1.0 Introduction

Deep learning is a subfield of machine learning that is based on artificial neural networks, and is a form of computational model that was developed in imitation of the workings of the human brain. This contributes to why deep learning works so well. Deep learning systems can learn complex patterns and relationships in data, which makes them mostly useful in tasks with vast amounts of data. A standard deep neural network includes an input layer, one or more hidden layers, and an output layer, all of this layer is interconnected by nodes or neurons. These layers act sequentially, with the input layer receiving raw data, the hidden layers extracting features and patterns, and the output layer producing predictions or classifications. The more layers this deep learning model has, the more abstraction it can learn about the data and therefore the more complex relationships can be captured.

Convolutional Neural Networks (CNNs) are a special kind of artificial neural network which processes data with a grid-like topology. This makes it particularly suitable for tasks with an image as input. The architecture of a CNN is inspired by the visual cortex in animals and designed to automatically learn and adaptively extract features in an image. A typical CNN has several layers which includes convolutional layers, pooling layer and the fully connected layers. In the convolutional layers, filters or kernels slide over the input image to detect various features such as edges, textures and shapes by computing their convolutions. This is how a CNN captures local patterns and spatial relationships effectively. Pooling layers, often following a convolutional layer decrease the dimensionality of your representation by down sampling which keeps most of what you want to have in the next layer while reducing computational complexity because it doesn't use as many neurons and at same time prevents over-fitting. The final layer of a CNN is usually a fully connected layer that takes the high-level features generated by previous groups and makes predictions based on those classifications.

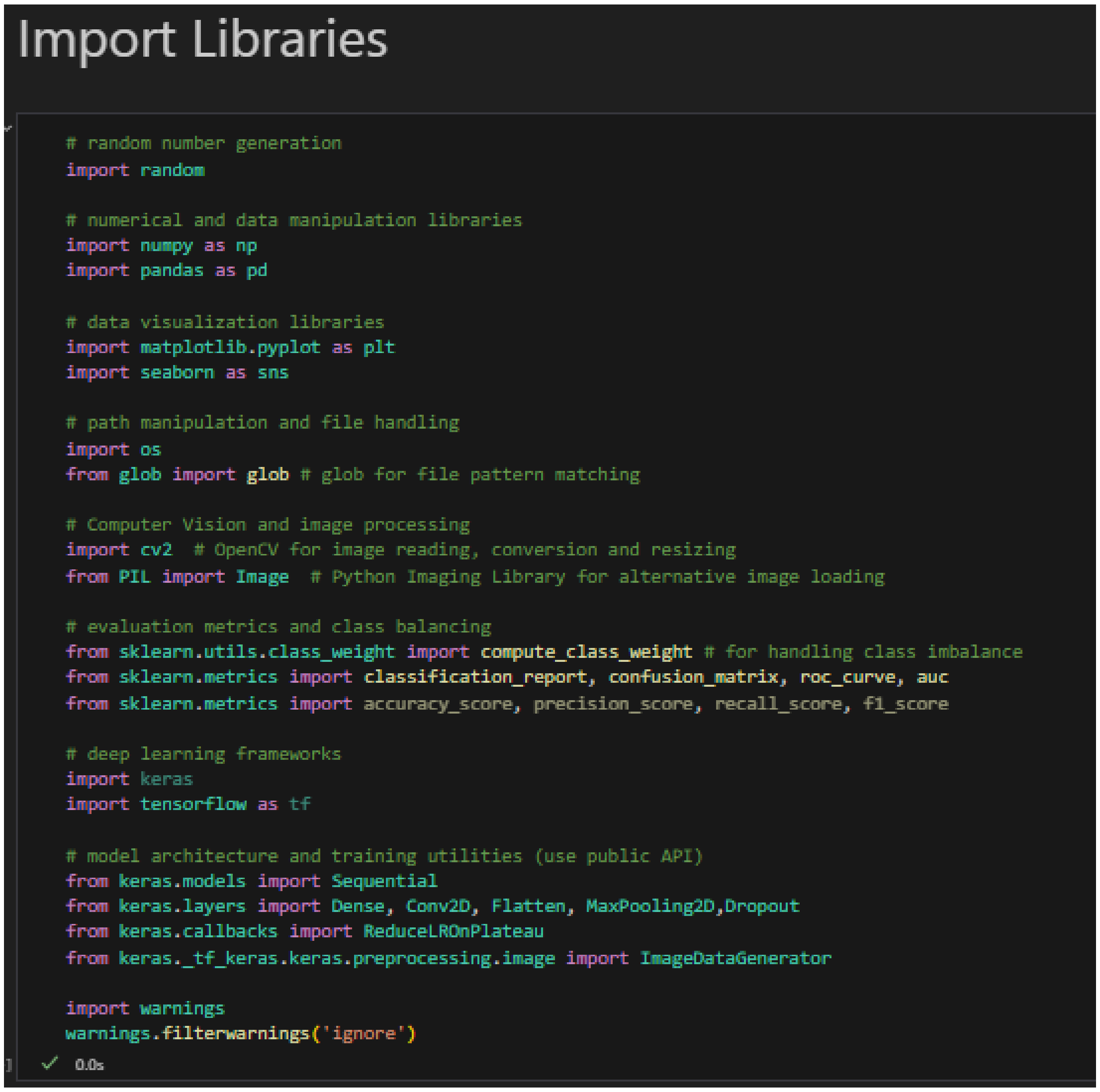
Lung diseases such as pneumonia are an inflammatory condition that affects the lungs tiny air sacs called the alveoli. It is usually due to infection with viruses or bacteria and characterized by the filling up the alveoli in lungs with fluid, which is causing the breathing difficult. Pneumonia is one of the leading causes of death globally, and especially for children under five. Timely diagnosis and treatment of pneumonia are essential for better patient outcomes. Chest X-rays are commonly used for the diagnosis of pneumonia, however, the interpretation of X-ray images requires specialized skills, training and experience. The automatic detection systems based on deep learning can be used as aids for the radiologists to diagnose more quickly and accurately. Hence, in this study we will create a Convolutional Neural Network model that can identify pneumonia from the chest X-ray images. The version of dataset used is adapted version of Paul Mooney's Pneumonia X-Ray Images. The dataset has been restructured to ensure a more balanced distribution between training, validation, and testing subsets.

This report investigates the application of Convolutional Neural Network to the Pneumonia X-ray image dataset, with the following objectives:

1. To develop a deep learning model that can automatically classify chest X-ray images, it is either healthy lungs or infected lungs.
2. To enhance the accuracy and efficiency of pneumonia diagnosis by reducing radiologist workload in clinical settings and improve patient outcomes.

2.0 Model Setup and Implementation

2.1 Import Libraries



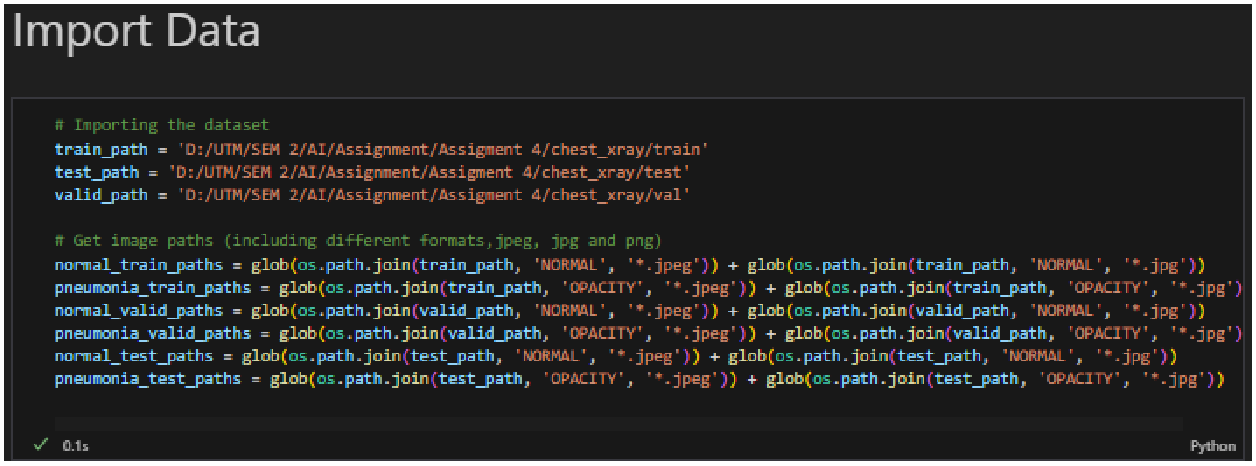
Import Libraries

```python
# random number generation
import random

# numerical and data manipulation libraries
import numpy as np
import pandas as pd

# data visualization libraries
import matplotlib.pyplot as plt
import seaborn as sns

# path manipulation and file handling
import os
from glob import glob # glob for file pattern matching

# Computer Vision and image processing
import cv2  # OpenCV for image reading, conversion and resizing
from PIL import Image  # Python Imaging Library for alternative image loading

# evaluation metrics and class balancing
from sklearn.utils.class_weight import compute_class_weight # for handling class imbalance
from sklearn.metrics import classification_report, confusion_matrix, roc_curve, auc
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# deep learning frameworks
import keras
import tensorflow as tf

# model architecture and training utilities (use public API)
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D,Dropout
from keras.callbacks import ReduceLROnPlateau
from keras._tf_keras.keras.preprocessing.image import ImageDataGenerator

import warnings
warnings.filterwarnings('ignore')
```

✓ 0.0s

First import the necessary library for running the CNN model.

2.2 Import Dataset



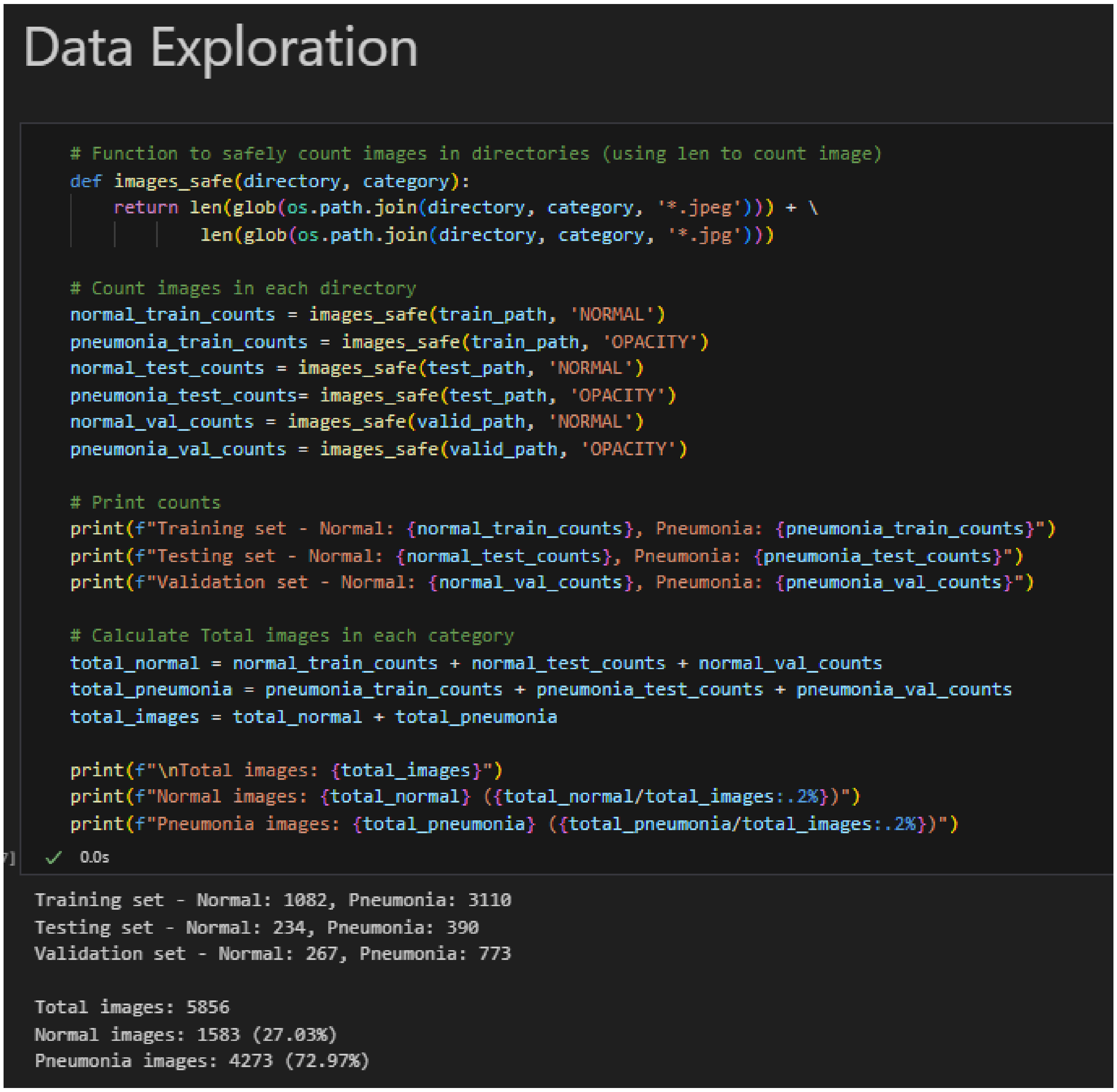
Import Data

```
# Importing the dataset
train_path = 'D:/UTM/SEM 2/AI/Assignment/Assigment 4/chest_xray/train'
test_path = 'D:/UTM/SEM 2/AI/Assignment/Assigment 4/chest_xray/test'
valid_path = 'D:/UTM/SEM 2/AI/Assignment/Assigment 4/chest_xray/val'

# Get image paths (including different formats,jpeg, jpg and png)
normal_train_paths = glob(os.path.join(train_path, 'NORMAL', '*.jpeg')) + glob(os.path.join(train_path, 'NORMAL', '*.jpg'))
pneumonia_train_paths = glob(os.path.join(train_path, 'OPACITY', '*.jpeg')) + glob(os.path.join(train_path, 'OPACITY', '*.jpg')
normal_valid_paths = glob(os.path.join(valid_path, 'NORMAL', '*.jpeg')) + glob(os.path.join(valid_path, 'NORMAL', '*.jpg'))
pneumonia_valid_paths = glob(os.path.join(valid_path, 'OPACITY', '*.jpeg')) + glob(os.path.join(valid_path, 'OPACITY', '*.jpg')
normal_test_paths = glob(os.path.join(test_path, 'NORMAL', '*.jpeg')) + glob(os.path.join(test_path, 'NORMAL', '*.jpg'))
pneumonia_test_paths = glob(os.path.join(test_path, 'OPACITY', '*.jpeg')) + glob(os.path.join(test_path, 'OPACITY', '*.jpg'))
```

✓ 0.1s Python

Next, import the dataset from the local directory and update these paths according to dataset location. The dataset is organized into 3 folders (train, test, val) and contains subfolders for each image category. Then, use glob() to match the file patterns from different formats including jpeg and jpg. This will help in loading images from the specified directories likewise, normal images are in 'NORMAL' folder and pneumonia images are in 'OPACITY' folder. This is useful for loading images from directories where the file names may vary. The paths are then concatenated to create a single list of file paths for each category.

2.3 Data Exploration



Data Exploration

```
# Function to safely count images in directories (using len to count image)
def images_safe(directory, category):
    return len(glob(os.path.join(directory, category, '*.jpeg'))) + \
            len(glob(os.path.join(directory, category, '*.jpg')))

# Count images in each directory
normal_train_counts = images_safe(train_path, 'NORMAL')
pneumonia_train_counts = images_safe(train_path, 'OPACITY')
normal_test_counts = images_safe(test_path, 'NORMAL')
pneumonia_test_counts= images_safe(test_path, 'OPACITY')
normal_val_counts = images_safe(valid_path, 'NORMAL')
pneumonia_val_counts = images_safe(valid_path, 'OPACITY')

# Print counts
print(f"Training set - Normal: {normal_train_counts}, Pneumonia: {pneumonia_train_counts}")
print(f"Testing set - Normal: {normal_test_counts}, Pneumonia: {pneumonia_test_counts}")
print(f"Validation set - Normal: {normal_val_counts}, Pneumonia: {pneumonia_val_counts}")

# Calculate Total images in each category
total_normal = normal_train_counts + normal_test_counts + normal_val_counts
total_pneumonia = pneumonia_train_counts + pneumonia_test_counts + pneumonia_val_counts
total_images = total_normal + total_pneumonia

print(f"\nTotal images: {total_images}")
print(f"Normal images: {total_normal} ({total_normal/total_images:.2%})")
print(f"Pneumonia images: {total_pneumonia} ({total_pneumonia/total_images:.2%})")
```

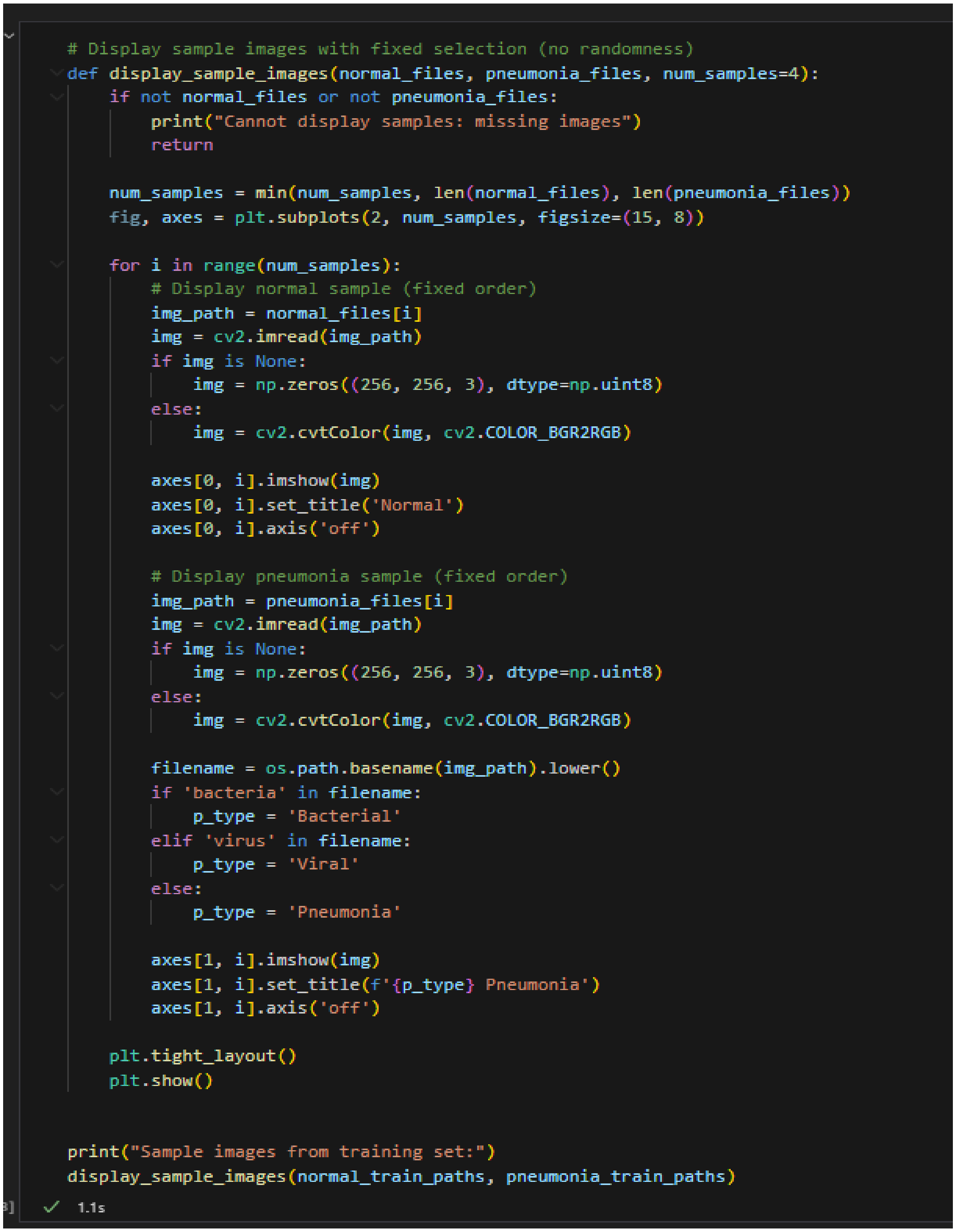
✓ 0.0s

```
Training set - Normal: 1082, Pneumonia: 3110
Testing set - Normal: 234, Pneumonia: 390
Validation set - Normal: 267, Pneumonia: 773

Total images: 5856
Normal images: 1583 (27.03%)
Pneumonia images: 4273 (72.97%)
```

Next part is data exploration. The images_safe function is created to takes the directory and a category (subfolder name), then uses the glob module to count all .jpeg and .jpg files in that subfolder. This helps ensure that images with either extension are included in the count. Then applies this function to each relevant subfolder in the training, testing, and validation directories. It will be storing the counts for both the "NORMAL" and "OPACITY" (pneumonia) categories. These counts are printed for each dataset split, giving a clear overview of how many images are available for each class in each set. After printing the individual counts, we calculate the total number of normal (total_normal) and pneumonia images (total_pneumonia) by summing across all splits and then computing the overall total number of images (total_images). Finally, prints the total counts and the percentage of images that belong to each class.

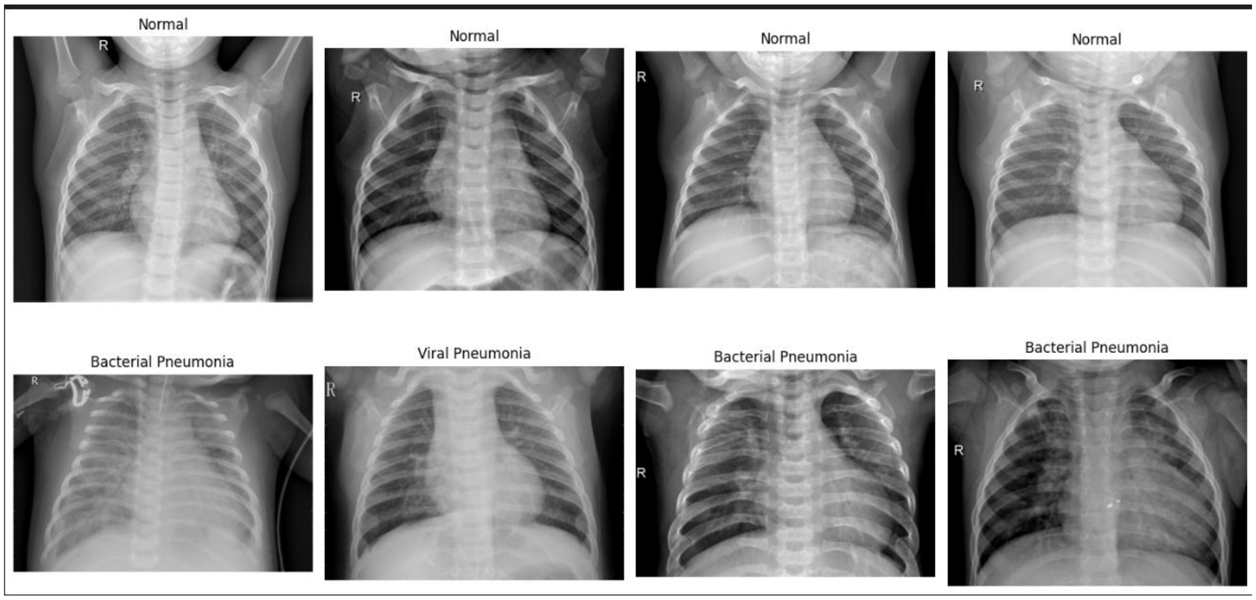
The result shows the training set has 1082 normal images and 3110 pneumonia images, the validation set has 267 normal image and 773 pneumonia images and the testing set has 234 normal image and 390 pneumonia images. There are 5,863 X-Ray images in total, among that, normal image has 1583 image which is 27.03% and pneumonia image has 4273 image which is 72.97%. It shows that the dataset is significantly imbalance between normal and pneumonia images, with pneumonia cases being more prevalent. This imbalance needs to be addressed during model training.

2.3.1 Image Visualize on Train dataset



```python
# Display sample images with fixed selection (no randomness)
def display_sample_images(normal_files, pneumonia_files, num_samples=4):
    if not normal_files or not pneumonia_files:
        print("Cannot display samples: missing images")
        return

    num_samples = min(num_samples, len(normal_files), len(pneumonia_files))
    fig, axes = plt.subplots(2, num_samples, figsize=(15, 8))

    for i in range(num_samples):
        # Display normal sample (fixed order)
        img_path = normal_files[i]
        img = cv2.imread(img_path)
        if img is None:
            img = np.zeros((256, 256, 3), dtype=np.uint8)
        else:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        axes[0, i].imshow(img)
        axes[0, i].set_title('Normal')
        axes[0, i].axis('off')

        # Display pneumonia sample (fixed order)
        img_path = pneumonia_files[i]
        img = cv2.imread(img_path)
        if img is None:
            img = np.zeros((256, 256, 3), dtype=np.uint8)
        else:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        filename = os.path.basename(img_path).lower()
        if 'bacteria' in filename:
            p_type = 'Bacterial'
        elif 'virus' in filename:
            p_type = 'Viral'
        else:
            p_type = 'Pneumonia'

        axes[1, i].imshow(img)
        axes[1, i].set_title(f'{p_type} Pneumonia')
        axes[1, i].axis('off')

    plt.tight_layout()
    plt.show()


print("Sample images from training set:")
display_sample_images(normal_train_paths, pneumonia_train_paths)
```

✓ 1.1s

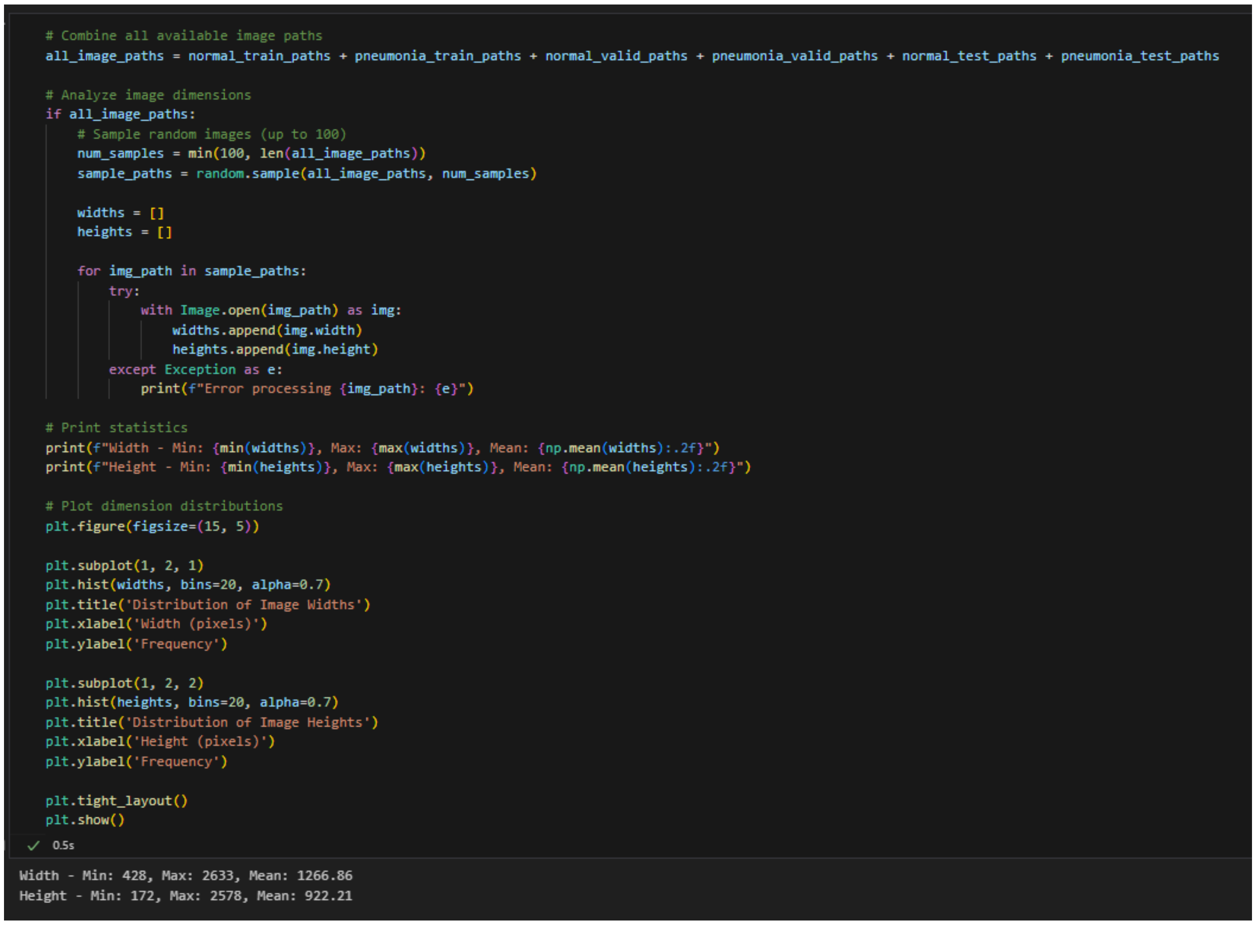


Next is visualizing the x-ray image in the train dataset. We define the display_sample_images function so that the function can visually display a grid of sample images from two categories, normal and pneumonia. The function takes lists of file paths for normal and pneumonia images, and an optional number of samples to show (default is 4). It first checks if both lists are non-empty and if both are missing, it prints a warning and exits.

The function determines the number of samples to display, ensuring it does not exceed the available images in either category. It then creates a 2-row subplot, the first row displays normal images, and the second row displays pneumonia images. For each sample, it loads the image using OpenCV. If the image cannot be loaded, it substitutes a blank image to avoid errors. Images are converted from BGR (OpenCV's default) to RGB for correct color display with Matplotlib.

For pneumonia images, the function inspects the filename to label the sample as "Bacterial," "Viral," or just "Pneumonia" if neither keyword is found. Each image is shown without axis ticks, and the layout is adjusted for clarity. Finally, the function "display_sample_images" is called to display sample images from the training set, providing a quick visual check of the dataset's contents and class distinctions.

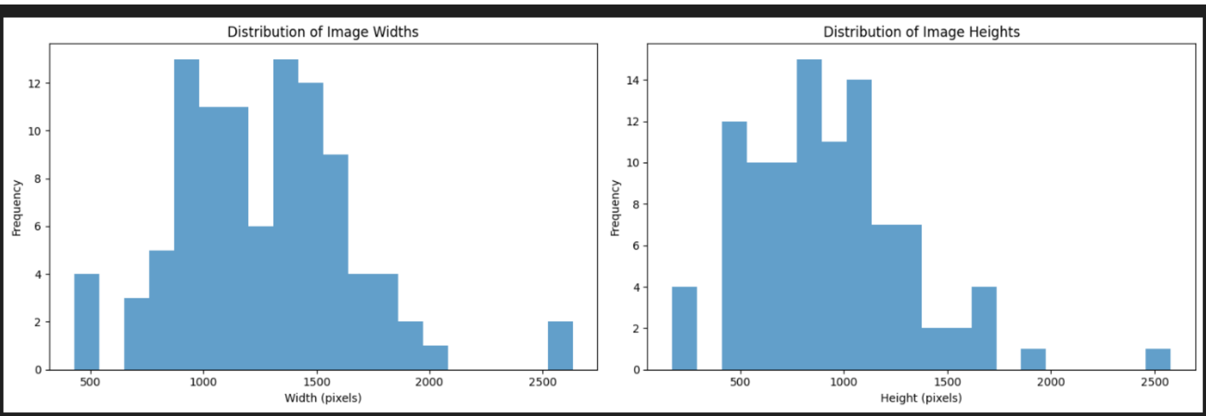
2.3.2 Image Dimensions Analysis



```python
# Combine all available image paths
all_image_paths = normal_train_paths + pneumonia_train_paths + normal_valid_paths + pneumonia_valid_paths + normal_test_paths + pneumonia_test_paths

# Analyze image dimensions
if all_image_paths:
    # Sample random images (up to 100)
    num_samples = min(100, len(all_image_paths))
    sample_paths = random.sample(all_image_paths, num_samples)

    widths = []
    heights = []

    for img_path in sample_paths:
        try:
            with Image.open(img_path) as img:
                widths.append(img.width)
                heights.append(img.height)
        except Exception as e:
            print(f"Error processing {img_path}: {e}")

# Print statistics
print(f"Width - Min: {min(widths)}, Max: {max(widths)}, Mean: {np.mean(widths):.2f}")
print(f"Height - Min: {min(heights)}, Max: {max(heights)}, Mean: {np.mean(heights):.2f}")

# Plot dimension distributions
plt.figure(figsize=(15, 5))

plt.subplot(1, 2, 1)
plt.hist(widths, bins=20, alpha=0.7)
plt.title('Distribution of Image Widths')
plt.xlabel('Width (pixels)')
plt.ylabel('Frequency')

plt.subplot(1, 2, 2)
plt.hist(heights, bins=20, alpha=0.7)
plt.title('Distribution of Image Heights')
plt.xlabel('Height (pixels)')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

✓ 0.5s

```
Width - Min: 428, Max: 2633, Mean: 1266.86
Height - Min: 172, Max: 2578, Mean: 922.21
```



The X-ray images have varying dimensions, which will require resizing to a standard size for our deep learning models.

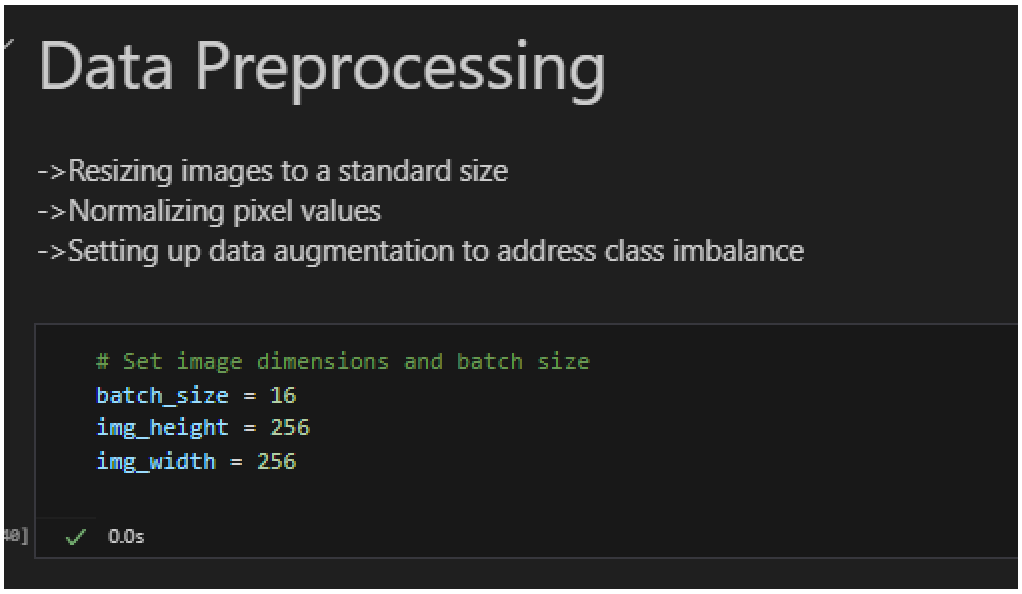
Next section is analyze the dimensions of images in the x-raydataset. First, combines all image file paths from the training, validation, and test sets for both normal and pneumonia categories into a single list called “all_image_paths”. If this list is not empty, the code randomly selects up to 100 images from the combined set to sample the dimensions.

For each sampled image, it attempts to open the file using the Python Imaging Library (PIL). If successful, it records the image's width and height. If an image cannot be opened, it prints an error message but continues processing the rest.

After collecting the dimensions, the code prints the minimum, maximum, and mean values for both widths and heights of the sampled images. Finally, we visualizes the distribution of image widths and heights using histograms, allow to see how much the image sizes vary across the dataset. The results shows the width minimum value is 428 pixels and maximum value is 2633 pixels, whereas the minimum value for height is 172 pixels and maximum value is 2578 pixels. The X-ray images have varying dimensions, which will require resizing to a standard size for our Convolution Neural Network models.

2.4 Data Preprocessing

2.4.1 Image Resizing



Data Preprocessing

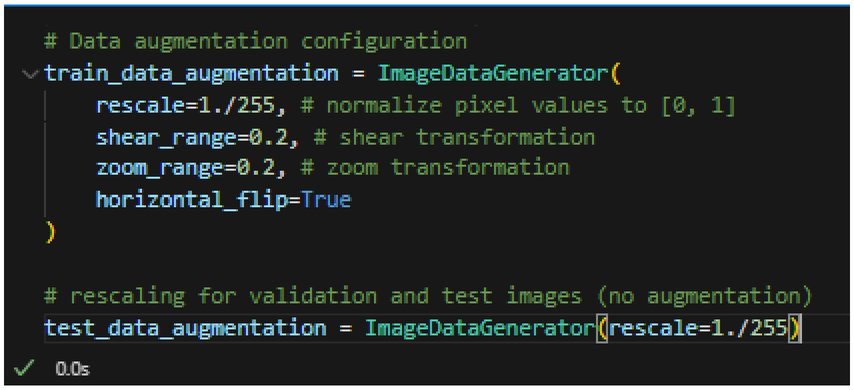
->Resizing images to a standard size
->Normalizing pixel values
->Setting up data augmentation to address class imbalance

```
# Set image dimensions and batch size
batch_size = 16
img_height = 256
img_width = 256
```

```
✓ 0.0s
```

Next is resizing the images. It is important to set the configuration parameters for image pre-processing and model training. We set the batch_size to 16, which makes the model able to handle 16 images per training step. This avoids a trade-off between memory use and speed of training. We define img_height and img_width as 256 both. This means, the input images will be resized to 256x256 pixels before we enter them as vectors to the neural network.

2.4.2 Data Augmentation and Normalization

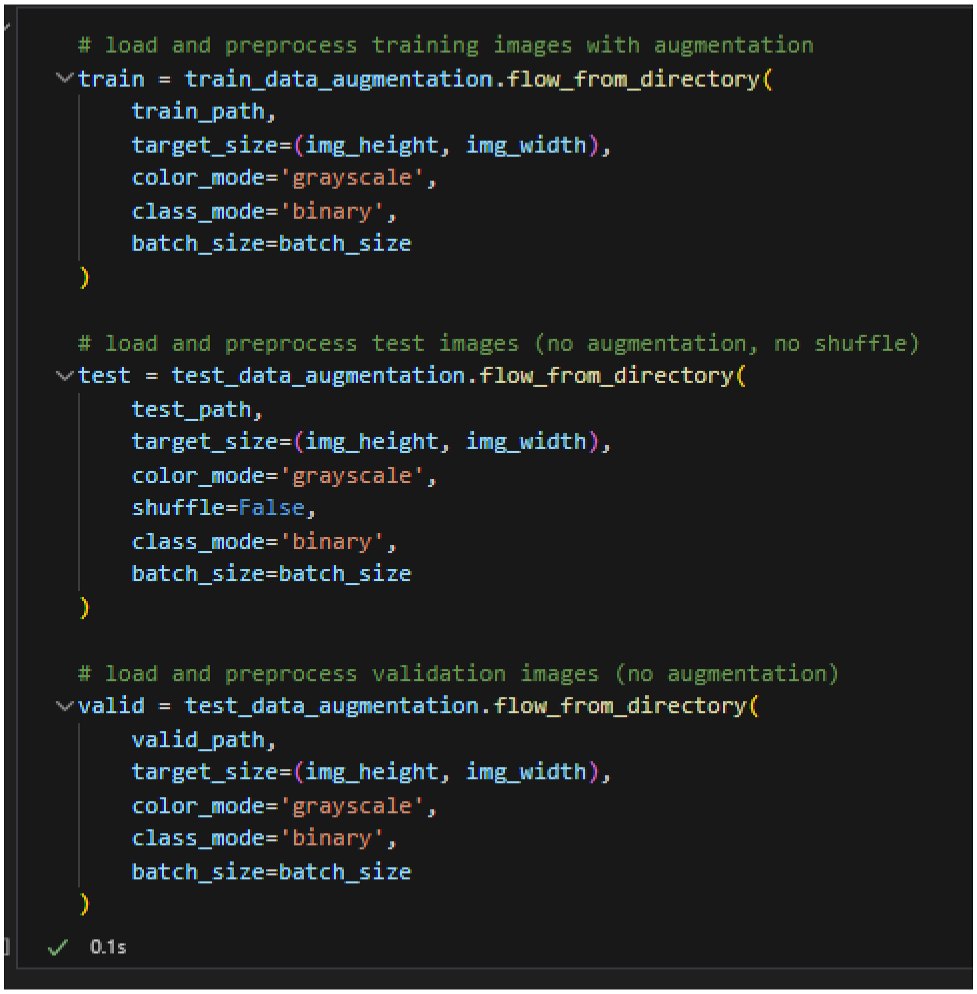


```
# Data augmentation configuration
train_data_augmentation = ImageDataGenerator(
    rescale=1./255, # normalize pixel values to [0, 1]
    shear_range=0.2, # shear transformation
    zoom_range=0.2, # zoom transformation
    horizontal_flip=True
)

# rescaling for validation and test images (no augmentation)
test_data_augmentation = ImageDataGenerator(rescale=1./255)
```

```
✓ 0.0s
```

Moving on is setting up two separate image data generators for preprocessing and augmenting images before they are fed into neural network models. The train_data_augmentation object is an instance of ImageDataGenerator. It is used to perform several transformations on training images. The function will rescale the image pixel values from the range [0, 255] to [0, 1] by "using rescale=1./255", applies random shearing (shear_range=0.2), random zooming (zoom_range=0.2), and random horizontal flipping (horizontal_flip=True). These augmentations are used to help increase the diversity of training data and improve the model robustness and prevent overfit.

The test_data_augmentation object is also an ImageDataGenerator, but this one only rescales the images and did not apply any random transformations. This approach will ensures that validation and test images are evaluated in their original form. This provides a fair assessment of the model's performance. This separation between training and validation or test preprocessing is the best practice in deep learning workflows.



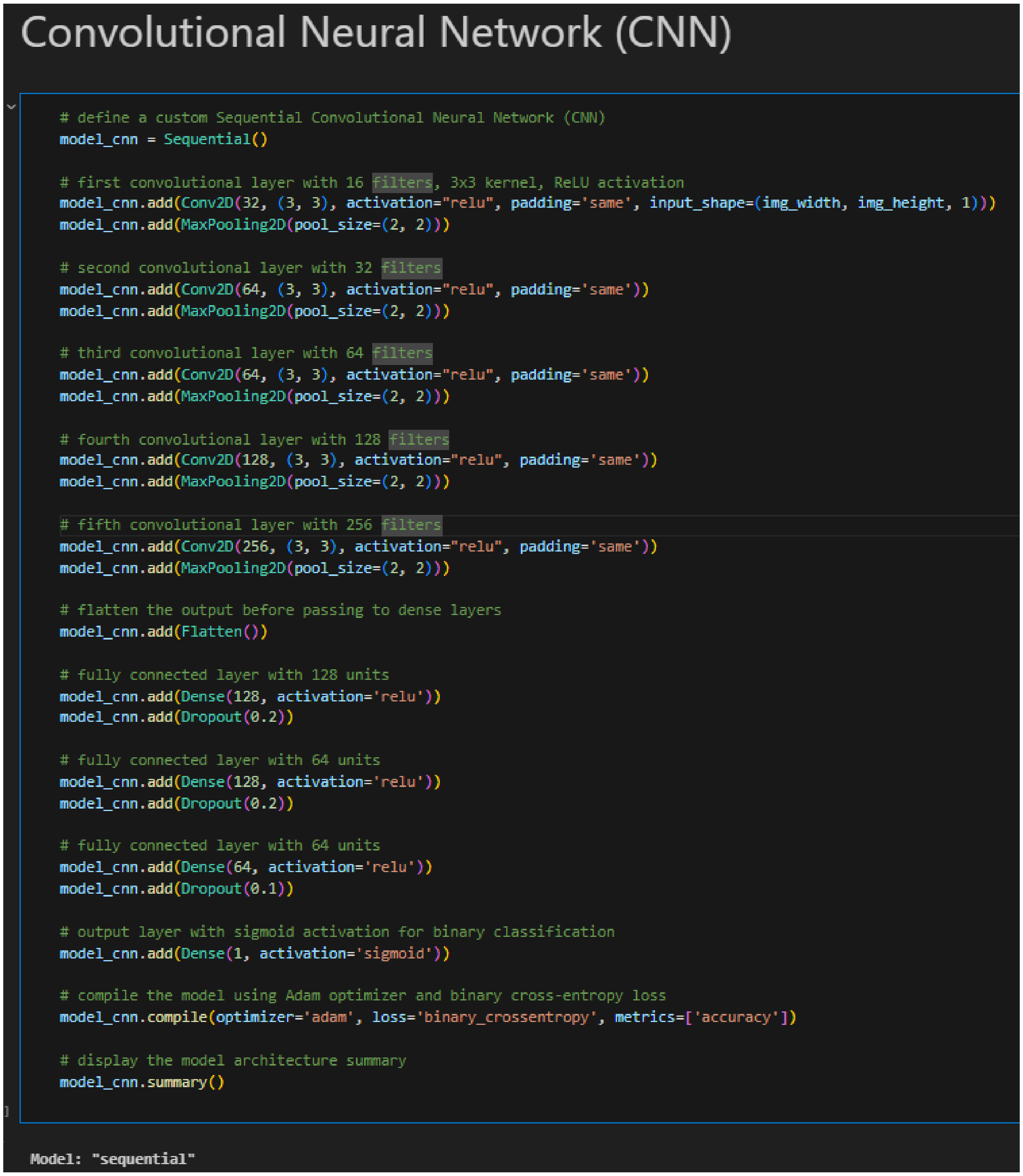
```
# load and preprocess training images with augmentation
train = train_data_augmentation.flow_from_directory(
    train_path,
    target_size=(img_height, img_width),
    color_mode='grayscale',
    class_mode='binary',
    batch_size=batch_size
)

# load and preprocess test images (no augmentation, no shuffle)
test = test_data_augmentation.flow_from_directory(
    test_path,
    target_size=(img_height, img_width),
    color_mode='grayscale',
    shuffle=False,
    class_mode='binary',
    batch_size=batch_size
)

# load and preprocess validation images (no augmentation)
valid = test_data_augmentation.flow_from_directory(
    valid_path,
    target_size=(img_height, img_width),
    color_mode='grayscale',
    class_mode='binary',
    batch_size=batch_size
)
```

Next is uses train_data_augmentation.flow_from_directory reads images from train_path, resizes them to the specified dimensions (256x256), converts them to grayscale, and applies data augmentation as previously configured. The images are grouped into batches of size (16), and the labels are provided in binary format (0 or 1), suitable for binary classification.

For the test and validation sets, test_data_augmentation.flow_from_directory is used. These generators also resize images and convert them to grayscale, but do not apply augmentation, only rescaling is performed (normalize the image into scale of 0,1). The test generator has shuffle=False to ensure the order of images is preserved, which is important for evaluation and matching predictions to ground truth labels. Both test and validation generators also use binary class mode and the same batch size.

2.5 Model Development

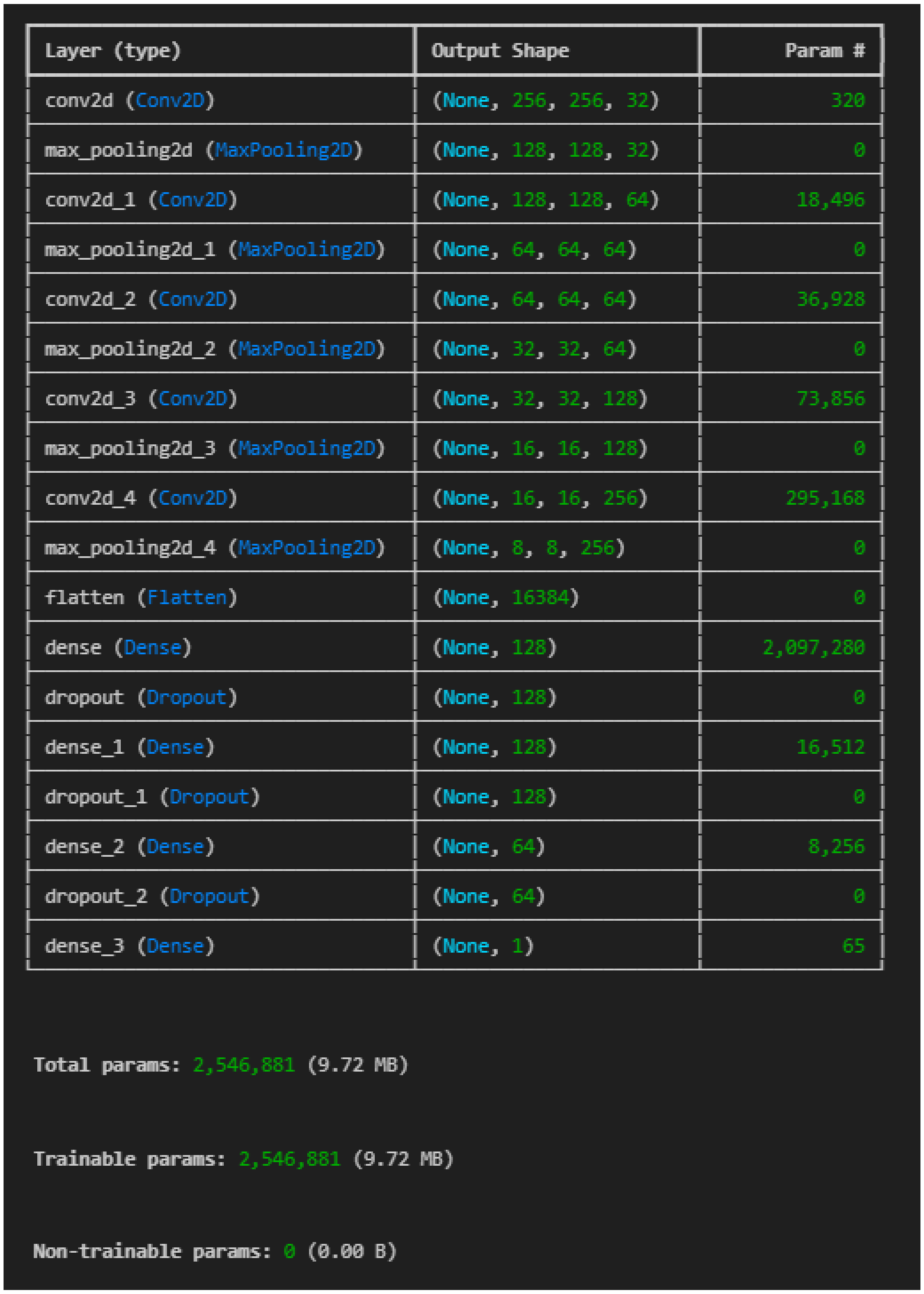
2.5.1 Model Architecture



Convolutional Neural Network (CNN)

```
# define a custom Sequential Convolutional Neural Network (CNN)
model_cnn = Sequential()

# first convolutional layer with 16 filters, 3x3 kernel, ReLU activation
model_cnn.add(Conv2D(32, (3, 3), activation="relu", padding='same', input_shape=(img_width, img_height, 1)))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

# second convolutional layer with 32 filters
model_cnn.add(Conv2D(64, (3, 3), activation="relu", padding='same'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

# third convolutional layer with 64 filters
model_cnn.add(Conv2D(64, (3, 3), activation="relu", padding='same'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

# fourth convolutional layer with 128 filters
model_cnn.add(Conv2D(128, (3, 3), activation="relu", padding='same'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

# fifth convolutional layer with 256 filters
model_cnn.add(Conv2D(256, (3, 3), activation="relu", padding='same'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

# flatten the output before passing to dense layers
model_cnn.add(Flatten())

# fully connected layer with 128 units
model_cnn.add(Dense(128, activation='relu'))
model_cnn.add(Dropout(0.2))

# fully connected layer with 64 units
model_cnn.add(Dense(128, activation='relu'))
model_cnn.add(Dropout(0.2))

# fully connected layer with 64 units
model_cnn.add(Dense(64, activation='relu'))
model_cnn.add(Dropout(0.1))

# output layer with sigmoid activation for binary classification
model_cnn.add(Dense(1, activation='sigmoid'))

# compile the model using Adam optimizer and binary cross-entropy loss
model_cnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# display the model architecture summary
model_cnn.summary()
```

```
Model: "sequential"
```

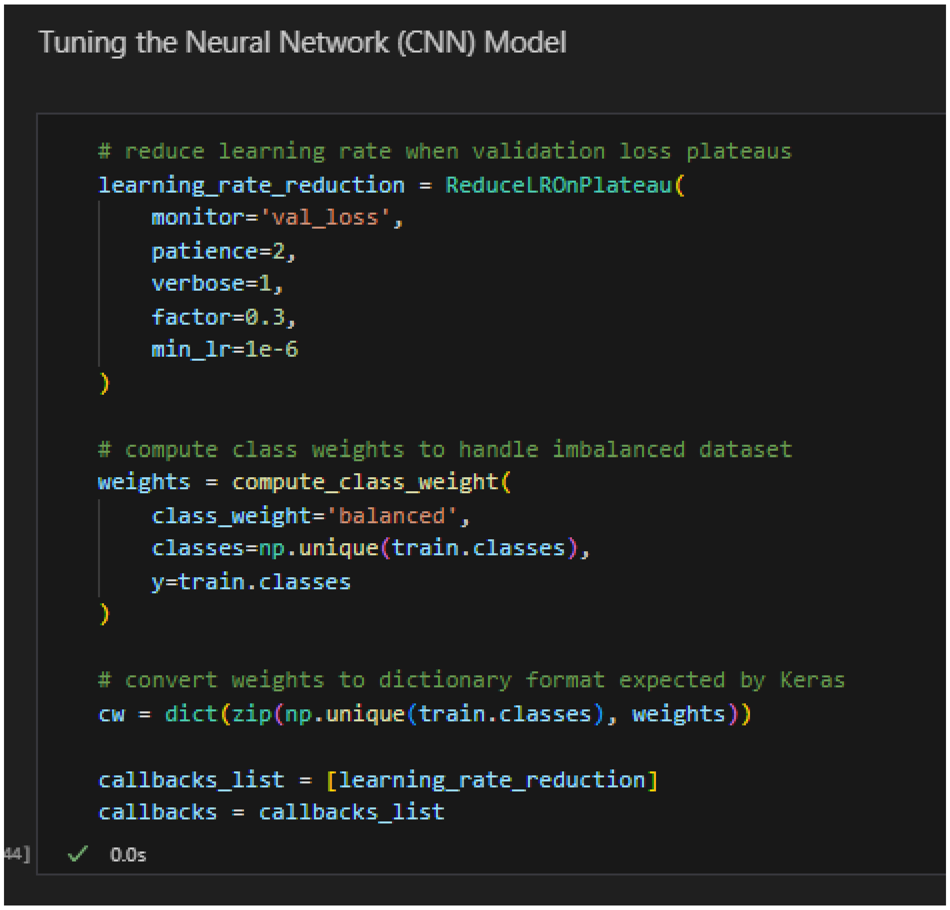


```
Layer (type)                      Output Shape              Param #
conv2d (Conv2D)                   (None, 256, 256, 32)          320
max_pooling2d (MaxPooling2D)      (None, 128, 128, 32)            0
conv2d_1 (Conv2D)                 (None, 128, 128, 64)       18,496
max_pooling2d_1 (MaxPooling2D)    (None, 64, 64, 64)              0
conv2d_2 (Conv2D)                 (None, 64, 64, 64)         36,928
max_pooling2d_2 (MaxPooling2D)    (None, 32, 32, 64)              0
conv2d_3 (Conv2D)                 (None, 32, 32, 128)        73,856
max_pooling2d_3 (MaxPooling2D)    (None, 16, 16, 128)             0
conv2d_4 (Conv2D)                 (None, 16, 16, 256)       295,168
max_pooling2d_4 (MaxPooling2D)    (None, 8, 8, 256)               0
flatten (Flatten)                 (None, 16384)                   0
dense (Dense)                     (None, 128)             2,097,280
dropout (Dropout)                 (None, 128)                     0
dense_1 (Dense)                   (None, 128)                16,512
dropout_1 (Dropout)               (None, 128)                     0
dense_2 (Dense)                   (None, 64)                  8,256
dropout_2 (Dropout)               (None, 64)                      0
dense_3 (Dense)                   (None, 1)                      65

Total params: 2,546,881 (9.72 MB)

Trainable params: 2,546,881 (9.72 MB)

Non-trainable params: 0 (0.00 B)
```

The model architecture of the Convolutional Neural Network (CNN) using Keras' Sequential API for binary image classification can be described as follows. The model is built layer by layer, starting with five convolutional layers followed by MaxPooling layers. Each Conv2D layer extracts features from the input images using different numbers of filters (starting at 32 and increasing up to 256), a 3x3 kernel size, and the ReLU activation function. The padding='same' argument ensures the output size matches the input size for each convolution. After each convolutional layer it is followed by a MaxPooling2D layer. This is to reduce the spatial dimensions and help to down sample the feature maps and control overfitting.

After proceeding five convolutional layers, a flatten layer is used to receive and flatten the output of convolutional layer by converting the 2D feature maps into a 1D vector. Then the output of the flatten layer is passed through three fully connected (Dense) layers. These three dense layers use the ReLU activation function and are include with Dropout layer with a dropout rate of 0.1-0.2. The uses of dropout is to randomly deactivate a fraction of neurons during training and to help prevent overfitting for the model. The final output layer uses a single neuron with a sigmoid activation, making it suitable for binary classification tasks such as distinguishing between normal and pneumonia X-rays. Finally, the model is compiled with the Adam optimizer, binary cross-entropy loss appropriate for binary classification, and accuracy as the evaluation metric.

2.5.2 Model Tuning


Tuning the Neural Network (CNN) Model

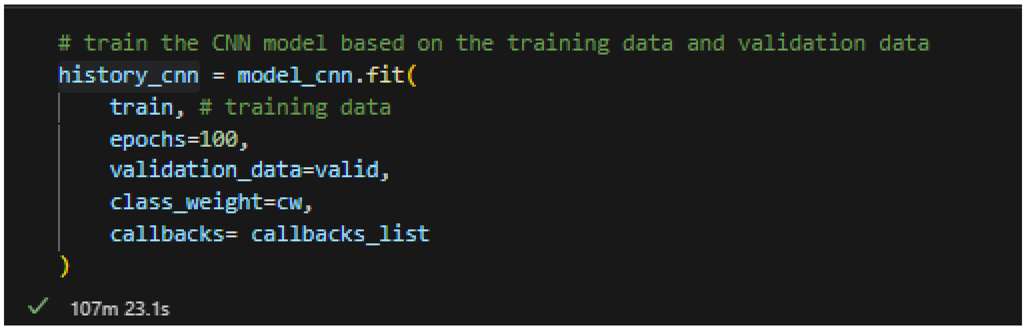
```
# reduce learning rate when validation loss plateaus
learning_rate_reduction = ReduceLROnPlateau(
    monitor='val_loss',
    patience=2,
    verbose=1,
    factor=0.3,
    min_lr=1e-6
)

# compute class weights to handle imbalanced dataset
weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(train.classes),
    y=train.classes
)

# convert weights to dictionary format expected by Keras
cw = dict(zip(np.unique(train.classes), weights))

callbacks_list = [learning_rate_reduction]
callbacks = callbacks_list
```

Next is model tuning for improving neural network training by using adaptive learning rate scheduling and class balancing. First, we creates a ReduceLROnPlateau callback named learning_rate_reduction. This callback monitors the validation loss (val_loss) during training. If the validation loss does not improve for 2 consecutive epochs (patience=2), the learning rate is reduced by multiplying it by 0.3 (factor=0.3). The learning rate will not go below 1e-6 (min_lr=1e-6). Setting verbose=1 ensures that a message is printed whenever the learning rate is reduced. This adaptive adjustment helps the model converge more effectively, especially when progress stalls.

Next, we computes the class weights using compute_class_weight with the 'balanced' option. This function calculates weights inversely proportional to class frequencies, helping to address class imbalance in the training data. The resulting weights are paired with their corresponding class labels using zip and converted into a dictionary (cw), which is the format expected by Keras during model training.

Finally, the callbacks_list is created to hold the learning rate reduction callback, and callbacks is set to this list. This list can be passed to the fit method of the model to activate the learning rate scheduling during training.

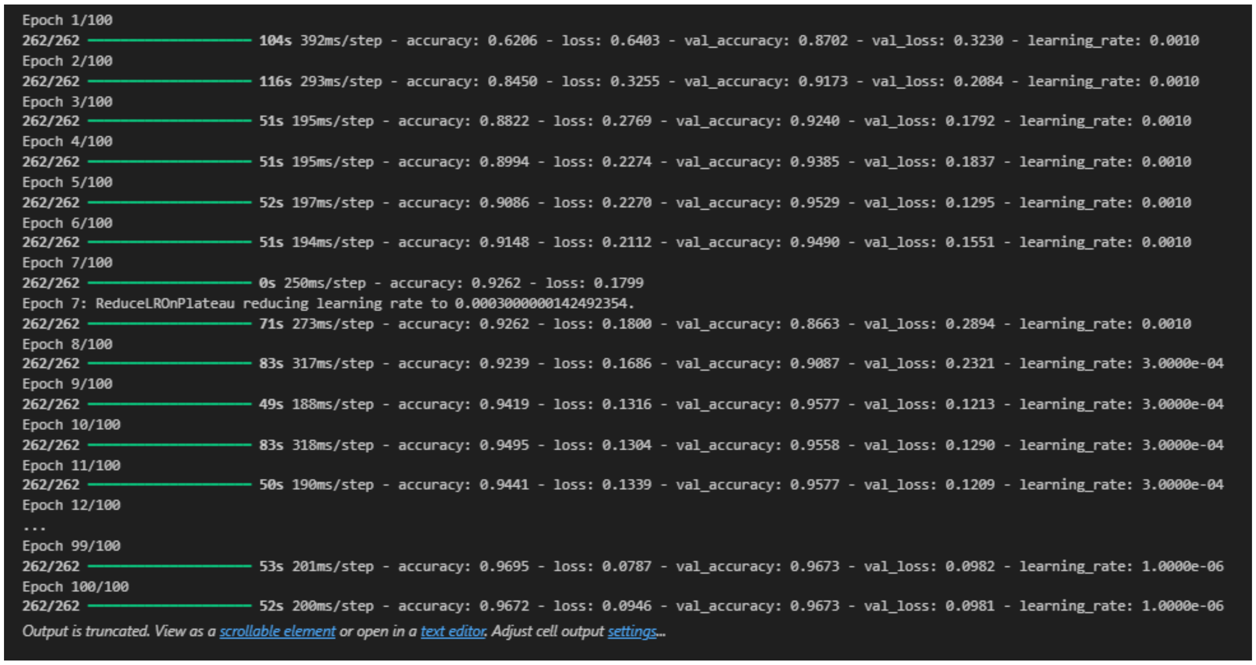
2.5.3 Model Training



```
# train the CNN model based on the training data and validation data
history_cnn = model_cnn.fit(
    train, # training data
    epochs=100,
    validation_data=valid,
    class_weight=cw,
    callbacks= callbacks_list
)
```

```
✓ 107m 23.1s
```

Next is trains the CNN model using the Keras fit method. The train generator provides batches of preprocessed and augmented training images. The model will train for up to 100 epochs, where each epoch is a full pass through the training data. The validation_data=valid argument specifies a separate validation set, allowing the model to be evaluated on unseen data at the end of each epoch. The class_weight=cw parameter is used to address class imbalance by assigning higher importance to underrepresented classes during training. The callbacks=callbacks_list argument enables additional training features, such as reducing the learning rate if the validation loss stops improving. The result of fit is stored in history_cnn, which contains details about the training and validation loss and accuracy over all epochs. This information can be used later for analysis or visualization of the training process.



```
Epoch 1/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 104s 392ms/step - accuracy: 0.6206 - loss: 0.6403 - val_accuracy: 0.8702 - val_loss: 0.3230 - learning_rate: 0.0010
Epoch 2/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 116s 293ms/step - accuracy: 0.8450 - loss: 0.3255 - val_accuracy: 0.9173 - val_loss: 0.2084 - learning_rate: 0.0010
Epoch 3/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 51s 195ms/step - accuracy: 0.8822 - loss: 0.2769 - val_accuracy: 0.9240 - val_loss: 0.1792 - learning_rate: 0.0010
Epoch 4/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 51s 195ms/step - accuracy: 0.8994 - loss: 0.2274 - val_accuracy: 0.9385 - val_loss: 0.1837 - learning_rate: 0.0010
Epoch 5/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 52s 197ms/step - accuracy: 0.9086 - loss: 0.2270 - val_accuracy: 0.9529 - val_loss: 0.1295 - learning_rate: 0.0010
Epoch 6/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 51s 194ms/step - accuracy: 0.9148 - loss: 0.2112 - val_accuracy: 0.9490 - val_loss: 0.1551 - learning_rate: 0.0010
Epoch 7/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 0s 250ms/step - accuracy: 0.9262 - loss: 0.1799
Epoch 7: ReduceLROnPlateau reducing learning rate to 0.0003000000142492354.
262/262 ━━━━━━━━━━━━━━━━━━━━ 71s 273ms/step - accuracy: 0.9262 - loss: 0.1800 - val_accuracy: 0.8663 - val_loss: 0.2894 - learning_rate: 0.0010
Epoch 8/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 83s 317ms/step - accuracy: 0.9239 - loss: 0.1686 - val_accuracy: 0.9087 - val_loss: 0.2321 - learning_rate: 3.0000e-04
Epoch 9/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 49s 188ms/step - accuracy: 0.9419 - loss: 0.1316 - val_accuracy: 0.9577 - val_loss: 0.1213 - learning_rate: 3.0000e-04
Epoch 10/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 83s 318ms/step - accuracy: 0.9495 - loss: 0.1304 - val_accuracy: 0.9558 - val_loss: 0.1290 - learning_rate: 3.0000e-04
Epoch 11/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 50s 190ms/step - accuracy: 0.9441 - loss: 0.1339 - val_accuracy: 0.9577 - val_loss: 0.1209 - learning_rate: 3.0000e-04
Epoch 12/100
...
Epoch 99/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 53s 201ms/step - accuracy: 0.9695 - loss: 0.0787 - val_accuracy: 0.9673 - val_loss: 0.0982 - learning_rate: 1.0000e-06
Epoch 100/100
262/262 ━━━━━━━━━━━━━━━━━━━━ 52s 200ms/step - accuracy: 0.9672 - loss: 0.0946 - val_accuracy: 0.9673 - val_loss: 0.0981 - learning_rate: 1.0000e-06
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

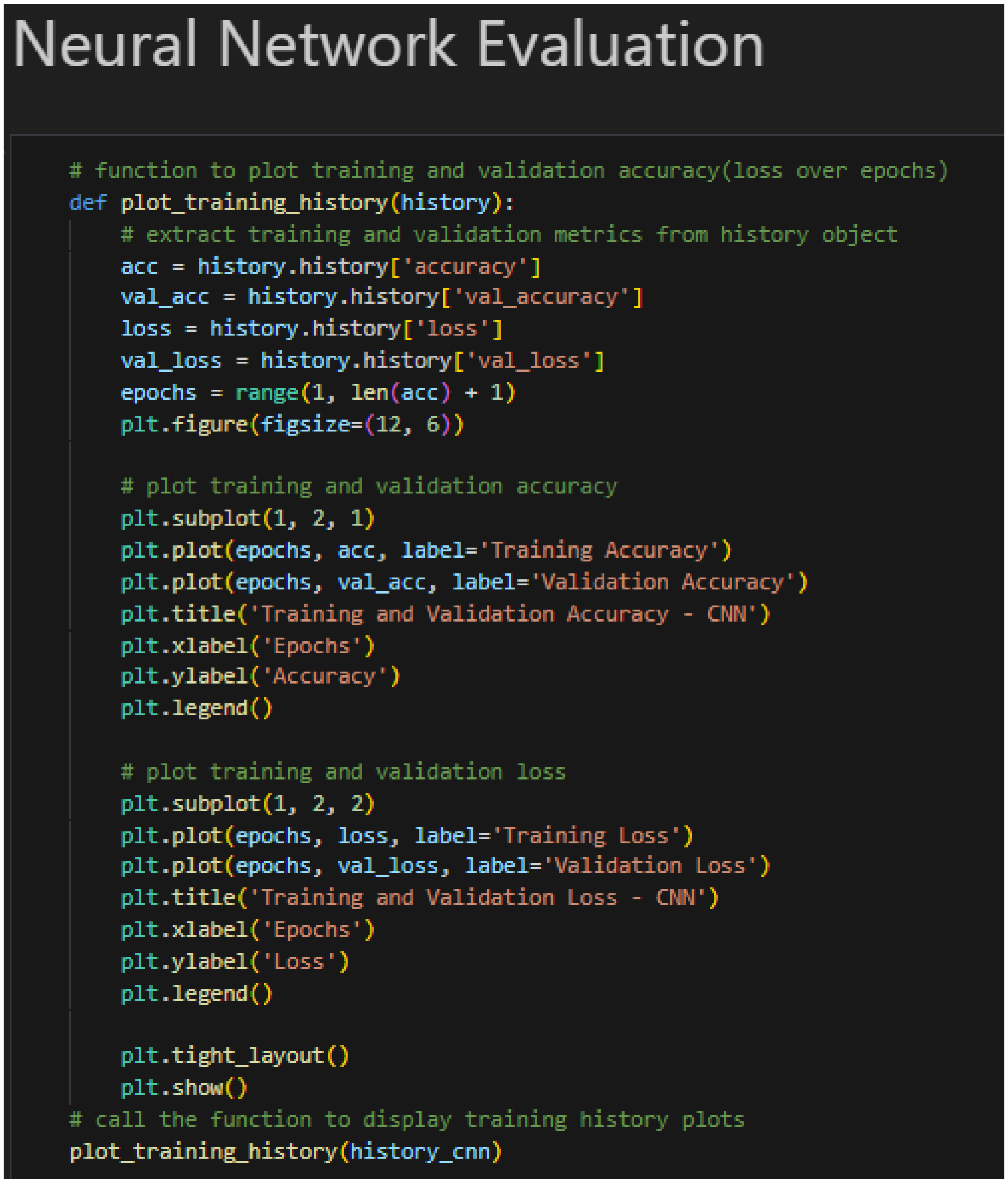
The output shows the training progress of CNN model over 100 epochs. Each line provides a result for a single epoch including the training accuracy, loss, validation accuracy, validation loss, and the current learning rate.

At start, the CNN model accuracy is relatively low with 62.06% on training and 87.02% on validation. But it improves to 96.72% on training and 96.73% by the end of 100epoch. This indicates the model is learning the training data well. For the training loss, the model starts at 0.6403 at epoch 1 and decreases to 0.0981 at the epoch 100. This means the decreasing trend of training loss is good. The validation accuracy (val_accuracy) climbs above 95% by epoch 5, indicate the CNN model is learning to distinguish between classes effectively. The learning rate starts at 0.001 and is automatically reduced several times by the ReduceLROnPlateau callback. By the end, the learning rate was fix at 1e-6.

Overall, the CNN model is learning well, because both training and validation accuracy increase and loss decreases. The validation accuracy 96.73% is very close to the training accuracy 96.72% at epoch 100. This suggests that the model is generalizing well and not overfitting.

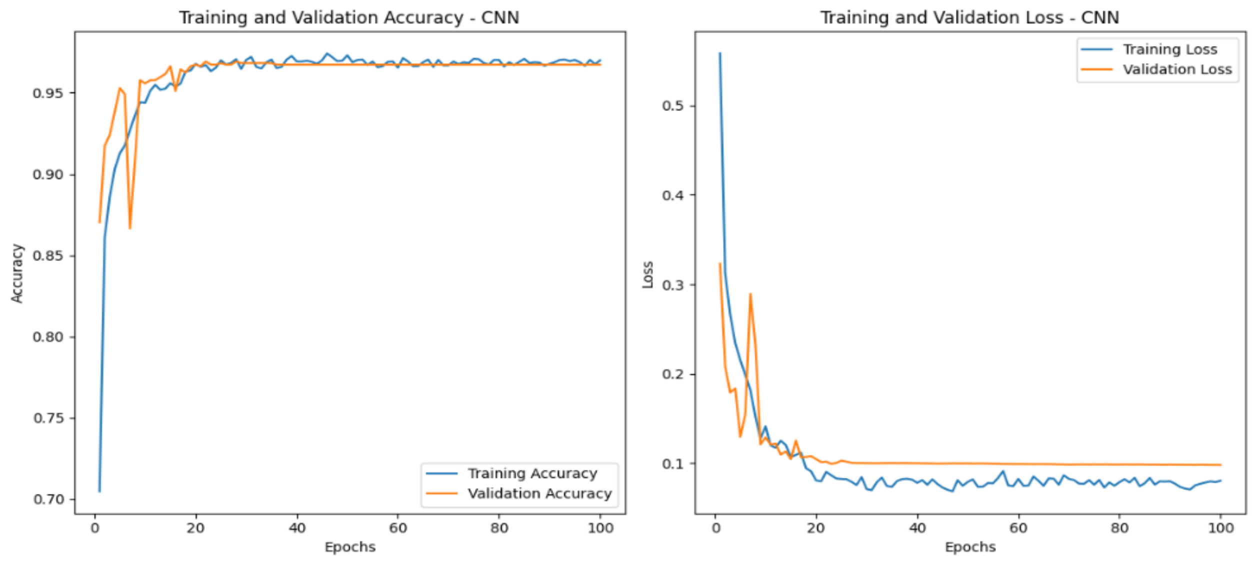
3.0 Interpretation and Evaluation

3.1 Model Evaluation



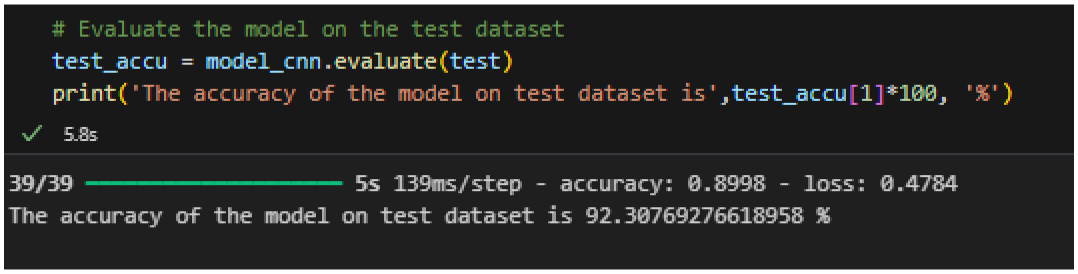
Neural Network Evaluation

```python
# function to plot training and validation accuracy(loss over epochs)
def plot_training_history(history):
    # extract training and validation metrics from history object
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs = range(1, len(acc) + 1)
    plt.figure(figsize=(12, 6))

    # plot training and validation accuracy
    plt.subplot(1, 2, 1)
    plt.plot(epochs, acc, label='Training Accuracy')
    plt.plot(epochs, val_acc, label='Validation Accuracy')
    plt.title('Training and Validation Accuracy - CNN')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # plot training and validation loss
    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, label='Training Loss')
    plt.plot(epochs, val_loss, label='Validation Loss')
    plt.title('Training and Validation Loss - CNN')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()
# call the function to display training history plots
plot_training_history(history_cnn)
```



The accuracy plots indicate that there is a large difference in the training accuracy (the blue line) which goes from 0.62 to 0.97 and the validation accuracy (the orange line) which goes from 0.87 to 0.97, but with quite a few fluctuations. Training and validation accuracy both stabilize around 96–97% after 30 epochs, and the validation loss plateaus around 0.1. This plateau reveals that the model has been trained to the optimal performance in the validation set and further training does not yield significant improvements**.** Overall, the model maintains high accuracy and low loss, indicating good generalization and convergence. The mild overfitting is controlled by regularization dropout and learning rate reduction.

3.2 Evaluation on Test Set

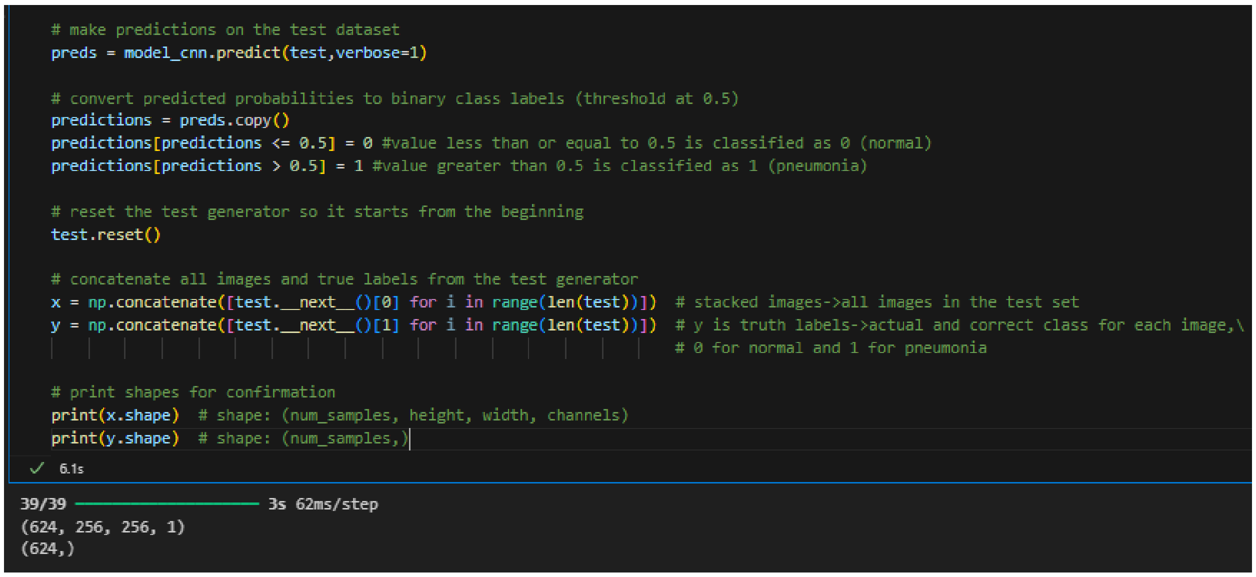


```
# Evaluate the model on the test dataset
test_accu = model_cnn.evaluate(test)
print('The accuracy of the model on test dataset is',test_accu[1]*100, '%')
```

✓ 5.8s

```
39/39 ━━━━━━━━━━━━━━━━━━━━ 5s 139ms/step - accuracy: 0.8998 - loss: 0.4784
The accuracy of the model on test dataset is 92.30769276618958 %
```

The output shows the evaluation results of the trained CNN model on the test dataset. The accuracy of the model on test dataset is 92.31. There is a small gap between validation (96.73%) and test (92.31%) which indicates the model generalize well. This suggests that the model performs well on unseen data, correctly predicting the class for most test images.

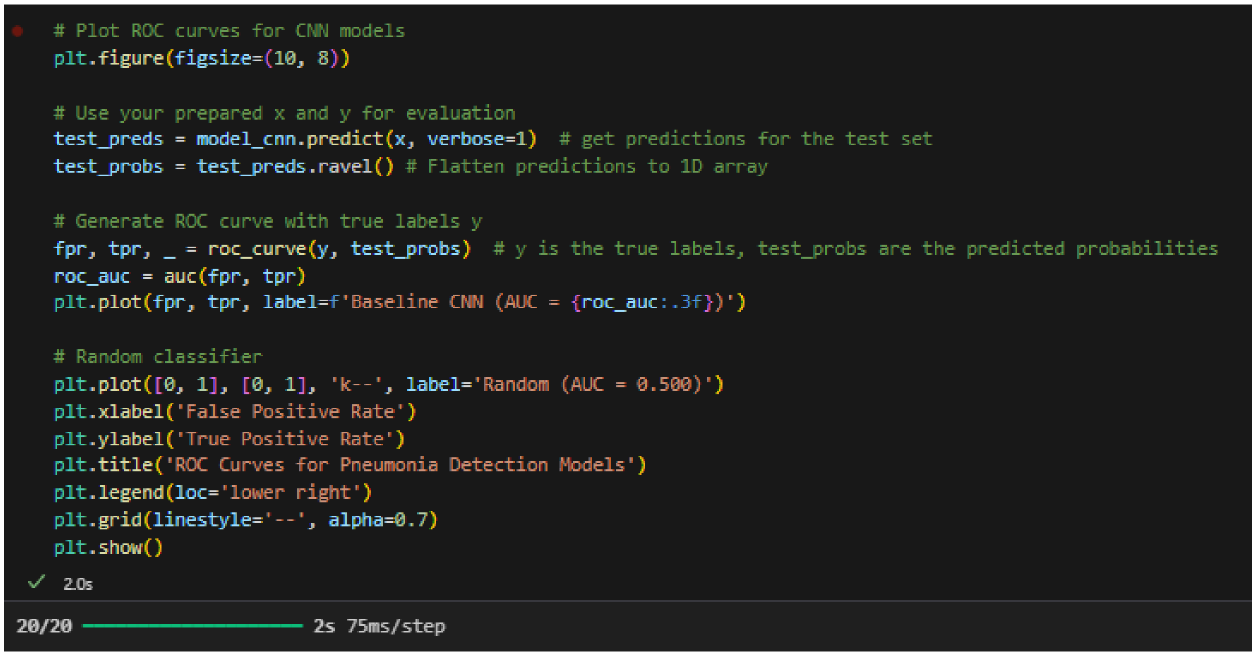


```
# make predictions on the test dataset
preds = model_cnn.predict(test,verbose=1)

# convert predicted probabilities to binary class labels (threshold at 0.5)
predictions = preds.copy()
predictions[predictions <= 0.5] = 0 #value less than or equal to 0.5 is classified as 0 (normal)
predictions[predictions > 0.5] = 1 #value greater than 0.5 is classified as 1 (pneumonia)

# reset the test generator so it starts from the beginning
test.reset()

# concatenate all images and true labels from the test generator
x = np.concatenate([test.__next__()[0] for i in range(len(test))])  # stacked images->all images in the test set
y = np.concatenate([test.__next__()[1] for i in range(len(test))])  # y is truth labels->actual and correct class for each image,\
                                                                     # 0 for normal and 1 for pneumonia

# print shapes for confirmation
print(x.shape)  # shape: (num_samples, height, width, channels)
print(y.shape)  # shape: (num_samples,)
```

```
✓ 6.1s
39/39 ━━━━━━━━━━━━━━━━━━━━ 3s 62ms/step
(624, 256, 256, 1)
(624,)
```

Next is to handle the model prediction and prepare the test data for further evaluation. First, we use model_cnn.predict(test, verbose=1) to generate predicted probabilities for each test image, where test is a data generator. The predictions are probabilities between 0 and 1, so the next lines convert these to binary class labels. The predicted values ≤ 0.5 will be set to 0, and the predicted values > 0.5 will be set to 1, so the prediction results can be used for binary classification. After making predictions, test.reset() is called so that it starts yielding batches from the beginning of the dataset again. Next, the code collects all test images and their true labels by iterating through the generator and concatenating the batches using np.concatenate(). This results in two arrays where x contains all test images, and y contains the corresponding truth labels. Finally, the shapes of x and y are printed for confirmation. The x.shape should show the total number of test samples and their dimensions, while y.shape should show the total number of labels. This setup is useful for comparing predictions to true labels in later evaluation steps, such as ROC curve, confusion matrix or classification report calculations.

3.2.1 ROC Curve

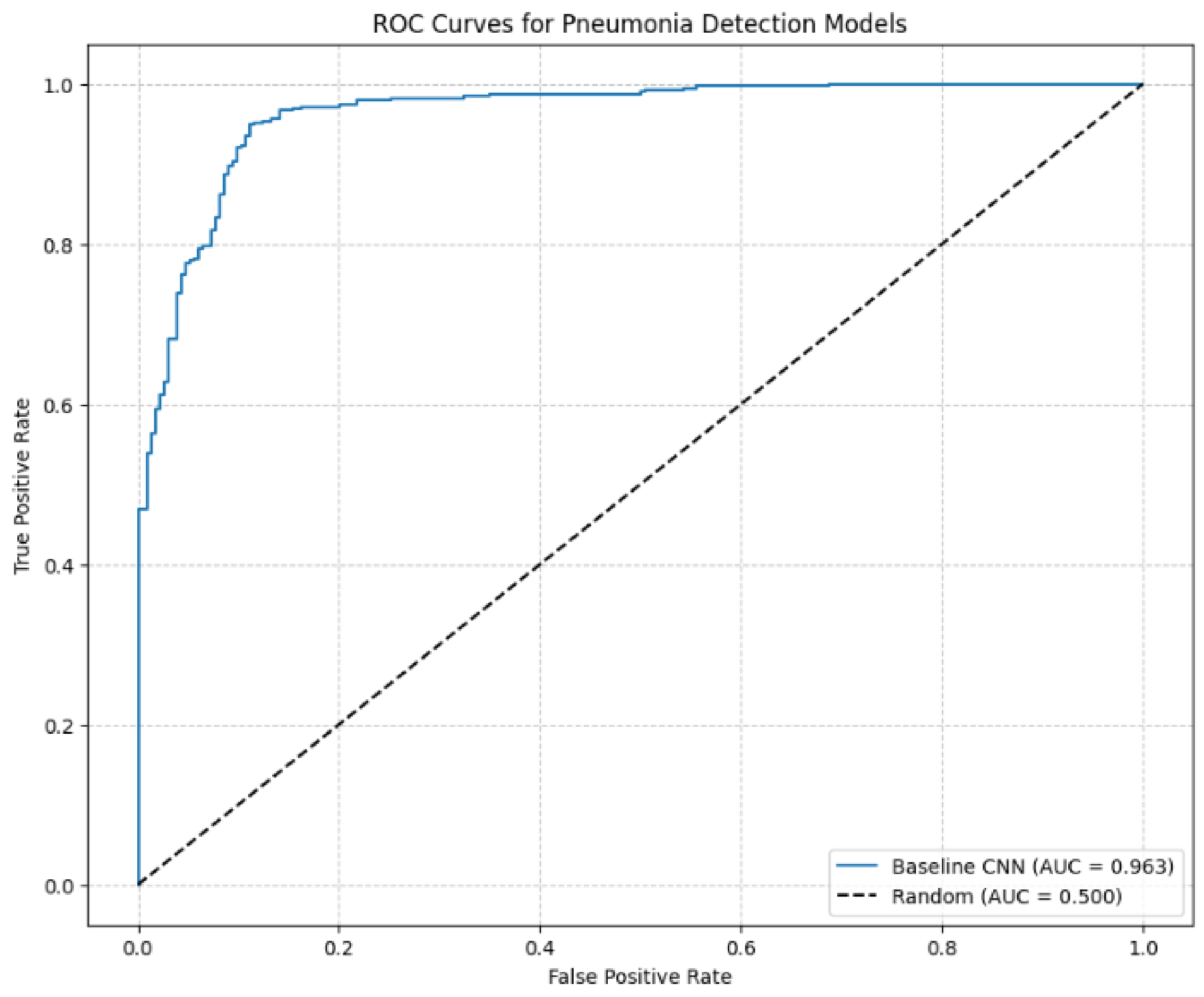


```
# Plot ROC curves for CNN models
plt.figure(figsize=(10, 8))

# Use your prepared x and y for evaluation
test_preds = model_cnn.predict(x, verbose=1)  # get predictions for the test set
test_probs = test_preds.ravel() # Flatten predictions to 1D array

# Generate ROC curve with true labels y
fpr, tpr, _ = roc_curve(y, test_probs)  # y is the true labels, test_probs are the predicted probabilities
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label=f'Baseline CNN (AUC = {roc_auc:.3f})')

# Random classifier
plt.plot([0, 1], [0, 1], 'k--', label='Random (AUC = 0.500)')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Pneumonia Detection Models')
plt.legend(loc='lower right')
plt.grid(linestyle='--', alpha=0.7)
plt.show()
```

✓ 2.0s

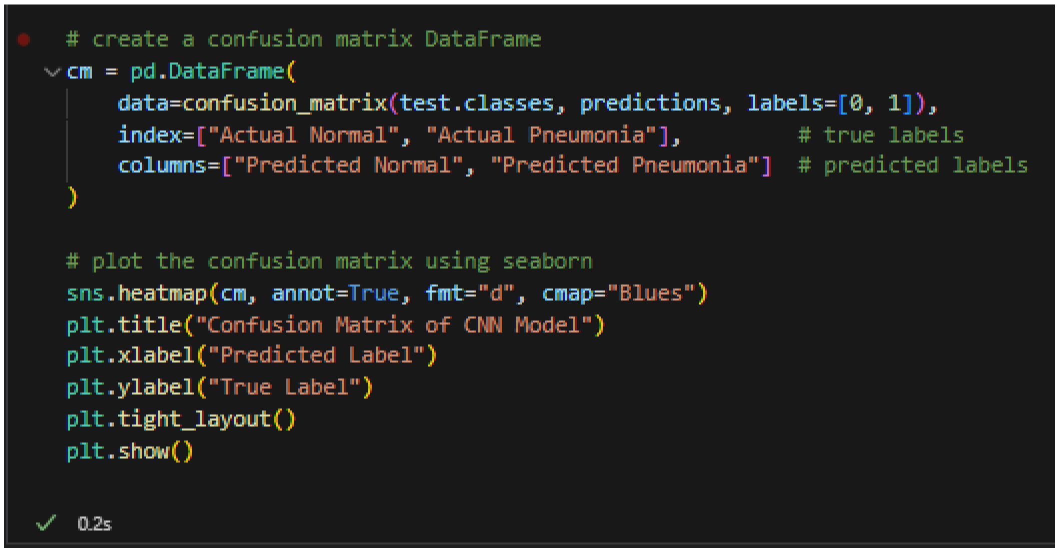
```
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 75ms/step
```

Next is plotting a Receiver Operating Characteristic (ROC) curve to evaluate the performance of CNN model on the test dataset. The model predicts probabilities for each test image using model_cnn.predict(x, verbose=1), and these predictions are flattened into a one-dimensional array with ravel(). The roc_curve function is then used to calculate the false positive rates (fpr) and true positive rates (tpr) at various threshold levels, using the true labels (y) and the predicted probabilities (test_probs). True labels (y) tell the function which samples are actually positive (pneumonia) and which are negative (normal). Predicted probabilities (test_probs) are used to determine how the model would classify each sample at different thresholds. For each threshold, roc_curve compares the predicted class (based on the threshold) to the true class (y). It then counts the false positives where normal cases (y=0) that the model incorrectly predicts as pneumonia (predicted probability above the threshold) and true positives where Pneumonia cases (y=1) that the model correctly predicts as pneumonia. The area under the ROC curve (AUC) is computed with the auc function, providing a single metric that summarizes the model's ability to distinguish between classes. The ROC curve for the model is plotted, with its AUC value shown in the legend. For comparison, a diagonal line representing a random classifier (AUC = 0.5) is also plotted.



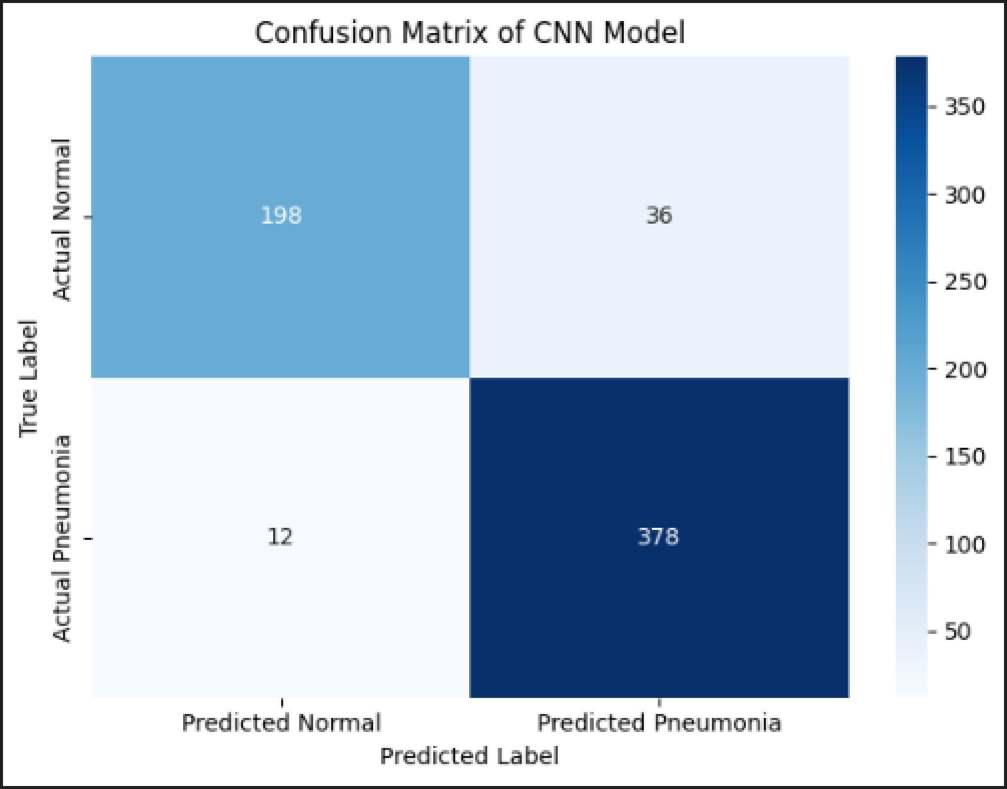
The CNN model shows an excellent diagnostic capability with AUC=0.963 for pneumonia detection. The ROC curve confirms it maintains high sensitivity across all operating points, making it clinically valuable. For example, for FPR value is 0.05, the TPR value is approximately equal to 0.75 (low false positive case, means low percentage of confirm the lungs is normal but diagnose as pneumonia). This means that the model can detect 75 percent of pneumonia cases with almost zero false alarms. This indicates that it is ideal for initial screening for patients where false alarms are not acceptable. But after the FPR values goes above 0.20, the TPR values are almost 0.97 or higher. This means that the model can detect 97% of pneumonia cases but having 20% of false positive. This scenario is suitable for critical situations where missing pneumonia cases are dangerous.

3.2.2 Confusion Matrix



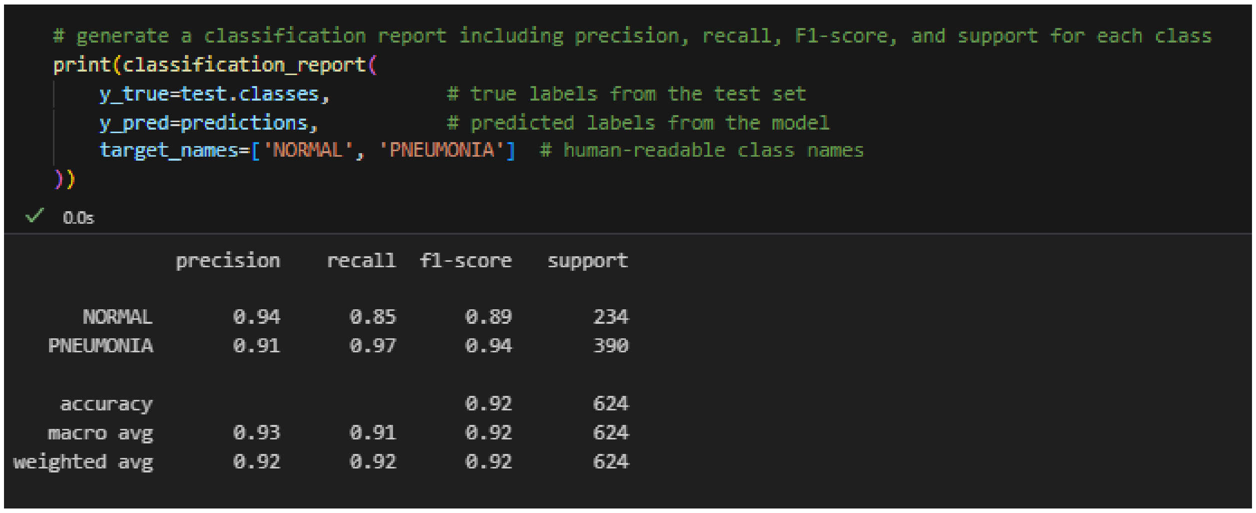
```
# create a confusion matrix DataFrame
cm = pd.DataFrame(
    data=confusion_matrix(test.classes, predictions, labels=[0, 1]),
    index=["Actual Normal", "Actual Pneumonia"],          # true labels
    columns=["Predicted Normal", "Predicted Pneumonia"]  # predicted labels
)

# plot the confusion matrix using seaborn
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix of CNN Model")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.tight_layout()
plt.show()
```

✓ 0.2s



The confusion matrix shows, there are 378 cases correctly identified pneumonia cases (true positive), 198 cases correctly identified as normal case (true negative), 36 case incorrectly classified normal as pneumonia (false positive, type 1 error) and 12 case incorrectly classified pneumonia as normal (false negative, type 2 error). From the confusion matrix, out of a total of 390 pneumonia cases, our CNN model misses 12 cases. These type 2 errors are critical in medical diagnosis because they mean a sick patient is missed by the model, potentially leading to lack of treatment. Other than that, there is 36 false positives cases which mean some healthy patients might undergo unnecessary tests.

3.3.3 Classification Report



```
# generate a classification report including precision, recall, F1-score, and support for each class
print(classification_report(
    y_true=test.classes,           # true labels from the test set
    y_pred=predictions,            # predicted labels from the model
    target_names=['NORMAL', 'PNEUMONIA']  # human-readable class names
))
```

```
              precision    recall  f1-score   support

      NORMAL       0.94      0.85      0.89       234
   PNEUMONIA       0.91      0.97      0.94       390

    accuracy                           0.92       624
   macro avg       0.93      0.91      0.92       624
weighted avg       0.92      0.92      0.92       624
```

The classification report shows that, for “Normal” x-ray image (234 data in test set) the CNN model achieves 0.94 for the precision indicate that when model predicts the “Normal” image is correct 94% for all time. For the recall, it achieves 0.85, meaning that the model is correctly identify 85% of all actual normal cases but misses 15%. So, the model sometimes incorrectly labels normal cases as pneumonia. For the F1-score it shows 0.89, which reflects a balanced measure of precision and recall for normal cases.

For “Pneumonia” x-ray image (390 data in test set), the CNN model achieves 0.91 for precision, which means it is correct 91% of the time when predict the pneumonia images. For the recall, the model achieves 0.97 indicates the model is especially strong at detecting pneumonia cases with only 3% miss cases. The F1 score of 0.94 shows the harmonic mean of precision and recall for pneumonia cases.

In summary, the overall accuracy of the model achieves 0.92, with macro averages of 0.93 precision, 0.91 recall, and 0.92 F1 score, demonstrating the model is well balanced, but slightly favors catching pneumonia over avoiding false alarms for normal cases, which is often desirable in medical screening.

4.0 Conclusion

In this study, we have successfully developed a CNN deep learning model for the automatic detection of pneumonia disease using chest X-ray images. The model had an excellent ability to make a diagnosis of normal and pneumonia, with a test accuracy of 92.31% and AUC score was 0.963. This confirmed that our CNN model had a great discriminative ability in differentiating normal and pneumonia cases. Most impressively, the model's recall for pneumonia detection was 0.97, meaning it detected 97% of actual pneumonia cases. This is a crucial characteristic since in medical diagnosis missing true positives cases will have serious consequences.

The confusion matrix showed only 12 false negatives in 624 test cases, indicating high reliability for screening purposes. The small difference of validation accuracy (96.73%) and test accuracy (92.31%) show a little sign of overfitting that requires to be addressed in the future work through transfer learning, larger datasets and other methods. These findings indicate that deep learning methods can be used to assist radiologists in accurately diagnosing the pneumonia disease.