Pneumonia detection using Transfer learning

MSc Research Project
MSc in Artificial Intelligence

Pavan Kumar Govind
Student ID: x23229896

School of Computing
National College of Ireland

Supervisor: Kislay Raj

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Pavan Kumar Govind
Student ID:	x23229896
Programme:	M.Sc. in Artificial Intelligence
Year:	2024
Module:	MSc Research Project
Supervisor:	Kislay Raj
Submission Due Date:	12/12/2024
Project Title:	Pneumonia detection using Transfer Learning
Word Count:	XXX
Page Count:	16

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Pavan Kumar Govind
Date:	12th December 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Pavan Kumar Govind
x23229896

1 Introduction

This configuration manual will give step by step procedures in order to reproduce the experiments for the research study on “Pneumonia Detection Using Transfer Learning.” This manual describes software, tools and methods employed in the course of the research with regard to the application of transfer learning models for pneumonia detection. It also explains about customized code that contains many functions separately created to enhance the model performance and generate useful outputs for analysis. The manual is intended to help the researchers and practitioners identifying the configuration steps and the resources that are required for the purpose of this further research and development on the subject.

2 Hardware Overview

All research was performed in an Acer Aspire 5 laptop with the Windows 10 operating system. The system is equipped with an Intel Core i5 processor (specific model: i5-1135G7 model comprising 4 cores with 8 GB of RAM. The laptop also features 512 GB SSD as storage and has Intel Iris Xe Graphics for its graphical processing unit. They supplied the hardware needed to smoothly perform the experiments on pneumonia detected by transfer learning approach and to handle large data sets and train complex deep learning models.

3 Environment

Each of the experiments in this paper was conducted using the Python programming language. The experiments were designed and run as a Jupyter Notebook(.ipynb) in Google Colab environment. To support this, Google Colab was selected as the preferred environment to work in, it is cloud based, this eliminates incidents of running out of space to save and the use of GPU as a hardware accelerator was easily available, furthermore most packages in Python can be accessed directly from Colab without necessarily needing to install them locally. Since deep learning models require huge files and consume sizeable memories for training, this environment was sufficiently extensive without prior hardware limitations. To successfully accomplish the experiments described in the previous sections, all used code was written into an ipynb file, which was located in a Google Drive account. Google Drive and Colab can be freely used only if the user currently has an active Google account.

4 Dataset Source

In this project, dataset was collected from Kaggle where there are a collection of chest X-ray images which are classified as either normal or pneumonia Mooney (2018). It is, therefore, useful to determine that the dataset underpins an important opportunity for developing deep learning models in the medical imaging area, with emphasis on pneumonia identification in chest X-rays. The images in this dataset were pre-processed in order to be ready to be fed into deep learning models and to increase accuracy.

First of all, we put the chest X-ray dataset into the working space and investigate the contents. This dataset is stored in Google Drive and contains images classified into two categories: Normal and Pneumonia. For easy operation, we first engage Google Drive and unzip the document files of the dataset.

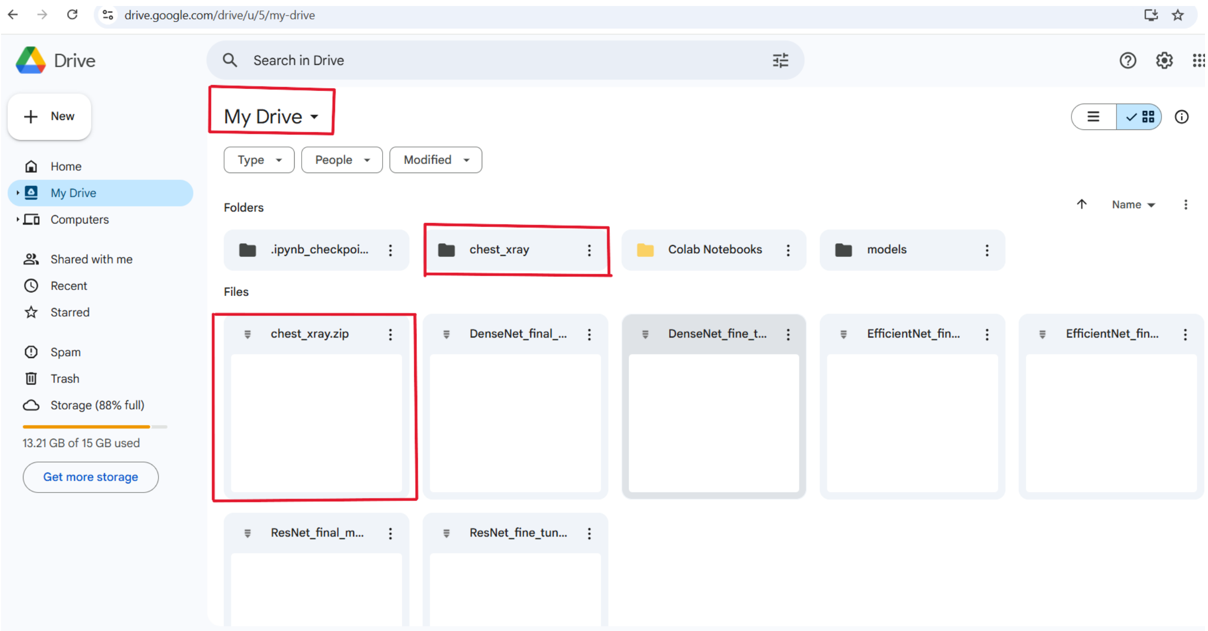

Figure 1: Dataset, Unzip dataset & saved models in Google drive

```python
# Step 1: Define the path to the zip file and output directory
zip_file_path = '/content/drive/MyDrive/chest_xray.zip'
output_directory = '/content/drive/MyDrive/chest_xray'

# Step 2: Check if the output directory already exists
if not os.path.exists(output_directory):
    print("Output directory not found. Unzipping the dataset...")
    !unzip -q "{zip_file_path}" -d "{output_directory}"
    print("Unzipping complete.")
else:
    print("Output directory already exists. Skipping the unzipping step.")

# Step 3: Verify the unzipping by listing the files in the output directory
print("Contents of the output directory:")
!ls "{output_directory}"

# Step 4: Define base_dir and paths for train, validation, and test directories
base_dir = os.path.join('/content/drive/MyDrive/chest_xray', 'chest_xray')
train_dir = os.path.join(base_dir, 'train')
val_dir = os.path.join(base_dir, 'val')
test_dir = os.path.join(base_dir, 'test')

print("Directories set:")
print(f"Train directory: {train_dir}")
print(f"Validation directory: {val_dir}")
print(f"Test directory: {test_dir}")

# Check if these directories exist
for dir_path in [train_dir, val_dir, test_dir]:
    if os.path.exists(dir_path):
        print(f"{dir_path} exists.")
```

Figure 2: Dataset loading and define path

```
Output directory already exists. Skipping the unzipping step.
Contents of the output directory:
chest_xray
Directories set:
Train directory: /content/drive/MyDrive/chest_xray/chest_xray/train
Validation directory: /content/drive/MyDrive/chest_xray/chest_xray/val
Test directory: /content/drive/MyDrive/chest_xray/chest_xray/test
/content/drive/MyDrive/chest_xray/chest_xray/train exists.
/content/drive/MyDrive/chest_xray/chest_xray/val exists.
/content/drive/MyDrive/chest_xray/chest_xray/test exists.
```

Figure 3: Output & loading dataset

5 Implementation

This section explains in detail the code to do the experiments and obtain the results of the Research Project included in this document. Results and discussion: The most important pieces of code for replication is presented visually using screenshots of the code, inputs and outputs.

Here's an updated version incorporating all the libraries you mentioned, written in a normal, cohesive format:

5.1 Establishment of environment

This research called for several python libraries and these where imported at the start of each session to facilitate a proper working environment as shown in the figure 1. Some of them were NumPy that is created to perform fastest multidimensional array computations, TensorFlow that can be used to build and train neural networks, lastly data visualization packages such as Matplotlib and Seaborn.

Importing libraries

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import DenseNet121, EfficientNetB0, ResNet50
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
```

Figure 4: importing libraries

5.2 Image augmentation

For image augmentation during preprocessing, data generators, the load image function and image to array function were employed. Popular transfer learning models: DenseNet121, EfficientNetB0, and ResNet50 were downloaded and imported, using Keras from TensorFlow. Furthermore, libraries like Scikit-learn were used in order to render performance metrics including the confusion matrix, classification report, ROC curve, and AUC.

Data Augmentation for the training set

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=30,
    width_shift_range=0.3,
    height_shift_range=0.3,
    shear_range=0.3,
    zoom_range=0.3,
    horizontal_flip=True,
    fill_mode='nearest',
    brightness_range=[0.2, 1.5]
)

test_datagen = ImageDataGenerator(rescale=1./255)
```

Figure 5: Data augmentation

- Rotation: The images were rotated up to 20 degrees, simulating slight variations in X- ray orientation.
- Width and Height Shifts: Additional small random displacements in the x & y coordinates for width and height areas were added to assist the model in alignment shifts.
- hearing: Shearing transformations were used to distort the model slightly to replicate minimal angulation.
- Zooming: The rotation of the images offered close-up view that allowed the model to interpret them differently.
- Horizontal Flipping: The increase of horizontal mirror images was used as a transition to make the dataset more diverse.

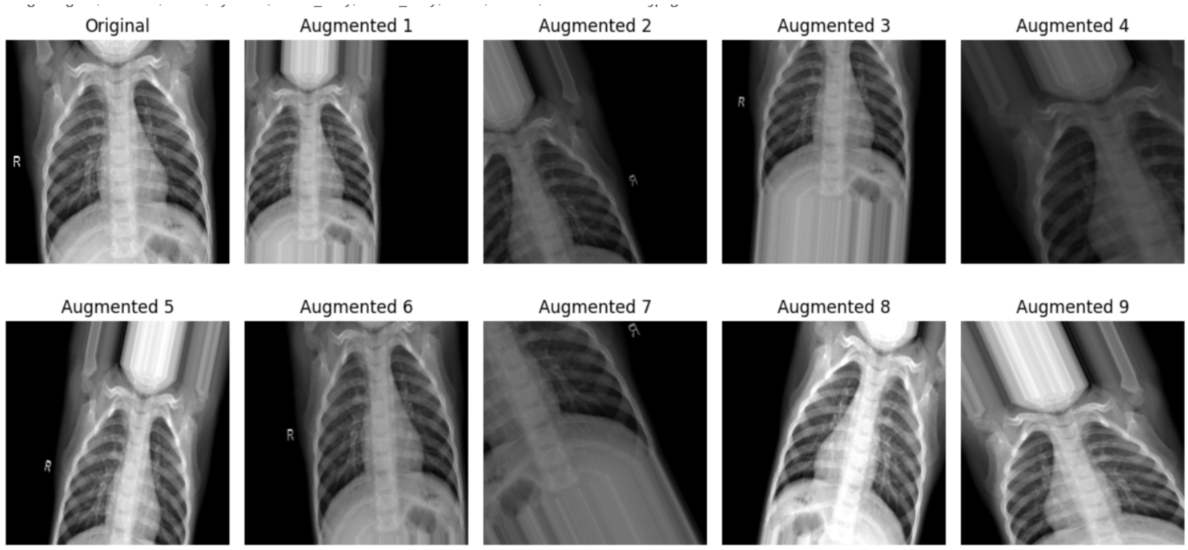

Figure 6: Sample augmentation output

5.3 Callback & Early stopping

Other callback functions including EarlyStopping were also incorporated so as to minimize overfitting during training. Preprocessing processes, model construction, and output assessment in all experiments were conducted with adjusted code implementation and a rigid procedure pipeline. After loading of the required packages and functions (Figure 2), environment was set to load the datasets and start the experiments.

```
[ ] from tensorflow.keras.callbacks import EarlyStopping

    # Add EarlyStopping callback to stop training when the validation loss stops improving
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=5,  # Number of epochs with no improvement after which training will stop
        restore_best_weights=True,  # Restore the model weights from the epoch with the best value of the monitored metric
        verbose=1
    )
```

Figure 7: Early stopping

5.4 model building

we define the architectures for three different models: DenseNet, EfficientNet, and ResNet.

```
[ ] def build_model(base_model):
        """
        Builds a binary classification model on top of a pre-trained base model.
        """
        x = base_model.output
        x = GlobalAveragePooling2D()(x)  # Adds global average pooling to reduce dimensions
        x = Dense(1024, activation='relu')(x)
        predictions = Dense(1, activation='sigmoid')(x)  # Output layer for binary classification
        model = Model(inputs=base_model.input, outputs=predictions)
        return model
```

Figure 8: Model building

5.5 model training

After defining the models, we can now proceed to train each of them.

```
# Train DenseNet model and visualize training
print("Training DenseNet Model...")
model_densenet = compile_and_train_model(model_densenet, 'DenseNet', learning_rate=0.0001, epochs=30)
```

Figure 9: DenseNet Training

```
Training DenseNet Model...
Epoch 1/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 877s 4s/step - accuracy: 0.8979 - loss: 0.2424 - val_accuracy: 0.7500 - val_loss: 0.8909
Epoch 2/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 156s 924ms/step - accuracy: 0.9475 - loss: 0.1270 - val_accuracy: 0.8750 - val_loss: 0.4171
Epoch 3/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 197s 885ms/step - accuracy: 0.9642 - loss: 0.0903 - val_accuracy: 0.9375 - val_loss: 0.1428
Epoch 4/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 202s 895ms/step - accuracy: 0.9702 - loss: 0.0810 - val_accuracy: 0.9375 - val_loss: 0.1136
Epoch 5/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 152s 891ms/step - accuracy: 0.9747 - loss: 0.0685 - val_accuracy: 1.0000 - val_loss: 0.0779
Epoch 6/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 204s 914ms/step - accuracy: 0.9729 - loss: 0.0678 - val_accuracy: 1.0000 - val_loss: 0.0430
Epoch 7/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 215s 985ms/step - accuracy: 0.9779 - loss: 0.0640 - val_accuracy: 1.0000 - val_loss: 0.0872
Epoch 8/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 161s 939ms/step - accuracy: 0.9757 - loss: 0.0721 - val_accuracy: 1.0000 - val_loss: 0.0421
Epoch 9/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 201s 939ms/step - accuracy: 0.9831 - loss: 0.0472 - val_accuracy: 1.0000 - val_loss: 0.0055
Epoch 10/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 162s 947ms/step - accuracy: 0.9809 - loss: 0.0612 - val_accuracy: 1.0000 - val_loss: 0.0380
Epoch 11/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 158s 929ms/step - accuracy: 0.9864 - loss: 0.0433 - val_accuracy: 0.9375 - val_loss: 0.1731
Epoch 12/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 159s 946ms/step - accuracy: 0.9775 - loss: 0.0547 - val_accuracy: 0.7500 - val_loss: 0.3101
Epoch 13/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 195s 901ms/step - accuracy: 0.9868 - loss: 0.0378 - val_accuracy: 1.0000 - val_loss: 0.0142
Epoch 14/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 157s 924ms/step - accuracy: 0.9855 - loss: 0.0389 - val_accuracy: 1.0000 - val_loss: 0.0319
```

Figure 10: DenseNet Test acc & loss

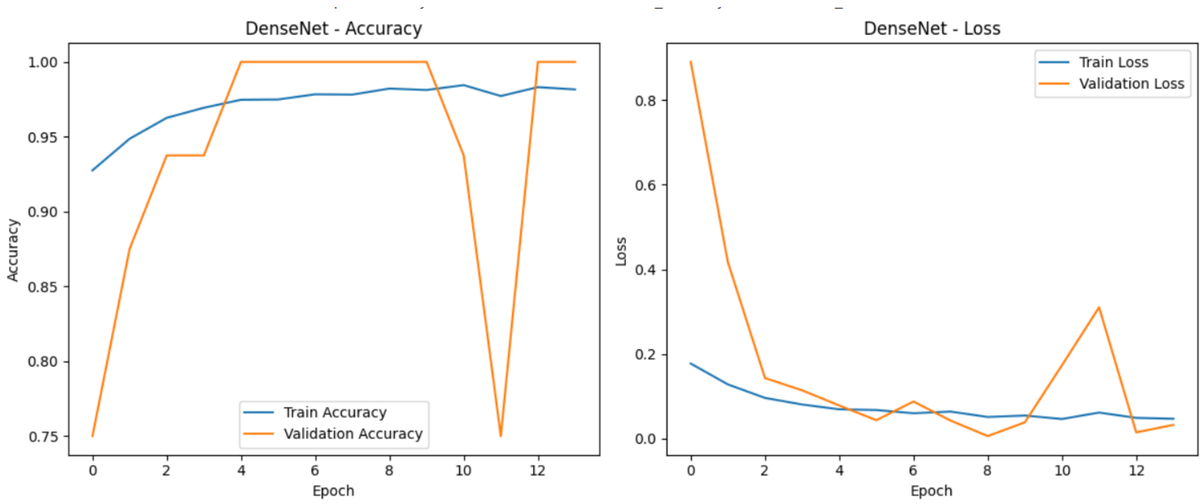

Figure 11: DenseNet Plot

```python
# Train EfficientNet model and visualize training
print("Training EfficientNet Model...")
model_efficientnet = compile_and_train_model(model_efficientnet, 'EfficientNet', learning_rate=0.0001, epochs=30)
```

Figure 12: EfficientNet Training

```
Training EfficientNet Model...
Epoch 1/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 245s 909ms/step - accuracy: 0.8493 - loss: 0.3285 - val_accuracy: 0.5625 - val_loss: 0.7808
Epoch 2/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 213s 934ms/step - accuracy: 0.9517 - loss: 0.1361 - val_accuracy: 0.5000 - val_loss: 1.1781
Epoch 3/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 146s 853ms/step - accuracy: 0.9541 - loss: 0.1108 - val_accuracy: 0.5000 - val_loss: 2.4057
Epoch 4/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 143s 849ms/step - accuracy: 0.9640 - loss: 0.0868 - val_accuracy: 0.6875 - val_loss: 0.5302
Epoch 5/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 205s 859ms/step - accuracy: 0.9681 - loss: 0.0836 - val_accuracy: 0.8750 - val_loss: 0.2558
Epoch 6/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 138s 814ms/step - accuracy: 0.9784 - loss: 0.0571 - val_accuracy: 1.0000 - val_loss: 0.0549
Epoch 7/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 144s 828ms/step - accuracy: 0.9754 - loss: 0.0670 - val_accuracy: 0.9375 - val_loss: 0.0964
Epoch 8/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 140s 815ms/step - accuracy: 0.9749 - loss: 0.0662 - val_accuracy: 1.0000 - val_loss: 0.0312
Epoch 9/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 139s 820ms/step - accuracy: 0.9744 - loss: 0.0614 - val_accuracy: 1.0000 - val_loss: 0.0978
Epoch 10/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 141s 827ms/step - accuracy: 0.9769 - loss: 0.0613 - val_accuracy: 0.8750 - val_loss: 0.1747
Epoch 11/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 145s 848ms/step - accuracy: 0.9780 - loss: 0.0543 - val_accuracy: 0.8750 - val_loss: 0.1671
Epoch 12/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 142s 830ms/step - accuracy: 0.9755 - loss: 0.0696 - val_accuracy: 0.7500 - val_loss: 0.2921
Epoch 13/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 137s 800ms/step - accuracy: 0.9819 - loss: 0.0498 - val_accuracy: 1.0000 - val_loss: 0.0744
```

Figure 13: EfficientNet Test acc & loss

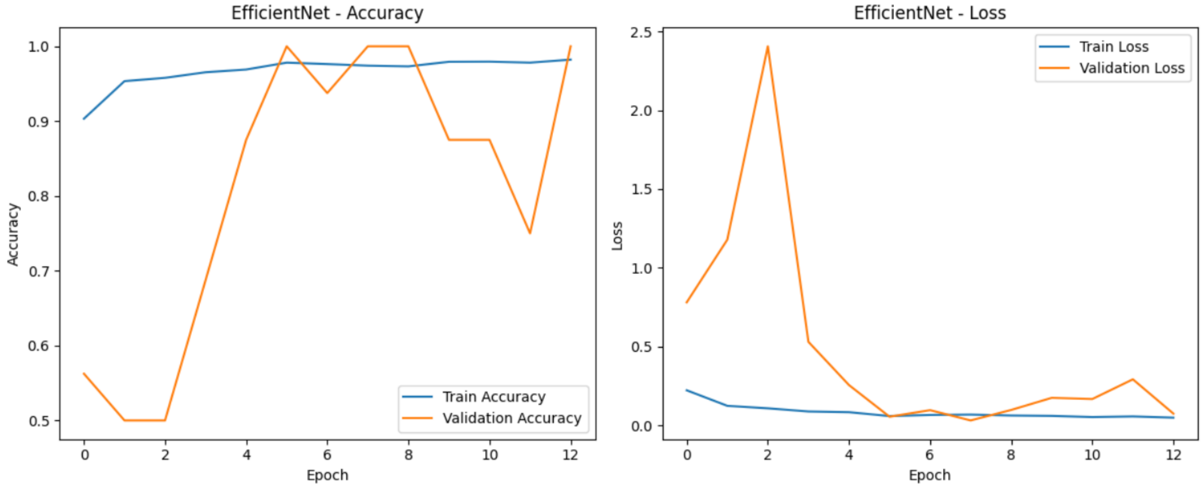

Figure 14: EfficientNet Plot

```python
# Train ResNet model and visualize training
print("Training ResNet Model...")
model_resnet = compile_and_train_model(model_resnet, 'ResNet', learning_rate=0.0001, epochs=30)
```

Figure 15: ResNet Training

```
Training ResNet Model...
Epoch 1/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 223s 933ms/step - accuracy: 0.8840 - loss: 0.2724 - val_accuracy: 0.5000 - val_loss: 0.9357
Epoch 2/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 167s 921ms/step - accuracy: 0.9473 - loss: 0.1282 - val_accuracy: 0.3750 - val_loss: 0.8054
Epoch 3/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 196s 889ms/step - accuracy: 0.9619 - loss: 0.1022 - val_accuracy: 0.5000 - val_loss: 1.6334
Epoch 4/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 212s 949ms/step - accuracy: 0.9706 - loss: 0.0761 - val_accuracy: 0.5000 - val_loss: 5.9751
Epoch 5/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 196s 911ms/step - accuracy: 0.9677 - loss: 0.0865 - val_accuracy: 0.5000 - val_loss: 9.7571
Epoch 6/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 204s 931ms/step - accuracy: 0.9753 - loss: 0.0781 - val_accuracy: 0.5000 - val_loss: 2.4017
Epoch 7/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 150s 887ms/step - accuracy: 0.9750 - loss: 0.0685 - val_accuracy: 0.7500 - val_loss: 0.3367
Epoch 8/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 206s 907ms/step - accuracy: 0.9756 - loss: 0.0620 - val_accuracy: 0.9375 - val_loss: 0.1260
Epoch 9/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 152s 904ms/step - accuracy: 0.9774 - loss: 0.0640 - val_accuracy: 1.0000 - val_loss: 0.0181
Epoch 10/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 163s 952ms/step - accuracy: 0.9774 - loss: 0.0648 - val_accuracy: 0.9375 - val_loss: 0.2502
Epoch 11/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 155s 909ms/step - accuracy: 0.9798 - loss: 0.0502 - val_accuracy: 1.0000 - val_loss: 0.0024
Epoch 12/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 200s 906ms/step - accuracy: 0.9823 - loss: 0.0556 - val_accuracy: 1.0000 - val_loss: 0.0158
Epoch 13/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 200s 871ms/step - accuracy: 0.9834 - loss: 0.0497 - val_accuracy: 0.9375 - val_loss: 0.0841
Epoch 14/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 210s 941ms/step - accuracy: 0.9865 - loss: 0.0424 - val_accuracy: 1.0000 - val_loss: 0.0268
Epoch 15/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 190s 867ms/step - accuracy: 0.9862 - loss: 0.0389 - val_accuracy: 0.6875 - val_loss: 0.9543
Epoch 16/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 157s 924ms/step - accuracy: 0.9843 - loss: 0.0438 - val_accuracy: 1.0000 - val_loss: 0.0164
```

Figure 16: Resnet Test acc & loss

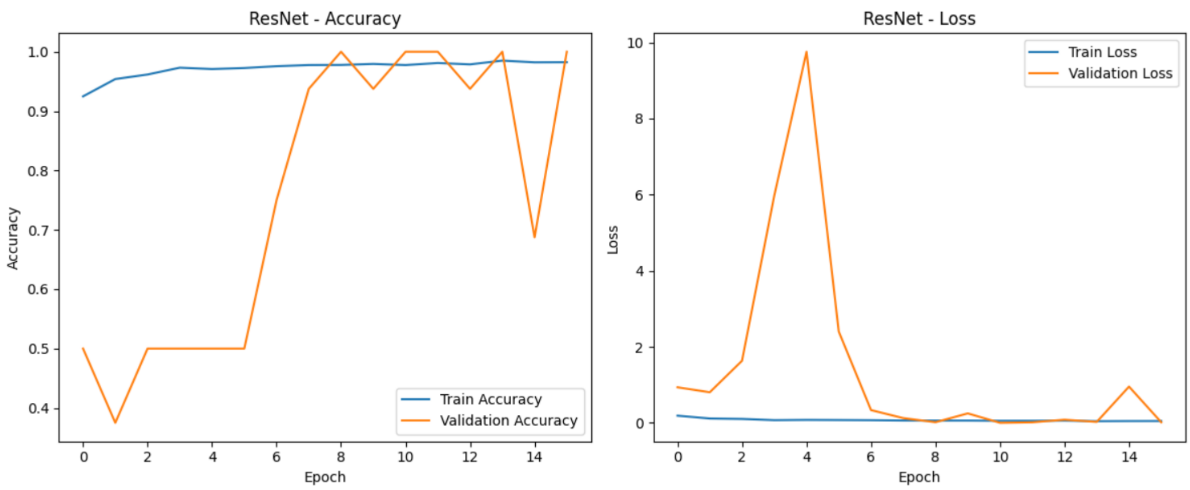

Figure 17: ResNet Plot

5.6 model saving

Once the models are trained, we save them for future use and evaluation.

```
# Save the trained DenseNet model to Google Drive
model_densenet.save('/content/drive/MyDrive/DenseNet_final_model.keras')  # Save in .keras format
print("DenseNet model saved to Google Drive.")
```

Figure 18: DenseNet Model saving to drive

```
# Save the trained EfficientNet model to Google Drive
model_efficientnet.save('/content/drive/MyDrive/EfficientNet_final_model.keras')  # Save in .keras format
print("EfficientNet model saved to Google Drive.")
```

Figure 19: EfficientNet Model saving to drive

```
# Save the trained ResNet model to Google Drive
model_resnet.save('/content/drive/MyDrive/ResNet_final_model.keras')  # Save in .keras format
print("ResNet model saved to Google Drive.")
```

Figure 20: ResNet Model saving to drive

5.7 model loading and traing with fine tuning

Once the models are savsed, we load them for fine tuning evaluation.

```python
# Load the previously saved models
densenet_model_path = '/content/drive/MyDrive/DenseNet_final_model.keras'
efficientnet_model_path = '/content/drive/MyDrive/EfficientNet_final_model.keras'
resnet_model_path = '/content/drive/MyDrive/ResNet_final_model.keras'

model_densenet = load_model(densenet_model_path)
model_efficientnet = load_model(efficientnet_model_path)
model_resnet = load_model(resnet_model_path)

# Define the path to your chest_xray dataset
base_dir = '/content/drive/MyDrive/chest_xray/chest_xray'

# Path to training, validation, and test directories
train_dir = os.path.join(base_dir, 'train')
val_dir = os.path.join(base_dir, 'val')
test_dir = os.path.join(base_dir, 'test')  # Path to the test data

# Set image size and batch size
IMG_SIZE = (224, 224)
BATCH_SIZE = 32

# Prepare the ImageDataGenerator for rescaling
test_datagen = ImageDataGenerator(rescale=1./255)
```

Figure 21: Loading saved models for fine-tuning

The below defines a Python function, evaluatemodel, which evaluates a trained deep learning model on a test dataset. The function generates key performance metrics and visualizations to assess the model's classification performance.

```python
# Evaluate the model using a custom threshold
def evaluate_model(model, test_generator, model_name, threshold=0.5):
    """
    Evaluates the model on the test data and generates various metrics like classification report,
    confusion matrix, and ROC curve.

    Args:
    - model: The trained model
    - test_generator: The data generator for the test data
    - model_name: The name of the model (for display purposes)
    - threshold: The threshold for classification (default: 0.5)
    """
    # Get true labels
    y_true = test_generator.classes

    # Get predicted probabilities
    y_pred_prob = model.predict(test_generator, verbose=1)

    # Convert probabilities to binary predictions using the threshold
    y_pred = (y_pred_prob > threshold).astype(int).flatten()

    # Classification Report
    print(f"{model_name} Classification Report (Threshold: {threshold}):")
    print(classification_report(y_true, y_pred, target_names=['Normal', 'Pneumonia']))
```

Figure 22: Evaluate and classification report

```
#Evaluate models with adjusted threshold (e.g., 0.3)

print("Evaluating DenseNet with adjusted threshold 0.3:")
evaluate_model(model_densenet, test_generator, 'DenseNet', threshold=0.3)

print("Evaluating EfficientNet with adjusted threshold 0.3:")
evaluate_model(model_efficientnet, test_generator, 'EfficientNet', threshold=0.3)

print("Evaluating ResNet with adjusted threshold 0.3:")
evaluate_model(model_resnet, test_generator, 'ResNet', threshold=0.3)
```

Figure 23: Evaluate fine-tuning models

```
Fine-tuning DenseNet Model...
Epoch 1/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 57s 2s/step - accuracy: 0.9331 - loss: 0.1962 - val_accuracy: 0.9038 - val_loss: 0.2692
Epoch 2/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 15s 660ms/step - accuracy: 0.8884 - loss: 0.3470 - val_accuracy: 0.9038 - val_loss: 0.2613
Epoch 3/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 24s 807ms/step - accuracy: 0.8644 - loss: 0.3890 - val_accuracy: 0.9087 - val_loss: 0.2548
Epoch 4/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 17s 655ms/step - accuracy: 0.8789 - loss: 0.3178 - val_accuracy: 0.9119 - val_loss: 0.2496
Epoch 5/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 15s 637ms/step - accuracy: 0.8910 - loss: 0.3009 - val_accuracy: 0.9151 - val_loss: 0.2442
Fine-tuned DenseNet model saved to: /content/drive/MyDrive/DenseNet_fine_tuned.keras
Fine-tuning EfficientNet Model...
Epoch 1/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 54s 2s/step - accuracy: 0.6675 - loss: 1834306.0000 - val_accuracy: 0.6250 - val_loss: 989264.3125
Epoch 2/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 53s 637ms/step - accuracy: 0.6737 - loss: 1490716.7500 - val_accuracy: 0.6250 - val_loss: 919060.4375
Epoch 3/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 18s 768ms/step - accuracy: 0.6382 - loss: 1582678.7500 - val_accuracy: 0.6250 - val_loss: 842986.4375
Epoch 4/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 17s 627ms/step - accuracy: 0.6955 - loss: 1067287.5000 - val_accuracy: 0.6250 - val_loss: 780935.1250
Epoch 5/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 18s 825ms/step - accuracy: 0.6967 - loss: 1106307.1250 - val_accuracy: 0.6250 - val_loss: 710208.1250
Fine-tuned EfficientNet model saved to: /content/drive/MyDrive/EfficientNet_fine_tuned.keras
```

Figure 24: DenseNet & EfficientNet Test acc & loss

```
Fine-tuning ResNet Model...
Epoch 1/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 32s 1s/step - accuracy: 0.8845 - loss: 0.3699 - val_accuracy: 0.8974 - val_loss: 0.3451
Epoch 2/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 15s 557ms/step - accuracy: 0.9232 - loss: 0.2471 - val_accuracy: 0.8990 - val_loss: 0.3301
Epoch 3/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 23s 787ms/step - accuracy: 0.8227 - loss: 0.6446 - val_accuracy: 0.9038 - val_loss: 0.3111
Epoch 4/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 22s 786ms/step - accuracy: 0.8359 - loss: 0.5903 - val_accuracy: 0.9071 - val_loss: 0.2975
Epoch 5/5
20/20 ━━━━━━━━━━━━━━━━━━━━ 17s 676ms/step - accuracy: 0.9256 - loss: 0.2186 - val_accuracy: 0.9135 - val_loss: 0.2878
Fine-tuned ResNet model saved to: /content/drive/MyDrive/ResNet_fine_tuned.keras
```

Figure 25: ResNet Test acc & loss

6 Results

```
Evaluating DenseNet with adjusted threshold 0.3:
20/20 ━━━━━━━━━━━━━━━━━━━━ 27s 885ms/step
DenseNet Classification Report (Threshold: 0.3):
              precision    recall  f1-score   support

      Normal       0.92      0.85      0.88       234
   Pneumonia       0.91      0.95      0.93       390

    accuracy                           0.91       624
   macro avg       0.91      0.90      0.91       624
weighted avg       0.91      0.91      0.91       624
```

Figure 26: DenseNet Classification report

```
                                  False Positive Rate
Evaluating EfficientNet with adjusted threshold 0.3:
20/20 ━━━━━━━━━━━━━━━━━━━━ 20s 646ms/step
EfficientNet Classification Report (Threshold: 0.3):
              precision    recall  f1-score   support

      Normal       0.00      0.00      0.00       234
   Pneumonia       0.62      1.00      0.77       390

    accuracy                           0.62       624
   macro avg       0.31      0.50      0.38       624
weighted avg       0.39      0.62      0.48       624
```

Figure 27: EfficientNet Classification report

```
Evaluating ResNet with adjusted threshold 0.3:
20/20 ━━━━━━━━━━━━━━━━━━━━ 13s 450ms/step
ResNet Classification Report (Threshold: 0.3):
              precision    recall  f1-score   support

      Normal       0.95      0.75      0.84       234
   Pneumonia       0.87      0.98      0.92       390

    accuracy                           0.89       624
   macro avg       0.91      0.86      0.88       624
weighted avg       0.90      0.89      0.89       624
```

Figure 28: ResNet Classification report

7 Grad CAM

To further understand the model's decision-making process, we use Grad-CAM (Gradient-weighted Class Activation Mapping) to visualize which parts of the chest X-ray images the model is focusing on. This helps in interpreting the model's predictions.

```python
def make_gradcam_heatmap(img_array, model, last_conv_layer_name):
    """
    Generate Grad-CAM heatmap to visualize model focus.
    Args:
        img_array: Preprocessed input image array.
        model: Trained model.
        last_conv_layer_name: Name of the last convolutional layer.
    Returns:
        Heatmap indicating areas of model focus.
    """
    grad_model = Model(inputs=model.input, outputs=[model.get_layer(last_conv_layer_name).output, model.output])

    with tf.GradientTape() as tape:
        conv_outputs, predictions = grad_model(img_array)
        loss = predictions[:, 0]

    grads = tape.gradient(loss, conv_outputs)[0]
    pooled_grads = np.mean(grads, axis=(0, 1))  # Compute average gradient per channel
    conv_outputs = conv_outputs[0]  # Get the output of the last convolutional layer for the image

    # Reshape pooled_grads to [1, 1, channels] for broadcasting
    pooled_grads = pooled_grads[np.newaxis, np.newaxis, :]  # Shape: [1, 1, channels]

    # Element-wise multiplication using broadcasting
    conv_outputs = conv_outputs * pooled_grads  # Broadcasting happens here

    heatmap = np.mean(conv_outputs, axis=-1)  # Average across channels to get the heatmap
    heatmap = np.maximum(heatmap, 0)  # Remove negative values (we only care about positive importance)
    heatmap = heatmap / np.max(heatmap)  # Normalize the heatmap to [0, 1]

    return heatmap
```

Figure 29: Grad CAM

```python
import matplotlib.pyplot as plt
import cv2

def display_gradcam(img_array, model, last_conv_layer_name, class_index):
    """
    Displays Grad-CAM heatmap overlayed on input image.
    Args:
        img_array: Preprocessed image array.
        model: Trained model.
        last_conv_layer_name: Last convolution layer name.
        class_index: Index of the class to visualize.
    """
    heatmap = make_gradcam_heatmap(img_array, model, last_conv_layer_name)

    # Rescale heatmap to range [0, 255]
    heatmap = cv2.resize(heatmap, (img_array.shape[2], img_array.shape[1]))
    heatmap = np.uint8(255 * heatmap)

    # Apply a colormap to the heatmap
    heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)

    # Convert the input image to RGB
    img_array_rgb = img_array[0] * 255  # Reverse the preprocessing

    # Overlay the heatmap on the image
    superimposed_img = cv2.addWeighted(img_array_rgb.astype(np.uint8), 0.6, heatmap, 0.4, 0)
```

Figure 30: Enter Caption

```python
    # Display the image
    plt.imshow(superimposed_img)
    plt.axis('off')  # No axes for the image
    plt.show()

# Example usage:
img, label = next(test_generator)  # Fetch a batch using the built-in next() method
sample_img = np.expand_dims(img[0], axis=0)  # Take one image from the batch
display_gradcam(sample_img, model_densenet, 'conv5_block16_2_conv', class_index=1)
```

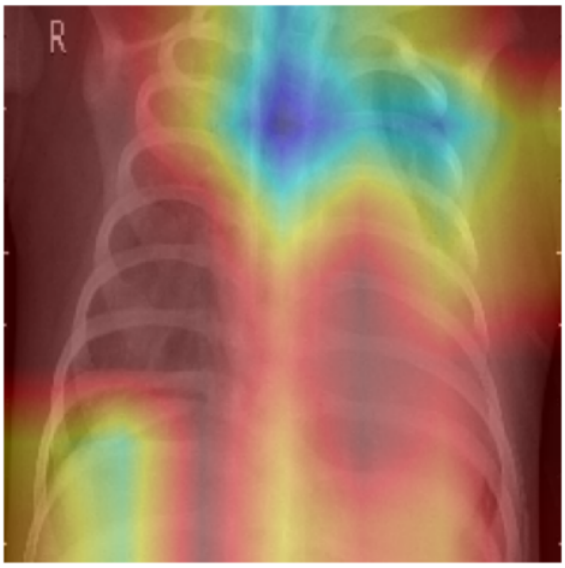

Figure 31: Grad CAM sample output

References

Mooney, P. (2018). Chest x-ray images (pneumonia). Available: https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia?resource=download.