Advanced Topics in Machine Learning and Neural Networks

Course project

Andrea Corvaglia, Francesco Pisu, Davide Serra, Donato Tiano

December 2025

1 Introduction

The goal of this project is to develop a web application capable of segmenting lungs in thoracic X-ray images. When functioning correctly, the app should allow the user to upload a thoracic X-ray image (in `.png` format) and receive as output the corresponding lung mask generated by a trained segmentation model.

The project is organized into the following main steps:

- Implementation and training of a lung segmentation model using Kaggle Notebooks and the *COVID-19 Radiography Database*;
- Creation of a dedicated `Lung Segmentation` directory to be uploaded to GitHub;
- Development of the `LungSegmentationApp.py` interface using the Streamlit framework;
- Deployment of the final web application.

This structure ensures a complete workflow, from data preparation and model training to deployment and practical use through an accessible web interface. To start, see this GitHub repo.

2 Segmentation[1]

Image segmentation is the process of dividing an image into meaningful regions. In practice, this means separating an image into parts that share similar characteristics such as color, intensity, shape, or anatomical structure; these characteristics are referred to as 'features'. Segmentation is a fundamental step in medical image analysis because it allows algorithms and clinicians to focus on the specific regions of interest within complex medical images. In healthcare applications, segmentation is essential for identifying and outlining organs, tissues, or pathological areas such as tumors or lesions. By automating this step, we reduce manual processing time and support clinicians in obtaining accurate measurements, visualizations, and disease monitoring tools. For example, segmentation can highlight abnormal tissue, help track cancer progression, or extract quantitative biomarkers from MRI, CT, or ultrasound scans. Semantic segmentation, a specific type of segmentation, assigns a class label to every pixel in the image. This makes it especially useful in medical projects where the goal is to detect and delineate well-defined structures, improving both the precision of the analysis and the quality of visualization.

2.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) have successfully implemented feature representation extraction for images, thus eliminating the need for hand-crafted features in image segmentation, and their superior performance and accuracy make them the main choice in this field. As the name suggests, these networks are based on convolutions. But what is a convolution? (For those of you who may be interested in the mathematical explanation and intuition behind this operator, we suggest seeing the linked video). We can explain

[1] This section was written largely based on "*Medical Image Segmentation Review: The Success of U-Net*", by Reza Azad et al.

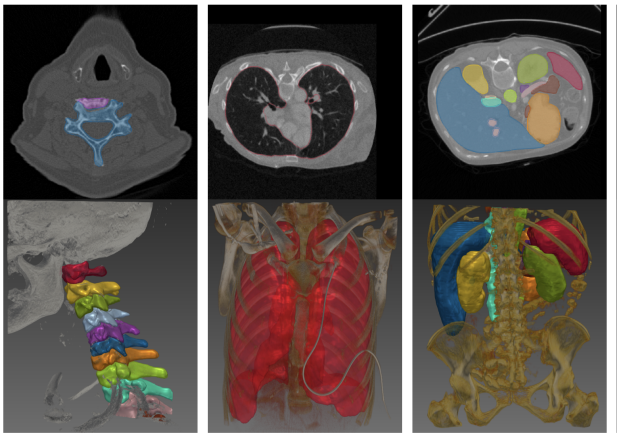

Figure 1: Examples of segmentation for various body parts.

it as follows: a small filter (kernel), represented by a matrix, slides over the image and computes a weighted sum of the local pixel values, where the weights are exactly the values inside the kernel. Different filters learn to detect specific visual patterns, such as edges, textures, or shapes. By stacking many convolutional layers, a CNN progressively builds a hierarchical representation of the image, moving from low-level details to high-level structures. Because convolutions preserve spatial information and greatly reduce the number of parameters compared to fully connected layers, CNNs are particularly well-suited for pixel-based tasks such as image segmentation.

2.2 The U-Net

Building on the progress made by Fully Convolutional Networks, Ronneberger et al. introduced U-Net, a deep learning architecture specifically designed for biomedical image segmentation. Fully Convolutional Networks, and in particular U-Net, became highly successful in medical image segmentation because they can achieve strong performance even with limited annotated datasets. Thanks to data augmentation strategies, such as random elastic deformations, U-Net can extract rich and detailed features without requiring large amounts of new training data. U-Net is built around a symmetric encoder–decoder architecture.

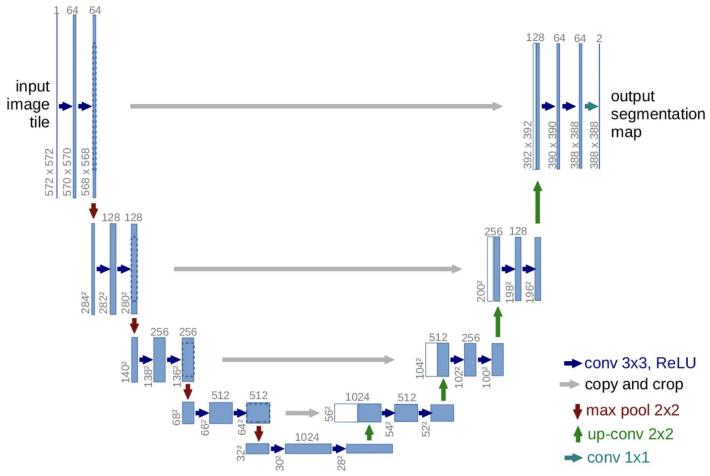

Figure 2: A 2D U-Net architecture with all the typical blocks.

The contracting path (encoder) progressively downsamples the image using convolutional blocks and

pooling operations. This part of the network captures semantic and contextual information by learning increasingly abstract features. The expansive path (decoder) gradually upsamples the feature maps, typically doubling their spatial resolution at each step. Convolutional blocks then refine these upsampled features to produce a dense, pixel-wise segmentation map. The key innovation of U-Net lies in its **skip connections** (grey in Figure 2), which directly copy feature maps from each stage of the encoder to the corresponding stage of the decoder. These connections preserve fine spatial details that would otherwise be lost during downsampling, allowing the decoder to combine low-level high-resolution information with high-level semantic features. This design significantly improves localization accuracy and boundary precision. Since its introduction in 2015, U-Net has become the backbone architecture for biomedical image segmentation, inspiring numerous variants and extensions that continue to dominate the field.

2.3 Implementing segmentation: main blocks

In this project you will learn to implement some of the basic blocks of a segmentation task.

Database importation

The first passage will be to correctly import all the data needed from a certain database in Kaggle.

Dataset and DataLoader classes

In order to load the data in the right way, you will need to define a `Dataset` class. This class is dedicated to organizing the samples and providing a clean interface to access them. In practice, it stores the paths to the images and masks (or the data arrays themselves) and implements two key methods: `__len__()`, which returns the number of samples, and `__getitem__()`, which returns the image-label pair corresponding to a given index, applying transformations when needed. In this way, the `Dataset` behaves like an indexed collection that can be used directly by PyTorch.

On top of this, the `DataLoader` class takes a `Dataset` instance and handles the logic of batching and shuffling of samples. During training, the `DataLoader` iterates over the dataset and yields mini-batches of images and labels, which are then fed to the network. This separation between `Dataset` (data definition) and `DataLoader` (data iteration) makes the code modular, efficient, and easy to extend to new datasets or preprocessing pipelines.

Transforms

During both training and validation, the images must undergo a series of transformations. These transformations serve two main purposes. First, they ensure that images and masks share the same spatial dimensions and datatype, which is essential for correct model input. Second, they perform data augmentation, such as rotations, flips, or elastic deformations, which helps the model generalize better by exposing it to varied versions of the same anatomical structures. As a result, the network becomes more robust to variations in orientation, scale, and noise.
For a detailed overview of the available operations, see the MONAI Transforms documentation.

Model

The model you will use is a simple UNet imported from MONAI framework.

Loss Function

The choice of the loss function plays a crucial role in training segmentation models, as it directly influences how well the network learns to distinguish foreground structures from the background. This choice should reflect the nature of the dataset and the clinical objective. For example, tasks that require precise boundary detection may benefit from boundary-aware losses, while highly imbalanced datasets often require overlap-based or focal variants. Selecting an appropriate loss is therefore essential to achieve robust segmentation performance. For a detailed overview of the available loss functions, see the MONAI Loss functions documentation.

Metric

To evaluate the performance of the segmentation model, an appropriate metric is required to quantify how well the predicted masks match the ground truth annotations. A widely adopted metric in medical image segmentation is the Dice Similarity Coefficient (DSC). This score computes the overlap between the prediction and the reference mask, taking values between 0 (no overlap) and 1 (perfect agreement). Monitoring the Dice score during training and validation provides a clear and interpretable indication of the model's segmentation quality. See MONAI documentation to select the right metric.

Training Loop

In this section, the model begins learning to segment the lungs in thoracic X-ray images. The training loop must be carefully structured so that you can reliably monitor whether the network is actually improving over time. Once the optimizer is defined, each iteration of the training loop performs a forward pass through the model, computes the loss, and then applies gradient descent to update the model parameters. By tracking the loss and evaluation metrics across epochs, you can assess convergence and ensure that the model is learning meaningful representations rather than simply memorizing the data.

Test

Finally, you can directly check how your model performs by computing the mask for a new image (not present in the test set), and superimposing it to the original mask.

3 GitHub

Once your model is trained and validated, the next step toward building an efficient web application with Streamlit is to organize your work in a dedicated GitHub repository. This repository should contain all the essential components of your project, including the trained model weights, the requirements of the environment you used in your notebook, and the Streamlit application files containing the pre- and post-processing parts. Using GitHub not only helps maintain a clean and reproducible workflow, but also allows you to track changes, collaborate with others, and easily deploy or share your work.

Basic Git Commands

To work efficiently with GitHub, it is useful to understand a few essential Git commands. After creating or cloning a repository, you can add files, track changes, connect your local project to GitHub, and upload your work using the following operations:

- **Initialize a repository**

```
git init
```

- **Check the status of your files**

```
git status
```

- **Add files to the staging area**

```
git add <filename>
git add .
```

- **Commit your changes with a message**

```
git commit -m "Describe your update"
```

- **Connect your local project to a GitHub repository**

```
git remote add origin https://github.com/<username>/<repository>.git
```

- **Upload (push) your commits to GitHub**

```
git push -u origin main
```

- **Download updates from GitHub**

```
git pull origin main
```

These commands cover the typical workflow: tracking changes, preparing commits, linking your local project to GitHub, and synchronizing the two repositories.

4 Streamlit Webapp

Once the model and preprocessing pipeline are in place, the final step is to build an interactive web application that allows users to upload medical images and visualize the segmentation results. Streamlit provides a simple and efficient framework for creating such applications directly in Python, without requiring any frontend development skills. Streamlit also offers seamless integration with GitHub, allowing you to host your application's code in a repository and connect it directly to a Streamlit deployment.