**KOCAELİ UNIVERSITY ENGINEERING FACULTY**

Logo

Description automatically generated

**MEDICAL IMAGE CLASSIFICATION AND SEGMENTATION**

**Furkan YARDIMCI**

**2022-2023**

**DEPARTMENT: Electronic and Communication Engineering**

**TEACHER: Dr. Lecturer Ayhan KÜÇÜKMANİSA**

**KOCAELİ**

**TABLE OF CONTENTS**

[TABLE OF FIGURES: 4](#_Toc137643377)

[PURPOSE 5](#_Toc137643378)

[INTRO 5](#_Toc137643379)

[1. SECTION 1 MEDICAL IMAGING 5](#_Toc137643380)

[1.1. MEDICAL IMAGING 5](#_Toc137643381)

[1.1.1. X-Ray 5](#_Toc137643382)

[1.1.2. MRI (Magnetic Resonance Imaging) 6](#_Toc137643383)

[1.1.3. CT (Computed Tomography) 8](#_Toc137643384)

[1.1.4. PET (Positron Emission Tomography) 9](#_Toc137643385)

[1.2. MEDICAL IMAGE PREPROCESSING 10](#_Toc137643386)

[1.2.1. Pre-processing Steps Specific To X-Ray Images 10](#_Toc137643387)

[1.2.2. Pre-processing Steps Specific To CT images 10](#_Toc137643388)

[1.2.3. Pre-processing Steps Specific To MRI Images 10](#_Toc137643389)

[1.2.4. Pre-processing Steps Specific To PET Images 11](#_Toc137643390)

[1.2.5. Example For Implementation Of The Pre-processing Steps 11](#_Toc137643391)

[1.3. MEDICAL FILE FORMATS 15](#_Toc137643392)

[1.3.1. DICOM (Digital Imaging and Communications In Medicine) 15](#_Toc137643393)

[1.3.2. NIfTI (Neuroimaging Informatics Technology Initiative) 15](#_Toc137643394)

[2. SECTION 2 MODULES 16](#_Toc137643395)

[2.1. PYDICOM 16](#_Toc137643396)

[2.1.1. Most Used Classes and Methods In PyDicom 16](#_Toc137643397)

[2.1.2. Example for Usage Of The PyDicom 16](#_Toc137643398)

[2.2. NIBABEL 23](#_Toc137643399)

[2.2.1. Most Used Classes and Methods In Nibabel 23](#_Toc137643400)

[2.2.2. Example 23](#_Toc137643401)

[2.3. TORCHIO 30](#_Toc137643402)

[2.4. Pytorch 31](#_Toc137643403)

[2.5. PytorchLightning 32](#_Toc137643404)

[3. SECTION 3 CLASSIFICATION 34](#_Toc137643405)

[3.1. Introduction 34](#_Toc137643406)

[3.2. Shallow Learning Algorithms 34](#_Toc137643407)

[3.2.1. Logistic Regression 34](#_Toc137643408)

[3.2.2. K-Nearest Neighbors (KNN): 35](#_Toc137643409)

[3.2.3. Support Vector Machines (SVM): 35](#_Toc137643410)

[3.2.4. Gaussian Naive Bayes: 36](#_Toc137643411)

[3.2.5. Decision Trees: 37](#_Toc137643412)

[3.2.6. Random Forest Classifier: 37](#_Toc137643413)

[3.3. PCA: 37](#_Toc137643414)

[3.4. IMPLEMENTATION: 39](#_Toc137643415)

[3.4.1. Dataset Inspection: 39](#_Toc137643416)

[3.4.2. Data Loading: 41](#_Toc137643417)

[3.4.3. Image Pre-processing: 41](#_Toc137643418)

[3.4.4. Data Splitting and Oversampling: 41](#_Toc137643419)

[3.4.5. Dimensionality Reduction: 41](#_Toc137643420)

[3.4.6. Model Training and Evaluation: 41](#_Toc137643421)

[3.5. CODE: 42](#_Toc137643422)

[4. Section 4 Segmentation 46](#_Toc137643423)

[4.1. Introduction 46](#_Toc137643424)

[4.2. CNN 46](#_Toc137643425)

[4.3. U-Net Model 48](#_Toc137643426)

[4.4. Dice Score 49](#_Toc137643427)

[4.5. IMPLEMENTATION 50](#_Toc137643428)

[4.5.1. Dataset Inspection 50](#_Toc137643429)

[4.5.2. Dataset Creation and Preparation 50](#_Toc137643430)

[4.5.3. Pre-processing and Augmentation 51](#_Toc137643431)

[4.5.4. Data Loader: 51](#_Toc137643432)

[4.5.5. Model Development Custom U-Net Architecture 51](#_Toc137643433)

[4.5.6. Model Training and Validation 51](#_Toc137643434)

[4.5.7. Prediction and Evaluation 52](#_Toc137643435)

[4.5.8. Visualization 52](#_Toc137643436)

[4.5.9. Logging with TensorBoard 52](#_Toc137643437)

[4.6. Code 52](#_Toc137643438)

[4.6.1. 01-DataInspection 53](#_Toc137643439)

[4.6.2. 0-2 Model 53](#_Toc137643440)

[4.6.3. 0-3 Train 56](#_Toc137643441)

[4.6.4. 0-4 Evaluation 60](#_Toc137643442)

[5. Source 64](#_Toc137643443)

# TABLE OF FIGURES:

[Figure 1: Xray Machine Diagram 5](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829399)

[Figure 2: Xray Machine 5](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829400)

[Figure 3: Xray Image 6](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829401)

[Figure 4: MRI Machine Diagram 6](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829402)

[Figure 5: MRI Machine 6](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829403)

[Figure 6: MRI Image 7](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829404)

[Figure 7: CT Diagram 7](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829405)

[Figure 8: CT Machine 7](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829406)

[Figure 9: CT Image 8](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829407)

[Figure 10: PET Machine 8](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829408)

[Figure 11: PET Image 9](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829409)

[Figure 12: Logistic Regression 38](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829410)

[Figure 13: K-NN Algorithm 39](#_Toc135829411)

[Figure 14: Linear SVM 40](#_Toc135829412)

[Figure 15: Kernel Trick 40](#_Toc135829413)

[Figure 16: Naïve Bayes 40](file:///C:\Users\furka\Desktop\MEDİCAL%20İMAGİNG%20Teknikleri.docx#_Toc135829414)

[Figure 17: Decision Tree 41](#_Toc135829415)

[Figure 18: Random Forest 41](#_Toc135829416)

[Figure 19: PCA1 42](#_Toc135829417)

[Figure 20: PCA2 42](#_Toc135829418)

[Figure 21: Dataset Visualization 44](#_Toc135829419)

[Figure 22: Dataset Labels 44](#_Toc135829420)

[Figure 23: CNN Layers 50](#_Toc135829421)

[Figure 24: CNN Layers2 50](#_Toc135829422)

[Figure 25: Convolution Process 51](#_Toc135829423)

[Figure 26: Pooling layer 51](#_Toc135829424)

[Figure 27: U-Net 53](#_Toc135829425)

[Figure 28: Dice Score 54](#_Toc135829426)

# PURPOSE

The aim of this report is to provide a detailed guide for conducting tumor classification and segmentation within medical imaging. The first part of this project involves using machine learning techniques for the classification of liver tumors using a decathlons dataset. The second part of the project then leverages a custom U-Net model for the task of 3D image segmentation of the tumors and the liver. The guide offers insight into various medical image processing techniques and how to handle them with Python-based tools and libraries. This document serves as a comprehensive resource for those interested in applying Python and machine learning methodologies to tackle complex tasks within the field of medical imaging, specifically related to tumor classification and 3D image segmentation.

# INTRO

This report presents a comprehensive study on medical imaging, primarily focused on liver tumors within 3D CT images. The project unfolds in three distinct sections. The first section delves into the theoretical and practical aspects of medical imaging techniques, discussing various types of medical files and their unique properties. It also explores how these files can be accessed and manipulated using Python, accompanied by a review of specific pre-processing methods required for each type of medical image. The second section of the project is dedicated to the classification of liver tumors, employing machine learning methodologies on the 3D CT images. The final section of this study presents a sophisticated 3D image segmentation approach to isolate the liver and the tumor within. This is accomplished using a custom U-Net model, demonstrating the powerful capabilities of deep learning in medical imaging tasks. This project serves as an in-depth exploration into the intersection of Python, machine learning, and medical imaging, providing valuable insights for similar future endeavours.

# SECTION 1 MEDICAL IMAGING

# MEDICAL IMAGING

# 1.1.1. X-Ray

X-ray imaging is the oldest and most frequently used form of medical imaging. It works by exposing a part of the body to a small dose of ionizing radiation to produce pictures of the inside of the body. X-rays are most used to visualize bones, and can detect bone fractures, infections, asnd cancers. They can also be used to visualize the chest, looking for signs of lung diseases, and the abdomen, identifying blockages or perforations in the digestive system.

Diagram of a cathode support system

Description automatically generated with low confidence

Figure 1: Xray Machine Diagram

A large machine in a room

Description automatically generated with low confidence

Figure 2: Xray Machine

X-ray imaging works by passing a controlled amount of X-ray radiation through the body and capturing the radiation that comes out on the other side on a film or a digital sensor. The radiation is absorbed by different tissues in varying degrees. Dense structures like bone absorb much of the radiation and appear white on the X-ray image, while softer tissues absorb less and appear darker.

X-ray machines consist of two separate components: an X-ray tube and an imager. The tube generates X-rays by accelerating electrons and crashing them into a metal plate, usually made of tungsten. This process produces X-rays that exit through a window in the tube and are directed toward the patient.

Figure 3: Xray Image

A picture containing x-ray film, medical imaging, radiology, radiography

Description automatically generated

# 1.1.2. MRI (Magnetic Resonance Imaging)

MRI uses a large magnet and radio waves to look at organs and structures inside your body. It can produce detailed images of the body without the use of ionizing radiation. MRI is especially useful for imaging soft tissues and organs like the brain, spinal cord, muscles, and heart. It is also useful for detecting diseases in these organs and tissues. It can detect structural abnormalities, tumors, infection, inflammation, or bleeding. MRI scans can take longer than other imaging methods and require the patient to remain still for extended periods.

Figure 4: MRI Machine Diagram

Figure 5: MRI Machine

MRI uses a powerful magnetic field and radio waves to generate images. The human body is primarily made of water molecules, which consist of hydrogen and oxygen atoms. At the center of each hydrogen atom is a proton, which acts like a tiny magnet. When you are placed into the MRI machine, the magnetic field aligns these proton "magnets". When the radio frequency waves are turned off, the protons "relax" and return to their original alignment, emitting radio signals in the process. The MRI machine picks up these signals and a computer processes them to create detailed, 3D images of the body. Different tissues send back different signals, allowing the MRI to differentiate between various types of tissue.

A close-up of a magnetic resonance imaging machine

Description automatically generated with medium confidence

A picture containing linear accelerator, wall, indoor, medical equipment

Description automatically generated

Figure 6: MRI Image

A picture containing medical imaging, radiology, medical, x-ray film

Description automatically generated

# 1.1.3. CT (Computed Tomography)

CT scans, also known as CAT scans, use a combination of X-ray technology and computer processing to create detailed images of the body. A CT scan can create detailed images of any part of the body, including the bones, muscles, fat, organs, and blood vessels. CT scans are more detailed than standard X-rays. They can detect bone and joint problems, like complex bone fractures and tumors, detect and monitor diseases and conditions like cancer, heart disease, lung nodules, and liver masses.

Figure 7: CT Diagram

Figure 8: CT Machine

A picture containing medical equipment, linear accelerator, medical, healthcare

Description automatically generated

A picture containing black, darkness, circle, black and white

Description automatically generated

CT scans are essentially a series of X-rays taken from many different angles around the body. These images are processed by a computer to produce cross-sectional slices of the body, like slices of bread. Each slice can be examined individually, and they can also be stacked together to create a 3D image.

During a CT scan, you lie on a table that moves through the CT machine, which is shaped like a large doughnut. The X-ray tube inside the machine rotates around your body and sends a thin beam of X-rays through your body at various angles. The X-ray detectors opposite the tube measure the radiation that has passed through your body and this information is used by the computer to generate the images.

Figure 9: CT Image

A close-up of an x-ray

Description automatically generated with medium confidence

# 1.1.4. PET (Positron Emission Tomography)

PET imaging uses a small amount of radioactive material, a special camera, and a computer to help evaluate organ and tissue functions. By looking at body changes at the cellular level, PET may detect the early onset of disease before it is evident on other imaging tests. PET scans are often combined with CT scans to produce even more detailed images. This combination (known as a PET/CT scan) allows any abnormality on the PET scan to be precisely located within the body, allowing for more accurate diagnosis and treatment planning.

Figure 10: PET Machine

A ct scan machine in a room

Description automatically generated with low confidence

PET scans involve the injection of a radioactive tracer into the body. This tracer is a special form of a substance (such as glucose) that collects in cells that use a lot of energy, such as cancer cells.

The PET scanner is a large machine with a round, doughnut-shaped hole in the middle, similar to a CT scanner. As the tracer begins to decay, it emits particles called positrons that immediately combine with electrons. This process, called annihilation, results in the emission of gamma rays, which can be detected and measured by the PET scanner. The scanner tracks the gamma rays and uses this information to create images that show where the tracer has collected in the body. This information can then be used to identify areas of disease.

A picture containing x-ray film, medical imaging, screenshot, radiology

Description automatically generated

Figure 11: PET Image

# 1.2. MEDICAL IMAGE PREPROCESSING

Medical image pre-processing is a crucial step in most medical imaging tasks, including radiology imaging analysis, computer-aided diagnosis, image-guided therapy, and more. Below are some common pre-processing steps:

# 1.2.1. Pre-processing Steps Specific To X-Ray Images

* Z-Normalization: This process involves subtracting the mean and dividing by the standard deviation of the pixel intensity values, bringing the data into a standard normal distribution (mean of 0 and standard deviation of 1). This technique can improve the performance of many machine learning models.
* Scaling: X-ray images can be scaled to a range between 0 and 255, which corresponds to the range of an 8-bit image. This scaling is often done as a way to standardize the dynamic range of the pixel values across different images.
* Noise Reduction: X-ray images may contain noise due to various factors, including electronic noise and quantum noise. Filters or denoising algorithms may be applied.
* Data Augmentation: Techniques like flipping, rotation, and zooming are used in deep learning applications to artificially increase the size of the dataset and make the model more robust.

# 1.2.2. Pre-processing Steps Specific To CT images

* Scaling: CT images can be scaled to a range between 0 and 3071. This corresponds to the typical range of Hounsfield Units, the unit of measurement used in CT scans, from -1000 (representing air) to +3071 (representing dense bone).
* Windowing: Windowing is a process that adjusts the range of pixel values that are displayed in the image, thus enhancing the contrast of certain tissues. For example, a typical window setting for viewing the lungs would have a window center of -500 HU and a window width of 1500 HU.
* Cropping: Cropping can be applied to focus on a region of interest within the image, which can be beneficial for both computational efficiency and model performance.
* Windowing: Windowing, also known as contrast stretching, is particularly important for CT images due to the wide range of pixel values, and different tissues being visible in different ranges of values.
* Slice Thickness Normalization: Since a CT scan is a volume composed of multiple 2D slices, ensuring all volumes have the same slice thickness can be an important pre-processing step.

# 1.2.3. Pre-processing Steps Specific To MRI Images

* Z-Normalization: Just like X-ray images, Z-normalization can be applied to MRI images as well, which is crucial for models that are sensitive to the scale of the input features.
* Min-Max Scaling: This technique involves scaling the intensity values of an image so that they lie between a desired minimum and maximum value. In the context of MRI images, it is often done individually for each subject due to the lack of a standardized intensity scale across different MRI machines and protocols.
* Cropping: Cropping can be applied to MRI images to focus on the region of interest, which can increase computational efficiency and potentially improve model performance.

# 1.2.4. Pre-processing Steps Specific To PET Images

* Scaling: PET images are typically scaled by clipping values at 20-40 to avoid extreme outliers, as these images can have a very high dynamic range due to the nature of the PET imaging process.
* Log Transformation: A logarithmic transformation can be applied to PET images to help reduce the dynamic range of the values, making it easier to visualize and process the images.

# 1.2.5. Example For Implementation Of The Pre-processing Steps

A screenshot of a computer program

Description automatically generated with low confidence

A screenshot of a computer

Description automatically generated with medium confidence

A picture containing text, screenshot

Description automatically generated

# MEDICAL FILE FORMATS

Medical file formats are standardized ways of organizing and storing digital medical images, along with associated metadata. These file formats are often used in radiology, cardiology, oncology, and other medical specialties that rely heavily on imaging. They are designed to support a range of needs, including clinical care, research, and teaching.

# 1.3.1. DICOM (Digital Imaging and Communications In Medicine)

DICOM is a standard for storing and transmitting information in medical imaging. It was developed by the American College of Radiology (ACR) and the National Electrical Manufacturers Association (NEMA). The standard includes protocols for imaging techniques such as radiography, computed tomography (CT), magnetic resonance imaging (MRI), and ultrasound. DICOM files contain both image data and corresponding metadata. The image data is usually stored as a two-dimensional array of pixels, where each pixel corresponds to a particular tissue or structure in the body. The metadata, which can include patient information, details about the imaging device, and parameters for the imaging procedure, is stored alongside the image data in a DICOM file. One important feature of the DICOM format is that it allows for the inclusion of a patient's entire medical history with the image data. This comprehensive view of the patient's medical history can greatly facilitate clinical decision-making. Python, a widely used language in data science and machine learning, offers libraries like PyDicom, which provide convenient interfaces to work with these files. There is an example for usage of the PyDicom is at Section 2.

# 1.3.2. NIfTI (Neuroimaging Informatics Technology Initiative)

The NIfTI format is a file format primarily used in neuroimaging research. It was developed by the Neuroimaging Informatics Technology Initiative, a project that aims to develop standards for the collection and sharing of neuroimaging data. A NIfTI file contains both imaging data and a header. The imaging data is typically a three-dimensional array of voxels (volumetric pixels), where each voxel represents a particular volume of tissue in the brain. The header contains metadata about the image, such as the dimensions of the image, the units of measurement, and the spatial orientation of the image. One key feature of the NIfTI format is that it allows for the storage of three-dimensional and even four-dimensional data in a single file. A fourth dimension can be used to store a time series, making NIfTI a popular choice for functional MRI (fMRI) data, which captures changes in brain activity over time. Even though it is primarily used in brain imaging, you can store any type of MRI, CT images in this format and it is more convenient than DICOM files most of the time. Python, a widely used language in data science and machine learning, offers libraries like NiBabel, which provide convenient interfaces to work with these files. There is an example for usage of the NiBabel is at Section 2.

# SECTION 2 MODULES

# 2.1. PYDICOM

PyDicom is an open-source Python package specifically designed to parse DICOM files which are the standard for transmitting, storing, and sharing medical imaging data. DICOM files not only store the image data but also carry a substantial amount of metadata, such as patient information, details about the scanning procedure, and image dimensions. PyDicom enables Python scripts to read and write data in the DICOM format with ease.

The main reasons for using PyDicom in this project were:

* Ease of Access: PyDicom can read all standard types of DICOM data, regardless of the imaging modality.
* Rich Metadata: The library allows access to all the metadata stored in the DICOM header, which can provide valuable context for the imaging data.
* Integration with Python: Being a Python library, PyDicom can easily be integrated into a wider data processing or machine learning pipeline.

# 2.1.1. Most Used Classes and Methods In PyDicom

* **pydicom.dcmread():**

This is a function used for reading a DICOM file. It takes the path to the file as a string and returns a FileDataset object representing the DICOM file. The returned object allows access to all the image data and metadata stored in the DICOM file.

* **FileDataset** Class:

A FileDataset object represents a DICOM file. The object contains all the data elements specified in the DICOM file, and these can be accessed as attributes of the object.

* **DataElement** Class:

A DataElement object represents a single data element in a DICOM file. It contains the tag, value representation (VR), value multiplicity (VM), and value of the data element. You can access these properties through the attributes of the DataElement object.

* **Dataset.pixel**\_array:

This is a property of the FileDataset class that gives access to the image data stored in the DICOM file as a NumPy array. This is useful for processing or visualizing the image data using other Python libraries.

* **Dataset.save\_as():**

This is a method of the FileDataset class that is used to save the DICOM data to a new DICOM file. It takes the path to the new file as a string.

# 2.1.2. Example for Usage Of The PyDicom

In this example, we will see how to access DICOM files in various directories with python, how to use PyDicom, how to access volume information in DICOM files with PyDicom and visualize**:**

A screenshot of a computer

Description automatically generated with medium confidenceDICOM files.

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

# 2.2. NIBABEL

Nibabel is another open-source Python library specifically designed to read and write data in various common neuroimaging file formats, including NIfTI and ANALYZE. It is a simple and flexible tool that can be a valuable part of the neuroimaging data analysis ecosystem. It provides read and write access to some common medical and neuroimaging file formats, including ANALYZE (plain, SPM99, SPM2), GIFTI, NIfTI1, and NIfTI2.

The main reasons for using Nibabel in this project were:

• Format Flexibility: Nibabel supports a wide variety of neuroimaging file formats, including the popular NIfTI format, which is frequently used for MRI data.

• Efficient Handling: Nibabel handles data array storage efficiently, which is vital when dealing with large 3D images common in medical imaging.

• Integration with Python: As a Python library, Nibabel easily integrates with data processing or machine learning pipelines.

# 2.2.1. Most Used Classes and Methods In Nibabel

* nibabel.load() This function is used to load image data from a file. It takes the file path as a string and returns a spatial image object that holds the image data and the associated metadata.
* Image Class The Image class is a standard representation for neuroimaging data in Nibabel. An Image object contains a 3D or 4D data array and an affine array that aligns the data array to a reference space.
* Image.get\_fdata() This method of the Image class is used to retrieve the image data as a NumPy array, which can be manipulated using standard NumPy operations.
* Image.affine This property of the Image class provides access to the affine array that aligns the data array to a reference space.
* nibabel.save() This function is used to save an Image object to a file. It takes an Image object and the path to the output file as strings.

# 2.2.2. Example

In this example, we will see how to access NIFTI files in various directories with python, how to use Nibabel, how to access volume information in NIFTI files with Nibabel, how to turn DICOM files into NIFTI file using dicom2nifti library and visualize**:**



A screenshot of a computer

Description automatically generated with medium confidence,

A screenshot of a computer code

Description automatically generated with low confidence

A screenshot of a computer screen

Description automatically generated with low confidenceKey features of nibabel are:

A screenshot of a computer screen

Description automatically generated with low confidence

A screenshot of a computer screen

Description automatically generated with low confidence

# 2.3. TORCHIO

We have used the TorchIO API to load images and pre-process them. TorchIO is an open-source Python library developed to efficiently pre-process, augment, and manage three-dimensional (3D) medical images. Leveraging the capabilities of PyTorch, it offers a powerful framework for machine learning applications in the field of medical imaging. It also facilitates the use of advanced techniques in deep learning, such as 3D convolutional neural networks (CNNs).

TorchIO provides tools that operate seamlessly on 3D images and are compatible with other existing libraries in the PyTorch ecosystem, thereby optimizing and streamlining the workflow. The library includes a comprehensive set of features, including:

* **Image Loading**: TorchIO supports a variety of 3D medical image formats, including DICOM, NIfTI, and more. This broad support enables its usage in a variety of medical imaging contexts.
* **Pre-processing and Augmentation**: The library includes numerous pre-processing functions and augmentation techniques specifically designed for 3D medical images. These operations help to prepare the data for a machine learning model and augment the dataset, respectively.
* **Interoperability**: TorchIO is built on top of PyTorch and is compatible with other PyTorch-based libraries, providing a streamlined workflow within the PyTorch ecosystem.

Now, let's delve into some of the key classes and methods in TorchIO:

**Compose**: This class comes from the **torchio.transforms** module. Compose is used to group together a list of transformations, which are applied sequentially. Each transformation in the list should be an instance of a class that works as a transformation. This functionality mirrors the Compose function found in **torchvision.transforms.**

**ScalarImage**: ScalarImage is a class used to denote 3D scalar images, such as MRIs or CT scans, which have one value per voxel. The ScalarImage class contains data and metadata for 3D scalar images. It can be used to load images from a desired directory.

Queue: The Queue class in TorchIO is a patch-based queue used to load 3D patches from subjects on demand. This provides an efficient way to handle large volumes of data. This class is especially useful when the entire dataset cannot fit into memory.

**Patcher:** The tio.PatchSampler class is responsible for dividing the volumes into patches during the patch-based processing. It works in conjunction with the tio.Queue class to sample patches from the dataset. Here's some more information about the **tio.PatchSampler:**

* Patchers are objects that determine how the volumes are divided into patches.
* tio.PatchSampler is a subclass of torchio.data.sampler.GridSampler.
* It takes a volume or a list of volumes as input and samples patches from them.
* The patch size and overlap can be specified to control the size and coverage of the patches.
* By using patch-based processing, the memory requirements can be reduced, making it easible to process large volumes of data.

**Sampler:** In TorchIO, a sampler is used to define the sampling strategy when extracting patches from the volumes. TorchIO provides various sampling strategies, such as random sampling, grid sampling, and stratified sampling. These strategies determine how patches are selected from the volumes. The tio.data.sampler module contains different sampler classes, such as RandomSampler, GridSampler, and StratifiedSampler, which can be used based on the specific requirements of the application.

By using Patchers and Samplers in TorchIO, it becomes possible to efficiently handle large volumes of 3D medical image data, divide them into patches, and sample the patches for processing or training deep learning models.

# 2.4. Pytorch

PyTorch is an open-source machine learning library that provides a high-level interface for tensor computations and deep neural networks. Developed by Facebook's AI Research lab, PyTorch has garnered widespread acclaim for its flexibility and intuitive design.

Key Features:

* Tensor Computing: PyTorch provides a multi-dimensional array called a Tensor, like NumPy's ndarrays, with the added feature that Tensors can also be used on a GPU to accelerate computing.
* Deep Neural Networks: PyTorch offers a robust framework for building and training complex neural network architectures, offering flexibility and modularity through its class-based structure.
* Automatic Differentiation and Gradients: PyTorch supports automatic differentiation, which automates the computation of backward passes in neural networks. Its auto grad package provides the functionality to calculate the gradient automatically.
* Interoperability: PyTorch is designed to be deeply integrated into Python. It can seamlessly work with other Python libraries like NumPy or SciPy.
* Distributed Training: PyTorch supports distributed training which enables the usage of multiple GPUs, helping to scale up computations and reduce training time.
* Dynamic Computational Graphs: PyTorch offers dynamic computation graphs, meaning that the graph is built up from scratch at each iteration, allowing for changes in the way the graph operates for each epoch. This is particularly useful in situations where you do not know how much memory will be required for creating a neural network.
* Easy Debugging: As PyTorch is deeply integrated with Python, you can use any Python debugger to debug your model.
* Support for CUDA: PyTorch provides good native support for NVIDIA GPUs (Graphics Processing Units) using CUDA (Compute Unified Device Architecture). This feature allows tensors to be stored on GPUs for efficient computation.

Key Classes and Methods:

**torch.Tensor**

* .requires\_grad\_(): Sets the tensor’s requires\_grad attribute in-place, instructing PyTorch to track computations on this tensor and compute gradients during the backpropagation step.
* .backward(): Computes the gradient of the current tensor with respect to graph leaves.
* .detach(): Returns a new tensor, detached from the current computation graph, preventing derivative propagation back through this tensor.

**torch.nn.Module**

* .parameters(): Returns an iterator over module parameters, typically used when optimizing a model.
* .zero\_grad(): Sets gradients of all model parameters to zero.
* .to(device): Moves all module parameters to a given device (like a GPU).
* .train() and .eval(): Switches the module between training and evaluation mode.

**torch.optim**

* .step(): Updates the optimizer's parameters based on gradients computed in the backward pass.
* .zero\_grad(): Clears the gradients of all optimized parameters.
* .state\_dict(): Returns the optimizer's state as a dictionary, useful for checkpointing the model.
* .load\_state\_dict(state\_dict): Loads the optimizer state, typically used when resuming a training job.

In summary, PyTorch is a comprehensive library that provides flexible and powerful tools for machine learning research and application development. Its dynamic computation graphing system, easy-to-use API, and deep integration with Python make it an excellent choice for both the research community and the industry.

# 2.5. PytorchLightning

PyTorch Lightning is a lightweight PyTorch wrapper with a high-level interface for building and training complex deep learning models. It was developed with the aim to provide a high-level interface to PyTorch, reducing the boilerplate code and making code more readable and reusable. Here are some of the core aspects:

* Simplicity: PyTorch Lightning structures your code so it's abstracted in the right places. It simplifies the engineering part of deep learning without oversimplifying and providing so much abstraction that the internals are not accessible or customizable.
* Flexibility: It allows for seamless switching between CPUs, GPUs, TPUs and even multi-node training, making your code device-agnostic.
* Reproducibility: It has built-in features that ensure the total reproducibility of experiments. This is extremely important in scientific settings where it's crucial to be able to reproduce results.
* Built-in best practices: PyTorch Lightning incorporates the latest best practices and standards determined by the research community right out of the box.
* Scale with minimal changes: You can begin prototyping in Jupyter notebooks with CPU, then effortlessly scale your code to run on any hardware setup with minimal changes.
* Easy debugging: Debugging is much easier in PyTorch Lightning because it allows you to do distributed computing, sequential running, and fast dev run for easy debugging.
* Advanced features support: PyTorch Lightning also supports advanced features like 16-bit precision, gradient accumulation, and more.

Here are some of the core components:

**LightningModule**: This is the main class in PyTorch Lightning. It's a torch.nn.Module, but with added functionality. A LightningModule organizes your PyTorch code into 5 sections: Computations (forward), Training loop (training\_step), Validation loop (validation\_step), Test loop (test\_step), and Optimizers (configure\_optimizers).

Trainer: The trainer class contains the logic for training loop, validation loop, and testing loop. It also includes the training configurations like num\_epochs, gpus, etc.

**DataLoader:** PyTorch Lightning uses the standard PyTorch data loaders. You specify the data loaders for training, validation, and testing in your LightningModule.

**DataLogger:** It can use TensorBoard tool to log desired datas, images etc.

In essence, PyTorch Lightning is a tool that helps structure your machine learning code to ensure it is more maintainable and reusable, while also ensuring the reproducibility of experiments. It's compatible with the PyTorch ecosystem and allows users to leverage all the features of PyTorch while making the process of developing complex models easier and more intuitive.

# SECTION 3 CLASSIFICATION

# 3.1. Introduction

Cancer, specifically liver cancer, has been a leading cause of death worldwide, posing severe public health challenges. Early detection and classification of liver tumors significantly increase the survival rate among patients. Medical imaging techniques like CT, MRI, PET, etc., play a crucial role in these detection processes, producing a plethora of imaging data. However, manually analysing these images can be time-consuming, error-prone, and often unfeasible given the vast volume of data. Therefore, automated, accurate, and efficient classification systems are in dire need.

This report outlines a machine learning project aimed at classifying 3D liver tumors using shallow learning techniques. The project involves several stages, including data importing, pre-processing, dimension reduction, and model building with various shallow learning algorithms. All these stages were performed on a dataset composed of 3D NIfTI files of liver images. In next steps we will learn some of the important subjects we have used to tackle classification problem.

# 3.2. Shallow Learning Algorithms

# 3.2.1. Logistic Regression

Just like linear regression, logistic regression is a[machine learning model](https://learn.microsoft.com/en-us/windows/ai/windows-ml/what-is-a-machine-learning-model#:~:text=A%20machine%20learning%20model%20is,and%20learn%20from%20those%20data.) used to determine the relationship between a dependent variable and one or more independent variables. However, this model is used for classification analysis. This is because logistic regression predicts the probability of an event occurring. For a probability greater than 0.5, a value of 1 is assigned, and for less than that 0. For example, you can use logistic regression to predict whether a student will pass (1) an exam, or they will fail (0).

Figure 12: Logistic Regression

A picture containing line, plot, diagram

Description automatically generated

# 3.2.2. K-Nearest Neighbors (KNN):

A picture containing line, screenshot, diagram, design

Description automatically generatedK-Nearest Neighbour is another simple [machine learning algorithm](https://datasciencedojo.com/blog/machine-learning-for-seo-marketing/) that classifies new cases based on the category/class of the data points nearest to the new data point. That is, if most neighbours of an unknown item belong to class 1, then we assign class 1 to this unknown item. The number of neighbours to take into consideration is the value K assigned. If k=10, we will look at the 10 nearest neighbours of this item. The nearest neighbours are determined by measuring the distance using distance measures such as Euclidean distance, and the nearest are those that have the shortest distance.

Figure 13: K-NN Algorithm

# 3.2.3. Support Vector Machines (SVM):

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labelled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. In two-dimensional space, this hyperplane is a line dividing a plane into two parts where each class lays on either side. You have used two kinds of SVM in your code:

* Linear SVM: This is a SVM that uses a linear function to create a hyperplane to separate the classes.
* Kernel SVM or Radial basis function (RBF) SVM: This SVM uses a non-linear function to map the original data points into a higher-dimensional space where a hyperplane can be used to separate the classes. The RBF kernel is one of the most used kernels.

A picture containing diagram, screenshot, origami, design

Description automatically generatedA picture containing diagram, line, screenshot, text

Description automatically generatedFigure 14: Linear SVM

Figure 15: Kernel Trick

# 3.2.4. Gaussian Naive Bayes:

Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features. GaussianNB implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian.

Figure 16: Naïve Bayes

A diagram of a classifier

Description automatically generated with low confidence

# 3.2.5. Decision Trees:

A picture containing diagram, circle

Description automatically generatedDecision tree is a supervised machine learning model that repeatedly splits the data based on a question corresponding to the features. The model learns the best way to reduce randomness and drafts a decision tree that can be used to predict the category of an item based on answering a selection of questions. For example, in the case of whether it will rain today or not, the questions can be whether it is sunny, did it rain yesterday, whether it is windy, and so on.

Figure 17: Decision Tree

# 3.2.6. Random Forest Classifier:

Random Forest is a machine learning algorithm that works similarly to a decision tree. The difference is that random forest uses multiple decision trees to make a prediction and hence decreases overfitting. The process of majority voting is carried out and the class selected by most trees is assigned to an item. For example, if two trees predict it to be 0, and one tree predicts it to be 1, then the class of 0 will be assigned to the item.

A diagram of a tree

Description automatically generated with medium confidence

Figure 18: Random Forest

# 3.3. PCA:

Principal component analysis (PCA) is a technique that transforms high-dimensions data into lower-dimensions while retaining as much information as possible.

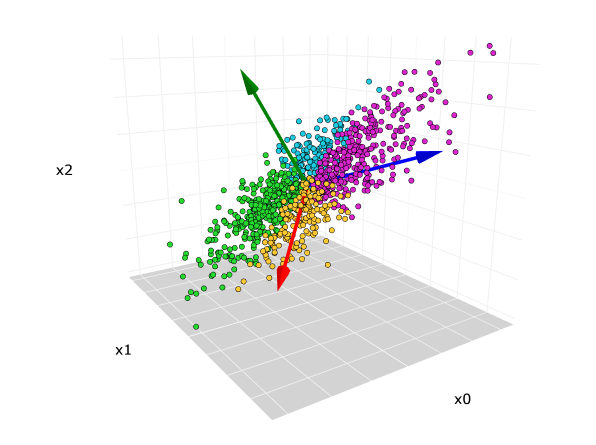


Figure 19: PCA1

The original 3-dimensional data set. The red, blue, green arrows are the direction of the first, second, and third principal components, respectively. Image by the author.

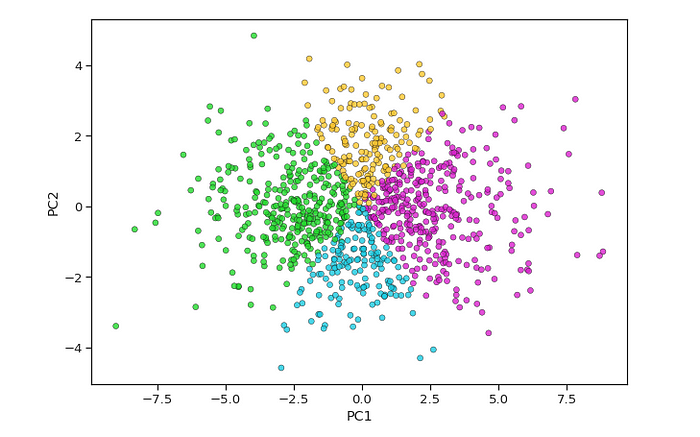


Figure 20: PCA2

Scatterplot after PCA reduced from 3-dimensions to 2-dimensions. Image by the author.

PCA is extremely useful when working with data sets that have a lot of features. Common applications such as image processing, genome research always have to deal with thousands-, if not tens of thousands of columns.

While having more data is always great, sometimes they have so much information in them, we would have impossibly long model training time and the [curse of dimensionality](https://towardsdatascience.com/the-curse-of-dimensionality-50dc6e49aa1e) starts to become a problem. Sometimes, less is more. Curse of dimensionality means that if we have more dimension, we need more data. If we don’t have enough data, we simply use PCA like methods to reduce dimension.

# 3.4. IMPLEMENTATION:

this code is primarily used for classifying liver tumor condition using CT images. The images and labels used for this process are pre-processed and then various machine learning models are trained and tested. Here is a summary of the code:

# 3.4.1. Dataset Inspection:

We used more lightened versions of Decathlon's 3d CT liver images. We used smaller and resized versions of these images, which are normally close to 1000x1000 in size. The biggest problem with these data is that not all of them are equal in size. It has an uneven size distribution. To remove this imbalance, we will use the TorchIO library and fix all images to 256x256x200. Since this size is too large for video cards with video ram such as 4gb-6gb, we will again use the sampler class of the TorchIO library with the que class and divide the 3d volume into small patches and train our U-Net network that way. In this way, the diversity of our data set, which is small, and the amount of data will increase even more.

Here is the size distribution of the data set and some images from dataset.

* (180, 180, 125)
* (221, 221, 156)
* (189, 189, 201)
* (256, 256, 43)
* (256, 256, 51)
* (256, 256, 42)
* (256, 256, 37)
* (256, 256, 41)
* (256, 256, 44)
* (216, 216, 231)
* (183, 183, 205)
* (170, 170, 201)
* (198, 198, 231)
* (181, 181, 79)

Figure 21: Dataset Visualization

A picture containing text, number

Description automatically generatedNo label data is provided by Decathlon for classification. I examined all the images myself with a 3d slicer and created the labels myself. Below is part of the list.

Figure 22: Dataset Labels

# 3.4.2. Data Loading:

The script begins by defining the path to the training images. It then iterates over these file paths and uses helper functions to find and assign labels from a CSV file based on the file name.

In this part of the script, TorchIO, a Python package for efficient manipulation and pre-processing of 3D medical images, is used. Specifically, tio.ScalarImage class is used to read and store the 3D CT scan data. Each 3D CT scan file is loaded and converted into a TorchIO image object for further preprocessing.

# 3.4.3. Image Pre-processing:

Two sets of transformations are prepared for the images, one for the training dataset (Train) and another for the test dataset (Test). Both sets include the tio.CropOrPad and tio.RescaleIntensity transformations, packaged together using the tio.Compose function.

The tio.CropOrPad((200, 200, 100)) transformation is used to ensure all 3D CT scans have a consistent size of (200, 200, 100). This is important for training machine learning models as they often require input data of a consistent shape.

The tio.RescaleIntensity((-1, 1)) transformation rescales image intensities to be between -1 and 1, thereby standardizing the images. This step is crucial as it helps to standardize the images, improving the performance and stability of the subsequent machine learning models.

# 3.4.4. Data Splitting and Oversampling:

The script then splits the data into a training set and a test set, reshapes the images, and applies the transformations. This involves converting the TorchIO images to NumPy arrays and reshaping them into one-dimensional vectors, which are then appended to the training or test datasets.

After preparing the training data, an oversampling operation is performed to handle any class imbalance in the dataset using the RandomOverSampler from the imbalanced-learn library.

# 3.4.5. Dimensionality Reduction:

Principal Component Analysis (PCA) is used to reduce the dimensionality of the data, keeping 98% of the total variance in the data. This reduces computational costs while retaining most of the useful information.

# 3.4.6. Model Training and Evaluation:

Several different machine learning models, including Logistic Regression, K-Nearest Neighbors, Support Vector Machines, Naive Bayes, Decision Trees, and Random Forest, are trained and their training accuracy is reported.

Finally, confusion matrices and accuracy, precision, and recall metrics for each model are computed on both the training and test datasets, providing a comprehensive evaluation of each model's performance.

In summary, this script employs the power of the TorchIO library for handling 3D medical imaging data and prepares this data for machine learning models to classify liver tumors in CT scans. The pre-processing steps applied to the CT scans, the dimensionality reduction technique, and the variety of machine learning models all contribute to the robustness of this classification task

# 3.5. CODE:

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

# Section 4 Segmentation

# 4.1. Introduction

Medical imaging is a cornerstone of modern healthcare, serving critical roles in diagnosis, treatment planning, and monitoring disease progression. The advent of deep learning has made it possible to tackle complex segmentation tasks in medical imaging with increased accuracy and efficiency. A notable architecture in this regard is the U-Net, a type of convolutional network that has proven highly effective for such tasks. This report details the application of a custom U-Net model, comprising 3 down- and 3 up-convolutions with two convolutions in each stage, for the segmentation of liver tumors in 3D images.

# 4.2. CNN

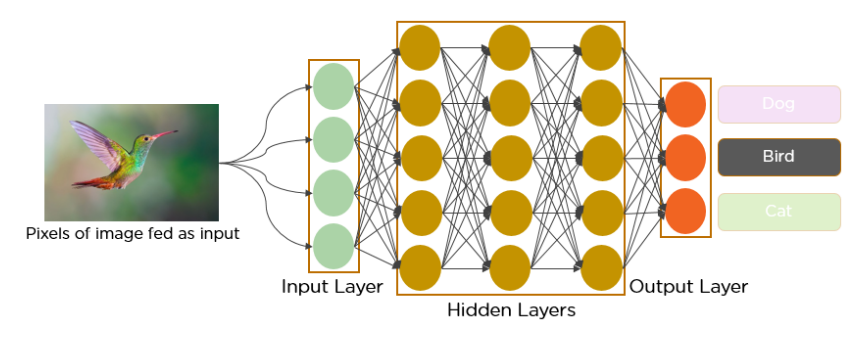
A Convolutional Neural Network (CNN) is a type of artificial neural network that has proven very effective in areas such as image recognition and classification. CNNs have been successful in identifying faces, objects, and traffic signs apart from powering vision in robots and self-driving cars.

Figure 23: CNN Layers

A picture containing car, vehicle, land vehicle, screenshot

Description automatically generated**Convolutional Layer:** This is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input, and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire objects or parts of objects on deeper layers of the network.

Figure 24: CNN Layers2

A picture containing diagram, text, line, plan

Description automatically generatedFigure 25: Convolution Process

**ReLU Layer:** This layer applies an elementwise activation function, such as the max(0,x) thresholding at zero. This leaves the size of the volume unchanged. The purpose of this layer is to introduce nonlinearity in the network, as the earlier layers are primarily linear.

A picture containing text, screenshot, diagram, line

Description automatically generated**Pooling Layer**: This layer reduces the spatial size of the representation to reduce the number of parameters and computation in the network. Pooling layer operates on each feature map independently. The most common approach used in pooling is max pooling.

Figure 26: Pooling layer

**Fully Connected Layer**: These are the traditional multi-layer perceptrons that perform high-level reasoning in the neural network. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular Neural Networks.

**How CNNs work:**

CNNs are designed to automatically and adaptively learn spatial hierarchies of features from a training dataset. These are very effective in areas where prior knowledge and human effort in feature design is not required.

The network architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units.

A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, ReLU layers, pooling layers, and fully connected layers.

In the convolutional layer, the output of each neuron is computed as a dot product between its weights and a small region it is connected to in the input volume. This may result in the network learning filters that become activated when they see certain visual features such as edges from some orientation or colour channel, or other more sophisticated features in case of deeper layers.

CNNs use a variation of multilayer perceptrons designed to require minimal pre-processing. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics.

In summary, CNNs are designed to automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks, such as convolutional layers, ReLU layers, pooling layers, and fully connected layers. This makes them particularly effective for image recognition and classification tasks.

# 4.3. U-Net Model

U-Net is a specific type of Convolutional Neural Network (CNN) that has proven to be successful in tasks such as medical image segmentation. Introduced by Ronneberger et al. in 2015, U-Net gets its name from its characteristic "U"-shaped architecture.

While U-Net was originally developed with the aim of segmenting cellular structures from microscopic images, U-Net and its various variants have been used in a wide range of medical imaging modalities, including MRI, CT, and other imaging technologies.

**U-Net Architecture and How It Works:**

U-Net is divided into two parts: an "encoder" (contracting path) and a "decoder" (expanding path).

**Encoder:** The encoder, similar to a traditional CNN, extracts features from the input image. In each layer, the image size decreases (down sampling), but the depth of the feature map increases. This provides a more abstract representation of the features.

**Decoder:** The decoder uses the features from the encoder and performs a pixel-level classification by upscaling back to the original size of the image (up sampling). Here, higher resolution feature maps are concatenated with lower resolution feature maps (skip-connection).

These "skip-connections" between the encoder and decoder ensure that the localization information from the encoder is combined with the contextual information in the decoder, allowing U-Net to precisely localize and segment the desired structures.

U-Net in Medical Image Segmentation:

Medical image segmentation is a crucial task in many medical applications. It involves the identification of regions of interest in a medical image, such as identifying tumors in a CT scan or blood vessels in an MRI scan. U-Net has become a popular choice for such tasks due to its efficient training and superior performance on various benchmarks. Its ability to work with a small amount of labelled data, and the precise localization it offers due to the use of skip-connections, make U-Net particularly suited for medical image segmentation.

A picture containing text, screenshot, diagram, line

Description automatically generatedIn summary, U-Net has significantly contributed to advancements in the field of medical image segmentation. Its unique architecture allows for the efficient and accurate segmentation of structures in medical images, leading to more precise diagnoses and improved patient outcomes.

Figure 27: U-Net

# 4.4. Dice Score

The Dice score, named after Lee Raymond Dice, is a statistical tool used for comparing the similarity of two samples. It is used frequently in image analysis, including medical image segmentation, where it measures the similarity between the predicted segmentation and the ground truth.

The Dice score is a number between 0 and 1, where a larger number represents a better match between the two samples. A Dice score of 1 represents a perfect match, meaning that the predicted segmentation is exactly the same as the ground truth. A score of 0, on the other hand, indicates that there is no overlap at all.

**A picture containing font, symbol, screenshot, logo

Description automatically generatedFormula:**

Figure 28: Dice Score

The Dice score is defined as follows:

DICE = (2 \* |X ∩ Y|) / (|X| + |Y|)

where:

* X and Y are the predicted and ground truth binary masks respectively.
* |X ∩ Y| is the common region between predicted and ground truth.
* |X| and |Y| are the total number of pixels in X and Y respectively.

This is equivalent to twice the number of overlapping pixels between the predicted and ground truth segmentations, divided by the total number of pixels in both the predicted and ground truth segmentations.

Dice Score in Medical Image Segmentation:

In the field of medical image segmentation, the Dice score is a common evaluation metric for the quality of a segmentation model. This is due to its ability to handle unbalanced data and the fact that it gives a good sense of spatial overlap.

# 4.5. IMPLEMENTATION

# 4.5.1. Dataset Inspection

The same data set introduced in the previous step was used in this section too.

# 4.5.2. Dataset Creation and Preparation

The dataset used in this project consisted of 3D NIfTI files, which were loaded using TorchIO's tio.Subject, tio.ScalarImage, and tio.LabelMap classes. These classes are utilized to load NIfTI images from the desired path and convert them into TorchIO objects. This conversion allows for the easy implementation of data augmentations, pre-processing, and other operations using TorchIO's functionalities. By using these classes, the NIfTI images are effectively transformed into TorchIO-compatible data structures, enabling seamless integration with TorchIO's rich set of image processing tools.

# 4.5.3. Pre-processing and Augmentation

The pre-processing pipeline encompassed resizing the volumes to a consistent size (256x256x200) and rescaling the intensity values to lie within -1 and 1. The Compose function from torchio was utilized to consolidate this pipeline.

For data augmentation, we employed the RandomAffine transformation from torchio, which randomly applies scale and rotation transformations to the volumes. This augmentation was incorporated only into the training pipeline, thereby increasing the diversity of the training data and enhancing the model's generalization capabilities.

# 4.5.4. Data Loader:

In the liver and liver tumor segmentation task, the input volumes are often large and cannot be directly processed by the model due to memory limitations. To overcome this challenge, the volumes are divided into smaller patches that can be individually processed by the model. The predicted patches are then aggregated to obtain the segmentation for the entire volume.

In this project, TorchIO's tio.Queue class is used to handle the patch-based processing. During the training and validation phase, two queues are created. These queues utilize a sampler, which divides the 3D volume into patches. The model is trained using this patch-based approach, where each patch serves as an individual input to the model. This allows for efficient training and utilization of memory resources.

During the evaluation phase, a slightly different approach is taken. Instead of using tio.Queue, the tio.GridSampler class is employed to divide the test images into patches. These patches are then fed to the model for inference. Subsequently, the predictions from the patches are merged into a single 3D image using the tio.GridAggregator class. This merging process combines the patch predictions to obtain the final segmentation for the entire volume.

By leveraging TorchIO's patch-based processing and the appropriate classes such as tio.Queue, tio.GridSampler, and tio.GridAggregator, the memory limitations associated with large volumes can be effectively addressed, enabling accurate and efficient segmentation of liver and liver tumors.

# 4.5.5. Model Development Custom U-Net Architecture

The model designed for this project was a custom U-Net architecture, constituted of three down- and three up-convolutions, with two additional convolutions at each step. Defined within PyTorch's nn.Module, this architecture was instantiated within a PyTorch Lightning Module, termed Segmenter. Within this module, we also defined the optimization method (Adam) and the loss function (CrossEntropyLoss).

# 4.5.6. Model Training and Validation

With the model architecture, optimizer, and loss function defined, the training and validation steps were set up within the Segmenter class. Each step involved processing a batch of volumes and their corresponding masks, computing the predictions, and calculating the loss. These values, alongside images, were logged for visualization in TensorBoard. The PyTorch Lightning Trainer conducted the training over 100 epochs, with the best models saved via a ModelCheckpoint callback.

# 4.5.7. Prediction and Evaluation

After training, we loaded the best-performing model and used it to make predictions on the validation set. We utilized torchio's GridSampler and GridAggregator to efficiently compute the predictions over the entire volumes. The patches were overlapped and combined to form full volume predictions.

To evaluate the model's performance, Dice score was used. The Dice score, or the Sørensen–Dice coefficient, is a statistic used for comparing the similarity of two sets, and it's often used in image segmentation tasks. Here, the two sets are the predicted segmentation and the ground truth segmentation.

The Dice score is defined as follows:

DICE = (2 \* |X ∩ Y|) / (|X| + |Y|)

where:

* X and Y are the predicted and ground truth binary masks respectively.
* |X ∩ Y| is the common region between predicted and ground truth.
* |X| and |Y| are the total number of pixels in X and Y respectively.

The score ranges from 0 to 1 where a Dice score of 1 signifies perfect agreement between two samples.

In the code, the Dice score was calculated by first flattening the prediction and mask tensors, then calculating the sum of their element-wise multiplication (which is equivalent to the intersection of the two binary masks) and dividing it by the sum of the total number of elements in both tensors (which corresponds to the union of the two binary masks).

# 4.5.8. Visualization

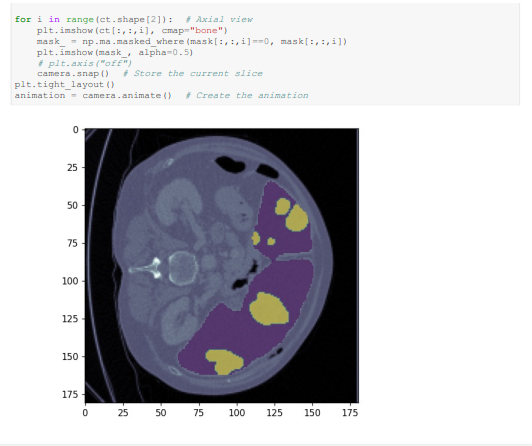
To visualize the segmentation results, we employed matplotlib and the celluloid library. For each volume in the validation set, we produced an animation that displayed the predicted and ground truth masks overlaying the original CT image for each slice of the volume. This visualization provided an intuitive understanding of the model's segmentation performance throughout the volume.

# 4.5.9. Logging with TensorBoard

Throughout the training and validation process, we logged loss values and images using TensorBoardLogger from PyTorch Lightning. Viewing these logs in TensorBoard enabled us to monitor the training progress, identify potential issues, and evaluate the model's performance.

# 4.6. Code

# 4.6.1. 01-DataInspection



# 4.6.2. 0-2 Model

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

# 4.6.3. 0-3 Train

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

# 4.6.4. 0-4 Evaluation

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

# Source

1. **https://paperswithcode.com/method/u-net**
2. **https://towardsdatascience.com/understanding-u-net-61276b10f360**
3. **https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/**
4. **https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network**
5. **https://www.ibm.com/topics/convolutional-neural-networks**
6. **https://www.arm.com/glossary/convolutional-neural-network**
7. <https://malsami.github.io/documentation/machine-learning/shallow-learning/shallow-learning/>
8. <https://datasciencedojo.com/blog/machine-learning-algorithms-explanation/>
9. <https://towardsdatascience.com/11-most-common-machine-learning-algorithms-explained-in-a-nutshell-cc6e98df93be>
10. <https://towardsdatascience.com/principal-component-analysis-pca-explained-visually-with-zero-math-1cbf392b9e7d#0226>
11. <https://www.simplilearn.com/what-is-pytorch-article>
12. <https://www.pytorchlightning.ai/index.html>
13. https://towardsdatascience.com/from-pytorch-to-pytorch-lightning-a-gentle-introduction-b371b7caaf09