香港城市大學
City University of Hong Kong

# Department of Computer Science

# BSCCS Final Year Project 2020-2021
# Interim Report II

**20CS095**

**Computer-Aided Diagnosis with CT Scan for Marine Mammals**

**(Volume ___ of ___)**


Student Name : **TAI Hsiang-Yu**

Student No. : **55597349**

Programme Code : **BSCEGU4**


**For Official Use Only**

Supervisor : **Dr LEE, Chung Sing Victor**

Date : **June 15, 2021**

# 1. EXTENDED ABSTRACT

A CT scan, short for computed tomography scan, is a medical imaging technique that involves scanning and combining a series of X-ray measurements in order to reconstruct a more detailed scanning result than traditional X-ray scans. While CT scan imaging is one of the best imaging techniques in the medical field, it is sometimes difficult for doctors to interpret CT scan images manually. Recent advancement in deep learning and image processing, however, enables the possibility for an automated, computer-aided diagnosis system.

There has been an abundance of research regarding this topic, from creating a model for automated detection of lung cancer nodules in CT scan images (Makaju, et al., 2017), to the development of a CNN model that detects intracranial hemorrhage by directly interpreting sinograms from CT scans without image reconstruction (Lee, et al, 2019). There are also some products and tools on the market that utilizes deep learning for medical imaging purposes, e.g. qure.ai. The majority of these works, nonetheless, are targeted at humans. The main aim of this project is to develop a system that is able to automate the scanning and interpretation of CT scan results of marine mammals and detect anomalies by labeling them for each body part.

The CT scan images are provided by the State Key Laboratory of Marine Pollution (SKLMP) at CityU. The dataset consists of CT scan results of 13 dolphins, in DICOM format.

In this project, I first tried implementing the YOLO model and built a bounding-box classifier that is able to localize the position of the lung by drawing a rectangle box around it. Later, in order to improve the level of precision and thoroughness of the localization, I switched my method and used PyTorch to build a U-Net model from scratch. I trained two U-net classifiers that are able to localize the lung and the unhealthy region respectively by the means of image segmentation. Then, the models are thoroughly tested on the test set, achieving an accuracy of 0.99 and a Dice coefficient of 0.94. Lastly, by overlapping the diseased area with the whole lung area

and calculating the percentage of unhealthy lung pixels, I can come up with a numeric value that quantifies the level of sickness, which is the expected outcome of the project.

# 2. ACKNOWLEDGEMENTS

Firstly, I would like to give my sincerest gratitude to Dr. Victor Lee, who is my FYP supervisor and has given me valuable guidance throughout the course of my final year project. His mentorship, support, and patience have played a huge part in the completion of this project. Dr. Lee read through my project plan and two interim reports and gave me suggestions on how to improve; he also helped me communicate with Dr. Brian Kot, arranging meetings with him when I need to and helping me negotiate with Dr. Kot when we had a discrepancy in our understanding of the project expectation. Moreover, the bi-weekly meetings held by him also helped me continuously evaluate the direction my project should go toward, decide the best approach to go about technical tasks, and make sure I am on the right track. I would also like to give my utmost appreciation to Dr. Antoni Chan, who has also given me extremely helpful advice during our project meetings. Without the guidance and supervision of the two professors, I would have never been able to carry out this project and learn so much throughout the process. Furthermore, special thanks go to Dr. Brian Kot and Ms. Maria Robles from the State Key Laboratory of Marine Pollution (SKLMP) at CityU. They gave me this opportunity to conduct this project, provided me with the dataset, helped me with the labeling work, and gave me invaluable advice on the biological side of things. A huge thank you to everyone involved in this project and have been of guidance to me, without whom I would never be able to carry out this project by myself.

# 3. TABLE OF CONTENTS

# 4. INTRODUCTION

## 4.1. Project Aims & Objectives

This project aims to build a model that automates the process of reading and interpreting the CT scans of deceased marine mammals with the help of machine learning and computer vision. With the current process of researchers manually analyzing 3D CT scan models being rather error-prone and at risk of misjudgment, I aspire to design a machine learning network that is able to achieve the following goal: automatically locating the target body region/organs from the scan, determining the region that is unhealthy, and ultimately computing a percentage of unhealthy region within the entire lung.

## 4.2. Problem Scope

The scope of the project is to provide an autonomous CT scan analysis system for dolphins and, by extension, marine mammals in general. It focuses on two main aspects, namely the accurate localization of target organs/region within the animal's body, whether or not it is healthy, and how healthy it is. Our dataset is kindly provided by Mr. Brian Kot, a researcher at the State Key Laboratory of Marine Pollution (SKLMP) at CityU. It consists of a set of CT scan results of deceased dolphins, with each scan result consisting of a series of around 2,400 2D scanned images, which, with the help of computer software, can be reconstructed to a 3D model. More details about the dataset are provided in a later section. Ideally, it can assist marine biologists in efficiently inspecting the inner structure of marine mammals and figure out the disease they suffered from, helping them with further figuring out the cause of death; in terms of its usages in the medical field, it can potentially also be deployed by veterinarians for clinical use.

## 4.3. Problem Motivation

Computer tomography scanning, or CT scans for short, has long been one of the standard medical imaging approaches that allow doctors and researchers detailed

insight into the examinee's body. Through computer processing, it combines a series of X-ray images taken from different angles to reconstruct a 3D model of the scanned object to allow the viewer to examine the internal tissues and organs without cutting. It is a powerful imaging procedure that is found to be extremely useful both in hospitals and for research purposes. However, the current interpretation process of CT scans is mostly done by doctors and radiologists manually; the most current software help with is the processing and construction of the 3D CT scan model, without any analytical ability. Misjudgments are unavoidable for doctors manually analyzing the scan results. Also, it takes time and experience for a medical student to be trained to correctly interpret CT scan results of each body part. Furthermore, with the three-dimensional nature of CT scan images, it is even less ideal to solely rely on the observational prowess of the naked eye.

With the rapid advancements in deep learning and computer vision, however, computers are more than capable of assisting humans with the interpretation of medical images. Its application ranges from automated detection of lung cancer nodules in chest CT scan results [9], to checking for cerebral hemorrhage, cranial fracture, and infarctions by directly interpreting sinograms from head CT scans without image reconstruction [7]. Other use cases of machine learning in the medical field include the categorization of benign/malignant lesions, with the precision being on par with professional radiologists [5]. All of the above are robust proofs that the advancement in machine learning and computer vision technology can be of a great boost to the betterment of automated CT scan analysis. What is noticeable is that most developments on this topic seem to center around the analysis of human CT scans; not much research has been dedicated to those of animals. Therefore, for this project, I am aiming to build a system that is able to automate the interpretation of CT scan results of dolphins and accurately localize the abnormal body parts through relevant machine learning techniques.

# 5. LITERATURE REVIEW

## 5.1. Organ localization/segmentation

When it comes to the localization of organs in medical images, there are quite a plethora of related works available.

Andréanne Lemay [8] proposed a kidney detection model for CT scans using YOLOv3, a popular object detection network. Lemay opted to use the YOLO network because of its fast processing speed in comparison to other existing networks, which can be hugely beneficial for medical treatments such as laparoscopic surgeries or adaptive radiotherapy, both of which require real-time evaluation. Lemay focused on both 2D and 3D kidney recognition with YOLO and compared the results with another popular model, SSD (single shot detector).

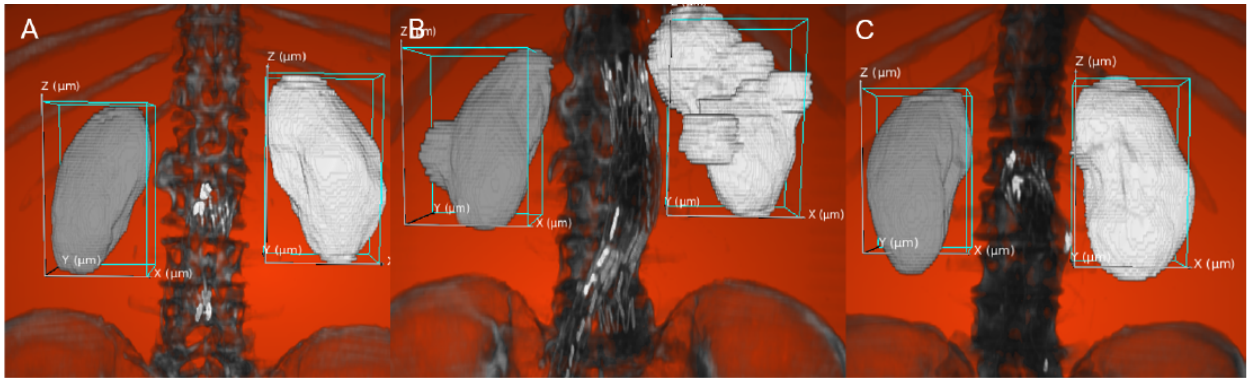
The dataset they used (KiTS2019) contains 14 CT scans, including both healthy and tumoral kidneys and contains around 3000 2D images of size 512*512, of which 1200 contain kidneys. The inclusion of both normal and cancerous kidneys offers variation and heterogeneity to the dataset, teaching the model invariance to such differences in sizes, shapes, and other conditions.

As far as the results go, Lemay witnessed quite satisfactory outcomes. An average Dice coefficient of 0.851 was achieved, on par with that of the SSD model; it also beats the latter in terms of IoU (Intersection over Union, a classification evaluation metric) score by two percentiles. As for the 3D localization, Lemay took the predicted bounding boxes on each 2D CT scan slice and performed a 3D generalization of the non-maximum suppression algorithm to obtain the smoothest, most optimal 3D result [Fig. 1]. Comparing the results with the ground truth, it is shown to achieve a Dice score of 0.742 and an IoU of around 0.61.

There is still room for improvement, of course. Regarding the 2D localization, the model tends to face difficulties in identifying organs of unusual ratio or morphology. It is found out that pathologically bloated or shrunken kidneys are harder to be detected due to their unknown ratio configurations. As for the 3D recognition aspect, since they obtain the 3D region by reconstructing it from 2D slices, it is observed to not work too well if the target object/organ is not aligned with the Z-axis, or in other

words, not completely vertically upright with respect to the X-Y horizontal plane. Furthermore, it is found out that 3D localization is more prone to inaccuracy since any errors and deviations made in the 2D predictions tend to be propagated upward to the 3D bounding boxes.

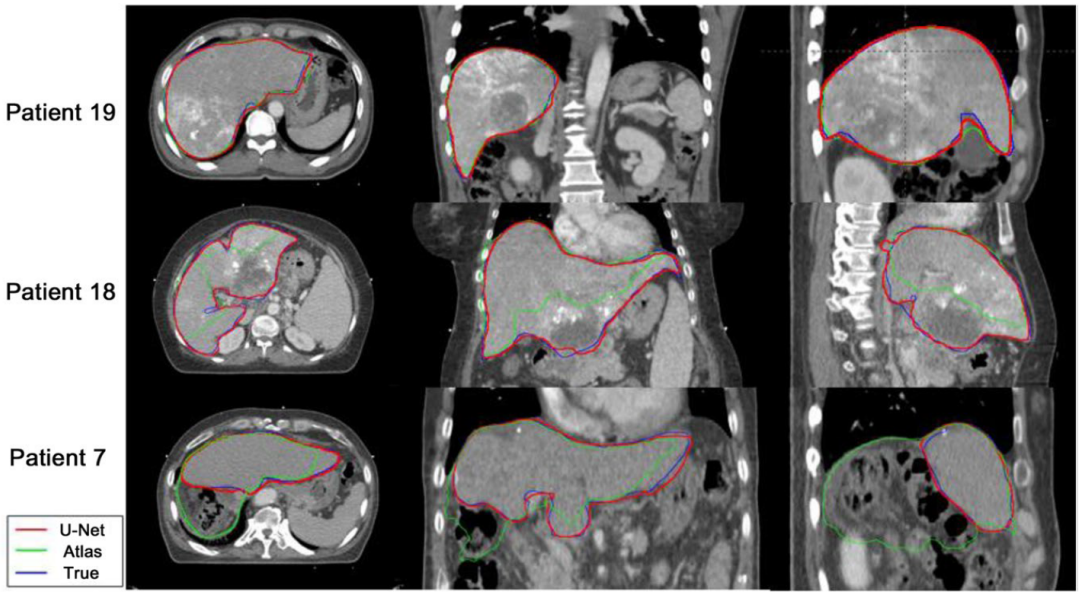
To conclude, improvements can be made by ameliorating the model's ability to generalize extremities in sizes and shapes as well as finding ways to improve the non-max suppression algorithm used to construct the 3D prediction.




[Fig. 1] Demonstration of reconstructing a 3d lung detection prediction with 2d bounding box predictions

Another popular image segmentation technique is the usage of U-net, a 3D patch-based deep CNN. Hojin Kim et al. [2] developed an abdominal multi-organ auto segmentation model using U-net for the purpose of improving organ detection efficiency in radiotherapy. Their data consists of 3D CT scans of size 64*64*64 and include organs such as livers, stomachs, duodenums, and kidneys, adding up to a total of 120 CT scans.

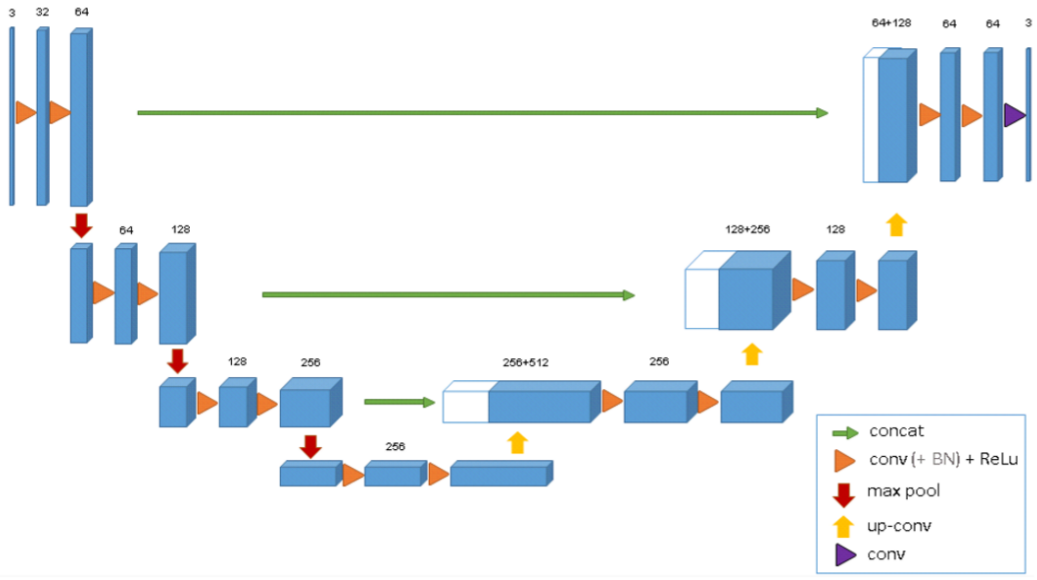
For evaluation, they compare the results against those produced by the Atlas-based method, another fairly common medical image segmentation technique, as well as the ground truth. It is shown that the model achieves Dice scores of 0.96, 0.81, 0.6, and 0.91 for livers, stomachs, duodenum, and kidney detection respectively. Apart from the slight drop in duodenum detection, which can be fairly challenging due to its complex structure and placement, it performs exceptionally well in all other aspects. Below is a figure showing its segregation results compared with that of atlas-based and ground truth [Fig. 2].




[Fig. 2] Segregation results using U-Net (red) juxtaposed with that of the atlas-based segmentation (green), and ground truth manual contours (blue)

The main usage of U-net is to take into account the scarcity of data. Higher-dimensional medical image data typically means it requires a larger dataset size to build the network; with U-net, however, it is a convolutional network that makes the most out of data augmentation to exploit the data in hand in a more efficient manner. The motivation behind the creation of U-net was primarily to make it easier for network training regarding medical images, which are mostly either 3D or relatively scarce in number. Also, more often than not, the objective of medical image processing includes not only the detection of a certain component but also the exact location of it, i.e. assigning a class label to each pixel, meaning the vanilla CNN model would not suffice. Previously, in order to combat this obstacle, Ciresan et al. [13] came up with a solution with which they make use of the “sliding window” technique; instead of the whole image, they provide as input a “patch” of the image at a time. This not only enables them to pin down the location of the target object but also in a sense creates more input images to train, since they only provide a portion of the image at a time, which essentially serves as a makeshift data augmentation technique. There are problems with this method though: it drastically increases the training time, having to train for each patch separately; there also tends to be a lot of

overlap among neighboring patches. With U-net, this problem can potentially be less of a concern. The architecture of the contracting/encoder path and expanding/decoder path [Fig. 3] ensures that the network can both pick up the feature and localize it.




[Fig. 3] Volumetric segmentation with 3D U-net

## 5.2. Image Classification

There has been an abundance of related research on the integration of computer-aided systems for medical image analysis. In “Feature extraction and LDA based classification of lung nodules in chest CT scan images”, Aggarwal, et al. designed a model to detect lung nodules from chest scans by using LDA as the classifier. The model extracts features through geometrical, statistical, and gray-level characteristics. Optimal thresholding is applied for the segmentation task. While they did achieve satisfactory accuracy and sensitivity, the specificity was not up to the acceptable standard, averaging at 53% [1].

In terms of a more machine-learning-oriented method, Chilamkurthy, et al. [3] developed a set of algorithms that automatically detect cranial-related pathology such as calvarial fractures, ICH (intracranial hemorrhage), SAH (subarachnoid hemorrhage), etc. They used two datasets for their head CT scan source, the

CQ500 dataset (contains a total of 491 head scans) and the Qure25k dataset (12051 male and 9043 female scans respectively), and proposed a hybrid and rather original method of labeling their data; for one dataset, they have professional radiologists manually label their data for the presence of each pathological observation; for the other dataset, however, they make use of an NLP algorithm to read the diagnostic description off of the corresponding clinical radiology report. In regards to building the algorithm itself, they used U-net and a DeepLab-based architecture for the semantic image segmentation of the hemorrhages and cranial fractures respectively. ResnNet18 was applied for the detection of each type of hemorrhage in a slice. The combination of these networks yielded a fairly satisfactory result, with the sensitivity averaging at an astonishingly high 86%, and an equally impressive average specificity of 81%. Despite the result, there is still room for improvement, nonetheless; that is, the relatively small size of the dataset for certain types of illness. For instance, there were only 13 CT scans in which extradural hemorrhage was present, inevitably making the result of extradural hemorrhage detection lose some statistical significance.

And in regards to the analysis of medical images of animals specifically, there has been a rather limited amount of research regarding this topic. Ziyue Xu, et al. [12] proposed a computer-aided model to analyze lung scan images for infectious pulmonary nodules in small animals. They collected a total of 133 lung CT scans of rabbits and ferrets and performed pathological lung segmentation with an average Dice similarity coefficient (index for evaluating image segmentation performance) of 90%. While their model performs relatively well on certain datasets, the use of threshold-based segmentation techniques fails to take into consideration the variance in the intensity of CT scans, meaning the performance might fluctuate according to the nature of the dataset.

# 6. PROPOSED DESIGN & DETAILED IMPLEMENTATION

## 6.1. Dataset

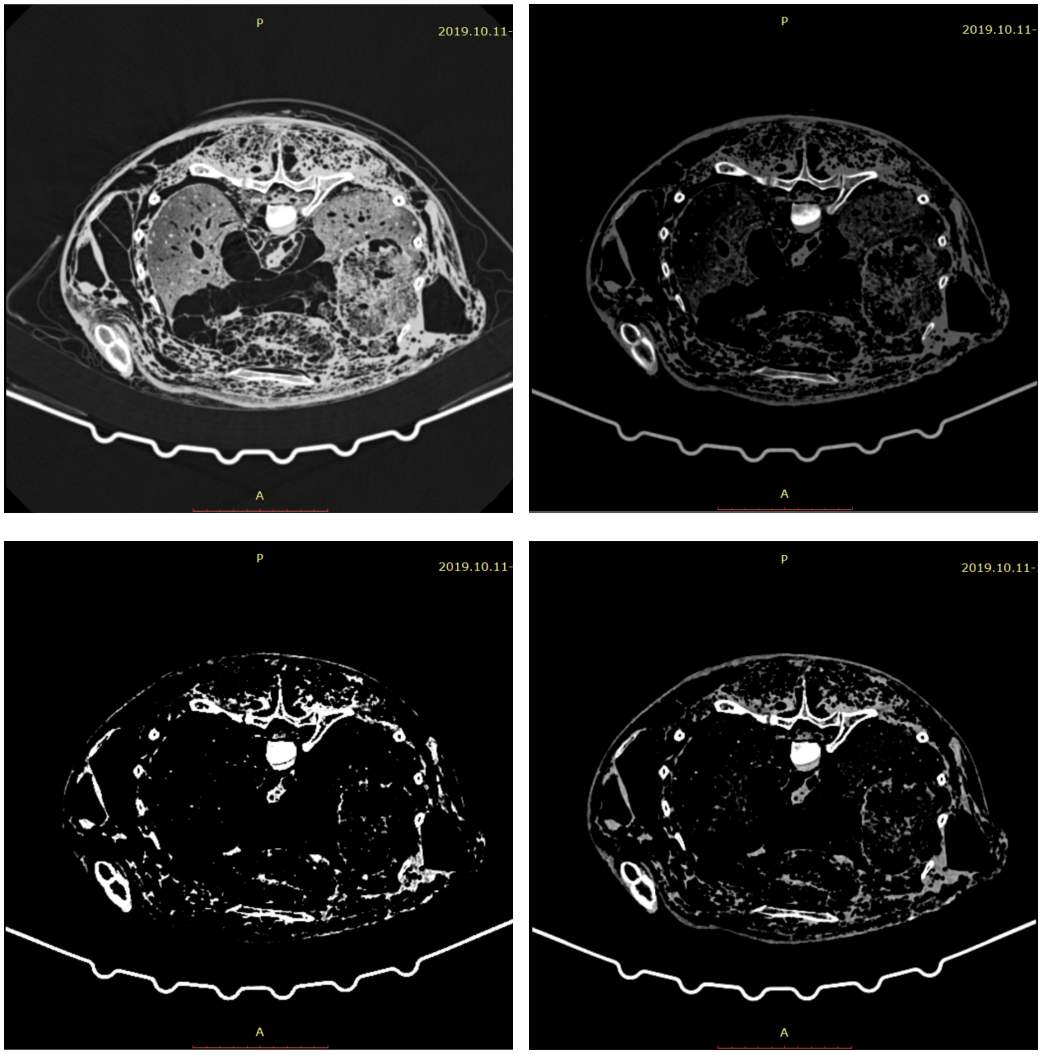
### 6.1.1. Background & Quantity

I would like to start off by elaborating on the dataset I obtained, kindly provided by Mr. Brian Kot, a marine biologist at the State Key Laboratory of Marine Pollution at CityU. The dataset is made up of a number of 3D CT scan models of deceased dolphins; each CT scan result consists of around 2,000 2D scan images, with each of them being a horizontal slice of the dolphin. With the help of certain computer software, these 2D images can be reconstructed into the original 3D model, enabling doctors and researchers to further examine the scanned creature in a more detailed nature.

For my project, I am using RadiAnt DICOM Viewer, a popular medical image viewer software that comes with a number of handy CT image viewing, reconstruction, and manipulation tools. For instance, one of the most important parts of my project is to detect the location of the lungs within the dolphin. However, due to the variation in the intensity of different CT scans, the lungs might not always show very clearly in the raw images under the default settings. Fortunately, RadiAnt DICOM Viewer comes with a built-in “gray-level mapping adjustment” function, which allows me to switch between the different levels of intensity/grayscale-ness to see the desired organs and regions more clearly [Fig. 4]. To offer a bit of context, gray-level mapping, also known as windowing, contrast enhancement, or histogram modification, is essentially the process of adjusting the greyscale level of CT scan and manipulating the appearance of the images to allow the user to highlight specific tissues or structures. The window level modifies how bright the picture is, whereas the window width takes care of the picture’s contrast.

RadiAnt also offers another feature, which is the automatic conversion from DICOM files to jpg images. It is especially convenient since the raw images are all in DICOM format, a standard file format for medical images, and are less convenient to operate on in my models compared to traditional JPG images. With these handy

functionalities, I am able to select the “CT Lung” windowing level -- which highlights the lungs most clearly -- and then convert them all into JPG format, which helps tremendously in manipulating and operating on the images later. As for how to select an optimal gray level to highlight the lungs clearly, the coworkers at SKLMP have kindly provided me help in this regard by selecting the grayscale level for me beforehand. Ms. Maria Robles has helped me with data labeling, dolphin selection, and CT windowing adjustment.




[Fig. 4] The same CT scan image slice under different gray-level mapping scheme

Below is a list of dolphin CT scans I have in my dataset and their relevant information such as scan date, code (decomposition condition, with 1 being mildest and 4 being most decomposed), sec, age group, and pathology description.

| ID | SCAN DATE | CODE | SEX | AGE GROUP | PATHOLOGY |
|---|---|---|---|---|---|
| NP20-1703 (PM1) | 17/03/2020 | 1 | Female | Juvenile | Fluid aspiration leads to pulmonary insufficiency: Drowning |
| SC15-1601 | 2015-01-16 | 1 | Male | Sub-adult | Lung consolidation, GGO (ground-glass opacity) |
| QTW14-00 579 | 2021-04-15 | 1 | Female | Juvenile | Lung consolidation |
| NP15-2101 | 2015-01-21 | 2 | Female | Juvenile | Suspect pneumonia or other pulmonary diseases |
| OT20-0201 | 2020-01-02 | 2 | Female | Sub-adult | Severe pulmonary complication (GGO) |
| NP14-0604 | 2014-04-06 | 2 | Female | ? | Lung consolidation, GGO |
| NP19-1412 | 2019-12-14 | 2 | Male | Juvenile | Lung consolidation, GGO (infection-related) |
| NP19-1412 | 2019-12-17 | 2 | Male | Juvenile | Severe parasitic pneumonia |
| NP17-2610 | 2017-10-26 | 3 | Male | Sub-adult | Pneumonia |
| NP14-2308 | 2014-08-23 | 3 | Female | ? | Lung consolidation, GGO |
| SC15-1505 | 2015-05-15 | 3 | Female | Neonate | Lung consolidation |
| SC14-3004 | 2014-05-02 | 3 | Male | ? | Focalized granulomas |

| | | | | | |
|---|---|---|---|---|---|
| SC21-2404 | 2021-04-27 | 4 | Male | Adult | Lung consolidation and GGO |
| SC21-2204 | 2021-04-27 | 4 | Female | Adult | Severe lung consolidation and GGO |
| NP15-1811 | 2015-11-18 | 4 | Female | Adult | Lung consolidation, parasitic infection |
| NP18-2112 | 2018-12-22 | 4 | Female | Juvenile | Severe parasitic infection |
| NP18-2012 | 2018-12-20 | 4 | Female | Juvenile | Parasitic infection |
| NP18-0412 | 2018-12-04 | 4 | Female | Juvenile | Severe parasitic infection |
| NP18-0701 | 2018-01-08 | 4 | Male | juvenile to sub-adult | Moderate parasitic infection |
| NP19-1811 B | 2019-11-19 | 4 | Male | Adult | Moderate parasitic infection |
| NP19-2010 | 2019-10-21 | 4 | Male | Sub-adult to adult | Pulmonary edema, mild parasitic infection |
| NP19-1110 | 2019-10-11 | 4 | Female | Juvenile | Severe parasitic pneumonia |
| NP20-0912 | 2020-12-10 | 4 | Male | Sub-adult | Ground glass opacity and pulmonary edema. Moderate parasitic pneumonia |

### 6.1.2. Labeling

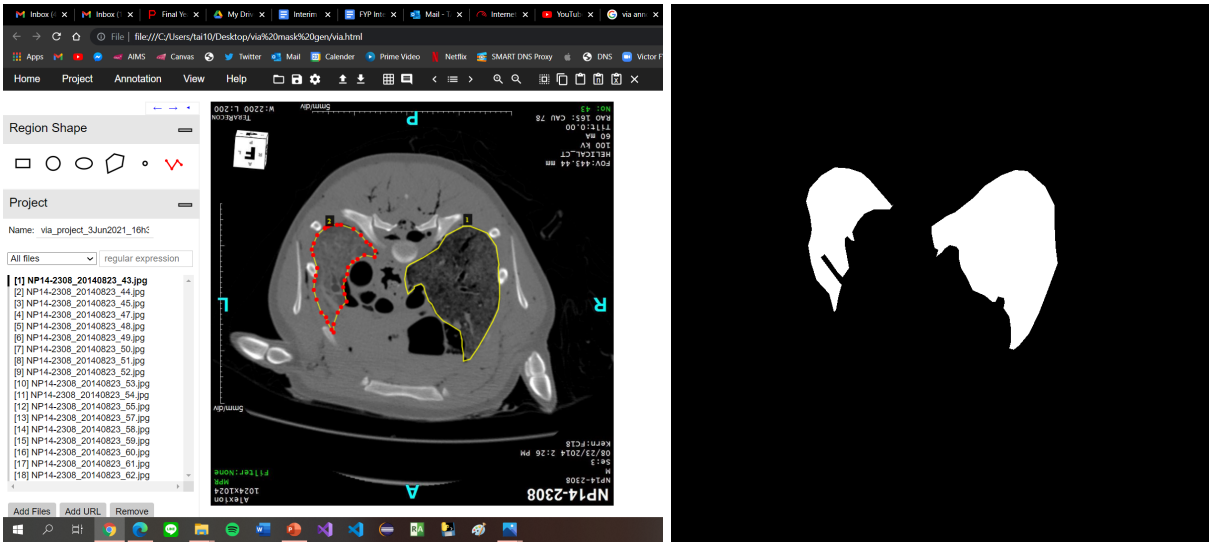
#### 6.1.2.1. For YOLO bounding box method

The dataset was initially offered to me unlabeled, so I had to label the dataset myself in the early stages when I was working on the bounding box identification

method. Mr. Brian Kot was kind enough to teach me how to locate the lungs within the dolphin's CT scans with the naked eye, so I was able to manually label them by myself on each of the images containing the lungs. I am making use of a tool called LabelImg (https://github.com/tzutalin/labelImg), a graphical image annotation software that allows me to draw rectangular bounding boxes around the target object to detect [Fig. 6]. It then automatically generates either a text file or an XML file (based on the user's need) corresponding to the relative position of the object and the class it should be classified as. I can later feed both the images and their corresponding text file with their positioning information as the ground truth to my model (mainly YOLO) for training purposes.

#### 6.1.2.2. For U-Net image segmentation

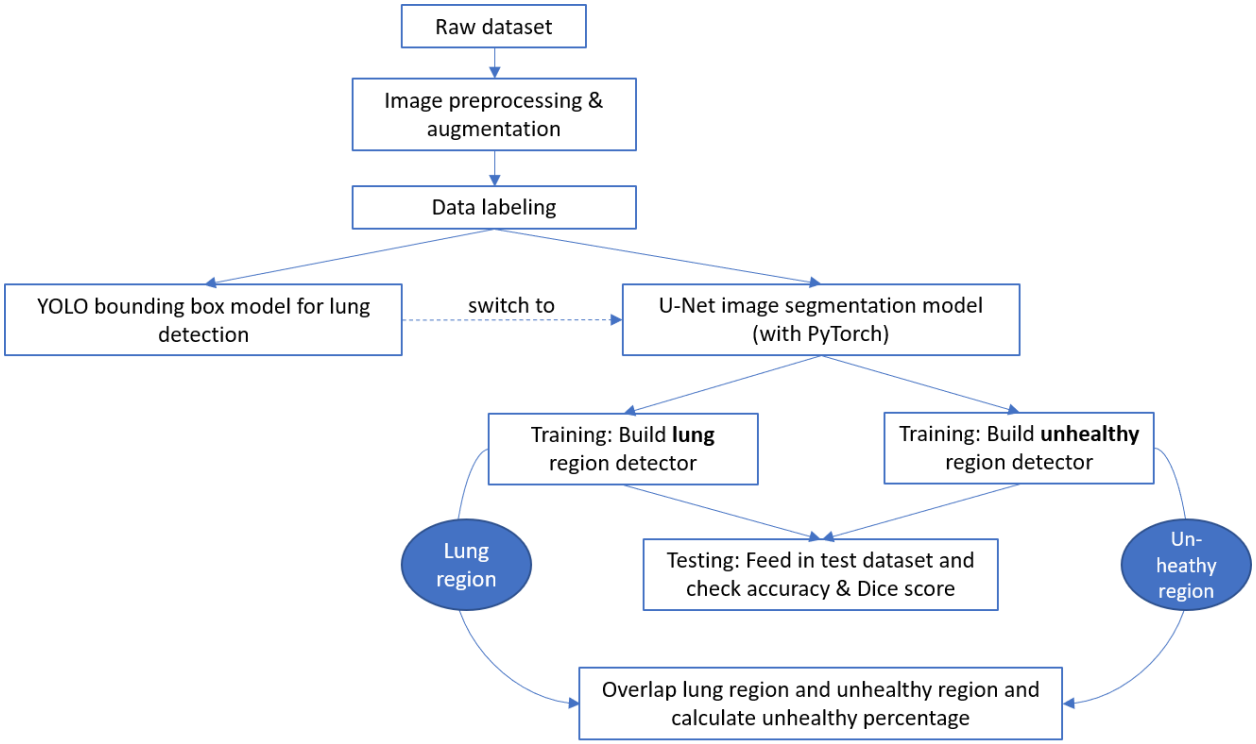
As for the image segmentation task that I have later worked on, the labeling is a bit more complicated and time-consuming. I was lucky enough to have Ms. Maria Robles, one of Dr. Kot's Ph.D. students, help me in this regard. For image segmentation, the exact shape/contour of the object is needed. I made use of the VIA Image Annotator, a very popular online image annotation tool (https://www.robots.ox.ac.uk/~vgg/software/via/app/via_image_annotator.html); for each image; I trace the edge of the target object (in our case, the lung) and keep clicking along the way to mark points that eventually form a polygon representing the area selected [Fig. 5]. A JSON file is then generated, storing all the points forming the shape. Lastly, I wrote a simple Python program to convert the JSON files into actual black-and-white mask images, in which the white area is the part we want. Now the raw images, along with the corresponding mask images, can be fed into the U-net to train the lung/pathology detection model.



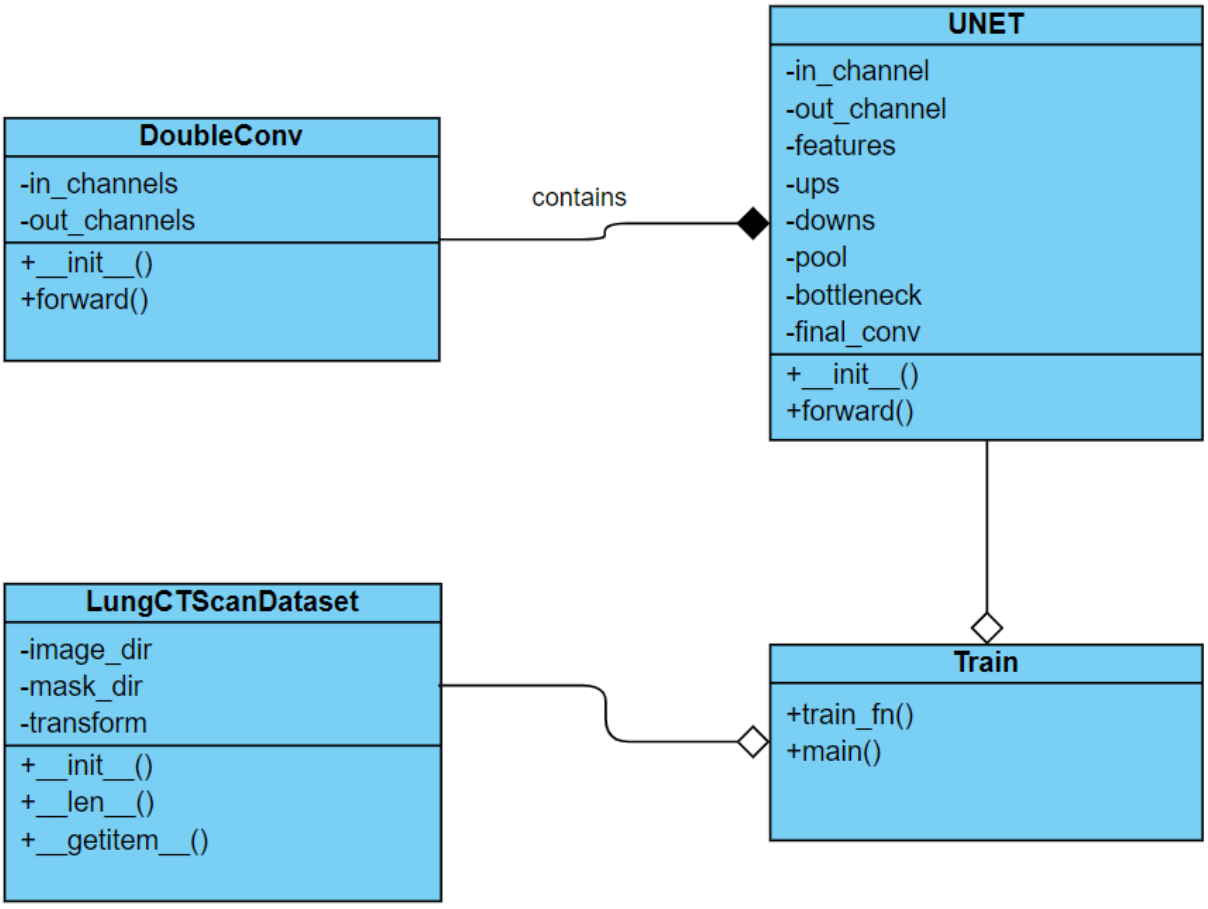

[Fig. 5] Left: the interface of VIA Image Annotator. Right: the result mask image.

## 6.2. Brief System Overview

This project aims to locate the lung region from a CT scan, analyze its health status, and output an index quantifying how healthy/unhealthy the animal is. For the lung localization, I tried two approaches. First, I implemented the YOLO object detection network, building a bounding box lung detection model. Then, in order to enhance the precision, I switched to U-Net, a network famous for image segmentation tasks, to capture the exact contours of the lung. After that, I trained another U-net segmentation classifier, but this time for the detection of the unhealthy region. Therefore, given a CT image, my lung detector and pathology detector can determine the lung and unhealthy region respectively. Finally, I can calculate the percentage of unhealthy pixels over the lung pixels as the final result. Below is a brief system diagram demonstrating the workflow:




And below is a class diagram of my U-Net project written in Python/PyTorch. There is a `LungCTScanDataSet` class, which loads the dataset via the designated directory and applies transformation/augmentation on the images. The `DoubleConv` class is the double 3*3 convolution layer repeatedly seen in the U-Net structure and is the building block for our model. The `UNET` class is the actual U-Net model and the heart and soul of our program, containing a down (contracting) path, an up (expansive) path, a bottleneck layer in between, and a final convolution layer. Finally, the `train_fn()` represents the process of loading the dataset via `LungCTScanDataSet` and training a `UNET` model for one epoch, and the `main()` function runs `train_fn()` for the specified number of epoch times. The details will be elaborated on in a later section (section 5.5: *U-Net Image Segmentation & Detailed Implementation*).




## 6.3. Data Preparation/preprocessing & Image Segmentation

As widely known, machine learning networks for biomedical images often suffer gravely from the insufficiency of data. One of the most common approaches to resolving this problem is to adopt data augmentation. In essence, it is the technique of generating more training from the original data source through a series of deformation techniques, including scaling, rotation, horizontal/vertical flipping, cropping, etc. This teaches the model invariance to such differences and increases the size of our dataset [11], making the model more robust even when examining objects of unusual distortion or ratio. It is a technique particularly common for image segmentation of medical images since the deformation and variation in size and color are very commonplace in organs, tissues, and cells. For this task, I am planning on using the ImageDataGenerator tool provided by Keras, which does

exactly what we need -- it helps generate more new data by augmenting the original dataset. It allows users to explicitly specify the rotation angle, brightness alteration, the rescaling factor, random vertical/horizontal flipping, zooming ratio, etc., perfectly catering to my need for mass-generating more image data from my relatively small data batch. This tool is utilized when I augment the images in the test set of YOLO object detection.

I will also perform image augmentation on the fly in my program. In my dataset class written in Python (which will be elaborated on later), it takes a parameter "*transform*", which indicates the type of transformation to perform on the images. I use the *albumentation* library for this task, in my U-Net PyTorch project. The usage of U-Net, in my case, is not to generate more images and to enlarge the dataset per se, but to augment the images directly to create some variation (in brightness, scaling, contrast, …) and enhance the program's ability to generalize. This will be elaborated on further in a later section (Section 5.5.4.1. `dataset.py`).

## 6.4. YOLO Bounding Box Approach

### 6.4.1. Brief Introduction & Past Use Case

YOLO is a relatively new model, having only emerged to the scene circa. 2016, but has gained tremendous attention and praise for its good performance on object detection tasks. It has been proven to work well with medical images and organ localization too; as mentioned in the literature review section, Andréanne Lemay et al. [8] built a kidney recognition model from CT scans using YOLO. She obtained 14 CT scans from the public KiTS2019 dataset. It contains around 3000 2D CT images, of which 1200 contain kidneys. The data is fed to the YOLOv3 model after some standard gray-level mapping and result in a Dice coefficient of 0.85 and an IoU of around 0.74.

### 6.4.2. How it works

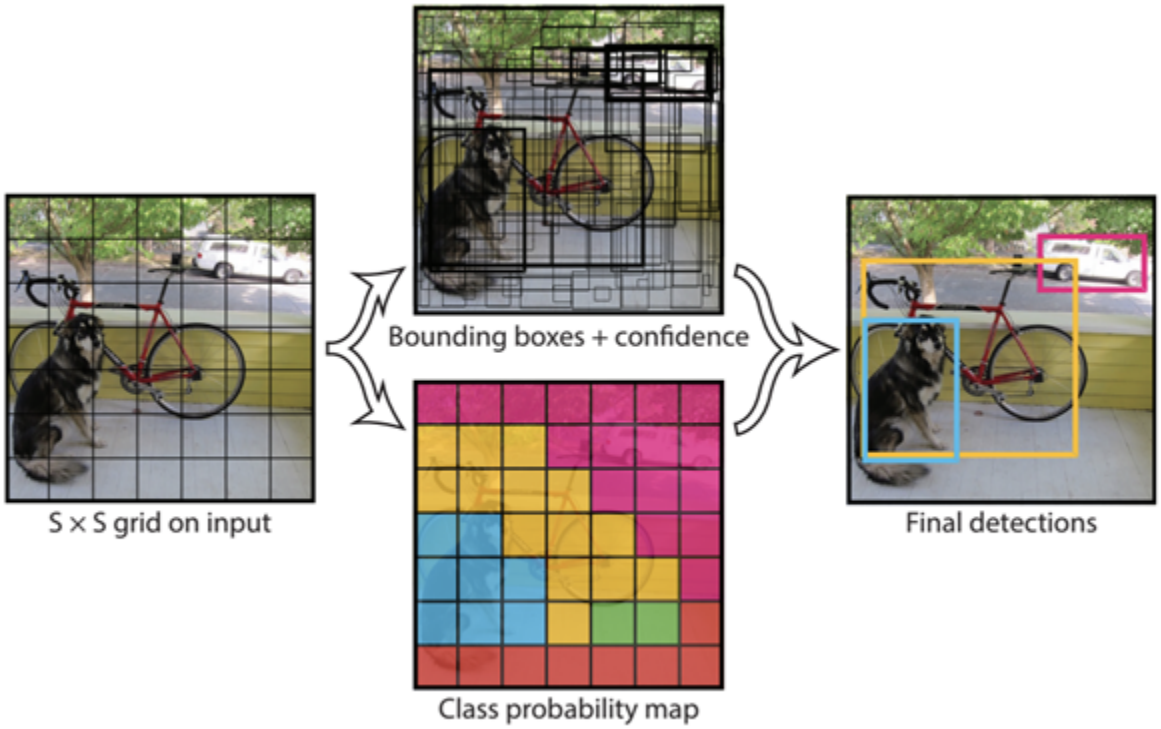
To start the detection task, the input image is split into S*S grids, each of which (if inside a ground truth box) predicts B bounding boxes and confidence scores for those boxes.

Each of the bounding boxes comprises five prediction-related variables. The (x, y) coordinates, representing the relative position of the center of the bounding box. Two more variables, h and w, record the height and width of the box [Fig. 6]. Lastly, the confidence prediction evaluates the probability for the box to contain the target object multiplied by the IOU between the prediction and any ground-truth box, or ***P(Obj) * IoU***. IOU (intersection over union) is a very common performance evaluation metric that will be further elaborated in a later section, but it basically evaluates the proportion of overlap divided by union.

For each grid, the model also calculates a conditional probability for each of the C classes, under the condition that the grid contains an object, therefore denoted as $\boldsymbol{P(Class_i \mid Obj)}$. At test time, by multiplying the class-specific probability with the individual box confidence,

$$\boldsymbol{P(Class_i \mid Obj) * P(Obj) * IoU = P(Class_i) * IoU},$$

leaving us with class-specific confidence scores of each box.




[Fig. 6] The YOLO Model. The model considers detection to be a regression-like task. It splits the input image into an S*S grid, each of which predicts B bounding boxes, those boxes' confidence, and C class probabilities. The model encodes the predictions as an S*S*(5B+C) tensor.

The final aggregated prediction can be contained in a tensor of size S*S*(B*5+C), with S*S grids, each of which evaluates B bounding boxes, the respective confidence for those boxes, and C class probabilities. The original YOLO network was trained on the Pascal VOC dataset, which contains 20 classes (C=20), plugging in the other variables that they used (S=7, B=2), they add up to around 1470 outputs, which really is not that many parameters for a neural network to predict, hence the performance boost.

### 6.4.3. Advantages of YOLO

One of the main advantages of using YOLO is its speed. Most of the other object detection models repurpose classifiers for detecting purposes. YOLO, on the other hand, reframes object detection as a regression task of spatially separate bounding boxes and the corresponding class-specific probabilities. A single neural network directly predicts bounding boxes and classification probabilities from the full image in one iteration, instead of drawing multiple sliding windows and performing predictions with multiple classifiers. Because the whole detection procedure is one single network, it can be directly optimized end-to-end on its recognition performance. With the YOLO model, it is able to determine what objects are in the image and where they are in one go, instead of having to go through multiple evaluations. In terms of statistics, YOLO can run at 45 FPS (frame/sec), which is 900 times faster than R-CNN, and 90 times faster than Fast R-CNN. It could be greatly beneficial to my project since in order to perform the lung localization, the model has to skim through around 2,400 images per model to come up with a 3D volumetric bounding box prediction. YOLO's speed can save the system quite some time in that regard, especially considering lung detection is just the first part of the system's design.

Another plus side to using YOLO is its capability of generalizing different representations of objects. Despite being trained with normal images, when being tested on images of varied styles and texture, e.g. artworks, cartoons. Etc., it still maintains a high level of accuracy seen on normal images, outperforming other detection models such as DPM (Deformable Part Models) and R-CNN by quite a margin. With it being highly generalizable, it is expected to perform well when faced with

new domains or unexpected inputs, which suits medical image analysis greatly, where heterogeneity and random morphology are expected.

### 6.4.4. Integration In My Project & Current Result

Over the past couple of months, aside from reviewing relevant literature about my project topic, I also went ahead and implemented the lung detection system with YOLO, as I had planned in the previous section.

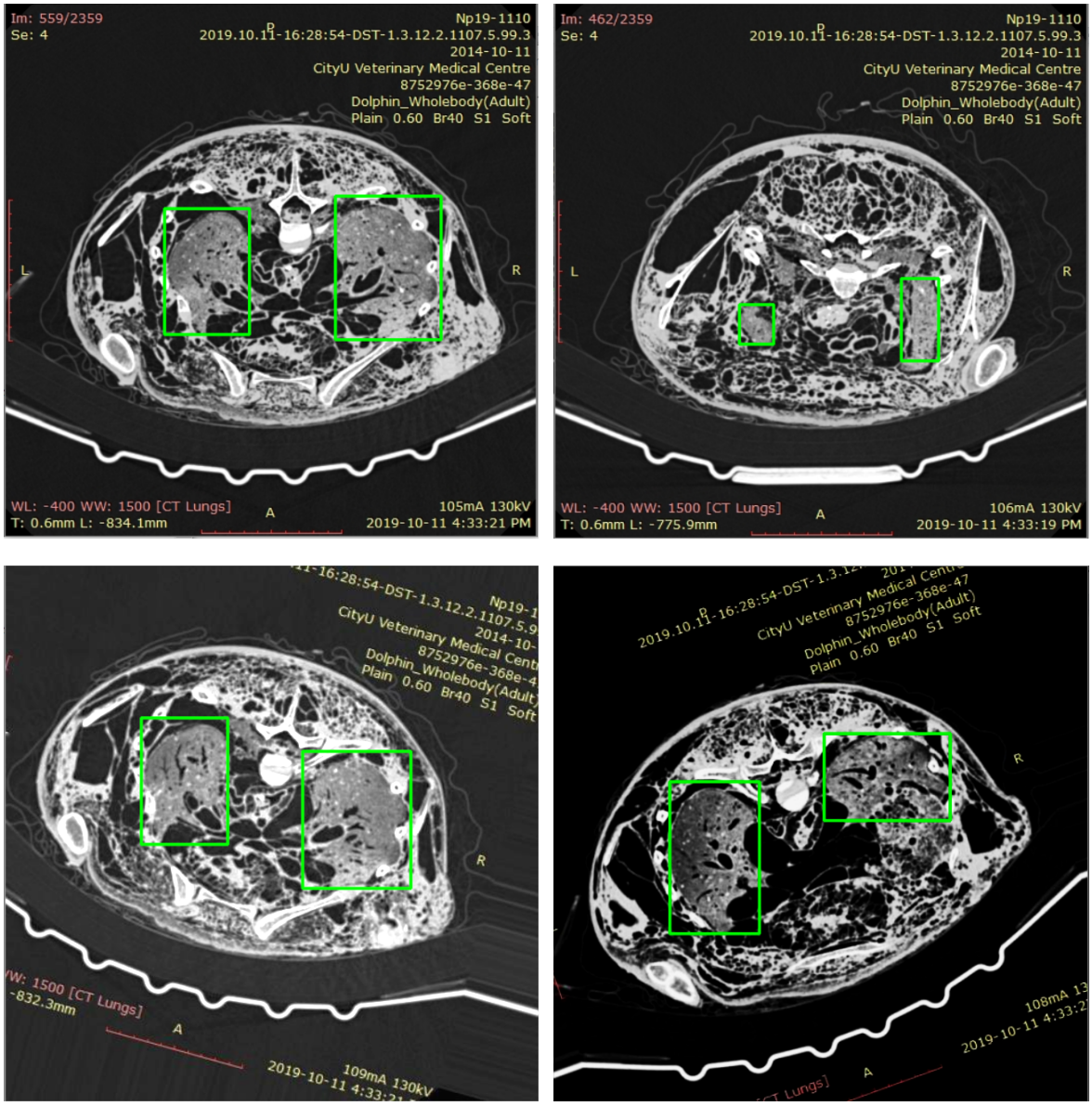
I first made use of the aforementioned LabelImg tool to manually label each of my images with lungs present. At the time (during which I was implementing the YOLO as I was still working on the bounding-box approach), I had six dolphins, each containing around 200-230 images with lungs in them, which adds up to 1307 2D images. Out of them, 90% are used for training, while the other 10% are included in the test set. I also make use of the ImageDataGenerator tool offered by Keras to perform image segmentation on my test set to increase the dataset. I ended up with 1176 training images and 260 test images.

I then installed Darknet, an open-source neural network framework set up by the creator of YOLO, Joseph Redmon, and downloaded a pre-trained YOLO model, “darknet53”, off of Darknet. It is a classification weights file containing the weights for a convolutional network pre-trained on ImageNet, with already 1000 classes classifiable. And then, I further trained the model by plugging in my dataset (containing the raw images and corresponding text files denoting the lung’s position) and the required configuration file. It is now able to detect dolphin lungs within 2D CT scan images, with a brand-new class “Lung” being added to the model. Below is a list of hyperparameters set up for the training process:

| Parameter | Value |
|---|---|
| Batch size | 64 |
| Learning rate | $10^{-3}$ |

| | |
|---|---|
| Number of epochs | 4000 |
| Image height | 512 |
| Image width | 512 |

After completing the training, it is time to test the result of the classifier. I prepared my test set by making use of the Keras ImageDataGenerater and created more images of different scaling ratios, rotation angles, brightness, etc. for testing purposes. I then fed those augmented images, along with the raw testing images that weren't used for training, into the detection model and examined the results. Shown below is the outcome of the lung detector [Fig. 7]:




[Fig. 7] The model correctly locates the lungs of different sizes and shapes within the CT images. Upper row: the model tested raw images. Bottom row: images augmented by ImageDataGenerator

Additional remarks: all the operations and coding were done on Google Colab with GPU enabled. I wanted to make use of the NVIDIA GPU Google Colab offers to accelerate the training process. The pre-trained model is downloaded off of Darknet.

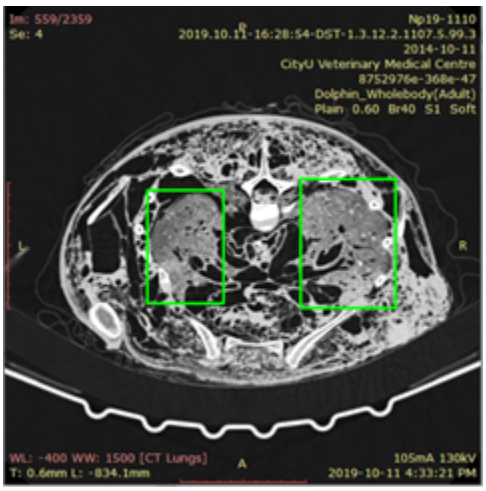
## 6.5. Changes In Project Requirements, Outcome Expectations, & Methodology

Some fairly drastic changes occurred after the submission of Interim Report I; before that, I finished the aforementioned YOLO detection in the previous section and was about to start with the image classification part. However, the researchers at the State Key Laboratory of Marine Pollution, who commissioned me to develop this computer-aided CT scan analysis model, indicated a modification in project requirement and outcome, which I have decided to mention in detail in the report for the sake of clarification. The changes include a switch in the lung detection method, dataset labeling, and a change in the expected outcome.

### 6.5.1. Change in Lung Detection Method

- Previous approach:

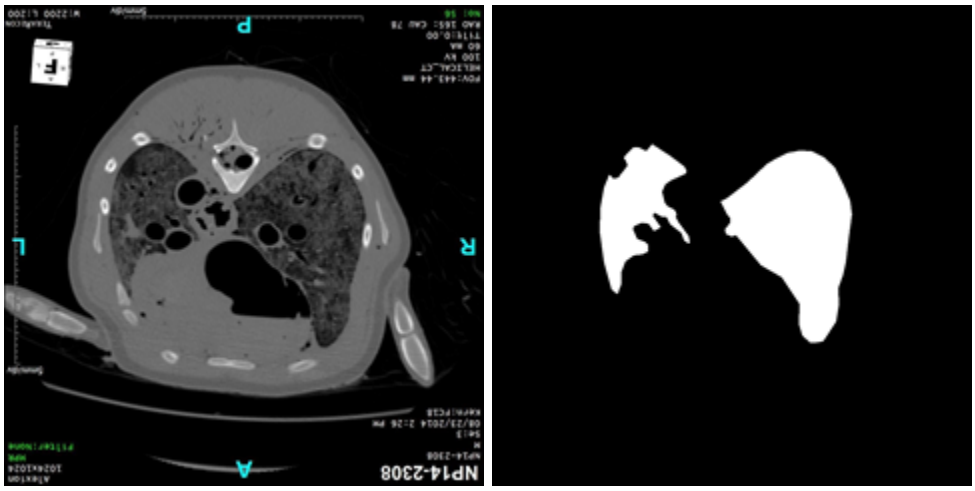
Previously, I was performing lung detection using the YOLO bounding box detection method. For the dataset, I marked the bounding box areas that contain a lung on each 2D image slice. Then, I made use of a pre-trained model from Darknet and further trained it on my data. I came up with a bounding-box lung detection model [Fig. 8] with a satisfactory Dice score of 0.83.




[Fig. 8] YOLO bounding box lung detection model

- New approach:

However, in order to improve the precision and specificity of my model, I have decided to adopt the technique of image segmentation to find the exact contour of each lung. I used the fairly new but highly developed U-net structure which has an exceptional reputation in terms of image segmentation tasks. Given an image, it is now able to determine whether each pixel is a part of the lung or not; therefore, it is able to resolve the specific shape of the lung, instead of just drawing a rectangular bounding box which contains other irrelevant, adjacent parts most of the time. Its output is in the form of a black and white image called a “mask” image, with the white area representing the region selected [Fig. 9].



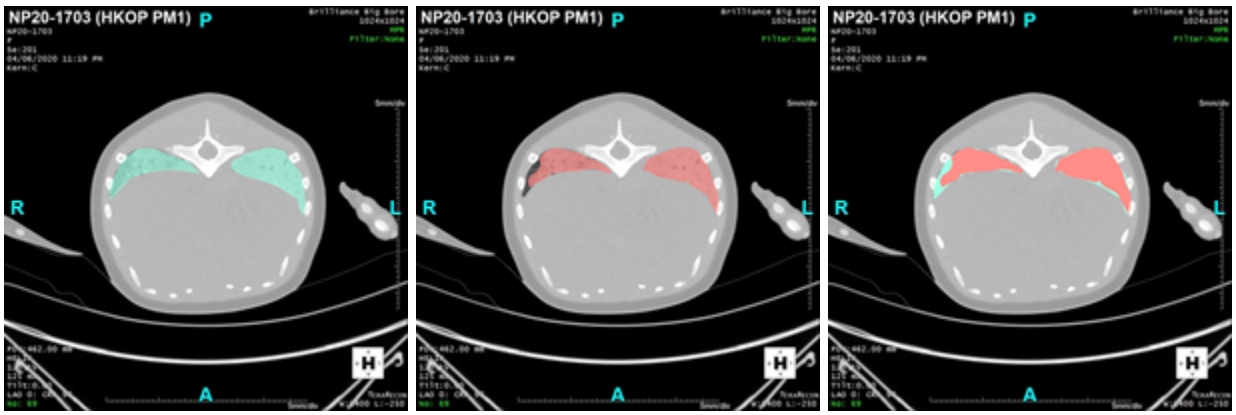
[Fig. 9] Left: original CT image. Right: label mask image

### 6.5.2. Change in Labeling Procedure

Of course, this also means a consequent change in the labeling procedure beforehand. Before, I used to only need to mark the box(es) on each image containing the lung (a text file containing the coordinates of each rectangle is generated for each image). Now, I need to trace the edge of the lung, mark points along the way (which are then saved in a JSON file), then convert the JSON file into mask images as the ground truth label. It is quite tedious work, and I am more than grateful that the colleagues at SKLMP are willing to lend a helping hand. Ms. Maria Robles, a Ph.D. student from the marine lab has been helping me with the majority of the labeling work.

### 6.5.3. Change in Expected Result

Originally, it was planned that the eventual result of the model would be to classify a lung as normal or abnormal, or perhaps into a number of categories (mild/moderate/severe) since it is only a matter of granularity. I thus initially planned to take the lung ROI images from the lung detection stage and perform image classification on the images. However, Dr. Brian Kot later suggested that he expected the model to be able to determine the severity of the sickness by presenting a percentage value, indicating the portion of the lung infected, instead of merely a binary/multi-class classification. Therefore, I have changed my approach in order to quantify the sickness level. First, as aforementioned, I have an image segmentation model that detects the lung area. Then, I plan to train another separate identifier for the pathologically infected area (e.g. the parts with lung consolidation). By overlapping the diseased area with the whole lung area and calculating the percentage of unhealthy lung pixels, the level of sickness can be quantified and represented by a value [Fig. 10]. And this is only for one image slice; by aggregating the percentage result of the entire dolphin/lung(s), we can have a general idea of the overall healthiness level of the dolphin.




[Fig. 10] Left: lung region prediction. Middle: unhealthy/pathological area prediction. Right: the unhealthy region.

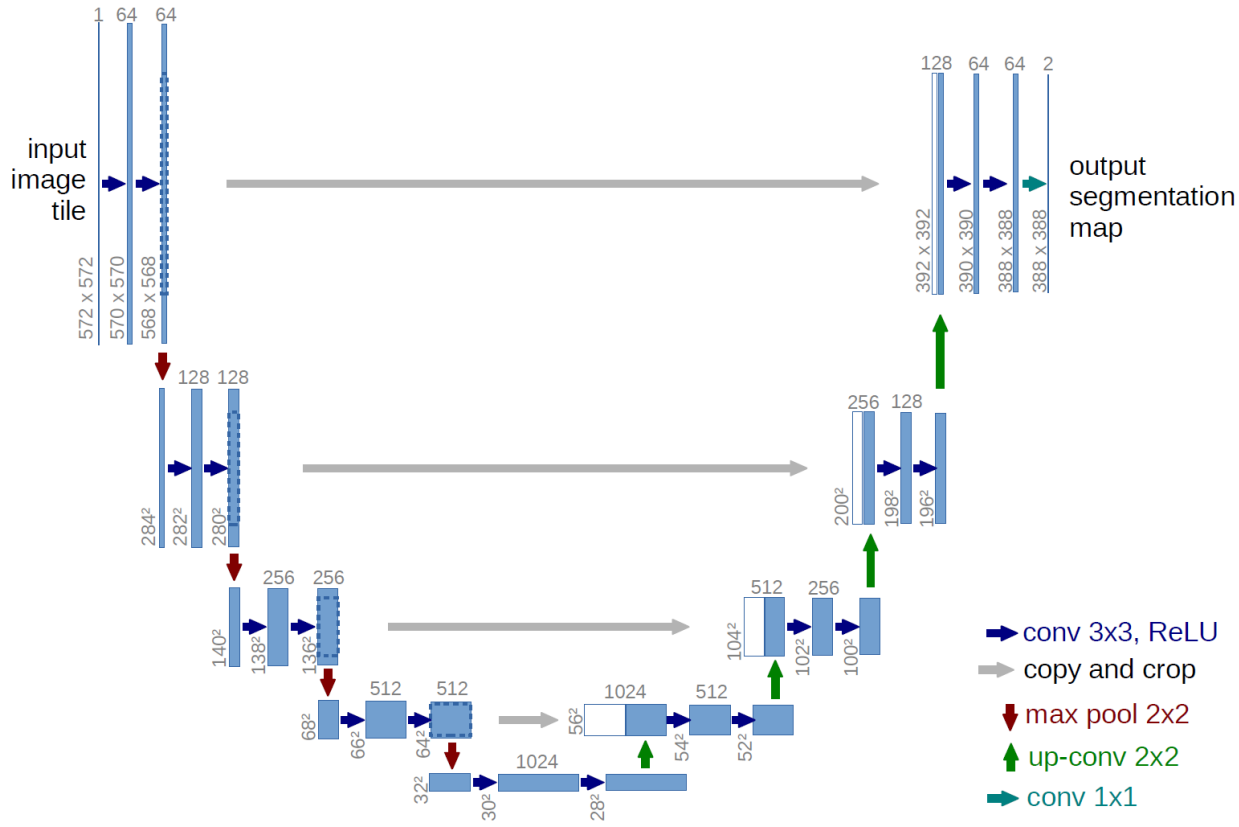
## 6.6. U-Net Image Segmentation & Detailed Implementation

### 6.6.1. Brief Introduction & Relevant Literature Review

Just like YOLO, the U-net is a relatively new network, with the original paper *U-Net: Convolutional Networks for Biomedical Image Segmentation (Olaf Ronneberger et. al)* coming out in 2015. However, it has soon established a reputation in image segmentation and the paper has also become one of the most influential papers in that field. Prior to the emergence of the U-net, there was not a good neural network approach that specializes in the efficient localization of objects. Large, fast convolutional networks focusing on image classification have been developed; however, few catered to the need for image segmentation/object detection where the ability to localize the object is required (in other words, classification for each pixel, so to speak). This is especially needed in medical image analysis, which is also what prompted Olaf Ronneberger et. al to develop U-net, as shown by its inclusion in the paper title. Some networks have been developed to resolve the need; Ciresan et al. developed a network using the sliding-window technique to make classification predictions for each pixel by providing a local region (called “patch”) adjacent to said pixel. While this does achieve localization, there are some major setbacks. Firstly, the speed is less than optimal due to the need to run the network independently for each pixel (and its patch around it), causing a lot of repetitiveness and redundancy when pixels are run repeatedly. Also, there is the trade-off between patch size and accuracy. Larger patches require more max-pooling layers (downsampling), reducing the accuracy; while smaller patches offer insufficient context around the pixel, which is also not ideal.

Olaf Ronneberger et. al proposed a more elegant, streamlined approach (Fig. 11). The first part of the network is the “contracting path”, where two 3x3 valid convolution layers (the blue arrows in Fig. 11) and a 2x2 max-pooling layer with a stride of 2 (the red downward arrows) are alternated, reducing the tensor size. However, unlike standard neural network models, where the usual structure consists of merely the contracting path, U-net adds an “expansive” path after that, which does the opposite (upsampling convolutional layers that increase the output size (the green upward arrows)). The outputs from the previous contracting path are then concatenated to the

corresponding counterpart in the expansive path forming the so-called “skip connections” (the gray arrows from left to right). Finally, a single 1x1 convolution layer is used to map the second-to-last output to the desired number of classes (in the paper the number is 2, but in our case, it would be 1, since we only have one class for our model). The contracting path helps determine what object it is (the “what”) (as is the function of traditional CNN structures with only the contracting part), and the expansive part, in conjunction with the concatenation of skip connections, helps localize the object within the image (the “where”).




[Fig. 11] The structure of U-Net

## 6.6.2. Advantages of U-net

As mentioned, it not only has the ability to classify but also accurately localize the object’s position within the image. Also, compared to the approach of Ciresan et al., it is a lot quicker and more memory/time efficient. Lastly, and most importantly, it works extremely well on small datasets, which is crucial for medical image-related analysis, since one of the most common bottlenecks of these tasks is the acquisition of sufficient

medical images, as they are often hard to access and have privacy concerns. This is also why the original paper used U-Net for cell segmentation in medical images.

### 6.6.3. Adaptation & Customization For My Project

In order to cater to my needs in this project, some modifications to the original paper are made, as listed below. They will also be demonstrated in detail in a later section.

#### 6.6.3.1. Adjustment in Number of Classes & Output Channels

Firstly, in the original implementation, the number of final output channels is two, corresponding to the number of classes they had. However, in this project, it is going to be set to one, since I am doing binary classification for both lung detection and pathology detection. That is why in my code implementation, the `out_channel` attribute of my `UNET` class is set to 1 by default, as will be shown later.

#### 6.6.3.2. Use of “same convolution” instead of “valid convolution”

In the original implementation, for the 3x3 convolutional layers, they used “valid convolution”, meaning the output size will shrink by 2 after each convolution. Therefore, even after applying the up-sampling, the right hand side of the U-shape will not be of the same size as its counterpart on the left hand side (e.g. 280 → 200, 136 → 104). Because of that, the eventual output size (388*388) is also not the same as the initial input size (512*512). That is why when concatenating the skip connection, cropping is needed. I figured it would be a bit troublesome to implement, so for the sake of simplicity, I decided to use “same convolution” in my project instead, where the input image is additionally padded before applying the convolution to ensure that the output has the same size as the input. The code implementation is as so (3*3 conv with stride 1 and same convolution (of padding 1)):

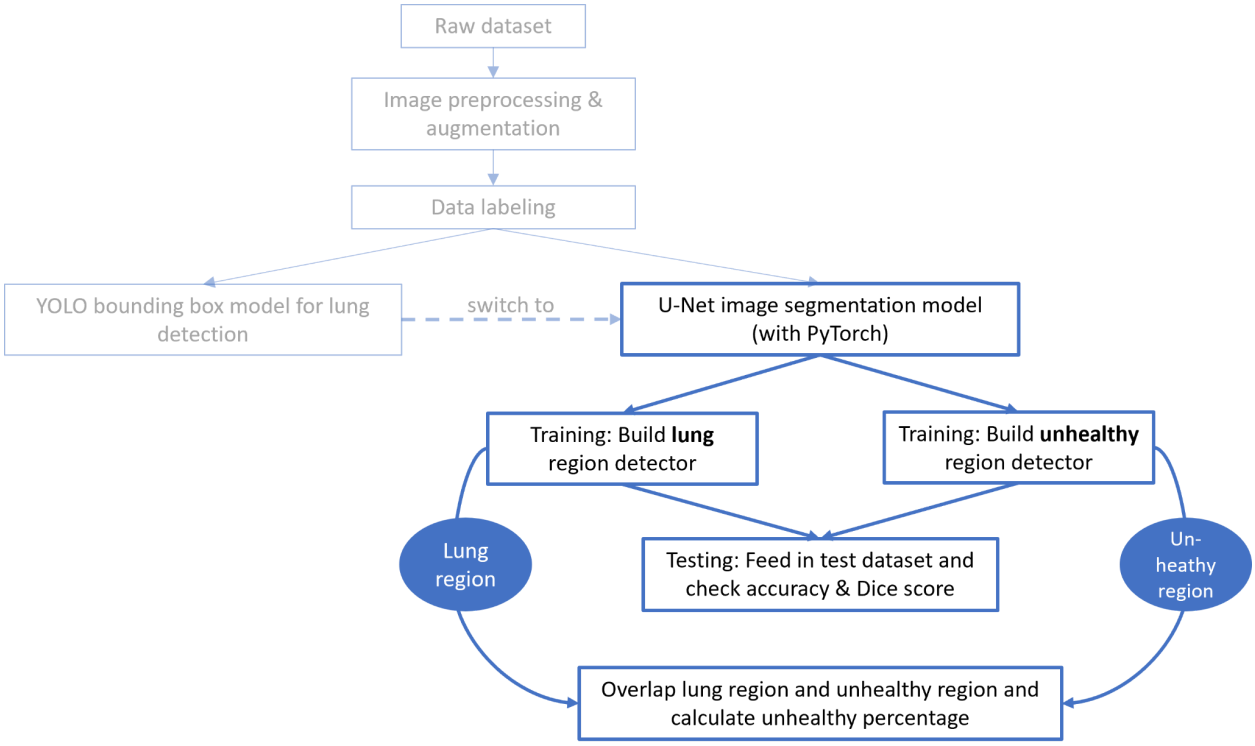
```
torch.nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
```

#### 6.6.3.3. Change in Activation & Loss function

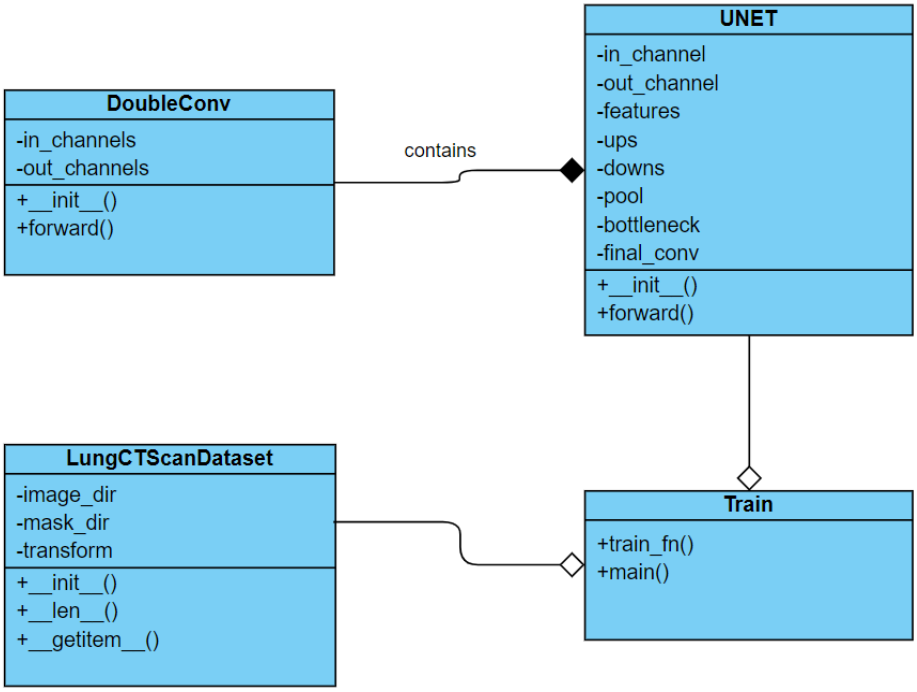
In the original paper, they use a **softmax** activation function with **cross-entropy** loss, since they have more than one class. In my project, as I plan to train two networks that determine whether each pixel is lung/non-lung and healthy/unhealthy respectively, I will use the **sigmoid** function with a **binary cross-entropy** loss function instead.

### 6.6.4. Implementation & Components Interaction

In this project, I will use PyTorch to build the U-Net architecture from scratch. The key components in my project include a Dataset class, with which I load the dataset into the project and perform data augmentation; a DoubleConv class, which is the double 3x3 convolutional layer repeatedly used in the U-Net architecture; the actual U-Net model class that I write from scratch containing a contracting (down) path and an expansive (up) path as per the paper, a number of utility functions (e.g. save/load model checkpoints, check the accuracy, get data loaders, …), and a training function that carries out the training process according to the hyperparameters set. A classifier for the whole lung region as well as the unhealthy region will be trained, which will then be used to calculate the percentage of the unhealthy region within the entire lung. Later, in the testing stage, the trained model is tested on the test set, and its accuracy and Dice score are to be calculated. Finally, the healthiness level can be quantified by a percentage value, which is calculated by overlaying the unhealthy region prediction on top of the lung region prediction and counting the proportion of the unhealthy pixels out of the whole lung.




System diagram (with only the image segmentation part being highlighted)




The class diagram of the image segmentation task

#### 6.6.4.1. Dataset configuration: `dataset.py`

Here, I created a `LungCtScanDataset` that inherits the `Dataset` class in PyTorch. The constructor `__init__()` takes three parameters: `image_dir`, the actual lung images' directory; `mask_dir`, the mask label ground truth (b/w images), and `transform`, which stores the image transformation/augmentation tool. The function `__len__()` returns the size of the dataset, and the function `__getitem__()` gets the raw image and its corresponding mask label (same name as the image but with suffix "_mask"), applies image augmentation (line 23), and return them both.

```
1. import os
2. from PIL import Image
3. from torch.utils.data import Dataset
4. import numpy as np
5.
6. class LungCtScanDataset(Dataset):
7.     def __init__(self, image_dir, mask_dir, transform=None):
8.         self.image_dir = image_dir
9.         self.mask_dir = mask_dir
10.            self.transform = transform
11.            self.images = os.listdir(image_dir)
12.
13.    def __len__(self):
14.        return len(self.images)
15.
16.    def __getitem__(self, index):
17.        img_path = os.path.join(self.image_dir, self.images[index])
18.        mask_path = os.path.join(self.mask_dir, self.images[index][:-4]
   + '_mask.png')
19.        image = np.array(Image.open(img_path).convert("RGB"))
20.        mask = np.array(Image.open(mask_path).convert("L"),
   dtype=np.float32)
21.        mask[mask == 255.0] = 1.0
```

```
22.
23.         if self.transform is not None:
24.             augmentations = self.transform(image=image, mask=mask)
25.             image = augmentations["image"]
26.             mask = augmentations["mask"]
27.
28.         return image, mask
```

#### 6.6.4.2. Building the U-Net model from scratch: `model.py`

Next, I create the components needed for the U-net model. There are two classes in this file, DoubleConv, which represents the double convolution layers often seen in the U-net structure, and the UNET class itself.

##### 6.6.4.2.1. DoubleConv

It inherits the `nn.Module` class from PyTorch, which is the base class and fundamental building block for all neural network modules. It consists of two `Conv2d` layers, each of which is followed by batch normalization and ReLU activation. It takes `in_channels` and `out_channels` as its two parameters. The first `Conv2d` layer has the same `in_channels` and `out_channels` as the entire `DoubleConv` block as it needs to change its number of channels (line 5), while the second `Conv2d` layer just maintains the same channel count (line 8). As aforementioned, I am using “same convolution”, meaning a padding of 1 is needed (with kernel_size and stride being 3 and 1 respectively), as opposed to no padding, which is the original paper’s approach.

```
1. class DoubleConv(nn.Module):
2.     def __init__(self, in_channels, out_channels):
3.         super(DoubleConv, self).__init__()
4.         self.conv = nn.Sequential(
5.             nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1,
   padding=1, bias=False),
```

```
6.            nn.BatchNorm2d(out_channels),
7.            nn.ReLU(inplace=True),
8.            nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False),
9.            nn.BatchNorm2d(out_channels),
10.              nn.ReLU(inplace=True),
11.           )
12.
13.   def forward(self, x):
14.       return self.conv(x)U-Net
```

Next up, is the UNET class itself. Similarly, it inherits the nn.Module class. I will break down its structure with respect to its functions.

- **`def __init__():`** In the constructor, it takes as attributes an in_channel, out_channel, and a features list storing the feature/channel count of each level.
    - `self.downs` and `self.ups` are two `ModuleLists` that represent the contracting and expansive paths respectively, each consisting of a list of `DoubleConv` objects (added by the two for loops, line 11 & 16).
    - `self.pool` is the max-pooling layer used for downsampling (line 8).
    - `self.bottleneck` is the `DoubleConv` layer on the very bottom of the U shape (line 24).
    - `self.final_conv` of the final convolutional layer that shows the final result.
- **`def forward():`** The forward function takes a parameter x and is called to run x through the layer/network. First, it iterates through self.downs, runs `forward()` on each down layer -- followed by a max-pooling layer -- and appends each layer to the skip_connections list to be used later (line 30-33). Then, it runs through the `self.bottleneck` layer (line 35). After that, it loops through self.ups and concatenates each up layer with its

corresponding skip connection layer (line 38-46). Lastly, the function returns the result after running x through the `final_conv` layer.

```
1. class UNET(nn.Module):
2.     def __init__(
3.             self, in_channels=3, out_channels=1, features=[64, 128, 256,
   512],
4.     ):
5.         super(UNET, self).__init__()
6.         self.ups = nn.ModuleList()
7.         self.downs = nn.ModuleList()
8.         self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
9.
10.            # Down part of UNET
11.            for feature in features:
12.                self.downs.append(DoubleConv(in_channels, feature))
13.                in_channels = feature
14.
15.            # Up part of UNET
16.            for feature in reversed(features):
17.                self.ups.append(
18.                    nn.ConvTranspose2d(
19.                        feature*2, feature, kernel_size=2, stride=2,
20.                    )
21.                )
22.                self.ups.append(DoubleConv(feature*2, feature))
23.
24.            self.bottleneck = DoubleConv(features[-1], features[-1]*2)  #
   (512, 512*2)
25.            self.final_conv = nn.Conv2d(features[0], out_channels,
   kernel_size=1)
26.
27.    def forward(self, x):
28.        skip_connections = []
```

```
29.
30.         for down in self.downs:
31.             x = down(x)
32.             skip_connections.append(x)
33.             x = self.pool(x)
34.
35.         x = self.bottleneck(x)
36.         skip_connections = skip_connections[::-1]
37.
38.         for idx in range(0, len(self.ups), 2):
39.             x = self.ups[idx](x)
40.             skip_connection = skip_connections[idx//2]
41.
42.             if x.shape != skip_connection.shape:
43.                 x = TF.resize(x, size=skip_connection.shape[2:])
44.
45.             concat_skip = torch.cat((skip_connection, x), dim=1)
46.             x = self.ups[idx+1](concat_skip)
47.
48.         return self.final_conv(x)
```

#### 6.6.4.3. Training Process: **`train.py`**

Firstly, I set a number of hyperparameters related to training my model. The list is as follows:

| Parameter | Value |
|---|---|
| Batch size | 16 |
| Learning rate | $10^{-4}$ |
| # of epochs | 15 |

| | |
|---|---|
| Image height | 512 |
| Image width | 512 |
| NUM_WORKERS (for DataLoader) | 2 |
| PIN_MEMORY (for DataLoader) | True |

I then define a function `train_fn()`. It is the training process of one epoch and will thus be called for `NUM_EPOCH` times. For parameters, it has a `loader` that loads the data, a (UNET) `model`, an `optimizer` (e.g. Adam, stochastic gradient descent, RMSprop, ...), a `loss_fn` (in our case a binary cross-entropy loss function), and a `scaler` used to speed up the training process and lower memory required with the usage of float16 precision. It contains a for loop that reads in a batch of data for each iteration (will loop NUM_IMAGES/BATCH_SIZE times). In the forward step, `predictions` stores the prediction made by the model (line 10), and the loss is calculated with the `loss_fn` function passed into the function. In the backpropagation step, `scaler.scale(loss).backward()` (or just loss.backward() if scaler is not used) calculates the gradient of each parameter. Note that since calling `backward()` multiple times accumulates the gradient (by addition) for each parameter, it is necessary to first call `optimizer.zero_grad()` to zero the gradients in the preceding line (line 14). Then, the optimizer, which takes the parameters to update and the learning rate on initialization, performs the updates with its `step()` function (line 16 & 17).

```
1. def train_fn(loader, model, optimizer, loss_fn, scaler):
2.     loop = tqdm(loader)
3.
4.     for batch_idx, (data, targets) in enumerate(loop):
5.         data = data.to(device=DEVICE)
6.         targets = targets.float().unsqueeze(1).to(device=DEVICE)
7.
8.         # forward
```

```
        with torch.cuda.amp.autocast():
            predictions = model(data)
            loss = loss_fn(predictions, targets)

        # backward
        optimizer.zero_grad()
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        # update tqdm loop
        loop.set_postfix(loss=loss.item())
```

Then, in the main function, we first initialize the parameters needed for the training. Firstly, the transform parameter that is to be passed into the `LungCtScanDataset` object upon creation, as mentioned before. In my project, I elect to use the “albumentations” library for image transformation/augmentation. It is a data augmentation library widely used for computer vision tasks. It easily enables image resizing, rotation, horizontal/vertical flip, RGB shifting, blurring, brightness/contrast change, etc. The transformation for training is defined as so (that of validation is defined similarly):

```
import albumentations as A
from albumentations.pytorch import ToTensorV2

train_transform = A.Compose(
   [
       A.Resize(height=IMAGE_HEIGHT, width=IMAGE_WIDTH),
       A.Rotate(limit=35, p=1.0),
       A.HorizontalFlip(p=0.5),
       A.VerticalFlip(p=0.1),
       A.Normalize(
```

```
            mean=[0.0, 0.0, 0.0],
            std=[1.0, 1.0, 1.0],
            max_pixel_value=255.0,
        ),
        ToTensorV2(),
    ],
)
```

Other parameters to create include the model, the loss function, the optimizer, and the scaler.

```
model = UNET(in_channels=3, out_channels=1).to(DEVICE)
loss_fn = nn.BCEWithLogitsLoss()  # binary cross-entropy loss function
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)  # the Adam
optimizer
scaler = torch.cuda.amp.GradScaler()
```

After creating all the parameters required, they are passed into a utility function `get_loaders` (defined in another file utils.py) to create the training and validation loader.

```
train_loader, val_loader = get_loaders(
    TRAIN_IMG_DIR,
    TRAIN_MASK_DIR,
    VAL_IMG_DIR,
    VAL_MASK_DIR,
    BATCH_SIZE,
    train_transform,
    val_transforms,
    NUM_WORKERS,
    PIN_MEMORY,
```

```
)
```

The `get_loaders` function first creates a `LungCtScanDataset` for the train dataset by passing in the image directory, mask label directory, and transform object. It then creates a train loader object (of type DataLoader from PyTorch) with the training dataset. It does the same for the validation part before returning both `train_loader` and `val_loader`.

```
def get_loaders(
    train_dir,
    train_maskdir,
    val_dir,
    val_maskdir,
    batch_size,
    train_transform,
    val_transform,
    num_workers=4,
    pin_memory=True,
):
    train_ds = LungCtScanDataset(
        image_dir=train_dir,
        mask_dir=train_maskdir,
        transform=train_transform,
    )

    train_loader = DataLoader(
        train_ds,
        batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=pin_memory,
        shuffle=True,
    )
```

```
    val_ds = LungCtScanDataset(
        image_dir=val_dir,
        mask_dir=val_maskdir,
        transform=val_transform,
    )

    val_loader = DataLoader(
        val_ds,
        batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=pin_memory,
        shuffle=False,
    )

    return train_loader, val_loader
```

Lastly, in the main function, a for loop runs for NUM_EPOCHS iterations, during each of which `train_fn` is called. The model is also saved after each epoch, and the accuracy is checked through another util function, `check_accuracy`.

```
for epoch in range(NUM_EPOCHS):
    train_fn(train_loader, model, optimizer, loss_fn, scaler)

    # save model
    checkpoint = {
        "state_dict": model.state_dict(),
        "optimizer":optimizer.state_dict(),
    }
    save_checkpoint(checkpoint, filename="my_checkpoint_" + CLASS + "_" + str(IMAGE_WIDTH) + ".pth.tar")

```

```
    # check accuracy
    check_accuracy(val_loader, model, device=DEVICE)
```

The `check_accuracy` function takes the (validation) loader, the (UNET) model, and the device type as parameters. It first calls `model.eval()` to enter evaluation mode (line 5). In PyTorch, it is necessary to call `model.eval()` before evaluation and `model.train()` to switch to training mode afterward (as in line 23) because the behavior of certain layers (e.g. BatchNorm layers, Dropout layers) can differ during training/evaluation time. Then, the prediction pred is computed by calling `torch.sigmoid(model(x))` and compared against `y` (ground truth). The accuracy and dice score can then be calculated and printed out, before `model.train()` is called to return to training mode.

```
def check_accuracy(loader, model, device="cuda"):
    num_correct = 0
    num_pixels = 0
    dice_numerator = 0
    dice_denominator = 0

    model.eval()
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device)
            y = y.to(device).unsqueeze(1)
            preds = torch.sigmoid(model(x))
            preds = (preds > 0.5).float()
            num_correct += (preds == y).sum()
            num_pixels += torch.numel(preds)
            dice_numerator += (2 * (preds * y).sum())
            dice_denominator += (
                    (preds == 1).sum() + (y == 1).sum()
            )

```

```
    print(
        f"Got {num_correct}/{num_pixels} with acc
{num_correct/num_pixels*100:.2f}"
       )
    print(f"Dice score: {dice_numerator / dice_denominator}")
    model.train()
```

## 6.7. Potential Constraints and Challenges

### 6.7.1. Scarcity of Dataset

One of the most prominent problems with medical image-related research is the scarcity of the dataset. The creation of a machine learning network typically involves a large amount of training data, and it is often not practical nor possible to obtain datasets of such size when it comes to medical images. The current source of my dataset is directly obtained from the CityU Veterinary Medical Center, which conducts the scans on deceased dolphins by themselves. It is to be expected that there might not be a large enough raw dataset readily available for training. Also, the sharing and obtaining of medical image resources often raise the question of potential invasion of privacy, making the hunt for available datasets online even tougher.

I initially suffered from this problem due to not having enough CT scan data at the early stage of my project. However, the kind colleagues at SKLMP have since provided me with 10 dolphins, which are sufficient for the task in hand.

### 6.7.2. Memory and Time Requirement for Training

The training process, especially that of U-Net network training, requires a lot of computer memory and time. So much so that I find it borderline impossible to train with all the training images I have with the original image size (1024*1024). I find myself having to settle for resizing the images to a smaller size to speed up the training process and reduce the memory space required (otherwise my computer hardware would not be able to run). I have also tried making use of

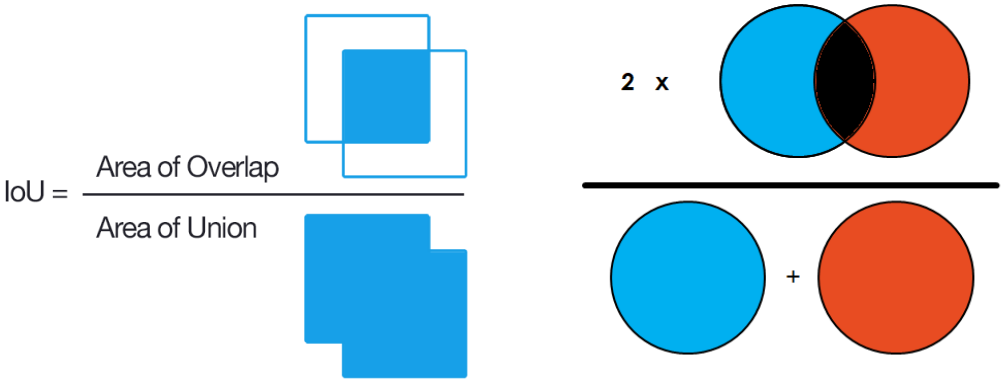
GradScaler, a gradient scaling tool provided by PyTorch, to lower the precision from the default float32 to float16 when calculating the gradient descent. It did help to a certain extent; I am able to train on images of sizes 256*256 and 512*512, but it still struggles to process images of sizes larger than a certain threshold. I will further try to resolve the problem by attempting to optimize my training approach or reducing the batch size.

# 7. TESTING, IMPROVEMENTS, & RESULTS

## 7.1. Testing of YOLO Bounding Box Model

After obtaining a predicted RoI via the YOLO model, we can compare it against the ground truth and evaluate how close the model's prediction is to the actual wanted region. There are two major metrics used when it comes to object detection, IoU and Dice Coefficient:

- **Intersection of Union (IoU)** [Fig. 12]
  The Intersection of Union is calculated by dividing the "intersection of the prediction and the ground truth" with their "union", also denoted as **TP/(FP+TP+FN)*** or Overlap/Union. It was used in the YOLO to calculate the confidence score of each bounding box [10].
  (**TP = true positive, FN = false negative, FP = false positive, etc.)*
- **Dice Coefficient** [Fig. 13]
  Short for the Sørensen–Dice similarity coefficient, the Dice score is another very popular evaluation metric. It takes two times the overlap of prediction and ground truth and divides it by the number of pixels/voxels of both combined, or denoted by the following formula: 2*Overlap/(Prediction+Ground Truth) or **2*TP/(2*TP+FP+FN)**. It is commonly used in the field of biomedical imaging to evaluate the performance of segmentation.




[Fig. 12 & 13] Visualization of the respective meaning of IoU and Dice coefficient

I calculated both metrics for my YOLO lung detection model and got an IoU of 0.89 and a Dice score of 0.825, which are satisfactory but not exactly top-notch. This is why I later switched to U-net image segmentation and achieved far greater results and better precision.

In terms of the value of the metrics, both are very common in object localization and image segmentation and are very similar to each other concept-wise. As a matter of fact, they are positively correlated, meaning if classifier A has a higher Dice score than classifier B, it is bound to perform better when evaluated by IoU as well. There are some very subtle minor differences, nonetheless. When it comes to averaging the scores over a number of inferences to evaluate the overall performance of each classifier instead of how they do on individual cases, the IoU metric penalizes single cases of bad classification more than Dice quantitatively. In other words, if classifier does marginally better than classifier B in most instances but performs substantially worse in a couple of cases, it is going to be ranked lower than classifier B by IoU, which has a magnifying effect on bad/wrong predictions, as opposed to the Dice metric, which would put it higher than classifier B.

## 7.2. Testing of U-Net Image Segmentation

The testing procedure is somewhat similar to that of validation, except this time it is not run after each epoch finishes training. Also, the model will be loaded from a saved file instead of starting from scratch. I have a total of 13 dolphin CT scans, 3 of which are used as the test set (the other ten are used for training), which add up to around 260 images in total).

I first create the variables needed to run the model. I create a `LungCtScanDataset` object, `test_ds`, then use it to create the test `DataLoader`. I then create a `UNET` model of `in_channel` 3 and `out_channel` 1, similar to before. Note that here we load the model from a pre-existing model that we just trained and want to test the accuracy of.

```
1. test_ds = LungCtScanDataset(
```

```
    image_dir=val_dir,
    mask_dir=val_maskdir,
    transform=val_transform,
)

test_loader = DataLoader(
    test_ds,
    batch_size=batch_size,
        num_workers=num_workers,
        pin_memory=pin_memory,
        shuffle=False,
    )
    # load model
    checkpoint = torch.load("my_checkpoint_" + CLASS + "_" +
str(IMAGE_WIDTH) + ".pth.tar")
    model.load_state_dict(checkpoint["state_dict"])

    num_correct = 0
    num_pixels = 0
    dice_numerator = 0
    dice_denominator = 0
```

After that, I call `model.eval()` to switch the model to evaluation mode, as explained before. Then, a for loop iterates through the `test_loader`. In each iteration, `x` (input) and `y` (ground truth) are extracted; the input `x` is fed into the model and the prediction is stored in preds. Then comes the performance assessment process. I mainly use two metrics to determine the model's performance: accuracy and Dice score. Below I will elaborate on the formula and difference between the two.

- **Accuracy:** ***(TP+TN)/(TP+TN+FP+FN) = True/All***

  Accuracy is calculated by taking the number of correctly classified cases (in our case, pixels) and dividing it by the total number of cases. While this is a straightforward and commonly used method to judge the model, it doesn't have

some problems. If the majority of the ground truth cases are false (which is the case here since the lung area only makes up a rather small part of the entire CT image), the model could achieve relatively high accuracy by simply guessing "false" for all cases. This issue is called "class imbalance", and it can lead to misleadingly high accuracy when classes are very imbalanced, meaning a class dominates a large proportion of the image.

- **Dice score:** ***2*TP/(2*TP+FP+FN) = 2*Overlap/(Prediction+Ground Truth)***
  Dice score is two times the area of overlap divided by the total number of pixels in both images. It is a better judging factor of a model's performance in that it takes into account the size of the target area (instead of just blindly using the entire image size as the denominator), eliminating the class imbalance problem.

**Clarification**:

Note that, the evaluation (of both the lung classifier and the pathology classifier) is calculated over the whole image, instead of just the lung. The reason the "universal set" is not just the lung pixels but the entire rectangular image is that, there are images outside of the lung that may be misclassified as the lung, which should also be taken into consideration when calculating the accuracy/Dice score. Therefore, during the evaluation, the whole image is considered and not just the lung region. Take the calculation of the accuracy metric for example. For the lung classifier, it is all the correctly labeled pixels -- the lung pixels classified as lung and the non-lung background pixels classified as not lung -- divided by the number of all pixels in the image. The same goes for the accuracy of the pathology classifier. The correctly classified pixels include the unhealthy pixels labels as so, PLUS the healthy pixels as well as the background pixels labeled as not-unhealthy, as they both are not considered in the unhealthy region. The denominator, similar to the lung classifier's calculation, is all the pixels in the entire image.

```
model.eval()

with torch.no_grad():
    for idx, (x, y) in enumerate(test_loader):
```

```
        x = x.to(DEVICE)
        y = y.to(DEVICE).unsqueeze(1)
        preds = torch.sigmoid(model(x))
        preds = (preds > 0.5).float()
        num_correct += (preds == y).sum()
           num_pixels += torch.numel(preds)
           dice_numerator += (2 * (preds * y).sum())
           dice_denominator += (
               (preds == 1).sum() + (y == 1).sum()
           )
           torchvision.utils.save_image(
               y.unsqueeze(1),
               f"saved_images/{CLASS}/individual_images/{idx}.png"
           )
           torchvision.utils.save_image(
               preds,
               f"saved_images/{CLASS}/individual_images/pred_{idx}.png"
           )

    print(
       f"Got {num_correct}/{num_pixels} with acc {num_correct / num_pixels
* 100:.2f}"
    )
    print(f"Dice score: {dice_numerator / dice_denominator}")
```
Here are the testing results after running the program:

| Metric \ object | **Lung detection** | **Pathology detection** |
|---|---|---|
| **Accuracy** | 0.9886 | 0.9882 |
| **Dice score** | 0.9293 | 0.8804 |

As shown in the table, the result is very satisfactory. The accuracies of both classifiers are close to 100%, and the Dice scores are at around 0.94-0.95 as well. Lastly, I saved the prediction as images using `torchvision.utils.save_image` (line 19).

## 7.3. Improvements

In order to improve the model even further, I tried to tweak and adjust the hyperparameters and train the model over and over again to come up with a final, optimal model by comparing the various results. Note that since I had not gotten my entire dataset labeled by the time I was doing this step, I did not use the entire training set for this parameter tweaking process, but rather a subset of the eventual dataset, but I reckon the findings still stand. The relevant parameters include:

- **Number of epochs:**
  The number of epochs is one of the most important factors in training a model. Currently, I have found that the loss starts to increase and the training tends to overfit after 10-12 epochs with a batch size of 16, so a figure within that range seems to be optimal. I will continue to test it with different hyperparameters and find the sweet spot that achieves the highest accuracy without it overfitting.
- **Batch size:**
  Batch size is another crucial hyperparameter that controls the learning speed and the stability of the training process. In a gradient descent algorithm, error gradients are calculated by comparing the prediction to the ground truth and denote the magnitude and direction in which to optimize the model weights. They are calculated in each epoch. The more training examples used (aka bigger batch size), the more accurate and representing the error gradient is estimated; on the other hand, a smaller batch size will yield a less accurate and more noisy estimate. Nonetheless, these noisy updates can sometimes lead to faster learning and even a more robust model. The small size indicates faster training time and less memory needed, and the noisy updates offer a regularizing effect and lower generalization error. I am currently using a batch size of 16 (a rather

small figure), and I would like to further explore the trade-off between small and large batch sizes to optimize my network.

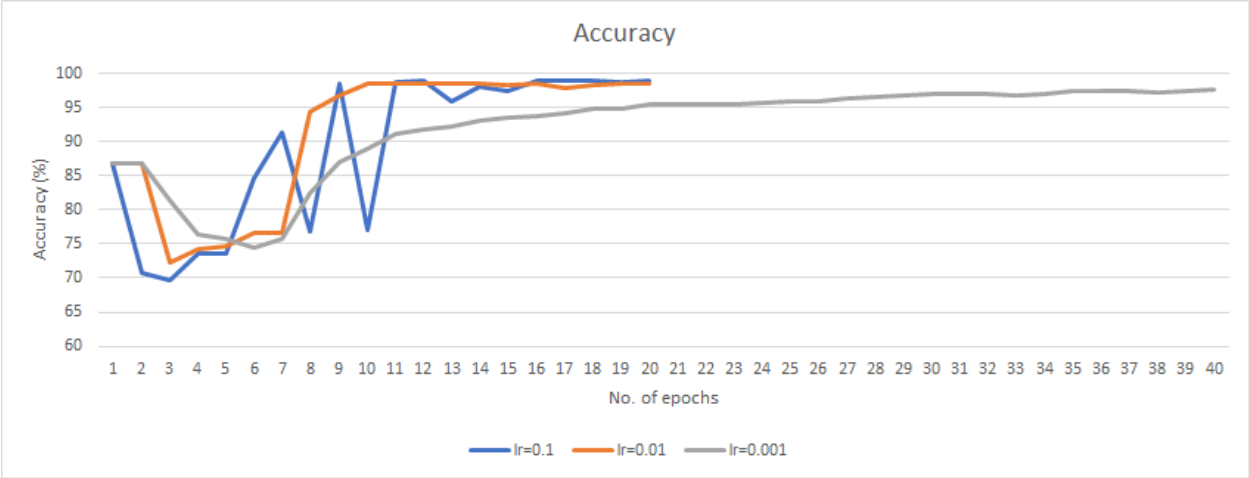
- **Optimizer & Learning Rate:**
  Currently, I am using the Adam optimizer, which is widely regarded as the best optimizer with its adaptive learning rate and ability to work with sparse data. It is suggested by some to first use Adam in any case, because it is more likely to yield good results without an advanced fine-tuning; then, one can explore other options such as Stochastic Gradient Descent (SGD) and RMSprop. I plan on trying SGD with different learning rates to see if it outperforms Adam.

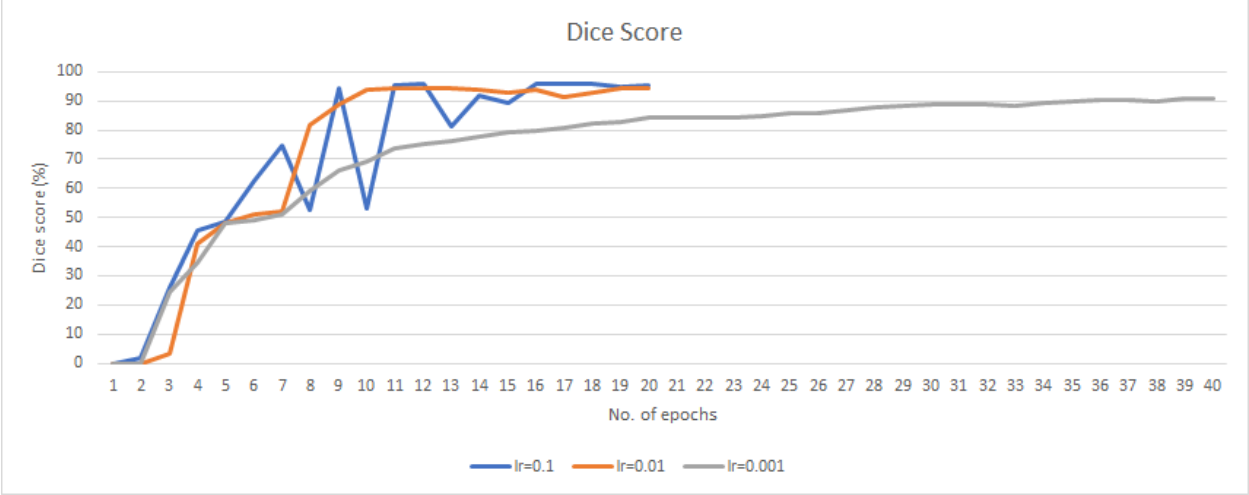
### 7.3.1. Learning Rate

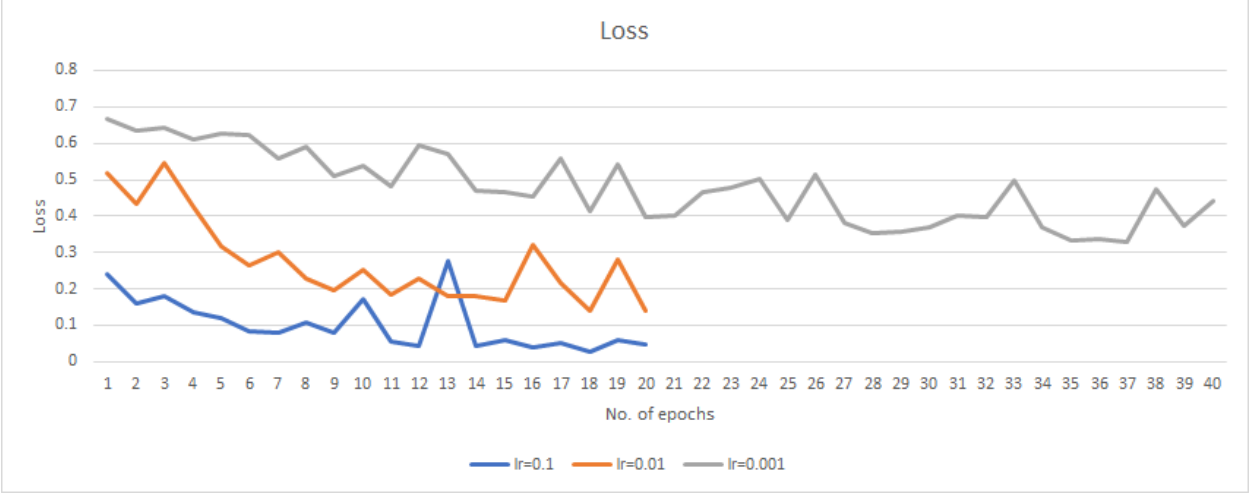
Firstly, I try to examine the impact of different learning rates on the model's performance. They control the progressiveness of the model's learning, or more specifically, they determine how big of a stride to take when updating the model after each iteration. Basically, a big learning rate means quicker learning, at the risk of unstable training (resulting in performance oscillation and thus weights diverging) and subpar results. Small learning rates, on the other hand, lead to a more globally optimal set of weights, but the training could be substantially slower. Worse case scenario, too low of a learning rate might result in getting stuck at a subpar solution. Therefore, it is important to analyze the trade-off between learning speed and performance and to find the most suitable learning rate value.

Since the learning rate and the optimizer used are closely related, I tested different learning rates under various optimizers in order to accurately capture the impact of learning rate changes. The learning rates I have tested are $10^{-1}$, $10^{-2}$, $10^{-3}$, and $10^{-4}$. The optimizers I have tried are Stochastic Gradient Descent (SGD), Adam, and RMSprop. The performance evaluations are done by analysing and plotting out the accuracy, Dice score, and loss.

#### 7.3.1.1. Different Learning Rates Using **SGD**



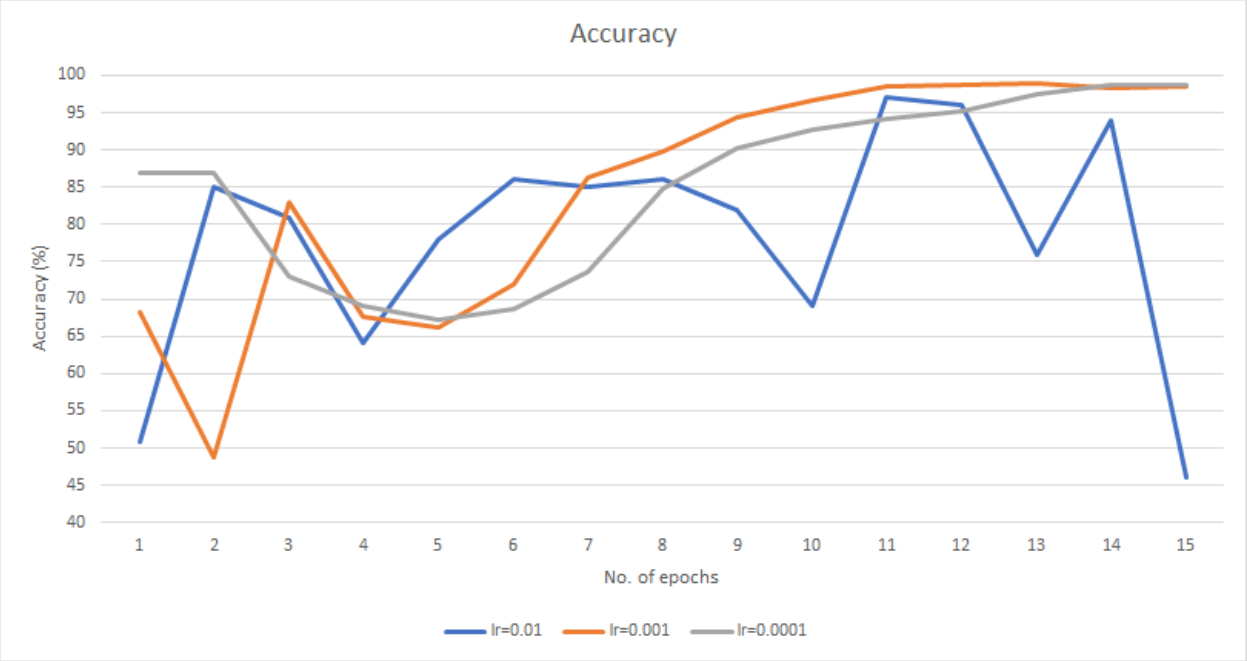



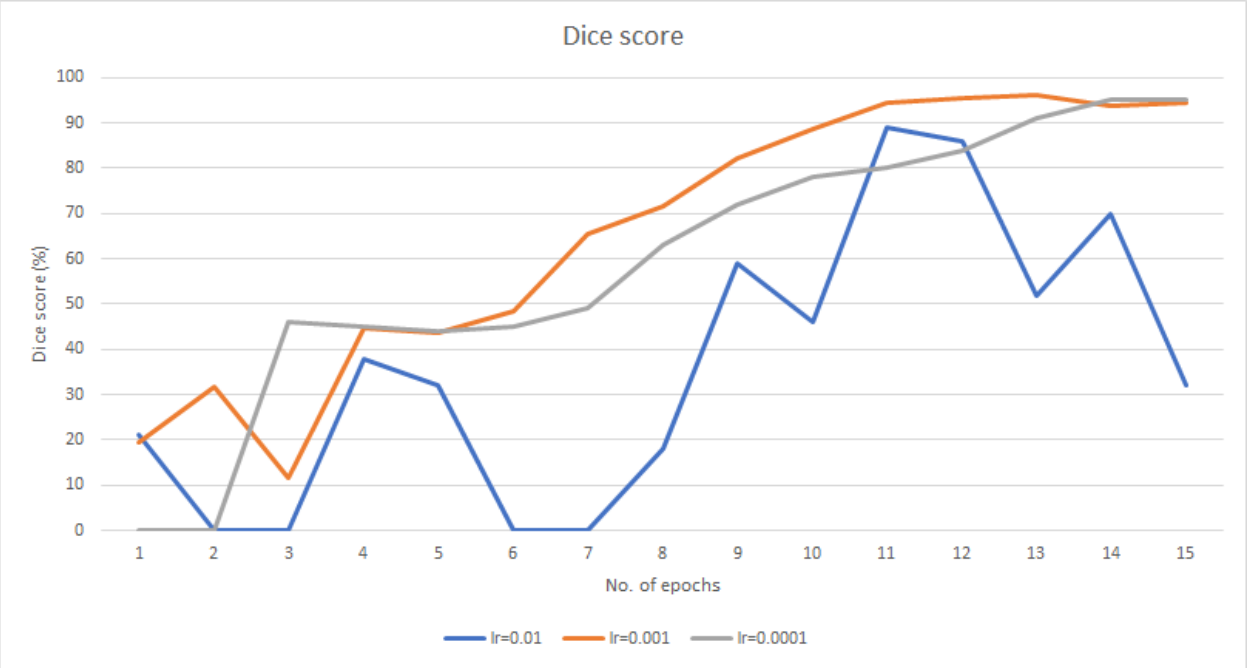



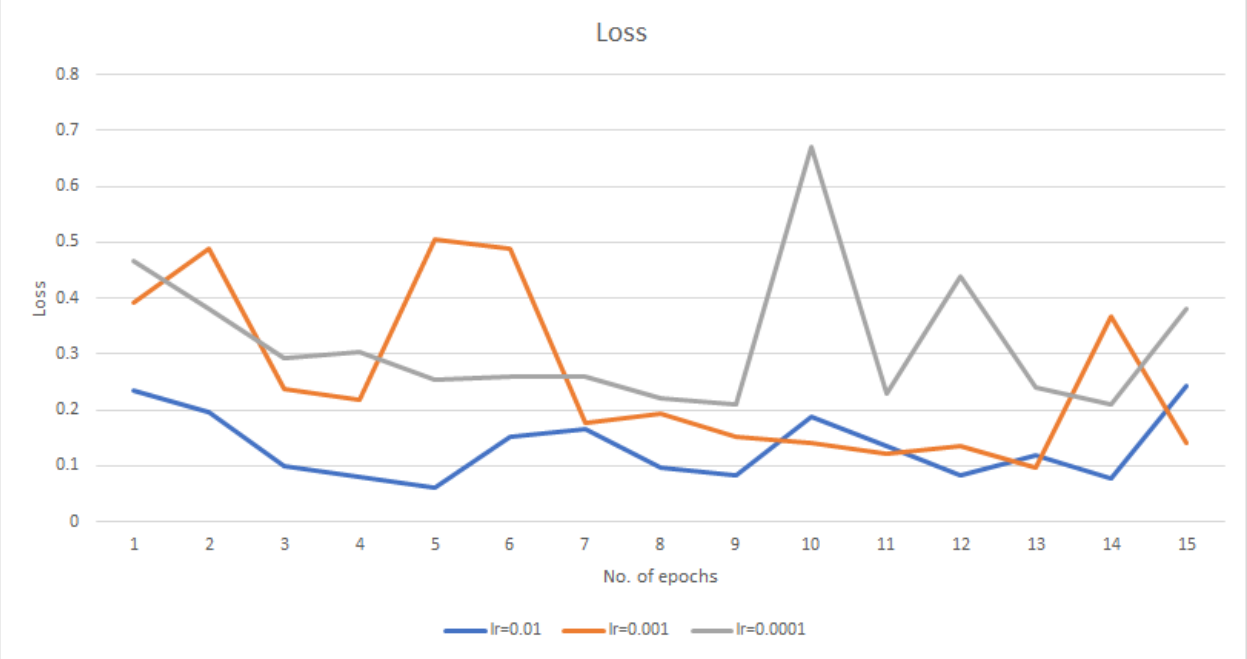

I compared the model performance using different learning rates with SGD. For learning rate = 0.1 and 0.01, I ran the training for 20 epochs, by the end of which both processes

have converged and maintained a fairly steady accuracy, Dice score, and loss. For learning rate = 0.001, however, I ran it for 40 epochs, as its accuracy and Dice score after 20 epochs were not high enough but still steadily rising. The result showed that both learning reat 0.1 and 0.01 were able to achieve very satisfactory accuracy and Dice score (at around 98.5% and 95% respectively), with learning rate 0.1 having a lower loss than 0.01. On the other hand, for learning rate 0.001, it is perhaps due to its too small of steps, that it not only took almost twice as long to converge, the final results were also not exactly satisfying, achieving a slightly lower accuracy of 97% and a Dice score of around 90%. The loss is also the highest.

#### 7.3.1.2. Different Learning Rates Using **Adam**



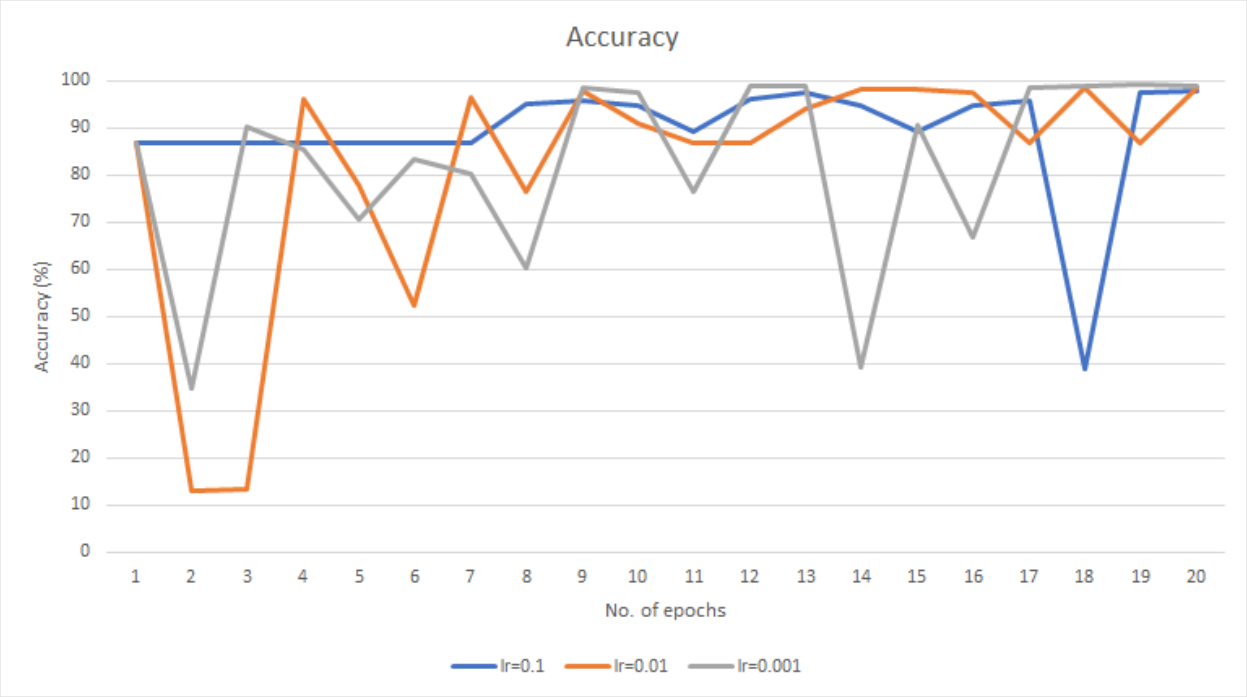


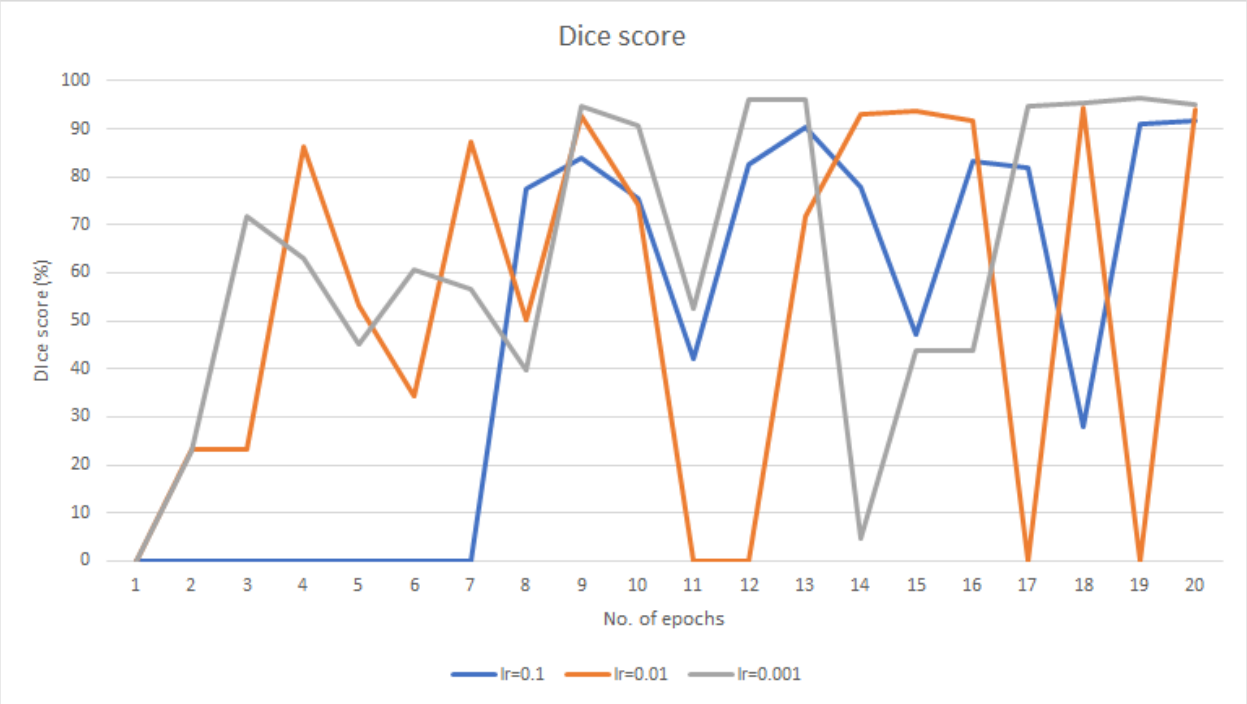



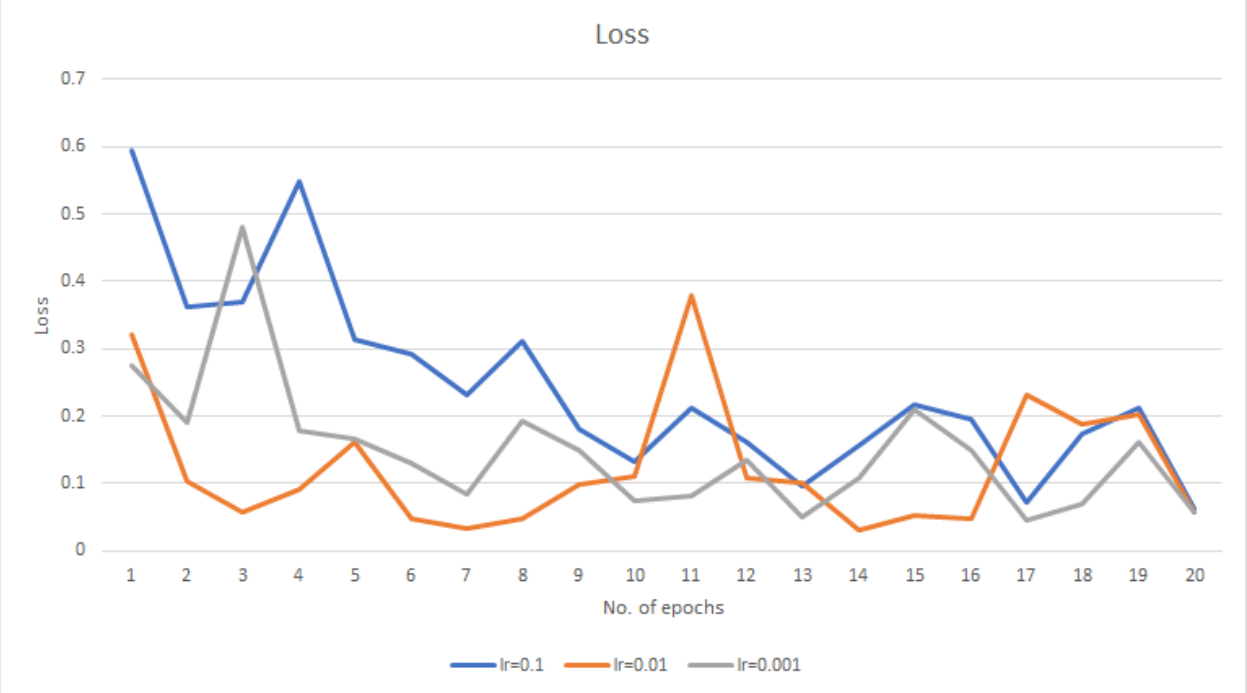

I also tried Adam, a state-of-the-art optimizer and many's go-to choice to start off the model training -- its adaptive learning rate method allows it to compute learning rates for each individual parameter, finding the most optimal value along the way. As a matter of fact, it was also the first optimizer I used to train my classifiers, coupled with a learning rate of $10^{-4}$. I also tested out rates like $10^{-2}$ and $10^{-3}$ and plotted the results. I took each rate and trained the lung classifier for 15 epochs with a batch size of 32. As shown in

the charts, both learning $10^{-3}$ and $10^{-4}$ work pretty well with Adam; their accuracy dipped slightly around 4-5 epochs in but then quickly rose to around 98%. The Dice score of both also rose steadily and converged at approx. 95%. The performance of learning rate $10^{-1}$, on the other hand, fluctuated rather drastically throughout the training process and did not perform very well. In terms of the loss, all three rates are rather similar and do not differ much.

#### 7.3.1.3. Different Learning Rates Using **RMSprop**



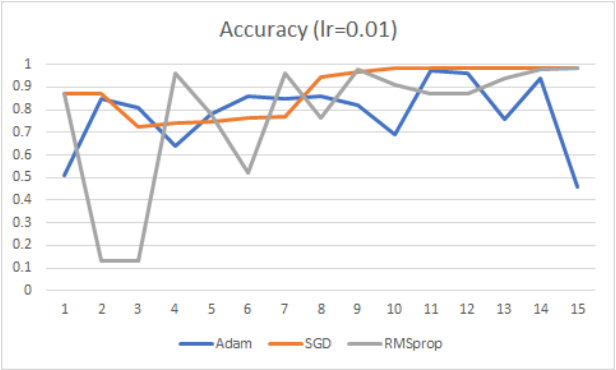


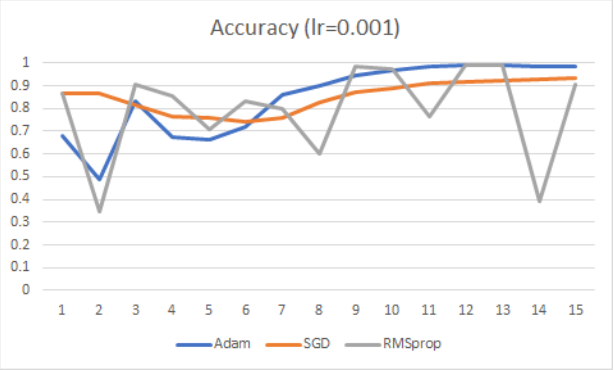



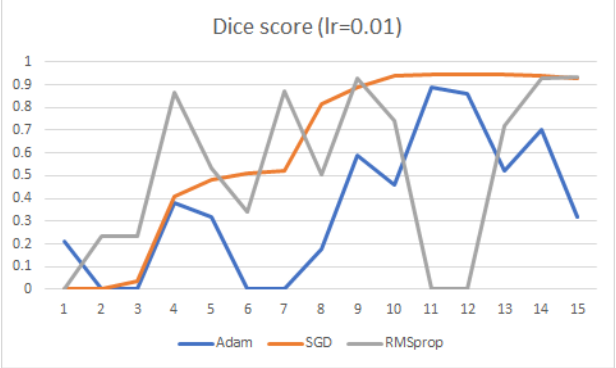

Lastly, I gave RMSprop a try and trained my model for 20 epochs for learning rate $10^{-1}$, $10^{-2}$, and $10^{-3}$ each. The performance was quite disappointing; the accuracy and Dice score oscillated continuously during the training and were never able to stabilize regardless of the learning rate, and as a consequence, I did not use RMSprop in my project any further.

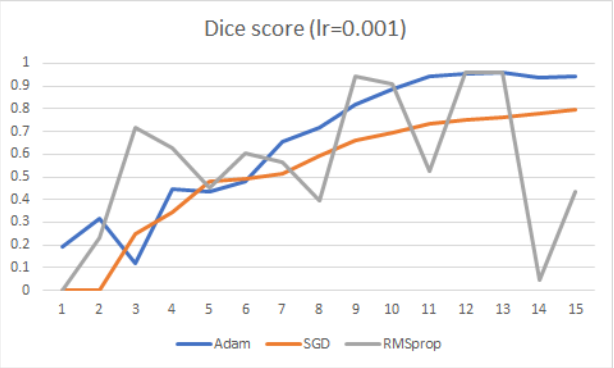
### 7.3.2. Optimizer

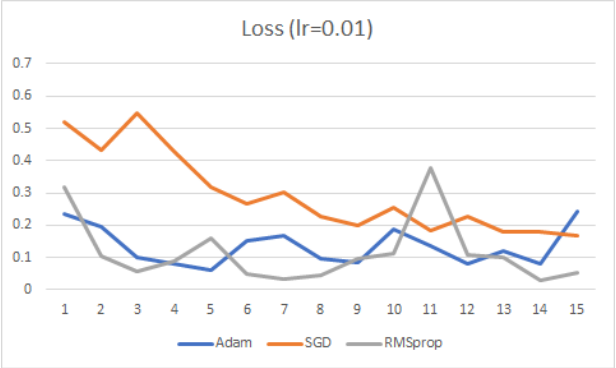
I also wanted to make direct comparisons between optimizers to decide which one is the most optimal. I compared their accuracy, Dice score, and loss under learning rate $10^{-2}$ and $10^{-3}$. For fairness, I am only comparing the first 15 epochs, as I only trained for this many epochs for Adam.

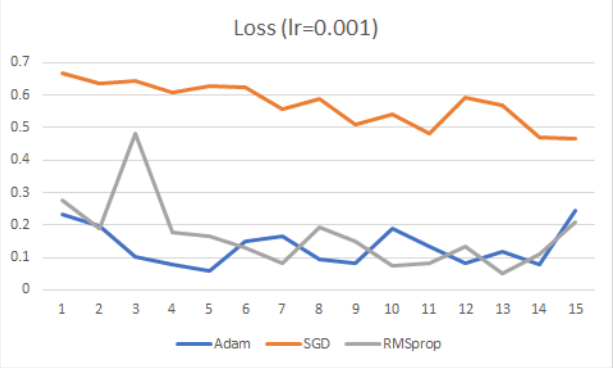


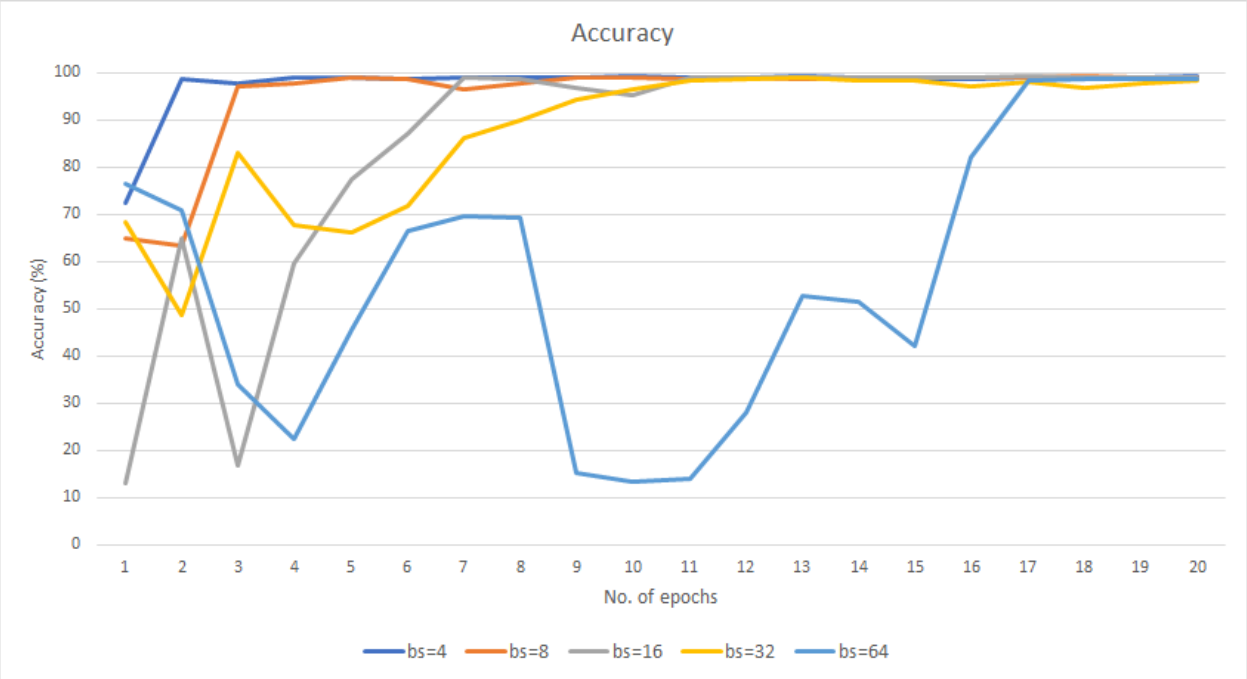



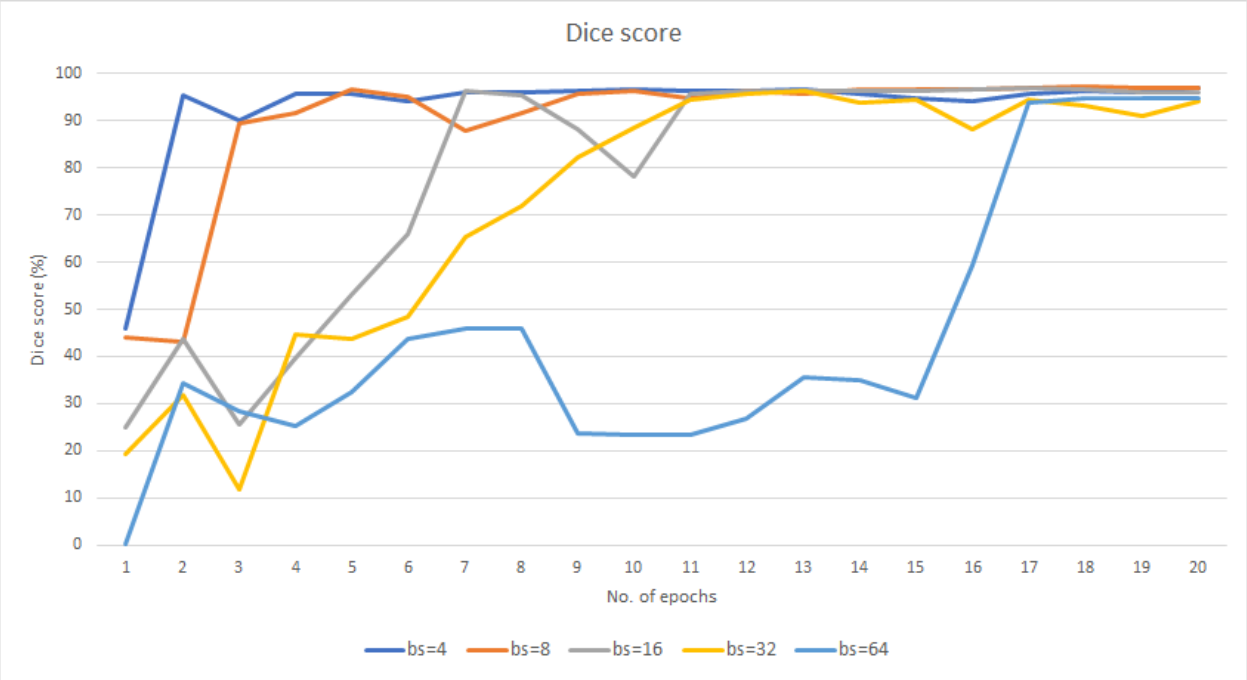

Here are my observations from the above graphs. RMSprop, as shown in previous experiments, does not perform very well with my model, as the metrics continue to oscillate during the training. Stochastic Gradient Descent performs pretty well when paired with a learning rate of $10^{-2}$, and Adam has great performance with a learning rate

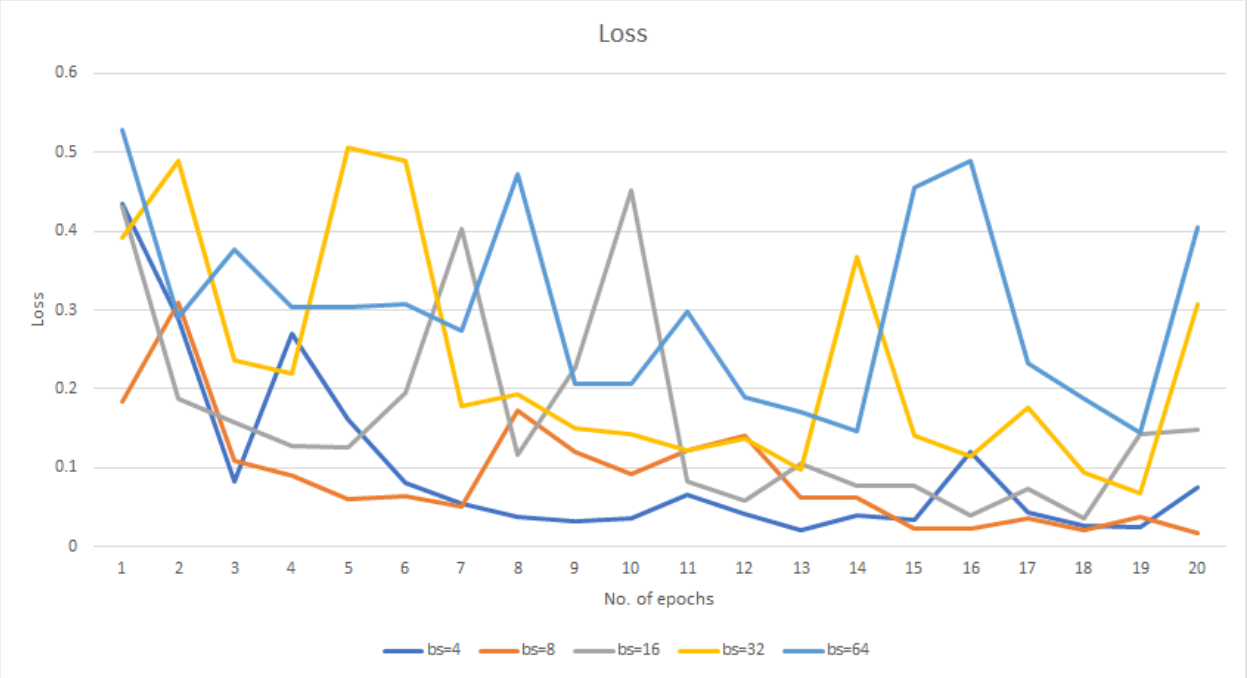
of $10^{-3}$. Eventually, I decided that the Adam optimizer, coupled with a learning rate of $10^{-3}$, is the most ideal combination, as it achieved a slightly higher accuracy (99%) and Dice score (96.2%) than SGD with $10^{-2}$ learning rate (98.5% accuracy and 94.3 Dice score).

### 7.3.3. Batch Size

I also wanted to find the optimal batch size for my training. Larger batch sizes yield more representing error gradient each time it's calculated, but tend to generalize worse and achieve lower accuracy [14]. Smaller batch sizes on the other hand yield more noisy gradient estimates but ultimately result in higher accuracy and converge faster. With an eye to figuring this out by myself, I trained my model on five different batch sizes: namely 2, 4, 8, 16, and 32. I used the Adam optimizer and a learning rate of $10^{-3}$, a combination that worked well, as shown in previous experiments.






The most obvious discovery from this experiment is that the smaller the batch size, the faster it converges, as has been established by many previous experiments by others. The training process with a batch size of 4 converges around 7 epochs in; batch size 8 converges after 10 epochs; both batch size 16 and 32 converge after 12 epochs; lastly, batch size 64 converges right towards the end, at 17 epochs. All of the above

batch sizes (barring 64) yielded a very good accuracy of close to 99% and a Dice score of 96%. What impressed me the most are batch size 4 and 8, which converge the quickest and also achieve the highest accuracy and Dice score, both having around 99.3% accuracy and a 97% Dice score. In terms of loss, those two are also the lowest. Eventually, I choose batch size 8 as the optimal value, as it slightly outperforms batch size 4 by a thin margin in all three metrics.

## 7.4. Comparison With Benchmark Model -- The Original U-Net

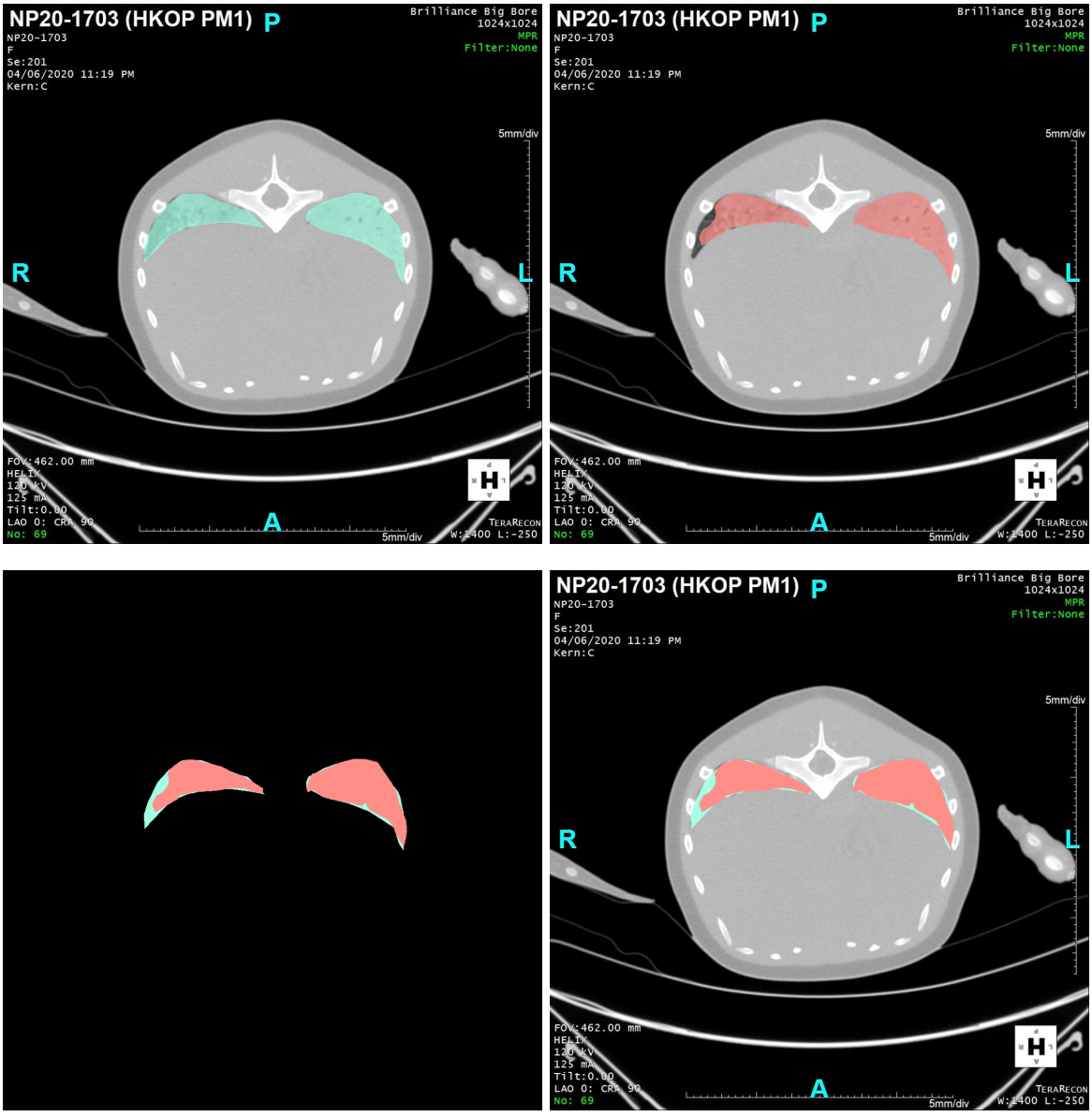
After all model enhancement and parameter optimization were set and done, I tested it on the entire dataset (the experiments in the previous section were conducted on a smaller subset of the entire dataset because the whole dataset wasa not done getting labeled until later), then compared my U-net with the original U-net without my modification and adjustments. The results are shown in the below table.

| | | **Lung detection** | **Pathology detection** |
|---|---|---|---|
| **My customized network** | **Accuracy** | 0.9886 | 0.9882 |
| | **Dice score** | 0.9293 | 0.8804 |
| **The original U-Net** | **Accuracy** | 0.9288 | 0.9794 |
| | **Dice score** | 0.8916 | 0.858 |

As shown by the results, my modified network for my project outperforms the original U-net in both the lung and pathology dataset as well as both metrics evaluated. For lung detection, my model has an accuracy of 98.86% and a Dice score of 92.93%, compared to the original U-net's 92.88% and 89.16%. As for pathology detection, my customization also outshines the original benchmark with 98.82% and 88.04% in accuracy and Dice score, exceeding the OG versions' 97.94% and 85.8% respectively.

## 7.5. Result

Now that the model is well-trained and the accuracy is thoroughly tested, it is now time for the final step: calculate the percentage of sickness using our lung & pathology detectors. As shown in the pictures below, the area of the lung, as well as the unhealthy area, can be determined by the two classifiers respectively. The two areas can then be overlapped (programmatically), and by calculating the percentage of unhealthy pixels within the whole 2D lung slice and aggregating the per-slice results across the entire 3D lung, we can come up with a percentage indicating the unhealthiness level of the lung.



By inputting the file paths where the mask images for lung and pathology predictions are saved, the program can calculate the unhealthiness percentage. It reads in the images, converts them to grayscale (line 11, 12) and then to integer NumPy arrays (black/false = 0 and white/true = 1). It then calculates the number of lung and unhealthy pixels using NumPy array addition (line 13) and counts the lung and overlapping pixels. Finally, the percentage of unhealthy -- a numeric value quantifying the healthiness level of the lung -- can be calculated by dividing the number of pathological pixels by the number of lung pixels.

```
1. lung_filenames = os.listdir('saved_images/lung/NP20')
2. pathology_filenames = os.listdir('saved_images/pathology/NP20')
3.
4. num_images = len(lung_filenames)
5. n_lungs = 0
6. n_patho = 0
7. rgb_weights = [0.2989, 0.5870, 0.1140]
8. for i in range(num_images):
9.     lung_path = os.path.join(lungs_dir, lung_filenames[i])
10.    pathology_path = os.path.join(pathology_dir, pathology_filenames[i])
11.    lung = (np.dot(plt.imread(lung_path)[..., :3], rgb_weights) >
   0.5).astype(int)
12.    pathology = (np.dot(plt.imread(pathology_path)[..., :3], rgb_weights)
   > 0.5).astype(int)
13.    n_lungs += np.count_nonzero(lung == 1)
14.    n_patho += np.count_nonzero(lung + pathology == 2)
15.
16.unhealthy_percentage = n_patho / n_lungs
17.print('Percentage of unhealthy lung = ' + unhealthy_percentage)
```

Below is the unhealthiness percentage of dolphins we have tested the model on:

| Dolphin ID | Percentage of unhealthy region |
|---|---|

| | |
|---|---|
| NP14-2308 | 93.82% |
| NP20-1703 | 79.47% |
| NP14-0604 | 82.05% |

# 8. CONCLUSION

## 8.1. Critical Review

To conclude, I am fairly proud of myself for completing this project. I did not have much experience in machine learning prior to this project, so it was not easy for me to have to learn tools like PyTorch from scratch and build a rather complex network. It definitely did not always go smoothly. At one point, I was stuck at the training phase where the accuracy and Dice score would never go up to a remotely acceptable standard. After some extensive research online (mostly Stack Overflow) and tweaking of parameters, I was able to figure out the root of the problem. It turned out that I set the batch size too high, causing it to never converge before my training epochs end. It was then that I learned first-hand that a higher batch size could lead to longer training/convergence time. It was a wonderful feeling seeing the model performing nicely and the lung contours being localized accurately after I adjust the batch size and retrain. I felt like I was starting to get a hold of what is truly happening under the hood in regard to machine learning, albeit it being a rather straightforward problem that I solved.

There are also parts where I reckon I should have done better. Regarding data labeling, I was initially given an unlabeled dataset. Instead of contacting my supervisor straight away or asking the data providers for help, I decided to go ahead and label the data manually by myself, despite not having any biological background or knowledge in marine mammal CT scans. I ended up wasting the majority of the early stages labeling (rather inaccurately, even) image after image and not spending much time on actually conducting more literature review or studying potential network architectures. In the end, Dr. Victor Lee was able to help me get in contact with the kind colleagues at the marine lab, and Dr. Kot assigned a Ph.D. student, Ms. Maria Robles, to me, who took over the labeling work since then. Had I asked for help earlier, I would have saved more time doing such tedious labeling tasks and focused more on the main part of my project. While it still worked out in the end, I think this is definitely one of the main takeaways from this project experience.

## 8.2. Potential Extensions

This system is currently able to detect the pathological/unhealthy area within a lung. I think it could be further enhanced by extending its scope to not only just pathology detection, but determining the exact kind of pathology in existence. Right now, given a set of CT scan images, the system outputs a numeric number indicating the proportion of unhealthy areas; however, in the future, it might be possible to show the percentage of each sickness. It could be something like:

| **Category** | **Percentage (%)** |
|---|---|
| Overall unhealthy percentage | 78% |
| Parasitic infection | 45% |
| Lung consolidation | 11% |
| Pulmonary fibrosis | 7% |
| Some other disease... | 15% |

This would be able to offer professionals an even more detailed look into the mammal's health status and understand the severity of each disease in comparison to just a general unhealthy percentage. I think this is a feasible direction to go towards for this project in the future.

# 9. REFERENCES


9.1. Aggarwal, T., Furqan, A., Kalra, K. (2015). Feature extraction and LDA based classification of lung nodules in chest CT scan images. 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)

9.2. Chen, Y., Ruan, D., Xiao, J., Wang, L., Sun, B., Saouaf, R., ... & Fan, Z. (2019). Fully Automated Multi-Organ Segmentation in Abdominal Magnetic Resonance Imaging with Deep Neural Networks. arXiv preprint arXiv:1912.11000.

9.3. Chilamkurthy, S., Ghosh, R., Tanamala, S., Biviji, M., Campeau, N. G., Venugopal, V. K., ... & Warier, P. (2018). Deep learning algorithms for detection of critical findings in head CT scans: a retrospective study. The Lancet, 392(10162), 2388-2396

9.4. Çiçek, Ö., Abdulkadir, A., Lienkamp, S. S., Brox, T., & Ronneberger, O. (2016, October). 3D U-Net: learning dense volumetric segmentation from sparse annotation. In International conference on medical image computing and computer-assisted intervention (pp. 424-432). Springer, Cham

9.5. Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. Nature, 542(7639), 115-118. doi:10.1038/nature21056

9.6. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

9.7. Lee, H., Huang, C., Yune, S., Tajmir, S. H., Kim, M., & Do, S. (2019). Machine Friendly Machine Learning: Interpretation of Computed Tomography Without Image Reconstruction. Scientific Reports, 9(1). doi:10.1038/s41598-019-51779-5

9.8. Lemay, A. (2019). Kidney Recognition in CT Using YOLOv3. arXiv preprint arXiv:1910.01268.

9.9. Makaju, S., Prasad, P., Alsadoon, A., Singh, A., & Elchouemi, A. (2018). Lung Cancer Detection using CT Scan Images. Procedia Computer Science, 125, 107-114. doi:10.1016/j.procs.2017.12.016

9.10. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 779-788).

9.11. Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention (pp. 234-241). Springer, Cham

9.12. Xu, Z., Bagci, U., Mansoor, A., Kramer-Marek, G., Luna, B., Kubler, A., . . . Mollura, D. J. (2015). Computer-aided pulmonary image analysis in small animal models. Medical Physics, 42(7), 3896-3910. doi:10.1118/1.4921618

9.13. Ciresan, D.C., Gambardella, L.M., Giusti, A., Schmidhuber, J.: Deep neural networks segment neuronal membranes in electron microscopy images. In: NIPS. pp. 2852–2860 (2012)

9.14. Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836.

# 10. APPENDICES

## 10.1. Section A): Monthly Log

### 10.1.1. October 2020

- Literature review: studied papers regarding CT/medical images recognition, 3D CNN, object detection techniques (etc. YOLO), ...
- Dataset obtainment: received 2359 2D CT images of a full-sized dolphin
- Familiarization with the dataset: obtained relevant knowledge in regards to the location and identification of the thorax area and lungs within a dolphin CT scan model
- Model building: built a YOLO object detection model to locate the lung areas inside dolphin CT scan models, with a success rate of close to 100%

### 10.1.2. November 2020

- Interim Report I: finished the interim report I
- Continued studying YOLO, Fast-RCNN, ResNet, and other network structures that might be used for the detection & classification task
- Meeting with Dr. Brian Kot and co.: Had a meeting with Dr. Brian Kot and his other colleagues at SKLMP to demonstrate my current working progress and discuss the future direction of the project
- Dataset acquisition: obtained the full dataset of dolphin CT scans with substantially more scans available (10 in total), containing those of varied health and decomposition condition

### 10.1.3. December 2020

- Labeling data: there are eight more dolphin CT scans given to me, each of which consists of 2000+ images that need to be labeled if lungs are present
- Continuous study of research topic: continue to study different neural networks including YOLO, U-Net, ResNet, etc.
- Determine precise research direction (3D -> 2D) and begin to explore image segmentation methods

### 10.1.4. January 2021

- Continue labeling the lungs in CT scan images
- Evaluate the object detecting model by measuring its accuracy, specificity, and sensitivity
- Study how to implement image segmentation on medical images
- Study CNN networks for binary (medical) image classification

### 10.1.5. February 2021

- Finish labeling all CT scans
- Finish training lung detecting model for extracting lung bounding boxes
- Start building the classification model
- Prepare the dataset for the classification (extract lung bounding boxes from CT scans, apply paddings to unify the size, etc.)
- Evaluate the accuracy of the lung detection model on the test set by calculating its IoU (intersection over union) index, achieving an IoU of around 90%

### 10.1.6. March 2021

- Have a meeting with Dr. Brian Kot and Ms. Maria Robles (Dr. Kot's PhD student) in regard to the expected outcome and future direction of the project to go towards
- Arrange Ms. Robles to help with data labeling
- Request more CT scan data from the marine lab
- Continue studying papers related to classification of medical images

### 10.1.7. April 2021

- Had a meeting with Dr. Brian Kot, Ms. Maria Robles, Dr. Victor Lee, and Dr. Antoni Chan to discuss and clarify project scope and expected outcome. Confirmed latest project direction to be using image segmentation to contour diseased lung area in dolphin CT scan and its percentage (of abnormality)
- Study image segmentation and U-net
- Study potentially viable implementation of U-net, e.g. PyTorch, Keras, ...
- Request help from Dr. Kot and co. for help with data labeling for image segmentation

#### 10.1.8. May 2021

- Finish labeling the entire dataset (for image segmentation)
- Study PyTorch & U-net
- Build U-net image segmentation model using PyTorch
- Evaluate model performance with test set and achieve dice score of 0.948
- Study ways to finetune the model
- Work on interim report 2

#### 10.1.9. June 2021

- Finish Interim Report II
- Continue to finetune/enhance the model
- Tweak model parameters e.g. learning rate, batch size, # of epochs to achieve highest accuracy
- Work on Final Report
- Prepare for demonstration and presentation

## 10.2. Section B): Sample of Label Format

This section will show how the image segmentation data is labeled, its format, and how it's converted to mask images in JPG format. An example of the raw JSON file annotated on the VGG Annotator website (https://www.robots.ox.ac.uk/~vgg/software/via/via.html) is as follows:


```
{
  "SC15-1505_20150515_1.jpg205781": {
    "filename": "SC15-1505_20150515_1.jpg",
    "size": 205781,
    "regions": [],
    "file_attributes": {}
  },
  "SC15-1505_20150515_2.jpg209739": {
    "filename": "SC15-1505_20150515_2.jpg",
    "size": 209739,
    "regions": [],
    "file_attributes": {}
  },
  …,
  "SC15-1505_20150515_38.jpg205944": {
    "filename": "SC15-1505_20150515_38.jpg",
```

```
"size": 205944,
"regions": [
  {
    "shape_attributes": {
      "name": "polyline",
      "all_points_x": [
        345,
        365,
        395,
        421,
        440,
        447,
        442,
        377,
        363,
        346,
        309,
        292,
        273,
        249,
        224,
        201,
        179,
        200,
        225,
        257,
        306,
        324,
        342,
        345
      ],
      "all_points_y": [
        275,
        275,
        290,
        322,
        367,
        382,
        397,
        440,
        446,
        447,
        441,
        433,
        425,
        421,
        420,
        421,
```

```
        432,
        375,
        330,
        294,
        265,
        266,
        274,
        275
      ]
    },
    "region_attributes": {}
  },
  {
    "shape_attributes": {
      "name": "polyline",
      "all_points_x": [
        713,
        746,
        773,
        810,
        826,
        834,
        837,
        837,
        812,
        781,
        747,
        712,
        688,
        661,
        643,
        614,
        625,
        634,
        638,
        646,
        663,
        694,
        715
      ],
      "all_points_y": [
        286,
        286,
        298,
        339,
        367,
        387,
        403,
```

```
            407,
            393,
            379,
            369,
            368,
            369,
            367,
            361,
            356,
            341,
            322,
            309,
            299,
            291,
            289,
            286
          ]
        },
        "region_attributes": {}
      }
    ],
    "file_attributes": {}
  },
  …,
}
```

Each entry in the JSON file (with the file name being its key) represents an image's label. "regions" contains a list of annotated areas that is an occurrence of the object in question. Each region within "regions" is represented by a polygon, in the form of a list of (x, y) coordinates. Then, I wrote a simple Python program to convert the JSON file into JPG mask images for my program's usage. The code is as follows:

```
with open(r'C:\Users\tai10\Downloads\via_project_28Jun2021_11h3m_json.json') as f:
    data = json.load(f)

for itr in data:
    regions = data[itr]['regions']
    img = np.zeros((1024, 1024))
    for region in regions:
        if region and 'shape_attributes' in region and 'cx' in region['shape_attributes']
and 'cy' in region['shape_attributes']:
            if region['shape_attributes']['name'] == 'ellipse':
                img = cv2.ellipse(img,
                                  (region['shape_attributes']['cx'],
region['shape_attributes']['cy']),
                                  (round(region['shape_attributes']['rx']),
round(region['shape_attributes']['ry'])),
                                  round(region['shape_attributes']['theta'] /
```

```
3.14159265358979 * 360),
                                    0,
                                    360,
                                    255,
                                    -1)
            elif 'r' in region['shape_attributes']:
                img = cv2.circle(img, (region['shape_attributes']['cx'],
region['shape_attributes']['cy']),
                                 int(region['shape_attributes']['r']), (255, 255, 255), -1)
            else:
                img = cv2.circle(img, (region['shape_attributes']['cx'],
region['shape_attributes']['cy']), 5,
                                 (255, 255, 255), -1)
        elif region and 'shape_attributes' in region and 'all_points_x' in
region['shape_attributes'] and 'all_points_y' in region['shape_attributes']:
            points = []
            for i in range(len(region['shape_attributes']['all_points_x'])):
                points.append(
                    [region['shape_attributes']['all_points_x'][i],
region['shape_attributes']['all_points_y'][i]])
            img = cv2.fillPoly(img, np.array([points], dtype='int32'), 255)
    if len(regions) == 0:
        print(f'Skipped {data[itr]["filename"]}')
    cv2.imwrite(r'C:\Users\tai10\Desktop\test vgg circle and point\patho\\' +
data[itr]["filename"][:-4] + '_mask.png', img)
```

And here is a demonstration of the mask image generated, juxtaposed with the original raw image: