PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru – 560 085, Karnataka, India

Report on

'LUNG CANCER DETECTION AND CLASSIFICATION USING IMAGE PROCESSING AND CNN-DEEP LEARNING ARCHITECTURE'

Submitted by

Rhitesh Kumar Singh (PES1201700254)
Rajath Gadagkar (PES1201700136)
Dhruv K.C (PES1201701265)
Nitin Kumar P (PES1201701837)

August 2020 - May 2021

under the guidance of

Internal Guide

Prof. H R Vanamala
Associate Professor
Department of ECE
PES University
Bengaluru -560085

FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGG
PROGRAM B.TECH

CERTIFICATE

This is to certify that the Report entitled

'LUNG CANCER DETECTION AND CLASSIFICATION USING IMAGE PROCESSING AND CNN-DEEP LEARNING ARCHITECTURE'

is a bonafide work carried out by

Rhitesh Kumar Singh (PES1201700254)
Rajath Gadagkar (PES1201700136)
Dhruv K.C (PES1201701265)
Nitin Kumar P (PES1201701837)

In partial fulfillment for the completion of 8th semester course work in the Program of Study B.Tech in Electronics and Communication Engineering, under rules and regulations of PES University, Bengaluru during the period Aug 2020 – May 2021. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The report has been approved as it satisfies the 8th semester academic requirements in respect of Capstone project work.

Signature with date & Seal
(Prof H R Vanamala.)
Internal Guide

Signature with date & Seal
Dr. Anuradha M
Chairperson

Signature with date & Seal
Dr. B. K. Keshavan
Dean - Faculty of Engg. &Technology

Name and signature of the examiners:
1.

2.

DECLARATION

We, Rhitesh Kumar Singh, Rajath Gadagkar, Dhruv K C, Nitin Kumar Pandit**,** hereby declare that the report entitled, '***Lung Cancer Detection and Classification using Digital Image Processing and CNN-Deep Learning Architecture',*** is an original work done by us under the guidance of **Prof. Vanamala H R**, *Associate Professor*, ECE Department and is being submitted in partial fulfillment of the requirements for completion of 8th Semester course work in the Program of Study, B.Tech in Electronics and Communication Engineering.

PLACE: Bangalore
DATE: 23/12/2020

NAME AND SIGNATURE OF THE CANDIDATES

1. Rhitesh Kumar Singh, 

2. Rajath Gadagkar 

3. Dhruv K C 

4. Nitin Kumar Pandit 

ABSTRACT

A large number of cancer deaths in the world is due to lung cancer, which is caused due to unbalanced cell growth. Through our work, a CNN-DEEP learning model is proposed with the help of image pre-processing techniques for detecting and classifying lung cancer in a dataset we aim to help in the diagnosis of the patient's cases: benign, or malignant.

Here we use the Kaggle Lung CT Challenge dataset and employ different image pre-processing techniques like Median filter to remove salt and pepper noise, Gaussian filter to remove high frequency noises and Adaptive Histogram Equalization to increase the contrast of the image. We utilize our 2D CNN model for lung cancer detection and classification and acquire a superior evaluation of the model. We divide our pre-processed dataset into 80%, 10% and 10% for training, validation and testing respectively, and obtain testing accuracy of 99.28% and precision of 1, recall of 0.99, and F1-score of 1.

ACKNOWLEDGEMENT

We would like to express our gratitude and thanks to all those who made this project possible. We sincerely thank our internal guide Prof. H R Vanamala, Professor of the Department of Electronics and Communication Engineering for providing us the opportunity to take up this project and providing us with support, guidance and motivation which helped us to complete our project. We are also grateful to Prof. Rajini M and Prof. Karpagavalli P for providing us useful insights on our project which helped us improve our project further. We would also like to sincerely thank Prof. Anuradha M, Head of the Department of Electronics and Communication Engineering for her encouragement and support. During the course of this project, we have learnt a lot from various sources and are indebted to all.

Table of Contents

Table of Figures:

Table of Contents: Tables

LUNG CANCER DETECTION AND CLASSIFICATION USING IMAGE PROCESSING AND CNN-DEEP LEARNING ARCHITECTURE

Chapter 1:

1.1 Introduction:

Hereditary factors, as well as exposure to radon gas, asbestos, second-hand smoke, or other forms of air pollution, all contribute to the development of lung cancer. Cigarette smoking and smoke exposure will increase the risk of contracting lung cancer. Lung cancer can develop too for people who have a history of exposure to inhaled chemicals or other toxins. Even if you were exposed to chemicals and other pollutants a long time ago, it could produce lung cell abnormalities that lead to cancer.

Symptoms of lung cancer do not often appear until the disease has progressed to an advanced stage.

Certain individuals, on the other hand, may detect symptoms that they mistake for those of a lesser serious, acute sickness.

Examples of these symptoms include:

- Loss of appetite
- Voice changes, such as hoarseness
- Chest infections, such as bronchitis and pneumonia, are common.
- lingering cough which will start to worsen
- Asthma
- Headaches that aren't going away
- Loss of weight
- Wheezing

1.2 Problem statement:

- Image processing algorithms and CNN were used to classify and detect lung cancer from CT scan images.
- Comparison of performance with different neural networks based on accuracy.

1.3 Motivation:

Currently, there are many techniques to detect and diagnose lung cancer. Most of the methods used by doctors are a long procedural task and might take some time to diagnose lung cancer. Lung cancer identification at an early stage is both necessary and beneficial to the patient. A low-dose CAT scan or CT scan (LDCT) has been explored on people who are at a greater risk of developing lung cancer in recent years. LDCT scans can aid in the detection of cancerous spots in the lungs. CT scanning also seems to be less harmful than regular chest x-ray scans.

For detecting, an artificial neural network might be employed. of cancer nodules in these CT scan images. A detection model with good accuracy can be used for the early detection of lung cancer as it is the key to increasing survival rates for patients with lung cancer. When cancer is detected early, the chances of effective therapy increase. Certain initial malignancies may have detectable signs and symptoms; however this is not always the case. Following a cancer diagnosis, staging offers critical information about the extent of the disease in the body as well as the expected response to therapy.

Chapter 2:

2.1 Literature Review:

An Effective Lung Cancer Detection And Classification Using Enhanced Fully Convolution Neural Networks

K. Narmada, G. Prabakaran

It is suggested here to use a fully convolutional neural network to categorise lung cancer stages and predict diagnosis (FCN). The methods of this technique can be refined through candidate creation throughout the screening step to produce more optimum outcomes. The focus leads to a learning process over the installation of FCN to deliver outcomes with a huge dataset. The enhanced findings will demonstrate the performance of the suggested system rather than the current efforts.

The CT images are used and also training samples can be identified. Even though the increase of training samples, will give more accurate results over the detection methodology.

In the future, the optimization technique will be used to improve the effectiveness of the detection and classification mechanism. Even more, the treatment prediction will also identify further improvements.

Convolutional Neural Networks Promising in Lung Cancer T-Parameter Assessment on Baseline FDG-PET/CT

Margarita Kirienko, Martina Sollini, Giorgia Silvestri, Serena Mognetti, Emanuele Voulaz, Lidija Antunovic, Alexia Rossi, Luca Antiga, and Arturo Chiti

To create a convolutional neural network (CNN)-based method for classifying lung cancer lesions as T1-T2 or T3-T4 on staging fluorodeoxyglucose positron emission tomography (FDG-PET)/CT images. A CNN architecture with two neural networks, one as a feature extractor and the other as a classifier, was utilised, and the model was trained using a cross-validation technique.

The method, which was built and evaluated, was built of two networks (a "feature extractor" and a "classifier"). It attained accuracy, recall, and specificity, and AUC of 87%, 69%, 69%, and 0.83; 86%, 77%, 70%, and 0.73; and 90%, 47%, 67%, and 0.68 in the training, validation, and test sets, respectively. Finally, a proof of concept was achieved showing CNNs may be utilised as a tool to aid in the staging of lung cancer patients.

Chapter 3:

3.1 Proposed method:

To identify lung cancer nodules, we use image processing methods and convolutional neural networks (CNN). We use image preprocessing to eliminate the noises that exist in the CT-scan images so that we can identify if there are any abnormal masses present in the lungs.

To minimize noise and improve contrast, the image is preprocessed with the Median, Gaussian filters, and Adaptive Histogram Equalization function. Watershed segmentation is then used to extract nodules present in the image. Finally, CNN-Deep Learning is utilized to extract numerous characteristics and spatial features of the image, as well as to categorize the identified nodules as malignant or benign.

3.2 Scope:

This method used with the combination of image processing and deep learning architecture helps us to classify the cancerous nodules as malignant or benign with the help of a single CT scan of the lungs. This thereby is a cost-cutting procedure for the affected patients since they don't have to get multiple scans done for the confirmation of cancerous nodules.

3.3 Block Diagram:

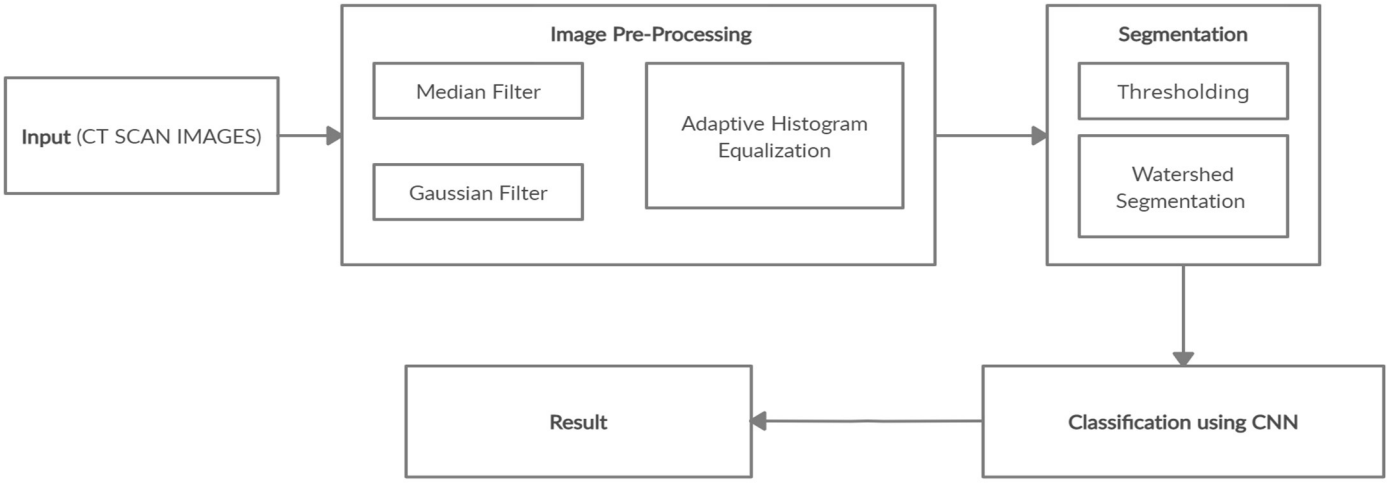


Fig 3.0 Block Diagram of our proposed methodology

3.4 CT Scan Images:

The CT scan images used in this model were taken from the Kaggle dataset. These datasets contain DICOM images. To read these DICOM images in Matlab, first, we have to use the “dicominfo” function which reads the metadata of the DICOM image and stores it in a variable. The name of the image is passed as an argument to this function. Then “dicomread” is used where the dicominfo data is passed as an argument that reads the DICOM image.
The goal of the dataset was to test several techniques for analyzing patterns in CT images that are related to contrast and age of the patient. The fundamental concept is to find visual textures, statistical patterns, and features that are strongly linked to these attributes, then build straightforward tools to automatically recognize these images when they are misclassified(or identifying abnormalities, which could be dubious cases/devices that aren't calibrated properly). It is made up of the middle slices of all CT pictures taken from patients who had valid age, modality, and contrast tags.

3.5 Data Analysis:

1. Age Distribution:

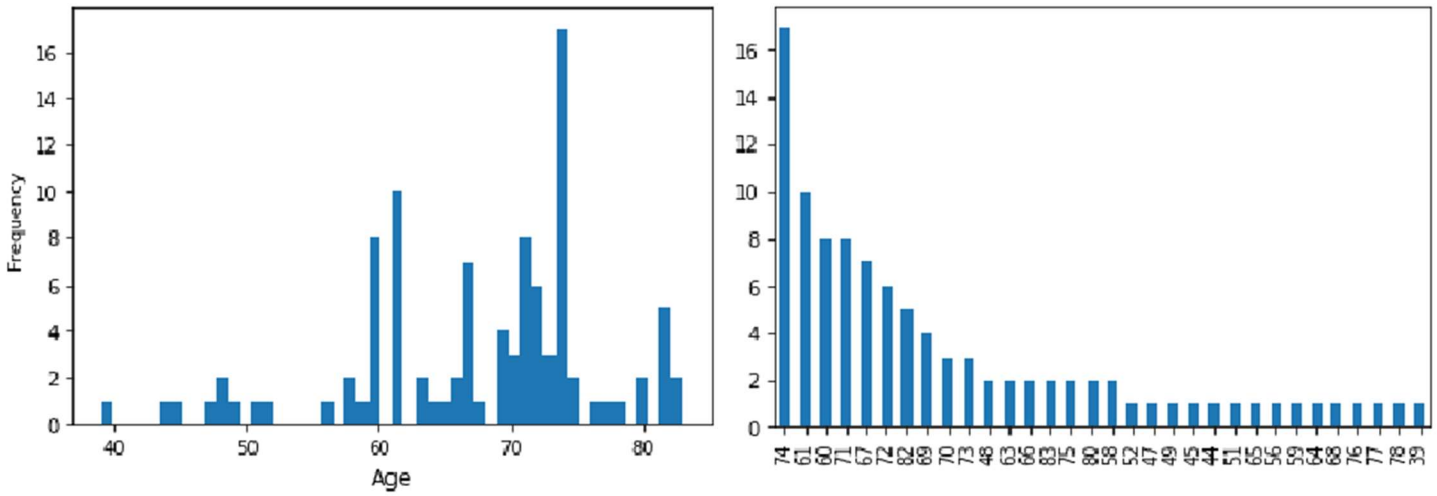


Fig 3.5.1 Age Distribution

The distribution is left skewed. The dataset contains most of the people in the age range of 60-75 years. It seems that cancer is most commonly affected in ages above 60.

2. Patients having cancer and patients not having cancer vs Age Distribution

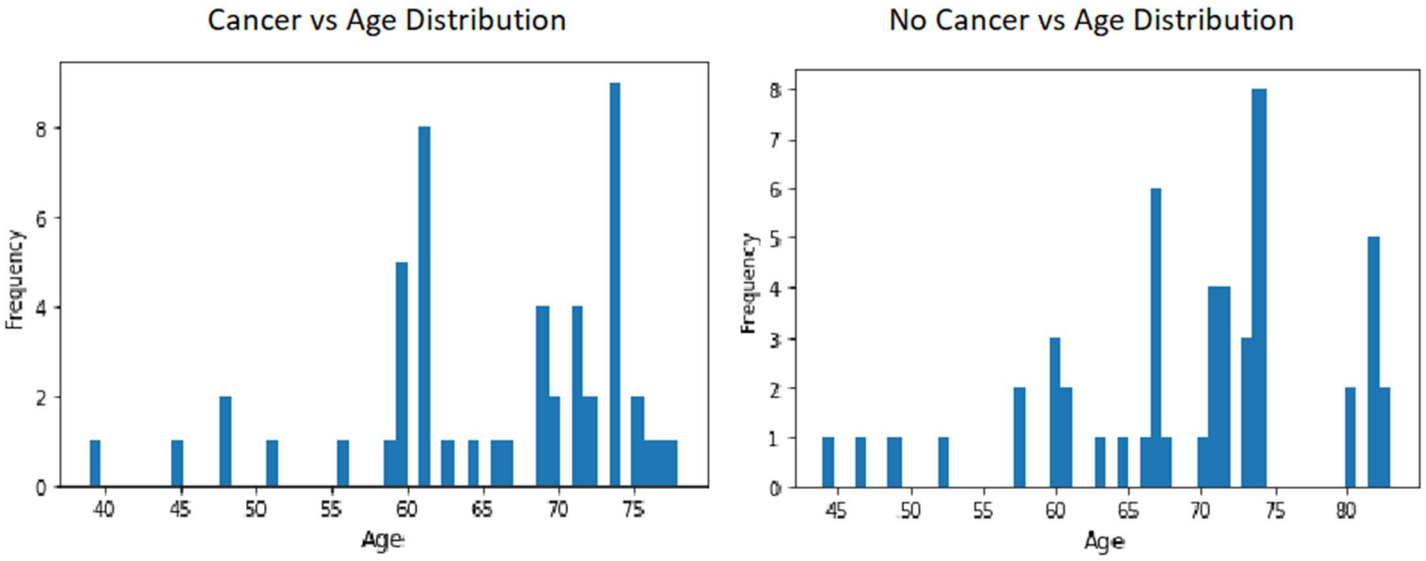


Fig 3.5.2 Cancer vs Age Distribution

Both the plots are left skewed distributions. People above 60+ age have a greater risk of cancer.

3.6 Image processing with the help of filters:

Only at the moment of picture acquisition or transmission is noise added into the image.

The change in brightness or information in images is referred to as image noise. (It refers to the deterioration of the visual signal as a result of external influences.)

SNR (signal to noise ratio) is used to measure the degree of noise in a CT scan image by comparing the frequency of the target signal to the level of background noise (pixels deviating from normal). The greater the signal-to-noise ratio, the lower the noise in the picture.

Types of noises present in CT scan images: Gaussian Noise, Salt Noise, Pepper Noise.

Gaussian Noise: Poor illumination, high temperature, transmission can be the sources of Gaussian noise in CT scan images.

The probability density function of a Gaussian random variable Z is given by:

$$P_G(Z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(Z-\mu)^2}{2\sigma^2}}$$ .... Eq 3.6 Probability density function of a Gaussian RV

Where **z** stands for the grey level, $\boldsymbol{\mu}$ stands for the mean value and $\boldsymbol{\sigma}$ stands for the SD.

The Gaussian noise exhibits both a random and a normal distribution of instantaneous amplitudes over time.

Salt and pepper noise are also known as impulse noise. This noise is typically created by or happens as a result of a sharp and rapid disruption in the visual signal. It generally appears as a randomly distributed white or black (or even both) pixel throughout the image.

Salt Noise: Salt noises are added by sprinkling random bright (255-pixel value) values across an image.
Pepper Noise: Pepper noise is induced by sprinkling random black (0-pixel) values across an image.
Therefore, to reduce the noise that is present in the CT scan image, the following filters were used

1. Median Filter
2. Gaussian Filter
3. Adaptive Histogram Equalization Function

3.6.1 Median Filter:

The median filter evaluates each pixel in the image and compares it to its neighbours to see if it is reflective of its surroundings, then substitutes it with the median of those values. The median is computed by arranging all of the pixel values in the surrounding neighbourhood in ascending order and then substituting the pixel under consideration with the center pixel value. If the number of pixels in the neighbourhood are even, the average of the two center pixel values are used.
The below figure illustrates an example calculation:

123	125	126	130	140
122	124	126	127	135
118	120	150	125	134
119	115	119	123	133
111	116	110	120	130

Fig 3.6.1 Median Filter

Neighborhood values: (arranged in ascending order)
115,119,120,123,124,125,126,127,150

Since 124 is the middle value, therefore the median value is: 124

From the above figure we see that the center pixel value is 150 that does not represent the surrounding /neighborhood pixels so it is substituted with the median value: 124

The median filter is a non-linear filter that is commonly used to reduce noise from an image or signal.

Outliers are especially effectively removed using median filters. Outliers, sometimes known as "salt and pepper" noise, are frequently present as a result of bit errors in transmission or produced during the signal acquisition stage.

3.6.2 Gaussian Filter:

The Gaussian function is represented by:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \text{...Eq 3.6.2 Gaussian Function}$$

Where $\boldsymbol{\sigma}$ represents the SD with zero mean.

The Gaussian function is employed in a variety of fields of study, including:

1. It establishes a probability distribution for noise or data.

2.It functions as a smoothing operator.

Convoluting an image with a Gaussian function is the same as applying a Gaussian blur. Since a Gaussian's fourier transform is another Gaussian, applying a Gaussian blur reduces the image's high-frequency components; hence, a Gaussian blur is a low pass filter.

The pixels closest to the kernel's core are given more weight in a Gaussian blur than those farther away. This averaging is performed channel by channel, and the average channel values become the filtered pixel's new value.

The Gaussian filter also reduces the contrast of the image due to which we use adaptive histogram Equalization.

3.6.3 Adaptive Histogram Equalization:

Histogram Equalization is a computer-based technique for boosting visual contrast (this strategy often boosts the global contrast of images when the data is expressed by close contrast values.). This permits for an increase in contrast for regions with poor local contrast. In images with bright or dark backgrounds and foregrounds, this technique works effectively.

Another image processing technique for increasing image contrast is adaptive histogram equalisation (AHE). It varies from standard histogram equalisation in how it calculates many histograms, each referring to a distinct

portion of the image, and uses them to redistribute the image's brightness values. It is used to enhance the edges in each section of an image and to improve the overall local contrast.

For contrast-limited adaptive histogram equalization(CLAHE), instead of using histeq, we can use the adapthisteq function. Unlike histeq, which works on the entire image, adapthisteq works on small regions of the image known as tiles. Adapthisteq improves the contrast of each tile such that the output region's histogram closely resembles a specified histogram.

Here we have an illustration of an image before and after applying Adaptive Histogram Equalization:

The original image:

Fig 3.6.3.1 Original Image

Image after applying AHE:

Fig 3.6.3.2 Image after applying AHE

3.7 Segmentation:

The method of separating a digital image into multiple segments is known as image segmentation(sets of pixels, also known as image objects). By simplifying or changing an image's representation, segmentation aims to make it more meaningful and easier to analyse. Segmentation is a technique used for identifying objects and boundaries in images (lines, curves, and so on). It labels each pixel in an image so that pixels with the same label share certain similar characteristics.

Before we finalized the segmentation techniques we were going to use for our model, we compared all the common segmentation techniques used for image processing and the comparison table below shows the advantages and disadvantages of each technique:

Segmentation Technique	Description	Advantages	Limitations / Disadvantages
Thresholding Method	This method uses the	It is the easiest approach	This method is highly

	image's histogram peaks to find specific threshold values.	and does not require any prior knowledge. Performs the fastest.	dependent on peaks and it doesn't consider spatial information.
Edge based Segmentation	Uses an image's discontinuous local features to detect edges and then identify an object's boundary.	It is good for images with higher contrast between their objects.	This technique does not produce the desired results when there are too many edges in the image and there isn't enough contrast between objects.
Region based Segmentation	Based on a threshold value, this method separates the image into homogeneous clusters.	It is more immune to noise and gives best results when it is easier to define similarity criteria. Has faster operation speed and requires simple calculations	It performs poorly when objects overlap or if there is very little difference in the grayscale values. Performs well only when the background and object have good contrast.
Clustering Method	The image is divided into homogeneous clusters using this method.	It performs brilliantly on small datasets and produces excellent clusters.	Low performance in terms of cost and computation time. The distance-based k-means method can't be used to cluster clusters that aren't convex.
Watershed Method	Topological interpretation is the basis of this approach.	The results are more stable, detected object boundaries are continuous and also performs really well even when objects are touching each other/overlapping and when there is little grayscale difference	The calculations of gradients are very complex and can be computationally expensive.

Table 3.7 shows the comparison of various segmentation techniques

After analyzing the advantages, disadvantages and performing extensive literature survey, we found that thresholding along with watershed segmentation are the best possible segmentation techniques that we can apply in our model.

So in our model, we have used Thresholding and Watershed Segmentation techniques to segment the CT scan images.

3.7.1 Thresholding:

Thresholding is a segmentation algorithm that is used to differentiate/separate out the objects from the background. It is a very popular and simple segmentation algorithm. It divides the pixels of the image based on their intensity values. There are 3 types of thresholding algorithms:

i) Global Thresholding

ii) Variable Thresholding

iii) Multi-level Thresholding

In our model, we have used global thresholding. This is done to differentiate the background and the foreground and is mainly implemented to convert the grayscale image into a binary image. A constant threshold value 'T' is considered throughout the process to classify/group the pixels of the image. The output image Q(x,y) can be obtained from the original image p(x,y) by using T as shown below:

$$Q(x,y) = \{1, if\ p(x,y) > T\ 0, if\ p(x,y) < T \quad \text{...Eq 3.7.1 Thresholding}$$

In MATLAB, we first use the "graythresh" function which computes the global threshold value of the grayscale image using Otsu's algorithm. The threshold in Otsu's approach is chosen to minimise the intraclass variation of the pixels. Then, using thresholding, the "imbinarize" function converts the grayscale image to the binary image. The threshold value is given as an input to the "imbinarize" function, which uses this threshold value to transform the grayscale image to binary.

After thresholding, the resultant complement of the binary image is subjected to negative distance transformation. Complementary of the binary image is chosen so that objects of interest appear in black and for which watershed segmentation can be applied. Negative distance transform is applied since we are taking the distance transform of a complemented image.

In MATLAB, the function called "bwdist" performs distance transformation. It returns a matrix after computing the true Euclidean distance transform. The gray-level intensities of points are updated to indicate the distance from each point to the closest boundary. Each pixel in the output image has a value equal to the distance between it and the nearest non-zero pixel of the binary image. This is done so that there can be local minima and catchment basins

for different objects in the image to perform watershed segmentation, instead of 0/1 representing the objects and backgrounds.

After applying Distance Transformation, those pixels which are white in the binary image are pushed to negative infinity in the distance transformed image so that the pixels in the objects are forced to be the only local minima in the image. After this, watershed transformation functions of MATLAB are applied, and we get the resultant segmented image.

3.7.2 Watershed Segmentation:

Watershed segmentation is an image segmentation algorithm that is mainly used to detect and separate out overlapping objects or objects having common boundaries with each other in the image. Watershed segmentation is based on local topography where each pixel is considered as an elevation where white pixels are considered to have high elevations while black pixels are considered to have low elevations. The highest point is referred to as maxima while the lowest point is called the minima. Every object has a minima and is called a “local minima " of that object. The region between the local minima and the maxima is called the catchment basin. This algorithm is analogous to flooding catchment basins with water and marking/drawing a line around parts where the water starts to overflow. If there are two objects in the image which are sharing a common boundary, there will be two minima’s for the two objects. This algorithm takes different colored “water” and floods the catchment basins with different colored water. When the waters from the two basins touch each other or if they overflow (touch the maxima), a 1-pixel thick dam is plotted. These lines/contours plotted are called watershed lines. The different colors represent the different objects/regions in the image. Superimposing these watershed lines onto the image results in the final segmented image

In MATLAB, the function ‘watershed’ implements this algorithm which computes and returns a label matrix. This matrix has values that are greater than or equal to zero where zero represents the “watershed pixels” (which do not belong to any watershed region also called as watershed lines), 1 represents the pixels that belong to the first watershed region, 2 represents the pixels that belong to the second watershed region and so on. A color map also can be plotted where each color will represent a region, which can be used to determine the regions detected using watershed segmentation. The colormap can be obtained by using the function “label2rgb” which will take labels (representing the regions) from L and apply the colormap passed as an argument to the “label2rgb” function. To superimpose the watershed lines on the filtered image, we must make those parts of the pixel black in the original image wherever L=0 in the label matrix (L=0 represents the watershed lines). Making those pixels 0 (black) in the

original image will result in the superimposed image. Since each local minima, no matter how small, becomes a catchment basin, watershed transformation is known for its tendency to over-segment an image. To address this, we must use ‘imextendedmin' to filter out tiny local minima and then adjust the distance transform such that no minima occur at the filtered-out locations; this is known as "minima imposition" and is accomplished using the ‘imimposemin' function. Therefore we change the distance transform so that it only contains minima in the chosen areas, and then we repeat the watershed stages and use the ‘imfuse’ function to get the final result which is the composite of the resultant watershed transformation and the output of adaptive histogram equalization.

In the algorithm, we applied thresholding and distance transform on the image before applying watershed segmentation to overcome over-segmentation of the image which is a disadvantage of the watershed segmentation technique.

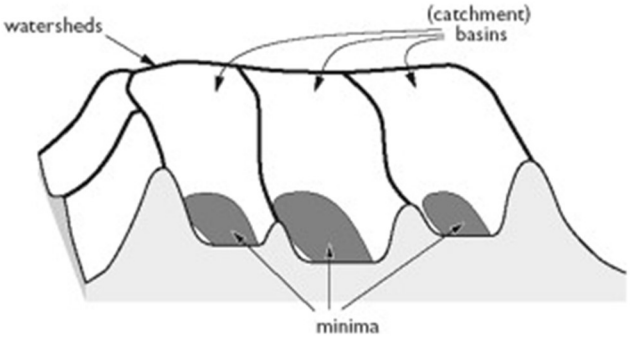


Fig 3.7.2.1

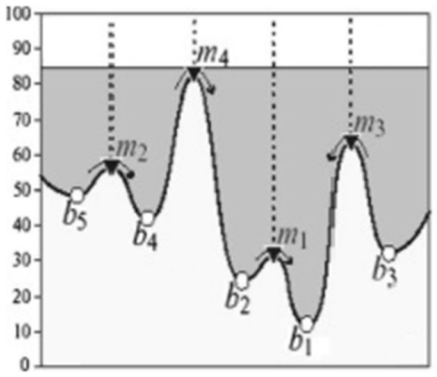


Fig 3.7.3.2

Let M_1,M_2,M_3, ….M_R be sets denoting the coordinates of the points in the regional minima of the image g(x,y). Let C(Mi) denote the coordinates of the catchment basin points associated with regional minima Mi, and let T[n] denote the set of coordinates (s,t) for which g(s,t)< n, i.e.

$$T[n] = \{ (s,t) \mid g(s,t) < n\} \quad \text{...Eq 3.7.2.1}$$

Geometrically, T[n] is the set of points in g(x,y) lying below the plane g(x,y) = n, where n represents the stage of flooding.

From n = min+1 through n = max+1, the topography will be inundated in integer flood increments. The algorithm needs to know the number of points below the flood depth at any stage n of the flooding process.

Let $C_n(M_i)$ denote the set of coordinates of points in the catchment basin associated with minimum M_i that are flooded at stage n. $C_n(M_i)$ maybe be viewed as a binary image given by:

$$C_n(M_i) = C(M_i) \cap T[n] \quad ...Eq\ 3.7.7.2$$

i.e $C_n(M_i)=1$ at location (x,y) if (x,y) ε $C(M_i)$ and (x,y) ε T[n]; otherwise $C_n(M_i) = 0$

Next, let C[n] denote the union of flooded catchment basins potions at stage n,

$$C[n] = U^{R}_{i=1}\, C_n(M_i) \quad ...Eq\ 3.7.2.3$$

Then C[max+1] is the union of all catchment basins, $C[max+1] = U^{R}_{i=1}\, C(M_i)$. ...Eq 3.7.2.4

The elements in both C[n] and T[n] are never altered during the algorithm's implementation, and the number of elements in these two sets either grows or remains constant as n progresses.

As a result, C[n-1] must be a subset of C[n]. Since C[n] is a subset of T[n], C[n-1] is also a subset of T[n], meaning that each connected component of C[n-1] is found in exactly one connected component of T[n].

The algorithm for finding the watershed lines is initialized with C[min+1] = T[min+1]. The algorithm then recurses, assuming that C[n-1] has been constructed at step n.

The process to obtain C[n] from C[n-1] is:

In T[n], let Q denote a set of connected components. Then, for each connected component q ε Q[n], there are three possibilities:

- q ∩ C[n-1] is empty
- q ∩ C[n-1] contains one connected component of C[n-1]
- q ∩ C[n-1] contains more than one connected component of C[n-1]

Based on these conditions, a dam must be built within q to prevent overflow between the catchment basins. Dilation is used to construct the dam.

The algorithm's efficiency is increased by only employing n values that correspond to existing gray-level values in g(x,y). We may deduce these values, as well as the min and max values, from the histogram g(x,y).

Chapter 4:

We are using Google Collaboratory to execute our model. It provides GPU's/TPU's runtime environments which help in handling GPU intensive tasks such as training deep learning models.

Libraries used:

Python:

1. Pandas: to analyze data
2. Numpy: for matrix tools
3. Sklearn: for machine learning and statistical modelling
4. Matplotlib.pyplot: for basic plots
5. Seaborn: for nicer plots
6. Keras: for evaluating deep learning models
7. Pydicom: for working with dicom files
8. Cv2: for image processing and analysis

4.1: Reading the pre-processed dataset

After the pre-processing stage, the results are stored in a folder retaining the original name of the file and is uploaded onto the google drive. To read the dataset using google Collaboratory, we must mount the drive and set datapath of the dataset in our google drive. After this, we use the 'pandas' library to read the excel sheet which consists of the ground truth/labels of each image and the corresponding file name, then we make a pandas dataframe which maps the file name and labels, and we make another list which will store all the file names. To read the images in the dataset, we make use of the list with all the names of the files present in the datapath, using the 'os' library in python to join the filenames with the datapath specified with the 'pydicom' library's function 'dicom.read' to read the dicom images and then we used the 'cv2' library to resize our images from 512x512 to 256x256.

4.2: Data Augmentation

Since our dataset consisted only of 100 images, our model was overfitting. Overfitting happens when the train dataset is more than the test dataset and the model has a better training accuracy and less validation accuracy with lesser training losses than validation losses as we split the dataset to 80%, 10%, 10% where 80% (80 images) is the train set size, 10% (10 images) validation set size and 10% (10 images) test set size. Due to overfitting, the model performs well on the train set but ineffectively on the new/unknown dataset. To overcome this, we used data augmentation which increased the dataset size from 100 images to 1400 images using data augmentation combinations of techniques like Scaling, Shear, Horizontal Flip, Vertical Flip on the dataset. So now, we have 1120 images for training, 140 images for testing, and 140 images for validation.

The data augmented dataset consists of the following image sets:

1. Pre-processed
2. Sheared
3. Scaled
4. Horizontally flipped
5. Vertically flipped
6. Sheared and Horizontally flipped
7. Scaled and Horizontally flipped
8. Vertically and Horizontally flipped
9. Sheared and Vertically flipped
10. Scaled and Vertically flipped
11. Horizontally flipped, Scaled, and Sheared
12. Vertically flipped, Horizontally flipped, and Scaled
13. Vertically flipped, Scaled, and Sheared
14. Sheared and Scaled

Each of these sets consists of 100 images and hence we have 1400 images in total. Data augmentation is employed by making use of the ‘imageio’ library in python and writing user-defined functions for each of the data augmentation techniques we have used. Next, we append all the pixel arrays from each set into a single new list and convert it into a NumPy array.

After data augmentation, since the label list we created earlier consists only of labels for 100 images, we need to duplicate this 14 times since we have 1400 images now, so we make use of 'pd.concatenate' in a loop that performs this operation 14 times.
The resultant NumPy array has the shape (1400,256,256) since there are 1400 images of size 256x256 stacked on top of one another in the array.

4.3 CNN:

The architecture of a Convolutional Network is inspired by the organization of the visual cortex and resembles the connectivity pattern of neurons in the human brain. The name for this type of Artificial Neural Network comes from one of the network's most important operations : Convolution

CNN architecture comprises the following layers:

1. Input Layer
2. Convolution Layer
3. Pooling Layer
4. Fully Connected Layer
5. Output Layer

In our proposed model, we have used 8 Convolution layers, 8 Batch Normalizations, 4 Max pooling layers, and 4 dense layers.

4.3.1 Input Layer:

The input layer is the input of the whole CNN. In the neural network of image processing, it generally represents the pixel matrix of the image in which each pixel is represented by a number between 0 and 255. The input layer takes input data which are 3-dimensional images with shape as Width x Height x Depth and by default the depth parameter is set to 3 which means that it's an RGB image. For our dataset, the 'Depth' is set to 1 since all the images in the dataset are grayscale images and the 'Width' and 'Height' are 256. The input layer consists of input neurons that contain the input data (the entire dataset for the CNN) which are connected to the next layer of CNN i.e the convolution layers.

4.3.2 Convolution Layer:

Convolutional layers apply convolution to the input and transfer the output through to the network's subsequent layers. A convolutional layer's sole purpose is to detect image features such as lines, edges, colour drops, etc.

In general, convolution layers operate on 3D tensors (a mathematical entity that is similar to but more general than a vector and is defined by an array of components that are functions of the coordinates of a space.) known as feature maps, which have 3 dimensions: height, width and depth. Since an RGB image has three channels, the depth axis/channels has a dimension of three (red, green, and blue) whereas the depth axis dimension for grayscale images, such as CT scan images, is one (gray level).

The convolution layer contains convolution filters that sweep out features from an image i.e., it tries to learn from an image. The depth of the filter/kernel is equal to the depth of the image.

It computes a sum of dot products between filter value and the image pixel values to form a convolution layer. This is shown by the Eq 4.3.2.1:

$$S(i, j) = (I*K)(i, j) = \sum_m I(m, n) \sum_n K(i - m, j - n) \quad \text{…Eq 4.3.2.1}$$

where S = feature map / output of the convolution operation

I = Input pixel array

K = Kernel / filter that is applied

The greater the convolution's dot product, the more similar the filter and the section of the image are.

Feature Map / Activation Map:

The results of the convolution layer are stored in what is known as a feature map/activation map. The feature maps consist of features learned by the convolution layer such as edges, lines, colors, etc. present in the image.

It is called a map since every position on the activation map reflects how well the shape filter fits on the corresponding location of the actual image, i.e. The feature map shows where a certain shape appears in the image and where it does not. The feature map's dimension is determined by the input image scale (w), feature detector size (f), stride (s), and image zero padding (p):

$$(w-f+2p)/s+1 \quad \text{...Eq 4.3.2.2}$$

Activation Function:

The activation function is a node at the end or in-between the layers of Neural Networks. They help to determine whether or not a certain neuron will fire. A nonlinear transformation is performed on the input

signal via the activation function. The transformed output is provided as input to the next layer of neurons after the activation function is applied.

We used the ReLu activation function in our model.

ReLU Activation Function

The word ReLU refers to a rectified linear unit, which is a type of activation function. Mathematically, it is defined as:

$$y = \max(0, x) \quad \ldots \text{Eq.4.3.2.3}$$

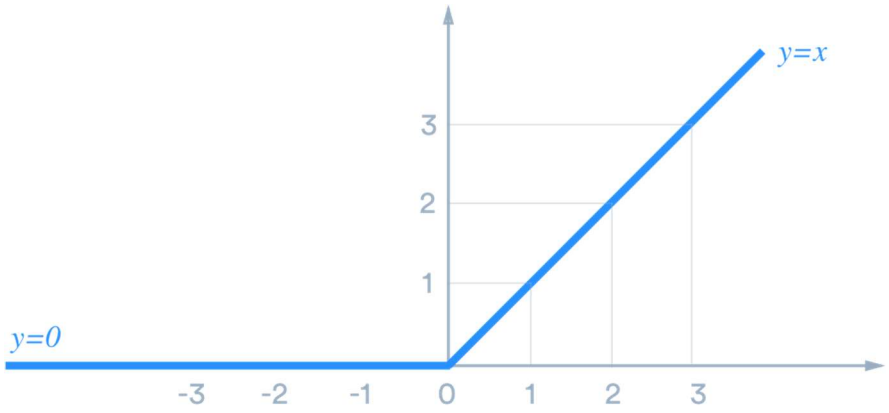


Fig 4.3.2.1 ReLU Function

ReLu is linear for all positive values and zero for all negative values, meaning that it has many of the desirable properties of a linear activation function when backpropagating a neural network. Since ReLu is linear in half of the input domain and nonlinear in the other, it is often referred to as a piecewise linear function/hinge function.

ReLU, unlike tanh and sigmoid activation functions, can output multiple true zero values for values less than zero (negative), allowing one or more true zero values to be used to activate hidden layers in neural networks, which can speed up learning and simplify the model Property like this is desired in our neural network where some weights are zero and also the neurons are more likely to be processing relevant aspects of the problem.

It also overcomes the 'vanishing gradient' problem which is present in sigmoid and tanh activation functions that cause larger values to snap to 1.0 and smaller values to snap to -1 or 0 and they are only particularly sensitive to changes in their mid-point of input: 0.5 - sigmoid and 0.0 - tanh. This means that sigmoid and tanh are activation saturated.

Networks with ReLU show better convergence and performance than those with sigmoid.

Fig 4.3.2.2 shows the output after the ReLU activation function is applied on a feature map obtained after convolution:

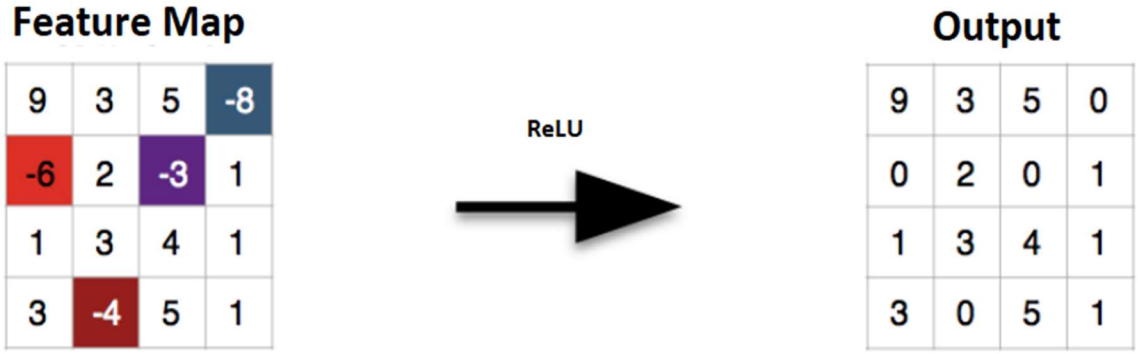


Fig 4.3.2.2 ReLu Function applied on a feature map

The convolution operation with an example is shown below:

Intuitively, we can imagine a 3×3 size window that slides along the entire 7×7 neuron layer of input containing the image. In the hidden layer, there is a neuron for each position of the window that processes this information, which is shown in figure 4.3.2.3 where I is the input pixel array, K is the convolution filter/kernel and I*K is the resultant feature map/activation map.

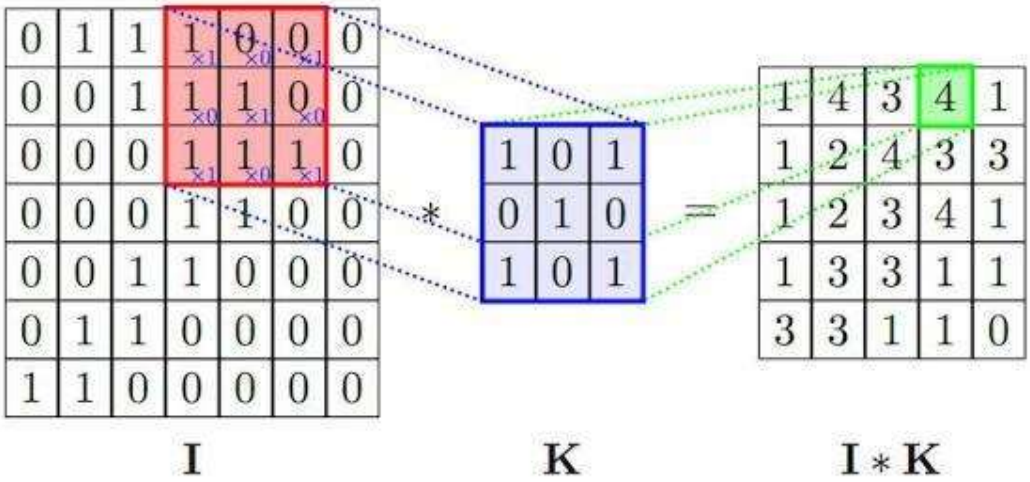


Fig 4.3.2.3 Convolution Operation

The output from the convolution layer represents high-level features it has learned from the data and the output shape is given by the formula in Eq 4.3.2.4:

$$\text{Output shape} = \frac{input - kernelsize + 2 * padding}{stride} + 1 \quad \text{...Eq 4.3.2.4}$$

Where input = input size

Kernelsize = size of the filter

Padding and Stride are values specified in the layer

and '+1' is the bias

The number of parameters in a convolution layer is given by:

((filter_size x filter_size x no_of_channels)+1) x no_of_filters ...Eq 4.3.2.5

After the Convolution layer, the feature maps are passed as inputs to the pooling layer.

4.3.3 Pooling Layer:

The pooling layer is commonly used to shrink the activation map's spatial size. Not only does it reduce the amount of computation required, but it also protects the model from overfitting. It summarizes the characteristics found in a region of the feature map produced by a convolution layer and simplifies the information collected by the previous layer.

We use Max-pooling, which is the most common pooling technique, which selects the brightest pixels from an image. It is useful when the background pixels are dark, and the pixels of interest are lighter. (which is the case in CT images).

Max Pooling:

Max pooling is a pooling operation that chooses the elements with the highest value from the region of the feature map covered by the filter. As a result, the feature map generated by the max-pooling layer contains the most prominent features of the feature map, which reflects the knowledge learned by the convolution layer in a condensed form.

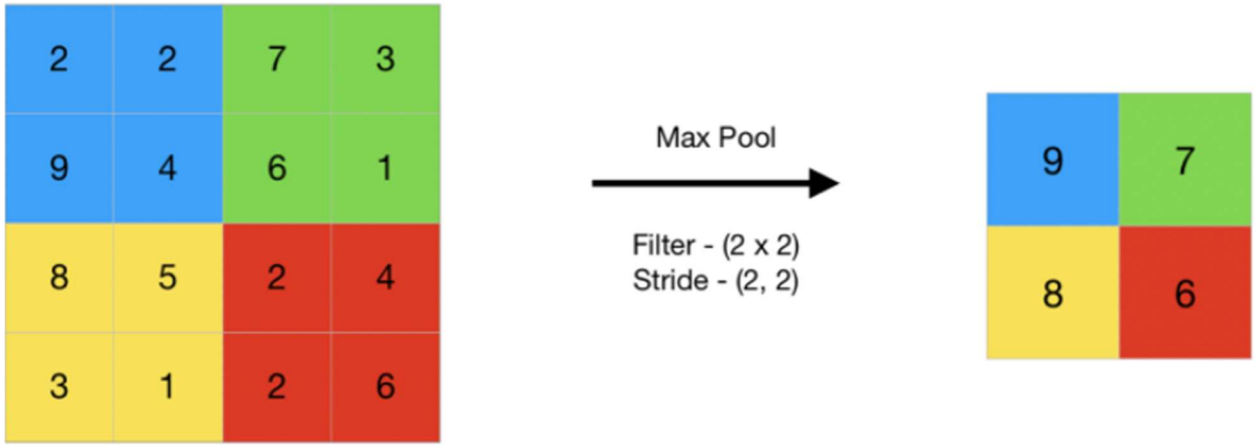


Fig 4.3.3 Max Pooling Operation

Max pooling takes 2 hyperparameters: stride and size.

Where size is the dimension of the filter/kernel of the pooling operation and Stride is a parameter of the pooling layers filter that modifies the amount of movement of the filter over the image.

The output shape of a pooling layer is given by the same equation (Eq 4.3.2.4) which is used to find the output shape of the convolution layer.

The pooling layer has 0 parameters since it doesn't have any parameters that affect the backpropagation of the model.

Apart from Max pooling, Average pooling is another commonly used pooling technique that computes the average of the elements present in the region of the feature map covered by the filter.

We used max-pooling instead of average pooling because average pooling retains a lot of data which may or may not be useful whereas max-pooling retains only the important features of the image.

4.3.4 Fully Connected Layers:

The output of the final convolution and pooling layers has to be sent to the fully connected layers but before this, we need to 'flatten' the inputs to the fully connected layer. By 'flattening' we convert the 3D outputs of the previous layers into a feature 1D vector i.e it is analogous to taking the pixel data, converting it into a single line, and then sending it as input to the final layer which will help in the classification. Fig 4.3.4.1 shows the 'flattening' operation performed on a matrix and the resultant output is a 1D feature map. These 1D array values are matrix values that contain details about the location of certain complex shapes. These complex patterns are also called trainable classifiers.

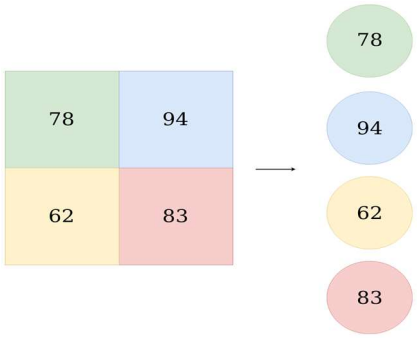


Fig 4.3.4.1 Flatten operation

Fully connected layers are the ones that perform discriminative learning in deep neural networks. It is a multi-layer perceptron that learns weights, identifies, and classifies an object class. Fig 4.3.4.2 shows a fully connected layer. Fully Connected Layer is simply feed-forward neural networks.

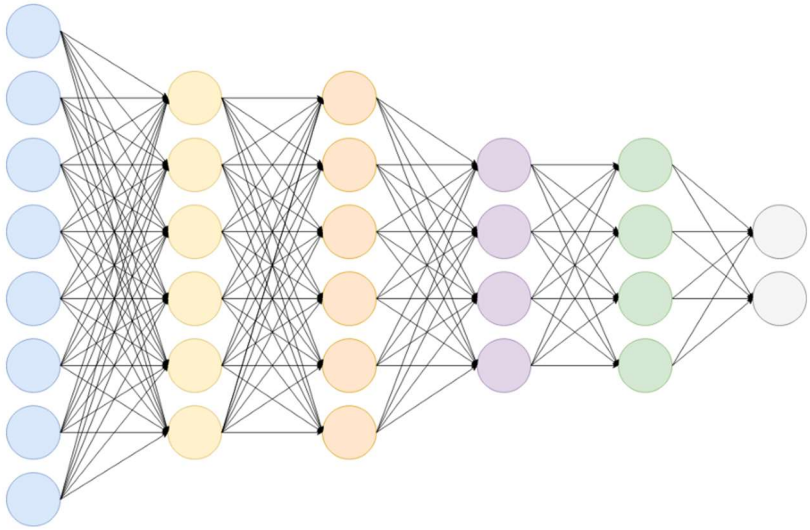


Fig 4.3.4.2 Fully Connected Layer

The fully connected layer produces a vector of probabilities for each class/category we trained the model on. For example, we have two classes: benign and malignant and the output is [0.98,0.02] then this means that there is a 98% probability that the image is benign and 2% that it's malignant.

The output weights are basically derived from:

I. What and all trainable classifiers are present in the image
II. and how the trainable classifiers are relative to each other

The output shape of a Fully connected layer is the same as the units specified in the layer.
The number of parameters of a Fully connected layer is given by the formula:

$$(\text{Input} \times N) + N \quad \text{...Eq 4.3.4}$$

Where N = Number of units in the layer

Input = Output size of the previous layer

4.3.5 Output Layer:

A CNN's output layer is the final fully connected layer, with the number of units/perceptrons/neurons equal to the number of classes defined or trained on. The final layer employs the softmax activation function, which aids in the solution of multi-class classification problems.

The Softmax Activation Function:

The softmax function converts a vector of K real values into a vector of K real values that add up to one. The softmax converts the input values into values between 0 and 1, allowing them to be interpreted as probabilities. The input values can be positive, negative, zero, or greater than one. If one of the inputs is small or negative, the softmax converts it to a small probability; if one of the inputs is large, it converts it to a large probability, but it will always remain between 0 and 1.

The softmax function is given as:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$ ...Eq 4.3.5.1

Where z(bar) = input vector to the softmax function

z_i = elements of the input vector to the softmax function

k = number of classes

As a result, since the output layer is the final fully connected layer, the output will be a vector of probabilities that add up to 1 (due to the softmax activation function), and each output will have the same number of probabilities as the number of groups (in our model, each output will have 2 probabilities since there are 2 classes).

As the number of classes is fixed to 2, the output shape of the output layer is (NONE,2) irrespective of the model/architecture we implement.

Loss function:

Binary cross-entropy loss function:

Binary cross-entropy is a loss function that is used for binary classification tasks (tasks that answer a question with only two choices). In our model, as our model has only 2 classes: Benign (not cancerous) and Malignant (cancerous) we used the binary cross-entropy function which helps in analyzing how "good" or how "bad" the prediction of our model is and this information helps in optimizing our model.

Binary cross-entropy compares each of the predicted probabilities generated by the model to the actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. That means how close or far the value is from the actual value.

Binary cross-entropy can be defined as the negative average of the corrected probabilities that were predicted and is given by the following equation:

$$\text{Log Loss} = \frac{1}{N}\sum_{i=1}^{N} -(y_i * log(p_i) + (1 - y_i) * log(1 - p_i)) \quad \text{...Eq 4.3.5.2 Binary Cross-entropy loss}$$

Where N = Total number of probabilities

p = predicted probability

y = label value / ground truth

Optimizer:

Adam:

Adam is an optimization algorithm that can be used to update network weights iteratively based on training data instead of the traditional stochastic gradient descent procedure.

Stochastic gradient descent is an optimization approach that estimates the error gradient for the current state of the model using instances from the training dataset, then utilizes the back-propagation of errors procedure to update the model's weights.

The amount by which the weights are updated during training is known as the "learning rate," also known as the "step size."

For all weight updates, stochastic gradient descent uses the same learning rate (called alpha), which remains constant throughout the training.

Each network weight (parameter) has its own learning rate, which is adjusted separately as learning progresses. The symbol η is often used to denote the learning rate.

Backpropagation of error calculates the amount of error induced by a node's weights in the network during training. The learning rate is used to scale the weight instead of updating it with the full amount.

Adam is better defined by combining the benefits of two other extensions of stochastic gradient descent, specifically: AdaGrad and RMSProp.

Since it is typically calculated on some small random batch of data, the gradient of a neural network's cost function can be thought of as a random variable whose first moment is mean, and the second moment is uncentered variance (this means that we do not deduct the mean while calculating variance).

Adam recognizes the advantages of AdaGrad and RMSProp. Instead of adjusting parameter learning rates based on the average first moment (the mean) like RMSProp, Adam also takes into consideration the average of the second moments (the uncentered variance).

This algorithm computes the gradient and squared gradient exponential moving averages, with the decay rates of these moving averages governed by the parameters$\beta_1$andβ_2.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \text{...Eq 4.3.5.3}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad \text{...Eq 4.3.5.4}$$

Equations 4.3.5.3 and 4.3.5.4 Represent the moving averages of gradient and squared gradient, where m and v are moving averages, g is gradient on current mini-batch, and betas — new introduced hyper-parameters of the algorithm.

Apart from the layers mentioned before, we have used 2 more layers that helps in improving the performance of the model during training and to reduce the loss:

1. BatchNormalization
2. Dropout

4.3.6 BatchNormalization:

When we update the previous weights during training, the distribution of the outputs of each intermediate activation layer shifts at each iteration, which is referred to as an internal covariant shift (ICS), and when all layers are shifted during an update, the update process is forever chasing a moving target. Batch normalization is used to avoid this.

Normalizing the input features increases our algorithm's convergence rate, which speeds up the learning process. But, in a neural network, we also have activations in the hidden layers that must be normalized. Normalizing the input features as well as inputs in the hidden layers helps in accelerating the training.

We use the mean and standard deviation (or variance) of the values in the current batch to normalize each layer's inputs during training, which is why it's called "batch" normalization.

By subtracting the empirical mean over the batch and dividing it by the empirical standard deviation, batch normalization normalizes the output of the previous layer. This will help the data to look like a Gaussian distribution.

$$\widehat{x_i} \leftarrow \frac{x_i - \mu B}{\left(\sqrt{\sigma_B^2 + \epsilon}\right)} \ldots \text{Eq 4.3.6.1.1 Normalize}$$

where $\mu B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$ ...Eq 4.3.6.1.2 mini-batch mean

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu B)^2 \ldots \text{Eq 4.3.6.1.3 mini-batch variance}$$

$$y_i \leftarrow \gamma \widehat{x_i} + \beta \ldots \text{Eq 4.3.6.2 Scale and shift}$$

Where $\gamma$and β are learnable parameters

To put it simply, batch normalization performs the operation throughout the network rather than just once at the start.On the other hand, normalization reduces the values to [0, 1], which is not always desirable, so it applies γ and β to the normalization values. These parameters are trained in the same way as the other hyperparameters during

the training phase via backpropagation. As a result, two parameters $\gamma$and β govern the normalisation mean and standard deviation.

For example, even if the exact values of hidden layer inputs vary, their mean and standard deviation nearly always remain constant, minimising the internal covariate shift. This weakens the coupling between the parameters of the previous layers and those of deeper layers, allowing each layer of the network to learn independently of the others.

4.3.7 Dropout:

Dropout is a technique that is used to prevent a model from overfitting. A neural network's dropout is implemented per layer. It works with a wide range of layers, including dense fully connected layers and convolutional layers. Dropout can be applied to any or all of the network's hidden layers, as well as the visible or input layer, but not the output layer. The dropout layer has a hyperparameter that determines the probability of the layer's outputs being dropped out, or the inversely, the probability of the layer's outputs being retained. The parameter accepts values between [0,1]. The network's weights would be larger than usual due to the dropout. As a result, the weights are initially scaled by the desired dropout rate before the network is finalized.

4.4 Training:

After reading the dataset and storing all the images in a NumPy array and storing all the labels in a panda data frame, we have to split the data set into:

1. Train set
2. Test set
3. Validation set

We split the dataset by using the ‘train_test_split’ function from the sklearn’s model selection library which takes the NumPy array containing the images, labels, and the test set size (value should be in the range from 0 to 1) as arguments.

We need to use the ‘train_test_split’ function twice since we have to split the dataset twice i.e.split once for validation set and again split the remaining train set again into test set and train set.

The sizes of these datasets are:

1. Train set - 80% (1,120 images)
2. Test set - 10% (140 images)
3. Validation set - 10% (140 images)

Before we finalized the train, test and validation dataset sizes, we tried splitting the dataset into various sizes such as (i) 70%,15%,15% (ii) 60%,20%,20% (iii) 80%,10%,10% and we found that the (iii) option gave us better performance and accuracy. With (ii) option, the train losses and validation losses were very high and with the (i) option, the train losses and validation losses were equal but higher in comparison with the (iii) option.

After splitting the dataset, we need to reshape each set in order to proceed with the training because, by default, after splitting the train set, the test set and validation set has a shape of (no_images,width,height) but the CNN input layer requires the shape(no_images,width,height,depth) so we have to reshape each of these sets by adding 'depth' as 1 since we are working on grayscale images. We need to normalize each dataset to one particular type too because if the sets have to be dealt with, they must have the same representation, so we normalize each set as 'float32' type and divide it by the maximum value present in the set. This will transform the values in the set to be in the same range.

After reshaping and normalizing the datasets, we create 2 variables called 'batch_size' and 'epochs'.

Epochs: When the entire dataset is passed forward and backward through a neural network, it is called one epoch

Batch size: If the datasets are large, it is difficult for one epoch to be fed to the computer at once and hence we divide it into smaller batches. The size of these batches is called batch size.

We set the batch_size to 20 and we are training the model for 100 epochs. After this, we create a model using the keras's 'sequential' library' to import the layers such as convolution layers, max pool layers, etc that we will use in our model and then compile it using the 'model.compile' function. We use the 'binary cross-entropy to compute the loss and 'Adam optimizer' as the optimizer for our model.

Then we train the model using the 'model.fit' function and pass arguments such as the train set, batch size, number of epochs we need to trail the model for, the validation set which the model will predict after each epoch and compute the validation accuracy and validation loss, and callbacks if any.

This training procedure remains the same no matter what model/architecture we use.

4.5 CNN Architecture:

Before we were able to arrive at our proposed model/architecture, we tested many other models to check if we can improve the accuracy, accelerate the training of the model and reduce the validation loss. The models we tested were:

1. LeNet Architecture
2. AlexNet Architecture
3. VGG16 Architecture

4.5.1 LeNet Architecture:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 254, 254, 6)       60
_________________________________________________________________
average_pooling2d (AveragePo (None, 127, 127, 6)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 125, 125, 16)      880
_________________________________________________________________
average_pooling2d_1 (Average (None, 62, 62, 16)        0
_________________________________________________________________
flatten (Flatten)            (None, 61504)             0
_________________________________________________________________
dense (Dense)                (None, 120)               7380600
_________________________________________________________________
dense_1 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 170
=================================================================
Total params: 7,391,874
Trainable params: 7,391,874
Non-trainable params: 0
_________________________________________________________________
```

Fig 4.5.1 LeNet model summary

Layer	# of filters	Filter size	Stride	Padding	Size of feature map	Activation function
Input	-	-	-	-	256 x 256 x 1	-
Conv 1	6	3 x 3	1	-	254 x 254 x 6	ReLU
Average Pool 1	-	3 x 3	1	-	127 x 127 x 6	-

Conv 2	16	3 x 3	1	-	125 x 125 x 16	ReLU
Average Pool 2	-	3 x 3	1	-	62 x 62 x 16	-
Fully Connected 1	-	-	-	-	120	ReLU
Fully Connected 2	-	-	-	-	84	ReLU
Fully Connected 3	-	-	-	-	2	Softmax

Table 4.5.1 LeNet Architecture

5 layers, 2 convolution layers, and 3 fully connected layers make up the LeNet architecture.

4.5.2 AlexNet Architecture:

AlexNet Model Summary:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 62, 62, 96)        11712
_________________________________________________________________
batch_normalization_5 (Batch (None, 62, 62, 96)        384
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 30, 30, 96)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 30, 30, 256)       614656
_________________________________________________________________
batch_normalization_6 (Batch (None, 30, 30, 256)       1024
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 14, 14, 256)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 14, 14, 384)       885120
_________________________________________________________________
batch_normalization_7 (Batch (None, 14, 14, 384)       1536
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 14, 14, 384)       147840
_________________________________________________________________
batch_normalization_8 (Batch (None, 14, 14, 384)       1536
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 14, 14, 256)       98560
_________________________________________________________________
batch_normalization_9 (Batch (None, 14, 14, 256)       1024
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_3 (Dense)              (None, 4096)              37752832
_________________________________________________________________
dropout_2 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_4 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_3 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_5 (Dense)              (None, 2)                 8194
=================================================================
Total params: 56,305,730
Trainable params: 56,302,978
Non-trainable params: 2,752
_________________________________________________________________
```

Fig 4.5.2 AlexNet Model summary

Layer	# of filters	Filter size	Stride	Padding	Size of feature map	Activation function
Input	-	-	-	-	256 x 256 x 1	-
Conv 1	96	11 x 11	4	-	62 x 62 x 96	ReLU
Max Pool 1	-	3 x 3	2	-	30 x 30 x 96	-
Conv 2	256	5 x 5	1	-	30 x 30 x 256	ReLU
Max Pool 2	-	3 x 3	2	-	14 x 14 x 256	-
Conv 3	384	3 x 3	1	-	14 x 14 x 384	ReLU
Conv 4	384	3 x 3	1	-	14 x 14 x 384	ReLU
Conv 5	256	3 x 3	1	-	14 x 14 x 256	ReLU
Max Pool 3	-	3 x 3	2	-	6 x 6 x 256	-
Fully Connected 1	-	-	-	-	4096	ReLU
Dropout 1	rate= 0.5	-	-	-	4096	
Fully Connected 2	-	-	-	-	4096	ReLU
Dropout 2	rate= 0.5	-	-	-	4096	
Fully Connected 3	-	-	-	-	2	Softmax

Table 4.5.2 AlexNet Architecture

They discovered that by employing ReLu as an activation function, they could speed up the training process by almost 6 times. Dropout layers were also used to prevent overfitting.

4.5.3 VGG 16 Architecture:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 256, 256, 64)      640
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 256, 256, 64)      36928
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 128, 128, 64)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 128, 128, 128)     73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 128, 128, 128)     147584
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 64, 64, 256)       295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 64, 64, 256)       590080
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 64, 64, 256)       590080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 32, 32, 512)       1180160
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 32, 32, 512)       2359808
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 32, 32, 512)       2359808
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 16, 16, 512)       0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 512)       2359808
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 16, 16, 512)       2359808
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 16, 16, 512)       2359808
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 8, 8, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 32768)             0
_________________________________________________________________
dense (Dense)                (None, 4096)              134221824
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 8194
=================================================================
Total params: 165,724,866
Trainable params: 165,724,866
Non-trainable params: 0
_________________________________________________________________
```

Fig 4.5.3 VGG16 model summary

Layer	# of filters	Filter size	Stride	Padding	Size of feature map	Activation function
Input	-	-	-	-	256 x 256 x 1	-
Conv 1	64	3 x 3	1	-	256 x 256 x 64	ReLU
Conv 2	64	3 x 3	1	-	256 x 256 x 64	ReLU

Max Pool 1	-	2 x 2	2	-	128 x 128 x 64	-
Conv 3	128	3 x 3	1	-	128 x 128 x 128	ReLU
Conv 4	128	3 x 3	1	-	128 x 128 x 128	ReLU
Max Pool 2	-	2 x 2	2	-	64 x 64 x 128	-
Conv 5	256	3 x 3	1	-	64 x 64 x 256	ReLU
Conv 6	256	3 x 3	1	-	64 x 64 x 256	ReLU
Conv 7	256	3 x 3	1	-	64 x 64 x 256	ReLU
Max Pool 3	-	2 x 2	2	-	32 x 32 x 256	-
Conv 8	512	3 x 3	1	-	32 x 32 x 512	ReLU
Conv 9	512	3 x 3	1	-	32 x 32 x 512	ReLU
Conv 10	512	3 x 3	1	-	32 x 32 x 512	ReLU
Max Pool 4	-	2 x 2	2	-	16 x16 x 512	-
Conv 11	512	3 x 3	1	-	16 x16 x 512	ReLU
Conv 12	512	3 x 3	1	-	16 x16 x 512	ReLU
Conv 13	512	3 x 3	1	-	16 x16 x 512	ReLU
Max Pool 5	-	2 x 2	2	-	8 x 8 x 512	-
Fully Connected 1	-	-	-	-	4096	ReLU
Fully Connected 2	-	-	-	-	4096	ReLU
Fully Connected 3	-	-	-	-	2	Softmax

Table 4.5.3 VGG16 Architecture

VGG16 is made up of 16 layers, 13 convolution layers, and three fully connected layers.

4.5.4 Proposed Model:

Layer	# of filters	Filter size	Stride	Padding	Size of feature map	Activation function
Input	-	-	-	-	256 x 256 x 1	-
Conv 1	96	11 x 11	4	-	62 x 62 x 96	ReLU
Max Pooling 1	-	3 x 3	2	-	30 x 30 x 96	-
Conv 2	256	5 x 5	1	-	30 x 30 x 256	ReLU
Max Pooling 2	-	3 x 3	2	-	14 x 14 x 256	-
Dropout 1	Rate = 0.7	-	-	-	14 x 14 x 256	-
Conv 3	384	3 x 3	1	-	14 x 14 x 384	ReLU
Conv 4	384	1 x 1	1	-	14 x 14 x 384	ReLU
Conv 5	256	1 x 1	1	-	14 x 14 x 256	ReLU
Max Pooling 3	-	3 x 3	2	-	6 x 6 x 256	-
Dropout 2	Rate = 0.7	-	-	-	6 x 6 x 256	-
Conv 6	512	5 x 5	1	-	6 x 6 x 512	ReLU
Conv 7	512	5 x 5	1	-	6 x 6 x 512	ReLU
Conv 8	384	5 x 5	1	-	6 x 6 x 384	ReLU
Max Pooling 4	-	3 x 3	2	-	2 x 2 x 384	-
Dropout 3	Rate =0.7	-	-	-	2 x 2 x 384	-
Fully Connected 1	-	-	-	-	4096	ReLU
Dropout 4	Rate = 0.7	-	-	-	4096	-
Fully Connected 2	-	-	-	-	4096	ReLU
Dropout 5	Rate = 0.7	-	-	-	4096	-
Fully Connected 3	-	-	-	-	4096	ReLU

Dropout 6	Rate = 0.7	-	-	-	4096	-
Fully Connected 4	-	-	-	-	2	Softmax

Table 4.5.4 Proposed Architecture

We used 2 convolution layers having 384 and 256 filters with filter size of 1x1 to retain the feature map sizes as they would be decreasing when sent to the subsequent layers which would lead to lesser features being learnt by the subsequent layers.

We used dropout layers between convolution layers and also in between the fully connected layers in order to avoid the model from overfitting during training with 0.7 dropout rate.

We used 8 convolution layers with decreasing filter sizes like 11x11, 5x5, 3x3, 1x1 and an increasing number of filters in each layer like 96, 256, 384, 512 to achieve higher accuracy and to improve the learning process in training.

We used 3 convolution layers in series - twice, to extract higher level features in each iteration.

We used Batch Normalization layers after each convolution layer in order to normalize the inputs to the next layers to overcome Internal Covariant Shift (ICS) and to accelerate the training.

Max Pooling was used as the pooling technique to reduce the feature map size and to represent the features learnt by the previous layers in a condensed form.

We used 3 Fully connected layers with 4096 neurons with the inputs from the previous layer linked to every neuron of the next layer. It is used to perform discriminative learning.

The last Fully connected layer with 2 output neurons are used to classify the output as benign or malignant.

The model summary of our proposed model:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 62, 62, 96)        11712
_________________________________________________________________
batch_normalization (BatchNo (None, 62, 62, 96)        384
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 30, 30, 96)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 30, 30, 256)       614656
_________________________________________________________________
batch_normalization_1 (Batch (None, 30, 30, 256)       1024
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 256)       0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 256)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 384)       885120
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 384)       1536
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 384)       1327488
_________________________________________________________________
batch_normalization_3 (Batch (None, 14, 14, 384)       1536
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 256)       98560
_________________________________________________________________
batch_normalization_4 (Batch (None, 14, 14, 256)       1024
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 256)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 6, 6, 512)         3277312
_________________________________________________________________
batch_normalization_5 (Batch (None, 6, 6, 512)         2048
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 6, 6, 512)         6554112
_________________________________________________________________
batch_normalization_6 (Batch (None, 6, 6, 512)         2048
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 6, 6, 384)         196992
_________________________________________________________________
batch_normalization_7 (Batch (None, 6, 6, 384)         1536
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 2, 2, 384)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 2, 2, 384)         0
_________________________________________________________________
flatten (Flatten)            (None, 1536)              0
_________________________________________________________________
dense (Dense)                (None, 4096)              6295552
_________________________________________________________________
dropout_3 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_4 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_5 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 8194
=================================================================
Total params: 52,843,458
Trainable params: 52,837,890
Non-trainable params: 5,568
_________________________________________________________________
```

Fig 4.5.4 Proposed Architectures Model Summary

4.6 Hyperparameter Tuning:

The process of selecting a set of optimal hyperparameters for a given learning algorithm is known as hyperparameter tuning (also known as hyperparameter optimization). A hyperparameter is a type of parameter whose value determines how the learning process flows and is regulated. To generalize different data patterns, our model can involve different constraints, weights, or learning rates. These measures are hyperparameters and they have to be tuned in order for our model to solve the problem optimally.

The Hyperparameters that we will be tuning in our model are:

Activation Function: Activation function introduces nonlinearity to the model. The most used activation functions are ReLu, Sigmoid and tanh functions. We will be using the ReLu activation function in our model. ReLu is faster to compute than Sigmoid-like functions because it only needs to select max(0, x) instead of performing costly exponential operations like Sigmoids. Practically, Relu networks perform better than sigmoid networks in terms of convergence.
ReLu is a nonlinear function that gives the same benefits as sigmoid but with better efficiency. It also avoids and solves the vanishing gradient problem.

No. of Filters: Filters in our model are feature detectors. Each filter generates a feature map which allows the model to learn explanatory factors within the image. More number of filters would extract more features but is not the case all the time. It would depend on the complexity and size of the input images. Suitable number of filters for our model was finalized by doing a lot of testing with different numbers of filters and verifying the results.

Filter/Kernel Size: There is no specific way of choosing the size of a filter. To minimize the computational cost and the incredibly long training time needed for the computation, the filter size is usually kept small in comparison to the size of the input images. The most popular kernel sizes used by CNNs are 1x1,3x3,5x5 and 11x11 (which was introduced by Alexnet Architecture). We have made use of 11x11 filters in the first layer and relatively small sized filters in the upcoming layers and fully connected layer to reduce computational cost and time in weight sharing which would lead to lesser weights being fed back to the architecture during back-propagation.

Stride and Padding: After having chosen the filter size, we must select the stride and padding. The stride of the filter determines how it convolves around the input volume. The stride is the number by which the filter moves. The stride value should be chosen such that the output volume is not a fraction but an integer. If we set the value of the stride too low compared to the input image, the filter will shift through the image too slowly but will be able

to extract a lot of features. Setting the value slightly higher will reduce the computational time at the cost of lower features being extracted. So, we have chosen a value of 4 for the first layer and relatively smaller value i.e. either 1 or 2 for the upcoming layers.
To maintain the dimensions of the output as in the input, we make use of padding. We make use of 'same' padding which would reserve the output as it is and add layers of zeros on the border to match the input dimension.

Batch Size: It is the number of training samples passed to the model in a single forward or backward pass. The higher the batch size, more memory would be used by the program running the model and lower the batch size would be a small set of data ranging from 100-1000 images. A batch size of 16 to 128 is generally used. We have initialized the batch size to 20, so it would take 56 iterations for the model to complete one epoch. We found this value to work well with our model producing results relatively quickly given the hardware resources on our end.

Number of Epochs: The number of epochs refers to how many times the entire training set is processed by the neural network. The number of epochs (100) was finalized after a series of trial and error when the model showed optimal consistency of training and validation accuracy.

Early Stopping: Too many epochs may lead to overfitting of the training dataset and too few may lead to underfitting. As a result, we employ early stopping, which allows us to determine an arbitrary number of training epochs and stop training once the model's output on a hold out validation dataset stops improving.

Chapter 5:

5.1 Results:

5.1.1 Image Pre-processing using filters:

The results of applying filters like Median Filter, Gaussian Filter, and Adaptive Histogram Equalization can be seen in Fig 5.1.1. The first image in Fig 5.1.1 shows the original Dicom image, the second image shows the result of the image after applying the Median Filter which removes the salt and pepper noises from the image. Then the output of the Median Filter is passed to the Gaussian filter which applies a Gaussian blur that removes the high-frequency components in the image but also decreases the contrast which can be seen in the third image of Fig 5.1.1. To increase the contrast of the Image, we then pass the output of Gaussian Filter to the Adaptive Histogram Equalization and the result of it can be seen in the fourth image of Fig 5.1.1.

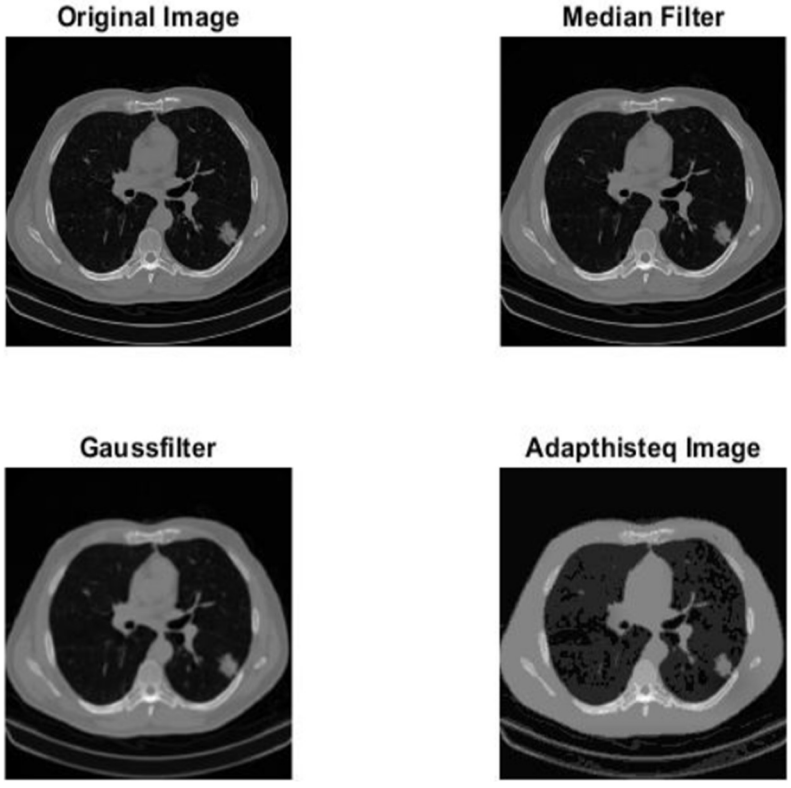


Fig 5.1.1 Image Pre-Processing

5.1.2 Segmentation:

To perform watershed segmentation, first, the grayscale image obtained after Adaptive histogram Equalization is converted to a binary image using Thresholding. The resultant binary image after applying thresholding can be seen in Fig 5.1.2.1. After obtaining the binary image, the complementary of the binary image is obtained and inverse distance transformation is applied and the result of it can be seen in 5.1.2.2 and 5.1.2.3. Then the pixels which are white in the binary image are pushed to negative infinity in the distance transformed image so that the pixels in the objects are forced to be the only local minima in the image which can be seen in Fig 5.1.2.4. Then we apply the 'imextendedminima' function that computes the extended-minima transform, which is the regional minima of the 2-minima transform to get a grayscale mask, the result is shown in Fig 5.1.2.5. Next, we use the 'imimposemin' function that modifies the grayscale mask image using morphological reconstruction, so it only has a regional minima wherever the binary image is nonzero. Then, on the image created by the 'imimposemin,' we apply watershed transformation and construct a new binary image by equating the pixel values in the binarized image to 0 wherever the watershed result has pixel values equal to 0. Fig 5.1.2.6 shows the watershed lines superimposed on the binary image; Fig 5.1.2.7 shows the result of "label2rgb" function which assigns different colors to different regions of the watershed. Finally, we apply the 'imfuse' function with arguments as the binary image generated before and the resultant image after applying adaptive histogram equalization which takes the composite of the two images. The final result of watershed segmentation is shown in Fig 5.1.2.8.

Fig 5.1.2.1

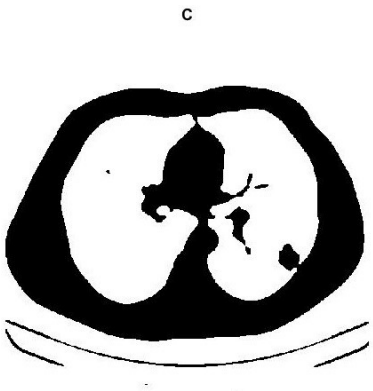


Fig 5.1.2.2

Fig 5.1.2.3

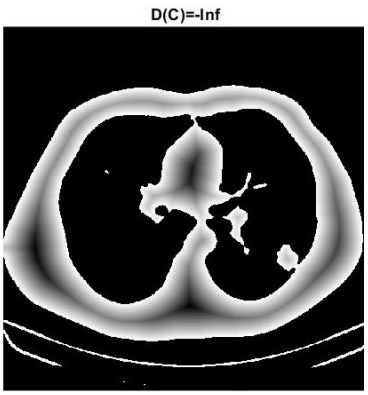


Fig 5. 1.2.4

Fig 5.1.2.5

Fig 5.1.2.6

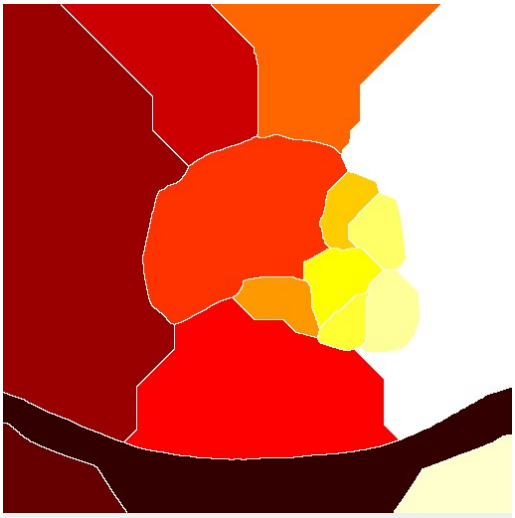


Fig 5.1.2.7

Fig 5.1.2.8

Fig 5.1.2.9

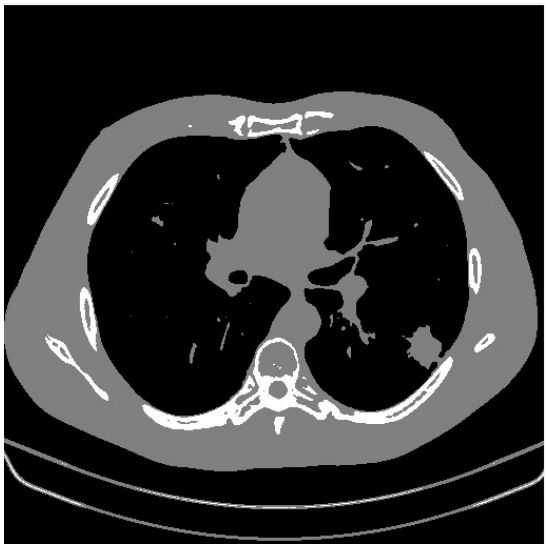


Fig 5.1.2.10

where 5.1.2.1-Binarized Image, 5.1.2.2-Complemented Image, 5.1.2.3-Inverse Distance Transform, 5.1.2.4-Negative Infinity transformation, 5.1.2.5-Grayscale mask, 5.1.2.6-Watershed Lines Image, 5.1.2.7-Watershed Image with Labels, 5.1.2.8-Final output of Watershed Segmentation, 5.1.2.9-K-means with Clusters=2, 5.1.2.10-K-means with Clusters=3.

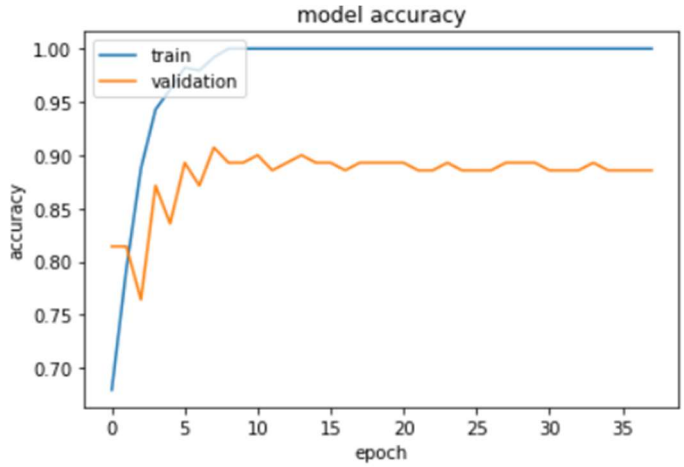
The images shown in Fig 5.1.2.9 and Fig 5.1.2.10 show the result of applying the K-means clustering algorithm instead of the watershed segmentation algorithm on the original image. The image shown in Fig 5.1.2.9 has the value of K=2 and Fig 5.1.2.10 has the value of K=3. (K represents the number of clusters).

5.1.3 CNN:

5.1.3.1 LeNet:

```
Epoch 29/100
56/56 [==============================] - 1s 13ms/step - loss: 4.9120e-05 - accuracy: 1.0000 - val_loss: 0.8610 - val_accuracy: 0.8929
Epoch 30/100
56/56 [==============================] - 1s 13ms/step - loss: 4.3142e-05 - accuracy: 1.0000 - val_loss: 0.8880 - val_accuracy: 0.8929
Epoch 31/100
56/56 [==============================] - 1s 13ms/step - loss: 4.0418e-05 - accuracy: 1.0000 - val_loss: 0.8778 - val_accuracy: 0.8857
Epoch 32/100
56/56 [==============================] - 1s 14ms/step - loss: 3.7757e-05 - accuracy: 1.0000 - val_loss: 0.8855 - val_accuracy: 0.8857
Epoch 33/100
56/56 [==============================] - 1s 13ms/step - loss: 4.0179e-05 - accuracy: 1.0000 - val_loss: 0.8885 - val_accuracy: 0.8857
Epoch 34/100
56/56 [==============================] - 1s 13ms/step - loss: 3.6079e-05 - accuracy: 1.0000 - val_loss: 0.8858 - val_accuracy: 0.8929
Epoch 35/100
56/56 [==============================] - 1s 13ms/step - loss: 3.4503e-05 - accuracy: 1.0000 - val_loss: 0.8926 - val_accuracy: 0.8857
Epoch 36/100
56/56 [==============================] - 1s 13ms/step - loss: 2.6925e-05 - accuracy: 1.0000 - val_loss: 0.9009 - val_accuracy: 0.8857
Epoch 37/100
56/56 [==============================] - 1s 14ms/step - loss: 3.0875e-05 - accuracy: 1.0000 - val_loss: 0.9052 - val_accuracy: 0.8857
Epoch 38/100
56/56 [==============================] - 1s 13ms/step - loss: 2.4172e-05 - accuracy: 1.0000 - val_loss: 0.9127 - val_accuracy: 0.8857
Epoch 00038: early stopping
```

Fig 5.1.3.1.1 Training results of LeNet Architecture




```
score:  [0.9126968383789062, 0.8857142925262451]
Wrong predictions:  17
```

Fig 5.1.3.1.2 Model accuracy & Model Loss plots and score of LeNet Architecture

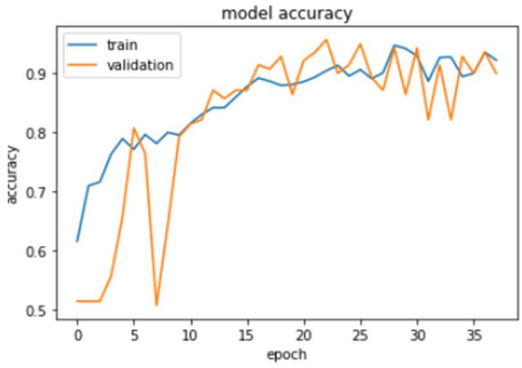
	precision	recall	f1-score	support
0	0.84	0.91	0.87	65
1	0.91	0.85	0.88	75
accuracy			0.88	140
macro avg	0.88	0.88	0.88	140
weighted avg	0.88	0.88	0.88	140

Fig 5.1.3.1.3 Classification report of LeNet Architecture

5.1.3.2 AlexNet:

```
56/56 [==============================] - 2s 44ms/step - loss: 0.1967 - accuracy: 0.9357 - val_loss: 0.5619 - val_accuracy: 0.8214
Epoch 35/100
56/56 [==============================] - 2s 44ms/step - loss: 0.3671 - accuracy: 0.8813 - val_loss: 0.2832 - val_accuracy: 0.9286
Epoch 36/100
56/56 [==============================] - 2s 44ms/step - loss: 0.3210 - accuracy: 0.8920 - val_loss: 0.3822 - val_accuracy: 0.9000
Epoch 37/100
56/56 [==============================] - 2s 44ms/step - loss: 0.2170 - accuracy: 0.9379 - val_loss: 0.1754 - val_accuracy: 0.9357
Epoch 38/100
56/56 [==============================] - 2s 44ms/step - loss: 0.2752 - accuracy: 0.9372 - val_loss: 0.9657 - val_accuracy: 0.9000
Epoch 00038: early stopping
```

Fig 5.1.3.2.1 Training results of AlexNet Architecture



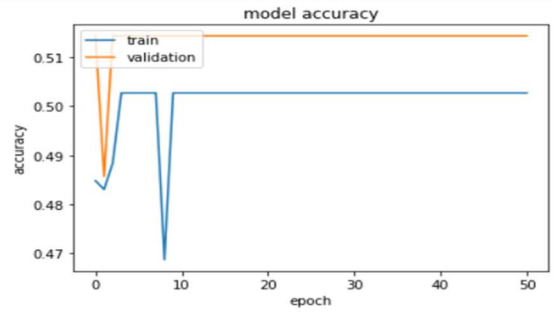

```
score:  [0.9656955003738403, 0.8999999761581421]
Wrong predictions:  14
              precision    recall  f1-score   support

           0       0.88      0.91      0.89        65
           1       0.92      0.89      0.91        75

    accuracy                           0.90       140
   macro avg       0.90      0.90      0.90       140
weighted avg       0.90      0.90      0.90       140
```

Fig 5.1.3.2.2 Model accuracy & Model Loss plots, score and classification report of AlexNet

5.2.3 VGG16:

```
Epoch 39/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6932 - accuracy: 0.5036 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 40/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6930 - accuracy: 0.5171 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 41/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6931 - accuracy: 0.5080 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 42/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6931 - accuracy: 0.5094 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 43/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6932 - accuracy: 0.4967 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 44/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6932 - accuracy: 0.4939 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 45/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6930 - accuracy: 0.5148 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 46/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6932 - accuracy: 0.5036 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 47/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6932 - accuracy: 0.4992 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 48/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6933 - accuracy: 0.4859 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 49/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6933 - accuracy: 0.4890 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 50/100
56/56 [==============================] - 12s 212ms/step - loss: 0.6934 - accuracy: 0.4802 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 51/100
56/56 [==============================] - 12s 213ms/step - loss: 0.6931 - accuracy: 0.5066 - val_loss: 0.6930 - val_accuracy: 0.5143
Epoch 00051: early stopping
```

Fig 5.1.3.2.1 Training results of VGG16 Architecture




```
score:  [0.6929967403411865, 0.5142857432365417]
Wrong predictions:  75
              precision    recall  f1-score   support

           0       0.46      1.00      0.63        65
           1       0.00      0.00      0.00        75

    accuracy                           0.46       140
   macro avg       0.23      0.50      0.32       140
weighted avg       0.22      0.46      0.29       140
```

Fig 5.1.3.2.2 Model accuracy & Model Loss plots, score, and classification report of VGG16

5.1.3.4 Proposed Model:

```
[7] Epoch 89/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.1042 - accuracy: 0.9698 - val_loss: 0.0048 - val_accuracy: 1.0000
    Epoch 90/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.1133 - accuracy: 0.9684 - val_loss: 0.0117 - val_accuracy: 1.0000
    Epoch 91/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.0781 - accuracy: 0.9849 - val_loss: 0.0251 - val_accuracy: 0.9857
    Epoch 92/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.0419 - accuracy: 0.9927 - val_loss: 0.0263 - val_accuracy: 0.9857
    Epoch 93/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.0214 - accuracy: 0.9934 - val_loss: 0.0071 - val_accuracy: 1.0000
    Epoch 94/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.0781 - accuracy: 0.9926 - val_loss: 0.0195 - val_accuracy: 0.9857
    Epoch 95/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.1241 - accuracy: 0.9850 - val_loss: 0.0158 - val_accuracy: 0.9857
    Epoch 96/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.0343 - accuracy: 0.9930 - val_loss: 0.0106 - val_accuracy: 0.9929
    Epoch 97/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.0541 - accuracy: 0.9853 - val_loss: 0.0026 - val_accuracy: 1.0000
    Epoch 98/100
    56/56 [==============================] - 4s 73ms/step - loss: 0.1063 - accuracy: 0.9890 - val_loss: 0.0471 - val_accuracy: 0.9929
    Epoch 99/100
    56/56 [==============================] - 4s 73ms/step - loss: 0.1492 - accuracy: 0.9663 - val_loss: 0.0075 - val_accuracy: 0.9929
    Epoch 100/100
    56/56 [==============================] - 4s 72ms/step - loss: 0.0397 - accuracy: 0.9908 - val_loss: 0.0115 - val_accuracy: 0.9929
```

Fig 5.1.3.4.1 Training results of proposed architecture

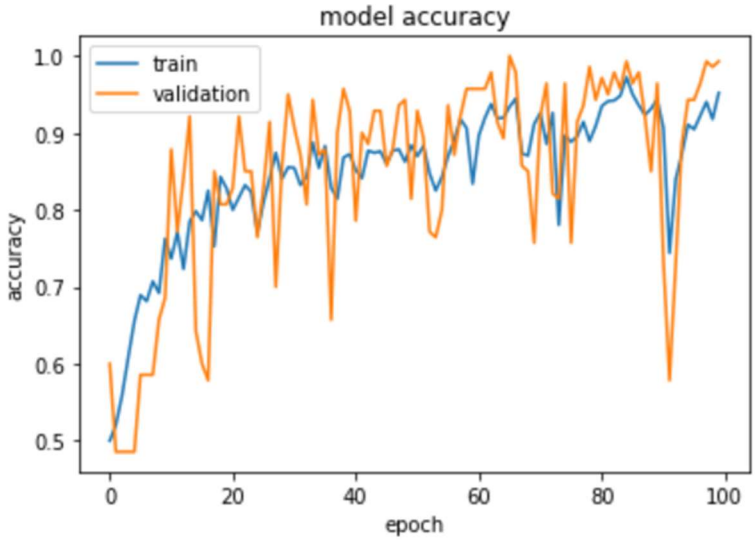


Fig 5.1.3.4.2 Model accuracy plot of proposed architecture

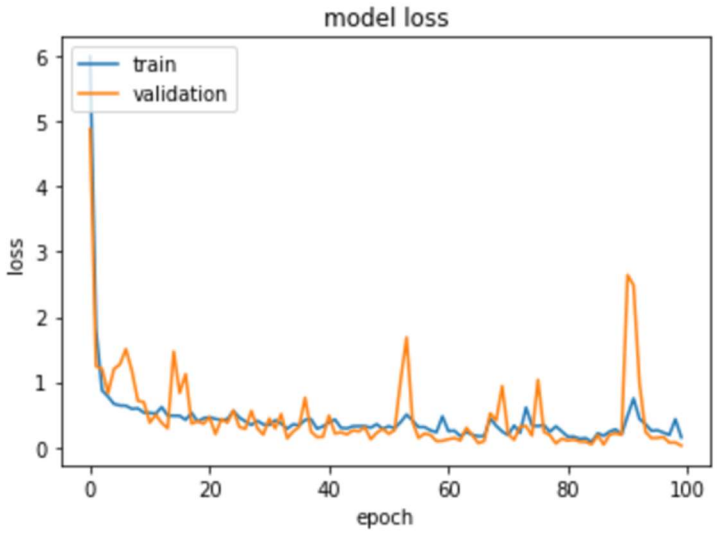


Fig 5.1.3.4.3 Model loss plot of proposed architecture

```
score:  [0.02355821058154106, 0.9928571581840515]
Wrong predictions:  1
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        65
           1       1.00      0.99      0.99        75

    accuracy                           0.99       140
   macro avg       0.99      0.99      0.99       140
weighted avg       0.99      0.99      0.99       140
```

Fig 5.1.3.4.4 Classification report of proposed architecture

5.2 Evaluation metrics:

True Positive (TP): The predicted and actual values are identical. This means that the model predicted a positive value, and the actual value was positive.

True Negative (TN): The predicted and actual values are identical. This means that the model predicted a negative value, and the actual value was negative.

False Positive (FP): The expected value turned out to be incorrect. While the actual value was negative, the model predicted it to be positive.

False Negative (FN): The expected value turned out to be incorrect. While the actual value was positive, the model predicted it to be negative.

Precision: Precision tells us how many of the correctly predicted cases actually turned out to be positive.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall: Recall tells us how many of the actual positive cases we were able to predict correctly with our model.

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1 Score: F1-score is a harmonic mean of Precision and Recall. It provides a combined picture of these two metrics. F1-score reaches its peak when precision equals recall.

$$\text{F1 - score} = \frac{2}{\frac{1}{Recall} + \frac{1}{Precision}}$$

Macro Average: It is the average of either the precision, recall or the f1-score.

$$\text{Macro average Precision} = \frac{P1 + P2}{2}$$

Weighted Average: It is the weighted average of either the precision, recall or the f1-score.

5.3 Comparison Analysis:

Model	Accuracy	Precision		Recall		F1 Score		Train loss	Validation loss
		0	1	0	1	0	1		
LeNet	88.57	0.84	0.91	0.91	0.85	0.87	0.88	2.4172e-05	0.9127
AlexNet	89.99	0.88	0.92	0.91	0.89	0.89	0.91	0.2752	0.9657
VGG16	51.43	0.46	0.00	1.00	0.00	0.63	0.00	0.6931	0.6930
Proposed Model	99.28	0.98	1.00	1.00	0.99	0.99	0.99	0.1812	0.0235

Table 5.4 Comparison of Evaluation metrics of experimented models

5.4 Conclusion

After comparing Fig 5.1.2.9, Fig 5.1.2.10 with image shown in Fig 5.1.2.8, we observe that the watershed segmentation algorithm can detect the objects(nodules) and plot a contour around the objects which represent the edge/boundaries of that object, whereas, in the k-means clustering algorithm, it only assigns a particular colour to each cluster but cannot differentiate between the objects. So, the watershed segmentation algorithm is found to be more advantageous than the K-means clustering algorithm.

Our proposed model was able to achieve higher accuracy (99.28) and performance when compared to other architectures and models we came across during literature survey.

When tested on a new dataset (test dataset), our model was able to classify 138 images out of 140 correctly with 2 wrong predictions.

Chapter 6:

6.1 Future Work:

6.1.1 Segmentation:

We can use the Mask R-CNN technique for segmentation. This technique gives three outputs for each object in the image: its class, bounding box coordinates, and object mask. It is the most advanced image segmentation technique currently available. It has a high training time and hence can be used when time is not a constraint. It is also computationally expensive.

6.1.2 CNN:

We can use parallel convolution layers to get better accuracy. We can also use sparse networks to try and isolate some of the neurons for better training and to avoid overfitting. Better architectures could be used if computation time and computation power are not a constraint.

References:

[1] Suren Makajua, P.W.C. Prasad, Abeer Alsadoon, A. K. Singh, A. Elchouemi
"**Lung Cancer Detection using CT Scan Images"**, 6th International Conference on Smart Computing and Communications, 2018.

[2] K.Mohanambal , Y.Nirosha , E.Oliviya Roshini , S.Punitha , M.Shamini, **"Lung Cancer Detection Using Machine Learning Techniques",** International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, Vol. 8, Issue 2, February 2019.

[3] Jeyaprakash Vasanth Wason and Ayyappan Nagarajan, **"Image processing techniques for analyzing CT scan images towards the early detection of lung cancer",** Biomedical Informatics, Department of Computer Applications, Alagappa University, September 12, 2019.

[4] Ibrahim M. Nasser, Samy S. Abu-Naser**, "Lung Cancer Detection Using Artificial Neural Network",** Department of Information Technology, Faculty of Engineering and Information Technology, Al-Azhar University- Gaza, Palestine 2018.

[5] Margarita Kirienko,1 Martina Sollini,1,2 Giorgia Silvestri,3 Serena Mognetti,3 Emanuele Voulaz,4 Lidija Antunovic,2 Alexia Rossi,1,5 Luca Antiga,3 and Arturo Chit, **Convolutional Neural Networks Promising in Lung Cancer T-Parameter Assessment on Baseline FDG-PET/CT,** 2019.

[6] Vaishnavi. D, Arya. K. S, Devi Abirami. T, M. N. Kavitha, **"Lung Cancer Detection Using Machine Learning",** INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY, VOLUME 7, ISSUE 01, 2019

[7] K.Narmada,G. Prabakaran,**"An Effective Lung Cancer Detection And Classification Using Enhanced Fully Convolutional Neural Networks",**INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH VOLUME 9, ISSUE 01, JANUARY 2020

[8] Ashok Kumar Yadav, Ramnaresh, Kamaldeep Joshi, Robin Singh, Shagun Rana, Ashish Krishan, Ritika Sharma,"**Lung Cancer Detection by using Adam Algorithm and Convolutional Neural Network",**International Journal of Engineering, Applied and Management Sciences Paradigms,2019

[9] Ibtihal D. Mustafa, Mawia A. Hassan,"**A Comparison between Different Segmentation Techniques used in Medical Imaging",**American Journal of Biomedical Engineering, 2016

[10] Asifullah Khan, Anabia Sohail1, Umme Zahoora1, and Aqsa Saeed Qureshi1" **A Survey of the Recent Architectures of Deep Convolutional Neural Networks",**Published in Artificial Intelligence Review, DOI: https://doi.org/10.1007/s10462-020-09825-6

Appendix A

Image Pre-Processing:

```
clc;
close all;
clear all;
dicomlist = dir(fullfile('D:\Dataset\','dicom_dir','*.dcm'));
result='D:\Dataset\PreProcessed';
display(numel(dicomlist))
for cnt = 1 : numel(dicomlist)
    display(cnt)
    O{cnt} =
dicomread(fullfile('D:\Dataset\','dicom_dir',dicomlist(cnt).name));
    I=O{cnt};
    Q = medfilt2(I);
    Z = imgaussfilt(Q,2);
    level=graythresh(Z);
    BW=imbinarize(Z,level)
    D = -bwdist(~BW);
    mask = imextendedmin(D,2);
    figure
    D2 = imimposemin(D,mask);
    Ld = watershed(D2);
    bw2=BW;
    bw2(Ld==0)=0;
    temp=imfuse(bw2,Z,'diff');
    imshow(temp)
    im=temp;
    b= im2double(im);
    c=imresize(b, [512 512]);
    name=dicomlist(cnt).name;
    baseFileName = sprintf('%s', name);
    fullFileName = fullfile(result, baseFileName);
    dicomwrite(c, fullFileName);
end
```

Appendix B

CNN & Data Augmentation:

```
import pandas as pd
import re
import os
from glob import glob
import numpy as np # matrix tools
import matplotlib.pyplot as plt # ftor basic plots
import seaborn as sns # for nicer plots
from skimage.io import imread
import keras
import pydicom as dicom
import cv2
from google.colab import drive
import tensorflow as tf
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
import random
from scipy import ndarray
import skimage as sk
from skimage import transform
from skimage import util
import imageio
import imgaug as ia
import imgaug.augmenters as iaa
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.optimizers import Adam,SGD
from keras.layers import Conv2D, MaxPooling2D, Dropout, MaxPool2D
from keras.callbacks import EarlyStopping, EarlyStopping
from sklearn.metrics import classification_report
drive.mount('/content/gdrive')
datapath='/content/gdrive/MyDrive/PreProcessed/'
overview_df = pd.read_csv((os.path.join(datapath,'overview.csv')))
overview_df.columns = ['idx']+list(overview_df.columns[1:])
overview_df['Contrast'] = overview_df['Contrast'].map(lambda x: 'Contrast' if x else 'N
o Contrast')
all_images_list = glob(os.path.join(datapath,'dcm','*.dcm'))
```

```
check_contrast = re.compile(r'ID_([\d]+)_AGE_[\d]+_CONTRAST_([\d]+)_CT')
label = []
id_list = []
for image in all_images_list:
    id_list.append(check_contrast.findall(image)[0][0])
    label.append(check_contrast.findall(image)[0][1])
label_list = pd.DataFrame(label,id_list)
IMG_SIZE = 256
################################################################################################
################
def horizontal_flip(image_arrays):
    num_files_desired = 100
    num_generated_files = 0
    num_transformations_to_apply = 1
    result=[]
    while num_generated_files < num_files_desired:
      image_to_transform = image_arrays[num_generated_files]
      num_transformations = 0
      transformed_image = None
      if num_transformations <= num_transformations_to_apply:
          transformed_image = image_to_transform[:,::-1]
          result.append(transformed_image)
          num_transformations += 1
      num_generated_files += 1
    num_generated_files = 0
    return result
def vertical_flip(image_arrays):
  result=[]
  for i in image_arrays:
    temp=np.array(list(reversed(i)))
    result.append(temp.tolist())
  return result
def scale(image_arrays):
  result=[]
  for i in image_arrays:
    scale_im=iaa.Affine(scale={"x": (0.9, 1.0), "y": (0.9, 1.0)})
    scale_image =scale_im.augment_image(i)
    result.append(np.array(scale_image))
  return result
def shear_images(image_arrays):
  result=[]
  for i in image_arrays:
      shear = iaa.Affine(shear=(0,20))
      shear_img=shear.augment_image(i)
      result.append(np.array(shear_img))
  return result
```

```
#################################################################################################
################
images=[]
imgs=[]
#1
for i in all_images_list:
    x = cv2.resize(dicom.dcmread(i).pixel_array,(IMG_SIZE,IMG_SIZE))
    imgs.append(x)
    images.append(x)
#2
sheared = shear_images(imgs)
for i in sheared:
  images.append(i)
#3
scaled = scale(imgs)
for i in scaled:
  images.append(i)
#4
h_flip = horizontal_flip(imgs)
for i in h_flip:
  images.append(i)
#5
v_flip = vertical_flip(imgs)
for i in v_flip:
  images.append(i)
#6
temp6 = horizontal_flip(np.array(sheared))
for i in temp6:
  images.append(i)
#7
temp7 = horizontal_flip(np.array(scaled))
for i in temp7:
  images.append(i)
#8
temp8 = horizontal_flip(np.array(v_flip))
for i in temp8:
  images.append(i)
#9
temp9 = vertical_flip(np.array(sheared))
for i in temp9:
  images.append(i)
#10
temp10 = vertical_flip(np.array(scaled))
for i in temp10:
  images.append(i)
#11
```

```
temp11 = shear_images(np.array(temp7))
for i in temp11:
  images.append(i)
#12
t12 = scale(np.array(h_flip))
te12 = shear_images(np.array(t12))
temp12 = vertical_flip(np.array(te12))
for i in temp12:
  images.append(i)
#13
t13 = scale(imgs)
te13 = shear_images(np.array(t13))
temp13 = vertical_flip(np.array(te13))
for i in temp13:
  images.append(i)
#14
temp14 = shear_images(np.array(scaled))
for i in temp14:
  images.append(i)
images = np.array(images)
dataset_size = len(images)
label_size = len(label_list)
temp = dataset_size/label_size
frames=[]
for i in range(int(temp)):
  frames.append(label_list)
labels_list = pd.concat(frames)
X_train, X_test, y_train, y_test = train_test_split(images, labels_list, test_size=0.1,
 random_state=23)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.111, ra
ndom_state=21)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_val = X_val.astype('float32')
output_train = keras.utils.to_categorical(y_train, 2)
output_test = keras.utils.to_categorical(y_test, 2)
output_val = keras.utils.to_categorical(y_val, 2)
input_shape=(IMG_SIZE,IMG_SIZE,1)
height=IMG_SIZE
depth=1
width=IMG_SIZE
input_train = X_train.reshape((X_train.shape[0], width,height,depth))
input_train.shape
input_train.astype('float32')
input_train = input_train / np.max(input_train)
input_test = X_test.reshape(X_test.shape[0], *input_shape)
```

```
input_test.astype('float32')
input_test = input_test / np.max(input_test)
input_val = X_val.reshape(X_val.shape[0], *input_shape)
input_val.astype('float32')
input_val = input_val / np.max(input_val)
batch_size = 20
epochs=100
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='rel
u', input_shape=input_shape),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu
', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Dropout(0.7),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu
', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu
', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(1,1), strides=(1,1), activation='relu
', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Dropout(0.7),
    keras.layers.Conv2D(filters=512, kernel_size=(5,5), strides=(1,1), activation='relu
', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=512, kernel_size=(5,5), strides=(1,1), activation='relu
', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(1,1), strides=(1,1), activation='relu
', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Dropout(0.7),
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.7),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.7),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.7),
```

```
    keras.layers.Dense(2, activation='softmax')
])
model.summary()


early = EarlyStopping(monitor='val_accuracy', min_delta=0, patience=30, verbose=1, mode
='auto')
model.compile(loss='binary_crossentropy', optimizer=Adam(), metrics=['accuracy'])
history = model.fit(input_train, output_train,batch_size=batch_size, epochs=epochs, ver
bose=1, validation_data=(input_val, output_val),callbacks=early)
from sklearn.metrics import classification_report
#'Accuracy'
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
# "Loss"
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')

plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
score = model.evaluate(input_val, output_val, verbose=0) #validation_loss & validation_
accuracy
print("score: ",score)
predictions = model.predict(input_test)
pred=np.rint(predictions)
y_pred = np.argmax(predictions,axis=1)
y_true = np.argmax(output_test,axis=1)
count=0
for i in range(len(y_true)):
    if y_pred[i]!=y_true[i]:
        count=count+1
print("Wrong predictions: ",count)
print(classification_report(y_true, y_pred,labels=[0,1]))
model_save_name = 'Proposed_Model'
path = F"/content/gdrive/My Drive/{model_save_name}"
model.save(path)
```

'LUNG CANCER DETECTION AND CLASSIFICATION USING IMAGE PROCESSING AND CNN-DEEP LEARNING ARCHITECTURE'

ORIGINALITY REPORT

30%	19%	19%	20%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Ahsanullah University of Science and Technology Student Paper	4%
2	Submitted to University of East London Student Paper	2%
3	www.analyticsvidhya.com Internet Source	2%
4	Ivana Shopovska, Ljubomir Jovanov, Wilfried Philips. "Efficient Training Procedures for Multi-Spectral Demosaicing", Sensors, 2020 Publication	1%
5	machinelearningmastery.com Internet Source	1%
6	www.scribd.com Internet Source	1%
7	medium.com Internet Source	1%