DE-41 (EE) ABDUL REHMAN, AYESHA, MASHHOOD

FPGA IMPLEMENTATION OF CNN FOR LUNG CANCER DIAGNOSIS

COLLEGE OF
ELECTRICAL AND MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY
RAWALPINDI
2023

COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING

DE-41 EE
PROJECT REPORT

FPGA IMPLEMENTATION OF CNN FOR LUNG CANCER DIAGNOSIS

Submitted to the Department of Electrical Engineering
in partial fulfillment of the requirements
for the degree of
Bachelor of Engineering
in
Electrical
2023

Submitted By:

Abdul Rehman Faisal
Ayesha Babar
Mashhood Ahmad Khan

CERTIFICATE OF APPROVAL

It is to certify that the project **"FPGA IMPLEMENTATION OF CNN FOR LUNG CANCER DIAGNOSIS"** is done by **NS Abdul Rehman Faisal, NS Mashhood Ahmad Khan,** and **NS Ayesha Babar** under the supervision of **Dr. Shahzad Amin Sheikh** and **A/P Kamran Aziz Bhatti.**

Submission: This project is submitted to the College of Electrical and Mechanical Engineering (Peshawar Road Rawalpindi), National University of Sciences and Technology, Pakistan, as part of the Bachelor of Electrical Engineering degree program.

Students:

1. Abdul Rehman Faisal

NUST ID: ________________________ Signature: ____________________

2. Ayesha Babar

NUST ID: ________________________ Signature: ____________________

3. Mashhood Ahmad Khan

NUST ID: ________________________ Signature: ____________________

Approved By:

Project Supervisor: __________________________ Date: ______________

A/P Kamran Aziz Bhatti

Project Co-Supervisor: ______________________ Date: ______________

Dr. Shahzad Amin Sheikh

DECLARATION

We thus certify that no part of this Project Thesis has been submitted in support of an application for another degree or qualification from this or any other university or other educational institution. We are totally liable for any disciplinary action taken against us based on the nature of the proved offence, including the revocation of our degree.

1. **Abdul Rehman Faisal ____________________________________**

2. **Ayesha Babar ____________________________________**

3. **Mashhood Ahmad Khan ____________________________________**

ACKNOWLEDGEMENTS

First and foremost, we are humbled and grateful for the blessings and guidance bestowed upon us by Allah Almighty. It is his divine intervention and blessings that we have been able to overcome numerous challenges throughout our project.

We are deeply indebted to our supervisors; Sir Kamran Aziz Bhatti and Sir Shahzad Amin Sheikh whose continuous guidance, expertise, and patience have been instrumental in bringing this project to fruition.

Lastly, we are immensely grateful to our families who acted as our support systems and their unwavering love, encouragement, and sacrifices have been the foundation upon which we successfully completed this project.

ABSTRACT

Lung cancer is a fatal disease taking more than 1.8 million lives every year, necessitating timely and accurate diagnosis for effective treatment. This research investigates the utility of deep learning models to diagnose lung cancer and its hardware implementation on an FPGA (Field Programmable Gate Array). Notably, this research is distinguished due to the utilization of a dataset consisting of bone scans. The dataset comprises bone scans of more than 3247 patients, where some cases exhibit bone metastasis. This dataset undergoes stages of comprehensive processing to standardize image resolutions and remove any potential artifacts. Subsequently, CNN (Convolutional Neural Network) models are trained and evaluated using these bone scans in order to extract relevant features and classify them according to the presence or absence of metastatic lung cancer. The performance of the four CNN architectures and hyperparameter configurations is evaluated using accuracy, precision, recall, and F1 score metrics. A hardware implementation of the trained model is realized on an FPGA due to its efficient and parallel processing capability, enabling effective diagnosis. In the latter chapters, this thesis evaluates the performance and computational efficiency of FPGA, considering factors such as resource utilization, inference speed, and power consumption.

SUSTAINABLE DEVELOPMENT GOALS

Pakistan affirmed its commitment to the 2030 Agenda for Sustainable Development by adopting the Sustainable Development Goals (SDGs) as its own national development agenda through a unanimous National Assembly Resolution in 2016. Since then, the country has made considerable progress by mainstreaming these goals in national policies and strategies and developing an institutional framework for SDGs implementation in Pakistan.

Our project is working towards the following SDGs:

Ensure healthy lives and promote well-being for all, by making healthcare more accessible.

Build resilient infrastructure, promote inclusive and sustainable industrialization, foster innovation.

Reducing inequality within country by providing affordable healthcare to all regardless of income/status.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

SCLC	Small cell lung cancer
AI	Artificial Intelligence
CAD	Computer-Aided Diagnostics
CNN	Convolutional Neural Networks
DL	Deep Learning
CT Scan	Computed Tomography Scan
FPGA	Field Programmable Gate Array
ReLU	Rectified Linear Unit
DNN	Deep Neural Network

CHAPTER 1 – INTRODUCTION

1.1. Background Information

Cancer has emerged as a significant contributor to global mortality, affecting approximately one in every six individuals annually. With an estimated 10 million deaths occurring each year, cancer stands as a prominent factor in the loss of human lives worldwide. Cancer is a complex group of diseases characterized by the uncontrolled growth and spread of abnormal cells in the body. Unlike normal cells in the body which grow, divide, and eventually die in a regulated manner; cancer cells do not follow this orderly process and continue to divide and grow uncontrollably, forming a mass of cells called Tumour.

Lung cancer stands as the foremost contributor to cancer-related fatalities, accounting for approximately 1.5 million global deaths annually. Metastasis is the primary factor behind the overwhelming majority of lung cancer deaths; it refers to the process by which cancer cells break away from the original tumour in a certain organ (such as the lungs) and spread to other parts of the body through the bloodstream or lymphatic system. These cancer cells can then form new tumours, known as metastatic tumours, in different organs or tissues, contributing to the progression and severity of the disease. The spread of cancer cells to other parts of the body hampers treatment effectiveness and leads to severe complications, resulting in significant loss of life. The diagnosis of cancer at early stages is important to increase the chances of successful treatment and prognosis; enabling less aggressive and invasive treatment options, reducing potential side effects, and lowering the cost of cancer treatment. Detecting skeletal metastasis in its early stages is vital to decrease morbidity and disease staging, outcome prediction and treatment planning.

1.2. Types of Lung Cancers

Lung cancer is a type of cancer that originates in the lungs and may spread to lymph nodes or other parts of the body. It is characterized by the uncontrolled growth of abnormal cells in lung tissues which can interfere with normal lung function.

Lung Cancer can be mainly divided into two types:

i) Small cell lung cancer (SCLC): These are also classed as neuroendocrine tumours since they originate from the neuroendocrine tissue of the lungs. These are caused by smoking and account for 15-20 % of lung cancers diagnosed.

ii) Non-small cell lung cancer (NSCLC): These are the most common type of lung cancer accounting for at least 80-85% of all lung cancers diagnosed. It has several types:

a. Adenocarcinoma: It often begins from the peripheral regions of the lungs and is associated with smoking.

b. Squamous cell carcinoma: It is also known as epidermoid carcinoma and originates from the airways of the lungs.

c. Large cell carcinoma: It is the least common subtype of NSCLC and can originate from any part of the lungs.

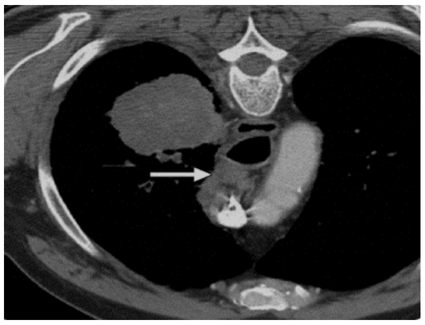


Figure1.1. Bone scan of SCLC

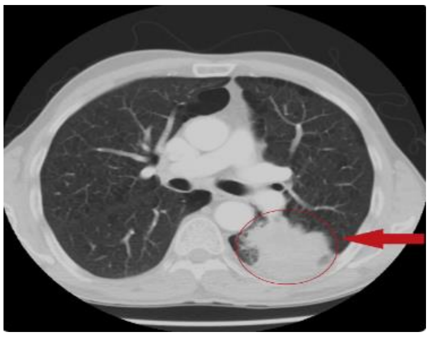


Figure1.2. Bone scan of adenocarcinoma

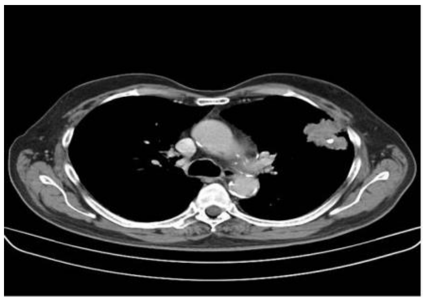


Figure1.3. Bone scan of small cell carcinoma

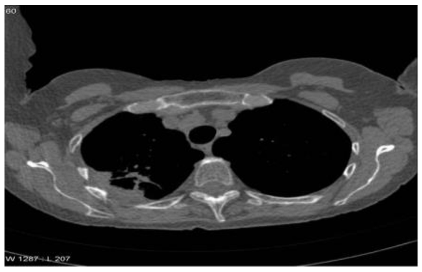


Figure1.4. Bone scan of large cell carcinoma

1.3. Diagnostic techniques

1.3.1. Introduction

The primary approaches utilized for diagnosing bone metastasis predominately rely on noninvasive diagnostic imaging techniques. The invasive procedures to diagnose lung cancer are used less often due to the complications and patient discomfort associated with them. These procedures involve higher costs as compared to non-invasive techniques. The diagnostic yield of non-invasive techniques such as CT scans or bone scans is also considered better than that of invasive procedures. In many cases, non-invasive techniques provide sufficient diagnostic information without the need for invasive procedures.

1.3.2. Invasive Diagnostic Procedures

Here are some of the invasive diagnostic procedures used in diagnosing Lung Cancer:

i) Needle biopsy: It is also known as percutaneous biopsy because it involves the insertion of the needle through the skin into the lung tissue to extract sample tissue for analysis.
   It can be further classified into:
   a. Transthoracic Needle Biopsy
   b. Endobronchial Ultrasound-Guided Biopsy (EBUS)
   c. Transbronchial Needle Aspiration (TBNA)

ii) Bronchoscopy: It is a procedure to allow direct visualization of the airways by inserting a thin, flexible bronchoscope (tube-like) through the mouth or nose of the patient.

iii) Thoracoscopy: It is a video-assisted thoracoscopic surgery involving the insertion of a thoracoscope (a thin tube with a camera) into the small incisions made in the chest wall. This technique allows the physician to access the lungs and pleura in the chest.

iv) Mediastinoscopy: It is a surgical procedure involving the insertion of a scope to examine the mediastinum (the space between the lungs) and obtain tissue samples from lymph nodes.

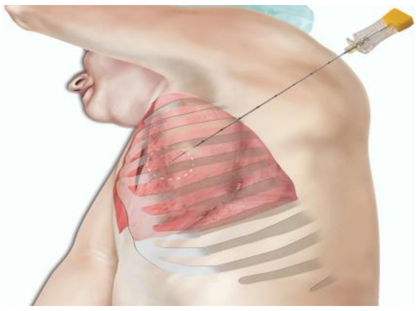


Figure1.5. Needle biopsy

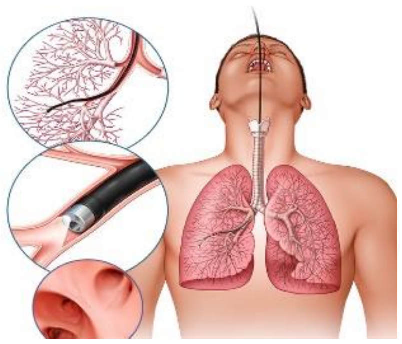


Figure1.6. Bronchoscopy

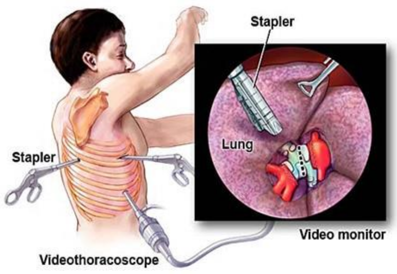


Figure1.7. Thoracoscopy

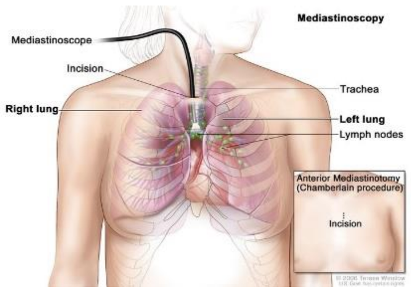


Figure1.8. Mediastinoscopy

1.3.3. Non-invasive Diagnostic Procedures

These are commonly used to diagnose lung cancer and do not involve the insertion of instruments into the body of the patient. Some of the techniques are listed below:

i) Imaging tests

   a. Chest X-ray: It uses low levels of radiation to create images of the lungs. It can detect abnormalities in the lungs such as masses or nodules.
   b. Computerized tomography (CT): CT scans provide cross-sectional images of the chest to help detect smaller lesions and the size, location, and shape of the tumour.
   c. Magnetic Resonance Imaging (MRI): It is an imaging test that assesses the involvement of structures nearby lesions. It uses powerful magnets to generate detailed images of the lungs.
   d. Bone scans: Bone scans are the primary tests to diagnose lung cancer and determine bone metastasis.

ii) Positron Emission Tomography: This imaging test utilizes a radioactive tracer that is injected into the body to detect areas of high metabolic activity. Cancer cells have increased metabolic activity and PET scans can help detect tumours and metastasis.

iii) Sputum Cytology: This test utilizes sputum samples coughed up from the lungs of the patient. This sputum is studied under a microscope to detect any cancerous or precancerous changes in the lungs.

iv) Blood tests: These tests are employed to identify any biomarkers associated with lung cancer such as carcinoembryonic antigen (CEA) or other genetic mutations.

v) Liquid Biopsy: This test involves testing the body fluids or urine of the patient to help identify any circulating tumour cells (CTCs) or fragments of tumour DNA.

The following images below show how lung cancer appears under various non-invasive diagnostic procedures.

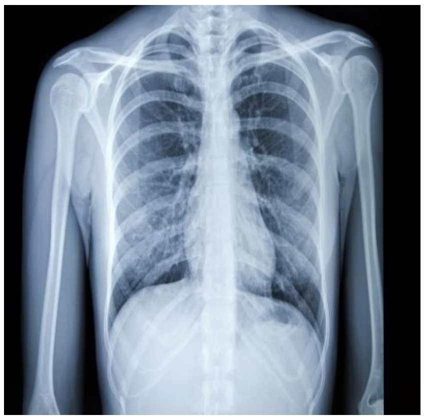


Figure1.9. Chest X-ray

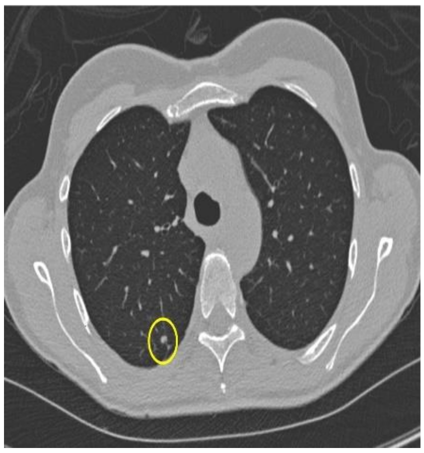


Figure1.10. CT Scan

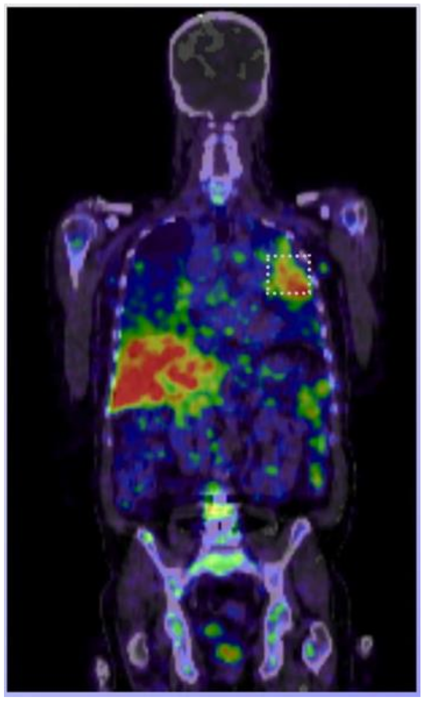


Figure1.11. PET Scan

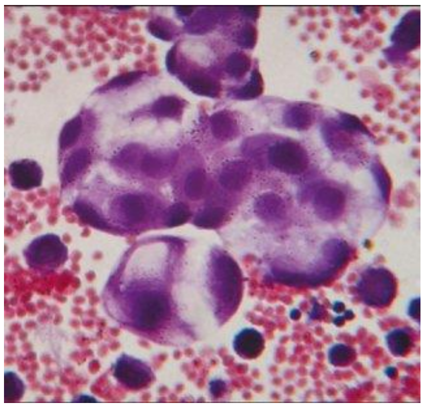


Figure1.12. Sputum Cytology

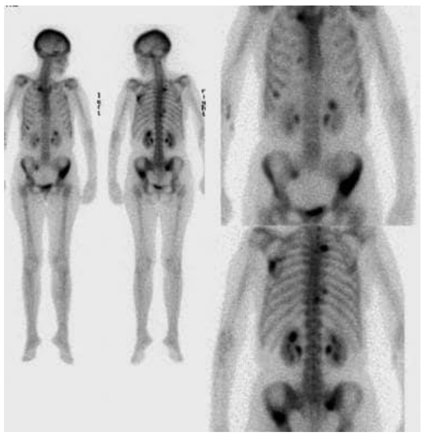


Figure1.13. Bone Scans

1.4. Inspecting the Utility of bone scans in Lung Cancer Diagnosis

Bone scans (skeletal scintigraphy) with technetium-99 methylene phosphonic acid are employed in this project due to their ability to provide a holistic assessment of the entire skeletal system enabling the detection of metastasis in multiple bone sites simultaneously. The procedure involves injecting a small amount of a radioactive substance into the bloodstream, which gathers in areas of increased bone activity, such as sites affected by metastatic cancer. Special cameras then detect the emitted radiation and create detailed images of the skeleton, helping identify any abnormal areas. Bone scans have the ability to detect both primary cancers and bone metastasis which is a cancer that has spread to bones from other parts of the body.

In contrast, CT and PET scans have limitations in capturing the entire skeletal system in a single image. Additionally, Bone scans have high sensitivity for detecting bone metastasis, even in the early stages or when lesions are small. This is particularly important for diagnosing metastatic bone involvement, as CT and PET scans may not always be as sensitive in detecting such lesions. Bone scans have relatively higher sensitivity, specificity, positive predictive value, and negative predictive value than

CT/PET scans.

Moreover, bone scans can also be performed relatively quickly and are generally well-tolerated by patients. Due to more holistic and accurate results provided by bone scans as compared to other diagnostic techniques, bone scans have been utilized in this research for diagnosing lung cancer.

CHAPTER 2 – CANCER DIAGNOSIS AND AI

2.1. Introduction

The use of artificial intelligence in the diagnosis and prognosis of lung cancer has great potential in the field of medical sciences. Artificial intelligence is a tool that pathologists can utilize to analyze large amounts of patient data. However, it is important to note that AI in cancer diagnosis is still evolving and its usage in clinical diagnosis and prognosis requires careful validation and integration with doctor’s judgment. While AI has the potential to improve the accuracy and efficiency of lung cancer diagnosis, it is currently used as an auxiliary tool to help healthcare professionals rather than replace their expertise.

2.2. Comparison between AI and Traditional Medicine

Traditional medicine is a term that refers to the conventional approach to medicine that completely depends on the clinical expertise and discretion of healthcare professionals to diagnose lung cancer. It does not involve the integration of AI or advanced technologies.

Computer Aided Diagnosis (CAD) is a term that refers to the use of AI or computer technologies to assist and aid healthcare professionals to diagnose diseases and predict prognosis.

AI algorithms have shown favorable results in achieving high levels of objectivity and accuracy in lung cancer diagnosis. AI algorithms can detect subtle abnormalities in lung cancer diagnosis that may be overlooked by pathologists, reducing the rates of false positive and false negative outcomes.

The time response of AI is better than that of healthcare professionals since it can analyze large amounts of data such as patient records or medical images in a shorter span of time. AI can rapidly learn features and patterns that may be indicative of lung cancer, aiding in early diagnosis and treatment of lung cancer.

AI systems can provide decision support to healthcare professionals in making more informed diagnosis and provide prognosis more efficiently.

2.3. Use of Convolutional Neural Networks in lung cancer diagnosis

A convolutional neural network (CNN) is a subset of machine learning particularly designed for visual inputs such as videos and images. It is a kind of deep learning architecture that is specifically involved in pixel manipulation and image classification. The architecture of CNN is inspired by the visual cortex of the human brain which consists of multiple nodes each performing a particular operation on visual input.

CNNs have the ability to detect and locate lung nodules which are small round lesions indicative of lung cancer. CNNs can be trained to identify such nodules in bone scans and classify bone scan images as cancerous and non-cancerous, assisting radiologists in the early diagnosis of lung cancer.

Each layer in a CNN architecture plays a crucial role in extracting meaningful features and aiding in detecting and classifying lung cancer accurately.

The input layer in a CNN architecture receives medical image data (bone scans in this research). The image data prior to being fed into the input layer is preprocessed to normalize pixel values.

Convolutional layers consist of filters or kernels that perform the operation of convolution on each input image. These filters tend to extract important features such as edges, shape, and location of tumour.

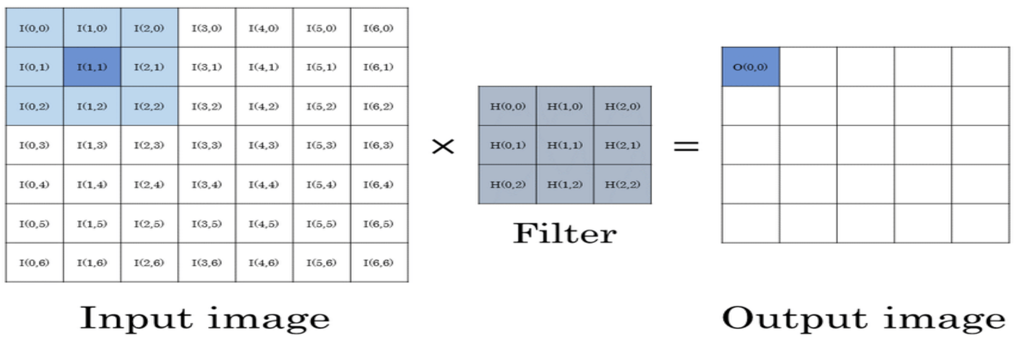


Figure 2.1. Convolutional Layer

Activation functions such as ReLU add non-linearity into the CNN architecture to help capture intricate relationships within the extracted features and enable the CNN to understand the complex patterns associated with lung cancer.

Pooling layers tend to reduce the spatial dimensions of feature maps and retain the most salient features that contribute to lung cancer diagnosis.

Fully connected layers, in the end, produce the classification results from the learned features and help determine whether the patient has cancer. The details of CNN are further explained in the latter chapters of this thesis.

CHAPTER 3 – DATA ACQUISITION AND PRE-PROCESSING

3.1. Introduction

Image preprocessing refers to a series of operations or techniques applied to an image before it is used for further analysis or processing. It involves manipulating the image data to enhance its quality, remove noise or artifacts, standardize the format or size, and prepare it for subsequent tasks such as feature extraction, classification, or object detection. Image Processing of bone scans is a crucial step in this project to enhance the quality of data being fed into the CNN models for it to accurately diagnose and classify cancer. Image processing techniques are employed to enhance the bone scan images to improve their clarity, contrast, and visibility of lung structures. Moreover, Image processing algorithms are used to isolate the regions of interest from the bone scan images. By specifically focusing on these regions, CNN models can prioritize the analysis of lung-related features, effectively reducing irrelevant information and noise that may be present in the bone scans. Image processing techniques also assist in extracting relevant features from the bone scan images that are indicative of lung cancer. These techniques include edge detection, texture analysis, or morphological operations to identify potential tumour regions. These extracted features are then directly fed into CNN models aiding in the detection and classification of lung cancer. Image processing also allows us to augment the datasets by applying transformations such as rotation, scaling, or mirroring the bone scan images. This helps to increase the diversity and variability of the training dataset, enabling CNN models to generalize better and improve their accuracy in predicting cancer. By applying image processing techniques to bone scans, CNN models can benefit from enhanced image quality, focused analysis on lung regions, extraction of relevant features, and increased training data variability which then increases their accuracy to detect lung cancers.

3.2. Acquisition of Bone Scans

A benchmark dataset is used for the purpose of research for this project. BS-80K is the first open-source bone scan dataset consisting of 82544 bone scan images associated with 3247 patients from the West China Hospital. In BS-80K, each patient provides two whole-body bone scan images corresponding to the anterior and posterior views. For each view, there are 13 anatomical slices of body parts susceptible to bone metastasis. For this project, chest slices are used to train CNN models because lungs are enclosed in the chest and this would result in a more precise diagnostic conclusion.

This project utilizes four different angles of chest bone scan including anterior and posterior views for the right and left sides of the chest to comprehensively analyze any bone metastasis. Four different CNN models are employed on four angles of the chest bone scan to evaluate potential metastatic lesions on four sides of the chest. By examining the scans from different angles, the model can capture a broader range of features and patterns that may indicate the presence of metastasis. The output of the CNN model can assist radiologists and oncologists in the early detection and accurate characterization of metastatic lesions, enabling timely intervention and appropriate treatment planning for patients.

The anterior view of the chest bone scan includes the front portion of the chest capturing the heart, lungs, ribs, sternum, and clavicle. This anterior angle is further divided into right anterior and left anterior chest bone scans. InceptionV3 is applied on the left anterior part of the chest bone scans whereas ConvNeXt is applied on the right anterior part of the chest bone scans to identify and localize any potential lesions in the anterior part of the chest.

The posterior view of a chest bone scan captures the back portion of the chest, providing an assessment of structures such as the spine, ribs, lungs, and surrounding tissues. This view allows for the evaluation of potential abnormalities or pathologies in the posterior aspect of the chest. The posterior angle is further divided into left posterior and right posterior chest bone scans. DenseNet-169 is applied on the left posterior part of the chest bone scans whereas EfficientNet is applied on the right posterior part of the chest bone scans to identify and localize any potential lesions in the posterior part of the chest.

3.3. Data Preprocessing

3.3.1. Image resizing

Image resizing is a crucial preprocessing step performed to ensure uniform dimensions of bone scan images prior to feeding them into CNN models. By resizing the images, they are brought to a consistent size, enabling compatibility and facilitating efficient processing within the CNN models. To achieve consistent dimensions in bone scan images, the approach is to calculate the median height and width of all the images in the dataset and resize the height and width of all the bone scale images to the median value. After resizing, all bone scan images have a uniform dimension of 171x75.

3.3.2. Image enhancement

Image enhancement refers to the pixel-manipulating operations that tend to enhance the salient features of the images to convey meaningful information. This is done in mainly two steps: Noise reduction and Image Sharpening.

3.3.2.1. Noise Reduction

Noise reduction in bone scan images is performed by using Gaussian Filter. The Gaussian filter works by convolving the image with a Gaussian kernel, which is a bell-shaped function that assigns higher weights to the central pixels and gradually decreases the weights as we move away from the center. The Gaussian filter helps reduce noise in bone scan images by smoothing out the high-frequency components that are often associated with noise. By blurring the image slightly, the filter effectively reduces the impact of noise while preserving the overall structure and important features of the image.

3.3.2.2. Image Sharpening

Image sharpening is performed by applying a High pass Filter to the bone scan images. The Laplacian filter enhances fine edges and details by subtracting the blurred version of the image from the original image, improving the visibility and clarity of potential tumour lesions in the lungs. It amplifies the high-frequency components resulting in sharper edges and improved image details.

3.3.3. Data Augmentation

This local bone scan dataset had a class imbalance issue where cancerous images are fewer than non-cancerous ones. Data Augmentation Techniques are employed for cancerous images to replicate them. Furthermore, rotation is performed encompassing different orientations and angles at which cancer is present, application of shear factor to skew or deform the bone scan images around an axis to introduce variations, zoom to introduce variations in the scale and size of bone scan images, and horizontal flip to mirror the bone scan images along the vertical axis. These augmentation techniques contribute to reducing the model’s sensitivity to the exact arrangement or symmetry of lesions in bone scans, making the model more robust and generalized to detect cancer accurately.

3.3.4. Training and Testing Datasets

The training datasets are employed to train a CNN model to help it learn patterns and features in input data and tune its internal parameters to conclude accurate predictions. The larger the training dataset allocated to a CNN model; the better it learns and generalizes. The testing dataset, on the other hand, is used to evaluate the performance of the training dataset. By evaluating the model's performance on this unseen data, we can assess its ability to generalize and make predictions accurately on new, unseen chest bone scan images. The bone scan datasets have been divided into training and testing datasets in the ratio of

0.8:0.2 to provide an estimate of how well the model performs in real-world scenarios.

CHAPTER 4 – DEEP LEARNING MODEL ARCHITECTURES

4.1. Introduction

Our project uses a model based on a Convolutional Neural Network (CNN) to categorize a bone scan as either cancerous or non-cancerous. A CNN is a type of deep learning architecture ideal for processing structured grid-like data, such as images. It utilizes convolutional layers to automatically learn and extract relevant features from the input data. CNNs are especially effective in capturing spatial relationships and patterns in the data by applying filters and pooling operations. This enables them to achieve excellent performance in image classification tasks.

A basic CNN model is shown below:

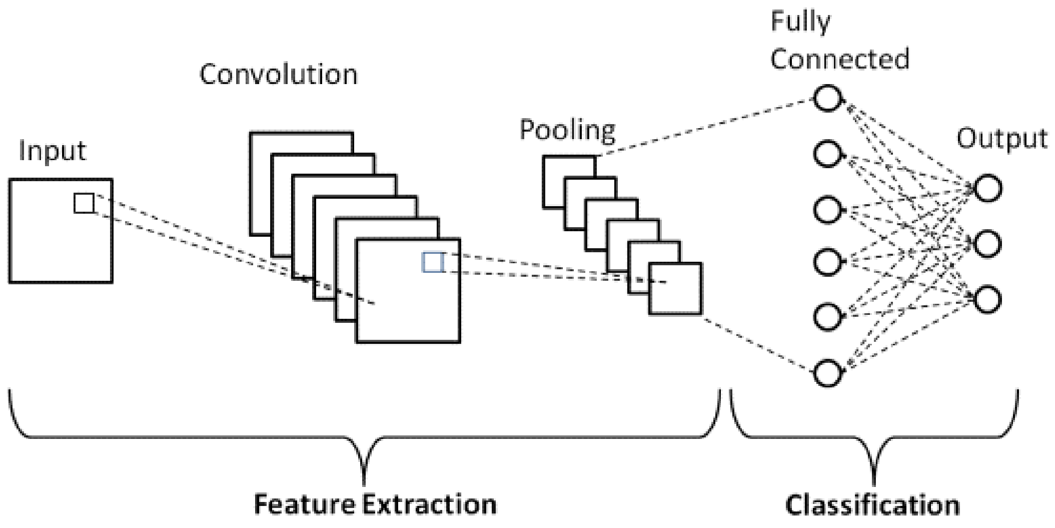


Figure 4.1. Basic CNN model

4.2. Components of a Basic Convolutional Neural Network

Following are some components of a typical CNN:

4.2.1. Convolutional layer

The convolutional layer takes the input image and applies a filter that moves across the image, calculating dot products to extract features.

4.2.2. Pooling layer

The pooling layer reduces the spatial size of the feature map, helping to decrease the computational requirements and extract key features that are invariant to position and rotation.

4.2.3. Flattening layer

After the convolutional and pooling layers, the flattening layer converts the multidimensional feature map into a one-dimensional array, preparing it for the fully connected layer.

4.2.4. Fully connected layer

Following the flattening layer, the fully connected layer receives the flattened feature vector as input. It connects every neuron in the previous layer to each neuron in this layer, allowing for complex combinations of features. This layer performs the final computations and produces the output predictions based on the learned features.

4.2.5. Dropout layer

To prevent overfitting, the dropout layer randomly ignores a portion of neurons during training, temporarily removing their contribution to downstream activation and avoiding excessive training. Dropout values range between 0 and 1 to control the dropout rate.

4.3. CNN Architectures

CNN architectures refer to the specific design and structure of Convolutional Neural Networks. Over the years, various CNN architectures have been developed, each with its own characteristics and strengths. In our project we have experimented with the following architectures:

- InceptionV3
- ConvNext
- DenseNet169
- EfficientNet

4.3.1 Inception-v3

Inception-v3 is a convolutional neural network architecture, also known as GoogLeNet-v3, that is introduced by Google researchers in 2015 as an evolution of the original Inception architecture. It is designed to achieve high accuracy while being more computationally efficient compared to its predecessors.

Inception-v3 focuses on achieving better computational efficiency compared to previous architectures. It achieves this through factorization, efficient use of convolutional filters, and reducing the number of operations.

Some notable features and characteristics of Inception-v3:

Inception Modules: Inception-v3 uses a series of Inception modules, which consist of parallel convolutional layers of different filter sizes (1x1, 3x3, and 5x5) along with max pooling. The outputs of these parallel branches are concatenated and fed into subsequent layers, allowing the network to capture features at different scales.

Factorization: To reduce computational complexity, Inception-v3 employs factorization techniques. It replaces large convolutions with a combination of smaller convolutions, such as a 5x5 convolution being factorized into two 3x3 convolutions. This reduces the number of parameters and operations, making the network more efficient.

Auxiliary Classifiers: Inception-v3 includes auxiliary classifiers at intermediate layers during training. These auxiliary classifiers are used to combat the vanishing gradient problem and provide additional supervision. They encourage the network to learn more discriminative features and help with gradient propagation.

Pre-training on ImageNet: Inception-v3, like many other CNN architectures, is typically pre-trained on large-scale datasets such as ImageNet. This pre-training provides a good initialization for the network's weights and allows the model to leverage learned features before fine-tuning on a specific task or dataset.

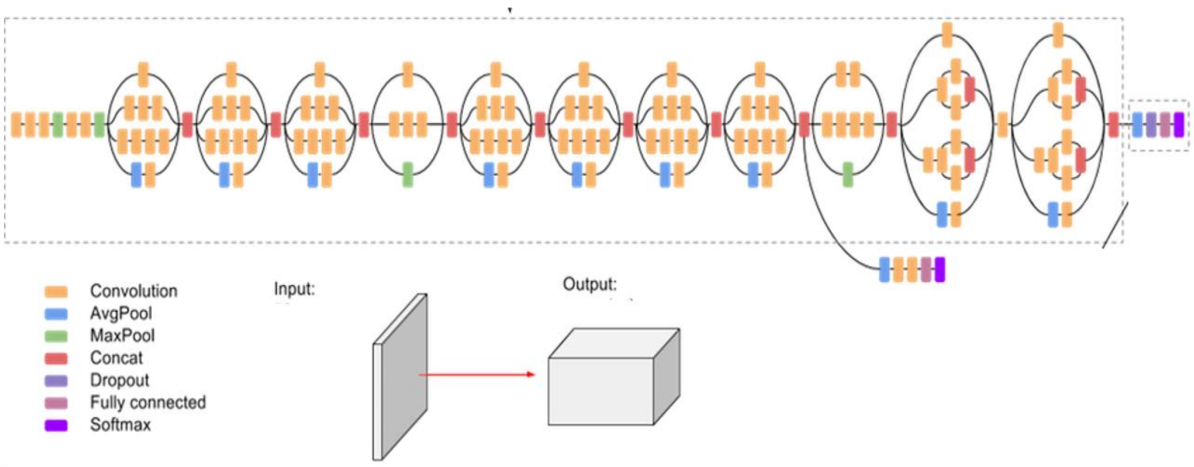


Figure 4.2. Inception-v3 architecture

4.3.2. ConvNeXt

ConvNeXt is a convolutional neural network architecture that was proposed in 2022 by researchers at Facebook AI. It is designed to be a more efficient and accurate alternative to Vision Transformers, which are currently state-of-the-art for image classification.
ConvNeXt achieves its efficiency and accuracy by combining several architectural innovations, including:

Using depthwise separable convolutions, which allow the network to learn more complex features with fewer parameters.

Introducing a new attention mechanism called Swin Transformer, which allows the network to better capture long-range dependencies in images.

Using a new training technique called mixup, which helps the network to learn more robust features.

ConvNeXt has been shown to achieve state-of-the-art results on a variety of image classification benchmarks, including ImageNet and CIFAR-10. It is also more efficient than Vision Transformers, making it a more practical choice for deployment on mobile devices and other resource-constrained platforms.

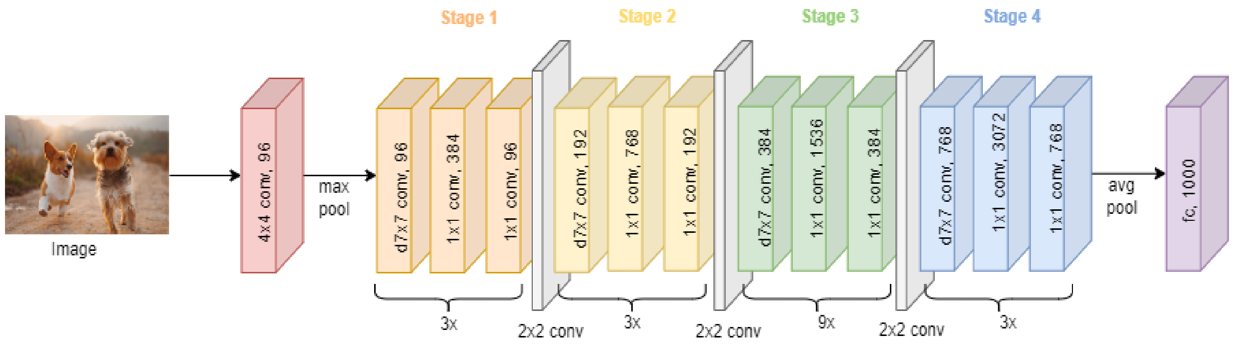


Figure 4.3. ConvNeXt architecture

4.3.3. DenseNet169

DenseNet-169 is a variant of the DenseNet architecture proposed by Huang et al. in 2017. It is part of a family of CNN architectures designed to address the challenges of gradient propagation and parameter efficiency.

DenseNet architectures introduce the concept of dense connections, where each layer is connected to every other layer in a feed-forward manner. This dense connectivity enables direct information flow between layers, promoting feature reuse and enhancing gradient flow throughout the network.

Here are some key features of the DenseNet architectures:

Dense Blocks: DenseNet consists of multiple dense blocks, each composed of multiple layers. Within a dense block, each layer is connected to all preceding layers, both within the same dense block and across different dense blocks. This dense connectivity leads to rich feature representations and facilitates gradient propagation.

Bottleneck Layers: DenseNet incorporates bottleneck layers, which reduce the number of input feature maps before the 3x3 convolution is applied. The bottleneck layers help reduce computational complexity while retaining the expressive power of the network.

Transition Layers: Between dense blocks, DenseNet includes transition layers that reduce the spatial dimensions of feature maps by using 1x1 convolution and average pooling. This compression aids in reducing the number of parameters and allows for a smooth transition between different scales of feature representations.

Growth Rate: DenseNet introduces a growth rate parameter, which determines the number of feature maps produced by each layer within a dense block. The growth rate controls the capacity of the network and influences the balance between model complexity and memory usage.

Global Average Pooling and Classifier: DenseNet architectures often conclude with a global average pooling layer, which aggregates spatial information from feature maps. This is typically followed by a fully connected layer or a SoftMax classifier for the final prediction.

The choice between DenseNet-169, DenseNet-121, or DenseNet-201 depends on various factors, including the specific task, available computational resources, and the trade-off between model complexity and performance. For our project, after experimenting with different DenseNet variants on our dataset and evaluating the performance of each, we used DenseNet-169.

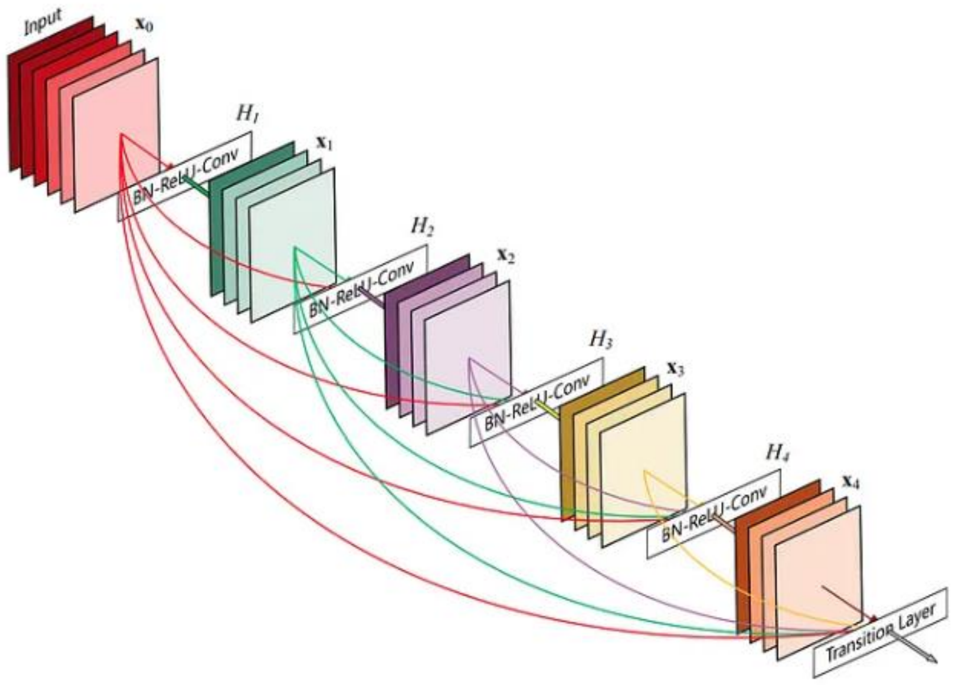


Figure 4.4. DenseNet architecture

4.3.4. EfficientNet

EfficientNet is a family of CNN architectures introduced by Tan et al. in 2019. It aims to achieve state-of-the-art performance while being computationally efficient and parameter efficient. It stands out for its innovative approach to scaling the model dimensions, including depth, width, and resolution, in a principled and balanced manner. It utilizes a compound scaling method to achieve better performance by carefully scaling these dimensions.

EfficientNet achieves its efficiency and accuracy by combining several architectural innovations, including:

Using compound scaling: EfficientNet scales the model dimensions in a balanced way by applying a compound coefficient to control the depth, width, and resolution of the network. This coefficient is derived through a neural architecture search and achieves a better trade-off between accuracy and computational efficiency.

Introducing a new building block called the mobile inverted bottleneck convolution (MBConv), which is designed to be more efficient than traditional convolutions. The MBConv block combines depthwise separable convolutions, which reduce computational cost, with a bottleneck structure to capture more complex features.

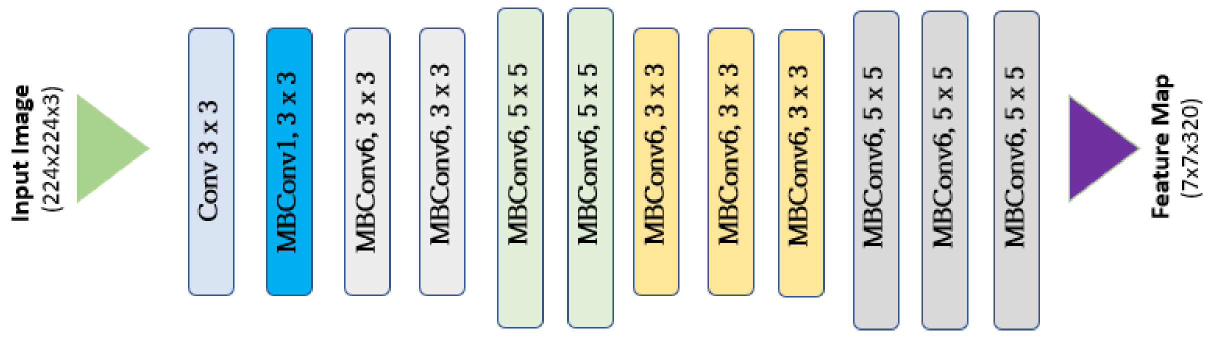


Figure 4.5. EfficientNet architecture

CHAPTER 5 – CODE DESCRIPTION AND EXPLANATION

5.1. Platform Used

We used Google Colab to write the code of our project. Google Colab is a free cloud-based platform that provides a Jupyter notebook environment for running and sharing code. It offers powerful GPU acceleration, allowing for faster execution of deep learning models. Its accessibility, convenience, and collaborative features make it an excellent choice for experimentation, learning, and collaborative coding projects.

5.2. Libraries Used

We have used the following python libraries in our project:

5.2.1. TensorFlow

An open-source library for numerical computation and machine learning that provides a flexible and efficient framework for building and training various types of deep learning models. With its extensive ecosystem and powerful features, TensorFlow has become one of the most widely used libraries for developing deep learning solutions.

5.2.2. Keras

A high-level neural networks API that runs on top of TensorFlow, providing a user-friendly interface for building and training deep learning models, including CNNs.

5.2.3. OpenCV

A computer vision library that provides tools for image and video processing, including functions for pre-processing images before feeding them into CNN models.

5.2.4. scikit-learn

A comprehensive machine learning library that includes various tools for classification, regression, clustering, and more, including utilities for CNN evaluation and preprocessing. We used scikit-learn in our project for its evaluation metrics.

5.2.5. Matplotlib.pyplot

Matplotlib.pyplot is a module within the Matplotlib library that provides a high-level interface for creating static, animated, and interactive visualizations in Python, making it a popular choice for data plotting and exploration tasks.

5.2.6. NumPy

A powerful library for numerical computing in Python, offering multi-dimensional arrays, mathematical operations, linear algebra, and more.

5.2.7. OS

OS provides a way to interact with the operating system, allowing you to perform various operations like file and directory manipulation.

5.2.8. Shutil

Shutil offers high-level file and directory operations, providing functions to copy, move, and delete files or entire directory trees.

5.2.9. Zipfile

A zip file allows you to create, read, write, and extract files from ZIP archives, providing functionalities for compression and decompression.

5.2.10. PIL

PIL (Python Imaging Library) is a library for opening, manipulating, and saving many different image file formats, widely used for image processing tasks.

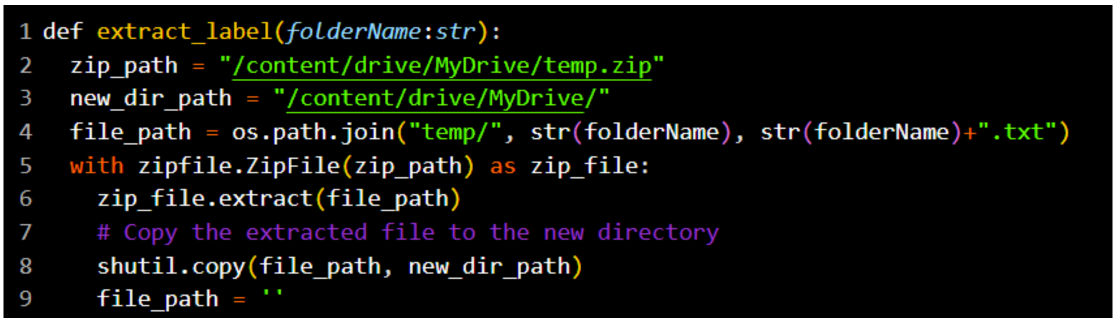
5.3. Code Explanation

Code is divided into multiple functions and independent blocks using Jupyter Notebook format on Google Colab. The aim is to make the code easier to understand, debug and scale according to our needs without compromising the efficiency.

Note that the dataset we used in **BS-80K**, and is publicly available in a **temp.zip** file. The required contents of this file are directly extracted and used with our code. Since our objective is detection of lung cancer, CNN models are only trained for bone scans of the chest i.e. **chestLANT, chestLPOST, chestRANT** and **chestRPOST**.

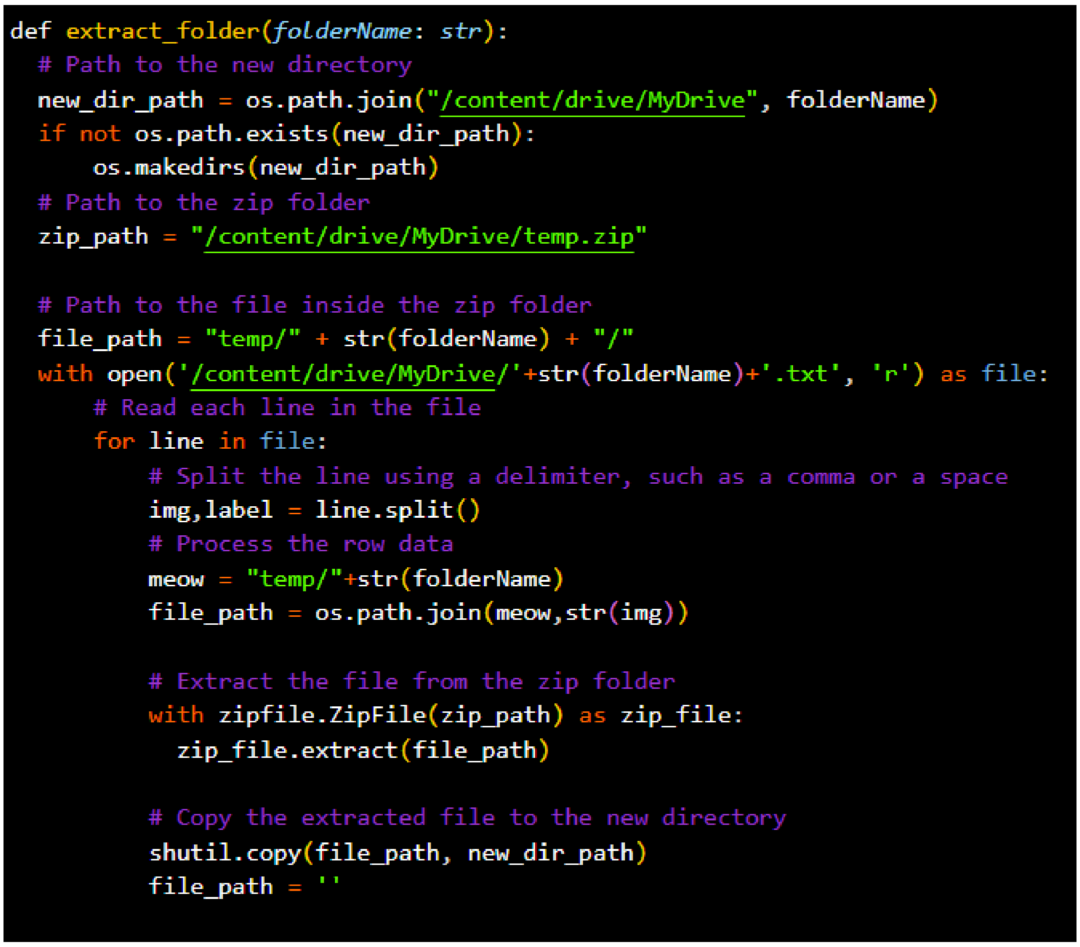
5.3.1. Extract Label Function

The function provided allows for the extraction of the label file for a specific sub-dataset. To use the function, the name of the sub-dataset must be specified as an argument. The label file, which contains image names and their corresponding labels, is in text format. Once extracted, the label file is saved within our drive.



```
def extract_label(folderName:str):
  zip_path = "/content/drive/MyDrive/temp.zip"
  new_dir_path = "/content/drive/MyDrive/"
  file_path = os.path.join("temp/", str(folderName), str(folderName)+".txt")
  with zipfile.ZipFile(zip_path) as zip_file:
    zip_file.extract(file_path)
    # Copy the extracted file to the new directory
    shutil.copy(file_path, new_dir_path)
    file_path = ''
```

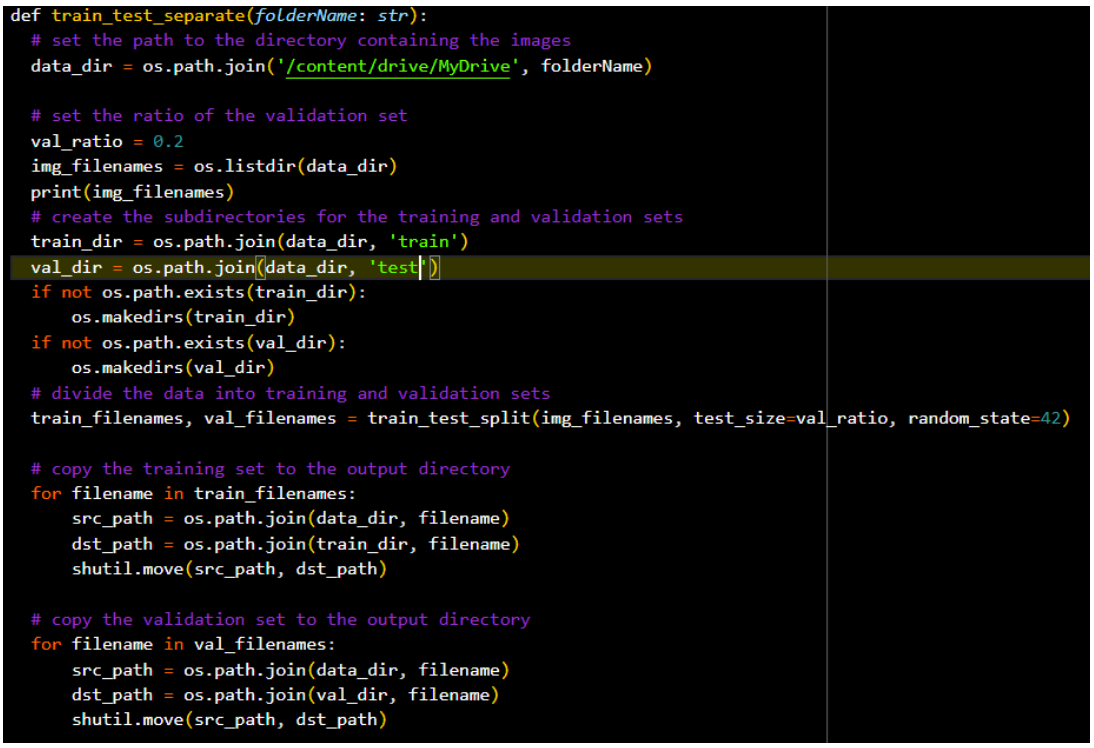
5.3.2. Extract Folder Function



```
def extract_folder(folderName: str):
  # Path to the new directory
  new_dir_path = os.path.join("/content/drive/MyDrive", folderName)
  if not os.path.exists(new_dir_path):
      os.makedirs(new_dir_path)
  # Path to the zip folder
  zip_path = "/content/drive/MyDrive/temp.zip"

  # Path to the file inside the zip folder
  file_path = "temp/" + str(folderName) + "/"
  with open('/content/drive/MyDrive/'+str(folderName)+'.txt', 'r') as file:
      # Read each line in the file
      for line in file:
          # Split the line using a delimiter, such as a comma or a space
          img,label = line.split()
          # Process the row data
          meow = "temp/"+str(folderName)
          file_path = os.path.join(meow,str(img))

          # Extract the file from the zip folder
          with zipfile.ZipFile(zip_path) as zip_file:
            zip_file.extract(file_path)

          # Copy the extracted file to the new directory
          shutil.copy(file_path, new_dir_path)
          file_path = ''
```

This function extracts the whole sub-folder from the **temp.zip** file. The name of the sub-folder needs to be added into the argument of the function when calling it. Since the zip file has many sub-folders, this function needs some time to execute completely.

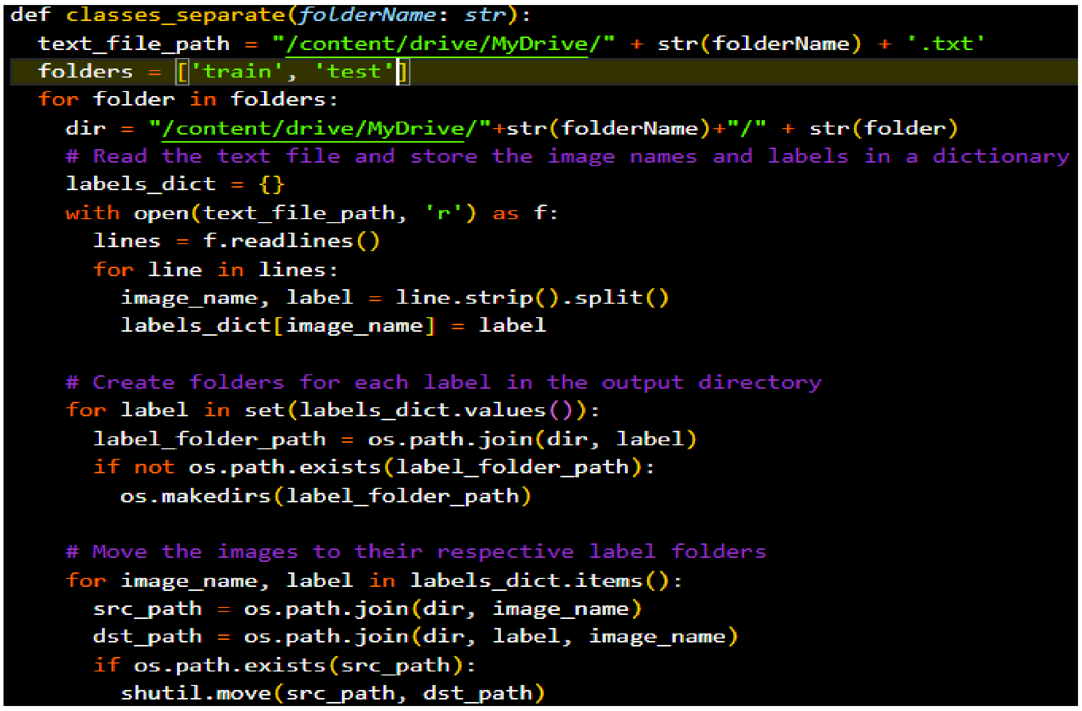
5.3.3. Train Test Separation Function



```python
def train_test_separate(folderName: str):
  # set the path to the directory containing the images
  data_dir = os.path.join('/content/drive/MyDrive', folderName)

  # set the ratio of the validation set
  val_ratio = 0.2
  img_filenames = os.listdir(data_dir)
  print(img_filenames)
  # create the subdirectories for the training and validation sets
  train_dir = os.path.join(data_dir, 'train')
  val_dir = os.path.join(data_dir, 'test')
  if not os.path.exists(train_dir):
      os.makedirs(train_dir)
  if not os.path.exists(val_dir):
      os.makedirs(val_dir)
  # divide the data into training and validation sets
  train_filenames, val_filenames = train_test_split(img_filenames, test_size=val_ratio, random_state=42)

  # copy the training set to the output directory
  for filename in train_filenames:
      src_path = os.path.join(data_dir, filename)
      dst_path = os.path.join(train_dir, filename)
      shutil.move(src_path, dst_path)

  # copy the validation set to the output directory
  for filename in val_filenames:
      src_path = os.path.join(data_dir, filename)
      dst_path = os.path.join(val_dir, filename)
      shutil.move(src_path, dst_path)
```

The folders extracted must have sub-folders of train and test. The ratio used to separate them randomly is 0.2, which infers that 80% of the data is allotted to the train folder and 20% is allotted to the test folder. **Train_test_split** function is used within this function to serve this purpose. The images selected are then moved to their folders.

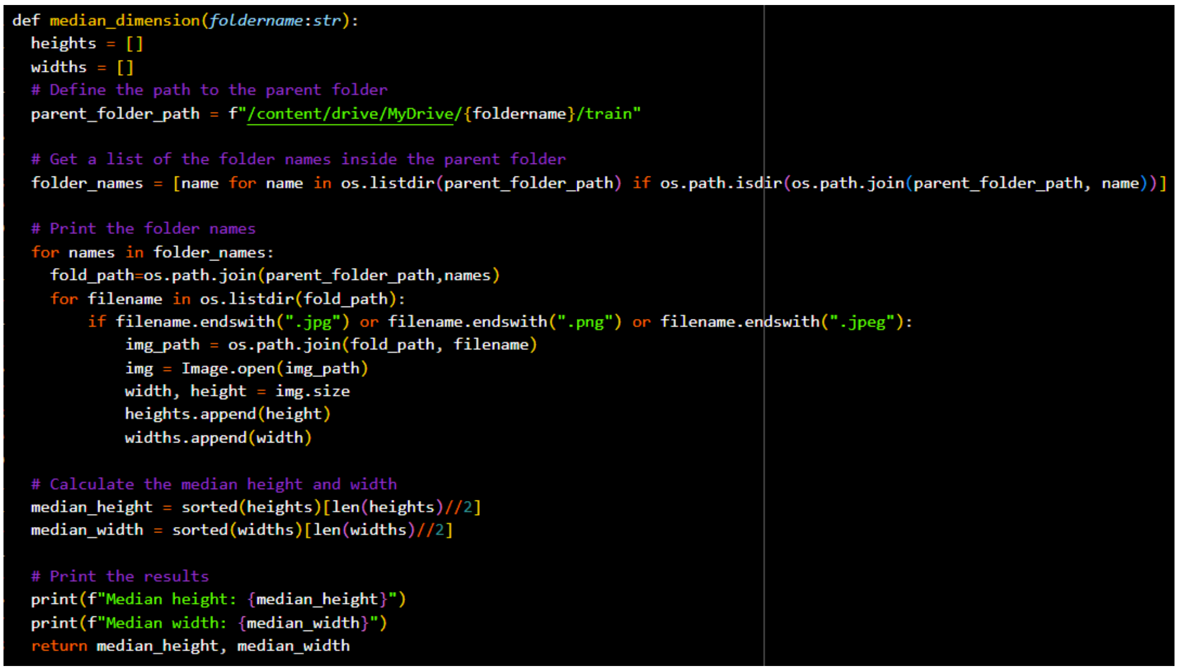
5.3.4. Classes Separation Folder



```python
def classes_separate(folderName: str):
  text_file_path = "/content/drive/MyDrive/" + str(folderName) + '.txt'
  folders = ['train', 'test']
  for folder in folders:
    dir = "/content/drive/MyDrive/"+str(folderName)+"/" + str(folder)
    # Read the text file and store the image names and labels in a dictionary
    labels_dict = {}
    with open(text_file_path, 'r') as f:
      lines = f.readlines()
      for line in lines:
        image_name, label = line.strip().split()
        labels_dict[image_name] = label

    # Create folders for each label in the output directory
    for label in set(labels_dict.values()):
      label_folder_path = os.path.join(dir, label)
      if not os.path.exists(label_folder_path):
        os.makedirs(label_folder_path)

    # Move the images to their respective label folders
    for image_name, label in labels_dict.items():
      src_path = os.path.join(dir, image_name)
      dst_path = os.path.join(dir, label, image_name)
      if os.path.exists(src_path):
        shutil.move(src_path, dst_path)
```

To separate the images according to their labels, this function is developed. This gives us more control over our data for manual changes and augmentation. Two subfolders in the train and test folders are created with names **0** and **1**. **1** is for cancerous and **0** is for non-cancerous.

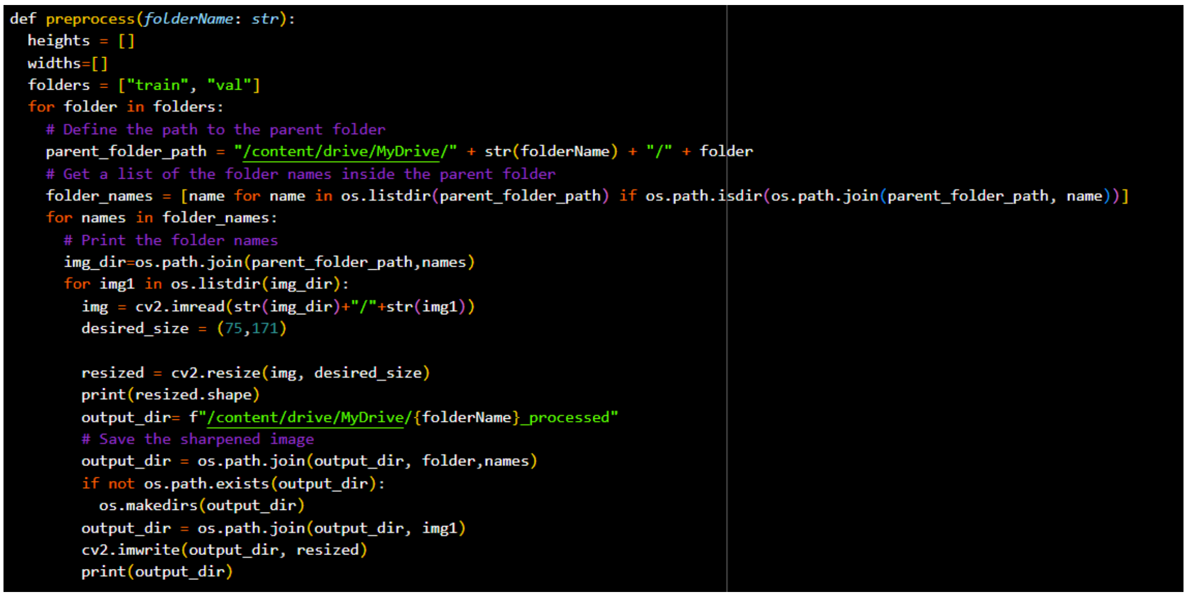
5.3.5. Median Height Calculation Function



```python
def median_dimension(foldername:str):
  heights = []
  widths = []
  # Define the path to the parent folder
  parent_folder_path = f"/content/drive/MyDrive/{foldername}/train"

  # Get a list of the folder names inside the parent folder
  folder_names = [name for name in os.listdir(parent_folder_path) if os.path.isdir(os.path.join(parent_folder_path, name))]

  # Print the folder names
  for names in folder_names:
    fold_path=os.path.join(parent_folder_path,names)
    for filename in os.listdir(fold_path):
        if filename.endswith(".jpg") or filename.endswith(".png") or filename.endswith(".jpeg"):
            img_path = os.path.join(fold_path, filename)
            img = Image.open(img_path)
            width, height = img.size
            heights.append(height)
            widths.append(width)

  # Calculate the median height and width
  median_height = sorted(heights)[len(heights)//2]
  median_width = sorted(widths)[len(widths)//2]

  # Print the results
  print(f"Median height: {median_height}")
  print(f"Median width: {median_width}")
  return median_height, median_width
```

This function calculates the median dimensions of the images inside the train folder of a sub-dataset. As the train folder contains 80% of the entire dataset, computing the median length from this subset provides a reliable approximation. Subsequently, all the images are resized to match the calculated median length.

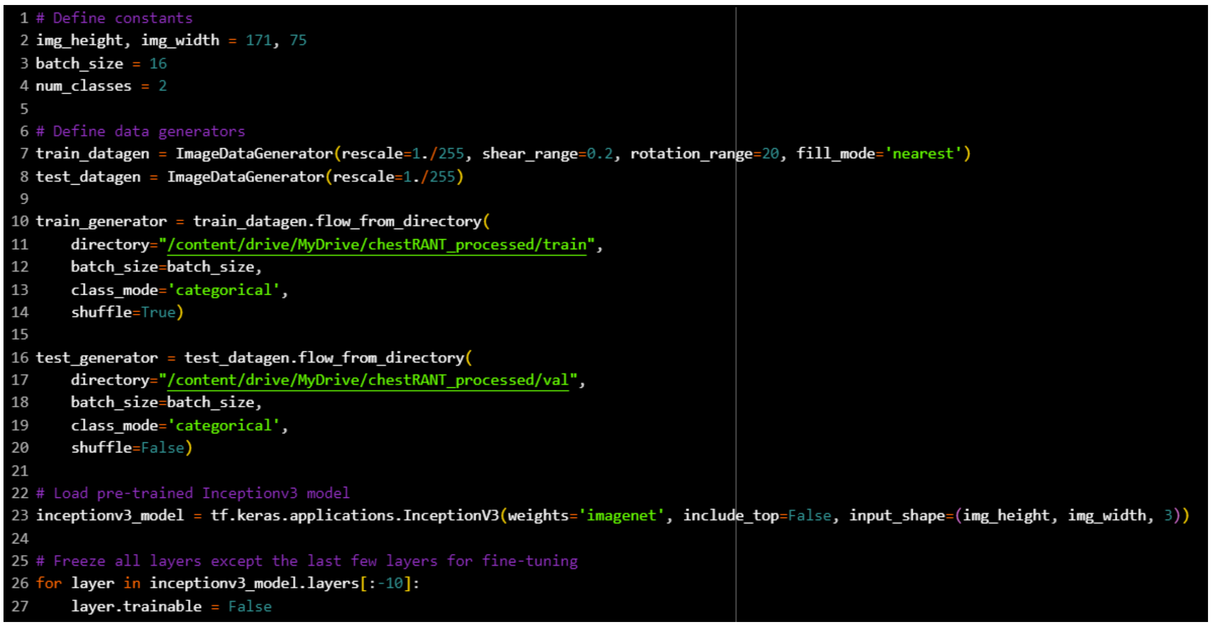
5.3.6. Preprocessing Function



```python
def preprocess(folderName: str):
  heights = []
  widths=[]
  folders = ["train", "val"]
  for folder in folders:
    # Define the path to the parent folder
    parent_folder_path = "/content/drive/MyDrive/" + str(folderName) + "/" + folder
    # Get a list of the folder names inside the parent folder
    folder_names = [name for name in os.listdir(parent_folder_path) if os.path.isdir(os.path.join(parent_folder_path, name))]
    for names in folder_names:
      # Print the folder names
      img_dir=os.path.join(parent_folder_path,names)
      for img1 in os.listdir(img_dir):
        img = cv2.imread(str(img_dir)+"/"+str(img1))
        desired_size = (75,171)

        resized = cv2.resize(img, desired_size)
        print(resized.shape)
        output_dir= f"/content/drive/MyDrive/{folderName}_processed"
        # Save the sharpened image
        output_dir = os.path.join(output_dir, folder,names)
        if not os.path.exists(output_dir):
          os.makedirs(output_dir)
        output_dir = os.path.join(output_dir, img1)
        cv2.imwrite(output_dir, resized)
        print(output_dir)
```

In this function, the images within a subfolder are resized based on the previously computed median dimensions. Following the resizing process, these images undergo a series of enhancements. First, they are subjected to a median filter to reduce noise and improve overall smoothness. Next, a Laplacian filter is applied to further enhance the images, emphasizing fine details and sharpening them. These refined and sharpened images are then utilized to enhance the training of our model, leading to improved performance.

5.3.7. Data Augmentation and Making the CNN Model



```python
# Define constants
img_height, img_width = 171, 75
batch_size = 16
num_classes = 2

# Define data generators
train_datagen = ImageDataGenerator(rescale=1./255, shear_range=0.2, rotation_range=20, fill_mode='nearest')
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    directory="/content/drive/MyDrive/chestRANT_processed/train",
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=True)

test_generator = test_datagen.flow_from_directory(
    directory="/content/drive/MyDrive/chestRANT_processed/val",
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=False)

# Load pre-trained Inceptionv3 model
inceptionv3_model = tf.keras.applications.InceptionV3(weights='imagenet', include_top=False, input_shape=(img_height, img_width, 3))

# Freeze all layers except the last few layers for fine-tuning
for layer in inceptionv3_model.layers[:-10]:
    layer.trainable = False
```

ImageDataGenerator API is used to augment the training dataset with a shear_range, rotation_range, and fill_model. Data augmentation is performed to make a model more generalized and prevent it from overfitting. Additionally, data augmentation plays a significant role in reducing data bias. In this project, where the ratio of cancerous images to non-cancerous images in the dataset is approximately 1:10, data augmentation techniques are used to augment the number of cancerous samples, effectively increasing their representation in the dataset.

Four different models are imported: **Inception-v3, ConvNext, EfficientNet and DenseNet**. Each model is used to predict a different sub-dataset. These specific architectures are known to perform well with data that exhibits similarities to the dataset being utilized. The objective is to evaluate each architecture on each sub-dataset and analyze their performance. This analysis will enable the selection of the model that demonstrates the most favorable performance characteristics for each specific sub-dataset.

To provide flexibility in handling variable image sizes, the include_top argument is set to false, allowing the model to accommodate different image dimensions. In this case, the images being fed to the model have dimensions of 175x75x3. Since the dataset comprises only two classes (cancerous and non-cancerous), the num_classes parameter is set to 2.
The batch_size, an essential hyperparameter, is adjusted through experimentation with the model. Initially, a batch size of 32 is used, but it lead to overfitting. Consequently, the batch size is reduced to 16 to alleviate the

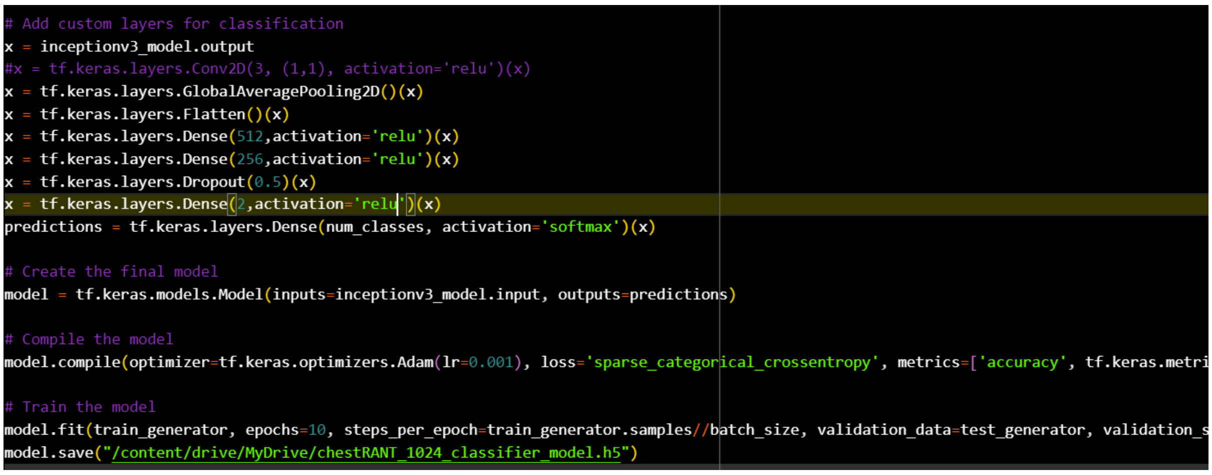
overfitting issue and improve the model's performance.

The smaller batches have more noise, and the images are likely to be different from each other, which essentially adds a regularization effect on the model. Furthermore, using smaller batch sizes means that the weights and biases of the model are updated more frequently. After each batch is processed, the gradients are calculated, and the weights and biases are adjusted accordingly to minimize the loss. This frequent updating of parameters allows the model to converge faster and potentially achieve a better convergence point, leading to improved performance.

Data normalization is also done using the ImageDataGenerator API. It is done on both train and test images because both should be passed through the same pre-processing techniques for correct model prediction. The goal is to make our cost function symmetric for quicker and better convergence at the minimum point.

Transfer learning is used, only the last 10 layers of each CNN model are trained by us while the weights and biases of layers beyond them are simply imported from tensor flow. During training, the last 10 layers are fine-tuned on the bone-scan dataset and their weights and biases are accordingly set. This technique is very useful to save time in training large CNN models. The lower layers, which only detect low-level features like edges, are not very important so these layers can be trained on any big dataset. The final layers detect high-level features which are important and hence these layers are trained on the actual dataset. Transfer learning is known to give good results, save time and also prevent the model from overfitting due to the less amount of data.

5.3.8 Adding Custom Layers and Training the Model



```
# Add custom layers for classification
x = inceptionv3_model.output
#x = tf.keras.layers.Conv2D(3, (1,1), activation='relu')(x)
x = tf.keras.layers.GlobalAveragePooling2D()(x)
x = tf.keras.layers.Flatten()(x)
x = tf.keras.layers.Dense(512,activation='relu')(x)
x = tf.keras.layers.Dense(256,activation='relu')(x)
x = tf.keras.layers.Dropout(0.5)(x)
x = tf.keras.layers.Dense(2,activation='relu')(x)
predictions = tf.keras.layers.Dense(num_classes, activation='softmax')(x)

# Create the final model
model = tf.keras.models.Model(inputs=inceptionv3_model.input, outputs=predictions)

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(lr=0.001), loss='sparse_categorical_crossentropy', metrics=['accuracy', tf.keras.metri

# Train the model
model.fit(train_generator, epochs=10, steps_per_epoch=train_generator.samples//batch_size, validation_data=test_generator, validation_s
model.save("/content/drive/MyDrive/chestRANT_1024_classifier_model.h5")
```

Custom layers are connected to the model after convolutional/max-pooling blocks. The first layer is the Global Average Pooling layer which is used to reduce the tensor size. It can be considered as a regularization technique because it reduced the number of features that are to be fed into dense layers for prediction. The reduced tensor size is then flattened into a single column by the flattening layer. Then dense layers with 512 and 256 neurons are added to study the feature maps inside tensors.

To tackle overfitting and reduce model complexity, dropout regularization is implemented. A dropout probability of 50% is used, indicating that half of the neurons in a 256-neuron layer are randomly deactivated during training. When neurons are dropped, the remaining neurons tend to distribute their weights to adjacent neurons, leading to an overall reduction in the magnitude of weights. Dropout is a common technique employed for computer vision applications and prevents the model to get too computationally intensive and complex. Although it helps to address some issues, it makes the calculation of the cost function difficult and makes it less defined due to which the training time can increase.

The learning rate, an essential hyperparameter, is adjusted based on experimentation with the model. Initially set at 0.0001, the model encountered challenges in converging to the optimal minima and tended to get trapped in a local minimum. Consequently, it experienced high loss and did not yield significant improvements during training after each epoch. To address this, the learning rate was increased to 0.001. This adjustment facilitated the model's escape from the local minimum and enabled convergence at a better minimum with reduced loss.

In this scenario, the loss function chosen is sparse_categorical_crossentropy, despite the problem being binary in nature. Typically, for binary classification problems, binary_crossentropy (log loss) is commonly used. However, due to the restriction imposed by the FPGA model, which requires the number of output neurons to match the number of classes, the problem had to be transformed into a multi-class problem. As a result, the sparse_categorical_crossentropy loss function is utilized.Another consideration is the label format, which is not in one-hot encoded vector form. Consequently, the commonly used multi-class loss function, categorical_crossentropy, could not be employed in this case.
Regarding the optimizer, Adam is selected as it is known to offer advantages over stochastic gradient descent (SGD) and simple gradient descent algorithms. Adam combines elements of both adaptive learning rates and momentum methods, enabling faster convergence and better optimization performance.

5.3.9. Activation Functions

In all models, the Rectified Linear Unit (ReLU) activation function is utilized for the hidden layers, while the SoftMax activation function is employed for the output layer. Activation functions are crucial as they introduce non-linearity, enabling our models to learn complex functions. Without activation functions, our models would resemble simple linear regression models, limiting their learning capacity. The sigmoid function is not used because it has a vanishing gradient and saturation problem if the output of the node is either lesser than -3 or greater than 3. This ceases the learning of the model and makes training useless after the first few epochs.

Tanh function, though being zero-centered, makes the learning easier by making the loss function symmetric however it also suffers from vanishing gradient and saturation when the output of nodes falls below -3 or exceeds 3.

ReLU function which is a relatively advanced function, does not face vanishing gradient and saturation problems since it is linear when input is greater than 0. It is more efficient than both sigmoid and tanh due to which it converges faster to the global minimum. Though, ReLU is not suitable for negative values because the gradient for negative values is zero which means that back propagation will play no role in updating the weight values. This problem can be addressed by leaky ReLU but to comply with our neural network on FPGA which had to be an exact replication of the software neural network, leaky ReLU is not used.

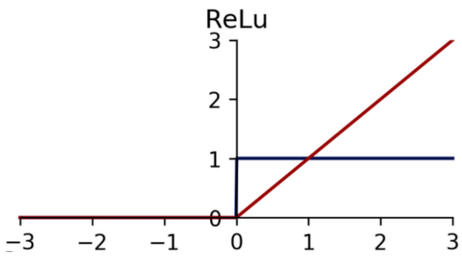


Fig 5.1. ReLU Activation Function and its Derivative

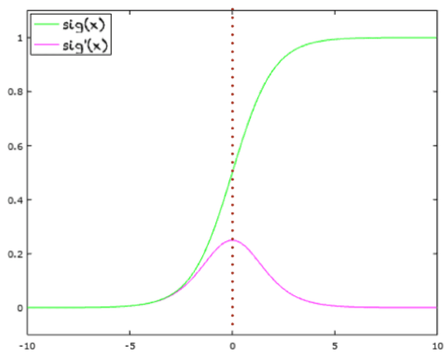


Fig 5.2. Sigmoid Activation Function and its Derivative

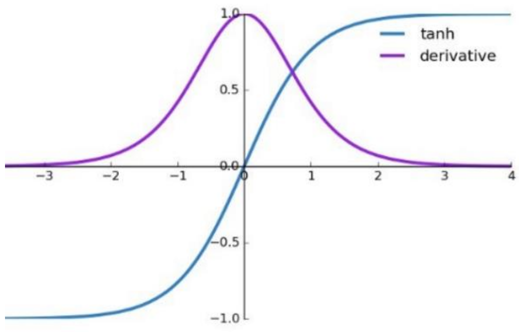
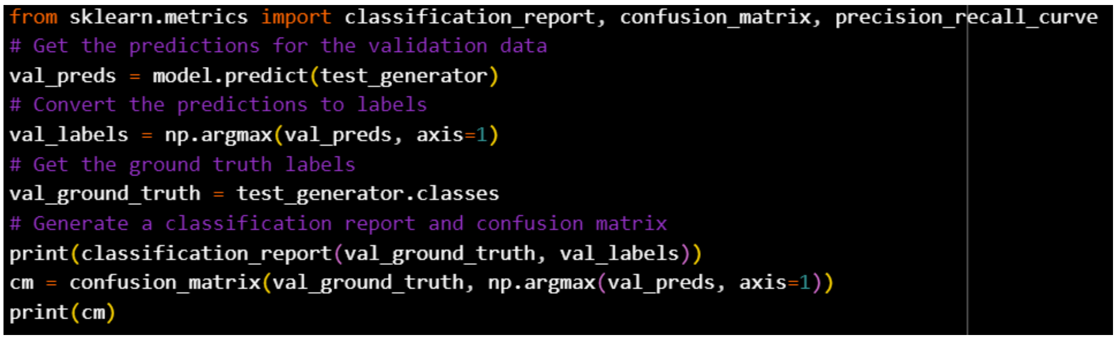


Fig 5.3. Tanh Activation Function and its Derivative

5.3.10. Performance Metrics


```
from sklearn.metrics import classification_report, confusion_matrix, precision_recall_curve
# Get the predictions for the validation data
val_preds = model.predict(test_generator)
# Convert the predictions to labels
val_labels = np.argmax(val_preds, axis=1)
# Get the ground truth labels
val_ground_truth = test_generator.classes
# Generate a classification report and confusion matrix
print(classification_report(val_ground_truth, val_labels))
cm = confusion_matrix(val_ground_truth, np.argmax(val_preds, axis=1))
print(cm)
```

In this piece of code, we used performance metrics to evaluate our model including: accuracy, precision, recall and F1-score. Since the dataset is biased, F1-score is the ideal metric for evaluation. F1-score is calculated at the end on our **test data** using scikit-learn.

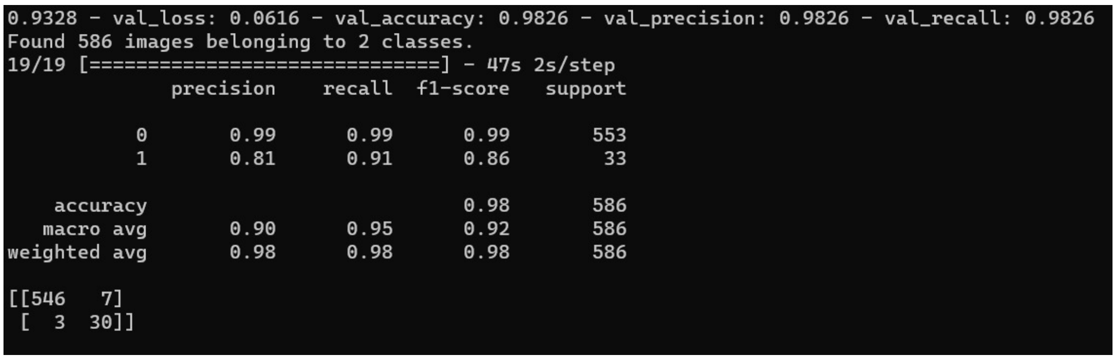
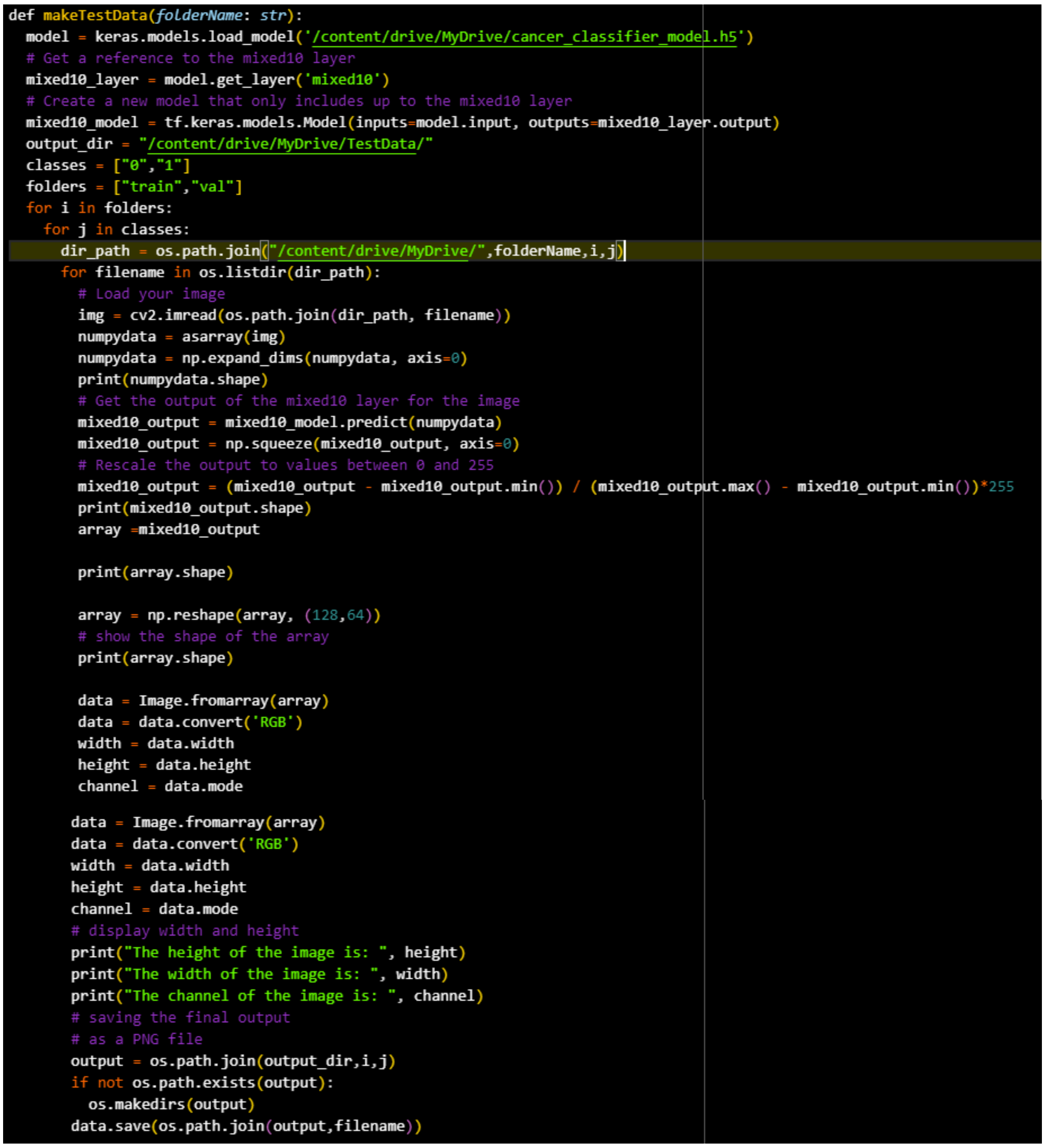

```
0.9328 - val_loss: 0.0616 - val_accuracy: 0.9826 - val_precision: 0.9826 - val_recall: 0.9826
Found 586 images belonging to 2 classes.
19/19 [==============================] - 47s 2s/step
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       553
           1       0.81      0.91      0.86        33

    accuracy                           0.98       586
   macro avg       0.90      0.95      0.92       586
weighted avg       0.98      0.98      0.98       586

[[546   7]
 [  3  30]]
```

Fig 5.4. Classification Report Example

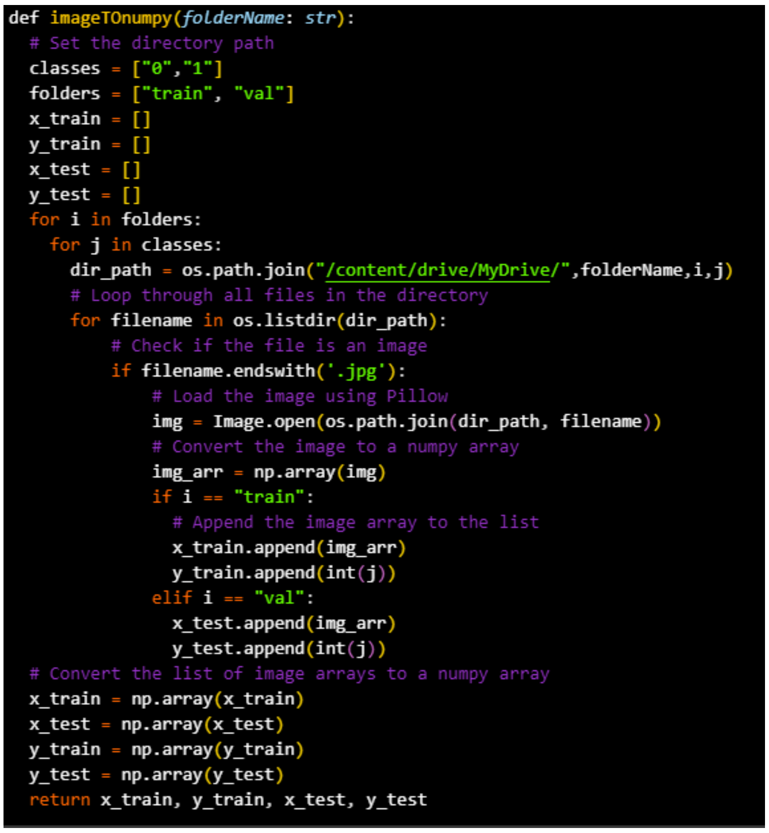
5.3.11. MakeTestData Function



```
def makeTestData(folderName: str):
  model = keras.models.load_model('/content/drive/MyDrive/cancer_classifier_model.h5')
  # Get a reference to the mixed10 layer
  mixed10_layer = model.get_layer('mixed10')
  # Create a new model that only includes up to the mixed10 layer
  mixed10_model = tf.keras.models.Model(inputs=model.input, outputs=mixed10_layer.output)
  output_dir = "/content/drive/MyDrive/TestData/"
  classes = ["0","1"]
  folders = ["train","val"]
  for i in folders:
    for j in classes:
      dir_path = os.path.join("/content/drive/MyDrive/",folderName,i,j)
      for filename in os.listdir(dir_path):
        # Load your image
        img = cv2.imread(os.path.join(dir_path, filename))
        numpydata = asarray(img)
        numpydata = np.expand_dims(numpydata, axis=0)
        print(numpydata.shape)
        # Get the output of the mixed10 layer for the image
        mixed10_output = mixed10_model.predict(numpydata)
        mixed10_output = np.squeeze(mixed10_output, axis=0)
        # Rescale the output to values between 0 and 255
        mixed10_output = (mixed10_output - mixed10_output.min()) / (mixed10_output.max() - mixed10_output.min())*255
        print(mixed10_output.shape)
        array =mixed10_output

        print(array.shape)

        array = np.reshape(array, (128,64))
        # show the shape of the array
        print(array.shape)

        data = Image.fromarray(array)
        data = data.convert('RGB')
        width = data.width
        height = data.height
        channel = data.mode

        data = Image.fromarray(array)
        data = data.convert('RGB')
        width = data.width
        height = data.height
        channel = data.mode
        # display width and height
        print("The height of the image is: ", height)
        print("The width of the image is: ", width)
        print("The channel of the image is: ", channel)
        # saving the final output
        # as a PNG file
        output = os.path.join(output_dir,i,j)
        if not os.path.exists(output):
          os.makedirs(output)
        data.save(os.path.join(output,filename))
```

This crucial function marks the commencement of the neural network's transition onto the FPGA. The previously trained model is imported, and a new sub-model is created comprising only the layers above the global average pooling layer. The test data is then fed through this sub-model, generating predictions in the form of tensors. The test data is passed through the model and predictions are taken in the form of tensors. The tensors which are in 4 dimensions are de-normalized and reshaped into a single 2D plane. This 2D plane is converted into an RGB image by replicating this 2D plane twice. Now our tensor has a batch size of 1 which means that it only contains the image itself and is ready to be stored as an image file in JPEG format. The data is saved in a drive to be further processed by FPGA.

5.3.12. ImageToNumpy Function

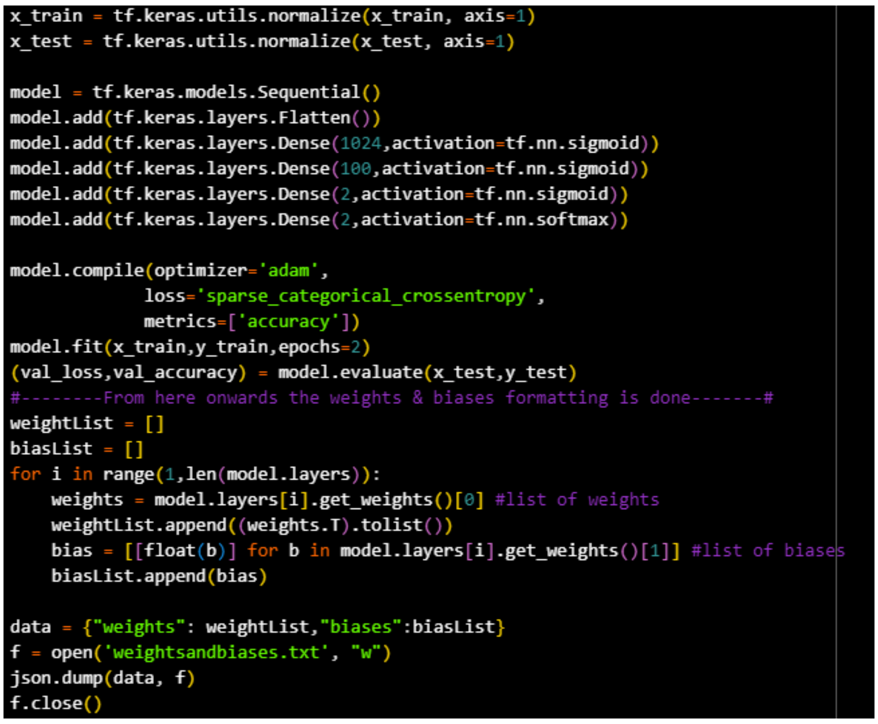


```python
def imageTOnumpy(folderName: str):
  # Set the directory path
  classes = ["0","1"]
  folders = ["train", "val"]
  x_train = []
  y_train = []
  x_test = []
  y_test = []
  for i in folders:
    for j in classes:
      dir_path = os.path.join("/content/drive/MyDrive/",folderName,i,j)
      # Loop through all files in the directory
      for filename in os.listdir(dir_path):
          # Check if the file is an image
          if filename.endswith('.jpg'):
              # Load the image using Pillow
              img = Image.open(os.path.join(dir_path, filename))
              # Convert the image to a numpy array
              img_arr = np.array(img)
              if i == "train":
                # Append the image array to the list
                x_train.append(img_arr)
                y_train.append(int(j))
              elif i == "val":
                x_test.append(img_arr)
                y_test.append(int(j))
  # Convert the list of image arrays to a numpy array
  x_train = np.array(x_train)
  x_test = np.array(x_test)
  y_train = np.array(y_train)
  y_test = np.array(y_test)
  return x_train, y_train, x_test, y_test
```

This function serves the purpose of taking the resized tensors stored in JPEG format and splitting them into four arrays: x_train, y_train, x_test, and y_test. These arrays, filled with data, will be utilized to train a new neural network that will later be replicated onto the FPGA.

It's important to note that while these may appear as simple images, they actually contain feature maps within them. Training a neural network with feature maps proves to be easier and more effective compared to training with simple images.
The necessity for retraining arises due to the resizing process, which is essential from the FPGA perspective. The FPGA model requires images directly as input, hence the feature maps are stored as images. If resizing was not performed, this retraining step would have been unnecessary. Nonetheless, the resizing step is crucial for compatibility with the FPGA model, enabling the storage of feature maps as images.

5.3.13. Training of Resized Neural Network to be Implemented on FPGA



```python
x_train = tf.keras.utils.normalize(x_train, axis=1)
x_test = tf.keras.utils.normalize(x_test, axis=1)

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(1024,activation=tf.nn.sigmoid))
model.add(tf.keras.layers.Dense(100,activation=tf.nn.sigmoid))
model.add(tf.keras.layers.Dense(2,activation=tf.nn.sigmoid))
model.add(tf.keras.layers.Dense(2,activation=tf.nn.softmax))

model.compile(optimizer='adam',
             loss='sparse_categorical_crossentropy',
             metrics=['accuracy'])
model.fit(x_train,y_train,epochs=2)
(val_loss,val_accuracy) = model.evaluate(x_test,y_test)
#--------From here onwards the weights & biases formatting is done-------#
weightList = []
biasList = []
for i in range(1,len(model.layers)):
    weights = model.layers[i].get_weights()[0] #list of weights
    weightList.append((weights.T).tolist())
    bias = [[float(b)] for b in model.layers[i].get_weights()[1]] #list of biases
    biasList.append(bias)

data = {"weights": weightList,"biases":biasList}
f = open('weightsandbiases.txt', "w")
json.dump(data, f)
f.close()
```

Initially, the inputs of both the test and training sets are normalized to ensure consistent preprocessing. Following normalization, multiple dense layers are added to the neural network, each employing the sigmoid activation function. While it has been discussed and acknowledged previously in this thesis that ReLU performs better but sigmoid is used to save up FPGA resources. By opting for sigmoid, the neural network can be replicated precisely on the FPGA while managing resource utilization effectively.

Once the model is trained, the weights and biases are stored in a text file. This text file will further be converted into multiple weight files, so that each neuron gets a separate weight file for itself. However, all neurons in a layer will have to share the same bias which is a compromise we made to save resources.

Although we are using SoftMax function for the final layer of our software code, we will be using HardMax function in the final layer of our hardware code. This is again done to save resources with some compromise on the performance of our hardware model.

CHAPTER 6 – IMPLEMENTATION ON FPGA

6.1. Introduction

Due to the increased demand for data processing, a new IT architecture is used nowadays to ease up the load on the internet and on the data centers by moving some of the storage and computer units close to the data-producing nodes. This is also known as **edge computing**. This provided a new opportunity for low-cost FPGA devices to act as edge computing neural network nodes for users who want computation power to run their ML/DL applications. In this work, we introduce DNN models on FPGA, solely for oncologists who want to detect cancer in their patients using bone Scan images. These models are compatible with low-cost hybrid FPGA platforms such as the Xilinx Zynq. Based on a software-hardware co-design approach, this project supports pre-trained networks. Simulation results show that the performance of DNNs implemented on the FPGA is very close to their software implementations in terms of performance metrics such as accuracy, precision, recall, and F1-score. At the same time, it gave better processing performance than CPUs and GPUs with a lower energy footprint because their concurrent computation model is neural network friendly.

6.2 The Architecture of Neuron:

The architecture of neurons consists of multiple registers, LUTs and designated processing units. The depth of the weight memory is proportional to the number of inputs to the neuron. When it comes to images, each pixel acts as an input to a neuron, so the bigger the image size, the larger the weight of memory we will need. For this purpose, the output tensor sizes are reduced to limit the model complexity. The weight, bias, and activation memories will be like a ROM memory because they must not be changed when the optimal point is reached during the training of our models. The neurons will use MAC units to perform the necessary mathematical operations and each operation will have a delay, denoted as **d** in our Verilog code. All inputs are first multiplied with their corresponding weights and added together using an adder block. Finally, at the last step, a bias is added to the whole equation. The equation will be a linear equation with multiple inputs, to add non-linearity in our model for better prediction and generalization, a non-linear activation function is used.

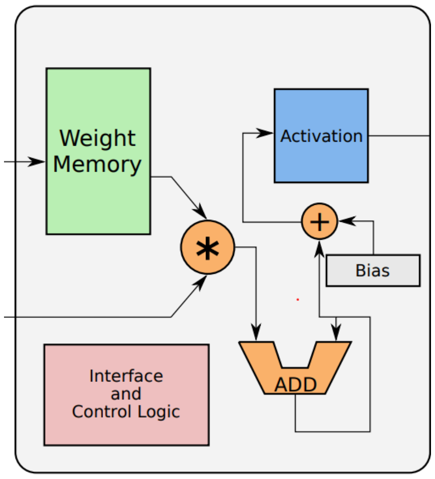
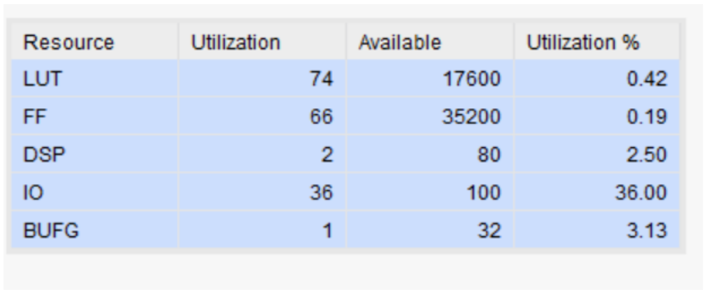


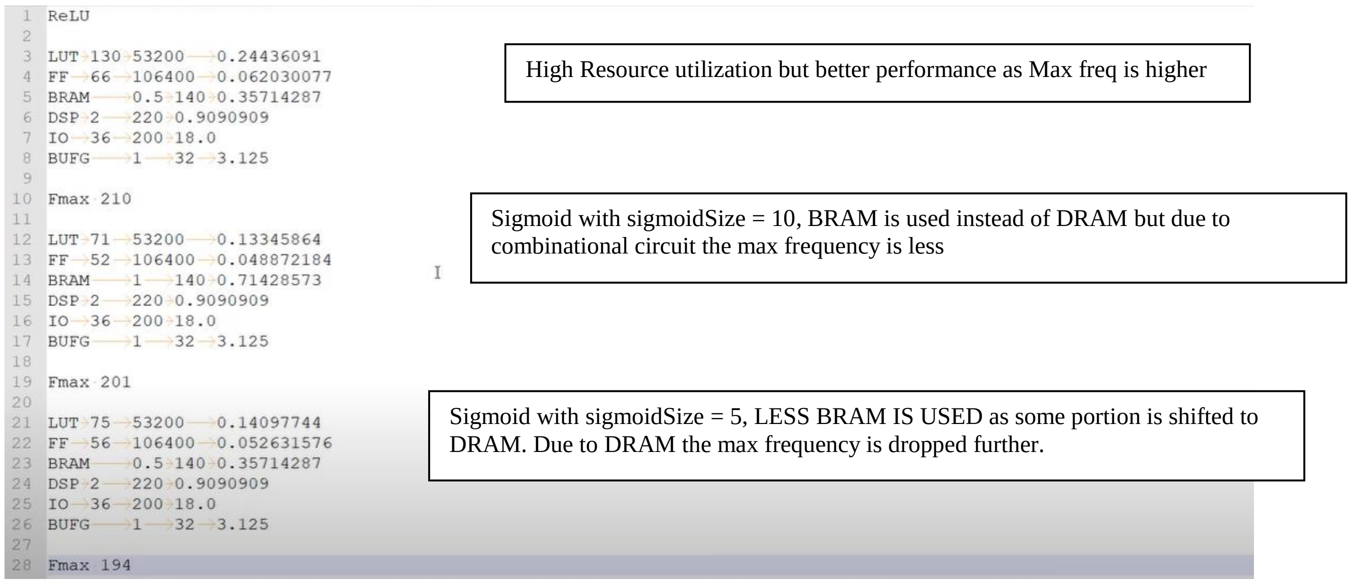
Fig 6.1 Architecture of a Neuron



Resource	Utilization	Available	Utilization %
LUT	74	17600	0.42
FF	66	35200	0.19
DSP	2	80	2.50
IO	36	100	36.00
BUFG	1	32	3.13

Fig 6.2 Resource Utilization table of a neuron with data width of 16

We have developed two activation functions in our model: Sigmoid and ReLU. The Sigmoid function is developed using Look-Up Tables (LUTs), with the depth of the LUTs being proportional to the number of inputs to our neurons. On the other hand, the ReLU function is developed using dedicated circuitry.
The choice of activation function has a significant impact on the performance of our model in terms of speed, latency, and accuracy. While the sigmoid function developed using LUTs is not as accurate as ReLU, it consumes fewer FPGA resources compared to ReLU. However, ReLU provides better prediction results and its zero-mean nature contributes to faster and more accurate predictions. To strike a balance between resource utilization and prediction performance, we decided to use the sigmoid function, as saving FPGA resources allows us to have more neurons and deeper neural networks, which are crucial for handling the complex task of cancer detection.
In terms of memory utilization, the smallest available Block RAM (BRAM) is 36kbits, and we aim to utilize at least half of it. To optimize our BRAM resources, we have implemented the activation memory using Dynamic Random-Access Memory (DRAM), as the required memory size is less than 18kbits. This decision helps us save the crucial BRAM resources for other parts of our model.



```
ReLU

LUT  130  53200   0.24436091
FF   66   106400  0.062030077
BRAM      0.5 140 0.35714287
DSP  2    220 0.9090909
IO   36   200 18.0
BUFG      1    32  3.125

Fmax 210

LUT  71   53200   0.13345864
FF   52   106400  0.048872184
BRAM      1    140 0.71428573
DSP  2    220 0.9090909
IO   36   200 18.0
BUFG      1    32  3.125

Fmax 201

LUT  75   53200   0.14097744
FF   56   106400  0.052631576
BRAM      0.5 140 0.35714287
DSP  2    220 0.9090909
IO   36   200 18.0
BUFG      1    32  3.125

Fmax 194
```

High Resource utilization but better performance as Max freq is higher

Sigmoid with sigmoidSize = 10, BRAM is used instead of DRAM but due to combinational circuit the max frequency is less

Sigmoid with sigmoidSize = 5, LESS BRAM IS USED as some portion is shifted to DRAM. Due to DRAM the max frequency is dropped further.

Fig 6.3 Performance of Activation Functions

6.3 Layers:

Our DNN consists of multiple layers that have the ability to learn complex non-linear functions. The first layer in our network is a flatten layer, responsible for converting the input, which is in the form of an image, into a single column of pixel values. This particular operation does not involve any weights, biases, or mathematical computations. However, it plays a crucial role in preparing the data for further processing in the subsequent dense layers of the network.

The output of the neurons is stored in a bus and when the bus is full which is indicated by the **ol_valid**, it is unloaded into a big register called **holdData**. This acts as the input register for the next layer of neurons, this register is unloaded to an output layer by shifting one per clock cycle. This shifting is exactly equal to the data width needed for the next neuron.

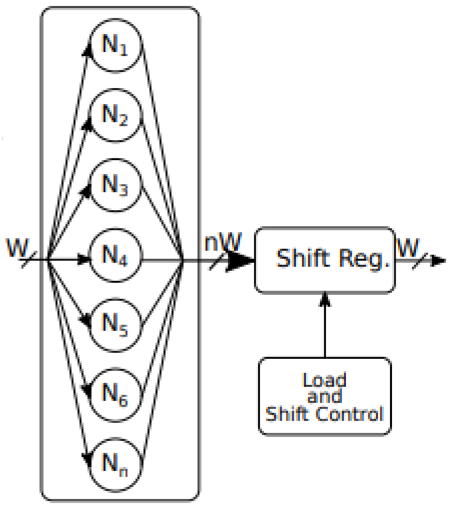
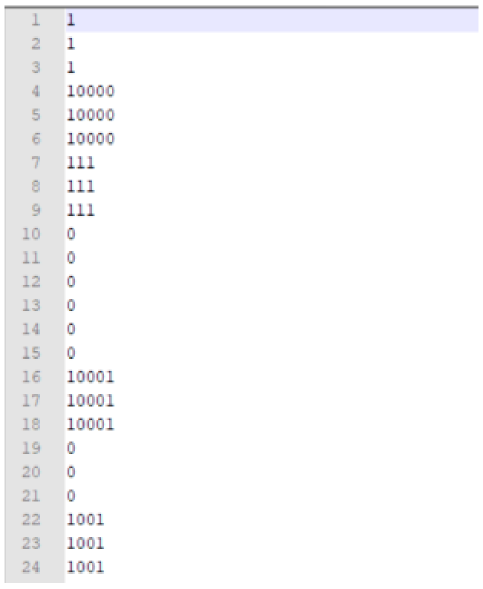


Fig 6.4 Layer Architecture

6.4 Test Data:

The test data is derived directly from the tensors of the final block of our CNN. We extract the feature map of each image from these tensors and save them as JPEG images. These images are then converted into text files, where each row represents a pixel value of the image in binary format, adhering to the appropriate data width and Qn.m format. In the Qn.m format, 'n' indicates the number of bits allocated for the integer part, while 'm' represents the number of bits allocated for the fractional part. Since the pixel values are normalized, the value of 'n' is always set to 1. The header file of the test image contains the corresponding label of the image. To facilitate data transfer, the AXI-stream is connected to the DMA Controller, while the AXI lite interface is connected to the processing system. The DMA controller enables the efficient transfer of blocks of external data with minimal intervention from the FPGA's processor.



```
1
1
1
10000
10000
10000
111
111
111
0
0
0
0
0
0
10001
10001
10001
0
0
0
1001
1001
1001
```

Fig 6.5 First 23 pixel values of a test image converted into a text file

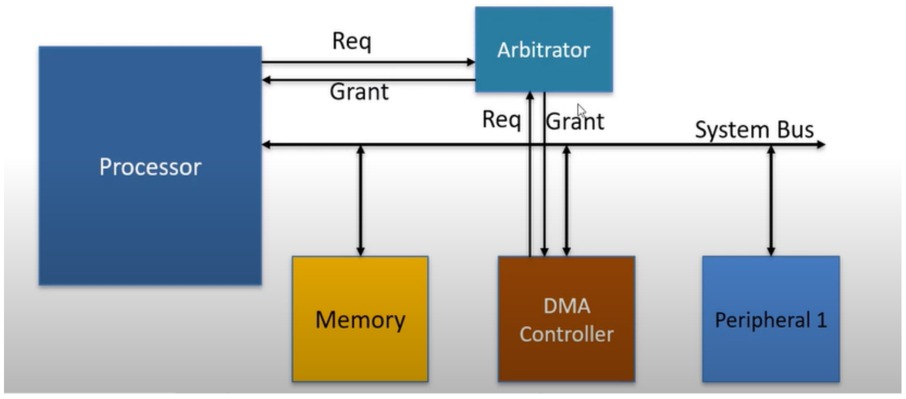


Fig 6.6 DMA controller and Processor

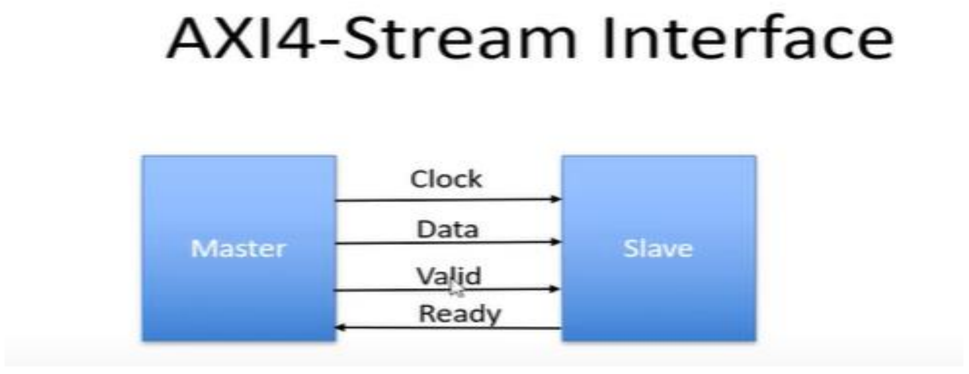


Fig 6.7 AXI4-Stream Interface

6.5 Additional Information:

Throughout the project, a fixed-point representation is used, although errors due to value estimation accumulate during mathematical operations. This can lead to some accuracy degradation, particularly when overflow occurs. However, fixed-point representation is preferred over floating-point representation due to its efficient resource utilization, especially considering the limited resources available.By carefully selecting an appropriate data width, such as 8 bits in our case, a compromise between accuracy and resource utilization can be achieved.

Xilinx IP cores are not directly utilized in the project; instead, their codes are copied and adjusted to meet our requirements. This approach provides greater flexibility and control over our model. Moreover, it introduces additional hyperparameters to fine-tune, including data width, activation function depth, and memory allocation. Utilizing a similar code ensures that our implementation remains efficient, akin to an IP core.

$$Sum[n] = (w\_out[n] * in[n]) + sum[n-1]$$

$$Sum[last] = \left(w_{out[last]} * in[last]\right) + sum[last] + bias$$

$$W\_out[n] = weightValue[n-1]\,; \qquad in[n] = myinput[n-1]$$

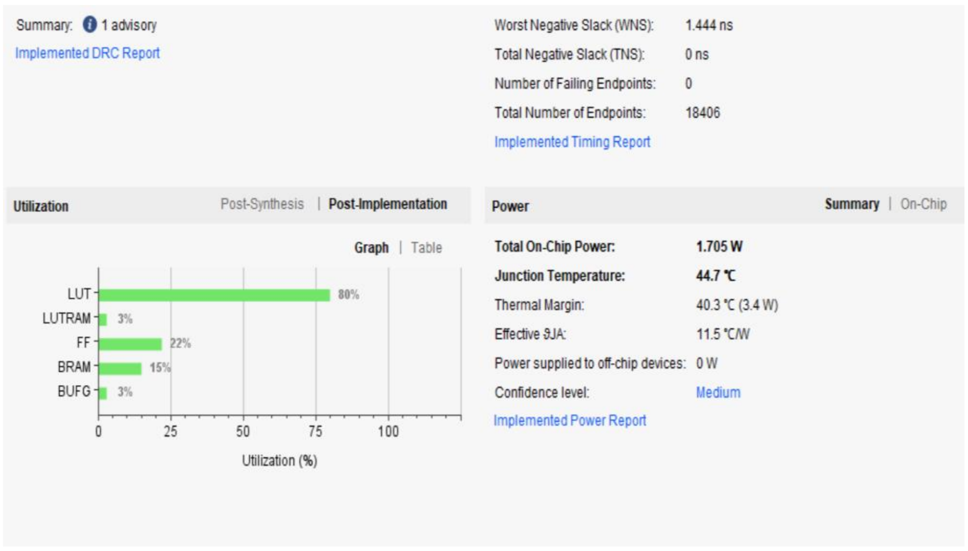
Fig 6.8 Equations of a Neuron

6.6 Resource Utilization:

From Figure 6.2, it is evident that a single neuron with a data width of 16 requires 2 digital signal processing (DSP) blocks. In the Xilinx Zynq Zybo FPGA, we have a total of 80 DSP blocks available. This means that we can accommodate a maximum of 40 neurons with a data width of 16, which is a relatively small number considering the desired number of neurons.

To overcome this limitation, we can reduce the data width to 8, which allows the neurons to be implemented using Look-Up Tables (LUTs) instead of DSP blocks. Figure 6.9 illustrates the implementation report of a model with 100, 8-bit neurons. As the LUTs and flip-flops (FFs) are not exhausted, we still have the potential to incorporate more neurons into the model.

Figure 6.10 displays the implementation report for 170 neurons, which represents the maximum number of neurons we can include. However, some slight adjustments can be made by modifying the density of each layer. Neurons in a layer with fewer input nodes tend to consume fewer resources, allowing for potential resource optimization. Additionally, the depth of the activation function can be fine-tuned, as higher depths will require more LUTs.



Summary: 1 advisory
Implemented DRC Report

Worst Negative Slack (WNS):	1.444 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	18406

Implemented Timing Report

Utilization Post-Synthesis | Post-Implementation

Graph | Table

Power Summary | On-Chip

Total On-Chip Power:	**1.705 W**
Junction Temperature:	**44.7 ℃**
Thermal Margin:	40.3 ℃ (3.4 W)
Effective ϑJA:	11.5 ℃/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

Fig 6.9 Implementation Report of 100 neurons with sigmoid depth of 5

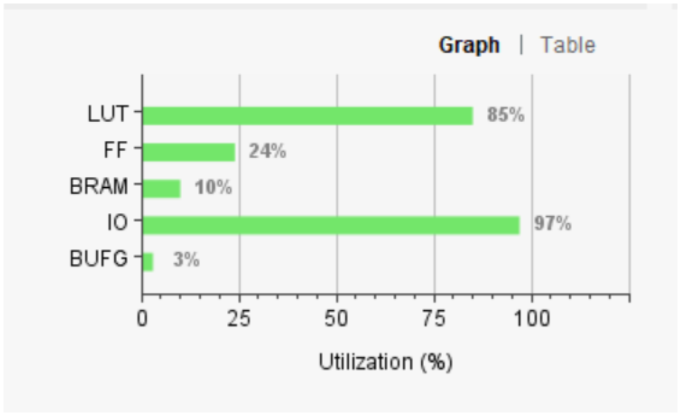
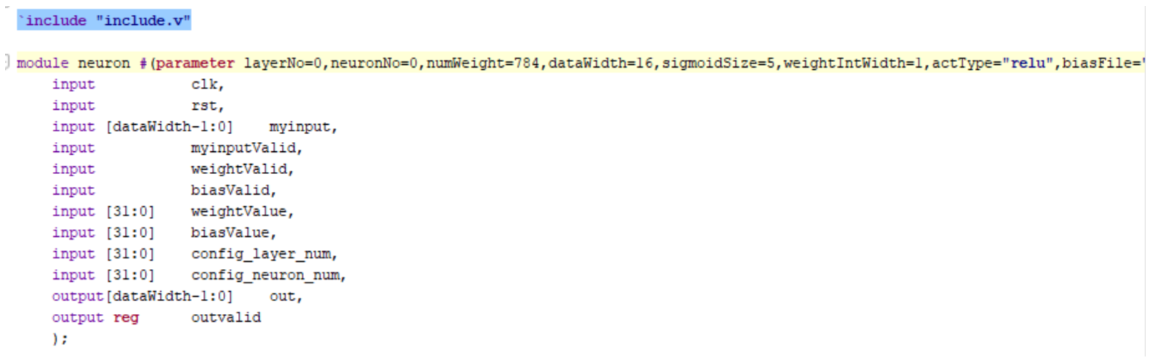
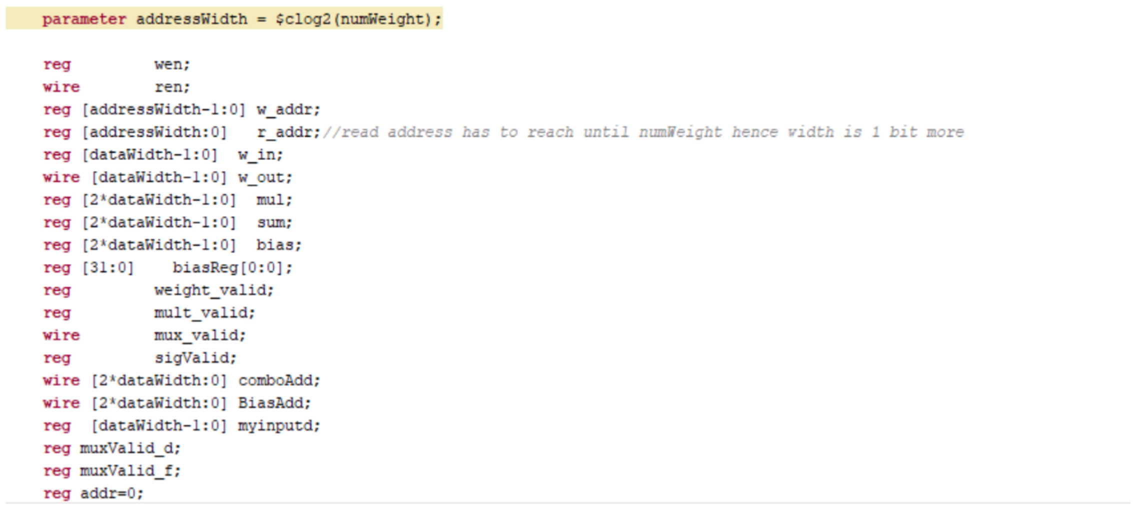


Fig 6.10 Implementation table of 170 Neurons

6.7 Verilog Code:

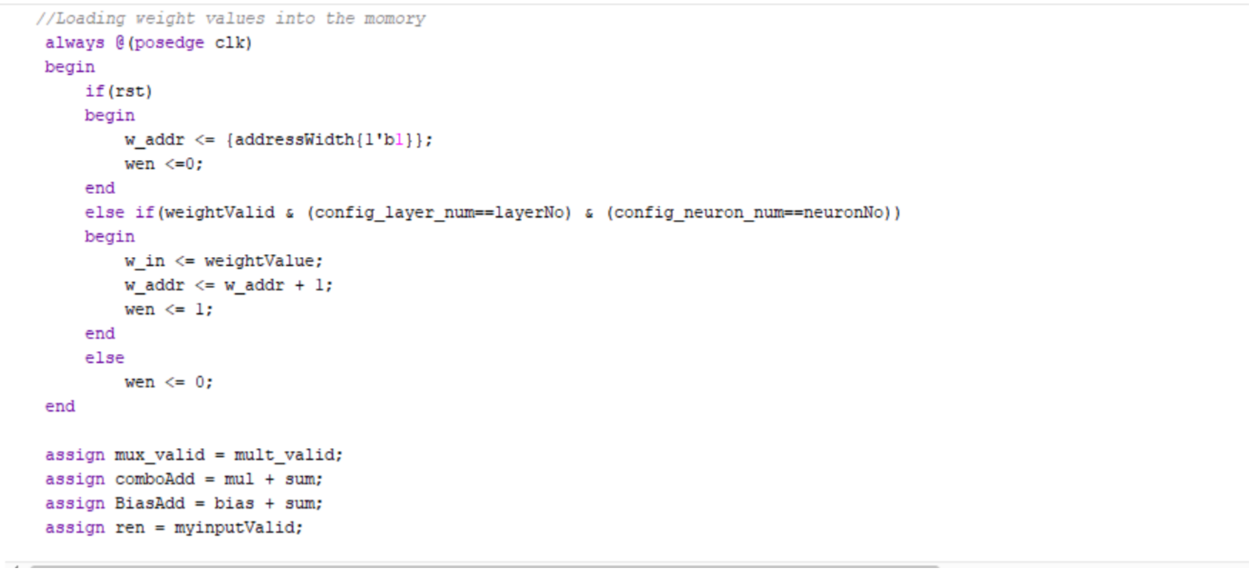


```
`include "include.v"

module neuron #(parameter layerNo=0,neuronNo=0,numWeight=784,dataWidth=16,sigmoidSize=5,weightIntWidth=1,actType="relu",biasFile='
    input           clk,
    input           rst,
    input [dataWidth-1:0]    myinput,
    input           myinputValid,
    input           weightValid,
    input           biasValid,
    input [31:0]    weightValue,
    input [31:0]    biasValue,
    input [31:0]    config_layer_num,
    input [31:0]    config_neuron_num,
    output[dataWidth-1:0]    out,
    output reg      outvalid
    );
```



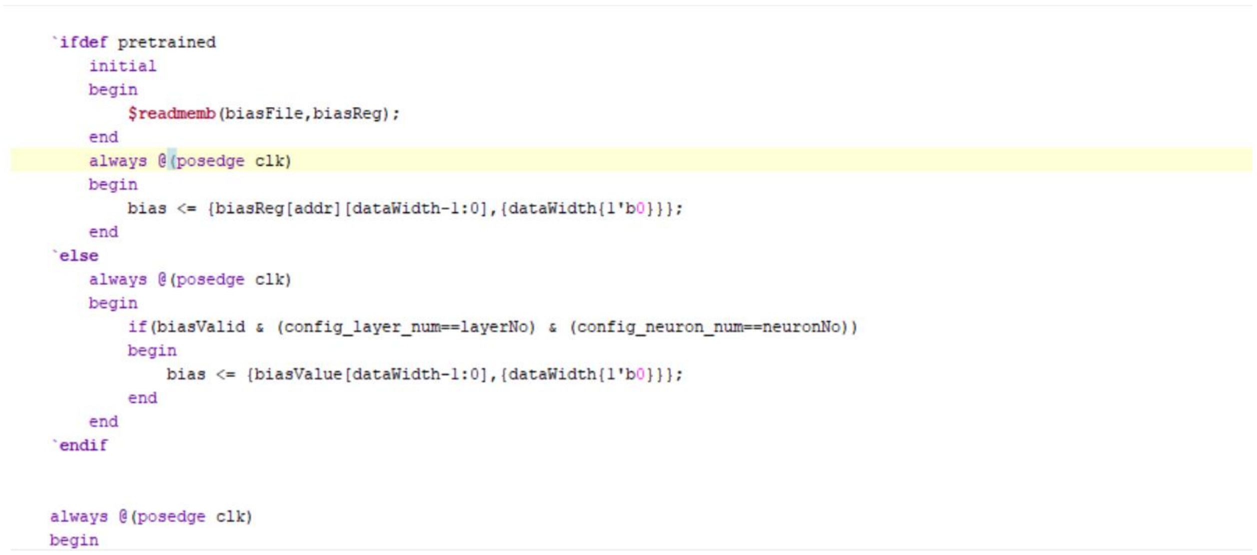
```
parameter addressWidth = $clog2(numWeight);

reg         wen;
wire        ren;
reg [addressWidth-1:0] w_addr;
reg [addressWidth:0]   r_addr;//read address has to reach until numWeight hence width is 1 bit more
reg [dataWidth-1:0]  w_in;
wire [dataWidth-1:0] w_out;
reg [2*dataWidth-1:0]  mul;
reg [2*dataWidth-1:0]  sum;
reg [2*dataWidth-1:0]  bias;
reg [31:0]    biasReg[0:0];
reg         weight_valid;
reg         mult_valid;
wire        mux_valid;
reg         sigValid;
wire [2*dataWidth:0] comboAdd;
wire [2*dataWidth:0] BiasAdd;
reg  [dataWidth-1:0] myinputd;
reg muxValid_d;
reg muxValid_f;
reg addr=0;
```

Above is the code for a single neuron. All the input and output registers are declared with the necessary data width. The **include** file will provide the neuron with the necessary hyperparameters like the type of activation function, number of inputs, number of outputs and data width assigned to the neuron. Some variables like mul are given a size equal to double the data width, this is because when two Qn.m numbers are multiplied, the result is Qn+n.m+m. This means that it will need twice the size than the size of the numbers being multiplied. As the product is added to the equation, the sum variable also had to be initialized with twice the data width.



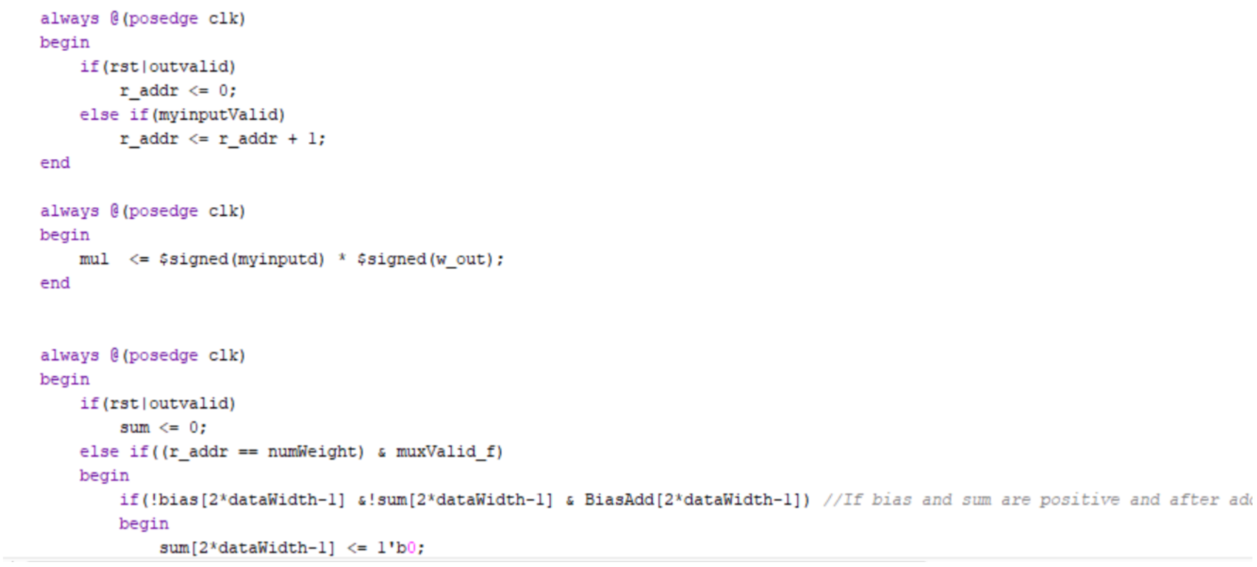
```
//Loading weight values into the momory
always @(posedge clk)
begin
    if(rst)
    begin
        w_addr <= {addressWidth{1'b1}};
        wen <=0;
    end
    else if(weightValid & (config_layer_num==layerNo) & (config_neuron_num==neuronNo))
    begin
        w_in <= weightValue;
        w_addr <= w_addr + 1;
        wen <= 1;
    end
    else
        wen <= 0;
end

assign mux_valid = mult_valid;
assign comboAdd = mul + sum;
assign BiasAdd = bias + sum;
assign ren = myinputValid;
```

The values of weights are added in the neuron; if the reset signal is set high, all weights are made zero; this will make an empty model with no learning. If the weight matches with the specifications of the neuron; it will be loaded into the neuron. After the action on weights, some important calculations are performed to form the linear equation of the neuron.



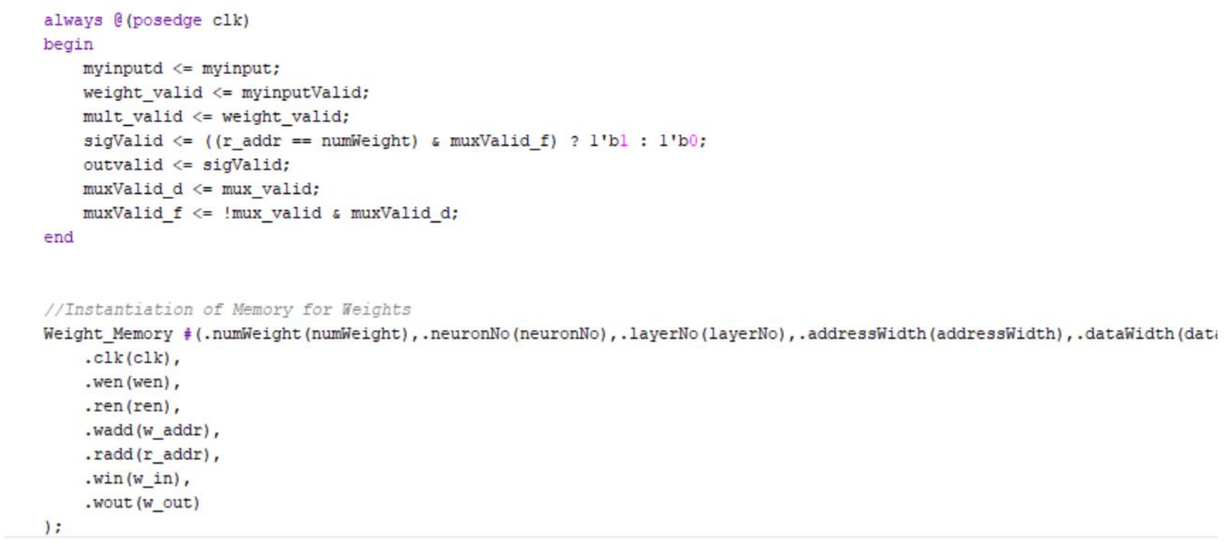
```
`ifdef pretrained
    initial
    begin
        $readmemb(biasFile,biasReg);
    end
    always @(posedge clk)
    begin
        bias <= {biasReg[addr][dataWidth-1:0],{dataWidth{1'b0}}};
    end
`else
    always @(posedge clk)
    begin
        if(biasValid & (config_layer_num==layerNo) & (config_neuron_num==neuronNo))
        begin
            bias <= {biasValue[dataWidth-1:0],{dataWidth{1'b0}}};
        end
    end
`endif


always @(posedge clk)
begin
```

The bias file is read and if the bias matches with the specifications of the neuron, it is fed into the neuron.



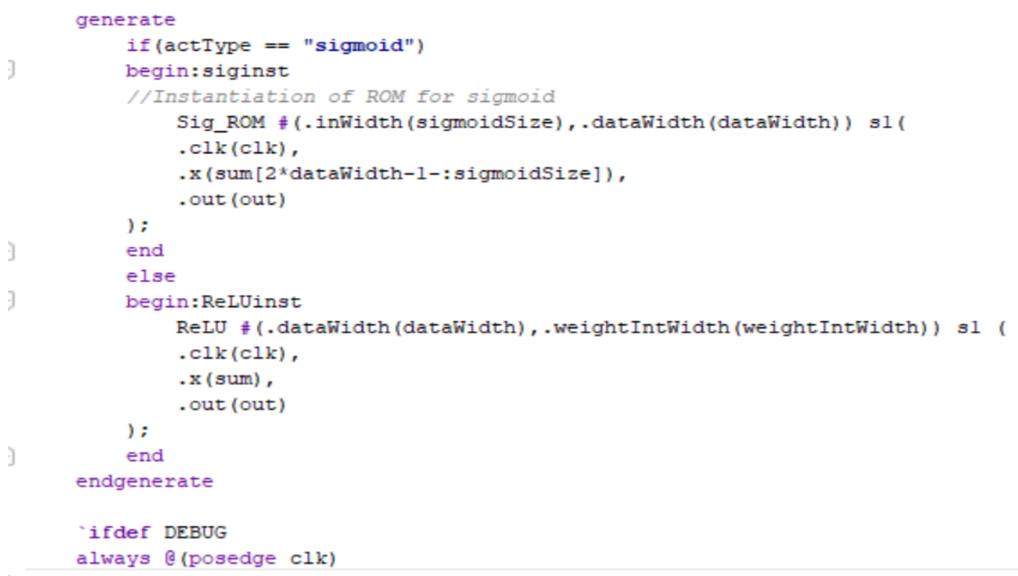
```
always @(posedge clk)
begin
    if(rst|outvalid)
        r_addr <= 0;
    else if(myinputValid)
        r_addr <= r_addr + 1;
end

always @(posedge clk)
begin
    mul  <= $signed(myinputd) * $signed(w_out);
end


always @(posedge clk)
begin
    if(rst|outvalid)
        sum <= 0;
    else if((r_addr == numWeight) & muxValid_f)
    begin
        if(!bias[2*dataWidth-1] &!sum[2*dataWidth-1] & BiasAdd[2*dataWidth-1]) //If bias and sum are positive and after ad
        begin
            sum[2*dataWidth-1] <= 1'b0;
```

To mitigate the potentially detrimental impact of overflow in our mathematical operations, we implement a saturation mechanism. Whenever an overflow is detected, the corresponding variable is saturated by assigning it the highest positive number or the lowest negative number, depending on the nature of the overflow. While this approach may result in some data loss, it is the optimal choice to obtain reliable and accurate results.

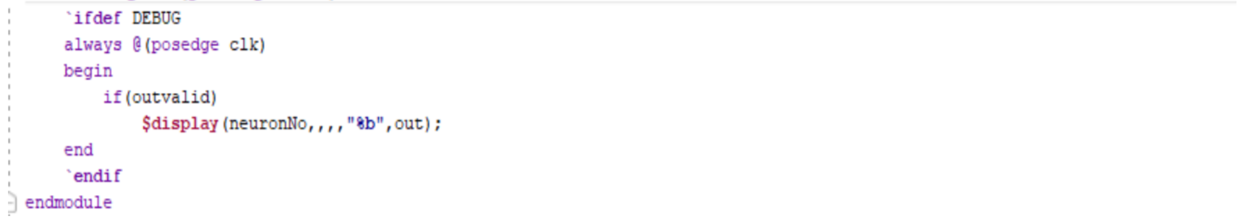


```verilog
always @(posedge clk)
begin
    myinputd <= myinput;
    weight_valid <= myinputValid;
    mult_valid <= weight_valid;
    sigValid <= ((r_addr == numWeight) & muxValid_f) ? 1'b1 : 1'b0;
    outvalid <= sigValid;
    muxValid_d <= mux_valid;
    muxValid_f <= !mux_valid & muxValid_d;
end


//Instantiation of Memory for Weights
Weight_Memory #(.numWeight(numWeight),.neuronNo(neuronNo),.layerNo(layerNo),.addressWidth(addressWidth),.dataWidth(data
    .clk(clk),
    .wen(wen),
    .ren(ren),
    .wadd(w_addr),
    .radd(r_addr),
    .win(w_in),
    .wout(w_out)
);
```

The output signals are adjusted to either delay or start the operation of next neuron of the next layer. The weight memory is instantiated with the variable names used in the code of neuron.

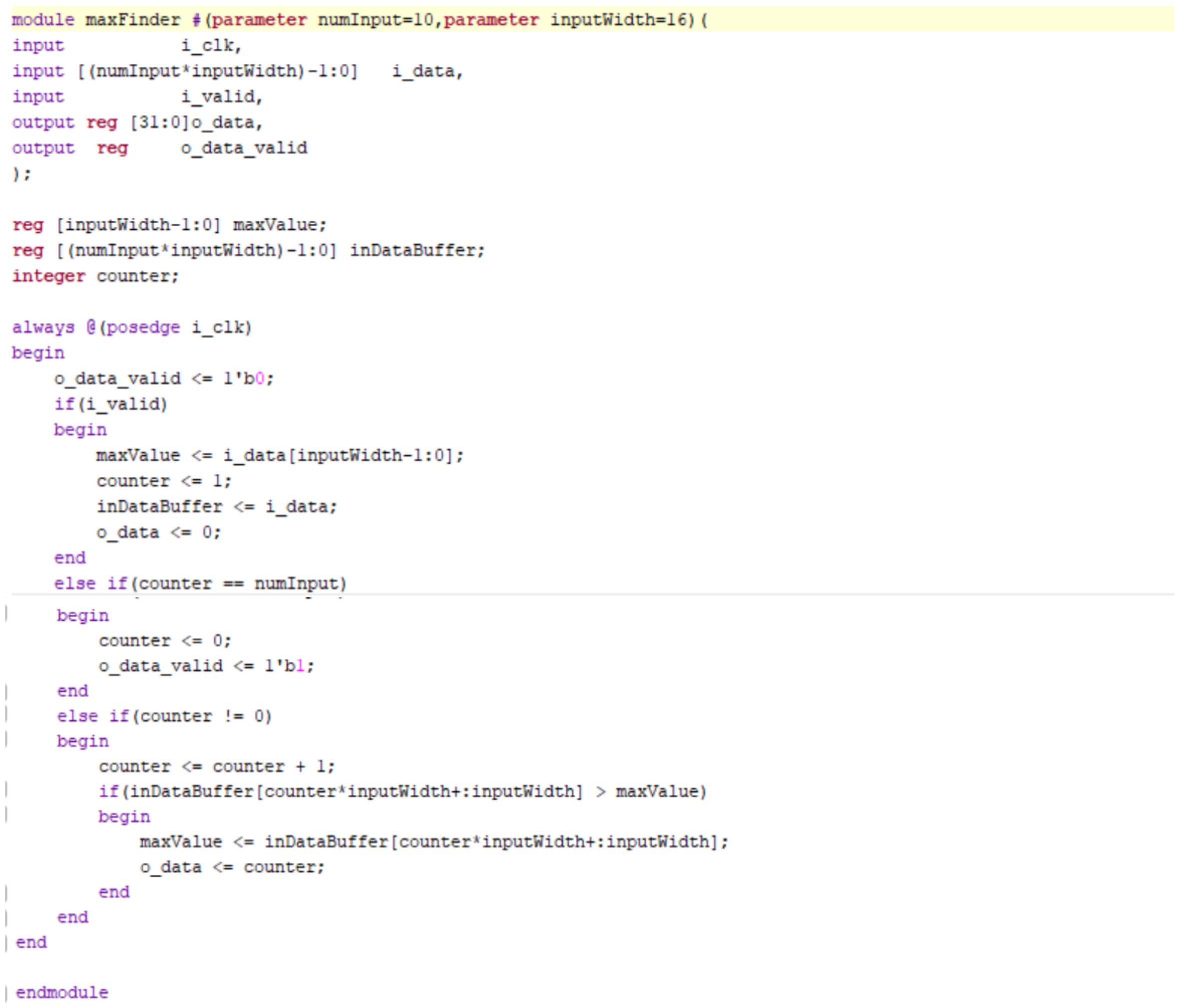


```verilog
generate
    if(actType == "sigmoid")
    begin:siginst
    //Instantiation of ROM for sigmoid
        Sig_ROM #(.inWidth(sigmoidSize),.dataWidth(dataWidth)) s1(
        .clk(clk),
        .x(sum[2*dataWidth-1-:sigmoidSize]),
        .out(out)
    );
    end
    else
    begin:ReLUinst
        ReLU #(.dataWidth(dataWidth),.weightIntWidth(weightIntWidth)) s1 (
        .clk(clk),
        .x(sum),
        .out(out)
    );
    end
endgenerate

`ifdef DEBUG
always @(posedge clk)
```



```verilog
`ifdef DEBUG
always @(posedge clk)
begin
    if(outvalid)
        $display(neuronNo,,,,"%b",out);
end
`endif
endmodule
```

The final step involves selecting the activation function for the neuron. When using the sigmoid activation function, only a portion of the left bits is retained while discarding the remaining right bits to conserve FPGA resources. This may result in some data loss, but it has minimal impact on the results. In contrast, the ReLU activation function does not cause any data loss as it is specifically designed for FPGA circuitry.



```
module maxFinder #(parameter numInput=10,parameter inputWidth=16)(
input           i_clk,
input [(numInput*inputWidth)-1:0]   i_data,
input           i_valid,
output reg [31:0]o_data,
output  reg     o_data_valid
);

reg [inputWidth-1:0] maxValue;
reg [(numInput*inputWidth)-1:0] inDataBuffer;
integer counter;

always @(posedge i_clk)
begin
    o_data_valid <= 1'b0;
    if(i_valid)
    begin
        maxValue <= i_data[inputWidth-1:0];
        counter <= 1;
        inDataBuffer <= i_data;
        o_data <= 0;
    end
    else if(counter == numInput)
    begin
        counter <= 0;
        o_data_valid <= 1'b1;
    end
    else if(counter != 0)
    begin
        counter <= counter + 1;
        if(inDataBuffer[counter*inputWidth+:inputWidth] > maxValue)
        begin
            maxValue <= inDataBuffer[counter*inputWidth+:inputWidth];
            o_data <= counter;
        end
    end
end

endmodule
```

Here is the code for our hardmax function. It is designed as an alternative to the softmax function, considering the trade-off between accuracy and computational efficiency. While the softmax function provides higher accuracy, it is computationally more expensive. In contrast, the hardmax function aims to maximize the output of the neuron with the highest probability, while nullifying the outputs of other neurons. This results in an output vector with a length equal to the number of output neurons. The neuron with the highest probability is assigned a value of 1, while the remaining neurons are assigned a value of zero.

CHAPTER 7 CONCLUSIONS AND FUTURE WORK

7.1 Block Diagram:

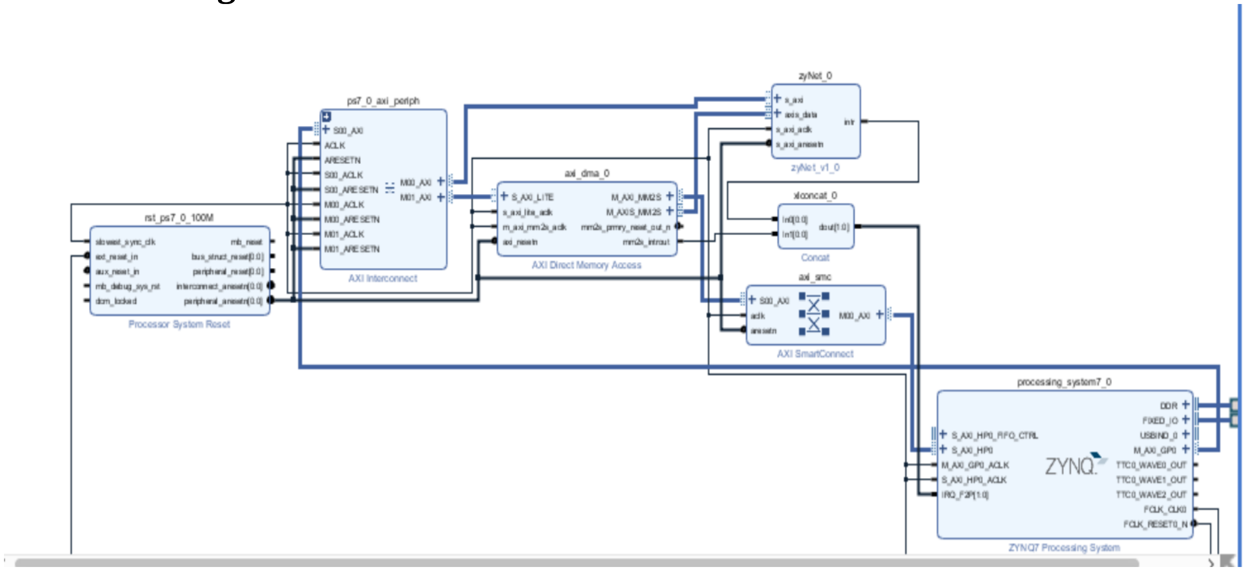


Fig 7.1 Block Diagram of FPGA NN

ZYNQ processor of our FPGA is used to display the FPGA output on teraterm. Since, large amount of data had to be transferred from our computer to the FPGA board, we used AXI_DMA for this purpose. AXI_SMARTCONNECT which is a configurable interconnect component which provided us with a scalable system which is a plug and play solution for connecting multiple masters and slaves.

Processor System Reset IP is used to ensure proper initialization and synchronization of the processor for more reliable and predictable system operation. It generated the necessary reset signals to initialize the processor including the asserting and de-asserting of the reset signals. Path for communication between neural network and processor is developed using AXI_LITE interface. This is necessary for the situation when our bias and weight files or other memory files cannot provide input, in this case, the processor will take the command and will provide the necessary data.

The design underwent validation, wiring, synthesis, and implementation using Vivado tools on a Xilinx Zybo board. The processor was programmed using the Vivado SDK kit. For data input and output, a UART port was utilized, enabling the communication between the FPGA design and external devices.

7.2 Constraints and Discussion:

FPGAs, with their support for parallel processing, can be an ideal device for implementing neural networks. However, the limited memory resources, especially in low-cost hybrid FPGAs, can present challenges when dealing with deeper neural networks. If we utilize a more powerful board like the Zed-board instead of the Zybo board we used previously, a 32-bit implementation with 80 neurons using the sigmoid activation function would require 50,349 LUTs, 15,543 flip-flops, 70 BRAMs, and 200

DSP slices. Similarly, an implementation using the ReLU activation function would need 54,560 LUTs, 18,000 flip-flops, 30 BRAMs, and 200 DSP slices. These numbers correspond to approximately 95% of the LUTs, 17% of the flip-flops, 22% of the BRAMs, and 100% of the DSP slices available on a Zed-board.

While the use of 32-bit neurons can outperform 8-bit neurons, a mere 80 neurons may not be sufficient for tackling complex and sensitive problems such as cancer detection.

One interesting result is the determination of the depth of the Sigmoid Look-Up Table (LUT), which is calculated using the expression 2^(address_bits). Simulation results have shown that when the number of bits is less than the sum of integer bits used to represent weight and input values, the accuracy is significantly reduced. Typically, the number of integer bits for input is set to 1 due to normalization, which doesn't pose a significant issue. However, for better estimation of weight values, the integer bits for weight values are usually set to 4. To address this issue, the minimum number of address bits for the sigmoid function is set to 5. If higher prediction accuracy is required, the depth of the sigmoid can be increased further, although this comes at the expense of BRAM resources.

When it comes to cancer detection in a hospital setting, high-end Xilinx FPGA devices should be used as edge devices. Incorrect diagnosis can be extremely dangerous and reduce the chances of patient survival if treatment is delayed. The way forward for improved diagnosis is clear: we need to deploy deeper Deep Neural Networks (DNNs) on our edge devices. This can be achieved through a hybrid approach, where FPGAs are connected to cloud data centers to increase storage capacity, or by deploying multiple low-cost FPGAs in parallel to increase processing power and storage. Alternatively, a single high-end FPGA device capable of handling the complexity of the model can be utilized.

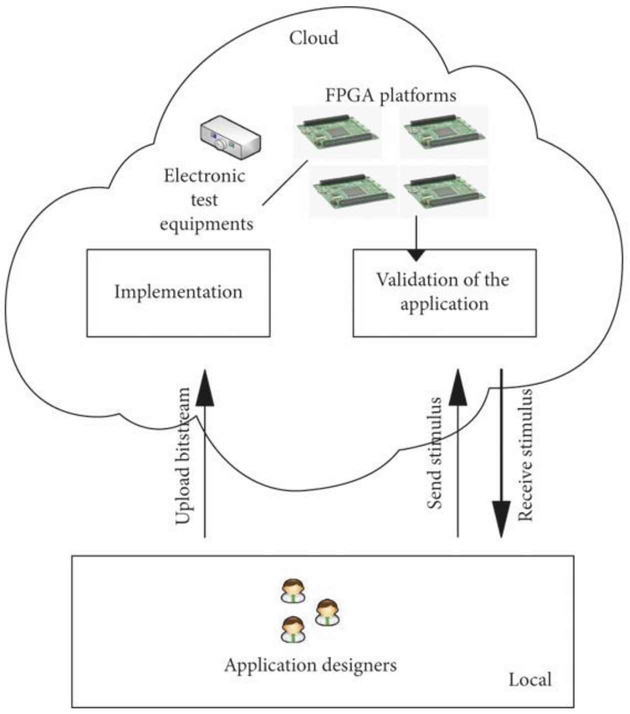
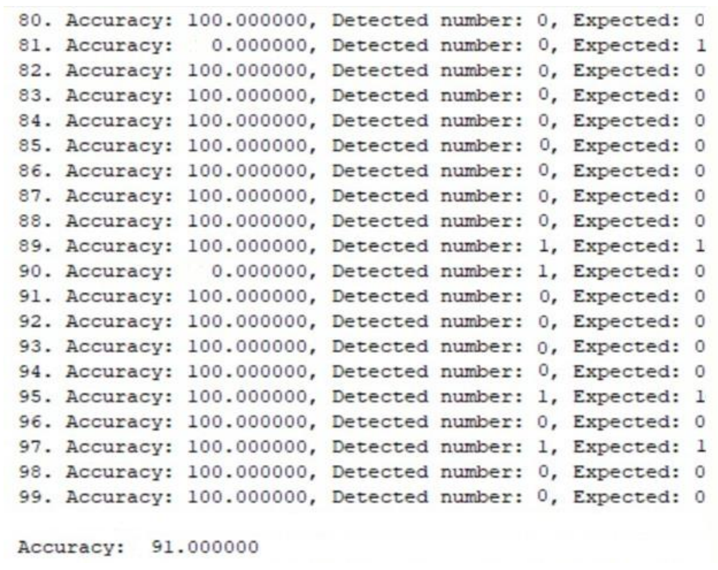


Fig 7.2 Cloud and FPGA proposed architecture

7.3 Results:



```
80. Accuracy: 100.000000, Detected number: 0, Expected: 0
81. Accuracy:   0.000000, Detected number: 0, Expected: 1
82. Accuracy: 100.000000, Detected number: 0, Expected: 0
83. Accuracy: 100.000000, Detected number: 0, Expected: 0
84. Accuracy: 100.000000, Detected number: 0, Expected: 0
85. Accuracy: 100.000000, Detected number: 0, Expected: 0
86. Accuracy: 100.000000, Detected number: 0, Expected: 0
87. Accuracy: 100.000000, Detected number: 0, Expected: 0
88. Accuracy: 100.000000, Detected number: 0, Expected: 0
89. Accuracy: 100.000000, Detected number: 1, Expected: 1
90. Accuracy:   0.000000, Detected number: 1, Expected: 0
91. Accuracy: 100.000000, Detected number: 0, Expected: 0
92. Accuracy: 100.000000, Detected number: 0, Expected: 0
93. Accuracy: 100.000000, Detected number: 0, Expected: 0
94. Accuracy: 100.000000, Detected number: 0, Expected: 0
95. Accuracy: 100.000000, Detected number: 1, Expected: 1
96. Accuracy: 100.000000, Detected number: 0, Expected: 0
97. Accuracy: 100.000000, Detected number: 1, Expected: 1
98. Accuracy: 100.000000, Detected number: 0, Expected: 0
99. Accuracy: 100.000000, Detected number: 0, Expected: 0

Accuracy:  91.000000
```

Fig 7.3 Hardware Results

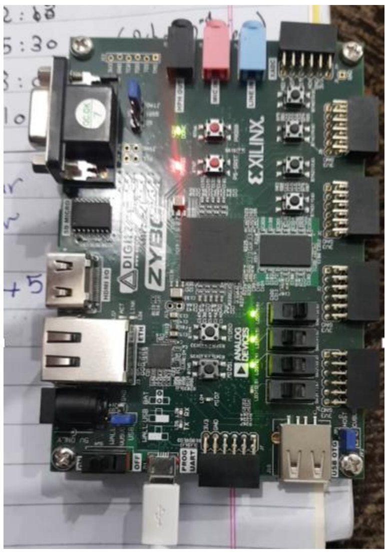


Fig 7.4 Zybo Board

In the results, **1** represents cancerous while **0** represents non-cancerous. The accuracy achieved of our models is in between **85-93%**. This depreciation is due to the compromises made in the neural network deployed in FPGA.

7.3 FUTURE WORK

In future work, several key aspects can be addressed to enhance the implementation and usability of the FPGA-based neural network for cancer detection. The following points outline potential areas of focus:

1. Optimization for Higher-End FPGAs: Explore the utilization of high-end FPGA devices with increased resources, such as the Xilinx Virtex-5 XC5VLX330, to replicate denser neural networks. These FPGA devices offer greater capacity for implementing larger models and accommodating more layers and neurons. Investigate the performance improvements achieved by deploying deeper neural networks on these high-end FPGAs.

2. Dataset Enhancement: Consider expanding the dataset used for training and validation by sourcing locally available data, such as the NORI dataset. Incorporating a diverse and representative dataset can enhance the robustness and generalization capabilities of the trained neural network model.

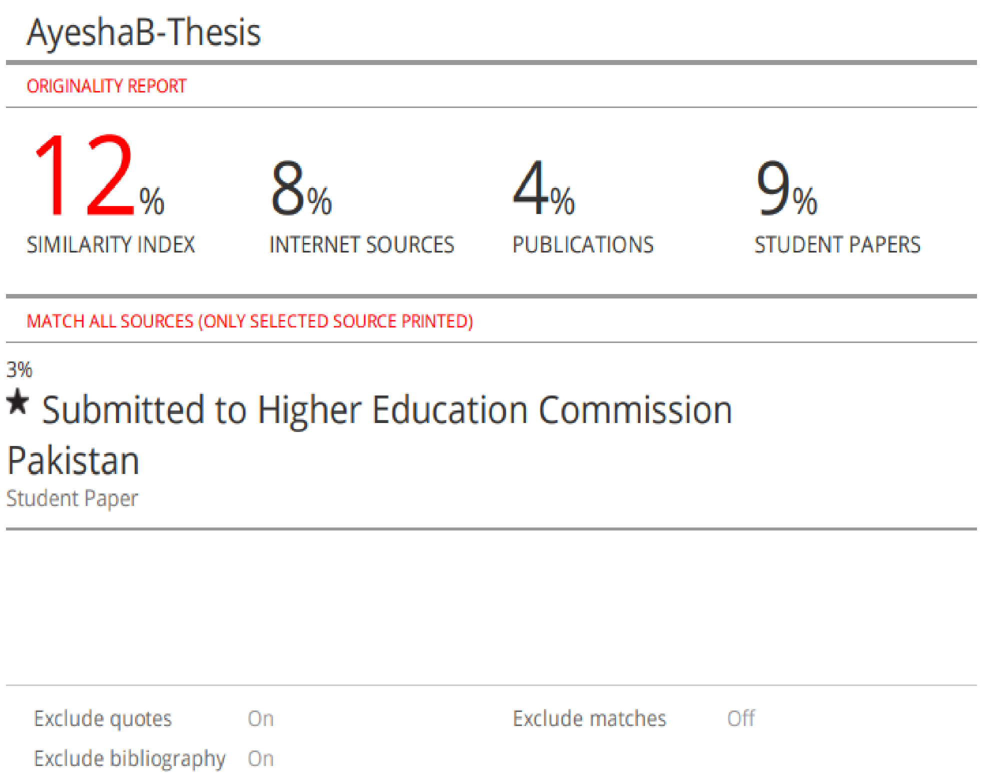
3. Shift Convolutional and Pooling Blocks: Investigate the optimization of convolutional and pooling blocks within the FPGA design. Explore techniques such as parallelization, pipelining, and resource sharing to maximize the utilization of FPGA resources and improve the computational efficiency of these blocks.

4. User-Friendly Interface: Design and develop a user-friendly end product that can be easily utilized by doctors and medical professionals. Consider incorporating a graphical user interface (GUI) that provides intuitive controls for loading input data, initiating the cancer detection process, and presenting the results in a clear and interpretable manner.

REFERENCES

- Kizheppatt Vipin (2020) **ZyNet: Automating Deep Neural Network Implementation on Low-Cost Reconfigurable Edge Computing Platforms**
- Zongmo, Pu, et al. (2022) BS-80K: The first large open-access dataset of bone scan images
- Tan, Le. (2019) EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks
- Szegedy, Vanhoucke, et al. (2015) Rethinking the Inception Architecture for Computer Vision
- Zhuang, Hanzi, et al. (2022) A ConvNet for the 2020s
- Gao, Zhuang, et al. (2016) Densely Connected Convolutional Networks

PLAGIARISM REPORT


AyeshaB-Thesis

ORIGINALITY REPORT

12%	8%	4%	9%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

MATCH ALL SOURCES (ONLY SELECTED SOURCE PRINTED)

3%

★ Submitted to Higher Education Commission Pakistan

Student Paper

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		