Disease classification using Chest X-Ray images and Deep Learning

BIRMINGHAM CITY University

Hemal Nakrani (S21154546)

Submitted in fulfilment of the requirement
for the degree of

Master of AI and Machine Learning

Faculty of Computing, Engineering and the Built Environment
Project supervisor: Dr Essa Shahra

2022-2023

Acknowledgements

I want to start by expressing my gratitude to my supervisor, Dr Essa Sahra, whose knowledge and patience greatly helped with my graduate experience. Your advice and guidance has been essential and very helpful during this journey.

I am also grateful to the Computer engineering and built environment department staff and professors for providing me access to knowledge and tools to conduct this research. My comprehension for this topic has been improved by priceless discussion, workshops and lectures.

My sincere gratitutde is extended to all my peers and other researchers who have contributed to this study and development of this project. At every steps and stage of the development of the project, they have helped me with their critique and opinions.

I would also acknowledge the creators, developers and various open source tools and platforms like Kaggle which helped me processes like data collection, training models and also gave free access to hardware and software resources.

Last but not least, my deepest gratitude is reserved for my family. Their unwavering faith in my abilities ,have been pillars supporting me through challenges and triumphs of this research journey.

Abstract

The focus of this project is to use various deep-learning networks and techniques to classify multiple diseases that can be diagnosed using chest X-rays. Every year we see a rise in cardiovascular and respiratory diseases. To tackle this we need a reliable and robust automated diagnosing tool.

The study and research done in this project will explore the use of different large pre-trained CNN (Convolution neural network) models such as ResNet-152, VGG19, and EfficientNet to classify chest X-rays. We are also using a self-attention mechanism architecture-based pretrained visual transformer (ViT). This project also focuses on other deep learning techniques like ensembling which takes advantage of all these trained models.

Training and evaluation of all this model will be done in this dissertation using the NIH Chest X-ray dataset, which is an open-source dataset. The main study in this dissertation will investigate the strengths and weaknesses of these models, thereby conducting a comparative analysis of the performance of these models.

All the steps taken like image pre-processing, finetuning of the models, data cleaning, data augmentation and balancing of the dataset will be assessed and explained in detail. Later after creating an architecture using all these models. This project will also discuss the techniques used to deploy this architecture as a web application. The deployment of this web application will make use of technologies such as docker, FastAPI and streamlit. The web application will help users to identify the disease and diagnose the health of the patient using the image of a chest x-ray image.

In this dissertation, we will be developing an image classifier using deep-learning CNN models, vision transformers, transfer learning, and ensembling techniques. We will be developing an end-to-end tool, which is a web application that anybody can use to diagnose their chest X-rays without having any prior knowledge of the medical field.

This web application will be able to detect 14 diseases including lung diseases like Pneumonia, Infiltration which can be detected using chest x-ray and will also provide probabilities of the same.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The arena of medical imaging consistently highlights the significance of chest X-rays, making them among the most common radiological procedures conducted globally. These images are vital, shedding light on the internal structures of the thoracic region, thus helping identify a diverse range of conditions from lung-related disorders to issues surrounding the heart. However, the soaring numbers of X-ray images, combined with the myriad of potential diseases, make their manual assessment not only lengthy but also susceptible to inaccuracies. With the pressing demand for swift, precise, and effective diagnostic approaches, there's a clear call for groundbreaking techniques.

In this research endeavour, we navigate the multifaceted realm of multi-label image classification tailored for chest X-rays. We'll be journeying through the conceptualization, development, and verification of an avant-garde model designed to pinpoint a range of thoracic conditions. By harnessing cutting-edge machine learning tools and expansive image databases, our pursuit is to propel the current benchmarks in automated chest X-ray analysis, ultimately supporting healthcare experts and potentially enhancing patient care.

Lung diseases like pneumonia and tuberculosis are one of the major reasons for many fatalities all around the world. This disease can generally lead to symptoms like fever, cough, fatigue, and vomiting and in some cases like Pneumothorax can also lead to partial or complete failure of the lungs. Early diagnosis of such lung diseases is essential to avoid life-threatening events in patients. Disease like Tuberculosis is one of the highly transmissible diseases through the air. Therefore, it is necessary to detect such diseases as early as possible, so doctors can start the treatment as soon as possible. The novel Coronavirus that took the whole world by storm was curable if it was detected in its early stages Soriano et al. (2020).

Most lung diseases are detected by doctors by observing chest X-ray radiographic images (CXR). X-rays are taken by radiologists and doctors use this to get current conditions of the lungs of the patient. Using X-rays and other symptoms, doctors can detect the disease and start the treatment accordingly Kim et al. (2022a)

It is not possible to get X-rays checked by experts or doctors in remote and rural areas. Deep Learning can be used generally to detect diseases from X-rays. Convolution neural networks (CNN) are one of the best algorithms which can be used to extract important features from the image, these features are further used for the classification of the images. During the COVID period, there was a huge shortage of doctors and there were not enough doctors to diagnose the X-rays of the patient. The deep Learning model can fill this gap and predict the disease by using X-ray images Bharati et al. (2020).

To train a deep learning model, we need a huge, labelled dataset of patients with different types of diseases and health conditions to make the model more robust, accurate and reliable. Many big health institutions have made their data open source for developers. The National Institutes of Health (NIH) has made its dataset of chest X-rays open source for data analysts and data scientists. Many researchers have tried different approaches and techniques to achieve high accuracies using this dataset. These developed models can greatly help doctors detect lung diseases and can decrease death rates if such models are integrated into health sectors and departments. Many developing and underdeveloped countries can take advantage of this technology as it is also low-cost and reasonable Bharati et al. (2020).

This project will be discussing various ideas and approaches that other researchers applied in this field of classification of diseases using X-rays. We will understand the advantages and disadvantages of the ideas used and compare their results using relevant evaluation metrics. To solve this problem, we will be making a hybrid architecture that will take advantage of various state-of-the-art pre-trained models using transfer learning. We will try to integrate current technologies into our hybrid architecture, like visual transformers which are used for image processing. After creating our hybrid AI model, we will evaluate it and compare it with the previously used AI model.

After developing an AI model, we will also deploy this AI model by using a web app. This web app can be used by anyone who has internet access. This will make this technology available to everyone and will reduce life-threatening events for patients and will help them to get early treatments.

1.1 Background

Recently there has been a rapid increase in lung-related diseases throughout the globe and millions of people died due to late treatment, for example, COVID-19. There was a huge shortage of doctors during the lockdown period and therefore, there is a need to come up with different solutions to detect such diseases as early as possible. Artificial Intelligence can play the main role in solving this problem of classifying lung diseases by using X-rays. Many researchers and developers have come up with different AI models which make use of different algorithms and techniques. For example, CNN, Transfer learning and ensembling techniques are used by many

researchers and developers to achieve higher accuracies in the classification of the disease. We will analyse all this research and methods and try to come up with solutions that will provide higher accuracies than previously developed AI models Brunese et al. (2020).

1.2 Aim

Train and deploy a hybrid AI model architecture using pre-trained CNN models and a Visual transformer to detect lung diseases in patients using chest X-ray images.

1.2.1 Objectives

1. Perform research on previously used techniques and approaches used to detect disease using CNN, Visual Transformer, and X-ray images.
2. Acquire labelled and high-quality data related to chest X-ray images.
3. Research and identify new techniques and approaches towards this problem that can be used to classify diseases accurately and efficiently.
4. Design and experiment using a newly found approach and develop a deep learning model or architecture using Convolutional Neural Network (CNN) models, Visual Transformer models, Transfer Learning, and ensembling techniques.
5. Evaluate this deep learning model using suitable evaluation metrics and compare them to previously used techniques and models.
6. Deploy and develop a web app using the newly developed deep learning model.

1.2.2 Ethical Consideration

The details like the name and address of the patient are removed from the dataset to protect the privacy of the patient.

Chapter 2

Sytematic Literature Review

2.1 Introduction

In any academic pursuit, especially in fields as dynamic as medical imaging and deep learning, it is imperative to ground one's research within the broader context of existing literature. By understanding the milestones achieved, challenges faced, and gaps yet to be addressed, we not only enrich our knowledge but also ensure that our endeavours are both relevant and contributory. This foundation is laid through a meticulous and robust review of the literature, and in our case, a systematic literature review serves this purpose.

Systematic literature reviews distinguish themselves from traditional reviews by their rigorous and structured approach. The aim is to provide an unbiased overview and summary of studies conducted on the topic of medical imaging and the use of AI in detecting diseases using X-rays. This ensures that the knowledge of the researchers who tried different approaches and techniques will become a definitive guide while pursuing new approaches in this field.

In the subsequent sections, we will delve deep into the vast ocean of research surrounding image classification of chest X-rays. This literature review aims to record the timeline of methodologies, techniques and key findings and their strengths and weaknesses evident in this field. By studying and summarizing all these studies, we hope to establish a clear picture of where AI stands today in classifying chest X-rays and where it might be headed in the coming years.

2.2 Defining Research Objectives and Questions

Initially, we define specific study objectives and queries. This framework will guide our search strategy, enabling us to identify the specific approaches and techniques used in detecting diseases using chest X-rays and deep learning techniques and models.

2.3 Developing Search Strategy

1. **Keywords:**
   - Main terms used include "Chest X-rays" and "Multi-label classification."
   - Synonyms like "Thoracic imaging," "Radiographs," and "Multi-tag classification" were also considered.
2. **Boolean Combinations:**
   - Operators like "AND" were used to combine primary terms with related terms, e.g., "Chest X-rays" AND "Multi-label classification."
3. **Databases Targeted:**
   - Primary sources for the literature review included:
     - PubMed
     - IEEE Xplore
     - Web of Science
     - Google Scholar
4. **Filters Applied:**
   - The search was limited to articles published within the past 10 years.
   - Emphasis was placed on peer-reviewed articles, conference proceedings, and reviews.
   - The primary language of the articles was English.
5. **Additional Searches:**
   - Hand searching and the snowballing method were employed by reviewing reference lists of primary articles.
6. **Documentation:**
   - Detailed records of all search parameters and results were maintained to ensure transparency and reproducibility.
7. **Review Process:**
   - The review process consisted of:
     - Initial screening of titles and abstracts.
     - In-depth full-text reviews for final article selection.

2.4 Classification of chest x-rays using AI

Many other developers have previously tried to classify various lung diseases using X-ray images. There are varying techniques and approaches used to solve this problem statement. We will discuss some of those solutions given by other authors, in this brief literature review.

2.4.1 CNN and Transfer learning in the classification of chest x-rays

One of the most used and novel techniques used to classify images in Artificial Intelligence is the Convolution Neural Network (CNN). CNN are one of the most reliable ways for the classification of images. CNN works by extracting features from an image using various filters. While training CNN, it finds the value of these filters used to extract crucial features from an image. CNN uses these extracted features to classify an image. Therefore CNN is considered one of the important technologies for medical image recognition Huang & Liao (2022).

Training a CNN model from scratch requires lots of data and computational power and it can be very time-consuming, therefore most authors have used Transfer Learning. Transfer Learning is a technique where we freeze the weights of convolution layers of the pre-trained models and just change the output layer according to our needs. Some of the trending CNN models are the efficient V2-M model, ResNet, Alexnet, VGGNet, Alexnet and many more. Some of the pre-trained CNN model like ResNet is trained and developed by big companies like Google using huge datasets and infrastructure. Transfer Learning takes advantage of these state of art pre-trained models and we can fine-tune those models according to the application and problem statement Basu et al. (2020).

One of the papers discusses the use of transfer learning in classifying healthy patients and patients suffering from COVID-19. They experimented with various pre-trained models like AlexNet, ResNet, and VGGNet and found that VGGNet was the best-performing model among all other models with an accuracy of 99 per cent for classifying COVID-19 patients. ResNet was the second best due to its deep architecture. The author used 5-fold cross-validation to evaluate all the used models Basu et al. (2020).

The xception model was used by one author. The xception model is 71 layers deep CNN model. They fine-tuned this model to be used for the classification of multiple lung diseases using X-ray images. They used various pre-processing steps before training the model on data. Preprocessing steps like images were passed through a median filter to remove salt and pepper noise. CLAHE was also used to enhance the image and to increase the contrast of the image. They compared different pre-processing filters like CLAHE and AHE. Fine-tuned Xception model with pre-processing of CLAHE provided an accuracy of 97.3 per cent Yimer et al. (2021).

The authors also tried to use the architecture of pre-trained models like EfficientNet

v2-M but without using transfer learning. Instead of transfer learning to train their model from scratch using just the architecture of EfficientNet v2-M. they used this model to identify 3 classes of lung diseases. The overall accuracy that this model was able to achieve was 82 per cent, which is very less compared to models built using transfer learning. This proves that without using transfer learning, pre-processing steps and data augmentation, it is very difficult for multi-level classification of chest X-rays Kim et al. (2022b).

Authors have also used Residual Network (ResNet) to train their model. They modified the ResNet by replacing global average pooling with adaptive dropout. This allows them to convert multiclass classification into binary classification and the model can be trained parallelly using a GPU. Basically, the model will be trained multiple times parallelly as per the number of classes. This allows the author to take advantage of parallel computing and external GPU and thereby reducing the time required for training Zhang et al. (2020). Another pre-trained CNN model used by another author is DenseNet121, where they used this model to classify 8 lung diseases using chest X-rays. The model managed to score 75 per cent overall which is not good as compared to other models.

2.4.2 Ensembling in classification of chest X-rays

Ensembling is also used by one of the authors, where they used Inception v3, MobileNetV2, ResNet101, NASNet and Xception (pre-trained models). Each of these models was fine-tuned on the dataset of chest X-rays and each model was given a copy of the input image. Each model's output is taken and passed through a fully connected layer which will be trained on the chest X-ray dataset. At the output of the fully connected layer, a softmax activation function is used. The weighted ensemble hybrid model showcased an overall accuracy of 93 per cent which was higher as compared to any individual model Iqball & Wani (2023).

Semi-supervised deep learning is also used to classify chest X-ray images. The developer has used a GAN-based semi-supervised deep learning model which generates images which look similar to labelled data. This increases the size of the data and allows us to train a model with a relatively small amount of data. If we don't have enough labelled data, then this technique is proven to be effective. The author was able to achieve 73.08 per cent accuracy by using just 10 labelled images of each class, whereas a CNN model requires at least 400 labelled images per class. Another advantage of using GAN based semi-supervised model is its robustness compared to CNN and also it is less prone to overfitting Madani et al. (2018).

After reading different articles and published papers, there are various techniques used for the classification of X-ray images. Deep learning models like Convolution Neural Network (CNN) are trained from scratch whereas the majority of papers used transfer learning and pre-trained CNN models, for example, ResNet, VGGNet, Xception etc. Hybrid models were also used by ensembling various pre-trained CNN

models and taking the weighted sum of their output and later passing them to dense layer networks. Lastly, the author also used ViT (Visual Transformer) which is a new concept and technology. Pre-processing and data augmentation was also used in many papers in order to improve the quality of the data and to also increase the robustness of the developed model. After looking into all the concepts, techniques and approaches to the problem of X-ray image classification, we understood the general trend and technologies used. We also studied the techniques which show great potential and can be improved with further development.

2.4.3 Visual Transformer in classification of chest x-rays

AI technologies like visual transformers are also used to classify X-ray images. The visual transformer works on the principle of self-attention mechanism and has various encoder decoder pair in its architecture. They compared the performance of Convolution Neural Network (CNN), hybrid model (VDSNet) and visual transformer (ViT). ViT is a visual transformer pre-trained by hugging face and can be used by anyone using transfer learning. A comparison of all these models was done using various evaluation metrics like accuracy, precision, recoil and F1 score. ViT performed better as compared to other models by a fraction. ViT is easier to train and its performance increases as the number of data increases. This shows that ViT as a promising technology in the field of image classification Uparkar et al. (2023).

The Input Enhanced Vision Transformer (IEViT) is a novel architecture developed by Gabriel Iluebe Okolo and his team, tailored for chest X-ray image classification. Building upon the Vision Transformer (ViT) foundation, IEViT introduces significant enhancements, such as the integration of a convolutional block for in-depth image embedding and an iterative input enhancement feature. This design ensures a comprehensive image view, facilitating richer feature extraction. Notably, the IEViT model consistently surpasses the traditional ViT in performance metrics across various datasets. For instance, the IEViT-B/32 variant achieved a perfect F1-score on the Tuberculosis dataset. When compared against other established CNN models, IEViT remains competitive, showcasing its potential for future research and clinical applications. In essence, the IEViT model, with its advancements over the standard ViT architecture, emerges as a leading contender in chest X-ray image classification, demonstrating superior performance metrics across diverse datasets Okolo (2022).

For multi-label chest x-ray classification, different variants of ViT are also used. pyramid transformer (MXT), a new variant of ViT is performing better as compared to traditional ViT. The self-attention layers of the MXT can capture both short-range as well as long-range visual information Jiang et al. (2022).

2.4.4 Gap Findings

Experimentation and development of hybrid models which include a visual transformer, and pre-trained CNN models like ResNet have yet to be seen as ViT is fairly new compared to CNN models. The results of using Ensembling techniques like majority voting, weighted average and stacking with all these models have yet to be discovered.

Chapter 3

Proposed Methodology

This chapter provides our proposed solution to solve the problem statement of classifying Chest X-rays

3.1 Introduction

In this academic exploration, understanding what has been done in previous chapter is one part of the equation. The innovation lies in moving beyond the known, discovering solutions that are yet-to-be-discovered, and making relevant contributions to the field. The previous chapters meticulously combed through the extensive body of literature, revealing the strengths, weaknesses, and gaps of existing methodologies. Having grounded ourselves in that knowledge, this chapter transitions from the retrospective to the prospective, introducing a novel methodology tailored to address the identified challenges in the multi-label image classification of chest X-rays.

The nature of chest X-rays, laden with intricate details and nuanced variations, demands a method that is not only precise but also adaptable. It is essential to accommodate the fact that patients may present with multiple co-existing conditions, making the task of classification multifaceted. the methodology proposed in this chapter seeks to leverage advanced AI models and insights drawn from both the domains of medical imaging and deep learning.

By the end of this chapter, readers will gain an understanding of the reasoning guiding this new approach, the technical components, deep learning models used and the rationale behind each decision. Our main goal is to enhance the accuracy and efficiency of chest X-ray interpretations, potentially making way for better diagnostic outcomes and improved patient care.

3.2 Problem Statement

In the realm of medical diagnostics, chest X-rays stand as one of the most widely employed imaging techniques, offering a non-invasive window into thoracic structures. A crucial challenge encountered in interpreting these images is the potential existence of multiple concurrent anomalies or conditions within a single X-ray. Traditional diagnostic approaches often rely on radiologists' expertise to identify and label these conditions, a process that can be time-consuming, subject to human error, and potentially inconsistent due to variations in individual expertise Shelke et al. (2021).

Moreover, as the volume of diagnostic imaging continues to grow, the strain on radiologists intensifies, increasing the risk of oversight and misdiagnoses. The overarching problem, therefore, lies in developing a reliable, efficient, and automated system capable of performing multi-label classifications on chest X-rays. Such a system should not only identify the presence of anomalies but accurately label multiple conditions simultaneously, ensuring comprehensive and precise patient diagnosis.

3.3 Work Implementation Environment

In the intricate world of medical image classification, the need for a well-structured, consistent, and efficient development environment is paramount. The chosen environment for this research was shaped by several contemporary tools and platforms, each serving a distinct purpose yet collectively ensuring the project's success. The following provides an outline of this environment:

3.3.1 Development and Training Environment

- **Python**: Serving as the foundational pillar of this research, Python was the primary programming language employed. Renowned for its simplicity and readability, Python boasts extensive libraries tailored for data science and machine learning, such as TensorFlow, PyTorch, and sci-kit-learn. Its widespread acceptance and open-source nature ensured up-to-date resources, community support, and continual enhancements vital for the project.
- **Jupyter Notebook**: An integral component of the Python ecosystem, Jupyter provides an interactive platform ideal for data manipulation, visualization, and algorithm development. The capacity to combine live code, equations, and visualizations made it instrumental in monitoring and documenting the iterative progress of the model training.
- **Kaggle Notebook**: Kaggle further extends the Python development environment with its cloud-based platform. Its in-built GPU capabilities ensured accelerated model training, while access to community-driven datasets and insights enriched the research process.

3.3.2 Deployment Environment

- **Visual Studio**: To transition from a research prototype to an operational model, a more encompassing IDE was required. Visual Studio, while supporting Python, offered a robust environment for deployment. Its array of tools for web services and API development ensured the trained models were seamlessly integrated into scalable solutions suitable for real-world scenarios.

Together, the harmonious blend of Python, Jupyter, Kaggle, and Visual Studio provided a comprehensive and agile work environment. This ensured not only the effective development and training of the multi-label classification models but also their successful deployment in a real-world context.

3.4 Information About Dataset

At the heart of this research, anchoring its methodologies and determining the scope of its findings, lies the dataset. A comprehensive examination of this foundation is therefore imperative. This research leans on the substantive NIH Chest X-Rays dataset for insights and training. The NIH Chest X-rays dataset is a public dataset readily available on Kaggle.

3.4.1 Overview of the Dataset

Spanning a massive size of roughly 45 GB, the NIH Chest X-rays dataset serves as a treasure trove of diagnostic images. This expansive collection houses images classified into 15 distinct categories:

1. Atelectasis
2. Cardiomegaly
3. Effusion
4. Infiltration
5. Mass
6. Nodule
7. Pneumonia
8. Pneumothorax
9. Consolidation
10. Edema

11. Emphysema
12. Fibrosis
13. Pleural Thickening
14. Hernia
15. No Findings

These categories encompass a broad spectrum of potential thoracic conditions.

3.4.2 Data Organization and Metadata

A cursory glance at the dataset reveals a plethora of images scattered across multiple folders. The organizational structure, while seemingly chaotic, holds layers of information waiting to be unravelled. Assisting researchers in this navigation is an accompanying CSV file, meticulously curated to provide context to the sea of images. This file catalogues each image with:

	Image Index	Finding Labels	Follow-up #	Patient ID	Patient Age	Patient Gender	View Position	OriginalImage[Width	Height]	OriginalImagePixelSpacing[x	y]
34173	00008951_006.png	No Finding	6	8951	56	M	PA	2500	2048	0.168	0.168
43540	00011237_004.png	No Finding	4	11237	55	F	PA	2978	2991	0.143	0.143
68580	00016934_031.png	No Finding	31	16934	56	M	AP	2500	2048	0.168	0.168

Figure 3.1: dataframe of the metadata csv

- The diagnostic class it belongs to.
- Non-identifying, generalized patient details ensuring utmost confidentiality.
- The specific orientation of the X-ray, offers insights into the imaging perspective.
- A file name, acting as the key to retrieve the image from its maze of folders.

3.4.3 Relevance and Implications of the Dataset

The sheer volume of the NIH Chest X-rays dataset is not its only commendable feature. The breadth of classes and the granularity of associated metadata position it as a rich resource for in-depth research. By plumbing the depths of this dataset, there lies an opportunity to sculpt machine learning models that can discern and classify nuanced variations in X-ray images with heightened precision. This dataset, thus, serves as both a foundation and a compass, guiding the trajectory of the research.

3.4.4 Limitations of the Dataset

While the NIH Chest X-Rays dataset offers lots of information and a diverse range of classes, it's essential to be aware of its limitations. One significant concern revolves around the labelling of the images. The dataset relies on Natural Language Processing (NLP) for label extraction, potentially introducing errors in classification. Although the NLP labelling mechanism boasts an impressive estimated accuracy exceeding 90%, it implies that up to 10% of the labels might contain inaccuracies. It is paramount to factor in these possible inaccuracies when interpreting results or deriving insights from the dataset, ensuring a balanced and realistic perspective on findings.

3.5 Proposed Approach

In this research, we embark on a journey to harness the capabilities of some of the most sophisticated neural network architectures, fine-tuning them for the specific task of chest X-ray image classification. Our approach strategically integrates the strengths of individual models and orchestrates them in harmony to achieve more robust and accurate predictions.

3.5.1 Transfer Learning

Transfer learning is a machine learning technique where a model developed for one task is reused as the starting point for a model on a second task. It's particularly useful in deep learning where training a large model from scratch requires a massive amount of data and computational resources.

Advantages of Transfer Learning

- **Insufficient Data:** Deep learning models often require vast amounts of labelled data to train from scratch. In real-world scenarios, collecting this amount of data is very difficult and resource extensive. Transfer learning allows us to take advantage of a pre-trained model which are generally trained on large datasets.
- **Computational Efficiency:** Training a deep learning model from scratch can be computationally intensive and time-consuming. Using pre-trained weights can significantly reduce training time.
- **Avoid Overfitting:** When training on a small dataset, there's a risk of overfitting. Starting with a pre-trained model can mitigate this risk, as the model has already learned robust features from a larger dataset.
- **Improved Performance:** Transfer learning often leads to better performance than training from scratch, especially when the source and target tasks are sim-

ilar. This is because the pre-trained model has already learned useful features from its source task, which can be beneficial for the target task.

S

3.5.2 Model Selection and Training

ViT Model

The Vision Transformer (ViT) has been a revolutionary model in computer vision since its introduction. Unlike traditional Convolutional Neural Networks (CNNs), which process images in a hierarchical and local manner, ViT uses the Transformer architecture, which has shown great success in natural language processing, to process images.

Here's a general explanation of how ViT works:

1. **Image Patching**:
   - An image is split into fixed-size patches (e.g., 16x16 pixels).
   - These patches are then linearly embedded into flat vectors. So, if an image is divided into N patches and each patch is embedded into a D dimensional vector, the image would be represented by a sequence of N vectors, each of size D.
2. **Position Embeddings**:
   - Since the Transformer architecture doesn't have any inherent sense of the order of input sequences (like CNNs have for images), position embeddings are added to provide the model with information about the position of each patch in the image.
   - These embeddings are learned during training.
3. **Transformer Encoder**:
   - The sequences of embedded patches are then fed into a series of Transformer encoder blocks.
   - The Transformer processes the patches collectively, allowing it to capture global interactions between patches.
   - Multi-head self-attention mechanisms inside the Transformer can focus on different parts of the image, ensuring that the model recognizes patterns that span over large regions of the image.
4. **Variants and Sizes**:

- There are various sizes and configurations of the Vision Transformer, with different numbers of layers, heads, and dimensions. For example: `ViT-BASE`, `ViT-LARGE`, etc.
- Different configurations offer trade-offs between computational cost and accuracy.

ResNet152 Model

ResNet, or Residual Network, is a type of convolutional neural network (CNN) designed to overcome the vanishing gradient problem. As the depth (number of layers) of the network increases, this problem becomes more pronounced. ResNet addresses this issue by introducing skip connections, or shortcut connections. These connections allow the gradients to flow more easily through the network. The "152" in ResNet-152 denotes the total number of layers in the architecture.

- **Initial Convolutional Layers:** The network begins with a convolutional layer, followed by a max-pooling layer.
- **Residual Blocks:** At the core of the architecture lie the bottleneck blocks, which are essentially groups of layers. Each of these blocks contains:
  - A 1x1 convolution.
  - A 3x3 convolution.
  - Another 1x1 convolution.

  These bottleneck blocks are repeated several times across the architecture.
- **Shortcut Connections:** A defining feature of ResNet is its use of shortcut connections. These connections skip over one or more layers and are subsequently added to the output of later layers. By doing so, they prevent the vanishing gradient problem and facilitate the creation of deeper networks.
- **Final Layers:** Post the residual blocks, there exists a global average pooling layer. This is followed by a fully connected layer that steers us to the final classification.

VGG19 Model

VGG19 is a variant of the VGG (Visual Geometry Group) model from the University of Oxford. The "19" in VGG19 refers to the number of weight layers in the network which includes 16 convolutional layers and 3 fully connected layers.

- **Convolutional Layers:**
  1. The architecture starts with two consecutive convolutional layers with 64 filters followed by a max-pooling layer.

2. Then, two consecutive convolutional layers with 128 filters followed by a max-pooling layer.
3. Four consecutive convolutional layers with 256 filters followed by a max-pooling layer.
4. Four consecutive convolutional layers with 512 filters followed by a max-pooling layer.
5. And lastly, four consecutive convolutional layers with 512 filters again followed by a max-pooling layer.

- **Fully Connected Layers:** After the convolutional layers, there are three fully connected layers. The first two have 4096 nodes each, and the final layer is the softmax classification layer.
- **Activation Function:** ReLU (Rectified Linear Unit) is used as the activation function in all layers except the final softmax layer.

Xception Model

The Xception architecture, standing for "Extreme Inception," diverges from traditional convolutional networks in its approach to convolution. The core idea behind Xception is the implementation of depthwise separable convolutions as opposed to standard convolutions.

- **Depthwise Separable Convolution:** This type of convolution decomposes the convolution operation into two stages:
  - **Depthwise Convolution:** A spatial convolution is performed independently over each channel of the input. For each channel in the input data, a distinct convolutional filter is used.
  - **Pointwise Convolution:** Following the depthwise convolution, a 1x1 convolution is employed. This amalgamates the outputs from the depthwise convolution across channels.
- **Architecture:**
  - **Entry Flow:**
    - Starts with a standard convolution operation (Conv2D) succeeded by batch normalization and a ReLU activation.
    - Subsequently, a sequence of depthwise separable convolution blocks is utilized. Each block consists of:
      - Depthwise separable convolution.
      - Batch normalization.
      - ReLU activation.
      - Max pooling with a stride, effectively reducing the spatial dimensions.

  - **Middle Flow:**
    * This portion of the model is constituted of eight repetitive blocks.
    * Each block contains:
      · Depthwise separable convolution.
      · Batch normalization.
      · ReLU activation.
    * The output of each block is routed into the subsequent one without any reduction in spatial dimensions.
  - **Exit Flow:**
    * Initiates with a depthwise separable convolution block, akin to the blocks in the entry and middle flows.
    * Followed by a max pooling layer.
    * Culminates with a global average pooling layer, producing feature vectors that are then channelled to the final classification layer (absent in models where `include_top=False`).

- This structure empowers the Xception model to adeptly process spatial and channel-wise details distinctly, potentially enabling more detailed feature representations with a minimized parameter count.

Each of these models starts with their respective pre-trained weights. They undergo fine-tuning, wherein their output layers are adjusted to cater to the specific categories present in our dataset. The primary training involves feeding our dataset into these models and calibrating their weights for optimal performance on the task of chest X-ray classification.

After the individual training phase, we venture into an advanced ensemble technique – stacking. The predictions from these models for the validation set are combined (stacked horizontally) and used as input to train a meta-model. This ensemble technique aims to harmonize the decision boundaries from individual models, thereby aiming to capitalize on their combined strengths and mitigate individual weaknesses.

3.5.3 Rationale and Advantages of Selected Models

- **ViT (Vision Transformer)**:
  - **Rationale**: Transformers have brought about revolutionary advancements in natural language processing. Vision Transformer, applying these principles to visual data, treats images as sequences of patches, akin to textual sequences.
  - **Advantages**:
    * Attention Mechanisms: Focuses on intricate details in the image.
    * Scalability: Efficient on large datasets.

    - Less Reliance on Data Augmentation: Reduced need for traditional augmentation techniques.

- **ResNet152**:
  - **Rationale**: Mitigates the challenges of deep networks, particularly the vanishing gradient problem, through residual connections.
  - **Advantages**:
    - Deep Architecture: Identifies diverse image features.
    - Residual Connections: Facilitates effective learning across layers.
    - Widely Recognized: A trusted choice for image classification.
- **VGG19**:
  - **Rationale**: Uniform architecture that's shown consistent performance across visual tasks.
  - **Advantages**:
    - Uniformity: Computationally efficient due to 3x3 filters.
    - Transfer Learning: Fine-tuning pre-trained models accelerates training.
    - Receptive Field: Captures global image features.
- **Xception**:
  - **Rationale**: An evolved model using depthwise separable convolutions for enhanced efficiency.
  - **Advantages**:
    - Efficiency: Reduced parameters without performance compromise.
    - Parallel Processing: Channel-wise processing speeds up computations.
    - Adaptable: Modifiable, modular architecture.

The ensemble of these architectures aims to strike a balance between performance, computational efficiency, and innovative design principles. By converging the predictions of these diverse architectures, the objective is to capture a comprehensive insight into the data, potentially counterbalancing the limitations of one model with the strengths of another Büyükçakir et al. (2018).

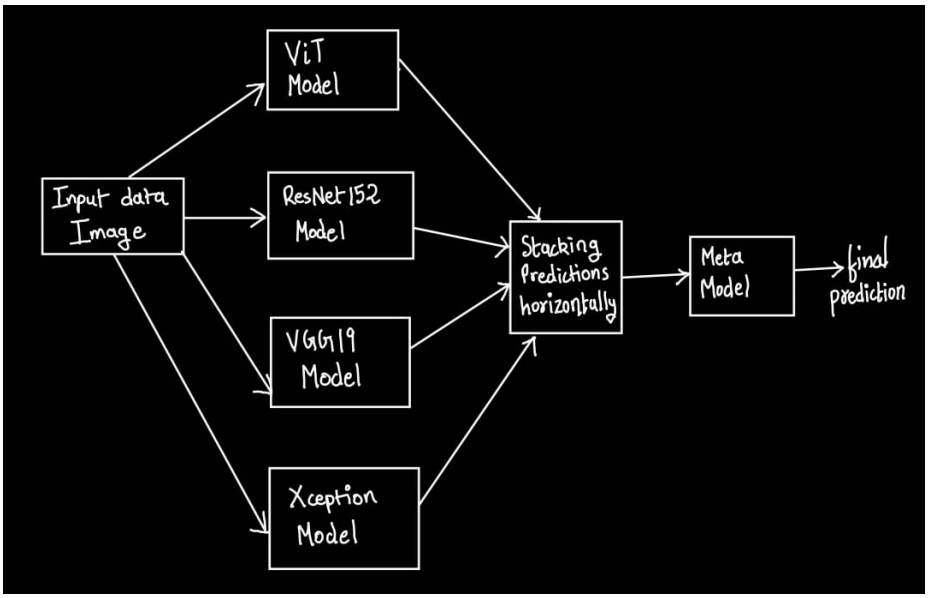

Figure 3.2: hybrid architecture and proposed approach

3.5.4 Deployment

Upon successful training and validation, the models, alongside the meta-model, are integrated into a web application. For this purpose, we employ FastAPI and Streamlit, providing an interactive interface for users to upload chest X-ray images and receive predictions. The entire application is containerized using Docker, ensuring easy scalability and deployment.

Advantages of Deployment Tools

FastAPI:

- *Fast Performance*: Notably faster compared to other Python frameworks.
- *Type Checking*: Utilizes Python type hints for data validation.
- *Automatic API Documentation*: Interactive documentation generation.
- *Asynchronous Support*: Enables concurrent request handling.
- *Easy to Use*: Intuitive, especially for those acquainted with Flask or Express.

Streamlit:

- *Rapid Prototyping*: Swift transformation from data scripts to web apps.
- *Data Integration*: Seamless integration with popular data science libraries.

- *Customizable*: Extendable with advanced features.
- *Widgets*: A variety of widgets for interactive data exploration.
- *Data Caching*: Built-in feature to avoid repeated data operations.

Docker:

- *Consistency*: Ensures uniformity across various environments.
- *Isolation*: Provides encapsulated environments for apps.
- *Portability*: Guarantees the application runs consistently on any Docker-installed machine.
- *Microservices Architecture*: Perfectly suited for this architectural style.
- *Resource Efficiency*: Lightweight due to shared host kernel.
- *Version Control for Containers*: In-built container versioning system.

The combination of these technologies facilitates fluid and straightforward deployment and version control, enhancing the web application's performance.

Chapter 4

Implementation

This chapter examines the implementation of the project.

4.1 Visualization And Data Analysis

The dataset used here has multiple labels consisting of 14 diseases and No findings This is the value count found in the dataset for each class

```
print(label_counts)
```

```
No Finding                 60361
Infiltration                9547
Atelectasis                 4215
Effusion                    3955
Nodule                      2705
Pneumothorax                2194
Mass                        2139
Effusion|Infiltration       1603
Atelectasis|Infiltration    1350
Consolidation               1310
Atelectasis|Effusion        1165
Pleural_Thickening          1126
Cardiomegaly                1093
Emphysema                    892
Infiltration|Nodule          829
Name: Finding Labels, dtype: int64
```

Figure 4.1: value counts of the classes

4.2 Loading And Cleaning Dataset

In any data-driven methodology, ensuring the dataset is clean, consistent, and reliable paves the way for successful outcomes. Given the vast nature of the NIH Chest X-rays dataset, meticulous preparation was essential to ensure it was ready for effective model training and validation.

1. **Replacing "No Finding" Labels**: The dataset had instances marked as "No Finding" to indicate the absence of any diagnostic issues. For better alignment with our sparse binary representation approach, these labels were replaced with an empty string. Consequently, they can be represented as 00000000000000 in binary, offering a streamlined categorization.
2. **Filtering Classes by Count**: To avoid undue class imbalances and ensure a robust learning environment for our models, only those classes with a minimum count of 1200 instances were retained. This approach, while selective, ensures that the models have sufficient data to understand and distinguish between different diagnostic categories.

After these preprocessing steps, the dataset was distilled down to 14 classes, providing a focused and comprehensive scope for the subsequent machine learning processes.

```
In [5]: all_xray_df['Finding Labels'] = all_xray_df['Finding Labels'].map(lambda x: x.replace('No Finding', ''))
        from itertools import chain
        all_labels = np.unique(list(chain(*all_xray_df['Finding Labels'].map(lambda x: x.split('|')).tolist())))
        all_labels = [x for x in all_labels if len(x)>0]
        print('All Labels ({}): {}'.format(len(all_labels), all_labels))
        for c_label in all_labels:
            if len(c_label)>1: # Leave out empty labels
                all_xray_df[c_label] = all_xray_df['Finding Labels'].map(lambda finding: 1.0 if c_label in finding else 0)
        all_xray_df.sample(3)

        All Labels (14): ['Atelectasis', 'Cardiomegaly', 'Consolidation', 'Edema', 'Effusion', 'Emphysema', 'Fibrosis', 'Hernia', 'Infi
        ltration', 'Mass', 'Nodule', 'Pleural_Thickening', 'Pneumonia', 'Pneumothorax']
```

Figure 4.2: data cleaning code snippet

4.3 Data Balancing

Medical datasets, inherent to their domain, frequently exhibit an imbalance in class distribution. Such imbalances arise because certain conditions or diseases manifest less frequently than others. When machine learning models trained on these imbalanced datasets, they may produce biased predictions, predominantly favouring the more abundant class. This skew is particularly problematic in the medical realm, where an erroneous classification can have dire repercussions.

To mitigate this inherent class disproportion, it becomes paramount to achieve a balanced data distribution. Balancing aids model training and augments predictive accuracy. A prevalent approach to rectify this imbalance is through weighted sampling, which amplifies the likelihood of under-represented classes being chosen during sampling Sharma et al. (2022).

```
In [9]: sample_weights = all_xray_df['Finding Labels'].map(lambda x: len(x.split('|')) if len(x)>0 else 0).values + 4e-2

        sample_weights /= sample_weights.sum()

        all_xray_df = all_xray_df.sample(40000, weights=sample_weights)
```

Figure 4.3: data balancing code snippet

The provided code segment elucidates this balancing process:

1. **Computing Sample Weights**: The 'Finding Labels' column from the dataframe is processed. For each entry, the code discerns the count of individual findings. In instances devoid of findings, a count of zero is recorded. Subsequently, a minute value (0.04) is appended to every weight, safeguarding that even labels with no counts retain a slight chance of inclusion. The cumulative weights are then normalized to ensure their sum equates to one. This step primes the weights for their role in the subsequent sampling function.

2. **Weighted Sampling**: Using the generated weights, 40,000 rows are selectively sampled from the data frame. Entries with augmented weights witness an elevated selection likelihood, ensuring the final sample provides an enhanced representation of previously under-represented classes.

Through the application of the above strategy, the initial dataset, skewed in its class distribution, undergoes a transformation into a more balanced avatar. This harmonized dataset now stands poised to facilitate further explorations, training, and validations, fortifying the promise of optimal model efficacy.

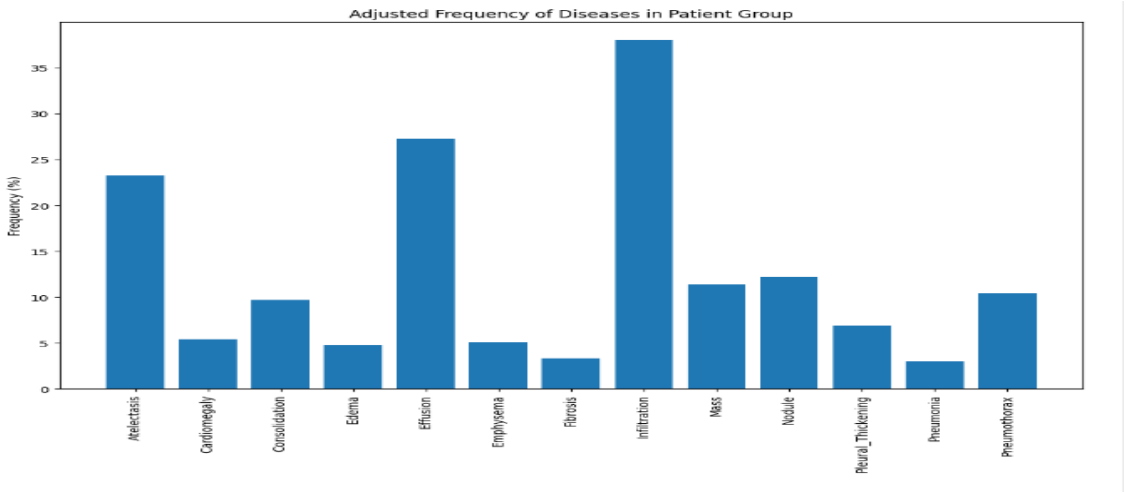

Figure 4.4: bar graph after data balancing

4.4 Dataset Splitting

In the realm of machine learning, it's a general best practice to partition available data into distinct sets for specific purposes, ensuring the robustness and generalizability of the model:

1. **Training Set**: This set is pivotal for training the model.
2. **Validation Set**: Beyond its traditional use for tuning and hyperparameters adjustment, in this study, the validation set also plays a dual role. It will be utilized to train the meta-model during the ensemble stacking process, ensuring that the meta-model learns how to optimally combine predictions from the base models.
3. **Test Set**: It serves as an unbiased evaluator, gauging the model's performance on new, unseen data.

The provided code achieves this three-way split employing the `train_test_split` function from `sklearn.model_selection`. Here's its detailed breakdown:

1. **Train-Temp Split**:
   - The entire dataset (`all_xray_df`) is bifurcated into two parts: a training set (`train_df`) and a temporary set (`temp_df`).
   - 70% of the data forms the training backbone, and the subsequent 30% is relegated to the temporary set.
   - The split is deterministic, anchored by the `random_state` parameter set at 2018.

```
In [24]: from sklearn.model_selection import train_test_split

         # Split into train and temporary set (70% train + 30% temp)
         train_df, temp_df = train_test_split(all_xray_df,
                                              test_size=0.3,
                                              random_state=2018,
                                              stratify=all_xray_df['Finding Labels'].map(lambda x: x[:4]))

         # Further split temp set into validation and test (20% validation + 10% test)
         valid_df, test_df = train_test_split(temp_df,
                                              test_size=1/3,
                                              random_state=2018,
                                              stratify=temp_df['Finding Labels'].map(lambda x: x[:4]))

         print('train', train_df.shape[0], 'validation', valid_df.shape[0], 'test', test_df.shape[0])

         train 28000 validation 8000 test 4000
```

Figure 4.5: code snippet for splitting dataset

- Furthermore, the `stratify` parameter ensures the 'Finding Labels' distribution remains consistent between the two datasets. To this end, the stratification utilizes the first four characters of each label (`x[:4]`).

2. **Validation-Test Split**:
   - The temporary reservoir (`temp_df`) undergoes another split, yielding a validation set (`valid_df`) and a test set (`test_df`).
   - The validation set corners 20% of the original data, leaving 10% for the test set.
   - `random_state` once again ensures the replicability of the splits, while stratification, guided by the initial quartet of characters from the 'Finding Labels', retains class distribution.

By meticulously partitioning the data this way, you're not only set for primary model training but also perfectly poised for the ensembling process. The stratified sampling ensures a balanced representation of classes across all datasets, an essential detail for an unskewed evaluation of the model's prowess.

4.5 Preparing Train, Validation, and Test Data

4.5.1 Image Preprocessing

Custom Data Generator and the Essence of CLAHE

The process initiates with the development of a *CustomImageDataGenerator* that functions as an extension to Keras's foundational *ImageDataGenerator*. The primary enhancement offered by this custom generator is the seamless integration of the **CLAHE** algorithm.

```
In [25]: import cv2
         import numpy as np
         from keras.preprocessing.image import ImageDataGenerator

         # Define a custom data generator with CLAHE
         class CustomImageDataGenerator(ImageDataGenerator):
             def __init__(self, clahe_clip_limit=3.0, **kwargs):
                 super().__init__(**kwargs)
                 self.clahe_clip_limit = clahe_clip_limit

             def clahe_augment(self, image):
                 clahe = cv2.createCLAHE(clipLimit=self.clahe_clip_limit)
                 return clahe.apply(image)

             def random_transform(self, x, seed=None):
                 x = self.clahe_augment(x)
                 return super().random_transform(x, seed)
```

Figure 4.6: custom data generator with CLAHE

CLAHE (Contrast Limited Adaptive Histogram Equalization) is a sophisticated variant of the conventional histogram equalization method, primarily employed to enhance the contrast of an image. Unlike traditional methods that operate over the entire image, CLAHE functions on localized sections. The image is partitioned into smaller tiles, with each segment undergoing individual histogram equalization. This localized approach is immensely beneficial, especially for images like medical scans where minute details are paramount.

The advantages of CLAHE encompass:

1. *Detail Magnification*: By design, CLAHE emphasizes localized features within images. In the realm of medical imagery, this can elucidate intricate structures that might otherwise remain subdued.
2. *Noise Counteraction*: Traditional histogram equalization, often amplifies noise inadvertently. In contrast, CLAHE mitigates this, ensuring clearer images.
3. *Adaptability*: With a configurable 'clip limit', users can modulate the equalization degree, making CLAHE adaptable to diverse requirements.

The *clahe_augment* function embodies the process of applying CLAHE to an image. Integrated into the generator's pipeline, it ensures the consistent enhancement of every processed image.

4.5.2 Data Augmentation

Data augmentation lies at the core of efficient deep learning. By introducing controlled variations to training images, models encounter a richer dataset. This not only deepens the model's learning but also fortifies its resistance to overfitting.

```
In [26]: from keras.preprocessing.image import ImageDataGenerator
         IMG_SIZE = (256, 256)
         train_datagen = CustomImageDataGenerator(samplewise_center=True,
                                                  samplewise_std_normalization=True,
                                                  horizontal_flip = True,
                                                  vertical_flip = False,
                                                  height_shift_range= 0.05,
                                                  width_shift_range=0.1,
                                                  rotation_range=5,
                                                  shear_range = 0.1,
                                                  fill_mode = 'reflect',
                                                  zoom_range=0.15)
```

Figure 4.7: data augmentation

A closer look at the augmentations employed:

1. *Samplewise Centering and Normalization*: It accelerates training convergence and reduces erratic behaviour potential in certain activations.
2. *Horizontal Flip*: It trains the model to recognize features regardless of their orientation.
3. *Height and Width Shift*: Enables feature recognition even if they're positioned slightly differently.
4. *Rotation*: Equips the model to handle real-world images that might be tilted.
5. *Shear*: Prepares the model for potential distortions.
6. *Reflect Fill Mode*: Ensures 'empty' image sections mirror adjacent pixel values.
7. *Zoom*: The model learns to discern features irrespective of their frame size.

For the *train_datagen*, these augmentations act as a training regimen. Conversely, the *val_datagen* and *test_datagen* are unaltered, ensuring a genuine assessment of the model's performance.

4.5.3 Splitting Labels

The code's intention is straightforward: it aims to break the concatenated string labels present in the column 'Finding Labels' into a list of distinct labels. This transformation is especially necessary for compatibility with the "flow from dataframe" functionality of Keras, which requires categorical labels to be present in a specific format for streamlined data ingestion and preprocessing.

1. **Defining the `split_labels` Function**: This function is designed to take a row of the dataframe and segregate the compound labels found in the 'Finding Labels' column. The labels are concatenated with a pipe ('—') separator, and this function essentially splits them into a list.

```
def split_labels(labels):
    list1 = []
    for i in labels['Finding Labels'].split('|'):
        if i != "":
            list1.append(i)
    return list1
```

```
valid_df['newLabel'] = valid_df.apply(split_labels, axis=1)
train_df['newLabel'] = train_df.apply(split_labels, axis=1)
test_df['newLabel'] = test_df.apply(split_labels, axis=1)
```

Figure 4.8: label splitting code snippet

- `labels['Finding Labels'].split('|')`: This splits the string at each pipe character, resulting in a list of labels.
- The condition `if i != ""` ensures that empty strings, which might result from trailing or leading pipes, aren't added to the list.
- The resulting list of distinct labels is then returned.

2. **Applying the Function to DataFrames**: The next step involves applying this function to the entire dataframe for each set (validation, training, and test). This is accomplished with the `apply` method in pandas, with `axis=1` ensuring the function operates row-wise.

   - `valid_df['newLabel'] = valid_df.apply(split_labels, axis=1)`: This line creates a new column, 'newLabel', in the validation dataframe. Each entry in this column contains the list of labels corresponding to that row's 'Finding Labels' column.

4.6 Data Generators: Using `flow_from_dataframe`

The `flow_from_dataframe` function is a method of Keras' `ImageDataGenerator` class. It's designed to read images directly from a pandas dataframe, streamlining the process of feeding image data into a neural network. This method is especially useful when working with datasets where metadata, including file paths and labels, is stored in a structured format like a dataframe.

4.6.1 Using `flow_from_dataframe`

1. **Dataframe Parameter**: `dataframe=train_df` specifies the source dataframe from which the images and labels will be read. In this instance, it's `train_df`, which contains information about the training dataset.

```
val_generator = val_datagen.flow_from_dataframe(
dataframe=valid_df,
directory= None,
x_col="path",
y_col="newLabel",
#subset="training",
batch_size=32,
seed=42,
shuffle=False,
class_mode='categorical',
color_mode ='rgb',
target_size=(256,256))
```

```
Found 8000 validated image filenames belonging to 14 classes.
```

Figure 4.9: data generator,flow from dataframe

2. **Directory**: Given `directory=None`, since we are providing absolute paths in the dataframe, there's no need to specify a directory. If only filenames were present in the dataframe, this argument would be used to provide the common directory path.
3. **Image Paths**: `x_col="path"` points to the column in the dataframe which contains the paths to the images.
4. **Labels**: `y_col="newLabel"` indicates the column where the (previously split) labels are stored.
5. **Batch Size**: `batch_size=32` denotes the number of images that will be read, processed, and returned in each batch when the generator is iterated.
6. **Random Seed**: `seed=42` is set for reproducibility. This ensures that random operations (like shuffling) will produce consistent results every time.
7. **Shuffling Data**: With `shuffle=True`, the training dataset will be shuffled before being fed into the model, aiding in better generalization during training.
8. **Class Mode**: `class_mode='categorical'` specifies that the labels are categorical. Multiple classes are expected, and internally, the labels will be one-hot encoded.
9. **Color Mode**: `color_mode='rgb'` indicates that images will be read in RGB (Red, Green, Blue) color mode. This implies that images will have three channels.
10. **Target Size**: `target_size=(256,256)` argument resizes every image to 256x256 pixels. Resizing is crucial for neural networks, as they expect all input data to have a consistent shape.

4.6.2 Validation and Test Generators

The generators for validation and test datasets are crafted in a similar fashion, with a notable distinction: `shuffle=False`. This ensures data remains consistent in order, which is particularly pertinent during the evaluation phase, guaranteeing that labels and predictions align properly.

Using such data generators in Keras provides an efficient way of handling voluminous datasets. Images are read in batches, implying only a minimal fraction of the dataset is loaded into memory concurrently, thus making it feasible to work with expansive image datasets on conventional hardware Malialis et al. (2022).

4.6.3 Visualization of Train Generator

The objective in this section is to visually validate the images and labels being processed by the `train_generator`. This step ensures that the image and label pairs are correctly matched and undergo intended preprocessing.

1. **Fetching Data**: A batch of images (`t_x`) and their corresponding labels (`t_y`) are obtained using the command `next(train_generator)`.
2. **Plot Setup**: A plotting grid of size 4x4 is set up with the command `plt.subplots(4, 4, figsize = (16, 16))` to visually inspect a subset of the batch.
3. **Displaying Images & Labels**: A loop is used to iterate over the images, labels, and individual plots. Each image is displayed with properties mimicking X-ray visuals. The labels (with scores greater than 0.5) are then set as the titles for the respective images. Additionally, for clarity, the axes labels and ticks are disabled.

This visualization serves as a confirmation of the correct operation of the data loading, preprocessing, and labelling mechanisms.

```
t_x, t_y = next(train_generator)
fig, m_axs = plt.subplots(4, 4, figsize = (16, 16))
for (c_x, c_y, c_ax) in zip(t_x, t_y, m_axs.flatten()):
    c_ax.imshow(c_x[:,:,0], cmap = 'bone', vmin = -1.5, vmax = 1.5)
    c_ax.set_title(', '.join([n_class for n_class, n_score in zip(all_labels, c_y)
                              if n_score>0.5]))
    c_ax.axis('off')
```

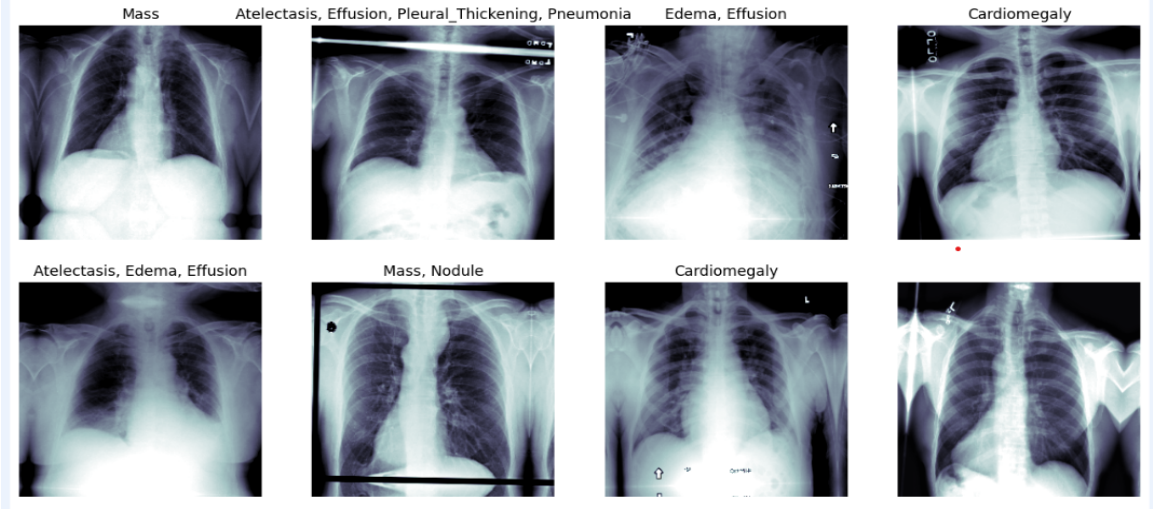

Figure 4.10: data generator,visualization

4.7 Training ViT Model

The primary advantage of ViT is its ability to look at global interactions between different parts of the image, as opposed to CNNs that look at local interactions and gradually build up a global understanding through a hierarchy of layers. The self-attention mechanism in Transformers allows each patch to attend to all other patches, helping in capturing patterns that are spatially distant.

Here are the specifics of the implementation:

1. **Pre-processing Layers**:
   - `Resizing (224, 224)`: Resizes input images to 224x224 pixels, which is the typical input size for many pre-trained models.
   - `Rescaling (1./255)`: Scales the pixel values of images between 0 and 1. Neural networks generally converge faster and more reliably on normalized data.
   - `Permute ((3,1,2))`: Rearranges the dimensions of the tensor. This is mainly done to match the expected input format of the pre-trained model.
2. **Loading the Pretrained ViT Model**:
   - The code loads a pretrained Vision Transformer model (`vit-base-patch16-224-in21k`) using the `transformers` library.

```
import tensorflow as tf
from tensorflow.keras.layers import Resizing, Rescaling, Permute
resize_rescale_hf = tf.keras.Sequential ( [
    Resizing (224, 224),
    Rescaling (1./255),
    Permute ((3,1,2))
])
```

```
from tensorflow.keras.layers import Input,Flatten,BatchNormalization, Dropout
from tensorflow.keras.layers import Dense
from transformers import AutoImageProcessor, TFViTModel

vit_model = TFViTModel.from_pretrained("google/vit-base-patch16-224-in21k")
vit_model.trainable = False
inputs = Input(shape = (256,256,3))
x= resize_rescale_hf(inputs)
x = vit_model.vit(x)[0]#[:,0,:]
x = tf.keras.layers.MaxPooling1D()(x)[:,0,:]
x = Flatten()(x)
x = Dense(512,activation = 'relu')(x)
output = Dense(14,activation='sigmoid')(x)  #ATTACH NEW CLASSIFICATION HEAD
hf_model = tf.keras.Model(inputs = inputs,outputs = output)
```

Figure 4.11: Training ViT Model

- `vit_model.trainable = False` ensures that the weights of this pre-trained model remain fixed during training, so only the attached custom layers will be updated.

3. **Custom Neural Network Architecture**:
   - The custom model takes images of shape `(256,256,3)` as inputs.
   - After resizing and rescaling, these images are passed to the ViT model.
   - `tf.keras.layers.MaxPooling1D()`: Given the output sequence from the transformer, the max pooling layer in 1D is applied. It reduces the spatial size (length of the sequence) by taking the maximum value over a window. In this context, it helps reduce the dimensionality and retain significant features.
   - After max pooling, the sequence is flattened to be fed into dense layers.
   - The next dense layer with 512 neurons and ReLU activation functions as an intermediary layer for feature extraction.
   - The final output layer has 14 neurons (corresponding to 14 classes) and uses the sigmoid activation function, suitable for multi-label classification.
4. **Compiling the Model**:
   - The model is compiled using the Adam optimizer and binary cross-entropy loss, ideal for binary and multi-label classification tasks.

```
hf_model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
                 metrics = ['binary_accuracy', 'mae'])
```

Figure 4.12: compiling ViT Model

- Metrics `binary_accuracy` and `mae` (Mean Absolute Error) is used to evaluate the model's performance during training.

5. **Training Callbacks**:

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint_callback = ModelCheckpoint(
    filepath="/kaggle/working/best_weights_vit.h5",  # Replace with th
    monitor='val_binary_accuracy',
    save_best_only=True,
    save_weights_only=True,
    mode='max',
    verbose=1
)
```

```
early_stopping_callback = EarlyStopping(
    monitor='val_binary_accuracy',
    patience=2,  # Number of epochs with no improvement before stoppir
    restore_best_weights=True,
    verbose=1
)
```

Figure 4.13: compiling ViT Model

- **ModelCheckpoint**: Saves the weights of the model with the best validation binary accuracy. It ensures that even if the training process gets interrupted or if the model starts overfitting in later epochs, we retain the best-performing version.
- **EarlyStopping**: Monitors the validation binary accuracy and will stop training if it doesn't improve for 2 consecutive epochs. It also restores the weights of the epoch with the best validation binary accuracy.

6. **Model Training**:

- The model is trained using the `fit` method for 10 epochs.
- Training and validation data are provided via the data generators.
- `steps_per_epoch` and `validation_steps` determine the number of batches per epoch for training and validation, respectively.

```
history = hf_model.fit(
train_generator,
steps_per_epoch=STEP_SIZE_TRAIN,
validation_steps=STEP_SIZE_VALID,
validation_data = val_generator,
epochs = 10,
verbose = 1,
callbacks = [checkpoint_callback,early_stopping_callback]
)
```

Figure 4.14: fitting ViT Model

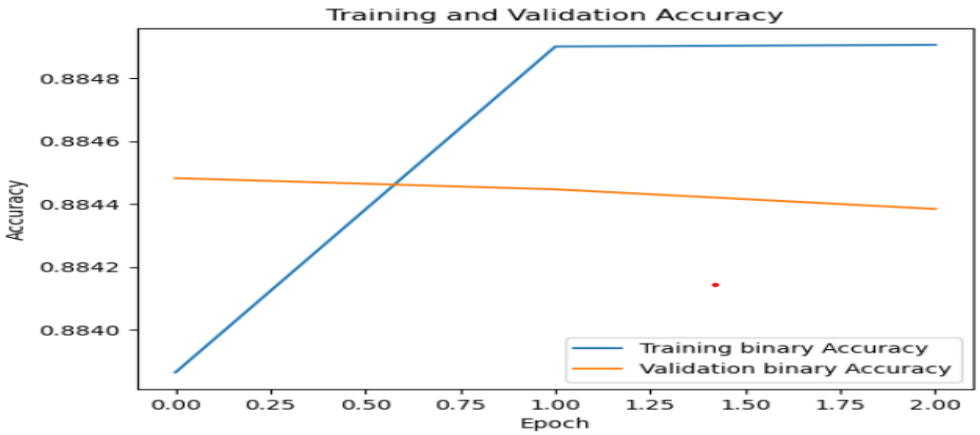

Figure 4.15: ViT train, val binary accuracy graph

4.8 Training ResNet152 Model

The provided code outlines the procedure to train a ResNet152 model, a deep convolutional neural network known for its performance in image recognition tasks.

1. **Preprocessing Layer**:
   - The code begins by defining a sequence of operations to resize and rescale images.
   - `Resizing(224, 224)`: Images are resized to 224x224 pixels, which is the standard input size for ResNet152.
   - `Rescaling(1./255)`: Images are rescaled to a [0, 1] range by dividing each pixel value by 255.
2. **Model Preparation**:
   - TensorFlow and necessary layers and applications are imported.

- The pretrained `ResNet152` model is loaded with weights pre-trained on ImageNet.
- The layers of this pre-trained model are frozen to prevent them from being updated during training. This is typically done to utilize the learned features without overfitting the new dataset.

3. **Extending the Model**:
   - New layers are added on top of the pre-trained ResNet152 to make it suitable for the specific task. This includes:
     - An input layer to accept images of size 256x256 with 3 channels.
     - The preprocessing layers to resize and rescale.
     - Flattening the output to create a 1D array.
     - A dense layer with 512 neurons and ReLU activation function to learn complex patterns.
     - A final dense layer with 14 neurons and sigmoid activation function, presumably to predict 14 different classes.

```
from tensorflow.keras.layers import Resizing, Rescaling, Permute
resize_rescale_hf_without_transpose = tf.keras.Sequential ( [
    Resizing (224, 224),
    Rescaling (1./255),
])
```

```
import tensorflow as tf
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.applications import ResNet152
from tensorflow.keras.models import Model

# Load the pretrained ResNet152 model
pretrained_model = ResNet152(
    include_top=False,
    input_shape=(224, 224, 3),
    weights='imagenet'
)

# Freeze layers of the pretrained model
for layer in pretrained_model.layers:
    layer.trainable = False

# Add layers for classification on top of ResNet152
inputs = Input(shape = (256,256,3))
x= resize_rescale_hf_without_transpose(inputs)
x = pretrained_model(x)
x = Flatten()(x)
x = Dense(512, activation='relu')(x)
predictions = Dense(14, activation='sigmoid')(x)

# Create the final model
resnet152_model = Model(inputs=inputs, outputs=predictions)
```

Figure 4.16: Training ResNet152 Model

4. **Model Compilation**:
   - The final model is compiled with the Adam optimizer and binary cross-entropy loss. This suggests a binary classification task for each of the 14 classes.
   - The metrics used are binary accuracy and mean absolute error (MAE).

```
resnet152_model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
                        metrics = ['binary_accuracy', 'mae'])
```

Figure 4.17: Compiling ResNet152 Model

5. **Callbacks for Model Training**:

   - `ModelCheckpoint`: This callback saves the model weights when the validation binary accuracy improves. The best weights are saved to a file named "best_weights_resnet.h5".
   - `EarlyStopping`: If the validation binary accuracy doesn't improve for 2 consecutive epochs, the training is stopped, and the best weights are restored. This prevents overfitting and reduces training time.

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint_callback = ModelCheckpoint(
    filepath="/kaggle/working/best_weights_resnet.h5",  # Replace with
    monitor='val_binary_accuracy',
    save_best_only=True,
    save_weights_only=True,
    mode='max',
    verbose=1
)
```

```
early_stopping_callback = EarlyStopping(
    monitor='val_binary_accuracy',
    patience=2,  # Number of epochs with no improvement before stoppin
    restore_best_weights=True,
    verbose=1
)
```

Figure 4.18: ResNet152 Callbacks

6. **Model Training**:

   - The model is trained using the `fit` method.
   - Data is provided via `train_generator` and `val_generator`, which are presumably data generators that provide batches of training and validation data respectively.
   - Training will run for a maximum of 10 epochs, but may stop earlier due to the EarlyStopping callback.

- The checkpoint and early stopping are provided as callbacks to monitor and enhance the training process.

```
history = resnet152_model.fit(
train_generator,
#batch_size = 32,
steps_per_epoch=STEP_SIZE_TRAIN,
validation_steps=STEP_SIZE_VALID,
validation_data = val_generator,
epochs = 10,
verbose = 1,
callbacks = [checkpoint_callback,early_stopping_callback]
#class_weight = class_weights,
#callbacks = [checkpoint_callback]
)
```

Figure 4.19: fitting on ResNet152

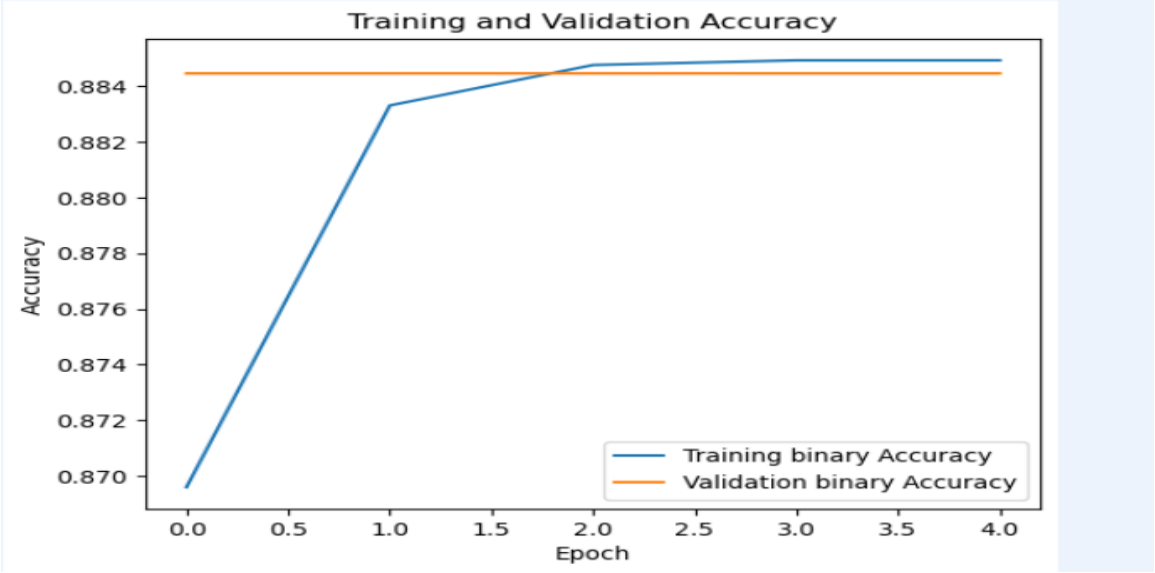

Figure 4.20: ResNet152 Train, Val binary accuracy graph

4.9 Training VGG19 Model

In this section, the VGG19 model, a renowned deep convolutional neural network designed primarily for image classification tasks, is fine-tuned using transfer learning.

```
from tensorflow.keras.layers import Resizing, Rescaling, Permute
import tensorflow as tf
resize_rescale_hf_without_transpose = tf.keras.Sequential ( [
    Resizing (224, 224),
    Rescaling (1./255),
])
```

```
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.applications import VGG19
from tensorflow.keras.models import Model

# Load the pretrained VGG19 model
pretrained_model = VGG19(
    include_top=False,
    input_shape=(224, 224, 3),
    weights='imagenet'
)

# Freeze layers of the pretrained model
for layer in pretrained_model.layers:
    layer.trainable = False

# Add layers for classification on top of VGG19
inputs = Input(shape = (256,256,3))
x= resize_rescale_hf_without_transpose(inputs)
x = pretrained_model(x)
x = Flatten()(x)
x = Dense(512, activation='relu')(x)
predictions = Dense(14, activation='sigmoid')(x)

# Create the final model
vgg19_model = Model(inputs=inputs, outputs=predictions)

# Print model summary
vgg19_model.summary()
```

Figure 4.21: vgg19 model training

1. **Preprocessing**:
   - A sequence of preprocessing operations is defined using TensorFlow's Keras API. This sequence resizes the input images to 224×224 pixels and rescales the pixel values to lie in the range [0, 1].
   - `Resizing(224, 224)`: Resizes images to match the standard input size for VGG19.
   - `Rescaling(1./255)`: Normalizes the pixel values to the [0, 1] range.
2. **Loading the pretrained VGG19 Model**:

- The pre-trained VGG19 model, trained on the ImageNet dataset, is loaded without the top classification layer.
- Its layers are then frozen to ensure the pre-trained weights remain unchanged during further training. This step is essential in transfer learning to preserve the knowledge acquired on the previous dataset.

3. **Extending the Model**:
   - An input layer designed to accept images of size $256 \times 256 \times 3$ is defined.
   - The preprocessing sequence (resizing and rescaling) is applied to this input.
   - The output from these preprocessing steps is then fed into the pre-trained VGG19 model.
   - The output of the VGG19 model is flattened into a one-dimensional vector, followed by a dense layer with 512 neurons using the ReLU activation function.
   - Finally, a dense layer with 14 neurons and a sigmoid activation function is added, suggesting the model aims to predict 14 different classes.
4. **Model Compilation**:

```
vgg19_model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
                            metrics = ['binary_accuracy', 'mae'])
```

Figure 4.22: vgg19 model compiling

- The extended VGG19 model is compiled using the Adam optimizer and binary cross-entropy loss. This implies a binary classification task for each of the 14 classes.
- The metrics used to evaluate the model's performance are binary accuracy and mean absolute error (MAE).

5. **Callbacks**:

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint_callback = ModelCheckpoint(
    filepath="/kaggle/working/best_weights_vgg19.h5",  # Replace with t
    monitor='val_binary_accuracy',
    save_best_only=True,
    save_weights_only=True,
    mode='max',
    verbose=1
)
```

```
early_stopping_callback = EarlyStopping(
    monitor='val_binary_accuracy',
    patience=4,  # Number of epochs with no improvement before stopping
    restore_best_weights=True,
    verbose=1
)
```

Figure 4.23: vgg19 model callbacks

- A `ModelCheckpoint` callback is defined to save the model weights whenever there's an improvement in validation binary accuracy. The best weights are saved to a file named `"best_weights_vgg19.h5"`.

6. **Model Training**:

```
history = vgg19_model.fit(
train_generator,
#batch_size = 32,
steps_per_epoch=STEP_SIZE_TRAIN,
validation_steps=STEP_SIZE_VALID,
validation_data = val_generator,
epochs = 10,
verbose = 1,
callbacks = [checkpoint_callback,early_stopping_callback]
#class_weight = class_weights,
#callbacks = [checkpoint_callback]
)
```

Figure 4.24: vgg19 model fitting

- The model is trained using the `fit` method.
- Data is supplied via `train_generator` and `val_generator`, which likely provide batches of training and validation data, respectively.
- Training runs for a maximum of 10 epochs but may halt earlier if the `EarlyStopping` callback (defined earlier in the ResNet152 section) is triggered.

This approach leverages the powerful feature extraction capabilities of VGG19, fine-tuning it on a new dataset to achieve potentially better performance than training a model from scratch.

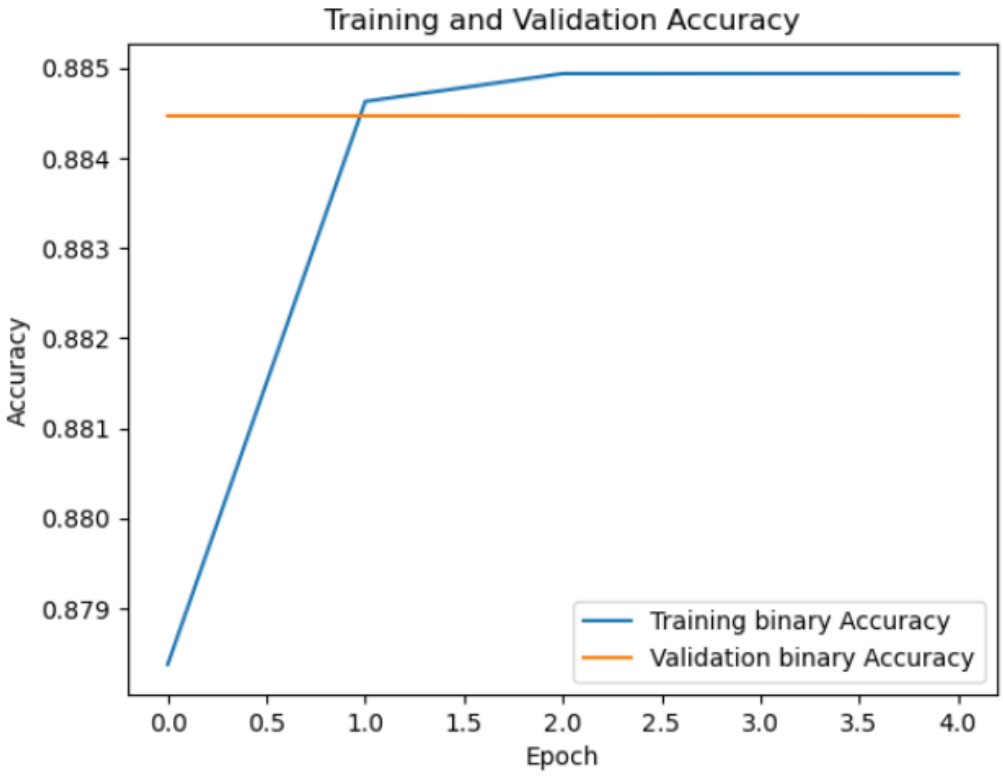

Figure 4.25: vgg19 model train val binary accuracy graph

4.10 Training Xception Model

1. **Preprocessing:**
   - A sequence of preprocessing steps is set up to manage the input images:
     - **Resizing**: The images are resized to dimensions of 224 × 224 pixels. This matches the standard input size expected by the Xception model.
     - **Rescaling**: Pixel values of the images are rescaled to lie between 0 and 1 by dividing them by 255. This normalization aids in faster model convergence and higher accuracy.
2. **Loading the pretrained Xception Model:**
   - The Xception model, pre-trained on the ImageNet dataset, is loaded. This provides a robust set of feature extractors useful for various image classification tasks.
   - The top layer (or classification head) of the pre-trained model is excluded, indicating a custom classification layer will be added.
   - All layers of the pre-trained model are frozen, ensuring that their weights do not get updated during further training.

```python
import tensorflow as tf
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.applications import Xception
from tensorflow.keras.models import Model

# Load the pretrained Xception model
pretrained_model = Xception(
    include_top=False,
    input_shape=(224, 224, 3),
    weights='imagenet'
)

# Freeze layers of the pretrained model
for layer in pretrained_model.layers:
    layer.trainable = False

# Add layers for classification on top of Xception
inputs = Input(shape = (256,256,3))
x= resize_rescale_hf_without_transpose(inputs)
x = pretrained_model(x)
x = Flatten()(x)
x = Dense(512, activation='relu')(x)
predictions = Dense(14, activation='sigmoid')(x)

# Create the final model
large_xception_model = Model(inputs=inputs, outputs=predictions)

# Print model summary
large_xception_model.summary()
```

Figure 4.26: xception model training

3. **Model Extension for Custom Task:**
   - An input layer is defined for images of size $256 \times 256 \times 3$.
   - The preprocessing steps (resizing and rescaling) are applied to this input.
   - The modified input goes through the pre-trained Xception model.
   - The output from the Xception model is flattened into a one-dimensional array.
   - A dense layer with 512 neurons and a ReLU activation function is added.
   - The final layer has 14 neurons with a sigmoid activation function, indicating binary decisions across 14 classes.
4. **Model Compilation:**
   - The model uses the Adam optimizer and binary cross-entropy loss, indicating binary classification for each class.
   - The metrics for model performance evaluation are binary accuracy and mean absolute error (MAE).

```
large_xception_model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
                             metrics = ['binary_accuracy', 'mae'])
```

Figure 4.27: xception model compiling

5. **Callbacks for Training:**
   - **ModelCheckpoint**: Saves the model weights if there's an improvement in the validation binary accuracy.
   - **EarlyStopping**: Stops training if the validation binary accuracy does not improve for 2 consecutive epochs, preventing overfitting.

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint_callback = ModelCheckpoint(
    filepath="/kaggle/working/best_weights_xception.h5",  # Replace wi
    monitor='val_binary_accuracy',
    save_best_only=True,
    save_weights_only=True,
    mode='max',
    verbose=1
)
```

```
early_stopping_callback = EarlyStopping(
    monitor='val_binary_accuracy',
    patience=2,  # Number of epochs with no improvement before stoppin
    restore_best_weights=True,
    verbose=1
)
```

Figure 4.28: xception model callbacks

6. **Model Training:**
   - The model trains using data from `train_generator` and validates with `val_generator`.
   - Training runs for 10 epochs but may stop earlier if early stopping conditions are met.

- Steps per epoch for training and validation are defined by `STEP_SIZE_TRAIN` and `STEP_SIZE_VALID`, respectively.

```
history = large_xception_model.fit(
train_generator,
#batch_size = 32,
steps_per_epoch=STEP_SIZE_TRAIN,
validation_steps=STEP_SIZE_VALID,
validation_data = val_generator,
epochs = 10,
verbose = 1,
callbacks = [checkpoint_callback,early_stopping_callback]
#class_weight = class_weights,
#callbacks = [checkpoint_callback]
)
```

Figure 4.29: xception model fiting

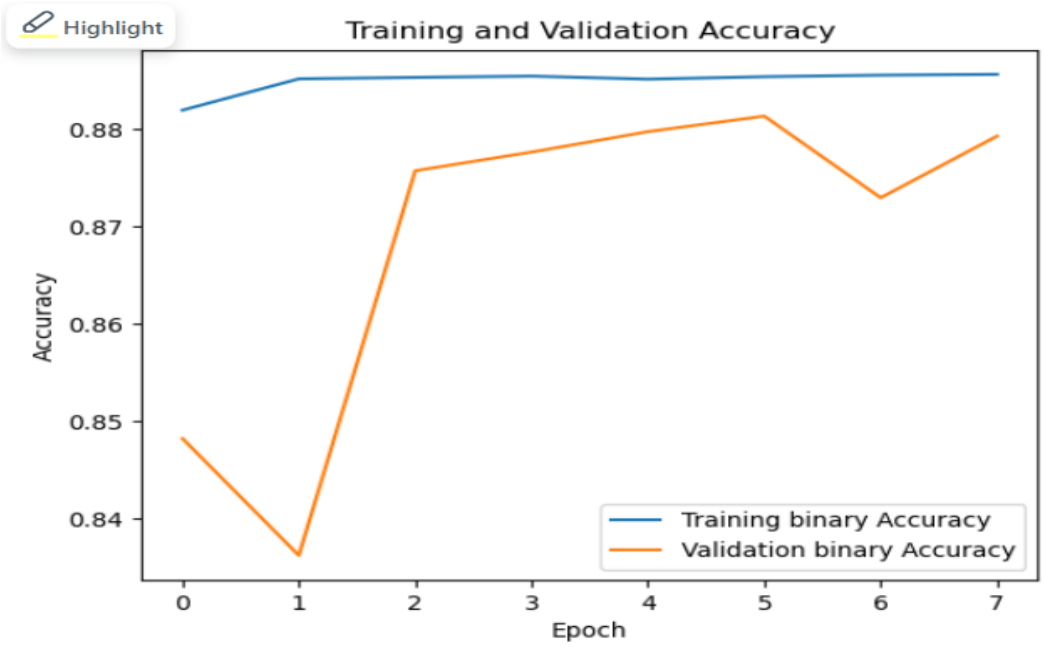

Figure 4.30: xception model train, val binary accuracy graph

4.11 Ensembling with Meta Model

In this section, we implement the technique of "ensembling" to harness the combined predictive power of multiple models. By stacking the predictions from four distinct models horizontally, we utilize a meta model to train on these aggregated predictions. Here, the predictions from the ensemble of models serve as features for the meta-model, which then performs the final classification.

1. **Generating Predictions**: Predictions are derived from each model (VIT, ResNet152, VGG19, Xception) on the validation data using their respective `predict` methods.
2. **Stacking Predictions**: Using `np.hstack`, the individual predictions from each model are stacked horizontally. This results in an array whose size is $(N, 56)$, assuming each model produces an array of size $(N, 14)$, with N being the number of validation samples and 14 representing the classes.

```
pred1_test = model1.predict(test_generator)
pred2_test = model2.predict(test_generator)
pred3_test = model3.predict(test_generator)
pred4_test = model4.predict(test_generator)
#... and so on for each model.
stacked_predictions_test = np.hstack((pred1_test, pred2_test, pred3_test, pred4_test))
# stacked_prediction = [pred1,pred2,pred3,pred4]  , len(stacked_prediction) = 56
```

```
125/125 [==============================] - 1005s 8s/step
125/125 [==============================] - 559s 4s/step
125/125 [==============================] - 548s 4s/step
125/125 [==============================] - 211s 2s/step
```

Figure 4.31: stacking val predictions

here, as shown in the example, the predictions are stacked one after another in a horizontal manner. The first 14 probability values of the predictions from ViT, Resnet152, VGG19, Xception are concatenated, thereby getting 56 values as the stacked predictions. These stacked predictions become feature vectors for the meta-model and are then further used for training the meta-model Mallick et al. (2022).

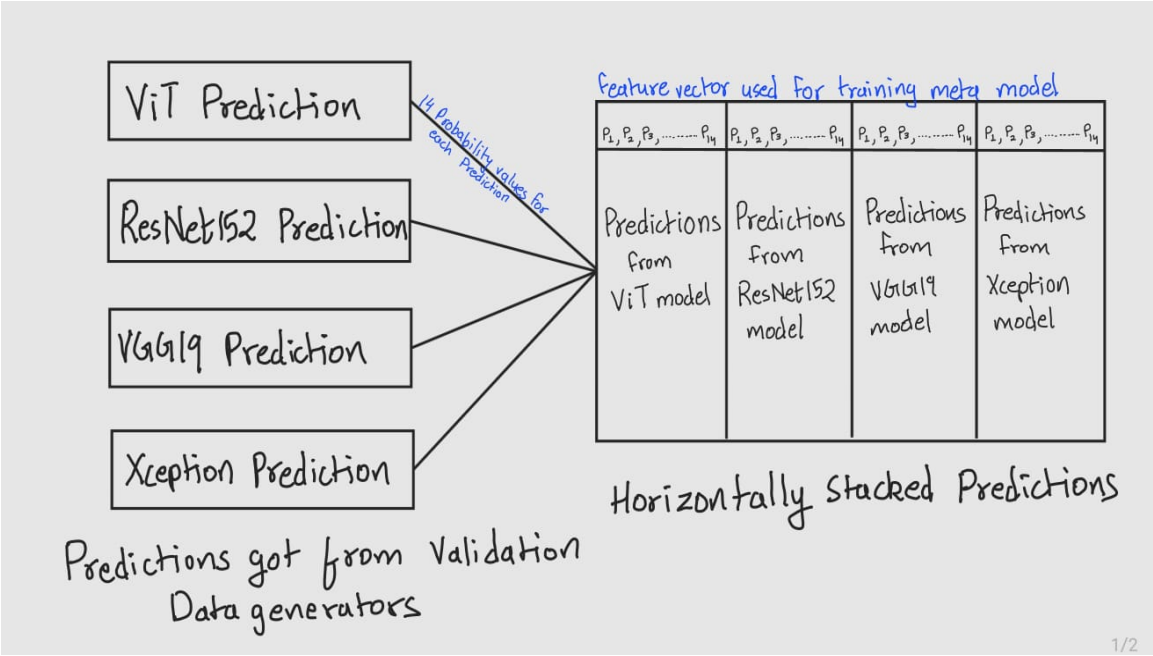

Figure 4.32: horizontal stacking of model for meta-model training

After training the meta-model, in order to get a prediction for any X-ray image using our meta-model, first we need to pass our image as input to all 4 models used, to get their respective predictions. After getting predictions from all these 4 models, we stack those predictions horizontally i.e. (concatenating them one after another). We then pass this stacked prediction as input to the meta-model to get our final prediction Lu et al. (2023).

```
In [74]: print(f'prediction from vit--->{pred1[0]}')
         print('-------------------------------------')
         print(f'prediction from resnet50--->{pred2[0]}')
         print('-------------------------------------')
         print(f'prediction from vgg19--->{pred3[0]}')
         print('-------------------------------------')
         print(f'prediction from xception--->{pred4[0]}')
         print('-------------------------------------')
         print(f'stacked predictions----->{stacked_predictions[0]}')

prediction from vit--->[0.25474072 0.14282958 0.16281486 0.07838243 0.28752932 0.18293774
 0.19222884 0.04390055 0.43276727 0.2895394  0.29755658 0.18856244
 0.12761593 0.21827292]
-------------------------------------
prediction from resnet50--->[0.27488905 0.0776007  0.15126301 0.04756674 0.31351697 0.04942755
 0.03591837 0.00613678 0.35807464 0.14142014 0.17263266 0.08981376
 0.04735346 0.12194947]
-------------------------------------
prediction from vgg19--->[1.6326134e-01 1.2160796e-02 4.9716074e-02 1.4681849e-02 1.9803664e-01
 1.2541892e-02 5.9460080e-03 1.3290271e-04 3.2645887e-01 4.3351192e-02
 6.0146749e-02 1.6881604e-02 6.1840983e-03 3.9241418e-02]
-------------------------------------
prediction from xception--->[1.0228627e-01 5.4541539e-04 3.3940000e-03 1.4051871e-04 3.6269311e-02
 2.4555766e-03 1.1256974e-03 1.9673340e-05 2.7172521e-01 2.8945575e-02
 5.0540775e-02 7.6699671e-03 3.4659414e-04 8.0678696e-03]
-------------------------------------
stacked predictions----->[2.5474072e-01 1.4282958e-01 1.6281486e-01 7.8382432e-02 2.8752932e-01
 1.8293774e-01 1.9222884e-01 4.3900553e-02 4.3276727e-01 2.8953940e-01
 2.9755658e-01 1.8856244e-01 1.2761593e-01 2.1827292e-01 2.7488905e-01
 7.7600695e-02 1.5126301e-01 4.7566742e-02 3.1351697e-01 4.9427554e-02
 3.5918374e-02 6.1367834e-03 3.5807464e-01 1.4142014e-01 1.7263266e-01
 8.9813761e-02 4.7353458e-02 1.2194947e-01 1.6326134e-01 1.2160796e-02
 4.9716074e-02 1.4681849e-02 1.9803664e-01 1.2541892e-02 5.9460080e-03
 1.3290271e-04 3.2645887e-01 4.3351192e-02 6.0146749e-02 1.6881604e-02
 6.1840983e-03 3.9241418e-02 1.0228627e-01 5.4541539e-04 3.3940000e-03
 1.4051871e-04 3.6269311e-02 2.4555766e-03 1.1256974e-03 1.9673340e-05
 2.7172521e-01 2.8945575e-02 5.0540775e-02 7.6699671e-03 3.4659414e-04
 8.0678696e-03]
```

Figure 4.33: stacking val predictions example

3. **Fetching Validation Labels**: For the meta-model training, the actual labels from the validation generator are extracted and stored in the `val_labels` array.

```
val_labels = []
for i in range(len(val_generator)):
    _, labels_batch = val_generator[i]
    val_labels.extend(labels_batch)

val_labels = np.array(val_labels)
```

Figure 4.34: getting val labels from val generator

4. **Creating the Meta Model**:
    - The meta-model, a neural network, is designed to accept the stacked predictions as input.
    - This model comprises three dense layers interspersed with batch normalization and dropout for effective regularization.
    - The concluding layer encompasses 14 neurons with a sigmoid activation function, aligning with the class count.

```
from  tensorflow.keras.layers import BatchNormalization, Dense, Dropout
import tensorflow as tf


meta_model = tf.keras.Sequential([
    Dense(512, activation='relu', input_shape=(stacked_predictions.shape[1],)),
    BatchNormalization(),
    Dropout(0.5),
    Dense(256, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    #Dense(64, activation='relu'),
    #BatchNormalization(),
    #Dropout(0.5),

    Dense(14, activation='sigmoid')
])
```

Figure 4.35: meta model training

5. **Compiling and Training the Meta Model**:
    - A checkpoint callback is configured to preserve the best meta-model weights based on binary accuracy.
    - With the Adam optimizer and binary cross-entropy loss, the meta-model is compiled.
    - The model undergoes training using the stacked predictions, with the genuine validation labels as the reference. The training spans 250 epochs.

Leveraging the meta-model to train on the stacked predictions facilitates the incorporation of combined insights and patterns discerned by individual models, potentially yielding a more robust and precise classifier.

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

checkpoint_callback = ModelCheckpoint(
    filepath="C:/Users/hemal nakrani/DeepLearningProjects/AI_Projects/chest_x_ray
    monitor='binary_accuracy',
    save_best_only=True,
    save_weights_only=True,
    mode='max',
    verbose=1
)
```

Figure 4.36: meta model callback

```
from sklearn.metrics import hamming_loss
meta_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['binary_accuracy']) # Adjust loss d
meta_model.fit(stacked_predictions, val_labels, epochs=250, batch_size=32,callbacks = [checkpoint_callback])
```

Figure 4.37: meta model fiting

4.12 Deployment

We are deploying a web application that utilizes a meta-model to make predictions. The technologies chosen for deployment are FastAPI, Streamlit, and Docker.

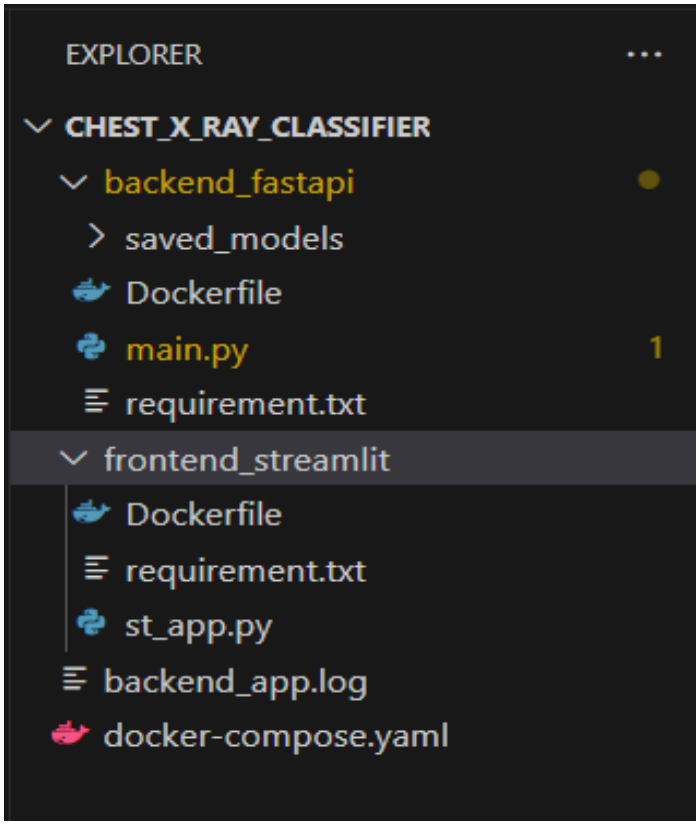

Figure 4.38: project directory structure

4.12.1 FastAPI Code:

1. **Imports & Setup:**
   - Modules like numpy, FastAPI, tensorflow, transformers, and others are imported.
   - Logging is set up to write logs to "backend_app.log".
2. **Model Loading:**
   - Custom objects for the TFViTModel are defined.
   - Several pre-trained models (VIT MODEL, RESNET152 MODEL,VGG19 MODEL and XCEPTION MODEL) are loaded from specified paths.
3. **API Endpoints:**

- A basic endpoint (‘/‘) is provided that returns ”hello world”.
- The main endpoint (‘/predict‘) takes an uploaded image, processes it, and runs it through the pre-loaded models to get a prediction. The prediction is then returned as a JSON response.

4. **Image Processing Functions:**
   - There are several utility functions such as `read_file_as_image` which reads the image file and resizes it and `apply_clahe_to_image` which apply CLAHE to the read image and returns that image.
5. **Prediction Functions:**
   - Multiple functions like `sparse_binary_convertor` which converts the output to sparse binary using threshold = 0.28 and `get_labels_from_output` which is used to process model predictions to get predicted class names
6. **Main Execution:**
   - If the script is executed as the main program, it will run a FastAPI server on `0.0.0.0:5000`.

4.12.2 Streamlit Code:

1. **Function Definitions:**
   - `post_run`: Sends the uploaded image to the FastAPI backend and returns the response.
   - `run`: The main function that sets up the Streamlit interface, handles file uploads, sends the image for prediction and displays results.
2. **Interface:**
   - The app displays instructions, provides a file uploader widget and shows prediction results once the ”Predict health” button is clicked.

4.12.3 Dockerfile for FastAPI:

1. **Base Image:** Uses `python:3.10-slim` as the base image.
2. **Copying and Setting Working Directory:** Copies the backend code to `/chest_x_ray_classifier/backend_fastapi` in the container and sets it as the working directory.
3. **System Dependencies:** Installs system packages needed by OpenCV and other Python libraries.
4. **Python Dependencies:** Installs Python packages from `requirement.txt`.

```
FROM python:3.10-slim

COPY . /chest_x_ray_classifier/backend_fastapi
#set working directory
WORKDIR /chest_x_ray_classifier/backend_fastapi

# Update and install system dependencies required
RUN apt-get update && apt-get install -y \
    build-essential \
    curl \
    software-properties-common \
    git \
    libgl1-mesa-glx \
    && rm -rf /var/lib/apt/lists/*

#install dependencies
RUN pip install -r requirement.txt

EXPOSE 5000

ENTRYPOINT [ "python" ]
CMD ["main.py"]
```

Figure 4.39: fastapi dockerfile

5. **Exposed Port:** Exposes port `5000` for the API.
6. **Execution Command:** Executes `main.py` using Python when the container starts.

4.12.4 Dockerfile for Streamlit:

1. **Base Image:** Uses `python:3.10-slim` as the base image.
2. **Copying and Setting Working Directory:** Copies the frontend code to `/chest_x_ray_classifier/frontend_streamlit` in the container and sets it as the working directory.
3. **System Dependencies:** Installs essential system packages.
4. **Python Dependencies:** Installs Python packages from `requirement.txt`.
5. **Exposed Port:** Exposes port `8501` for the Streamlit app.
6. **Execution Command:** Executes `st_app.py` using Streamlit when the container starts.

```
frontend_streamlit > Dockerfile > ...
1   FROM python:3.10-slim
2
3
4   COPY . /chest_x_ray_classifier/frontend_streamlit
5   #set working directory
6   WORKDIR /chest_x_ray_classifier/frontend_streamlit
7
8   RUN apt-get update && apt-get install -y \
9       build-essential \
10      curl \
11      software-properties-common \
12      git \
13      && rm -rf /var/lib/apt/lists/*
14
15  #install dependencies
16  RUN pip install -r requirement.txt
17
18  EXPOSE 8501
19
20  ENTRYPOINT [ "streamlit","run" ]
21  CMD ["st_app.py"]
```

Figure 4.40: streamlit dockerfile

4.12.5 Docker Compose File:

1. **Services:**
   - **backend_fastapi**: It builds the backend from the `./backend_fastapi` directory and exposes it on port `5000`.
   - **frontend_streamlit**: It builds the frontend from the `./frontend_streamlit` directory and exposes it on port `8501`. It also depends on `backend_fastapi`, meaning it waits for the backend service to start before starting itself.

```yaml
version: '3.8'

services:

  backend_fastapi:
    build: ./backend_fastapi
    ports:
      - 5000:5000

  frontend_streamlit:
    build: ./frontend_streamlit
    ports:
      - 8501:8501
    depends_on:
      - backend_fastapi
```

Figure 4.41: docker compose file

4.12.6 Summary

We are deploying a web application using Docker containers. The backend is built using FastAPI, which loads deep learning models and makes predictions. The front end is a Streamlit application that lets users upload an image and see model predictions. Both the frontend and backend are containerized using Docker, and their orchestration is managed using Docker Compose.

4.12.7 Future Scope

The versatility and modular nature of our web application design, combined with containerization using Docker, open up the possibility for seamless deployment on major cloud platforms in the future.

1. **Oracle Cloud**: We can utilize Oracle's cloud infrastructure to host our application. Given the support for Docker on Oracle Cloud, deploying the app is as simple as pushing our code and executing the `docker-compose up` command.
2. **Google Cloud Platform (GCP)**: GCP's Kubernetes Engine can be leveraged

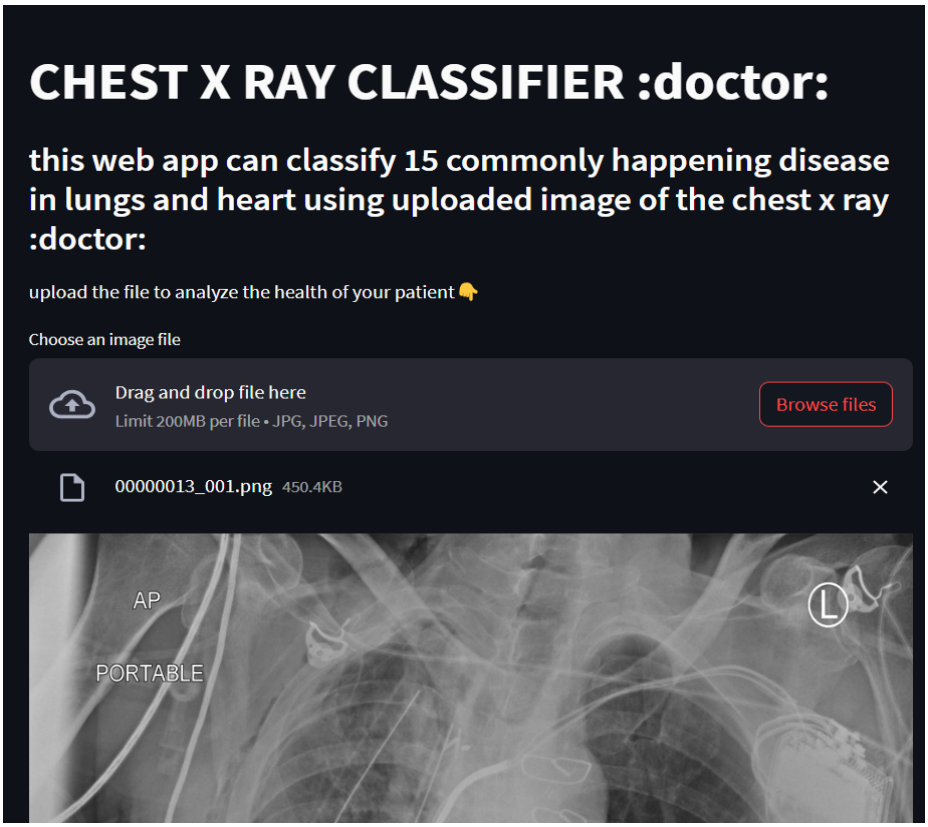

Figure 4.42: deployed web application

for container orchestration. With our Dockerfiles for Streamlit and FastAPI in hand, we can easily push our containers to the Google Container Registry and then deploy them on GCP. This would allow for robust, scalable deployment with the ability to handle a high number of simultaneous users.

Such cloud deployments not only ensure scalability but also enhance the availability, resilience, and global reach of our application.

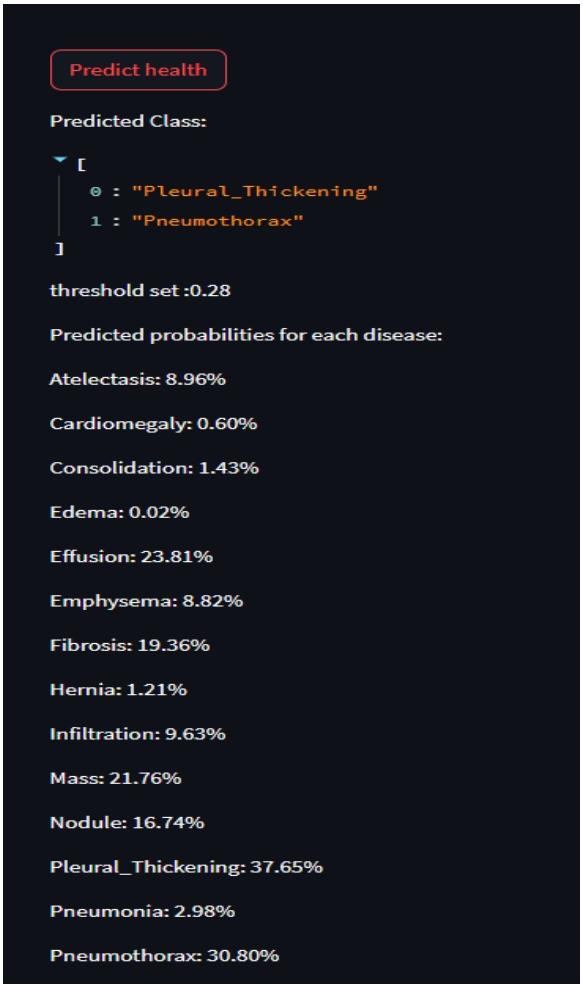

Figure 4.43: web application prediction output

Chapter 5

Evaluation And Testing

5.1 Introduction

In this journey of deep learning model development, one of the important steps is the evaluation and testing phase. This step not only ascertains that our model transcends mere rote learning of the training data but also stands competent in generalizing and deducing accurate predictions on unfamiliar data. As we navigate this chapter, our main focus will be the granular aspects of our chosen evaluation and testing methodologies.

At the core of our strategy is the adoption of test generators. Opting for test generators facilitates a consistent, reproducible, and structured means of presenting data to our model during the evaluation phase. It's imperative to underscore that we've decisively set data shuffling to `false`. Such a choice ensures that the data sequence remains unaltered throughout the testing, a facet that is quintessential for upholding the fidelity of our assessment. By preserving the original sequence of data, we can foster trustworthy comparisons and derive consistent insights regarding the model's efficacy.

As this discourse unfolds, the intricate layers of our approach will be illuminated, underlining the rigour and dependability of our testing and evaluation blueprint.

5.2 Evaluation Metrics

5.2.1 Hamming Loss

Description: Hamming Loss is a measure of how well our model's predictions match the true labels across all classes. For a given instance, it calculates the fraction of labels that are incorrectly predicted. In other words, it measures the average fraction of incorrect labels Stemerman et al. (2021).

Rationale for Choosing: Given that we are dealing with a multi-label classification problem, each image might belong to more than one category. Hamming Loss provides a measure that encapsulates the performance across all labels. The benefit of using this metric is that it inherently handles multiple labels and provides a singular value that can be used to judge performance Tsoumakas et al. (2010).

Interpretation: A Hamming Loss of 0 indicates perfect multi-label classification, meaning all labels for all samples are classified correctly. As the value approaches 1, the performance of the classifier worsens. Thus, lower values of Hamming Loss indicate better performance.

5.2.2 ROC-AUC Curve

Description: The Receiver Operating Characteristic (ROC) curve is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The Area Under the Curve (AUC) of the ROC represents the classifier's ability to distinguish between the positive and negative classes. For multi-label classification, the ROC-AUC can be computed for each label individually and then averaged.

Rationale for Choosing: In multi-label image classification, it is important to understand how well the model can distinguish between each label. The ROC-AUC provides a measure of this ability without being dependent on a specific threshold, making it a valuable metric when classes are imbalanced or when the cost of false positives is different from false negatives Grandini et al. (2020).

Interpretation: An AUC of 1 indicates perfect classification, while an AUC of 0.5 suggests the model is performing no better than random guessing. For multi-label classifiers, we often consider the micro-average (considering all labels collectively) and macro-average (averaging the AUCs for each label) ROC-AUC values. A high AUC indicates that the model is capable of distinguishing between the positive and negative instances for each label effectively.

With these metrics, you'll have a comprehensive understanding of your multi-label classifier's performance. They provide both a holistic view of performance across labels (Hamming Loss) and the ability of the model to discriminate between labels (ROC-AUC).

5.3 Evaluation of Trained ViT Model

5.3.1 Hamming Loss:

The Hamming Loss for the trained Vision Transformer (ViT) model is 0.24258. The Hamming Loss gives us a measure of how well the model's predictions match the

true labels across all classes. A Hamming Loss of 0 would mean a perfect multi-label classification, where all labels for all samples have been predicted correctly. In our case, a value of 0.24258 suggests that on average, about 24.25% of the labels are incorrectly predicted for a given instance. The closer this value is to 0, the better the performance.

5.3.2 ROC-AUC Curve Values:

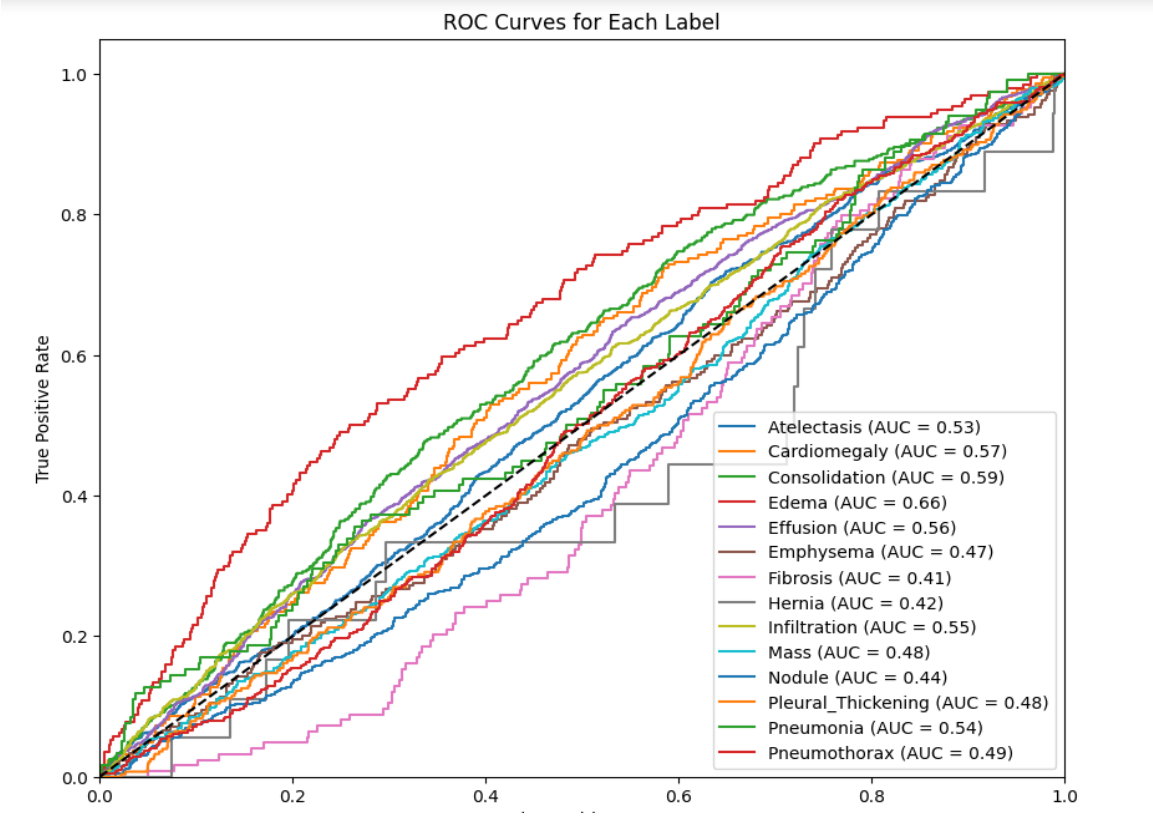

Figure 5.1: ViT ROC AUC curve

The Receiver Operating Characteristic (ROC) curve is a plot that describes the performance of a binary classifier system. The Area Under the Curve (AUC) represents the classifier's ability to differentiate between the positive and negative classes for each label. An AUC of 1 indicates perfect classification, whereas an AUC of 0.5 suggests the model is performing similarly to random guessing.

The highest AUC was achieved for Edema (0.66), suggesting that the model has a better ability to distinguish this condition compared to others. On the other hand, the model struggled the most with distinguishing Fibrosis, having the lowest AUC (0.41).

The AUC values indicate that there is considerable variability in the model's per-

formance across different classes. Improvements are needed to enhance the model's ability to discriminate between the different conditions more effectively, as most of the AUC values are near or below 0.6.

The evaluation of the trained ViT model using Hamming Loss and individual ROC-AUC values for each class reveals a need for further optimization. The overall performance across the different classes is not consistent, with some conditions being better distinguished by the model than others. The hamming loss also points towards a significant number of incorrect predictions, emphasizing the need for model refinement and optimization to achieve better and more consistent performance across all classes.

5.4 Evaluation of Trained ResNet152 Model

5.4.1 Hamming Loss:

The Hamming Loss for the trained ResNet152 model is 0.18126. This metric offers insights into, how well the model's predictions align with the true labels across all the disease categories. A Hamming Loss value of 0 would symbolize perfect multi-label classification, with every label for every sample correctly predicted. In the current scenario, a score of 0.18126 means that on average, approximately 18.12% of the labels are predicted inaccurately for a given instance. Ideally, we aim for lower values in this metric; thus, our result indicates a reasonably strong performance.

5.4.2 ROC-AUC Curve Values:

The ROC (Receiver Operating Characteristic) curve is a tool designed to illustrate the performance of a binary classification model. The AUC (Area Under the Curve) value that accompanies it quantifies the classifier's proficiency in distinguishing between positive and negative instances for each label. An AUC value of 1 signifies perfect classification, while a score of 0.5 suggests performance equivalent to random guessing.

From the ResNet model evaluation, the AUC values for each disease label are as detailed: The ROC-AUC values signify the model's competency in distinguishing positive and negative instances for each individual class. Higher values, approaching 1, are indicative of superior discriminative capability. The ResNet152 model has shown proficient discriminative ability, particularly for Edema, with an AUC of 0.68, suggesting an enhanced capability in identifying this condition in comparison to others. Conversely, the model experienced difficulties in discerning Fibrosis, which demonstrated the lowest AUC of 0.36, indicating a need for improvement in discriminating this condition. The assessment of the ResNet152 model using Hamming Loss and individual ROC-AUC values uncovers that the model is performing reasonably well,

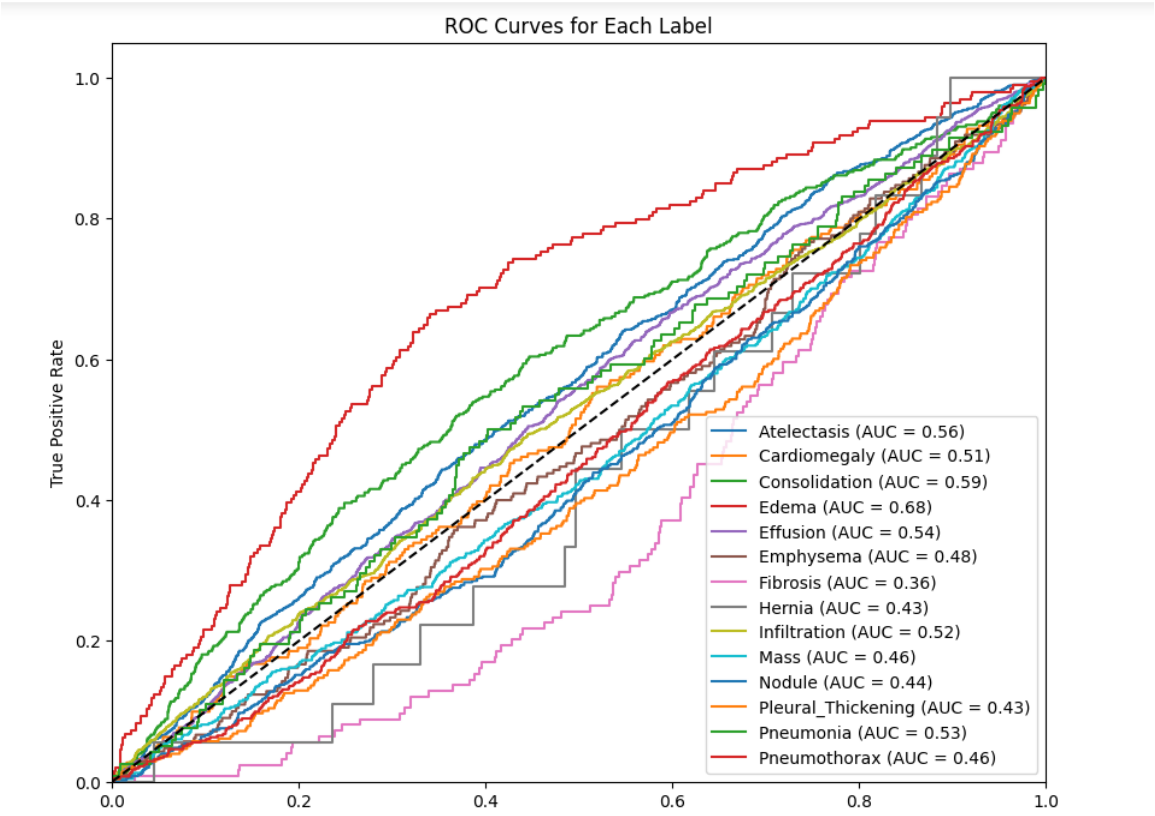

Figure 5.2: resnet152 roc auc curve

with a significant reduction in incorrect predictions compared to the ViT model. The ROC-AUC values exhibit variability in performance across different conditions, with notable success in discerning Edema but apparent struggle with Fibrosis. Despite the advancements, there still exist disparities in the model's ability to distinguish between various conditions efficiently and consistently, necessitating further refinement and enhancement for achieving optimal and uniform performance across all classes.

5.5 Evaluation of Trained VGG19 Model

5.5.1 Hamming Loss:

The Hamming Loss for the trained VGG19 model stands at 0.13355. This metric provides insight into the alignment between the model's predictions and the true labels across the disease categories. A value of 0 in Hamming Loss would indicate impeccable multi-label classification, where every single label for every sample has been predicted flawlessly. In this context, a score of 0.13355 suggests that, on average, about 13.35% of the labels are predicted inaccurately for a particular instance. Generally, lower values are desired for this metric, hence the result from the VGG19 model is indicative

of a reasonably solid performance.

5.5.2 ROC-AUC Curve Values:

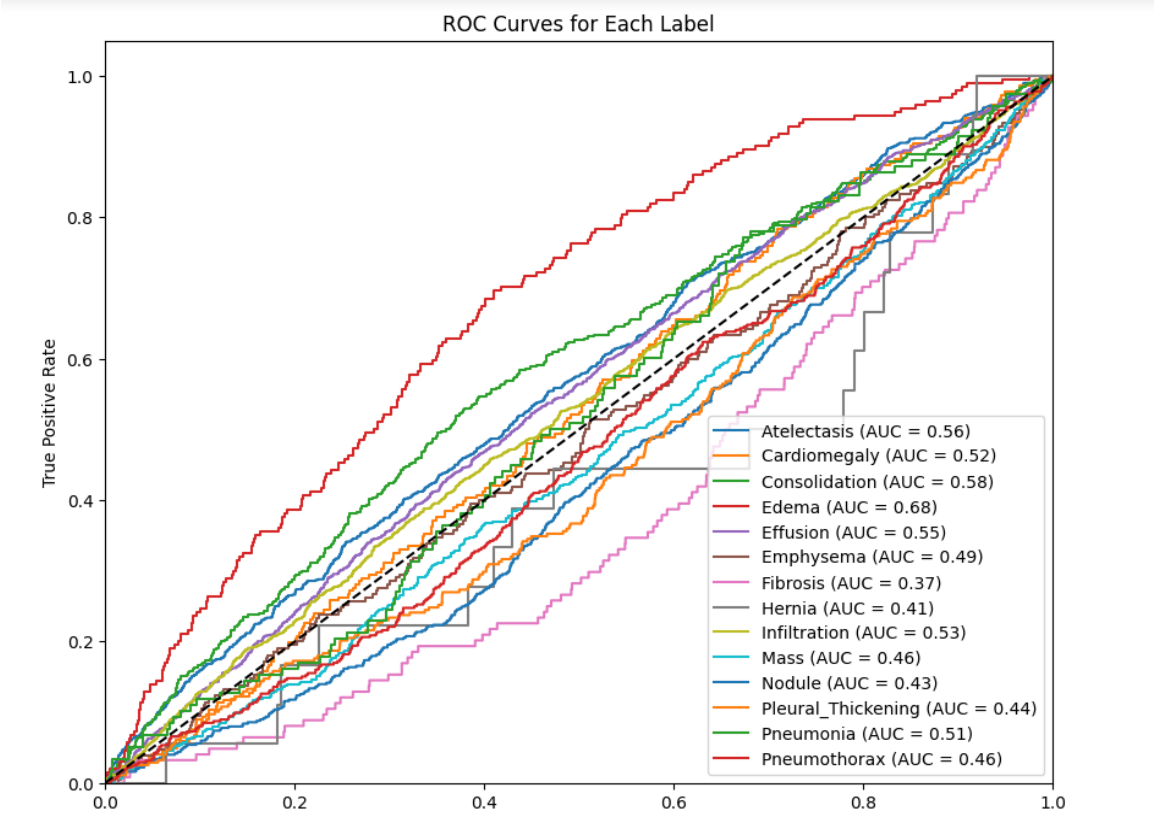

Figure 5.3: vgg19 roc auc curve

The Receiver Operating Characteristic (ROC) curve is a graphical representation tailored to express the performance of a binary classification model. The Area Under the Curve (AUC) accompanying the ROC measures the classifier's capacity to differentiate between positive and negative instances for each label. An AUC of 1 signifies unparalleled classification, whereas a score of 0.5 would denote a model performing on par with random chance.

Evaluating the AUC values from the VGG19 model for each disease label: These values signify the model's ability to discriminate between positive and negative instances for each class. The highest AUC value, 0.68 for Edema, indicates that the model is most proficient at distinguishing this condition, while the lowest AUC value of 0.37 for Fibrosis suggests that the model struggles more with discerning this condition

In conclusion, The VGG19 model has exhibited superior performance compared to the previously mentioned models, as denoted by a lower Hamming Loss. This

model seems particularly capable of identifying cases of Edema, but like the other models, struggles with conditions like Fibrosis. While the model displays a promising and balanced performance across different conditions, there remains room for further enhancements to improve its discriminative ability for classes with lower AUC values, ensuring a more consistent and reliable performance across all conditions

5.6 Evaluation of Trained Xception Model

5.6.1 Hamming Loss:

The Xception model, trained on our dataset, exhibits a Hamming Loss of 0.16085. To elucidate, the Hamming Loss metric is an indicator of how aligned the model's predictions are with the true labels across the various classes. A Hamming Loss of 0 represents a scenario where every label for every sample has been perfectly predicted. The observed value of 0.16085 implies that, on average, about 16.08% of the labels are inaccurately predicted for an individual sample. This metric acts as an inverse indicator of performance, where lower values are preferable. The score achieved by the Xception model suggests a reasonably commendable performance.

5.6.2 ROC-AUC Curve Values:

The Receiver Operating Characteristic (ROC) curve offers a visualization of a model's binary classification performance. The Area Under the Curve (AUC) quantifies the classifier's capability to distinguish between positive and negative classes for each individual label. An ideal AUC value of 1 denotes perfect classification abilities, while a value of 0.5 indicates the model's performance is akin to a random guess.

Upon evaluating the AUC values for the Xception model across various disease labels: These AUC values are indicative of the model's capabilities in discriminating between positive and negative instances for each condition. The model has the highest discriminative ability for the condition "Nodule" with an AUC value of 0.57, and the lowest ability for "Emphysema," with an AUC value of 0.40. Most of the AUC values hover around 0.5, suggesting that the model struggles to distinguish between the conditions and would benefit from further optimization and refinement.

In summation, The Xception model has shown intermediate performance, with its Hamming Loss being lower than some models and higher compared to others. It demonstrated a modest ability to distinguish between various medical conditions, with no particular condition showing exceptionally high AUC values. This uniform performance indicates that the model needs substantial improvements to increase its reliability and accuracy in diagnosing various conditions, as it is currently performing near the level of random guessing for most of the conditions.

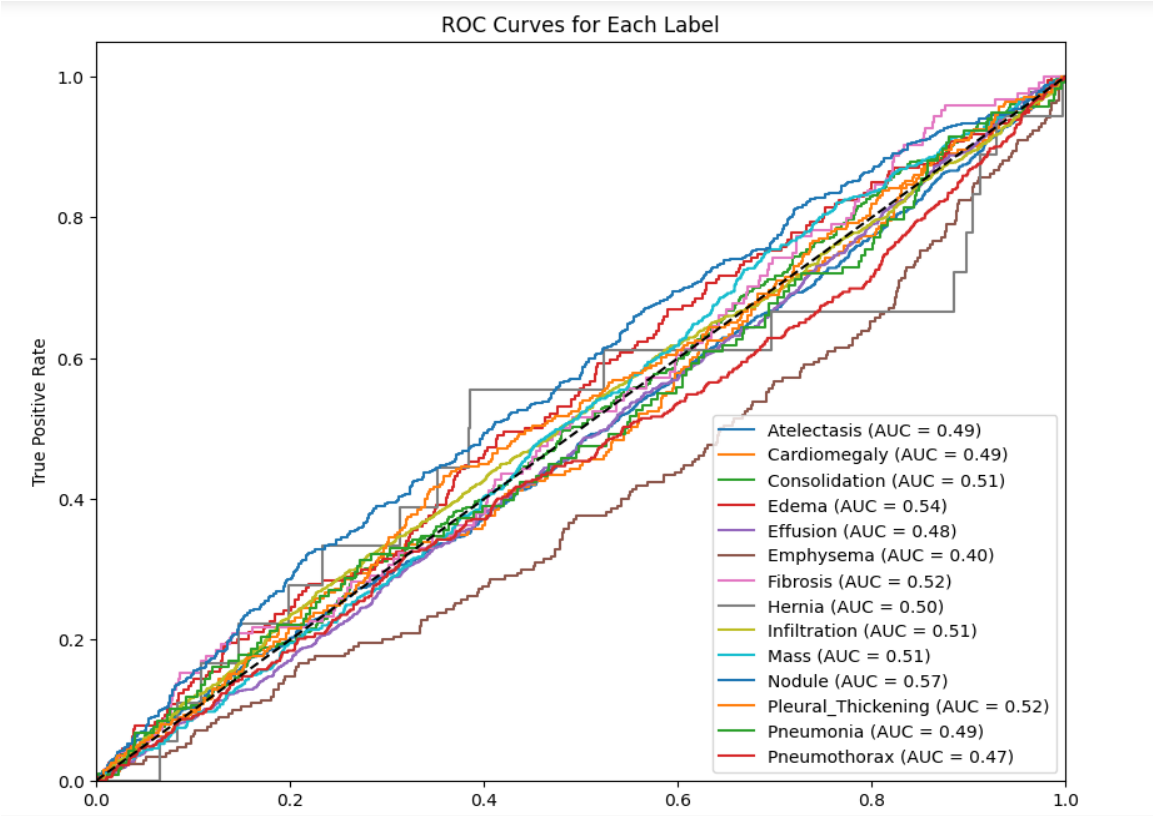

Figure 5.4: xception roc auc curve

5.7 Evaluation of Trained Meta Model

5.7.1 Hamming Loss

The evaluated Hamming Loss for the trained Meta Model stands at 0.140803. To put this metric into perspective, the Hamming Loss measures the discrepancy between the predicted labels by the model and the actual labels. A perfect score of 0 would mean that every single label has been predicted flawlessly for all samples. In our scenario, the value of 0.140803 suggests that on average, approximately 14.08% of the labels are incorrectly predicted per sample. Lower values are indicative of better performance. In comparison to many conventional models, the Meta Model's performance, as gauged by this metric, seems commendably accurate.

5.7.2 ROC-AUC Curve Values

The ROC-AUC values provide insight into the binary classification capabilities of the model for each disease label. An AUC value of 1 represents impeccable classification, while a value closer to 0.5 suggests performance akin to random guessing.

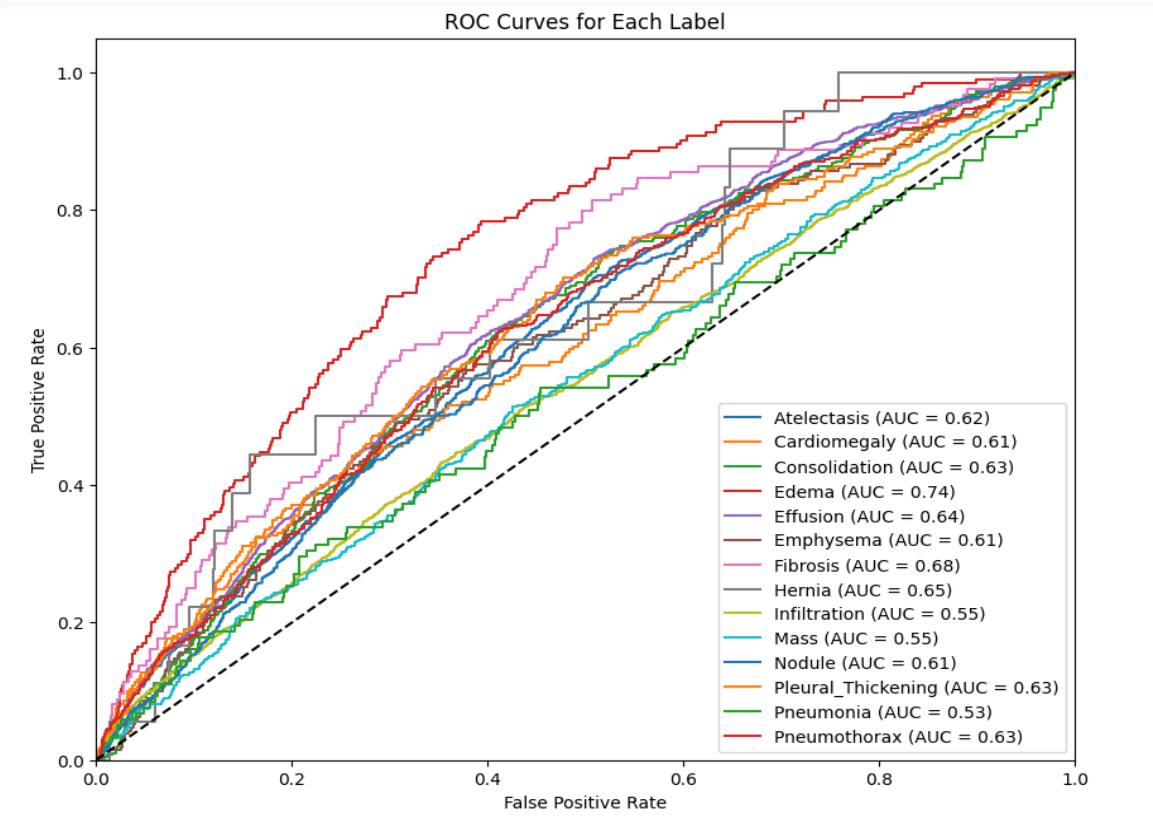

Figure 5.5: meta model roc auc curve

Here's a breakdown of the AUC values obtained for the Meta Model for the respective diseases:

- "Edema" achieves the highest AUC value of 0.74, indicating that the model exhibits its most distinguished performance in discerning between positive and negative cases for this particular disease.
- In contrast, "Infiltration" and "Mass" are the least distinguishable conditions, both with an AUC value of 0.55

The results suggest that the Meta Model exhibits a strong performance across multiple diseases, with some room for improvement in specific areas, especially for diseases like "Mass" and "Infiltration".

The Meta model has shown superior performance with its lower Hamming Loss and higher ROC-AUC values compared to other models. The model's discriminative power is particularly pronounced in conditions such as Edema, Fibrosis, and Hernia, allowing for more reliable and accurate predictions in these areas. However, despite its advanced capabilities, there are still areas, like Infiltration and Mass, where the model's discriminatory ability could be further improved for optimal performance in

multi-label medical image classification tasks.

5.8 Comparison of Models based on Evaluation Metrics

Table 5.1: Comparison of Hamming Loss and AUC values across models

Disease/Model	**ViT**	**ResNet**	**VGG19**	**Xception**	**Meta Model**
Atelectasis	0.53	0.56	0.56	0.49	0.62
Cardiomegaly	0.57	0.51	0.52	0.49	0.61
Consolidation	0.59	0.59	0.58	0.51	0.63
Edema	0.66	0.68	0.68	0.54	0.74
Effusion	0.56	0.54	0.55	0.48	0.64
Emphysema	0.47	0.48	0.49	0.40	0.61
Fibrosis	0.41	0.36	0.37	0.52	0.68
Hernia	0.42	0.43	0.41	0.50	0.65
Infiltration	0.52	0.52	0.53	0.51	0.55
Mass	0.48	0.46	0.46	0.51	0.55
Nodule	0.44	0.44	0.43	0.57	0.61
Pleural_Thickening	0.48	0.43	0.44	0.52	0.63
Pneumonia	0.54	0.53	0.51	0.49	0.53
Pneumothorax	0.49	0.46	0.46	0.47	0.63
Hamming Loss	0.24258	0.18126	0.13355	0.16085	0.140803

Chapter 6

Conclusion

In this dissertation, a meticulous exploration and analysis have been undertaken, delving into the capabilities of several prominent deep learning architectures including **ViT (Vision Transformer)**, **ResNet152**, **VGG19**, **Xception**, and a **Meta model** in multi-label chest X-ray image classification. The overarching goal was to identify various diseases present in these images with optimal accuracy.

Employing both *Hamming Loss* and the *ROC-AUC curve* as evaluation metrics allowed for an in-depth, multifaceted assessment of each model's performance. *Hamming Loss* provided insights into the proportion of incorrect predictions, while the *ROC-AUC curve* gauged the proficiency of each model in distinguishing the presence and absence of specific diseases.

The findings from this exploration have unearthed several insightful observations:

- **ViT (Vision Transformer)** showcased moderate efficacy, demonstrating an aptitude for detecting *Edema*, but conversely, faced considerable challenges with *Hernia*.
- **ResNet152** and **VGG19** displayed comparable outcomes, indicating nearly identical AUC values for each disease and registering similar *Hamming Loss*, suggesting a paralleled level of accuracy in their predictions.
- **Xception**, maintaining consistency, mirrored the performances of **ResNet152** and **VGG19**, signifying a balanced act across diverse conditions.
- Remarkably, the **Meta model** overshadowed its counterparts by not only achieving the lowest *Hamming Loss* but also exhibiting consistently higher AUC values across the diseases, indicating its superior discriminatory capabilities.

In light of these observations, the **Meta model** emerges as the superior model, illustrating its potential applicability and reliability in clinical diagnostic settings where precision is of paramount importance.

Even though some architectures might not have revealed their full potential in this study, the extensive analysis conducted has laid a profound foundation for future endeavors in this domain. The comprehensive insights gained from understanding the unique strengths and nuances of each model will play a pivotal role in guiding subsequent research, fostering an informed and optimized approach to model selection and refinement.

Continued exploration into ensemble models and emerging architectures remains a promising avenue, focusing on the enhancement of diagnostic accuracy and efficiency in interpreting chest X-rays to elevate patient care standards progressively.

References

Basu, S., Mitra, S., & Saha, N. 2020, 2020 IEEE Symposium Series on Computational Intelligence (SSCI), 2521. https://api.semanticscholar.org/CorpusID:216056619

Bharati, S., Podder, P., & Mondal, M. R. 2020, Informatics in Medicine Unlocked, 20, 100391, doi: 10.1016/j.imu.2020.100391

Brunese, L., Mercaldo, F., Reginelli, A., & Santone, A. 2020, Computer Methods and Programs in Biomedicine, 196, 105608, doi: 10.1016/j.cmpb.2020.105608

Büyükçakir, A., Bonab, H., & Can, F. 2018, Proceedings of the 27th ACM international conference on information and knowledge management, 1063

Grandini, M., Bagli, E., & Visani, G. 2020, arXiv preprint arXiv:2008.05756

Huang, M.-L., & Liao, Y.-C. 2022, Computers in Biology and Medicine, 146, 105604 . https://api.semanticscholar.org/CorpusID:248671385

Iqball, T., & Wani, M. A. 2023, International Journal of Information Technology, 15, 557–564, doi: 10.1007/s41870-022-01149-8

Jiang, X., Zhu, Y., Cai, G., et al. 2022, Cognitive Computation, 14, 1362, doi: https://doi.org/10.1007/s12559-022-10032-4

Kim, S., Rim, B., Choi, S., et al. 2022a, Diagnostics, 12, doi: 10.3390/diagnostics12040915

—. 2022b, Diagnostics, 12, 915

Lu, M., Hou, Q., Qin, S., et al. 2023, Water, 15, 1265, doi: 10.3390/w15071265

Madani, A., Moradi, M., Karargyris, A., & Syeda-Mahmood, T. 2018, in 2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018), IEEE, Washington, DC, USA, 1038–1042, doi: 10.1109/ISBI.2018.8363749

Malialis, K., Papatheodoulou, D., Filippou, S., Panayiotou, C., & Polycarpou, M. 2022, in 2022 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE, 1–1, doi: 10.1109/SSCI51031.2022.10022133

Mallick, J., Talukdar, S., & Ahmed, M. 2022, Applied Water Science, 12, 1, doi: 10.1007/s13201-022-01599-2

Okolo, G. I. 2022, Computer Methods and Programs in Biomedicine, 226, 107141, doi: https://doi.org/10.1016/j.cmpb.2022.107141

Sharma, C. M., Goyal, L., Chariar, V. M., & Sharma, N. 2022, Journal of Healthcare Engineering, 2022, 9036457, doi: 10.1155/2022/9036457

Shelke, A., Inamdar, M., Shah, V., et al. 2021, SN computer science, 2, 300

Soriano, J. B., et al. 2020, The Lancet. Respiratory medicine, 8, 585, doi: 10.1016/S2213-2600(20)30105-3

Stemerman, R., Arguello, J., Brice, J., et al. 2021, JAMIA Open, 4, ooaa069, doi: 10.1093/JAMIAOPEN/OOAA069

Tsoumakas, G., Katakis, I., & Vlahavas, I. 2010, Data mining and knowledge discovery handbook, 667

Uparkar, O., Bharti, J., Pateriya, R. K., Gupta, R. K., & Sharma, A. 2023, Procedia Computer Science, 218, 2338, doi: 10.1016/j.procs.2023.01.209

Yimer, F., Tessema, A., & Simegn, G. 2021, International Journal of Advanced Trends in Computer Science and Engineering, 10, 1–7

Zhang, Q., Bai, C., Liu, Z., et al. 2020, Information Sciences, 536, 91–100, doi: 10.1016/j.ins.2020.05.013

Appendix A

The First Appendix

A.1 Project Code Repository

all the code for deployment and model training are available on the github repository GitHub Project: chest_x_ray_classifier[1]

[1] https://github.com/HEMAL60/chest_x_ray_classifier.git