Image Analysis

Sondre Wikberg
Prakash Nair
Zheng Cao
Yingzhe He

Year 4 Project
School of Mathematics
University of Edinburgh
Report Date

Abstract

This project provides a detailed investigation into the mathematical aspects of several image preprocessing and segmentation algorithms, and object-detection methods in machine learning. We provide a balanced discussion on the strengths and weaknesses of each approach, culminating in their application to medical anomaly detection, specifically in breast mammography.

Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text.

(*Sondre Wikberg*
Prakash Nair
Zheng Cao
Yingzhe He)

We would like to thank our supervisor, Dr Stuart King, for his insight, patience and unwavering motivation.

Contents

Chapter 1

Introduction

1.1 Image storage and representation

Two-dimensional digital images have two main variants: vector graphics and raster images. The former consists of shapes described by mathematical equations on a Cartesian plane, and the latter is based on pixels that are in a grid-like pattern. This project is focuses on raster images, as they are the main format of medical images, and we henceforth use the terminology "image" instead of "raster image" in the following discussion. In the rest of this section, we define the representation of digital images. The content in this section takes inspiration from "Digital Image Processing" by W. Burger and M. J. Burge [Burger and Burge, 2016].

The basic unit of an image is the pixel, which is usually a single square point. Figure 1.1 gives an analog of the arrangement of pixels in an image. An $M \times N$ image means that the image has N pixels in a row and M pixels in a column. The size of an image is determined by the number of pixels it contains. The coordinate system can be used to specify the location of pixels. In Python, $(0, 0)$ refers to the pixel at the top-left of an image, and $(M-1, N-1)$ corresponds to the pixel at the M^{th} row, N^{th} column of the image. In this example, we see that an image can be mapped to a matrix of the same size. In general, any greyscale image I can be considered as a discrete function that maps the location of each pixel to its corresponding value. We note that the coordinates of any pixel, x and y, are both natural numbers. If we use $\mathbb{P}$ to denote all possible values of pixels, then I is given by,

$$\begin{aligned} I : \mathbb{N} \times \mathbb{N} &\to \mathbb{P} \\ (x, y) &\mapsto p_{x,y} \end{aligned}$$

The formula above can be expressed more explicitly, for an $M \times N$ image I,

$$I(x, y) = \begin{bmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,N-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{M-1,0} & p_{M-1,1} & \cdots & p_{M-1,N-1} \end{bmatrix}. \tag{1.1}$$

An advantage of this representation is that the application of linear transformations to an image is relatively straightforward.

	0	1	2	3	$\cdots$	$N-1$
0	•	•	•	•	$\cdots$	•
1	•	•	•	•	$\cdots$	•
2	•	•	•	•	$\cdots$	•
3	•	•	•	•	$\cdots$	•
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$M-1$	•	•	•	•	$\cdots$	•

Figure 1.1: Analog of representation of $M \times N$ image in Python

A pixel is a representation of a specific brightness and colour, with each pixel having a value or a set of values to indicate its hue. The most fundamental type of image is the black and white image, in which each pixel only has two possible values, 0 or 1. A value of 0 denotes black and 1 signifies white. This type of image is also called a binary image. In addition to black and white, images can also display various levels of colour intensity between these two extremes. This type of image is referred to as a greyscale image. In greyscale images, only integers are assigned as the value of pixels instead of a fraction between 0 and 1. Pixel values are commonly binary words of length k and thus a pixel can correspond to any of 2^k colour between black and white [Burger and Burge, 2016]. The number k is the bit depth of the image. The most common value of k is 8, such that each pixel is assigned a value between 0 and 255 . Figure 1.2 shows how Python reads an 8-bit greyscale image, and we see that for a greyscale image, $\mathbb{P}$ is a subset of $\mathbb{N}$. An important definition associated with the value of pixels is contrast. It refers to the difference in intensity between the brightest pixel and the darkest pixel. This definition implies that images with a high contrast always have better visibility compared to images with low contrast, as textures within a high contrast image can be displayed better in clarity. In some segmentation techniques, such as thresholding, a high-contrast image can preserve more detail when converting the image into a binary representation. A more specific discussion will be presented in section 2.2.

$\Rightarrow$

97	90	56	96	116	65	92	128	$\cdots$
76	64	71	80	54	41	88	69	$\cdots$
80	58	89	123	97	108	111	107	$\cdots$
62	36	112	115	99	104	108	95	$\cdots$
77	111	111	61	45	86	88	58	$\cdots$
107	111	68	76	113	60	76	97	$\cdots$
87	85	74	137	125	96	107	109	$\cdots$
138	114	101	121	104	123	118	112	$\cdots$
91	91	78	80	74	75	87	58	$\cdots$
68	92	97	105	81	85	104	111	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 1.2: Representation of a greyscale image in Python

Most colour images use the RGB model, in which each image has three channels of colour: red, green and blue. In these images, pixels correspond to a set of natural numbers with length 3, and each number denotes a colour component

respectively. Thus, for a k-bit RGB image, each pixel would need 3×2^k bits to encode all colours. As a consequence, this type of image would typically occupy three times the amount of computer storage space compared to a greyscale image. Another common colour model is CMYK, which is an abbreviation of Cyan-Magenta-Yellow-Black. This model is usually used in the digital prepress process.

1.2 Image Segmentation Methods

In this section, we are going to provide an overview of common image segmentation methods. A more specific discussion will be presented in the next two chapters. Image segmentation is the process that subdivides an image into several small regions based on some criteria, which is a fundamental area in computer vision. Let I represent the whole spatial region of a digital image, subregions of this image after applying a segmentation algorithm can be viewed as $I_1, I_2, \ldots, I_n$, such that [Gonzalez and Woods, 2014],

1. $\forall i, j \in [1, n]$ and $i \neq j$, $I_i \cap I_j = \emptyset$;
2. $\bigcup_{k=1}^{n} I_k = I$.

Many segmentation algorithms are based on the similarity and discontinuity of pixel values in an image. Because these algorithms can be implemented via statistical models on the spatial domain or the frequency domain of an image, they are usually categorised as "classical" methods. We shall use region growing as an example of statistical approaches. This algorithm is introduced by R. Adams and L. Bischof in 1994, which takes a grey-scale image and several "seeds" as input [Adams and Bischof, 1994]. Each "seed" can be a single pixel or a connected region. When performing the segmentation, the similarity in intensity between the "seeds" and their neighbour is calculated, if the similarity criteria are satisfied then the neighbour pixel would be added to the segmented area. A few variants and improvements of region growing was proposed after Adams and Bischof's work. In 1997, A. Mehnert and P. Jackway created a new region growing algorithm that is independent of pixel order [Mehnert and Jackway, 1997]. This development makes the segmentation result independent of the order of pixels being processed. F. Shih and S. Cheng presented another improved region growing method [Shih and Cheng, 2005] which allows for the segmentation of colour images via region growing; it uses a colour histogram to select seeds automatically. Because of the similarity and discontinuity, properties of the segments of an image defined previously may be extended by the following:

1. I_k is a connected set, for $k = 1, \ldots, n$
2. $Q(I_k) = \text{TRUE}$ for $k = 1, \ldots, n$
3. $Q(I_k \cup I_j) = \text{FALSE}$ for any adjacent regions I_k and I_j

In the last two properties, $Q(I_k)$ is true if every pixel in I_k is at the same intensity level.

The approach of machine learning methods significantly differs to that of "classical" methods. In machine learning, every image is flattened into a single row, and each pixel in the image is treated as a feature. Traditional machine learning methods such as support vector machines (SVM), and random forest (RF) have advantages in terms of their explainability. These models can perform well on small datasets and few features. However, implementing those models sometimes requires prior knowledge and feature engineering. These models have difficulties in capturing non-linear relationships as well [Seo et al., 2020]. Previous studies have also found that such methods are ineffective in identifying edges in radiological images [Seo et al., 2020]. Based on a large training set, "modern" machine learning methods such as convolutional neural networks (CNNs) provide significantly better performance, as they are able to conduct automatic feature extraction [Yamashita et al., 2018]. However, due to the sensitive nature of medical images and the tedium involved in annotating them, it is difficult to find a publicly available dataset that contains a large number of images with accurate annotations. In the following chapters, we examine the applicability of neural networks on a particular publicly available dataset.

1.3 Medical Images

Biomedical images serve as a visual representation of the human anatomy and are instrumental in identifying and diagnosing various illnesses. We introduce some main techniques used to produce medical images. Discussions in this section are based on the book "Introduction to Medical Imaging: Physics, Engineering and clinical application" by N. Smith and A. Webb [Smith and Webb, 2011].

Radiography

Radiography is the most common diagnostic imaging method, which uses penetrative radiation to project internal parts of the body onto a 2-dimensional image. The two main methods of radiography are X-ray (plain radiography) and computed tomography (CT) scan. From October 2021 to October 2022, over 21 million X-ray tests and over 6 million CT scans were performed on NHS patients in England, accounting for 64.6% of medical imaging activity in England [a20, 2023]. Both plain radiography and CT scans are based on the principle that different tissues have various absorption rates of X-ray. When performing a plain radiography test, a beam of X-rays is produced and directed towards the body part being imaged. A solid-state flat panel detector placed below the patient is used to detect X-rays passing through the body. To minimize the effects of scattered X-rays and enhance image contrast, an anti-scatter grid can be positioned directly in front of the detector. The pattern of X-ray absorption is recorded and used to create an image. In contrast to an X-ray, CT scans can be used to produce full 3-dimensional images of a particular region of the body. In a helical CT scanner, hundreds of detectors are arranged in parallel alignment to each other. They are rotated with an X-ray tube while the patient is slowly moving through the beam. The 2-dimensional rotational and 1-dimensional linear translation together

allow for the construction of a complete 3-dimensional image. The application of X-ray includes producing digital subtraction images and digital mammography, and producing digital chest imaging [Webb and Flower, 2012]. Compared to plain 2-dimensional X-ray imaging, CT scans have various benefits, allowing precise determination of the size, shape and location of a tumour, or allowing the doctor to take targeted needle biopsies. All types of radiographic imaging subject patients to radiation which may be harmful at high doses. Since the radiation dose of a CT scan is much higher than that of an X-ray, the benefits of CT do not always outweigh the risks. Consequently, CT scans are not normally used unless the patient presents with symptoms [Power et al., 2016].

Ultrasound

The basis of this diagnostic imaging method is the differential reflection rate of ultrasound (1 to 15 MHz for clinical use) by various tissues. Typically, to perform an ultrasound scan, a probe with an array of up to 512 individual active sources is used to generate the sound wave. Those individual sources are divided into small subgroups and fired sequentially to produce parallel ultrasound beams. Because of the difference in acoustic and physical properties, a small amount of sound is reflected when it reaches the boundary between tissues. These reflected waves are detected by the transducer and the distance to the boundary can be calculated by the transmission time. Then, a real-time image is produced based on the transmission time and the strength of the reflected wave. This method has the advantage of portability, in that the probe can move over the skin, be inserted into the body, or even be attached to an endoscope which passes further into the body to scan the stomach or oesophagus.

MRI

MRI stands for magnetic resonance imaging. An MRI system consists of three main components: a superconducting magnet, a set of three magnetic field gradient coils and a radio frequency transmitter. Typically, the strength of the surrounding magnetic field is 3 Tesla, which is about 6×10^4 greater than the Earth's magnetic field. While the patient's body lies on the machine, the magnetic field aligns protons within the body's hydrogen atoms and precesses protons at the 'resonance' frequency. The magnetic field gradients determine the resonance frequency depending on the location of protons. Then, radio waves with a frequency of 128 Hz are transmitted by the radio frequency transmitter, kicking protons out of alignment. When the radio wave is turned off, protons realign and release energy in the form of radio signals, which are detected by the receiver. These signals, and the different rates at which protons realign, provide detailed information about the location of protons in the body, which can be used to create images of internal structures. An MRI scan can be used to examine most parts of the body, including the brain, breast, and internal organs.

1.4 Motivation

In the context of anomaly detection, although medical images can display the region that contains lesions or tumours, the quality of an image might be influenced by various factors. Thus the symptomatic organ could be obscured and mislead the diagnosis. In Figure 1.3, we can see that it is hard to identify the tumour without the annotation by a medical professional. By doing image segmentation, a specific region in an image can be identified and isolated. This technique can sometimes allow doctors or researchers to more easily extract information such as the size and shape of a tumour. With modern techniques, such as machine learning, image segmentation can be used to develop systems that analyse a large number of images quickly and accurately. Such systems allow the clinical workflow to be more efficient. In this project, we are going to implement several classical image segmentation methods such as region growing and thresholding, as well as machine learning approaches for this task. These methods will be applied to the same dataset, and we will compare the advantages and disadvantages of each method.

Figure 1.3: Example of two CESM images with and without annotation

For the application of our methods, we use the "Categorized Digital Database for Low Energy and Subtracted Contrast Enhanced Spectral Mammography Images (CDD-CESM)" dataset, provided by The Cancer Image Archive [Khaled et al., 2021]. This dataset contains 2006 high-resolution images (with an average of 2355×1315 pixels) taken from 326 patients, including annotations and medical reports. As a part of preprocessing, every image in this collection is converted from DICOM (a common format of medical images which stores related information such as the detail of patients) to JPEG via RadiAnt with 100% image quality. Images in this dataset are produced by standard digital mammography equipment, with additional software to conduct the dual-energy image acquisition. This technique performs recombination and subtraction between a low-energy image ($26 \sim 31$ kVp) and a high-energy image ($45 \sim 49$ kVp) to reduce the visibility of breast parenchyma in the background. A pair of low and high energy images are two

exposures in the same view.

1.5 Project Outline

In Chapter 2 we are going to introduce various "classical" preprocessing methods and segmentation methods, including non-local mean denoising, region growing, thresholding, binary operations and edge detection. We will discuss the scientific aspects and the implementation of these methods along with their pseudo-code. We also provide several examples of our algorithms. Three of the test images (board, coin, remote) are taken by a member in the group, and one is a CT scan showing a large tumour in the patient's left lung, from Wikimedia, CC BY-SA 2.0. In Chapter 3 we discuss the fundamental principles and architecture of the "modern" machine learning approaches, as well as a few variants of neural networks, such as CNNs and faster R-CNNs. Chapter 4 presents our image segmentation pipeline that uses the classical methods covered in chapter 2. The results after applying the classical methods pipeline and the faster R-CNN on the CDD-CESM dataset are also included in that chapter with discussions. Chapter 5 contains the conclusions of this project.

Chapter 2

Classical Methods

2.1 Image Denoising

The presence of noise in biomedical images is a ubiquitous phenomenon arising from a variety of causes, including the detection of photons, and the transformation of analog signals into digital signals [Morin and Mahesh, 2018]. The reconstruction algorithms, conversion of the analogue signal and detection of photons play a significant role in determining the level of noise in CT scans. In an ultrasound, noise is affected by the expected location of the anatomy [Morin and Mahesh, 2018]. Noise is one of the main factors that characterises the quality of an image and the presence of a significant amount of noise inevitably weakens the result of segmentation algorithms. Thus, image-denoising techniques become essential for many aspects of image analysis. We begin by introducing the common types of noise seen in images.

2.1.1 Image Noise Models

Gaussian Noise

This eponymous noise model assumes that the signals of noise are normally distributed [Gonzalez and Woods, 2014], where the probability density function (PDF) of a Gaussian random variable z is:

$$f(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(z-\bar{z})^2}{2\sigma^2}}$$

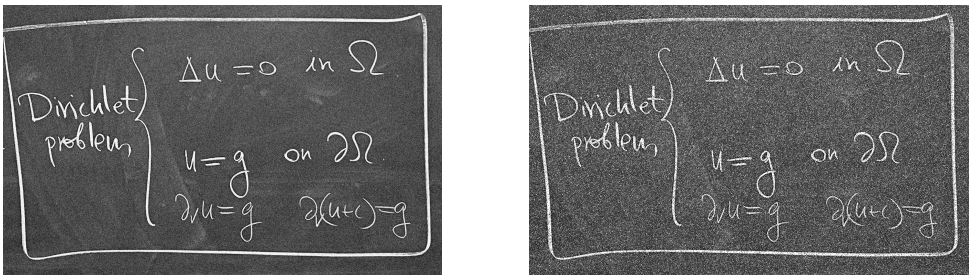
where z is the brightness of a pixel and σ is the standard deviation of the noise. Gaussian noise is the most widely used model [Gonzalez and Woods, 2014]. A reason for its popularity is that the Gaussian distribution can approximate noise in a variety of contexts well, e.g. electronic noise and thermal noise. This phenomenon is a consequence of the Central Limit Theorem, which states that the sum of multiple random variables with different PDFs leads to a signal with a Gaussian PDF [Gravel et al., 2004].

Impulse Noise

The PDF of impulse noise is given by [Gonzalez and Woods, 2014]:

$$f(z) = \begin{cases} P_a, & z = a \\ P_b, & z = b \\ 0, & \text{otherwise} \end{cases}$$

Notice that $P_a + P_b = 0$ and the noise is called unipolar noise if either P_a or P_b is zero. When a and b represent the intensity of purely white and black pixels, respectively, this special case of impulse noise has a more common name: Salt and Pepper Noise. This type of noise is usually generated during the process of image acquisition and transmission [Gupta and Sunkaria, 2017].



(a) **(b)**

Figure 2.1: (a) image with Gaussian noise (b) image with Salt and Pepper noise

2.1.2 Spatial Filtering Methods

Filters in image analysis can act on the spatial domain, or the frequency domain. In this section, we will focus on the former. If we categorise spatial filtering methods by the relationship between the pixel to be denoised and its corresponding region of analysis, there are two types: local filters and non-local filters.

Local Filters

When applying local filters, we usually consider points that are within a fixed distance of the pixel to be denoised in the spatial domain. The most intuitive spatial filter would be the *mean filter*, i.e. the intensity of every pixel in the denoised image is the mean of the intensity of pixels adjacent to it in the noisy image. We formulate its process as follows:

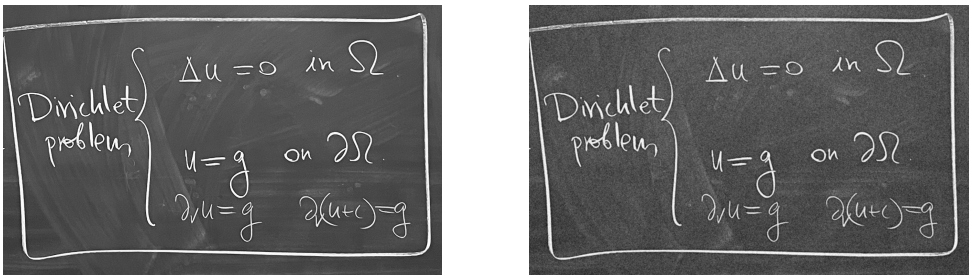
$$g(i,j) = \frac{1}{9} \sum_{n=i-1}^{i+1} \sum_{m=j-1}^{j+1} f(n,m),$$

where $g(i,j)$ and $f(i,j)$ represent the intensity of pixel (i,j) in the denoised and noisy images, respectively. This mean filter is the simplest type of linear filter.

The general form of a linear filter is given by [Szeliski, 2011]:

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l) \tag{2.1}$$

The terms $h(k,l)$ are called filter coefficients. The family of linear filters also includes the Gaussian filter, Weiner filter, and others. Figure 2.2 is an example of using the mean filter to denoise an image. We observe two disadvantages of this filter: it cannot remove all the noisy pixels, and it blurs the edges. It is noted that the latter is a common drawback for most linear filters [Goyal et al., 2020].



(a) **(b)**

Figure 2.2: (a) original image. (b) denoised image after applying mean filter.

Another way to representat the above mean filter is using a 3×3 matrix with all entries scaled by $\frac{1}{9}$. To see this, consider the following analogue,

$$\begin{pmatrix} 13 & 78 & 191 \\ 9 & 178 & 91 \\ 208 & 176 & 10 \end{pmatrix} * \frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix},$$

where the first matrix contains the intensities of 9 adjacent pixels in an image. We apply the mean filter to the pixel of intensity 178 by *convolving* the right matrix with the left. Then, the intensity of that pixel after processing would be,

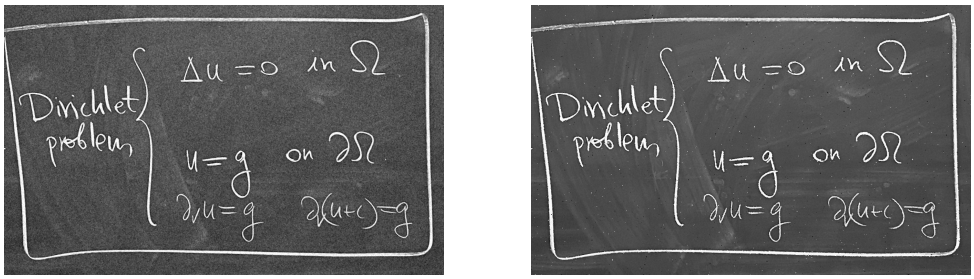
$$\frac{13+78+191+9+178+91+208+176+10}{9} = 106.$$

Matrices with similar functionalities as the right matrix above are called kernels, masks, or convolution matrices. When convolved with each pixel in an image, such matrices can be used to alter or analyse a given image in many ways, such as sharpening, blurring, and computation of gradients. The general expression for the convolution of W to a pixel $I(i,j)$ is given by [Gonzalez and Woods, 2014],

$$I(i,j) * W = \sum_{s=-a}^{a} \sum_{t=-b}^{b} I(i-s, j-t)W(s,t), \tag{2.2}$$

where the size of W is $(2a+1) \times (2b+1)$. In the analogue above, as W is a 3×3 matrix, both a, b have a value of 1. We recognise that this formula is very similar

to our general equation of linear filters.
The *median filter* is a straightforward non-linear filter that uses the median of the region around the pixel to be denoised so as to replace the intensity of that pixel. Figure 2.3 is the result of using the median filter to denoise the same image as in Figure 2.2 with Gaussian noise and salt and pepper noise respectively. We see that although the median filter does not work particularly well with the Gaussian noise, it is effective in denoising the salt and pepper noise. This is because the brightness of salt and pepper noise is always 0 or 255 (in an 8-bit image). Those values are too extreme to be the median of a region unless most of the pixels in the region are purely black or white. For most images, the median filter struggles to preserve image details and overcome Gaussian blurring as corroborated by [Goyal et al., 2020].



(a) **(b)**

Figure 2.3: (a) denoised image after applying the median filter on image with Gaussain noise. (b) denoised image after applying the median filter on image with Salt and Pepper noise.

Non-Local Filter

A *non-local filter* models the similarity between one particular region and every other region in the image. We focus on the non-local mean filter in this section, which was introduced in 2005 by A. Buades [Buades et al., 2005]. The algorithm builds on an abstraction of a noisy image as a continuous two-dimensional domain, on which an integrable intensity v function is defined. Meanwhile, $\Omega \subset \mathbb{R}^2$ denotes the domain and $\mathbf{z}, \hat{\mathbf{z}} \in \Omega$ denote locations in the domain. The denoised intensity of $\mathbf{z}$ is set to be the integral over all locations whose neighbourhood in intensity space is similar to the neighbourhood of $\mathbf{z}$. This may be expressed as:

$$NL[v](\mathbf{z}) = \frac{1}{C(\mathbf{z})} \int_{\Omega} \exp\left(\frac{(G_a * |v(\mathbf{z}+.) - v(\hat{\mathbf{z}}+.)|^2)(0)}{h^2} \right) v(\hat{\mathbf{z}}) d\hat{\mathbf{z}}, \tag{2.3}$$

where h is a filtering parameter and its value is not fixed (usually $h = 10$). G_a is a Gaussian kernel with standard deviation a. The normalising factor is given by,

$$C(\mathbf{z}) = \int_{\Omega} \exp\left(\frac{(G_a * |v(\mathbf{z}+.) - v(\hat{\mathbf{z}}+.)|^2)(0)}{h^2} \right) d\hat{\mathbf{z}}.$$

In the convolution $(G_a * |v(\mathbf{z}+.) - v(\hat{\mathbf{z}}+.)|^2)(0)$, $v(\mathbf{z}+.)$ and $v(\hat{\mathbf{z}}+.)$ represent the intensities of small patches centered on pixels $\mathbf{z}$ and $\hat{\mathbf{z}}$, respectively. The (0) indicates that the convolution is evaluated at 0, i.e. the origin. This notation aligns the centre of the kernel with the origin when evaluating the convolution. The convolution can then be expressed as:

$$(G_a * |v(\mathbf{z}+.) - v(\hat{\mathbf{z}}+.)|^2)(0) = \int_{\mathbb{R}^2} G_a(\mathbf{t}) \, |v(\mathbf{z}+\mathbf{t}) - v(\hat{\mathbf{z}}+\mathbf{t})|^2 \, d\mathbf{t},$$

$$\text{where } G_a(\mathbf{t}) = \frac{1}{2\pi a} \exp(\frac{\|\mathbf{z}\|^2}{a^2})$$

Equation (2.3) assumes that every point in the relevant image is continuous, but that is not the case in reality. Thus, we need to reformulate the convolution operation for the discrete case. For a noisy image, whose pixels are contained in the set $\mathbf{I}$, $v = \{v(z)|z \in I\}$, the non-local mean for a pixel z is computed as:

$$NL[v](z) = \sum_{\hat{z} \in I} w(z, \hat{z}) v(\hat{z}). \tag{2.4}$$

In the equation above, $w(z, \hat{z})$ represents a weight, which positively correlated to the similarity in intensity between pixels z and $\hat{z}$. We define w such that $\sum_{\hat{z}} w(z, \hat{z}) = 1$. The similarity between z and $\hat{z}$ is calculated by the squared Euclidean distance $\|v(\mathcal{N}_z) - v(\mathcal{N}_{\hat{z}})\|_2^2$, where $\mathcal{N}_z$ is the square neighbourhood centred at i with a fixed width. Then $w(z, \hat{z})$ is calculated as:

$$w(z, \hat{z}) = \frac{1}{Z(\hat{z})} \exp\left(\frac{\|v(\mathcal{N}_z) - v(\mathcal{N}_{\hat{z}})\|_2^2}{h^2}\right),$$

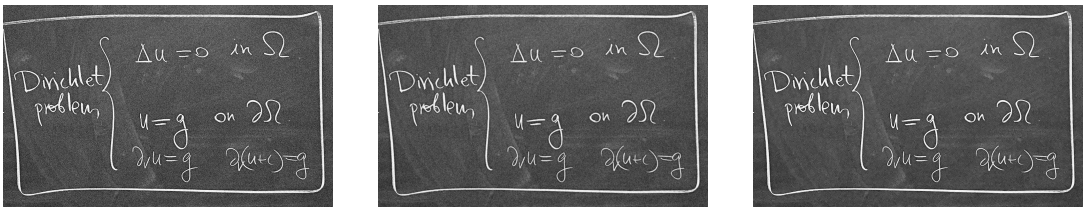
where $Z(z) = \sum_{\bar{z} \in I} \exp\left(\|v(\mathcal{N}_z) - v(\mathcal{N}_{\bar{z}})\|_2^2 \cdot h^{-2}\right)$ is the normalising constant to ensure $\sum_z w(z, \hat{z}) = 1$, and where h controls the decay of the exponential function. The value of h is a constant and, as in the continuous case, is usually $h = 10$. Figure 2.5 gives an example of applying the non-local filter with $h = 10$ to the image seen above. We see that the result of the non-local mean filter is better in overcoming blurring.

h = 1 h=3 h=10 h=100

Figure 2.4: Example of applying the non-local mean filter with different h values

The drawback of the non-local mean filter is that it requires significant computation time. The average time of denoising the 200×200 images in Figure 2.4

was 62 seconds, and denoising the 1277×783 image in Figure 2.5(**a**) took 1578 seconds. In practice, when applying the non-local mean algorithm, we always set a search window and compare the pixel to be denoised with every pixel within the search window instead of using the whole image, to reduce the computational cost. Suppose the search window is a square with width D, and $\mathcal{N}_z$, $\mathcal{N}_{\hat{z}}$ are square neighbourhoods of width d, then the non-local mean algorithm on a grey-scale image containing N pixels has a computational cost of $\mathcal{O}(Nd^2D^2)$ [Froment, 2014].



(a) **(b)** **(c)**

Figure 2.5: (a) image with Gaussian noise. (b) denoised image after applying the local mean filter. (c) denoised image after applying the non-local mean filter.

Operation	Kernel	Image
Identity	$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	
Gaussian Blur	$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$	
Mean Filter (Box Blur)	$\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	
Sharpen	$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$	
Laplacian	$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$	

Table 2.1: Some examples of kernels and the visualisation of their application

2.2 Thresholding

Thresholding is one of the most straightforward classical methods for image segmentation. It seeks to convert a greyscale image (or colour image) into a binary image by separating it into background and foreground. This is done by partitioning the image into two disjoint sets C_0 and C_1, representing background and foreground respectively [Burger and Burge, 2016]. In this report colour images will be converted into greyscale images using the following equation:

$$Y = 0.2125R + 0.7154G + 0.0721B$$

Where R, G, B are the red, green, blue values of a colour image (input), respectively, and Y is the value of the greyscale image (output). $W_R = 0.2125$, $W_G = 0.7154$, $W_B = 0.0721$ are the weights for R, G, B for the conversion respectively. The aim of converting a colour image is to predict the luminance value of each pixel. The weights mentioned above are the standard weights for computing luminance from RGB values as given in the International Telecommunication Union Broadcasting Service (Television) Standards 709 [ITU-R, 2015].
Let (u, v) be the coordinates of a M by N greyscale image. We thus partition

the image by the following:

$$(u,v) \in \begin{cases} C_0 \text{ if } I(u,v) \leq q \\ C_1 \text{ if } I(u,v) > q \end{cases},$$

where $I(u,v)$ is the pixel value of the image at (u,v) and q is the threshold [Burger and Burge, 2016]. We are then faced with the question of how to choose an optimal value of q. The threshold may be the same for all pixels in the image or if may depend on individual pixel and varies across the image (denoted by $T(x,y)$ instead of q). There are several ways to do that, and we focus on four such methods.

2.2.1 Histogram

Before the discussion of threshold selection methods, it's important to define the histogram of an image first. The histogram of a greyscale image is the distribution of intensity value for an in a range $I(u,v) \in [0, K-1]$. Examples of histograms are given in Figure 2.6. For a typical 8-bit greyscale image $K = 2^8 = 256$ and will be the range used in our following discussion. The information derived from the histogram of an image is used in many thresholding techniques and we proceed to discuss two such methods.

2.2.2 Median

One of the simplest ways to find a threshold value for a greyscale image is to compute the median of $I(u,v)$ of all the pixels of the image. By using the median as the threshold, the foreground and the background set have roughly the same number of pixels [Burger and Burge, 2016]. We provide examples with their histograms in Figure 2.7.
As illustrated by Figure 2.7, the median threshold is not very effective in segmenting images with an unequal number of pixel counts in foreground and background (all examples except the lung image). Even on the lung image, this method fails to categorise parts of the lung as the background. Therefore the median threshold method is not generally practical as it only works for images with almost exactly the same number of pixels in the foreground and background.

2.2.3 Otsu's Method

Otsu's method is a statistical method that assumes an image contains pixels from two classes. The method works by seeking a threshold that maximally separates the background and foreground based on that assumption. In statistical terms, this means the background and foreground partitions have minimal variances and maximal distance between their respective means [Burger and Burge, 2016].
There are three key parameters for Otsu's method. The first one is $\sigma_w(q)^2$, which denotes the within class variance, measuring the combined distance of distribution for the foreground and background. The second one is $\sigma_b(q)^2$, which

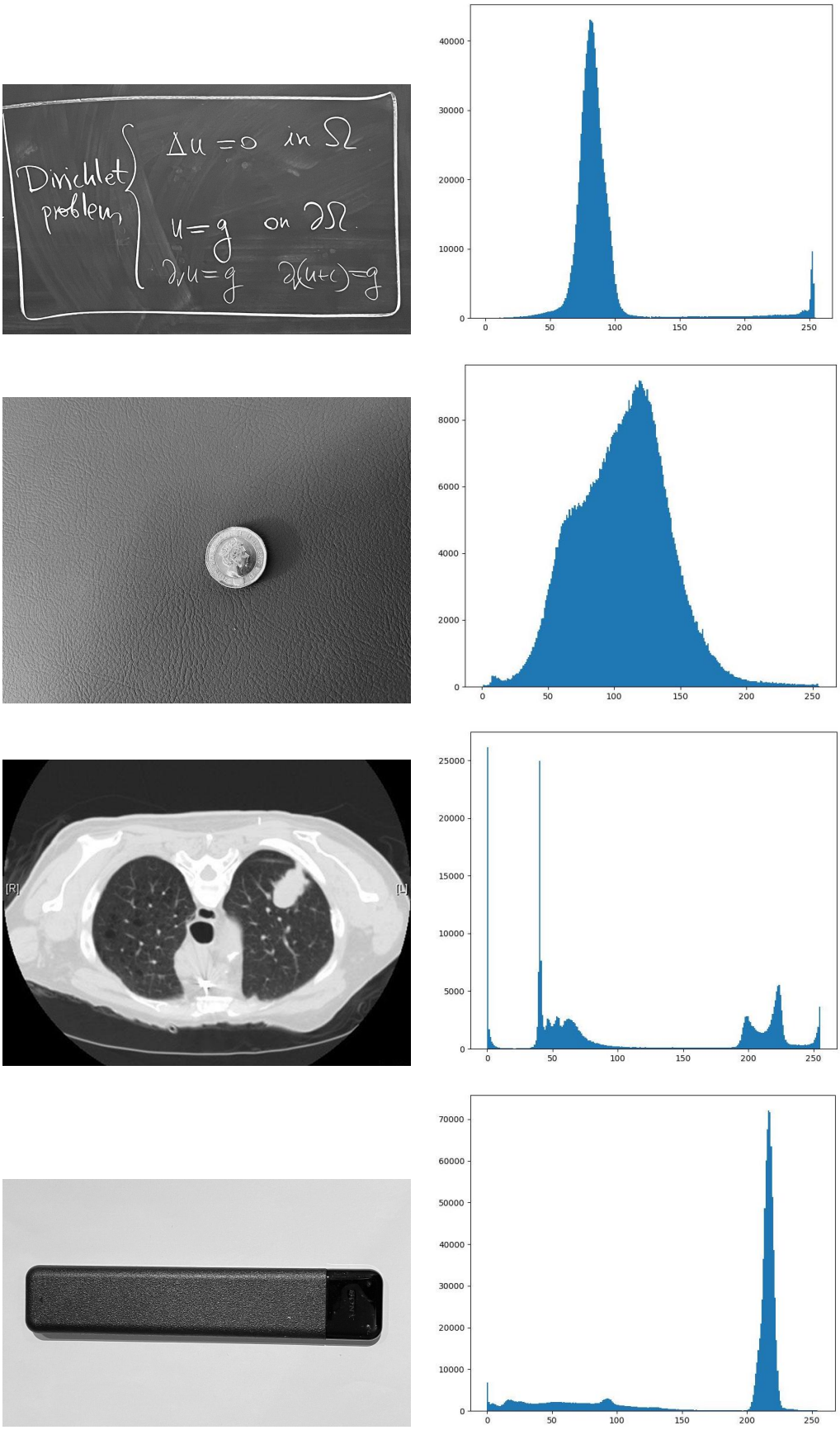


Figure 2.6: Grayscale images and their histograms (From top to bottom: Board, Coin, Lung, Remote)

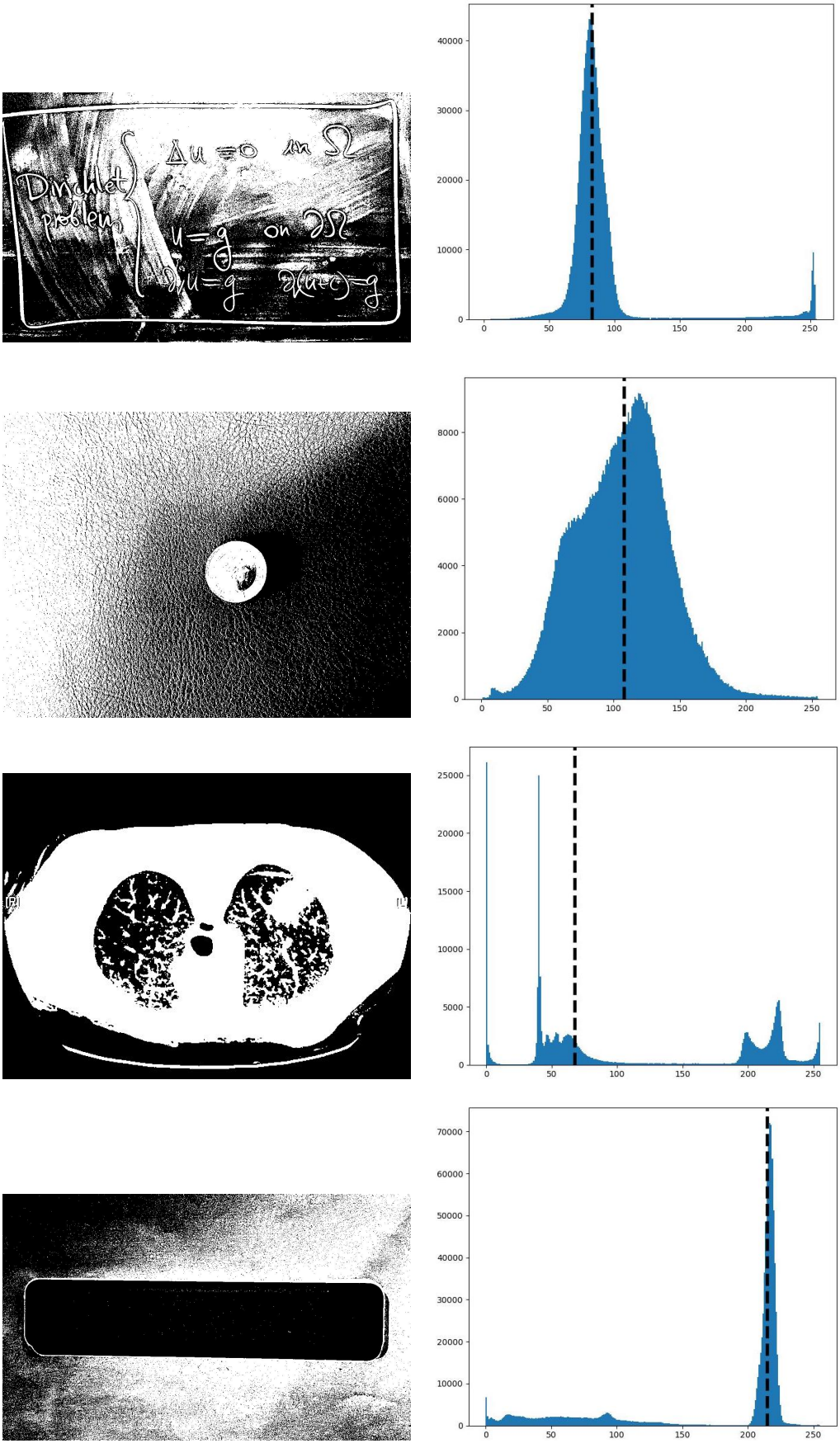


Figure 2.7: Results using median threshold and their histograms indicating the threshold value (From top to bottom: Board, Coin, Lung, Remote)

measures the distance between cluster means for the foreground and the background class, and the overall mean of the whole image. The third is total image variance $\sigma_{\mathbf{I}}^2$, which is a constant for an image I, given by $\sigma_{\mathbf{I}}^2 = \sigma_w(q)^2 + \sigma_b(q)^2$. The threshold q can be found by either minimizing $\sigma_w(q)^2$ or maximizing $\sigma_b(q)^2$ [Burger and Burge, 2016]. For our purposes, the threshold will be computed by maximizing $\sigma_b(q)^2$.
The expression for $\sigma_b(q)^2$ is:

$$\sigma_b(q)^2 = \frac{1}{(NM)^2} n_0(q) n_1(q) (\mu_0(q) - \mu_1(q))^2 \tag{2.5}$$

$$\text{where} \quad n_0(q) = \sum_{i=1}^{q} h(i), \; n_1(q) = \sum_{i=q+1}^{K-1} h(i),$$

$$\mu_0(q) = \frac{1}{n_0(q)} \sum_{i=1}^{q} ih(i), \; \mu_1(q) = \frac{1}{n_1(q)} \sum_{i=q+1}^{K-1} ih(i)$$

and $h(i)$ is the number of pixels with intensity i [Burger and Burge, 2016].
The threshold q that has the maximal $\sigma_b(q)^2$ will be computed by iterating through all $q \in [0, K-2]$. The value $q = K-1$ is omitted, since if $q = K-1$ the foreground set would be empty $\Rightarrow n_1 = 0 \Rightarrow \sigma_b(q)^2 = 0$ and lead to an error in the code (iterating from K to $K-1$). For visualisation, we provide examples with their respective histograms in Figure 2.8.
As shown from Figure 2.8, Otsu's method can effectively perform image segmentation when the intensity values for foreground and background of the image are well separated as illustrated in the histograms for all the images except the coin. Due to the complexity of the texture for the coin image, the intensity for the foreground and the background form one large cluster. Thus, parts of the background gets categorised as foreground and vice-versa. For this image, the sought after foreground is the coin, and the background is everything else.

2.2.4 Maximum Entropy

The *maximum entropy* method uses the histogram to find an optimum threshold, and does so by first normalizing the histogram and then using it to estimate the probability density function for the intensity levels of the image
[Burger and Burge, 2016]. For visualisation, we once again provide examples with their respective histograms in Figure 2.9.
Let $K-1$ be the maximum intensity of the image, in this case $K-1 = 255$. Furthemore, let $p(g)$, $g \in \{0, \ldots, K-1\}$ is the probability that any given pixel has intensity g. The vector $(p(0), p(1), \ldots, p(K-1))$ is the probability density function (PDF) of the image. Given a particular image with $N \times M$ pixels and $h(g)$ denoting the number of pixels with intensity g, $p(g) \approx \hat{\mathbf{p}}(g) = h(g)/(NM)$. The ratio $\hat{\mathbf{p}}(g)$ is thus the estimate of the image's PDF and clearly $0 \leq \hat{\mathbf{p}}(g) \leq 1$ with $\sum_{g=0}^{K-1} \hat{\mathbf{p}}(g) = 1$. Hence, the associated cumulative distribution function

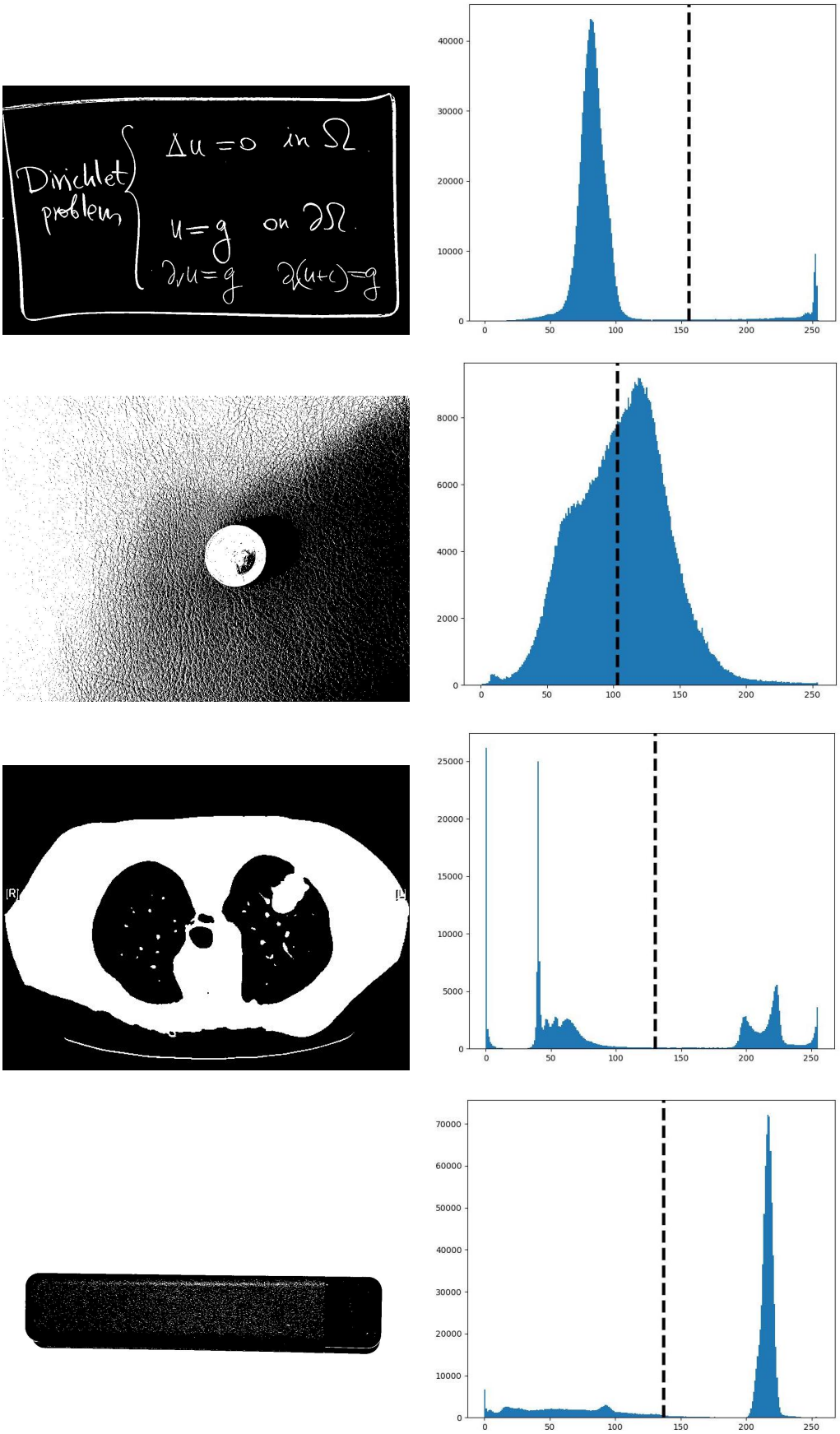


Figure 2.8: Results using Otsu threshold and their histograms indicating the threshold value (From top to bottom: Board, Coin, Lung, Remote)

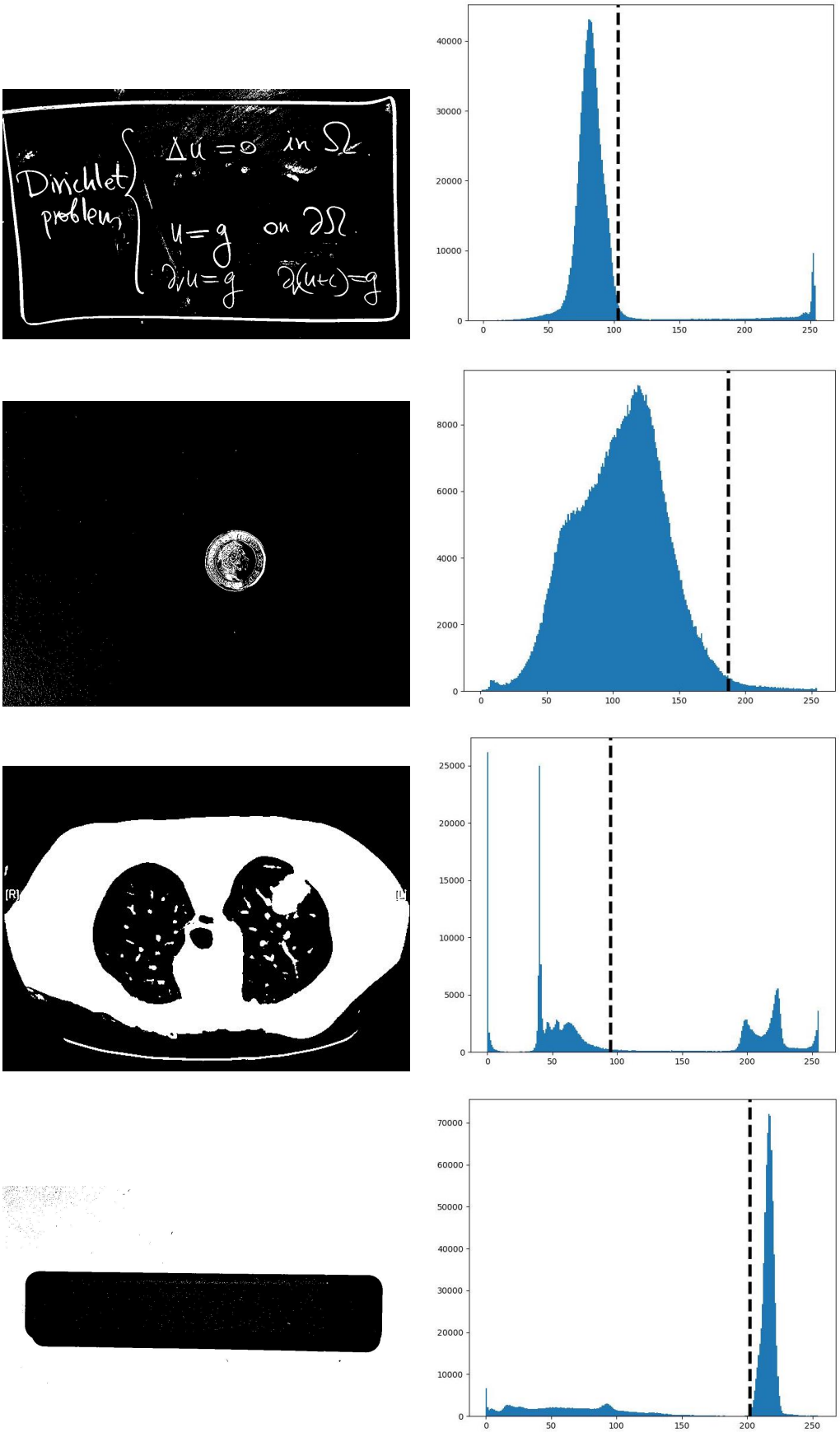


Figure 2.9: Results using maximum entropy threshold and their histograms indicating the threshold value (From top to bottom: Board, Coin, Lung, Remote)

(CDF) is $\hat{\mathbf{P}}(g) = \sum_{i=1}^{g} \mathbf{p}(i)$. The *entropy* of the image is defined as

$$H(Z) = -\sum_{g \in Z} \hat{\mathbf{p}}(g) \log_b(\hat{\mathbf{p}}(g))$$

where $Z = \{0, \ldots, K-1\}$ and $b > 1$. This measures the average amount of information (or uncertainty) contained in the image [Burger and Burge, 2016]. The goal of the maximum entropy threshold method is to find a threshold q such that the overall entropy of the image

$$H_{01}(q) = H_0(q) + H_1(q)$$

is maximized, whereby,

$$H_0(q) = -\sum_{i=0}^{q} \frac{\hat{\mathbf{p}}(i)}{\hat{\mathbf{P}}_0(q)} \log_b(\frac{\hat{\mathbf{p}}(i)}{\hat{\mathbf{P}}_0(q)}), \quad H_1(q) = -\sum_{i=q+1}^{K-1} \frac{\hat{\mathbf{p}}(i)}{\hat{\mathbf{P}}_1(q)} \log_b(\frac{\hat{\mathbf{p}}(i)}{\hat{\mathbf{P}}_1(q)})$$

$$\hat{\mathbf{P}}_0(q) = \sum_{i=1}^{q} \hat{\mathbf{p}}(i), \quad \hat{\mathbf{P}}_1(q) = \sum_{i=q+1}^{K-1} \hat{\mathbf{p}}(i).$$

$H_0(q)$, $H_1(q)$ represent the entropy of the background and foreground image (partitioned by threshold q), respectively. The ratios $\hat{\mathbf{p}}(i)/\hat{\mathbf{P}}_\mathbf{0}(q)$, $0 \leq i \leq q$, $\hat{\mathbf{p}}(i)/\hat{\mathbf{P}}_\mathbf{1}(q)$, $q+1 \leq i \leq K-1$ are the estimated PDFs of the background and foreground respectively. The term in the denominator ensures the sum of the PDFs over all possible i adds up to 1. The main idea behind this method is to partition the image into background and foreground where the information contained in the partitioned images are as large as possible. Generally speaking, an image that has a "flat" histogram (with intensity equally spread over a large range) will have higher entropy than an image that has a "peaked" histogram (with intensity condensed over a small range) [Burger and Burge, 2016].
We may compare the results of the maximum entropy with that of Otsu's method by observation of Figures 2.8 and 2.9. The maximum entropy method produces similar results except for the coin example. The entropy method focuses on identifying the background, while Otsu's method focuses on identifying the coin (the foreground). Looking at the histogram, Otsu's method draws the threshold line in the middle of the wide peak while the entropy method draws the line at the right hand side of the peak where it starts to flatten. As discussed previously, the entropy of a flat histogram is generally higher than a peaked histogram. The histogram segment to the right of the line is quite flat, resulting in a high entropy. Therefore, the entropy method is good at dealing with images that have a foreground (background) that covers a wide range of intensity values and has a relatively uniform distribution.

2.2.5 Sauvola Thresholding

All the thresholding methods discussed above are global thresholding methods, as these methods predict a single threshold q for the entire image. As illustrated

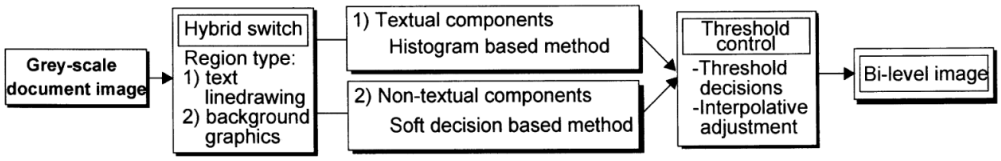


Figure 2.10: Overview of Sauvola thresholding method. From [Sauvola and Pietikäinen, 2000]

by the coin example, these methods do not perform well when the background of the image is textured.
One way to address this issue is to apply local thresholding methods to the image. Local thresholding methods sub-divide an image into regions, finding a threshold value for each region. One such method is the Sauvola Thresholding method. This method was designed for document analysis, specifically separating its contents (texts, diagrams, etc.) from the background and converting it to a binary image. Note that the background for documents written in dark ink on bright paper will correspond to the bright regions of the image. Unless specified, we will refer to the background as the part of the image that is below a given threshold from now on.
The main idea of Sauvola thresholding is to classify pixels into two categories, namely background (of document) and pictures (BP), and textual and line-drawing (TL). Then, a soft decision method (SDM) is applied to BP pixels and a specialized text binarization method (TBM) is applied to TL pixels. Finally, the outcomes of these methods will be combined into one single algorithm. The details of classifying pixels, SDM and TBM are discussed in-depth in the paper "Adaptive Document Image Binarization" [Sauvola and Pietikäinen, 2000]. Figure 2.10 is a flowchart that provides an overview of the method.
Given a greyscale image $\mathbf{I}$ we define square patches $P(x, y)$ with $n \times n$ pixels (n being odd) centered at pixel $(x, y) \in \mathbf{I}$. The threshold $T(x, y)$ of pixel (x, y) is then calculated by

$$T(x, y) = m(x, y)\left(1 + k\left(\frac{s(x, y)}{R} - 1\right)\right), \tag{2.6}$$

where $m(x, y)$, $s(x, y)$ are the mean and standard deviation of $P(x, y)$, $0 < k < 1$ is a user defined parameter. R is the dynamic range of $s(x, y)$ define by $(\max_{(x,y)\in\mathbf{I}} s(x, y) - \min_{(x,y)\in\mathbf{I}} s(x, y))$. This can be approximated by taking half the dynamic range of the image, which is 128 for an 8-bit image (the default value used in the function
`skimage.filters.threshold_sauvola` [ski, 2023]) [Sauvola and Pietikäinen, 2000].

Figure 2.11 shows the result of applying `skimage.filters.threshold_sauvola` to the coin image with different n, k values. As observed, increasing k will lead to more pixels being labelled as foreground. This can be explained by observing (2.6), in which increasing k increases the contribution of the $s(x, y)/R - 1$ term. By definition of R, $s(x, y)/R \leq 1 \Rightarrow (s(x, y)/R) - 1 \leq 0$. Thus increasing

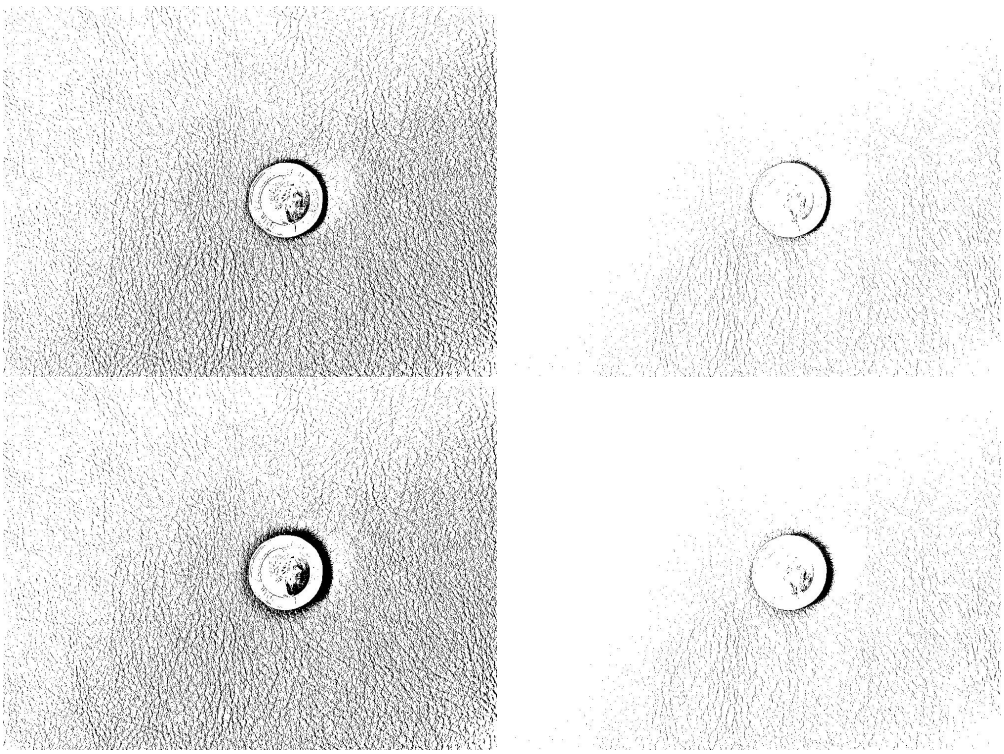


Figure 2.11: Results of Sauvola thresholding on the coin image, with $R = 128$, $n = 25, k = 0.2$ (top left), $n = 25, k = 0.5$ (top right), $n = 51, k = 0.2$ (bottom left), $n = 51, k = 0.5$ (bottom left).

k decreases $T(x, y)$, and lowering the value of $s(x, y)$ exacerbates this decrease. Therefore, increasing k lowers the thresholds of individual pixels leading to more pixels being labelled as foreground.
On the other hand, increasing n (increasing patch size around pixels) has a subtle effect on the result, whereby more pixels in the shadow area of the image get labelled as background while everything else remains relatively unchanged. This might be due to the existence of bright spots around some shadow pixels. Having a bigger $P(x, y)$ around these shadow pixels leads to an increase of the $m(x, y)$ value, thus increasing the threshold value (assuming there is no significant drop in $s(x, y)$). This results in more pixels being labelled as background.
Comparing the results from Sauvola thresholding with the results from the global thresholding methods, we observe that the Sauvola method focuses on picking up the textural details of the image while the global methods focus on identifying whole objects in the image.

2.3 Region Growing

One of the simplest, yet most widely used, algorithms in image analysis, particularly segmentation, is that of *region growing*. Under this approach, a region starts off as one or several pixels, "seeds", selected based on predefined criteria. Pixels adjacent to the region are added to it if they are deemed similar, based on some function, to those already in the region. If a pixel is added to a region, its

unclassified neighbouring pixels are in turn assessed for similarity and added to, or left out of, the region accordingly. This process continues iteratively until no more pixels can be added to a particular region. If there are pixels in the image that still do not belong to a region, seeds for a new region are chosen among them, and growing proceeds as above. Otherwise the algorithm halts. Pseudo-code for a simple region growing method will look like Algorithm 1.

Algorithm 1 Region Growing

Input: Image $\mathbf{I}$ of size $a \times b$, where each pixel $z \in \mathbf{I}$ takes RGB value, seed-pixels for first region $\{z_1, z_2, \ldots, z_m\} \subset \mathbf{I}$, function to find neighbours of pixel z: $nbr(z) = \{\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_j\} \subset \mathbf{I}$, function to check for similarity of pixel to a region $f(R_i, z) \in \mathbb{R}_+$, similarity threshold $s \in \mathbb{R}_+$
Output: Regions $\{R_1, R_2, \ldots, R_n\}$ satisfying $\bigcup_{i \leq n} R_i = I$, $\bigcap_{i \leq n} R_i = \emptyset$

1. $n = 1$
2. $R_1 = \{z_1, z_2, \ldots, z_m\}$
3. **while** $\exists z \in R_1$, such that for $\bar{z} \in nbr(z)$, $\bar{z} \notin R_1$, $f(R_1, \bar{z}) < s$ **do**
4. append $\bar{z}$ to R_1
5. **end while**
6. **while** $\exists z \in \mathbf{I}$ such that $z \notin R_i$ $\forall i \in \{1, \ldots, n\}$ **do**
7. $n = n + 1, R_n = \{z\}$
8. **while** $\exists z \in R_n$, such that for $\bar{z} \in nbr(z)$, $\bar{z} \notin R_n$, $f(R_n, \bar{z}) < s$ **do**
9. append $\bar{z}$ to R_n
10. **end while**
11. **end while**

Clearly, the region growing algorithm is heavily dependent on the definition of the similarity function f and the selection of seed pixels. Flexibility is also present in the definition of the function nbr for a given pixel $z \in \mathbf{I}$. A typical choice for the nbr, would be such that for a pixel $z \in \mathbf{I}$, $nbr(z)$ returns the 8 adjacent pixels to z [Gonzalez and Woods, 2014].

2.3.1 Assessing Similarity

The determining factor for the outcome of a region-growing algorithm is the means by which an unclassified pixel, $\bar{z}$, is deemed "similar" to an existing region R_n. This process is directly controlled by the function $f(R_n, \bar{z})$ and the constant $s \in \mathbb{R}_+$ appearing in the above pseudo-code. In a greyscale image, let $I(z)$ denote the intensity of a pixel z and $\bar{R}_n$ denote the average intensity of pixels in region R_n. For greyscale images, the simplest method lets

$$f(R_n, \bar{z}) = |\bar{R}_n - I(\bar{z})|,$$

and $\bar{z}$ is appended to R_n if $f(R_n, \bar{z}) < s$ for some threshold $s \in [0, 255]$ which is predetermined and context specific. Assessing similarity becomes more interesting when the distribution of a region is taken to be known. In this case, an

unclassified point $\bar{z}$ is appended to the region if it falls within some confidence interval as calculated using the sample mean and variance of the existing region at a predefined significance level α .

2.3.2 Preprocessing

Assessing every pixel in an image for similarity, potentially with respect to several regions, can be very computationally expensive. As an initial step for a region growing algorithm, thresholding is commonly used to classify relatively uniform pixels that may represent a background or a region within which segmentation is unnecessary. Possible methods by which to identify a threshold value are histogram analysis [Qi and Snyder, 1998], Otsu's method, median thresholding, maximum entropy, and Sauvola thresholding as discussed above.

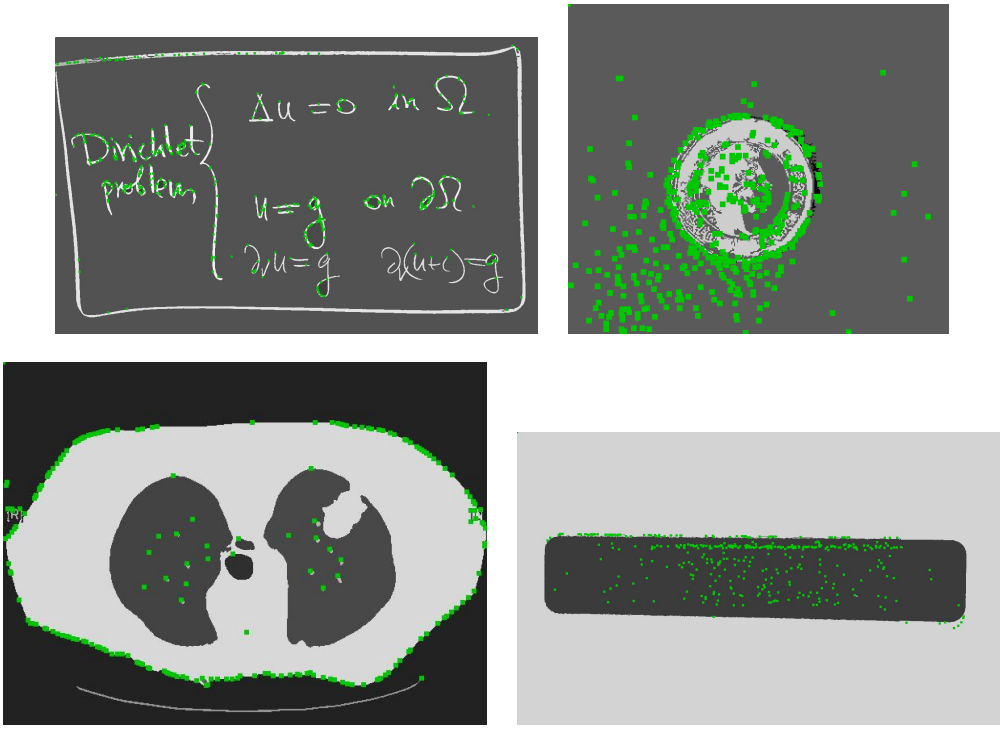


Figure 2.12: Results using simple region growing algorithm with similarity function $f(R_i, z) = |\bar{R}_i - p(z)|$ and threshold $s = 2\sigma_I$ (σ for the lung). Here, $\bar{R}_i$ denotes the average intensity of region i and σ_I denotes the standard deviation in pixel intensity for the image. Pixels are coloured by the average intensity of the region to which they belong. Green dots indicate seed pixels (From left to right: Board, Coin, Lung, Remote)

2.4 Clustering

In the context of image analysis, clustering algorithms seek to group pixels that are similar in terms of colour intensities and, possibly, location. For our discussion

of clustering algorithms, we will define the feature space of a greyscale image as the three-dimensional space indicating the x-index, y-index, and intensity of each pixel.

2.4.1 K-Means

The most well known form of clustering algorithm, known as *K-means clustering*, takes as input a given set of points (means) in the feature space of an image. This set of points serve as "seeds" that determine the number of clusters that will be identified. Each pixel in the image is assigned to its closest mean (seed) in terms of Euclidean distance [Snyder and Qi, 2017] upon initialisation. In an iterative process, each mean is then moved to the average position (in the feature space) among the pixels assigned to it. Pixels are then reassigned, as earlier, to the closest mean. This process is repeated until the Euclidean difference between all pairs of previous and current means are lower than some chozen threshold. A cluster corresponds to all pixels that have been assigned to a specific K-mean. Figure 2.13 shows the process by which K-means are updated and pixels reassigned for an image whose feature space has only two dimensions, α and β.

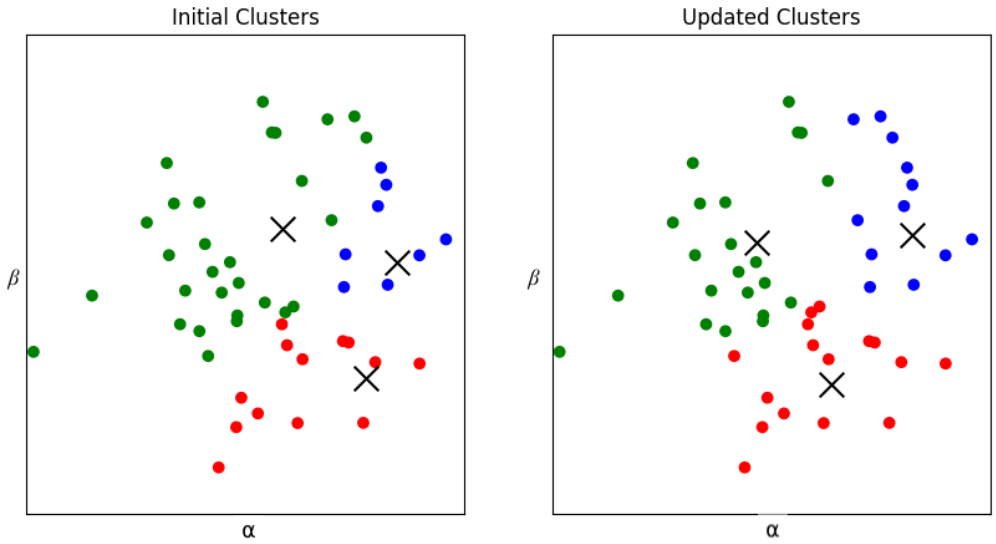


Figure 2.13: Left: Points in the plane colour coded according to which seed point (cluster) lies closest. Right: Updated location of clusters and corresponding change of colour codes

For the greyscale images we consider, we normalise the x and y coordinates of each pixel, in addition to its intensity before applying the K-means algorithm. Algorithm 2 provides the pseudo-code for the algorithm and its application to examples are shown in 2.14.

Algorithm 2 K-Means

Input: Image $\mathbf{I}$ of size $N \times M$, where each pixel $z \in \mathbf{I}$ takes greyscale value; seed-locations for K-means $\mathbf{C} = \{c_1, c_2, \ldots, c_m\}$, where each seed $\mathbf{c}_i = (x_i, y_i, \iota_i)^T \in [0,1]^3$ indicates a unit-scaled location, (x_i, y_j) and intensity, ι_i; vectors $\mathbf{v}_z = (x_z, y_z, \iota_z)^T$ containing unit-scaled values for the location-indices and intensity of each pixel z.

Output: Clusters $\{K_1, K_2, \ldots, K_m\}$ satisfying $\bigcup_{i \leq m} K_i = I$, $\bigcap_{i \leq m} K_i = \emptyset$; corresponding cluster means $\{\mathbf{d}_1, \ldots, \mathbf{d}_m\}$

1. **for** $i \in \{1, 2, \ldots, m\}$ **do**
2. $\quad K_i = \{z \in \mathbf{I} : ||\mathbf{v}_z - \mathbf{c}_i||_2 \leq ||\mathbf{v}_z - \mathbf{c}_j||_2 \quad \forall j \in \{1, 2, \ldots, m\}\}$
3. $\quad \mathbf{d}_i = |K_i|^{-1} \sum_{z \in K_i} \mathbf{v}_z$
4. **end for**
5. $\mathbf{D} = (\mathbf{d}_1, \ldots, \mathbf{d}_m)$
6. **while** $\max_{i \in \{1,2,\ldots,m\}} \{||(\mathbf{D} - \mathbf{C})_i||_2\} > \varepsilon$ **do**
7. $\quad$ **for** $i \in \{1, 2, \ldots, m\}$ **do**
8. $\quad\quad K_i = \{z \in \mathbf{I} : ||\mathbf{v}_z - \mathbf{d}_i^T||_2 \leq ||\mathbf{v}_z - \mathbf{d}_j^T||_2 \quad \forall j \in \{1, 2, \ldots, m\}\}$
9. $\quad\quad \mathbf{c}_i = \mathbf{d}_i$
10. $\quad\quad \mathbf{d}_i = |K_i|^{-1} \sum_{z \in K_i} \mathbf{v}_z$
11. $\quad$ **end for**
12. **end while**

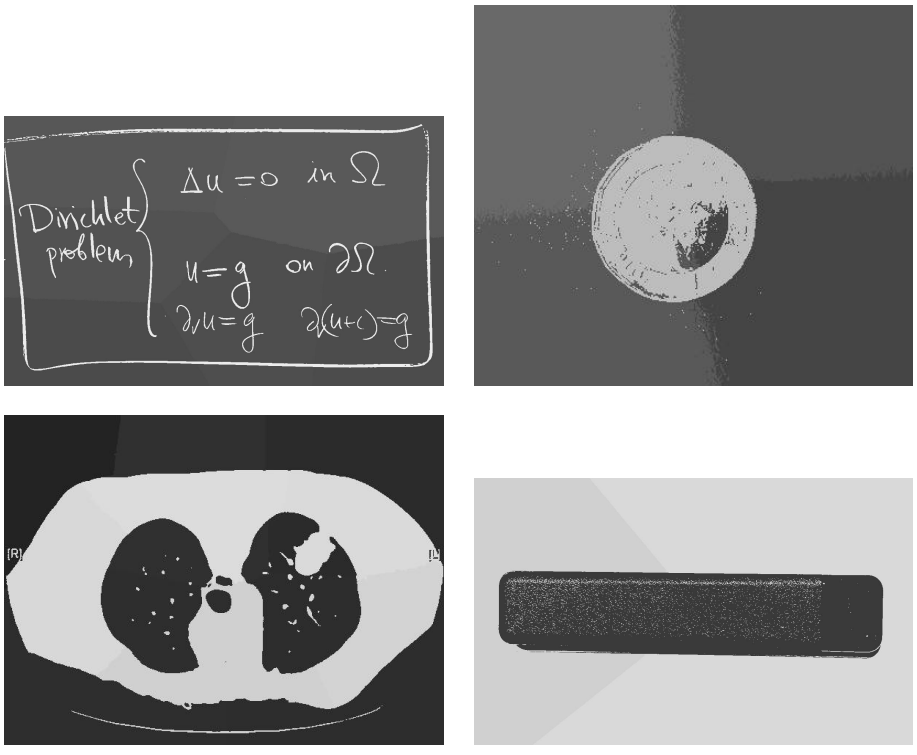


Figure 2.14: K-means clustering for example images with different predefined cluster numbers. Number of clusters for Board: 8, Coin: 5, Lung: 10, Remote: 4. The intensity of each region corresponds to the (scaled-up) intensity entry in the final region mean location $\mathbf{d}_i$. For all images we used a halting threshold $\varepsilon = \sqrt{0.001}$

2.4.2 Mean-Shift

Another common clustering algorithm is the *mean-shift* algorithm. In contrast to K-means, the mean-shift algorithm does not require a set of seed points as input. This is advantageous when the number of clusters in an image are unknown [Snyder and Qi, 2017]. On the other hand, the mean-shift algorithm does not inherently specify the cluster to which each datum belongs, possibly necessitating post-processing of the results. The mean-shift algorithm introduced here also uses a normalised feature space with x-index, y-index and intensity dimensions. Under the mean-shift algorithm, a centroid, $\boldsymbol{\phi}_z \in [0,1]^3$, is identified for each pixel $z \in \mathbf{I}$. The initial positions of the centroids are set to $\boldsymbol{\phi}_{z,0} = \mathbf{v}_z$, with $\mathbf{v}_z$ denoting the position of pixel z in the normalised feature space. We also define a kernel $h : [0,1]^3 \times [0,1]^3 \to \{0,1\}$, for which

$$h(\mathbf{v}_1, \mathbf{v}_2) = \begin{cases} 1 & \text{if } ||\mathbf{v}_1 - \mathbf{v}_2||_2 < d \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

where d is some predefined threshold. In other words, $\{\bar{z} \in \mathbf{I}\text{:} \quad h(\boldsymbol{\phi}_{z,n}, \mathbf{v}_{\bar{z}}) > 0\}$ is the set of pixels whose position in the feature space lies within distance d of centroid $\boldsymbol{\phi}_{z,n}$. The threshold d is called the bandwidth. For a pixel $z \in \mathbf{I}$, the position of centroid $\boldsymbol{\phi}_{z,n}$ is updated by taking a weighted mean of all $\mathbf{v}_{\bar{z}}$ for which $h(\boldsymbol{\phi}_{z,n}, \mathbf{v}_{\bar{z}}) > 1$. We may write this as

$$\boldsymbol{\phi}_{z,n+1} = \sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,n}, \mathbf{v}_z)\mathbf{v}_z \Big/ \sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,n}, \mathbf{v}_z) \tag{2.8}$$

For each centroid $\boldsymbol{\phi}_{z,n}$, this process is repeated until movements between iterations are small, i.e. $||\boldsymbol{\phi}_{z,n} - \boldsymbol{\phi}_{z,n+1}||_2 < \varepsilon$ for some threshold ε. The crux is that one may assign to the same cluster K_i all pixels whose final centroid locations, $\boldsymbol{\phi}_z$, are equal or similar to within some threshold γ. However, this must be performed in an informed manner during post-processing. Algorithm 3 demonstrates a full implementation of the mean-shift algorithm.
In Algorithm 3, the function h is what is called a flat kernel, assigning equal weight to pixel $\bar{z}$ for which $||\boldsymbol{\phi}_{z,n} - \mathbf{v}_z||_2 < d$. Other possibilities include kernels that are weighted according to, for example, the Euclidean distance between pixels. Popular choices include a Gaussian kernel such that $\hat{h}(\boldsymbol{\phi}_{z,n}, \bar{z}) \propto \exp\left(-((x_z - x_{\bar{z}})^2 + (y_z - y_{\bar{z}})^2)/2\right)$, where x_z, y_z denote the x- and y-indices of pixel z in image $\mathbf{I}$.

While Algorithm 3 may succeed in finding the approximate location of the cluster to which each pixel $z \in \mathbf{I}$ belongs, it is extremely expensive computationally. In an image, a single pixel, z, may bear resemblance to thousands of other pixels, $\bar{z}$, such that $h(z, \bar{z}) = 1$. Consequently, the cost of step 7 in Algorithm 3 may increase dramatically if the kernel h is too wide. Conversely, if h is small, we are at risk of assessing z against too few other pixels. The remedy is to define some threshold, $\rho < d$, such that if $||\boldsymbol{\psi}_{z,n} - \mathbf{v}_{\bar{z}}|| < \rho$, we assign z and $\bar{z}$ to the same cluster, K_i [Comaniciu and Meer, 2002]. We shall say that $\bar{z}$ lies in the same *basin of*

Algorithm 3 Mean-Shift

Input: Image $\mathbf{I}$ of size $a \times b$, where each pixel $z \in \mathbf{I}$ takes greyscale value; vectors $\mathbf{v}_z = (x_z, y_z, \iota_z)^T$ containing unit-scaled values for the location-indices and intensity of each pixel z; function h taking centroid inputs $\mathbf{v}_1, \mathbf{v}_2 \in [0,1]^3$ for which

$$h(\mathbf{v}_1, \mathbf{v}_2) = \begin{cases} 1 & \text{if } ||\mathbf{v}_1 - \mathbf{v}_2||_2 < d \\ 0 & \text{otherwise} \end{cases}$$

for some predefined bandwidth $d < \sqrt{3}$; difference-threshold $\varepsilon < 1$
Output: final centroids $\boldsymbol{\phi}_z$ for each pixel $z \in \mathbf{I}$

1. **for** $z \in \mathbf{I}$ **do**
2. n = 0
3. $\boldsymbol{\phi}_{z,0} = \mathbf{v}_z$
4. $\boldsymbol{\phi}_{z,1} = (\sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,0}, \bar{z})\mathbf{v}_{\bar{z}})/(\sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,0}, \bar{z}))$
5. **while** $||\boldsymbol{\phi}_{z,n+1} - \boldsymbol{\phi}_{z,n}||_2 > \varepsilon$ **do**
6. n= n+1
7. $\boldsymbol{\phi}_{z,n+1} = (\sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,n}, \bar{z})\mathbf{v}_{\bar{z}})/(\sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,n}, \bar{z}))$
8. **end while**
9. **end for**

attraction as z. As the centroid, $\boldsymbol{\psi}_{z,n}$, whose initial position is $\boldsymbol{\phi}_{z,0} = \mathbf{v}_z$, moves around the feature space, it will encounter many such pixels, drastically reducing the number of pixels from which we have to initiate the for-loop in Algorithm 3. Clearly, a drawback is that some pixels may be assigned to the wrong cluster. However, this alteration is necessary to reasonably limit computation times. Algorithm 4 shows how this methodology is implemented in practice. Figure 2.15 demonstrates the process over two iterations of the updated mean-shift algorithm on an image with a single normalised colour channel giving the greyscale intensity. The other dimensions correspond to the normalised indices in the x and y directions. For some pixel z, we let $\boldsymbol{\phi}_{z,0} = \mathbf{v}_z$, the location of the sphere centres (green dot) in the left panel of Figure 2.15. A new centroid location, $\boldsymbol{\phi}_{z,1}$ is calculated as the mean of all points within the grey sphere of radius d (green, black and red points). All the (red or green) points within the red sphere of radius ρ are assigned to the same cluster as z. In the next iteration, the spheres will be relocated to have centre at $\boldsymbol{\phi}_{z,1}$, as in the right panel of Figure 2.15. The mean of all points within the outer sphere (black and red) is calculated to be at $\boldsymbol{\phi}_{z,2}$. All points within the red sphere around $\boldsymbol{\phi}_{z,1}$ are assigned to the same cluster as z. The process is repeated until $||\boldsymbol{\phi}_{z,n+1} - \boldsymbol{\phi}_{z,n}|| < \varepsilon$. The algorithm then finds a new, unassigned pixel $\hat{z}$ from which to repeat the procedure. The algorithm halts when all pixels have been considered at least once, either as initiation point for a centroid or having fallen into the basin of attraction for another pixel.

As before, for two centroids with nearly coincident final positions, the corre-

sponding pixel clusters K_i and K_j can be merged to a single cluster K. The merging of clusters can be performed by means of a density gradient approach based on the final locations of the centroids [Comaniciu and Meer, 2002]. Consider the case of a pixel $\bar{z}$ lying in the basin of attraction for two pixels z_1 and z_2, for which the centroids converge to entirely different locations in the feature space. By the above, $\bar{z}$ would be assigned to two different clusters K_α and K_β. However, we have designed Algorithm 4, such that $\bar{z}$ will simply be assigned to the same cluster as whichever pixel, z_1 or z_2, was considered first. The consequence is that the outcome is dependent on the pixels from which initiation is done. This issue could be resolved by assigning $\bar{z}$ to whichever pixel z_1 or z_2 lies closest to it in the normalised feature space. However, evaluating this distance for each such pixel $\bar{z}$ would increase computational cost, and so we have decided not to include this step in Algorithm 4.

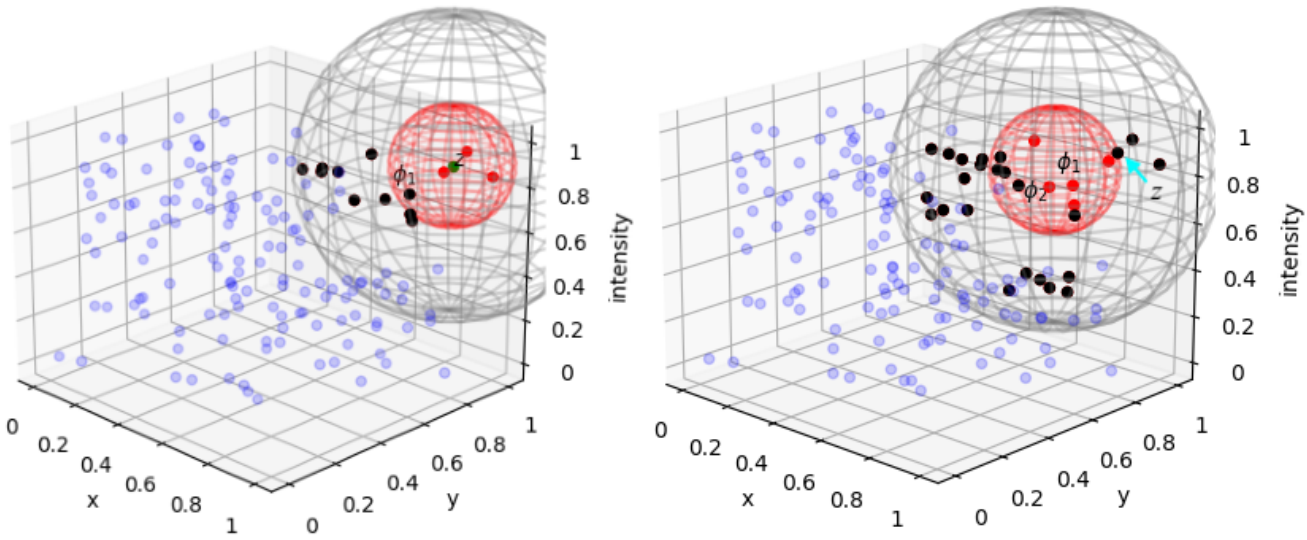


Figure 2.15: Left: First iteration of mean-shift algorithm. Position of pixel z in feature space is indicated by green dot. $\boldsymbol{\phi}_1$ is calculated as mean of all points within grey sphere of radius γ. All (red) points within red sphere will be assigned to same cluster as z. Right: Second iteration of mean-shift algorithm. Spheres are centred at $\boldsymbol{\phi}_1$. $\boldsymbol{\phi}_2$ is calculated as mean of all black and red points. Red points (within red sphere) are assigned to same cluster as z.

Algorithm 4 Mean-Shift Updated

Input: Image $\mathbf{I}$ of size $a \times b$, where each pixel $z \in \mathbf{I}$ takes greyscale value; vectors $\mathbf{v}_z = (x_z, y_z, \iota_z)^T$ containing unit-scaled values for the location-indices and intensity of each pixel z (location is feature space); Kernel, h, taking inputs $\mathbf{v}_1, \mathbf{v}_2 \in [0,1]^3$ for which

$$h(\mathbf{v}_1, \mathbf{v}_2) = \begin{cases} 1 & \text{if } ||\mathbf{v}_1 - \mathbf{v}_2||_2 < d \\ 0 & \text{otherwise} \end{cases}$$

for some predefined threshold $d < \sqrt{3}$; Second kernel, s, taking pixel inputs $\mathbf{v} \in [0,1]^3$ and $\bar{z} \in \mathbf{I}$ for which

$$s(\mathbf{v}, \bar{z}) = \begin{cases} 1 & \text{if } ||\mathbf{v} - \mathbf{v}_{\bar{z}}||_2 < \rho \\ 0 & \text{otherwise} \end{cases}$$

for some predefined threshold $\rho < d$; difference-threshold $\varepsilon < 1$

Output: Clusters $\{K_1, K_2, \ldots, K_m\}$ satisfying $\bigcup_{i \leq m} K_i = I$; $\bigcap_{i \leq m} K_i = \emptyset$; Final centroid locations $\{\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_m\}$ corresponding to each cluster.

1. $P = \{z | z \in \mathbf{I}\}$
2. **while** $|P| \neq 0$ **do**
3. n = 0
4. $z = P_0$
5. $\boldsymbol{\phi}_{z,0} = \mathbf{v}_z$
6. $\boldsymbol{\phi}_{z,1} = \sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,0}, \mathbf{v}_{\bar{z}}) \boldsymbol{\phi}_{z,0} / \sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,0}, \mathbf{v}_{\bar{z}})$
7. $P = P \setminus \{\bar{z} \in \mathbf{I} | s(\mathbf{v}_z, \bar{z}) = 1\}$
8. **while** $||\boldsymbol{\phi}_{z,n+1} - \boldsymbol{\phi}_{z,n}||_2 > \varepsilon$ **do**
9. n = n+1
10. $\boldsymbol{\phi}_{z,n+1} = \sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,n}, \mathbf{v}_{\bar{z}}) \boldsymbol{\phi}_{z,n} / \sum_{\bar{z} \in \mathbf{I}} h(\boldsymbol{\phi}_{z,n}, \mathbf{v}_{\bar{z}})$
11. $P = P \setminus \{\bar{z} \in \mathbf{I} | s(\mathbf{v}_z, \bar{z}) = 1\}$
12. **end while**
13. **end while**

Figure 2.16 demonstrates the mean-shift algorithm applied to the examples seen earlier. The corresponding bandwidth values, d, and ρ are indicated. Without merging any centroids, the resulting number of clusters (centroids) was 58 for the Board, 83 for the coin, 35 for the lung and 6 for the remote. Several of these centroids converged to points in the normalised feature space with similar intensity values however. After scaling up these intensity values and rounding to the nearest integer, the centroids for the board, coin, lung, and ranged over 19, 22, 15, and 3 different intensity levels, respectively. The processed images given in 2.16 assign to each pixel this rounded and scaled intensity.

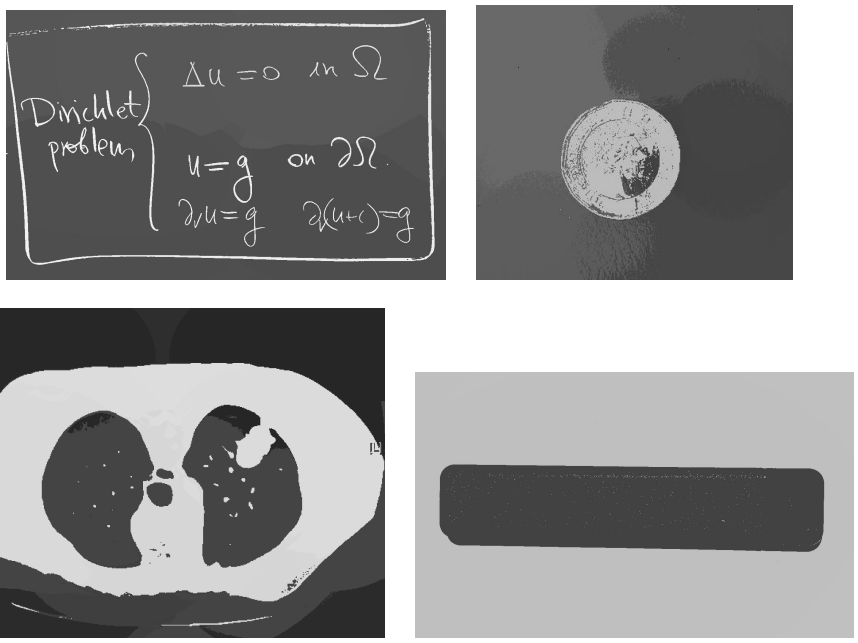


Figure 2.16: Mean-shift clustering for example images with different parameters γ and ρ defined on the normalised feature space. The intensity of each pixel corresponds to the intensity of the (scaled up) corresponding centroid. Board: $d = 0.1, \rho = 0.05$; Coin: $d = 0.1, \rho = 0.03$; Lung: $d = 0.1, \rho = 0.05$, Remote: $d = 0.4, \rho = 0.3$. For all images, we used a halting threshold $\varepsilon = 0.01$

2.5 Binary Operations

The binary mask generated by thresholding are sometimes poor in quality and cannot be used for some applications without post-processing. In this section we will discuss several image processing methods that are commonly used on binary images. These methods in general also work on greyscale/colour images but we will only be focusing on their application to binary images in this section.

2.5.1 Erosion and Dilation

Let $\mathbf{I}$ be a 1-bit binary image with intensity $I(x, y)$ for $(x, y) \in \mathbf{I}$ ($I(x, y) \in \{0, 1\}$). Let $\mathbf{I}_{er}$ and $\mathbf{I}_{di}$ be the resulting image after preforming erosion and dilation to $\mathbf{I}$, with intensity $I_{er}(x, y)$ and $I_{di}(x, y)$ respectively ($\mathbf{I}$, $\mathbf{I}_{er}$, $\mathbf{I}_{di}$ have same dimension). Let $P(x, y)$ be an arbitrary neighbourhood of (x, y), then

$$I_{er}(x, y) = \min_{(u,v)\in P(x,y)} I(u, v), \qquad I_{di}(x, y) = \max_{(u,v)\in P(x,y)} I(u, v). \tag{2.9}$$

In other words, erosion will set a pixel (x, y) to black if there is a black pixel in the neighbourhood $P(x, y)$, dilation will set a pixel (x, y) to white if there is a white pixel in the neighbourhood $P(x, y)$. Common choices of $P(x, y)$ are $\overline{Sq}_n(x, y)$ odd square centred at (x, y) in $\mathbf{I}$, and $\overline{D}_r(x, y) = \{(u, v) \in \mathbf{I} \mid ||(u, v) - (x, y)||_2 \leq r\}$ a disc of radius r centred at (x, y) in $\mathbf{I}$.
Figure 2.17 shows the result of erosion and dilation on the mask of the coin

Figure 2.17: Erosion (left) and dilation (right) results of the coin image mask from Figure 2.8, with $P(x, y) = \overline{D}_3(x, y)$.

Figure 2.18: Opening (left) and closing (right) results of the coin image mask from Figure 2.8, with $P(x, y) = \overline{D}_3(x, y)$.

image from Figure 2.8. As we can see erosion and dilation essentially expand every black/white pixel to a disk of radius 3 and isolates the regions with pure white/black colour from the image. However in our case, since the original binary mask is very noisy, the results are also noisy. To address this issue, we introduce the opening and closing operations. Opening is performing erosion followed by dilation with the same $P(x, y)$. Analogously, closing is performing dilation followed by erosion with the same $P(x, y)$ [Burger and Burge, 2016].
Figure 2.18 shows the result of opening and closing on the result (coin image) from Figure 2.8, which is essentially performing dilation and erosion on the left and right image in Figure 2.17, respectively. As we can see the results of opening and closing are marginally cleaner than that of erosion and dilation. This is due to dilation clearing up black spots from erosion and erosion clearing up white spots from dilation. Hence, as their names suggest, opening "opens" up small black gaps between white objects, while closing "closes" up black gaps between white objects. In other words, if erosion/dilation leaves white/black spots, these spots will be expanded by the opposite operation as illustrated on the right image of 2.18.

Figure 2.19: Results of applying remove small objects (left) and holes (right) to images from Figure 2.18, with $A = 100$ and $c = 1$.

2.5.2 Remove Small Objects

Remove small objects is a function from the `skimage.morphology` class. Given a binary image $\mathbf{I}$, the function removes white pixel clusters with area less than A, which is a predefined value. Let $P_c(p)$ denote the neighbourhood of a pixel $p \in \mathbf{I}$. We shall only consider values $c \in \{1, 2\}$, for which $P_1(p) = \overline{D}_1(p)$ (p and pixels adjacent to p) and $P_2(p) = \overline{Sq}_3(p)$ [ski, 2023]. A cluster K is a set of pixels within which each pixel is *connected* to at least one other pixel in the cluster. Pixels $p_1, p_2 \in \mathbf{I}$ are connected if $p_2 \in P_c(p_1)$ or equivalently $p_1 \in P_c(p_2)$. *Remove small holes* is a function from the same class that removes black pixel clusters using a method analogous to *remove small objects* [ski, 2023].
Figure 2.19 shows the result of the *remove small objects* and *remove small holes* functions acting on the images in Figure 2.18. As we can see from the left image, small white spots are removed by *remove small objects* while *remove small holes* removes small black spots. Both functions failed to remove larger spots, usually in the form of union of several small spots. This can be addressed by increasing A but at the risk of removing actual desired features.

2.6 Boundary Identification

Algorithms such as region growing and clustering output a label matrix $\mathbf{L}$. These methods assign pixels in image $\mathbf{I}$ to clusters $\{K_1, \ldots, K_n\}$. Let $L(p)$ be the label of pixel p in $\mathbf{L}$, $L(p) = i$ if $p \in K_i$. Binary masks, discussed in Section 2.2 and Section 2.5, are also a form of label matrix. A pixel is labelled 1 (or 255) if it is in the foreground and 0 if it is in the background, or vice-versa.
In this section we briefly introduce algorithms from the `skimage.segmentation` class [ski, 2023] that identify boundaries in label matrices for subsequent use in Chapter 4. The function `skimage.segmentation.find_boundaries` takes as input a label matrix $\mathbf{L}$, the connectivity parameter, c, as in 2.6, a marking mode and a background label. It outputs a label matrix $\mathbf{L}_b$ that has value 1 at boundaries and 0 otherwise. We will only be focusing on the "thick" marking mode where the inputted background label is irrelevant. The thick marking mode of the function identifies a pixel $p \in \mathbf{L}$ as a boundary pixel if there exists $p' \in P_c(p)$

s.t. $L(p') \neq L(p)$.
Furthermore, the function `skimage.segmentation.mark_boundaries` takes a label matrix **L** and an image **I** (usually the original image that is being segmented) with the same dimension, and identifies the boundaries using the aforementioned `find_boundaries` method and marks it on **I** with an user selected colour.

2.7 Edge Detection

Edge detection is concerned with determining the location and shape of edges, which demarcate borders between areas of different colours in an image. In the context of greyscale images, such borders occur between areas of different intensities. Intuitively, edges should correspond to areas where the change from one pixel to the next is large. Alternatively, if the transition between distinct segments of an image are gradual, we would be interested in the areas where this change is large. In this vein, we discuss two of the main underlying methodologies for edge detection: first-derivative methods and second-derivative methods.

2.7.1 First-Derivative Methods

The idea is to estimate the gradient at each pixel of an image in order to find sections where the intensity between pixels changes rapidly. For the pixel (x, y), a forward difference approximation to the gradient in the x-direction may be estimated by $I(x+1, y) - I(x, y)$. Similarly, a backward difference approximation may be calculated as $I(x, y) - I(x-1, y)$. We average the two to find a centred difference approximation in the x-direction of $\delta_{x,\cdot} I(x, y) = (I(x+1, y) - I(x-1, y))/2$. For an image, we are only interested in the relative gradient of pixels, and so the factor of two is usually disregarded. Furthermore, due to noise, the approximation $\delta_{x,\cdot}$ may not be a good representation of the gradient. We can reduce the effect of noise by considering the weighted difference approximations, in the x-direction, of pixels directly above and below (x, y). We assign a weight $w_x > 1$ to pixel (x, y), relative to those above and below. Again, as we are only interested in the relative gradient of pixels, we can omit the operation of averaging the difference operators and scaling for the weight. The resulting operator may then be written as

$$\hat{\delta}_x I(x, y) = w_x \delta_{x,\cdot} I(x+1, y) + \delta_{x,\cdot} I(x, y+1) + \delta_{x,\cdot} I(x, y-1). \tag{2.10}$$

Or equivalently,

$$\begin{aligned} \hat{\delta}_x I(x, y) = {} & w_x I(x+1, y) + I(x+1, y+1) + I(x+1, y-1) \\ & - w_x I(x-1, y) - I(x-1, y+1) - I(x-1, y-1). \end{aligned} \tag{2.11}$$

We see that the entire process may be encapsulated by a kernel containing the weights for each adjacent pixel in terms of the centred difference approximations:

-1	0	1
$-w_x$	0	w_x
-1	0	1

An entirely analogous kernel may be defined for approximation of the gradient in the y-direction, $\hat{\delta}_y(x, y)$, this time with the zeros along the middle row:

-1	$-w_y$	-1
0	0	0
1	w_y	1

Larger kernels, of size for example 5×5 or 7×7 are also possible when the plain difference approximation involves more pixels. We find the relative weights by considering one-dimensional Taylor series. For example by considering a function $f(x)$ evaluated at $x \pm \Delta x$ and $x \pm 2\Delta x$ and solving for $f'(x)$ yields

$$f'(x) \approx \frac{f(x + 2\Delta x) + 2f(x + \Delta x) - 2f(x - \Delta x) - f(x - 2\Delta x)}{8\Delta x} \tag{2.12}$$

Again ignoring the scaling factor, this translates to a 5×5 kernel in the x-direction for an image $\mathbf{I}$ of

-1	-2	0	1	2
$-\tilde{w}_x$	$-2\tilde{w}_x$	0	$\tilde{w}_x$	$2\tilde{w}_x$
$-w_x$	$-2w_x$	0	w_x	$2w_x$
$-\tilde{w}_x$	$-2\tilde{w}_x$	0	$\tilde{w}_x$	$2\tilde{w}_x$
-1	-2	0	1	2

A 5×5 kernel in the y-direction can be constructed in an analogous manner. Note that an extra weight $\tilde{w}_x$ must be added to the pixels directly above and below the relevant pixel (x, y) in order that these pixels be given higher importance than the pixels lying two rows above and below. Given the approximated gradient at each pixel

$$\nabla I(x, y) = \begin{bmatrix} \hat{\delta}_x I(x, y) \\ \hat{\delta}_y I(x, y) \end{bmatrix} \tag{2.13}$$

we may compute the corresponding magnitude, $||\nabla I(x, y)||$, and angle (with respect to the positive x-axis), $\phi(\nabla I(x, y))$, as

$$||\nabla I(x, y)|| = \sqrt{\hat{\delta}_x^2 I(x, y) + \hat{\delta}_y^2 I(x, y)}, \qquad \phi(\nabla I(x, y)) = \arctan \frac{\hat{\delta}_y I(x, y)}{\hat{\delta}_x I(x, y)} \tag{2.14}$$

The map $(x, y) \to ||\nabla I(x, y)||$ may now be represented as an image in which the edges are more brightly coloured than more uniform areas. Furthermore, the map $(x, y) \to \phi(\nabla I(x, y))$ represents the angle of most rapid change throughout the image.

We note that the essential choice within this methodology is that of picking the kernel weights, w_x and w_y. The Sobel Operator is a famous 3×3 operator for which $w_x = w_y = 2$ and whose kernel is given:

In most applications, an image first undergoes denoising, for example by means

$\partial/\partial x$

-1	0	1
-2	0	2
-1	0	1

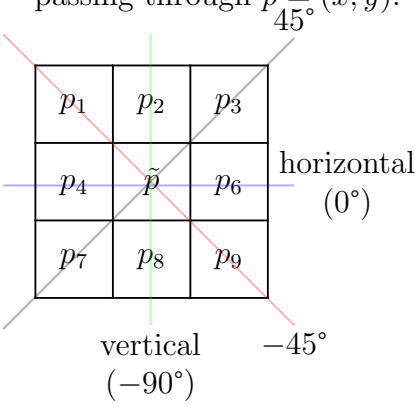
$\partial/\partial y$

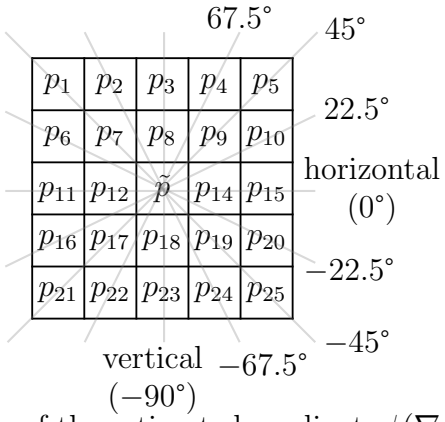
-1	-2	-1
0	0	0
1	2	1

of a Gaussian blur filter, as shown in 2.1, before being passed to a first-derivative method (such as the Sobel operator).

Canny Edge Detection

We notice that for an image I, the image corresponding to $||\nabla I(x, y)||$ will not make clear distinctions between edges and uniform areas. The Canny edge detection method, proposed by John Canny, is an extension of the general first derivative method, which seeks to address this issue. Our description of this algorithm will be based on that given in [Gonzalez and Woods, 2014]. After Gaussian blurring and calculating the gradient map, $\nabla I(x, y)$, by means of the Sobel operator, Canny's method proceeds to use a procedure known as *nonmaxima suppression*. Nonmaxima suppression seeks to reduce the thickness of edges to one pixel. Let $n \times n$ denote the size of the kernel used for the gradient estimation and let ν denote the number of weights in the outermost layer (i.e. 8 for a 3×3 kernel, 16 for a 5×5 kernel, etc.). For some pixel (x, y), consider then the $n \times n$ array of pixels centred at (x, y). As in the Figure below, we may define $\nu/2$ directions of an edge, each direction passing through opposite pixels in the outermost layer of the $n \times n$ array surrounding (x, y). These are the *possible* directions of an edge passing through $\tilde{p} = (x, y)$.





Note that for some pixel (x, y), the angle of the estimated gradient, $\phi(\nabla I(x, y))$, indicates the direction of greatest change in terms of pixel intensity. In other words, supposing there is an edge at (x, y), $\phi(\nabla I(x, y))$ represents the direction perpendicular to that edge. If we round $\phi(\nabla I(x, y))$ (up or down) to the closest of the $\nu/2$ directions as defined above, we can identify the pixels whose intensity should differ most from the intensity at the edge.

For example, say we are using a 3×3 kernel and that the angle of the gradient at some pixel (x, y) is $\phi(\nabla I(x, y)) = 19°$. Among the $\nu/2 = 4$ directions defined on the 3×3 array centred at (x, y), the horizontal one has the closest angle (0°). Along this direction we find the pixels $(x - 1, y)$, (x, y), and $(x + 1, y)$ (corresponding to p_4, $\tilde{p}$, and p_6 in the left panel above). Similarly for a 5×5 kernel,

with $\phi(\nabla I(x,y)) = 19°, 22.5°$ is the closest of the angles for any of the $\nu/2 = 8$ directions, as can be seen from the right panel above. Along this direction we find the pixels $(x-2,y-1),(x-1,y-1),(x-1,y),(x,y),(x+1,y),(x+1,y+1)$, and $(x+2,y+1)$ (corresponding to $p_{16}, p_{17}, p_{12}, \tilde{p}, p_{14}, p_9$, and p_{10} in the panel). In general we denote by $P(x,y)_n$ the set of pixels laying along the direction with angle closest to $\phi(\nabla I(x,y))$, when an $n \times n$ kernel is being used.
The Canny edge detection algorithm looks to identify edges whose width is one pixel, and so the assumption is made that (x,y) lies on an edge only if

$$||\nabla I(x,y)|| > \max_{(\hat{x},\hat{y})\in P'(x,y)_n} ||\nabla I(\hat{x},\hat{y})||, \quad \text{where } P'(x,y)_n := P(x,y)_n \backslash (x,y).$$

Accordingly, the algorithm defines a new array g, for which

$$g(x,y) = \begin{cases} ||\nabla I(x,y)|| & \text{if } ||\nabla I(x,y)|| > \max_{(\hat{x},\hat{y})\in P'(x,y)_n} ||\nabla I(\hat{x},\hat{y})|| \\ 0 & \text{else} \end{cases}$$

Applying this rule to each pixel (x,y) effectively limits the number of pixels potentially corresponding to an edge. The penultimate step in the Canny algorithm consists of thresholding to identify weak and strong edges, respectively. Given two thresholds U and L, with $U > L$, two new arrays, g_S and g_W, are constructed given the formulae

$$g_S = \begin{cases} g(x,y) & \text{if } g(x,y) > U \\ 0 & \text{else,} \end{cases} \qquad g_W = \begin{cases} g(x,y) & \text{if } L < g(x,y) \leq U \\ 0 & \text{else.} \end{cases}$$

Previous works suggest that the ratio U/L should lie somewhere in the range $[2,3]$ [Gonzalez and Woods, 2014]. The non-zero entries in g_S and g_W may now be interpreted as strong and weak pixels, respectively. It is mentioned that, due to noise or obstructions of the edges at hand, the edges present in the image corresponding to g_S will typically have missing pixels or gaps [Gonzalez and Woods, 2014]. In order to fill in the gaps, connectivity analysis is performed according to the below algorithm.
The output of the above connectivity analysis is an updated array g_S of strong edges. The non-zero entries in g_S are taken as pixels corresponding to edges in the original image I. We may summarise the Canny edge detection algorithm in five steps:

1. Smooth image to reduce noise using Gaussian convolution
2. Apply first Sobel operator to calculate $\nabla I(x,y)$ for each pixel $(x,y) \in I$
3. Nonmaxima suppression
4. Two-layer threshold to identify weak and strong edges
5. Connectivity Analysis to fill in gaps among strong edges

We note that the Canny edge detection algorithm relies on the choices of the first derivative Kernel, and its size $n \times n$, the thresholds U and L and the definition of a neighbourhood. The choice of Kernel size can have a large impact on the edges identified in the final array g_S. For a large Kernel, more potential edge pixels will

Algorithm 5 Connectivity Analysis

Input: $R_W = ((x, y) \in I : g_W(x, y) \neq 0)$ (set of remaining weak-edged pixels), $R_S = ((x, y) \in I : g_S(x, y) \neq 0)$ (set of strong-edged pixels), $nbr(x, y) = \overline{Sq}_m(x, y)$ an $m \times m$-square of centred at (x, y) in g_S ($m \in \mathbb{N}$ being odd).
Output: Updated g_S array.

1. Define set $\mathcal{R}_S := \bigcup_{r_s \in R_S} nbr(r_s)$
2. **while** $\mathcal{R}_S \cap R_W \neq \emptyset$
3. **for** $r_0 \in \mathcal{R}_S \cap R_W$
4. $g_S(r_0) = ||\nabla I(r_0)||$
5. $g_W(r_0) = 0$
6. Update sets R_S, R_W and $\mathcal{R}_S$ (by their definition)
7. **end while**
8. **for** $r_1 \in R_W$ (remaining weak-edged pixels)
9. $g_W(r_1) = 0$
10. Update R_W

be eliminated. Additionally, the choice of U and L will have a large impact on the number of pixels that are filtered out at the threshold stage. Finally, the size of the neighbourhood in the final connectivity analysis will determine the number of weak-edge pixels that end up being identified with an edge in the final output. For example, to ensure that the connectivity analysis only adds weak-edged pixels that are directly adjacent to a strong-edged one, an 8-connectivity neighbourhood must be chosen. Figure 2.20 shows the Canny edge-detection algorithm applied to the example images seen before. Each image was blurred with a Gaussian kernel of size 5×5 and $\sigma_x = \sigma_y = 1.4$ using openCV's `GaussianBlur` function. We then used openCv's `canny` function to apply the algorithm [its, 2023]. By default, the openCV function uses a Sobel operator of size 3×3. The caption indicates the lower, L, and upper, U, thresholds used to identify strong and weak edges. Note that the gradients calculated by openCV's function are not normalised by the weights in the Sobel operator, in line with what was mentioned above. The connectivity analysis is done with an 8-connectivity neighbourhood.

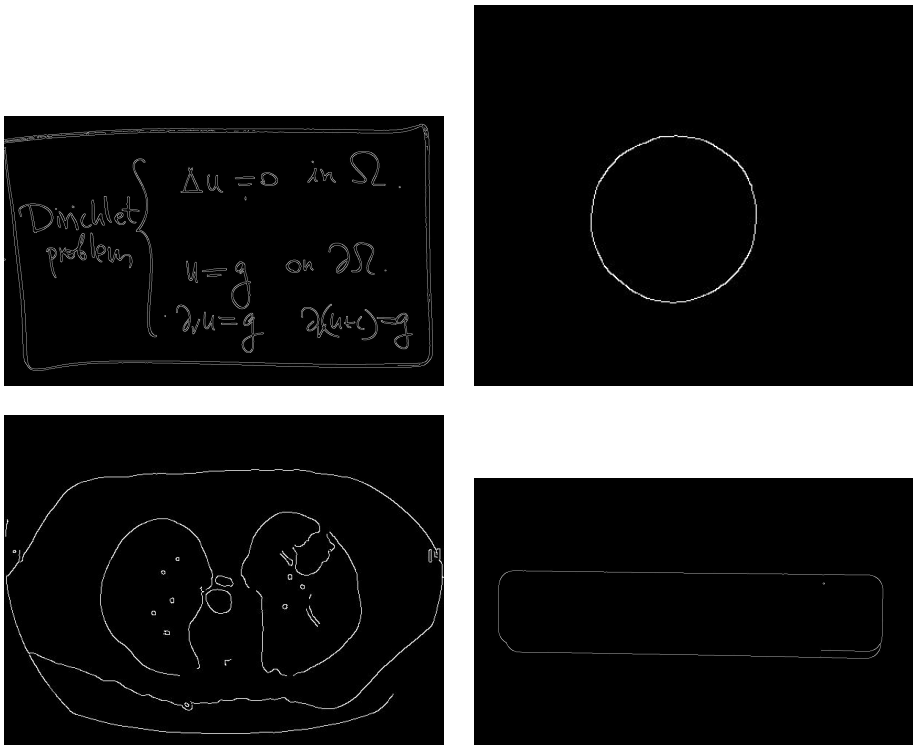


Figure 2.20: Canny edge detection algorithm applied to example images. The thresholds used for the weak, L and strong, U, edges are as follows: Board: $L = 250, U = 500$; Coin: $L = 250, U = 500$; Lung: $L = 150, U = 300$; Remote: $L = 250, U = 500$

2.7.2 Second-Derivative Methods

As mentioned earlier, we are sometimes interested in identifying edges associated with gradual changes in pixel intensity. As such, we are interested in areas where the estimated norm of the gradient is maximised. Maximising the gradient norm is a computationally expensive procedure, however, as it involves maximising a function in two variables. Furthermore, there is the question of the direction in which to find and maximise the gradient. The issue is solved by utilising the only orientation-independent second-order differential operator: the Laplacian [Marr and Hildreth, 1980]. In order to find points at which the gradient is large, we look for zero-crossings of the Laplacian, $\nabla^2 I = \partial_{xx} I + \partial_{yy} I$. A *zero crossing* is defined as two adjacent points (x, y) and (x', y') for which $\nabla^2 I(x, y) > 0$ and $\nabla^2 I(x', y') < 0$. We may interpret a greyscale image as a discrete representation of a two-dimensional scene in which the light intensity, I', varies continuously with coordinates x and y. Denote by l the line passing between the centres of pixels (x, y) and (x', y') as above. In the two-dimensional scene there must be a point along l for which $\nabla^2 I' = 0$. As always, the computation of partial derivatives of the light intensity in a scene must be computed in a discrete manner given the intensities of pixels in the image I. In the x-direction, an approximation to the second order derivative of the light intensity is found by taking the difference of

the forward and backward difference approximations as

$$\begin{aligned}\delta_{xx}I(x,y) &= I(x+1,y) - I(x,y) - (I(x,y) - I(x-1,y)) \\ &= I(x+1,y) - 2I(x,y) + I(x-1,y)\end{aligned} \tag{2.15}$$

This is called a centred, second order derivative approximation since we use both the forward and backward first order approximations. Combining the second order centred difference approximations in the x- and y-directions gives the kernel for the discrete Laplacian operator as

$$\nabla^2$$

0	1	0
1	−4	1
0	1	0

As above, the width of the Laplacian kernel may be increased at the cost of longer computational times. The Marr-Hildreth edge detection algorithm works by first applying a smoothing filter, such as a Gaussian blur, followed by the Laplacian operator, to each pixel in an image. In combination, these operators are called a Laplacian of Gaussian (LoG). Applying this operator to an image $\mathbf{I}$, we shall denote the resulting image by $\text{LoG}(\mathbf{I})$. Zero-crossings are then identified as the positive pixels $(x,y) \in \text{LoG}(\mathbf{I})$ for which an adjacent pixel is negative. The resulting image, with zero-crossings represented as non-zero pixels, will typically contain many more zero crossings than we are interested in. Therefore, it is suggested in [Gonzalez and Woods, 2014], one should keep only zero-crossings (x,y) for which $\text{LoG}(\mathbf{I}) > T$, where T is some threshold. Figure 2.21 shows this methodology applied to the example images seen earlier. For both the Laplacian (shown below) and Gaussian filter, we have used a Kernel of size 5×5 and $\sigma = 1.4$ in the Gaussian filter. This was done using the `cv2.GaussianBlur` and `skimage.filters.laplace` functions from [its, 2023] and [ski, 2023], respectively. Zero-crossings were identified by means of the simple code found at [a20, 2022]. The threshold used for each image, to filter out weak edges, is given in the caption.

$$\nabla^2$$

0	0	1	0	0
0	1	2	1	0
1	2	−16	2	1
0	1	2	1	0
0	0	1	0	0

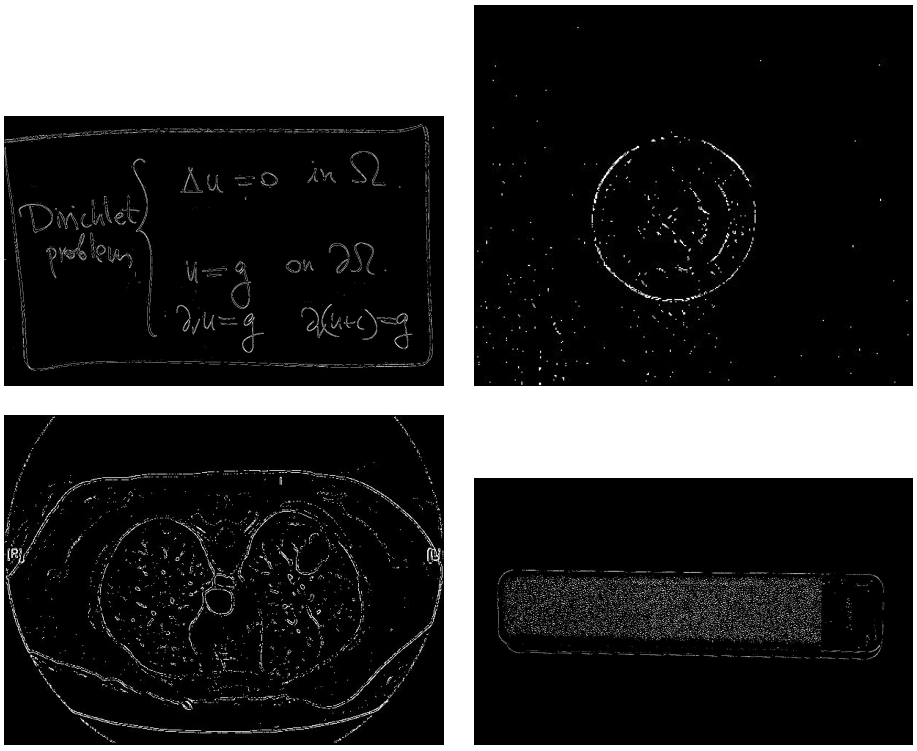


Figure 2.21: Marr-Hildreth edge detection applied to example images. Thresholds T used: Board: $T = 0.1$; Coin: $T = 0.3$; Lung: $T = 0.1$, Remote: $T = 0.1$

Chapter 3

Machine Learning Methods

3.1 Neural Networks

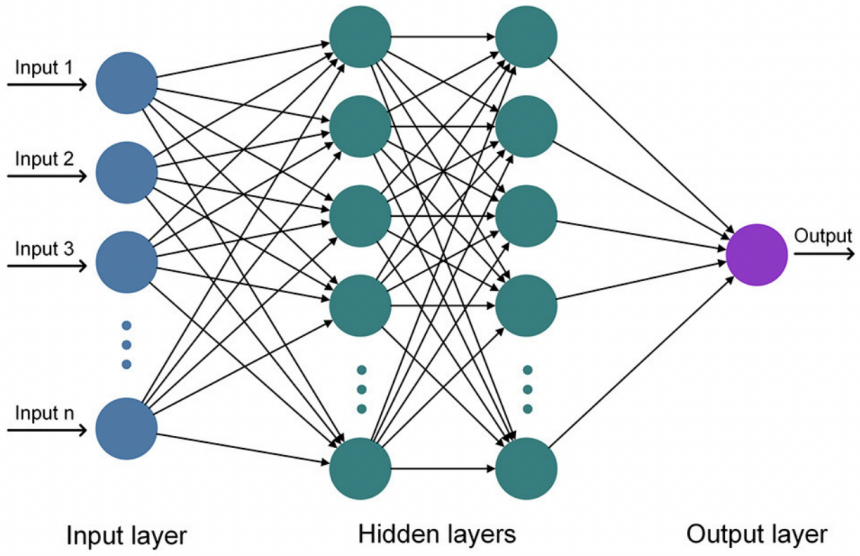


Figure 3.1: Schematic diagram of an ANN with 2 hidden layers. From [Sahraei et al., 2021].

As a family of modern machine learning methods, artificial neural networks (ANNs) were first implemented in 1971 by Ivakhnenko in his seminal paper "Polynomial theory of complex systems" [Ivakhnenko, 1971]. Inspired by information processing in biological systems, modern ANNs are structured layers of "neurons" where each neuron has its own set of weights which it uses to create linear combinations of inputs from upstream in the network. The result is then passed through a non-linear activation function. Accordingly, we attain a model that learns non-linear associations inherent in data and generalises well.

3.1.1 Back-propagation

First introduced in [LeCun et al., 1989], back-propagation is a numerical method that facilitates "learning" of a neural network architecture following a "forward

pass" of data by allowing us to compute and minimise a loss/objective function with respect to the weight space of the neural network.
In linear regression problems with a mean-square-error loss, we are able to write down an expression for the weights $\hat{\boldsymbol{\theta}} = \text{argmin}_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$, given a training dataset $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$. However, in general, we cannot find closed-form expressions and we resort to optimisation algorithms to lower the value of the loss function to a small value as opposed to finding a global solution [Goodfellow et al., 2017].
A standard trick for overcoming large amounts of training data is to use optimisation algorithms, in which the update direction $\boldsymbol{\nabla}_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ is estimated on a subset of the training data.
While optimisation algorithms may vary, they all update the parameters based on estimates of the gradient $\nabla_{\theta} L(\boldsymbol{\theta})$ via the basic equation of the form

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}\right). \tag{3.1}$$

The parameter η here is a *learning rate* and the negative sign ensures we are stepping downhill in our parameter space in order to minimize the loss. We focus here on the computation of the gradient.
Fortunately, computing the gradient can be done efficiently via back-propagation, which is simply an implementation of the chain rule. Once we have done the derivation, it is clear that the value of the loss at a given neuron depends only on:

- (i) the value of the neuron weight at that position in the network after a "forward pass" of the data
- (ii) the value of the loss at the connected downstream neurons (which we compute via "back-propagation")

Therefore, each layer needs only the abilities to (1) pass information forward, (2) receive information from one layer upstream and (3) compute derivatives with respect to the parameters of that layer.
For illustration, let us consider an input $\mathbf{x}$ and M hidden layers $\left\{\mathbf{z}^k\right\}_{k=1}^{M}$. We first write our loss as $L(\boldsymbol{\theta}) = \sum_i L_i(\boldsymbol{\theta})$, separating out contributions from each observation in the training set. Consider first the derivative with respect to one of the parameters used to compute the output, β_j (this weight is associated with neuron j in the final hidden layer, K). The final output f, can be written as follows:

$$f(\mathbf{x}) = g_f\left(\beta_0 + \sum_{n=1}^{M_K} \beta_n z_n^K\right),$$

where g_f is a non-linear activation at the output layer $(K+1)$. By implementation of the chain rule, we have that:

$$\begin{aligned}\frac{\partial L_i}{\partial \beta_j} &= \frac{\partial L_i}{\partial f}\frac{\partial f}{\partial \beta_j} \\ &= \underbrace{\frac{\partial L_i}{\partial f} g'_f\left(\beta_0 + \sum_n \beta_n z_n^K\right)}_{\delta_i^f} z_j^K\left(\mathbf{x}_i\right).\end{aligned}$$

Note that the evaluation of the derivative requires knowledge of an error term δ_i^f evaluated at the output (or "loss" layer) and an evaluation of the state of neuron z_j^K from a forward pass through the network.
Consider now the derivative with respect to one of the parameters which are used to combine features in the final hidden layer $\mathbf{z}^K, \alpha_{jn}^K$ say. This is the weight which pre-multiplies the output of neuron z_n^{K-1} for evaluation of the value of neuron z_j^K. Again, via the chain rule we have

$$\begin{aligned}\frac{\partial L_i}{\partial \alpha_{jn}^K} &= \frac{\partial L_i}{\partial f}\frac{\partial f}{\partial z_j^K}\frac{\partial z_j^K}{\partial \alpha_{jn}^K} \\ &= \underbrace{\frac{\partial L_i}{\partial f}\frac{\partial f}{\partial z_j^K} g'_K\left(\alpha_{j0}^K + \sum_n \alpha_{jn}^K z_n^{K-1}\right)}_{\delta_{ji}^K} z_n^{K-1}\left(\mathbf{x}_i\right).\end{aligned}$$

So again, we may write this derivative as an error multiplying the state at the associated neuron. However, using the definition of f to evaluate the derivative $\partial f/\partial z_j^K$, we have

$$\delta_{ji}^K = \frac{\partial L_i}{\partial f} g'_f\left(\beta_0 + \sum_n \beta_n z_n^K\right)\beta_j g'_K\left(\alpha_{j0}^K + \sum_n \alpha_{jn}^K z_n^{K-1}\right)$$

or, using the definition of δ_i^f above,

$$\delta_{ji}^K = \delta_i^f \beta_j g'_K\left(\alpha_{j0}^K + \sum_n \alpha_{jn}^K z_n^{K-1}\right).$$

This is the back-propagation equation. In practice, computer programs use the method of *automatic differentiation* for computation of gradients, δ. Automatic differentiation exploits the fact that computer programs execute a sequence of elementary arithmetic operations, and with repeated application of the chain-rule, gradients can be computed to a very high level of accuracy. As such, all machine learning libraries implement automatic differentiation in the computation of gradients.

3.1.2 Optimisation Algorithms

As elucidated above, within the context of neural networks, optimisation algorithms are numerical methods that largely serve to optimise our loss/objective function with respect to the network's weight space using the gradients computed via back-propagation and their use is necessitated by inherent non-linearities in the network [Goodfellow et al., 2017]. By seeking convergence toward a minimum in our parameter space, the optimisation algorithm ultimately updates network weights in order to learn representations in the data.

Stochastic Gradient Descent

Stochastic Gradient Descent is a simple optimisation algorithm that differs from the basic equation, (3.1), by introducing a mini-batch in the computation of the gradient estimate.

Algorithm 6 Stochastic Gradient Descent

Require: Learning rate, η
Require: Initial parameter θ
$k \leftarrow 1$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient estimate: $g \leftarrow m^{-1} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
    Apply update: $\theta \leftarrow \theta - \eta_k g$
    $k \leftarrow k + 1$
end while

Adam

As machine learning tasks increased in complexity and difficulty, neural network architectures became deeper and more complex to allow for accurate pattern recognition and generalisation across tasks. In recognising the difficulties in selecting an appropriate learning rate parameter, η_k, for such large networks the "Adam" [Kingma and Ba, 2014] optimiser was introduced and is currently the de-facto default optimiser in most neural network architectures. Adam implements adaptive learning rate optimisation across mini-batches of data by computing first-order and second-order moments of the gradient by a weighted average and incorporating them into the gradient update. Furthermore, Adam iteratively implements bias correction of the first-order and second-order moments to correct for the fact that they are zero-initialised, resulting in an optimiser that, compared to other adaptive learning rate optimisers, retains relatively low bias in the early iterations of backpropagation.

We note that in 3.1.2, $\odot$ refers to the *Hadamard product*, which is element-wise multiplication of g on itself.

Algorithm 7 Adam

Require: Learning rate, η
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$.(Suggested defaults: 0.9 and 0.999, respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters θ
Initialize 1st and 2nd moment variables $s = 0$, $r = 0$
Initialize time step $t = 0$
while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient estimate: $g \leftarrow m^{-1} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1) g$
    Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$
    Correct bias in first moment: $\hat{s} \leftarrow s/(1 - \rho_1^t)$
    Correct bias in second moment: $\hat{r} \leftarrow r/(1 - \rho_2^t)$
    Compute update: $\Delta\theta = -(\eta \hat{s})/\sqrt{\hat{r} + \delta}$ (operations applied element-wise)
    Apply update: $\theta \leftarrow \theta + \Delta\theta$
end while

3.1.3 Convolutional Neural Networks

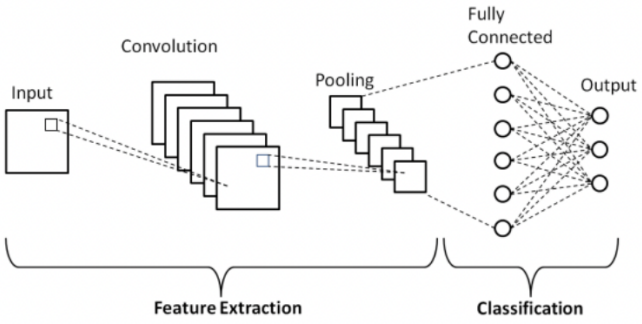


Figure 3.2: Schematic diagram of a simple CNN architecture. From [Phung and Rhee, 2019].

Convolutional neural networks (CNNs) are a paradigm-specific structure of modern neural networks used for processing data in a grid topology [Goodfellow et al., 2016]. Time-series data, for example, can be considered as a 1-dimensional grid of values across t. More pertinently, we can consider image data to be a 2-dimensional grid of values (i.e. pixels). While CNNs function in a largely similar manner to an ANN, the key operations in CNNs maintain the spatial features of the image input between layers allowing downstream layers to learn their respective weights based on a smaller local spatial region of the origi-

nal input image. In doing so, a CNN makes use of the following novel operations, which we shall describe below:

- Convolution
- Padding
- Pooling

Input dimensions

In Figure 3.2, we note the general structure of a CNN and denote all layers before the fully-connected layer as "spatial layers", which serve the purpose of extracting features from an input image. Analogous to ANNs, in a typical CNN structure, the first spatial layer is also known as the "input layer" while downstream spatial layers are known as "hidden layers". In the input layer, parameter values are decided by the nature of the input image and its total number of pixels. For example, for an RGB image we have an intensity of the three primary colors, corresponding to red, green, and blue, respectively. Therefore, if the spatial dimensions of an image are 128×128 pixels and the depth is 3 (corresponding to the RGB color channels), the overall number of pixels in the image is $128 \times 128 \times 3$. Conversely, for a greyscale image of identical spatial dimensions, the total number of pixels is $128 \times 128 \times 1$, where the pixel values are mapped to integers in the range $[0, 255]$ as described in Section 1.1. For the hidden layers, however, the input dimensions correspond to the dimensions of the output of the convolution operation from the previous layer as described in 3.1.3. For illustration, we take that the input in the q^{th} layer is of size $L_k \times B_k \times d_k$, where L_k refers to the height, B_k refers to the width and d_k is the depth. It is important to note that in nearly all image tasks, $L_k = B_k$, and using our RGB example above for our first layer, $L_1 = 128$, $B_1 = 128$ and $d_1 = 3$. For hidden layers ($k > 1$), the value of d_k is usually much larger than the number of input channels as the number of independent features in local regions of the original image extracted by upstream layers tends to be quite large. For $k > 1$, the inputs to the hidden layers can thus no longer be considered as raw pixels and are thus referred to as *feature maps* [Aggarwal, 2019] and are analogous to the hidden layer values in a typical ANN.

Filters and Convolutions

The model parameters in a CNN are structured as 3-dimensional units, called *filters* or *kernels*. These filters are square and odd-numbered in spatial dimension and tend to be significantly smaller than that of the hidden layer they are applied to. Again, for illustration, if we consider a filter applied at the q^{th} layer whose dimensions are: $F_q \times F_q \times d_q$. At each hidden layer, the convolution operation itself positions the filter at every possible location on the input image such that the filter overlaps the entire image, and at each position, performs a dot product between the $F_q \times F_q \times d_q$ filter parameter and the corresponding grid in the input to the hidden layer [Aggarwal, 2019]. Specifically, for a task involving RGB image data with 3 channels, if we denote the parameters of filter p in layer

q by the 3-dimensional tensor $\boldsymbol{W}^{p,q} = \left[w_{ijk}^{(p,q)}\right]$ and the feature maps in layer q by $\boldsymbol{M}^{(q)} = \left[m_{ijk}^{(q)}\right]$, the convolutional operation from the q$^{\text{th}}$ to (q+1)$^{\text{th}}$ layer is defined as:

$$m_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} m_{i+r-1,j+s-1,k}^{(q)} \quad \begin{array}{l} \forall i \in \{1, \ldots, L_q - F_q + 1\} \\ \forall j \in \{1, \ldots, B_q - F_q + 1\} \\ \forall p \in \{1, \ldots, d_{q+1}\} \end{array} \tag{3.2}$$

We first note that the input feature maps to layer q are of spatial dimension $L_q \times B_q$. Subsequently, when performing convolution from the q$^{\text{th}}$ to the (q+1)$^{\text{th}}$ layer, the operation results in a reduction in the spatial dimensions of the subsequent feature maps as a filter can only be applied at $L_q - F_1 + 1$ and $B_q - F_q + 1$ positions along the height and width dimensions. This subsequently determines the spatial dimensions of the input feature maps to the (q+1)$^{\text{th}}$ layer as: $(L_q - F_1 + 1) \times (B_q - F_q + 1)$, as noted in (3.2). Furthermore, the case where $q = 1$ corresponds to the input layer and $\boldsymbol{M}^{(1)}$ is the input image.

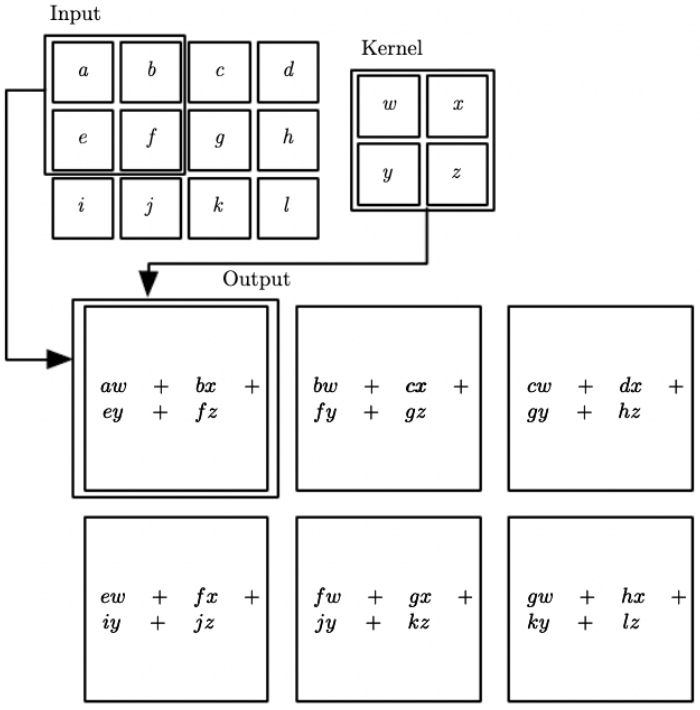


Figure 3.3: Example of a 2-dimensional convolution between a 2×2 filter and 3×4 input, resulting in a feature map of height $= 3 - 2 + 1 = 2$ and width $= 4 - 2 + 1 = 3$. From [Goodfellow et al., 2016].

Contrary to the concept of context specific filters as shown in Table 2.1, the values of filters in CNNs are set as weights to be learned iteratively via back-propagation. Specifically, given a chosen initialisation of filter values (e.g. He initialisation [He et al., 2015]), the network learns the values of filters at each hidden layer that best reduce the chosen loss function for the task at hand, e.g.

classification of whether the input image depicts a cat or a dog.

Padding and Pooling

We observe from (3.2), that the convolution operation reduces the size of the feature map from the $(q)^{th}$ to the $(q+1)^{th}$ layer. Typically, this results in a loss of information along the borders of an image (or feature map, for hidden layers) as the filters are not able to fit along the full width without "sticking out" beyond the edges. As this is undesirable, we make use of *padding* to resolve this issue. Padding refers to the technique of adding pixels to the borders of an image (or feature map) in order to maintain the spatial dimensions between the input and output feature maps throughout the CNN structure. Padding is used widely in CNNs and in practice, the value of the added pixels is set to 0 (known as *zero-padding*) such that the padded sections do not contribute to the dot product of the convolution operation [Aggarwal, 2019]. Zero-padding thus facilitates the convolution operation with a section of the filter "sticking out" beyond the borders of a feature map and only computing the dot product over the areas of the feature map where values are defined.

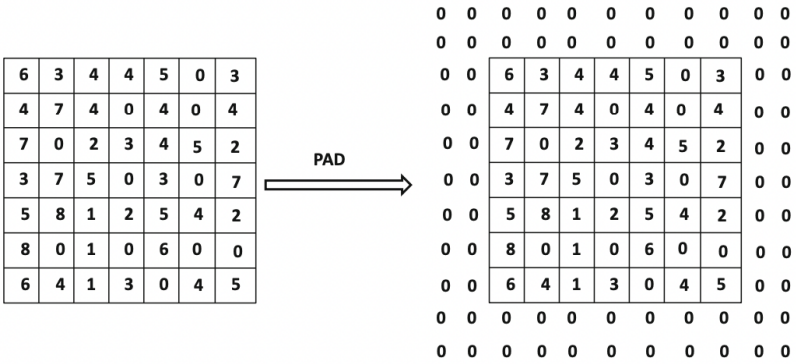


Figure 3.4: Example of zero-padding on a 7×7 feature map. Padding thus allows for the convolution operation over the entire feature map where values are defined. From [Aggarwal, 2019].

Pooling is an operation performed following convolution that down-samples the feature maps created by the convolution operation itself. Pooling combines an arbitrary $n \times n$ region of values in a given feature map into a singular value, thereby creating a sub-sampled feature map of smaller spatial dimension [Scherer et al., 2010]. Typically, the combination is applied by simply taking the largest value within the $n \times n$ section, known as *max-pooling* and is given by:

$$a_j = \max_{N \times N} \left(a_i^{n \times n} u(n, n) \right), \tag{3.3}$$

where $u(x, y)$ is a window function that takes the maximum over each non-overlapping $n \times n$ region. While in practice, a common choice for n in max-pooling is $n = 2$, we visualise the effect of the max-pooling operation on an image for $n = 64$, $n = 16$ and $n = 1$ (corresponding to no pooling) below:

Figure 3.5: Visualisation of max-pooling operation on an image for $n = 64$ (left), $n = 16$ (middle) and $n = 1$ (right) From [Gholamalinezhad and Khosravi, 2020].

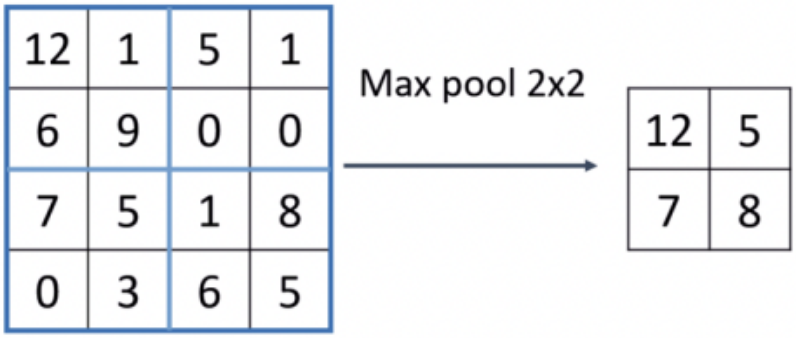


Figure 3.6: Example of 2×2 max-pooling operation on a feature map. From [Nirthika et al., 2022].

Non-linear Activation and Fully-Connected Layers

Following the convolution operation, a non-linear activation is typically applied to each of the $L_q \times B_q \times d_q$ feature map values in a given hidden layer, and is analogous to the non-linear activations used in ANNs. This results in $L_q \times B_q \times d_q$ thresholded values which are passed on to the next hidden layer. Therefore, applying the non-linear activation does not change the dimensions of a layer as it is a one-to one mapping of feature map values. In practice the *Rectified Linear (ReLu)* activation function is used due to its superiority in terms of speed and convergence over other activation functions [Krizhevsky et al., 2017] and is currently the conventional choice. Given a feature map value, x, the ReLu activation, $f(x) = \max(0, x)$ simply returns the maximum of the value itself and 0.
In practice, a typical CNN architecture consists of multiple hidden layers where convolution, non-linear activation and pooling operations are applied, in that order, to the input feature maps. The output from the final hidden layer in our spatial layers is then flattened and fed into a fully-connected layer, which functions identically to a traditional ANN as described in Section 3.1. Similar to the hidden spatial layers, more than one fully-connected layer is used in order to improve the classification and generalisation ability of the model. For example, AlexNet, the CNN architecture that achieved state-of-the-art accuracy on the ImageNet dataset in 2012, consisted of 5 hidden spatial layers and 3 fully-connected layers

[Krizhevsky et al., 2017]. In other words, a modern CNN architecture consists of multiple iterations of convolution and pooling operations followed by a multi-layer ANN. Finally, the output layer is fully-connected to every neuron in the final fully-connected layer, and has a weight associated with it. The number of neurons and choice of non-linear activation in the output layer is largely dependent on the nature of the classification task at hand. For binary classification (e.g. whether an image depicts a dog or a cat), the output layer consists of one neuron and a sigmoid activation function is used:

$$f(x) = \frac{1}{1 + e^{-x}}.$$

Conversely, in a multi-class classification setting, the number of neurons is chosen to be equal to the number of classes in our classification problem and a softmax activation function is used to convert the output of the final fully-connected layer into class probabilities:

$$f_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}.$$

3.1.4 Backpropagation through convolutions

We conclude this section by providing a mathematical description of backpropagation in a CNN. For simplicity, we only derive the backpropagation equations for a filter in a given layer. In a given hidden layer k with a 2×2 filter, $\mathbf{F}$ and a 3×3 feature map, $\mathbf{X}$, the convolution operation is described by the schematic below:

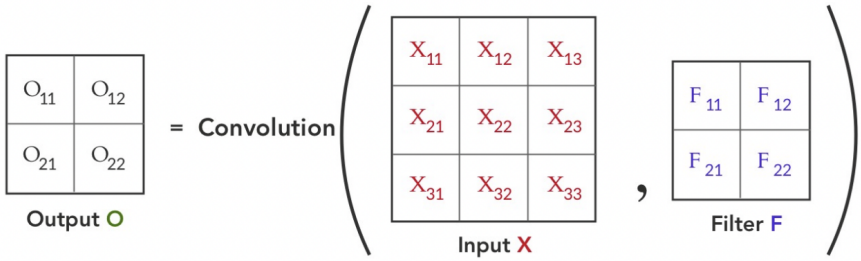


Figure 3.7: Schematic of convolution operation between a 2×2 filter and a 3×3 feature map. From [Sadiq, 2019].

Specifically, by application of (3.2), the outputs, $\mathbf{O}$, are computed as,

$$\begin{aligned}
\mathrm{O}_{11} &= \mathrm{X}_{11}\,\mathrm{F}_{11} + \mathrm{X}_{12}\,\mathrm{F}_{12} + \mathrm{X}_{21}\,\mathrm{F}_{21} + \mathrm{X}_{22}\,\mathrm{F}_{22} \\
\mathrm{O}_{12} &= \mathrm{X}_{12}\,\mathrm{F}_{11} + \mathrm{X}_{13}\,\mathrm{F}_{12} + \mathrm{X}_{22}\,\mathrm{F}_{21} + \mathrm{X}_{23}\,\mathrm{F}_{22} \\
\mathrm{O}_{21} &= \mathrm{X}_{21}\,\mathrm{F}_{11} + \mathrm{X}_{22}\,\mathrm{F}_{12} + \mathrm{X}_{31}\,\mathrm{F}_{21} + \mathrm{X}_{32}\,\mathrm{F}_{22} \\
\mathrm{O}_{22} &= \mathrm{X}_{22}\,\mathrm{F}_{11} + \mathrm{X}_{23}\,\mathrm{F}_{12} + \mathrm{X}_{32}\,\mathrm{F}_{21} + \mathrm{X}_{33}\,\mathrm{F}_{22}
\end{aligned} \tag{3.4}$$

Using the chain-rule, we can compute the gradient of the loss, L, w.r.t $\mathbf{F}$ as

$\frac{\partial L}{\partial F} = \frac{\partial L}{\partial 0} * \frac{\partial 0}{\partial F}$. For every element of $\mathbf{F}$, the gradient is computed as:

$$\frac{\partial L}{\partial F_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial F_i}.$$

By expanding the summation, we note the gradient for each element of $\mathbf{F}$:

$$\begin{aligned}
\frac{\partial L}{\partial F_{11}} &= \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{11}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{11}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{11}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{11}} \\
\frac{\partial L}{\partial F_{12}} &= \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{12}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{12}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{12}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{12}} \\
\frac{\partial L}{\partial F_{21}} &= \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{21}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{21}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{21}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{21}} \\
\frac{\partial L}{\partial F_{22}} &= \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{22}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{22}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{22}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{22}}.
\end{aligned}$$

By taking derivatives of each O_{ij} w.r.t to each F_{ij} in (3.4), we can simplify the equations for the gradients of each element of $\mathbf{F}$:

$$\begin{aligned}
\frac{\partial L}{\partial F_{11}} &= \frac{\partial L}{\partial O_{11}} * X_{11} + \frac{\partial L}{\partial O_{12}} * X_{12} + \frac{\partial L}{\partial O_{21}} * X_{21} + \frac{\partial L}{\partial O_{22}} * X_{22} \\
\frac{\partial L}{\partial F_{12}} &= \frac{\partial L}{\partial O_{11}} * X_{12} + \frac{\partial L}{\partial O_{12}} * X_{13} + \frac{\partial L}{\partial O_{21}} * X_{22} + \frac{\partial L}{\partial O_{22}} * X_{23} \\
\frac{\partial L}{\partial F_{21}} &= \frac{\partial L}{\partial O_{11}} * X_{21} + \frac{\partial L}{\partial O_{12}} * X_{22} + \frac{\partial L}{\partial O_{21}} * X_{31} + \frac{\partial L}{\partial O_{22}} * X_{32} \\
\frac{\partial L}{\partial F_{22}} &= \frac{\partial L}{\partial O_{11}} * X_{22} + \frac{\partial L}{\partial O_{12}} * X_{23} + \frac{\partial L}{\partial O_{21}} * X_{32} + \frac{\partial L}{\partial O_{22}} * X_{33}.
\end{aligned}$$

Thus completing the derivation of the backpropagation equations of the loss, L, w.r.t each element of filter, $\mathbf{F}$. It is important to note that when taking into account the max-pooling operation and non-linear activations, the computation of the backpropagation equations becomes highly involved and we omit such derivations here. A detailed explanation of backpropagation involving pooling and non-linear activations can be found in [Aggarwal, 2018].

3.1.5 Faster R-CNN

While traditional CNNs akin to those described in Section 3.1.3 delivered strong gains in simple image classification tasks, recent advances have led to the development and popularisation of models that are able to localise instances of objects in a given image, known as *object-detection models*. In this section, we describe one particular choice of object-detection model, called *Faster R-CNN*, that has achieved state-of-the-art results on such tasks. Below, we describe the two novel modules that Faster R-CNN implements: *Region Proposal Networks* and *Fast R-CNN detection*.

Figure 3.8: Examples of object-detection in images. From [Ren et al., 2017].

Region Proposal Networks

As the first module in the Faster R-CNN model, the Region Proposal Network (RPN) a deep fully-convolutional network that proposes regions of interest (RoI) in our input image. With the ultimate aim of sharing computation between the RPN and the Fast R-CNN detector, both modules share a common set of convolution layers. A 3×3 window is passed along the output feature map of the final shared convolution layer and **at each sliding-window location**, we simultaneously predict multiple region proposals, where the number of maximum possible proposals for each location is denoted as k. These k proposals are parameterized relative to k reference boxes, called *anchors*. Importantly, these k anchors are centred at the position of the sliding window and are parameterised by *scale* and *aspect ratio* parameters as visualised below in Figure 3.10. Thus, for a convolutional feature map of a size $W \times H$, there are $W \times H \times k$ anchors in total. By default, Faster R-CNN uses 3 different scale and aspect ratio parameters, leading to a total of $k = 9$ anchor boxes.

Fast R-CNN detection

First proposed in [Girshick, 2015], Fast R-CNN proved to be, at the time, the state-of-the-art in object detection by introducing the *ROI-Pooling* operation in conjunction with two fully-connected layers. In Faster R-CNN, the Fast R-CNN model is used as a classification head. For each of the k anchor boxes generated in Section 3.1.5, we apply a variation of the max-pooling operation, known as *ROI-Pooling*, to reduce them to a 256-dimensional feature vector. While we do not describe ROI-Pooling in-depth, the operation splits each of the k region proposals into equally sized grids. The max-pooling operation is then applied to each grid

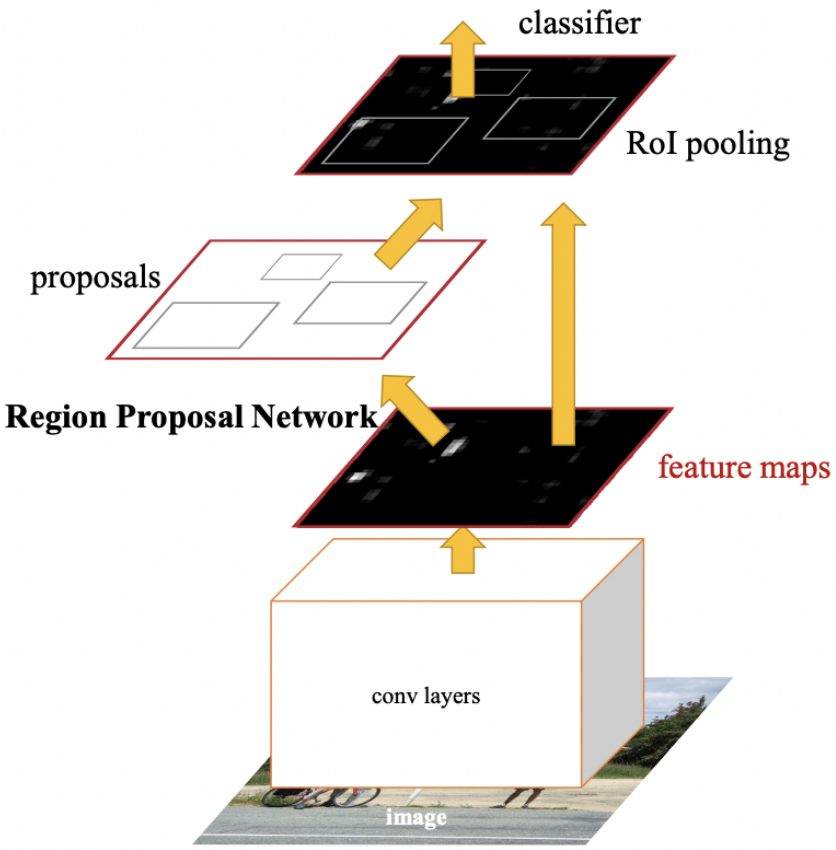


Figure 3.9: Schematic diagram of Faster R-CNN structure. From [Ren et al., 2017].

in the proposal to return a single maximum value [Girshick et al., 2014]. These maximum values from all grids then represent the feature vector.
The feature vector is then fed into two sibling fully-connected layers, similar to that described in Section 3.1.3. The first of which is a box-regression layer (denoted *reg*), which performs bounding-box regression on the coordinates of the region proposals and thus, outputs a 4-dimensional vector denoting the coordinates of each of the k region proposals. The second fully-connected layer is a box-classification layer (denoted *cls*) which performs binary classification on the region proposal and outputs a 2-dimensional vector of scores that estimate probability of object or not object for each proposal [Ren et al., 2017].

Training Faster R-CNN

The method for training a Faster R-CNN is relatively straightforward. A binary class label is assigned to each region proposal anchor (whether is an object or not) and positive labels are assigned to two kinds of anchors:

1. region proposals with the highest Intersection-over Union (IoU) overlap with a ground truth box

2. region proposals that have an IoU overlap higher than 0.7 with any ground

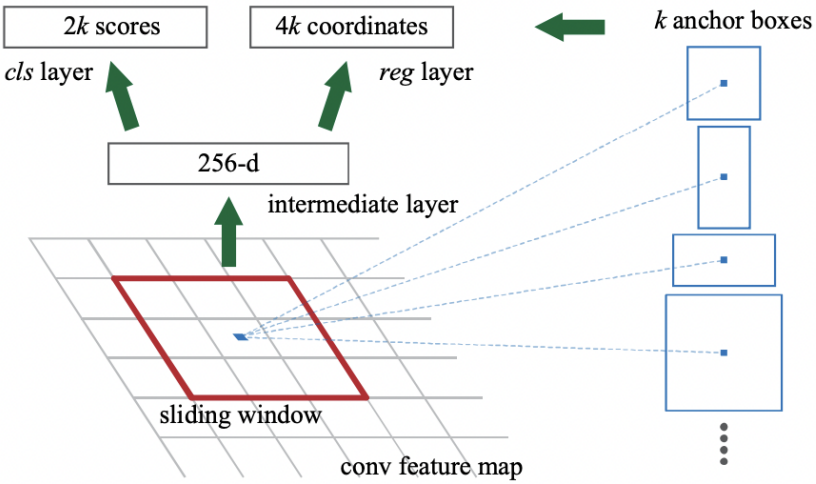


Figure 3.10: Anchor boxes for a singular sliding window position in the RPN. From [Ren et al., 2017].

truth box

Taking into account the above conditions, Faster R-CNN minimises the following loss function, for a single input image:

$$\begin{aligned} L\left(\{p_i\},\{t_i\}\right) = & \frac{1}{N_{cls}} \sum_i L_{cls}\left(p_i, p_i^*\right) \\ & +\lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}\left(t_i, t_i^*\right) \end{aligned} \tag{3.5}$$

Here, i is the index of a region proposal in an image and p_i is the predicted probability of the proposal i being an object, which is the output of the *cls* fully-connected layer. The ground truth label p_i^* is 1 if the proposal is positive, and is 0 if negative. t_i is a vector representing the 4 parameterized coordinates of the predicted bounding box i.e. the output of the *reg* layer, and t_i^* is that of the ground truth box associated with a positive proposal. The classification loss L_{cls} is a log-loss over two classes (object or not object). For the regression loss, we use $L_{\text{reg}}\ (t_i, t_i^*) = \text{smooth}\left(t_i - t_i^*\right)$ where the smooth_{L1} is defined as [Girshick, 2015]:

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

The term $p_i^* L_{reg}$ implies that the regression loss is activated only for positive anchors ($p_i^* = 1$) and is disabled otherwise ($p_i^* = 0$). The two terms are normalized by N_{cls} and N_{reg} and weighted by a balancing parameter λ. By default, the *cls* term in (3.5) is normalized by the batch size of inputs to Faster R-CNN while the *reg term* is normalized by the number of region proposal locations. Furthermore, λ is set to $\lambda = 10$ to allow equal weighting to the *reg* and *cls* terms. We note, however, that empirically, model performance is relatively insensitive to the values of λ in a wide range.

Chapter 4

Results

4.1 Classical Methods

After exploring various classical image processing methods, we have created a pipeline that applies multiple classical methods to the input image in a given order and outputs a final segmentation result. All methods and their respective parameters were selected manually, and tested on a subset of 5 images from the subtracted images data set. We adjusted the methods/parameters based on the test results.
Compared to modern machine learning methods, this is an inefficient way to generate a segmentation pipeline. Hence, if the output of a specific method with specific parameters adequately fulfills the tasks, it will be selected as part of the pipeline and other methods will not be considered for that specific step. Therefore some methods discussed earlier will not be considered at all, not because they are unsuitable for all the steps in the pipeline but because there are already other methods that can adequately perform given tasks at specific steps.
Furthermore, the reason for only using subtracted images is that they are very different from the low-energy images in terms of intensity composition in the breast tissue. It is difficult to design a pipeline that adequately works for both image sets. We decided to only focus on the implementation of the classical pipeline on one of the image sets. The subtracted set was selected because the areas labelled by ground-truth labels give cleaner clusters of pixels compared to the low-energy set. The subtracted set is therefore easier to handle for the classical methods.

4.1.1 Pipeline Overview

Algorithm 8 is a summary of the classical segmentation pipeline for the subtracted images data set.

Algorithm 8 Classical Segmentation Pipeline

Input: Image $\mathbf{I}$ from subtracted image data set, with pixels $p \in \mathbf{I}$ and intensities $I(p)$.
Output: Image $\mathbf{I}$ with abnormal regions marked.

1. Denoise $\mathbf{I}$ using non-local mean filter to create $\mathbf{I}_1$ with intensities $I_1(p)$.
2. Create a mask $\mathbf{M}_1$ with labels $M_1(p)$ that separates tissue ($M_1(p) = 1$) and background ($M_1(p) = 0$) using Sauvola Thresholding.
3. Refine $\mathbf{M}_1$ using binary operations.
4. Create an copy $\mathbf{I}_2$ of $\mathbf{I}_1$ with intensity $I_2(p)$, set $I_2(p) = 0$ if $M_1(p) = 0$.
5. Perform masked thresholding (defined in later section) on $\mathbf{I}_2$ with mask $\mathbf{M}_1$ using Otsu's method to create mask $\mathbf{M}_2$ with labels $M_2(p)$.
6. Refine $\mathbf{M}_2$ using binary operations.
7. Set $R_1 = \{p \in \mathbf{M}_2 \mid M_2(p) = 0\}$ and preform region growing algorithm in Section 2.3 to get regions $\{R_1 \dots R_n\}$ in the form of a label matrix $\mathbf{L}_1$ with labels $L_1(p)$.
8. Set some threshold $T \in \mathbb{N}$, create a copy $\mathbf{L}_2$ of $\mathbf{L}_1$ with labels $L_2(p)$ and set $L_2(p) = 1$ if $|R_{L_1(p)}| < T$.
9. Mark the boundaries of $\mathbf{L}_2$ on $\mathbf{I}$ using `mark_boundaries` function discussed in Section 2.6.

We will base our discussion of the pipeline on the image in the left panel in Figure 4.1. We apply the procedures in the pipeline step-by-step and justify our choices along the way. Our aim is to produce a result similar to the image in the center panel of Figure 4.1, which was labelled by a medical professional.

4.1.2 Denoising

This section corresponds to step 1 in Algorithm 8. The right of Figure 4.1 shows the denoised result ($\mathbf{I}_1$) of test image ($\mathbf{I}$) using the non-local mean filter discussed in Section 2.1.2 with parameters $h = 10$, size of search window $= 5$ and size of comparison window $= 2$. As we can see, the denoising filter has reduced the Gaussian-like noise on the image and can potentially lead to less noisy masks when we apply thresholding methods later. A comparison of the final results for the noisy and denoised image, when processed by the pipeline, will be discussed in a later section (in Section 4.1.6).

4.1.3 Mask for Thresholding

This section corresponds to steps 2, 3, 4 in Algorithm 8. In this section, we aim to create a mask that labels the black background of the scan and the bright parts

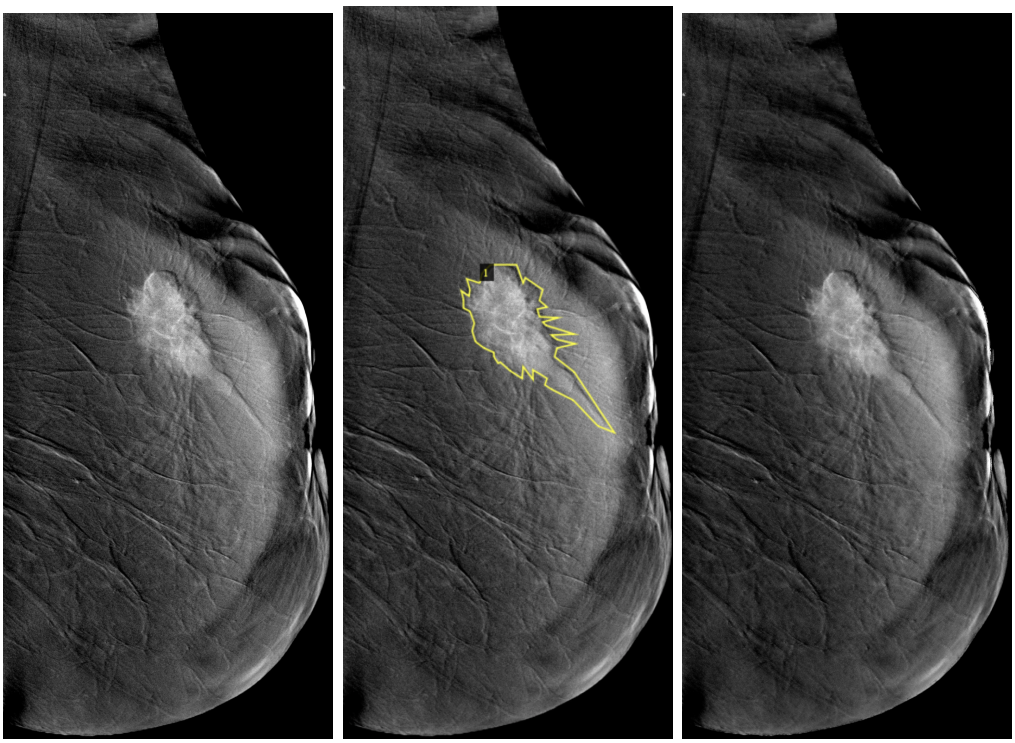


Figure 4.1: Images for classical segmentation pipeline discussion, original unlabelled (**I**, left), ground truth label (middle) and denoised unlabelled ($\mathbf{I}_1$, right).

along the edge of the breast as background. In general terms, this mask allows the algorithm to focus on the interior of the tissue, the details will be discussed in the following section.
The decision of using Sauvola Thresholding (discussed in Section 2.2.5) to generate a mask that separates tissue and background was due to a coding mistake. As we were testing different thresholding methods on the image to achieve the mask described above, Sauvola Thresholding, with $n = 25$, $k = 0.2$, was faultily implemented. It created a mask, $\mathbf{M}_1$, that clearly separated the tissue from the background, as shown in Figure 4.2 (1). However $\mathbf{M}_1$ is not the mask from Sauvola Thresholding correctly implemented. Its labels were accidentally set so that $M_1(p) = 0$ for $T(p) = 0$ and $M_1(p) = 1$ otherwise. A correctly implemented mask, under Sauvola Thresholding, would have labels $M_1(p) = 0$ for $I_1(p) \leq T(p)$ and $M_1(p) = 1$ for $I_1(p) > T(p)$. Although $\mathbf{M}_1$ was obtained due to a mistake in the code, it successfully achieved the aims of step 2 in Algorithm 8 as outlined above. Hence, we decided to keep the incorrect result, $\mathbf{M}_1$, and the incorrect implementation as a part of the pipeline.
Recall the expression for $T(p)$ in Sauvola Thresholding is given

$$T(p) = m(p)(1 + k(\frac{s(p)}{R} - 1)). \tag{4.1}$$

For any $p \in \mathbf{I}_1$, $T(p) = 0$ if $m(p) = 0$ or $(1 + k(s(p)/R - 1)) = 0$. The latter case is not possible for $k = 0.2$ (the parameter used here). Since $0 \leq s(p)/R$, it follows that $-0.2 \leq (k(s(p)/R - 1) \Rightarrow (1 + k(s(p)/R - 1)) > 0$.

Figure 4.2: Thresholding mask refinements, from left to right (1): original $\mathbf{M}_1$, (2): dilated (1) with $P(p) = \overline{D}_5(p)$, (3): small holes removed (2) with $A = 400$, $c = 1$, (4): edge linked (3), (5): edge expanded (4) with $r = 150$ and (6): $\mathbf{I}_2$.

Thus $T(p) = 0 \Leftrightarrow m(p) = 0 \Leftrightarrow$ all pixels in $P(p)$ (the 25×25 square patch centred at p) is black. Since the background of the scan is solid black, the algorithm can identify pixels in the background and label these accordingly in $\mathbf{M}_1$. However, for some background pixels $\bar{p}$ around the tissue edge, $\exists \hat{p} \in P(\bar{p})$ for which $I(\hat{p}) \neq 0 \Rightarrow T(\bar{p}) \neq 0 \Rightarrow M_1(\bar{p}) = 1$, leading to a shrunken background label. Furthermore as we can see from Figure 4.2, some tissue pixels p_T also got labelled as background, having a black patch in their neighbourhood. To address these issues, we applied several binary operations on the mask.
The pipeline of binary operations on $\mathbf{M}_1$ will be discussed in Algorithm 9. Its steps correspond to the transition between panels Figure 4.2.

Algorithm 9 $\mathbf{M}_1$ Binary Operations

Input: Binary mask $\mathbf{M}_1$.
Output: Refined binary mask $\mathbf{M}_1$.

1. (1) $\rightarrow$ (2) Perform dilation on $\mathbf{M}_1$ with $P(p) = \overline{D}_5(p)$ to reduce the size of incorrectly labelled pixels in the tissue.
2. (2) $\rightarrow$ (3) Remove the incorrectly labelled pixels using `remove_small_holes` (discussed in Section 2.5.2) on (2) with parameter $c = 1$, $A = 400$.
3. (3) $\rightarrow$ (4) Identify any outermost row or column of (3) with at least 50% black pixels in each half (split down the middle). Then draw a black line 5 pixels wide along the respective row or column. This step aims at restoring edge pixels that were faultily labelled as foreground due to the shrinkage of the mask (either caused by modified Sauvola or dilation).
4. (4) $\rightarrow$ (5) Expand the black region in (4) by removing white pixels within a 150 pixel radius when going along the existing edge between white and black. In other words, first identify the boundaries $\mathcal{B}$ using `find_boundaries` (discussed in Section 2.6) with $c = 1$. Then for all $b \in \mathcal{B}$, let $\mathcal{S} = \overline{D}_{150}(b) \cap \mathbf{M}_1$. Subtract $\mathbb{1}_{\mathcal{S}}(p)$ from all $p \in$ (4) ($\mathbb{1}_{\mathcal{S}}(p) = 1$ if $p \in \mathcal{S}$ and 0 otherwise). Then set all pixels with negative values to 0.
5. Return (5).

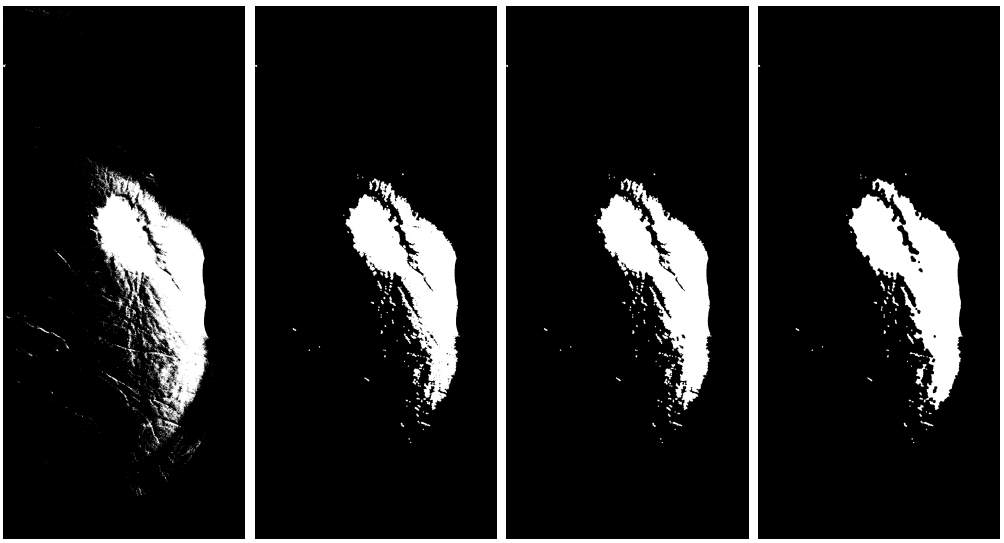


Figure 4.3: Mask $\mathbf{M}_2$ refinements, from left to right (1): original $\mathbf{M}_2$, (2): (1) after opening with $P(p) = \overline{D}_3(p)$, (3): small holes removed (2) with $A = 144$, $c = 1$, (4): (3) after closing with $P(p) = \overline{D}_5(p)$.

Then we perform step 4 of Algorithm 8 to generate $\mathbf{I}_2$.

4.1.4 Masked Thresholding and Mask Refinement

This section corresponds to step 5, 6 in Algorithm 8. In this section, we will perform masked Otsu thresholding on $\mathbf{I}_2$ with mask $\mathbf{M}_1$ (refined version) to obtain $\mathbf{M}_2$. Masked Otsu thresholding simply excludes pixels in $\mathbf{I}_2$, that are marked as background by $\mathbf{M}_1$, from the histogram used to calculate the threshold. Then we perform binary operations to refine $\mathbf{M}_2$.
Figure 4.3 shows the results for masked thresholding and refinement. Algorithm 10 discusses the pipeline for refining $\mathbf{M}_2$ (labelling based on the corresponding subfigures in Figure 4.3).

Algorithm 10 $\mathbf{M}_2$ Binary Operations

Input: Binary mask $\mathbf{M}_2$.
Output: Refined binary mask $\mathbf{M}_2$.

1. (1) $\rightarrow$ (2) Perform opening on $\mathbf{M}_2$ with $P(p) = \overline{D}_3(p)$ to reduce the number of white gaps in the mask.
2. (2) $\rightarrow$ (3) Remove the small black spots by using `remove_small_holes` (discussed in Section 2.5.2) function on (2) with parameters $c = 1$, $A = 144$.
3. (3) $\rightarrow$ (4) Perform opening on (3) with $P(p) = \overline{D}_5(p)$ to further reduce the number of black spots in the mask and clear up the boundaries.
4. Return (4).

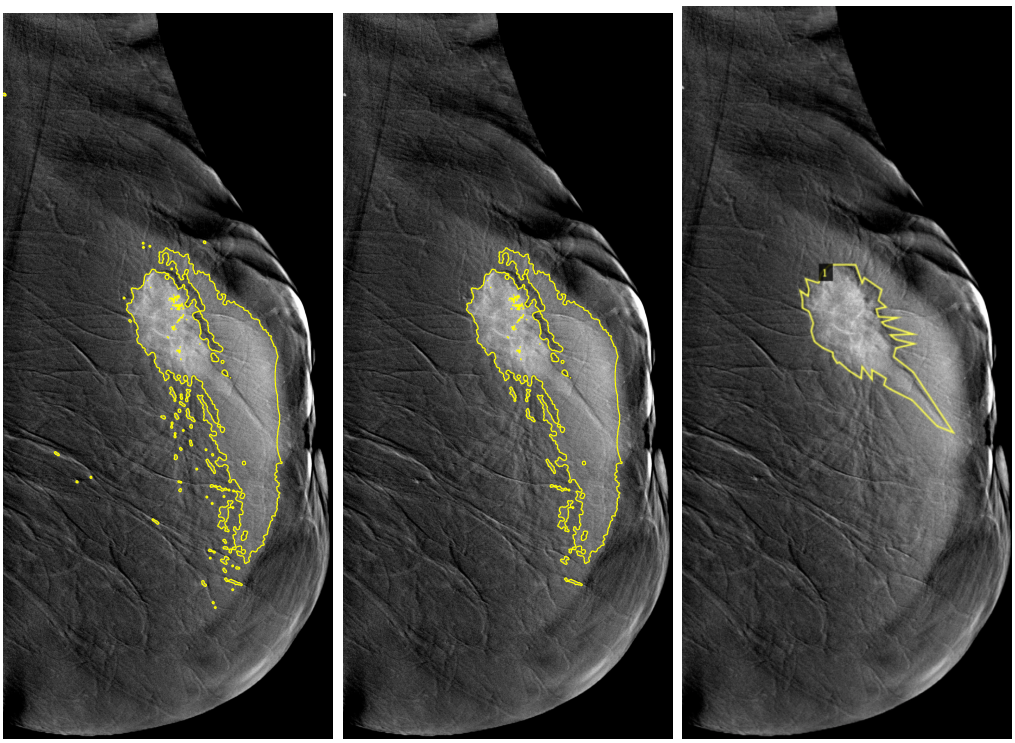


Figure 4.4: $\mathbf{I}_2$ (with mask $\mathbf{M}_2$) region growing results (and ground truth label), marked on $\mathbf{I}$ (1) before removing small regions (left), (2) after removing small regions (middle) and (3) the ground truth label (right).

4.1.5 Region Growing With Mask

This section corresponds to step 7, 8, 9 in Algorithm 8. We perform region growing on $\mathbf{I}_2$ (discussed in Section 2.3, with the same parameter as Figure 2.12) by classifying all the background pixels into region R_1 and start the growing algorithm to generate $\{R_2, \ldots, R_n\}$ (assuming we have n total regions). We get our result in the form of a label matrix $\mathbf{L}_1$ that takes values $L_1(p)$ where $L_1(p) = k$ if $p \in R_k$ for $p \in \mathbf{L}_1$. Then we re-label the regions with less than $T = 200$ pixels as region 1 (the background region) and generate a new label $\mathbf{L}_2$. The previous step allows us to remove minor regions generated due to remaining spots (impulse noise) on $\mathbf{M}_2$ or any kind of remaining noise on $\mathbf{I}_2$. We then use the `mark_boundaries` function (discussed in Section 2.6) to mark the boundaries of $\mathbf{L}_2$ on $\mathbf{I}$, Figure 4.4 shows the result. Before the removal of small regions, the total region number is 164 while after the process the region number is 9 which is a significant decrease while leaving regions with (somewhat) significant size unchanged.
Comparing with Figure 4.4 (3) (the ground truth label) we see our compiled algorithm labels a larger region that mostly contains the small region. This might be due to the fact the edge between the ground truth labelled region and the additional region we labelled has similar intensity leading to failure of thresholding algorithm (in terms of separating these two regions). Furthermore, the unclear edge leads to the failure of region growing algorithm. Thus, our classical algorithm complication failed to separate these two regions, and resulting

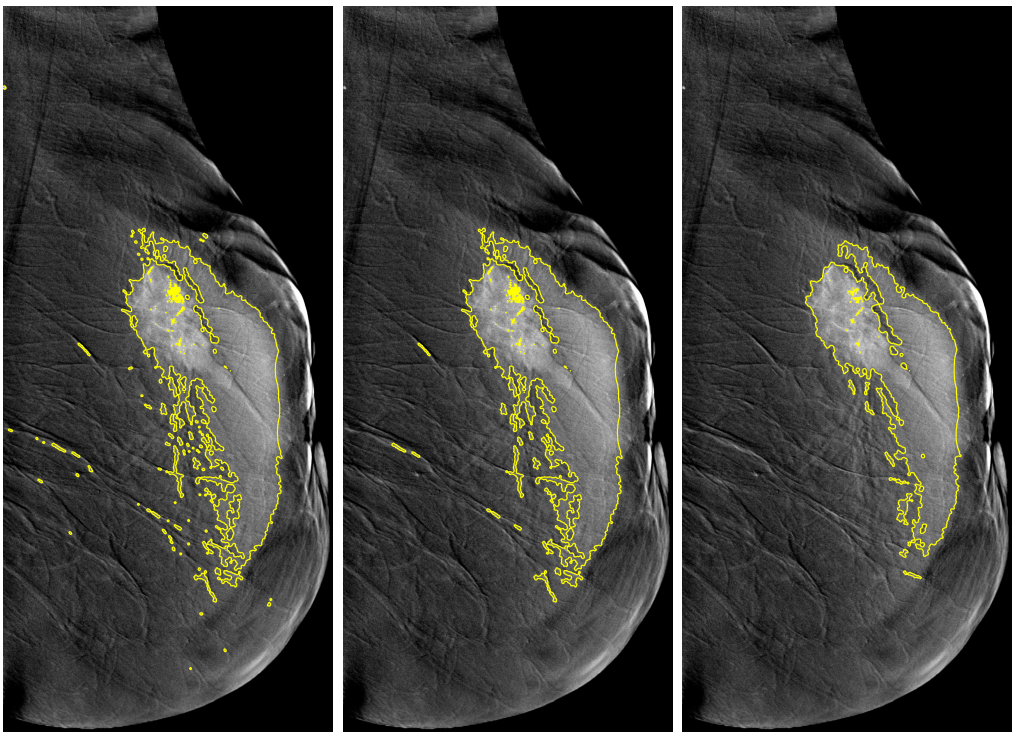


Figure 4.5: $\mathbf{I}_2$ (with mask $\mathbf{M}_2$) region growing results, marked on $\mathbf{I}$ (1) undenoised image, small regions not removed (left), (2) undenoised image, small regions removed (middle) and (3) denoised image, small regions removed (right).

in labelling a larger region containing the true label as one region.

4.1.6 Effect of Denoising on Result

In this section, we will briefly discuss the result of Algorithm 8 if we omit step 1 of the algorithm (denoising), since it is the most computationally intensive step that takes around 30 minutes to run while the rest of the steps takes around 30 seconds to run (on a single image). In practice, we perform step 1 separately and store the denoised image, and then we perform the rest of the steps on that image. But if we want to systematically run Algorithm 8 on the larger set, the high run time of the denoising algorithm may be undesirable.
Figure 4.5 shows the result comparison, as we can see the denoised result has cleaner bounding box with fewer small regions (with more than 200 pixels) scattered around the image and has fewer regions than the undenoised result (17 vs. 9 after small region removal). So denoising the image can indeed improve the result in the sense that the bounding box is cleaner thus closer to the true bound. However if starting from an image that has not been denoised, the run time will be approximately 60 times higher if we do implement the denoising step.

4.1.7 Test Set Results

We now discuss the results obtained from implementing Algorithm 8 to a hold-out test set of images. Figure 4.6 shows the results. As we can see from the results,

regions marked by Algorithm 8 seem to be the brightest patches on the image which in actuality may not correspond to a ground truth area of interest (AoI), which are specifically labelled by a medical professional. This is an unavoidable caveat of the algorithm since all of the classical methods we have discussed are intensity orientated. Thus the most feasible way to build the pipeline is by assuming the brightest patches on the image are AoIs, which is true to a certain extent as the ground truth label in general contains part of the brightest spots.
Notice we try to exclude the nipple-like object (addressed as NLO) from our marked regions as it is not labelled in the ground truth label. If the NLO is located around the edge of the tissue, the mask expansion algorithm (Algorithm 9 step 4) is able to label it as background and successful masking of NLOs is shown by Figure 4.6 (1r), (2r) and (4r) as the NLO was labelled as background in these images (it is not being marked around). However the NLO in (3r) is at the inner part of the tissue, thus we cannot mask it with the mask expansion algorithm. Therefore it becomes part of the marked region in that image.
Furthermore, notice that (2r) has no marked region. This is due to almost all of the pixels are labelled as background in $\mathbf{M}_2$ by Otsu's thresholding (Algorithm 8 step 5) as illustrated in Figure 4.7. The unusual result may be due to the histogram has a single normal-distribution like shape and the principle of Otsu's method is to try to find a threshold that separates two clusters in the histogram, it is not suited for this kind of histogram thus generating a wildly inaccurate result. However the coin image in Figure 2.8 has a similarly shaped histogram as the one in Figure 4.7 but it does not draw the threshold line that classifies all the pixels into the background. At first glance we suspect it's due to missing pixels with certain intensity (represented by the white stripes in Figure 4.7), however, after inspecting (2.2.3) we see that the parameters used to calculate $\sigma_b^2(q)$ (the value to maximize), $n_0(q), n_1(q), \mu_0(q), \mu_1(q)$ are summations over $h(i)$s the number of pixels with intensity i. So the impact of the missing intensity values are minimal on $\sigma_b^2(q)$. Therefore our conjecture of the discrepancy between the result of the coin image and (2r) is due to slight differences between those histograms, and the fact that Otsu's method is not designed to deal with these kind of histograms so a slight change in histogram shape may lead to large difference between the threshold value q that maximizes $\sigma_b^2(q)$. The ground truth label (2t) labelled a tiny bright spot that is only slightly brighter than the surroundings and this is beyond the capability of our classical pipeline.
The inability to recognize regions of interest beyond bright patches on the image and the failing result are intrinsic issues for the classical methods we have selected. Since the methods we used cannot "learn" the shape of the regions of interest from the labelled data it cannot accurately label these regions accurately if it deviates from our assumption (regions of interest are the bright patches) or if the region is difficult to distinguish from the surrounding. These problems may be addressed if we implement machine learning methods that allow the computer to "learn" the shape of those ground truth labelled region and use that data to predict the AoI on an unlabelled image. We will be discussed such method in the next section.

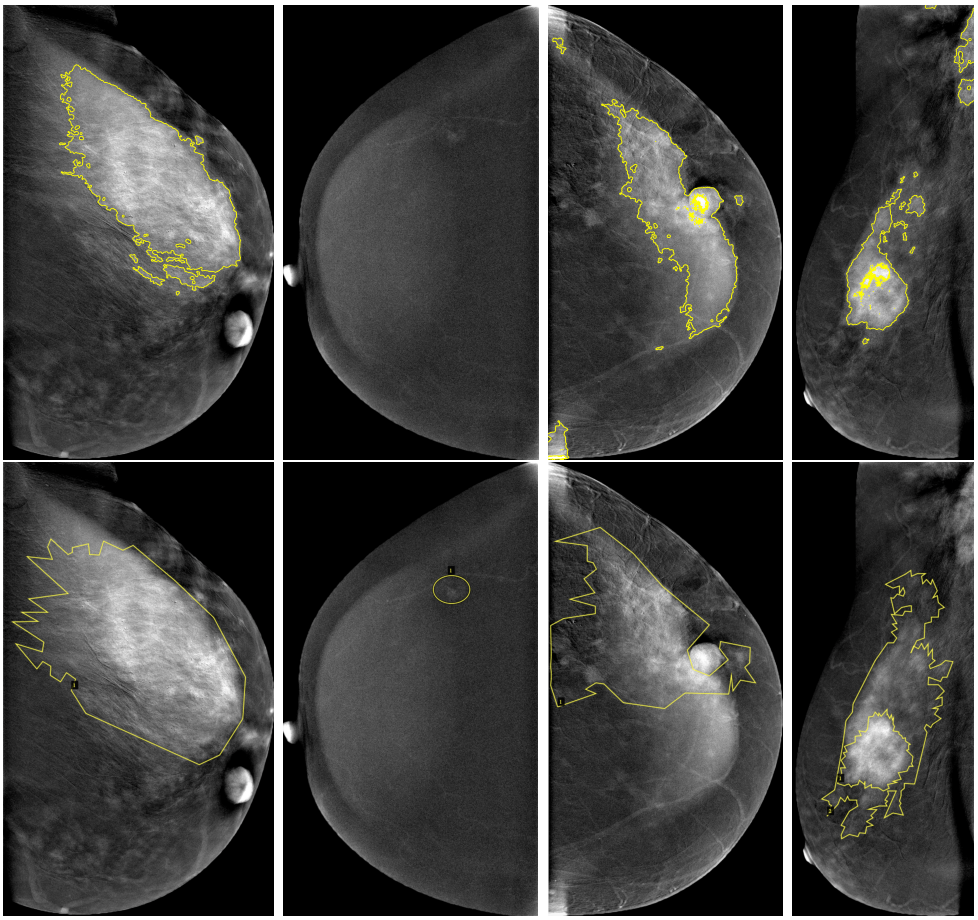


Figure 4.6: Classical segmentation results on test images (top row, labelled (1r) to (4r) from left to right) and ground truth labels of these images (bottom row, labelled (1t) to (4t) from left to right).

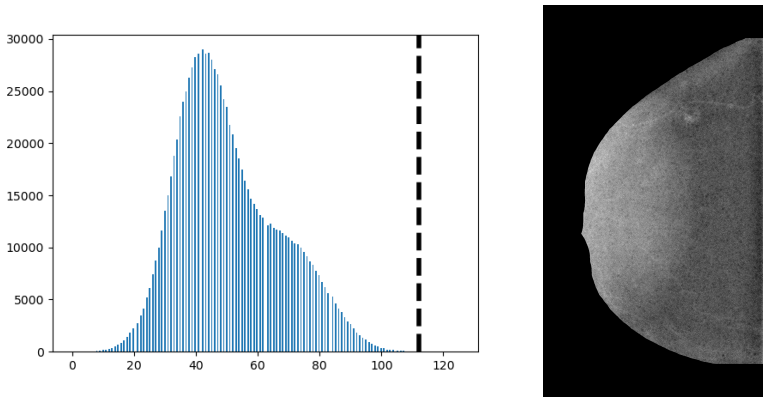


Figure 4.7: Figure 4.6 (2r) histogram excluding pixels labelled as background by threshold mask $\mathbf{M}_1$ black dashed line representing Otsu's threshold value (left) and (2r) with $\mathbf{M}_1$ background coloured black (corresponds to $\mathbf{I}_2$ in Algorithm 8) (right).

4.2 Faster R-CNN Object Detection

4.2.1 Implementation

As a comparison to the classical methods, we implement the Faster R-CNN model on CDD-CESM data. From the original set of images of 2006 images, we retain images with suitable annotations that allow us to compute ground truth bounding boxes. Due to the large amount of data required for fine-tuning, we choose to use both Low-Energy and Subtracted images for this task. This leads to a final dataset size of $n = 466$. For the model implementation, we use a high resolution Faster R-CNN model with a MobileNetV3-Large FPN backbone. This backbone serves as the convolutional layers in the Faster R-CNN structure and generates the final output feature map that is fed into the region proposal network as described in Section 3.1.5. Due to the significant computational resources required to train such a model, we use a variant of that has been pre-trained on the "COCO train2017" dataset, with the aim of fine-tuning it to our task. We implement fine-tuning of our model with the Adam optimiser as described in Section 3.1.2 along with the following hyper-parameters:

Parameter	Value
Learning Rate, η	0.001
ρ_1	0.9
ρ_2	0.999
δ	1×10^{-8}
Fine-Tuning Epochs	5
Batch Size	8

Table 4.1: Hyperparameters used in Fine-Tuning Faster R-CNN model with a MobileNetV3-Large FPN backbone.

In fine-tuning a model, we are able to specify how many of the model layers we would like to allow for training, in order to adjust the pre-trained weights to our specific task. Due to constraints in computation and runtime, we only allow the final layer in the backbone to be trainable and we perform the fine-tuning for 5 epochs. In order to preserve the majority of the data for fine-tuning, we use the following train and test set sizes: $n_{train} = 436$ and $n_{test} = 30$.

4.2.2 Intersection over Union

In object-detection tasks, a typical metric of performance is *Intersection over Union (IoU)*, which computes an accuracy metric by comparing the ground truth bounding box to the predicted bounding-box. Specifically, given a ground truth bounding-box, A, and a prediction, B, IoU is computed as:

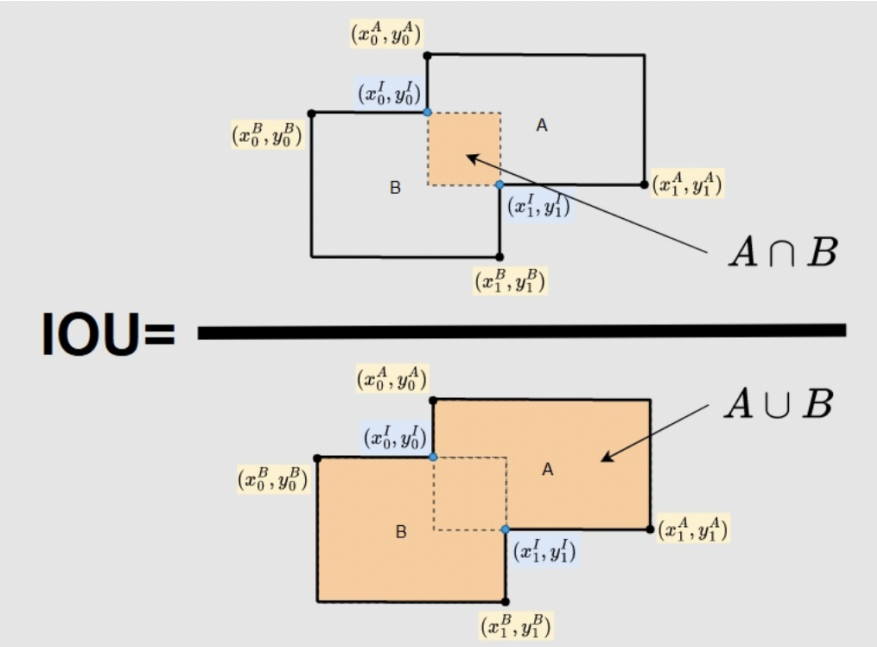


Figure 4.8: Computation of IoU metric. From [Sadli, 2023].

4.2.3 Test Set Results

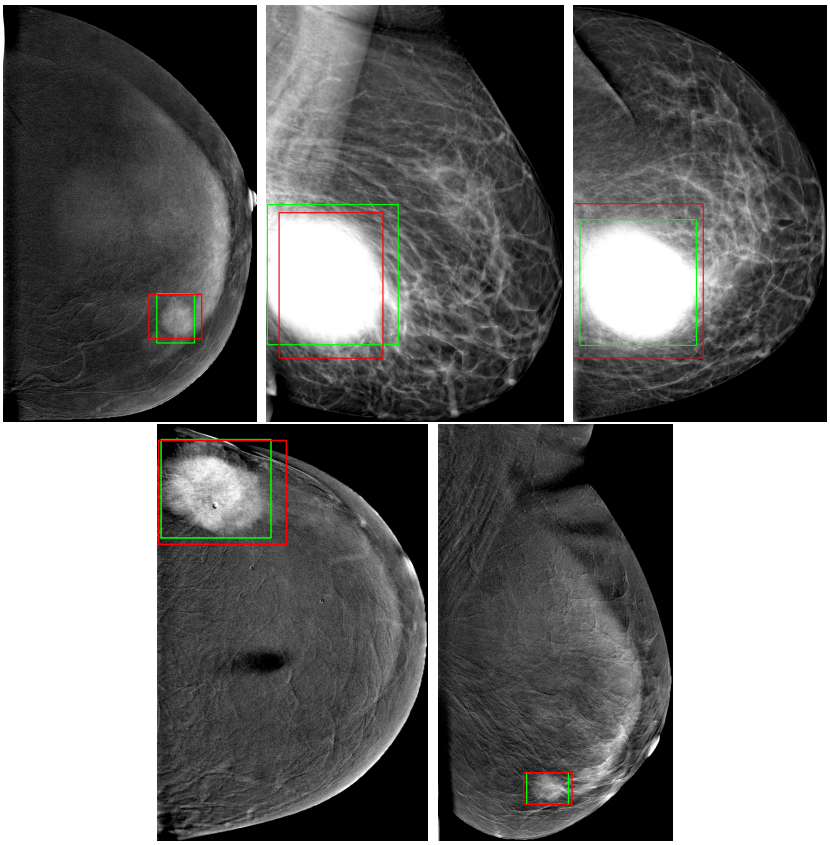


Figure 4.9: Faster R-CNN prediction (red box) and the ground truth label (green box)

Across our test set we select the 5 images with the largest IoU from our test set and visualise the results in Figure 4.9. Note that the green bounding-box denotes the ground truth, while the red bounding-box denotes the prediction from the Faster R-CNN model. We observe that on images where the AoI is clearly visible and segmented, the model performs rather well, and the ground truth AoI is well within the predicted bounding box. Across these 5 images, the average IoU is $\overline{\text{IoU}}_{H5} = 0.843$.
Conversely, the model performs rather poorly on certain test images and we now shift the discussion to the limitations of the model. We show the 3 images with the lowest IoU ($\overline{\text{IoU}}_{L3} = 0.054$) in Figure 4.10. We note that the problems in these 3 images generalise to other images in our test set that the model performs poorly on and can be categorised into two cases.

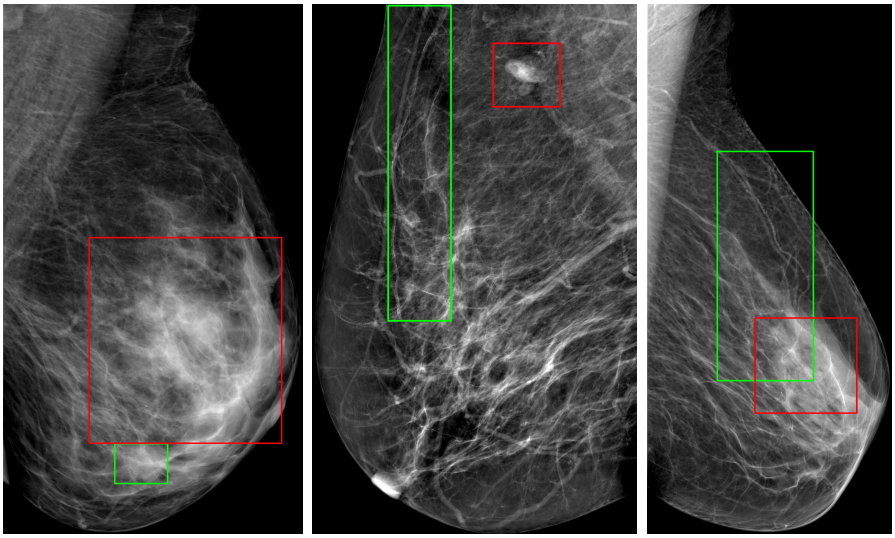


Figure 4.10: Faster R-CNN prediction (red box) and the ground truth label (green box)

The first limitation can be observed on the left of Figure 4.10, where the model predicts a region of the image that is highly similar in texture to that of the ground truth AoI and is unable to distinguish between AoIs and non-AoIs that are visually alike. The second limitation can be seen in the middle of Figure 4.10, where the Faster R-CNN model predicts a bounding-box around a region that looks like a significant AoI which in actuality, is not. There are several potential reasons for this. First, the image may be poorly labelled, resulting in a ground truth AoI that is not representative of the actual AoIs location in an image. The more likely scenario, however, is that the predicted AoI is simply a normal tissue variation or gland within the breast which would appear as "white spots" [Nazari and Mukherjee, 2018] that could be mistaken as AoIs. The image on the right is similar in that the patient's breast is of high fibroglandular density (large proportion of fibroglandular tissue in the breast related to the amount of fatty tissue). In such cases, the model tends to predict this normal tissue as AoIs.
The first limitation of our model can be considerably alleviated by introducing a larger fine-tuning dataset with a greater proportion of images where ground truth AoIs are visually similar to non-AoIs, allowing the model to improve in its

ability to distinguish between the two. By the same token, the model's ability to predict ground truth AoIs in cases of high fibroglandular density can be improved by having more of such cases in the fine-tuning dataset.
A large consideration in the application of an object-detection model to mammography images is the ability to accurately detect ground truth AoIs in cases where the AoI itself may be occluded or blocked by fibroglandular tissue. The fibroglandular tissue absorbs ionizing radiation (X-rays) and projects white on mammography and thus, may "mask" cancers on mammography. As such, patients with high fibroglandular density are at an increased risk of late diagnosis of breast cancer [Mann et al., 2022]. Hence, an improved object-detection model which accurately predicts ground truth AoIs in such cases could provide significant utility to the field of medical anomaly detection.

Chapter 5

Conclusions

With the proliferation of accessible image data over the last decade, wide-ranging and robust image analysis techniques have experienced great demand in their applications. In this report, we have investigated a variety of such techniques, ranging from classical methods to object-detection models in machine learning. Jointly, we have assessed their application to the field of medical anomaly detection, specifically on breast mammography. Our analyses has shown that while classical methods provide satisfactory results on such tasks, they succeed best when used to identify relatively simple, monotonic AoIs. While the object-detection model gives an adequate overall performance in anomaly detection and performs well on most unseen images, it struggles on specific cases.
To build upon our analyses, we could implement data augmentation using a selection of classical methods discussed prior to the usage of Faster R-CNN which could result in a more robust model. On the other hand, using the image annotations as part of the CDD-CESM dataset, we could create masks for the ground-truth AoIs and implement an instance segmentation model such as Mask R-CNN to generate localised predictions as opposed to bounding-boxes.

Bibliography

[a20, 2022] (2022). Edge detection with canny, log/zero-crossing, and wavelets, packtpub.com, https://tinyurl.com/2ryveskf, retrieved on 22.03.2023.

[ski, 2023] (2023). Api reference for skimage 0.20.0 — skimage v0.20.0 docs.

[a20, 2023] (2023). Diagnostic imaging dataset statistical release.

[its, 2023] (2023). *The OpenCV Reference Manual*. Itseez, 4.7.0 edition.

[Adams and Bischof, 1994] Adams, R. and Bischof, L. (1994). Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:641–647.

[Aggarwal, 2018] Aggarwal, C. C. (2018). *8.3.1.* Springer International Publishing.

[Aggarwal, 2019] Aggarwal, C. C. (2019). *Neural networks and deep learning: A textbook*. Springer.

[Buades et al., 2005] Buades, A., Coll, B., and Morel, J. M. (2005). A review of image denoising algorithms, with a new one. *Multiscale Modeling & Simulation*, 4:490–530.

[Burger and Burge, 2016] Burger, W. and Burge, M. J. (2016). *Digital Image Processing : an Algorithmic Introduction Using Java*. Springer, second edition edition.

[Comaniciu and Meer, 2002] Comaniciu, D. and Meer, P. (2002). Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:603–619.

[Froment, 2014] Froment, J. (2014). Parameter-free fast pixelwise non-local means denoising. *Image Processing On Line*, 4:300–326.

[Gholamalinezhad and Khosravi, 2020] Gholamalinezhad, H. and Khosravi, H. (2020). Pooling methods in deep neural networks, a review.

[Girshick, 2015] Girshick, R. (2015). Fast r-cnn. *2015 IEEE International Conference on Computer Vision (ICCV)*.

[Girshick et al., 2014] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition*.

[Gonzalez and Woods, 2014] Gonzalez, R. C. and Woods, R. E. (2014). *Digital image processing*. Dorling Kindersley.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Goodfellow et al., 2017] Goodfellow, I., Bengio, Y., and Courville, A. (2017). *Deep learning*. The MIT Press.

[Goyal et al., 2020] Goyal, B., Dogra, A., Agrawal, S., Sohi, B., and Sharma, A. (2020). Image denoising review: From classical to state-of-the-art approaches. *Information Fusion*, 55:220–244.

[Gravel et al., 2004] Gravel, P., Beaudoin, G., and DeGuise, J. (2004). A method for modeling noise in medical images. *IEEE Transactions on Medical Imaging*, 23:1221–1232.

[Gupta and Sunkaria, 2017] Gupta, S. and Sunkaria, R. K. (2017). Real-time salt and pepper noise removal from medical images using a modified weighted average filtering. *2017 Fourth International Conference on Image Information Processing (ICIIP)*.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*.

[ITU-R, 2015] ITU-R (2015). Parameter values for the hdtv standards for production and international programme exchange.

[Ivakhnenko, 1971] Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(4):364–378.

[Khaled et al., 2021] Khaled, R., Helal, M., Alfarghaly, O., Mokhtar, O., Elkorany, A., El Kassas, H., and Fahmy, A. (2021).

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Krizhevsky et al., 2017] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

[LeCun et al., 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551.

[Mann et al., 2022] Mann, R. M., Athanasiou, A., Baltzer, P. A., Camps-Herrero, J., Clauser, P., Fallenberg, E. M., Forrai, G., Fuchsjäger, M. H., Helbich, T. H., Killburn-Toppin, F., and et al. (2022). Breast cancer screening in women with extremely dense breasts recommendations of the european society of breast imaging (eusobi). *European Radiology*, 32(6):4036–4045.

[Marr and Hildreth, 1980] Marr, D. and Hildreth, E. (1980). Theory of edge detection. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 207(1167):187–217.

[Mehnert and Jackway, 1997] Mehnert, A. and Jackway, P. (1997). An improved seeded region growing algorithm. *Pattern Recognition Letters*, 18:1065–1071.

[Morin and Mahesh, 2018] Morin, R. and Mahesh, M. (2018). Role of noise in medical imaging. *Journal of the American College of Radiology*, 15:1309.

[Nazari and Mukherjee, 2018] Nazari, S. S. and Mukherjee, P. (2018). An overview of mammographic density and its association with breast cancer. *Breast Cancer*, 25(3):259–267.

[Nirthika et al., 2022] Nirthika, R., Manivannan, S., Ramanan, A., and Wang, R. (2022). Pooling in convolutional neural networks for medical image analysis: A survey and an empirical study. *Neural Computing and Applications*, 34(7):5321–5347.

[Phung and Rhee, 2019] Phung and Rhee (2019). A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets. *Applied Sciences*, 9(21):4500.

[Power et al., 2016] Power, S. P., Moloney, F., Twomey, M., James, K., O'Connor, O. J., and Maher, M. M. (2016). Computed tomography and patient risk: Facts, perceptions and uncertainties. *World Journal of Radiology*, 8:902.

[Qi and Snyder, 1998] Qi, H. and Snyder, W. E. (1998). Lesion detection and characterization in digital mammography by bezier histograms. *Proceedings of the 20th Annual International Conference of the IEEE Engineering in Medicine and Biology Society. Vol.20 Biomedical Engineering Towards the Year 2000 and Beyond (Cat. No.98CH36286)*, 2:1021–1024 vol.2.

[Ren et al., 2017] Ren, S., He, K., Girshick, R., and Sun, J. (2017). Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149.

[Sadiq, 2019] Sadiq, R. (2019). Cnn backpropagation.

[Sadli, 2023] Sadli, R. (2023). Intersection over union (iou): A comprehensive guide.

[Sahraei et al., 2021] Sahraei, A., Chamorro, A., Kraft, P., and Breuer, L. (2021). Application of machine learning models to predict maximum event water fractions in streamflow. *Frontiers in Water*, 3.

[Sauvola and Pietikäinen, 2000] Sauvola, J. and Pietikäinen, M. (2000). Adaptive document image binarization. *Pattern Recognition*, 33(2):225–236.

[Scherer et al., 2010] Scherer, D., Müller, A., and Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. *Artificial Neural Networks – ICANN 2010*, page 92–101.

[Seo et al., 2020] Seo, H., Badiei Khuzani, M., Vasudevan, V., Huang, C., Ren, H., Xiao, R., Jia, X., and Xing, L. (2020). Machine learning techniques for biomedical image segmentation: An overview of technical aspects and introduction to state-of-art applications. *Medical Physics*, 47.

[Shih and Cheng, 2005] Shih, F. Y. and Cheng, S. (2005). Automatic seeded region growing for color image segmentation. *Image and Vision Computing*, 23:877–886.

[Smith and Webb, 2011] Smith, N. and Webb, A. (2011). *Introduction to medical imaging : physics, engineering and clinical applications*. Cambridge University Press.

[Snyder and Qi, 2017] Snyder, W. E. and Qi, H. (2017). *Fundamentals of Computer Vision*. Cambridge University Press.

[Szeliski, 2011] Szeliski, R. (2011). *Computer vision : algorithms and applications*. Springer.

[Webb and Flower, 2012] Webb, S. and Flower, M. A. (2012). *Webb's physics of medical imaging*. Taylor & Francis.

[Yamashita et al., 2018] Yamashita, R., Nishio, M., Do, R. K. G., and Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9:611–629.

Appendix A

Python Code of Thresholding

```
# Import modules
import numpy as np
import skimage.io as sio
import matplotlib.pyplot as plt
import skimage.color as col

# Grayscale conversion function
def gray(img):
    gray = col.rgb2gray(img)*255
    gray_int = (np.rint(gray)).astype(int)
    gray_int_clipped = np.clip(gray_int,0,255)
    return gray_int_clipped

# coin_gray = gray(coin)
# Threshold function
def threshold(img, q):
    '''
    Input:
    img: a numpy array of a greyscale image (value ranges form 0
    ↪ - 255)
    q: the threshold
    Output:
    binary: a binary image obtained by applying threshold filter
    ↪ to img
    '''
    # Store the dimension of the img array
    m = len(img)
    n = len(img[0])
    # Create a new array with entries of ones representing the
    ↪ binary image
    binary = np.ones((m,n))*255
    # Loop over all pixels of the image
    for y in range(m):
        for x in range(n):
```

```
32              # if the intensity of the image at pixel (x,y) is
                ↪  less than q
33              # set the intensity of pixel (x,y) of the binary
                ↪  image to 0
34              if img[y,x] <= q:
35                  binary[y,x] = 0
36      # return the binary image
37      return binary
38
39
40  # Function calculating the imbetween variance given a histogram
    ↪  of an image
41  def btw_var(hist, q, m, n):
42      '''
43      Input:
44      hist: histogram of the image
45      q: the threshold
46      m,n: the dimension of the image
47      Output:
48      varb: The inbetween variance
49      '''
50      # Standard calculation of these quantities by definition
51      n0 = 0
52      n1 = 0
53      mu0 = 0
54      mu1 = 0
55      for i in range(0, q+1):
56          n0 += hist[0][i]
57          mu0 += i*hist[0][i]
58      for i in range(q+1,256):
59          n1 += hist[0][i]
60          mu1 += i*hist[0][i]
61      mu0 = mu0/n0
62      mu1 = mu1/n1
63      varb = (1/((m*n)**2))*n0*n1*((mu0 - mu1)**2)
64      return varb
65
66  # Function for finding threshold using otsu method
67  def otsu(img):
68      # Store the dimension of the img array
69      m = len(img)
70      n = len(img[0])
71      # Generate the histogram for the image
72      hist = np.histogram(img, bins = 256)
73      # Create variables to store the maximun inbetween variation
74      # and the optimal threshold q
75      maxvar = 0
```

```
    opt_q = 0
    # Iterate over all q from 0 to 254 to find the optimal q
    for q in range(255):
        varb = btw_var(hist, q, m, n)
        if varb > maxvar:
            maxvar = varb
            opt_q = q
    # Return the optimal q
    return opt_q

# Function of calculating the overall entropy
def H01(hist, q, m, n):
    '''
    Input:
    hist: histogram of the image
    q: the threshold
    m,n: the dimension of the image
    Output:
    The overall entropy
    '''
    # Standard calculation of these quantities by definition
    p = np.array(hist[0])/(m*n)
    H0 = 0
    H1 = 0
    P0 = 0
    P1 = 0
    for i in range(0,q+1):
        P0 += p[i]
    P1 = 1 - P0
    for i in range(0,q+1):
        if p[i] != 0:
            H0 -= (p[i]/P0)*np.log(p[i]/P0)
    for i in range(q+1,256):
        if p[i] != 0:
            H1 -= (p[i]/P1)*np.log(p[i]/P1)
    return H0+H1

# Entropy threshold
def entropy(img):
     # Store the dimension of the img array
    m = len(img)
    n = len(img[0])
    # Generate the histogram for the image
    hist = np.histogram(img, bins = 256)
    # Create variables to store the maximun entropy
    # and the optimal threshold q
    maxH01 = 0
```

```
123     opt_q = 0
124     # Iterate over all q from 0 to 254 to find the optimal q
125     for q in range(255):
126         H_01 = H01(hist, q, m, n)
127         if H_01 > maxH01:
128             maxH01 = H_01
129             opt_q = q
130     # Return the optimal q
131     return opt_q
```

Appendix B

Python Code of Region Growing

```
from PIL import Image, ImageFilter, ImageDraw
import numpy as np
import math
from distinctipy import distinctipy
import skimage.io as sio
import skimage.color as col
import Threshold
from scipy import stats
import statistics as stats
from scipy.stats import t

# save image
img = Threshold.gray(sio.imread("SampleImage.jpg"))
std = np.std(img)

'''create new image of same size as original in order
to look at distinct regions'''
im = Image.new("L", (img.shape[1], img.shape[0]))
pixelsNew = im.load()

'''Each region has a number assigned to it. The pixel map
represents which region each pixel belongs to. A zero
means the pixel has not been assigned yet'''
pixel_map = np.zeros((img.shape[0], img.shape[1]))

region_nr = 0
region_sums = []
sums_squared = []
region_size = []
region_means = []

# Define new region
def new_reg():
```

```
34      global region_nr, region_size, sums_squared, region_sums,
        ↪ region_means
35      region_nr += 1
36      region_size.append(0)
37      sums_squared.append(0)
38      region_sums.append(0)
39      region_means.append(0)
40
41
42  # Add point to region
43  def new_point(loc):
44      global region_nr, region_size, sums_squared, region_sums,
        ↪ pixel_map
45      sums_squared[-1] += img[loc]**2
46      region_sums[-1] += img[loc]
47      region_size[-1] += 1
48      pixel_map[loc[0], loc[1]] = region_nr
49
50
51  def adjacents(loc):
52      # define list of 8 adjacent locations (including potential
        ↪ ghost pixels outside image)
53      ghost_indices = np.array(
54          loc) + np.array([[0, 1], [1, 0], [1, 1], [0, -1], [0, -1],
            ↪ [-1, -1], [1, -1], [-1, 1]])
55
56      # restrict ghost_indices to locations that lie within image
57      adjacents = [tuple(x) for x in ghost_indices if (
58          0 <= x[0] < img.shape[0]) and (0 <= x[1] < img.shape[1])
            ↪ and (pixel_map[x[0], x[1]] == 0)]
59
60      return adjacents
61
62
63  def z_test(px1):
64      x = float(img[(px1[0], px1[1])])
65      mean = region_sums[-1] / region_size[-1]
66      n = region_size[-1]
67      var = (sums_squared[-1] - n * (mean**2)) / (n - 1)
68      return (abs(mean - x) < 1.96 * std)
69
70
71  while len(np.where(pixel_map == 0)[0]) != 0:
72      # set seed to first element in first row where there is a
        ↪ zero (an unassigned point)
73      seed = (np.where(pixel_map == 0)[0][0], np.where(pixel_map ==
        ↪ 0)[1][0])
```

```
74      new_reg()
75
76      # add unassigned neighbours to region
77      current_reg = [seed] + [i for i in adjacents(seed) if
        ↪  pixel_map[i] == 0]
78
79      # set region nr of new_reg to the new number
80      for pixel in current_reg:
81          new_point(pixel)
82
83      # we must visit neighbours of seed
84      to_visit = current_reg[1:].copy()
85
86      # loop over elements in to_visit
87      while len(to_visit) != 0:
88
89          # Look at adjacent elements
90          for i in adjacents(to_visit[0]):
91
92              '''if the difference in grayscale value is
                ↪  sufficiently small and the adjacent pixel
93              does not already belong to region, append it to
    ↪  to_visit and change its region value'''
94              if z_test(i) and pixel_map[i] == 0:
95                  to_visit.append(i)
96                  new_point(i)
97
98          # remove first element from to_visit as it has already
            ↪  been checked
99          to_visit = to_visit[1:]
100
101     # update mean
102     region_means[-1] = region_sums[-1] / region_size[-1]
103
104 # get number of regions found
105
106 # Display regions with mean of region intensity in place of
    ↪  actual intensity
107 for i in range(img.shape[0]):
108     for j in range(img.shape[1]):
109         number = int(pixel_map[i, j] - 1)
110         pixelsNew[j, i] = int(region_means[number])
111
112
113 im.show()
```

Appendix C

Python Code of Binary Operations

```
# Import Modules
from PIL import Image, ImageFilter, ImageDraw
import numpy as np
from distinctipy import distinctipy
import skimage.io as sio
import skimage.color as col
from scipy import stats
import statistics as stats
from scipy.stats import t
import skimage.morphology as mor
from skimage.segmentation import (mark_boundaries,
    find_boundaries,
                                  clear_border)
from scipy.signal import convolve2d
import matplotlib.pyplot as plt

# Function calculating the imbetween variance given a histogram
    of an image
def btw_var(hist, q, m, n):
    '''
    Input:
    hist: histogram of the image
    q: the threshold
    m,n: the dimension of the image
    Output:
    varb: The inbetween variance
    '''
    # Standard calculation of these quantities by definition
    n0 = 0
    n1 = 0
    mu0 = 0
    mu1 = 0
```

```
    for i in range(0, q+1):
        n0 += hist[0][i]
        mu0 += i*hist[0][i]
    for i in range(q+1,256):
        n1 += hist[0][i]
        mu1 += i*hist[0][i]
    mu0 = mu0/n0
    mu1 = mu1/n1
    varb = (1/((m*n)**2))*n0*n1*((mu0 - mu1)**2)
    return varb

# Function for finding threshold using otsu method with mask
def otsu_mask(img,mask):
    # Store the dimension of the img array
    m = len(img)
    n = len(img[0])
    # Generate the histogram for the image with mask
    hist = masked_histogram(img, mask)
    # Create variables to store the maximun inbetween variation
    # and the optimal threshold q
    maxvar = 0
    opt_q = 0
    # Iterate over all q from 0 to 254 to find the optimal q
    for q in range(255):
        varb = btw_var(hist, q, m, n)
        if varb > maxvar:
            maxvar = varb
            opt_q = q
    # Return the optimal q
    return opt_q

# Function of calculating the overall entropy
def H01(hist, q, m, n):
    '''
    Input:
    hist: histogram of the image
    q: the threshold
    m,n: the dimension of the image
    Output:
    The overall entropy
    '''
    # Standard calculation of these quantities by definition
    p = np.array(hist[0])/(m*n)
    H0 = 0
    H1 = 0
    P0 = 0
    P1 = 0
```

```
78      for i in range(0,q+1):
79          P0 += p[i]
80      P1 = 1 - P0
81      for i in range(0,q+1):
82          if p[i] != 0:
83              H0 -= (p[i]/P0)*np.log(p[i]/P0)
84      for i in range(q+1,256):
85          if p[i] != 0:
86              H1 -= (p[i]/P1)*np.log(p[i]/P1)
87      return H0+H1
88
89  # Masked entropy thresholding
90  def entropy_mask(img,mask):
91       # Store the dimension of the img array
92      m = len(img)
93      n = len(img[0])
94      # Generate the histogram for the image with mask
95      hist = masked_histogram(img, mask)
96      # Create variables to store the maximun entropy
97      # and the optimal threshold q
98      maxH01 = 0
99      opt_q = 0
100     # Iterate over all q from 0 to 254 to find the optimal q
101     for q in range(255):
102         H_01 = H01(hist, q, m, n)
103         if H_01 > maxH01:
104             maxH01 = H_01
105             opt_q = q
106     # Return the optimal q
107     return opt_q
108
109 # Masked entropy thresholding
110 def mask_edge_link(mask):
111     newmask = mask.copy()
112     m = len(mask)
113     n = len(mask[0])
114     mask_bin = np.clip(mask,0,1)
115     le_mean_u = np.mean(mask_bin[0,:int(n/2)])
116     le_mean_d = np.mean(mask_bin[0,int(n/2):])
117     re_mean_u = np.mean(mask_bin[m-1,:int(n/2)])
118     re_mean_d = np.mean(mask_bin[m-1,int(n/2):])
119     ue_mean_l = np.mean(mask_bin[:int(m/2),0])
120     ue_mean_r = np.mean(mask_bin[int(m/2):,0])
121     de_mean_l = np.mean(mask_bin[:int(m/2),n-1])
122     de_mean_r = np.mean(mask_bin[int(m/2):,n-1])
123
124     if le_mean_u < 0.5 and le_mean_d < 0.5:
```

```
125         newmask[:5,:] = 0
126     if re_mean_u < 0.5 and re_mean_d < 0.5:
127         newmask[m-5:,:] = 0
128     if ue_mean_l < 0.5 and ue_mean_r < 0.5:
129         newmask[:,:5] = 0
130     if de_mean_l < 0.5 and de_mean_r < 0.5:
131         newmask[:,n-5:] = 0
132     return newmask
133 
134 # Link up edges of the mask by splitting each edge into two
135 # and link the edge if each half has at least 50% black pixels
136 def mask_expand(mask, radius):
137     m = len(mask)
138     n = len(mask[0])
139     ker = mor.disk(radius)
140     binary_mask = np.clip(mask,0,1)
141     expanded_mask = binary_mask.copy()
142     mask_bound = find_boundaries(mask)
143     i = 0
144     for y in range(m):
145         for x in range(n):
146             if mask_bound[y,x]:
147                 lbound = max(0,x-radius)
148                 rbound = min(n,x+radius+1)
149                 ubound = max(0,y-radius)
150                 dbound = min(m,y+radius+1)
151                 klb = abs(min(0,x-radius))
152                 krb = radius+1 + min(radius,abs((n-1)-x))
153                 kub = abs(min(0,y-radius))
154                 kdb = radius+1 + min(radius,abs((m-1)-y))
155                 expanded_mask[ubound:dbound,lbound:rbound] -=
                    ↪  ker[kub:kdb,klb:krb]
156     expanded_mask_clipped = np.clip(expanded_mask,0,1)
157     return expanded_mask_clipped*255
```

Appendix D

Python Code of Mean Shift

```
from PIL import Image, ImageFilter, ImageDraw
import numpy as np
import skimage.io as sio

#import image
img = sio.imread("sample_gray_image.jpg")

#define new image in which regions will be displayed
im = Image.new("L", (img.shape[1], img.shape[0]))
pixelsNew = im.load()

#Define feature matrix that will contain normalised features of
  ↪  each pixel
feature_matrix = np.zeros((img.shape[0], img.shape[1], 3))
for i in range(img.shape[0]):
    feature_matrix[i, :, 0] = i / img.shape[0]

for j in range(img.shape[1]):
    feature_matrix[:, j, 1] = j / img.shape[1]

#list of all pixels
to_check = []

for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        feature_matrix[i, j, 2] = img[i, j] / 255
        to_check.append((i, j))

#preliminary threshold that will reduce cost of searching through
  ↪  feature space
sqrt_01 = np.sqrt(0.4)

#Threshold/bandwidth values
d = 0.1
```

```
rho = 0.05
epsilon = 0.01

#Define function that finds pixels within certain
#euclidean distance of input p_n in feature space
def adjacents(p_n):
    lb_y = max(int((p_n[0] - sqrt_01) * img.shape[0]), 0)
    ub_y = min(int((p_n[0] + sqrt_01) * img.shape[0]),
    ↪  img.shape[0])

    lb_x = max(int((p_n[1] - sqrt_01) * img.shape[1]), 0)
    ub_x = min(int((p_n[1] + sqrt_01) * img.shape[1]),
    ↪  img.shape[1])

    nbrs = feature_matrix[lb_y:ub_y, lb_x:ub_x, :]

    space = []
    friends = []
    for n in range(nbrs.shape[0]):
        for m in range(nbrs.shape[1]):
            b = nbrs[n, m, :]
            dot = np.dot(p_n - b, p_n - b)
            if dot < d:
                space.append(b)
            if dot < rho:
                friends.append((int(lb_y + n), int(lb_x + m)))

    return np.array(space), friends


#Run while loope while there are still
# unassigned pixels
while len(to_check) > 0:
    point = to_check[0]

    #initiate a centroid
    current = feature_matrix[point[0], point[1], :]
    adj = adjacents(current)


    adj_this = adj[0] #pixels lying within distance d
    friends_big = set(adj[1]) #pixels lying with distance rho

    if len(adj_this) == 0:
        pixelsNew[point[1], point[0]] = int(current[2] * 255)
        to_check = to_check[1:]
        continue
```

```
78
79      next_ = np.mean(adj_this, axis=0)
80
81      #update centroid until convergence
82      while np.dot(current - next_, current - next_) > epsilon**2:
83          current = next_
84          adj = adjacents(current)
85          adj_this = adj[0]
86          friends_big = friends_big.union(set(adj[1]))
87          next_ = np.mean(adj_this, axis=0)
88
89      #subtract from to_check
90      to_check = list(set(to_check) - friends_big)
91
92      #assign to all pixels that fell within basin of attraction
93      #the rounded intensity corresponding to the location of
        ↪  converged centroid
94      for loc in friends_big:
95          pixelsNew[loc[1], loc[0]] = int(next_[2] * 255)
96
97
98  #im.save("mean_shift.jpg")
99  im.show()
```

Appendix E

Python Code of K-Means

```
from PIL import Image, ImageFilter, ImageDraw
import numpy as np
import skimage.io as sio

# number of clusters
k = 4

# import image and define image on which to
# display intensities of clusters
img = sio.imread("Lungexample.jpeg")
im = Image.new("L", (img.shape[1], img.shape[0]))
pixelsNew = im.load()

# Map assigning to each pixel the number of nearest mean
meanMap = np.zeros((img.shape[0], img.shape[1]))

# Construct feature matrix
feature_matrix = np.zeros((img.shape[0], img.shape[1], 3))
for i in range(img.shape[0]):
    feature_matrix[i, :, 0] = i / img.shape[0]

for j in range(img.shape[1]):
    feature_matrix[:, j, 1] = j / img.shape[1]

for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        feature_matrix[i, j, 2] = img[i, j] / 255

# Function to find nearest mean for each pixel and label meanMap
  ↪ correspondingly


def nearest_mean(means):
    for i in range(img.shape[0]):
```

```
34          for j in range(img.shape[1]):
35              meanMap[i, j] = np.argmin([np.dot(
36                  (feature_matrix[i, j, :] - means[k_]),
                    ↪ (feature_matrix[i, j, :] - means[k_])) for k_
                    ↪ in range(len(means))])
37
38  # Function that updates position of each mean by taking average
      ↪ position of all
39  # pixels assigned to it. Removes cluster K_i if there is no pixel
      ↪ for which K_i
40  # is the closest cluster
41
42
43  def update(means):
44      means_ = []
45      means_saved = []
46      for k_ in range(k):
47          if len(feature_matrix[meanMap == k_]) > 0:
48              means_.append(
49                  np.mean(feature_matrix[meanMap == k_], axis=0))
50              means_saved.append(means[k_])
51
52      shift = np.array(means_saved) - np.array(means_)
53
54      return np.array(means_), np.max([np.dot(shift[l, :], shift[l,
          ↪ :]) for l in range(len(means_))])
55
56
57  max_shift = 3
58
59  np.random.seed(0)
60  means = np.random.rand(k, 3)
61  nearest_mean(means)
62
63
64  while max_shift > 0.001:
65      upd = update(means)
66      means = upd[0]
67      max_shift = upd[1]
68      nearest_mean(means)
69
70  # in new image, assign to each pixel the intensity value of the
      ↪ corresponding mean
71  for i in range(img.shape[0]):
72      for j in range(img.shape[1]):
73          pixelsNew[j, i] = int(means[int(meanMap[i, j])][2] * 255)
74
```

```
75 # im.save("sample_grayscale_image.jpg")
76 im.show()
```

Appendix F

Python Code of Faster R-CNN

```
import pandas as pd
import numpy as np
import torchvision
import torch
from torchvision.models.detection.faster_rcnn import
    FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import
    MaskRCNNPredictor
import ast
import glob
import os
from torchvision import transforms as T
from utils import collate_fn
from PIL import Image
import shutil
#!pip install natsort
from natsort import natsorted
import math
import matplotlib.pyplot as plt
from matplotlib import patches
import cv2
from engine import train_one_epoch, evaluate
from google.colab.patches import cv2_imshow

# Sample from torchvision
# Segmentations preprocessing to bounding boxes
segmentations = pd.read_csv("filtered_segmentations.csv")

# List and indices of image names that match filtered
    segmentations.
segmentation_names = segmentations['#filename'].to_list()
image_names =
    natsorted(os.listdir("/content/drive/MyDrive/Diss/Images
    filtered"))
```

```
imageidx_filtered = []
imagenames_filtered = []
for image_idx in range(len(image_names)):
  if (image_names[image_idx] in segmentation_names):
    imageidx_filtered.append(image_idx)
    imagenames_filtered.append(image_names[image_idx])


final_segmentations = segmentations[segmentations['#filename']. \
isin(imagenames_filtered)].reset_index()

# Define PyTorch image transforms
def get_transform():
    transforms = []
    transforms.append(T.PILToTensor())
    transforms.append(T.ConvertImageDtype(torch.float))
    return T.Compose(transforms)

# Create dataset class that inherits from Pytorch Dataset class
class CancerDataset(torch.utils.data.Dataset):
    def __init__(self, img_root, segmentations, transform):
        self.img_root = img_root
        self.transform = transform
        self.segmentations = segmentations

        self.imgs = list(os.listdir(img_root))

    def __getitem__(self, idx):
        image_path = os.path.join(self.img_root, self.imgs[idx])
        img = Image.open(image_path)

        # get bounding box coordinates for each mask
        boxes = []
        coords_dict =
        ↪  ast.literal_eval(self.segmentations.region_shape_attributes[idx])
        seg_names = self.segmentations['#filename'][idx]
        max_x = max(coords_dict['all_points_x'])
        min_x = min(coords_dict['all_points_x'])
        max_y = max(coords_dict['all_points_y'])
        min_y = min(coords_dict['all_points_y'])


        box = [min_x, min_y, max_x, max_y]
        boxes.append(box)

        # convert to torch tensor
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
```

```
76          # area of the bounding boxes
77          area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] -
            ↪  boxes[:, 0])
78          # no crowd instances
79          iscrowd = torch.zeros((boxes.shape[0],),
            ↪  dtype=torch.int64)
80          # labels to tensor
81          labels = torch.ones((boxes.shape[0],), dtype=torch.int64)
82
83          image_id = torch.tensor([idx])
84
85          # prepare the final `target` dictionary
86          target = {}
87          target["boxes"] = boxes
88          target["labels"] = labels
89          target["area"] = area
90          target["iscrowd"] = iscrowd
91          target["image_id"] = image_id
92          #print(seg_names)
93          #print(image_path)
94
95          if self.transform is not None:
96              img = self.transform(img)
97
98          return img, target
99
100     def __len__(self):
101         return len(self.segmentations)
102
103 # Define model instance
104 def get_model_instance_segmentation(num_classes):
105     # load Faster RCNN pre-trained model
106     model =
        ↪  torchvision.models.detection.fasterrcnn_mobilenet_v3_large_fpn
        ↪  \
107     (pretrained=True, trainable_backbone_layers = 1)
108     # get the number of input features
109     in_features =
        ↪  model.roi_heads.box_predictor.cls_score.in_features
110     # define a new head for the detector with required number of
        ↪  classes
111     model.roi_heads.box_predictor = FastRCNNPredictor(in_features,
        ↪  num_classes)
112     return model
113
114 from engine import train_one_epoch, evaluate
115
```

```
116
117
118
119 # train on the GPU or on the CPU, if a GPU is not available
120 device = torch.device('cuda') if torch.cuda.is_available() else
    ↪ torch.device('cpu')
121
122 # our dataset has two classes only - background and tumor
123 num_classes = 2
124 # use our dataset and defined transformations
125 dataset = CancerDataset("//content/drive/MyDrive/Diss/Images
    ↪ filtered",
126                         final_segmentations, get_transform())
127 dataset_test = CancerDataset("/content/drive/MyDrive/Diss/Images
    ↪ filtered",
128                              final_segmentations,
                                 ↪ get_transform())
129
130 # split the dataset in train and test set
131 indices = list(range(len(dataset)))
132 dataset = torch.utils.data.Subset(dataset, indices[:-30])
133 dataset_test = torch.utils.data.Subset(dataset_test,
    ↪ indices[-30:])
134
135 # define training and validation data loaders
136 data_loader = torch.utils.data.DataLoader(
137     dataset, batch_size=8, shuffle=False, num_workers=4,
138     collate_fn=collate_fn)
139
140 data_loader_test = torch.utils.data.DataLoader(
141     dataset_test, batch_size=1, shuffle=False, num_workers=4,
142     collate_fn=collate_fn)
143
144 # get the model using our helper function
145 model = get_model_instance_segmentation(num_classes)
146
147 # move model to the right device
148 model.to(device)
149
150 # construct an optimizer
151 params = [p for p in model.parameters() if p.requires_grad]
152
153
154 optimizer = torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999),
    ↪ eps=1e-08, weight_decay=0,
```

```
                                amsgrad=False, foreach=None,
                                ↪ maximize=False,
                                ↪ capturable=False,
                                differentiable=False, fused=None)


lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=4,
                                               gamma=0.1)
# let's train it for 5 epochs
num_epochs = 10

for epoch in range(num_epochs):
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model, optimizer, data_loader,
    device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model, data_loader_test, device=device)

print("That's it!")

# pick one image from the test set
z = 1

img, gt = dataset_test[z]
# put the model in evaluation mode
model.eval()
with torch.no_grad():
    prediction = model([img.to(device)])

from google.colab.patches import cv2_imshow

# Plot image itself
img_path = os.path.join("/content/drive/MyDrive/Diss/Images
↪ filtered",

                        ↪ os.listdir("/content/drive/MyDrive/Diss/Images
                        ↪ filtered")[-30:][z])

img = cv2.imread(img_path)


# Plot GT bounding box
coords_dict = ast.literal_eval(list( \
final_segmentations.iloc[-30:].region_shape_attributes)[z])
```

```
max_x = max(coords_dict['all_points_x'])
min_x = min(coords_dict['all_points_x'])
max_y = max(coords_dict['all_points_y'])
min_y = min(coords_dict['all_points_y'])
# Add into a list for IoU calculation
gt_bb = [min_x, max_y, max_x, min_y]
img = cv2.rectangle(img,(min_x,min_y),(max_x,max_y),(0,255,0), 5)

# Plot pred bounding box
max_x_p = int(list(prediction[0].values())[0][0][2])
min_x_p = int(list(prediction[0].values())[0][0][0])
max_y_p = int(list(prediction[0].values())[0][0][3])
min_y_p = int(list(prediction[0].values())[0][0][1])
# Add into a list for IoU calculation
pred_bb = [min_x_p, max_y_p, max_x_p, min_y_p]
img =
    ↪  cv2.rectangle(img,(min_x_p,min_y_p),(max_x_p,max_y_p),(0,0,255),
    ↪  5)
cv2_imshow(img)

def get_iou(gt, preds):
    """
    Implement the intersection over union (IoU) between box1 and
    ↪  box2
    """
    # ymin, xmin, ymax, xmax = box

    y11, x11, y21, x21 = gt
    y12, x12, y22, x22 = preds

    yi1 = max(y11, y12)
    xi1 = max(x11, x12)
    yi2 = min(y21, y22)
    xi2 = min(x21, x22)
    inter_area = max(((xi2 - xi1) * (yi2 - yi1)), 0)
    # Calculate the Union area by using Formula: Union(A,B) = A +
    ↪  B - Inter(A,B)
    box1_area = (x21 - x11) * (y21 - y11)
    box2_area = (x22 - x12) * (y22 - y12)
    union_area = box1_area + box2_area - inter_area
    # compute the IoU
    iou = inter_area / union_area
    return iou

# put the model in evaluation mode
model.eval()
ious = []
```

```
for i in range(30):
  # pick one image from the test set
  img, gt = dataset_test[i]
  with torch.no_grad():
      prediction = model([img.to(device)])
      coords_dict = ast.literal_eval(list( \
      final_segmentations.iloc[-30:].region_shape_attributes \
      )[i])
      max_x = max(coords_dict['all_points_x'])
      min_x = min(coords_dict['all_points_x'])
      max_y = max(coords_dict['all_points_y'])
      min_y = min(coords_dict['all_points_y'])
      # Add into a list for IoU calculation
      gt_bb = [min_y, min_x, max_y, max_x]

      #Plot pred bounding box
      max_x_p = int(list(prediction[0].values())[0][0][2])
      min_x_p = int(list(prediction[0].values())[0][0][0])
      max_y_p = int(list(prediction[0].values())[0][0][3])
      min_y_p = int(list(prediction[0].values())[0][0][1])
      # Add into a list for IoU calculation
      pred_bb = [min_y_p, min_x_p, max_y_p, max_x_p]

      iou = get_iou(gt_bb, pred_bb)
      ious.append(iou)
```