, which then reduces the dimension of the problem. The l_0-norm defines the number of non-zero elements in a vector:

$$\|\mathbf{X}\|_0 = card\{supp(x)\} \leq \kappa. \tag{8}$$

If $\kappa \ll N$, $\mathbf{x}$ is considered sparse, and thus a reconstruction method can be used to give an approximate solution the linear system of equations.

Restricted Isometry Property

The *restricted isometry property* (RIP) is one of the two important conditions to be met for a reconstruction to be successful, the other is the incoherence property which will be described after. It is used to describe approximately orthonormal matrices. Orthonormal matrices hold the relationship that the inverse of the matrix is equal to the transposed of the matrix: a matrix $\mathbf{O}$ is othonormal if $O^{-1} = O^T$. The relationship between Φ and Ψ is of importance in CS [25]. The matrix Φ satisfies the RIP if

$$(1-\delta)\,\|x\|^2 \leq \|\Phi x\|^2 \leq (1+\delta)\,\|x\|^2, \tag{9}$$

holds for the vector $\mathbf{x}$ with κ nonzero coefficients or less [26].

Incoherence Property

The other important condition to be met for successful reconstruction is the incoherence property. This is the mutual coherence between the measurement matrix Φ and transform domain matrix Ψ. The property gives a measure of the similarity between the two matrices. The maximum similarity needs to be as low as possible in order to reconstruct the signal. Larger incoherence means fewer samples required for good reconstruction [27]. The incoherence of matrix Φ and Ψ is expressed by $\mu(\Phi,\Psi)$:

$$\mu(\Phi,\Psi) = \sqrt{n}\max_{i,j}|\langle\varphi_i,\psi_j\rangle|. \tag{10}$$

Here Φ_i is the i-th row in matrix Φ and ψ_j is the j-th column in matrix Ψ. The coherence measure, μ is expressed in the range between 1 and $\sqrt{n}$ [21].

2.2 Optimization Problems

In computer science, an optimization problem is the question of finding the best solution of all possible solutions [28]. As described earlier, in order to solve Eq. (7) one can search for the sparsest solution. The search is done by minimizing the l_0-norm of $\mathbf{x}$:

$$min\,\|x\|_0 \text{ subject to } \mathbf{y} = \mathbf{A}\mathbf{x}. \tag{11}$$

This is a non-convex optimization problem, which replaces l_1-norm by l_p-norm [25] where, the solution is found by demanding searches over the subset of columns in the CS-matrix $\mathbf{A}$ [2]. All sparse vectors $\mathbf{x}$ with κ samples are needed to be searched over. The κ-positions of entries are from the set $\{1,2,...,N\}$. The total number of κ-position subsets are $\binom{N}{\kappa}$, this yields an exponential order of complexity. For example if one has $N = 256$ and $\kappa = 6$ there will be over 10^{11} systems to solve. Because of this complexity isn't scalable

and one gets a exponentially growing number of systems to solve, one use the closest convex l_1-norm of the transform instead [2].

$$min \|\mathbf{x}\|_1 \text{ subject to } y = \mathbf{A}\mathbf{x}. \tag{12}$$

Vector norms are mathematical expressions of distances between vectors. A vector $\mathbf{x}$ with i elements is given in the p-norm as [29]:

$$\|\mathbf{x}\|_p = \left(\sum_i |\mathbf{x}_i|^p \right)^{1/p}. \tag{13}$$

This enables the use of linear programming to solve the optimization problem. The l_1-norm is given by the following expression [2]:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{N} |\mathbf{x}_i|. \tag{14}$$

2.3 Machine Learning

Machine learning algorithms builds models established on sets of sampled data, called datasets, and makes predictions and decisions based on the learned patterns in the datset, without being directly programmed to do so [30]. The learning can be supervised, by using labeled data, or unsupervised, which learns patterns in unlabeled data. The end goal of a machine learning algorithm can be to cluster datasets, classify data or predict data [21].

2.4 Deep Learning

Deep learning is a machine learning branch in which features are automatically extracted from data. Deep learning falls into the category of end-to-end learning, which concerns about learning a mapping between a given input and the desired output automatically. The advantage of this branch is the continuous improvement of the network with increasing data [31]. A comparison of classical machine learning methods and deep learning methods is given in Fig. 2.

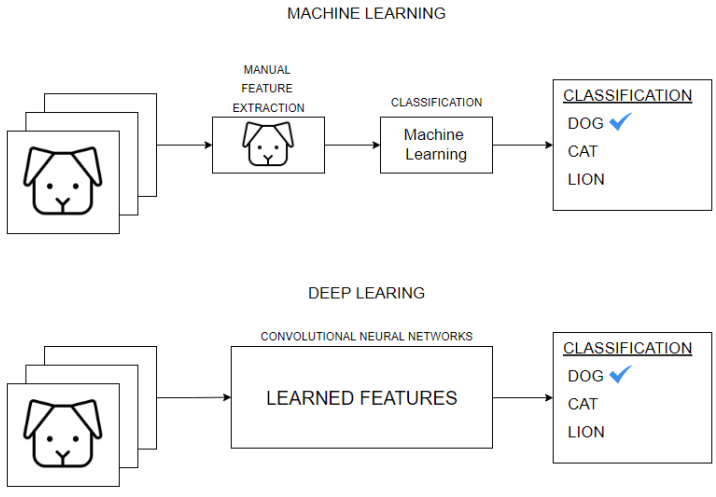


Figure 2: Difference between classical machine learning methods in classifying animals and deep learning methods.

The figure above shows the advantage of automatically feature extraction with the use of deep learning methods, by classifying images. Whereas the classical method relies on manually chosen features, in order for correct classification [32].

2.5 Neural Networks

In 1980 the computer scientist Kunihiko Fukushima made the first artificial neural network called Neocognitron. This where a mathematical model of neural networks, and shared the same characteristics of today's deep convolutional neural networks, with multi-layered perceptron structure, convolution, nonlinear dynamical nodes, and max pooling operation being some of them. The network was inspired by Huble and Wiesel's discoveries of the structure of the visual nervous system [33]. Today the convolutional neural network (CNN) is a popular deep learning design which is specialized in the work of dimensional data, such as images. CNNs have been used in image segmentation applications, for classifying structures in images, due to their ability to learn image features [31]. Popular applications of CNNs are *ResNet* [34], *VGG* [35], *Inception* [36] and *AlexNet* [37].

Layers

CNN's "building blocks" are their layers. A CNN consists of multiple layers, and each layer can learn different features of an image. The layers are arranged accordingly: input layer - hidden layers - output layer, and define filters for the training images at different resolutions [21]. The output of each layer is the input of the next. Fig. 3 shows an overview of the layers in a classical CNN.

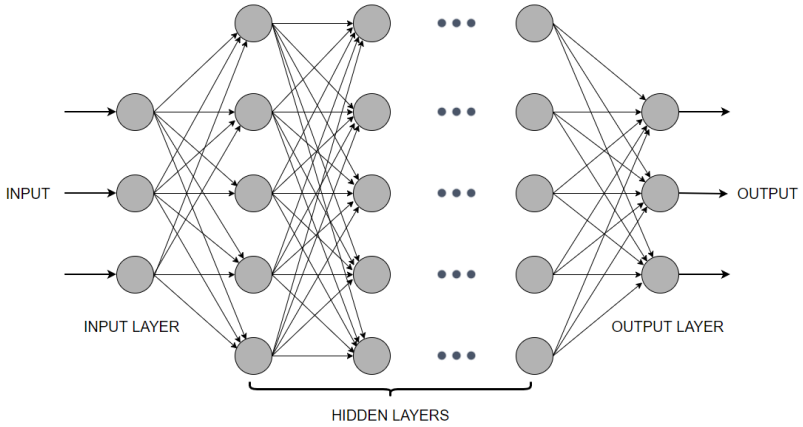


Figure 3: CNN with its layer categories

The feature extraction gets more complex the deeper into the hidden layers one go and this is the advantage of neural networks. The operation which makes the layers learn features is the convolutions.

Convolution

The word "convolutional" in convolutional neural networks implies that the network employs a mathematical operation called convolution on the data. The CNNs use convolution in at least one of its layers instead of general matrix multiplication. The operations are linear, where a set of weights are multiplied by the input. The method was designed for the use of multi-dimensional inputs, thus the multiplication is done on multi-dimensional input and multidimensional weights, called kernels or filters [21]. There is a size difference between the input data and the filters, where the filters are smaller in size. The reason for this is to let the filter be multiplied by the input data multiple times at different input points. The filter is systematically sliding over the image, left to right and top to bottom, and detecting features along the image. Fig. 4 shows an illustration of the convolution operation where the input image with dimensions 4x4 is convolved with a kernel or filter with dimensions 3x3 which gives a output of dimensions 2x2 called the feature maps.

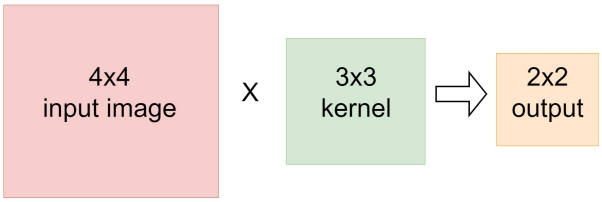


Figure 4: A basic illustration of the convolution operation with a 4x4 pixel input image convolved with a kernel of size 3x3 and the corresponding output of 2x2 pixels.

Max-pooling

Max-pooling is the operation that down-samples the data into smaller dimensions. The down-sampling of data is important for the networks for learning features in the sub-regions of the down-sampled input [21]. Max-pooling helps to prevent overfitting, which means that the model learns too many details in the training data and makes the model perform worse on unseen data [21]. Max-pooling also reduces the computational cost by reducing parameters by progressively reducing the dimension size of the representation. Fig. 5 shows the max-pooling operation on a 4 x 4 matrix and a filter size of 2 x 2. In order to not overlap regions the stride is set to 2. The stride can be visualized in Fig. 5 as the filter is jumping 2 steps in-between the max pooling as seen in the four color tiles. The output is then a 2 x 2 matrix with each of the maximum values from the initial regions.

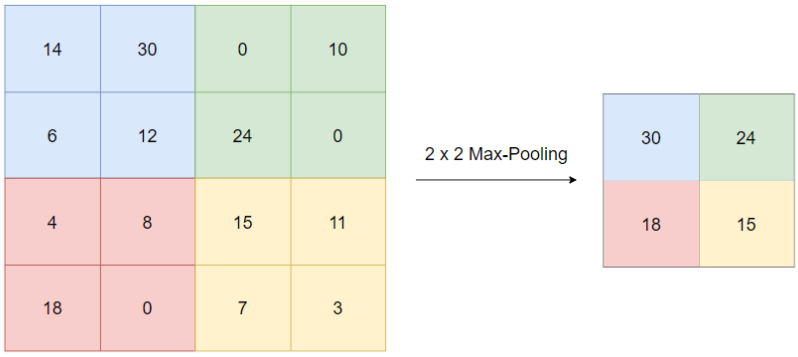


Figure 5: Max-Pooling operation with a filter size (pool-size) of 2 x 2.

Loss Functions/Cost Function

A cost function is used to calculate the cost which is the difference between the predicted value from the model on the validation data and the actual value. By using the loss function the gradients, which are used to update the weights, can be found. The cost is the average over all losses. The neurons in each layer process information by using a non-linear activation function. By propagating information through layers of neurons the network is gradually learning. Through a process called backpropagation, the weights are iteratively changed and by combining these, with input information the outputs can be motivated towards the expected outcome.

The operation of one single neuron in a network can be viewed as the data is multiplied by a random weight and added to a random bias. The result of this will be the input of the neuron. In order to change the result, bias and weight are found during training. Eq. (15) shows the input to one neuron:

$$input = data \times weight + bias \tag{15}$$

The neuron represents a non-linear activation function, and this function transforms the input into a value within a specified range related to the chosen activation function. The output is compared to the expected output and the difference is measured with a cost function. The difference is sent back to the beginning and used to update the weight and the bias, this is known as backpropagation in the network. This procedure is repeated until the output is reasonably close to the expected output [21].

Optimization Functions

Training times can be reduced exponentially with the right optimization algorithm. In order for the neural network to reduce the losses, the optimizer changes network parameters such as weights and learning rates. When mapping the input to the output the optimization algorithm finds the weights that minimize the error of the mapping. During the training of the network, the weights are updated each epoch and the loss function is minimized. Choosing these weights is a complex task as the network can consist of millions of parameters, which sets the bar for the optimization algorithm's performance for the given task. There exist different optimization algorithms for giving the most accurate result possible and some of them are given below.

- *Gradient Descent*: the *gradient descent* algorithm is widely used in linear regression and classification algorithms, as well as in backpropagation in neural networks. The gradient descent optimizer is depending on the first-order derivative of the loss function and uses the information to calculate which direction to alter the weights such that the loss function can reach the minima. Using gradient descent as an optimizer is computationally simple, and implementation is straightforward. The downsides are that it might get stuck in a local minima, which happens when the gradient of the loss function is calculated to be zero at a point that is not corresponding to the global minima. Datasets can be large, and the gradient descent to calculate all gradients for the whole datasets can often take a substantially long time.
- *Stochastic Gradient Descent*: the *stochastic gradient descent* (SGD) optimizer is a variant of the gradient descent, which updates parameters more frequently in the model. On each training example, the parameters are updated, compared to the standard gradient descent where the parameters are updated after one cycle through the training data. This results in faster convergence, and fewer values to store hence requiring less memory. SGD has some downsides as well, with model parameters having high variance and learning rates are slowly reduced in order to converge at the same point as gradient descent.
- *Adaptive Moment Estimation* (Adam): is an extension of the *stochastic gradient descent* (SGD) for updating weights during training. Adam optimizer updates the learning rates of each weight individually compared to the SGD which maintains a single learning rate. It is widely used throughout the deep learning community, with its faster running time, minor memory requirements, and fewer adjustments than other optimization algorithms are required. Adam optimizers also have a downside, which is that they focus on faster computation times and thus generalize less than SGD.

2.6 U-Net

U-Net is a neural network architecture that is focused on segmentation. The name comes from the shape of the network layer structure that forms the shape of a U, as can be seen in Fig. 6. The U-shape divides the network into two sides, where the left slope is the encoder and the right slope is the decoder. The left side of the U-Net is referred to as the contracting part used to capture the context of the data, which follows the same architecture as a standard CNN. On the right side is the expansive path, used to capture the data context [38].

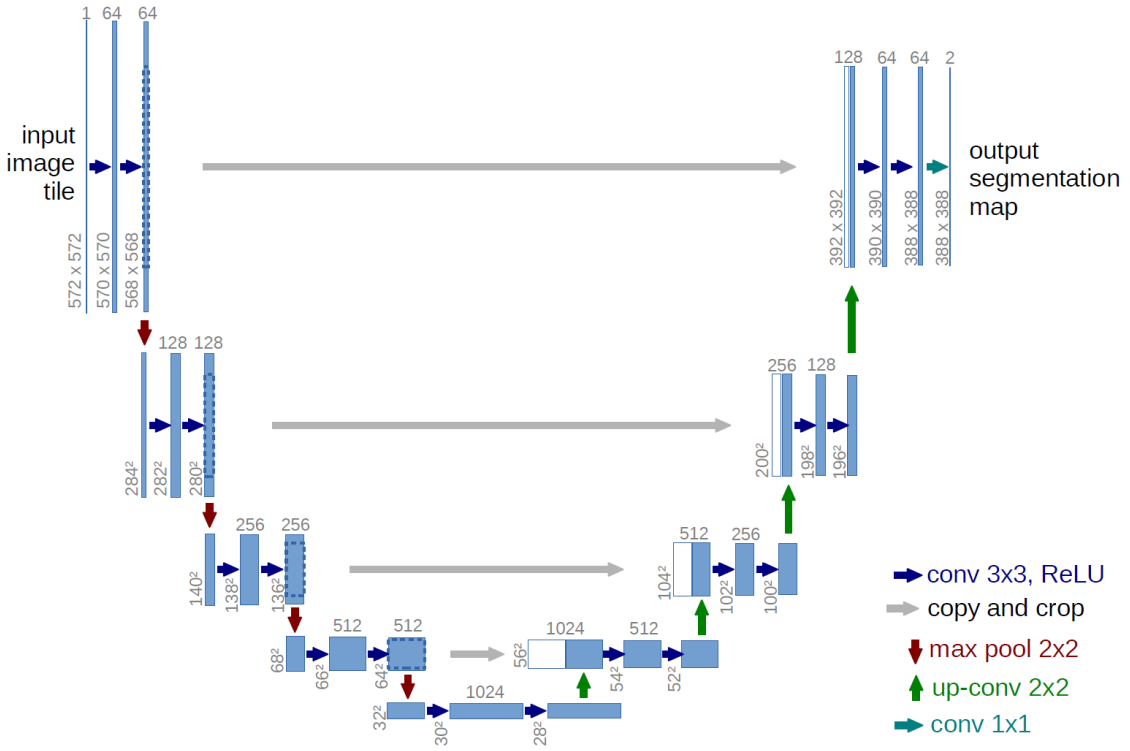


Figure 6: U-Net architecture, each blue rectangle corresponds to a feature map with multiple channels. The arrows represent different operations characteristic of the U-Net architecture [39].

The contractive path consists of convolutional blocks, and the amount of blocks is configurable. A skip connection or concatenation is applied to the outputs from each convolutional block, except the last block. The blocks are composed of two convolutional layers with a kernel size of 3x3. With a stride of 1 and a kernel of 3x3, the image height and width are reduced by 2 pixels for each convolution, which can be seen in the example in Fig. 6, where the highest block has reduced the image from 572 x 572 to 568 x 568 after the last convolution. Each convolution is followed by a ReLU activation function. The max-pooling operation is performed between each block, this has a 2x2 pool size and a stride of 2. The max-pooling is the operation which down-samples the data by half the pixel width and height, which is seen in the figure above where the input starts at 572 x 572 and ends up with a dimension of 32 x 32 at the bottom layer. The compensation is a doubling in feature map size where the last layer has 1024 feature maps and the upper layer has 64. This shows that the U-Net learns more features at lower dimensions than higher ones, and thus balances computational resources. The expansive path is similar to the contractive path only inverse. The data is being upsampled instead of downsampled, with the same configurations. The upsampling decreases the feature maps by 2. The skip connections or the concatenation doubles the number of feature maps. After the two convolutional layers with the same configuration as the contractive path, the feature maps are back to the original size. At the end of the expansive path, the output will be the upsampled image with an initial amount of feature maps [38]. The concatenation serves as an "information-keeper", which adds extra information from the decoder side to the encoder side that might be lost due to the down-sampling of the network. The concatenation is a way to combine the contracting path with the expansive path, this utilizes end-to-end training approaches and thus learning can increase [40].

2.7 Hyperspectral Remote Sensing

Remote sensing is the technique of long-range detection of objects or substances and natural phenomena by capturing information from a distant scene. Remote sensing can perceptually define a phenomenon or object by measuring the collected reflected electromagnetic wave. What makes this technique possible and further makes hyperspectral images useful is the fact that all objects or substances reflect different electromagnetic waves due to different features and environmental conditions. Hyperspectral imaging or spectroscopic

imaging is an imaging technique that captures spatial and spectral information of a scene. The information acquired in the acquisition process is stored in a three-dimensional datacube, where the spatial information is represented in the **x**- and **y**-axis. In contrast, the spectral information is represented in the **z**-axis or the depth of the cube. Fig. 7 shows a hyperspectral cube of Jasper Ridge dataset. The hyperspectral image is a representation of the characteristically electromagnetic waves reflected by substances and objects from the scene. Hyperspectral images have gained popularity in the area of remote sensing due to their potential to classify different phenomena, like the detection of mammal herds, seasonal variations and toxation of algae blooms [41]. Different techniques exist when capturing HSIs, where the most used are whisk broom, push broom and snapshot [9, 17]. Whisk- and push broom are scanning methods for capturing the spatial-spectral information, and snapshots are capturing the scene in one ”shot”. [3]

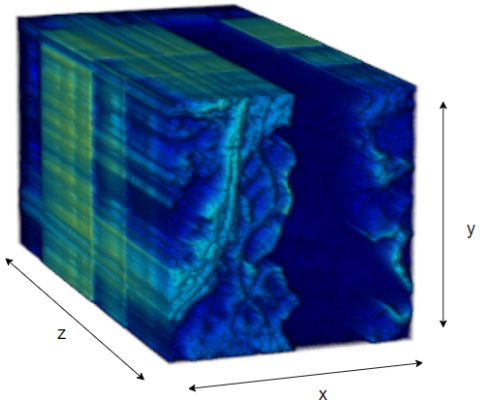


Figure 7: Hyperspectral data cube with spatial and spectral information. **x**- and **y**-axis are the spatial dimensions, while **z**-axis is the spectral bands dimension. The colors are not calibrated and are only for visualisation. Figure is made in MATLAB using the *Volume Viewer* app.

The spectral dimension (depth) contains the wavelength that the imager is capturing, this dimension is divided into sampled wavelengths called bands. If the hyperspectral imager is capable of capturing wavelengths between 400 nm and 700 nm with 100 bands, this will give each band a wavelength of 3 nm. For a snapshot imager the spatial resolution is defined by the number of pixels in the image detector, while for the scanning methods the spatial resolution is defined by the duration of the scanning. Looking at one individual pixel in the spectral domain reveals spectras, which is one pixel with all of its bands, shown in Fig. 8. This wave representation of a pixel makes it possible to see what object or signature the pixel represents in the scene being captured [3].

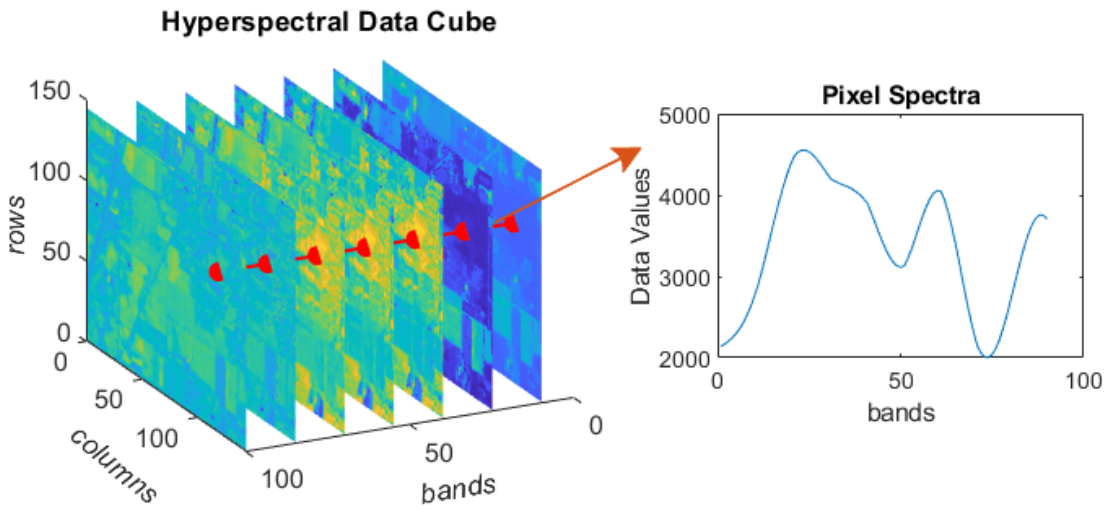


Figure 8: A hyperspectral cube separated by its different bands into spatial images. The spectra of one pixel is shown in the plot to the right [42].

2.8 Quantitative Image Quality Metrics

In order to quantify the quality of compression reconstruction, there are some useful metrics. These compare the reconstructed image to the original image. Three such metrics will be explained below.

Peak Signal-to-Noise Ratio (*PSNR*) and Mean Squared Error (*MSE*)

PSNR is used to quantify the reconstruction of a compressed image compared to the uncompressed version. The ratio concerns the maximum power of the signal and the power of the corrupted noise that affects the fidelity of the representation [43]. The *PSNR* is logarithmic using the decibel scale, and higher dB means better quality. Eq. (17) and (16) shows how the *PSNR* is calculated:

$$PSNR = 10\log_{10}\left(\frac{R^2}{MSE}\right), \tag{16}$$

where R is the maximum fluctuation in the image data type. If the image is 8-bit unsigned integer type data, the R will be 255. Eq. (17) shows the expression of mean square error (*MSE*), where M and N are the spatial dimensions of the images, and I_1 and I_2 are the two images being compared:

$$MSE = \frac{\sum_{M,N}[I_1(m,n) - I_2(m,n)]^2}{M * N}. \tag{17}$$

As with *PSNR* the *MSE* is a measurement for comparing the quality of image compressions. *MSE* is the cumulative squared error between the compressed and the original image, whereas *PSNR* represents a measure of the peak error.

Structural Similarity Measure (*SSIM*)

The *SSIM* is a perception-based model which measures degradation in structural information in images. The model does not measure absolute errors such as *MSE* and *PSNR*. The spectral information is that close pixels have strong inter-dependencies, which carries important information about the image scene like structures and objects [44]. *SSIM* is calculated from three measurements: luminance (l), contrast (c) and structure (s), and the weighted combination of these reveals the *SSIM* as seen in the equations below:

$$l(x,y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1}, \tag{18}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2}, \tag{19}$$

$$s(x,y) = \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3}. \tag{20}$$

With **x** and **y** being the two images to measure between. μ the averages, σ^2 the variances and σ the covariance. c_1 and c_2 are variables and c_3 is equal to $c_2/2$. *SSIM* can then be expressed as:

$$SSIM(x,y) = \left[l(x,y)^\alpha, c(x,y)^\beta, s(x,y)^\gamma\right], \tag{21}$$

where α, β, γ are weights.

2.9 Compression Ratio (CR)

Compression ratio, or *CR* for short, is the ratio between the compressed number of samples *M* and the original number of samples *N*. This states how much of the original bands that are left from the true image. This means that if $M = 30$ and $N = 150$, the *CR* is 30/150 = 0.2, i.e. 80% of the original samples are discarded in the acquisition.

2.10 MUSI

This subsection goes through the theory from the paper "*Compressive Sensing Hyperspectral Imaging by Spectral Multiplexing with Liquid Crystal*", by Yaniv Oiknine et al [10].

MUSI stands for *miniature compressive ultra-spectral imaging system* and utilizes a single liquid crystal (LC) phase retarder, which encodes only the spectral domain. By applying different voltages to the LC cell, the refracting index is modified which again changes spectral modulation. Due to the LC-retarder, the signal is multiplexing entirely in the spectral domain. Two polarizers are placed on each side of a liquid crystal cell. By changing the applied voltage on the LC cell, variations in the cell's birefringence occur, which again causes the refractive index to change. This controls the spectral transmission. [10]

Birefringence

Birefringence is a property of optics a material has. The material has a refractive index which depends on the orientation and polarization of incoming light. The quantification of a material's birefringence is often defined by the maximum difference between the refractive indices and the material effects [45]. Transparent objects are optically isotropic, this means that the index of refraction is equal in all directions throughout the crystalline lattice. The entering light is reflected at a constant angle in an isotropic crystal, and passes through it at a constant velocity. The light is not undergoing the effects of polarization caused by interactions in the electronic components inside the crystalline lattice.

Measurements

The methods of measurements can be divided into two subgroups; direct measurements and indirect measurements, such as multiplexed or coded measurements. Coded measurements have gained popularity in the use of spectroscopy even though they are needing post-processing and are more complex systems than direct measurements. Sensing matrices for the indirect measurements are following a non-zero off-diagonal element structure.

Sensing Matrix

The sensing matrix is orthogonal with N columns and N rows. The hypercube grid comprises N points and the direct- and indirect measurement systems thus need several measurement points, M, greater or equal to N. However, MUSI requires significantly fewer measurement points M, than hypercube points N. A compact cost-effective, single LC variable retarder is used with the theory of compressive sensing to sample the hypercube with a great amount of measurement point less than before.

In the approach of MUSI the spectral encoding is accomplished in the spectral domain alone, thus no need for spectral-to-spatial transformation. The modulator comprises a liquid crystal cell (LCC) and photosensor array. The CS theory is preserved in the specific design of the LCC. The form factor of such a sensor device can be a width of only a few millimetres. Which is a huge advantage compared to other methods of spectral measurements. These methods tend to be based on direct spectral measurement with narrow-band spectral scanning, and they are often optically less efficient due to a few cascaded spectral filters.

The M spectrally multiplexed images can be recovered using reconstruction algorithms. The spectral encoding is done by the LCC, where the refractive index is controlled by changing the voltage over it. By changing the refractive index different phase delays for different wavelengths are implemented, which leads to wavelength-dependent attenuation. The representation of the spectral encoding of the system as a function of the voltage applied is the system spectral matrix. Each pixel of the sensor array integrates the modulated light that is passing through the LCC. The spectral and spatial encoding are separate because each pixel is similarly spectrally encoded. The separate encoding further means that the spatial-spectral matrix can be modelled.

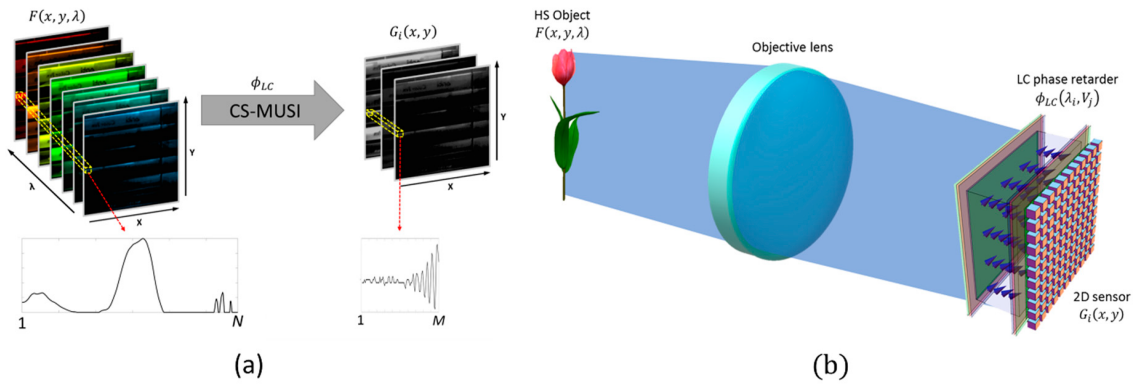


Figure 9: Illustration of MUSI acquisition process (a). MUSI optical scheme diagram (b) [17].

The CS-MUSI camera has a single liquid crystal (LC) phase retarder. This follows the same principles as described above (liquid crystal cell, spectral modulation). When the LC cell's optical axis is at 45° to two perpendicular polarizers, the spectral response of the phase retarder is given by:

$$\phi_{LC}(\lambda, V_i) = \frac{1}{2} - \frac{1}{2} cos\left(\frac{2\pi\Delta n(V_i)d}{\lambda}\right), \tag{22}$$

where the thikness of the cell is given by λ, $\Delta n(V_i)$ is the birefringence formed by the voltage V_i. The LC cell is applied either a sine or square wave with a frequency in the order of kHz. The spectral response map is measured in the calibration process and is showed in the figure bellow.

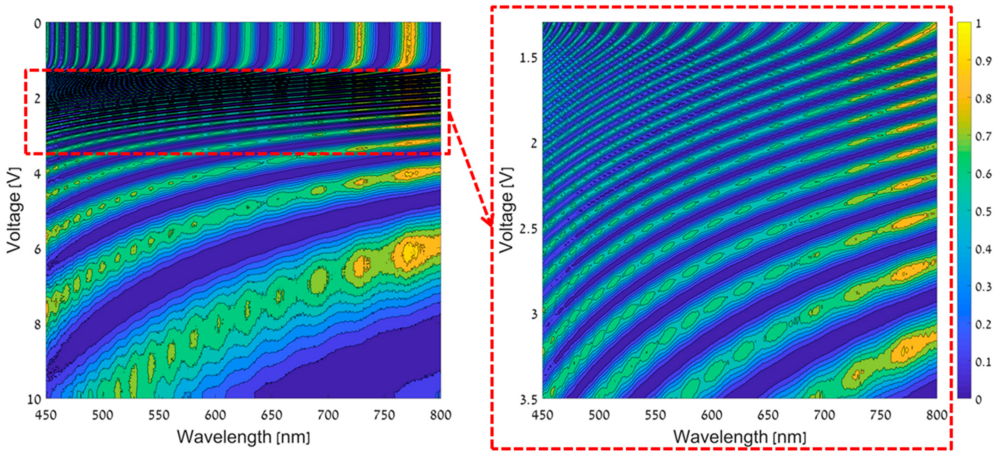


Figure 10: Spectral response map for the CS-MUSI with voltages from 0 to 10 volts at the left and a zoom-in at the voltages from 1.3 to 3.5 volts [10].

By selecting M rows from the response map in 10 a sensing matrix $\mathbf{A}$ can be found. The figure bellow shows the spectral sensing matrix with $M = 32$ and $N = 391$.

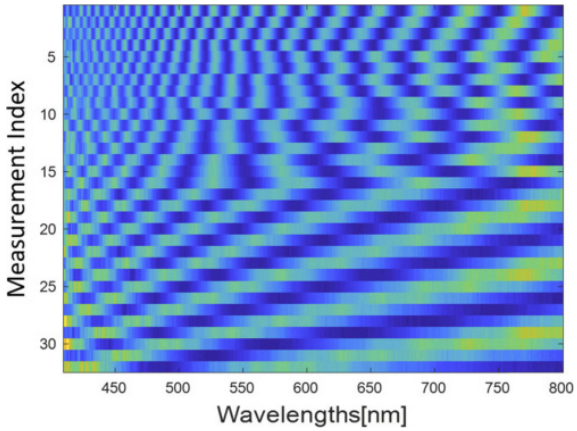


Figure 11: Sensing matrix $\mathbf{A}$ with 32 measurements (row) and 391 original bands (columns) [6].

2.11 Previous Work

Under this Section, previous research in compressive sensing, hyperspectral imaging and reconstruction methods are being provided. The aim is to give the reader an overview of the groundwork that was used to decide upon the methods proposed in this thesis.

Compressive Hyperspectral Imaging

Several methods have been proposed for capturing spatial information across many wavelengths. The push-broom spectral sensor is a hyperspectral imager that captures a spectral cube with one focal plane array (FPA) measurement per spatial line of the scene [9]. Spectrometers acquire hyperspectral data by scanning several zones linearly in proportion to the desired spatial and spectral resolution. Spectrometers are based on optical baseband filters. These filters are tuned in steps in order to scan the scene sequentially. The aforementioned acquisition techniques all share the same disadvantage: they are time-consuming and require considerable data storage. Thus, a different spectral imager called CASSI (coded aperture snapshot spectral imagers) exists. An advantage of the CASSI is that it preserves compressive sensing principles. The entire data cube is sensed with just a few FPA measurements [11]. Looking at the principles of CS, the random projections occur naturally due to the optical dispersion phenomena affecting coded aperture light fields as they traverse a prism before the imaging detector integrates these.

MUSI

MUSI stands for *miniature compressive ultra-spectral imaging system* and utilizes a single liquid crystal (LC) phase retarder, which encodes only the spectral domain. Changing the applied voltage on the LC cell, variations in the cell's birefringence, which again causes the refractive index to change. This controls the spectral modulation of the incoming light [10].

Reconstruction Methods for CS HSIs

ConvCSNet: A convolutional compressive sensing framework based on deep learning (2018):

Earlier works of compressive sensing reconstruction of hyperspectral images have suffered from blocking artefacts in the recovered images. Instead of recovering block by block, Lu, Xiaotong, et al. implemented a convolutional CS framework which sensed the entire image using a set of convolutional filters. Then the whole image was reconstructed from the linear convolutional measurements. The CNN in ConvCSNet performed both the convolutional sensing and the nonlinear reconstruction. The advantage of this framework is highlighted when comparing it with the substantial random sensing matrices and the slow convergence of the sparsity optimization reconstructions. Sensing the input image is done through the first layer by convolving the whole image with a set of random filters, followed by subsampling. The tiny filters are stored and use little storage space compared to the random matrices used in the sparse recovery methods. The authors claimed that their method substantially outperformed previous state-of-the-art methods in both visual quality and *PSNR* [8].

Hyperspectral image reconstruction using deep external and internal learning (2019)

Zhang, Tao, et al. presented a CNN-based channel attention reconstruction network developed to efficiently exploit the spatial-spectral correlation of HSIs. The reconstruction process consists of three steps: 1. Present a CNN-based reconstruction method for coded HSI to effectively combine deep external and internal learning. 2. Exploit the spatial-spectral correlation of the HSI by external learning. 3. Guarantee generalization ability and adapt itself to variant scenes by internal learning. They aimed to implement an efficient CNN-based method for coded HSI reconstruction learning the deep prior from an external dataset and internal input image, combining deep external and internal learning.

Their proposed systems were based on multiplexing a 3D cube into a 2D spatial sensor. This acquisition method previously sacrificed spatial resolution. And coding-based methods showed potential in overcoming the temporal and spatial resolution trade-off. The optical design elaborates coding-based techniques to encode the 3D HSI into a 2D compressive sensor. The bottleneck was now shifted to the reconstruction of the HSI. Due to the model-based methods relying upon carefully designed priors, learning-based methods were used, as they implicitly learned the priors from external datasets. The learning-based methods had a problem; however, they often attempted to brute-force the input and output mapping, ignoring the internal imaging model.

They suggested a convolutional neural network for coded HSI reconstruction, which learned deep priors for external datasets and spectral-spatial constraints from internal input image information. The method outperformed state-of-the-art, according to the authors. [46]

DeepCubeNet: reconstruction of spectrally compressive sensed hyperspectral images with deep neural networks (2019)

Gedalin, Daniel, Yaniv Oiknine, and Adrian Stern contributed to DeepCubeNet. The DeepCubeNet is set around the CS-MUSI and can reconstruct hundreds of spectral bands. The reconstruction is performed through a deep neural network consisting of two parts; a pseudo-inverse operation as an approximate solver and a U-net architecture where the 2D convolutions are replaced with 3D convolutions. The pseudo-inverse operation back-projects the CS-MUSI data from the compressed domain to the hyperspectral domain. This means transforming the CS-MUSI sensing matrix, which is the initial back-projection. It was introduced to prior information about the imaging system, which gave the network an initial guess, allowing it to converge to desired minima. In the U-net, all layers are 3D convolutions. The architecture exploits the spatial context of neighbouring pixels and the spectral correlation of neighbouring bands. The network is based on the acquisition of a compressive sensing system for hyperspectral imaging called CS-MUSI. DeepCubeNet claims to outperform previous state-of-the-art by at least 10 dB. Using pseudo-inverse projection as part of the network prevents overfitting, and the neural network gets prior knowledge of the physical measurement system. This knowledge reduces the complexity of the network, allowing a reduction in the number of parameters [6].

HyperMixNet: Hyperspectral image reconstruction with deep mixed network from a snapshot measurement (2021):

In [47], the authors Yorimoto, Kouhei, and Xian-Hua Han reconstructed the underlying HSI from a single snapshot image. Instead of conventional convolutional layers, HyperMixNet has integrated a MixConv block. This has the advantage of reducing the size of the reconstruction model, which in turn is handy when being embedded in the natural imaging system. The number of network parameters is decreased, and multi-level context is learned for more representative feature extraction. This outperforms the previous state-of-the-art methods in quantitative values, visual effect, and reconstruction model scale [47].

Reconstruction Methods in General

Deep generative adversarial neural networks for compressive sensing MRI (2019):

Mardani, Morteza, et al. focused on high diagnostic-quality image reconstruction from highly undersampled MR measurements. The authors propose a tandem network consisting of a generator, an affine projection operator, and a discriminator. The generator aims to create gold-standard images from the undersampled images. This is done using a deep residual network (ResNet) with skip connections to retain high-resolution information. The discriminator network consists of a multilayered convolutional neural network. The discriminator aims to distinguish fake images (from the generator) from real images (corresponding gold standards). The network's training is like playing a game with conflicting objectives between the discriminator and the generator. The generator aims to map the subsampled input image to a fake image that fools the discriminator into believing it is fed an actual image. The discriminator tries to score one for

the actual gold-standard image and zero for the fake. The perceptual quality was achieved with the ability to confirm diagnostics. The GANCS were significantly faster than the state-of-the-art methods [7].

Interesting approaches in the field of hyperspectral images

Super-Resolution

In "Deep hyperspectral prior: Single-image de-noising, inpainting, super-resolution" [48] Sidorov, Oleksii, and Jon Yngve Hardeberg, propose an approach to de-noising, inpainting and super-resolution of hyperspectral images by using properties of a convolutional neural network without training. The performance is comparable with previously trained networks and has the advantage of not being restricted by the available training data sets. The work extends the "deep-prior" algorithm to the hyperspectral imaging domain and 3D-convolutional networks. The algorithm is used in restoring hyperspectral images that suffer from noise, low dimensions and corruption.

Image-Fusion

In the article "Multispectral and Hyperspectral Image Fusion Using a 3-D-Convolutional Neural Network" [49] by Palsson, Frosti, Johannes R. Sveinsson, and Magnus O. Ulfarsson. The authors propose a method for using a 3D convolutional neural network to fuse multispectral and hyperspectral images to obtain a higher resolution hyperspectral image. The 3D CNN learns filters used to fuse the multispectral image with the hyperspectral image. The method is based on supervised learning and requires a hyperspectral target image. However, the author did not have the multispectral and hyperspectral pair and used spatial low-pass filtered and downsampled versions of the hyperspectral images instead. Experiments with simulated data show that the proposed method gives acceptable results.

3 System Description

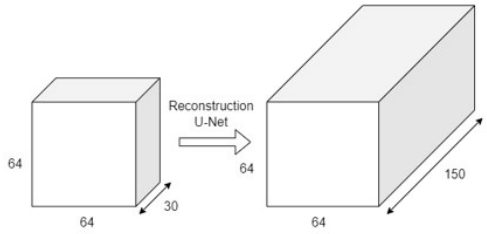


Figure 12: Illustration of the reconstruction of the sub-sampled cube.

The following Section addresses the steps that are followed throughout this project. First, the sensing of the data is presented here with transform domains like DFT and DCT. The different datasets used and their limitations are presented, as well as the training set preparations—followed by a preview of the U-Net architecture and its different components. The training process is further explained, and data types and tools will be highlighted at the end. When researching methods for compressive sensing Hyperspectral imaging systems, the goal was to find a creative solution that was inspired by the growing interest in deep learning in previous image applications. There is a few options for already developed methods for these kinds of systems which uses different structures of CNNs [50, 51], convolutional autoencoders [52] and MixConv [53] to name some of them. They all share the same advantage over the classical iterative methods like Orthogonal Matching Pursuit, Basis Pursuit, or Gradient Projection for Sparse Reconstruction [4], in terms of speed and no need for hyper-parameter tuning. Two methods stand out in particular when looking at imaging systems that could enable compressive sensing directly in the sampling process. These are the Coded Aperture Snapshot Spectral Imaging (CASSI) [11] and the Compressive Sensing Ultra-Spectral Imager (CS-MUSI) [17]. These are both in the category of snapshot hyperspectral imagers, meaning that the sensing times compared to scanning methods will decrease. CASSI has a more complex sensing strategy and uses a coded aperture, where the CS-MUSI only compress the image in the spectral domain. As a concept for a new hyperspectral imager for CubeSats, the CS-MUSI is the choice. Daniel Gedalin, Yaniv Oiknie and Adrian Stern, had great success with the combination of CS-MUSI and their deep learning reconstruction method using U-Net architecture [6]. They used the ICVL Hyperspectral Database [15], and with a compression ratio of approximately 0.1 (32 of 391 bands), they achieved *PSNR* results of 48 dB. That is over 10 dB better than state-of-the-art deep learning methods at that time. The results and documentation from DeepCubeNet and the CS-MUSI inspired this thesis goal of a new hyperspectral image reconstruction system for CubeSats. The first milestone would be to get results close to those of the original paper, and the second would be to get better results by changing some parameters.

3.1 Sensing the Data

The CS should be implemented directly in the acquisition of the data. With the limiting resources within the CubeSat, this plays a huge role, where power can be saved in operating the detector, storing data and transmitting data. Imagers built on compressive sensing principles like CS-MUSI use a liquid crystal phase retarder as a dispersive element. The LC encodes the spectral domain of the incoming signal. The voltage over the LC controls the encoding, and the LC's spectral transmission response will change. In this thesis, the aim is to emulate compressive sensing as done using a real LC phase retarder. Building an actual imager is beyond this project's scope, but using the same principles will be a proof of concept for this CS hyperspectral imaging.

3.2 Datasets

Since the thesis addresses the concept of reconstructing CS hyperspectral images, the testing will be done on already collected datasets. The datasets used in this thesis are well established in the hyperspectral image processing field, with some of the more widely used being Cuprite, Jasper Ridge, and Salinas. These are collected under the same dataset named Compact Dataset in this thesis. Another dataset used is the one provided by *The Interdisciplinary Computational Vision Laboratory* (ICVL) [15]. ICVL database consists of over 200 images with dimensions 1392x1300x519. These are images from an urban city scene. The Aviris dataset contains hyperspectral images from the database from the Aviris project, collected over the USA for a decade. The images from Aviris were handpicked, meaning that the images that are given in Table 1 are chosen due to their quality in the form of spatial and spectral dimensions. Fig.13 shows some examples of images in the ICVL database, and Fig. 14 shows some examples from some popular hyperspectral datasets. Table 1 gives an overview of the different earth observational datasets used.

Dataset overview	Images	Dimensions	Wavelengths	Data-type	Targets Category
ICVL	29	1392x1300x519	400-1000	double	City, urban
Compact Dataset					
Jasper Ridge	1	100x100x198	380-2500	double	Nature, river
Cuprite	1	250x190x188	370-2480	double	Nature, ridge
Airport-Beach-Urban	1	100x100x205	NA	double	City, urban
The China Dataset	2	420x140x154	NA	int16	Nature, fields
The USA Dataset	2	307x241x154	NA	int16	Nature, fields
Kennedy Space Centre	1	512x614x176	NA	double	Urban and nature
Salinas	1	512x217x224	NA	double	Nature, fields
Aviris					
SaltLake	1	2808x786x224	360-2500	int16	Nature, ridge, water
Olympic	1	2447x742x224	360-2500	int16	Nature, ridge, water
Campbell	1	2313x777x224	360-2500	int16	Nature, ridge, water
St. George	1	2128x749x224	360-2500	int16	Nature, ridge
Playmounth	1	5104x737x224	360-2500	int16	Nature, fields

Table 1: The table shows an overview of the different hyperpsectral datasets used throughout this project.

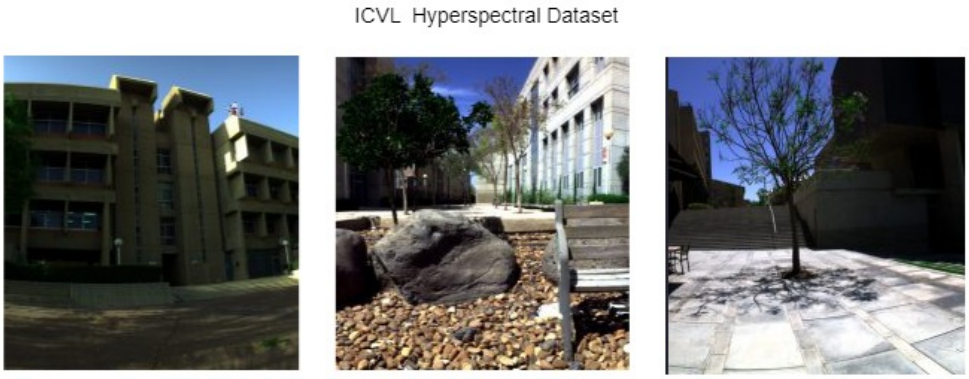


Figure 13: Example images from the ICVL Hyperspectral Database [15].

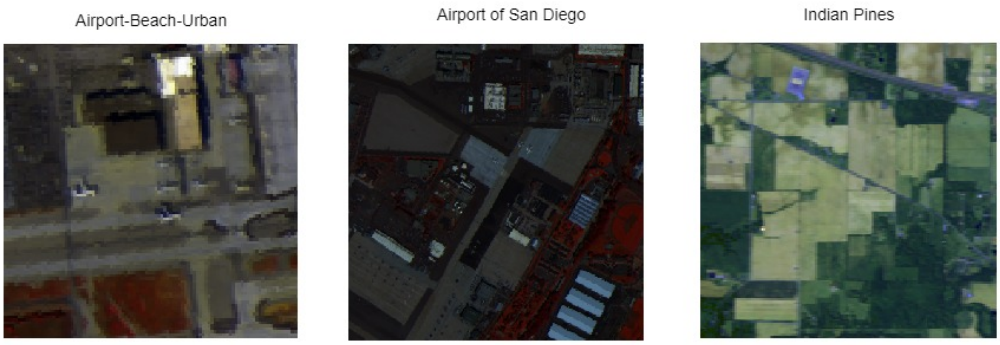


Figure 14: Example images from different public hyperspectral datbases.

ICVL database has high-quality images. These are taken at ground level in a city environment. The Compact Dataset and the Aviris dataset include both nature and urban scenes with a birds-eye-view or earth observational. In order to train the deep learning model, the training samples must have equal dimensions. The earth observational dataset has a broad spectrum of dimensions; some images have higher resolutions than others. In order to have as high a spectral dimension as possible with the limited datasets, the spectral boundary was set to 150 bands; however, this left out some of the images from the Compact Dataset. The bands for each image are down-scaled to 150 bands from their original length. This ensures that the bands' structure remains the same because the spectral resolution is different from image to image.

The hyperspectral image is the signal $\mathbf{f}$ from the CS theory. In the pre-processing of the data, the CS is performed on each pixel. Keeping the same sensing matrix throughout this process will ensure that the final result will be the spectral encoded image with M bands. Thus keeping the emulation of sampling as equal to a practical example as possible. Fig. 15 shows a schematic overview of the compressive sensing for this project. The hyperspectral image is compressive sensed according to the CS-theory and with a compression ratio given by the ratio between M and N.

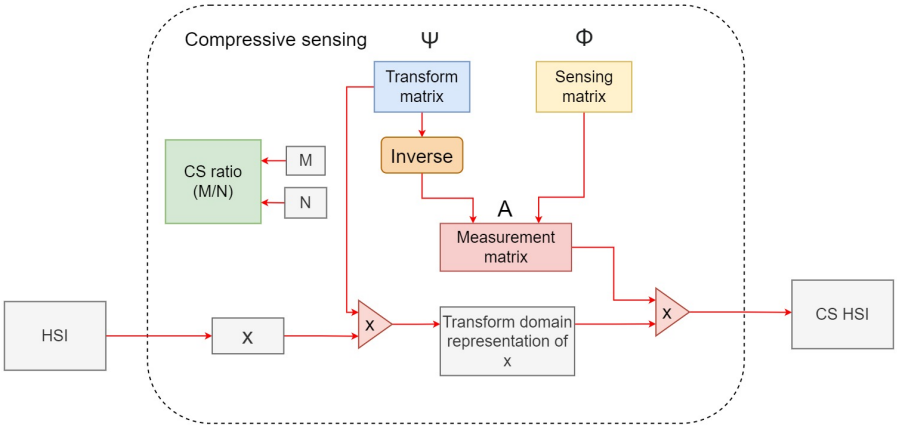


Figure 15: An overview of the compressive sensing stage for every hyperspectral image in the dataset to create patch-pairs of true image and CS-image.

Following the CS theory, the sensing process needs to follow certain constraints. The constraints are: sparsity, restricted isometry property and incoherence property between the measurement matrix and transform domain matrix. The first approach is to define the random measurement matrix Φ. The measurement matrices can be optimized for better incoherence with the transform domain matrix, which can improve the performance of the reconstructed signal [54]. For simplicity, the thesis will follow the principles of the

well-established Gaussian random matrix. The transform domain matrices used in this thesis are the *DCT* and *DFT*. The sensing matrix $\mathbf{A}$, where the M-samples are defining number of rows from Ψ to be acquired, and Φ defines which rows from Ψ to sample for making matrix $\mathbf{A}$. This process is shown below, where the figure shows the transform domain matrix as IDFT matrix with dimmensions 20 x 20 and how the M = 5 rows are randomly selected and creates the measurement matrix A. This will result in a compression of 20% sampled data:

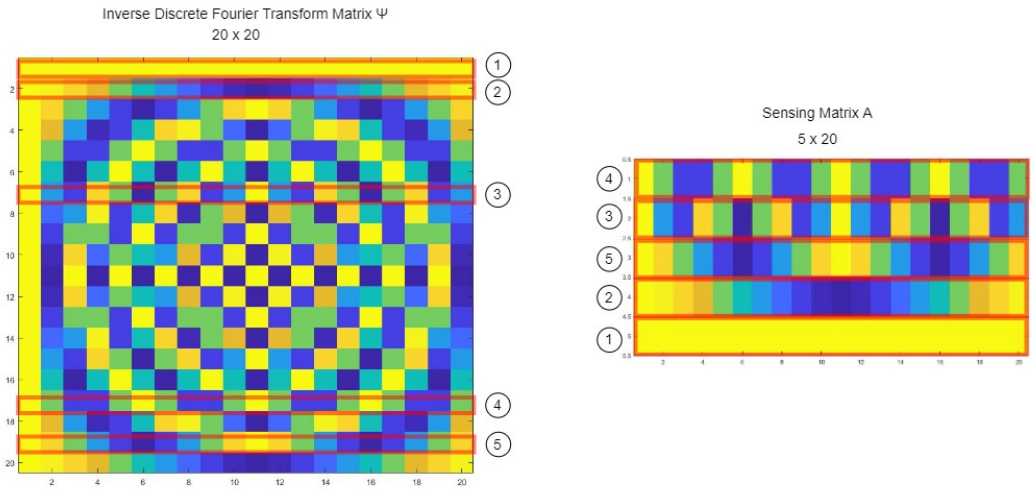


Figure 16: The figure shows the structure of matrix $\mathbf{A}$.

Fig. 17 shows the plot of one pixel from a hyperspectral image. The left plot is the compressed pixel with 20% of the data sampled, the middle is the original pixel, and the right plot shows the sparsified version of the original pixel.

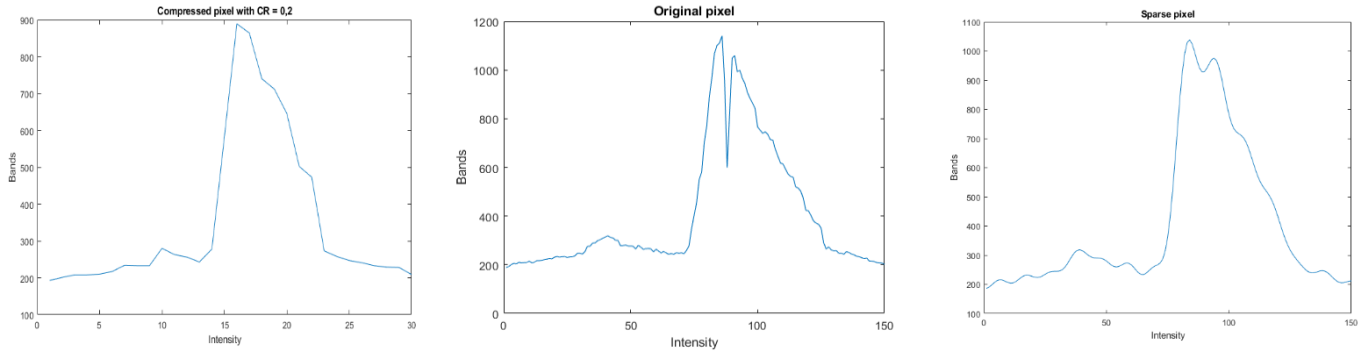


Figure 17: The figure shows one pixel after compressive sensing (left), original pixel (middle) and sparsified pixel (right). The spectral domain has been reduced from 150 samples to 30 samples on the compressive sensed plot.

3.3 Data Pre-processing

Three datasets used in this thesis, the ICVL, Aviris and one consisting of a dozen publicly available earth observational images, will be referred to as the Compact Dataset. The training will be performed by using

them individually for comparisons in the results section. For training-, validation- and test set, the data needs to be pre-processed to have the same dimensions. In order to reduce the parameter complexity and the computational times, the input training sample images are reduced in spatial dimensions. Each image is divided into patches of 64x64x150 with 32 pixels overlapping in the spatial domain. The spatial dimension of 64x64 includes enough spatial data for the model to learn features while keeping training memory low. By overlapping with 32 pixels, the amount of training data is increased. This operation is shown in Fig. 18.

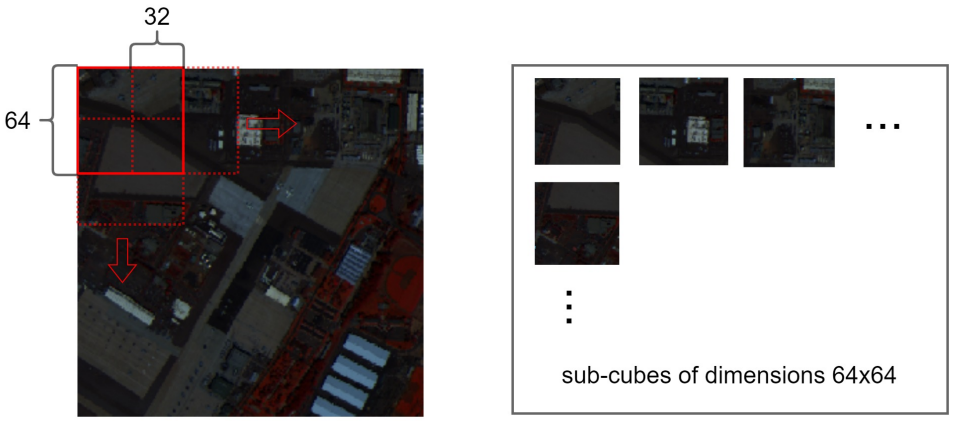


Figure 18: The figure shows the data being divided into sub-cubes.

Before the images are divided into patches, CS is performed over each image. Then one creates a patch-pair with both the original data and the CS data, as shown in Fig. 19 below.

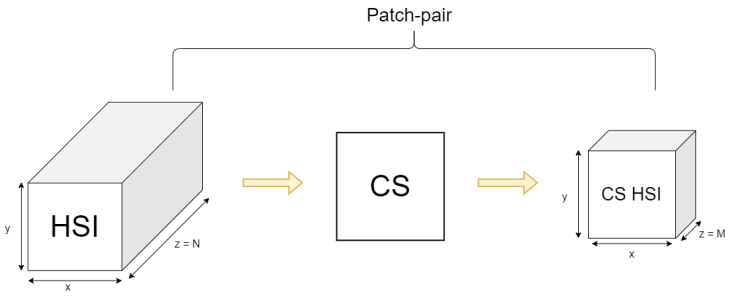


Figure 19: The figure shows the concept of patch-pairs of original image at the left and CS representation at the right.

Several training datasets are made for training the model to see which gave the best results. The different training sets are shown in the table below:

Models	Dataset	Sparse	CR	Patches
ICVL CR 0,2	ICVL	No	0,2	3809
ICVL CR 0,6	ICVL	No	0,6	2951
ICVL CR 0,2 Sparse	ICVL	Yes	0,2	3809
Aviris CR 0,2	Aviris	No	0,2	
Aviris CR 0,6	Aviris	No	0,6	2584
Aviris CR 0,2 Sparse	Aviris	Yes	0,2	2584

Table 2: The table shows an overview of the different training datasets built in the pre-processing stage.

Different compression ratios were tested for reconstruction, to see the impact that less samples have on the model's ability to keep quality in the reconstructed image. Compression ratios are given in Table 2 above. The data is divided into smaller patches to reduce computational times and complexity in training the model. The patching will also increase the training data samples, thus eventually better generalising the model.

3.4 DeepCubeNet

In order to reconstruct the CS hyperspectral image, a reconstruction method is needed, which for a long time has been sparse recovery algorithms [4], which can further be divided into convex optimization algorithms and greedy algorithms. These have been used for many compressive sensing tasks like imaging, MRI and HSIs, and with great results. However, the great results from these algorithms are bounded by handcrafted priors and their computational times. These reconstruction methods have evolved in the last years to pre-trained models that learn the priors by themself and take considerably less time to recover the data from the CS. These pre-trained models are from deep learning methods like CNNs. In this thesis, a U-Net is used as the structure for training the model for mapping the CS image to the original image. The U-Net architecture used is the DeepCubeNet. The DeepCubeNet differs from the original U-Net architecture [39], where it uses 3D convolutions instead of 2D convolutions. By changing 2D convolutions with 3D convolutions, the network can be used for hyperspectral data reconstruction, learning features from neighbouring spatial pixels and spectral bands. The 3D convolution is showed in Fig. 21. The max-pooling operation is also changed to work with 3D data like hyperspectral images. The 3D max-pooling is showed in Fig. 22. The original layout of the DeepCubeNet is given in the figure below.

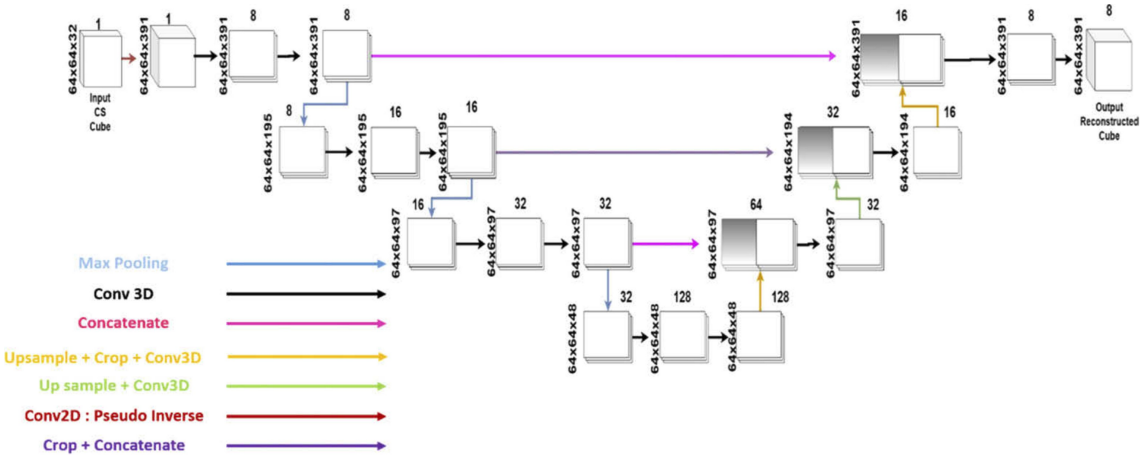


Figure 20: An overview of the DeepCubeNet U-Net architecture [6]

The original DeepCubeNet architecture is two parted. The first block is a pseudo-inverse operation that backprojects the input data from the compressed domain with 30 bands to the hyperspectral domain with 150 bands. This block is non-trainable to avoid overfitting the model, which means the training gets too specific to the training data.

Another difference from the standard U-Net is that the input data does not reduce spatially going down the different layers but spectrally. The data dimensions will be reduced in the spectral domain.

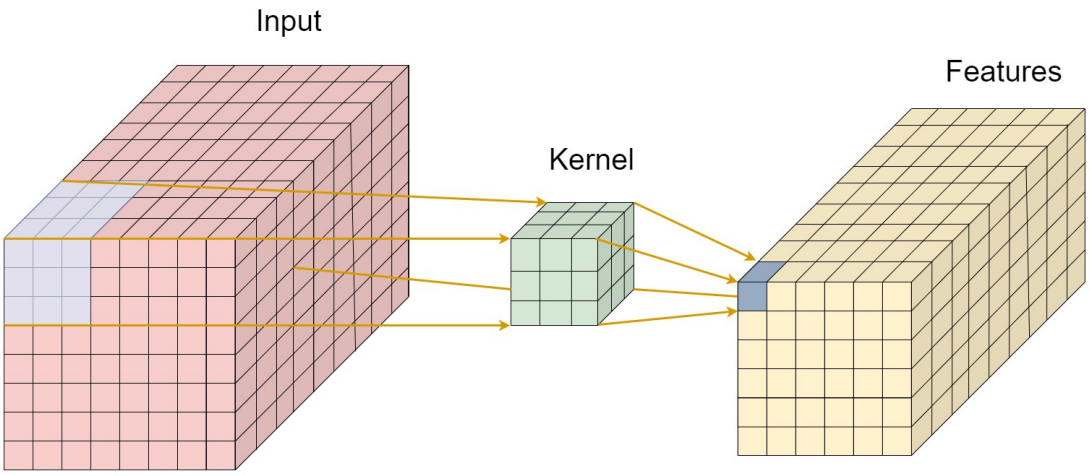


Figure 21: 3D convolution example with input data, kernel and features.

The figure above shows the 3D convolution. It has a 3 dimensional kernel that can move in 3 dimensional space (x, y, z) over the input data, in order to learn the low level features. The output is a 3 dimensional cube. Such convolutions are used for detection medical images, videos and hyperspectral images.

The 3D max-pooling is performed on the layer's 3D convoluted output from the last convolutional block. The operation is similar to the standard max-pooling. The difference is that the operation is performed on each convoluted output. In the example, in Fig. 22 the window size is 2 x 2 with a stride of two, which gives a 2 x 2 output. The outputs from each max-pooling operation are then stored back in the same order as the 3D convoluted output before max-pooling. By doing this particular max-pooling, one get an output that is half the height and width but has the same depth as before.

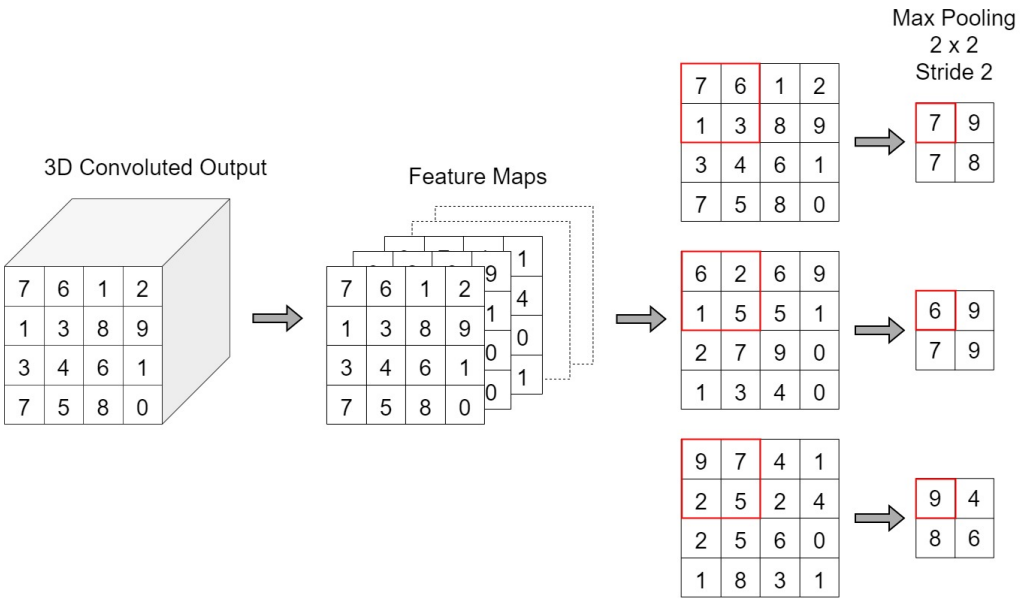


Figure 22: 2 x 2 3D max-pooling of 3D convolutional output, with stride 2 [55].

3.5 Training

The training sets tend to be significant when working with hyperspectral images, and this project is no exception. The ICVL and the Aviris datasets are too large for the graphic card memory to process. The graphic card used in this project is the GeForce 3080 RTX, which has a memory of 10 GB. ICVL CR 0.2 Sparse, which is the ICVL dataset sparsified, contains 3809 training pairs. Using 20% of the data for the validation set leaves 3047 samples for training which is 3,5 GB in memory for the compressive sensed samples and 17,8 GB for the true images. For each epoch, the model needs to process 20 GB of memory with only 10 GB in its hands. This will give an "Out of memory" error. This can be overcome by running training and validation on different GPUs, decreasing the number of samples or loading the training data as batches with generators in TensorFlow. Since the two first are not an option, generators to load batches are used. The generators load the data as batches to our memory from the disk.

For training, the validation data is chosen to be 20% of the training data. The batch size is set to 6, with 30 epochs. The batch size defines how many training samples the model will input before the internal parameters are updated. A batch size of six means that six and six samples are loaded into the model before the model parameters are updated. The epochs define how many times the whole training set is loaded into the model. An epoch of 30 means that the training set will be loaded 30 times into the model. Having an epoch of one means that every training sample has had the opportunity to adapt the internal parameters once. When creating the datasets, the data is stored naturally in an order which gives neighbouring patches a high correlation. If the data is loaded into the model in their original order, the data can easily overfit and learn the neighbouring patterns too much. This is prevented by enabling shuffling in the training step. Shuffle ensures that the training samples order is loaded randomly into the model. The shuffling still ensures that the patch-pairs are shuffled to the same position.

The model uses the Adam optimizer for training. Adam optimizer is used due to its ability to update weights with high computational efficiency and low memory requirements. Adam optimizers are also well suited for extensive data and parameter problems [56], which is suitable for training with HSIs and U-Nets with over 1 million trainable parameters. The learning rate, which is a hyperparameter that controls the change of internal parameters of the model from the error estimations every time these are updated, is set to 0.0001. The learning rate impacts training times, and the stability of the training process [57]. A lower learning rate can cause longer training processes, and too big can make the training process unstable, and weights can be sub-optimally. The loss function is the mean-square-error. The loss function is used to calculate how good the predicted data is compared to the true data: the higher loss, the worse predictions. The mean-square-error ensures that the model's performance is good on most of the data.

3.6 Reconstruction

After training the model through its epochs, the model is stored with a specified model name referring to the training data. A test set which is built on the same principles as the training set is used to test the model's ability to reconstruct the hyperspectral images. The reconstructed result will define the model's ability of generalization, which is how good the model performs on "unseen" data [58]. The test set must be of patches that are not in the training set and are therefore picked beforehand and stored in a separate folder.

Different models with different parameters and set-ups are tested and will further be presented in the result section. Reconstruction is done by using the *model.predict* function within TensorFlow, and the reconstruction quality valuation is done using *PSNR* values and *SSIM*. *PSNR* is calculated using the formula from the theory section and with the built-in function within MATLAB. The *SSIM* is also calculated for the reconstructed and original images.

3.7 Data Types

The ICVL dataset contains images of the data type double, which is 64-bit double-precision floating variables. These are too large to use for model training, so the training data should be normalized. Normalizing the data helps speed up the training process and reduce the computational complexity of higher value vari-

ables. The data is first cast to a 16-bits value or uint16 data class. Uint16 means that the variables are integers and can be valued from 0 - 65 535. After this, the training data is normalized between 0 and 1. The normalization is done by dividing each number by 65 535. After reconstruction, the data is multiplied by 65 535 to reveal the uint16 image.

3.8 Tools

The data processing, namely the compressive sensing, the patch subsampling, data post-processing, visualization and stitching of patches together, are done with MATLAB. Comparing the reconstructed images with the original images are done with Hyperspectral Viewer's help in MATLAB, where the images can be visually inspected. The U-Net implementation is done in Python with the use of TensorFlow API. TensorFlow is a machine learning platform that is open-source; it contains many libraries for developers to create powerful machine learning applications. Codes for both MATLAB and Python are given in the appendices.

3.9 Architecture

The building of the different datasets is done using MATLAB, and Fig. 23 shows the concept of how this is achieved. The database could be ICVL or AVIRIS, i.e. the source the images are from. The images are divided into CS patches and GT (ground truth), i.e. original patches. Fig. 24 shows how the folder structures are built up. First, the training data folder is created, and inside the different HSI images are represented with a name, i.e. Olympic, followed by one folder for CS patches and one for GT patches. Inside each of these folders, the patches of 64x64x30 (for CS) and 64x64x150 (for GT) will be in the order created when building the datasets. It is important that the respective GT patch is loaded into the model for a given CS patch. This is because the patch-pairs must be loaded at the same time. The patch files are stored as .mat files, a MATLAB format for storing matrices or cubes in this case. A Python package called *mat73* is used to load the files inside the Python environment. The process is the same for making the test data.

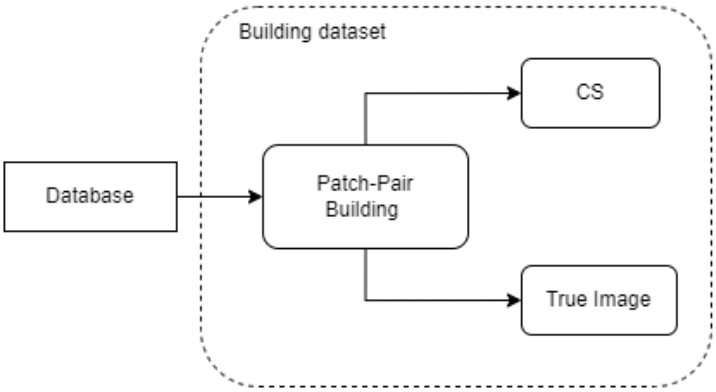


Figure 23: A chart showing the principle of building datasets for this project

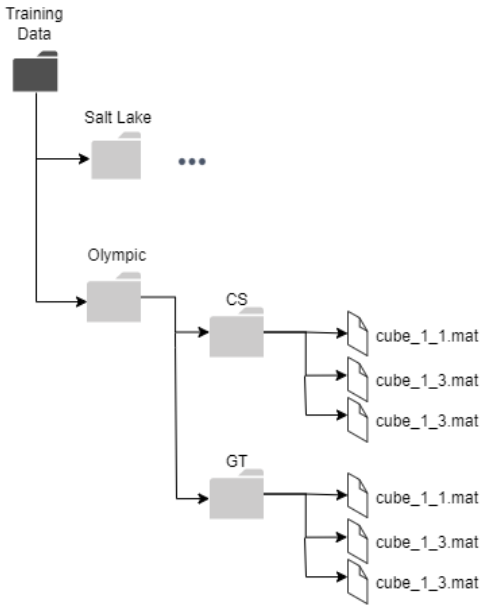


Figure 24: Figure shows the folder structure for the training data.

Fig. 25 shows the concept of training the model. This part is done in Python using TensorFlow. The U-Net is trained by using the training data as input to the train-test-split block, where the data gets divided into training data and validation data. The split is a percentage of the training data. Using a split of 20%, the model uses 20% of the training data as validation data. The data is then loaded into the model through a batch generator block. This is done such that the computer does not run into any out-of-memory errors. After the model has finished its epochs, it can be used to reconstruct the images from the testing data, using the *model.predict* function within TensorFlow. The code for both MATLAB and Python is provided in the Appendix.

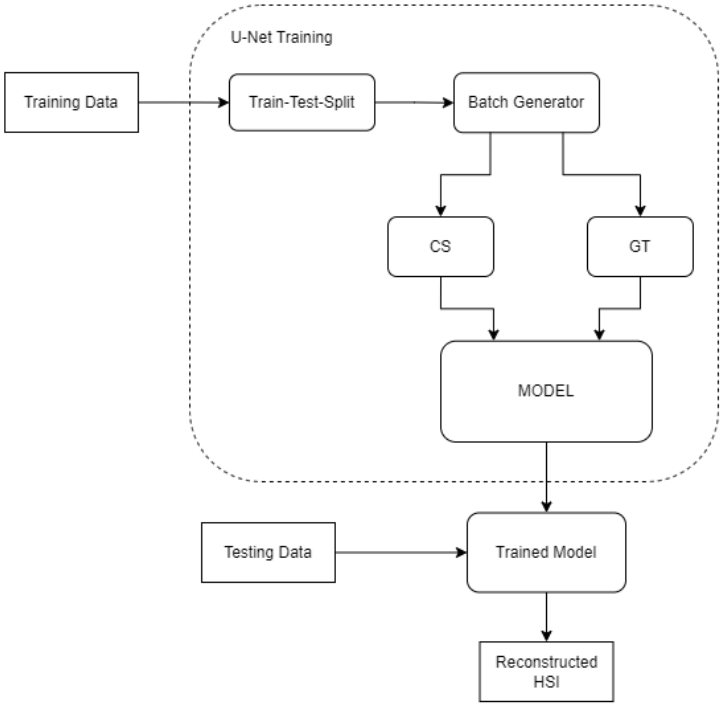


Figure 25: Figure shows the folder structure for the training data.

4 Results and Discussion

In this section the results are presented and discussed. The results are the reconstruction of the hyperspectral images with different trained models and datasets. They give an overview of the current state of the reconstruction system, and give important feedback that can later be combined with theory to find new solutions. The results will be discussed after each topic; these are under AVIRIS and ICVL. *Hyperspectral Viewer* is used for side-by-side comparisons of reconstructed and original images, and pixel comparison plots are used to see how the models perform on a detailed spectral level. In order to see the reconstruction performance in all axis of images, *PSNR* and *SSIM* are provided in tables.

4.1 ICVL Dataset

The ICVL dataset is the dataset with the urban hyperspectral images from the *Interdisciplinary Computational Vision Laboratory* [15]. The dataset is used here to train three models, one standard model with 80% discarded data before reconstruction, one with 40% discarded data and one with 80% discarded data which is sparsified. 27 images are used in the training process giving 3809 training samples. Training samples are referred to as patch-pairs of 64x64x30 CS-images and 64x64x150 true-images. Two images are referred to in this section in particular, this is The Bench images and The View image. These are renamed from the ICVL database, where The Bench has the filename bgu_0403-1439, and The View has the file name eve_0331-1705. These images are excluded from the training data and used to test the models' reconstruction ability of new unseen data. The Bench and The View are given in Fig. 26 below.

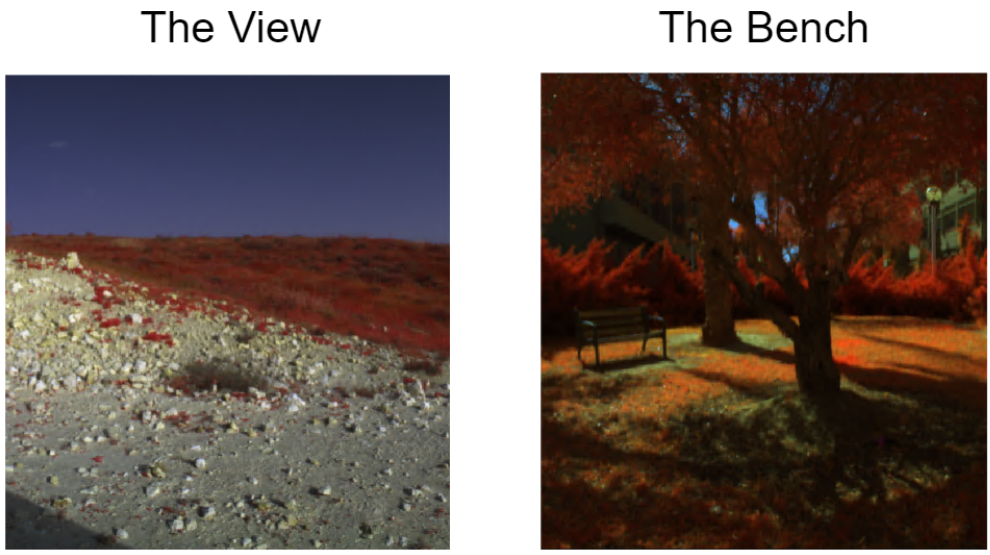
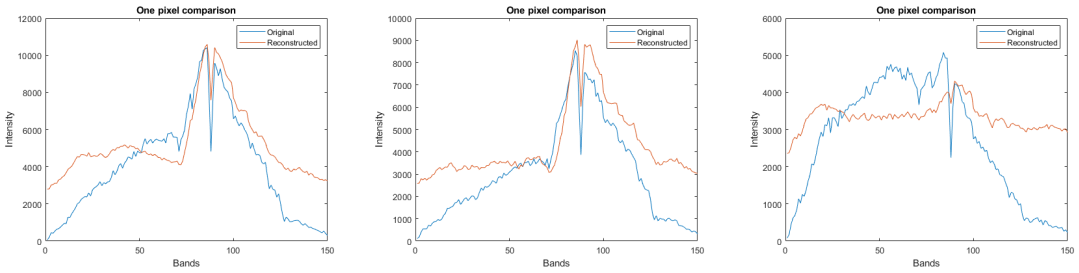


Figure 26: The left image is The View and the right image is The Bench from the ICVL dataset.

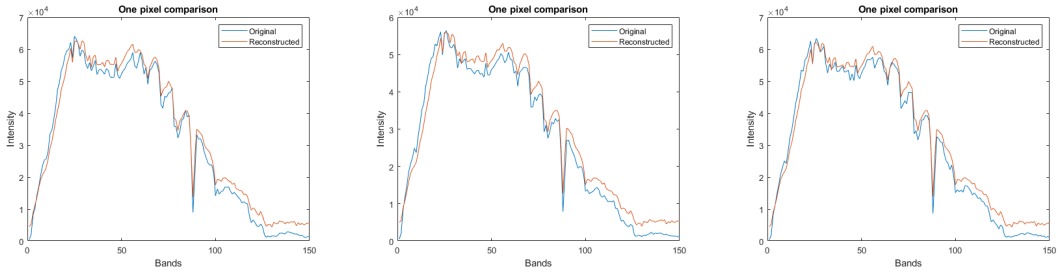
ICVL CR 0,2

The ICVL CR 0,2 is the training data of the ICVL dataset where 80% of the original samples have been discarded. This leaves us with a CS HSI with 30 bands compared to 150 original bands. 27 images are used in the training process to create patches that give 3809 patch-pairs in total, divided into compressed and true image folders. The model is trained with batch sizes of 6 and 30 epochs. Fig. 27 shows three randomly picked pixels from The Bench from the ICVL testing set. The test image has the same CR and is categorized as unseen data for the model. Fig. 28 shows the pixel comparison of The View image reconstructed with the ICVL CR 0.2 model. The pixel is picked from one of the recovered patches.



(a) (b) (c)

Figure 27: (a) *PSNR* = 31,2 dB and *SSIM* = 0,809 (b) *PSNR* = 32,1 dB and *SSIM* = 0,809 (c) *PSNR* = 32,9 dB and *SSIM* = 0,803.



(a) (b) (c)

Figure 28: (a) *PSNR* = 26,6 dB and *SSIM* = 0,867 (b) *PSNR* = 25,8 dB and *SSIM* = 0,866 (c) *PSNR* = 26,4dB and *SSIM* = 0,862.

Table 3 shows the *PSNR* and *SSIM* for a patch in The Bench image. Table 4 shows the *PSNR* and *SSIM* for a patch in The View image.

PSNR	Value [dB]	SSIM	Value
Whole image	31,5	Whole image	0,8
Pixel average	31,7	Pixel average	0,81
Band average	33,1	Band average	0,803
MatLab function	31,5		

Table 3: Table shows the *PSNR* and *SSIM* values for a reconstructed patch from The Bench image.

PSNR	Value [dB]	SSIM	Value
Whole image	24,6	Whole image	0,837
Pixel average	24,8	Pixel average	0,834
Band average	26,7	Band average	0,835
MatLab function	24,6		

Table 4: Table shows the *PSNR* and *SSIM* values for a reconstructed patch from The View image.

Fig. 29 and Fig. 30 shows the full reconstruction of The Bench image and The View image respectively. The patches are reconstructed one by one and stitched back together.

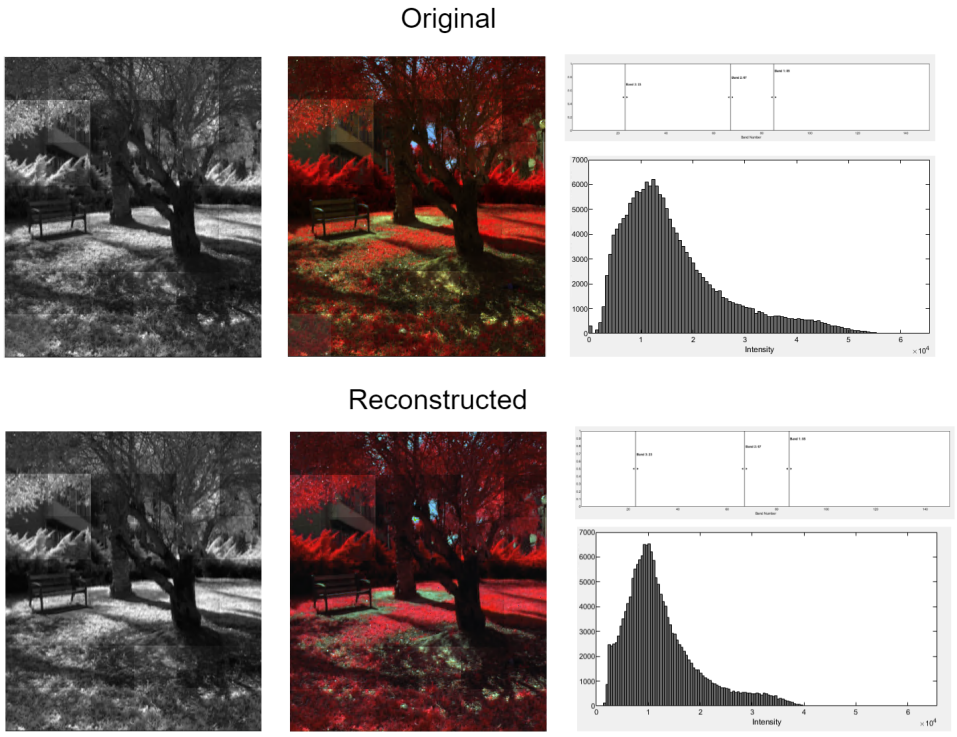


Figure 29: The Bench image stitched together with reconstructed and original image as reference. *PSNR* is 29,2 dB and *SSIM* is 0,826.

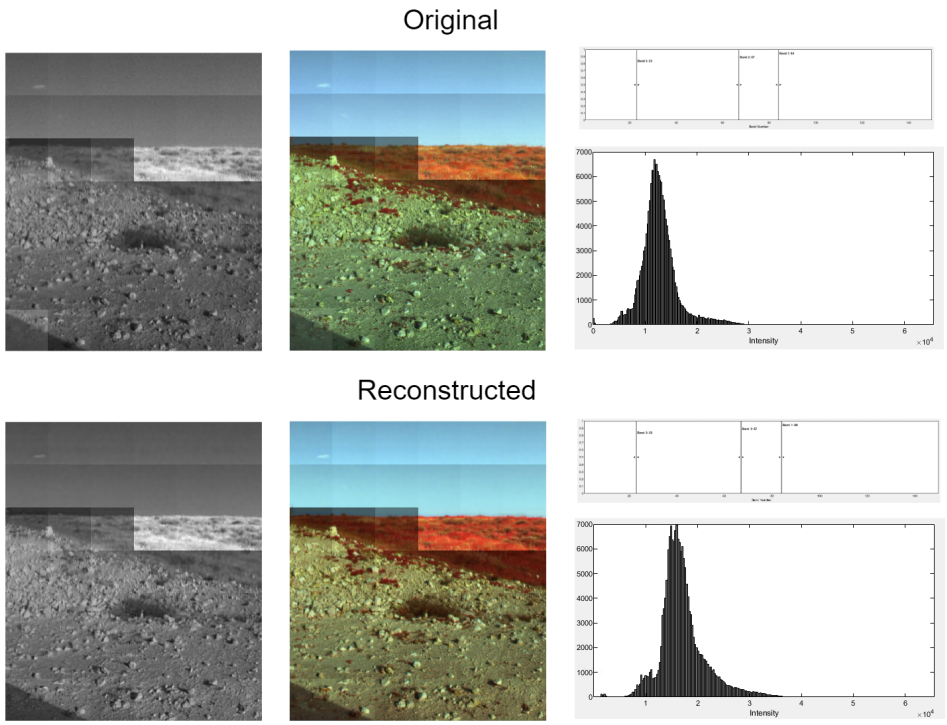
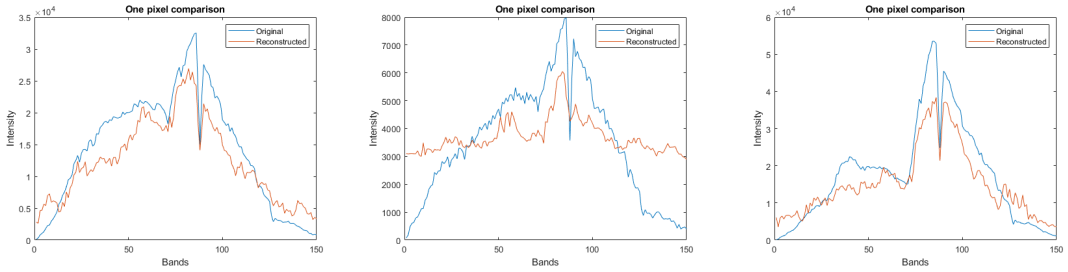


Figure 30: The View image stitched together with reconstructed and original image as reference. *PSNR* is 25,4 dB and *SSIM* is 0,850.

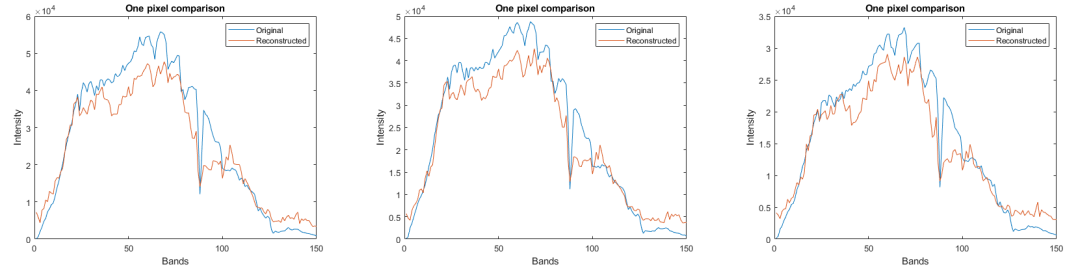
ICVL CR 0,6

The model is trained using the same ICVL images as with ICVL CR 0,2 model. However, the compression ratio is now 0,6, which means that 40% of the original bands are discarded. This means that 90 spectral bands from the original 150 are left for reconstruction. The batch size is set to 6 with 30 epochs. Fig. 31 shows three randomly picked pixels from The Bench from the ICVL testing set. The test image has the same CR and is categorized as unseen data for the model. Fig. 32 shows the pixel comparison of The View image reconstructed with the ICVL CR 0,6 model. The pixel is picked from one of the recovered patches.



(a) (b) (c)

Figure 31: (a) *PSNR* = 24,7 dB and *SSIM* = 0,795 (b) *PSNR* = 32,3 dB and *SSIM* = 0,825 (c) *PSNR* = 21 dB and *SSIM* = 0,699.



(a) (b) (c)

Figure 32: (a) *PSNR* = 20,9 dB and *SSIM* = 0,705 (b) *PSNR* = 22,7 dB and *SSIM* = 0,734 (c) *PSNR* = 25,5 dB and *SSIM* = 0,776.

Table 5 and Table 6 shows the *PSNR*s and *SSIM*s for The Bench and The View respectively.

PSNR	Value [dB]	SSIM	Value
Whole image	31,2	Whole image	0,81
Pixel average	31,6	Pixel average	0,822
Band average	32,6	Band average	0,82
MatLab function	31,2		

Table 5: Table shows *PSNR* and *SSIM* values for reconstruction of a patch from The Bench image in ICVL, with CR = 0,6.

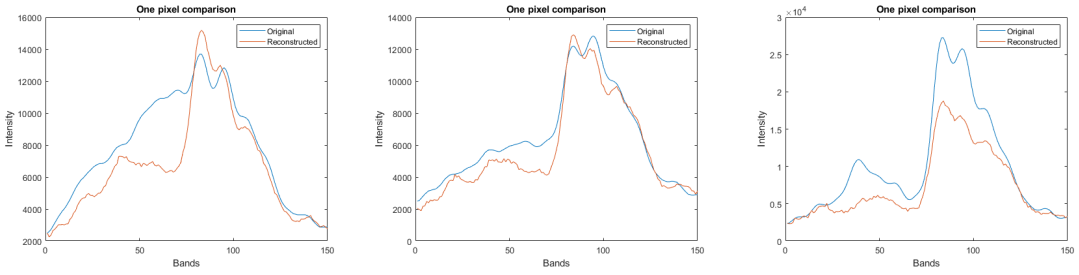
ICVL CR 0,2 Sparse

This time the U-Net was trained using sparsified images. The images are sparsified in the *DCT*-domain by setting all frequency coefficients above 30 to zero. Additionally, the training procedure is identical to the method used when training with non-sparsified CR 0,2 images from ICVL. Both training images, i.e. the

PSNR	Value [dB]	SSIM	Value
Whole image	23,5	Whole image	0,799
Pixel average	24	Pixel average	0,735
Band average	26,7	Band average	0,853
MatLab function	23,5		

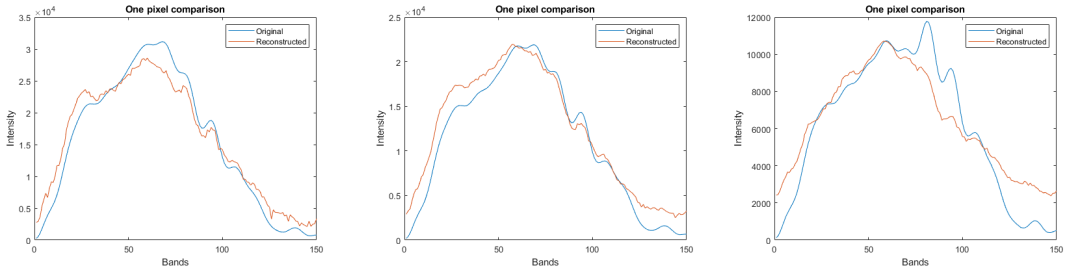
Table 6: Table shows *PSNR* and *SSIM* values for reconstruction of a patch from The View image in ICVL, with CR = 0,6.

CS images with 30 bands and the true images with 150, are sparsified. Fig. 33 shows the pixel comparisons between original and reconstructed pixels in The Bench image, while Fig. 34 shows the same for The View image.



(a) (b) (c)

Figure 33: (a) *PSNR* = 30,6 dB and *SSIM* = 0,956 (b) *PSNR* = 37,2 dB and *SSIM* = 0,978 (c) *PSNR* = 24,7 dB and *SSIM* = 0,916.



(a) (b) (c)

Figure 34: (a) *PSNR* = 28,7 dB and *SSIM* = 0,888 (b) *PSNR* = 31,0 dB and *SSIM* = 0,891 (c) *PSNR* = 32,9 dB and *SSIM* = 0,863.

Table 7 shows the quality metrics as *PSNR* and *SSIM* values for the reconstruction of a patch in The View image. Table 8 shows the same for The Bench image.

PSNR	Value [dB]	SSIM	Value
Whole image	27,3	Whole image	0,856
Pixel average	28,3	Pixel average	0,887
Band average	29,1	Band average	0,854
MatLab function	27,3		

Table 7: *PSNR* and *SSIM* values from a reconstructed patch from The View image in the ICVL dataset using model 30.

PSNR	Value [dB]	SSIM	Value
Whole image	19,1	Whole image	0,743
Pixel average	22,8	Pixel average	0,819
Band average	23,2	Band average	0,744
MatLab function	19,1		

Table 8: *PSNR* and *SSIM* values from a reconstructed patch from The Bench image in the ICVL dataset using model 30.

Fig. 29 and Fig. 30 show the full reconstruction of The Bench and The View respectively. The color images represent three bands of the hyperspectral images. Reconstructed images are displayed on the bottom and the original images at the top. Both reconstructed images show histograms that follow the same shape as their originals. The histograms are shifted a bit in both instances. These are both reconstructions with the ICVL CR 0,2 model. When using the ICVL CR 0,6 model, the reconstructions are similar to what is obtained by the ICVL CR 0,2 model. The *PSNR*s are mostly the same, with a slight decrease in *SSIM* and *PSNR* for the reconstruction of The View. The reconstructed image pixel follows the shape of the original, but there is a significant difference between the pixel comparisons for ICVL CR 0,2 and ICVL CR 0,2. When looking at Fig. 32 one can see that the model does not follow the drop around band number 90, but in Fig. 31 the reconstructed pixel follows the same drop. ICVL CR 0,2 Sparse provides results similar in *PSNR*s to the two beforehand, but with a higher *SSIM* for The Bench, with 0,856 for the whole image. The View, however, suffers in the reconstruction compared to the previous models. Looking at the first model, ICVL CR 0,2, which is trained on images with 80% discarded bands, the reconstruction of The Bench image achieves the highest *PSNR* values with a 31,5 dB when taking *PSNR* of the whole image. This is 6 dB better than what the reconstruction of The View did, even though looking at the pixel comparison plots in Fig. 27 and Fig. 28 the reconstructed pixel looks better for The View than The Bench. Here one have to look at the **y**-axis in the figure; the distance between the reconstructed pixel and the original pixel are sometimes in the order of thousands in the plot of The View, while in the reconstruction of The Bench the distance is at its most 1500. However when looking at the *SSIM* values The View gets the highest values with a total *SSIM* of 0,837 compared to 0,803 for The Bench. This shows the importance of having several quality measurements in order to state the performance of the model.

4.2 Aviris

The Aviris dataset contains images from *Airborne Visible/Infrared Imaging Spectrometer* hyperspectral database [16]. The images are named after the area they cover in The United States: Salt Lake, Olympic and St George. The image of St George is used for testing, while the other two are the training data. All images are divided into the same patch-pair configuration as the previous datasets. Salt Lake and Olympic makes 2584 patch-pairs of size 64x64x30 and size 64x64x150. Three models are trained individually and presented in this section with their respective results. The training pattern is the same as for the ICVL dataset models. One model is trained with 80% of bands discarded, another with 40% bands discarded, and a final one with 80% discarded bands and sparsified data. Fig. 35 shows RGB-representations of the three hyperspectral images used for the three upcoming models.

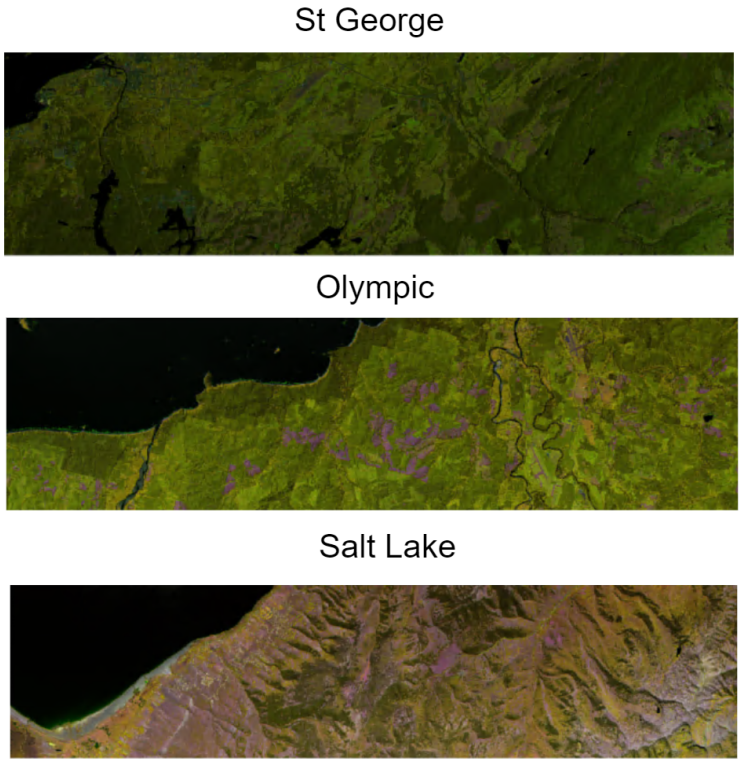
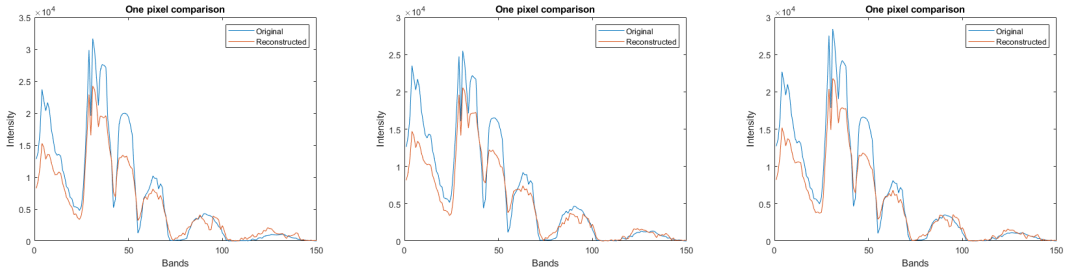


Figure 35: The Aviris dataset.

Aviris CR 0,2

The model is trained using images from the Avirs dataset. These images are handpicked from the Aviris database online, with roughly the same scenes and spatial dimensions. Salt Lake and Olympic are used for training, resulting in patch-pairs of 2584. St George is used as a test set for the reconstruction. The compressed data has a compression ratio of 0,2, meaning that 80% of the original samples are discarded. The batch size is set to 6 and epochs to 30. Fig. 36 shows the pixel comparison of pixels in St George.



(a) (b) (c)

Figure 36: *PSNR* = 26,8 dB and *SSIM* = 0,861 (b) *PSNR* = 27,9 dB and *SSIM* = 0,899 (c) *PSNR* = 28,1 dB and *SSIM* = 0,920.

Table 9 shows the *PSNR* and *SSIM* values from the reconstructions of a patch in the St George using Aviris CR 0,2.

PSNR	Value [dB]	SSIM	Value
Whole image	25,3	Whole image	0,863
Pixel average	25,4	Pixel average	0,869
Band average	34,9	Band average	0,886
MatLab function	25,3		

Table 9: The table shows the *PSNR* and *SSIM* values for a patch reconstruction from St George image.

Fig. 37 shows the full reconstruction of the St George image. The patches are individually reconstructed and then stitched together.

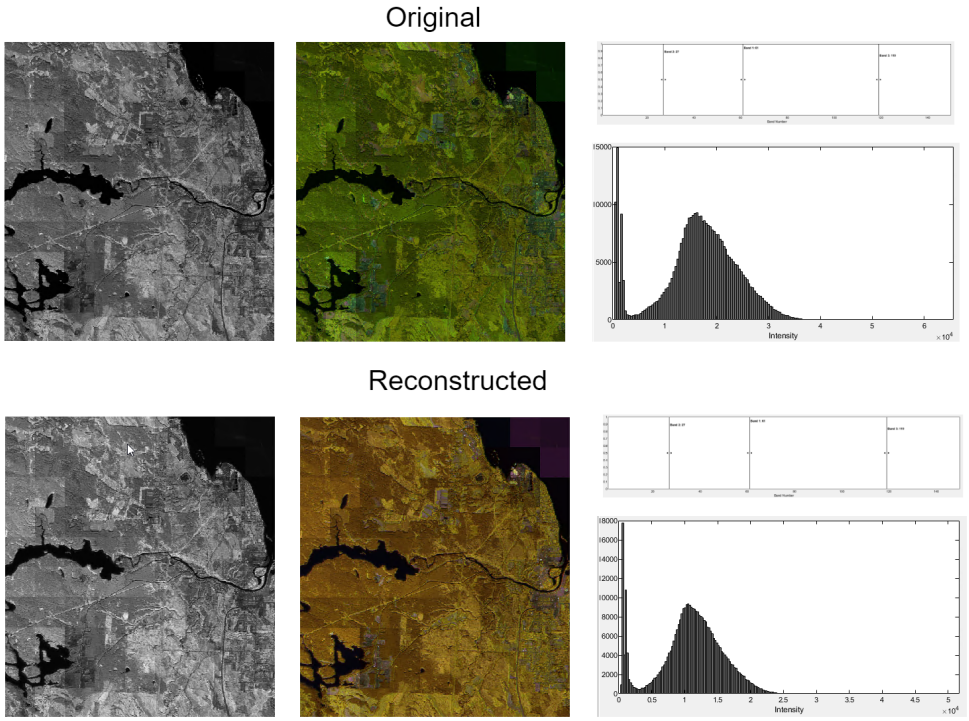
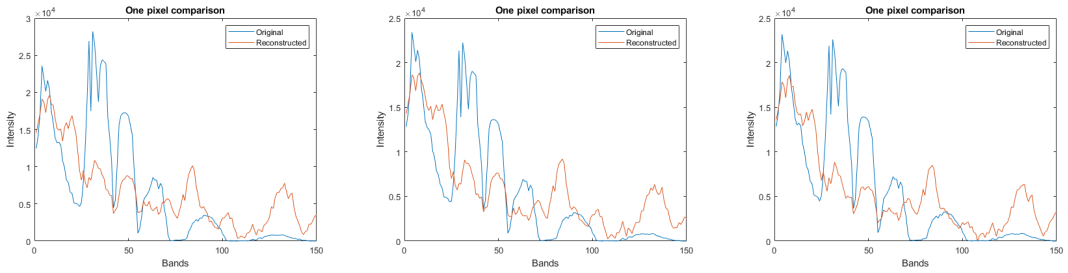


Figure 37: St George image stitched together with reconstructed and original image as reference. *PSNR* is 26,0 dB and *SSIM* is 0,846.

Aviris CR 0,6

The model is trained on Salt Lake and Olympic with a compression ratio of 0,6, meaning that 40% of the bands are discarded for the compressive sensed samples for training. The batch size is set to 6 with 30 epochs. Fig. 38 shows the pixel comparisons of the reconstructed St George image using the Aviris CR 0,6 model.



(a) (b) (c)

Figure 38: *PSNR* = 21 dB and *SSIM* = 0,407 (b) *PSNR* = 22,6 dB and *SSIM* = 0,462 (c) *PSNR* = 22,8 dB and *SSIM* = 0,445.

Table 10 shows the *PSNR* and *SSIM* for the reconstructed patch of St George.

PSNR	Value [dB]	SSIM	Value
Whole image	19,8	Whole image	0,42
Pixel average	20,4	Pixel average	0,424
Band average	25,3	Band average	0,552
MatLab function	19,8		

Table 10: The table shows the *PSNR* and *SSIM* values for a patch reconstruction from St George image with CR = 0,6.

Fig. 39 shows the full reconstruction of the St George image. The patches are reconstructed individually and stitched together.

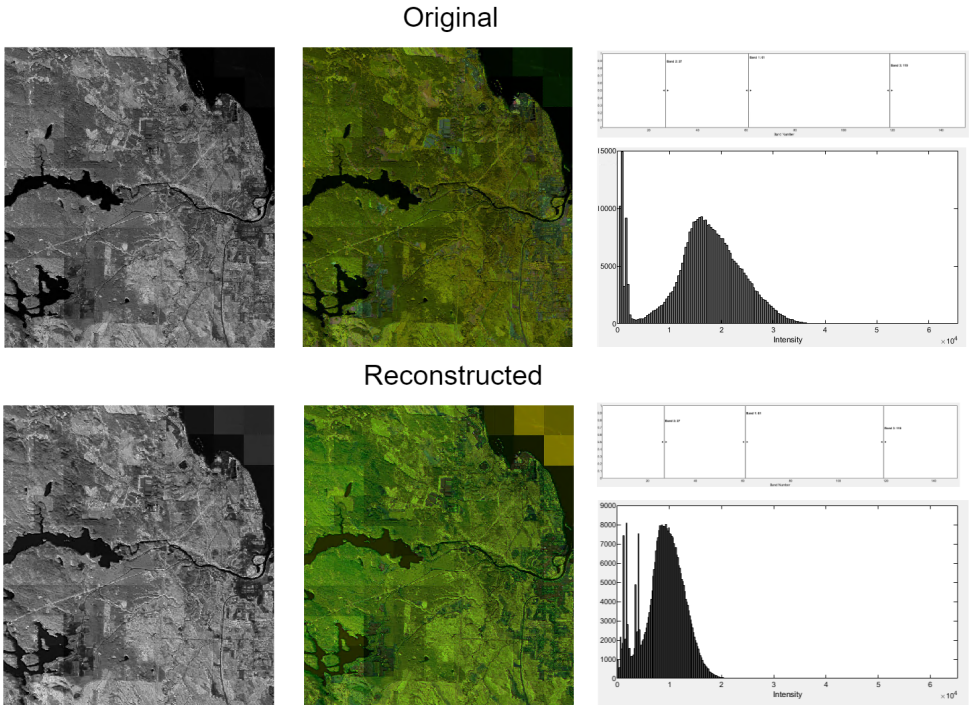
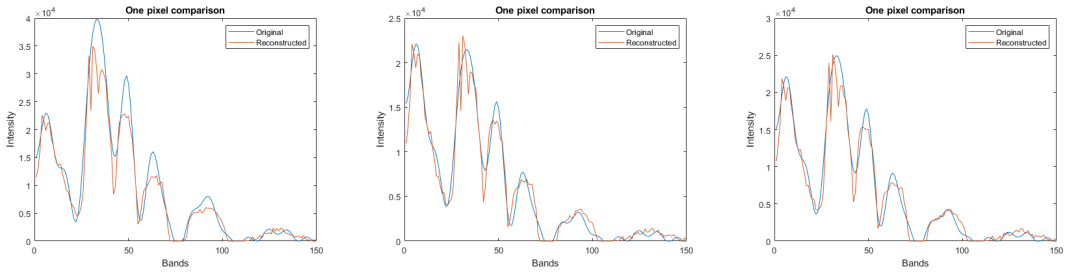


Figure 39: St George image stitched together with reconstructed and original image as reference. CR is 0,6. *PSNR* is 20,5 dB and *SSIM* is 0,486.

Aviris 0,2 Sparse

Aviris 0,2 Sparse model is trained on Salt Lake and Olympic with compression ratio 0,2, which means that 80% of bands are discarded for the compressive sensed samples for training. The spectral bands are sparsified by removing coefficients above 30 in the *DCT*-domain. The batch size is set to 6 with 30 epochs. Fig. 40 shows the pixel comparisons for pixels in St George.



(a) (b) (c)

Figure 40: (a) *PSNR* = 27,6 dB and *SSIM* = 0,832 (b) *PSNR* = 33,8 dB and SSIM = 0,870 (c) *PSNR* = 32,8 dB and *SSIM* = 0,857.

Table 11 shows the full reconstruction of St George. Patches are individually reconstructed and stitched together.

PSNR	Value [dB]	SSIM	Value
Whole image	29,1	Whole image	0,858
Pixel average	29,7	Pixel average	0,837
Band average	34,8	Band average	0,886
MatLab function	29,1		

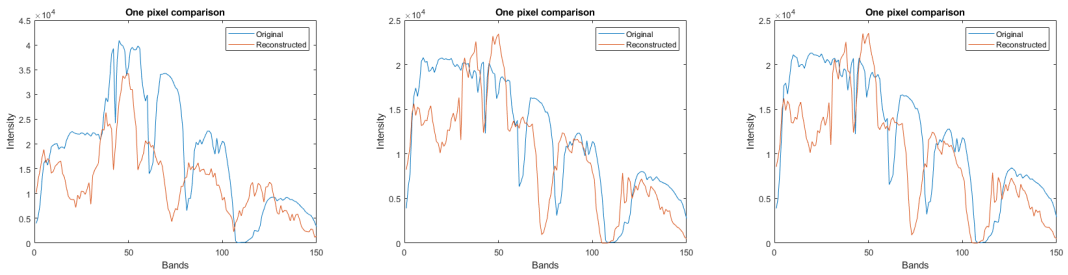
Table 11: The table shows the *PSNR* and *SSIM* values for a patch reconstruction from St George image which is sparsified.

When looking at the three models' reconstruction performances on St George, the sparsified model gives the highest *PSNR* values, presented in Table 11. This is expected as sparsified signals have fewer high-frequency components to recover. These high-frequency components make the small saw-like shapes in the spectral plots. As presented in the theory section, many signals and images are naturally sparse. By sparsifying the data, some information will always be lost from the removal of high-frequency components. The more sparsification one do, the less the example can be validated for real-life scenarios. Aviris CR 0,2 model, trained on 80% discarded bands, has its *PSNR* and *SSIM* values in Table 9. These are not as different as the sparsified ones. In Fig. 36 three pixels are presented with both reconstructed and original. Here one can see that the model can recover the original shape of the pixel at an adequate level, but it lacks in the intensity axis at points. This results in *PSNR* values lower than the sparsified model but with an *SSIM* value which can compete with it. Aviris CR 0,6 model, performed worst of the three models. This model is trained on images with 40% discarded bands. This means that the model with the most original bands performs worst on reconstruction. In Fig. 38 we can see that the pixel struggles to follow the shape of the original pixel. It does manage to keep the same shape for some of the peaks, but the overall reconstruction lacks quality. This can be seen in the *SSIM* values in Table 10 where the spectral similarity is too low for the reconstruction and original. The low performance of the reconstruction of patches where most bands are kept is the most interesting result from the Aviris dataset models. One should expect that keeping more data would result in easier reconstructions since more of the spectral bands are available for the model. However, this does not seam to be the case for this instance. This can have something to do with the model's training, but the training methods are the same as for the CR 0,2 and the sparse, with a batch size of 6 and 30 epochs. One explanation can be that there are more peaks and jittering in the pixel-spectral domain for the CS images when more bands are kept, while fewer samples lead to a more sparse pixel-spectral domain. When

looking at the results from the sparse training, it appears to deliver the best reconstruction results. The images used in the training and testing are quite similar regarding scenes and resolution, i.e. the number of pixels per meter matches quite well. The scenes are earth observations of fields, rivers, ridges, water and woods. Images that follow similar structures and scenes are used to train these models to observe results obtainable from maximizing the similarity in the training and testing sets. The model achieves acceptable reconstruction of St George, which is expected given that the model is trained on similar data.

4.3 Other Experiments and Generalizability

This section presents other types of experiments. Generalizability describes a model's performance on unseen data, which is data that is not used in the model's training. The models' ability to reconstruct images from datasets they are not trained on is presented in this section. This means that images from the Compact Dataset are reconstructed using models trained on the Aviris dataset. The results will state how adaptable the different models are for entirely different hyperspectral images by testing them in a more relevant scenario. The images in the three databases Aviris, ICVL, and the Compact Dataset are of different characters. ICVL is the database least similar to the two others since it consists of images from an eye-level view of a city scene. The Compact Dataset and Aviris dataset images have the same point-of-view; they are all earth observational images. This section will therefore contain reconstructions of the Compact Dataset using Aviris models. Fig. 33 shows the pixel comparisons for pixels from reconstructed Salinas image using the Aviris CR 0,2 model.



(a) (b) (c)

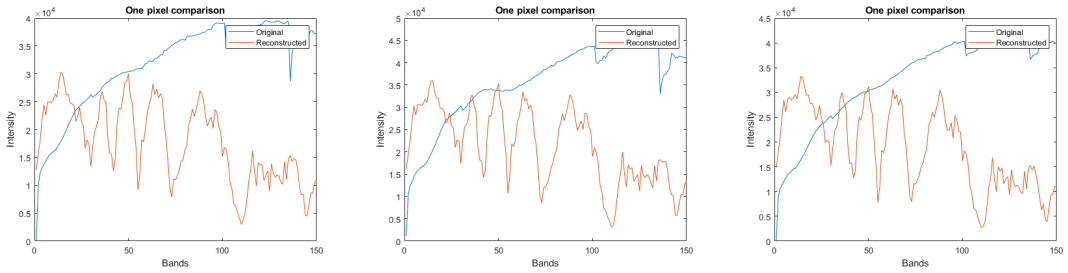
Figure 41: (a) *PSNR* = 16,0 dB and *SSIM* = 0,334 (b) *PSNR* = 22,5 dB and *SSIM* = 0,510 (c) *PSNR* = 22,1 dB and *SSIM* = 0,503.

Table 12 shows the *PSNR* and *SSIM* values from a reconstructed patch from the image Salinas from The Compact Dataset using Aviris CR 0,2 model.

PSNR	Value [dB]	SSIM	Value
Whole image	17,3	Whole image	0,427
Pixel average	18	Pixel average	0,383
Band average	21,2	Band average	0,724
MatLab function	17,3		

Table 12: *PSNR* and *SSIM* values of patch from Salinas image reconstructed with Aviris CR 0,2 model

Fig. 42 shows the pixel comparisons for pixels from reconstructed Cuprite image using the Aviris CR 0,2 model.



(a) (b) (c)

Figure 42: (a) *PSNR* = 10,8 dB and *SSIM* = 0,327 (b) *PSNR* = 9,9 dB and *SSIM* = 0,269 (c) *PSNR* = 10,1 dB and *SSIM* = 0,288.

Table 13 shows the *PSNR* and *SSIM* values from a reconstructed patch from the image Cuprite from The Compact Dataset using Aviris CR 0,2 model.

PSNR	Value [dB]	SSIM	Value
Whole image	10	Whole image	0,325
Pixel average	10	Pixel average	0,124
Band average	13,3	Band average	0,250
MatLab function	10		

Table 13: *PSNR* and *SSIM* values of patch from Cuprite image reconstructed with Aviris CR 0,2 model

Table 14 gives a summary of the models reconstruction performances, the model names are showed at the left of the table and and *PSNR*s and *SSIM* are showed accordingly at the right hand side. The *PSNR* values and *SSIM* values are given as the highest values obtained.

Table 14: Recovery of Compressed HSI Data Cubes : *PSNR* and *SSIM*

Data Set	Model Name	*PSNR* [dB]	SSIM	Recovered Bands [%]
ICVL	ICVL CR 0,2	31.7	0.837	80.0
	ICVL CR 0,6	31.6	0.822	40.0
	ICVL CR 0,2 Sparse	29.1	0.887	80.0
AVIRIS	Aviris CR 0,2	34.9	0.886	80.0
	Aviris CR 0,6	25.3	0.552	40.0
	Aviris CR 0,2 Sparse	34.8	0.886	80.0
Compact Dataset	Aviris CR 0,2	21.2	0.724	80.0

4.4 Discussion

This thesis has addressed the usage of U-Nets trained on existing datasets for reconstruction of compressive sensed hyperspectral images. Trying to generalize the models is important when dealing with machine learning. ICVL and AVIRIS are datasets which contains many high quality HSIs, making them suitable for training sets. This is visible in the reconstruction results from models trained on these datasets. The term "reconstructed images" is somewhat ambiguous because there is no boundary or limit deciding when a CS HSI can be called "reconstructed". The level of reconstruction is up to the user of a given observational system to decide. However, one should expect reconstruction results to be of a certain quality. Original signatures of substances and objects in scenes should be apparent in reconstructions so that classifying and detection tasks are not limited. The models trained using datasets from ICVL and AVIRIS can generalize to a point where the reconstruction follows the shape of the original pixels when reconstructing images within their datasets. The internal dataset contains images with similar scenes to those in the Aviris dataset. However, the results are rather poor when reconstructing images from the internal dataset using the Aviris models. The model cannot reconstruct the image, although the scenes are fairly similar. One reason for this could be the spatial resolution of the images. The Aviris dataset contains many high-resolution images, which give more prominent shapes and features than the images in the internal dataset. An image similar to St George, containing fields and mountain ridges, is reconstructed with *PSNR*s of 26 dB and *SSIM* of 0,846.

One of the goals of this project is to explore the possibility of using pre-trained models to reconstruct CS HSI for Earth observations. The expectations were high after reading about what the authors of DeepCubeNet are able to do with their U-Net architecture along with a snapshot imager. The goal is to get at least as good results as them and prove that the method can be used for Earth observational images, making it a candidate for future CubeSat-projects hyperspectral systems. The results do however not beat the current state-of-the-art model reconstructions. At one point, the results are close, but the model and its results can not be considered official results for this project due to the lack of reproducibility. The same results are not obtained when attempting to train a similar model. It is assured that the convincing first model is not trained on the test image. However, when using the same testing image, this time compressive sensed in a separate batch, the reconstructions are not as impressive, as the pixel spectra are shifted in the band domain. This suggests that the model suffered from overfitting to that particular compressive sensed batch.

4.5 Compressive Sensing

The CS in this project were achieved by following the CS theory and images were compressed in the spectral domain leaving the spatial resolution as is. The method for CS in this project distinguish it self from the method used in the original DeepCubeNet-paper, where they did compressive sensing as a spectral modulation due to their use of LC-cell phase retarders in the acquisition process. Here we also tried for a long time to figure out how to achieve this spectral modulation within or programs when trying to emulate the compressive sensing on established HSIs. This were harder than first thought, were several attempts were tried to get this to work. The way it were thought of was to use a transform domain at the spectral responsemap and that by following the same equations that were used in the DeepCubeNet project the compressive sensing would be done as spectral modulations. This were not successful and instead we decided to go for the more traditional CS. This could have an effect on the models ability to generalize, but is not confirmed since no experiments were done using spectral modulation method. In the future it would be highly interesting to explore this way of CS on HSIs and see the difference in results compared to this thesis. To our knowledge this would require a snapshot imager with CS integrated directly in the acquisition. The transform domains used in the CS were the *DFT* and *DCT*, these showed no noticeably difference in results, and in the end the *DCT* were used for all model training, because of the *DCT* domain not having any complex components to be sorted out. Keep in mind that the exploration of transform domains effect on compressive sensing were not of this project highest priority and therefore the difference in the CS domain were negligible for the two transform domains.

4.6 Compression Ratios

The CR is the most interesting parameter in CS in our minds. This decides how much data to keep from the original data, and then also how much data must be reconstructed. One would want to have the CR as low as possible and still manage to keep the quality of the reconstructions high. In this project we sat the benchmark of reconstructions at CR = 0,2, meaning that 80% of the original bands are discarded and needs to be recovered. The reason why the CR were set to 0,2 were that this were thought of being a achievable rate for the models to reconstruct HSIs. The state-of-the-art methods has sometimes lower than CR = 0,1, and authors of DeepCubeNet were using CR = 0,08. Since the project were to explore the possibilities of deep learning in the reconstruction of CS HSIs the aim where not set as high as CR = 0,1 and bellow, because getting results of reconstruction using deep learning was more important than achieving the best CR. However after proving reconstruction abilities on CR = 0,2 this could be for future improvements to try to see what the limit of CR could be for such a setup.

4.7 Sparsity

The sparsification of the data is a method used to make reconstruction "easier" for methods such as iterative algorithms searching for the sparsest solution. In this case when using deep learning methods sparsifying the data does not seam to have the larges impact on the performance of the models, even with the high sparsifying rate as used here. Here we tested the models by setting all frequency components higher than 30 to zero in the *DCT*-domain. The results were sometimes worse than the results atained at non-sparsified data. This could indicate that the pre-sparsification of the data does not have the same impact on model performance to reconstruct data as one could expect when comparing to iterative reconstruction methods such as OMP and Adaptive Gradient Descent. This is however a good thing to notice, bacause of the effect the pre-sparsification has to the data. By canceling out the higher frequency components one loses information about the scene, and spectral signatures will be distorted which can results in false predictions in segmentation and classification of substances in the HSI. As explained in the Theory natural data such as images and natural signals has already sparse characteristics. Some smaller sparsifications could filter out high frequency noise and make the image more smooth, but would have the disadvantage of training the models on images that are unrealistic in a real life scenario. In a real application one should expect distortions and noise in the images captured by a hyperspectral imager in space, whether its clouds or random noise from radiation in space.

4.8 Normalization and Artifacts

The ICVL dataset sparsified gives good results for reconstructing image of the same nature, using the testing image one can see that that the pixel is following somewhat the same shape as the original. The side-by-side view in *Hyperspectral Viewer* shows the patch-pairs built back togheter. One can clearly see the checker marks in the stitched image, this is because of the *exposure rescale intensity* function used in order to convert the data back to 16bit from the normalized version that contains values from 0-1. This function is done on each of the patches individually and therfore the rescale function will use the whole dynamic range for that particullarly patch, thats why in the original image we can also see the exposure levels are different in the tiles.

4.9 Other Problems

When testing different sizes of the network, one issue were more prominent than others, namely "OOM errors", where OOM stands for Out-Of-Memory. This error was showing whenever the network parameters increased, due to batch size increase, more training data or deeper U-Net configurations. By doing some research on the issue, revealed that this was a problem of large parameters when training networks. Hyperspectral data can quickly add up memory for the graphichs card and the internal RAM, and suggestions for avoiding these problems is to use two graphichs cards at once; one for training and one for validation. The workstation that were provided for this project had only one graphic card. The workaround for this problem were to use batch generators when training the model. These loads the desired batch size of data into the

model and prevent the memory to fill it self up by providing the model the whole dataset. This were a huge benefit, where we before got a lot of OOM-errors when testing higher batch sizes and larger datasets. In the current state of the project, the model is suffering of memory allocation problems when trying to train on bigger datasets. In future it would be of highly prioritization to fix this problem in order to see what bigger datasest would do to the generalization of the models. One should expect the model to perform better when trained on larger datsets as is proven by many deep learning methods before.

4.10 Overall

In some of the reconstruction instances the *PSNR* is in an adequate level but the reconstruction does not follow the original pixel wave well. When then looking at the *SSIM* values these shows low quality, this shows the importance of having different quality metrics when looking at compressions. One metric can focus on one aspect of the similarity between the two instances while the other can focus on another aspect. The DeepCubeNet were used in the paper *DeepCubeNet: reconstruction of spectrally compressive sensed hyperspectral images with deep neural networks*, to reconstruct images from ICVL database. In the paper they reconstructed 391 bands from 32 bands which were compressive sesed using a spectral modulation technique utilized by a liquid crystal phase retarder. They include pixel comparison plots between original and reconstructed pixel, with *PSNR*s between 38,4 dB and 41,8 dB. In this project we aimed to produce the same quality in reconstruction, and ended up with model *Aviris CR 0,2* giving *PSNR* of 34.9 dB. The compression ratio is less promising in this project with 80% discarded spectral bands compared to their 92%. If we are not comparing the results to results gained by Daniel Gedalin et. al. our trained model are capable of reconstructing the hyperpsectral image at a adequate quality. It even showed in one instance that data from other datasets were in fact “shape-reconstructed” in the spectral dimension. The reconstructed image would be useless for analysis of hazardous algae blooms if one look at the reconstruction of the Compact Dataset using Aviris model for example. If one expect the hyperspectral images delivered by a CubeSat to have the same quality as Aviris one should expect that the Aviris model would be able to reconstruct the images.

5 Conclusion

This project explored the potential of using deep learning methods for the reconstruction of compressive sensed hyperspectral images for CubeSats. Through this project, a U-Net architecture called DeepCubeNet was adapted for, and trained on existing hyperspectral Earth observational datasets such as AVIRIS. The trained model was able to reconstruct a compressive sensed hyperspectral image where 80% of the spectral bands were discarded. The reconstructed image was from the same database as the training images but is still characterized as unseen data for the model.

The reconstruction quality compared to the original image was measured by *PSNR*, *SSIM* and visual analysis. The reconstruction did not improve the current state-of-the-art methods for reconstructing compressive sensed hyperspectral imaging but shows the potential for using such a method for CubeSats, since one should expect similar images from the satellite as the one from AVIRIS. The project also managed to explore different public databases for hyperspectral images like AVIRIS and ICVL.

The trained models are able to reconstruct hyperspectral images from the same database as expected. The models are not able to reconstruct hyperspectral images that are from a different database source, and thus the ability of generalization of the model can be questioned. The compressive sensing was done in the spectral domain only, and the initial plan was to use compressive sensing as spectral modulation for the bands. However, this was unsuccessful, and the compressive sensing was done on a more traditional method.

The motivation for exploring compressive sensing reconstructions for hyperspectral images came from the research of CubeSats. The project presents a method that could be further developed for remote sensing CubeSats. This is shown from the reconstruction of St George hyperspectral image from the model trained on two images from AVIRIS datasets.

The U-net were also tested on the ICVL dataset, and the results showed that the models were able to reconstruct the images at an adequate level in terms of *PSNR*, *SSIM* and visualization. This did not beat the reconstructions from the original DeepCubeNet paper.

5.1 Further Work

Although the project managed to reconstruct compressive sensed hyperspectral images, the models could be drastically improved. Generalization should be further explored and would be the first thing to improve if the project was to be further worked on. At the end of the project period, there were attempts to train a model using far more data from the AVIRIS database. This contained 20 large hyperspectral images. Due to time limitations and errors regarding the size of the training data, the model was not successfully trained. This could have further improved the model and resulted in a more generalized model, due to the data being less hand-picked for the given model. The SmallSat Lab at NTNU was this thesis's project provider, and the research is a part of their HYPSO project. While writing this thesis, the HYPSO-1, a CubeSat launched by SmalSat Lab, was capturing hyperspectral images and transferring these back to SmallSat Lab. The images are not used in this thesis due to the fact that they are not publicly available at the time this thesis is being written. In the future, it would be interesting to see if the AVIRIS model would be able to reconstruct images from the HYPSO-1 satellite. An even more interesting approach would be to train a model on already existing images from the satellite to see if the images can be used to build a model used to reconstruct hyperspectral images from the next generation of earth observational CubeSats. The project's lowest compression ratio was 20% of data kept after compressive sensing. The compression ratios could be further explored to see what happens with the reconstructions at different steps. One should expect the results to get worse if one were to use the same dataset as used for the other models. The final stage for this project would be to build a fully connected system of snapshot imagers for CubeSats with integrated compressive sensing for capturing hyperspectral images, which could then be transferred down to be reconstructed by a trained U-Net architecture.

References

[1] Mariusz E. Grøtte et al. 'Ocean Color Hyperspectral Remote Sensing With High Resolution and Low Latency—The HYPSO-1 CubeSat Mission'. In: *IEEE Transactions on Geoscience and Remote Sensing* 60 (2022), pp. 1–19. DOI: 10.1109/TGRS.2021.3080175.

[2] Srdjan Stanković, Irena Orović and Ervin Sejdić. *Multimedia Signals and Systems: Basic and Advanced Algorithms for Signal Processing*. Springer, 2015.

[3] Liguo Wang and Chunhui Zhao. *Hyperspectral image processing*. Springer, 2016.

[4] Jon Álvarez Justo, Egil Eide and Milica Orlandić. 'Compressive Sensing on Three Dimensional SFCW Ground-Penetrating Radar'. In: *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. IEEE. 2020, pp. 1–6.

[5] Fei Zhang and Yan Piao. 'Design of Restoration Method Based on Compressed Sensing and TwIST Algorithm'. In: *Journal of Physics: Conference Series* 1004 (Apr. 2018), p. 012006. DOI: 10.1088/1742-6596/1004/1/012006. URL: https://doi.org/10.1088/1742-6596/1004/1/012006.

[6] Daniel Gedalin, Yaniv Oiknine and Adrian Stern. 'DeepCubeNet: reconstruction of spectrally compressive sensed hyperspectral images with deep neural networks'. In: *Opt. Express* 27.24 (Nov. 2019), pp. 35811–35822. DOI: 10.1364/OE.27.035811. URL: http://opg.optica.org/oe/abstract.cfm?URI=oe-27-24-35811.

[7] Morteza Mardani et al. 'Deep generative adversarial neural networks for compressive sensing MRI'. In: *IEEE transactions on medical imaging* 38.1 (2018), pp. 167–179.

[8] Xiaotong Lu et al. 'Convcsnet: A convolutional compressive sensing framework based on deep learning'. In: *arXiv preprint arXiv:1801.10342* (2018).

[9] James E. Fowler. 'Compressive pushbroom and whiskbroom sensing for hyperspectral remote-sensing imaging'. In: *2014 IEEE International Conference on Image Processing (ICIP)*. 2014, pp. 684–688. DOI: 10.1109/ICIP.2014.7025137.

[10] Isaac August et al. 'Miniature compressive ultra-spectral imaging system utilizing a single liquid crystal phase retarder'. In: *Scientific reports* 6.1 (2016), pp. 1–9.

[11] Ignacio Garcıa-Sánchez et al. 'Coded Aperture Hyperspectral Image Reconstruction'. In: *Sensors* 21.19 (2021), p. 6551.

[12] *NTNU Small Small Satellite Lab*. URL: https://www.ntnu.edu/ie/smallsat/project-overview.

[13] Redaksjon. *10 000 tonn død Laks Til Nå*. May 2019. URL: https://www.kyst.no/article/10-000-tonn-doed-laks-til-naa/.

[14] Av Redaksjonen. *Stabil sjømateksport i 2020 til tross for koronapandemien*. Jan. 2021. URL: https://fisk.no/fiskeri/7301-stabil-sjomateksport-i-2020-til-tross-for-koronapandemien.

[15] Boaz Arad and Ohad Ben-Shahar. 'Sparse Recovery of Hyperspectral Signal from Natural RGB Images'. In: *European Conference on Computer Vision*. Springer. 2016, pp. 19–34.

[16] *AVIRIS Data Portal 2006 - 2021*. URL: https://aviris.jpl.nasa.gov/dataportal/.

[17] Yaniv Oiknine et al. 'Compressive Sensing Hyperspectral Imaging by Spectral Multiplexing with Liquid Crystal'. In: *Journal of Imaging* 5.1 (2019). ISSN: 2313-433X. DOI: 10.3390/jimaging5010003. URL: https://www.mdpi.com/2313-433X/5/1/3.

[18] Srdjan Stanković et al. *Algorithms for compressive sensing signal reconstruction with applications*. 2016.

[19] Massimo Fornasier and Holger Rauhut. 'Compressive Sensing.' In: *Handbook of mathematical methods in imaging* 1 (2015), pp. 187–229.

[20] Biswa Nath Datta. *Numerical linear algebra and applications*. Vol. 116. Siam, 2010.

[21] Brunton & Kutz. *Data Driven Science & Engineering: Machine Learning, Dynamical Systems, and Control*. Brunton & Kutz, 2017.

[22] Charles Casimiro Cavalcante et al. 'Fast 1-Minimization Algorithms For Robust Face Recognition'. In: *Proceedings of the International Conference on Image Processing in*. 2010, p. 1.

[23] D.L. Donoho. 'Compressed sensing'. In: *IEEE Transactions on Information Theory* 52.4 (2006), pp. 1289–1306. DOI: 10.1109/TIT.2006.871582.

[24] Jian Wang, Seokbeop Kwon and Byonghyo Shim. 'Generalized Orthogonal Matching Pursuit'. In: *IEEE Transactions on Signal Processing* 60.12 (2012), pp. 6202–6216.

[25] Meenu Rani, S. B. Dhok and R. B. Deshmukh. 'A Systematic Review of Compressive Sensing: Concepts, Implementations and Applications'. In: *IEEE Access* 6 (2018), pp. 4875–4894. DOI: 10.1109/ACCESS.2018.2793851.

[26] Afonso S Bandeira et al. 'Certifying the restricted isometry property is hard'. In: *IEEE transactions on information theory* 59.6 (2013), pp. 3448–3450.

[27] Emmanuel J Candès and Michael B Wakin. 'An introduction to compressive sampling'. In: *IEEE signal processing magazine* 25.2 (2008), pp. 21–30.

[28] Stephen Boyd, Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[29] Eric W Weisstein. *Vector Norm.* URL: https://mathworld.wolfram.com/VectorNorm.html.

[30] John S Gero and Fay Sudweeks. *Artificial Intelligence in Design'96*. Springer Science & Business Media, 2012.

[31] Yann LeCun, Yoshua Bengio and Geoffrey Hinton. 'Deep learning'. In: *nature* 521.7553 (2015), pp. 436–444.

[32] *What is deep learning?: How it works, techniques & amp; applications*. URL: https://se.mathworks.com/discovery/deep-learning.html.

[33] Kunihiko Fukushima. 'Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.' In: 36 (1980), pp. 193–202. DOI: https://doi.org/10.1007/BF00344251.

[34] Qiao Zhang et al. 'Image segmentation with pyramid dilated convolution based on ResNet and U-Net'. In: *International conference on neural information processing*. Springer. 2017, pp. 364–372.

[35] Muhammad Mateen et al. 'Fundus image classification using VGG-19 architecture with PCA and SVD'. In: *Symmetry* 11.1 (2018), p. 1.

[36] Xavier Soria Poma, Edgar Riba and Angel Sappa. 'Dense Extreme Inception Network: Towards a Robust CNN Model for Edge Detection'. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. Mar. 2020.

[37] Md Zahangir Alom et al. 'The history began from alexnet: A comprehensive survey on deep learning approaches'. In: *arXiv preprint arXiv:1803.01164* (2018).

[38] Olaf Ronneberger, Philipp Fischer and Thomas Brox. 'U-net: Convolutional networks for biomedical image segmentation'. In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[39] *NET: Convolutional Networks for Biomedical Image Segmentation*. URL: https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/.

[40] Zhuokun Pan et al. 'Deep Learning Segmentation and Classification for Urban Village Using a Worldview Satellite Image Based on U-Net'. In: *Remote Sensing* 12.10 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12101574. URL: https://www.mdpi.com/2072-4292/12/10/1574.

[41] Peg Shippert et al. 'Why use hyperspectral imagery?' In: *Photogrammetric engineering and remote sensing* 70.4 (2004), pp. 377–396.

[42] *Hyperspectral viewer*. URL: https://se.mathworks.com/help/images/getting-started-with-hyperspectral-image-analysis.html.

[43] *PSNR*. URL: https://se.mathworks.com/help/vision/ref/psnr.html.

[44] *ssim*. URL: https://se.mathworks.com/help/images/ref/ssim.html.

[45] Mortimer Abramorwitz and Michael W. Davidson. *Optical Birefringence*. URL: https://www.olympus-lifescience.com/en/microscope-resource/primer/lightandcolor/birefringence/.

[46] Tao Zhang et al. 'Hyperspectral image reconstruction using deep external and internal learning'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 8559–8568.

[47] Kouhei Yorimoto and Xian-Hua Han. 'HyperMixNet: Hyperspectral Image Reconstruction with Deep Mixed Network from a Snapshot Measurement'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 1184–1193.

[48] Oleksii Sidorov and Jon Yngve Hardeberg. 'Deep hyperspectral prior: Single-image denoising, inpainting, super-resolution'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 2019, pp. 0–0.

[49] Frosti Palsson, Johannes R. Sveinsson and Magnus O. Ulfarsson. 'Multispectral and Hyperspectral Image Fusion Using a 3-D-Convolutional Neural Network'. In: *IEEE Geoscience and Remote Sensing Letters* 14.5 (2017), pp. 639–643. DOI: 10.1109/LGRS.2017.2668299.

[50] Tao Zhang et al. 'Hyperspectral Image Reconstruction Using Deep External and Internal Learning'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019.

[51] Xiaotong Lu et al. 'Convcsnet: A convolutional compressive sensing framework based on deep learning'. In: *arXiv preprint arXiv:1801.10342* (2018).

[52] Inchang Choi et al. 'High-Quality Hyperspectral Reconstruction Using a Spectral Prior'. In: *ACM Trans. Graph.* 36.6 (Nov. 2017). ISSN: 0730-0301. DOI: 10.1145/3130800.3130810. URL: https://doi.org/10.1145/3130800.3130810.

[53] Kouhei Yorimoto and Xian-Hua Han. 'HyperMixNet: Hyperspectral Image Reconstruction with Deep Mixed Network from a Snapshot Measurement'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 1184–1193.

[54] Vahid Abolghasemi et al. 'On optimization of the measurement matrix for compressive sensing'. In: *2010 18th European Signal Processing Conference*. IEEE. 2010, pp. 427–431.

[55] Sandeep Balachandran. *Machine learning - max pooling with color images*. Mar. 2020. URL: https://dev.to/sandeepbalachandran/machine-learning-max-pooling-with-color-images-2i21.

[56] Jason Brownlee. *Gentle introduction to the adam optimization algorithm for deep learning*. Jan. 2021. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

[57] Jason Brownlee. *Understand the impact of learning rate on neural network performance*. Sept. 2020. URL: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

[58] *Generalization; machine learning crash course; google developers*. URL: https://developers.google.com/machine-learning/crash-course/generalization/video-lecture.

Appendix

Making Datasets (MATLAB)

```
clc; clear all;

% give the path where the images are stored
filedir = 'C:\Users\NN\Desktop\Aviris\';

% gives lists of folder content
matfiles = dir(filedir);

t = 8; %length of the name list

% all file names are stored in cell
for i = 1:t
cell{i} = getfield(matfiles(i),'name');
end

% throw away the first two cells, these are "fill-ins"
cell(1:2) = [];

% waiting-bar to show the progress
wait = waitbar(0, 'Starting');

for l = 1:(t-2)
  % for the waitbar
  waitbar(l/20, wait, sprintf('Progress: %d %%', floor(l/20*100)));
  pause(0.1);

  % file path for the file
  file = append(filedir,cell{l});

  % loading the file
  folder = load(file);

  % extract the cube
  data = folder.cube;
  [x,y,z] = size(data);

  % normalize the spectral band
  k = z/150;
  data = data(:,:,1:k:end);

  [x y z] = size(data);

  % define the amount of patches in rows and columns from the original
  % image
  data_rad = floor(x/64);
  data_col = floor(y/64);
  jump=64;

  % devide the cube into vectors of each pixel
  for i=1:x
  for j=1:y
  f{i,j}=permute(data(i,j,1:150),[3 2 1]);
```

```
  end
  end

  % define the N and M for the CR
  N = 150;
  M = 30;

  % DCT-transfrom matrix
  B = dctmtx(N);
  Binv = inv(B);

  % selecting random rows of the DCT matrix
  q = randperm(N);
  q = q(1:M);
  q = sort(q);

  % sensing matrix A
  A = Binv(q,:);

  % CS is performed on each of the pixels in the spectral domain
  for i=1:x
  for j=1:y
    xf{i,j} = B*double(f{i,j});% taking dft of the signal
    ff = xf{i,j};
    tf = A*ff;
    tt{i,j} = tf;

  end
  end

  % the cs pixel vectors are re-aranged back to cubes
  Z = cellfun(@(x)reshape(x,1,1,[]),tt,'un',0);
  o = cell2mat(Z);

  % the gt pixel vectors are re-aranged back to cubes
  Y = cellfun(@(x)reshape(x,1,1,[]),f,'un',0);
  data = cell2mat(Y);
  % this is to have both cs and gt be the same class
  d = double(data);

cd(filedir)

% making foldernames for the images in order to create cs-folder and
% gt-folder
foldername = cell{l};
foldername=erase(foldername,'.mat');

% making folders
mkdir(foldername)
path = append(filedir,foldername);
cd(path)

sub_folder = ['gt','cs'];
  % storing d for data into gt and o for out into cs
  for a = 1:2
    mkdir('gt')
    mkdir('cs')
    counter = 0;
```

```
    if a == 1
    % build patches of 64x64x150 from the gt image
    for g=1:data_rad
      for h=1:data_col
        gt = d((g*jump)-(jump-1):64+(g*jump)-jump,...
        (h*jump)-(jump-1):64+h*jump-jump,...
        :);
        counter = counter+1;
        % define automatic cube-names
        if counter < 10
          filename_i = sprintf('cube_000%.f',counter);
        elseif counter < 100
          filename_i = sprintf('cube_00%.f',counter);
        elseif counter >= 100
          filename_i = sprintf('cube_0%.f',counter);
        elseif counter >= 1000
          filename_i = sprintf('cube_%.f',counter);

        end
          % store the patch into the folder gt
          filename = filename_i;
          save(['gt\', filename_i, '.mat'] ,'gt');

      end
    end
    end
    if a == 2
      % build patches of 64x64x30 from the cs image
      for g=1:data_rad
      for h=1:data_col
        gt = o((g*jump)-(jump-1):64+(g*jump)-jump,...
        (h*jump)-(jump-1):64+h*jump-jump,...
        :);
        counter = counter+1;
        % define automatic cube-names
       if counter < 10
          filename_i = sprintf('cube_000%.f',counter);
        elseif counter < 100
          filename_i = sprintf('cube_00%.f',counter);
        elseif counter >= 100
          filename_i = sprintf('cube_0%.f',counter);
        elseif counter >= 1000
          filename_i = sprintf('cube_%.f',counter);
       end
          % store the patch into the folder cs
          filename = filename_i;
          save(['cs\', filename_i, '.mat'] ,'gt');
      end
      end

    end
  end
% emptying f and tt for next iteration
f = {};
tt = {};
  end
% wait-bar is closed when last image is done
close(wait)
```

Stacking Reconstructed Patches (MATLAB)

```
clc; clear all;

% set the number of rows and columns according to number of patches in rows
% and columns in the image
rows = ;
col = ;

% define the reconstructed images folder path
filedir = 'C:\Users\NN\Desktop\Avirs CR 0.2 model outputs\'
matfiles = dir(filedir);

% number of elements in the dir list
t = ;

% store the file names in cell array
for i = 1:t
cell{i} = getfield(matfiles(i),'name')
end

% delete two first cells which are fill-ins
cell(1:2) = [];

% define the first cube of the reconstructed cubes
cube_i = load('C:\Users\NN\Desktop\Avirs CR 0.2 model outputs\cube_001.mat');
cube_i = cube_i.org;

cube1 = 0;
var = 1;
for m = 1:rows

%Stacking patches back together to full image
f_name = append(filedir,cell{1+col*(m-1)});
cube_i = load(f_name);
cube_i = cube_i.org;
  for n = 2:col
    filename = cell{(m-1)*col+n}
    f_name = append(filedir, filename);
    cube = load(f_name);
    cube_ = cube.org;
    cube_i = cat(2,cube_i, cube_);
  end
  if var
    var = 0;
    cube1 = cube_i;
  else
    % cube 1 is the full image
    cube1 = cat(1,cube1, cube_i);

  end
end
```

Training The Model (Python)

```
from encodings import normalize_encoding
from tabnanny import verbose
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
import tensorflow as tf
import time
import os
import numpy as np
import scipy.io
import mat73
#import Unet as Unet
#import Unet2 as Unet
import Unet3 as Unet
#import Unet4 as Unet
from skimage import exposure, img_as_ubyte, img_as_uint
from tensorflow.keras.layers import UpSampling3D #upsampling layers for 3D
 ↪  inputs
from net_blocks import *
from utils import *
from Unet import *
from scipy.io import loadmat
from tqdm import tqdm
from keras.callbacks import TensorBoard
from tensorflow.compat.v1 import ConfigProto
from tensorflow.compat.v1 import InteractiveSession


# define the patch dimensions
IMG_WIDTH = 64
IMG_HEIGHT = 64
IMG_CHANNELS_CS = 30
IMG_CHANNELS_GT = 150

#change folder-name for choosing the right training data
train_path = 'Big_model_training/'

length = 0
counter = 0

# find the folder names
datasets = next(os.walk(train_path))[1]
t_train = []

# go through each folder
for i in range(len(datasets)):

    # /Training_data/set1 training_path = /set1/
    train_path_ = train_path + datasets[i] + '/'
    train_ids = next(os.walk(train_path_))[1]
    cs_path = train_path_ + train_ids[0] + '/'
    gt_path = train_path_ + train_ids[1] + '/'


    train_ids_cs = next(os.walk(cs_path))[2]
    train_ids_gt = next(os.walk(gt_path))[2]
```

```
    length += len(train_ids_cs)

    # define zero_arrays to insert training data
    x_train = np.zeros(((length), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS_CS),
     ↪  dtype=int)
    y_train = np.zeros(((length), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS_GT),
     ↪  dtype=int)

    # define zero_arrays to insert scaled data to uint16
    x_train_uint16 = np.zeros(((length), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS_CS),
     ↪  dtype=np.uint16)
    y_train_uint16 = np.zeros(((length), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS_GT),
     ↪  dtype=np.uint16)


for i in range(len(datasets)):
    #walk through the folder, give the folder name plus the file name
    train_path_ = train_path + datasets[i] + '/'

    train_ids = next(os.walk(train_path_))[1]
    cs_path = train_path_ + train_ids[0] + '/'
    gt_path = train_path_ + train_ids[1] + '/'

    train_ids_cs = next(os.walk(cs_path))[2]
    train_ids_gt = next(os.walk(gt_path))[2]

    counter = counter

    # load the .mat files (patches) into x_train and y_train
    for j, id_ in tqdm(enumerate(train_ids_cs), total=len(train_ids_cs)):
        path = cs_path

        # load the .mat patch
        cs = mat73.loadmat(path + id_)

        counter = counter + 1
        x_tmp = cs['gt']

        # change data to integer
        x_tmp = x_tmp.astype(int)

        # insert array from .mat into x_train
        x_train[(counter-1)] = x_tmp

        # do the same for y_train as for x_train
        path = gt_path
        gt = mat73.loadmat(path + id_)
        y_tmp = gt['gt']
        y_tmp = y_tmp.astype(int)
        y_train[(counter-1)] = y_tmp

# set values that are negative to zero for y_train
for u in tqdm(range(y_train.shape[0])):
    for i in range(y_train.shape[1]):
        for j in range(y_train.shape[2]):
            for k in range(y_train.shape[3]):
                if(y_train[u,i,j,k] < 0):
                    y_train[u,i,j,k] = 0
```

```
# set values that are negative to zero for x_train
for u in tqdm(range(x_train.shape[0])):
    for i in range(x_train.shape[1]):
        for j in range(x_train.shape[2]):
            for k in range(x_train.shape[3]):
                if(x_train[u,i,j,k] < 0):
                   x_train[u,i,j,k] = 0


x_train_uint16 = np.zeros((x_train.shape[0], x_train.shape[1], x_train.shape[2],
 ↪  x_train.shape[3]), dtype=np.uint16)
y_train_uint16 = np.zeros((y_train.shape[0], y_train.shape[1], y_train.shape[2],
 ↪  y_train.shape[3]), dtype=np.uint16)

# exposure.rescale_intensity for x_train which then is stored as x_train_uint16
for i in tqdm(range(x_train.shape[0])):
    x_train_uint16[i] = img_as_uint(exposure.rescale_intensity(x_train[i]))
# exposure.rescale_intensity for y_train which then is stored as y_train_uint16
for k in tqdm(range(y_train.shape[0])):
    y_train_uint16[k] = img_as_uint(exposure.rescale_intensity(y_train[k]))

# normalize x_train_uint16 and y_train_uint16 between 0-1
x_train_uint16_norm = x_train_uint16 / 65535
y_train_uint16_norm = y_train_uint16 / 65535

# return a list of physical devices visible to the host runtime, here GPU
physical_devices = tf.config.list_physical_devices('GPU')
for dev in physical_devices:
    print(dev)

# helps for OOM errors
config = ConfigProto()
config.gpu_options.allow_growth = True
session = InteractiveSession(config=config)
# resets all state generated by Keras
tf.keras.backend.clear_session()

# split training into training and validataion set
x_train, x_test, y_train, y_test = train_test_split(
    x_train_uint16_norm,
    y_train_uint16_norm,
    test_size=0.2,
    random_state=5,
    )

# generator for loading batches of data into the model
class DataGenerator(tf.keras.utils.Sequence):
    def __init__(self, x_set, y_set, batch_size):
        self.x, self.y = x_set, y_set
        self.batch_size = batch_size

    def __len__(self):
        return int(np.ceil(len(self.x) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch_x = self.x[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.y[idx * self.batch_size:(idx + 1) * self.batch_size]
```

```
        return batch_x, batch_y

# defines the input to the generators, 2 referres to batch size of 2
train_gen = DataGenerator(x_train, y_train, 2)
test_gen = DataGenerator(x_test, y_test, 2)

# define the Unet to use
model = Unet.get_model_3()

# loggs tensorboard training graphs
NAME = "Hyp-Reconstruction-unet3-aviris-model_name-{}".format(int(time.time()))
tensorboard = TensorBoard(log_dir='new-logs/logs/{}'.format(NAME))

# define early-stopping parameters
callbacks = [
        tf.keras.callbacks.EarlyStopping(patience = 10, monitor = 'val_loss'),
        tensorboard]

# train the model using model.fit with train_gen and test_gen as inputs
history = model.fit(train_gen,
                    epochs=30,
                    validation_data=test_gen)
# give the model an appropriate name
model.save("model_43")
```

Testing The Model (Python)

```
# define the path to the cs patches
test_path = 'cuprite_30/cs/'
# define where the reconstructed patch will be saved
output_path = 'model_40_outputs/cuprite/rec/'

# get array with all file names
testing_data = next(os.walk(test_path))[2]

# reconstruct every patch in the folder
for i in tqdm(range(len(testing_data))):
    data = test_path + testing_data[i]

    # define empty array
    test = np.zeros((64, 64, 30), dtype=int)

    # load data into tmp
    # and test after casting to integers
    tmp = mat73.loadmat(data)
    tmp = tmp['gt']
    test = tmp.astype(int)

    # set negative values to 0
    for u in range(test.shape[0]):
        for t in range(test.shape[1]):
            for j in range(test.shape[2]):
                    if(test[u,t,j] < 0):
                        test[u,t,j] = 0

    # use exposure.rescale_intensity for data inside test
    test = img_as_uint(exposure.rescale_intensity(test))

    # normalize the test data between 0 and 1
    test = test/65535

    # define the pre-trained model to use for reconstruction
    model = tf.keras.models.load_model("model_40", custom_objects = ({'psnr':
     ↪  psnr, 'SSIM': SSIM}))

    # change the dimenion of the input data
    test = np.expand_dims(test, axis=0)

    # rec is the reconstructed patch from test
    rec = model.predict(test)

    # delete the first dimension which are made by the model
    rec = rec[0, :, :, :]

    # scale the patch back to 16-bit values
    rec = rec * 65535

    # set the output to uint16 datatype
    rec = rec.astype(np.uint16)

    # defines the name according to output path and the patch name
    file_path = output_path + testing_data[i]
```

```
    # patch is saved as .mat file
    scipy.io.savemat(file_path, {'rec': rec})

# Now we need to do the same scaling and casting procedure for the original data
 ↪  in order for the
# comparisons between reconstructed and original patches to be fair

org_path = 'cuprite_30/gt/'

output_path = 'model_40_outputs/cuprite/org/'

print(output_path)
orginal_data = next(os.walk(org_path))[2]

for i in tqdm(range(len(orginal_data))):
    data = org_path + orginal_data[i]

    test = np.zeros((64, 64, 30), dtype=int)

    tmp = mat73.loadmat(data)
    #tmp = mat73.loadmat('cuprite_set/30_no_overlap/cs/cube_1_1.mat')
    tmp = tmp['gt']
    test = tmp.astype(int)

    #test= np.round(tmp['gt'])
    #test = test.astype(int)
    for u in range(test.shape[0]):
        for t in range(test.shape[1]):
            for j in range(test.shape[2]):
                    if(test[u,t,j] < 0):
                        test[u,t,j] = 0

    org = img_as_uint(exposure.rescale_intensity(test))

    file_path = output_path + orginal_data[i]

    scipy.io.savemat(file_path, {'org': org})
```

DeepCubeNet (Python)

```
from tabnanny import verbose
import tensorflow as tf
import time
import os
import numpy as np
import scipy.io
from tensorflow.keras.layers import UpSampling3D
from net_blocks import *
from utils import *

# the U-Net model based on DeepCubeNet
def get_model_3():
    input1 = tf.keras.layers.Input(shape=(64, 64, 30), batch_size=None,
    ↪  name='input1')

    bp = tf.keras.layers.Conv2D(150,(1,1), activation=None, trainable=False,
    ↪  use_bias=False, name='bp1')(input1)
    # the backprojection from CS domain to HS domain
    bp_exp = tf.keras.layers.Lambda(expand)(bp)
    # the contracting path
    # 3D convolutional layers
    d_1 = conv3D_block(8,(3,3,11), input = bp_exp) #8 filters
    d_1 = conv3D_block(8,(3,3,11), input = d_1) #8 filters
    # 3D max-pooling
    d_1_p = tf.keras.layers.MaxPooling3D((1,1,2))(d_1)
     # 3D convolutional layers
    d_2 = conv3D_block(16,(3,3,9), input = d_1_p) #16 filters
    d_2 = conv3D_block(16,(3,3,9), input = d_2) #16 filters
    # 3D max-pooling
    d_2_p = tf.keras.layers.MaxPooling3D((1,1,2))(d_2)
    # 3D convolutional layers
    d_3 = conv3D_block(32,(3,3,7), input = d_2_p) #32 filters
    d_3 = conv3D_block(32,(3,3,7), input = d_3) #32 filters
    # 3D max-pooling
    d_3_p = tf.keras.layers.MaxPooling3D((1,1,2))(d_3)
    # 3D convolutional layers
    d_4 = conv3D_block(128,(3,3,5), input = d_3_p) #128 filters
    d_4 = conv3D_block(128,(3,3,5), input = d_4) #128 filters

    # the expanding path
    # 3D up-sampling layer
    u_1 = UpSampling3D((1,1,3))(d_4)
    # cropping
    u_1 = tf.keras.layers.Cropping3D(cropping=((0,0),(0,0),(0,17)))(u_1) #17
    # 3D convolutional layer
    u_1 = conv3D_block(32,(3,3,7), input = u_1) #32 filters
    # 3D convolutional layer + concatination
    u_1 = conv3D_block(32,(3,3,7), input = u_1, concat = d_3) #32 filters
    # 3D up-sampling layer
    u_2 = UpSampling3D((1,1,2))(u_1)
    # cropping
    d_2_c = tf.keras.layers.Cropping3D(cropping=((0,0),(0,0),(0,1)))(d_2)
    ↪  #cropping
    # 3D convolutional layer
    u_2 = conv3D_block(16,(3,3,9), input = u_2) #16 filters
    # 3D convolutional layer + concatination
```

```
u_2 = conv3D_block(16,(3,3,9), input = u_2, concat = d_2_c) #16 filters +
 ↪  concat
# 3D up-sampling layer
u_3 = UpSampling3D((1,1,3))(u_2)
# cropping
u_3_c = tf.keras.layers.Cropping3D(cropping=((0,0),(0,0),(0,72)))(u_3)
 ↪  #cropping
 # 3D convolutional layer
u_3 = conv3D_block(8,(3,3,11), input = u_3_c) #8 filters
 # 3D convolutional layer + concatination
u_3 = conv3D_block(8,(3,3,11), input = u_3, concat = d_1) #8 filters +
 ↪  concat

# Final Projection
pr = conv3D_block(1,(1,1,1), input = u_3)
# Creates the final HSI with initial dimensions
final = tf.keras.layers.Lambda(squeeze)(pr)

# defines the model
model = tf.keras.models.Model(input1, final)
# define optimizer
opt = tf.keras.optimizers.Adam(learning_rate = 0.0001, name = 'Adam')
# compile model
model.compile(optimizer = opt, loss='mse', metrics=[psnr,'mse','mae',SSIM])

return model
```

Utils for DeepCubeNet (Python)

```
import numpy as np
from hdf5storage import loadmat,savemat
from scipy.interpolate import griddata
import os
from tqdm import tqdm
import keras
import tensorflow as tf
import keras.backend as K
import skimage

# these are utils from the DeepCubeNet from GitHub
# these defines the calculations of SSIM and PSNR
def psnr(y_true, y_pred):
    return tf.image.psnr(y_true, y_pred, max_val=K.max(y_true))
def cos_distance(y_true, y_pred):
    def l2_normalize(x, axis):
        norm = K.sqrt(K.sum(K.square(x), axis=axis, keepdims=True))
        return K.maximum(x, K.epsilon()) / K.maximum(norm, K.epsilon())

    y_true = l2_normalize(y_true, axis=-1)
    y_pred = l2_normalize(y_pred, axis=-1)
    return K.mean(K.sum(y_true * y_pred, axis=-1))
def relRMSE(y_true,y_pred):
    true_norm = K.sqrt(K.sum(K.square(y_true), axis=-1))
    return K.mean(K.sqrt(keras.losses.mean_squared_error(y_true,
    ↪  y_pred))/true_norm)
def SSIM(y_true,y_pred):
    return tf.image.ssim(y_pred,y_true,K.max(y_true))
def spectral_TV(y_true,y_pred):
    return K.mean(K.mean(K.sqrt(K.square(y_pred[:, :, :, 1:] - y_pred[:, :, :,
    ↪  :314]))))
```

Layer Definitions DeepCubeNet (Python)

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import InputSpec, Layer

# these are definitions for the U-Net layers: 3D conv, expand,
 ↪  squeeze
# these are functions used in the original DeepCubeNet which is on
 ↪  GitHub
class ReflectionPadding3D(Layer):
    def __init__(self, padding=(1, 1 ,1), **kwargs):
        self.padding = tuple(padding)
        self.input_spec = [InputSpec(ndim=5)]
        super(ReflectionPadding3D, self).__init__(**kwargs)

    def compute_output_shape(self, s):
        """ If you are using "channels_last" configuration"""
        return (s[0], s[1] + 2 * self.padding[0], s[2] + 2 *
         ↪  self.padding[1], s[3]+2*self.padding[2],s[4])
```

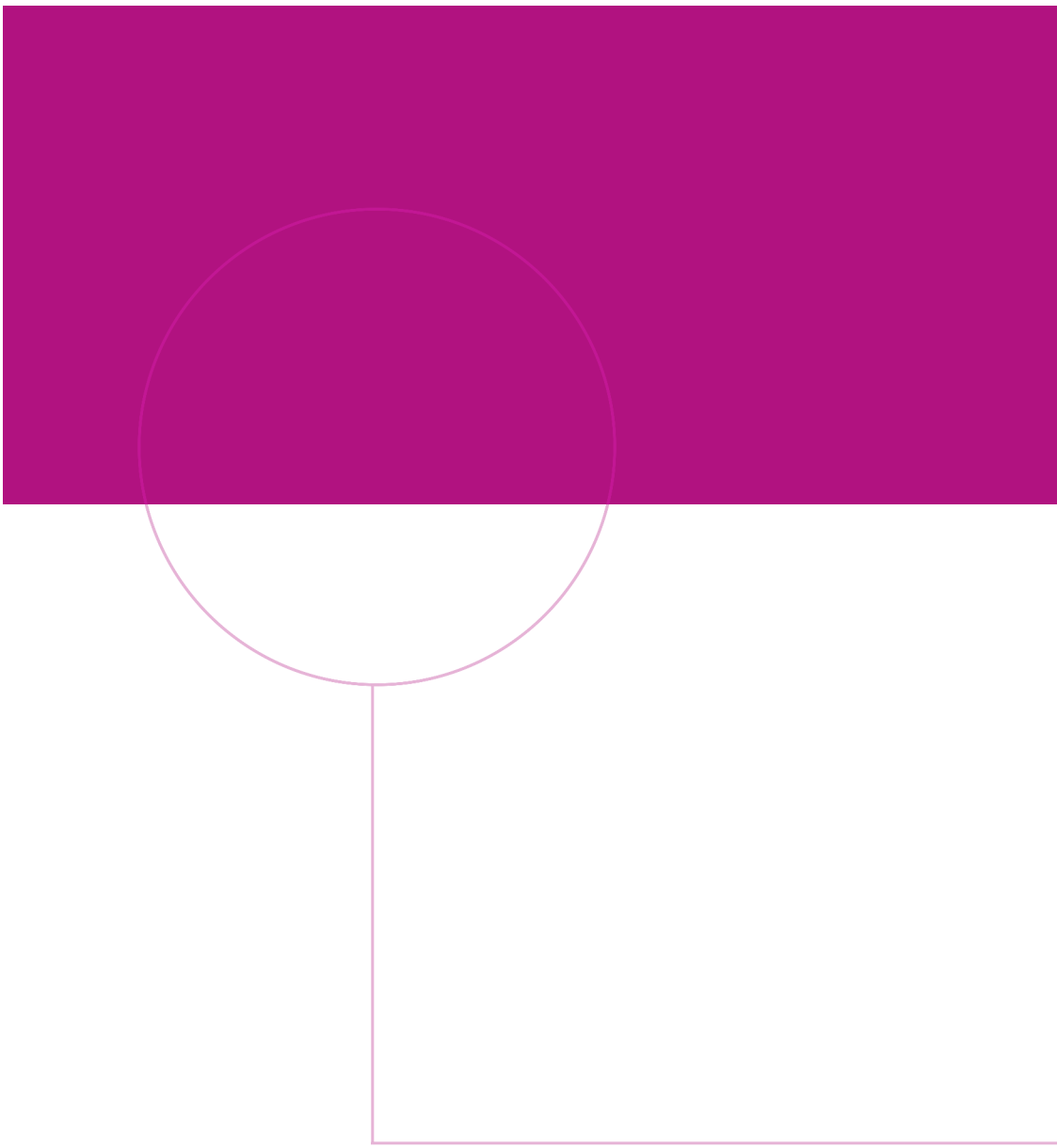
```
    def call(self, x, mask=None):
        w_pad,h_pad,z_pad = self.padding
        return tf.pad(x, [[0,0], [h_pad,h_pad], [w_pad,w_pad] ,
         ↪  [z_pad,z_pad], [0,0] ], 'REFLECT')

def conv3D_block(n_filters,kernel,input,stride=1,name=None,concat =
 ↪  None,dropout=0,bias_reg=None):
    if concat != None:
        input= tf.keras.layers.Concatenate()([input,concat])
    x = ReflectionPadding3D((int(np.floor(kernel[0]/2)),
     ↪  int(np.floor(kernel[1]/2)),
     ↪  int(np.floor(kernel[2]/2))))(input)
    x = tf.keras.layers.Conv3D(n_filters, kernel,
     ↪  padding='valid',strides=stride, name=name,
     ↪  activation=None,bias_regularizer=bias_reg)(x)
    x = tf.keras.layers.Activation('relu')(x)
    if dropout!=0:
       x = tf.keras.layers.Dropout(dropout)(x)

    return x

def squeeze(x):
    import keras
    return keras.backend.squeeze(x,axis=-1)

def expand(x):
    import keras
    return keras.backend.expand_dims(x,axis=-1)
```



NTNU

Norwegian University of
Science and Technology