Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

Helene Markeng

Exploratory Pipeline for Reconstructing Compressive Sensed Images and Small Cloud Removal using 3D convolutional U-nets

Masteroppgave i Elektronisk systemdesign og innovasjon
Veileder: Milica Orlandic
August 2023

NTNU
Kunnskap for en bedre verden

Helene Markeng

Exploratory Pipeline for Reconstructing Compressive Sensed Images and Small Cloud Removal using 3D convolutional U-nets

Masteroppgave i Elektronisk systemdesign og innovasjon
Veileder: Milica Orlandic
August 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for elektroniske systemer

NTNU
Kunnskap for en bedre verden

Exploratory Pipeline for Reconstructing Compressive Sensed Images and Small Cloud Removal using 3D convolutional U-nets

Helene Markeng

August, 2023
Version: 1.0

Norwegian University of Science and Technology

NTNU

Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Signal processing and machine learning

Exploratory Pipeline for Reconstructing Compressive Sensed Images and Small Cloud Removal using 3D convolutional U-nets

Helene Markeng

Reviewer Milica Orlandic
Department of electric systems
Norwegian University of Science and Technology, NTNU

Supervisor Milica Orlandic

August, 2023

Helene Markeng
Exploratory Pipeline for Reconstructing Compressive Sensed Images and Small Cloud Removal using 3D convolutional U-nets
Signal processing and machine learning August, 2023
Reviewer: Milica Orlandic
Supervisor: Milica Orlandic
Norwegian University of Science and Technology, NTNU

Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems
Høgskoleringen 1
7034, Trondheim

Abstract

This study examines a pipeline for reconstructing compressive sensed images while simultaneously removing small clouds. The research encompasses various subsystems designed and tested to address the reconstruction, cloud detection, cloud removal, and missing area reconstruction tasks, with the U-Net as an essential tool. The primary objective is to investigate potential solutions within the context of HYPSO-1 images.

Acknowledgement

I would like to express gratitude to Milica Orlandic, my supervisor and reviewer, for providing me with guidance and suggesting the project topic. Additionally, I am grateful to the Small-Sat team at NTNU for granting me access to the GPU, and sharing their inspiring perspectives on the project. Finally, I want to thank Jon Alvarez Justo for collaborating with me on the utilization of ML and for our discussion on compressive sensing.

Contents

List of Figures

List of Tables

1 Introduction

Helene Markeng
Håkon Jarls gate 2
7030 Trondheim
Norway

1.1 Motivation and Problem Statement

The use of hyperspectral images (HSIs) obtained from satellites is becoming increasingly important in various fields such as agriculture, environmental monitoring, and remote sensing. However, HSIs are usually data-intensive and high-dimensional, which makes it challenging to store, process, and transmit the data. To address these challenges, compressive sensing (CS) has emerged as a promising technique to obtain sparse measurements of the HSI, which allows for efficient data compression and transmission. In CS, the reconstruction of the original HSI from the sparse measurements is a crucial step. Traditional reconstruction methods often use complex algorithms that can be computationally demanding and time-consuming. To overcome these limitations, deep learning techniques such as U-Net have gained considerable attention. U-Net is a convolutional neural network that can capture and make use of the spatial and spectral correlations within HSIs, thereby improving the speed and quality of HSI reconstruction.

In addition to the challenges posed by data transmission and reconstruction, cloud cover can significantly impact the usability of hyperspectral images. Clouds can obscure the underlying features and make the acquired data unreliable for further analysis. Therefore, an essential task in HSI processing is cloud detection and removal. Accurate detection and removal of clouds can enhance the quality and reliability of the reconstructed HSIs, allowing more accurate analysis and interpretation of the data.

This thesis aims to investigate how compressive sensing, U-Net reconstruction, and cloud removal techniques can be used together to improve the speed, efficiency, and usability of hyperspectral images taken by satellite platforms. Specifically, the focus is on HSIs captured by the HYPSO-1 satellite, which aims to capture ocean-color images to support marine research. By using deep learning and cloud removal algorithms, the goal is to investigate the feasibility of a pipeline that can effectively detect and remove clouds from compressed measurements, resulting in faster and more reliable reconstruction of HSIs. To accomplish this, expertise in multiple technical fields is required. Therefore, the pipeline will be divided into separate components and studied individually and in combination with other parts to produce informative results that can pave the way for future advancements and improvements.

1.2 Thesis Structure

Chapter 2 - Concepts

In this chapter, I will give a detailed explanation of the theories and concepts that form the basis of the research presented in this thesis. Additionally, to avoid any confusion, I will clarify any abbreviations used throughout the work to ensure a common understanding.

Chapter 3 - Related work

This thesis builds upon prior research and incorporates numerous techniques. In this chapter, we examine relevant studies in the field, outlining their main discoveries and how they relate to our current research.

Chapter 4 - System Description

This chapter provides an overview of the system implementation. It includes the decisions made during the project, which were guided by the theory and related research in Chapter 1 and 2. The tools, dataset and code used for the system implementation are also described, as well as various approaches.

Chapter 5 - Results and discussion

This chapter reviews and analyzes the results achieved through various methods assessed in the study. Additionally, errors will be investigated to gain a better comprehension of the causes of any inconsistencies and pinpoint possible opportunities for enhancement.

Chapter 6 - Conclusions and future work

In this chapter, the study's primary discoveries and conclusions are summarized, emphasizing the significant outcomes. Additionally, potential directions for future research are examined, taking into account any shortcomings of the present study and possible areas for additional investigation.

2 Concepts

This chapter contains several technical concepts that are crucial for comprehending the studied system, with the purpose of providing a theoretical basis for the research presented in the thesis. By explaining these concepts in detail and establishing a shared vocabulary, the reader will be able to fully understand the system and the research presented in the thesis. In addition, this chapter will introduce pertinent research that serves as both a source of inspiration and a basis for decision-making.

2.1 Remote sensing and hyperspectral images.

Remote sensing is a scientific field that revolves around the collection, processing, and analysis of information concerning the Earth's surface and atmosphere using sensors deployed on satellites and aircraft. This technology enables researchers and scientists to investigate and monitor the Earth from a distance, eliminating the need for direct ground-based data collection. Remote sensing sensors fall into two main categories: passive and active sensors. Passive sensors, like cameras and spectrometers, detect and measure the natural energy emitted or reflected by the Earth's surface and atmosphere. On the other hand, active sensors, including radar and lidar, emit energy and capture the energy reflected back from the Earth's surface and atmosphere. Remote sensing data has a wide range of applications, such as studying land use, vegetation, hydrology, meteorology, and climate. [1].

2.1.1 Hyperspectral imaging

Hyperspectral images (HSI) are a type of image that captures information across a wide range of the electromagnetic (EM) spectrum. HSIs can be used as remote sensing data by capturing the reflectance of the Earth's surface in many narrow, contiguous wavelength bands, typically between 400-2500 nm [2]. This allows for identifying and mapping materials and features on the Earth's surface based on their unique spectral signatures. One pixel with all its samples across the electromagnetic spectra is called a spectral vector, a spectral array, or a pixel array. The frequency bands are usually the same for all spectral vectors in all images in one data set. Unlike RGB images, which contain three frequency bands (Red, Green, and Blue), HSIs can contain hundreds of bands. As illustrated in Fig. 2.1, an HSI has dimensions x and y in the spatial domain and dimension z in the spectral domain, making it a three-dimensional cube or tensor.

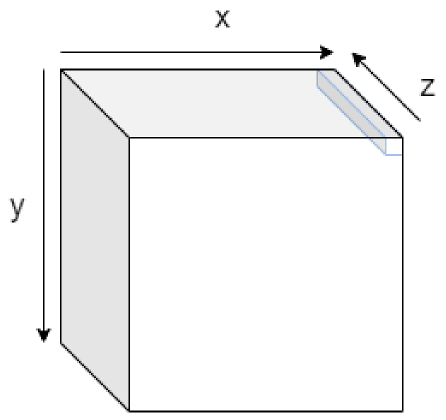


Fig. 2.1: A simple representation of a HSI cube, with dimensions x, y, and z. A spectral vector of one pixel is marked in light blue.

An HSI cube can be represented in RGB format by selecting bands corresponding to red, green, and blue frequencies. Examples of an HSI cube and its corresponding RGB representation are shown in Fig. 5.20a and 2.3, respectively.

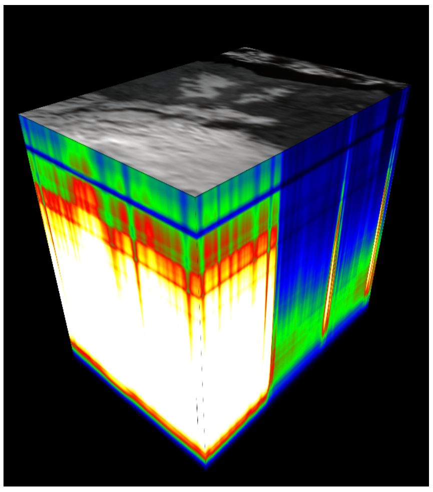


Fig. 2.2: HSI cube containing 120 bands. Note that the image is rotated compared to the image above so that the z-axis is pointing downwards.

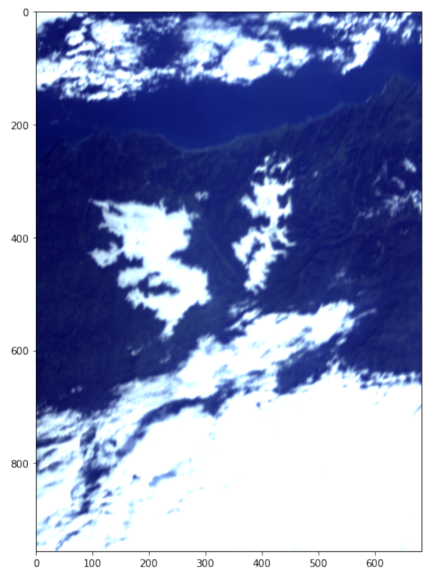


Fig. 2.3: RGB representation of HSI cube.

2.1.2 Cloud contamination

Clouds pose a significant challenge when conducting earth observation through remote sensing using satellite images due to their opaque nature. They can obstruct the visibility of ground objects and seamlessly blend with the underlying details, making them a major obstacle to accurate analysis and interpretation. The spectral signature of clouds can vary depending on factors such as cloud type, thickness, and altitude. For example, due to their ice crystals, high-altitude clouds tend to have a strong spectral signature in the mid-infrared wavelengths. In contrast, low-altitude cumulus clouds tend to have a strong signature in the visible and near-infrared wavelengths due to their water droplets [3]. Clouds can also affect the spectral signature of the underlying surface by reflecting or absorbing radiation, which can complicate the analysis of remote sensing data. A clouded image can be modeled using a cloud physical model using parameters such as atmospheric light, the reflectance of the ground object, cloud transmission, and the sunlight's attenuation coefficient. Using such models, one can get an expression for cloud-free images, and researchers can estimate the actual ground reflectance [4]. However, this task demands considerable expertise and customization specific to the remote sensor and landscape conditions.

Cloud detection

Cloud detection in remote sensing imagery is vital for applications like weather forecasting and environmental analysis. Different methods, such as thresholding, spectral analysis, and machine learning, are used for this purpose. Thresholding sets a specific value for a parameter, like reflectance, to classify pixels as clouds if their values exceed the threshold. Spectral analysis uses unique spectral signatures of clouds, stemming from their scattering and absorption properties. Machine learning, especially with Convolutional Neural Networks and UNet architectures, has proven effective due to their ability to learn complex patterns. [5]–[8]

Cloud detection methods can be categorized based on their application:

1. Snow/Cloud Detection: Differentiating between snow-covered areas and clouds is challenging due to similar spectral signatures.
2. Cloud/No Cloud Detection: This binary classification aims to identify whether a pixel represents a cloud or not. It serves as a fundamental step in cloud detection systems.

3. Thin/Thick Cloud Detection: Differentiating between thin and thick clouds is essential for assessing cloud properties and their impact on the Earth's radiation budget. Thin clouds are less opaque and may require more sophisticated techniques for accurate detection.

However, creating a universally applicable model is challenging due to sensor characteristics and varying conditions.

Cloud removal

Removing clouds from HSIs often consists of removing the contamination from the clouds and recovering areas lost due to the cloud removal. Image stacking and pixel replacement methods can be used if multiple images of the same place, close in time, and with similar lightning conditions are available. If no such images are available, physical model and spectral unmixing approaches can be utilized to recover the true ground reflectance. In recent years, deep learning techniques have shown promising results in this field, but it is still an area of active research, with new advancements and improved methods frequently emerging.

2.2 The HYPSO-1 satellite

The HYPSO-1 satellite is a CubeSat that orbits 500km above the Earth's surface. It captures hyperspectral images of 40km x 40km areas and transfers them to nearby ground stations. The HSIs have a spectral resolution of 5nm with wavelengths going from 387nm to 801nm and a spatial resolution of 100m per pixel. The satellite's mission is to provide high-resolution remote sensing data for analyzing sporadic ocean color events, particularly algal blooms. If the algal is malignant, it may cause considerable damage to marine environments, ecosystems, and fish. An algal bloom has wavelengths from 400nm to 700nm, depending on the type, and therefore, using HYPSO-1 HSIs might help discover malignant types. [9]

The HYPSO-1 imaging mode produces hyperspectral cubes that are 153 MB in size. Even with a high data rate S-band downlink of more than 1 Mbps, downloading a whole cube would take more than 20 minutes [10]. The NTNU ground station sees the satellite for an average of ten passes daily, with a total theoretical duration of 100 minutes. If the operation is streamlined and error-free, approximately 60 minutes could be available for downlinking hyperspectral data. This corresponds to about 450 MB of data daily, equivalent to three uncompressed hyperspectral image cubes. In HYPSO-1, lossless CCSDS123 compression is used, which reduces the data to be downlinked to 55%, depending on the contents of the image cube. However, it may still require two or three ground station passes to complete the downlink of a single cube.

2.3 Tensors

Tensors are mathematical objects used in various fields to model complex systems with multiple degrees of freedom. For example, in machine learning and deep learning, tensors represent and manipulate large amounts of data like images, audio, and text. In addition, it is used to represent the weights and biases of neural networks, as discussed in sections 2.6 and 2.8.

The order of a tensor is determined by the number of vector spaces it is a product of. A first-order tensor, **a**, is often referred to as an array or vector, while a second-order tensor, **A**, is referred to as a matrix. The dimension describes how many elements are in a particular axis. A third-order tensor, $\mathcal{A}_{ijk}$, is illustrated in Fig. 2.4, showing the three ways to slice it into arrays. [11]

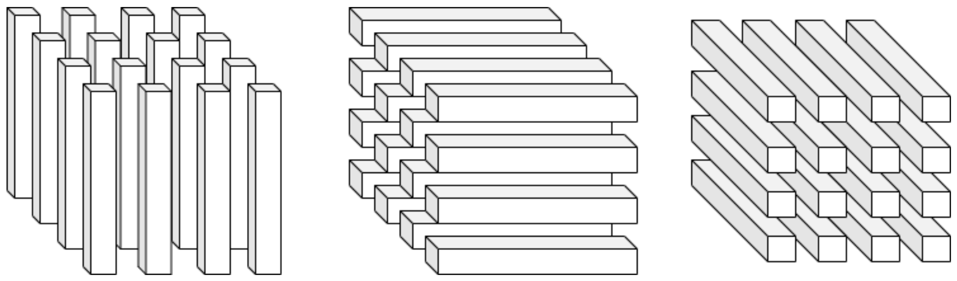


Fig. 2.4: A third-order tensor, sliced into columns (left), rows (middle), and tubes (right).

The rank of a tensor is given by the minimum number of rank one tensors needed to sum up the tensor. A decomposition of a third-order rank-one tensor is illustrated in Fig. 2.5, while a decomposition of a third-order rank N tensor is illustrated in Fig. 2.6.

Finding the rank of a tensor is an NP-hard problem, which means no known algorithm can solve it efficiently for all tensors. However, several methods can give an estimate of the rank, such as tensor decomposition methods like Principal Component Analysis (PCA).

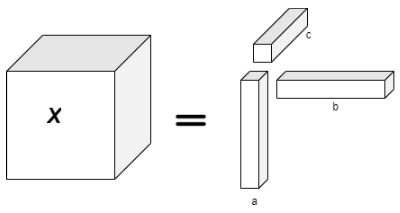


Fig. 2.5: A rank-one tensor, $\mathcal{X}$ can be represented by the outer product of three first-order tensors. This is called decomposition.

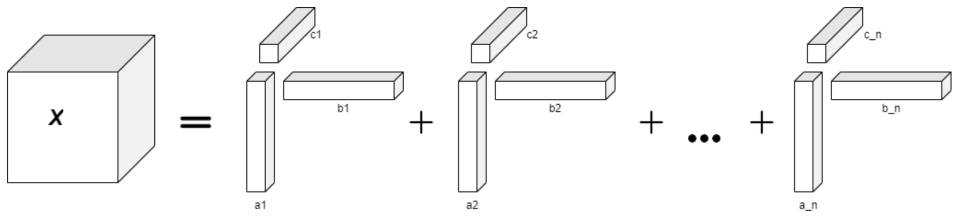


Fig. 2.6: A rank-N tensor, $\mathcal{X}$, decomposed into a linear combination of N rank-one decomposition.

2.4 Distance measurements and image reconstruction quality measurements

A variety of distance functions exist, and they can be used for a broad variety of purposes, from optimization to classification or as a measure of closeness, for example, reconstruction quality.

The image reconstruction quality is typically measured using metrics that provide information about the similarity or distance between an original image and a reconstructed/predicted image. These metrics can be divided into two main categories: subjective and objective [12]. Subjective metrics involve human observers who assess the quality of the reconstructed image, either visually or through the use of specialized image quality assessment tools. Subjective metrics help evaluate the perceived quality of the reconstructed image, but they can be time-consuming and subject to observer variability. On the other hand, objective metrics use mathematical algorithms to measure the reconstructed image's quality automatically. These metrics can include measures of similarity, such as the Peak Signal-to-Noise Ratio (PSNR) or the Structural Similarity index (SSIM), or measures of error, such as the mean squared error (MSE) or the mean absolute error (MAE). In addition, the Spectral Angle Mapper (SAM) and the Spectral Information Divergence (SID) are commonly used to measure the quality of the spectral domain of the reconstructed HSIs. Objective metrics provide a standardized way of evaluating the quality of reconstructed images, but they may not always accurately reflect the perceived quality of the image. Therefore, it is essential to use both in combination.

2.4.1 MSE and MAE

The Mean Squared Error of a predicted image compared to an original image is calculated as

$$MSE = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} (y[m,n] - \hat{y}[m,n])^2, \tag{2.1}$$

where $\hat{y}$ is the predicted image and y is the reference image. [13]

The Mean Absolute Error is calculated by taking the square root of the total absolute error [14], as given by

$$MAE = \frac{1}{NM} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} |y[m,n] - \hat{y}[m,n]|. \tag{2.2}$$

2.4.2 PSNR

Given a reference image and a predicted image of the same size, and with maximum pixel intensity, MAX_I, the $PSNR$ can be calculated as

$$PSNR = 10 \log_{10}(\frac{MAX_I^2}{MSE}).[13] \tag{2.3}$$

2.4.3 SSIM

SSIM is designed by modeling any image distortion as a combination of three factors that are loss of correlation, luminance distortion, and contrast distortion, and is given by

$$SSIM(f,g) = l(f,g)c(f,g)s(f,g) \tag{2.4}$$

where

$$\begin{cases} l(f,g) = \frac{2\mu_f\mu_g + C_1}{\mu_f^2+\mu_g^2+C_1} & \text{is the luminance distortion,} \\ c(f,g) = \frac{2\sigma_f\sigma_g + C_2}{\sigma_f^2+\sigma_g^2+C_2} & \text{is the contrast distortion,} \\ s(f,g) = \frac{\sigma_{fg} + C_3}{\sigma_f\sigma_g+C_3} & \text{is the loss of correlation.} \end{cases} \tag{2.5}$$

The positive constants C_1, C_2, and C_3 are used to avoid a null denominator, while μ and σ represent the mean and variance of the luminance of the two images f and g. The calculated value is in the continuous range of 0 to 1, where a value of 1 indicates that the images are identical, while a value of 0 indicates that the images are entirely different. [13]

2.4.4 SAM

The Spectral Angle Mapper (SAM) algorithm is a common method for evaluating the quality of a 3D reconstruction of hyperspectral image (HSI) data. The basic idea behind SAM is to compute the angle between the spectral vectors of the corresponding pixels in the reconstructed and the original HSI and then use this information to measure the reconstruction's spectral accuracy. This angle measures the similarity between the spectral information at those points, with smaller angles indicating higher similarity. The angle, α, between two spectral vectors, **X** and **Y** can be calculated as

$$\alpha = \cos^{-1} \frac{\mathbf{X} \cdot \mathbf{Y}}{\sqrt{(\mathbf{X} \cdot \mathbf{X})(\mathbf{Y} \cdot \mathbf{Y})}}. \tag{2.6}$$

The SAM is then calculated as

$$SAM = \cos\alpha = \frac{\mathbf{X} \cdot \mathbf{Y}}{\sqrt{(\mathbf{X} \cdot \mathbf{X})(\mathbf{Y} \cdot \mathbf{Y})}}, \tag{2.7}$$

and yields a number between 0 and 1, where 1 indicates perfect similarity. Finally, the overall spectral accuracy of the reconstruction is found by taking the average of the SAM of all corresponding points. [15]

Some potential limitations of SAM include sensitivity to noise since it is based on the dot product of spectral vectors. If the spectral vectors are corrupted by noise or other errors, the calculated angles may not accurately reflect the true similarity between the reconstructed 3D model and the reference HSI data.

2.4.5 SID

Spectral Information Divergence (SID) is a metric that originates from information theory, specifically the concept of divergence. It characterizes the statistical properties of a spectrum. Unlike SAM, which focuses on extracting geometric features between two spectra, SID treats each pixel spectrum as a random variable and quantifies the differences in their probabilistic behaviors. [16]

Given two normalized spectral vectors, $\boldsymbol{x}$ and $\boldsymbol{y}$, in the range $[0, 1]$, the SID can be calculated as

$$\text{SID}(\boldsymbol{x}, \boldsymbol{y}) = D(\boldsymbol{x}||\boldsymbol{y}) + D(\boldsymbol{y}||\boldsymbol{x}), \tag{2.8}$$

where $D(\boldsymbol{x}||\boldsymbol{y})$ is called the relative entropy of $\boldsymbol{y}$ with respect to $\boldsymbol{x}$, and is calculated as $D(\boldsymbol{x}||\boldsymbol{y}) = \sum_{i=0}^{i=N-1} x_i \log(x_i/y_i)$. The other way around for $D(\boldsymbol{y}||\boldsymbol{x})$.

2.5 Machine Learning

In the field of data science, machine learning (ML) involves programming a computer to learn from data. This can be achieved through various methods, such as optimizing statistical models or using deep neural networks. Regardless of the approach, the programmer must select a model that is suitable for the task and use data to facilitate learning. The learning process, referred to as training, requires a significant amount of data for effective models. The data may be labeled, indicating that the model is aware of how to organize the information or the goal during the learning phase. Alternatively, it may be unlabeled or unsupervised, meaning that the model must extract useful information by identifying patterns within the data without knowing how to categorize it. The data set is commonly divided into training, validation, and test sets. The training set is used in the model's learning process, while the validation set is utilized during training to monitor and control the learning process, preventing overfitting. Once the model is trained, the test set assesses its accuracy and generalization, reflecting its ability to handle new, unseen data. This test data set maintains the same structure as the training data set, containing only samples not part of the training process. [17]

2.5.1 Clustering

Clustering is an unsupervised machine-learning technique and a versatile tool for discovering structures within data. The primary goal of clustering is to group similar data points based on specific characteristics, allowing for the identification of patterns, relationships, and underlying structures that might not be immediately apparent. The process of clustering involves partitioning a data set into distinct groups, or clusters, where each cluster is composed of data points that share similarities. The fundamental challenge lies in determining how to measure similarity and define the boundaries that separate these clusters. Measures of similarity can be, for example, one of the distance measures defined in section 2.4. [18]

2.6 Artificial neural networks

Artificial neural networks (ANN), or just neural networks (NN), is a machine-learning approach inspired by how the neurons in the human brain work. NNs have shown remarkable success in a wide range of areas, such as computer vision, natural language processing, and bio-informatics [19]. It consists of nodes connected in a graph structure, with non-linear mapping as vertices of the graph/the connection between the nodes. One advantage is that, unlike other statistical techniques, the MLP makes no prior assumptions concerning the data distribution [20]. The most straightforward NN structure, or architecture, is the multi-layer perceptron (MLP), which is illustrated in Fig. 2.7.

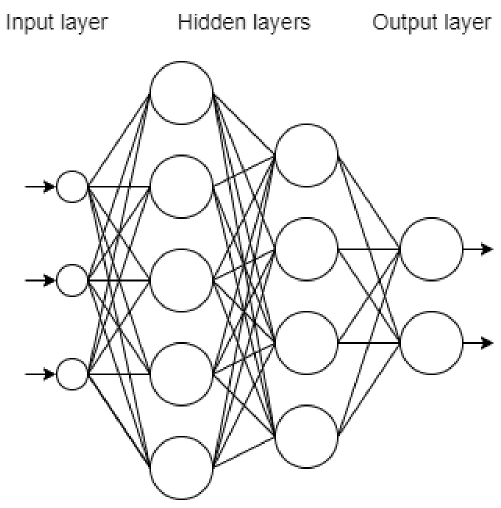


Fig. 2.7: An example of an MLP with an input, an output layer, and two hidden layers.

An MLP is composed of nodes that are arranged in layers. The input layer is where the data enters the network, while the output layer provides the network's prediction or output. The hidden layers are situated between the input and output layers. If a neural network has several hidden layers, it is referred to as a deep neural network (DNN). Each layer can be represented by a tensor, which is computed by multiplying the previous layer's outputs by a weight matrix plus a bias vector. Each element of the resulting vector is transformed by a non-linear function called an activation function. This activation function can be a sigmoid, a softplus, a softmax, or a rectified linear unit (ReLU). Tensors and tensor operations are important components of deep learning, allowing for the efficient computation of large-scale neural networks. Popular machine learning and deep learning libraries like PyTorch, TensorFlow, and Keras utilize tensors in their computation, providing high-level abstractions for working with them and making it easier to build and train complex neural networks. Tensors have revolutionized many fields in machine learning and deep learning, such as computer vision, natural language processing, and speech recognition, enabling the development of highly accurate and scalable machine-learning models that can be trained on massive data sets. [19]

2.6.1 Types of layers

ANNs have various architectures that are selected by the programmer. In addition to different types of activation functions and structures, they are made up of different layers. These layers are essential components of the network, and the NN is often named after the main layer type. The following sections describe some important layer types.

Convolutional layer

Convolutional layers utilize the mathematical operation convolution by combining two functions, namely the kernel/filter and the input signal, to generate a third function known as the output signal.

When performing a 1D convolution, a kernel is applied to the input data by sliding it along the length of the data and computing the dot product at each position. A 1-dimensional (1D) convolution is given as

$$h[n] = (f * g)[n] = \sum_{k=-\infty}^{\infty} f[k]g[n-k] \tag{2.9}$$

where $f[n]$ represents a discrete signal, and $g[n]$ acts as the kernel. The output of the convolution between $f[n]$ and $g[n]$ is a new signal, $h[n]$ [21].

When processing images in NN, 2D convolutions are commonly utilized. The kernel is a small, fixed-size matrix that is used to extract features from the input data, while the input is one or more larger matrices. In a 2D convolution, the kernel is moved vertically and horizontally over the entire input image, and the dot product is computed at each position, similar to a 1D convolution. The kernels are usually spatially-localized patterns that are learned from the training data. They are used to detect edges, corners, textures, and other visual patterns in the input data. An example of a 2D convolution using a 3×3 kernel is given in Fig. 2.8.

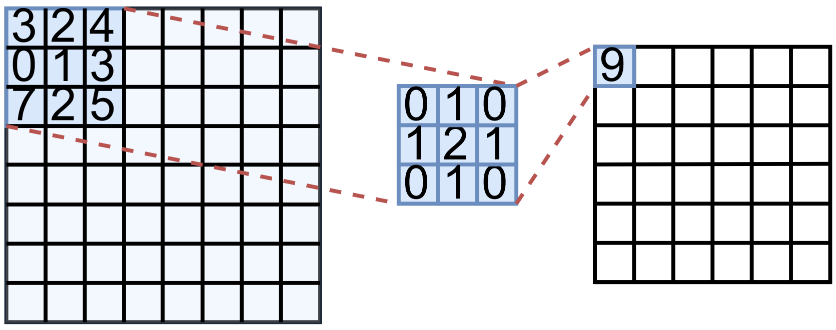


Fig. 2.8: A 2D convolution between a 3×3 kernel and an image.

A 3D convolutional layer in a neural network is similar in structure to a 2D convolutional layer but with a few notable differences. The primary distinction is that the 3D convolutional layer employs 3D kernels with three dimensions (length, width, and depth) instead of 2D kernels with only two dimensions (length and width). This enables the 3D convolutional layer to extract features from the third dimension of the input data, which corresponds to the spectral bands for an HSI or time in a video. Fig. 2.9 visually demonstrates the dimensional differences between a 2D and a 3D convolution.

A convolutional layer can have several filters that extract diverse information from the image. The result of the convolutional layer is a feature map, which is a collection of filtered versions of the input data that encode the local features detected by the filters. Besides the convolutional operation, a convolutional layer might introduce a bias and apply a non-linear transformation, such as the activation functions mentioned earlier [19].

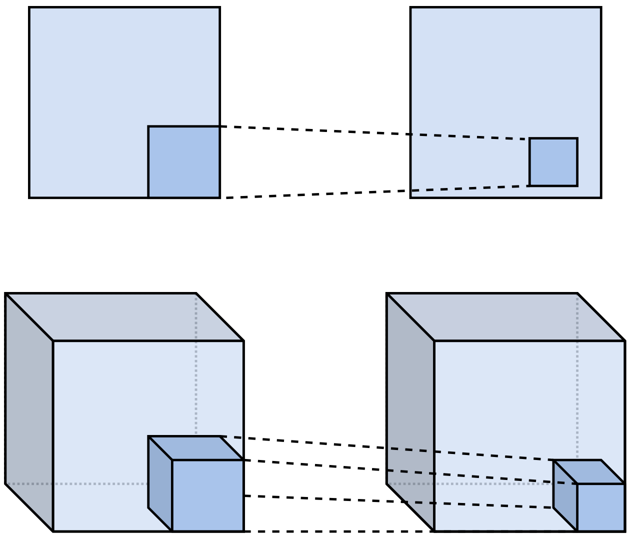


Fig. 2.9: A 2D convolution in comparison to a 3D convolution.

Dilated convolutional layers

Dilated Convolutions refer to convolution that involves widening the kernel by creating gaps between its elements. The degree of expansion is regulated by a parameter known as the dilation rate (d), which determines the extent of the widening. Typically, these gaps are achieved by leaving spaces between the kernel elements, as shown in Fig. 2.10.

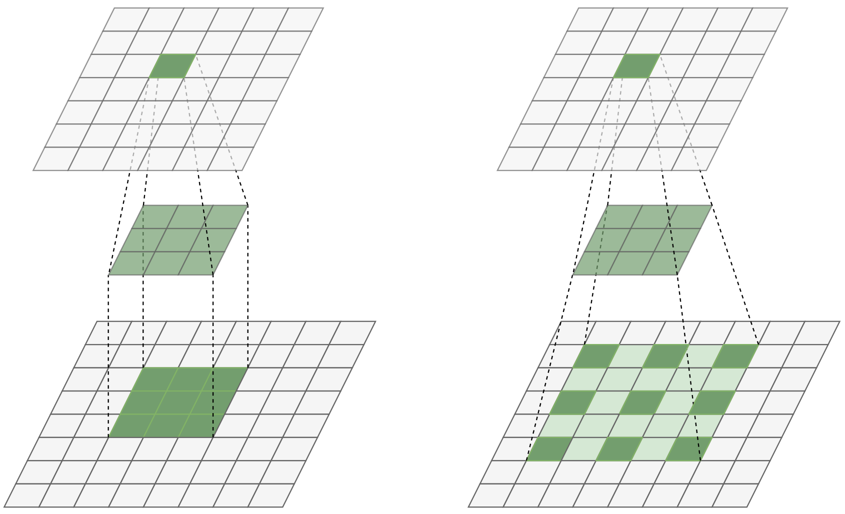


Fig. 2.10: A normal convolution with $d = 1$ and kernel size 3×3 to the left, and a dilated convolution with $d = 2$ and kernel size 3×3 to the right.

Max-pooling layers

Max-pooling is a method of reducing the size of a set of numbers by selecting only the highest value from a pool or patch. For instance, if the pool is of size 2×2, the output from the layer will be halved in both height and width. This is illustrated in Fig. 2.11.

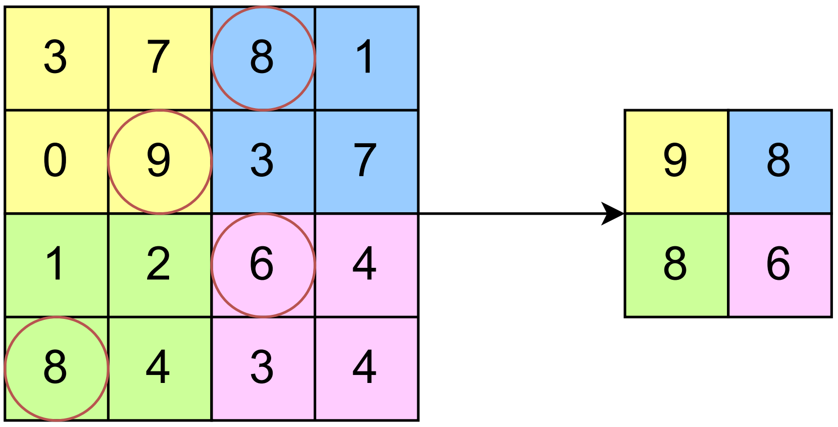


Fig. 2.11: Max-pooling with a pool size of 2×2. The different pools are color-coded, and the highest value within each pool is marked with a red ring.

Pools can come in varying sizes, and there are also different types of pooling available such as average, minimum, and median pooling [19].

2.6.2 Convolutional Neural Network

A convolutional neural network (CNN) is a type of DNN that is designed to take advantage of the spatial structure of data and uses convolutional layers and pooling layers to extract high-level features from the input. CNNs are widely used for image and video analysis tasks, such as image classification, object detection, and image segmentation, and they have achieved state-of-the-art performance. A CNN often consists of many convolutional layers in combination with pooling layers and fully or partially connected layers [19]. Fig. 2.12 illustrates an example of a CNN with two convolutional layers, two max-pooling layers, and two fully connected layers. The input to the CNN is an RGB photo with spatial dimension $3 \times 128 \times 128$, and the output is a 1×64 array.

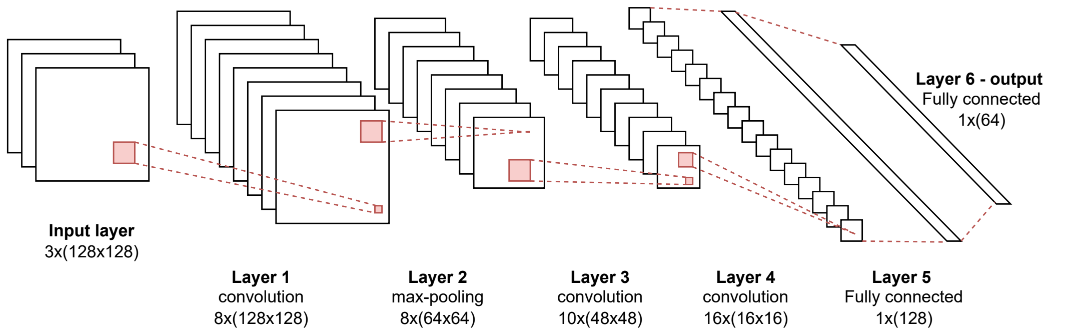


Fig. 2.12: A CNN with two convolutional layers and two max-pooling layers, every other, and with two fully connected layers in the end.

2.6.3 U-net architecture neural network

U-Net is a popular type of CNN utilized for image segmentation tasks, such as medical imaging, remote sensing, and industrial inspection. Its U-shaped architecture is a significant feature, comprising of a down-sampling path for encoding the input image into lower-resolution feature maps, and an up-sampling path for decoding the feature maps into a high-resolution segmentation map. The skip connections linking the down-sampling and up-sampling paths enable the network to merge information from different scales and resolutions, producing more accurate and detailed segmentation maps. U-Net can be trained end-to-end, thereby allowing it to learn from numerous examples and improve its performance on the segmentation task. Besides its U-shaped architecture, U-Net implements other design choices, such as batch normalization and ReLU activation functions, to enhance network performance and stability. Additionally, U-Net includes various loss functions to train the network for different types of segmentation tasks. [22]

2.6.4 Advantages and disadvantages with using NN

The field of machine learning has brought about significant changes by providing solutions to a wide range of problems. A well-designed model can extract useful information and make predictions from data, even without prior knowledge of the topic at hand. This often yields better results and faster computation times than traditional algorithms. Machine learning can even identify patterns that may not be discernible to human eyes. However, there are some potential drawbacks to using neural networks (NNs). For instance, they need a lot of labeled training data to learn effectively, which can be challenging and costly to obtain in certain circumstances. NNs can also require significant computational resources to train

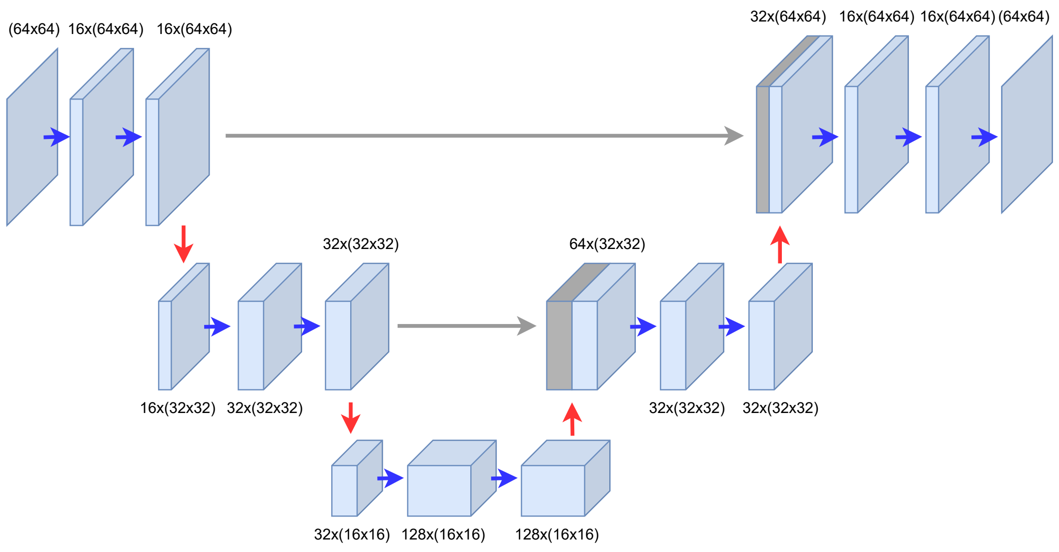


Fig. 2.13: A simple u-net of depth 3, taking a 64×64 image as input. The red arrows symbolize the up-and-down sampling, while the blue ones symbolize the convolutions.

and run, which could limit their application in some areas. Additionally, they can be prone to overfitting, meaning that the model learns the details of the training data too well, resulting in poor generalization to new data. NNs can be challenging to interpret, as their internal workings are often intricate and not easily understandable by humans. Moreover, NNs are sensitive to the quality and format of input data, and this may lead to poor results if the data is noisy or has other issues. [17]

2.7 Training of a neural network

When training a neural network, the goal is to optimize its performance for a specific task. To achieve this, the network is presented with many examples of desired input and output, and an optimization algorithm is used to adjust its parameters. During training, a loss function is used to measure the difference between predicted and desired outputs. The algorithm then uses this information to adjust the parameters and reduce the loss. This process is called back-propagation and is typically done using a variant of gradient descent [19]. The network's performance improves as training continues, and the parameters are adjusted to produce more accurate outputs. The process stops when the network reaches a satisfactory level of performance or when the optimization algorithm reaches a predefined stopping point [23].

2.7.1 Loss functions

In order to learn the parameters of the model, a it is necessary to establish a loss function that can identify and penalize errors [19]. Different types of loss functions are utilized depending on the optimization task (e.g. regression, classification, etc.). If $y_i \in \mathbb{R}$ represents the desired outcome and $X_i \in \mathbb{R}^p$ signifies the input for the ith sample, where i belongs to the set of values from 1 to n, the learned model parameters or weights are represented by $\beta \in \mathbb{R}^p$. This means that the predicted outcome of the NN is $X_i^T\beta$. Then, the targeted output can be modeled as a combination of the predicted outcome and an error $\epsilon_i \sim N(0, \sigma^2)$, as given by

$$y_i = X_i^T\beta + \epsilon_i. \tag{2.10}$$

A loss function, denoted as $L(y_i, X_i, \beta)$, is responsible for determining the distance between the predicted output and the desired output using the parameters outlined in (2.10). The distance functions described in section 2.4 are applicable as distance functions, with the MSE as the most common. [19]

2.7.2 Optimization algorithms

To find the best values for a model's parameters, one must minimize the average loss using the loss function across all parameters in β. This involves calculating an error gradient throughout the neural network, which is then used to minimize the total error [24]. There are various algorithms for optimizing model parameters, some of which are discussed in the following sections.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) algorithm is a simple and popular optimization technique based on gradient descent. SGD analyzes a single example from the training data to calculate the gradient of the loss function. It then adjusts the parameters of the network based on this gradient. While SGD can be effective for large-scale training, it may also be noisy and slow to converge. [24]

Mini-batch Gradient Descent

Mini-batch Gradient Descent is a modified version of gradient descent that uses a small batch of training data examples to calculate the gradient of the loss function. Compared to SGD, Mini-batch gradient descent is more stable and efficient. However, it necessitates additional computational resources and may still have slow convergence. [24]

Momentum-based Gradient Descent

Momentum-based Gradient Descent is a version of gradient descent that utilizes a momentum term to accelerate the optimization process. This term is derived from previous updates to the network's parameters and can prevent the optimization algorithm from being trapped in local minima or saddle points. [24]

Adaptive Moment Estimation

The Adaptive Moment Estimation (ADAM) algorithm is a frequently used optimization algorithm for training neural networks. It is a type of stochastic gradient descent that employs an adaptive learning rate to enhance the optimization process's convergence. ADAM combines mini-batch gradient descent and momentum-based gradient descent, using a mini-batch of examples from the training data to compute the loss function's gradient and a momentum term to speed up the optimization process. Additionally, ADAM incorporates a regularization term to avoid over-fitting and improve the trained network's generalization performance. [24], [25]

2.8 PyTorch

PyTorch is an open-source machine learning library widely used in industry and academia to develop and train deep neural networks. It was first released in 2016 by Facebook's AI research group and has since gained popularity due to its user-friendly interface and flexibility. It is build on the Python programming language and the Torch library. Its flexible architecture allows users to build and train machine learning models of varying sizes and complexities while providing a range of tools and libraries for working with data, training, and evaluating models. One of PyTorch's key features is its ability to work with tensors. This allows for efficient processing and manipulation of large data sets, enabling complex mathematical operations even on large data sets. [26]

GPUs (Graphics Processing Units) are specialized hardware components that excel at performing parallel computations. They are commonly used in tasks that require high computational power, such as graphics rendering, scientific simulations, and deep learning. PyTorch provides functionalities for creating and training deep learning models on GPUs, leveraging the parallel computing capabilities of NVIDIA GPUs to accelerate computations. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows developers to utilize the power of GPUs for general-purpose computing tasks. CUDA provides a programming framework and tools for GPU-accelerated computing, enabling developers to write code that can run efficiently on NVIDIA GPUs. [27]

2.8.1 Building a model with PyTorch

Building machine learning models in PyTorch typically involves defining a class that inherits from the nn.Module class. This class defines the structure of the model, including its layers, activation functions, and any other operations. The forward method of the class specifies how the input data is processed through the layers of the model. During training, the backward method is used to compute gradients and update the model parameters according to an optimization algorithm. When building large neural networks with PyTorch, it is common to use a modular approach and separate the network into smaller building blocks. This approach provides a better understanding of the network architecture, makes implementation faster, and reduces the likelihood of errors.

To utilize the `torch` library, one must use the data structure `torch.Tensor`. When working on the GPU, it is necessary to store all variables and models in the GPU memory. PyTorch provides a simple function to transfer data from a computer's CPU to the GPU. By using the function `x.to(device="cuda")`, one can easily move the `torch.Tensor`, here named `x`, to the GPU. [27]

2.9 Introduction to Compressive sensing

Compressive sensing (CS), also called compressive sampling or compressed sensing, is an emerging field within signal processing, computer vision, and information theory. CS exploits the property that a signal is often sparse in some transformed domain to recover it from a small number of linear, random measurements. Unlike conventional sampling methods, which are limited by the Nyquist–Shannon sampling theorem [28], compressive sensing allows for sub-Nyquist sampling, which reduces the number of samples needed to reconstruct an image. This, in turn, leads to significant reductions in memory usage and transmission and imaging time. Robust signal recovery is possible from a number of measurements proportional to the signal's sparsity level. This means that even when a signal is only partially sampled, it can still be reconstructed accurately using compressive sensing techniques. As such, compressive sensing has become a valuable tool in many areas, including remote sensing.

A compressive sensed signal $\boldsymbol{y} \in \mathbb{R}^{M\times 1}$ can be expressed by

$$\mathbf{y} = \Psi\mathbf{f} \tag{2.11}$$

where $\Psi \in \mathbb{R}^{M\times N}$ is a sensing matrix and $\boldsymbol{f} \in \mathbb{R}^{N\times 1}$ is the original signal. $\boldsymbol{f}$ can be represented as

$$\mathbf{f} = \Phi\mathbf{x} \tag{2.12}$$

where $\boldsymbol{x} \in \mathbb{R}^{N\times 1}$ is sparse vector and $\Phi \in \mathbb{R}^{N\times N}$ is a basis. **A** includes both Φ and Ψ, for easier notation, and is called a dictionary. The dictionary Combining (2.11) and (2.12) gives

$$\mathbf{y} = \Psi\Phi\mathbf{x} = \mathbf{A}\mathbf{x}. \tag{2.13}$$

It is important to note that the notation may vary slightly in the literature, which means that the variables used for different purposes may be different.

Certain conditions must be met to effectively reconstruct a signal using fewer samples than what is traditionally required. The signal must be sparse in some domain, meaning that it only has a few non-zero elements. If the signal itself is not sparse, it may still be possible to represent it as sparse in a different domain by expressing it with a basis. In addition to sparsity, the sampling of the signal must also meet certain requirements in terms of incoherence and the number of samples. This will be discussed in the following sections.

2.9.1 Degree of sparsity and sparsity approximation.

The sparsity level of a signal is determined by the number of non-zero elements in the sparsest domain, referred to as κ. Meanwhile, the degree of sparsity indicates the percentage or fraction of data represented by zero values. A high degree of sparsity implies that most of the data is represented by zero values, while a low degree of sparsity means that only a small fraction of the data is represented by zero values. The degree of sparsity impacts the effectiveness of compressive sensing algorithms, as it affects the amount of data that needs to be captured and reconstructed.

In practical situations, signals may not be completely sparse but rather have many values that are almost zero. These values can be considered noise and can usually be disregarded during the signal reconstruction process. This is called sparsity approximation and allows for a more effective and precise reconstruction by focusing only on retrieving the non-zero elements of the signal. In other words, sparsity approximation compresses a signal by keeping only the most significant coefficients in its sparsest representation.

2.9.2 Representing a signal by using a basis

A signal can be represented by a basis, $\mathbf{\Phi}$, such as the Fourier basis to the cosine transform. A set of atoms $\{\phi_\gamma\}_{\gamma\in\Gamma}$ is basis for a space $\mathbb{H}$ if the signals together span out $\mathbb{H}$, as given by

$$\text{span}\{\phi_\gamma\}_{\gamma\in\Gamma} = \mathbb{H}. \tag{2.14}$$

If the atoms are orthogonal, as defined by

$$\langle \phi_\gamma, \phi_{\gamma'} \rangle = \begin{cases} 1, & \text{if } \gamma = \gamma' \\ 0, & \text{if } \gamma \neq \gamma', \end{cases} \tag{2.15}$$

the basis is orthogonal. The basis is orthonormal if the signals also are unit vectors (the magnitude is equal to one) [29]. If $\{\phi_\gamma\}_{\gamma\in\Gamma}$ is a orthonormal basis for $\mathbb{H}$, any signal $f \in \mathbb{H}$ can be written as

$$f(t) = \sum_{\gamma\in\Gamma} \langle f(t), \phi_\gamma \rangle \phi_\gamma = \sum_{\gamma\in\Gamma} x_\gamma \phi_\gamma. \tag{2.16}$$

where the signal, $f(t)$, is a linear combination of the atoms ϕ_γ, and the sparse coefficients, $x_\gamma \in \boldsymbol{x}$. $\boldsymbol{x}$ is the sparse representation of the signal and can be calculated by taking the inner product of $f(t)$ and ϕ_γ. While using non-orthonormal bases is possible, it can introduce errors such as energy distortion. Orthonormal bases frequently used in compressive sensing include the Fourier basis, which breaks down a signal into its frequency components, the discrete cosine transform basis, which describes the signal using its cosine functions, and the wavelet basis, which is particularly useful for capturing localized features in a signal. [30]

2.9.3 Incoherent sampling and the sensing matrix

In terms of sensing or sampling, the process is done by applying a sensing matrix, Ψ to the signal, as denoted in (2.11). To effectively utilize compressive sensing, the sensing matrix and sparsity basis must be "incoherent," meaning they should be as uncorrelated as possible. This prevents the measurement process from introducing significant correlations into the signal, which can degrade the reconstruction [30].

Incoherent sampling can be achieved using a random or pseudo-random sensing matrix or a deterministic approach designed to have low coherence with the sparsifying basis. The choice of the sensing matrix and the sampling method will depend on the signal's characteristics and the application's requirements. When acquiring data, the sensing matrix can be applied to an already acquired signal or by taking the selected samples directly at the sensor level. This approach can be more efficient in certain cases, such as in a hyperspectral imager on a satellite.

Restricted Isometry Property

The Restricted Isometry Property (RIP) is an essential principle in compressive sensing that measures a sensing matrix's ability to accurately represent a signal. To ensure successful reconstruction, the sampling matrix must satisfy a condition described by RIP. A dictionary **A** is said to satisfy RIP of order κ if

$$(1-\delta_k)||\mathbf{x}||_2^2 \leq ||\mathbf{Ax}||_2^2 \leq (1+\delta_k)||\mathbf{x}||_2^2 \tag{2.17}$$

where $\delta_k \in (0,1)$ is a constant determined by the sparsity level, κ, and the allowed error. x is a sparse signal. Typically, δ_k is close to zero [30]. The subscript indicates the type of norm used, here, the l-2 norm (euclidean norm).

RIP can be interpreted as the conservation of energy, nearly orthogonal columns in **A**, and the uniqueness of transformed vectors. When RIP holds, the dictionary **A** preserves the Euclidean distances, or the MSE, between two sparse vectors x and y, as given in

$$(1-\delta_k)||\mathbf{x}\text{-}\mathbf{y}||_2^2 \leq ||\mathbf{Ax}\text{- }\mathbf{Ay}||_2^2 \leq (1+\delta_k)||\mathbf{x}\text{-}\mathbf{y}||_2^2, \tag{2.18}$$

and the data's information is nearly preserved. [30]

Compression rate

The compression rate, CR, is the ratio between the size of a compressed signal and its original, uncompressed version. For instance, if a signal, $\mathbf{x} \in \mathbb{R}^{N\times 1}$, is compressed and results in a smaller signal, $\mathbf{y} \in \mathbb{R}^{M\times 1}$, the compression rate can be determined by

$$CR = \frac{N}{M}, \tag{2.19}$$

where $N > M$.

Selection of compression rate

The compression rate should be selected accordingly to the allowed error, the coherence of the matrices, and the sparsity degree. There exist multiple bounds stated as a function of the sparsity level, and no exact answer can be provided. However, different limits in which the RIP holds with high probability are given.

Suppose **A** is obtained by randomly selecting M rows from an $N \times N$ orthonormal matrix, with values drawn independently from the Gaussian probability distribution with zero mean and variance $\frac{1}{M}$. Then RIP theorem holds with overwhelming probability when

$$\kappa \leq \frac{CM}{\log(\frac{N}{M})}. \tag{2.20}$$

where C is a constant [30]. By solving (2.20) for M, the number of samples can be estimated. Even when not working with dictionaries drawn from a Gaussian distribution, this gives an approximation of a lower bound.

2.9.4 Signal reconstruction

There are several approaches to reconstructing a compressive sensed signal. A traditional approach is to solve an optimization problem that recovers the original signal from the compressed measurements. This optimization problem typically involves minimizing the difference between the reconstructed signal and the original signal or ground truth, subject to certain constraints such as sparsity or rank. Recent research also explores the use of machine learning techniques for compressive sensing signal reconstruction. Deep neural networks, in particular, have shown promising results in this area. The following section will describe these methods closer.

Traditional reconstruction methods

Given a sparse signal **x**, a dictionary, **A**, and a compressive sensed signal $y = \mathbf{A}x$, the optimization problem to be solved is

$$\min_{\hat{x} \in \mathbb{R}^N} ||\hat{x}||_0 \quad \text{subject to} \quad \mathbf{A}\hat{x} = y. \tag{2.21}$$

The problem of finding the best solution for l-0 recovery is difficult and falls under the category of NP-hard problems [30]. However, in many cases, solving the l-1 recovery problem can provide a close approximation to the solution. This problem aims to minimize the l-1 norm of the solution, $\hat{x}$, while using the same constraint. If the signal is just approximately sparse, the solution can be obtained by solving the optimization problem given by

$$\min_{\hat{x} \in \mathbb{R}^N} ||\hat{x}||_1 \quad \text{subject to} \quad ||y - \mathbf{A}\hat{x}||_2 \leq \epsilon, \tag{2.22}$$

which allows some errors to be present in the reconstruction.

There are several algorithms available to solve optimization problems in compressive sensing. These algorithms use different optimization techniques and can have different levels of efficiency and accuracy. The reconstructed signal $\hat{x}$ is an approximation of the original signal x. The reconstruction accuracy depends on the CS algorithm's performance and the compressed measurements' quality.

Deep learning for reconstruction

Deep learning has shown great potential when it comes to the reconstruction of compressive sensed signals. CNNs, in particular, are able to learn how to approximate the inverse mapping from compressed measurements to the original signal by training on a data set of pairs of compressed measurements and their corresponding original signals. With a well-designed NN architecture, it becomes possible to identify correlations in specific cases, if they exist. This holds true for the recovery of compressive sensing data as well. The challenge primarily lies in the NNs design, as there is no set recipe for creating it.

Once trained, these NNs can accurately and efficiently reconstruct signals from new compressive measurements. Compared to traditional optimization-based methods, deep learning reconstruction typically results in faster and higher-quality outcomes, provided suitable NN architectures and training data are used.

However, there is skepticism among researchers about the reliability and generalizability of these NNs due to their black-box nature i.e., the difficulty in understanding how they arrive at their decisions, as discussed in section 2.6.4.

3 Related Work

This section presents research that is relevant to the thesis. The work has been divided into the following sections: compressive sensing and reconstruction, cloud detection and segmentation, and cloud removal and recovery of missing areas. These sources provide insight into the topics, serve as inspiration, provide a foundation for decision-making, and are used as a baseline for evaluating the performance of the work done in this thesis.

3.1 Compressive sensing methods for HSIs and reconstruction methods

This section presents some of the research done in the field of compressive sensing and reconstruction, with a focus on hyperspectral images and using deep learning for reconstruction.

3.1.1 Compressive sampling of hyperspectral images

Various approaches have been developed for compressive sampling in hyperspectral images, involving compression in the spectral domain, spatial domain, or a hybrid combination. For example, in [31], H. Xiao et al. employed random sampling matrices to compress HSIs in the spatial domain during image acquisition. Another technique presented by Golbabaee et al. in [32] performs compressive sensing on the entire hyperspectral image cube, encompassing both the spectral and spatial domains. However, their method utilizes a least-squares approach on the mobile system to reduce spectral bands and requires a comprehensive library of spectral signatures.

In the domain of hyperspectral imaging systems, Y. Oiknine et al. introduced the Miniature compressive Ultra-Spectral Imaging system (MUSI) in [33], which employs a liquid crystal (LC) phase retarder to compress the spectral domain. By modulating the refractive index of the LC cell through applied voltage, the sensing matrix, Ψ, is

constructed by selecting rows from the spectral response map.
Compressive sensing is also used for dimensionality reduction in settings other than acquisition, such as image classification as in [34].

3.1.2 Classical reconstruction methods for compressive sensing

There are various algorithms available for addressing the optimization problem described in 2.22. One such algorithm is the Fast Iterative Shrinkage-Thresholding Algorithm (FISTA), an iterative and convex algorithm combining a soft-thresholding operator with the gradient method [35]. It is commonly used in compressive sensing and is known for its rapid convergence. Another algorithm is the convex Alternating Direction Method of Multipliers (ADMM), which breaks down the original optimization problem into smaller sub-problems and then solves them separately [36]. This approach can make it more efficient than other methods and is also useful for handling constraints in the optimization problem. There are also several greedy algorithms, such as the Generalized Orthogonal Matching Pursuit (gOMP) [37], the Backtracking Iterative Hard Thresholding (BIHT) that uses hard thresholding to obtain a sparse solution [38], and the Compressive Sampling Matching Pursuit (CoSaMP) algorithm, which is a variant of the Matching Pursuit algorithm that leverages a sparse projection operator to enforce sparsity during the optimization process. CoSaMP is particularly effective at handling noise in the measurements [39].

Justo and colleagues conducted a study comparing the accuracy and speed of various algorithms for reconstructing hyperspectral images, as described in their article [40]. The greedy gOMP algorithm was found to be the most effective, with a max PSNR of over 50dB and a reconstruction time of 0.5 to 10 minutes. The other algorithms had an average PSNR of about 45 dB, and their reconstruction times ranged from a few minutes to over 24 hours. It's worth noting that the study tested different compression ratios, and the standard deviation in the results achieved by gOMP was 10.76 dB.

3.1.3 Deep learning-based reconstruction

ADMM-CSNet from [41] is a fast and accurate deep-learning approach using compressive sensing and data-driven techniques to produce images from sparse measurements. It has been successfully applied to complex-valued MR imaging and natural image reconstruction, achieving superior performance while remaining computationally efficient. The key contribution is the introduction of ADMM-CSNets that enable optimal image recovery for various tasks and image types. measurements and faster computational speed.

Xu et al. propose in [42] have developed a technique for rebuilding hyperspectral imaging (HSI) from compressive measurements while identifying anomalies in the data. Their approach uses tensor-based processing with tensor robust principal component analysis (TRPCA) and a Mahalanobis distance-regularized term. The method performs better than state-of-the-art techniques for both reconstruction quality and anomaly detection accuracy.

The study, given in [43], conducted by the University of Negev, explores using u-nets to reconstruct hyperspectral images. The researchers developed a CNN called DeepCubeNet that employs a U-net with 3D-convolutional layers and a 1D-convolutional layer to up-sample compressive sensed data, perform max pooling, and down- and up-sample data. The ADAM optimizer and MSE loss function are utilized. The images used for training and testing were captured using the CS-MUSI method. The study demonstrates that DeepCubeNet produces high PSNR results and significantly reduces reconstruction time compared to traditional methods. Average reconstruction time for a 64×64×391 HS patch from a 64×64×32 compressed patch captured with CS-MUSI is 0.25 seconds on a GPU and 6 seconds on a CPU. Overall, the study highlights the potential of deep learning-based methods for reconstructing spectrally compressed HSI and emphasizes the advantages of the proposed DeepCubeNet architecture.

The unpublished paper [44] is based on the DeepCubeNet. It looked at how u-net with 3D convolutions can be used to reconstruct HSIs from the HYPSO-1 and the ICVL data sets. The basic neural network architecture is the same as in the DeepCubeNet, except for the compression of the spectral bands, which is suited to fit the spectral dimensions of the ICVL and HYSPO-1 data. The thesis explores how different preprocessing techniques affect the reconstruction quality for the ICVL images and looks at potential generalization across preprocessing schemes. When reconstructing HYSPO images, the model reaches PSNRs up to 50.84 dB, with a mean of 45.46 dB. The reconstructions have, however, errors in the reconstructions for some of the sampled points, which should always be correct. In addition, some predictions reach values higher than one, even though the input pixel values are normalized in the range [0, 1].

In [45] compares deep learning-based image reconstruction with traditional methods in compressive sensing. The findings show deep learning approaches perform better without compromising robustness. However, the study also reveals susceptibility to noise and perturbations.

3.2 Cloud detection and segmentation

Cloud detection in HSI is an active research area, and new techniques are continuously emerging. It is challenging due to the data's high spectral resolution and complex nature. State-of-the-art cloud detection involves combining traditional computer vision techniques and machine learning algorithms.

In [7] Papin et al. (2018) use infrared images from METEOSAT to detect clouds. They extract features, such as motion-based measurements, intensity images, and thermal parameters, from sequences of images.A statistical labeling process categorizes pixels as "low clouds" or "clear sky" using a Bayesian estimation framework with Markov random field (MRF) models. The approach minimizes a global energy function consisting of data-driven terms (thermal and motion-based) and a regularization term representing prior knowledge on the label field. The authors use a progressive minimization procedure that starts with reliable labels and involves local computation to minimize a global energy function consisting of thermal and motion-based terms and a regularization term representing prior knowledge on the label field.

Gomez-Chova et al. present a four-step cloud detection algorithm in [6]. The first step is feature extraction, where they extract physical features to increase the separability between clouds and surface covers. They consider cloud brightness and atmospheric absorptions to estimate the optical path. The next step is finding the region of interest, where a nonrestrictive threshold and a region-growing algorithm identify cloud-like pixels. The third step is image clustering and labeling, where they use the Expectation-Maximization algorithm to cluster the pixels within the region of interest and label them based on their geophysical classes. They compute a cloud probability index by summing the posteriors of the cloud clusters. The final step is in the spectral domain, applying a spectral unmixing algorithm to obtain a cloud abundance map for each pixel. The researchers combine the cloud abundance map and the cloud probability using pixel-by-pixel multiplication to improve the accuracy of cloud detection. The algorithm performs well in detecting thin cirrus clouds and clouds over ice/snow and accurately identifies cloud borders.

B. Grabowski et al. describe in [5] a way of detecting clouds in remote sensing images using a CNN based on the U-Net architecture. The U-Net was trained on a dataset containing images with and without clouds, taken at different time instances but close in time. The images have four spectral bands, hence not HSIs. The results show that the U-Net, in general, can detect clouds but that the segmentation quality depends on the training data and that including challenging scenes like snow helps improve the generalization abilities of the NN.

3.3 Cloud removal and recovery of missing areas

Cloud removal methods can typically be classified into three groups: single-image, multi-spectral-image, and multi-temporal-image.

Zheng et al. describe in [46] a way of removing clouds from single images using a combination of two different NNs. The first one is a U-Net which removes thin clouds. The second one is a Generative Artificial Network (GAN) which reconstructs the missing parts due to thick clouds. It consists of three neural networks: a U-Net, a generator (G), and a discriminator (D). They have the same general encoder-decoder network structure, with 11 layers in total. The first five layers form a contracting path, and the sixth to the tenth layers create an expanding path. In the contracting path, convolution blocks perform multichannel 2D convolutions with a step size of 1 or 2. The expanding path has a symmetric structure with transposed convolution blocks for G and D, while the U-Net has a doubled number of channels in each layer due to copy and concatenation operations

The authors of [47] propose a multi-spectral method that utilizes the concept of the dark channel prior method to estimate the appearance of clouds. The method avoids altering the original spectral information by subtracting solely in the intensity channel. The intensity is then enhanced using gamma correction to recover some information unintentionally lost during the previous step and restore obscure details distorted by clouds. Additionally, considering that clouds affect both the intensity and saturation channels, the authors address the reduction in saturation caused by clouds by applying a logarithmic image transformation. This transformation helps to increase the saturation and compensate for the saturation loss resulting from the presence of clouds.

In [48], the authors propose a comprehensive approach to reconstructing missing data. They implement a neural network (NN) that incorporates spatial-temporal-spectral (STS) data to retrieve lost information caused by various factors such as deadlines and thick clouds. The framework requires input information that is spatial, temporal, and spectral in nature. The NN is comprised of two parts - the first part uses regular convolutions while the second part employs dilated convolutions, and both parts have a set of skip-connections. The resultant trained model is capable of recovering all types of missing data based on the STS data.missing data based on the STS data.

System

4

This thesis's primary objective is to assess the feasibility of creating a pipeline that encompasses the reconstruction of compressive sensed images while simultaneously removing small clouds. While the ultimate aim is to create a user-friendly system that seamlessly integrates all components automatically, as illustrated in Fig. 4.1, it should be noted that this aspect falls beyond the scope of this thesis.

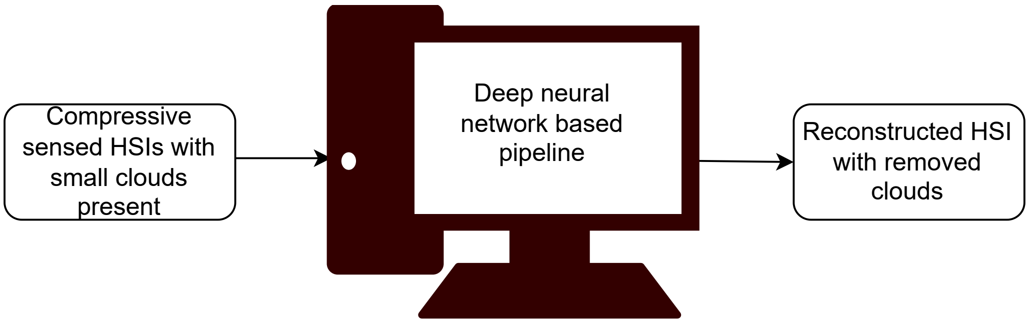

Compressive sensed HSIs with small clouds present
Deep neural network based pipeline
Reconstructed HSI with removed clouds

Fig. 4.1: By integrating cloud removal techniques and reconstructing compressive sensed images, a user can benefit from a seamless and user-friendly experience with a black box approach.

As outlined in the research section, there are various solutions available for each component of the pipeline. However, there is a lack of comprehensive solutions that take into account all aspects. Additionally, there is a shortage of solutions that can recover areas affected by clouds without requiring both cloudy and cloudless images of the same scene. This thesis aims to explore methods that address this gap. A significant portion of the work has been devoted to exploratory research, leading to the design and testing of various subsystems. The pipeline has been primarily focused on three key areas: reconstructing compressive sensed images, detecting and removing clouds, and the task of reconstructing missing areas. The objective is to explore potential solutions within the context of HYPSO-1 images. The different subsystems are noted in Fig. 4.2.

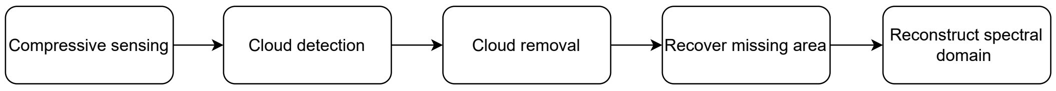


Fig. 4.2: Overview of all the sub-parts explored in this thesis and how they connect together.

In the following sections, all aspects of the explored systems will be described in detail. The first section introduces the data set and preprocessing, while the next one discusses basic building blocks for the system, such as hardware, algorithms for training, predicting, loading data, and evaluation. These sections are general and are used for all models designed and trained throughout the thesis. The following three sections describe the parts of the final pipeline, including some additional exploration. This includes the reconstruction of HSI compressive sensed in the spectral domain, cloud detection and removal with focus on the HYPSO-1 images, and how areas lost due to cloud contamination can be recovered.

4.1 The HYPSO-1 data set

All images used are captured by the HYPSO-1 satellite. The HYPSO-1 HSI data set currently contains about 1500 images of the earth's surface from across the globe captured in the period 4Th March 2022 up to date. Images are collected at 956x684 spatial resolution and a spectral resolution of 120 [9]. The data is stored in binary files which contain the image data, and header files which contain the metadata about the images needed to read the binary files. Due to memory and time restrictions, only 57 images from the HYPSO-1 data set are used in this thesis. Fig. 4.3 shows four HYPSO-1 images captured at different locations.

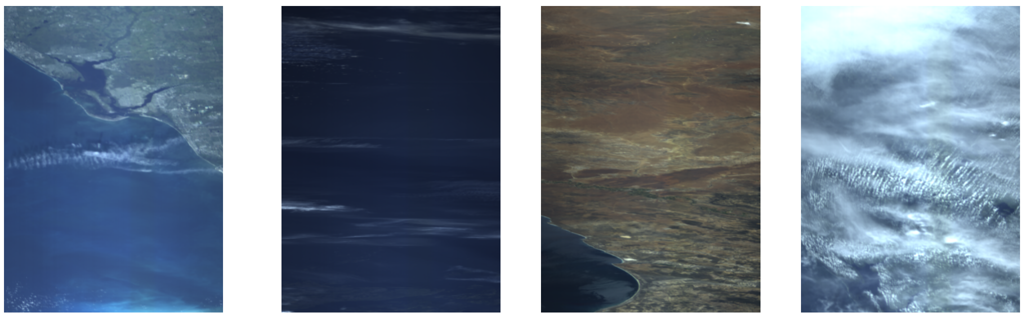


Fig. 4.3: Four selected HYPSO-1 images showcasing their diverse nature.

Working with HYPSO-1 images introduces certain limitations compared to other existing solutions. Although there are certain overlaps in some images, no images of the exact same location are captured close in time. Hence, labeled data containing images with and without clouds is unavailable to train a neural network. Consequently, solutions like the one presented in [5] are excluded from consideration. Additionally, cloud masks are not available for HYPSO-1 images at this moment. Hence their generation is necessary, with visual inspection being the primary measure of quality. Although data sets with cloud masks do exist, time constraints prevent their incorporation within the scope of this thesis. Another limitation arises from the diverse nature of HYPSO-1 images, making it challenging to identify effective general approaches. However, this diversity may contribute to improved model generalization.

4.2 Image preprocessing

Some preprocessing is necessary to deal with the images in a machine-learning sense. The preprocessing includes compressive sensing and test/train data split. Fig. 4.4 shows a flowchart of the steps of the prepossessing for the HYPSO-1 data set.

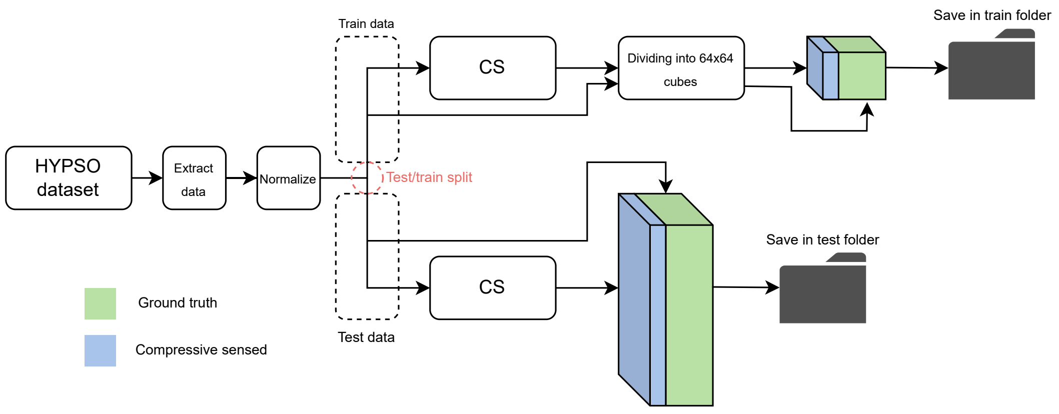


Fig. 4.4: Flowchart illustrating the different parts of the preprocessing stage. Note that the figure depicts only one compressive sensing/ground truth cube for the training branch, but multiple cubes are used.

The following steps are executed in the preprocessing:

1. Read information from the header file and extract data from the binary file using the header information.
2. Normalize image. (max-normalization)
3. Train/test split.
4. Compressive sensing of the images, with a description of the sensing matrix and compression ratio.
5. Divide the training set into smaller cubes.
6. Saving the pre-processed in dedicated folders.

In the following subsections, all the preprocessing steps are described in detail.

Data extraction

The header files provide details about the height, width, number of bands, data type, and other information for all images. As images from the same set may have some differences, it is essential to read this information for each image individually. To achieve this, the Python package spectral is utilized to read the binary files and extract data as per the format specified in the header file.

Normalizing the images

To ensure accurate reconstruction, the images undergo max-normalization. This process involves determining the maximum pixel value in each complete image and dividing the entire image by this value. As a result, the normalization produces images with values between zero and one while preserving their relative spatial information.

Compressive sensing of the images

To successfully compress a signal using compressive sensing, it's important that the signal is sparse or compressible in some domain. Even when using neural networks for reconstruction instead of traditional mathematical optimization algorithms, it's necessary to consider the sparsity level when choosing the compression rate to make meaningful comparisons of the results.

For hyperspectral images, the spectral domain isn't typically sparse. However, by applying the DCT, it's possible to obtain an approximately sparse representation of the signal. The DCT transform of the spectral vector shown in Fig. 4.9 is presented in Fig. 4.5. It's evident that using the DCT transform, a significant number of the coefficients are close to or equal to zero. However, determining an exact sparsity level, κ, is challenging.

Based on the lower bound presented in chapter 2.9.3 with regards to M, it is possible to determine an estimate of the number of samples that should be selected for use in compressive sensing. By solving the equation given in (2.20), a lower bound for the number of samples needed for RIP to hold with a high probability can be calculated. Fig. 4.6 shows a lower bound for $N = 120$, e.g. the number of spectral bands for the HYPSO-1 images. To guarantee to take enough samples, while still having good compression, the compression rate is set to 5, meaning that 80% of the data is discarded. This means that the number of samples, M, for the HYPSO-1

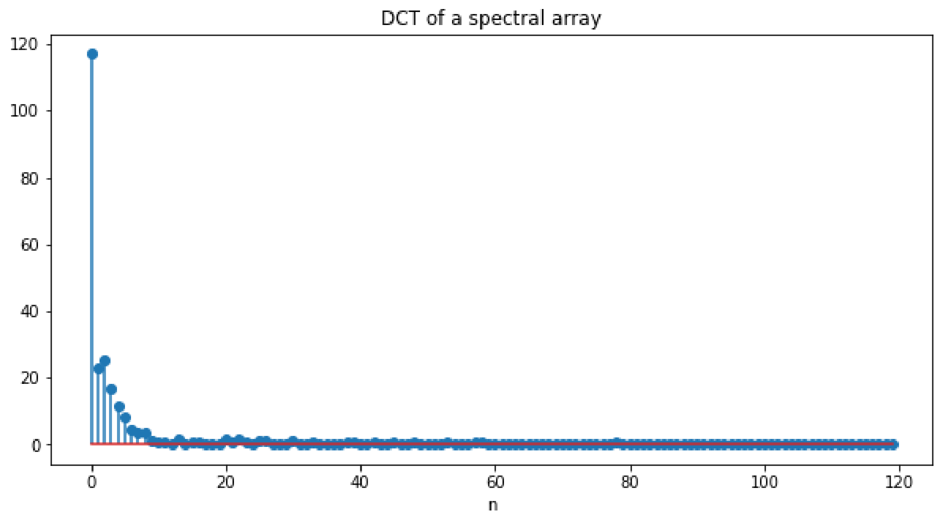


Fig. 4.5: The discrete cosine transform of a spectral array of a HYPSO-1 image.

images, is 24. This is still a high compression rate compared to classical compression methods for HSI, which is often around 2 [10].

Sensing Matrix

The sensing matrix is a simplification of the CS-MUSI. Instead of using a set of linear combinations of the spectral bands, M bands are selected and the rest is discarded. To ensure incoherent sampling, the sensing matrix, Ψ, is made by randomly choosing which bands to keep. The compressive sensing process is then done by applying the sensing matrix to the spectral domain of each pixel in the HSI. It is important that the same sensing matrix is used on all the images for one model, in order to train the neural network in a consistent and effective manner.

Fig. 4.7 shows the sensing matrix that is used in this project. The matrix is plotted as an image, where yellow pixels symbolize 1 and purple pixels symbolize 0. This sensing matrix was selected randomly, and stored in a text file for later use.

The sensing matrix is applied to the spectral vector of all the pixels in the HSI, as previously mentioned. This is presented in Fig. 4.8. Fig. 4.9 shows an instance of the spectral domain of one pixel with its CS version. The initial spectral domain is drawn using a blue line, while the compressed representation achieved using the sensing matrix is displayed using yellow $\times$'s. The compressed representation only has 24 bands, compared to the original 120 bands, which significantly reduces the data size. This demonstrates how CS can efficiently compress the spectral information of an HSI.

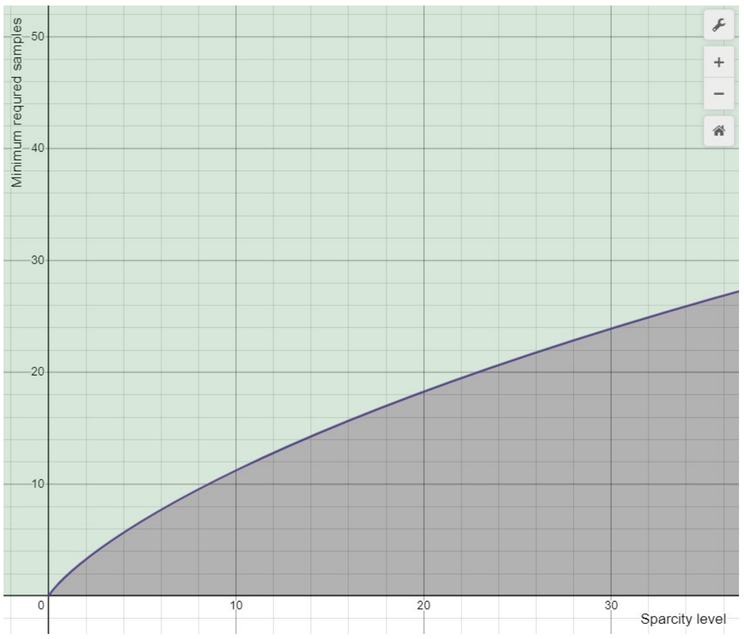


Fig. 4.6: This graph shows the minimum number of samples needed for RIP to have a high probability of holding. The y-axis represents the compressive samples, M, while the x-axis shows the signal's sparsity level, κ. The equation given in (2.20) is used to calculate the function by solving it for M, with $N = 120$. The green area indicates values above the lower bound, while the gray area indicates values below. The graph is created using Desmos.

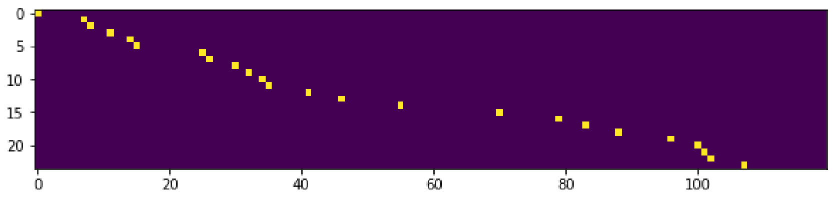


Fig. 4.7: The sensing matrix, denoted by Ψ, is a matrix with dimensions of $M \times N$ and it comprises ones and zeros. Each row has only one singular one. In the image, the yellow pixels represent 1 while the purple ones represent 0. Compressive sensing involves multiplying this matrix with all the spectral vectors in an HSI cube.

Splitting the images into train and test data-sets

The train/test split is set to 0.7/0.3, meaning 70% of the images are used for training and 30% for testing. Since there are 57 HYPSO-1 images downloaded and used, 40 are used for training and 17 for testing.

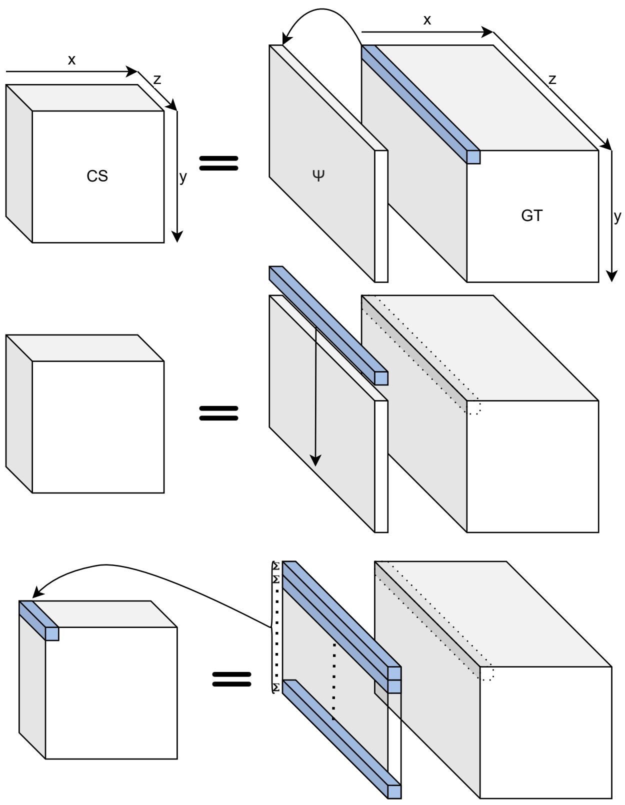


Fig. 4.8: This figure shows how the sensing matrix, Ψ, is used to compress the spectral domain of each pixel in the original image, which is referred to as GT in the figure. The resulting compressed image is named CS. It's important to note that the spectral domain, represented by z, is shorter in the compressed image compared to the ground truth.

Dividing training data into smaller cubes

Hyperspectral images are partitioned into smaller cubes to allow for efficient processing by the NN without exceeding GPU memory constraints. This approach also reduces the size of the neural network, resulting in faster computation and less risk of over-fitting. For training the data set, the images are divided into $64 \times 64 \times 24$ cubes for compressive sensed images and $64 \times 64 \times 120$ for ground truth images. When training the NN, these cubes act as input (cs) and label (ground truth). Each new cube is generated by shifting a 64×64 block by 32 pixels, resulting in the number of cubes given by

$$N_{cubes} = \texttt{floor}(\frac{(2x_i - 64)}{64}) \cdot \texttt{floor}(\frac{(2y_i - 64)}{64}). \tag{4.1}$$

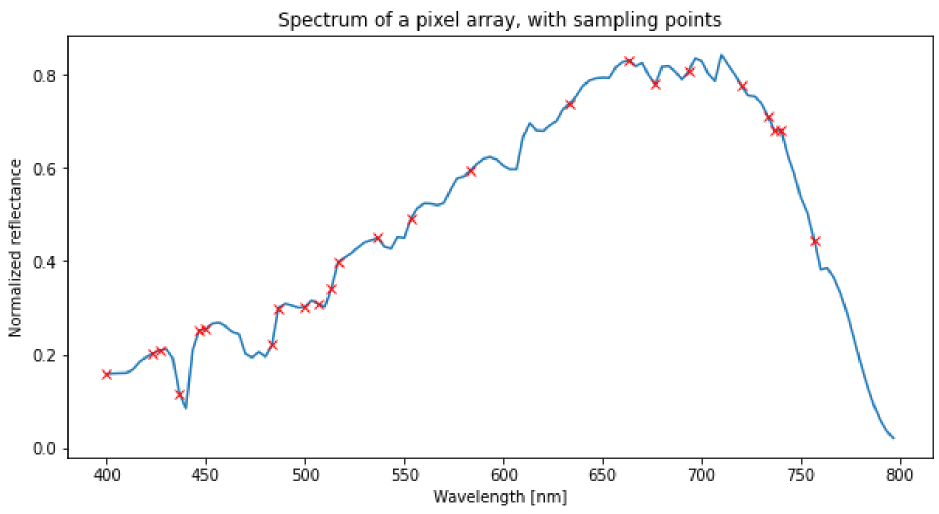


Fig. 4.9: This is an example of compressive sensing applied to the spectral domain of a single pixel. The original spectral domain is represented by the blue line, while the orange dots indicate the specific points used in the compressive sensing process. It's worth noting that only 20% of the original data points were utilized.

The variables x_i and y_i represent the number of pixels in the x- and y-direction. Fig. 4.10 illustrates a cube divided into smaller cubes. For the HYSPO-1 images this results in 560 cubes per HSI.

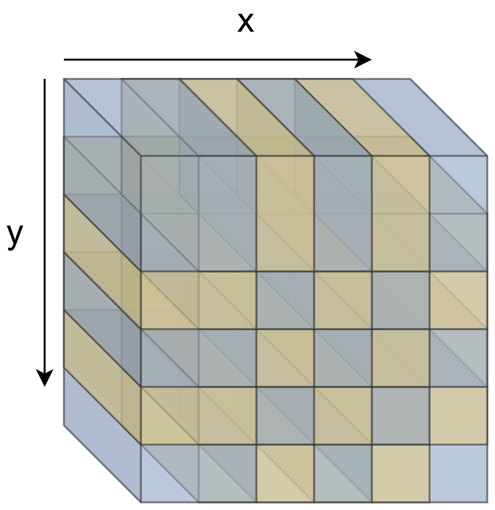


Fig. 4.10: In this illustration, an HSI cube is shown divided into smaller cubes with overlapping sections. The cubes highlighted in blue and yellow alternate every other one. It is important to note that only the spatial domain is being divided, while the spectral domain (which is pointing inward in the image) remains whole.

There are two benefits to using this method. First, it allows for more training data to be used, which in turn enhances the neural network's ability to learn. Second, it enables the network to learn from the borders of one block that are located in the middle of another block, improving its ability to make generalizations across various parts of the images.

Saving the pre-processed in dedicated folders

The training cubes, as seen in Fig. 4.11, are saved in the train folders as img-files accompanied by their corresponding hdr-files. The test images are chosen based on their names from the "full$_{i}mages''folder, andthentheyundergocompressivesensinginthepredictionalg$

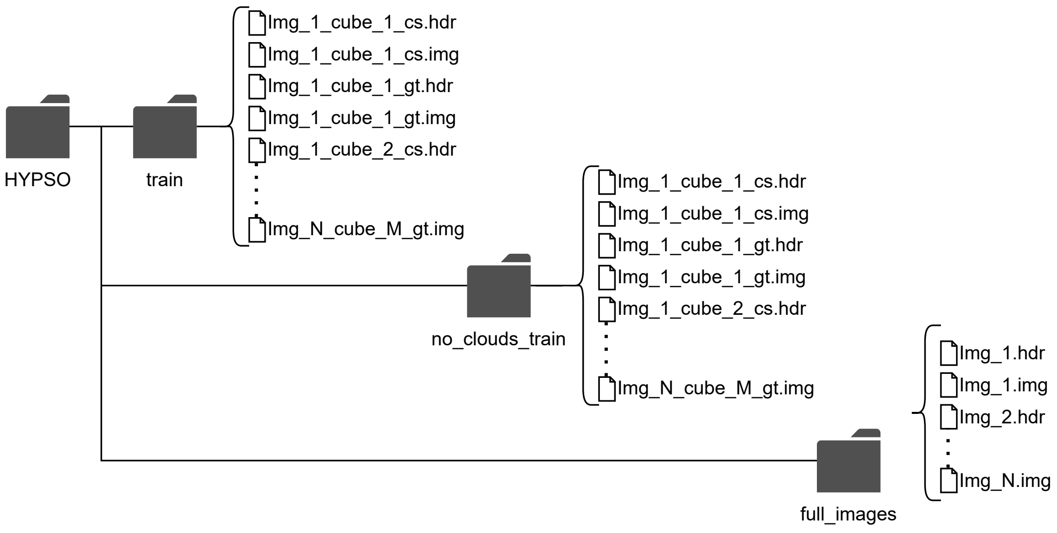


Fig. 4.11: The folder structure for the pre-processed images.

4.3 Neural network implementation with PyTorch

The Python package PyTorch is used for implementing all training and predicting functions, which is further elaborated in section 2.8. While PyTorch has many tools for training and prediction, it requires customization to fit the specific task. To enhance computational efficiency, an NVIDIA GeForce RTX 3080 GPU is utilized as it offers faster processing speeds than standard CPUs. The thesis uses various NN architectures, which are generally large and share similar building blocks. The following subsections explain the fundamental building blocks and functions for data loading, model training, and prediction using trained models.

4.3.1 The building blocks

The building blocks consist of smaller groups of functional components that can be found in the torch.nn library or other libraries compatible with PyTorch and tensors. This method simplifies the process of designing and arranging intricate neural networks with a diverse array of architectures and functions.

Upscale-Block

The “UpScaleBlock”, illustrated in Fig. 4.12, is a key component of the proposed system that is responsible for upscaling a compressed HSI to its original, uncompressed dimension. To accomplish this, the block utilizes the upsample class provided by the torch.nn module, which performs bilinear interpolation to scale up the tensor to the desired size based on the compression rate. As described in section 4.2, a compression rate of 5 is used. Following the upsampling operation, a 3D convolution with a kernel size of (1, 1, 1) is performed to enhance the quality of the upscaled data. The resulting tensor is then passed to the next block in the network for further processing.

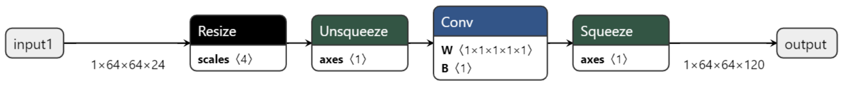


Fig. 4.12: The “Upscale-block” with all the operations done. The image is created using torch.onnx and Neutron.

Fig. 4.12 shows all the operations performed in the upscale. This method of plotting is however not convenient for larger models, hence a simplification, shown in figure 4.13 will be used in the larger models.

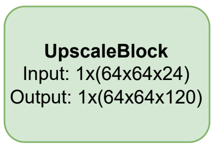


Fig. 4.13: A simple model of the “Upscale-block”.

Down Block

The “DownBlock” is depicted in Fig. 4.14. The figure shows that the input is passed twice through 3D convolutional blocks with kernel size (3, 3, 3), with a ReLU block after each convolutional block. Following this, max-pooling with a pool size of (1, 1,

2) is applied to the image, which results in reducing the third axis of the tensor, or the spectral dimension, to half its original size.

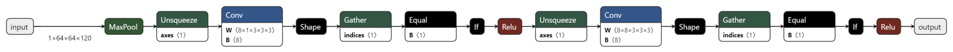


Fig. 4.14: The “DownBlock” with all its operations. The image is created using torch.onnx and Neutron.

To be able to perform the convolutions and pooling operations, PyTorch resizes and squeezes the data based on needs. Fig. 4.14 shows all the resizing and squeezing done by PyTorch. When using this block in bigger models, the block is abstracted, as illustrated in Fig. 4.15, to make the schematics easier to understand.

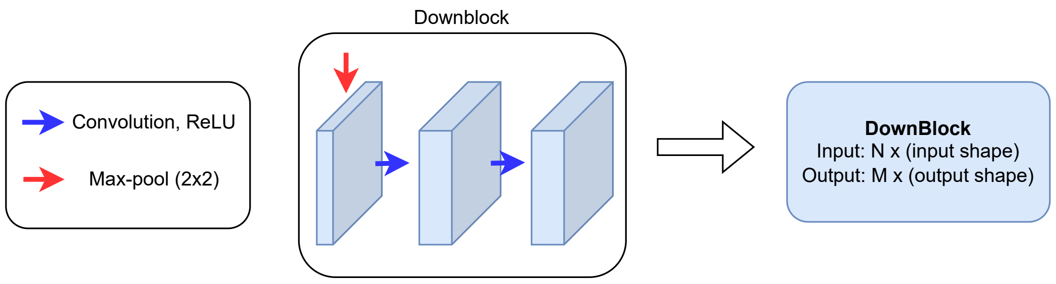


Fig. 4.15: A simplified model of the “DownBlock”. The middle part of the figure is from the u-net in Fig. 2.13, and is also correct for this case. The input is marked with the red arrow, and the output is illustrated with the rightmost block.

Up Block

The "UpBlock" is another important building block of the proposed system, which is responsible for up-sampling the HSI data on the up-sampling path of the U-net. The block first applies two 3D convolution operations with kernel size (3, 3, 3), followed by an up-sampling operation using the "Upsample" class provided by the torch.nn module. The resulting tensor is then concatenated with the corresponding tensor from the down-sampling path of the U-net, and a final 3D convolution operation, merging the concatenated data into the right shape, is performed to produce the output.

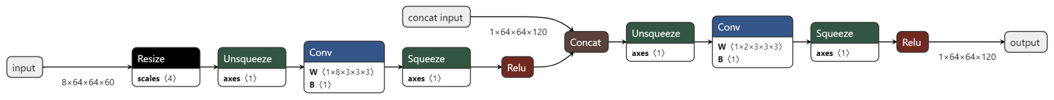


Fig. 4.16: The “UpBlock” with all its operations. The image is created using torch.onnx and Neutron.

Fig. 4.14 depicts the resizing and squeezing carried out by PyTorch for the operations. To ease understanding in larger models, the figures are denoted in a simplified manner, as illustrated in Fig. 4.17.

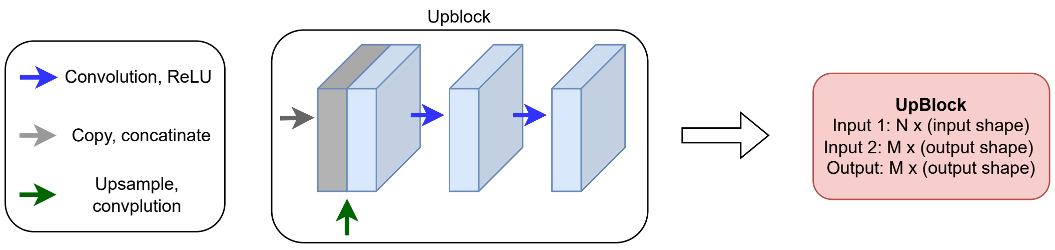

Convolution, ReLU
Copy, concatinate
Upsample, convplution
Upblock
UpBlock
Input 1: N x (input shape)
Input 2: M x (output shape)
Output: M x (output shape)

Fig. 4.17: A simplified model of the "UpBlock". Input 1 is marked by the green arrow, while input 2 is marked by the gray arrow. The output is illustrated by the rightmost block.

Reset values block

The reset values block uses the input data and the sensing matrix, $\mathbf{\Psi}$, to reset the inputted values to the correct positions. Since this is done on the Cuda device, the function `torch.put` needs to be used.

To use this function, the positions and values must be inputted in a specific way. The indices need to be organized in a second-order tensor with the first axis representing the tensor order and indicating the exact positions for the values. The values themselves should be flattened into a first-order tensor with the correct values in order. If formatted correctly, the line of code, `x.index_put(tuple(self.A_indices), torch.flatten(f))`, will insert the values `f` into `x` at the positions indicated by `A_indices`. Please note that this can only be done in parts of the neural network where the spectral dimension of the HSI is not compressed, as the sampling indices are only valid in these areas.

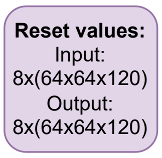


Fig. 4.18: The simplified model notation of the reset values block.

Convolution

This block represents a single convolution and is commonly utilized as an output layer in many neural networks. However, it can also be implemented in other sections of the network. Please refer to Fig. 4.19 for the notation of a singular convolution.

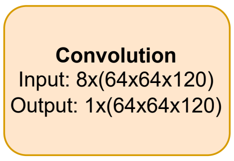


Fig. 4.19: The simplified notation for a singular convolution.

4.3.2 Loading data

Loading data with PyTorch requires tailored versions of the class "torch.utils.data.Dataset", suited for the data to be loaded. Since the training and testing data have different sizes, as described in section 4.2, different sub-classes have to be made for each. In addition to loading the data, it must have the right data structure and type. After prepossessing, the HSIs are stored as image files, later reloaded as binary files, and transformed to `torch.Tensors`.

4.3.3 Training the neural networks

The optimizer ADAM is used due to its superior convergence and results. Different learning rates are tested, but experiments show that a learning rate of 0.0005 or lower yields similar results with varying convergence speeds. However, using a higher learning rate than this tends to converge to a local minimum, resulting in poorer outcomes. The batch size is set to one, and the data is shuffled for each epoch. To load the data, all 16 CPUs of the computer are utilized for maximum efficiency.

The mean squared error between the ground truth and the output is used as the loss function. In addition, a small weight decay of 10^{-8}, is added to the optimizer function to avoid that a few of the weights play a too big part. Once all the relevant variables and functions are defined, the training data is iterated through, and the NN's weights are updated.

The packages tqdm and TensorBoard are used to monitor the training process in real-time. Tqdm provides progress updates, displaying the number of completed epochs, remaining epochs, progress within the current epoch (in percentage, number of images, and time), and the average loss of the last 100 batches. TensorBoard plots the loss as a function of the number of training cubes iterated through.

4.3.4 Predicting

The size of the input data used to train the model is $64 \times 64 \times 24$, while the HYPSO-1 images are compressed and sized at $956 \times 684 \times 24$. When reconstructing the images, they are divided into smaller blocks of $64 \times 64 \times 24$, which are individually reconstructed using the trained model and then patched together. Due to the spatial dimensions of the HSIs not being evenly divisible by 64, some portions of the image are left out during the reconstruction process. To make predictions, the model is put into evaluation mode, and the compressive sensed data is used as input to the model.

4.3.5 Evaluation functions

The quality of the reconstructions is evaluated using five different functions: PSNR, SSIM, SAM, SID, and MSE. These functions are described in section 2.4. To calculate SSIM, the Python package skimage.metrics is utilized, which requires two images and a data range as input. PSNR, SID, SAM, and MSE are calculated as described in sections 2.4.2, 2.4.5, 2.4.4, and 2.4.1, respectively. These evaluation functions are applied to both the reconstructed image and the ground truth. To simplify the evaluation process, a function is created to iterate through all the reconstructed images and calculate the average SSIM, SAM, PSNR, SID, and MSE for each image.

4.4 Reconstructing the spectral domain of compressive sensed HSIs using u-nets

Different u-net architectures for reconstructing compressive sensed HSI data are evaluated in this thesis. They are all based on the NNs presented in [43], [44], which are implemented using TensorFlow. The models in this project are however implemented using PyTorch, so some variation due to the library might occur.

4.4.1 The master model

This first model is the PyTorch implementation of the DeepCubeNet, tailored to fit the HYPSO-1 images same was as in [44], and is given the name master model. The aim is to achieve similar results as the TensorFlow implementation on the

HYPSO-1 images. The first up-sampling layer in this implementation is designed to be trainable, unlike the one in the DCN, where the layer parameters are set to be untrainable. However, the desired functionality was not successfully achieved due to challenges encountered during implementation. Apart from this particular difference, there are no other known discrepancies between the two implementations.

The architecture of this network, using the blocks described above, is presented in Fig. 4.20. As the DCN, the master model has a depth of three and utilizes 3D convolutions.

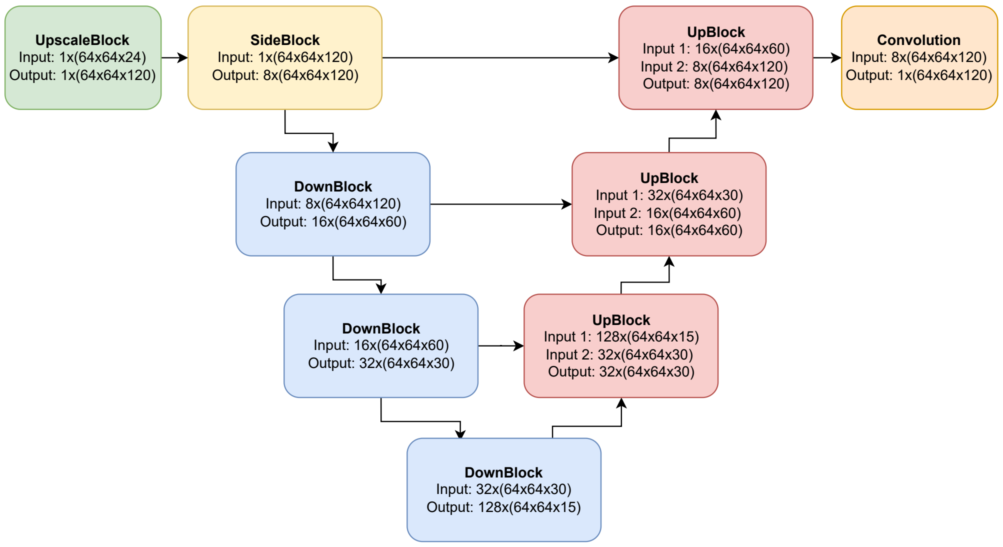


Fig. 4.20: The master model. It has the same architecture as the DeepCubeNet but is implemented using PyTorch and tailored to be usable with the HYPSO-1 images.

4.4.2 The simple model

When designing a neural network, it's important to aim for simplicity while maintaining high performance levels. To this end, it's worth investigating if similar results can be obtained through a simpler U-Net architecture. A simplified version of this architecture, named "simple model," is shown in Figure 4.21. This model is a 3D convolutional U-Net with a depth of two.

4.4.3 The backfire model

As noted in prior studies, there may be inaccuracies in the reconstructed data, even if the input values were correct. This neural network addresses this issue by resetting

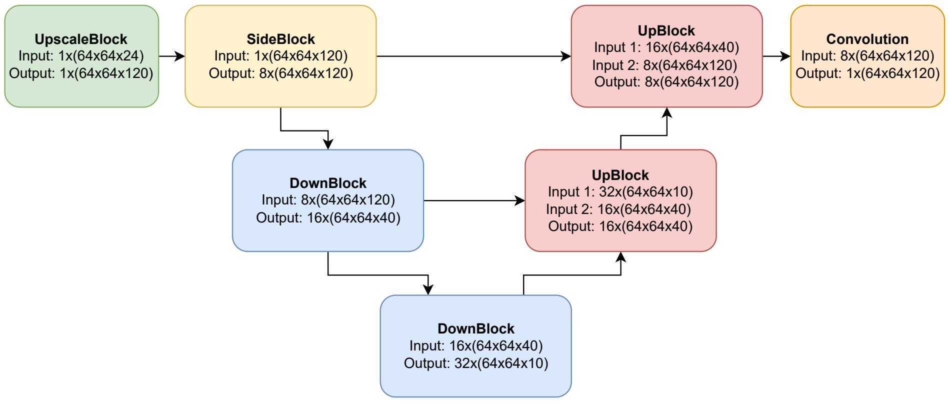


Fig. 4.21: A u-net with depth 2. This is a simplification of the DeepCubeNet

the erroneous values to their proper positions within the network. This is achieved by utilizing a reset-values block prior to the final convolution, as illustrated in figure 4.22. The overall architecture remains identical to the original model.

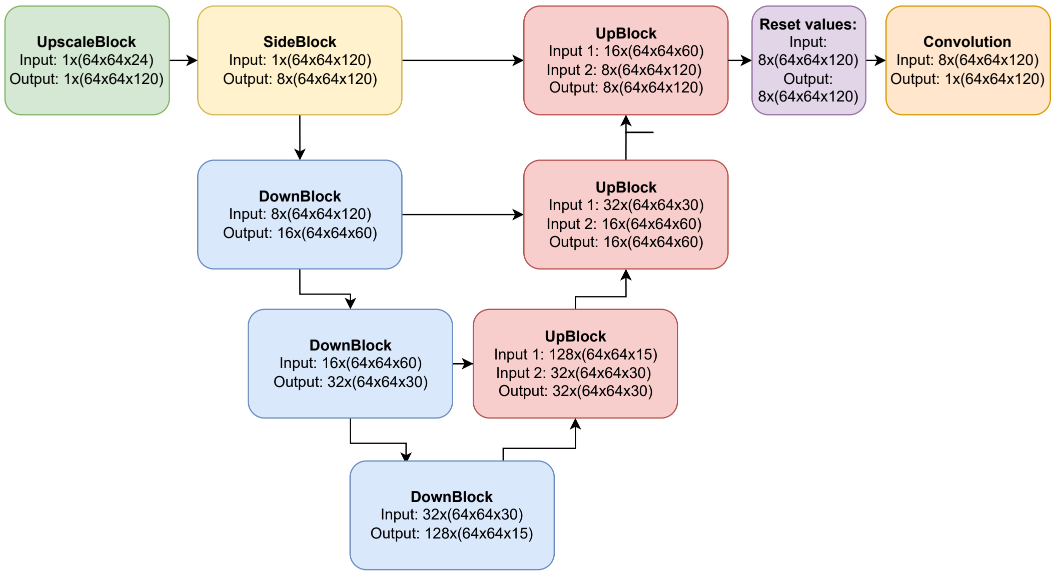


Fig. 4.22: The backfire model, which resets the know values to their correct values before the final convolution.

4.5 Cloud detection and segmentation

As described in section 2.1.2, the spectral signature of clouds is diverse. What a hyperspectral camera will measure is also dependent on the ground beneath the cloud, and the camera settings; hence the clouds are not easily separable from the background. Nonetheless, there exist several methods to detect them, as described in section 3.2. However, the functionality cannot easily be transferred to use for the HYSPO images. This is because some of them require an existing cloud mask, some require images taken at the same spot close in time, and other models are dependent on a specific range of frequencies in the bands. For the HYPSO-1 images, none of these are available. Therefore specific cloud detection algorithms have to be created.

Fig. 4.23 shows a plot of all the spectral arrays of an HYPSO-1 image that has both land, ocean, and sea. One can see that there is a continuous range of spectral signatures and that only using a threshold might not be a good approach.

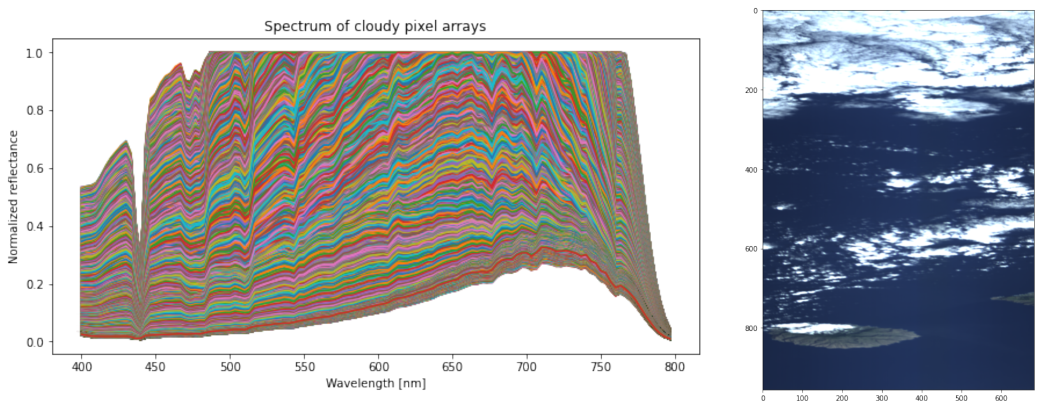


Fig. 4.23: A plot of all the spectral arrays in the image to the right.

4.5.1 Using clustering to make cloud masks

The k-means function delivered by Spectral Python (SPy) is used to divide the spectral vectors of an image into a given number of clusters based on a distance function. The different distance functions used are:

1. MAE
2. MSE
3. SAM
4. SID

The SPy function only has the option to use MAE and MSE. Thus, the SAM and SID are implemented by editing the installed package code. The goal is to separate clouds from the rest. For many images, the content is quite complex; thus, two clusters might not be the optimal number of clusters. Therefore, various numbers of clusters are explored.

Threshold and correlation-based cloud detection

Fig. 4.24 shows three examples of cloud-contaminated spectral arrays from three different HYPSO-1 images. The pixels are selected manually to be in the center of thick clouds. It is clear that the spectral signature is different from the three pixels, but they all have maximum values that are equal to 1. Even though the spectral signatures differ from the different clouds, pixels within the same clouds often have similar ones. These observations are utilized to make a function that combines thresholding and spectral comparison to detect and generate cloud masks.

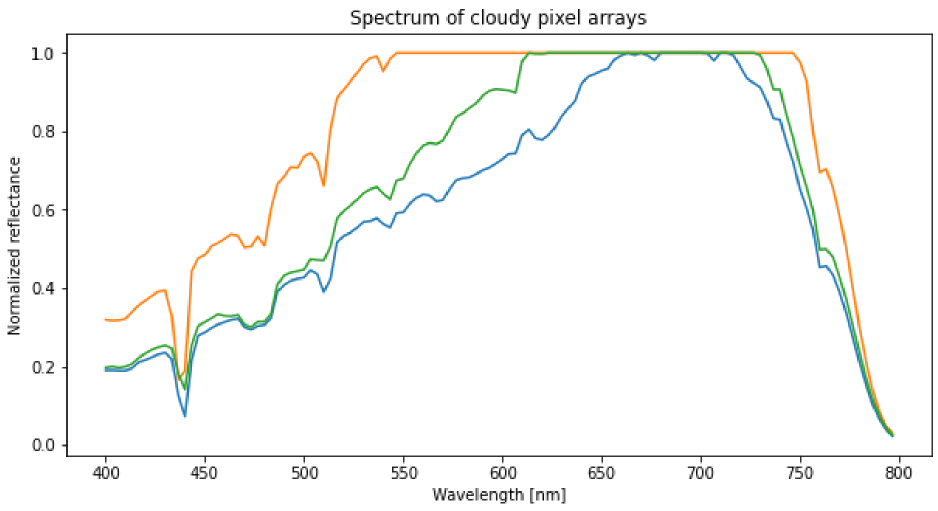


Fig. 4.24: Example of spectral arrays of cloud contaminated pixels from the HYPSO-1 dataset.

The function, named `spreading_cloud_detection_queue`, performs threshold-based cloud detection on an inputted HSI followed by an expansion process. In addition to the HSI to be analyzed, named `pred_img`, the function takes in the following variables and their default values; `queue_threshold=0`, `check_range=2`, `neighbor_threshold=None`, `threshold=None`, and `plotting=False`. The function returns a cloud mask with zeros and ones, where a one indicates cloud and a zero indicates no cloud. The first thresholding is done to capture the central thick cloud pixels, like the ones in Fig. 4.24. The expansion should add the surrounding cloud pixels that might not be saturated, hence not captured by the threshold. The threshold, `threshold`, is either set manually or calculated using the mean and standard deviation of the image. This threshold is applied pixel-wise by taking the mean of the 10 highest values in the spectral array, making a second-order representation of the HSI cube with ones and zeros. This is the cloud mask. An empty mask is returned if the number of cloud pixels identified through this thresholding is zero. If the number of detected cloud pixels is greater than 90% of the total number of pixels in the image, the function assumes that the image is heavily covered by clouds and returns the mask as it is, as the mask would have no meaningful value. If the number of cloudy pixels found is between these two thresholds, the function proceeds to identify neighboring pixels that have similar spectral signatures. All the cloud pixels identified through the first thresholding are additionally added to a queue structure with a first-in-first-out (FIFO) priority. In addition to this first threshold, two other thresholds are used; the `neighbor_threshold`, which is the threshold for adding a neighboring pixel to the cloud mask, and the `queue_threshold`, which is the threshold for adding a neighboring pixel to the FIFO-queue. The code listing below shows how the queue is iterated through and how the value, `val`, used for the `neighbor_threshold` and `queue_threshold` is calculated.

```python
while len(queue) > 0:
    node = queue.pop(0)
    i, j = node[0], node[1]
    main_pixel = pred_img[node]
    for k in range(-check_range, check_range+1):
        for l in range(-check_range, check_range+1):
            #Checking if the pixel is within the image borders
            if not (i+k>=0 and i+k< max_img.shape[0] and j+l>=0
                    and j+l < max_img.shape[1]):
                continue
            #Checking if the pixel is already in the cloud mask
            if (not max_img[i+k, j+l]):
                main_pixel = main_pixel-torch.mean(main_pixel)
                check_pixel = pred_img[i+k, j+l, :] -
                            torch.mean(pred_img[i+k, j+l, :])
                main_dot = torch.dot(main_pixel, main_pixel)
                val = torch.dot(main_pixel, check_pixel)/main_dot
```

The `main_pixel` is the detected cloud pixel, and the `check_pixel` is a neighboring pixel. The parameter `check_range` determines how many surrounding pixels will be compared to each detected pixel. Since one iterates in both negative and positive directions, for both x and y axis in the image, the number of compared pixels is calculated as $N_{compare\_pixels} = (2 \cdot check\_range)^2 - 1$. Once a pixel is added to the cloud mask, it will not be compared to other cloud pixels again to avoid going into loops. The function also allows generating a GIF animation of the expasion process by setting the plotting parameter to `True`.

4.6 Removing clouds

The clouds are removed by setting all the values that are within detected cloud pixel arrays to zero. This is implemented through a function that takes in a hyperspectral image and an associated cloud mask and sets the values in the HSI corresponding to the cloud mask to zero. All operations are performed on the GPU for optimal workflow and efficiency.

4.7 Recovering missing areas

This part of the system aims to recover missing data in the HSIs due to cloud removal. A U-net similar to those used for CS reconstruction is used to do this. In addition to the building block blocks described in section 4.3.1, blocks using 3D convolution with dilations, hereby called dilation blocks or dilated convolutional blocks, are utilized. Two different architectures are trained and tested. The models use several dilation blocks with different purposes, to fill in the missing areas with non-zero values, and to extract different frequency information about changes within the image to generate accurate reconstructions. Bigger kernels and larger dilations capture slower changes, while smaller kernels and dilations capture rapid and local changes. One of the models resets the known pixels to their correct values at the end of the model. The spectral dimension of the data in the models can have all sizes, given that it is big enough to be down-sampled in the down-sampling path of the UNet. However, the models are trained on compressive sensed HSIs from the HYSPO data set, with a size of 24 samples in the spectral domain.

4.7.1 Recovery Model 1

The architecture of this model, shown in Fig. 4.25, is based on an UNet structure with a depth of 2, but some of the normal convolutional blocks have been replaced with dilated convolutional blocks. The dilation blocks are placed in both the down-sampling and the up-sampling path.

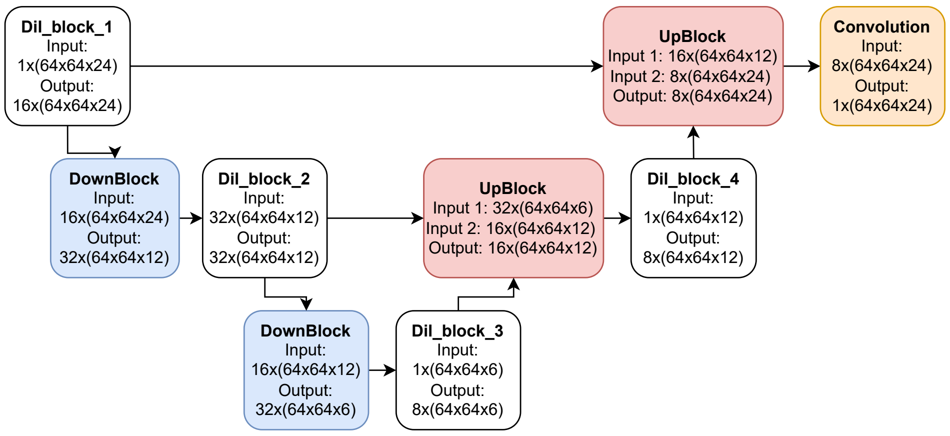


Fig. 4.25: The architecture of the Recovery Model 1, with each building block's input and output dimensions.

Table 4.1 describes the four dilation blocks used in this model. Each block is designed with a specific objective, and their collective cooperation aims to extract sufficient information for an accurate reconstruction.

4.7.2 Recovery Model 2

In this model, depicted in Fig. 4.26, dilated convolutional blocks are only used in the down-sampling path of the UNet, with a depth of three. Since the UNet has crossover connections from the down-sampling path to the up-sampling path, one can assume that dilation blocks exclusively on the down-sampling path are adequate. Moreover, the known pixel values are reset to their original values twice within the network, following a similar approach to the backfire model discussed in Section 4.4.3. Detailed descriptions of the different dilation blocks can be found in Table 4.2.

Resetting the values for an image lacking an unknown number of pixels requires additional processing compared to the previously described resetting block to avoid resetting the zero values within a cloud mask. Since each input has different and unknown missing parts, specific indices for resetting need to be determined for each input. A method defined within the model class deals with this by identifying the non-zero pixel indices and formatting the indices appropriately. This function is called once for every new input and is a time-consuming process. Another method that correctly applies the resetting on the input is performed twice within the neural network. First, after the first dilation block to recover information lost due to the size of the kernels and dilations, and the second right before the final convolution, Tab. 4.1: The dilated convolutional blocks used in Recovery Model 1, with a small description of the purpose of each block. Full architecture can be seen in image 4.25.

Block name	Parameters	Purpose
Dil_block_1	Dilation = 3, Kernel size = 7	The block aims to fill in the missing values by using a large kernel and dilation, then a smaller.
	Dilation = 1, Kernel size = 3	
Dil_block_2	Dilation = 3, Kernel size = 5	This block uses large kernels to fill in missing values. Three different dilations are used to extract frequency information within the image.
	Dilation = 1, Kernel size = 5	
	Dilation = 2, Kernel size = 3	
Dil_block_3	Dilation = 3, Kernel size = 3	This block aims to fill in finer details in the reconstruction, based on surrounding pixels.
	Dilation = 1, Kernel size = 3	
Dil_block_4	Dilation = 2, Kernel size = 3	This block aims to do a final touch, optimizing the reconstruction quality.
	Dilation = 1, Kernel size = 1	

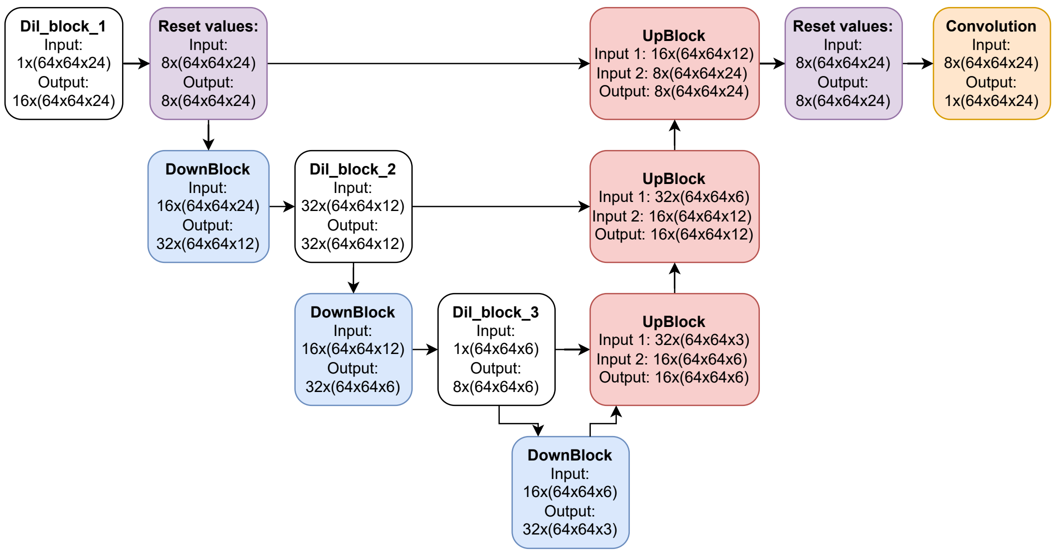

Dil_block_1 Input: 1x(64x64x24) Output: 16x(64x64x24)
Reset values: Input: 8x(64x64x24) Output: 8x(64x64x24)
UpBlock Input 1: 16x(64x64x12) Input 2: 8x(64x64x24) Output: 8x(64x64x24)
Reset values: Input: 8x(64x64x24) Output: 8x(64x64x24)
Convolution Input: 8x(64x64x24) Output: 1x(64x64x24)
DownBlock Input: 16x(64x64x24) Output: 32x(64x64x12)
Dil_block_2 Input: 32x(64x64x12) Output: 32x(64x64x12)
UpBlock Input 1: 32x(64x64x6) Input 2: 16x(64x64x12) Output: 16x(64x64x12)
DownBlock Input: 16x(64x64x12) Output: 32x(64x64x6)
Dil_block_3 Input: 1x(64x64x6) Output: 8x(64x64x6)
UpBlock Input 1: 32x(64x64x3) Input 2: 16x(64x64x6) Output: 16x(64x64x6)
DownBlock Input: 16x(64x64x6) Output: 32x(64x64x3)

Fig. 4.26: The architecture of the recovery model 2, with each building block's input and output dimensions.

to recover information lost in the rest of the network. The resetting ensures no loss of known information in the recovery process, but at the cost of speed.

Tab. 4.2: The dilated convolutional blocks used in Recovery Model 2, with a short description of the purpose of each block. Full architecture in Fig. 4.26.

Block name	Convolution parameters	Description
Dil_block_1	Dilation = 3, Kernel size = 7	This block uses a large area of surrounding pixels to fill in values for the missing pixels, capturing slow changes in the image.
	Dilation = 2, Kernel size = 5	
	Dilation = 1, Kernel size = 5	
Dil_block_2	Dilation = 3, Kernel size = 5	This block first analyzes moderate changes, then examines finer details with a smaller kernel.
	Dilation = 1, Kernel size = 3	
Dil_block_3	Dilation = 2, Kernel size = 3	This block first analyzes moderate changes, then examines finer details with a smaller kernel.
	Dilation = 1, Kernel size = 3	

4.7.3 Training and predicting with the recovery models

Compressive sensed HYPSO-1 images are used to train the NN. Both images with missing data and ground truth are needed in the training. For the HYPSO-1 images, there are no images that have the exact same spot and lighting with and without clouds, hence using actual cloud masks to train the network is not a possible method. In addition, for good generalization, lots of data are needed. A function for generating fake cloud mask, described below, are used instead of real cloud masks, and only images without clouds are used. To generate loads of training data and avoid overfitting, new random cloud masks are generated for each training cube in each iteration. This ensures the NN actually learns to reconstruct from the surrounding pixels rather than memorizing some particular cases.

Function for simulating cloud masks

The function “generate_fake_clouds” generates cloud masks for given input images. It optionally takes in two additional parameters, the probability for a new pixel being added to the mask, `prob_of_clouds`, and the maximum number of pixels allowed in the mask given as a fraction of the total number of pixels in the image, `maximum_size`. The default values for these two parameters are 0.249 and 0.25, respectively. The function starts by creating a binary mask of the same size as the input, with all elements initially set to 0. A few seed pixels within the image are selected at random locations, and the corresponding locations in the binary mask is set to 1. The number of seed pixels is selected within the range of 2 and $\log(\text{Total number of pixels in the image}) + 2.5$. The function then iteratively expands the mask by adding the neighboring pixels. The probability for adding a neighboring pixel is given by the parameter `prob_of_clouds`. The expansion process uses a Last-In-First-Out (LIFO) queue, meaning that the last neighboring pixel added to the queue is the next one to explore. This is done to avoid generating circular clouds centered around the seed pixels, which will happen if a First-in-First-Out (FIFO) queue is used. The algorithm stops when the maximum number of pixels allowed in the cloud mask is reached or the queue is empty. The function returns the final cloud mask as a binary matrix with the same shape in the x and y direction as the input image.

Recovering missing areas with the models

When trained, a model is applied normally, as described in section 4.3.4, on the desired images to recover the missing areas. As described above, the input shape is not defined, hence all image shapes and dimensions can be used. However, the image must be recovered in pieces due to local memory restrictions on the GPU. Since the models use surrounding pixels to recover the missing ones, it is natural that the performance is worse on the edges with fewer surrounding pixels. This is dealt with by utilizing stride, or overlap when dividing into smaller cubes and using the output of the previous predictions as input to the next one.

5 Results and Discussion

The results are divided into parts, covering the reconstruction of compressive sensed images, cloud detection, and the recovery of missing areas. The process has been very exploratory, hence many methods have been tested without further development.

5.1 Reconstructing the spectral dimension of the compressive sensed HSIs

5.1.1 The training process

Fig. 5.1 shows how the master model improves with more training. The training loss and the validation loss are both improving, indicating the neural network's ability to generalize and predict unseen data. The loss decreases quickly in the beginning and slower after more epochs. Considering that the MSE is used as a loss function and all values are between 0 and 1, the squared error will quickly become very small.

Fig. 5.1 has been created using TensorBoard, a real-time tool that displays the loss per epoch. This tool is utilized during the training of all models to track performance and prevent overfitting.

All the models trained for the reconstruction of compressive sensed data are trained on the same train data set and with approximately 35 epochs. Smaller models converge faster and require less training compared to larger models, as they have fewer parameters to tune.

5.1.2 Evaluation of reconstruction quality

Table 5.1 presents the results given by the evaluation functions for the various models, together with the TensorFlow implementation presenter in [44]. The evaluation scores are calculated using the same test images for all the models. Visual investigation of the results is conducted in the subsequent subsections.

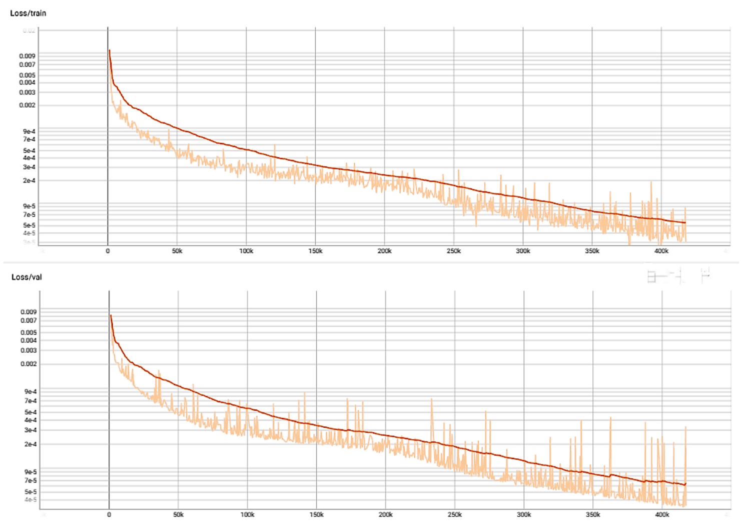


Fig. 5.1: MSE loss of the training and validation data given by the number of training cubes iterated through. This run consists of 19 epochs with a 90%/10% training/validation split. The light orange line gives the plot of the average of every 200 training cubes, while the darker orange line is a smoothed version. Note that the y-axis is logarithmic.

Preprocessing	Mean PSNR	Mean SSIM	Mean SAM	SID
TensorFlow model [44]	45.46 dB	0.9928	0.9997	N/A
Master model	48.19 dB	0.9946	0.9999	$3.02 \cdot 10^{-4}$
Simple model	43.22 dB	0.9858	0.9996	$6.84 \cdot 10^{-4}$
Backfire model	49.02 dB	0.9944	0.9999	$1.61 \cdot 10^{-4}$

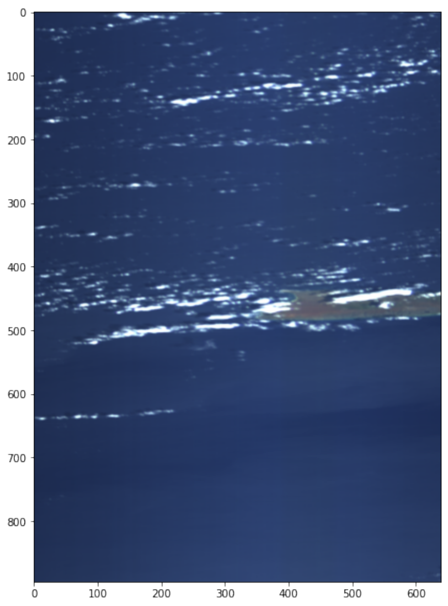
Tab. 5.1: HYPSO-1 image reconstruction quality, using different evaluation methods.

Master model

As seen in table 5.1, the reconstructions with this model have achieved higher average scores compared to the TensorFlow model. With an average PSNR score of 48.19 dB and a minimum of 40.40 dB, it seems that the NN has found a correlation between input and output that is general for all the HSIs in the data set.

Fig. 5.2 show the RGB representations of ground truth and the reconstruction of a test image. The image contains both ocean, land, and sky. It is impossible to see the difference between the ground truth and the reconstruction just by comparing the two.

Fig. 5.3 showcases two ground truth spectral arrays, their respective reconstructions, and the sampling points for compressive sensing, from the images in Fig. 5.2.


(a) Ground truth.

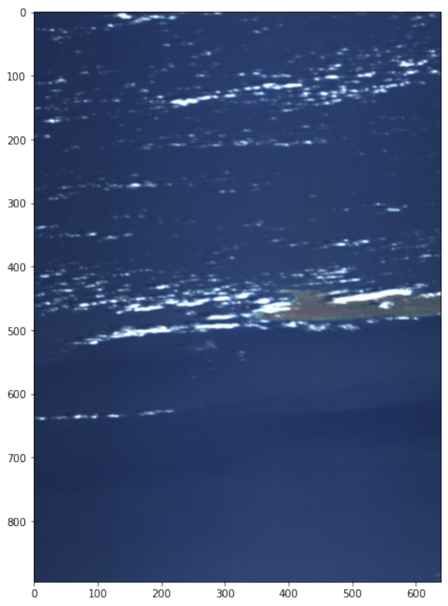

(b) Reconstruction.

Fig. 5.2: Ground truth and reconstruction for a test image. The reconstruction has a PSNR of 49.40 dB, SSIM of 0.9971, SAM of 0.9999, and a SID of $2.26{\cdot}10^{-4}$.

The reconstruction demonstrates the model's ability to capture spectral variations within an array. The pink ground truth 1 plot is hardly visible beneath the green reconstruction plot. Furthermore, even in the regions of the spectral array where there are no sampling points, the reconstruction successfully captures the spectral signature. However, it is worth noting that the reconstruction of ground truth 2 between 600 to 750 nm exhibits inaccuracies, failing to match the provided sampling points. This observation is intriguing, given the expectation that the model would learn, during the training process, that these provided points are always correct.

To further inspect the reconstruction quality, the pixel-wise error is used. Fig. 5.4 shows pixel-wise error given different distance functions. Each distance function offers unique insights into specific aspects of the error, allowing us to comprehensively understand the reconstruction quality. Higher MSE and SID indicate a greater level of error in the reconstruction, whereas a higher value of SAM indicates a higher degree of similarity between the reconstructed and original spectra. Analysis of the errors using the MSE and SAM reveals that the largest reconstruction errors occur at sharp transitions, which predominantly correspond to land and cloud regions. These areas, being outliers compared to the majority of the ocean scene, present greater challenges for accurate reconstruction. The SID highlights the edges, especially the corners of the patched cubes, indicating lower reconstruction quality in these

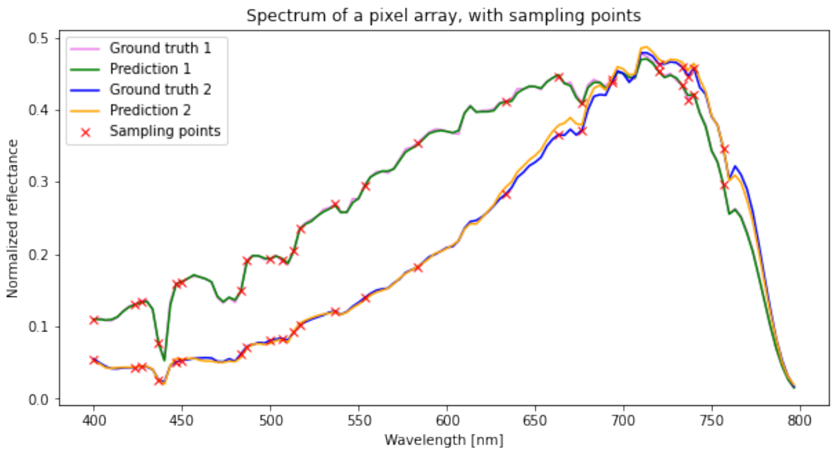
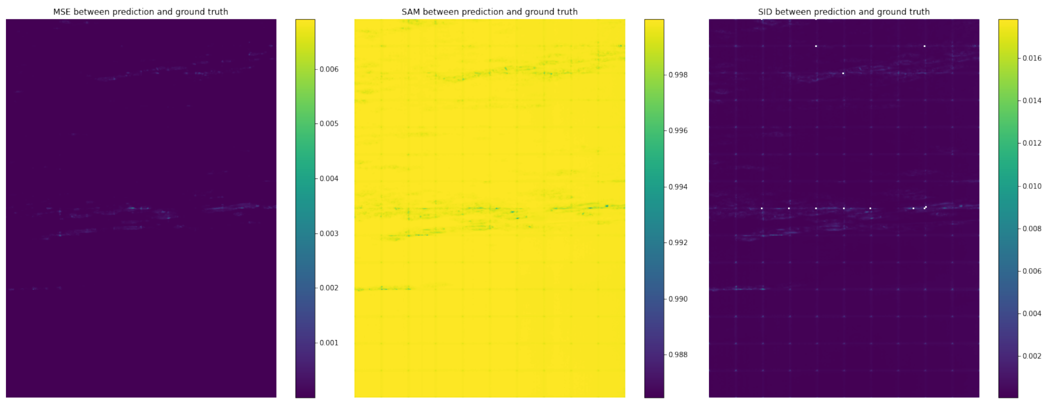


Fig. 5.3: Spectral arrays of the ground truth and the reconstruction of the image represented in Fig. 5.2. The pink (Ground truth 1) and green (Prediction 1) lines show the ground truth and the reconstruction for an ocean pixel, and the blue (Ground truth 2) and orange (Prediction 2) for a land pixel.

regions. This observation aligns with the fact that edges have fewer neighboring pixels available for analysis in the reconstruction process.



(a) MSE. **(b)** SAM. **(c)** SID.

Fig. 5.4: Pixel vise errors for the ground truth and reconstruction, using the master model, for the image shown in Fig. 5.2.

Simplification of the NN

By employing a smaller neural network, both the training and prediction phases are accelerated. The training process was terminated after 30 epochs due to the absence of any noticeable improvements in the loss over numerous iterations. The training duration was 239 minutes and 40.8 seconds. Using the simple model on the same image as in Fig. 5.2, it gets a PSNR of 45.37 dB, an SSIM of 0.9914, a SAM of 0.9997, and a SID of $6.73{\cdot}10^{-4}$. This is worse than the master model, but still in line with state-of-the-art methods for reconstruction.

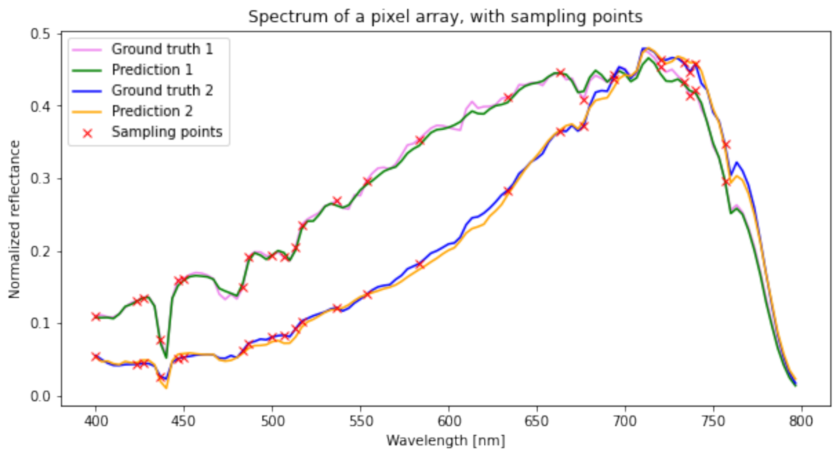


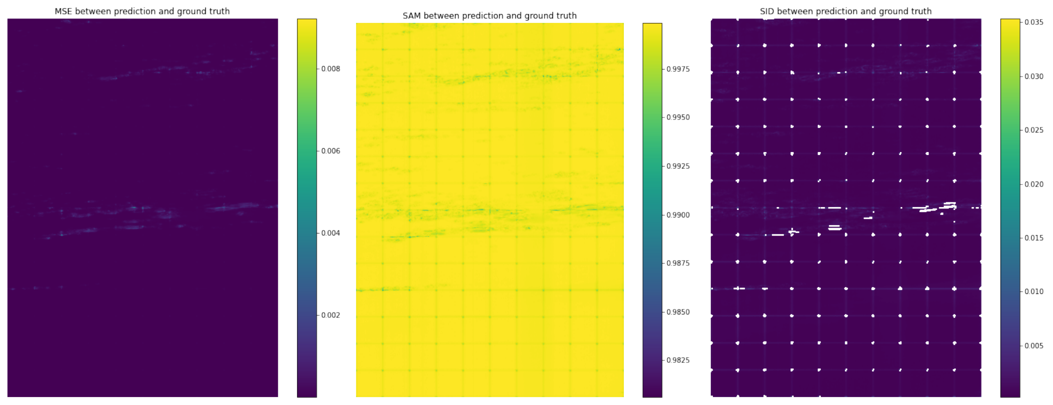
Fig. 5.5: Spectral arrays of the image represented in Fig. 5.2. The pink and green lines show the ground truth and the reconstruction for an ocean pixel, and the blue and orange for a land pixel.

Fig. 5.5 shows spectral arrays of the same pixels as in 5.3, including both ground truth and reconstruction. It is apparent that this model deviates more from the ground truth compared to the master model. The predictions tend to take shortcuts instead of accurately following the variations present in the ground truth, resulting in a smoothed version of the ground truth or a regression-like prediction.

The different distance functions used in Fig. 5.6 capture different aspects of the error. It is clear that the error, in general, has increased compared to the error given by the master model. Especially in the corners of the re-stitched cubes, the SID has increased. This suggests that the SID is sensitive to the reconstruction quality at the corners, indicating that the reconstruction performance may be relatively weaker in those areas. Also, the SID is high in the center of some of the clouds.

Backfire model

The training process of this mo was stopped after 32 epochs, and each epoch took approximately 28.5 minutes. As noted in table 5.1, the objective reconstruction quality measurements using this model are very similar to those of the master model. The PSNR and the SID have slightly improved, while the SSIM has slightly decreased. Using the same image as in Fig. 5.2, this model yields a PSNR of 50.85 dB, an SSIM of 0.9976, a SAM of 0.9999, a SID of $1.36 \cdot 10^{-4}$.



(a) MSE. **(b)** SAM. **(c)** SID.

Fig. 5.6: Pixel vise errors for the ground truth and reconstruction using the simple model.

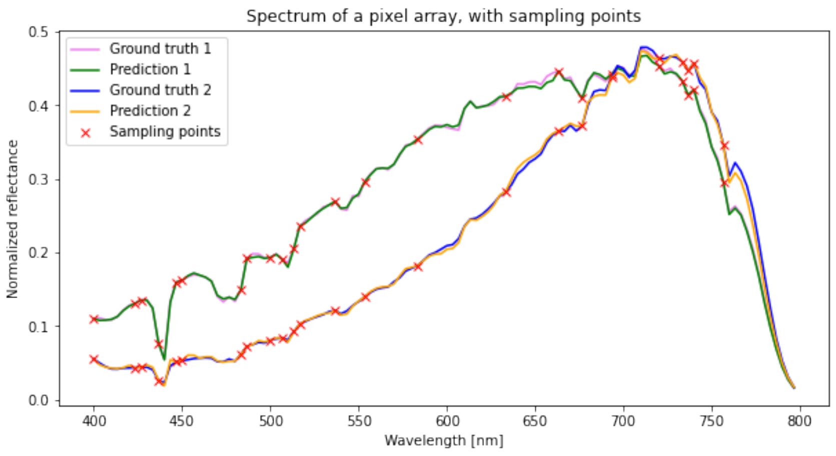


Fig. 5.7: Spectral domain for ground truth and predictions of the image represented in Fig. 5.2, using the backfire model. The pink and green lines show the ground truth and the reconstruction for an ocean pixel, and the blue and orange for a land pixel.

As observed in Fig. 5.7, the backfire model consistently predicts the correct values for the compressive sampling points, unlike the master model. However, there are still errors present between the sampling points. At approximately 660nm for both the ground truth and prediction 1, the trajectory of the reconstruction through the sampling point exhibits a slightly forced and unnatural behavior. Fig. 5.8 displays two additional spectral arrays from other test images, which demonstrate the same errors and unnatural behavior in their respective reconstructions.

Fig. 5.9 investigate further the spatial structure of the errors by presenting pixel-wise errors using different distance measures. Once again, it is when the image content has rapid changes, like the transition from ocean to land, or the thick small clouds, that the error is the biggest. However, it is worth noting that the errors in the

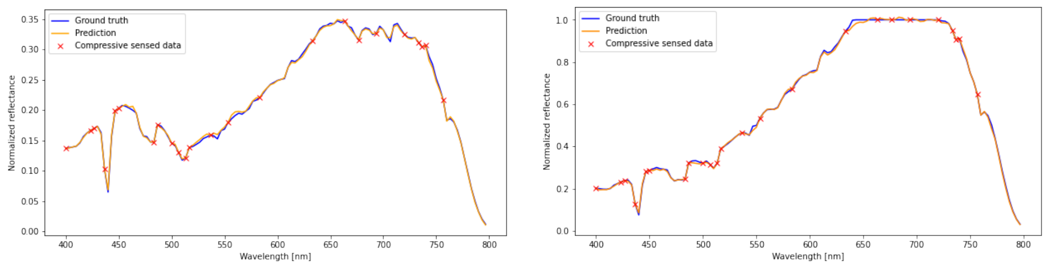
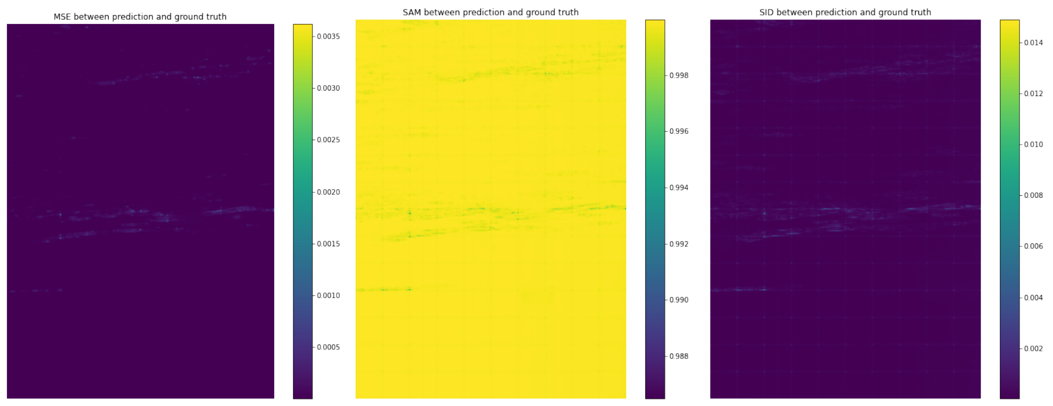


Fig. 5.8: Two spectral arrays together with the sampling points and the reconstructions.

backfire model are visibly smaller compared to the simple model, and the borders resulting from the concatenation of smaller cubes are barely noticeable.



(a) MSE. **(b)** SAM. **(c)** SID.

Fig. 5.9: Pixel vise errors for the ground truth and reconstruction using the backfire model.

Despite these imperfections, the backfire model performs well on unseen data. This is evidenced by the lowest reconstruction PSNR on the test data, which is 40.80 dB, indicating its adaptability to unfamiliar settings. The fact that the predictions are

5.1.3 Discussion of the spectral reconstruction results

As presented in table 5.1, the PyTorch implementations generally outperform the TensorFlow-based implementation in [43], [44]. The reason for this improvement is unclear, especially for the master model, considering that the implementations should be replicas. One notable difference is the Upscale block, which has non-trainable parameters in the TensorFlow implementation. This alone may account for the 3 dB increase in PSNR. However, other factors, such as the random initialization of neural network parameters, gradient calculation, and hardware-specific implementation, may also contribute. Despite the expected variability in results due to different parameter starting values, the PyTorch implementation consistently demonstrates better performance throughout the project compared to the TensorFlow-based implementation.

Machine learning has the potential to capture correlations when the architecture and size are appropriately chosen. However, in the case of the master model, it is observed that the prediction does not always match the input. This discrepancy cannot be solely attributed to the model's size. Larger models were trained and tested during the project, but their performance did not improve significantly. Moreover, training such large models is time-consuming as they converge slowly and require a greater number of epochs, each taking longer to complete. For instance, even after 40 epochs, the larger models were far from converging and performed worse than the master model.

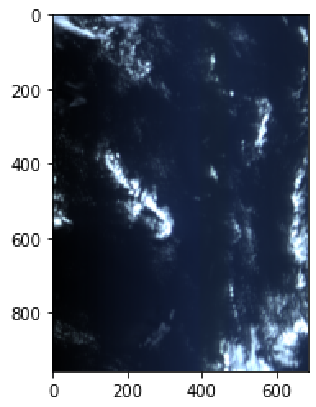
Out of the three presented models, the backfire model that resets the values is, arguably, the best-performing one. This model will therefore be used in the final pipeline, and the method for re-setting values will be adapted to other methods.

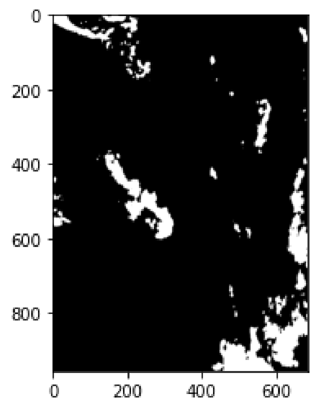
5.2 Part 2: Cloud detection and segmentation

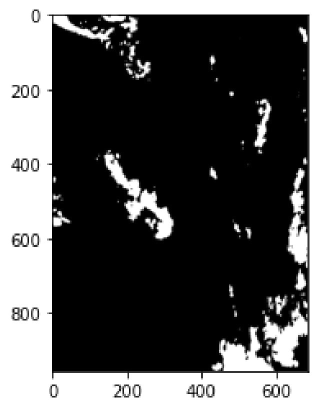
Evaluating the performance of cloud detection algorithms is done by visually inspecting the generated cloud masks in relation to the original images. It is important to detect entire clouds, including their thinner borders where the clouds are fading, as the presence of cloud borders can introduce unwanted distortions in the reconstructions. Therefore, the cloud mask should accurately identify and capture the complete cloud coverage, enabling reliable analysis and further processing.

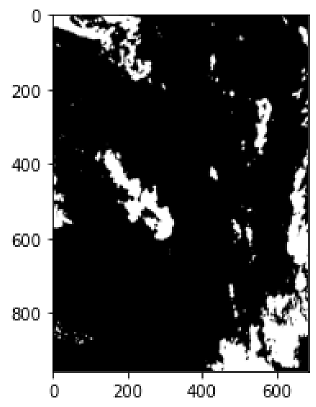
5.2.1 Clustering

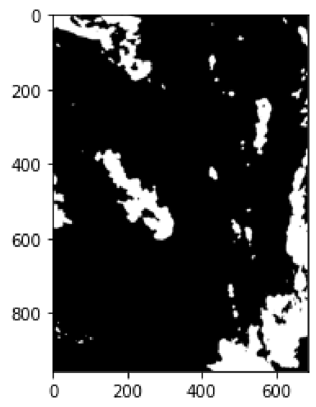
The cloud masks shown in Fig. 5.10 are generated using K-means clustering with 2 clusters, using different distance functions. The figure shows that the different distance functions have variable sensitivity when it comes to the border of the clouds. The MAE and the MSE are the least sensitive, only capturing the pixels with the thickest clouds present. Using SID gives the highest number of pixels in the cloud mask, with a total of more than 13% of the pixels in the mask, followed by SAM with 10.06%.


(a) Ground truth.


(b) MAE.


(c) MSE


(d) SAM.


(e) SID.

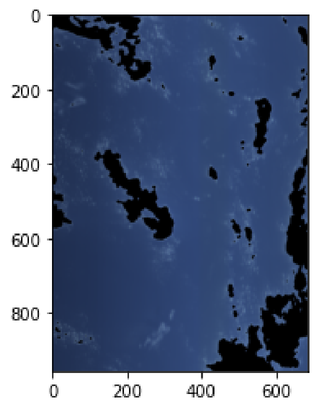

(f) Removed clouds using SID .

Fig. 5.10: Cloud masks generated using k-means clustering, with two clusters, using different distance functions. The original ground truth and ground truth with removed clouds using the cloud mask given SID are also compared.

Fig. 5.10f show how the cloud mask, generated using SID, overlaps with the ground truth. It is clear that the clustering is able to distinguish the clouds from the ocean. However, this is not a very challenging task when there are only oceans and clouds. Fig. 5.11 use clustering when, in addition to clouds and ocean, land is present. Both two and three clusters are tested.

It is clear from Fig. 5.11 that the clustering algorithm is not able to correctly separate the land from the clouds in all cases. Hence this approach might not be the most reliable when it comes to cloud segmentation. In addition, using this approach has no way of knowing which of the clusters represents clouds and whether clouds are present. It just sorts the pixel arrays into the given number of clusters based on similarity.

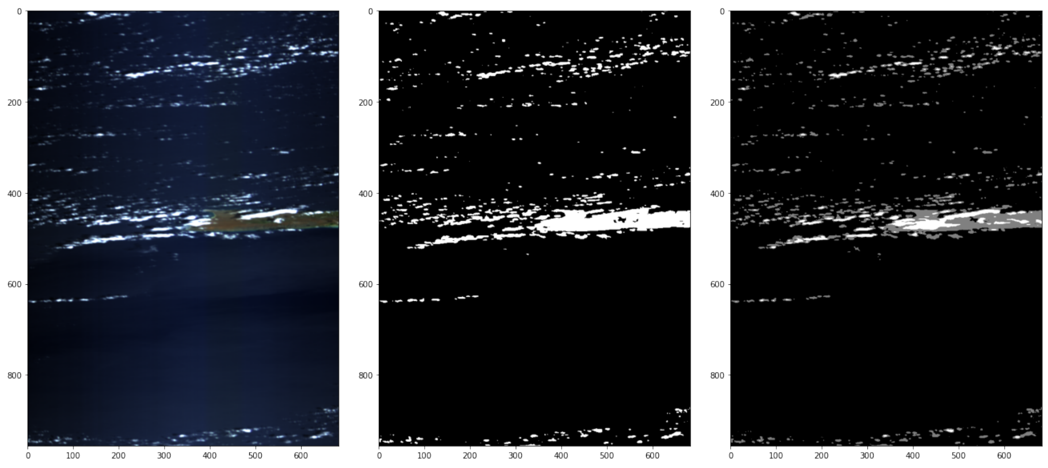

(a) Ground truth. (b) Two clusters. (c) Three clusters.

Fig. 5.11: Cloud masks generated using k-means with SAM as distance function, and using two and three clusters.

5.2.2 Threshold and correlation-based algorithm

The algorithm explained in section 4.5.1 yields, in general, more reliable cloud masks than the clustering. Fig. 5.12 shows the algorithm's performance on the same image used for clustering in Fig. 5.11. The `threshold` and the `queue_threshold` are set to 0.95, and the `neighbour_threshold` is set to 0.7. From the figure, it is clear that this algorithm does not mix up land and cloud pixels in this image, in contrast to the clustering. However, not all thin clouds are detected. Additionally, in Fig. 5.12c, it can be observed that the borders surrounding the eliminated clouds appear white, indicating the presence of residual thin clouds in those areas.

Fig. 5.13 shows three other images and the corresponding detected cloud masks.

The algorithm misses some of the cloud-contaminated pixels in all the images in Fig. 5.13. Image a is the same image used with clustering in Fig. 5.10. In this case, only 7.0% of the pixels are added to the cloud mask, in contrast to the 13.0% using clustering with the SID as the distance function. However, the number of detected cloud pixels is dependent on the given function parameters. Fig. 5.14 shows how changing the parameter `queue_threshold` influences the detected cloud mask.

When the threshold is set to 0.94, marginally more pixels are added to the mask compared to when the threshold is 0.95. At the same time, the run time increased from 6m and 48.5s to 141m and 16.9s. When the threshold is set to 0.93, suddenly all pixels are added to the mask. Since the function adds pixels to the mask based on the correlation between the pixel arrays already in the mask, it is understandable

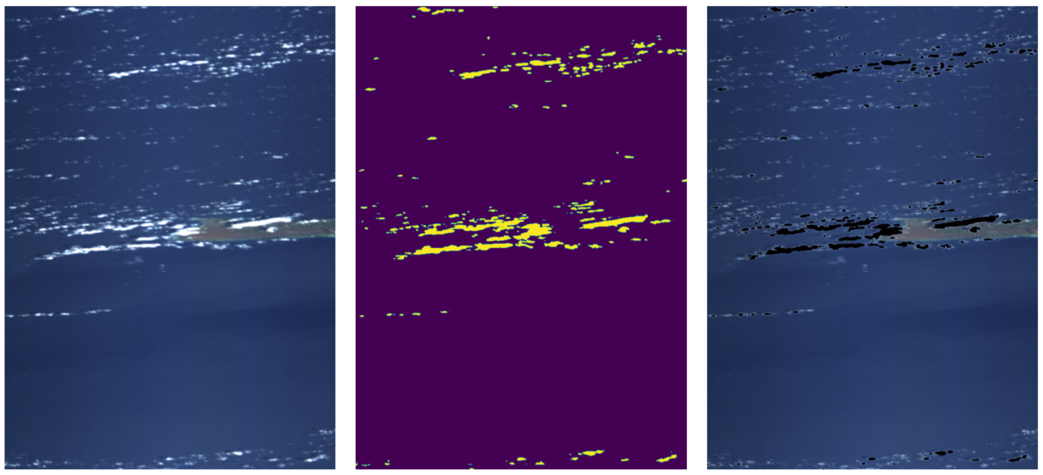

(a) Ground truth. (b) The detected cloud mask. (c) Removed clouds.

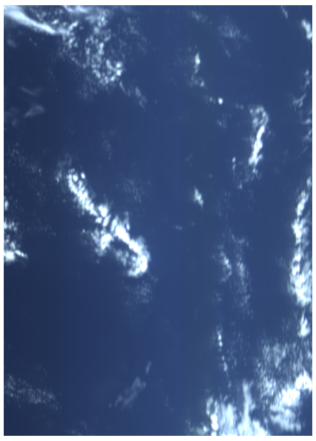
Fig. 5.12: Ground truth, cloud mask, and ground truth with removed clouds using the proposed cloud detection algorithm.

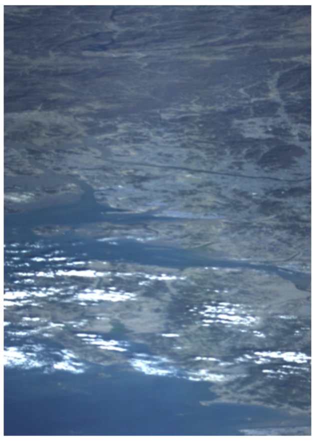
that once a non-cloud pixel is added it will continue adding non-cloud pixels. Hence, this parameter has to be tuned to fit a specific image, and the algorithm is not suited for unsupervised detection and segmentation of clouds.

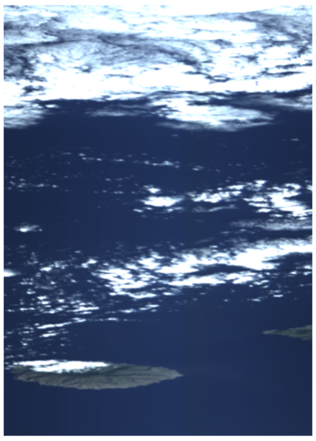
5.2.3 Discussion of the cloud detection and segmentation methods

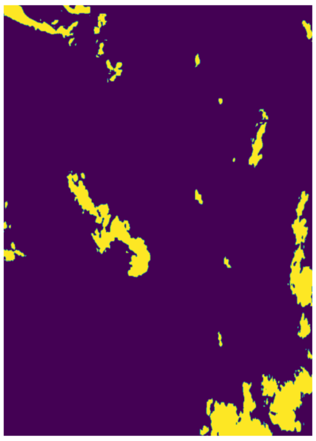
As discussed in section 5.2.1, clustering is not a suitable approach for unsupervised cloud detection/segmentation, especially in images with diverse content. Nevertheless, employing various distance functions with clustering provides valuable insights into how these distance functions function with the spectral arrays. This knowledge can be applied in other applications beyond cloud detection and segmentation.

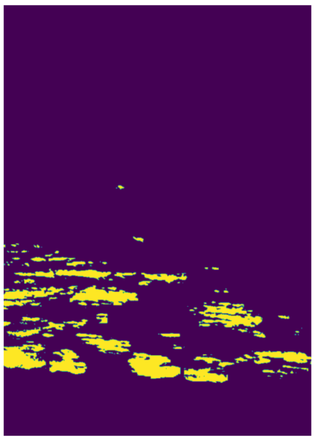
The proposed algorithm also has some disadvantages that should be considered. Its execution speed is relatively slow, and the more clouds present, the slower execution. The three cloud masks detected in Fig. 5.13 took 106 min, 86 min, and 102 min, respectively. As seen in Fig. 5.14, some of the algorithm's parameters may require adjustments to achieve optimal results for each individual image. To prevent the loop for adding cloud pixels from becoming infinite, a pixel is excluded from further checks once it has been added to the cloud mask. However, due to the difference in threshold values for adding pixels to the cloud mask and to the queue, certain clear cloud pixels that should have been added to the queue may be overlooked. This situation can occur if they are initially compared to a cloud pixel that is not sufficiently similar, leading to their addition to the cloud mask without being added

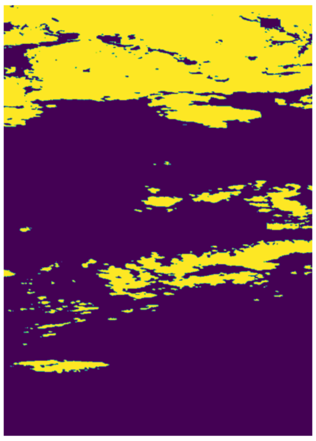

(a) Ground truth, Image a.

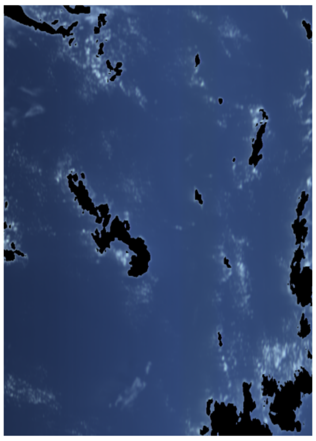

(b) Ground truth, Image b.

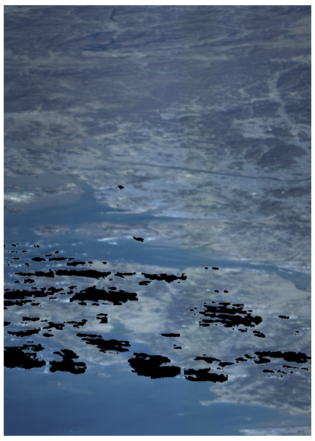

(c) Ground truth, Image c.


(d) Cloud mask, image a.


(e) Cloud mask, image b.


(f) Cloud mask, image c.


(g) Removed cloud, image a. 7.0% of the pixels removed.


(h) Removed cloud, image b. 6.1% of the pixels removed.

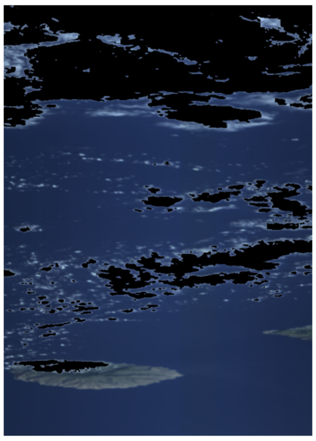
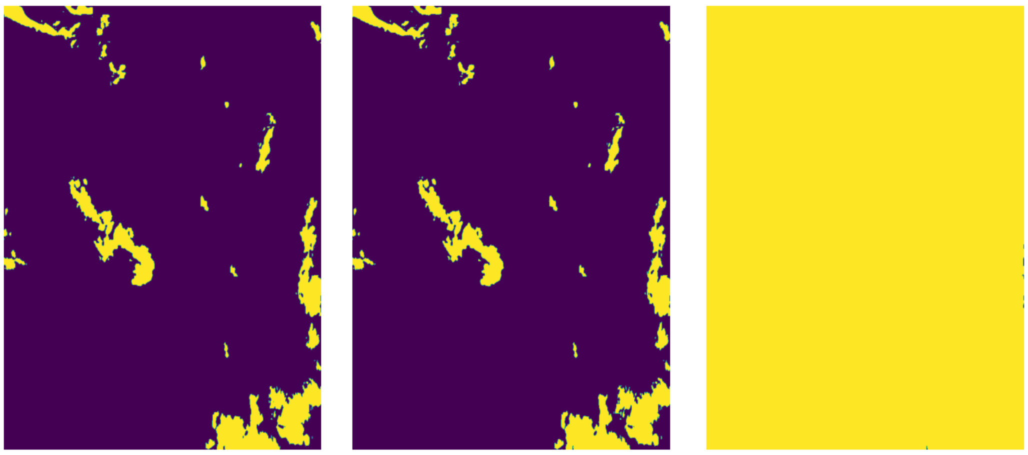

(i) Removed cloud, image c. 27.7% of the pixels removed.

Fig. 5.13: Cloud masks generated using the proposed algorithm on three different images.

to the queue. Once a pixel is added to the cloud mask, it will not be compared to other cloud pixels again, which helps improve the runtime. However, by not adding



(a) queue_threshold = 0.95. (b) queue_threshold = 0.94. (c) queue_threshold = 0.93.

Fig. 5.14: The detected cloud mask when queue_threshold is set to 0.95, 0.94, and 0.93 respectively. The three figures illustrate how the parameter influences the detected number of pixels added to the cloud mask.

these pixels to the queue, both isolated elements and significant portions can be missed, particularly as the mask expands spatially. If the algorithm mistakenly adds a non-cloud pixel to the queue, it will continue to add similar pixels, magnifying the error and leading to inaccurate results.

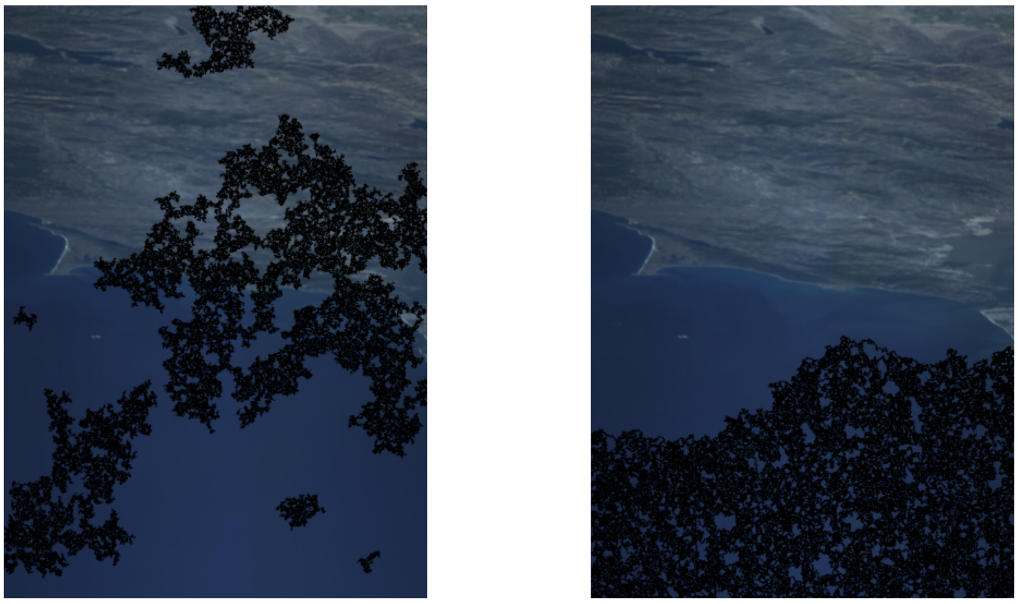
To summarize, none of the cloud detection methods presented rival the state-of-the-art algorithms, as they can be considered relatively simplistic. However, the methods given the right circumstances and parameter tuning can create cloud masks that include most of the cloud pixels. The methods were primarily developed to have something functional for the HYSPO-1 images to be utilized in the other parts. The algorithm does serve this purpose, with the given cloud masks. Due to time constraints, only a limited amount of time was assigned to this particular part, allowing numerous potential future improvements and refinements.

5.3 Recovery of missing areas

The results presented below look at the models' performances on randomly generated cloud masks using the algorithm described in section 4.7.3, and on detected cloud masks using the algorithm described in section 4.5.1. The detected cloud masks are tested on other images where no clouds are present in addition to the images they belong to. The spectral domain is also recovered using the backfire model to generate RGB representations of the images for further visual evaluation of the results.

5.3.1 The simulated clouds

The algorithm described in section 4.7.3, is used to simulate clouds. The parameter `prob_of_clouds` impact the structure of the simulated clouds. Fig. 5.15 show two simulated cloud masks are presented - one with the default parameter value of 0.249 and the other with an adjusted value of 0.3. The contrast between the two masks is apparent in terms of cloud structure, shape, and density. It is observed that a higher probability of clouds results in more compact clouds.



(a) `prob_of_clouds` = 0.249.

(b) `prob_of_clouds` = 0.3.

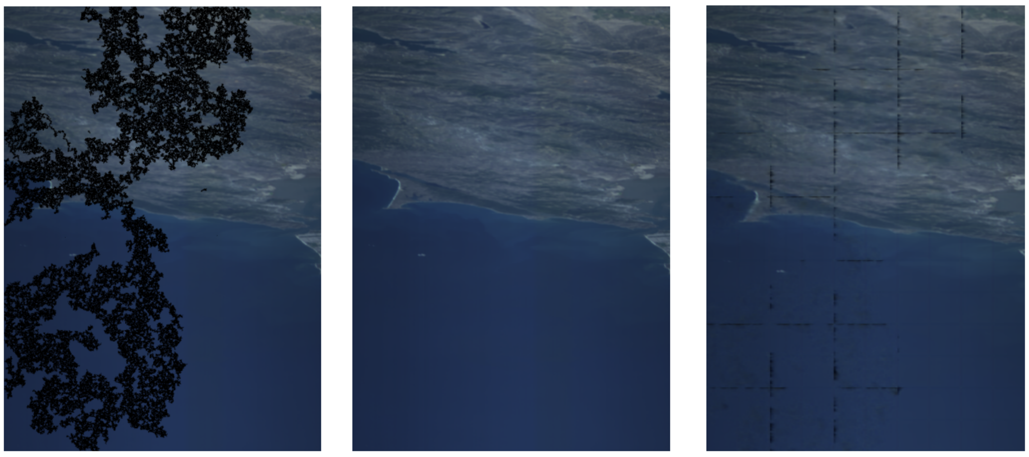
Fig. 5.15: Two simulated cloud masks using different values for the parameter `prob_of_clouds`.

In Fig. 5.15b, the cloud mask is concentrated at the bottom of the image. The algorithm uses a LIFO queue and adds seed pixels to the queue from the top to the bottom. Higher probabilities of adding a new pixel, e.g., higher value for `prob_of_clouds`, cause the algorithm to get stuck at the last added seed and expand the mask from there. As seen in 5.15a, lower probabilities result in more spread-out clouds. Both cloud masks have some “holes” or areas within the cloud masks that are not a part of the masks. The cloud mask generated using a lower probability of adding new pixels has more holes.

This characteristic differs from the detected cloud masks in section 5.2.2. In addition, the simulated clouds have a distinct shape compared to real clouds which are more dense. Consequently, training models with these simulated cloud masks may limit their ability to generalize to real-world scenarios. However, despite these limitations, this is the available data to work with.

5.3.2 Utilizing stride in the missing area recovery

As mentioned in section 4.7.3, using stride for image recovery can effectively address data loss along the edges of a cube. Figure 5.16 illustrates the outcome of recovery without using stride. As expected, the model struggles to recover missing parts at the edges of the patches.



(a) Removed cloud. (b) Ground truth. (c) Recovered image.

Fig. 5.16: Recovered image, using recovery model 1. The input of the network is the image with the removed cloud.

To address the problem, reconstruction with stride can be utilized while incorporating previous predictions as input, as explained in section 4.7.3. Figure 5.17 showcases the contrast with model 1 using this reconstruction method, which is applied for all subsequent reconstructions.

5.3.3 Result on simulated clouds

When it comes to recovering missing areas given simulated cloud masks, the ground truth is available, so the evaluation functions are used as normal. SAM is excluded due to bizarre errors given division with zero. Table 5.2 shows the performance of

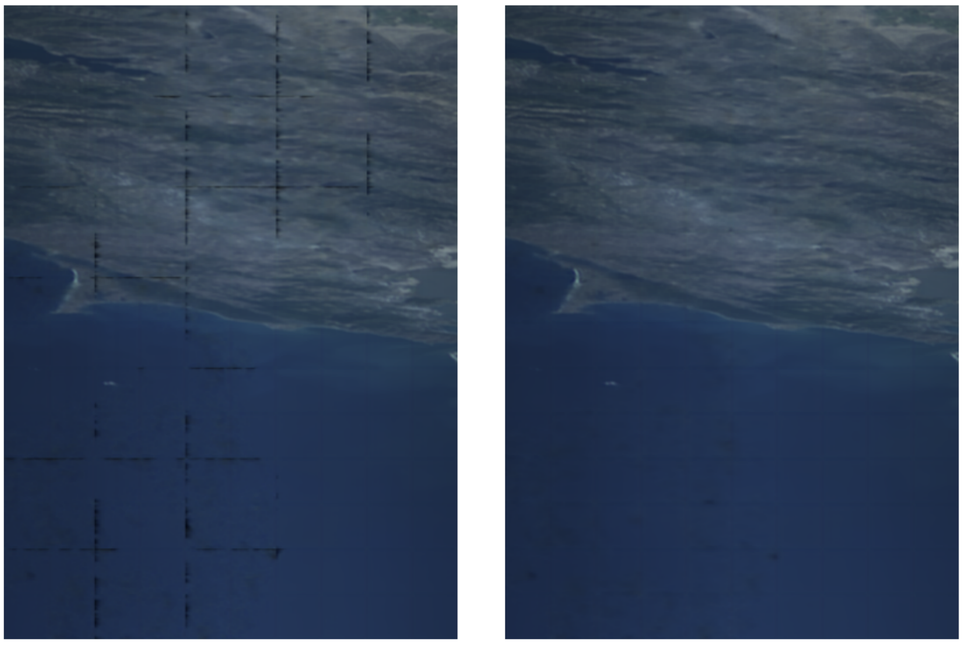

(a) Recovered image, no stride. (b) Recovered image, with stride.

Fig. 5.17: Recovered image by utilizing both no stride and stride with previous predictions as inputs in the reconstruction. The same image input and Model 1 are used.

the two models. The same set of images, all without clouds, and the same simulated clouds are used on each model. As a baseline for comparison, the scores have been calculated on the images with removed clouds without using any recovery models.

Tab. 5.2: Performance of the recovery models of simulated cloud masks, and the evaluation scores using no recovery model. The scores are calculated by taking the average of seven different images and cloud masks. The cloud masks are generated using prob_of_clouds in the range [0.249, 0.3].

Preprocessing	**Mean PSNR**	**Mean SSIM**	**SID**	**Mean recovery time**
No recovery	17.3 dB	0.7101	0.0	0.0s
Recovery Model 1	28.57dB	0.9535	0.000425	9.5s
Recovery Model 2	29.71 dB	0.9656	0.000209	12m 2s

As seen in the table, both models exhibit remarkably similar performance, with model 2 demonstrating slight improvements across all aspects except recovery time. The reconstructed images generally exhibit some loss of information during the reconstruction process, resulting in scores inferior to those achieved in section 5.1. However, this outcome was anticipated, as the task of recovering missing areas is more complex considering that the areas missing are different in each case. Despite this, both models demonstrate a significant increase in recovery scores compared to no recovery.

To conduct visual inspections, RGB representations of images are created by reconstructing the spectral domain through the backfire model. In Figure 5.18, the reconstructions made by model 1 and model 2 are compared using the same image and cloud mask. Although the evaluation scores are similar, it can be seen that model 1 has black marks in image 5.18c, indicating some content is missing. Model 2 has fewer of these areas, but there are some barely visible lines and impurities present. Both models, however, are able to recover most of the missing areas. Notice the thin cloud by the bay, where many pixels are lost in the randomly generated cloud mask. Despite this, both reconstructions preserve the cloud shape, with Model 2 doing so to a higher degree than Model 1.

Fig. 5.18 also shows reconstructions made with Model 1 and Model 2, using the same image and cloud mask for both, this time with a higher `prob_of_clouds`. The evaluation scores are similar also this time. It is clear that the denser cloud mask is making the recovery lack more data and more blurry. Given the marginally better performance of Recovery Model 2, this will be used in further testing. However, both model recovery manages to capture the shape of the coastal line. A part of the image is lost in the reconstruction, as described in section 4.7.3.

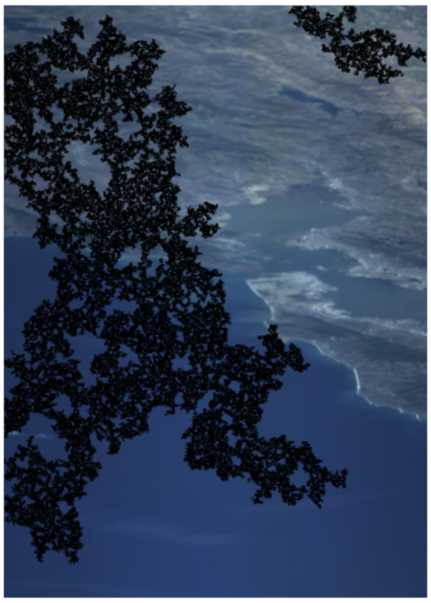
To closer investigate the details in the recoveries, smaller image cubes are investigated individually. Fig. 5.20 and 5.21 show the recovery of two random 64×64 cubes, with randomly generated cloud masks. The cloud masks are generated for the cubes, hence not using more than 25% of the cube pixels.

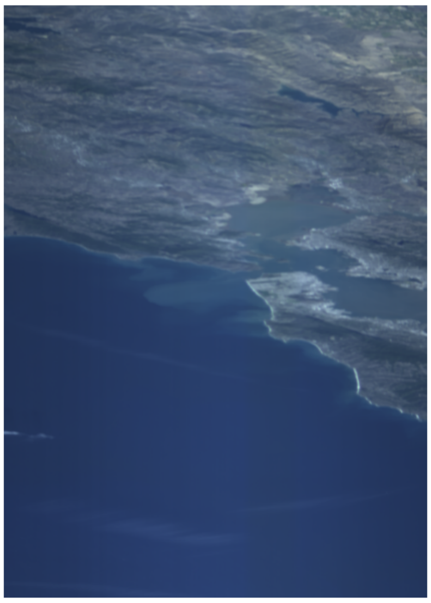
5.3.4 Results using real clouds

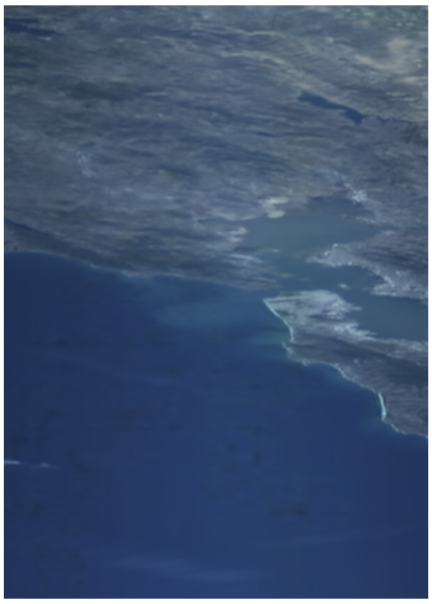
Fig. 5.22 displays the recovery of a cloud contaminated image, together with the original image and the removed cloud mask image. There are no without clouds to compare with, so only a visual inspection is possible.

From the figure, it is clear that this approach does not work. The cloud mask does not cover appropriately all cloud contaminated pixels, hence the surrounding pixels of the removed cloud mask still is contaminated. This leads to the recovery of thin clouds, instead of the intended recovery of the ground underneath.

In Fig. 5.23, the cloud mask from Fig. 5.22 is used on another image, without any clouds. In this case, the evaluation scores are available due to the access of the gt.


(a) Removed cloud mask.


(b) Ground truth.


(c) Model 1, with a PSNR of 38.43dB and an SSIM of 0.9952.

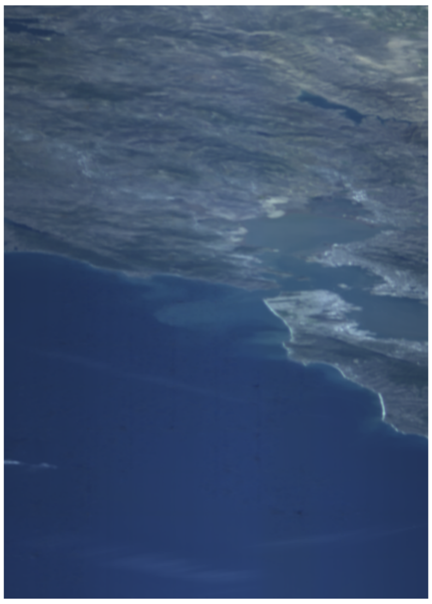
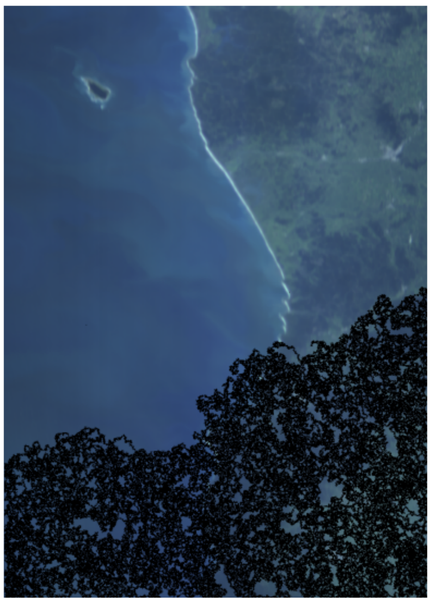
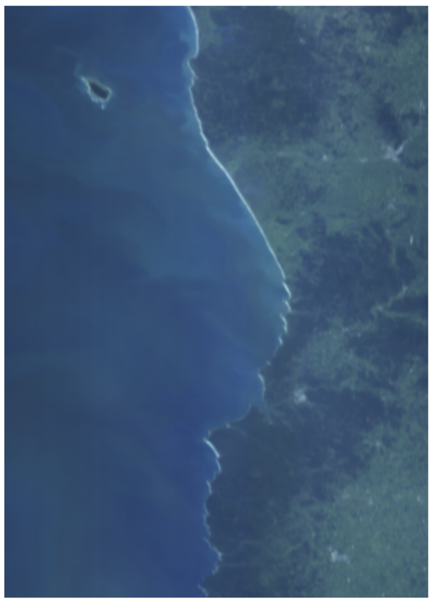
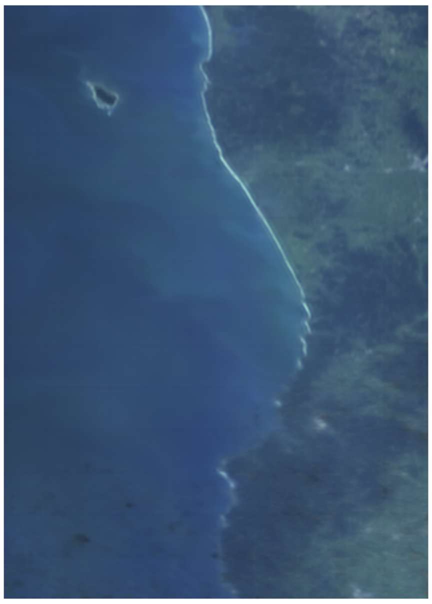

(d) Model 2, with a PSNR of 39.31dB and an SSIM of 0.9971.

Fig. 5.18: Recovered images using Model 1 and Model 2, together with the cloud mask and the ground truth.

The recovery has high evaluation scores, but less data is removed compared to the simulated clouds. The recovery is missing some parts parts centered in the biggest removed bulks of the cloud mask. The recovery process has also added some distortions around some of the pixels in the removed cloud mask, looking like small,


(a) Removed cloud mask.


(b) Ground truth.


(c) Model 1, with a PSNR of 22.95dB and an SSIM 0.9259.

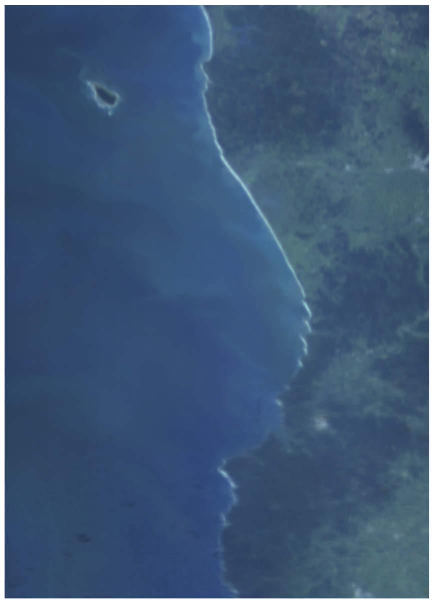
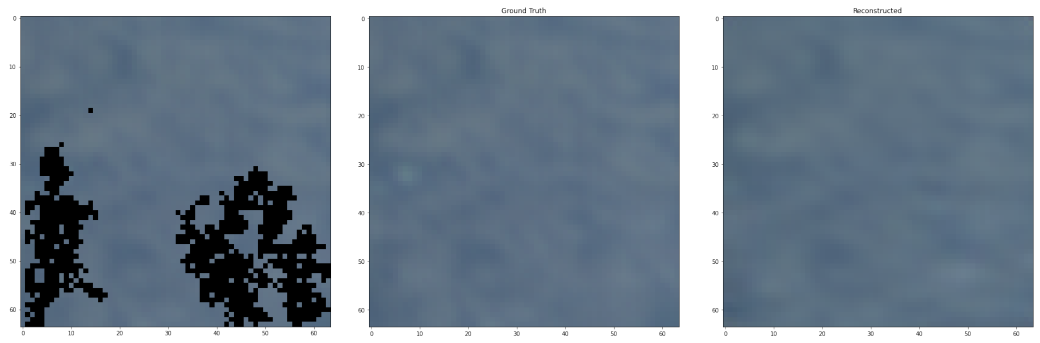

(d) Model 2, with a PSNR 22.92dB and an SSIM of 0.9237.

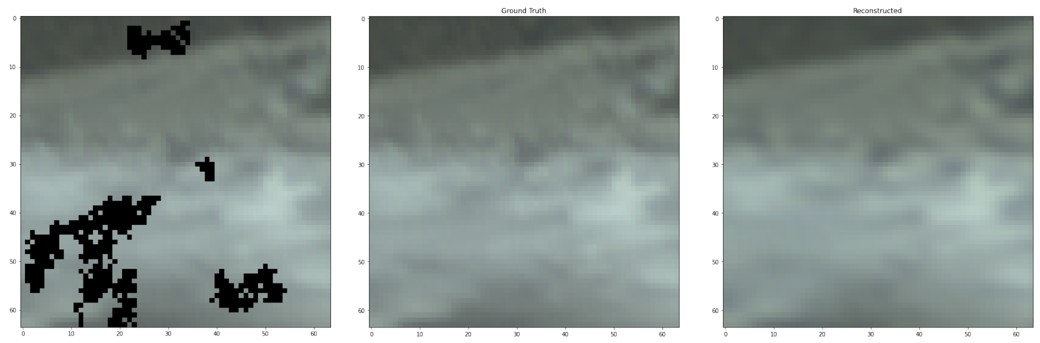
Fig. 5.19: Another recovered image using Model 1 and Model 2, together with the cloud mask and the ground truth.

thin clouds. However, otherwise the reconstruction appears to recover most of the missing parts.



(a) Removed cloud mask. **(b)** Ground truth. **(c)** Recovered image.

Fig. 5.20: The recovery of a 64×64 cube, using Recovery Model 2.



(a) Removed cloud mask. **(b)** Ground truth. **(c)** Recovered image.

Fig. 5.21: The recovery of another 64×64 cube, using Recovery Model 2.

5.3.5 Discussion of the results

The neural network is able to recover the missing areas with high accuracy when working with the simulated cloud masks. However, when working with real cloud masks, it is clear that it is not working as well. The generated clouds masks and the detected clouds masks have however quite different spatial structures. Whereas the real cloud masks have very compact blobs, where all the pixels within a mask are removed, the generated cloud masks have some pixels present within the clouds, and a more random outline of the masks. One can therefore argue that the simulated cloud masks do not comprehensively cover the structure and behavior of real clouds. Considering that the NN is trained on the generated cloud masks, it is understandable that the reconstruction on the real cloud masks is worse.

It is clear that both the recovery models are able to recover some missing data based on surrounding pixels. Both models perform best when using randomly simulated clouds with the same premises as the model is trained on. Changing the premises means a decrease in the models' performance. Using real cloud masks seems to be harder for the model to recover. The real cloud masks are denser, unlike the simulated ones that have some non-cloud pixels within the cloud masks. The bigger areas missing, the worse the reconstructions.

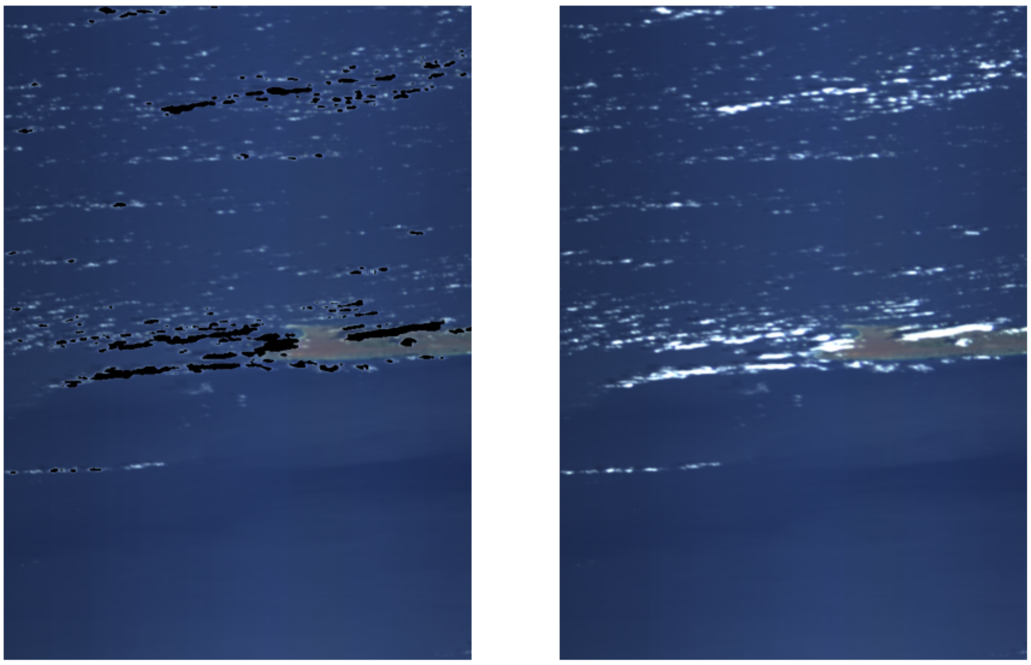
When comparing model 1 and model 2, certain differences become apparent. Model 1 exhibits a higher level of blurriness, resulting in the loss of finer details in the image, even in regions where no data is missing. On the other hand, model 2 presents a sort of edge in the distinct edge at the boundaries where the missing and known areas intersect. This edge creates a somewhat unnatural appearance in the image, clearly indicating the presence of missing data.

An important point to note is that the model was trained using only images without any clouds. This makes it difficult to determine if the reconstructions are simply based on memory or if the models are actually capable of filling in missing areas using the surrounding pixel arrays. It is also worth noting that the recoveries presented in this thesis were only made from compressed to compressed domain. This is because the model was specifically trained for this purpose. When tested in other situations, such as from uncompressed to uncompressed domain, the models yielded meaningless recoveries.

5.4 Discussion of pipeline parts

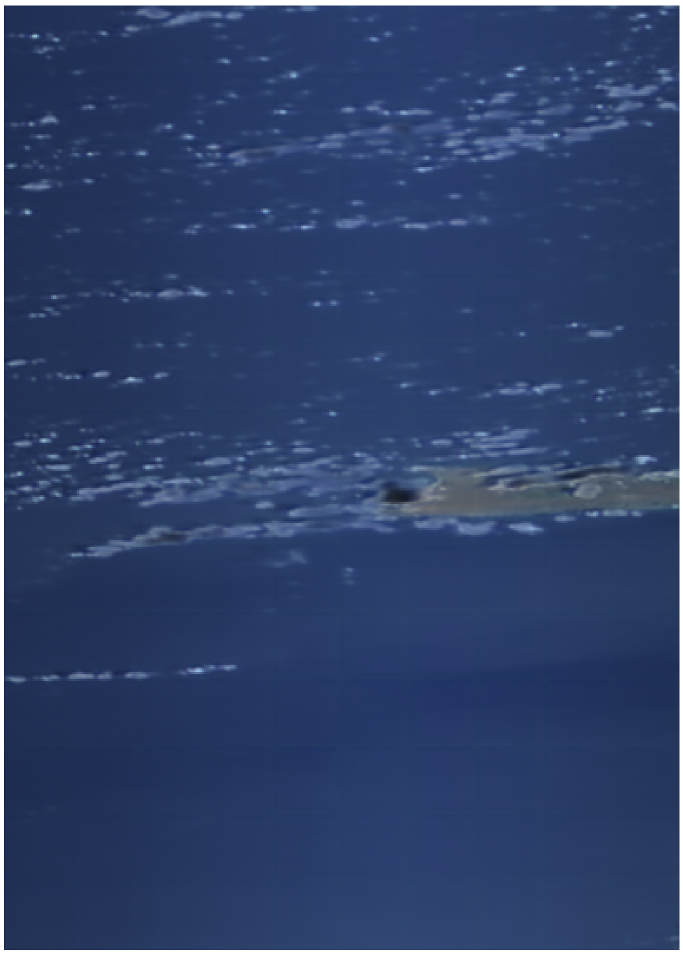
The previous sections present and discuss the results individually, given different pipeline parts. It has been proven that using a U-Net to reconstruct the spectral domain of compressive sensed HSI yields state-of-the-art results given an adequate model and training. This result follows the conclusions of the newest research on this topic. A U-Net has also been used to recover missing areas in the HSI based on surrounding pixels. In addition, two methods for detecting and segmenting clouds have been tested and evaluated. Now, the collaboration between the different parts and the feasibility of a pipeline will be discussed.

The culmination of the individual components into a pipeline for cloud detection and recovery presents both challenges and opportunities. The successful application of the U-Net architecture for both spectral domain reconstruction and cloud-free area recovery is promising. This demonstrates the adaptability of this architecture across different tasks within the same pipeline. However, integrating cloud detection methods with the recovery process is crucial. The choice of cloud detection approach significantly impacts the quality of the input data for the recovery model. A more accurate cloud detection process directly translates to better results in the subsequent recovery phase. Conversely, an erroneous cloud detection can propagate errors through the pipeline, leading to inaccurate or incomplete reconstruction. Likewise, refining the recovery model using more refined cloud detection and segmentation can improve the quality of the reconstructed image. It has been demonstrated in [5] that U-Nets can be utilized for cloud detection. Therefore, it is reasonable to assume that a well-designed U-Net may be capable of handling all three aspects of the pipeline at once. During the exploration, a UNet was attempted to combine recovery of missing areas and reconstruction of a compressive sensed domain. However, it was noted that the GPU couldn't handle the amount of storage required during training. So one consideration is whether this pipeline can be realistically implemented. The computational demands of deep learning models like U-Nets need to be thoughtfully evaluated to ensure they can be applied in real scenarios without overwhelming computational resources.



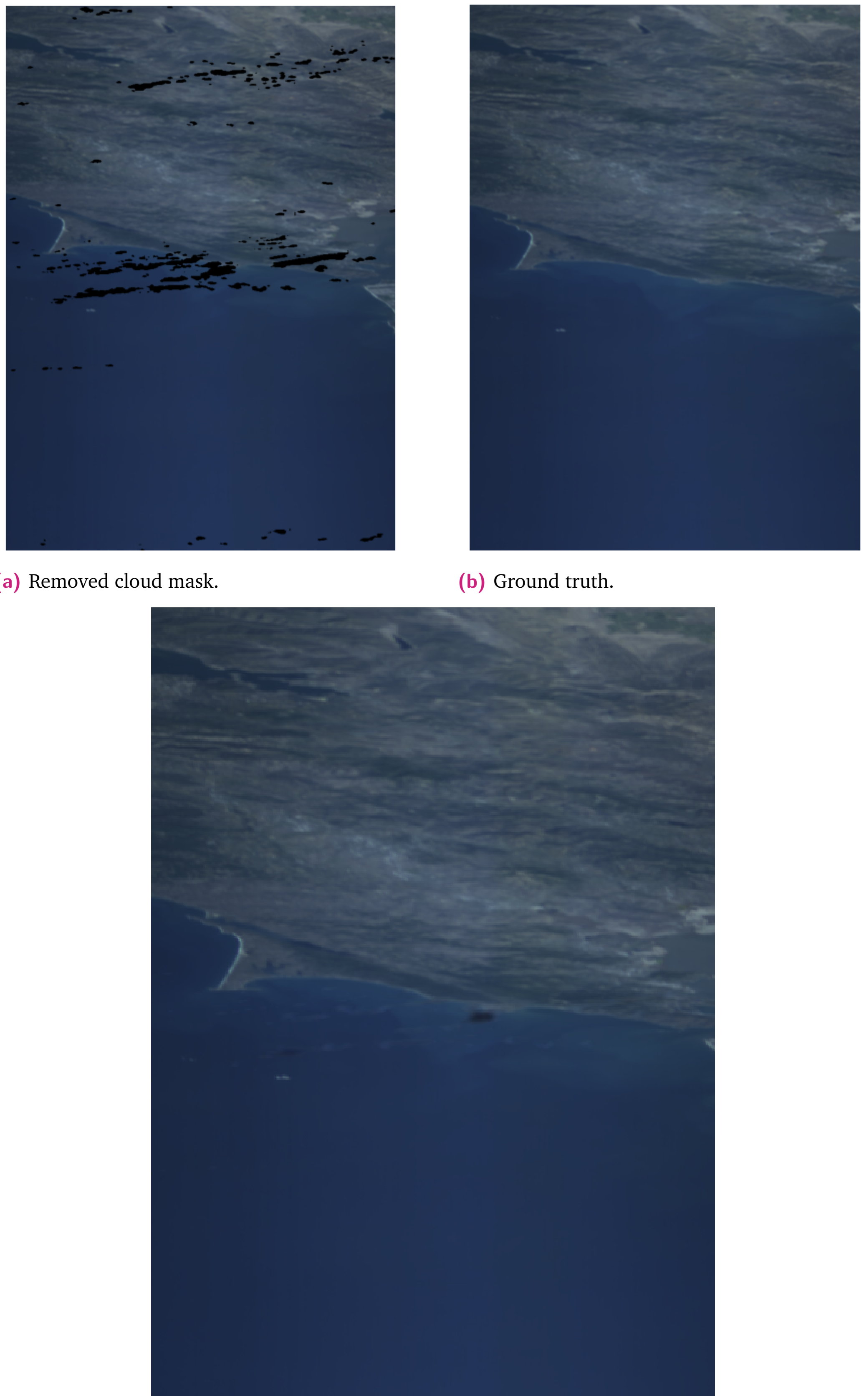
(a) Removed cloud mask.

(b) Ground truth.



(c) Recovery using model 2.

Fig. 5.22: Recovery of an image with real clouds removed. The cloud mask is detected using the new detection algorithm.


(a) Removed cloud mask.

(b) Ground truth.

(c) The Recovery using Model 2, with a PSNR of 42.53dB and an SSIM of 0.9978.

Fig. 5.23: Recovery when detected cloud mask from Fig. 5.22 is used with a HSI without any clouds.

6 Conclusions and future work

This study has presented and discussed the different components in a pipeline including cloud detection and removal, recovery of missing areas and reconstruction of a compressive sensed spectral domain of HSIs from the HYSPO-1 satellite. It has been demonstrated that employing a UNet for reconstructing the spectral domain of compressive sensed HSI yields state-of-the-art results with the appropriate model and training. Similarly, using a UNet to recover missing areas in the HSI based on surrounding pixels has also shown promising outcomes. Regarding cloud detection and segmentation, the methods used have significant limitations making them unsuitable for an unsupervised setting, such as the one in the suggested pipeline. Nonetheless, they have served their purpose in testing the recovery models on real clouds.

In regards to future work, it has been noted that the reconstruction of compressive sensed HSIs and the recovery of missing areas rely on U-Nets. Additionally, research suggests that U-Nets have potential for cloud detection. As a result, it may be worthwhile to investigate the combination of all three aspects into one model. This initiative would require high-quality data and powerful GPUs, but with the appropriate model, the results in this thesis indicate that it can be a feasible task.

Bibliography

[1] J. B. Campbell and R. H. Wynne, *Introduction to remote sensing*. Guilford Press, 2011 (cit. on p. 5).

[2] C.-I. Chang, *Hyperspectral imaging: techniques for spectral detection and classification*. Springer Science & Business Media, 2003, vol. 1 (cit. on p. 5).

[3] M. J. Costa and D. Bortoli, “Cloud detection and classification from multi-spectral satellite data,” *Proceedings of the SPIE*, vol. 7475, pp. 747 514–747 514, Sep. 2009 (cit. on p. 7).

[4] Z. Liu and B. R. Hunt, “A new approach to removing cloud cover from satellite imagery,” *Computer vision, graphics, and image processing*, vol. 25, no. 2, pp. 252–256, 1984 (cit. on p. 7).

[5] B. Grabowski, M. Ziaja, M. Kawulok, and J. Nalepa, “Towards robust cloud detection in satellite images using u-nets,” in *2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS*, 2021, pp. 4099–4102 (cit. on pp. 7, 34, 38, 82).

[6] L. Gomez-Chova, G. Camps-Valls, J. Amoros-Lopez, *et al.*, “New cloud detection algorithm for multispectral and hyperspectral images: Application to envisat/meris and proba/chris sensors,” in *2006 IEEE International Symposium on Geoscience and Remote Sensing*, 2006, pp. 2757–2760 (cit. on pp. 7, 34).

[7] C. Papin, P. Bouthemy, and G. Rochard, “Unsupervised segmentation of low clouds from infrared meteosat images based on a contextual spatio-temporal labeling approach,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 40, no. 1, pp. 104–114, 2002 (cit. on pp. 7, 34).

[8] Z. Li, H. Shen, Q. Cheng, *et al.*, “Deep learning based cloud detection for remote sensing images by the fusion of multi-scale convolutional features,” *arXiv preprint arXiv:1810.05801*, 2018 (cit. on p. 7).

[9] M. E. Grøtte, R. Birkeland, E. Honoré-Livermore, *et al.*, “Ocean color hyperspectral remote sensing with high resolution and low latency—the hypso-1 cubesat mission,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–19, 2021 (cit. on pp. 8, 38).

[10] R. Birkeland, S. Berg, M. Orlandic, and J. Garrett, “On-board characterization of hyperspectral image exposure and cloud coverage by compression ratio,” in *2022 12th Workshop on Hyperspectral Imaging and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, IEEE, 2022, pp. 1–5 (cit. on pp. 8, 41).

[11] J. M. Landsberg, “Tensors: Geometry and applications,” *Representation theory*, vol. 381, no. 402, p. 3, 2012 (cit. on p. 9).

[12]K.-H. Thung and P. Raveendran, “A survey of image quality measures,” in *2009 international conference for technical postgraduates (TECHPOS)*, IEEE, 2009, pp. 1–4 (cit. on p. 10).

[13]A. Hore and D. Ziou, “Image quality metrics: Psnr vs. ssim,” in *2010 20th international conference on pattern recognition*, IEEE, 2010, pp. 2366–2369 (cit. on pp. 11, 12).

[14]C. J. Willmott and K. Matsuura, “Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance,” *Climate research*, vol. 30, no. 1, pp. 79–82, 2005 (cit. on p. 11).

[15]O. A. De Carvalho and P. R. Meneses, “Spectral correlation mapper (scm): An improvement on the spectral angle mapper (sam),” in *Summaries of the 9th JPL Airborne Earth Science Workshop, JPL Publication 00-18*, JPL publication Pasadena, CA, USA, vol. 9, 2000 (cit. on p. 12).

[16]C.-I. Chang, “Spectral information divergence for hyperspectral image analysis,” in *IEEE 1999 International Geoscience and Remote Sensing Symposium. IGARSS’99 (Cat. No.99CH36293)*, vol. 1, 1999, 509–511 vol.1 (cit. on p. 13).

[17]M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning*. MIT press, 2018 (cit. on pp. 13, 20).

[18]E. H. Ruspini, “A new approach to clustering,” *Information and control*, vol. 15, no. 1, pp. 22–32, 1969 (cit. on p. 14).

[19]B. Pang, E. Nijkamp, and Y. N. Wu, “Deep learning with tensorflow: A review,” *Journal of Educational and Behavioral Statistics*, vol. 45, no. 2, pp. 227–248, 2020 (cit. on pp. 14–16, 18, 20, 21).

[20]M. W. Gardner and S. Dorling, “Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences,” *Atmospheric environment*, vol. 32, no. 14-15, pp. 2627–2636, 1998 (cit. on p. 14).

[21]E. W. Weisstein, “Convolution,” *https://mathworld. wolfram. com/*, 2003 (cit. on p. 16).

[22]N. Ibtehaz and M. S. Rahman, “Multiresunet: Rethinking the u-net architecture for multimodal biomedical image segmentation,” *Neural networks*, vol. 121, pp. 74–87, 2020 (cit. on p. 19).

[23]F. Günther and S. Fritsch, “Neuralnet: Training of neural networks.,” *R J.*, vol. 2, no. 1, p. 30, 2010 (cit. on p. 20).

[24]S. Ruder, “An overview of gradient descent optimization algorithms,” *arXiv preprint arXiv:1609.04747*, 2016 (cit. on pp. 21, 22).

[25]D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014 (cit. on p. 22).

[26]A. Paszke, S. Gross, F. Massa, *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, *et al.*, Eds., vol. 32, Curran Associates, Inc., 2019 (cit. on p. 23).

[27]E. Stevens, L. Antiga, and T. Viehmann, *Deep learning with PyTorch*. Manning Publications, 2020 (cit. on p. 23).

[28]E. Por, M. van Kooten, and V. Sarkovic, "Nyquist–shannon sampling theorem," *Leiden University*, vol. 1, p. 1, 2019 (cit. on p. 24).

[29]P. S. Heuberger, P. M. Van den Hof, and O. H. Bosgra, "A generalized orthonormal basis for linear dynamical systems," *IEEE Transactions on Automatic Control*, vol. 40, no. 3, pp. 451–465, 1995 (cit. on p. 25).

[30]S. Foucart and H. Rauhut, "An invitation to compressive sensing," in *A mathematical introduction to compressive sensing*, Springer, 2013, pp. 1–39 (cit. on pp. 26–28).

[31]H. Xiao, Z. Wang, and X. Cui, "Distributed compressed sensing of hyperspectral images according to spectral library matching," *IEEE Access*, vol. 9, pp. 112 994–113 006, 2021 (cit. on p. 31).

[32]M. Golbabaee, S. Arberet, and P. Vandergheynst, "Multichannel compressed sensing via source separation for hyperspectral images," in *2010 18th European Signal Processing Conference*, 2010, pp. 1326–1329 (cit. on p. 31).

[33]Y. Oiknine, I. August, V. Farber, D. Gedalin, and A. Stern, "Compressive sensing hyperspectral imaging by spectral multiplexing with liquid crystal," *Journal of imaging*, vol. 5, no. 1, p. 3, 2018 (cit. on p. 31).

[34]S. Yang, H. Jin, M. Wang, Y. Ren, and L. Jiao, "Data-driven compressive sampling and learning sparse coding for hyperspectral image classification," *IEEE Geoscience and Remote Sensing Letters*, vol. 11, no. 2, pp. 479–483, 2014 (cit. on p. 32).

[35]A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences*, vol. 2, no. 1, pp. 183–202, 2009 (cit. on p. 32).

[36]S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein, *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011 (cit. on p. 32).

[37]J. Wang, S. Kwon, and B. Shim, "Generalized orthogonal matching pursuit," *IEEE Transactions on signal processing*, vol. 60, no. 12, pp. 6202–6216, 2012 (cit. on p. 32).

[38]H.-R. Yang, H. Fang, C. Zhang, and S. Wei, "Iterative hard thresholding algorithm based on backtracking," *Acta Automatica Sinica*, vol. 37, no. 3, pp. 276–282, 2011 (cit. on p. 32).

[39]D. Needell and J. A. Tropp, "Cosamp: Iterative signal recovery from incomplete and inaccurate samples," *Applied and computational harmonic analysis*, vol. 26, no. 3, pp. 301–321, 2009 (cit. on p. 32).

[40]J. A. Justo, D. Lupu, M. Orlandić, I. Necoara, and T. A. Johansen, "A comparative study of compressive sensing algorithms for hyperspectral imaging reconstruction," in *2022 IEEE 14th Image, Video, and Multidimensional Signal Processing Workshop (IVMSP)*, IEEE, 2022, pp. 1–5 (cit. on p. 32).

[41]Y. Yang, J. Sun, H. Li, and Z. Xu, “Admm-csnet: A deep learning approach for image compressive sensing,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 3, pp. 521–538, 2020 (cit. on p. 32).

[42]Y. Xu, Z. Wu, J. Chanussot, and Z. Wei, “Joint reconstruction and anomaly detection from compressive hyperspectral images using mahalanobis distance-regularized tensor rpca,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 5, pp. 2919–2930, 2018 (cit. on p. 33).

[43]D. Gedalin, Y. Oiknine, and A. Stern, “Deepcubenet: Reconstruction of spectrally compressive sensed hyperspectral images with deep neural networks,” *Optics express*, vol. 27, no. 24, pp. 35 811–35 822, 2019 (cit. on pp. 33, 50, 67).

[44]H. Markeng, “Deep neural network-based reconstruction of compressive sensed hyperspectral images,” *Unplublished*, (cit. on pp. 33, 50, 61, 62, 67).

[45]M. Z. Darestani, A. S. Chaudhari, and R. Heckel, “Measuring robustness in deep learning based compressive sensing,” in *International Conference on Machine Learning*, PMLR, 2021, pp. 2433–2444 (cit. on p. 33).

[46]S. Malek, F. Melgani, Y. Bazi, and N. Alajlan, “Reconstructing cloud-contaminated multispectral images with contextualized autoencoder neural networks,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 4, pp. 2270–2282, 2017 (cit. on p. 35).

[47]T. Markchom and R. Lipikorn, “Thin cloud removal using local minimization and logarithm image transformation in hsi color space,” in *2018 4th International Conference on Frontiers of Signal Processing (ICFSP)*, 2018, pp. 100–104 (cit. on p. 35).

[48]Q. Zhang, Q. Yuan, C. Zeng, X. Li, and Y. Wei, “Missing data reconstruction in remote sensing image with a unified spatial-temporal-spectral deep convolutional neural network,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. 56, no. 8, pp. 4274–4288, 2018 (cit. on p. 35).

Colophon

This thesis was typeset with $\LaTeX\,2_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

NTNU
Kunnskap for en bedre verden