Politecnico di Torino

Master Degree course in Communications Engineering

Master Degree Thesis

Deep Learning for Onboard Real-Time Cloud Segmentation in Satellite Images

Supervisors
Prof. Enrico MAGLI
Prof. Diego VALSESIA

Candidate
Giuseppina Maria RIZZI

October 2024

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Professor Enrico Magli and Professor Diego Valsesia, for giving me the opportunity to work under their guidance.

Their valuable suggestions and constructive feedback have been fundamental for the development of this thesis work. I truly appreciate the support, dedication and patience throughout this process.

Abstract

The Earth observation satellites have allowed us to monitor the planet at a scale that seemed unattainable up to some years ago. Satellite imagery plays a fundamental role in many applications, like forestry, tracking wildfires, mapping deforestation, railway industry, urban planning or agriculture.

Terabytes of image data are produced everyday, making it necessary to have automated methods to recognise anomalies inside the images before using them for further analysis. One of the main obstacles in satellite imagery is represented by the presence of clouds, which regularly cover 66% of the Earth's surface, obscuring the ground objects we are interested in. This thesis studies the use of Deep Neural Networks to design an on-board algorithm able to perform cloud segmentation in real time with the aim of classifying each pixel of the satellite images as cloud or background.

This work investigates the design of a deep learning model based on a hybrid recursive attention mechanism that processes images one line at a time. This architecture, through its ability to model dependencies within sequences of complex data while maintaining system simplicity, offers significant advantages in terms of memory efficiency and reduced latency compared to traditional 2D approaches. Specifically, the novel Mamba layer handles the along-track direction of the images, while the 1D convolutional layers of the U-Net process the columns.

To assess the prediction capabilities of the developed solution, a comprehensive evaluation of performance metrics, such as accuracy, precision, recall, and computational efficiency is provided. Additionally, a comparative analysis with state-of-the-art approaches is conducted to demonstrate the superiority of the proposed architecture when handling real-time processing with memory constraints.

Contents

List of Figures

List of Tables

Acronyms

Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
ANN	Artificial Neural Network
CE	Cross-Entropy
CNNs	Convolutional Neural Networks
CPU	Central Processing Unit
ESA	European Space Agency
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory

IDE	Integrated Development Environment
IoU	Intersection over Union
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSI	MultiSpectral Instrument
NIR	Near Infra-Red
NN	Neural Network
RNNs	Recurrent Neural Networks
RGB	Red Green Blue
SGD	Stochastic Gradient Descent
SSH	Secure Shell
SRAM	Static Random Acess Memory
SSM	State Space Models
SWIR	Short Wave Infra-Red
TN	True Negative
TNR	True Negative Rate
TP	True Positive

TPR	True Positive Rate
VPN	Virtual Private Network
VSCode	Visual Studio Code
VNIR	Visible and Near-Infra-Red

Chapter 1

Introduction

1.1 Motivation

Satellite images have made the Earth's observation from high altitudes feasible. Having the possibility to monitor our planet from above is crucial for many applications, like forest degradation assessment, urban areas growth tracking, natural disasters' analysis, agriculture, industry, weather patterns discovery [16].

The access to the space is becoming more affordable lately, and new resources are put into orbit. However, the complexity and the amount of operations to be performed on board the satellite have increased, and for this reason the available work power and the transmission bandwidth may become limiting factors. A possible approach to reduce the impact of this problems is to lower the amount of images to be transmitted.

Cloud coverage reduces the visibility of the Earth's surface, making the satellite images useless for many of the applications previously listed. Having automatic methods that are able to discard the corrupted images can help in saving computing and bandwidth resources.

Artificial Intelligence (AI) is playing an important role in a variety of space-related applications that have to manage a massive amount of data. Most of the approaches focus more on the down-link image processing because of the higher computing resources availability. However new algorithms for on board cloud discovery in satellite images have been developed and the research field is still interested in finding new advances to optimize the up-link processing. Convolutional Neural Networks (CNNs) combined with the power of SSM represent a possible solution.

1.2 Objectives

The aim of the thesis is to develop a Deep Neural Network algorithm for effective semantic segmentation in the satellite imagery field. The objectives of this work are:

1. To design a Deep Learning model for cloud detection with low computational complexity, suitable for the implementation on board satellites, where the computing resources are limited. To achieve this, images will be processed line by line, significantly reducing the memory usage and computational demands. The approach will integrate the U-Net architecture, used for semantic segmentation, with the Mamba framework, which provides a memory of previous lines to ensure accurate segmentation without processing the entire image at once.

2. To achieve satisfying values of accuracy in the predictions on cloud coverage in satellite images

3. To reduce the waste of resources in terms of computational power, transmission bandwidth and latency.

4. To train, validate and test the model with real images coming from the Sentinel-2 mission for Earth's surface monitoring.

5. To perform model evaluation through typical Deep Learning metrics, like accuracy, precision, recall, F-1 score and Intersection over Union (IoU).

6. To find a solution to potential problems that may arise during the model implementation.

1.3 Methodology

This work is based on the use of deep learning for image segmentation. The approach follows the typical Neural Network workflow, which consists in processing the data collection and designing, training, validating and testing the model.

Satellite data come from the Sentinel-2 mission, which captures wide-swath, high-resolution, multi-spectral images that are publicly available. They show high resolution and multi-band capabilities, which make them suitable for cloud detection applications.

The model is built by integrating the U-Net framework, which is the leader architecture in semantic segmentation tasks, with the Mamba framework. The resulting model can reach higher accuracy levels thanks to the combination of Recurrent Neural Networks (RNNs) and SSM.

The dataset is split into training, validation and test sets to properly evaluate the capabilities of the model, also exploiting several performance metrics.

1.4 Organization

The thesis is organized into five chapters explaining in a more exhaustive way the topics previously introduced.

A more detailed description of the chapters' subjects is presented in the following:

- Chapter 1 is the current section, and it provides a general overview of the thesis' argument, organization and methodology.
- Chapter 2 describes the state of the art in the fields of satellite imagery and deep learning for image segmentation. An explanation of what SSM are and how they work is also provided to introduce the capabilities of the Mamba architecture.
- Chapter 3 shows the workflow implementation, starting from the description of the hardware and software set up, moving to the dataset discovery and processing and finally reaching the model design and training. Also a description of the performance metrics used during the training and test steps is provided.
- Chapter 4 reports and analyses the results obtained during the training and test steps, and also the performance comparison with other models that are typically exploited in cloud segmentation tasks.
- Chapter 5 presents the conclusion of this thesis and some possible further research topics for future works.

Chapter 2

State of the Art in Satellite Image Interpretation and Deep Learning

The space sector is currently exerting a strong appeal for new investors, with a significant growth of the small satellite industry. New private ventures are investing in this market to work alongside leader agencies and companies, highlighting how this sector is still full of opportunities to exploit. At the same time, a rapid expansion has also been registered in the AI field allowing for information extraction from big datasets. If applied to image processing scenarios, AI-powered Computer Vision techniques help solving complex problems such as medical diagnosis and autonomous driving.
By installing AI systems on edge devices like satellites, innovative solutions can be implemented to reduce memory usage and processing time with respect to traditional software approaches.

2.1 Exploring Satellite imagery

AI combined with its subset ML has enabled high accuracy in Remote Sensing for a large variety of Earth observation applications.

"Remote sensing is the practice of deriving information about the Earth's land and water surfaces using images acquired from an overhead perspective, using electromagnetic radiation in one or more regions of the electromagnetic spectrum, reflected or emitted from the Earth's surface." [5]

The first attempts to take Earth's pictures can be dated in the early 1800s, when the practice of photography started. However, it was at beginning of World War I that the need for military reconnaissance and surveillance, led to the acquisition of aerial pictures as a regular practice.

In the interwar years (1919-1939) new instruments specifically designed for the analysis of aerial photos were realized. Another important driver at that time was represented by the worldwide economic depression that started in 1929. The government started to be interested in remote images of the Earth to monitor the rural development as a means to counteract the economic crisis.

During World War II, the infrared and the microwave spectra were identified as really potential bands for the space images processing, however, there were not instruments able to work with them. This revelation set the basis for new researches in the photo-interpretation field, which led to the building of the CORONA satellite during the Cold War period, for image collection from the space.

A milestone in the history of satellite imagery is represented by the launch of Landsat1 in 1972. It was the first of many satellites orbiting around the Earth, and it provided pictures of large areas of the planet's surface in several regions of the electromagnetic field. Thanks to the Landsat experiment, the interest in the multispectral data analysis from the scientists increased, the use of digital analysis for remote sensing expanded and new land observation satellites were designed and operated by several organizations around the world.

A new mission called Sentinel-2 has begun in 2015 [10], when the European Space Agency (ESA) has launched the first of the two satellites S2A and S2B into the orbit. The aim of the mission is to monitor the green areas of the Earth to help facing natural disasters. The two satellites contain a MSI that works passively collecting sunlight reflected from the planet. The incoming light is collected by the instrument and split into two beams, one made up of Visible and Near-Infra-Red (VNIR) bands and the other of Short Wave Infra-Red (SWIR) bands.

The operating principle of the MSI is based on a push-broom concept. A push-broom sensor collects images one row at a time across the orbital swath, and new rows can be acquired thanks to the motion of the spacecraft along the orbit. The observation time varies between 17 and 32 minutes.

The MSI can collect reflected sunlight in 13 spectral bands that range from VNIR to SWIR, as shown in Table 2.1 :

Table 2.1: Spectral bands collected by the MSI

Band number	**S2A Central wavelength (nm)**	**S2A Bandwidth (nm)**	**S2B Central wavelength (nm)**	**S2B Bandwidth (nm)**
1	442.7	20	442.3	20
2	492.7	65	492.3	65
3	559.8	35	558.9	35
4	664.6	30	664.9	31
5	704.1	14	703.8	15
6	740.5	14	739.1	13
7	782.8	19	779.7	19
8	832.8	105	832.9	104
8a	864.7	21	864.0	21
9	945.1	19	943.2	20
10	1373.5	29	1376.9	29
11	1613.7	90	1610.4	94
12	2202.4	174	2185.7	184

These images are available for free, and can be obtained in entire swaths from ESA and from USGS Earth Explorer or Copernicus Open Access Hub.

2.2 Computer Vision

Computer vision is the area of Machine Learning that aims at interpreting and understanding images and videos [25].

The heart of the matter is represented by the research for mathematical models that can recover the three-dimensional shape and appearance of objects and that can reconstruct their properties, such as shape, illumination, and color distributions. In other words, there is the desire of reproducing the capabilities of humans in performing visual tasks. Up to now, it is possible to reconstruct a partial 3D model of an environment by processing thousands of overlapping photographs of it. However this process is not trivial, because vision is an inverse problem, meaning that there is the attempt to recover some unknowns given insufficient information to fully specify the solution.

Computer vision is now used in a variety of different applications like automotive safety, biomedical imaging, object recognition for automated systems, machine inspection, 3D model building, surveillance, fingerprint recognition [26].

Accurate and complex models are usually realized by relying on deep learning technologies. The main tasks [7] that can be performed in the field of object recognition are:

- **Classification**: the purpose is to assign one or more semantic labels to each image such as ’urban’, ’forest’, ’agricultural land’

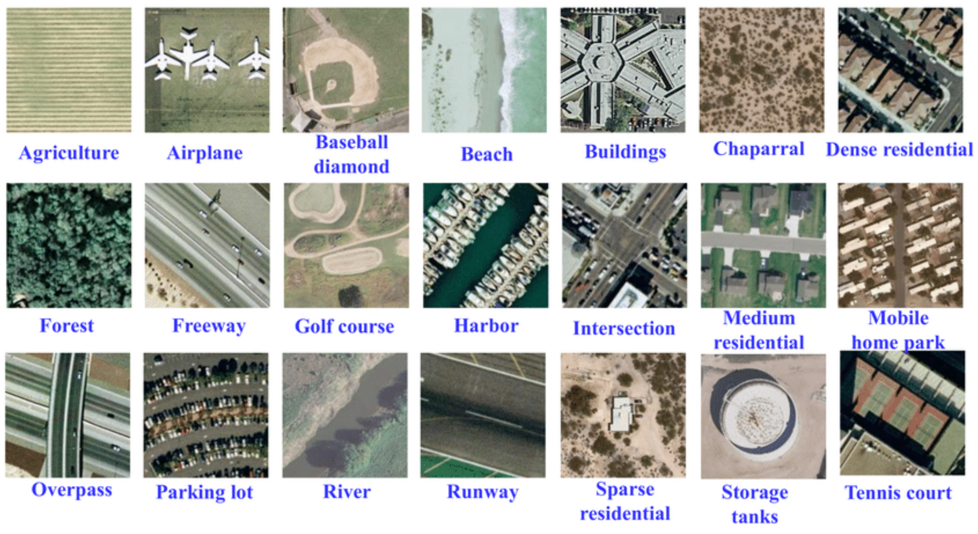


Figure 2.1: Image classification examples

- **Semantic Segmentation**: the aim is to divide the image into semantically meaningful segments or regions. A class label is assigned to each of the pixels inside the image to identify the class it belongs to.

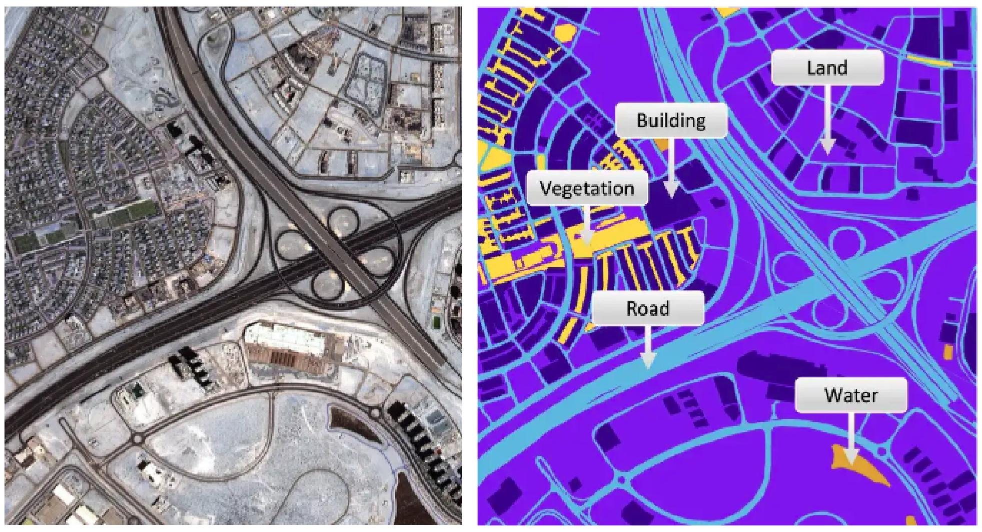


Figure 2.2: Semantic segmentation examples

- **Object detection**: the goal is to detect and surround objects with bounding boxes. The accuracy increases if the image resolution increases, so high resolution images are fundamental in the field of satellite imagery and remote sensing. Objects may be oriented in any direction, which means that bounding boxes have to be used also in a rotated way.

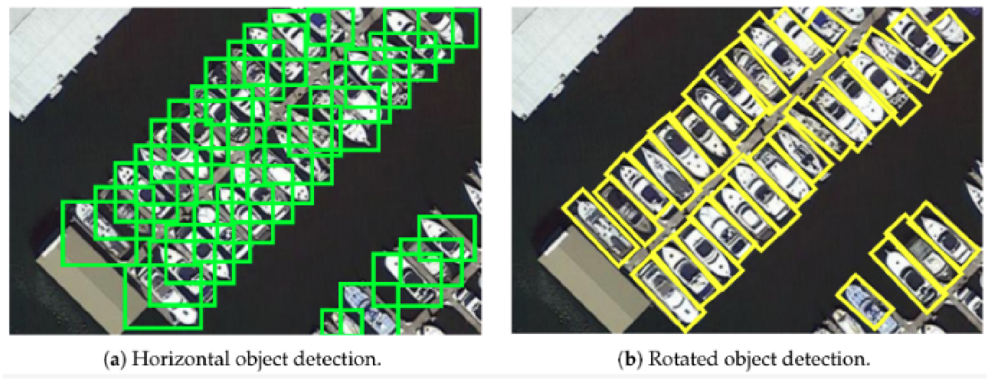

(a) Horizontal object detection.
(b) Rotated object detection.

Figure 2.3: Object detection examples

2.3 Deep Neural Networks for Earth observation

Deep learning has revolutionized the way images are analyzed and processed, solving also the issues related to the vast image sizes and to the wide arrays of object classes.

2.3.1 From Artificial Intelligence to Deep Learning

Artificial intelligence can be defined as the development of computer systems that can perform tasks typically requiring human intelligence, such as learning, problem-solving, and decision-making. [20]
There are two different kinds of AI, weak AI and strong AI. The former is easy and specific applications oriented, like autonomous driving. The latter, instead, can almost mimic human brain in solving problems, learning from already existing data and planning future actions.

When AI started to be used as a tool to make predictions, it relied on a set of predefined rules that had to be provided by an expert on the topic. Its aim was to simulate the human brain capability of solving problems through a computing machine. The amount of data to be processed, however, increased massively with the time, and new data-driven approaches started to be required. It was in this context that machine learning became a fundamental application of AI [22].

ML is the set of algorithms and tools that machines use to discover and understand patterns inside the input data with the aim of addressing specific tasks. The workflow of a ML algorithm is made up of three main parts [12]:

1. Decision process: according to the input data, an estimate of the pattern inside them is produced.

2. Error function: is used to evaluate the performance of the model in terms of accuracy of the prediction

3. Model Optimization Process: the model is adjusted so that the estimate can be as close as possible to the known data.

Usually **Deep Learning** is considered a competitive technology to ML, but actually it is a sub-domain, as shown in Figure 2.4. The main difference is represented by the higher computational capabilities of Deep Learning that allow for the parallelization of the work to perform predictions.

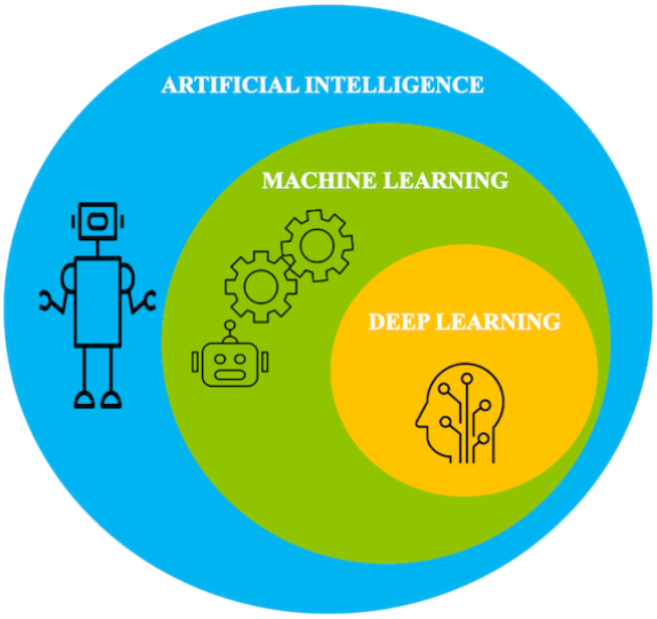


Figure 2.4: Relationship between AI, ML and Deep Learning

The fundamental basic building block of the architecture for deep learning is called perceptron. It mimics the behavior of a brain neuron, in fact, as a real neuron, it needs an input signal to activate and to produce some outputs. As for any ML algorithm, the actions that the perceptrons have to perform are oriented towards the recognition of specific patterns inside the data. By generating an entire layered network of perceptrons, deep learning is able to emulate the capabilities of the human brain; for this reason this architecture is called Neural Network (NN) orArtificial Neural Network (ANN).

2.3.2 Artificial Neural Networks

As already mentioned in subsection 2.3.1, the basic building block of an ANN is the perceptron. It receives some inputs x_i which are the equivalent of the neuron's dendrites.

They are weighted by factors w_i to determine their relative importance before being linearly combined and then summed to a bias parameter. Perceptrons have to be activated through a non-linear activation function, as well as neurons have to receive an electrical impulse to activate. A comparison of the biological perceptron and the artificial neuron is shown in figure 2.5

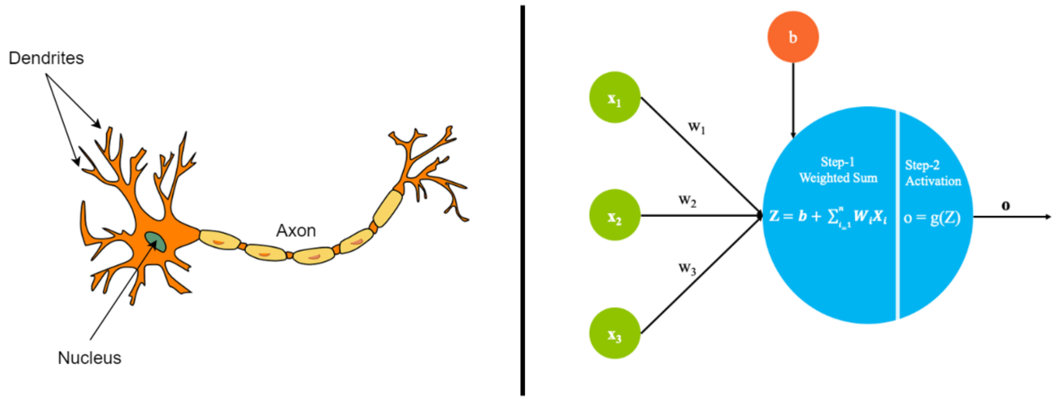


Figure 2.5: Biological neuron vs artificial perceptron

A neural network is a collection of many perceptrons which are arranged in parallel and in layers, as shown in Figure 2.6.

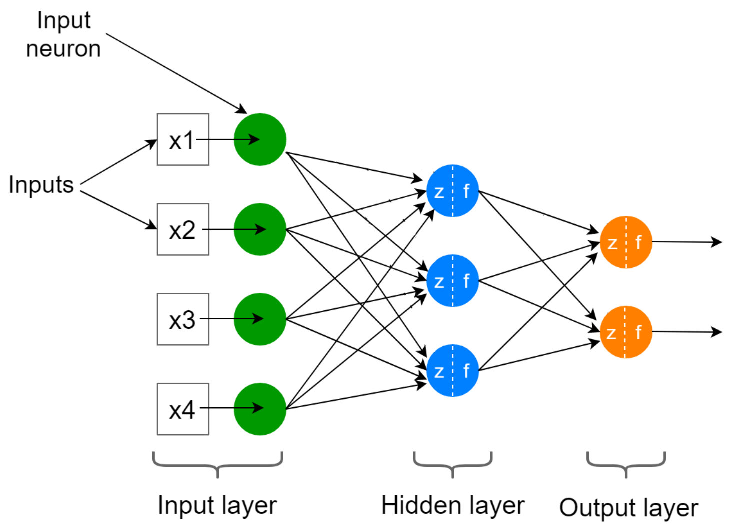


Figure 2.6: Shallow Neural Network scheme

The workflow of the perceptron described above is known as *forward propagation*. At the end of this process, the predicted output is compared to the ground truth label, allowing for weights adjustment to improve prediction accuracy at the next iteration. This step is known as *backpropagation*.

The way the neurons inside a layer are connected to the nodes of the next layer, strongly influences the output of the neural network. It is then fundamental for the developer to properly design the interconnections based on the goal of the designed model.

2.3.3 Fundamental Neural Network Layers

When dealing with image processing and segmentation, it is fundamental to design performing NNs. Understanding the contribution of the layers inside the architecture is crucial to fully leverage their potential. Each of them has an impact on the overall functionality and performance of the model. This section provides an overview of the layers commonly used in image processing tasks, focusing on their characteristics and functions.

- **THE FULLY CONNECTED LAYER**

  The Fully Connected Layer [28] is the simplest kind of NN layer. All the inputs are connected to all the neurons, and in this way it implements a matrix vector product as in 2.1:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + b) \tag{2.1}$$

  where:

  - **y**: $[F_{out}$ x $1]$ is the output vector,
  - **x**: $[F_{in}$ x $1]$ is the input vector,
  - **W**: $[F_{out}$ x $F_{in}]$ is the weight matrix,
  - **b**: $[F_{out}$ x $1]$ is the bias vector,
  - f is the non-linear activation function

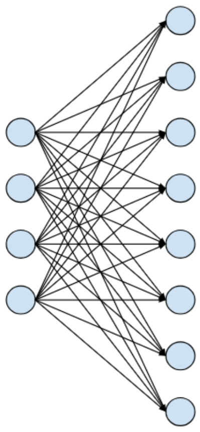


Figure 2.7: Fully Connected Layer scheme

Several non-linear activation functions are available in the literature, and the choice depends on the task to address. The most commonly used functions are outlined in the following:

- sigmoid: saturates for high or low values

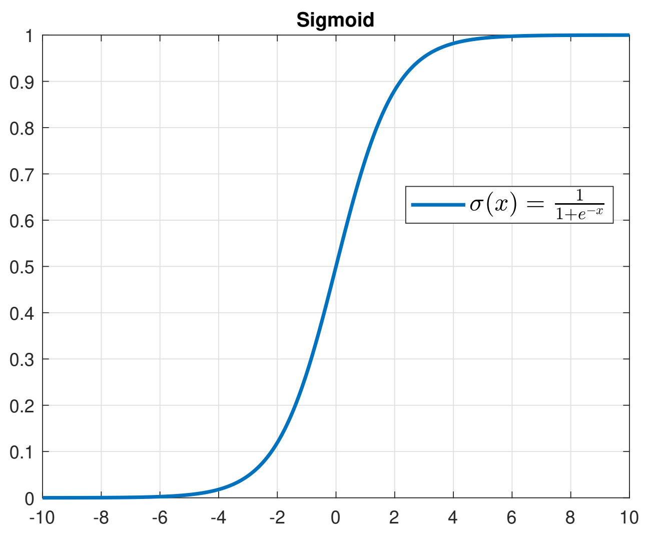


Figure 2.8: Sigmoid function

- ReLU (Rectifier Linear Unit): does not saturate, fast to implement, dying neurons due to hard zero for negative inputs

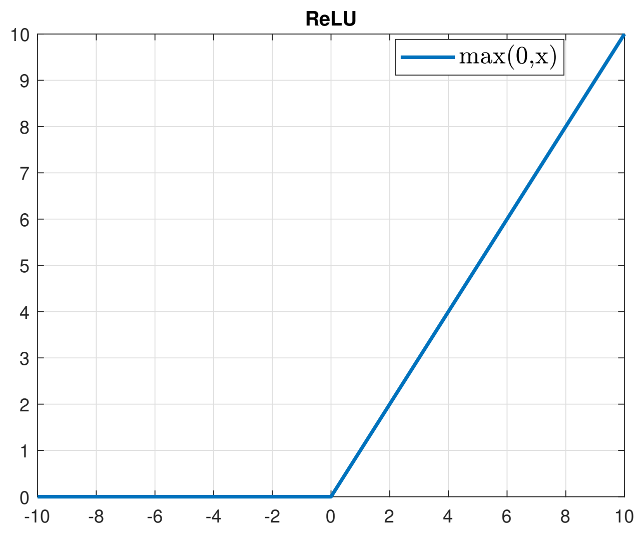


Figure 2.9: ReLU

– Leaky ReLU: does not saturate, non dying neurons due to small non-zero values for negative inputs

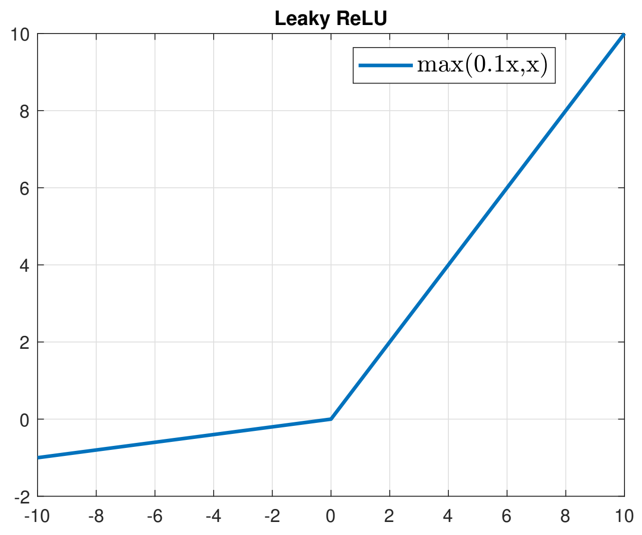


Figure 2.10: Leaky ReLU

- **THE CONVOLUTIONAL LAYER**

  The Convolutional Layer [11] uses some filters (convolutions) to exploit data properties and reduce the number of parameters. The ingredients to build a convolutional layer are the input data, a filter and a feature map. If the input is a color image in the Red Green Blue (RGB) scale, it will have three dimensions:

– **height**: number of pixels from the top to the bottom,
– **width**: number of pixels from the left to the right,
– **number of channels**: the three color layers (Red, Green, and Blue).

A kernel or a filter, will move across the receptive field of the image to check if the feature is present or not. This operation corresponds to a 2D convolution.

The filter is a matrix of weights whose dimension may vary, but typically is equal to 3x3. A dot product is applied between the filter and the pixels belonging to the region of the image inside the receptive field. The filter can then shift by a stride to repeat the same procedure until the entire image has been swept. A graphic representation of a convolutional layer is provided in Figure 2.11

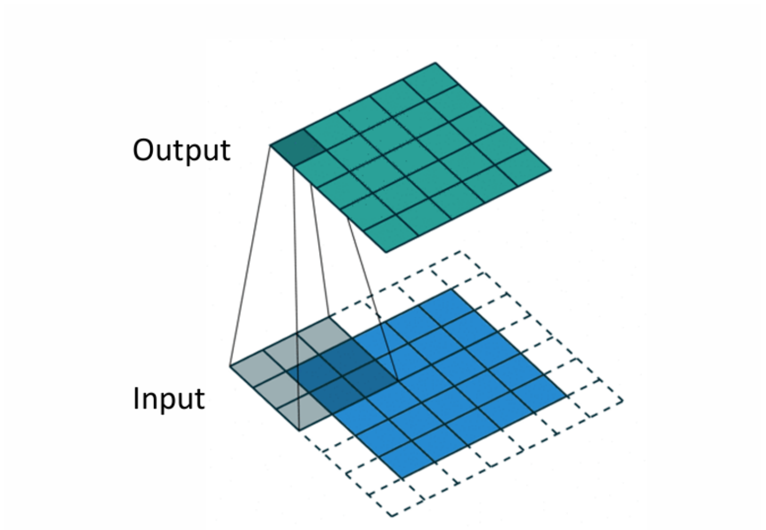


Figure 2.11: Convolutional layer

There is some degree of freedom in designing the convolutional layer:

– Number of filters: has an impact on the models depth
– Stride: number of pixels the kernels moves over the input matrix. It affects the final dimension of the output
– Zero-padding: all the elements of the filter that exceed the input matrix are set to zero to obtain an output matrix with the same size as the input one. There are three different padding options which are valid padding or no padding, same padding or full padding to increase the output dimension.

- **THE POOLING LAYER**

  The Pooling layer is used to reduce the dimensionality of the system by reducing the number of input parameters. A filter is used to aggregate the elements within the receptive field, adopting one of two approaches: in the case of max pooling, the maximum element inside the receptive field is selected to populate the output matrix, otherwise, in the case of average pooling, the mean value is computed. A graphical representation is provided in Figure 2.12.

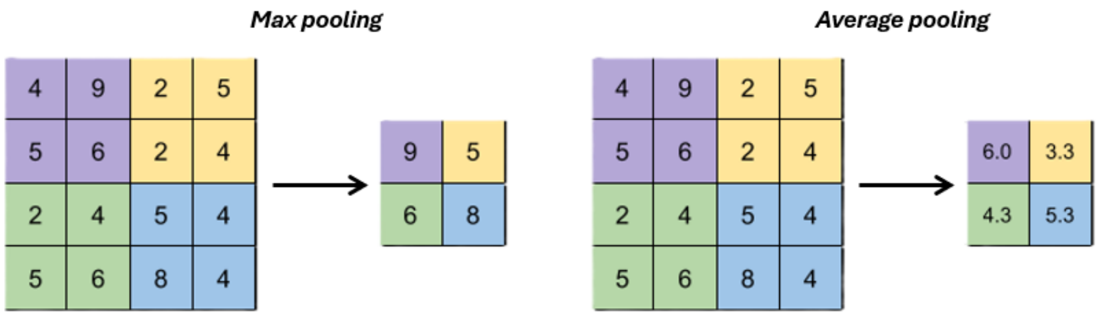


Figure 2.12: Max pooling vs Average pooling

- **THE DROPOUT LAYER**

  The Dropout layer has the aim of randomly removing a certain percentage of the neurons during training, as shown in Figure 2.13. In this way the chances of co-adaptation of neurons are reduced, and overfitting is avoided. The problem of overfitting is due to the model learning too much from the training data, to the point that it is not able to generalize anymore.

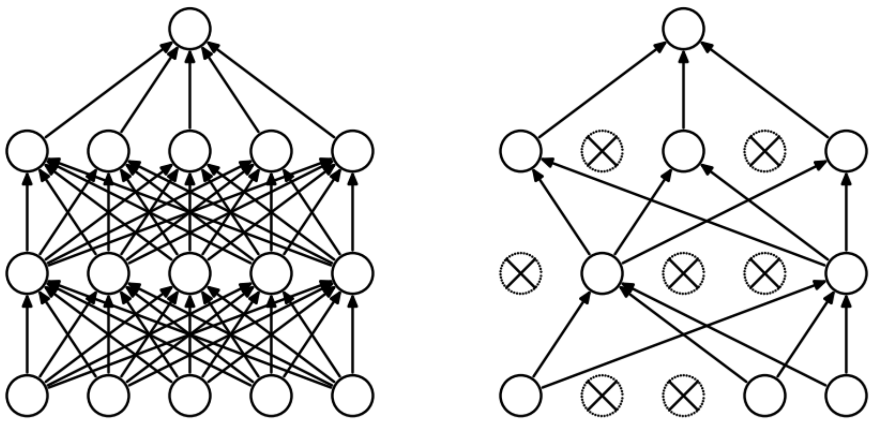


Figure 2.13: Neural Network before and after applying dropout

- **THE CONCATENATE LAYER**

  The Concatenate Layer is added to the model to combine several layers in such a way that they produce a single tensor.

- **THE BATCH NORMALIZATION LAYER**

  The Batch Normalization Layer is used to stabilize the network during training [8]. Usually, all the data is normalized to zero mean and unit variance before inputting it to the deep learning model, according to the formula in 2.2.

  $$x_i = \frac{x_i - mean_i}{std_i} \tag{2.2}$$

  In this way, all the feature values are on the same scale and the gradient computed during the backward pass does not oscillate too much along one dimension. Thanks to data normalization, in fact, it can smoothly reach the minimum without wasting time going back and forth along the trajectory.

  The reason why normalization is applied to the input data is the same as the logic used when normalizing the inputs to the hidden layers inside the networks. The Batch Normalization Layer is then placed between an hidden layer and the next one and computes the mean and standard deviation over a batch of inputs, that is a small set of images taken from the dataset.

  Batch Normalization layers work differently during training and inference. During training, the values are normalized according to the mean and standard deviation of the current batch of inputs; during inference, instead, normalization is performed using the moving average of the means and standard deviations computed over all the batches during the training process. For this reason, the test set has to show similar statistics to the training set.

2.4 Challenges and Opportunities of Real-Time On-Board Processing in Satellites

Satellite imagery plays a fundamental role in many applications, like forestry, tracking wildfires, mapping deforestation, railway industry, urban planning or agriculture. Thanks to remote sensing, it is possible to monitor the Earth's surface providing critical information about the planet's properties evolution over time.
While being undeniably valuable for a wide range of applications, satellites and satellite imagery have to address several challenges [6], such as:

- **Memory management**: the memory available on board the satellites is limited by the physical constraints of mass and volume. AI algorithms need considerable amounts of resources depending on the weights, the architecture of the model and the input data size. Moreover, moving bits and memory leak have to be addresses when dealing with memory management. Very complex semantic segmentation models may even surpass satellite's memory budget, therefore optimizing performance is fundamental when designing the algorithm.

- **Computational capacity**: the computing capabilities of the satellite's processor are limited by the mass, volume, ability to withstand radiations, heritage, safety and general satellite's requirements. For this reason AI algorithms may run too slow and the situation can become even worse when dealing with CNNs, which are made up of several convolutional layers.

- **Battery power consumption**: sensors, transmitting equipment, actuators, processors and satellite's hardware need electrical power to work. It is retrieved from solar panels, whose surface is limited by the satellite's extension, or generated on-board the satellites. Computing hardware used for AI algorithms and GPUs are responsible for high power consumption representing for this reason a limitation.

Despite all the challenges that on board image processing implies, it is still considered beneficial to implement AI-algorithms directly on satellites because of the significant benefits that they can provide to the scientific community.
The main advantages brought by real-time image processing include:

1. Eliminating the need to compress and transmit data acquired by the sensor, providing more precise information to the processor.

2. Reducing communication overhead between the satellite and the ground station.

3. Lowering the burden on ground-based data processing equipment.

4. Providing real-time output that can be used faster for the target operation.

Instead of focusing on the memory, computational capacity and battery improvement, most approaches focus more on the optimization of the deep learning architectures to obtain lightweight models allowing for an efficient real-time processing.

In this thesis a novel design for a cloud segmentation deep network is presented. It differs form already existing architectures for many reasons:

- It works recursively in the along-track direction in such a way that memory usage is only limited to the current line and all its spectral components.

- It can provide real-time processing as it does not need to wait for the entire image to be available because it works line by line.

- It exploits a hybrid attention-recursive mechanism that can model dependencies performing a content-based reasoning that scales linearly within a large context window.

- It employs the Mamba layer, which is designed for long sequence segmentation, to process an image.

2.5 Evolution of CNN architectures for semantic segmentation

Since the novel proposed method is specifically designed to work differently from all the traditional 2D methods, it is worth reviewing the conventional approaches. This will highlight the key distinctions and potential advantages of the new line by line approach with respect to standard techniques.

This section provides an insight of the CNNs and U-Net architecture, whose development has enhanced segmentation capabilities, allowing for more accurate and efficient predictions on complex data.

2.5.1 Convolutional Neural Networks

CNNs are the leader architectures in image-driven pattern recognition tasks. As traditional ANNs, they are able to self-optimize thanks to the process of backpropagation. The main advance that they introduce with respect to ANN is that they can manage the computational complexity that working with images implies [21].

They are made up of three types of layers, which are:

- **Convolutional layers**: to extract the features from the images
- **Pooling layers**: to downsample and reduce the number of input parameters
- **Fully-connected layers**: to combine the extracted features and output the class scores for the classification task

and they are organized as shown in Figure 2.14

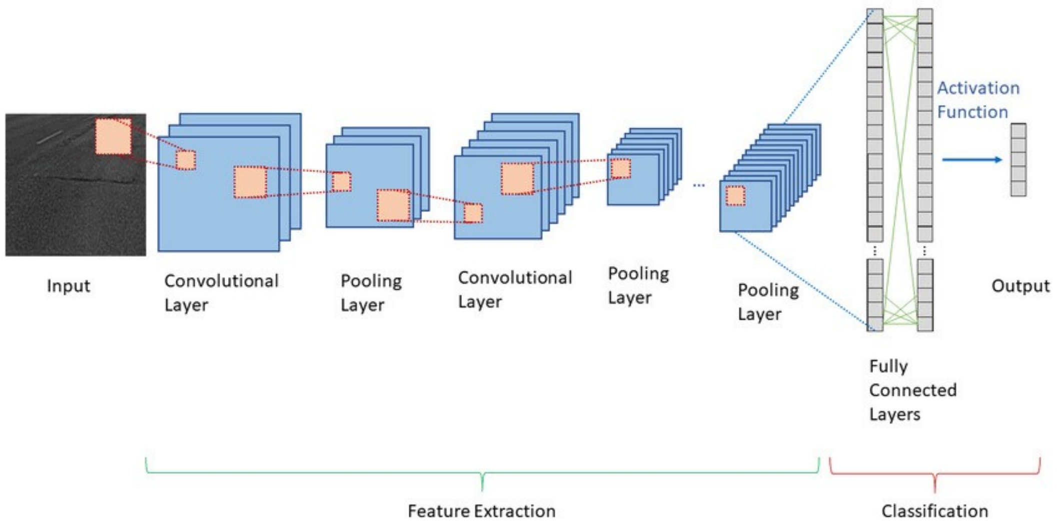


Figure 2.14: Convolutional Neural Network architecture

The sequence of convolutional layers is essential to exploit the compositional nature of the features extracted by the CNNs. Early layers can extract simple features, like edges or corners, while successive layers can extract higher semantic concepts, like small objects [28].

2.5.2 U-Net architecture

CNNs have been the most suitable model for many image recognition tasks for years, however their performance was limited for those applications that required large training sets and deep architectures.
In recent years, a new model, the so-called "UNet", has been built. It is able to provide more precise segmentations while keeping the size of the training dataset limited [19].

The architecture of the U-Net consists of the sequence of a contracting path and an expansive path. A graphical representation in provided in Figure 2.15.

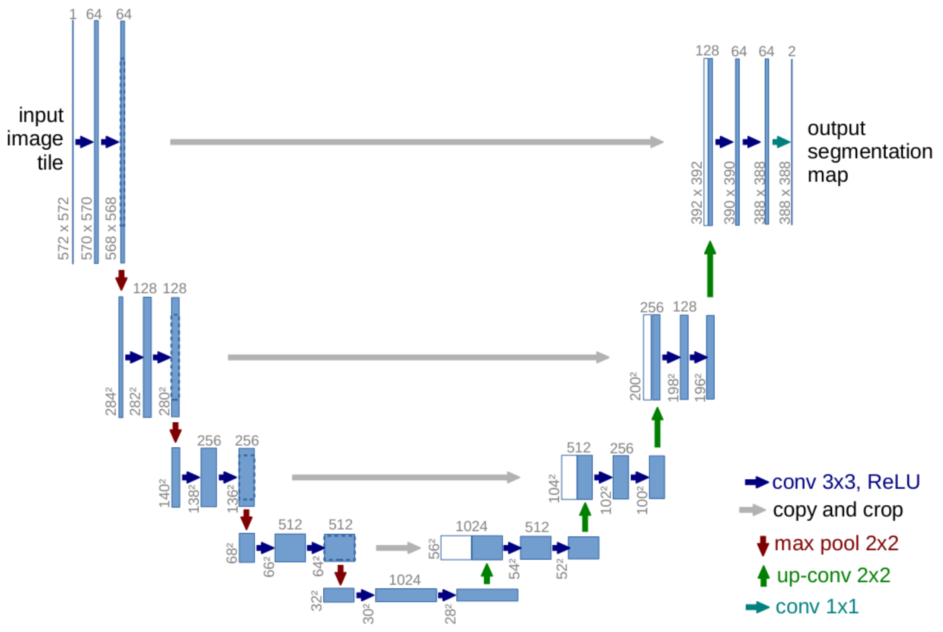


Figure 2.15: U-Net architecture

The **contracting path** is designed as a conventional CNNs, with the repetition of several blocks made up of:

- an unpadded 3x3 convolutional layer,
- a ReLU activation function,
- a 2x2 max pooling layer with a stride of 2.

Convolutional layers are responsible for the doubling of the extracted features, while pooling layers for the halving of the images dimensions. In this way, each downsampling step is associated to the doubling of the features number.

The **expansion path**, instead, is realized through the sequence of several blocks made up of:

- an upsampling layer,
- a 2x2 convolutional layer,
- a concatenation layer,
- a ReLU activation function.

The upsampling layer doubles the images' dimensions, while the 2x2 convolutional layer halves the number of feature channels. The concatenation layer is used to concatenate the current input with the correspondingly cropped feature map in the contracting path.

Finally, the last layer is a 1x1 convolution whose aim is to map each of the elements in the feature vector to the proper number of classes.

2.6 State Space Models and Mamba

The conventional CNNs and U-Net models presented in the previous section can only operate on 2D inputs. Some studies [15], however, have demonstrated that working with the recently developed 1D CNNs can provide numerous advantages for the following reasons:

- The computational complexity of 1D convolutions is significantly lower compared to the one of 2D convolutions. Considering an image with dimension NxN and kernel KxK, the computational complexity is $\sim O(N^2K^2)$ if 2D convolution is exploited; under equivalent conditions but using 1D convolutions, instead, complexity becomes $\sim O(NK)$.
- Models with 1D convolutions typically show more compact configurations, having even less than 10K parameters. 2D convolutional layers, instead, require deeper architectures with a number of parameters above 10M. This has implications in the complexity of the training and implementation of the algorithm.
- Training deep neural networks with 2D convolutional layers typically requires a specific hardware setup, like Cloud computing or GPU farms, while when using 1D convolutional layers even a standard CPU can be highly performing for simple architectures.

Thanks to the low computational complexity, 1D convolutional layers are the best approach for real-time applications having limited computational and storage capability. For this reason, they are also well-suited for a possible on-board the satellites implementation.

As previously introduced in Section 1.2, the novel architecture proposed in this thesis work integrates the U-Net architecture, designed with 1D convolutional layers, with the Mamba framework, which provides a memory of previous lines to ensure accurate segmentation without processing the entire image at once.

For this reason, a detailed overview about the Mamba layer and how it manages the selection mechanism is provided in the following sections.

2.6.1 From Multi-Head attention to the Transformer architecture

The process of exploiting the memory of past lines requires the implementation of the Multi-Head Attention mechanism. Attention mechanisms allow the modeling of the dependencies without any limitation related to their distance inside the input and output sequences.

An **attention function** is used to map a query and a set of key-value pairs to an output. The output is obtained as the weighted sum of the values, while the weights are assigned through a compatibility function applied to the query and the corresponding key. Query (Q), key-value (K, V) pairs and outputs are all vectors.

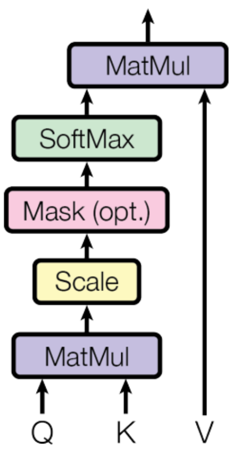
Given the dimension of the queries and keys d_k, and the dimension of the values d_v, the weights are obtained by computing the dot products between the query and all the keys, dividing each of them by $\sqrt{d_k}$ and applying a softmax function.

The outputs are then computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.3}$$

Multi-Head attention projects queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively, instead of just performing a single attention [4].

On each of the projected versions, attention is performed in parallel as shown in Figure 2.16



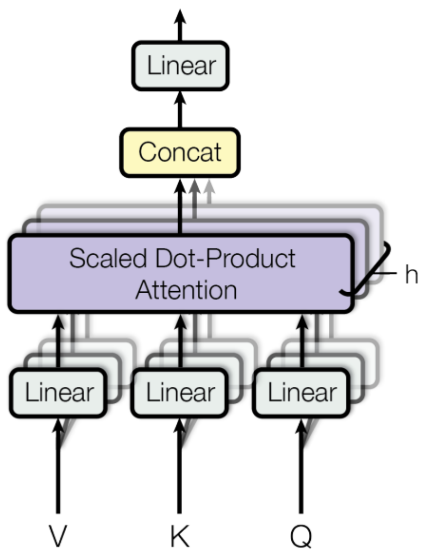


Figure 2.16: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention

Multi-Head Attention is the main block in the **Transformer** architecture [4]. This model is characterized by an encoder-decoder structure arranged as in Figure 2.17

The encoder is built using N=6 identical layers. Each of them is made up of a multi-head attention mechanism followed by a fully connected feed-forward network.

The decoder has also 6 identical blocks, each of them made up of a multi-head attention layer, a fully connected feed-forward layer and a multi-head attention layer which works over the outputs of the encoder.

The Transformer's power lies in its capability of discovering the dependencies between the input and the output totally relying on the attention mechanism. Previous architectures showed the need for recurrent models, which were less parallelizable and required a larger training time. In common architectures, the number of operations to be performed to find the dependencies was growing with the distance between the positions in the input and in the output. Transformers fix the number of steps to be performed, reducing in this way the computational time.

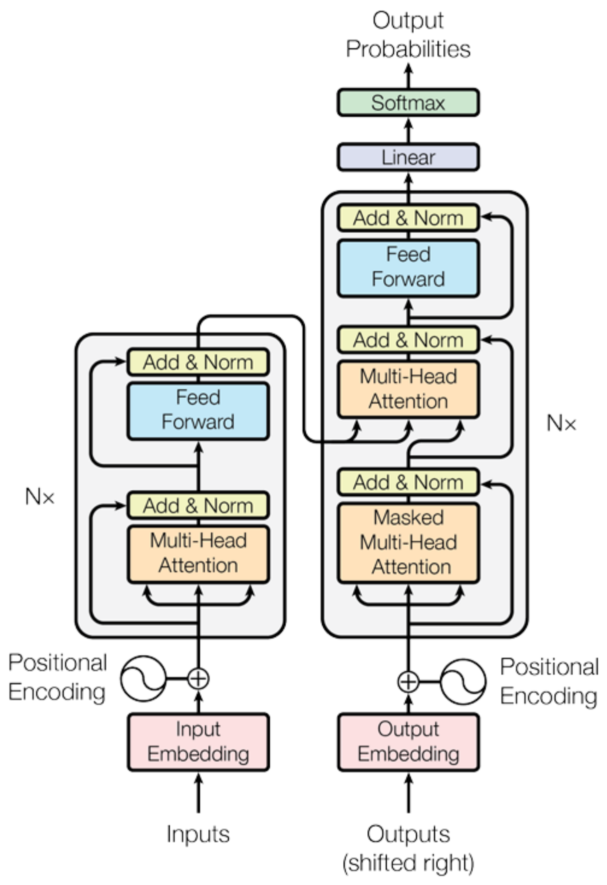


Figure 2.17: The transformer architecture

2.6.2 Selective State Space Models and Mamba architecture

Foundation Models are now the state of the art in many deep learning applications. They are built based on the Transformer architecture and their core attention layers [3]. Transformers, however, have shown low performance in the processing of long sequences, due to the inability to select data in an input-dependent way. The attention module of the Transformer, in fact, can work efficaciously only within a context window, outside of which anything can be modeled. Also, it suffers from quadratic scaling with respect to the window length.

To address these limitations, a new architecture called **Mamba** has been developed. It combines the power of prior SSM with the Multi-Layer Perceptron (MLP) block of the Transformer into a unified framework. The Mamba block solves two of the major issues characterizing prior models:

- by building a selection mechanism that parameterize the SSM internal parameters based on the input, irrelevant information are discarded by the model and long-term dependencies can be simulated,

- by designing a hardware-aware algorithm, improvements are registered also in handling time-variant inputs.

One of the core components of the Mamba block is the **Structured State Space Model**. SSM map a 1-dimensional function or sequence $x(t) \in \mathbb{R} \mapsto y(t) \in \mathbb{R}$ through an implicit latent state $h(t) \in \mathbb{R}^N$.
The workflow of a SSM can be totally defined through the four parameters $\mathbf{A}, \mathbf{B}, \mathbf{C}, \Delta$. It consists of the sequence of the following two stages:

- **discretization**: the continuous parameters $\mathbf{A}, \mathbf{B}, \Delta$ are converted into discrete parameters $\bar{\mathbf{A}}, \bar{\mathbf{B}}$ through fixed formulas:

$$h'(t) = \mathbf{A}h(t) + \mathbf{B}x(t) \qquad h(t) = \bar{A}h_{t-1} + \bar{B}x_t \qquad \bar{K} = (C\bar{B}, C\bar{A}\bar{B}, ..., C\bar{A}^K\bar{B}) \tag{2.4}$$

- **computation**: the model is computed through linear recurrence or global convolution.

$$y(t) = \mathbf{C}h(t) \qquad y_t = \mathbf{C}h_t \qquad y = x * \bar{K} \tag{2.5}$$

By integrating the architecture of structured SSMs with the selection mechanism, the Mamba architecture emerges as an outstanding tool to model long-range dependencies. A graphical representation of the Mamba pipeline is provided in Figure 2.18

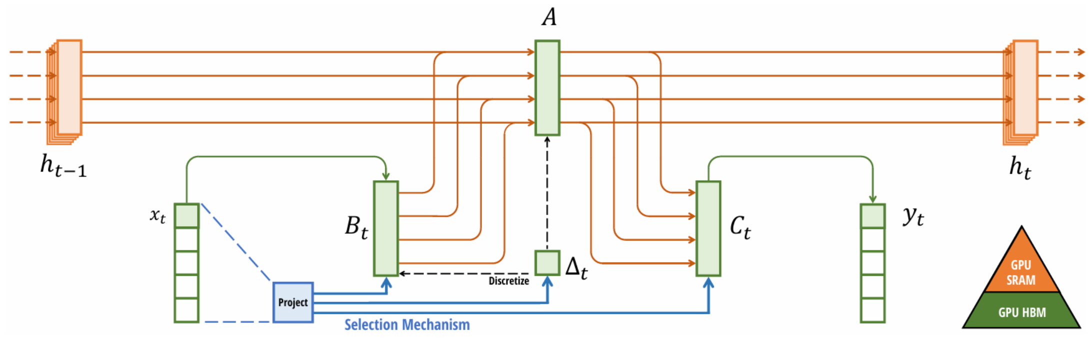


Figure 2.18: Structured SSM with a selective mechanism

It can be summarized as follows:

1. **Input**: h_{t-1} and $x(t)$ on the left of the diagram represent the previous hidden state and the current input
2. **Projection**: x_t is converted into B_t
3. **Selection mechanism**: it involves a discretization step and a selection step to decide which parts of the state to propagate forward and which parts to forget.

4. Δt: is the computation of the state update from h_{t-1} to h_t. It is the parameter used to control the importance of the current input x_t. Large values of Δ are used to reset the state h and focus on the current input, while low values of Δ to ignore the current input and persist the state. To summarize, if $\Delta \to \infty$ the system focuses on the current input for longer, instead, if $\Delta \to 0$, the current input is ignored.

5. **Output**: h_t on the right is the n-dimensional updated hidden state, which maps the word x_t

Figure 2.18 also shows the Graphics Processing Unit (GPU)'s memory hierarchy usage during the execution of the Mamba block. Memory management is one of the key aspects introduced by the Mamba layer, which tries to balance expressivity and speed. Prior models demonstrated to be slower but effective when working with a large hidden state dimension, thus Mamba tries to maximize the hidden state dimension without penalizing speed and memory usage.
The way this result is achieved is by exploiting the capabilities of the modern GPU accelerators to bound most operations by memory bandwidth. Instead of preparing the input into the High Bandwidth Memory (HBM) of the GPU, parameters are directly retrieved from the Static Random Acess Memory (SRAM), which is faster, then results are stored back in the HBM. In this way, the speedup is significant compared to standard implementations.
Also, to save memory usage, intermediate states are recomputed in the backward pass instead of being saved, leading to a reduction of the memory requirements.

Chapter 3

Model development

The aim of this thesis is to design a model for cloud segmentation in satellite imagery. The workflow followed to achieve the final goal is made up of different steps [27]:

1. **Data acquisition**: data can be retrieved from public datasets or can be collected on purpose by the model developer
2. **Preprocessing**: data must be cleaned from noise sources and scaled
3. **Splitting datasets**: the resulting dataset is split into training, validation and test sets. It may also be necessary to handle class imbalance.
4. **Model building and training**: the model is built according to the desired goal and also trained
5. **Validation**: the model's performance are tested on the validation set
6. **Model deployment**: the model is used to perform predictions on unknown data.

Steps 4. and 5. are often performed iteratively several times. The aim of the training step, in fact, is to finely tune the hyperparameters in order to achieve the highest possible accuracy level. The process of adjusting the weights may require many iterations to ensure optimal convergence.

3.1 Development environment

The entire work is developed on a remote server, whose access is guaranteed via Secure Shell (SSH) protocol. Since it has a private IP address, the connection is only possible through a Virtual Private Network (VPN) provided by PoliTo. The server runs on a Linux operating system, which supports command-line programming.

The language chosen for the model implementation is Python 3.8 because it provides a high performance package called "PyTorch" for deep neural network design [1].

There are several software platforms for machine learning development like Tensorflow, Microsoft CNTK, Theano, Caffe, which are available for free, however they do not offer dynamic solutions for models implementation. They require the design and reuse of the same structures many time, and the redefinition from scratch of the entire architecture whenever the designer needs to change the behavior of the network. With PyTorch, instead, thanks to the technique of reverse-mode auto-differentiation, it is easy and fast to change the network behavior with zero lag or overhead.

PyTorch offers two high-level features which are fundamental for the purpose of this thesis:

- Computations with tensors can be done using built-in packages like NumPy, also exploiting the acceleration provided by the GPUs.
- The deep neural networks are built on a tape-based autograd system, which means that the sequence of operations performed in the forward pass and their dependencies are saved into a tape. These information are then exploited during the backward pass to compute the gradient and optimize the weights.

The PyTorch version used for the model development is 2.2.2, because it is compatible with the 12.1 CUDA version. These requirements are strictly necessary for the Mamba layer execution, as clarified in section 3.3.

As already introduced, PyTorch supports the use of GPUs to accelerate operations with tensors. GPUs are hardware accelerators originally designed for graphics rendering. They show an extremely large amount of computational power, which makes them suitable for deep learning applications requiring the execution of a large number of matrix multiplications. Thanks to the highly parallel nature of GPUs, many pieces of data can be processed simultaneously, accelerating in this way processes that would take a larger amount of time to be executed on a Central Processing Unit (CPU) [13].

The server provides a NVIDIA CUDA RTX6000 GPU, whose specifications are available at [17].

Together with the GPU, the server also exploits an AMD Ryzen 7 3700X 8-Core Processor, which is an high-performance CPU made up of 8 physical cores and 16 virtual cores.
The simultaneous use of the CPU and GPU resources makes the parallel and sequential processing possible, improving the processing efficiency required by deep learning models.

The Integrated Development Environment (IDE) used to write and edit the Python code is Visual Studio Code (VSCode), which offers a user-friendly interface supporting almost all the programming languages. Since it offers debugging tools and the possibility to connect to remote servers via SSH, it is suitable for the purpose of this work.

3.2 Dataset discovery

Machine learning algorithms are mathematical models to represent the relationship between the input and the desired output. They require a feature extraction stage, to learn the properties inside the inputs, and a successive stage of decision making to predict the output based on the extracted features. In a deep learning task, both the phases are data-driven, so the entire model is learnt through the data with training.

For this reason, having a suitable dataset is crucial in NN algorithms. Training a supervised deep learning task requires the dataset to be labeled, so that the model can measure accuracy and learn over time.

As already mentioned in Section 1.3, the dataset used for this thesis work comes from the Sentinel-2 mission, which captures wide swath, high resolution, multi-spectral images that are publicly available at [23].

Data are taken from an open source repository called *Radiant MLHub* [9], that offers geospatial machine learning training data generated by the Radiant Earth Foundation team. Several folders of data are available, storing both source imagery and labels ready to be processed. Labels are in the form of masks, with pixels of zero to represent the background and of one to represent the clouds.

After the dataset download, the different folders appear to be organized in the following way:

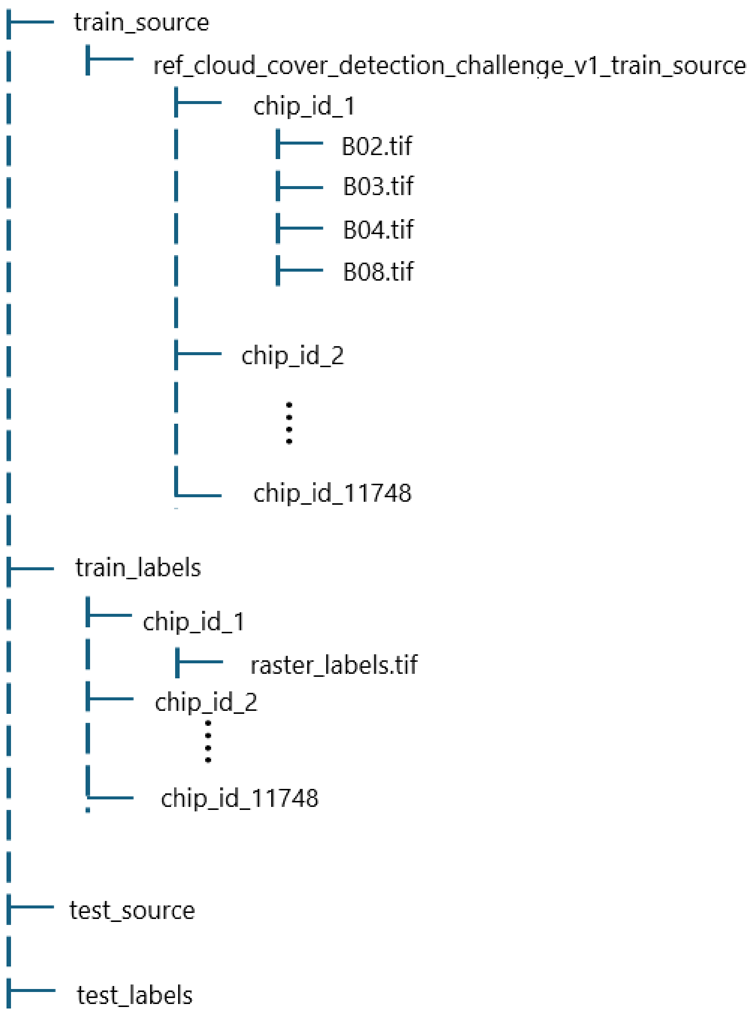


Figure 3.1: Dataset organization

The train_source folder contains the 11748 chips used for the training of the model. Chips correspond to images of specific areas of the Earth's surface at a specific time with a dimension 512x512 pixels. For each chip there is a folder containing 4 different images representing the same area but captured from a different range of wavelengths, according to the scheme in Table 3.1

Table 3.1: Frequency ranges available for each chip

Band name	**Light colour**	**Central frequency**
B02	Blue visible light	497 nm
B03	Green visible light	560 nm
B04	Red visible light	665 nm
B08	Near Infrared light	835 nm

The train_labels folder, instead, contains the labels associated to each of the chip. Finally, the test_source and the test_labels folders are organized respectively as train_source and train_labels, so there are 10980 chips folders containing the 4 bands images and a label image for each of them.

An example of what the images in the training set look like is provided in Figure 3.2

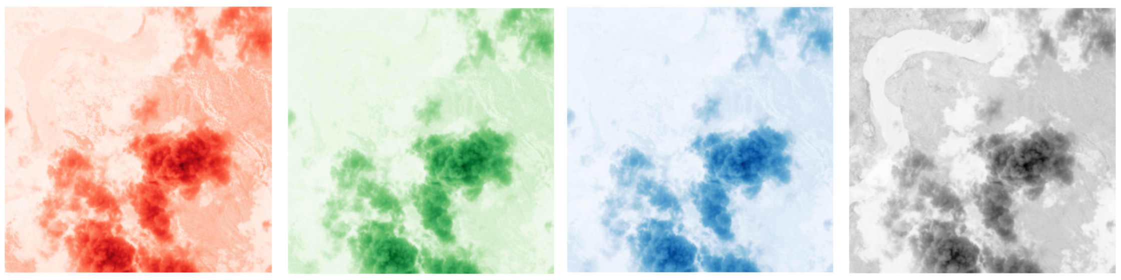


Figure 3.2: Same image in the R, G, B and NIR bands

By combining the RGB components of the images it is also possible to recover the original picture:

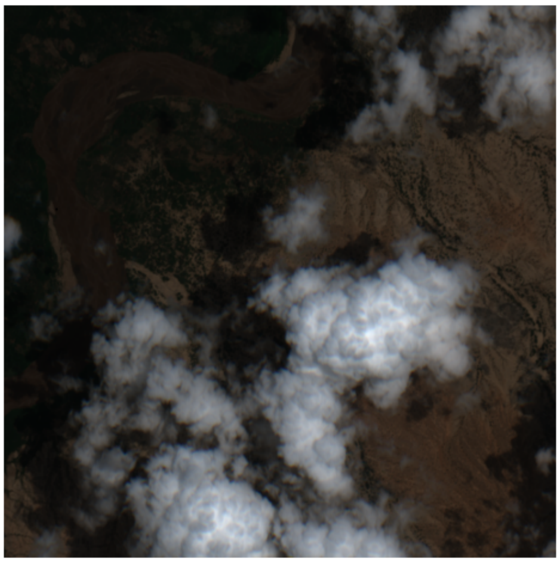


Figure 3.3: Real image

and the corresponding ground truth label is showed in Figure 3.4

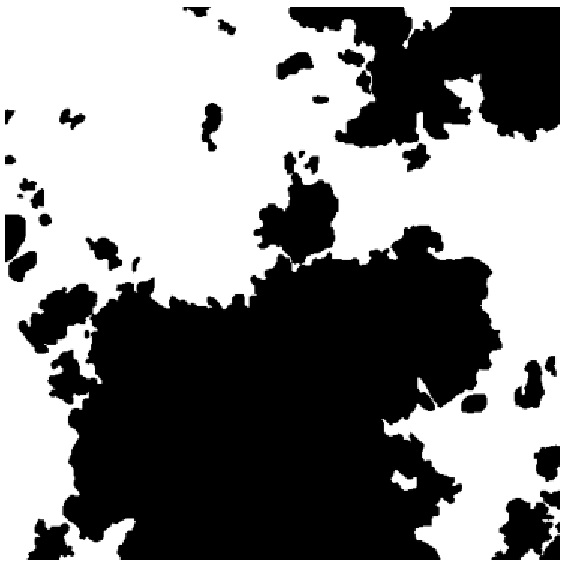


Figure 3.4: Ground truth label

In order to perform properly training and validation, the training chips are randomly split into training set and validation set, with a proportion of 2:1. In this way, the validation step can exploit unseen labelled data to generate a model that is more accurate in making predictions on the test dataset.

Also, the training, validation and test sets balancing are studied to investigate the model's generalization capabilities. Working with an unbalanced dataset, with some classes over-represented with respect to others, can lead to a biased learning because the more dominant classes become the ones having the more accurate predictions. This is a real issue in semantic segmentation tasks because they require accurate pixel-level classification across all the possible classes, so a well-balanced dataset is necessary to ensure the robustness of the model.

Figures 3.5, 3.6 and 3.7 show the cloud coverage distribution histograms used to qualitatively assess the class balance in the training, validation and test datasets. The values of the variance, standard deviation and min_max_ratio $=\frac{min}{max}$ of the cloud coverage distributions are also provided.

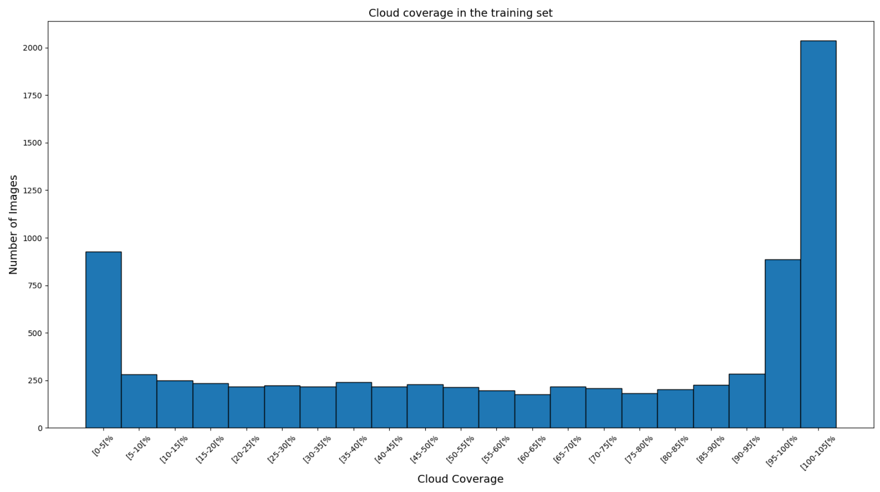


Figure 3.5: Cloud coverage distribution in the training set

- VARIANCE: 178833.202

- STANDARD DEVIATION: 422.887

- min_max_ratio : 0.0867

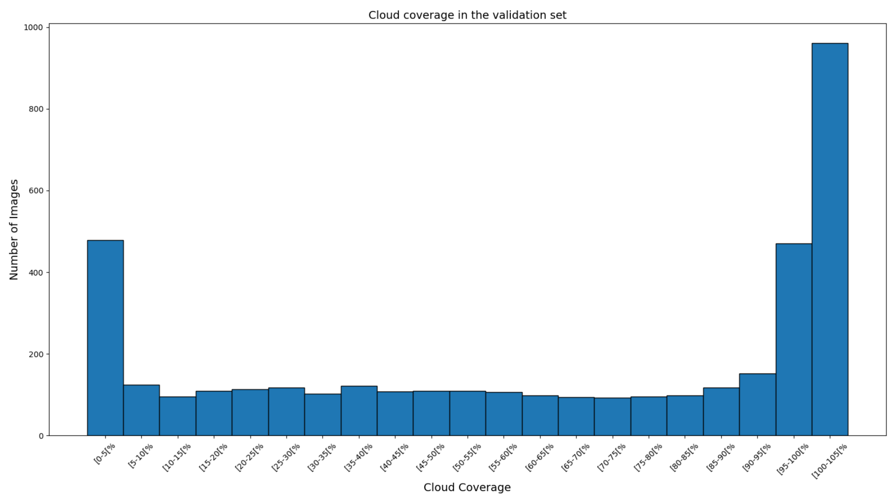


Figure 3.6: Cloud coverage distribution in the validation set

- VARIANCE: 41743.855
- STANDARD DEVIATION: 204.313
- min__max__ratio: 0.097

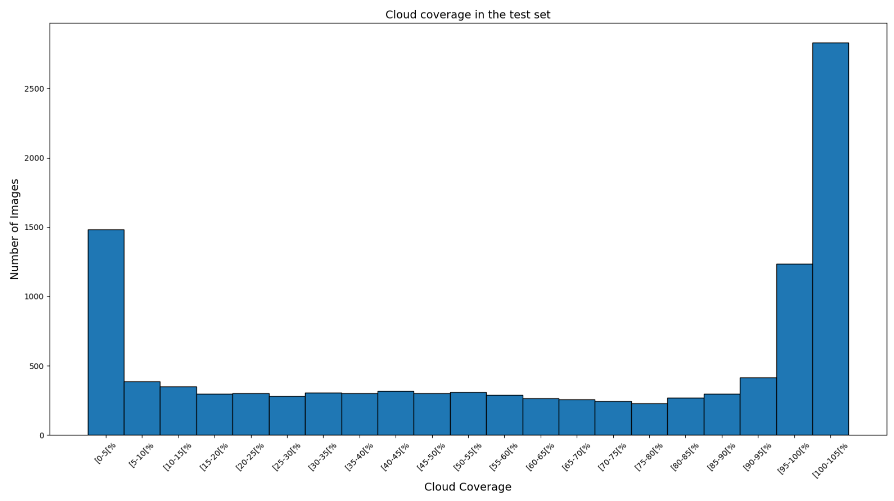


Figure 3.7: Cloud coverage distribution in the test set

- VARIANCE: 365440.694
- STANDARD DEVIATION: 604.517
- min__max__ratio: 0.080

All the dataset are characterized by a large variety of possible situations, ranging from total absence of clouds to total coverage. By looking at the values of the variance and standard deviation it is possible to understand that there is a high variability in the different cases' distributions. The low value of max_min_ratio, instead, means that the class with the minimum count is only 8-9% of the class with the maximum count.

The coverage distributions across the training, validation and test sets are quite similar, suggesting that the problem is balanced. The uniformity of distribution is a prerogative of any model, because it ensures that the system will be exposed to representative samples during all the phases of the task. In this way the risk of bias is reduced and the generalization capabilities increase.

The feature data are handled through a custom designed dataset class. It is called CloudDataset and it can retrieve the images, convert them into tensors, combine all the four provided bands, and divide them into batches to feed the model. The CloudDataset class exploits the PyTorch's "Dataset" and "DataLoader" classes, which are respectively used to create custom methods and to handle data loading in parallel.

The training images are organized into a "Pandas" dataframe to manage them easily. For each chip id, the direct paths to reach the 4 bands images and the ground truth label are provided, so that by using the "rasterio" library it is possible to directly access them.

3.3 Model implementation

The definition of the CloudDatastet class marks the beginning of the model implementation. The aim of the work is to determine the cloud coverage in satellite images , and for this purpose, semantic segmentation is the best approach to proceed. A possible deep neural network architecture for semantic segmentation is represented by the U-Net, whose structure was already discussed in section 2.5.2. However, a novel solution is implemented in this thesis aiming at reducing the complexity on board the satellites.

The design is based on a hybrid attention-recursive mechanism that, by working one line at a time, is able to reduce memory usage and latency with respect to traditional architectures based on 2D image processing. The sequence of rows is handled by the Mamba block, as explained in section 2.6.2, while the columns are processed through a more conventional U-Net exploiting the 1D convolutional layer.

A graphical representation of the model is showed in Figure 3.8:

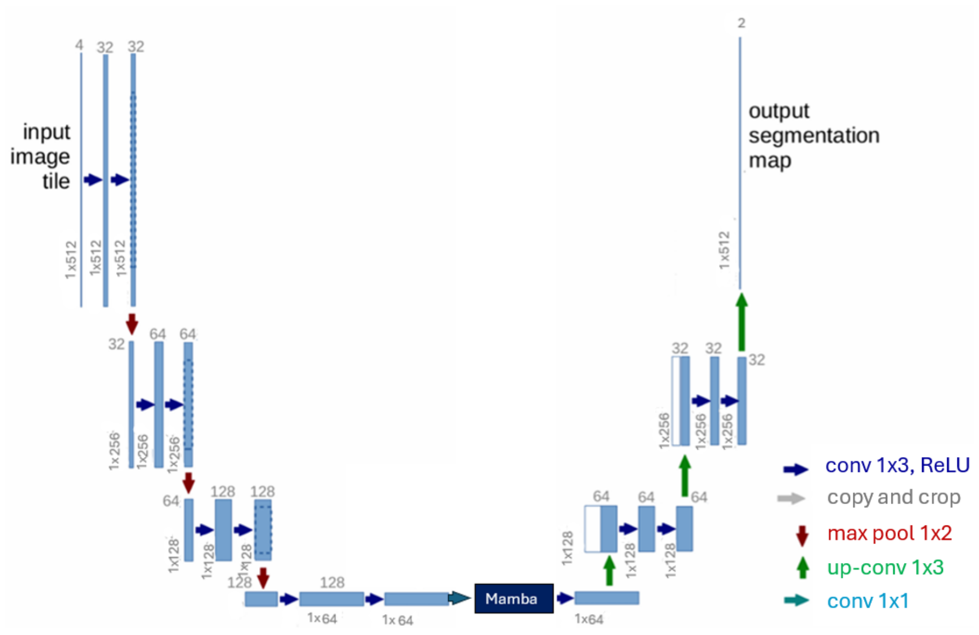


Figure 3.8: Neural network model

The architecture consists of a contraction path, followed by a Mamba layer and a final expansive path. For the training step, all the rows of the images are considered available at the same time, so that the operations can be executed independently and in parallel. At the input of the U-Net encoder there is always a tensor with dimension [batch size, channels, height, width], which has to be converted into a tensor with dimension [batch*height, channels, width]. Since images have a dimension 512x512, with 4 channels and batch_size = 16, the first layer of the U-Net encoder receives an input tensor with dimension [8192, 4, 512].
Inside the contraction path, 1D convolutional layers with kernel_size = 3, zero-padding and ReLU activation function are responsible for the doubling of the extracted features, while max pooling layers for the halving of the images' width. Dropout layers are also introduced to reduce the probability of overfitting.

The Mamba layer requires an input tensor with dimension [batch*width, height, features]. In this way it is able to process each pixel of each column in parallel. The Mamba code used in the architecture is taken from the open-source GitHub repository "https://github.com/state-spaces/mamba" [2].

Finally, the shape of the tensor is reset to [batch*height, channels, width], to be compatible with the U-Net decoder's layers.
In the expansion path, "ConvTranspose1d" layers are used to double the width of the image, while 1d convolutional layers are used to half the numbers of features. A final "Conv1d" layer is responsible for the output of images whose pixel values represent the final model's prediction. The predicted values can only be 0, if the pixel is classified as a

background pixel, or 1 if it is classified as a cloud pixel.

Differently from the typical U-Net architecture, the maximum number of features extracted from the implemented model is limited to 128. This choice is driven by two key considerations:

- Prevention of overfitting: when using a non extremely wide dataset, extracting an excessive amount of features can lead to overfitting. In such cases, the system may over-adapt to the training data, capturing noise or irrelevant patterns, and lose the generalization capabilities.

- GPU memory constraints: working with a large number of features significantly increases the demand for computational resources especially in terms of memory. By limiting the number of features to 128, the model can keep outstanding prediction capabilities within the memory limits of the available GPU.

A summary of the model implementation with the tensors' shape layer by layer is provided in Tables 3.2, 3.3 and 3.4:

Table 3.2: U-Net encoder summary

Layer	**Output shape**
	[batch*height, features, width]
Input layer	[8192, 4, 512]
Conv1d	[8192, 32, 512]
Conv1d	[8192, 32, 512]
MaxPool1d	[8192, 32, 256]
Conv1d	[8192, 64, 256]
Conv1d	[8192, 64, 256]
MaxPool1d	[8192, 64, 128]
Conv1d	[8192, 128, 128]
Conv1d	[8192, 128, 128]
MaxPool1d	[8192, 128, 64]

Table 3.3: Mamba layer summary

Layer	**Output shape**
	[batch*width, height, features]
Mamba	[1024, 512, 128]

Table 3.4: U-Net decoder summary

Layer	**Output shape** [**batch*height, features, width**]
Input layer	[8192, 128, 64]
ConvTranspose1d	[8192, 64, 128]
torch.cat	[8192, 128, 128]
Conv1d	[8192, 64, 128]
Conv1d	[8192, 64, 128]
ConvTranspose1d	[8192, 32, 256]
torch.cat	[8192, 64, 256]
Conv1d	[8192, 32, 256]
Conv1d	[8192, 32, 256]
Conv1d	[8192, 2, 512]

To stabilize and accelerate the training process, while reducing the risk of overfitting, batch normalization layers are added after all the convolutional layers of the encoder's architecture.

The implemented code is based on the PyTorch Lightning architecture. It is a PyTorch framework that allows for the design of the model, of the loss function, of the optimizer, and of the training and validation steps following a modular design. Thanks to its classes LightningModule e Trainer, it is possible to exploit advanced functionalities like checkpointing, early stopping and GPU parallelization while keeping the code organized and simple.

The LightningModule organizes the code into several sections:

- Initialization
- Train loop
- Validation loop
- Test loop
- Prediction loop
- Optimizers and LR Schedulers

All the sections are managed by the Lightning Trainer [18]. It not only handles training, but also:

- enables and disables the gradients (more details on the gradient are provided in section 3.3.3)
- runs train, validation and test DataLoaders

- calls the callback functions properly
- manages the data saving on the correct devices.

3.3.1 Train and validation loops

Pretrained weights are used as the starting point of the training step. They have been previously trained on a large dataset called ImageNet so that the model training can be faster and more performing. During training, they are optimized to improve the accuracy of the model.

The model training requires the definition of some parameters like the batch size and the epochs [28].

The **batch size** is the model hyperparameter that defines the number of samples to be processed before the update of the model's parameters. They are randomly selected from the training dataset, and represent the inputs used to minimize the loss function during the backward pass. The choice of the proper batch size is problem-dependent and can strongly influence the quality of the predictions. As a general distinction:

- Large batch size: the model achieves better approximation, faster convergence and higher computational efficiency on GPU.
- Small batch size: the model approximations are noisy, convergence is slower, and can escape the real minima of the loss function.

A good solution can be to choose a batch size close to the GPU memory limit, and adjust it during training.

The **epochs** are instead the hyperparameter representing the number of iterations necessary to see all the samples in the training set at least once. The number of epochs is typically large, so that the model can learn from the inputs as much as possible to minimize the loss function, and so to improve the accuracy of the predictions.

For this thesis work, the chosen batch size is equal to 16, which is the maximum size that the GPU memory can support.
As for the number of epochs, the algorithm was at first trained using the "EarlyStopping" callback provided by the PyTorch Lightning package. In this way the number of iterations is not statically defined, but it depends on the behavior of a metric under test. The monitored function is called IoU, and it is discussed more in the details in Section 3.4.
The "EarlyStopping" callback takes the parameter "patience" as an input. It represents the maximum number of consecutive iterations with no improvement in the max IoU that the system can perform. If the value of patience exceeds without the discovery of a new maximum, the training can be considered completed. Its value is set to 12 in the implemented code.

3.3.2 Test and prediction loops

The test stage is fundamental to verify the capabilities of the model in generating predictions on unseen data. This stage requires the test dataset preparation, the model loading, the inference and the predictions saving. As for the training loop, images are organized into dataframes mapping each chip ID to the corresponding links to reach the TIF images.

The pre-trained model is loaded using the weights saved during training, then images are processed in batches to improve the computing efficiency. A DataLoader object is used to manage the batches and transfer them to the GPU to generate predictions about cloud coverage.

Predictions are converted into binary masks and saved as TIF files, so that it is easier to compare them to the ground truth labels during the model's performance evaluation stage.

3.3.3 Optimizers and LR schedulers

Optimization consists in reducing the difference between the prediction output and the ground truth label, by minimizing a loss or cost function. Finding the minimum of a function that depends on a large amount of variables can be difficult, for this reason the iterative approach of the Gradient descent algorithm is used.
The aim of the algorithm is to iteratively update the initial weights $\mathbf{w}$ by a quantity $\Delta\mathbf{w}$ so that the cost C is reduced:

$$C(\mathbf{w} + \Delta\mathbf{w}) < C(\mathbf{w}) \tag{3.1}$$

By forcing $\Delta\mathbf{w}$ to be negative, and by manipulating equation 3.1 through Taylor expansion, the steps to be performed iteratively to reach optimization become:

- computation of the cost function gradient using the current weights
- update of the parameters according to

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\Delta_w C \tag{3.2}$$

where $\eta > 0$ is a scalar value called **learning rate**. It controls the update speed, and typically, a high η can be faster but may fail to converge, while a low η can be slower and more complex but more performing.

Since using all the training data to compute the cost function is computationally expensive, the gradient is computed on batches, as explained in section 3.3.1. This approach is called "Stochastic Gradient Descent (SGD)".

A popular improvement to the SGD is represented by the Adaptive Moment Estimation (Adam) optimizer, which speeds up convergence by exploiting the "exponentially weighted average" of the gradients.

For the purpose of this work, a binary Cross-Entropy (CE) cost function is used. The equation defining its behavior is reported in 3.3,

$$\text{CE} = -\sum_{i=1}^{C=2} y_i log(s_i) = -(y_1 log(s_1) + (1 - y_1) log(1 - s1)) \tag{3.3}$$

where y is the label, s the score, and C the number of classes, which is equal to 2 because the predicted outputs can only be either 0 or 1.
The Adam optimizer is used together with $\eta = self.learning\_rate$, which dynamically adapts to the cost function gradients to improve the efficiency and the stability of the model.

3.4 Performance metrics

Performance metrics are fundamental tools to evaluate the quality of the model. An insight on the evaluation metrics used in this work is provided in the following.

3.4.1 Training performance metrics

The best performance metric in a semantic segmentation task is represented by the IoU, also known as Jaccard index [24]. A semantic segmentation problem can be split into two consecutive steps, which are:

1. object localization
2. class assignment.

The first one consists in the definition of a bounding box around the object inside the image, which has to be compared to the bounding box in the ground truth label in order to evaluate the model's performance.
Calling A and B respectively the bounding boxes in the ground truth and in the prediction images, the IoU is computed as:

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} = \frac{|I|}{|U|} \tag{3.4}$$

and it measures the ratio between the overlapping area in the bounding boxes and the total area occupied by them. It can only assume values between 0 and 1, and the higher the score, the closer the intersection is to the union, which implies a higher match between the actual truth and the prediction. This is the reason why it is the most performing metric for model evaluation in semantic segmentation tasks.

IoU is the quality assessment criteria used during training, as already introduced in Section 3.3.1

3.4.2 Testing performance metrics

Performing the evaluation of the model also in the test step is a fundamental way to verify if the system can make accurate predictions on unseen data, or if it shows poor generalization capabilities needing for an improvement. Together with the IoU, some other metrics can be exploited for this task.

When looking at the predicted images, a pixel classified as a "Positive or 1" stands for detected cloud, otherwise, a pixel classified as a "Negative or 0" stands for background. Correctly classified pixels are referred to as *True Positive (TP)* and *True Negative (TN)*. Miss-classified pixels, instead, are referred to as *False Positive (FP)*, if the true label is Negative, but the model has predicted a Positive, and *False Negative (FN)*, if a Positive pixel is erroneously classified as Negative.
The graphical representation of the pixel classification is called "Confusion Matrix", and an example is shown in Figure 3.9:

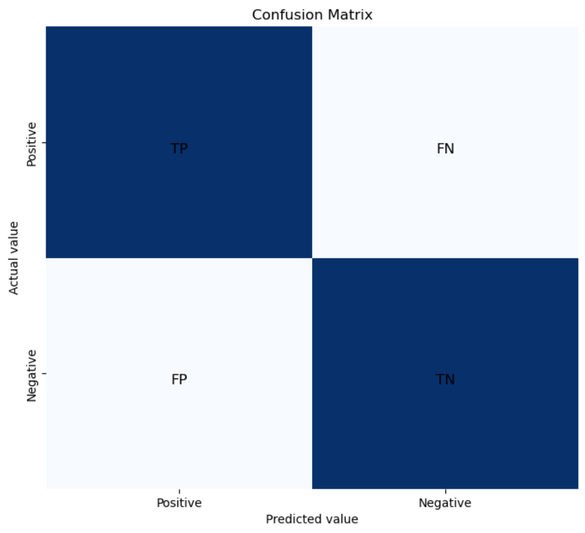


Figure 3.9: Confusion matrix

Starting from the confusion matrix, it is possible to evaluate some of the key performance metrics [14]:

- **Precision**: it measures the amount of positive predictions with respect to the total amount of positive predictions in order to determine if the cost of the FP predictions is too high.

$$Precision = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.5}$$

- **Recall/Sensitivity**: it measures the amount of positive predictions with respect to all the actual positive elements, so it gives an idea about the cost of the FN predictions.

$$Recall = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.6}$$

- **Accuracy**: it measures the amount of correct predictions with respect to all the predictions of the model

$$Accuracy = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{3.7}$$

- **F1 Score**: it measures a harmonic mean of precision and recall. It is a good measure to use when there is the need to balance Recall and Precision in case of an uneven class distribution.

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.8}$$

- **True Positive Rate (TPR)**: it measures the portions of positive predictions that are correctly classified by the model.

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.9}$$

- **False Positive Rate (FPR)**: it measures the portions of positive predictions that are wrongly classified by the model.

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{3.10}$$

- **True Negative Rate (TNR)**: it measures the portions of negative predictions that are correctly classified by the model.

$$\text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{3.11}$$

- **False Negative Rate (FNR)**: it measures the portions of negative predictions that are wrongly classified by the model.

$$\text{FNR} = \frac{\text{FN}}{\text{FN} + \text{TP}} \tag{3.12}$$

Chapter 4

Experimental results

The aim of this chapter is to analyze the performance on the training, validation and test sets obtained when using the model presented in Chapter 3. A comparison with other common models is also presented to better understand the potentialities of the new architecture in terms of quality of the predictions and memory usage.

4.1 Training performance

As previously mentioned in Section 3.3.1, selecting the appropriate batch size and number of epochs is a crucial step before initiating the training process. For the purposes of the experiment, a batch size of 16 images is used, while the number of epochs is not predefined since it is dynamically determined by the EarlyStop Callback. The following section illustrates the results obtained under these conditions, specifically analyzing how the model's accuracy, IoU, and loss metrics evolved throughout the training process.

The behavior of accuracy and IoU over the epochs during the model's training process is illustrated in Figures 4.1 and 4.2

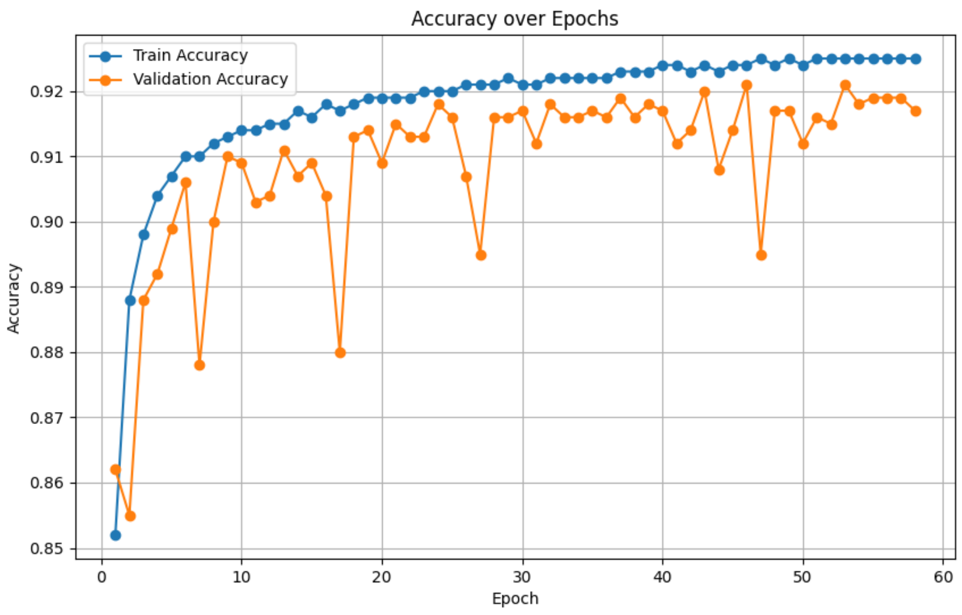


Figure 4.1: Accuracy

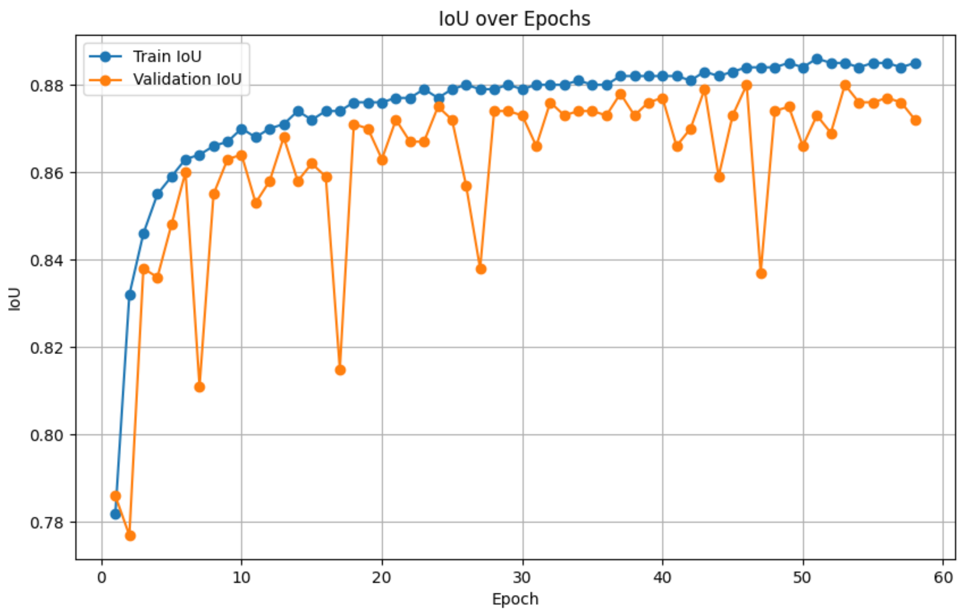


Figure 4.2: Intersection Over Union

The plots display an increasing trend for the training and validation curves of accuracy and IoU. Notably, the training curves exhibit a more stable growth compared to the validation curves. This is due to the presence of the batch normalization layers, which stabilize the training process while improving the generalization capability of the model. Moreover, it is common for the model to better adapt to the training data, potentially losing some accuracy on the validation data. The best values reached are respectively equal to 92,4%, 88,2%.

The losses trends are instead shown in Figure 4.3.

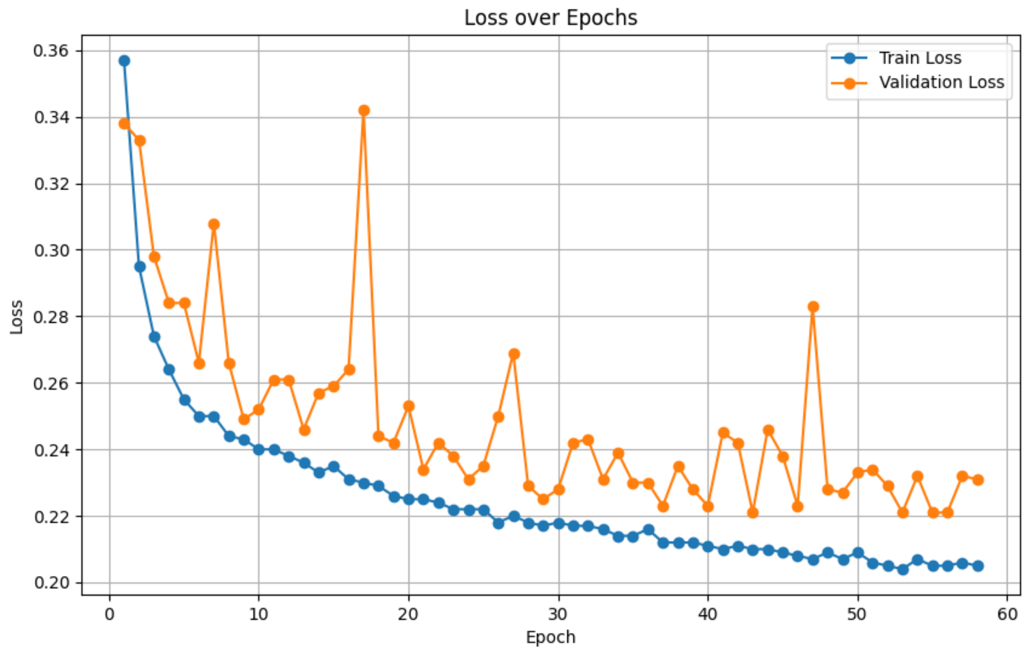


Figure 4.3: Losses

The losses decrease over time, suggesting that the model effectively minimizes them across epochs while learning the features within the images. The validation losses show more oscillations compared to the stable training trend, suggesting that the model may find it difficult to generalize consistently across the validation set. The lowest loss value reached is equal to 20,76%.

4.2 Test performance

After the training and validation steps, the prediction capabilities of the neural network have to be tested on a test set. The dataset must consist of unseen images, allowing for an objective assessment of the model's generalization performance. This is a critical step in evaluating whether the model can be successfully applied to new data in real-world scenarios.

The following section presents the results obtained on the test set by tracking key performance metrics such as accuracy, IoU, precision, recall, F1 score, TPR, and FPR. These metrics provide a comprehensive overview of the model's effectiveness, offering insights into whether further optimization or fine-tuning is necessary.

The quality of the predictions is evaluated pixel by pixel for each image. The number of correctly and incorrectly classified pixels is determined by computing the TP, TN, FP, and FN. These values are then used to calculate the main performance metrics as explained in Section 3.4.2.

The confusion matrix, built according to the pixel-wise evaluation, is presented in Figure 4.13 to visually show the distribution of the predictions across all the images.

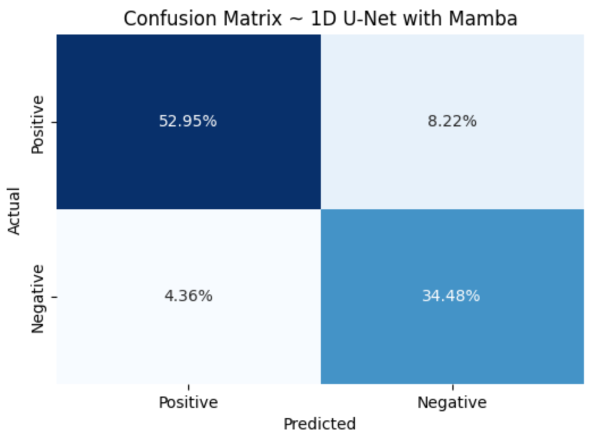


Figure 4.4: Confusion matrix

According to the confusion matrix, the model is able to correctly classify cloud pixels and background pixels with high accuracy. Most of the miss-classified pixels belong to the FN group, reaching a percentage of 8.22%, suggesting that the system is not always able to identify clouds.

A summary of the main performance metrics is provided in Table 4.1.

Table 4.1: Performance metrics

Metric	Value
Accuracy	91,11%
Precision	93,31%
Recall	92,06%
F1 Score	92,68%
IoU	86,37%
TPR	92,06%
FPR	10,39%
TNR	89,60%
FNR	7,94%

The following figures showcase some of the miss-classified chips, that achieved TP and TN values lower that 10%.

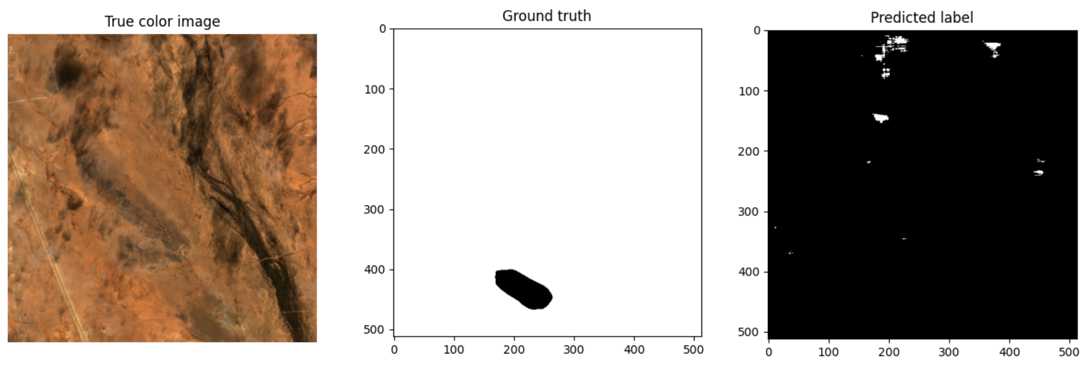


Figure 4.5: Miss-classified image

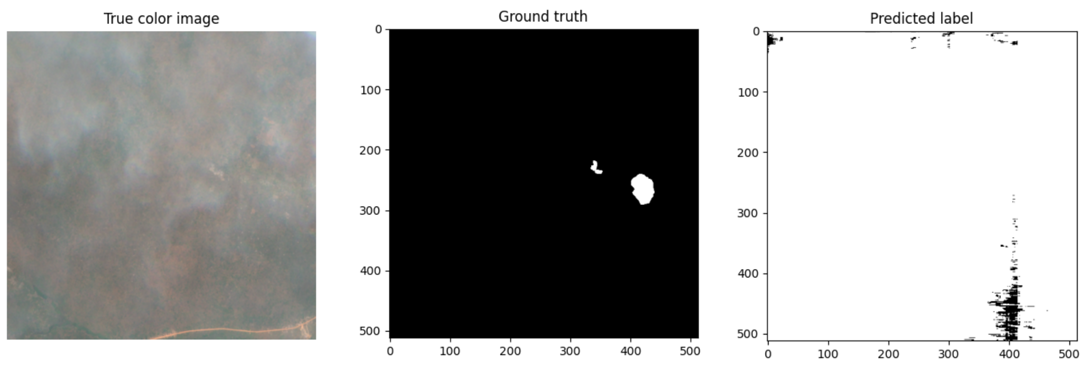


Figure 4.6: Miss-classified image

Figures 4.7 and 4.8, instead, show examples of perfectly classified chips.

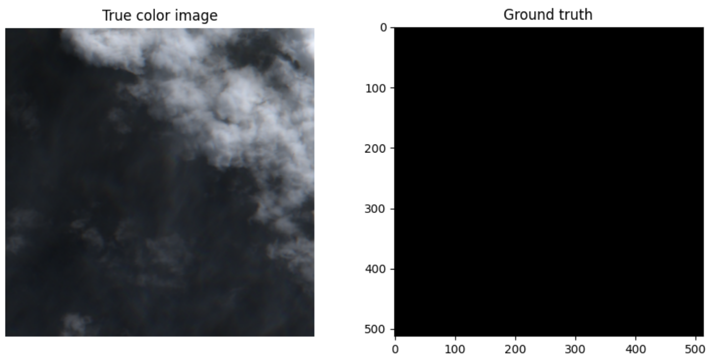


Figure 4.7: Perfectly classified image

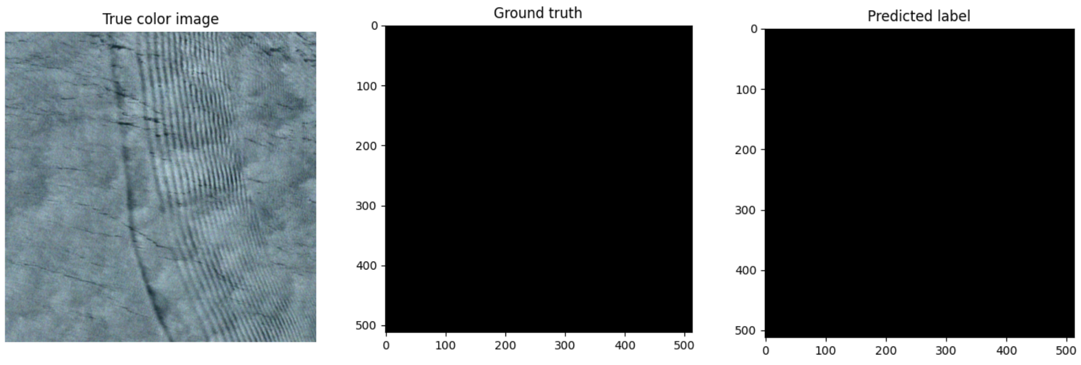


Figure 4.8: Perfectly classified image

4.3 Comparison with classical models

The aim of this thesis work is to design a neural network for the semantic segmentation of multi-spectral images on board the satellites. The model has to be optimized for low complexity to improve memory usage with respect to the conventional 2D methods, while preserving the quality of the predictions. The key element in this new architecture is represented by the hybrid recursive attention mechanism, which captures the local and global dependencies inside the images with lower computational cost.

The following section shows a comparative analysis of various neural network architectures to highlight the distinctive advantages and performance capabilities of the Mamba model.

4.3.1 UNet architecture with a ResNet34 encoder

A standard architecture for traditional 2D semantic segmentation is represented by the UNet. As explained in Section 2.5.2, this model is suitable for image recognition tasks that require deep architectures but with a limited training dataset.

Thanks to the multiple convolutional, pooling, and downsampling sequential steps of the encoder, the UNet is able to capture high and low-level features in the images. Then, through the sequential upsampling and convolutional layers of the decoder, it produces a label image containing the prediction for each pixel in the input image.

The first UNet architecture proposed for the comparison task presents a ResNet34 encoder pretarined with the ImageNet dataset and a base Unet implementation for the decoder. In this way, instead of training from scratch, the network performs a fine-tuning that can help in recognising simple features faster in the initial layers and more complex ones in the final layers.

The network architecture is shown in Figure 4.9

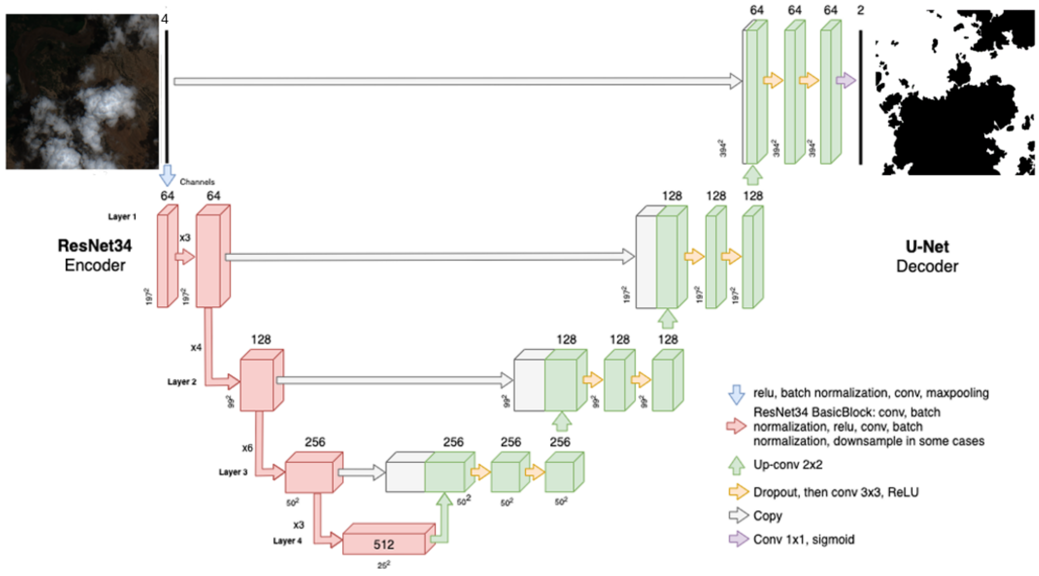


Figure 4.9: UNet with ResNet34 encoder architecture

ResNet34 is a deep CNNs made up of 34 layers, known for its capability of overcoming the vanishing gradient problem. This result is achieved thanks to the implementation of skip connections. This method consists in bypassing some of the levels in between to directly connect activations of subsequent layers. In this way blocks are created and stacked to generate resnets. A visual representation of a skip connection is provided in Figure 4.10

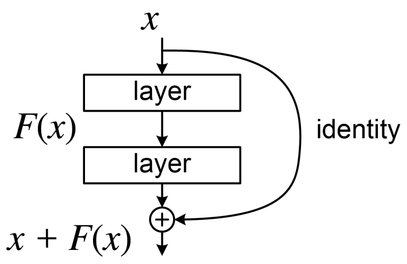


Figure 4.10: Skip connection scheme

The architecture of the encoder starts with a convolutional layer with 64 filters and kernel size 7x7, followed by a max-pooling layer with stride=2. Then blocks made up of a pooling layer and 2 convolutional layers are sequentially implemented. Finally an average pooling layer followed by a softmax activation function is located at the end.

The ResNet34 architecture is shown in Figure 4.11.

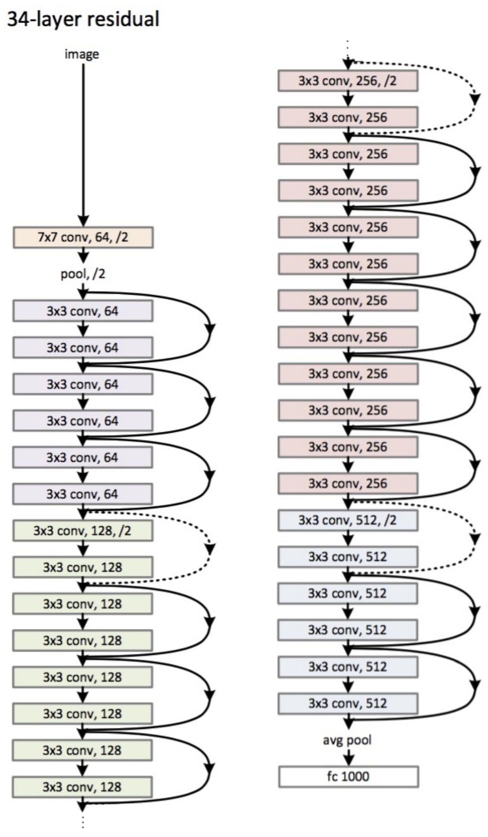


Figure 4.11: ResNet34 architecture

4.3.2 UNet

The second model used for the comparison is a traditional 1D UNet designed to extract at most 128 features at each level. This architecture provides a compromise between computational efficiency and performance while keeping the conventional encoder-decoder structure. The implemented architecture is shown in Figure 4.12.

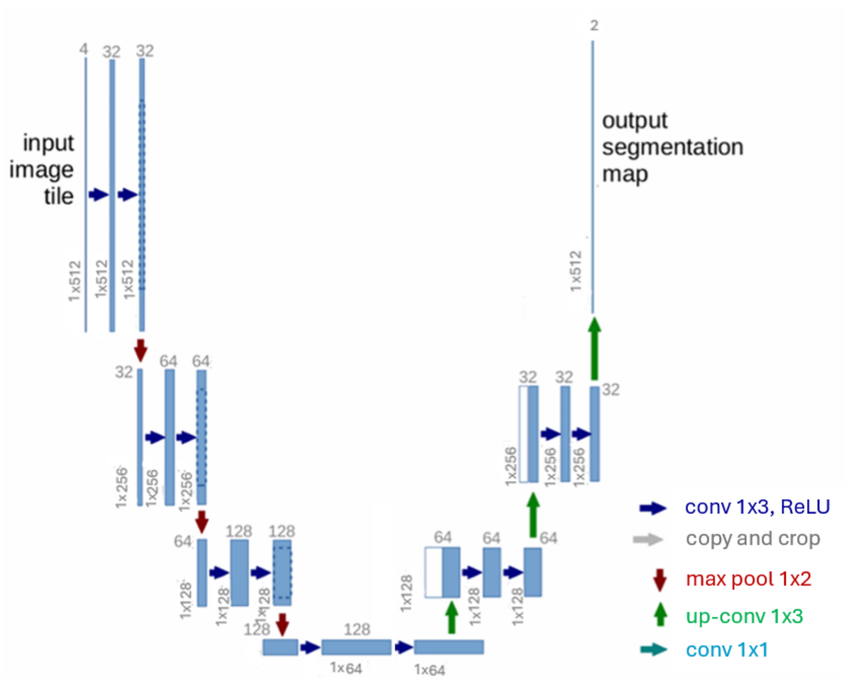


Figure 4.12: Implemented UNet architecture

The reduced dimension of the network makes it suitable for applications that require low computational cost, like on board processing. This version of the model is less complex than the one based on the ResNet34 encoder, making it less appropriate to perform semantic segmentation with complex images. This baseline model allows for the assess of the effectiveness of Mamba in enhancing performance while maintaining lower memory and computational requirements.

4.3.3 Performance comparison

The following section deals with the performance comparison of the three implemented models. The analysis outlines the key differences in the performance metrics evaluated on the same test dataset, highlighting the strengths and weaknesses of each approach.

Table 4.2 shows the accuracy, precision, recall, F1-score and IoU scores obtained for all the models.

Table 4.2: Performance metrics' comparison

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**	**IoU**
1D U-Net with Mamba	**91,11%**	**93,31%**	92,06%	**92,68%**	**86,37%**
2D U-Net with ResNet34	87,42%	92,39%	86,56%	89,38%	80,81%
1D U-Net	89,87%	90,99%	**92,63%**	91,80%	84,84%

The comparison between the three different models shows that the 1D U-Net with Mamba reaches the best performance across most of the metrics, with outstanding scores for Precision, F1-Score and IoU. These results confirm the superiority of the novel model in balancing the recognition of both positive and negative pixels, with respect to the other traditional models.

Looking at the performance of the 2D U-Net with ResNet34, it can be noticed that it has comparable scores in terms of Precision, but it suffers from low values of Recall and IoU, suggesting that it better predicts positive pixels, lacking of classification completeness.

Finally, the standard 1D U-Net, shows high values of Recall and F1-Score, but lower IoU compared to the Mamba model. This suggests that, despite the high capabilities of correctly classifying pixels, the generalization performance are slightly lower than the Mamba ones.

Table 4.3 shows instead the TPR, FPR, TNR, FNR scores obtained for all the models.

Table 4.3: Performance rates' comparison

Model	**TPR**	**FPR**	**TNR**	**FNR**
1D U-Net with Mamba	92,06%	10,39%	89,60%	7,94%
2D U-Net with ResNet34	86,56%	11,22%	88,77%	13,44%
1D U-Net	92.63%	14,45%	85,51%	7,37%

The 1D U-Net with Mamba shows a high value of TPR and a low FNR, suggesting that it is the best model in correctly classifying positive pixels while keeping the FNR relatively low. The low FPR also highlights that the system can minimize the false positive predictions preferring TP detections.

The 2D U-Net with ResNet34, instead, shows a lower capability of correctly predicting the positive cases. Values are not too far from the ones reached with the Mamba architecture, however the higher FPR makes it less reliable overall.

Finally, the standard 1D U-Net presents the best performance in detecting positives, while slightly struggling in distinguishing between false positives and negatives, as suggested by the scores of FPR and TPR.

A comparison of the confusion matrices is shown in the following to better visualize the distributions of TP, TN, FP, FN across all the images for all the models.

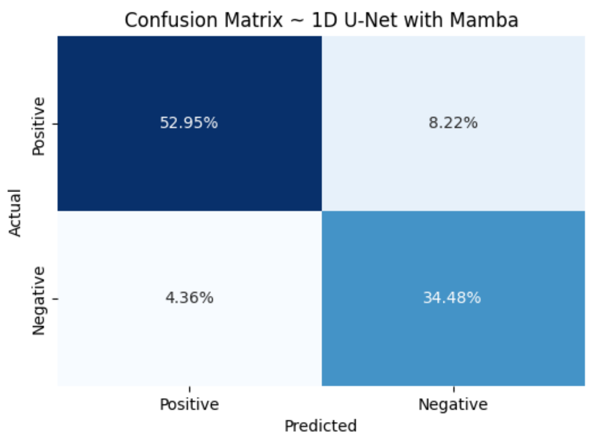


Figure 4.13: Confusion matrix

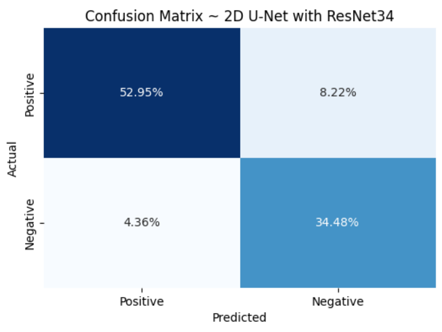


Figure 4.14: Confusion matrix of the 2D U-Net with ResNet34

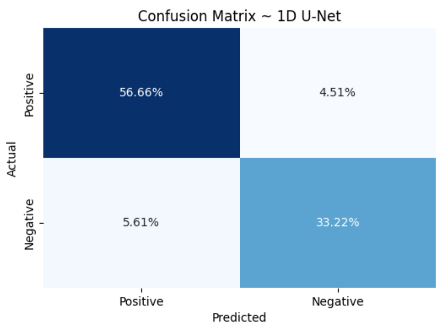


Figure 4.15: Confusion matrix of the 1D U-Net

Table 4.4 contains the number of parameters and the corresponding memory usage for the different models. In this way, a direct comparison between the different implemented solutions is provided to highlight the differences in memory utilization.

Table 4.4: Memory usage for parameters storage comparison

Model	**Memory Usage**	**Number of parameters**
1D U-Net with Mamba	1.121 MB	279 K
2D U-Net with ResNet34	97.76 MB	24.4 M
1D U-Net	0.849 MB	212 K

The 2D U-Net with ResNet34 encoder shows the highest number of parameters and so the largest memory usage among all the models. This is an expected result since 2D convolutional layers require a large amount of parameters to capture the spatial features.

In contrast, the model with the lowest memory occupation is instead the simple 1D U-Net, which also requires the lowest amount of parameters. The Mamba variant, despite consuming slightly more memory, can achieve better performance compared to the simple 1D U-Net, achieving the best trade-off between performance and memory usage for a possible on-board implementation.

Finally, Table 4.5 contains the memory usage values for the different models when only one row of the test image is given as an input of the test loop. In this way, the line-by-line processing capabilities of the models are compared with the aim of verifying that the novel architecture requires low memory with respect to the conventional 2D methods.

Table 4.5: Memory usage comparison

Model	**CPU Memory Usage**	**GPU Memory Usage**
1D U-Net with Mamba	111.80 MB	8.13 MB
2D U-Net with ResNet34	283.43 MB	0.12 MB
1D U-Net	138.07 MB	0.0 MB

The results show that the novel architecture is effectively able to optimize memory usage compared to the 2D U-Net architecture with ResNet34 encoder and to the 1D U-Net without the Mamba layer. This demonstrates the capabilities of the hybrid attention-recursive mechanism in saving memory while modelling long-term dependencies.

Chapter 5

Conclusions and future work

5.1 Conclusions

The objective of this thesis was to develop a neural network model for cloud semantic segmentation with low computational complexity, suitable for the implementation on board satellites. The dataset used for training, validation, and testing consisted of multi-spectral images comprises RGB and NIR components. The network architecture is based on a hybrid recursive-attention mechanism, leveraging the novel Mamba architecture, which enables real-time image processing. This architecture, through its ability to model dependencies within sequences of complex data while maintaining system simplicity, offers significant advantages in terms of memory efficiency and reduced latency compared to traditional 2D approaches. Specifically, the Mamba layer handles the along-track direction of the images, while the U-Net's conv1D layers process the columns.

The outstanding capabilities of deep learning in the field of semantic segmentation have made it the best choice for solving the problem of clouds identification in satellite images. Since the design of a neural network model requires the integration of several components and layers, a general overview of what deep learning is and what its main building structures are has been provided. Some important notions about CNNs and SSM have been included to contextualize the implementation choices.
This thesis' section realization has been carried out thanks to the high availability of conference papers, articles and scientific journals, which investigate deeply the potentialities of deep learning for object recognition.

A general background about satellite imagery has been also provided to highlight its importance in many applications for the Earth's monitoring. Details about the development of the techniques to capture and process satellites data have been discussed in order to understand how to manage the different bands images. This information was sourced from the official websites of governmental agencies overseeing satellite activities.

After the theoretical introduction, the neural network model has been designed. The architecture has been realized through the combination of the U-Net model with the Mamba framework. The U-Net structure has been modified to adapt it to the final application and to the available resources. The training, validation and test loops have been carried out using a set of labelled images coming from a Sentinel2 mission.

In order to evaluate the performance of the network, different metrics have been computed both during the training and the test steps. The model has reached an accuracy value of 92,4% and a IoU value of 88,2% during training, and of respectively 91,11% and 86,37% during testing. The confusion matrix has also been plotted to provide a visual representation of the prediction capabilities of the network.
Some mislabelled images have also been shown to visually check the entity of the miss-classification.

Finally, in order to better highlight the outstanding capabilities of the Mamba layer in sequence modelling, a comparison with other models has been provided. A typical 2D U-NET with a ResNet34 encoder and a simpler 1D U-Net have been trained, validated and tested to compare the final prediction capabilities and memory usage with the novel implemented architecture.

In conclusion, this study has demonstrated that it is possible to achieve high quality performance in a real-time cloud segmentation task by implementing a hybrid recursive-attention mechanism that combines the segmentation capability of the classical U-Net with the innovative Mamba framework. Latency and memory usage are also optimized, making the novel architecture suitable for on board implementation.

5.2 Future work

Even though the results of this work have addressed the proposed challenge, many improvements in several potential directions can be further investigated, and some of them are proposed in the following:

- Improve the datatset by applying augmentation techniques to let the model generalize more on new data and reduce overfitting.
- Develop a deeper network capable of extracting a larger number of features from the images.
- Extend the model to the other spectral bands to potentially improve the prediction quality in challenging conditions, like at night.
- Investigate new kinds of optimizations for different hardware architectures to make the model more performing when used with specialized satellite processors.
- Test the model performance through K-Fold Cross Validation. The dataset can be split into folds according to the location captured by the satellite sensors, and the model can be trained several times. Finally the accuracy can be computed for each of the folds to obtain more accurate evaluation of the model.

Appendix A

Python code

The following section contains the implementation of the three models. The code development was inspired by an existing codebase, available at [29], but many modification are present to adapt it to the thesis requirements.

1D U-NET WITH MAMBA

Listing A.1: 1D U-Net with Mamba model

```
class UNetEncoder(nn.Module):
    def __init__(self):
        super(UNetEncoder, self).__init__()
        self.conv1 = nn.Conv1d(4, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(32)
        self.conv2 = nn.Conv1d(32, 32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(32)
        self.pool = nn.MaxPool1d(2)

        self.conv3 = nn.Conv1d(32, 64, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm1d(64)
        self.conv4 = nn.Conv1d(64, 64, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm1d(64)

        self.conv5 = nn.Conv1d(64, 128, kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm1d(128)
        self.dropout = nn.Dropout(0.2)
        self.conv6 = nn.Conv1d(128, 128, kernel_size=3, padding=1)
        self.bn6 = nn.BatchNorm1d(128)

    def forward(self, x):
        # Contraction path
        c1 = F.relu(self.bn1(self.conv1(x)))
        c1 = F.relu(self.bn2(self.conv2(c1)))
        p1 = self.pool(c1)

        c2 = F.relu(self.bn3(self.conv3(p1)))
        #c2 = self.dropout(c2)
        c2 = F.relu(self.bn4(self.conv4(c2)))
        p2 = self.pool(c2)

        c3 = F.relu(self.bn5(self.conv5(p2)))
        #c3 = self.dropout(c3)
        c3 = F.relu(self.bn6(self.conv6(c3)))

        return c3, c2, c1

class UNetDecoder(nn.Module):
    def __init__(self):
        super(UNetDecoder, self).__init__()

        self.conv_trans1 = nn.ConvTranspose1d(128, 64, kernel_size
            =2, stride=2, padding=0)
        self.conv7 = nn.Conv1d(128, 64, kernel_size=3, padding=1)
        self.conv8 = nn.Conv1d(64, 64, kernel_size=3, padding=1)
        self.dropout = nn.Dropout(0.1)

        self.conv_trans2 = nn.ConvTranspose1d(64, 32, kernel_size
            =2, stride=2, padding=0)
        self.conv9 = nn.Conv1d(64, 32, kernel_size=3, padding=1)
        self.conv10 = nn.Conv1d(32, 32, kernel_size=3, padding=1)

        self.final_conv = nn.Conv1d(32, 2, kernel_size=1)

    def forward(self, c3, c2, c1):
        # Expansion path
        u4 = self.conv_trans1(c3)
        u4 = torch.cat((u4, c2), dim=1)
        c4 = F.relu(self.conv7(u4))
        c4 = F.relu(self.conv8(c4))

        u5 = self.conv_trans2(c4)
        u5 = torch.cat((u5, c1), dim=1)
        c5 = F.relu(self.conv9(u5))
        c5 = F.relu(self.conv10(c5))

        output = self.final_conv(c5)

        return output


# Define CloudSegmentationModel
class CloudSegmentationModel(nn.Module):
    def __init__(self):
        super(CloudSegmentationModel, self).__init__()
        self.unet_encoder = UNetEncoder()
        self.mamba = Mamba(
            d_model=128,  # Model dimension d_model
            d_state=16,  # SSM state expansion factor
            d_conv=4,  # Local convolution width
            expand=2  # Block expansion factor
        ).to("cuda")
        self.unet_decoder = UNetDecoder()


    def forward(self, x):
        batch_size, channels, height, width = x.size()

        # Permute to (batch, height, channels, width)
        x = x.permute(0, 2, 1, 3).contiguous()    #####
        # Reshape to (batch*height, channels, width)
        x = x.view(batch_size * height, channels, width)

        # Pass through UNet encoder
        c3, c2, c1 = self.unet_encoder(x)
        (batch_height, features, width) = c3.size()

        # Reshape to (batch, height, features, width)
        c3 = c3.view(batch_size, height, features, width)
        # Permute to (batch, width, height, features)
        c3 = c3.permute(0, 3, 1, 2).contiguous()
        # Reshape to (batch*width, height, features)
        c3 = c3.view(batch_size * width, height, 128)

        # Pass through Mamba
        c3 = self.mamba(c3)

        # Reshape to (batch, width, height, feautures)
        c3 = c3.view(batch_size, width, height, -1)
        # Permute to (batch, height, features, width)
        c3 = c3.permute(0, 2, 3, 1).contiguous()
        # Reshape to (batch*height, features, width)
        c3 = c3.view(batch_size * height, features, width)

        # Pass through UNet decoder
        output = self.unet_decoder(c3, c2, c1)

        # Reshape output to (batch, height, 1, width)
        output = output.view(batch_size, 512, 2, 512)
        # Permute to (batch, 1, height, width)
        output = output.permute(0, 2, 1, 3).contiguous()

        return output
```

1D U-NET

Listing A.2: 1D U-Net model

```
class UNetEncoder(nn.Module):
    def __init__(self):
        super(UNetEncoder, self).__init__()
        self.conv1 = nn.Conv1d(4, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv1d(32, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool1d(2)

        self.conv3 = nn.Conv1d(32, 64, kernel_size=3, padding=1)
        self.conv4 = nn.Conv1d(64, 64, kernel_size=3, padding=1)

        self.conv5 = nn.Conv1d(64, 128, kernel_size=3, padding=1)
        self.dropout = nn.Dropout(0.2)
        self.conv6 = nn.Conv1d(128, 128, kernel_size=3, padding=1)

    def forward(self, x):
        # Contraction path
        c1 = F.relu(self.conv1(x))
        c1 = F.relu(self.conv2(c1))
        p1 = self.pool(c1)

        c2 = F.relu(self.conv3(p1))
        c2 = F.relu(self.conv4(c2))
        p2 = self.pool(c2)

        c3 = F.relu(self.conv5(p2))
        c3 = self.dropout(c3)
        c3 = F.relu(self.conv6(c3))

        return c3, c2, c1


# Define UNetDecoder
class UNetDecoder(nn.Module):
    def __init__(self):
        super(UNetDecoder, self).__init__()

        self.conv_trans1 = nn.ConvTranspose1d(128, 64, kernel_size
            =2, stride=2, padding=0)
        self.conv7 = nn.Conv1d(128, 64, kernel_size=3, padding=1)
        self.conv8 = nn.Conv1d(64, 64, kernel_size=3, padding=1)
        self.dropout = nn.Dropout(0.1)

        self.conv_trans2 = nn.ConvTranspose1d(64, 32, kernel_size
            =2, stride=2, padding=0)
        self.conv9 = nn.Conv1d(64, 32, kernel_size=3, padding=1)
        self.conv10 = nn.Conv1d(32, 32, kernel_size=3, padding=1)
        self.final_conv = nn.Conv1d(32, 2, kernel_size=1)

    def forward(self, c3, c2, c1):
        # Expansion path
        u4 = self.conv_trans1(c3)
        u4 = torch.cat((u4, c2), dim=1)
        c4 = F.relu(self.conv7(u4))
        c4 = F.relu(self.conv8(c4))

        u5 = self.conv_trans2(c4)
        u5 = torch.cat((u5, c1), dim=1)
        c5 = F.relu(self.conv9(u5))
        c5 = F.relu(self.conv10(c5))

        output = self.final_conv(c5)

        return output

# Define CloudSegmentationModel with causal convolution instead of
   Mamba
class CloudSegmentationModel(nn.Module):

    def __init__(self):
        super(CloudSegmentationModel, self).__init__()
        self.unet_encoder = UNetEncoder()

        self.causal_conv = nn.Conv1d(in_channels=128, out_channels
           =128, kernel_size=3, padding=2, dilation=2)

        self.unet_decoder = UNetDecoder()

    def forward(self, x):

        batch_size, channels, height, width = x.size()

        # Permute to (batch, height, channels, width)
        x = x.permute(0, 2, 1, 3).contiguous()
        # Reshape to (batch*height, channels, width)
        x = x.view(batch_size * height, channels, width)

        # Pass through UNet encoder
        c3, c2, c1 = self.unet_encoder(x)

        # Pass through Causal Convolution
        c3 = self.causal_conv(c3)

        # Pass through UNet decoder
        output = self.unet_decoder(c3, c2, c1)
        # Reshape output to (batch, height, 2, width)
        output = output.view(batch_size, 512, 2, 512)
        # Permute to (batch, 1, height, width)
        output = output.permute(0, 2, 1, 3).contiguous()

        return output
```

Bibliography

[1] Ronan Collobert Koray Kavukcuoglu Clement Farabet Leon Bottou Iain Melvin Jason Weston Samy Bengio Johnny Mariethoz Adam Paszke, Soumith Chintala. Pytorch, 2024.

[2] Tri Dao Albert Gu. Mamba. https://github.com/state-spaces/mamba, 2024. Accessed: 27/07/2024.

[3] Tri Dao Albert Gu. Mamba: Linear-time sequence modeling with selective state spaces. pages 1–5, May 2024.

[4] Niki Parmar Jackob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Shazeer. Attention is all you need. pages 1–5, August 2023.

[5] James B. Campbell and Randolph H. Wynne. Introduction to remote sensing, fifth edition. pages 1–18. Guilford Press, 2011.

[6] Bharadwaj Chintalapati, Arthur Precht, Sougata Hanra, Rene Laufer, Marcus Liwicki, and Jens Eickhoff. Opportunities and challenges of on-board ai-based image recognition for small satellite earth observation missions. *Advances in Space Research*, 2024.

[7] Robin Cole. Satellite image deep learning, 2024.

[8] Ketan Doshi. Batch norm explained visually â how it works, and why neural networks need it, 2021.

[9] DRIVENDATA. On cloud n: Cloud cover detection challenge, 2024.

[10] ESA. Overview of sentinel-2 mission, 2024.

[11] IBM. What are convolutional neural networks?, 2020.

[12] IBM. What is machine learning (ml)?, 2020.

[13] Intel. What is a gpu?, 2024.

[14] Abhishek Jain. A comprehensive guide to performance metrics in machine learning, 2024.

[15] Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J. Inman. 1d convolutional neural networks and applications: A survey. *Mechanical Systems and Signal Processing*, 151:107398, 2021.

[16] Pablo Miralles, Kathiravan Thangavel, Antonio Fulvio Scannapieco, Nitya Jagadam, Prerna Baranwal, Bhavin Faldu, Ruchita Abhang, Sahil Bhatia, Sebastien Bonnart, Ishita Bhatnagar, Beenish Batul, Pallavi Prasad, HÃ©ctor Ortega-GonzÃ¡lez, Harrish Joseph, Harshal More, Sondes Morchedi, Aman Kumar Panda, Marco Zaccaria Di Fraia, Daniel Wischert, and Daria Stepanova. A critical review on the state-of-the-art and future prospects of machine learning for earth observation operations.

Advances in Space Research, 71(12):4959–4986, 2023.

[17] NVIDIA. Scheda grafica nvidia rtx 6000 ada generation, 2024.

[18] Creators of PyTorch Ligthnings. Trainer-pytorch lightning, 2024.

[19] Philipp Fischer Olaf Ronneberger and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. pages 1–4, May 2015.

[20] P. Ongsulee. Artificial intelligence, machine learning and deep learning. *15th International Conference on ICT and Knowledge Engineering(ICTKE)*, pages 1–6, 2017.

[21] Keiron OâShea and Ryan Nash. An introduction to convolutional neural networks. pages 1–4, December 2015.

[22] Samaya Madhavan Piyush Madan. An introduction to deep learning, 2020.

[23] @radiantearth. Sentinel-2 cloud cover segmentation dataset, 2024.

[24] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[25] run ai. Deep learnin for computer vision, 2024.

[26] Richard Szeliski. Computer vision: Algorithms and applications. pages 3–5. Springer, 2010.

[27] Codecademy Team. Deep learning workflow, 2024.

[28] Diego Valsesia. Signal, image and video processing and learning. Slide del Corso, 2023/2024. Politecnico di Torino.

[29] Katie Wetstone. How to use deep learning, pytorch lightning, and the planetary computer to predict cloud cover in satellite imagery. https://drivendata.co/blog/cloud-cover-benchmark/, 2021. Accessed: 27/07/2024.