7SENG013W Software Development Project
2022/23

Machine Learning for Liver Tumor Segmentation

Student: Markos Galikas (w1895147)
Supervisor: Dr. Ester Bonmati Coll
Date: 07/09/2023

MSc Software Engineering (Conversion)
School of Computer Science and Engineering
College of Design and Digital Industries
University of Westminster

Abstract

The proposed dissertation is about offering a way for doctors to perform diagnosis faster and better on a still standing problem in the medical field, the image segmantation of liver and liver tumors. The key reasons why automatic liver and liver tumor segmentation is still a work in progress are the considerable variety in their shape, appearance, and placement. This project is important because if offers a clear comparison of different machine learning models and how to perform hyperparameter tuning to increase the model's accuracy. The software is build with Python and it is very easy to be used by any doctor. Through research we found the appropriate dataset and performed preprocessing to prepare the dataset to perform better. Different machine learning model architectures were tested. The main difference of them was that of the encoder-decoder representation. Tuning was performed for the best model architecture of SegNet to improve its performance. The conclusion was that SegNet performed better than the rest of the tested models and tuning was of utmost importance to reach a state were the predictions are acceptable and close to other researchers results.

Keywords

Medical Imaging
Machine Learning
Segmentation
Liver tumor
UNet
Renet
SegNet

Declaration

This report is submitted in partial fulfilment of the requirements for the MSc Software Engineering (Conversion) Degree at the University of Westminster.

This report has been prepared based on my own work. Where other published and unpublished source materials have been used, these have been acknowledged in the text of the report and included in the list of references.

Word Count: 11186
Student Name: Markos Galikas
Date of Submission: 07/09/2023

Acknowledgements

I'd like to thank my supervisor, Dr. Ester Bonmati Coll, for her essential advice, continuous support, and mentorship during this journey. Her knowledge and encouragement have been invaluable in influencing my study.
I am also grateful to my family for their unending love, support, and understanding. Their unshakable faith in me has provided me with strength and motivation.
I consider myself extremely fortunate to have had such excellent people by my side, and their contributions have been important in my academic and personal development.

Contents

List of Figures

List of Tables

1 Introduction the project

1.1 Introduction

Liver is the largest, heaviest glandular organ and the second largest organ besides our skin. It weights about 1.5 kilograms and is located in the right upper quadrant of the abdominal cavity, below the diaphragm, to the right of the stomach, and overlying the gallbladder(Figure 1.1). It is responsible for up to 500 different functions including detoxification of the organism, and the synthesis of proteins. It is also the only organ in the human body that is capable of natural regeneration of lost tissue.

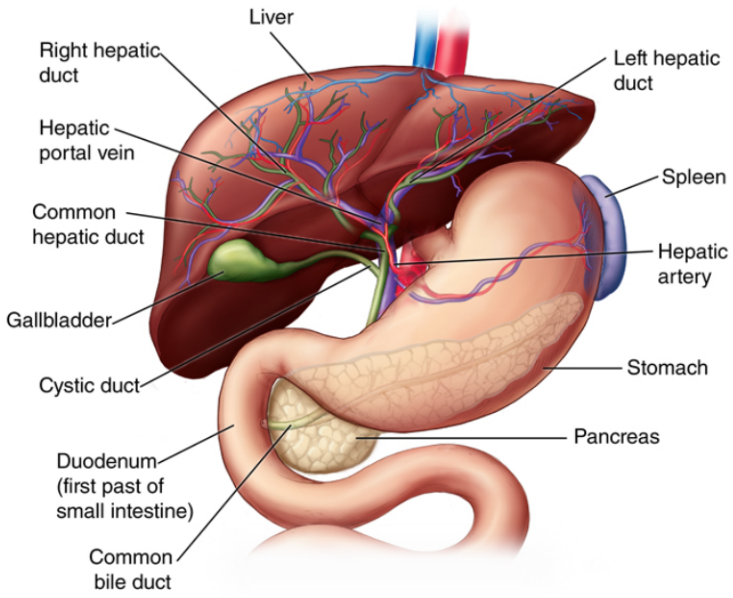

Figure 1.1: Liver anatomy and its parts

Because of its position and numerous functions, supporting all the other organs, it is prone to many diseases. Liver cancer is the third most common cause of cancer death globally, and it is among the five most common causes of cancer death in 90 countries across the world and is predicted to increase by 55% by 2040(WHO, 2022). The liver is a site for primary and secondary tumor as cancer is often metastasized from other organs.
In clinical routine, manual or semi-manual segmentation techniques are applied to interpret Computed Tomography(CT) and Magnetic Resonance Imaging(MRI) images that have been acquired in the diagnosis of the liver. These techniques, however, suffer from operator-dependent variations and are very

time consuming. The main reasons that up to this day the automatic liver and liver tumor segmentation is an open problem are that of high variability in their shape, appearance and localization. Challenges as a low-contrast between liver and lesion, different types of contrast levels (hyper/hypo-intense), abnormalities in tissues, size and varying number of lesions(Bilic Patrick et al., 2023) are still pressing. These can be attributed to varied contrast agent and the timing of their injection, different acquisition parameters, chronic liver disease, effects of the treatment.

The medical sector is experiencing a technological prosperity, and technical breakthroughs are allowing for the creation of new and inventive techniques to diagnose and treat numerous diseases, including cancer. Machine learning has emerged as a promising tool that has the potential to transform the way diagnoses are provided and overall healthcare is delivered. Many test are conducted on what machine learning model is the best, many more are being developed and the results are varying.

Machine learning allows computer systems to learn from data and make predictions based on it by applying algorithms and statistical models. Machine learning has the potential to significantly improve patient care in a variety of ways. For example machine learning algorithms are being trained on massive datasets of medical imaging such as X-rays, CT scans, and MRIs in order to detect malignant tumours earlier than traditional approaches. Google's DeepMind, for example, created a model that can analyze retinal images to detect signs of diabetic retinopathy. Moreover, benefits include more precise and prompt diagnosis by analysing the most effective medications with the fewest adverse effects by assessing a patient's genetic profile, tumor features, and treatment responses, identification of disease risk factors and drug discovery by investigating large databases of chemical compounds to determine which molecules are most likely to be useful in treating specific ailments. These shortens and lowers the cost of the healthcare system.

1.2 Project aims and objectives

This project proposes an investigation using machine learning in the medical industry, specifically for the illness of liver cancer. We shall concentrate on two major aspects. Implementing, and analyzing machine learning models that can find patterns from datasets and create synthetic or artificial data to augment the available datasets. We will train the machine learning model on the available datasets and finally on a combination of actual and synthetic data using the available methodologies, which will hopefully assist to increase its accuracy and generalization capabilities. This method may also eliminate the requirement for massive amounts of real-world data.

More specifically the aims and the objectives of the project are:

(PA1) Use machine learning models to segment liver and liver tumor.

PO1.1 Investigate and select a dataset.

PO1.2 Demonstrate an understanding of a significant programming language, such as Python.

PO1.3 Critical understanding of the issues and problems associated with the use of third-party packages and frameworks.

PO1.4 Implement data prepossessing techniques.

PO1.5 Develop machine learning models that can analyze the sets of medical data.

PO1.6 Analyze the effectiveness of the produced models.

PO1.7 Tune the best model for better performance.

1.3 System Architecture

To give an overview of the software system and the context it is intended to operate in, we include a system architecture diagram(Figure 1.2).

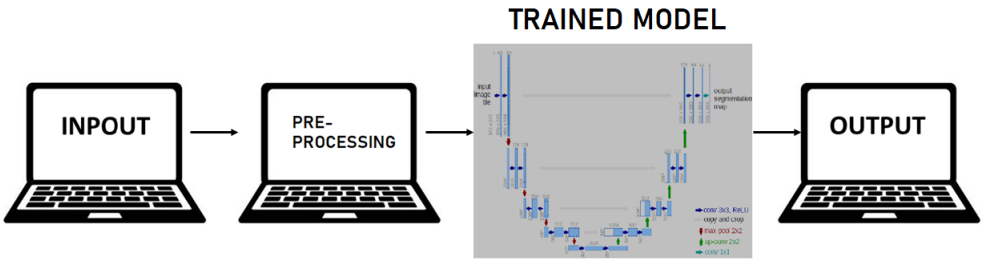

Figure 1.2: System architecture

This include the input data, which will be grayscale CT scan slices, the output, which will be the segmented image and the neural network architecture. In the Figure 1.2 we placed the UNet NN architecture, which more or less captures the whole number of system that we will be testing having an encoder-decoder architecture.

1.4 Project Resources

Programming language
This project will be coded with the programming language Python for the machine learning models and the creation of synthetic or fake data. To begin with, it provides a wide variety of libraries that make it simpler to create and use machine learning models. Secondly, it is a great option for beginners because it is straightforward to learn and has a basic syntax. Last but not least, Python includes a wide range of data processing and visualisation tools that are essential in machine learning projects.
IDE
Also installed must be an IDE of our choice for working with Python. We choose the Jupiter Notebook, because it is easy to work with and provides code interactivity and built-in visualization tools.
Specific tools and libraries for the ML models training, evaluating and visualizing must also be installed, such as TensorFlow, NumPy and Matplotlib. TensorFlow provides a high level of flexibility, allowing work with a wide range of data types and models. This comes related with other libraries such as NumPy and Pandas for preprocessing the data.

1.5 Outline of the report

Chapter 2
In this chapter we outline the related work that has previously being done with the most important the LiTS challenge. Also briefly discussed are the methods used to segment the liver and its lesions.

Chapter 3
Here we discuss the dataset used and its specific characteristics.

Chapter 4
This chapter is about the background knowledge we need to accomplice our aims and objectives. It has information about neurons, activations functions, backpropagation and gradient descent, convolutions, maxpooling, deconvolution, up-sampling and batch normalization.

Chapter 5
The next chapter is about the data preporcessing needed to effectively change and enhance the dataset.

Chapter 6
Chapter 6 introduces the system requirements. From system stakeholders to functional and non-functional requirements.

Chapter 7
To begin the design process, we will use case, sequence, and class/entity rela-

tionship diagrams.

Chapter 8
This chapter describes the different architectures and the criteria used to compare them. They are the U-Net, SegNet, ResNet18/50 and VGG16. MOreove the results of each network are displayed using the loss functions diagrams and displaying the models' predictions.

Chapter 9
Finally, in chapter 9 we perform a basic tuning changing the hyperparameters of the best model based on the results from the previous chapter. We change the batch size, teh learing rate and the resolution of the images.

Chapter 10
Concluding the project report we summarize and review the functional and non-functional requirements along with discussing further improvements to be made.

2 Background

2.1 Neurons

A neuron, also known as a node or a perceptron, is a key component of a neural network (NN) that accepts input signals, does calculations, and generates outputs. Essentially is a "thing" that holds a number between 0 and 1 or -1 and 1. The number is called its activation. Within a neural network, neurons are arranged in layers, providing the framework for information processing and learning.
A neuron's structure typically consists of the following elements:

- Neurons: can take in information directly from the input data or indirectly through other neurons. Each input has a weight assigned to it that denotes the significance or strength of the connection that exists between the input and the neuron.
- Weighted Sum: A neuron's inputs are multiplied by the appropriate weights before the outputs are added up. This weighted total accounts for how each input affects the output of the neuron.
- Bias: A bias term is frequently incorporated into a neuron to provide some degree of output flexibility. In addition to its own weight, the bias is an extra input to the neuron that aids in modifying the output value.
- Activation function: is used to add non-linearity to the neuron's output by passing the weighted sum—which includes the bias—through it. Based on the calculated total, the activation function chooses the neuron's ultimate output value. It aids in the representation of complicated relationships and

makes it possible for the neural network to pick up non-linear patterns in the data.

- Output: The result of the activation function becomes the neuron's output and is either used as the neural network's final output or sent to the next layer of neurons. Overall, activations in one layer determines the activations in the next layer.

Together, the neurons of a neural network process data, extract features, and provide predictions. The network's architecture is defined by how neurons are arranged and connected in different levels, and the number of neurons in each layer can change based on the difficulty of the job at hand.
The weighted sum for the first neuron at the second layer is: $x_0^{(1)} = w_{0,0} \cdot x_0^{(0)} + w_{0,1} \cdot x_1^{(0)} w_{0,n} \cdot x_n^{(0)} + b_0$, where 'w' is the weights, 'x' is the activations and 'b' is the bias,

2.2 Activation functions

While calculating the weighted sum of the neurons we can come with any number in a wide range. However we want this number to be between zero and one. This is the reason we squeeze with an activation function the output in the wanted range. In a neural network, an activation function is a mathematical function that adds non-linearity to a neuron's or layer of neurons' output.
There are tree main categories of activation functions. The step, the linear and the non-linear activation functions. Mainly we will use a non-linear activation function and specifically the rectified linear unit(ReLU)(Figure 2.1). The ReLU function sets all negative inputs to zero and keeps positive inputs unchanged. It provides a simple, computationally efficient activation and helps in addressing the vanishing gradient problem. It is as simple as setting the function $ReLU(x) = max(0, x)$.

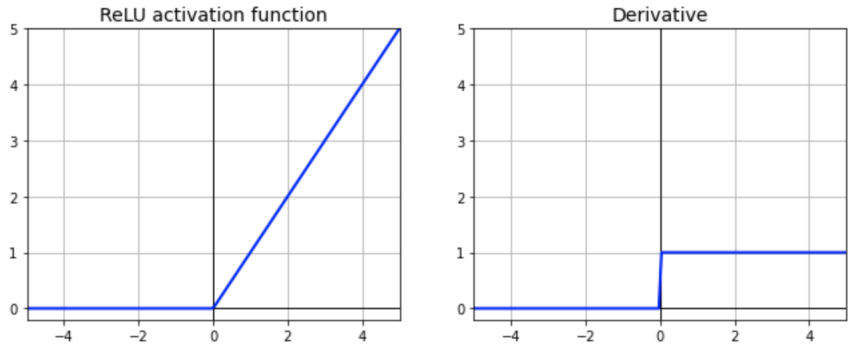

Figure 2.1: Graph for ReLU activation function and its derivative

Some of the drawbacks are that it is sensitive to weight initialization. A substan-

tial number of neurons may become inactive (outputting zero) during training if the initial weights are large and produce a negative weighted total for the majority of inputs. Another reason for that is when a significant gradient passes through, a neuron updates its weight and may end up with a large negative weight and bias(Luthfi Ramadhan, 2021). This neuron will always generate 0 if this occurs. The dead neurons, stuck in this state during training, stop learning and do not contribute to the network's output. To mitigate this problem variants of ReLU, such as Leaky ReLU exist to allow small negative values. The differentiation for the ReLU is relatively simple for the computation of neural network backpropagation. The sole assumption we make is that the derivative at the point zero is also zero(Bharath Krishnamurthy, 2022).
The final equation(Figure 2.2) for a neuron in the network including the weights, biases and activation function is: $x^{(1)} = ReLU(Wx^{(0)} + b)$

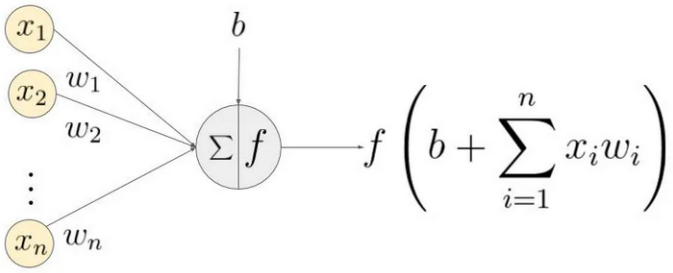

Figure 2.2: An example of a neuron. *Image taken from Temi Babs(2018)*

2.3 Backpropagation and Gradient Descent

Backpropagation works by propagating errors from the network's output layer back to the network's input layer, changing weights and biases along the way.
In more detail, firstly we have the forward pass. We initialize the weight and biases into random numbers and calculate the output layer and its prediction. Because we have a random input the output will not be what we expect. So, we define a cost function, meaning that the prediction is compared against the true label of the sample. The loss function used is determined by the task at hand, such as mean squared error for regression or cross-entropy loss for classification. Basically, the NN is a complicated function that from a number of inputs produces a number on outputs, by adjusting the weights and biases. On top of that the cost function from those weights and biases produces a single number(the average cost of all the training data) that measures how bad or good the predictions are. The lower the number the better and the goal is to minimize this function.
One note we should make at this point is about gradient descent. Gradient descent is an iterative optimization algorithm used to minimize a given function, typically the loss function, by adjusting the parameters of the model. it

is based on the observation that if the multi-variable function F(x) is defined and differentiable in a neighborhood of a point 'a' , then F(x) decreases fastest if one goes from 'a' in the direction of the negative gradient of F at 'a', namely $a_{n+1} = a_n - \gamma \nabla F(a_n)$, where γ is the learning rate (Wikipedia, 2023). While the gradient is indicating the direction the learning rate indicates how big of a step we should make to minimize the cost function. It is one of the most important hyperparameters, one that can lead to overshooting, oscillations and increasing cost or slow overall convergence.
The procedure of backpropagation begins by computing the gradient of the loss with regard to the output layer's parameters (weights and biases). The gradients of the loss with respect to the parameters of the preceding layer are then determined using the chain rule of calculus by multiplying the gradients of the current layer with the weights connecting the layers. This method is done layer by layer, backward across the network, until all layers' gradients are determined. With the gradients obtained, the neural network's weights and biases are modified to minimise the loss. The update is carried out with the use of an optimisation method, such as stochastic gradient descent (SGD), which updates the parameters based on the calculated gradients and a learning rate that regulates the step size. Mainly we will be using through out the project the Adam optimizer to perform the gradient descent, which includes adaptive learning rates based on first and second moments of the gradients.
These steps are repeated till a stopping conditions is met.

2.4 Convolution

Convolution is a fundamental technique in machine learning that is largely employed in convolutional neural networks (CNNs). Convolution is essential for extracting significant features from input data such as pictures, audio signals, and time series. CNNs may capture different types of patterns and features at different levels of abstraction by using convolutional layers with numerous filters and varied sizes. The features can then be combined and refined by subsequent layers in a CNN to create predictions or classifications.
Essentially, convolution is a mathematical technique in which two functions are combined to form a third function(Figure 2.3). It entails sliding a kernel over the input data and conducting element-wise multiplication and summation to build a feature map.
The general structure that we will be using is from the Tensoflow library:

Conv2D(filters, kernel_size, strides=(1, 1), padding='same', activation='relu')

- Filters: Numerous filters are usually utilised at the same time. Each filter extracts a separate set of features from the input image, resulting in a number of feature maps.
- Kernel Size: A small matrix (usually a square one) with learnable weights is defined as a kernel. The filter is often way smaller in size than the input image. The convolution operation is performed with The kernel

sliding across the picture and applied to the input image. At each place, element-wise multiplication between the kernel and the relevant portion of the input image is performed. In the output feature map, the resulting values are then totaled to produce a single value.

- Strides and Padding: The step size employed while sliding the filter, determining the amount of overlap between consecutive sections, is referred to as the stride. The technique of adding extra border pixels to the input image in order to preserve its spatial dimensions is known as padding. Stride and padding are used to control the output size. We can have 'valid' or 'same' padding.

For example having an image of 256x256x3, with a 4x4 kernel, stride 1, and no padding, the output feature map size will be reduced in both dimensions by (kernel_size - 1) = (4 - 1) = 3. Therefore, the resulting feature map will have dimensions of 253x253x3. With a stride of 2, the kernel will move by 2 pixels at each step while sliding across the input image. Assuming again no padding is used, the output feature map size will be determined by the following formula: output_size = (input_size - kernel_size) / stride + 1. Therefore, the resulting feature map will have dimensions of 127x127x3. If the number of filters is set to 64, it means that during the convolution operation, 64 different filters (kernels) will be applied to the input image. Each filter will generate its own feature map as a result. So finally we have the output feature map of 127x127x64.
It is vital to remember that the number of filters is a convolutional layer hyperparameter that may be modified based on the complexity of the task and the desired depth of the network. Increasing the number of filters allows the network to learn more complex features, but it also increases the processing needs.

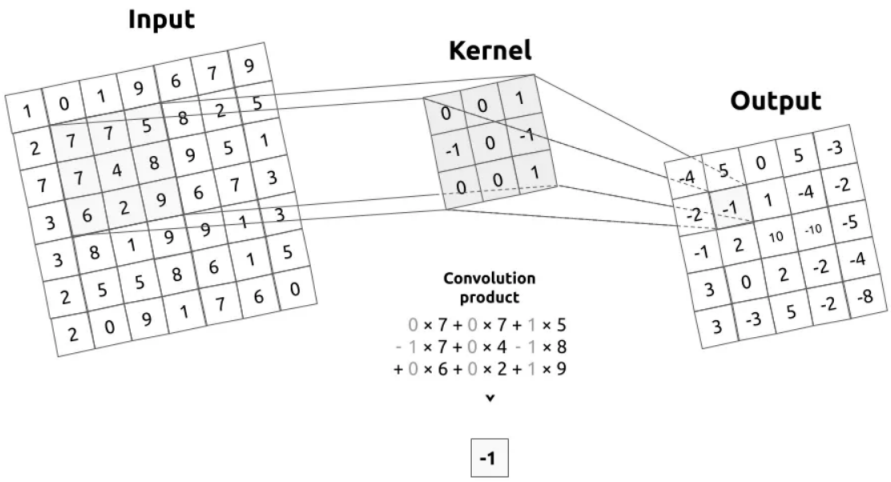

Figure 2.3: Convolution operation. *Image taken from Axel Thevenot(2020)*

2.5 Maxpooling

The primary principle underlying max pooling is to keep the most critical or distinguishing traits while rejecting less important details. Max pooling helps to capture the most dominant features while offering a degree of translation invariance because the largest value signifies the presence of a feature independent of its exact position within the pooling zone.
Max pooling is commonly used after convolutional layers to minimise the spatial dimensions of the data while maintaining the most important features. It aids in the management of overfitting, the reduction of computational complexity, and the provision of a type of spatial hierarchy in feature representations, all of which can contribute to increased efficiency and generalisation in CNNs.

2.6 Deconvolution

In convolutional neural networks (CNNs), the transpose operation, also known as transposed convolution, upsampling or deconvolution, is the inverse operation of the ordinary convolution operation. Conv2DTranspose, as named in the Tensorflow library, seeks to enhance the spatial dimensions of the input, whereas convolution diminishes them.
Conv2DTranspose is frequently used in image creation, upsampling, and semantic segmentation applications. It enables the network to learn how to upsample feature maps, reconstruct high-resolution images, and provide fine-grained details.
Conv2DTranspose aids in the recovery of spatial information lost during prior layers' downsampling techniques, such as max pooling or regular convolution. It allows the network to build higher-resolution feature maps or images, allowing it to capture finer features and provide a more accurate representation of the input. The general structure that we will be using is:

Conv2DTranspose(filters, kernel_size, strides=(2, 2), activation='relu', padding='same')

Assuming we use a specific set of parameters (e.g., kernel size of 4x4, stride of 2, and no padding) for the Conv2DTranspose operation, the resulting output dimensions would be: output_width = (input_width - 1) * stride + kernel_size = (127 - 1) * 2 + 4 = 254. The same with the output_height = 254.

2.7 Up-Sampling

Conv2DTranspose and UpSampling2D are both used in neural networks to upsample or increase the spatial dimensions of feature maps. They do, however, have separate functions and are frequently employed for distinct purposes.
Conv2DTranspose is frequently used for learning an upsampling process from data and filling in gaps between original data points. picture segmentation, picture-to-image translation, and image production (generative adversarial networks - GANs) are all common applications. UpSampling2D is utilized when a

simple and efficient upsampling technique is required without the need to understand extra parameters. It can be used to increase the resolution of feature maps before additional processing or when a basic interpolation-based upsampling is adequate.
UpSampling2D is a straightforward upsampling layer that just repeats the rows and columns of the input feature maps without learning any new parameters. It doesn't add any new learnable parameters or perform any convolutional operations. UpSampling2D is a deterministic procedure that does not suffer from the checkerboard artifacts that Conv2DTranspose does. Because it does not include any additional learnable parameters, UpSampling2D is less computationally expensive than Conv2DTranspose.

2.8 Batch Normalization

Internal covariate shift is the change in the distribution of layer inputs that occurs during deep neural network training. It occurs when the changing parameters of the preceding layers impact the distribution of input to each layer throughout the learning process.
As a network learns, the gradients computed from later layers are used to change the parameters of earlier layers. As a result, with each update, the input distribution to each layer moves. Internal covariate shift refers to this shift in distribution.
Batch normalization is introduced specifically to combat this problem. By normalizing the inputs, batch normalization ensures that the distributions of layer inputs remain more stable during training. This helps to improve training stability, convergence speed, and overall network performance.

3 Related work

One of the most important piece of work is the Liver Tumor Segmentation benchmark. It was organized in 2017 in conjunction with the IEEE International Symposium on Biomedical Imaging (ISBI) and the International Conferences on Medical Image Computing and Computer-Assisted Intervention (MICCAI). The aim of this challenge was to enhance and test different segmentation techniques on a more well defined dataset, as none of the previous attempts to collect medical images contained well-defined cohorts of patients with lesions, and there was no segmentation of the liver and its lesions.The results indicated a need for further research.
Previous attempts include SLIVER07, the first large segmentation challenge that was held at MICCAI 2007, the LTSC'08 segmentation competition at MICCAI 2008, the ImageCLEF 2015, the VISCERAL challenge and the most current CHAOS challenge.
In these attempts to segment the liver and the liver tumors many techniques, including atlas-based models, graphical models, and deformable models, traditional machine learning methods, for the automatic segmentation has been

conducted. Although these model-based methods can produce outstanding segmentation results, they frequently require the use of patient-specific parametric stages, which prevents the models from being more widely applied(Hyunseok Seo et al., 2020). Convolutional neural networks(CNNs) and Fully Convolutional Neural Networks(FCNs) have recently produced important advances in image segmentation. Numerous attempts have been made to apply these achievements to the segmentation of liver tumors in line with this trend.

4 Dataset

When compared to other organs, the liver datasets that are currently accessible either have a minimal number of images and reference segmentation, or none at all.In this project we used the training dataset from the LiTS challenge containing 131 contrast-enhanced abdominal CT scans coming from 7 clinical institutions(Figure 4.1). The CT scans come with manually reference annotations of the liver and tumors done by a radiologist with more than 3 years of experience in oncologic imaging. The in-plane image resolution ranges from 0.56 mm to 1.0 mm, and 0.45 mm to 6.0 mm in slice thickness. Also, the number of axial slices ranges from 42 to 1026. The number of tumors varies between 0 and 12. The size of the tumors varies between 38 mm^3 and 1231 mm^3(Bilic Patrick. et al., 2023 ; Grzegorz Chlebus et al., 2018). The dataset can be found in the popular website Kaggle(2023).

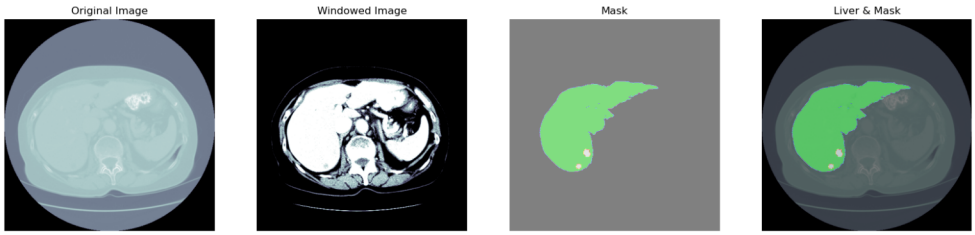

Figure 4.1: Slice of a CT liver image from LiTS challenge

4.1 Data preprocessing

For each file used, we read the CT scans(Figure 4.2) using a custom read_nii function. This functions reads the grayscale NIfTI files into an array while rotating it by 90 degrees the image. The files are images of dimensions 512x512. The code then resizes the images to 128x128 while keeping the aspect ratio, crops them into squares, and turns them to tensors. The image tensors that have been scaled and cropped are appended to separate lists for further processing. This is done because the computational capabilities of the PC used is not that high. Additional processing is made by normalizing the values of the train data. For the input data we subtract the minimum value from each element and divide it by the range of train data values (the difference between the maximum and

minimum values). This normalization step scales the values in a range of [0.0, 1.0], ensuring that the data is within a consistent and manageable range for subsequent model training and evaluation. The values of the mask(Figure 5.2) are three distinct integer numbers, meaning 0(background), 1(liver) or 2(tumor). We also add an extra dimension to the tensors along the last axis. The purpose of this step is to indicate that the data has a single channel(grayscale), The extra dimension is added to conform to the expected input shape for training. We also disregard any slices that are empty by calculating the mean value or the image. If it is zero we discard it. Also we perform enhancement(windowed) of the images to accentuate the regions where the liver is.
The data we have are separated into training and testing batches. The data in the batches are from different patients. Both of them receive the same data prepocessing(Appendix A). We use 616 slices(78%) for training and 176(22%) slices for testing.

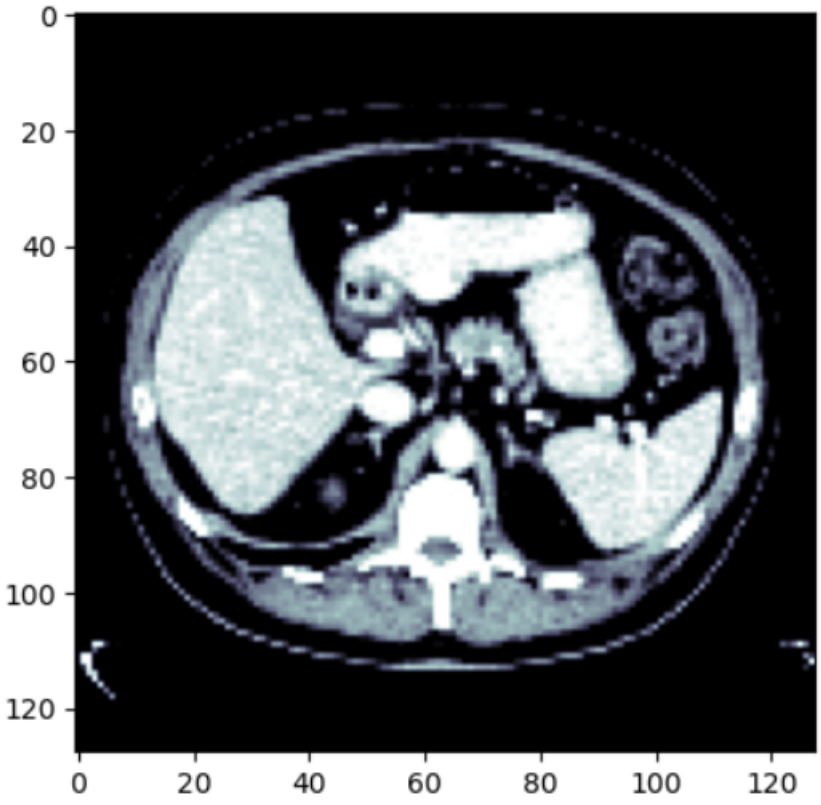

Figure 4.2: CT slice after preprocessng

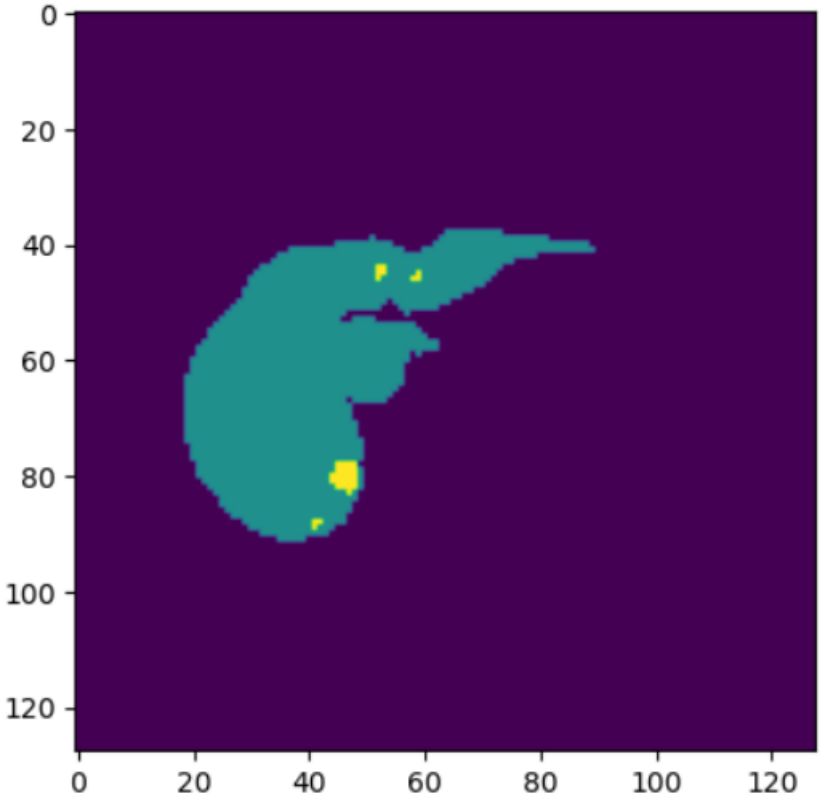

Figure 4.3: Mask image after preprocessng

5 System Requirements

5.1 Introduction

We conduct a stakeholder study to better understand the needs and expectations of major stakeholders both inside and outside the project environment. Understanding the attributes and their interrelationships helps us design our project strategically.
Understanding, extracting, and solidifying project requirements in recorded form is one of the most difficult components of a project. We will concentrate on system requirements that are both functional and non-functional.

5.2 System Stakeholders

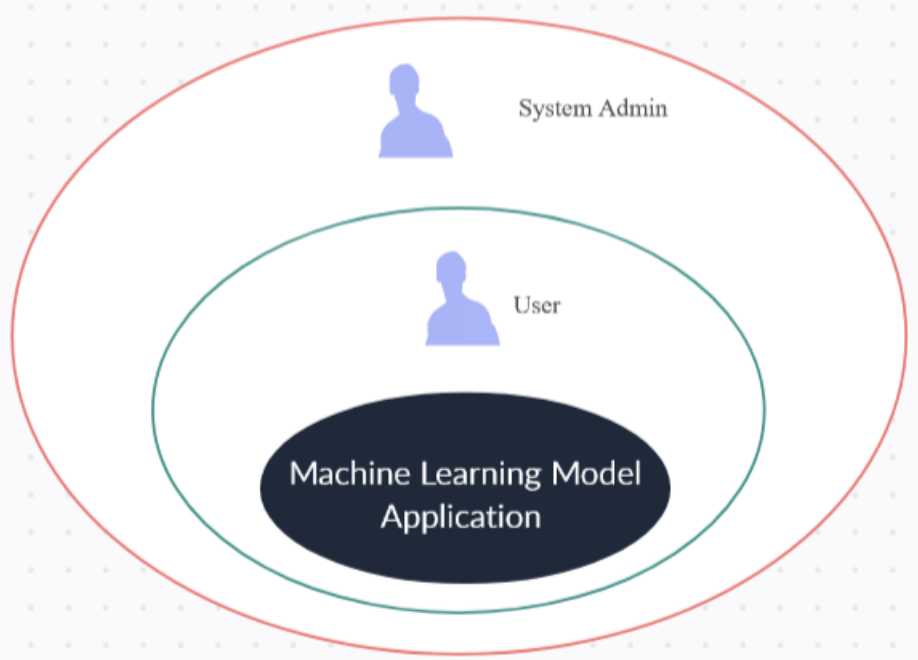

Figure 5.1: Stakeholders onion diagram

User
The User is the person or entity who uses or consumes the machine learning model's predictions or outputs. They engage with the model in order to obtain the required outcomes. Individuals, companies, or any entity that benefits from the model's capabilities are all examples of users. Our main machine learning model application user will be a healthcare professionals (doctors or medical practitioner) who use machine learning models for diagnosis or personalized treatment recommendations.

System Admin
The Admin is in charge of monitoring, maintaining, and supervising the machine learning model over its entire existence. They are in charge of its creation, deployment, monitoring, and maintenance. The Administrator function is critical for ensuring that the model runs effectively and fits with company goals. The Admin's tasks may include:

- Model Development: including working with data scientists and engineers to create, train, and validate a model using relevant datasets and methods.
- Model Monitoring: by keeping track of the model's performance and be-

havior in real-world circumstances, recognizing errors or abnormalities, and taking appropriate action.

- Model maintenance: by frequently updating the model to reflect changing data distributions and business requirements.
- Model governance: by ensuring that the model adheres to ethical and legal norms, as well as data protection and security rules.

5.3 System Context

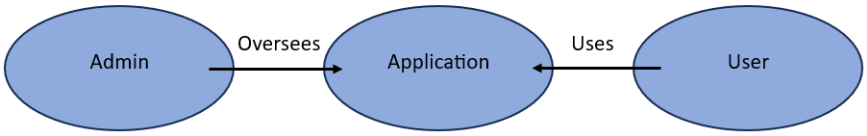

Figure 5.2: System Context

The main application is the machine learning model and its predictions based on the input images it receives. The system administrator oversees the application and ensures that everything is running smoothly. If not the admin perform corrective actions and updates the application. On the other hand the user is the doctor how inputs the patient's data and performs the analysis.

5.4 Functional Requirements

5.4.1 Essential Requirements

FR1 Develop and implement machine learning models that can analyze sets of medical data.

FR2 Segment successfully the CT images from the dataset.

FR3 Evaluate the effectiveness of the developed machine learning models

5.4.2 Desirable Functional Requirements

FR4 Analyze the effectiveness of the produced models and tune them for better performance.

5.4.3 Luxury Functional Requirements

FR5 Create synthetic or fake data to augment the available datasets and reduce the need for large amounts of real-world data.

FR6 Examining if the synthetic data generation techniques impact the accuracy and generalization capabilities of the models.

5.5 Non - Functional Requirements

5.5.1 Essential Non - Functional Requirements

NFR1 A suitable development environment to work on the code. Python is our choice language for machine learning, and we chose to work with an IDE of Jupyter Notebook.

NFR2 Critical understanding of the issues and problems associated with the use of third-party packages and frameworks.

NFR3 A solid understanding of machine learning algorithms and techniques, such as supervised and unsupervised learning, regression, classification, and clustering.

NFR4 Familiarity with relevant machine learning libraries and frameworks, such as Scikit-Learn, TensorFlow, Keras, and PyTorch.

NFR5 Access to cancer datasets, from publicly available sources. We will need to ensure that the data is accurate and reliable, and that we have the necessary permissions to use it.

NFR6 Good understanding of ethical and legal considerations related to working with medical data and artificial intelligence, including privacy and security regulations, and avoiding harm and bias.

NFR7 Familiarity with data analysis and data preprocessing techniques, such as data cleaning, feature extraction, and feature scaling.

5.5.2 Luxury Non - Functional Requirements

NFR8 Knowledge of synthetic data generation techniques, such as generative adversarial networks (GANs), autoencoders, or other methods to create synthetic data.

NFR9 Patience, perseverance, and attention to detail, as machine learning projects can be complex and challenging and may require multiple iterations and adjustments to achieve good results.

6 System Design

6.1 Introduction

We will be using case, sequence, and class/entity relationship models to begin the design process.This seeks to offer a clear picture of entities, relationships between them, and procedures required.
Using the description provided we create a use case diagram for the machine learning application. We identify and characterise the actors shown in the use case diagram, along with their connections (actor/actor, actor/use case, and use

case/use case).
Actor/Actor: This relationship indicates that two actors in the system can communicate or interact with each other. For example, the User actor and the Admin actor can communicate to collaborate on the project.
Actor/Use Case: This relationship indicates that an actor in the system can perform a specific use case. For example, the User actor can perform a diagnosis.
Use Case/Use Case: This relationship indicates that one use case can trigger another use case in the system. For example a report of an error can trigger a fix of the program application.

6.2 Use Case Diagram

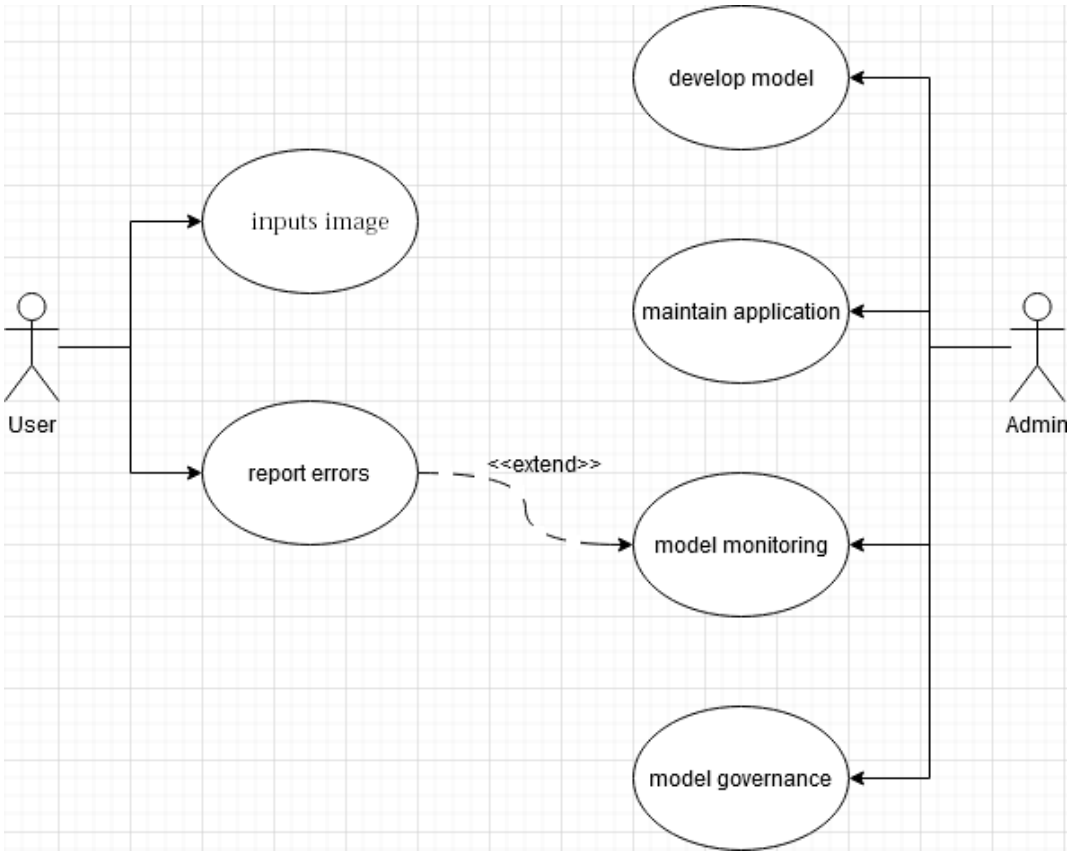

Figure 6.1: Use Case Diagram

The system has two users(Figure 6.1). One is the admin and the other is the doctor/user. The admin can perform actions like develop the model and monitor the system and the doctor can perform image insertion and report error that

occur. The last function is an extension of the system monitoring.

6.3 Use Case Descriptions

Use Case 1: Diagnosis

Brief Description	This use case allows a doctor to insert a patient's scan and perform diagnosis.
Actors	User
Preconditions	• The patient has performed a CT scan. • The system is up-to-date.
Postconditions	Doctor has performed a diagnosis.
Basic Path	1. The doctor uses the machine learning diagnostic system. 2. The doctor inserts the scan. 3. The machine learning model processes the patient's data and generates a diagnosis prediction. 4. The doctor reviews the diagnosis and, if necessary, validates it based on their medical expertise and knowledge of the patient's condition.
Alternative Path	If the system encounters technical issues or errors during the diagnosis process, the doctor may report the problem to the system administrator or use alternative diagnostic methods. The doctor may disagree with the system's diagnosis due to factors not captured by the model. In such cases, the doctor can provide their own diagnosis and treatment plan based on their expertise and judgment.

Table 1: Use case 1: Diagnosis

Use Case 2: Report Errors

Brief Description	This use case allows a doctor to report possible errors.
Actors	User, Admin
Preconditions	• The doctor has access to the machine learning model. • The system is up-to-date.
Postconditions	Doctor reported the error to the Admin.
Basic Path	1. The doctor uses the machine learning model to acquire predictions or recommendations for the patient's condition during a medical diagnosis. 2. The mistake report, including the documentation provided by the doctor, is reviewed by the system administrator. 3. If the error is validated, the admin changes the machine learning model, modifies its parameters, or inserts additional data based on the study to improve its accuracy and dependability. 4. The new model is validated and tested to ensure that the issue has been corrected and that it operates as expected. 5. The model is redeployed to the medical environment after it has been updated and validated, and the system administrator certifies its proper functionality.
Alternative Path	If the reported error is the result of poor data quality or insufficient training data, the admin may collaborate with the doctor to locate relevant data and incorporate it into the training process. In some circumstances, the doctor's report could be based on a rare or unique medical issue that the model has never seen before. To increase the model's effectiveness in handling rare circumstances, the data scientist may try to incorporate such cases in the model's training dataset.

Table 2: Use case 2: Report Error

Use Case 3: Develop Application

Brief Description	This use case allows the admin to develop the model.
Actors	Admin
Preconditions	• The admin has access to the machine learning model parameters.
Postconditions	Admin successfully created the application and the application provides accurate and reliable predictions
Basic Path	1. The administrator works with stakeholders such as business owners, end-users, and domain experts to find a use case or problem that can be solved by machine learning. 2. The administrator chooses relevant machine learning algorithms and approaches for the specified use case and designs the machine learning application's architecture. 3. The prepared data is used to create and train machine learning models. 4. The program is tested and quality assured to discover and correct any potential bugs or difficulties. 5. The program is deployed to a production environment and made available to end users after successful testing.
Alternative Path	If data is few or of poor quality, the administrator may investigate data augmentation techniques or locate alternate data sources to improve model performance. The administrator may face issues related to model interpretability, fairness, or ethical considerations during the development process.

Table 3: Use case 3: Develop Application

Use Case 4: Maintain the System

Brief Description	This use case allows the admin to maintain the model.
Actors	Admin
Preconditions	• The machine learning application has been deployed to a production.
Postconditions	The admin obtains a clear understanding of the application's performance.
Basic Path	1. The administrator looks into the root cause of any observed abnormalities, attempting to determine whether they are the result of application code errors. 2. Based on the findings of the investigation, the administrator may take corrective action to remedy the concerns as soon as possible. 3. To guarantee that the machine learning models continue to generate credible predictions, the admin monitors model performance parameters such as accuracy, precision, recall, and other assessment metrics.
Alternative Path	In the event of extended performance decline, the administrator may engage to assess and enhance the machine learning models for improved efficiency.

Table 4: Use case 4: Maintain the System

Use Case 5: Monitoring of the System

Brief Description	This use case allows the admin to monitor the model.
Actors	Admin
Preconditions	• The machine learning application has been deployed to a production.
Postconditions	The admin monitors the system performance.
Basic Path	1. In reaction to reports or aberrant metrics, the administrator analyzes the source of the problems. Collaboration with system administrators and data scientists may be required to determine the source of performance degradation or resource limitations. 2. To remedy the detected problems, the administrator takes corrective measures such as restarting services, optimizing system configurations.
Alternative Path	

Table 5: Use case 5: Monitoring of the System

Use Case 6: Model Governance

Brief Description	This use case allows the admin to enforce data governance.
Actors	Admin
Preconditions	• The machine learning application has been deployed to a production. • Model governance policies and guidelines have been developed and shared with the development team.
Postconditions	The admin ensures that the machine learning models comply with the defined model governance policies and regulations.
Basic Path	1. The administrator works with engineers to create model governance policies. 2. If potential biases or concerns with fairness are discovered, the administrator fixes them. 3. To maintain accuracy and relevance, the admin ensures that models are updated and re-validated at suitable intervals.
Alternative Path	When a data breach or security incident involving model data occurs, the administrator takes prompt steps to safeguard the models and assess the impact on user privacy. If new legislation or ethical principles are implemented, the administrator works to modify the model.

Table 6: Use case 6: Model Governance

6.4 Sequence diagrams

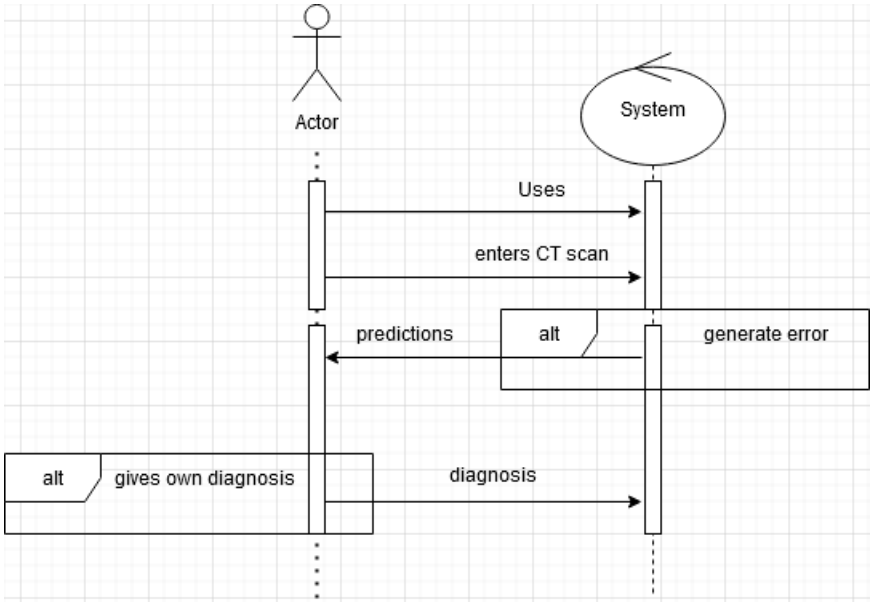

Figure 6.2: Use Case 1: Image insertion

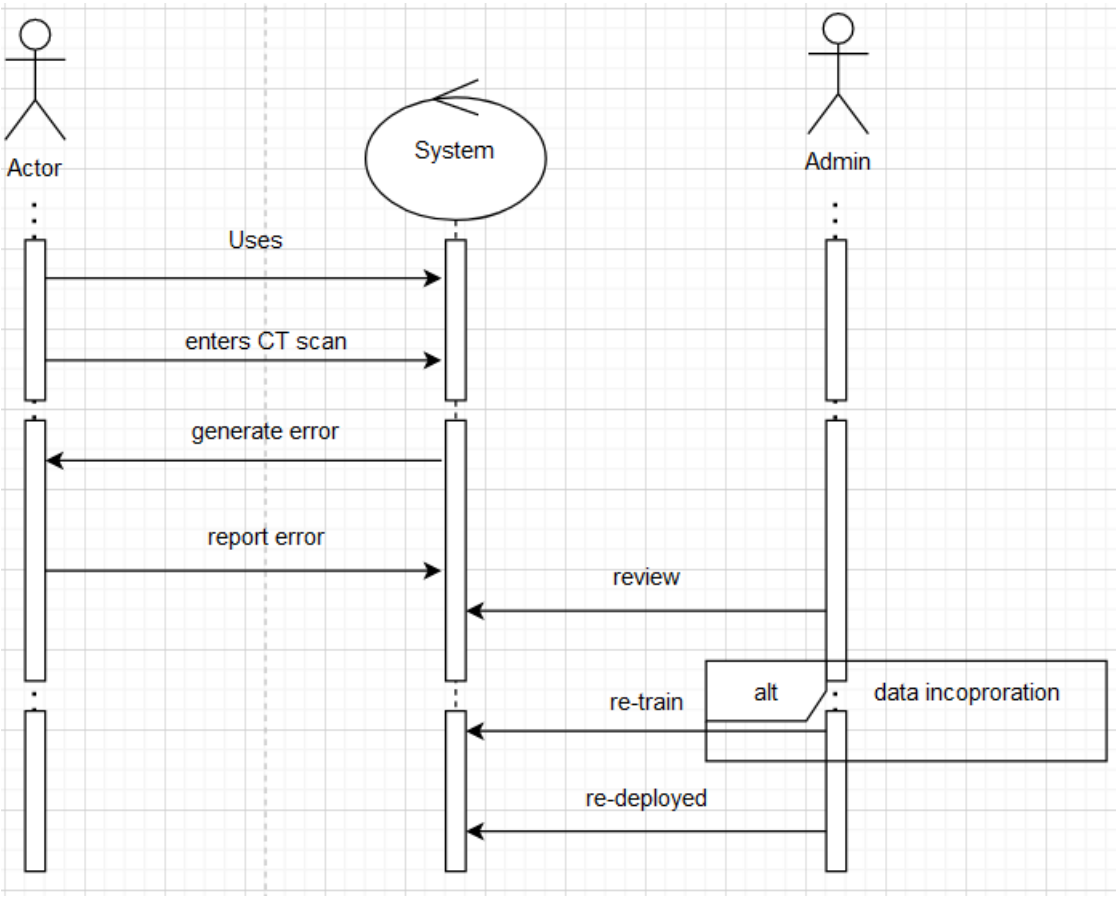

Figure 6.3: Use Case 2: Reporting errors

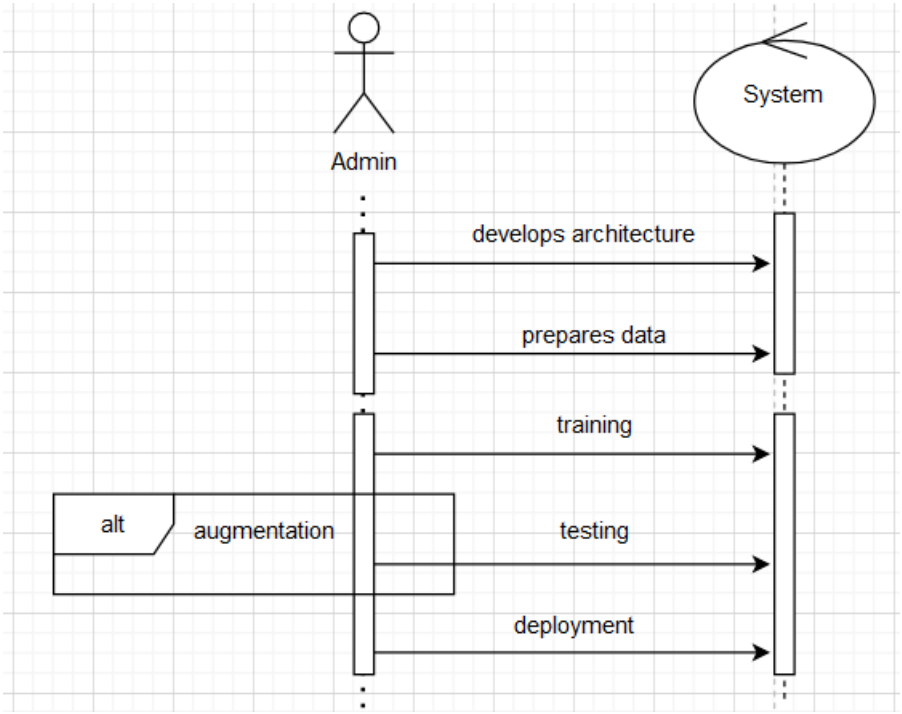

Figure 6.4: Use Case 3: Developing the model

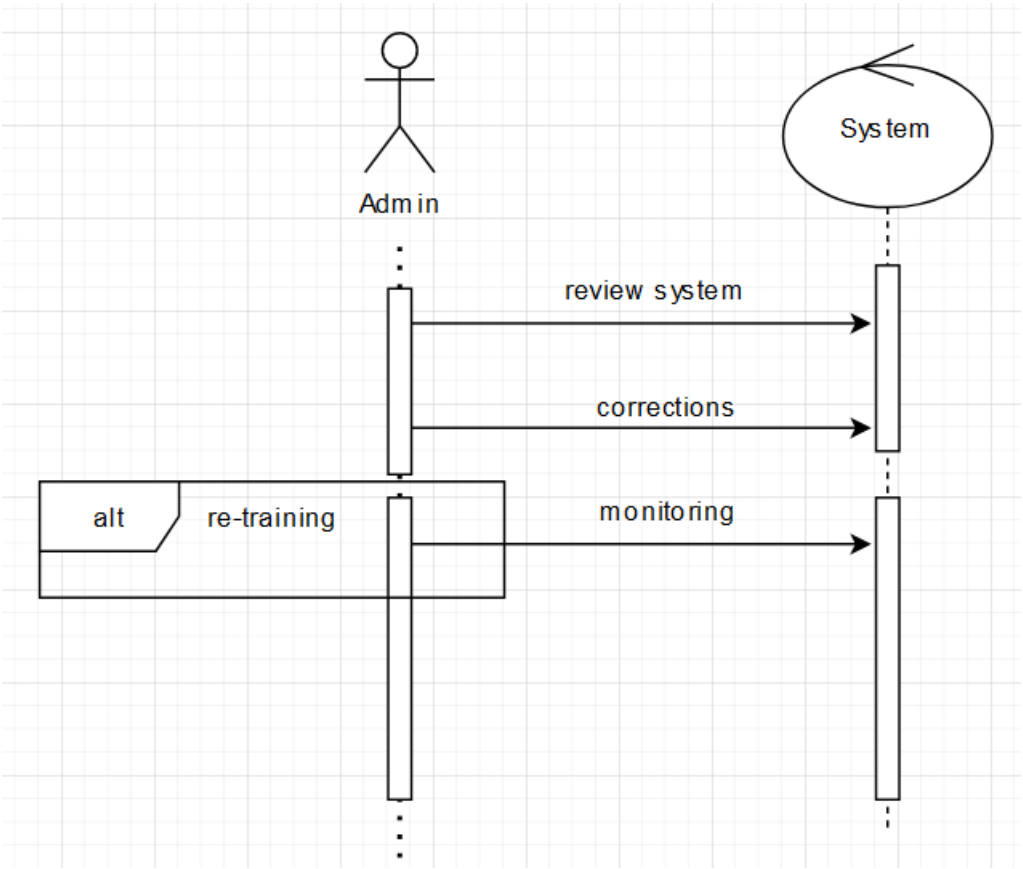

Figure 6.5: Use Case 4: Maintain the model

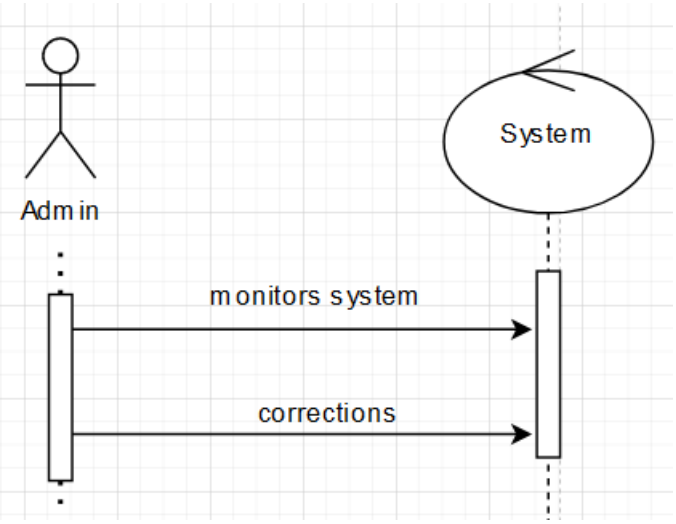

Figure 6.6: Use Case 5; Monitoring of system

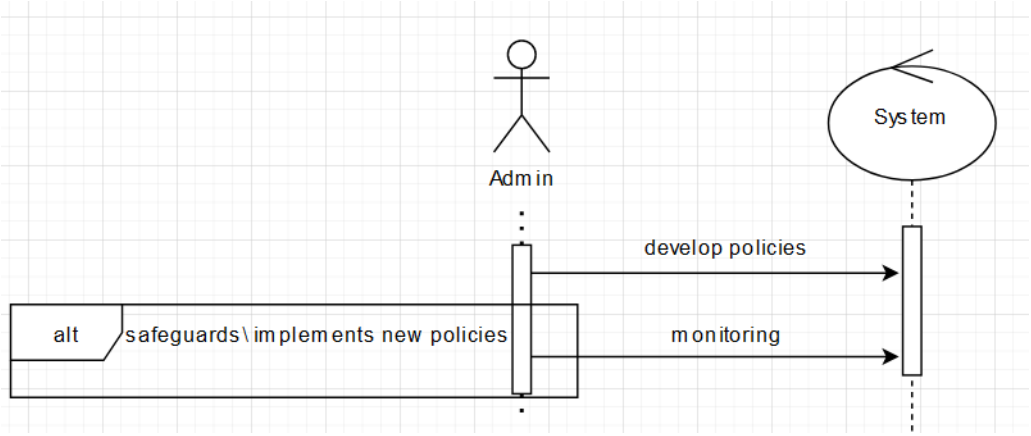

Figure 6.7: Use Case 6: Governance

6.5 Class Diagram

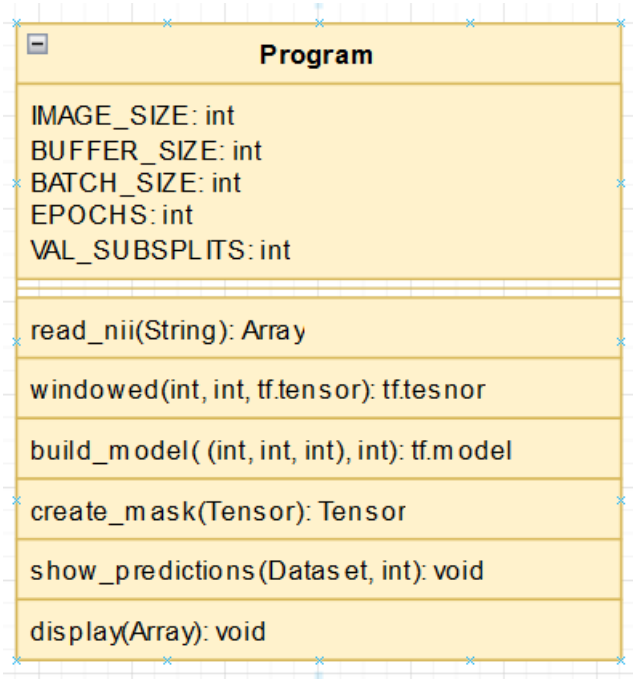

Figure 6.8: Class Diagram

The class diagram is depicted in Figure 6.8. It has some constants like batch size and epochs which are displayed with uppercase letters. The Dataset and Tensor variables are from the Tensorflow library.

7 Implementation

7.1 Introduction

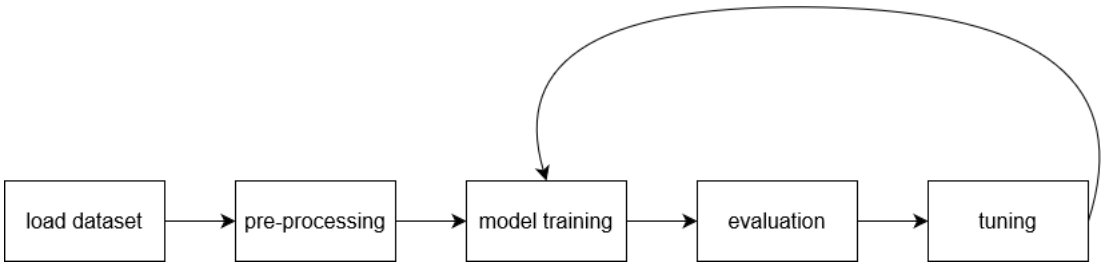

Figure 7.1: Methodology followed

In this project we will deal with four different network architectures. All of them have more or less the same structure. They consist of an encoder and a decoder. In a network design, the encoder and decoder components collaborate to learn from the input data in the encoder and then to reconstruct or synthesize the intended output in the decoder. This hierarchical feature extraction and reconstruction approach allows the model to acquire meaningful representations and generate accurate predictions in a variety of tasks, including picture synthesis, image segmentation, and more.
In more detail the encoder is in charge of converting the input data into a lower-dimensional or compressed representation that captures the most significant elements. The encoder in CNNs is typically composed of convolutional layers, followed by pooling or downsampling layers. These layers extract features with learnable filters and lowering spatial dimensionality. The encoder's convolutional layers are designed to extract hierarchical features and capture local patterns, gradually encoding the input data into a more abstract form. As the spatial dimensions reduce, the number of filters in the convolutional layers tends to rise, allowing the encoder to capture more complicated and high-level data. The encoder takes in an input sequence and produces a representation of it, often referred to as a hidden or latent representation.
The decoder on the other hand is in charge of reconstructing or the output data from the encoder's compressed form. The decoder in CNNs is often composed of transpose convolutional layers and usually employs skip connections. Transpose convolutional layers upsample the compressed representation by increasing spatial dimensions while decreasing the number of channels. Based on the encoder's learnt features, the decoder layers gradually recover spatial details and reconstruct the output data. Skip connections, aid in the propagation of characteristics from the encoder to the decoder, allowing for more precise object localisation and reconstruction. The output is produced by the final layer of the decoder.

7.2 Architectures to be Used

Several architectures have been widely employed and have demonstrated outstanding performance when examining neural networks for image segmentation. Here are some popular neural network topologies for image segmentation tasks:

- U-Net
- SegNet
- ResNet
- VGG

7.3 Review Criteria

The following criteria will be used to review the selected systems in order to influence the design process and assist with best practices.

RC1 Model Complexity

RC2 Time Training and Convergence

RC3 Accuracy / Dice score

7.4 Review of Neural Networks

7.4.1 U-Net

The U-Net architecture(Figure 7.2) is a well-known convolutional neural network (CNN) architecture for image segmentation. Olaf Ronneberger et al. announced it in 2015, and it has since become widely used in a variety of medical imaging applications.
The network consists of a left(downsampling) and a right(upsamping) path. In the middle there is the bottleneck layer and all together form a 'U' shape, hence the name. It is composed of two 3x3 convolutions applied four times, each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. We double the number of feature channels with each downsampling step. Every step(four steps in total) in the expansive path begins with an upsampling of the feature map, followed by a 2x2 upconvolution that cuts the number of feature channels in half, a concatenation(skip connection) with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. Finally, a 1x1 convolution is employed at the final layer to transfer each feature vector to the desired number of classes. In total 23 convolutional layers comprise the network.

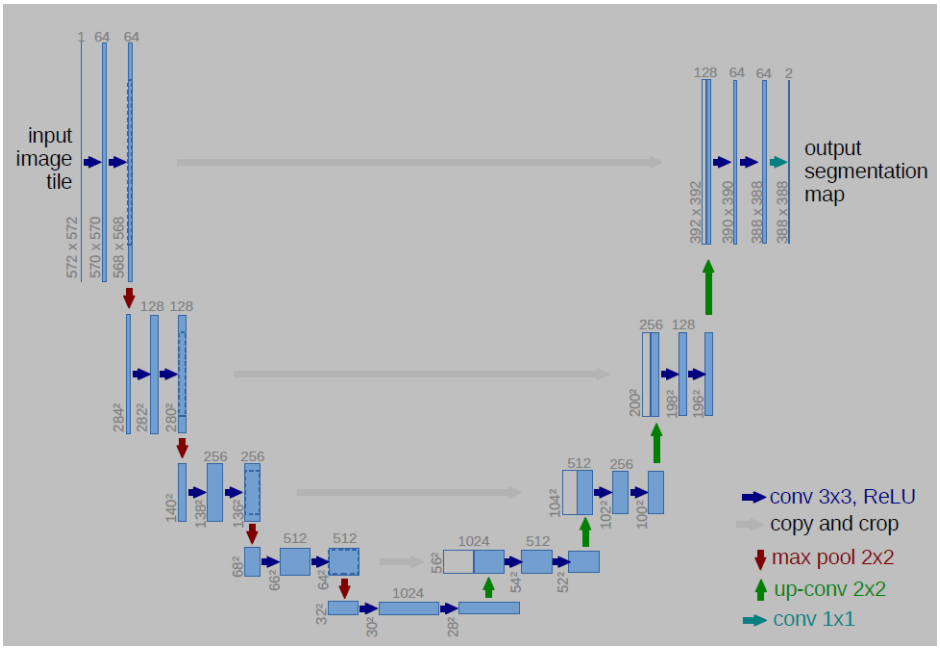

Figure 7.2: UNet model architecture

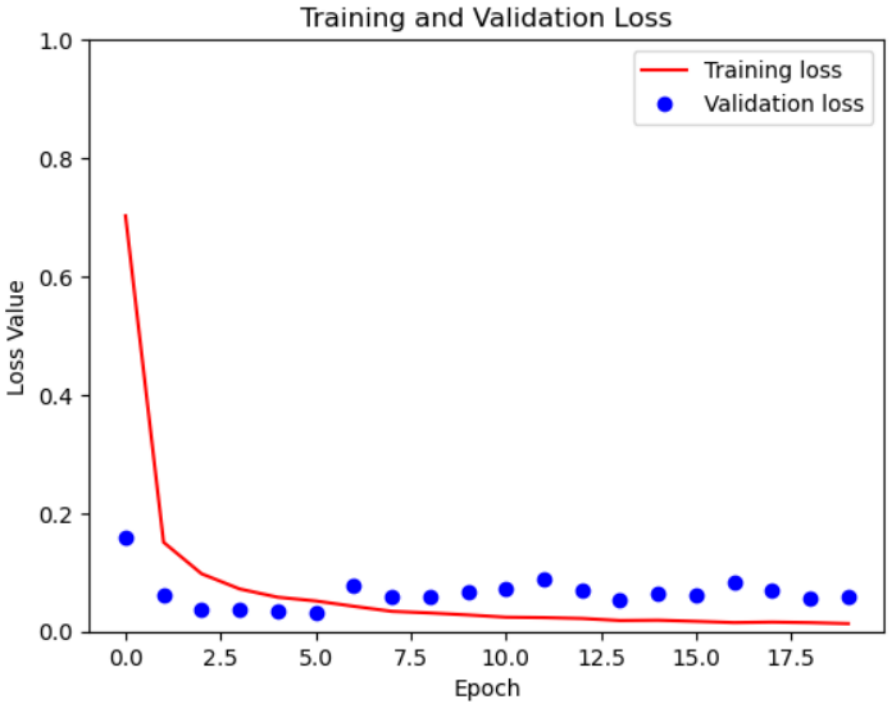

Figure 7.3: UNet Model Loss

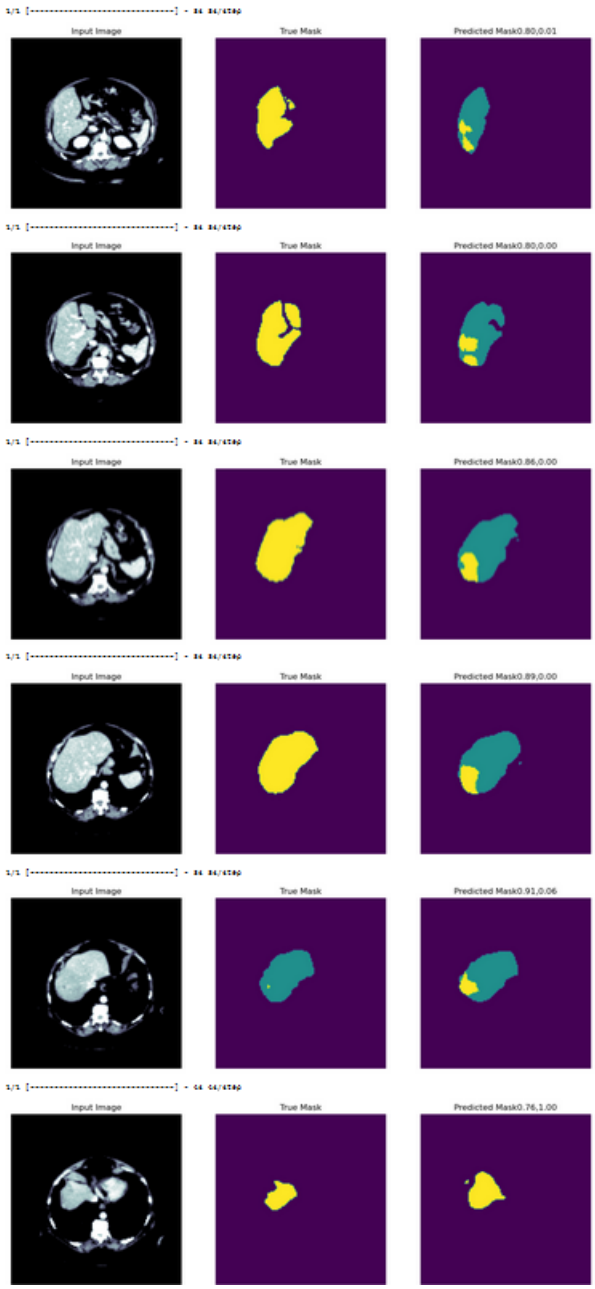

Figure 7.4: Predictions of UNet Model

The first column of the Figure 7.4 are the CT slices. The second column is the mask and the third column's images are the predicted masks. We see that we

don't have a good prediction of the liver and the tumor. This is also evident from the Dice scores we calculated for the overall model and for image separately. The Dice scores for liver and tumor are above each predicted mask.

7.4.2 Residual Networks

One can think that the deeper the network the better the result we get from training. And this is partially true. The network is able to capture more details from the training data and the depth is of crucial importance. But stacking layers not always results to better performance. In traditional learning approaches, each layer in a neural network is expected to learn a full mapping from its input to its output. However, for very deep networks the problem of vanishing or exploding gradients arise. This happens when the gradients during backpropagation become too little or too large, making it difficult to update the weights of earlier layers properly.
Kaiming He et al. in 2015 hypothesized that instead of learning the whole mapping it would be better optimizing the residual mapping than optimizing the initial, unreferenced mapping. Mathematically, a residual mapping(Figure 8.4) can be represented as F(x) = H(x) - x, where x is the input to a layer, H(x) is the desired mapping (ideal output), and F(x) represents the residual mapping that the layer aims to learn. By rewriting the equation to F(x) + x = H(x) the network learns to simulate the incremental adjustments required to refine the input and approach the intended output by adding the residual mapping to the input. During the training process, the network's parameters are optimized to minimize the difference between the actual and desired output. This is accomplished by modifying the settings to minimize the residual mapping F(x) value and bring it closer to zero. It is crucial to note, however, that achieving absolutely zero residual mapping is not necessarily the primary goal. The key goal is to adjust the network settings such that the layer can learn the residual mapping effectively and provide accurate predictions or representations. The shortcut connections introduced add neither extra parameters nor computation complexity.

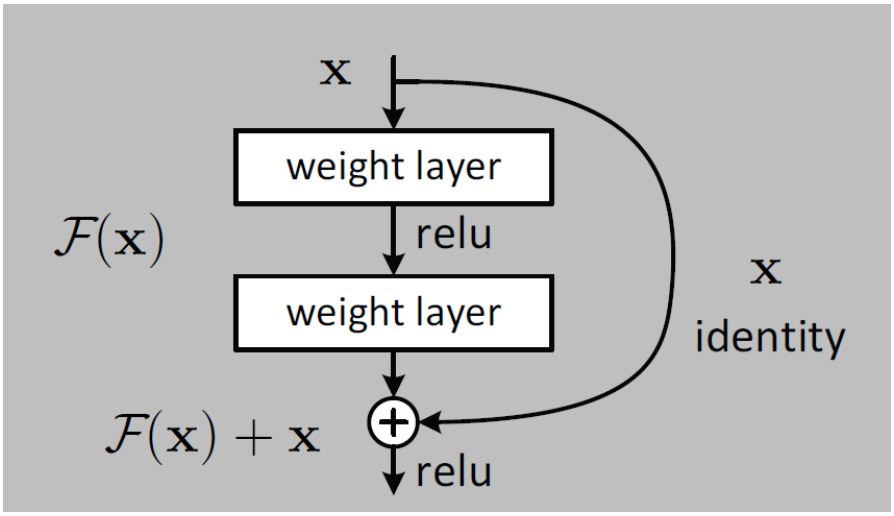

Figure 7.5: Residual learning

ResNet comes in different variations(Figure 7.6-7.7), such as ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152, which differ in terms of depth and number of parameters. We will be using the Resnet-18 and ResNet-50 as the backbone (encoder) of the UNet networks.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 7.6: Building blocks of ResNet variations

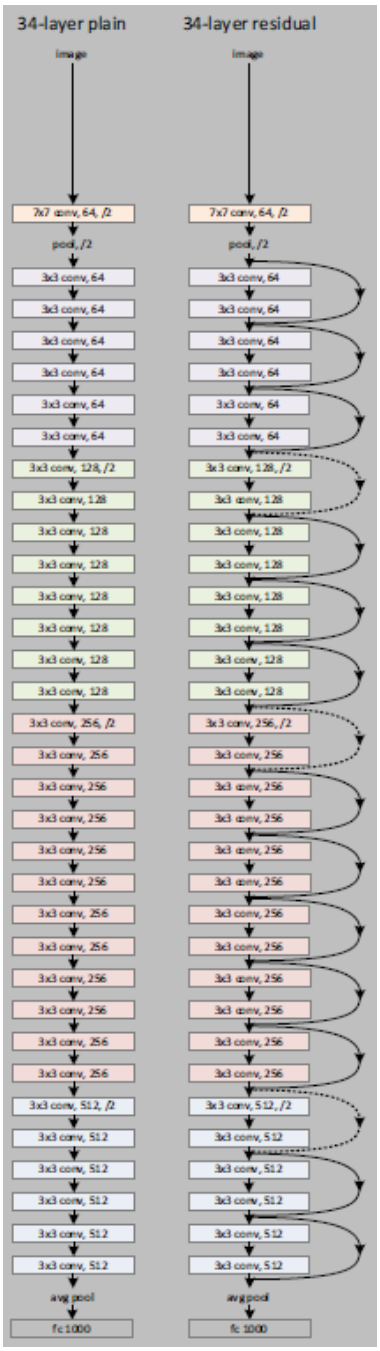

Figure 7.7: ResNet-34

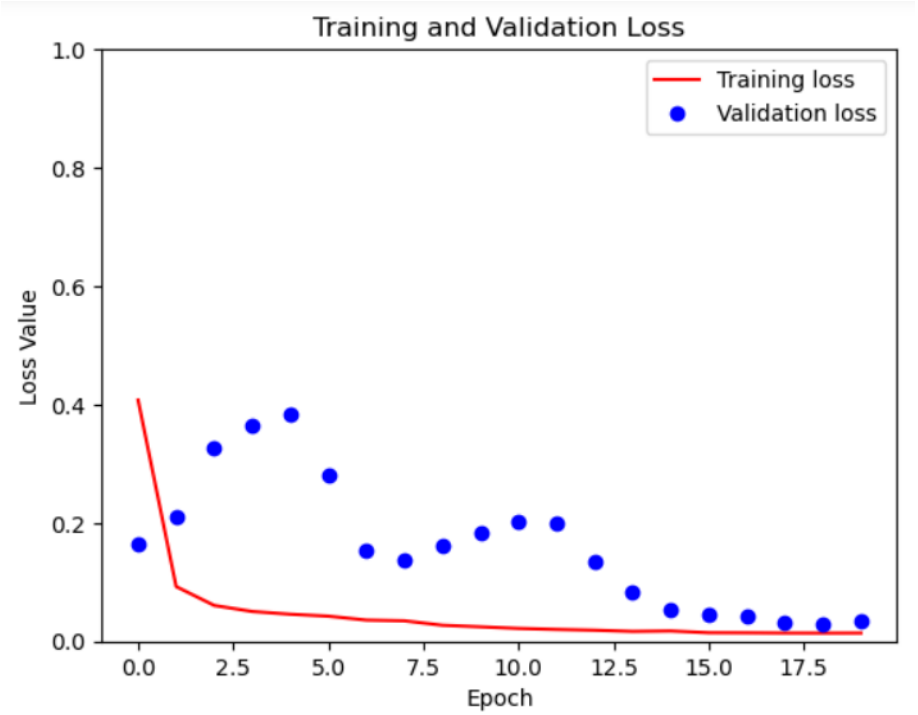

Figure 7.8: UNet-Resnet18 Model Loss

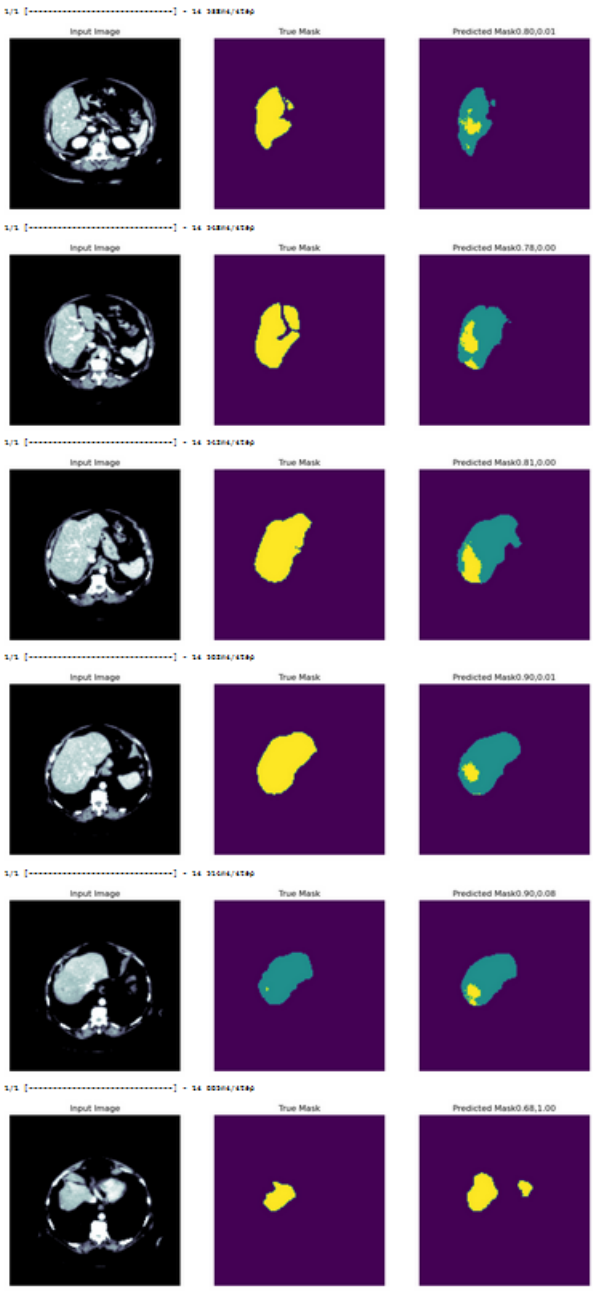

Figure 7.9: UNet-Resnet18 Model Predictions

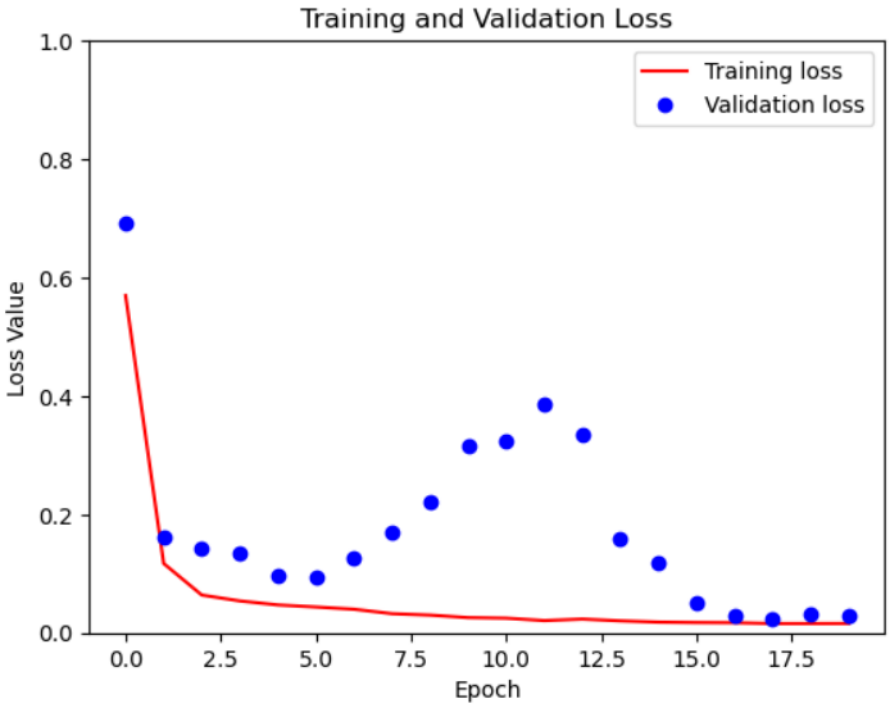

Figure 7.10: UNet-Resnet50 Model Loss

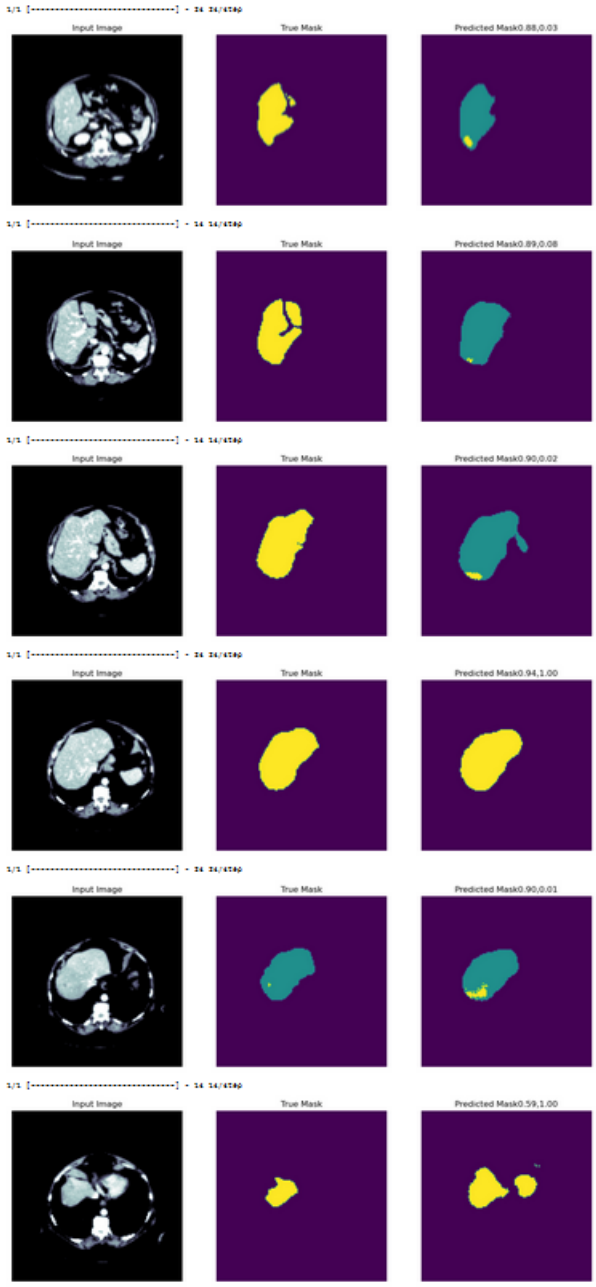

Figure 7.11: UNet-Resnet50 Model Predictions

7.4.3 VGG-16

Commonly known as VGGNet, one variation of it is referred to as VGG-16(Visual Geometry Group 16). It is a 16-layer convolution neural network (CNN) model as described by A. Zisserman and K. Simonyan(2015), and thus a relatively extensive network with a total of 138 million parameters.
The VGG16 architecture is well-known for its ease of use and extensive network depth. It has 16 weight layers, 13 convolutional layers, and 3 fully linked layers. To retain the spatial dimensions of the feature maps, the convolutional layers are built of modest 3x3 filters with a stride of 1 pixel and padding. The feature maps are downsampled using the max-pooling operation with a 2x2 filter and a stride of 2 pixels.
The consistent construction of VGG-16 is its most noticeable feature. The convolutional layers are piled on top of each other throughout the network, and the number of filters continuously grows while the spatial dimensions decrease
The VGG16 fully connected layers are followed by a softmax activation function, which allows the network to classify images into several classes. The architecture was first trained using the ImageNet dataset, which contains millions of annotated photos from 1000 different object categories.
The VGG architecture has also served as a basis for the development of deeper networks, such as VGG-19 and the ResNet family of networks. In the project we use the variation VGG-16D(Figure 7.12-7.13).

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 7.12: Different VGGNet models

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Figure 7.13: Number of Parameters in VGGNet models

7.4.4 Semantic Segmentation

SegNet is a deep fully convolutional neural network architecture for semantic pixel-wise segmentation(Figure 7.14). This core trainable segmentation engine is made up of an encoder network, a decoder network, and a pixel-wise classification layer. In was introduced(V. Badrinarayananet al., 2016) for scene un-

derstanding through object support relationships to autonomous driving. Due to its general success SegNet has been influential also in the field of image segmentation.

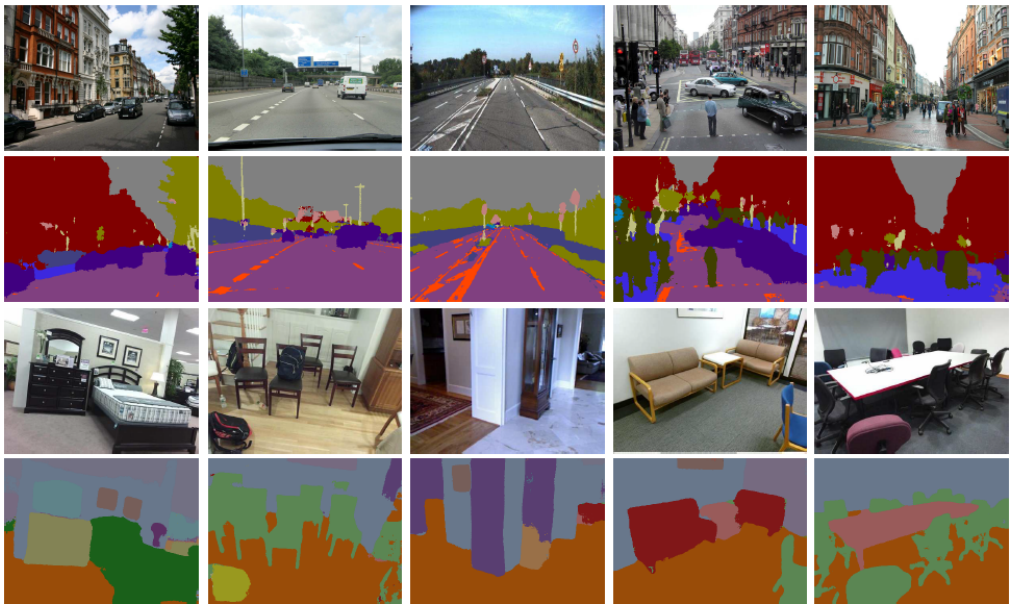

Figure 7.14: SegNet predictions on road scenes and indoor scenes

As previously mentioned the model architecture is based on encoder-decoder(Figure 7.15). More specifically we will be using the VGG-16 for the encoder part of the model architecture. We disregard the fully connected layers, reducing the parameters to 14.7 million, and transfer the softmax layer from the VGG-16 to the end of the decoder. The model also utilizes skip connections to propagate fine-grained information from the encoder to the decoder.
In contrast to SegNet, U-Net transmits the whole feature map (at the expense of extra memory) to the respective decoders and concatenates them to upsampled (through deconvolution) decoder feature maps.

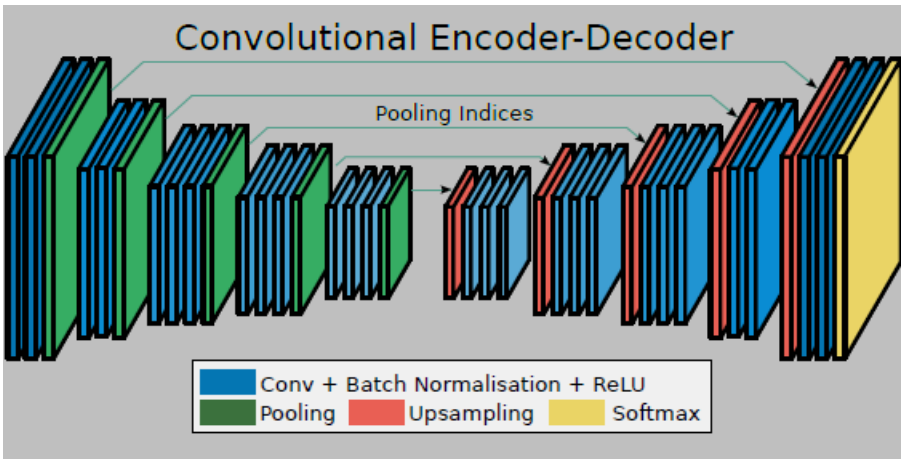

Figure 7.15: SegNet Architecture

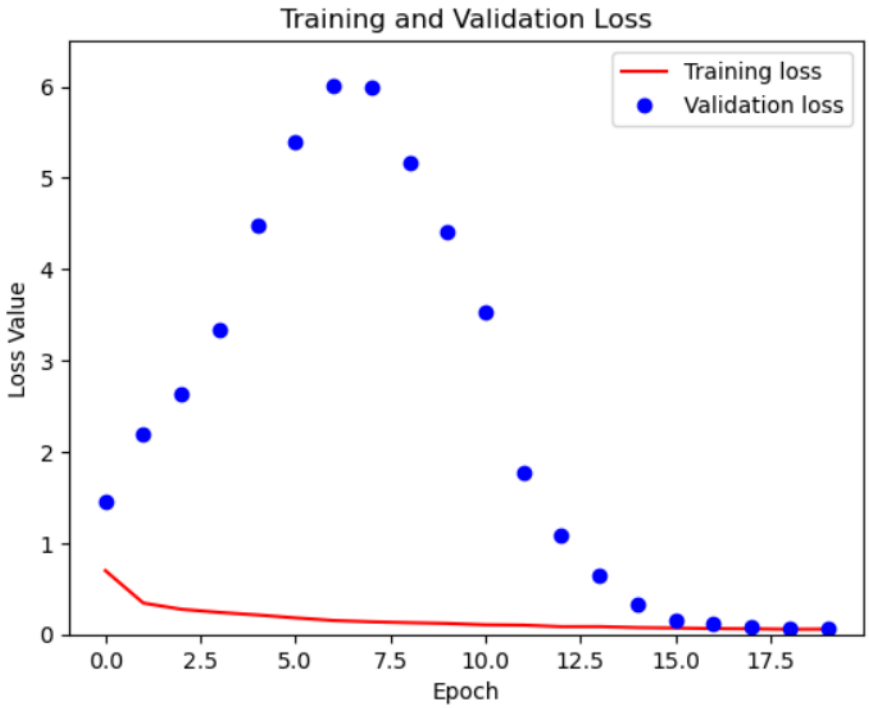

Figure 7.16: SegNet Model Loss

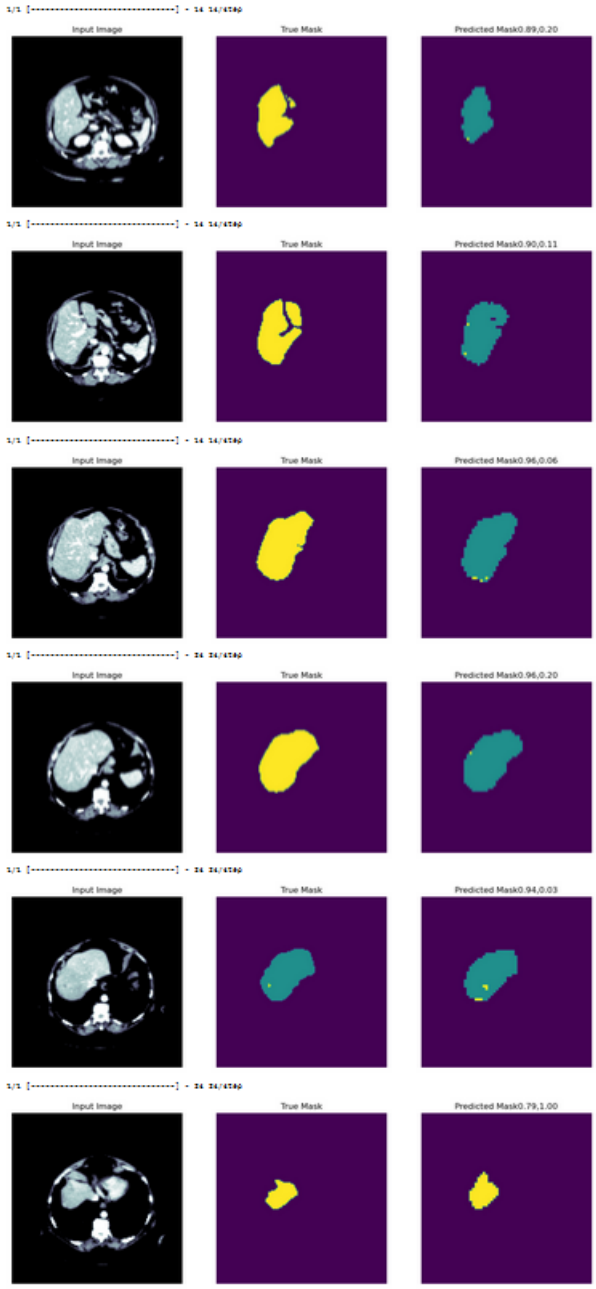

Figure 7.17: Predictions of SegNet Model

7.4.5 Custom Network

To conclude the model architectures we also added a custom network of our making. The network follows the UNet-like design, consisting of an encoder section that captures features from the input image and a decoder section that reconstructs the segmented output. The encoder section consists of three blocks of convolutional layers followed by batch normalization and max-pooling. Each block increases the number of feature channels while reducing the spatial dimensions through max-pooling.
The first block has two Conv2D layers with 32 filters each, and it is followed by max-pooling with a 2x2 kernel. The second block has two Conv2D layers with 64 filters each, followed by max-pooling with a 2x2 kernel. The third block has two Conv2D layers with 128 filters each, followed by max-pooling with a 2x2 kernel.
The decoder section consists of three blocks of convolutional layers followed by batch normalization and upsampling (UpSampling2D). The first block has two Conv2D layers with 256 filters each, followed by upsampling with a 2x2 kernel. The second block has two Conv2D layers with 128 filters each, followed by upsampling with a 2x2 kernel. The third block has two Conv2D layers with 64 filters each, followed by upsampling with a 2x2 kernel.
This inclusion to the number of systems was done only for comparison purposes to varifiy that a plain network without any special features like skip connections does poor job on image segmentation.

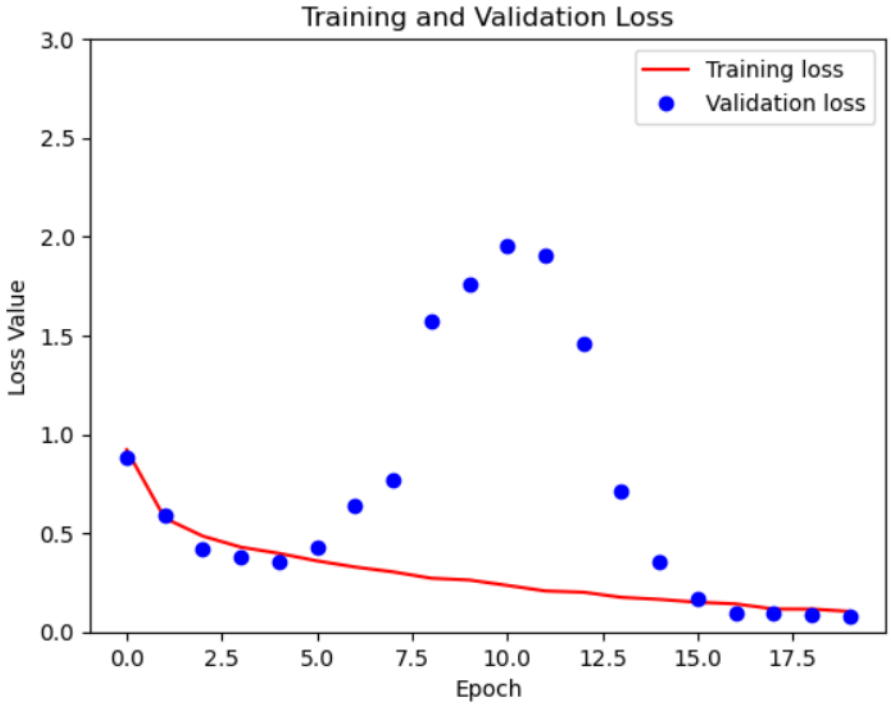

Figure 7.18: Loss of Custom Model

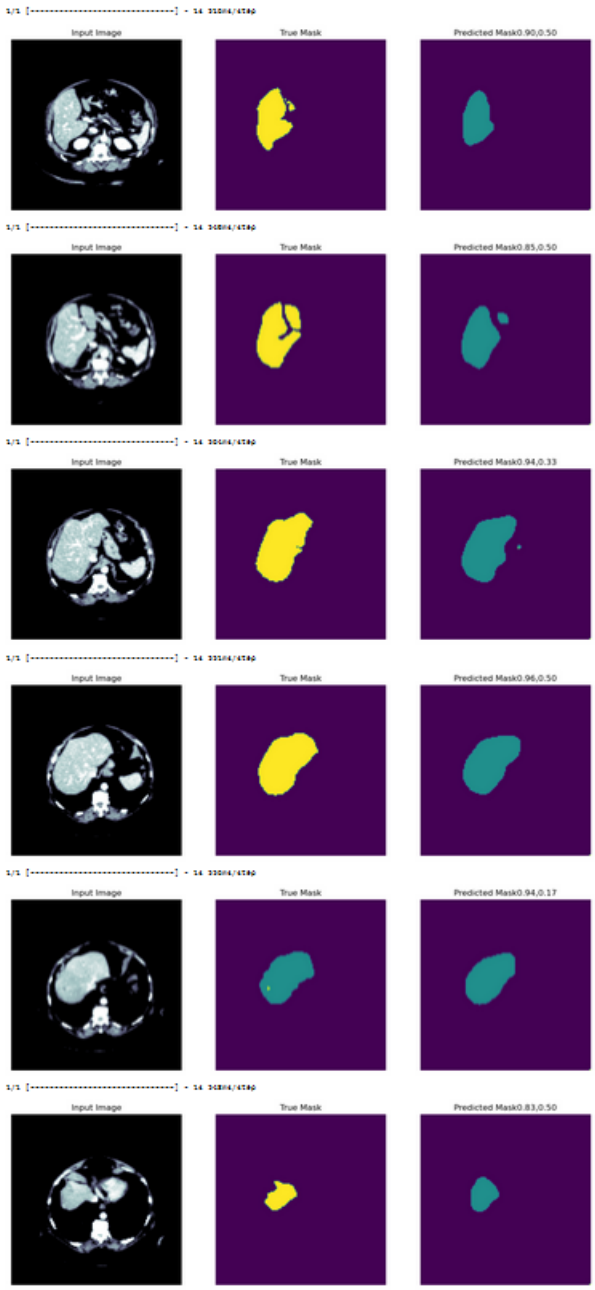

Figure 7.19: Predictions of Custom Model

7.5 Comparative analysis of Systems

7.5.1 Testing approach

We run each model for 20 epochs with BATCH_SIZE=16 and learning_rate=0.0001. After that we check the Dice score. The results were not always what we expected so we had many times to make changes. That meant changing the hyperparamaters. We establish training parameters such as learning rate, batch size, optimization method, and use approaches such as early halting and learning rate scheduling. The best results we got are in the following section(Table 7).

7.5.2 Comparison

	Complexity	Time training	Mean Dice Score(liver/tumor)	Standard Deviation(liver/tumor)
UNet	Very High	440sec/epoch	0.711/0.224	0.233/0.396
UNet-ResNet18	Low	130sec/epoch	0.737/0.273	0.234/0.429
UNet-ResNet50	High	209sec/epoch	0.794/0.509	0.230/0.453
SegNet	High	295sec/epoch	**0.806/0.554**	**0.249/0.454**
Custom	Low	140sec/epoch	0.754/0.482	0.294/0.128

Table 7: Comparison of different systems

We can see that the best model based on metrics is the SegNet one. The least good is the UNet which also has the highest complexity and even performs worse than the custom model. We can improve the performance of the standard UNet model by simply changing the encoder to capture better the features. Close second and counting also the standard deviation is the UNet-Resnet50.

8 Hyperparameter Tuning

BATCH_SIZE	Mean Dice Score(liver/tumor)	Standard Deviation(liver/tumor)
8	0.800/0.724	0.263/0.400
16	0.806/0.554	0.249/0.454
32	0.00074/0.00016	0.0011/0.00018

For BATCH_SIZE=32 we got no useful results back. The predictions were completely wrong. One solution would be to train the model for more epochs as we see a downward trend for the loss function.
For BATCH_SIZE=8 we get slightly worse results for the segmentation of the liver but better predictions for the lesions. Smaller batch sizes can aid model generalization by providing more frequent updates to the model's parameters,

which can lead to the model escaping local minima. We will continue the tuning with this value for the BATCH_SIZE.

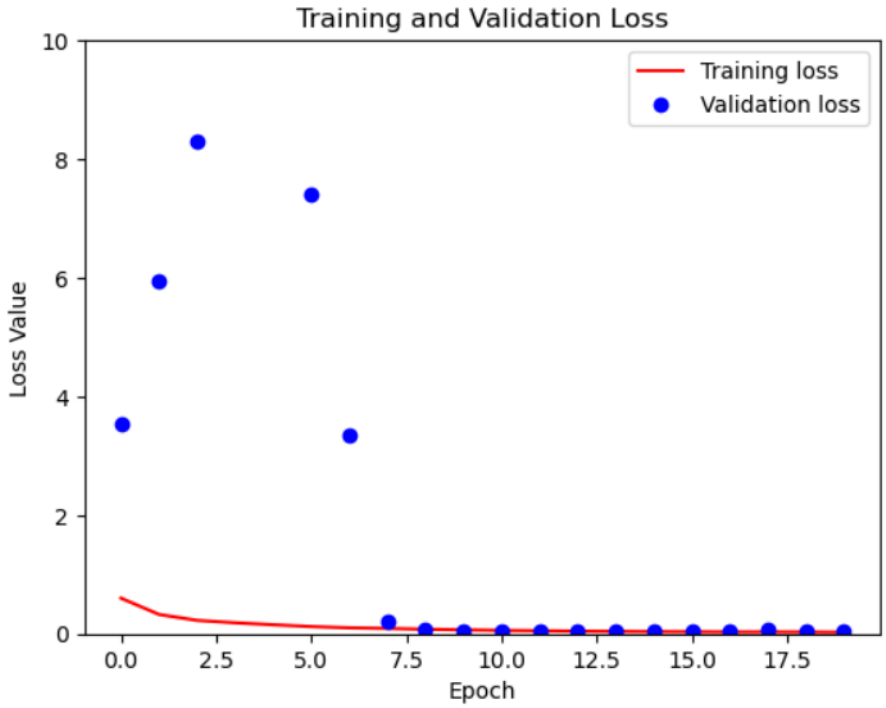

Figure 8.1: Loss of SegNet with BATCH_SIZE=8

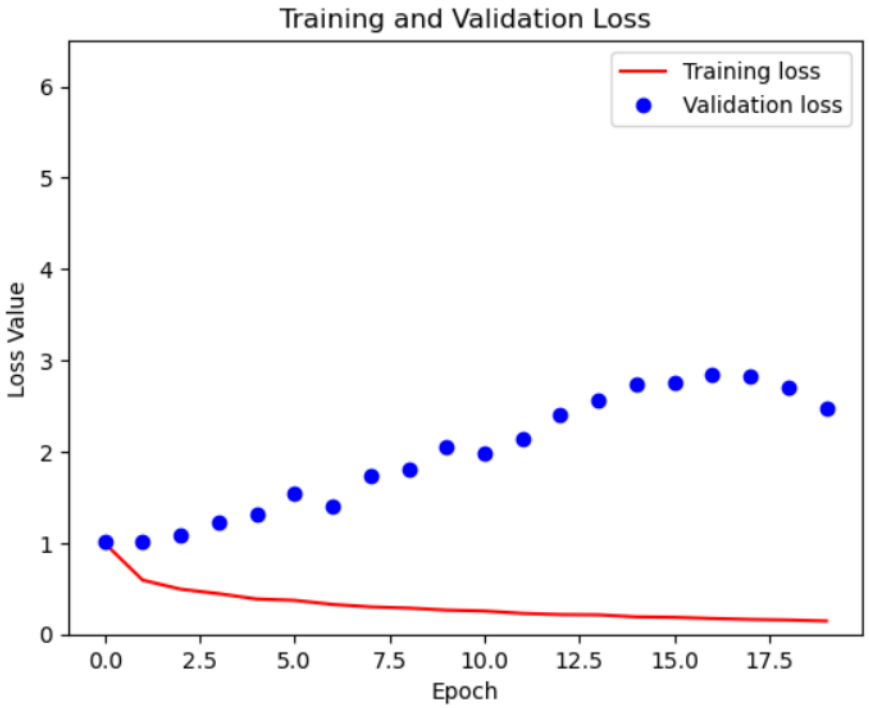

Figure 8.2: Loss of SegNet with BATCH_SIZE=32

Next thing we want to change is the learning rate. We will be testing a couple of values. One lower and one higher than the default of 0.0001.

Learning Rate	Mean Dice Score(liver/tumor)	Standard Deviation(liver/tumor)
0.01	0.8599/0.875	0.181/0.302
0.0001	0.800/0.724	0.263/0.400
0.00001	0.782/0.292	0.271/0.395

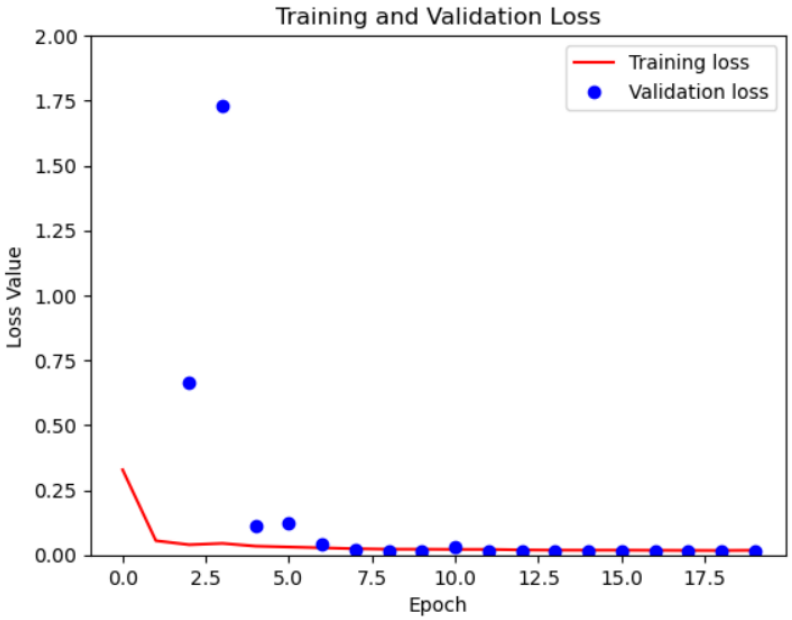

Figure 8.3: Loss of SegNet with Learning Rate=0.01

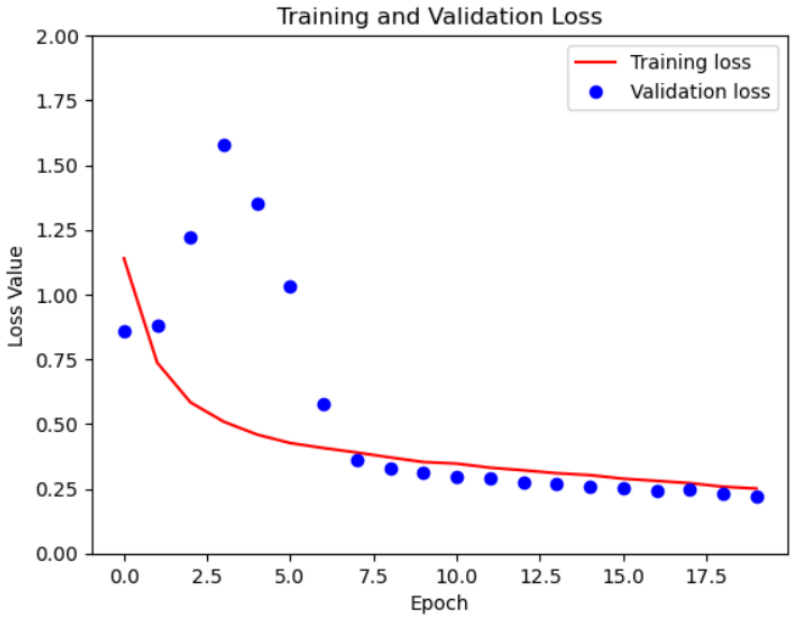

Figure 8.4: Loss of SegNet with Learning Rate=0.00001

From the resulting data we can see that the best results are when the learning rate=0.01. For that value, the validation loss starts at a high value that the diagram doesn't capture, but in the end it converges to a low acceptable value. This happens because of exploding gradients, where a high learning rate can lead to gradients that are too large at the start and the model's parameter updates are too large.
One last thing we want to check is related to the time the model trains for. Because the application has real word impact the time training is of crucial importance. One way to reduce the time is to reduce the resolution of the images.

Image Resolution	Mean Dice Score(liver/tumor)	Standard Deviation(liver/tumor)
64	0.777/0.862	0.272/0.308
128	0.860/0.875	0.181/0.302

The time training is reduced to half by reducing to half the resolution. We can see that this has the biggest impact on the liver segmentation. The higher the resolution the better the predictions we get. The results indicate the significance of striking a balance between accuracy and training time. If computing efficiency is a priority, reducing resolution may be a reasonable technique. Tested was also the resolution 256x256. However, we concluded that we need more computing power as the PC crashed consistently after the quadruple time we needed for training the model.
Reviewing the literature(Seo et al., 2020) we can see that our model is sufficiently good.

TABLE II
QUANTITATIVE SCORES OF THE LIVER-SEGMENTATION RESULTS (LITS DATASET). ALL METRIC IS DESCRIBED IN DETAIL IN [34]

	Liver				
	DSC (%)	VOE (%)	RVD (%)	ASSD (mm)	MSSD (mm)
Qin et al. [12]	97.18 ± 1.22	5.81 ± 2.48	0.91 ± 0.19	1.80 ± 0.55	12.48 ± 5.12
Han et al. [20]	97.36 ± 1.29	5.05 ± 2.29	0.72 ± 0.14	1.81 ± 0.56	13.75 ± 5.38
Men et al. [35]	97.50 ± 1.06	3.94 ± 2.17	0.45 ± 0.12	1.47 ± 0.38	10.35 ± 3.78
Li et al. [36]	98.05 ± **1.01**	3.31 ± **2.00**	0.32 ± **0.10**	1.16 ± **0.35**	9.17 ± **3.64**
mU-Net	**98.51** ± 1.02	**3.07** ± 2.01	**0.26** ± **0.10**	**0.92** ± 0.37	**8.52** ± 3.65

TABLE III
QUANTITATIVE SCORES OF THE LIVER-TUMOR-SEGMENTATION RESULTS LITS DATASET). ALL METRIC IS DESCRIBED IN DETAIL IN [34]

	Liver tumor				
	DSC (%)	VOE (%)	RVD (%)	ASSD (mm)	MSSD (mm)
Qin et al. [12]	39.82 ± 24.79	61.39 ± 29.04	-5.34 ± 0.87	4.70 ± 1.49	13.81 ± 4.72
Han et al. [20]	55.42 ± 26.37	50.73 ± 23.06	-0.82 ± 0.18	1.54 ± 0.43	5.99 ± 3.09
Men et al. [35]	83.14 ± 6.25	29.73 ± 16.31	-0.62 ± 0.14	0.96 ± 0.24	5.01 ± 1.98
Li et al. [36]	86.53 ± 5.32	24.46 ± 14.43	-0.53 ± **0.13**	0.83 ± 0.22	4.74 ± 1.97
mU-Net	**89.72** ± **5.07**	**21.93** ± **13.00**	**-0.49** ± 0.15	**0.78** ± **0.20**	**4.53** ± **1.95**

Figure 8.5: Result comparison against literature

9 Conclusions

9.1 Introduction and Summary

To summarize, combating liver cancer is one of the still standing problems in the medical sector. With the advances of technology, medicine can greatly benefit from ML algorithms. This dissertation examines the literature on different aspects of machine learning, from neurons, activation functions to up-sampling features. Through research we found the appropriate medical dataset about liver and liver tumor segmentation.
Moreover, we set the project aims, objectives and requirements to cement our understanding of needs of the project. The dissertation uses the programming language of Python and some of its ML packages, like Tensorflow. On the

dataset we performed preprocessing to change the resolution and enhance the CT slices. The data were separated to training and testing batches. It is also important to identify the users of the system -the admin and the doctor. Afterwards we identified use cases and their descriptions. In this dissertation we implemented four different ML architectures for image segmentation plus one custom model for comparison reasons. These are the UNet, the UNet with ResNet18/50 encoder and the SegNet with a VGG16 encoder. These were compared based on their Dice coefficient and the best one(SegNet) was selected for further tuning. We changed the batch size, the learning rate and the resolution. The best result were with a BATCH_SIZE of 8, a learning rate of 0.01 and a resolution of 128x128. The results were compared against the literature and have been found sufficient. The conclusion is that it is imperative that irregardless of with model we choose, image prepossessing, hyperparameter tuning are important to get good results which are on the same level as those of other researchers. Nevertheless, more research must be made to imporve the efficiency and accuracy of these models.

9.2 Review of Project Aims and Objectives

(PA1) Use machine learning models to detect cancer.

- PO1.1 *Research into the current practises for cancer detection with ML.* — We studied the literature about the modern techniques for image segmentation. Also to enhance our understanding we studied how to classify images, perform regression, prevent overfitting and underfitting, tune the hyperparameters.
- PO1.2 *Find the appropriate datasets.* — We started our journey with image classification with the dataset of Fashion MNIST and MedMNIST to understand how to perform ML predictions. Later we explored our reviewed dataset of LiTS.
- PO1.3 *Demonstrate an understanding of a significant programming language, such as Python.* — The language we used was Python and we come to understand its fundamental structure and the pythonic way of writing code.
- PO1.4 *Critical understanding of the issues and problems associated with the use of third-party packages and frameworks.* — Throughout the project we come across many different coding errors mainly involving the ML packages. We overcome those and we have our results in the report.
- PO1.5 *Implement data prepossessing techniques.* — We prerformed image resolution reduction keeping the aspect ration the same and also we enchanced the pictures by performing the windowing function to emphasize the areas where the liver and liver tumor were.
- PO1.6 *Develop machine learning models that can analyze the sets of medical data.* — We developed 5 ML models for this project analyzing their

performance on the dataset we have.

PO1.7 *Analyze the effectiveness of the produced models.* — By comparing the complexity, the time training and dice score we concluded to the best model that was the SegNet.

PO1.8 *Tune the best model for better performance.* — We performed a simple tuning by changing the hyperparameters of the batch size and learning rate. To find an appropriate balance between time training and performance we reduced the resolution of the CT slices.

9.3 Review of Requirements

9.3.1 Functional Requirements

Essential Requirements

FR1 *Develop and implement machine learning models that can analyze sets of medical data.* — Fully met by developing different model architectures.

FR2 *Segment successfully the CT images from the dataset.* — Fully met based on the results we present in this report.

FR3 *Evaluate the effectiveness of the developed machine learning models* — Fully met by mainly evaluating the Dice score.

Desirable Functional Requirements

FR4 *Analyze the effectiveness of the produced models and tune them for better performance.* — Fully met as we tuned for better performance the best model we produced.

Luxury Functional Requirements

FR5 *Create synthetic or fake data to augment the available datasets and reduce the need for large amounts of real-world data.* — Based on the project progress review we changed course and removed this part of the project. This has a luxury requirement as the time has pressing and we had more data available than needed.

FR6 *Examining if the synthetic data generation techniques impact the accuracy and generalization capabilities of the models.* — For the same reason as above this requirement was not met.

9.3.2 Non-Functional Requirements

NFR1 *A suitable development environment to work on the code.* — Python is our choice language for machine learning, and we chose to work with an IDE of Jupyter Notebook.

NFR2 *Critical understanding of the issues and problems associated with the use of third-party packages and frameworks.* — We understood and overcome the issues we faced.

NFR3 *A solid understanding of machine learning algorithms and techniques* — Supervised and unsupervised learning, regression, classification, and clustering were studied.

NFR4 *Familiarity with relevant machine learning libraries and frameworks* — We used TensorFlow, Keras, NumPy for developing the code.

NFR5 *Access to cancer datasets, from publicly available sources. We will need to ensure that the data is accurate and reliable, and that we have the necessary permissions to use it.* — We found the LiTS dataset which was online available. The data contained were reliable as per results.

NFR6 *Good understanding of ethical and legal considerations related to working with medical data and artificial intelligence, including privacy and security regulations, and avoiding harm and bias.* — We didn't deal with ethical and legal subjects as the dataset is anonymous and publicly available for everyone to use.

NFR7 *Familiarity with data analysis and data preprocessing techniques, such as data cleaning, feature extraction, and feature scaling.* — Fully met with implementing image resizing and enhancing.

Luxury Non - Functional Requirements

NFR8 *Knowledge of synthetic data generation techniques, such as generative adversarial networks (GANs), autoencoders, or other methods to create synthetic data.* — Not met as we had more data available than needed.

NFR9 *Patience, perseverance, and attention to detail, as machine learning projects can be complex and challenging and may require multiple iterations and adjustments to achieve good results.* — Fully met despite some periods where we had to produce some results and needed help from the supervisor. Also, we performed many iterations and despite the mental fatigue we produced some results.

9.4 Further Work and Improvements

The work contacted in the dissertation was a few months research. The results provide a direction and a base for further research.
First of all we can perform some more image preprocessing as this has a major impact on the results. Because the main issue with segmenting the liver and its lesions is the vaguely borders, shape and colors of them a good image enhancement must be made.
More data can be used to increase the reliability and generalization of the ML model. In this project we only used a small fraction of the data available as the

computational capabilities and time we had in our disposal was limited.
More model architectures exist and they are not limited to the one we used. Variations of them should be evaluated.
One of the main issues for liver segmentation is the availability of data. To address this problem we can use techniques to improve the generalization capabilities of the models, such as synthetic image generation as it has been suggested.

9.5 Conclusions

Based on the above the project concludes that for image processing of the liver and liver tumor segmentation the best architecture is that of the SegNet with a backbone of VGG16. It produces a Dice score of 0.8599/0.875. This is optimized with training on specific hyperparameters(batch size=8, learning rate=0.01, resolution= 128, epochs=20). Although most of the tasks outlined on the research criteria were completed, much more time is required to complete all of the aims and objectives.
The most valuable learning outcomes were that Python and its libraries are appropriate for ML image processing and that testing different architectures will produce varied results - some more accurate than others.
There is more to be explored on the topic by testing different architectures, using a wider range of datasets and more fine tuning.

10 References

WHO, (2022). Press Release No. 320: Number of new cases and deaths from liver cancer predicted to rise by more than 55% by 2040. IARC.

Bilic, P., Christ, P., Bran Li, H., Vorontsov, E., Ben-Cohen, A., Kaissis, G. et al., (2023). The Liver Tumor Segmentation Benchmark (LiTS). Medical Image Analysis 84.

Chlebus, G., Schenk, A., Moltz, J.H. van Ginneken ,B., Hahn, H.K. & Meine , H., (2018). Automatic liver tumor segmentation in CT with fully convolutional neural networks and object-based postprocessing. Sci Rep 8 (15497).

Kaggle, (2023). Liver Tumor Segmentation. Available from: https://www.kaggle.com/datasets/andrewmvd/liver-tumor-segmentation?resource=download-directory(Accessed: 10 June 2023).

Ramadhan, L., (2021). Neural Network: The Dead Neuron. Towards Data Science. Available from: https://towardsdatascience.com/neural-network-the-dead-neuron-eaa92e575748(Accesed: 10 June 2023).

Krishnamurthy, B. (2022). An Introduction to the ReLU Activation Function. Available from https://builtin.com/machine-learning/relu-activation-function(Accessed: 10 June 2023).

Babs T., (2018). The Mathematics of Neural Networks. Coinmonks. Available from https://medium.com/coinmonks/the-mathematics-of-neural-network-60a112dd3e05(Accessed: 10 June 2023).

Wikipedia, (2023). Gradient descent. Available from https://en.wikipedia.org/wiki/Gradient_descent(Accessed: 10 June 2023).

Thevenot A., (2020). Conv2d: Finally Understand What Happens in the Forward Pass. Available from https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148(Accessed 10 June 2023)

Ronneberger, O., Fischer, P., Brox, T., (2015). U-Net: Convolutional Networks for Biomedical. Medical Image Computing and Computer-Assisted Intervention – MICCAI.

He, K., Zhang, X., Ren, S., & Sun, J., (2015). Deep Residual Learning for Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 770-778).

Zisserman, A., and Simonyan, K., (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. International Conference on Learning Representations (ICLR

Badrinarayanan, V., Kendall, A., & Cipolla, R., (2015). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 39(12), 2481-2495.

Zhao, H., Shi, J., Qi, X., Wang, X., & Jia, J., (2017). Pyramid Scene Parsing Network. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Seo, H., Huang C., Bassenne, M., Xiao, R., and Xing, L., (2020). Modified U-Net (mU-Net) With Incorporation of Object-Dependent High Level Features for Improved Liver and Liver-Tumor Segmentation in CT Images. IEEE TRANSACTIONS ON MEDICAL IMAGING, Vol. 39(5)

11 Appendix A: Image preprocessing

Here follows the data preprocessing code snippets crucial for the machine learning algorithm to work more efficiently.

```
def read_nii(filepath):
    ct_scan = nib.load(filepath)
    array = ct_scan.get_fdata()
    array = np.rot90(np.array(array))
    return(array)
```

```
def windowed(w, l, tensor):
    px_min = l - w // 2
    px_max = l + w // 2
    px = tf.clip_by_value(tensor, px_min, px_max)
    return (px - px_min) / (px_max - px_min)

cwd = os.getcwd()
X_train = []
y_train = []
for i in range(0, 5):
    curr_ct = read_nii(os.path.join(cwd, "Desktop/input/liver-tumor-segmentation/volume_pt1/",f"volume-
{i}.nii"))
    curr_mask = read_nii(os.path.join(cwd, "Desktop/input/liver-tumor-segmentation/segmentations
",f"segmentation-{i}.nii"))
    curr_dim = curr_ct.shape[2]
    for curr_slice in range(0, curr_dim):
        image_ct = Image.fromarray(curr_ct[...,curr_slice])
        image_mask = Image.fromarray(curr_mask[...,curr_slice])
        # Calculate the scaling factor for resizing
        width, height = image_ct.size
        aspect_ratio = width / height
        new_width = int(IMAGE_SIZE * aspect_ratio)
        # Resize the image while maintaining the aspect ratio
        resized_image_ct = image_ct.resize((new_width, IMAGE_SIZE), Image.BILINEAR)
        resized_image_mask = image_mask.resize((new_width, IMAGE_SIZE),
Image.NEAREST)
        # Crop the resized image to (256, 256) by centering it
        left = (new_width - IMAGE_SIZE) // 2
        right = left + IMAGE_SIZE
        cropped_image_ct = resized_image_ct.crop((left, 0, right, IMAGE_SIZE))
        cropped_image_mask = resized_image_mask.crop((left, 0, right, IMAGE_SIZE))
        # Convert the cropped image back to a tensor
        resized_tensor_ct = np.array(cropped_image_ct)
        resized_tensor_mask = np.array(cropped_image_mask)
        X_train.append(windowed(150, 30, tf.convert_to_tensor(resized_tensor_ct)))
        y_train.append(tf.convert_to_tensor(resized_tensor_mask, dtype="uint8"))

X_train = tf.expand_dims(X_train, axis=-1)
y_train = tf.expand_dims(y_train, axis=-1)
```

12 Appendix B: U-Net

```
def build_model(input_shape, num_classes):
    input_layer = Input(input_shape)
```

```
    conv1 = Conv2D(64, (3, 3), activation="relu", padding="same")(input_layer)
    conv1 = Conv2D(64, (3, 3), activation="relu", padding="same")(conv1)
    pool1 = MaxPooling2D((2, 2))(conv1)
    conv2 = Conv2D(128, (3, 3), activation="relu", padding="same")(pool1)
    conv2 = Conv2D(128, (3, 3), activation="relu", padding="same")(conv2)
    pool2 = MaxPooling2D((2, 2))(conv2)
    conv3 = Conv2D(256, (3, 3), activation="relu", padding="same")(pool2)
    conv3 = Conv2D(256, (3, 3), activation="relu", padding="same")(conv3)
    pool3 = MaxPooling2D((2, 2))(conv3)
    conv4 = Conv2D(512, (3, 3), activation="relu", padding="same")(pool3)
    conv4 = Conv2D(512, (3, 3), activation="relu", padding="same")(conv4)
    pool4 = MaxPooling2D((2, 2))(conv4)
    convm = Conv2D(1024, (3, 3), activation="relu", padding="same")(pool4)
    convm = Conv2D(1024, (3, 3), activation="relu", padding="same")(convm)
    deconv4 = Conv2DTranspose(512, (3, 3), strides=(2, 2), padding="same")(convm)
    uconv4 = concatenate([deconv4, conv4])
    uconv4 = Conv2D(512, (3, 3), activation="relu", padding="same")(uconv4)
    uconv4 = Conv2D(512, (3, 3), activation="relu", padding="same")(uconv4)
    deconv3 = Conv2DTranspose(256, (3, 3), strides=(2, 2), padding="same")(uconv4)
    uconv3 = concatenate([deconv3, conv3])
    uconv3 = Conv2D(256, (3, 3), activation="relu", padding="same")(uconv3)
    uconv3 = Conv2D(256, (3, 3), activation="relu", padding="same")(uconv3)
    deconv2 = Conv2DTranspose(128, (3, 3), strides=(2, 2), padding="same")(uconv3)
    uconv2 = concatenate([deconv2, conv2])
    uconv2 = Conv2D(128, (3, 3), activation="relu", padding="same")(uconv2)
    uconv2 = Conv2D(128, (3, 3), activation="relu", padding="same")(uconv2)
    deconv1 = Conv2DTranspose(64, (3, 3), strides=(2, 2), padding="same")(uconv2)
    uconv1 = concatenate([deconv1, conv1])
    uconv1 = Conv2D(64, (3, 3), activation="relu", padding="same")(uconv1)
    uconv1 = Conv2D(64, (3, 3), activation="relu", padding="same")(uconv1)
    output_layer = Conv2D(num_classes, 1, padding="same")(uconv1)
    model = Model(inputs=input_layer, outputs=output_layer)
    return model
```

13 Appendix C: Helper functions for Resnet

```
# Convolutional Block
def conv_block(inputs, filters, kernel_size, strides):
    x = Conv2D(filters=filters, kernel_size=kernel_size, strides=strides, padding='same')(inputs)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    return x

# Identity Block
def identity_block(inputs, filters):
```

```
    x = conv_block(inputs, filters=filters, kernel_size=3, strides=1)
    x = conv_block(x, filters=filters, kernel_size=3, strides=1)
    x = Add()([x, inputs])
    x = Activation('relu')(x)
    return x

# Projection Block
def projection_block(inputs, filters, strides):
    shortcut = Conv2D(filters=filters, kernel_size=1, strides=strides, padding='same')(inputs)
    shortcut = BatchNormalization()(shortcut)
    x = conv_block(inputs, filters=filters, kernel_size=3, strides=strides)
    x = conv_block(x, filters=filters, kernel_size=3, strides=1)
    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x
```

14 Appendix D: UNet - ResNet18

```
def unet_resnet18_model(input_shape, num_classes):      inputs = tf.keras.Input(shape=input_shape)
    # Stage 1
    x = conv_block(inputs, filters=64, kernel_size=7, strides=2)
    encoder1 = x
    x = MaxPooling2D(pool_size=3, strides=2, padding='same')(x)
    # Stage 2
    x = projection_block(x, filters=64, strides=1)
    x = identity_block(x, filters=64)
    encoder2 = x
    # Stage 3
    x = projection_block(x, filters=128, strides=2)
    x = identity_block(x, filters=128)
    encoder3 = x
    # Stage 4
    x = projection_block(x, filters=256, strides=2)
    x = identity_block(x, filters=256)
    encoder4 = x
    # Stage 5
    x = projection_block(x, filters=512, strides=2)
    x = identity_block(x, filters=512)
    encoder5 = x
    # Upsampling path
    up1 = Conv2DTranspose(512, 2, strides=(2, 2), activation='relu', padding='same')(encoder5)
    merge1 = Concatenate()([encoder4, up1])
    conv1 = Conv2D(512, 3, activation='relu', padding='same')(merge1)
    conv1 = Conv2D(512, 3, activation='relu', padding='same')(conv1)
    up2 = Conv2DTranspose(256, 2, strides=(2, 2), activation='relu', padding='same')(conv1)
```

```
    merge2 = Concatenate()([encoder3, up2])
    conv2 = Conv2D(256, 3, activation='relu', padding='same')(merge2)
    conv2 = Conv2D(256, 3, activation='relu', padding='same')(conv2)
    up3 = Conv2DTranspose(128, 2, strides=(2, 2), activation='relu', padding='same')(conv2)
    merge3 = Concatenate()([encoder2, up3])
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(merge3)
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(conv3)
    up4 = Conv2DTranspose(64, 2, strides=(2, 2), activation='relu', padding='same')(conv3)
    merge4 = Concatenate()([encoder1, up4])
    conv4 = Conv2D(64, 3, activation='relu', padding='same')(merge4)
    conv4 = Conv2D(64, 3, activation='relu', padding='same')(conv4)
    up5 = Conv2DTranspose(32, 2, strides=(2, 2), activation='relu', padding='same')(conv4)
    conv5 = Conv2D(num_classes, 1, padding='same')(up5)
    model = Model(inputs=inputs, outputs=conv5)
    return model
```

15 Appendix E: UNet - Resnet50

```
def unet_resnet50_model(input_shape, num_classes):
    # ResNet-50 backbone
    inputs = tf.keras.Input(shape=input_shape)
    # Stage 1
    x = conv_block(inputs, filters=64, kernel_size=7, strides=2)
    encoder1 = x
    x = MaxPooling2D(pool_size=3, strides=2, padding='same')(x)
    # Stage 2
    x = projection_block(x, filters=64, strides=1)
    x = identity_block(x, filters=64)
    x = identity_block(x, filters=64)
    encoder2 = x
    # Stage 3
    x = projection_block(x, filters=128, strides=2)
    x = identity_block(x, filters=128)
    x = identity_block(x, filters=128)
    x = identity_block(x, filters=128)
    encoder3 = x
    # Stage 4
    x = projection_block(x, filters=256, strides=2)
    x = identity_block(x, filters=256)
    x = identity_block(x, filters=256)
    x = identity_block(x, filters=256)
    x = identity_block(x, filters=256)
    x = identity_block(x, filters=256)
    encoder4 = x
    # Stage 5
```

```
    x = projection_block(x, filters=512, strides=2)
    x = identity_block(x, filters=512)
    x = identity_block(x, filters=512)
    encoder5 = x
    # Upsampling path
    up1 = Conv2DTranspose(512, 2, strides=(2, 2), activation='relu', padding='same')(encoder5)
    merge1 = Concatenate()([encoder4, up1])
    conv1 = Conv2D(512, 3, activation='relu', padding='same')(merge1)
    conv1 = Conv2D(512, 3, activation='relu', padding='same')(conv1)
    up2 = Conv2DTranspose(256, 2, strides=(2, 2), activation='relu', padding='same')(conv1)
    merge2 = Concatenate()([encoder3, up2])
    conv2 = Conv2D(256, 3, activation='relu', padding='same')(merge2)
    conv2 = Conv2D(256, 3, activation='relu', padding='same')(conv2)
    up3 = Conv2DTranspose(128, 2, strides=(2, 2), activation='relu', padding='same')(conv2)
    merge3 = Concatenate()([encoder2, up3])
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(merge3)
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(conv3)
    up4 = Conv2DTranspose(64, 2, strides=(2, 2), activation='relu', padding='same')(conv3)
    cropped_encoder1 = Cropping2D(cropping=((1, 0), (1, 0)))(encoder1)
    cropped_encoder1 = tf.image.resize(cropped_encoder1, tf.shape(up4)[1:3])
    merge4 = Concatenate()([cropped_encoder1, up4])
    conv4 = Conv2D(64, 3, activation='relu', padding='same')(merge4)
    conv4 = Conv2D(64, 3, activation='relu', padding='same')(conv4)
    up5 = Conv2DTranspose(32, 2, strides=(2, 2), activation='relu', padding='same')(conv4)
    conv5 = Conv2D(num_classes, 1, padding='same')(up5)
    model = Model(inputs=inputs, outputs=conv5)
    return model
```

16 Appendix F: Segnet

```
def segnet_model(input_shape, num_classes):
    inputs = Input(shape=input_shape)
    # Encoder
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(inputs)
    conv1 = BatchNormalization()(conv1)
    conv2 = Conv2D(64, 3, activation='relu', padding='same')(conv1)
    conv2 = BatchNormalization()(conv2)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(pool1)
    conv3 = BatchNormalization()(conv3)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(conv3)
    conv4 = BatchNormalization()(conv4)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv4)
    conv5 = Conv2D(256, 3, activation='relu', padding='same')(pool2)
    conv5 = BatchNormalization()(conv5)
```

```
conv6 = Conv2D(256, 3, activation='relu', padding='same')(conv5)
conv6 = BatchNormalization()(conv6)
conv7 = Conv2D(256, 3, activation='relu', padding='same')(conv6)
conv7 = BatchNormalization()(conv7)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv7)
conv8 = Conv2D(512, 3, activation='relu', padding='same')(pool3)
conv8 = BatchNormalization()(conv8)
conv9 = Conv2D(512, 3, activation='relu', padding='same')(conv8)
conv9 = BatchNormalization()(conv9)
conv10 = Conv2D(512, 3, activation='relu', padding='same')(conv9)
conv10 = BatchNormalization()(conv10)
pool4 = MaxPooling2D(pool_size=(2, 2))(conv10)
conv11 = Conv2D(512, 3, activation='relu', padding='same')(pool4)
conv11 = BatchNormalization()(conv11)
conv12 = Conv2D(512, 3, activation='relu', padding='same')(conv11)
conv12 = BatchNormalization()(conv12)
conv13 = Conv2D(512, 3, activation='relu', padding='same')(conv12)
conv13 = BatchNormalization()(conv13)
pool5 = MaxPooling2D(pool_size=(2, 2))(conv13)
# Decoder
upsample1 = UpSampling2D(size=(1, 1))(pool5)
conv14 = Conv2D(512, 3, activation='relu', padding='same')(upsample1)
conv14 = BatchNormalization()(conv14)
conv15 = Conv2D(512, 3, activation='relu', padding='same')(conv14)
conv15 = BatchNormalization()(conv15)
conv16 = Conv2D(512, 3, activation='relu', padding='same')(conv15)
conv16 = BatchNormalization()(conv16)
upsample2 = UpSampling2D(size=(2, 2))(conv16)
x1 = concatenate([pool4, upsample2])
conv17 = Conv2D(512, 3, activation='relu', padding='same')(x1)
conv17 = BatchNormalization()(conv17)
conv18 = Conv2D(512, 3, activation='relu', padding='same')(conv17)
conv18 = BatchNormalization()(conv18)
conv19 = Conv2D(256, 3, activation='relu', padding='same')(conv18)
conv19 = BatchNormalization()(conv19)
upsample3 = UpSampling2D(size=(2, 2))(conv19)
x2 = concatenate([pool3, upsample3])
conv20 = Conv2D(256, 3, activation='relu', padding='same')(x2)
conv20 = BatchNormalization()(conv20)
conv21 = Conv2D(256, 3, activation='relu', padding='same')(conv20)
conv21 = BatchNormalization()(conv21)
conv22 = Conv2D(128, 3, activation='relu', padding='same')(conv21)
conv22 = BatchNormalization()(conv22)
upsample4 = UpSampling2D(size=(2, 2))(conv22)
x3 = concatenate([pool2, upsample4])
conv23 = Conv2D(128, 3, activation='relu', padding='same')(x3)
```

```
    conv23 = BatchNormalization()(conv23)
    conv24 = Conv2D(64, 3, activation='relu', padding='same')(conv23)
    conv24 = BatchNormalization()(conv24)
    upsample5 = UpSampling2D(size=(2, 2))(conv24)
    x4 = concatenate([pool1, upsample5])
    conv25 = Conv2D(64, 3, activation='relu', padding='same')(x4)
    conv25 = BatchNormalization()(conv25)
    conv26 = Conv2D(64, 3, activation='relu', padding='same')(conv25)
    conv26 = BatchNormalization()(conv26)
    upsample6 = UpSampling2D(size=(2, 2))(conv26)
    outputs = Conv2D(num_classes, 1, padding='same')(upsample6)
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

17 Appendix G: Custom Network

```
def encoder_decoder(input_shape, num_classes):
    # Input tensor
    inputs = tf.keras.Input(shape=input_shape)
    # Encoder      conv1 = Conv2D(32, 3, activation='relu', padding='same')(inputs)
    conv1 = BatchNormalization()(conv1)
    conv1 = Conv2D(32, 3, activation='relu', padding='same')(conv1)
    conv1 = BatchNormalization()(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(64, 3, activation='relu', padding='same')(pool1)
    conv2 = BatchNormalization()(conv2)
    conv2 = Conv2D(64, 3, activation='relu', padding='same')(conv2)
    conv2 = BatchNormalization()(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(pool2)
    conv3 = BatchNormalization()(conv3)
    conv3 = Conv2D(128, 3, activation='relu', padding='same')(conv3)
    conv3 = BatchNormalization()(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
    # Decoder
    conv4 = Conv2D(256, 3, activation='relu', padding='same')(pool3)
    conv4 = BatchNormalization()(conv4)
    conv4 = Conv2D(256, 3, activation='relu', padding='same')(conv4)
    conv4 = BatchNormalization()(conv4)
    up1 = UpSampling2D(size=(2, 2))(conv4)
    conv5 = Conv2D(128, 3, activation='relu', padding='same')(up1)
    conv5 = BatchNormalization()(conv5)
    conv5 = Conv2D(128, 3, activation='relu', padding='same')(conv5)
    conv5 = BatchNormalization()(conv5)
    up2 = UpSampling2D(size=(2, 2))(conv5)
```

```
    conv6 = Conv2D(64, 3, activation='relu', padding='same')(up2)
    conv6 = BatchNormalization()(conv6)
    conv6 = Conv2D(64, 3, activation='relu', padding='same')(conv6)
    conv6 = BatchNormalization()(conv6)
    up3 = UpSampling2D(size=(2, 2))(conv6)
    conv7 = Conv2D(32, 3, activation='relu', padding='same')(up3)
    conv7 = BatchNormalization()(conv7)
    conv7 = Conv2D(32, 3, activation='relu', padding='same')(conv7)
    conv7 = BatchNormalization()(conv7)
    # Output      output = Conv2D(num_classes, 1, padding='same')(conv7)
    model = tf.keras.Model(inputs=inputs, outputs=output)
    return model
```

18 Appendix H: Model Compilation

```
X_train_dataset = tf.data.Dataset.from_tensor_slices(X_train)
y_train_dataset = tf.data.Dataset.from_tensor_slices(y_train)
train_dataset = tf.data.Dataset.zip((X_train_dataset, y_train_dataset))
X_test_dataset = tf.data.Dataset.from_tensor_slices(X_test)
y_test_dataset = tf.data.Dataset.from_tensor_slices(y_test)
test_dataset = tf.data.Dataset.zip((X_test_dataset, y_test_dataset))
BUFFER_SIZE = 1000
BATCH_SIZE = 16
train_batches = (
    train_dataset
    .cache()
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE)
    .repeat()
    .prefetch(buffer_size=tf.data.AUTOTUNE)
)
test_batches = test_dataset.batch(BATCH_SIZE)

# Compile the model
random = encoder_decoder(input_shape=(128, 128, 1), num_classes=3)
tf.random.set_seed(42)
random.compile(tf.keras.optimizers.Adam(0.0001),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'])

# Train the model
EPOCHS = 20
VAL_SUBSPLITS = 5
TRAIN_LENGTH = len(X_train)
VALIDATION_STEPS = TRAIN_LENGTH//BATCH_SIZE//VAL_SUBSPLITS
```

```
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE
model_history = random.fit(train_batches, epochs=EPOCHS,
                  steps_per_epoch=STEPS_PER_EPOCH,
                  validation_steps=VALIDATION_STEPS,
                  validation_data=test_batches)
```

19 Appendix I: Dice score

```
def dice_coef_for_labels(y_true, y_pred, labels):
    y_true = tf.cast(y_true, tf.float32)
    y_pred = tf.cast(y_pred, tf.float32)
    dice_scores =
    for label in labels:
        # Create binary masks for the specific label
        y_true_label = K.cast(K.equal(y_true, label), tf.float32)
        y_pred_label = K.cast(K.equal(y_pred, label), tf.float32)
        # Flatten the masks
        y_true_f = K.flatten(y_true_label)
        y_pred_f = K.flatten(y_pred_label)
        intersection = K.sum(y_true_f * y_pred_f)
        dice_score = (2. * intersection + 1.) / (K.sum(y_true_f) + K.sum(y_pred_f)
+ 1.)
        dice_scores[label] = dice_score
        return dice_scores

y_pred = np.argmax(predictions, axis=-1)
dice_1 = []
dice_2 = []
for image1, image2 in zip(y_test, tf.convert_to_tensor(y_pred)):
    dice_1.append( dice_coef_for_labels(image1, image2, labels=[1]) )
    dice_2.append( dice_coef_for_labels(image1, image2, labels=[2]) )

dice_values_label_1 = [dice_scores[1] for dice_scores in dice_1]
dice_values_label_2 = [dice_scores[2] for dice_scores in dice_2]
mean_dice = np.mean(dice_values_label_1), np.mean(dice_values_label_2)
std_dice = np.std(dice_values_label_1), np.std(dice_values_label_2)
```