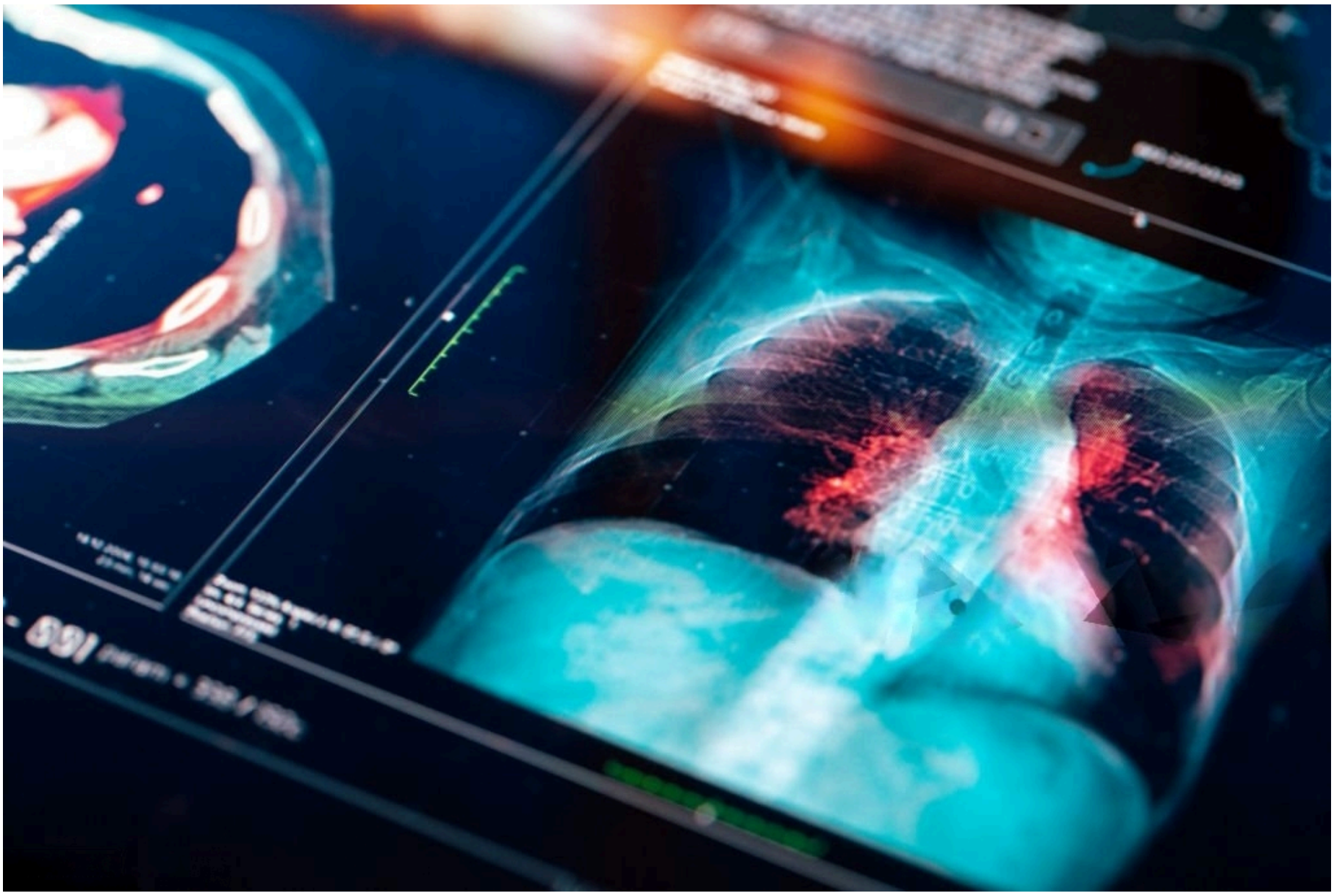
In [80]:
```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

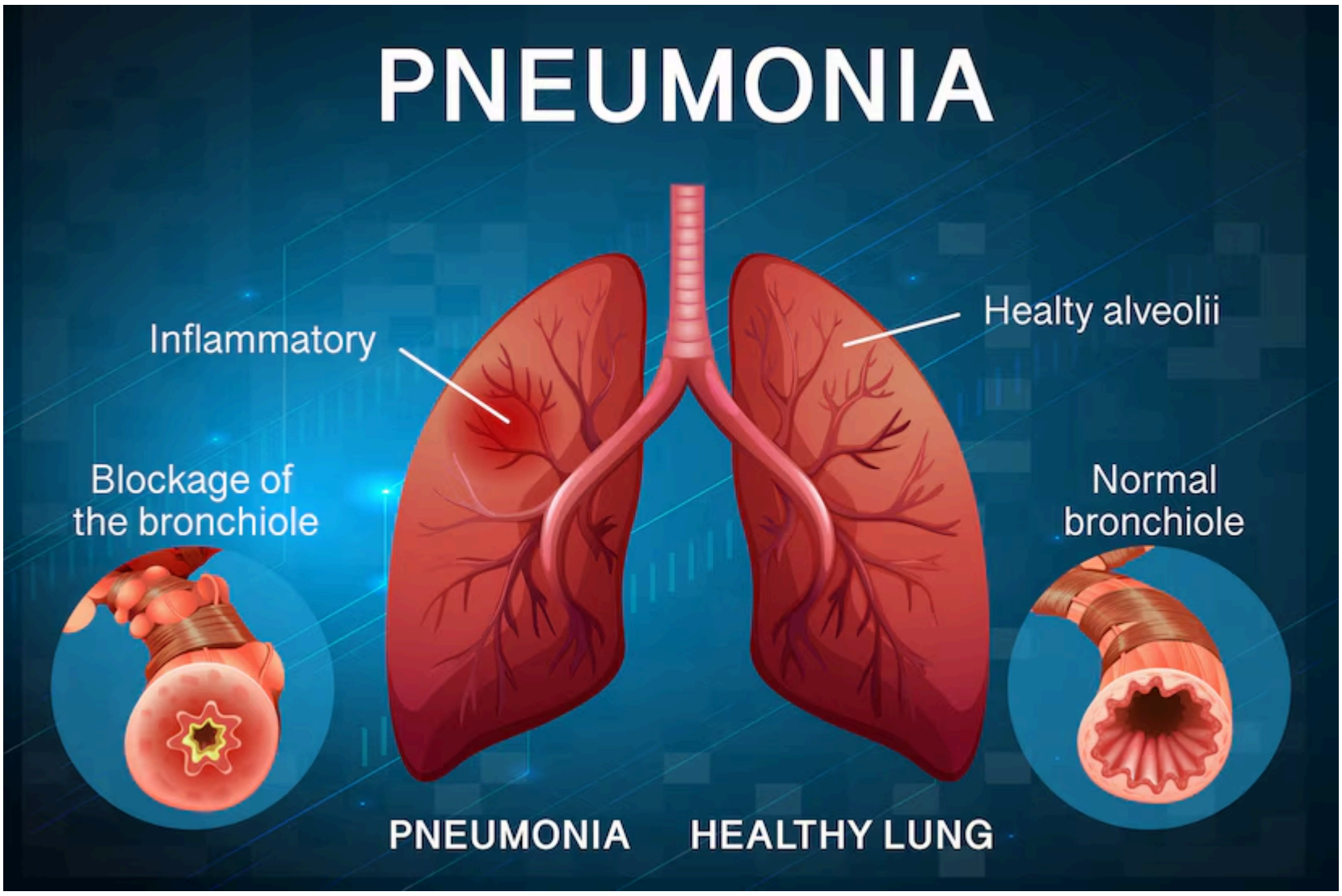


A Deep Learning Approach to Pneumonia Detection

Executive Summary

In this project, we address the critical task of pneumonia detection through the analysis of chest X-ray images using deep learning methodologies. Our comprehensive approach involved careful data preprocessing to ensure data quality and consistency. We conducted an extensive model comparison, evaluating various architectures including a baseline CNN, a tuned CNN with hyperparameter optimization, a modified CNN architecture, and a pre-trained ResNet50v2 model. Through rigorous experimentation, the ResNet50v2 model emerged as the top performer, achieving an impressive test accuracy of 92%. Our findings underscore the potential of deep learning in medical image analysis, with implications for improved diagnostic accuracy and patient care. Moving forward, we recommend further optimization of model deployment to effectively utilize computational resources, alongside exploration of advanced data augmentation techniques to bolster model robustness and generalization.

Problem Statement



`Pneumonia` is a leading cause of respiratory illness to humanity. It poses a significant threat to vulnerable populations, particularly young children and older adults. Therefore, early and accurate diagnosis is essential for successful treatment and improved patient outcomes. Current methods for diagnosing pneumonia, primarily relying on chest X-ray interpretation by radiologists, have limitations that include time constraints that lead to delays in treatment, potential for subjectivity leading to misdiagnosis, and high dependence on specialized expertise not always readily available. A recent article by the National Library of Medicine reveals that the average accuracy of human readiologits in detecting pneumonia cases is about 60%, **Link here (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8506182/)**

To curb the increasing fatalities from Pneumonia, the need for faster, more objective, and accessible diagnostic tools for pneumonia detection is rapidly growing, especially with the continuous advancement in technology.

With the above in mind, we explore the potential of deep learning to address these challenges by developing an automated system for pneumonia identification from chest X-ray images. Deep learning has revolutionized medical image analysis due to its ability to learn complex patterns from large datasets.The ability to extract intricate patterns from vast datasets, have transformed the field of medical image analysis by offering a unique opportunity to automate such analyses. We shall therefore harness this approach and capabilities to automate pneumonia detection with an objective to improve efficiency, accuracy, and accessibility of diagnosis.

Research Questions

1. Which deep learning model architecture achieves the best performance in terms of accuracy, sensitivity, specificity in detecting pneumonia from chest X-ray images?
2. How can data augmentation techniques be employed to improve the generalizability and robustness of the deep learning model for pneumonia detection on unseen data?
3. Can transfer learning from pretrained models such as ResNet50v2 improve pneumonia detection performance?
4. How do pneumonia detection models perform compared to human radiologists, and what are the implications for clinical practice?

Outline Approach

1. `Data Acquisition and Preprocessing` - This involves use of representative data that contains both healthy and pneumonia-infected cases. We annotate the images with labels indicating the presence or absence of pneumonia. We also enhance image quality and consistency by resizing, normalization/standardizations of values to a common range, facilitating better training for the deep learning model and reduction of noise to improve the model's ability to identify the true underlying structures in the X-ray.
2. `Model Development` - This involves selection of a suitable deep learning architecture such as Convolutional Neural Network(CNN) that has had good traction in medical image analysis. We then train the model on the preprocessed dataset, splitting it into training, vaidation, and tests sets. We also address class imbalance, if any, by data augmentation to artificially expand the training data then monitor the training process by adjusting parameters for optimal performance.
3. `Model Evaluation` - In the model evaluation phase, we assess the performance of the trained models using key metrics such as accuracy and confusion matrices. Through the interpretation of these metrics, we gain insights into the models' efficacy in distinguishing between normal and pneumonia cases in X-ray images. Additionally, visualization techniques are employed to dissect the models' predictions, providing valuable insights into their strengths and weaknesses. This comprehensive analysis allows for informed decisions regarding model refinement and optimization, ultimately ensuring the development of robust classifiers for accurate binary image classification.

4. `Interpretation and Deployment` - We explore techniques for explaining the model's decision-making process for instance, saliency maps, class activation maps to gain insights into how it differentiates between healthy and pneumonia-infected X-rays. If the model achieves satisfactory performance, consider deploying it in a controlled clinical environment for further validation and potential real-world use cases.

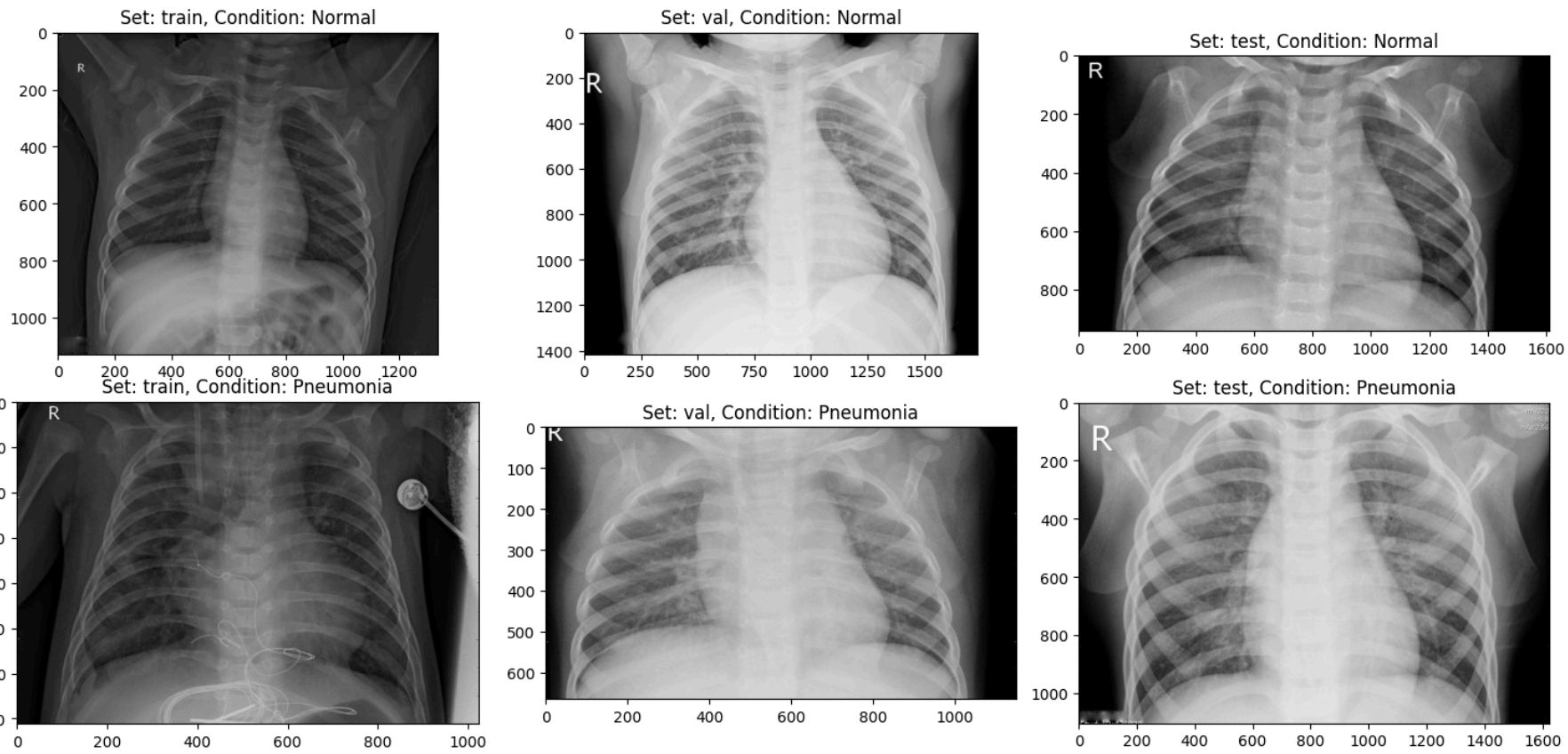
Importing Libraries and Dataset

In [2]:
```
# Data manipulation libraries
import pandas as pd
import numpy as np

# System libraries
import os
import re

# Image processing libraries
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from PIL import Image
%matplotlib inline
import seaborn as sns

from sklearn.metrics import confusion_matrix
from PIL import Image

# Deep learning libraries
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img
from tensorflow.keras.utils import plot_model
from tensorflow.keras.applications import ResNet50V2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten

# Suppress warnings
import warnings
warnings.filterwarnings("ignore")

import datetime
```

Exploratory Data Analysis (EDA)

The Exploratory Data Analysis (EDA) section serves as a crucial initial step in understanding the dataset's characteristics and structure before model development. This phase involves analyzing fundamental attributes such as the number of images per directory, distribution of classes, and image sizes. By examining these key aspects, we gain insights into the dataset's composition and potential challenges, informing subsequent preprocessing and modeling decisions. EDA acts as a foundation for effective data-driven strategies, facilitating the development of robust and reliable deep learning models.

In [3]:
```
# Random visualization from all directories (train, test, validation)

input_path="../input/chest-xray-pneumonia/chest_xray/chest_xray/"
fig, ax = plt.subplots(2, 3, figsize=(15, 7))
ax = ax.ravel()
plt.tight_layout()

for i, _set in enumerate(['train', 'val', 'test']):
    set_path = input_path+_set
    ax[i].imshow(plt.imread(set_path+'/NORMAL/'+os.listdir(set_path+'/NORMAL')[0]), cmap='gray')
    ax[i].set_title('Set: {}, Condition: Normal'.format(_set))
    ax[i+3].imshow(plt.imread(set_path+'/PNEUMONIA/'+os.listdir(set_path+'/PNEUMONIA')[0]), cmap='gray')
    ax[i+3].set_title('Set: {}, Condition: Pneumonia'.format(_set))
```

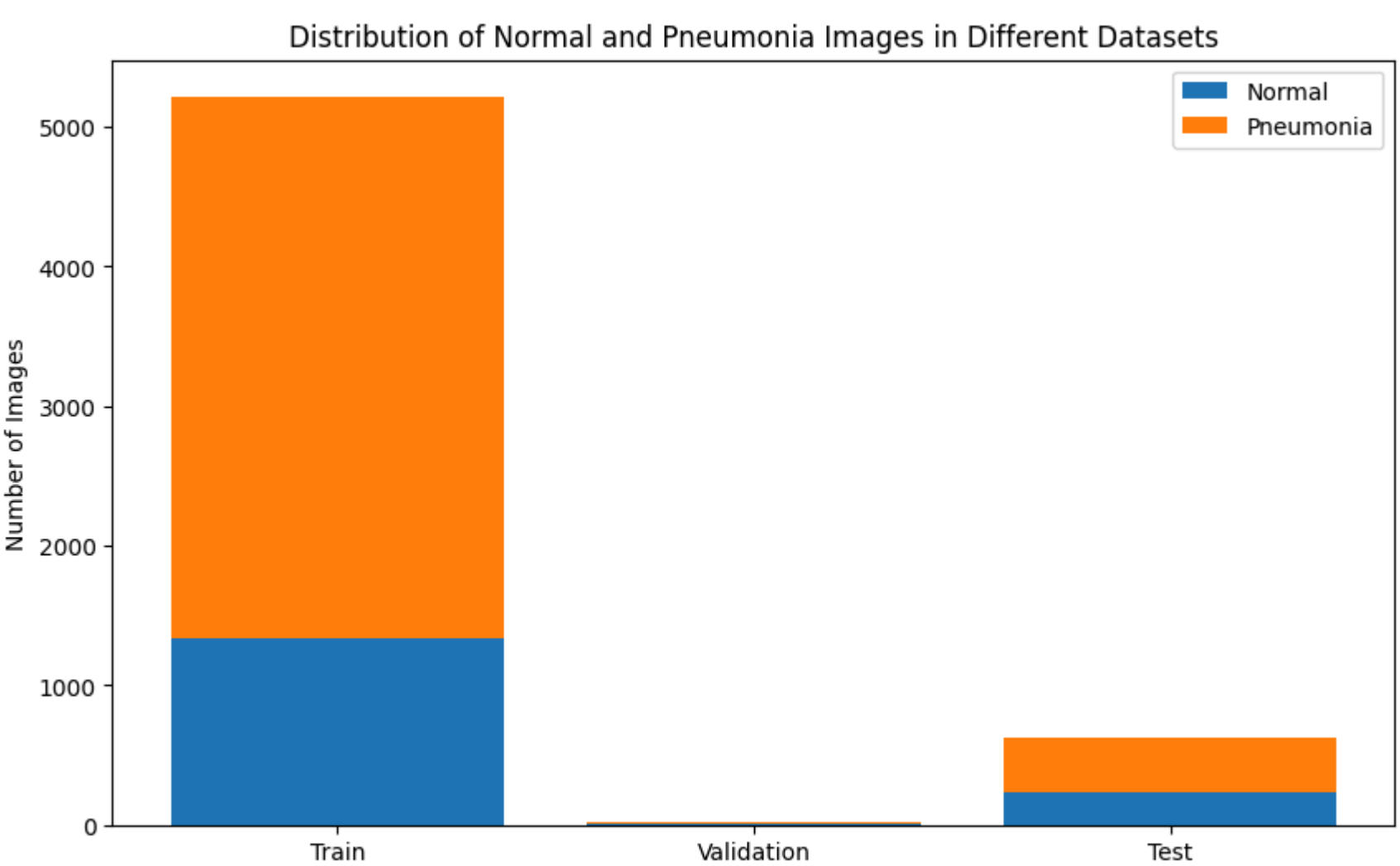


In [4]:
```
# Looping through the directories to do an image count

for _set in ['train', 'val', 'test']:
    n_normal = len(os.listdir(input_path + '/' + _set + '/NORMAL'))
    n_infect = len(os.listdir(input_path + '/' + _set + '/PNEUMONIA'))
    print('Set: {}, Normal images: {}, pneumonia images: {}'.format(_set, n_normal, n_infect))
```

```
Set: train, Normal images: 1342, pneumonia images: 3876
Set: val, Normal images: 9, pneumonia images: 9
Set: test, Normal images: 234, pneumonia images: 390
```

In [5]:
```
# Initialize lists to store counts
normal_counts = []
pneumonia_counts = []

# Loop through the sets
for _set in ['train', 'val', 'test']:
    n_normal = len(os.listdir(input_path + '/' + _set + '/NORMAL'))
    n_pneumonia = len(os.listdir(input_path + '/' + _set + '/PNEUMONIA'))

    # Append counts to lists
    normal_counts.append(n_normal)
    pneumonia_counts.append(n_pneumonia)

    print('Set: {}, Normal images: {}, pneumonia images: {}'.format(_set, n_normal, n_pneumonia))

# Create bar plot
plt.figure(figsize=(10, 6))
plt.bar(['Train', 'Validation', 'Test'], normal_counts, label='Normal')
plt.bar(['Train', 'Validation', 'Test'], pneumonia_counts, bottom=normal_counts, label='Pneumonia')
plt.xlabel('Dataset')
plt.ylabel('Number of Images')
plt.title('Distribution of Normal and Pneumonia Images in Different Datasets')
plt.legend()
plt.show()
```

```
Set: train, Normal images: 1342, pneumonia images: 3876
Set: val, Normal images: 9, pneumonia images: 9
Set: test, Normal images: 234, pneumonia images: 390
```



- From the bar plots above, it's evident that the distribution of images across the train, validation, and test datasets varies, with the validation set containing the fewest number of images. Given the potential risk of overfitting associated with a small validation set, implementing data augmentation emerges as a viable strategy.
- No significant class imbalance is present in the dataset.

In [6]:
```
# Saving directory paths into variables for easier reference

train_dir="/kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/train/"
val_dir="/kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/val/"
test_dir="/kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/test/"
```

In [7]:
```
import os
import cv2

# Looping through the 3 directories and extracting the max, min, and average image dimensions

def analyze_image_sizes(directory):
    class_labels = [label for label in os.listdir(directory) if os.path.isdir(os.path.join(directory, label))]
    for class_label in class_labels:
        class_dir = os.path.join(directory, class_label)
        image_sizes = []
        for image_file in os.listdir(class_dir):
            image_path = os.path.join(class_dir, image_file)
            if os.path.isfile(image_path):
                image = cv2.imread(image_path)
                if image is not None:
                    width, height = image.shape[:2]
                    image_sizes.append((width, height))
                else:
                    print("Warning: Failed to read image:", image_path)
            else:
                print("Warning: File not found:", image_path)

        if len(image_sizes) > 0:
            print(f"Class: {class_label}")
            print(f"Number of images: {len(image_sizes)}")
            min_width, min_height = min(image_sizes)
            max_width, max_height = max(image_sizes)
            average_width = sum(width for width, _ in image_sizes) / len(image_sizes)
            average_height = sum(height for _, height in image_sizes) / len(image_sizes)
            print(f"Minimum size: ({min_width}, {min_height})")
            print(f"Maximum size: ({max_width}, {max_height})")
            print(f"Average size: ({average_width:.2f}, {average_height:.2f})")
            print("-" * 20)
        else:
            print(f"No images found in class: {class_label}")

# Directories
train_dir = "/kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/train/"
val_dir = "/kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/val/"
test_dir = "/kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/test/"

# Analyze image sizes for each directory
print("Analysis of Train Directory:")
analyze_image_sizes(train_dir)

print("\nAnalysis of Validation Directory:")
analyze_image_sizes(val_dir)

print("\nAnalysis of Test Directory:")
analyze_image_sizes(test_dir)
```

```
Analysis of Train Directory:
Warning: Failed to read image: /kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/train/PNEUMONIA/.DS_Store
Class: PNEUMONIA
Number of images: 3875
Minimum size: (127, 384)
Maximum size: (2304, 2160)
Average size: (825.03, 1200.48)
--------------------
Warning: Failed to read image: /kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/train/NORMAL/.DS_Store
Class: NORMAL
Number of images: 1341
Minimum size: (672, 912)
Maximum size: (2663, 2373)
Average size: (1381.43, 1667.73)
--------------------

Analysis of Validation Directory:
Warning: Failed to read image: /kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/val/PNEUMONIA/.DS_Store
Class: PNEUMONIA
Number of images: 8
Minimum size: (592, 968)
Maximum size: (1128, 1664)
Average size: (814.00, 1217.00)
--------------------
Warning: Failed to read image: /kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/val/NORMAL/.DS_Store
Class: NORMAL
Number of images: 8
Minimum size: (928, 1288)
Maximum size: (1416, 1776)
Average size: (1191.88, 1479.50)
--------------------

Analysis of Test Directory:
Class: PNEUMONIA
Number of images: 390
Minimum size: (344, 888)
Maximum size: (1456, 2000)
Average size: (765.29, 1140.82)
--------------------
Class: NORMAL
Number of images: 234
Minimum size: (496, 984)
Maximum size: (2713, 2517)
Average size: (1369.09, 1800.30)
--------------------
```

Data Preprocessing

In this data preprocessing section, the raw image data is to be transformed and prepared for model training. This critical step involves various operations/transformations such as resizing images to a consistent size, normalizing pixel values, and obtaining train, validation and test generators for efficient loading during training. Additionally, augmenting the dataset through techniques like rotation, flipping, or shifting to enhance model generalization will be explored. By carefully preprocessing the data, we ensure that it is in a suitable format for the subsequent training and evaluation phases, laying the groundwork for effective deep learning

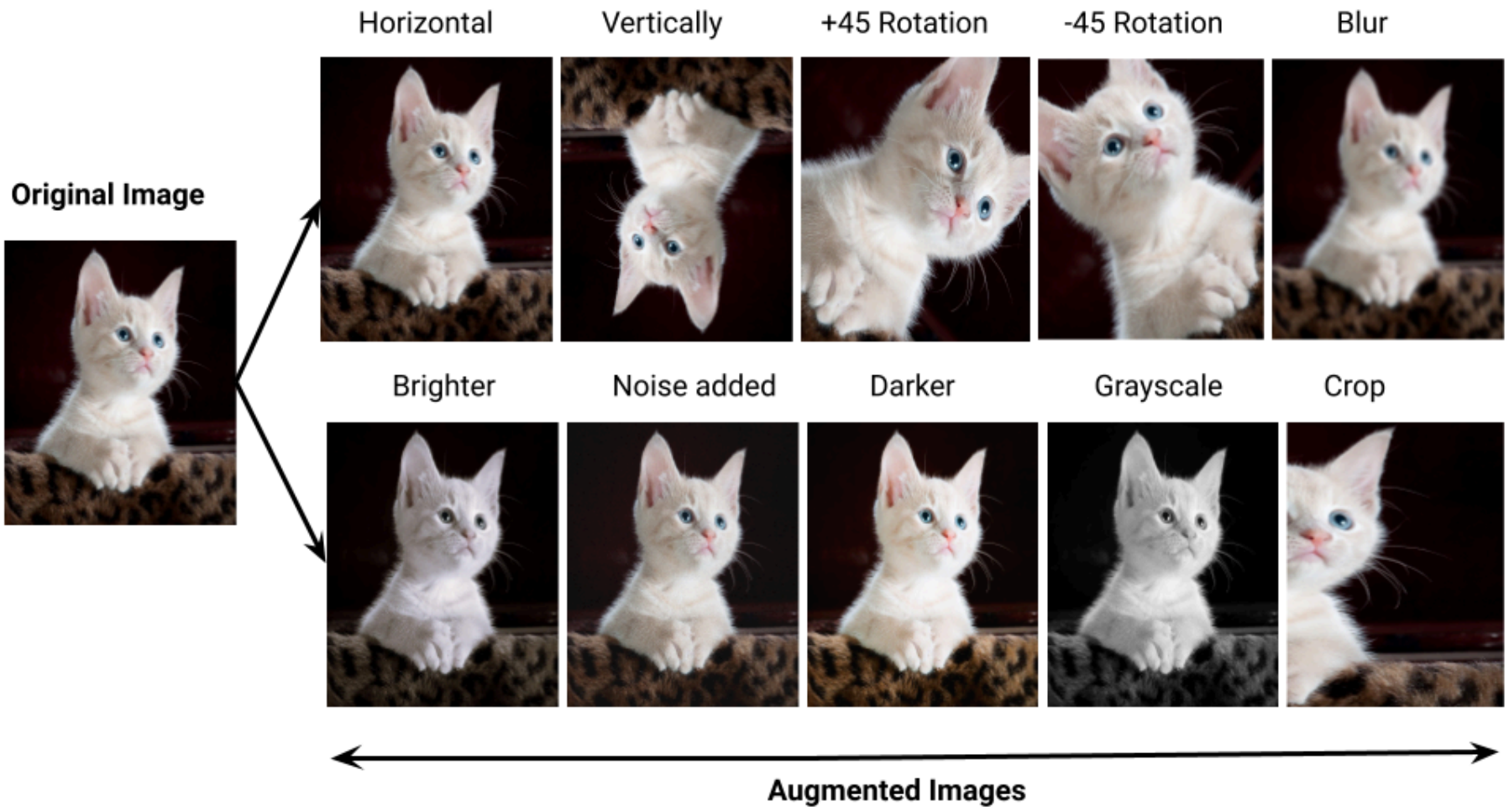
Data Augmentation

Why Augment Data?

- Data augmentation aims to enhance model performance by artificially/sythetically expanding the training dataset. By systematically applying transformations like rotations, flips, and zooms to existing data samples, data augmentation diversifies the dataset, enabling the model to encounter a broader range of scenarios during training. This process fosters robustness, mitigates overfitting, and promotes better generalization to unseen data.

Importance of Data Augmentation

- Data augmentation plays a critical role in addressing common challenges in deep learning projects. It enhances model robustness by exposing the model to a wider array of data variations, while also mitigating overfitting by providing additional training samples. With a larger and more diverse training dataset, the model can learn more effectively, leading to improved convergence rates and better optimization performance. Overall, data augmentation serves as a cornerstone technique, empowering models to learn from richer and more diverse training data while achieving better performance on real-world tasks.



Data Normalization & Image Resizing

`Normalization` is a crucial preprocessing step in deep learning with dual benefits:

- It accelerates the training process by ensuring that all input features are on the same scale, thereby preventing certain features from dominating others during gradient descent.
- Facilitates model convergence to a stable solution by stabilizing the learning process, allowing the model to learn more efficiently and effectively from the data.

Normalization will be performed independently for each set to help maintain the integrity of the data and ensures that the model generalizes well to unseen data.

In [8]:
```
# Normalization and Augmentation

Image_gen = ImageDataGenerator(
        rescale = 1/255,
        shear_range=10,
        zoom_range=0.3,
        horizontal_flip=True,
        vertical_flip=True,
        brightness_range=[0.5,2.0],
        width_shift_range = 0.2,
        rotation_range=20,
        fill_mode = 'nearest'
)
val_Datagen = ImageDataGenerator(
        rescale = 1/255
)
```

- Above we initialized two ImageDataGenerator objects, Image_gen and val_Datagen, responsible for data normalization and augmentation. The Image_gen object is configured to perform normalization by rescaling the pixel values to the range [0, 1]. Additionally, it includes augmentation techniques such as shearing, zooming, horizontal and vertical flipping, brightness adjustments, width shifting, rotation, and nearest fill mode. These augmentations help enhance the diversity and robustness of the training data, ultimately improving the model's generalization capabilities. The val_Datagen object, on the other hand, solely applies rescaling for normalization to the validation data, ensuring consistency with the training process.

In [83]:
```
# Define data generators for training, validation, and test sets

train = Image_gen.flow_from_directory(train_dir,
                                      batch_size=32,
                                      class_mode='binary',
                                       target_size=(224,224)
                                      )

validation = Image_gen.flow_from_directory(val_dir,
                                           batch_size=2,
                                           class_mode='binary',
                                           target_size=(224,224)
                                          )

test = val_Datagen.flow_from_directory(test_dir,
                                      batch_size=2,
                                      class_mode='binary',
                                       target_size=(224,224)
                                     )
```

```
Found 5216 images belonging to 2 classes.
Found 16 images belonging to 2 classes.
Found 624 images belonging to 2 classes.
```

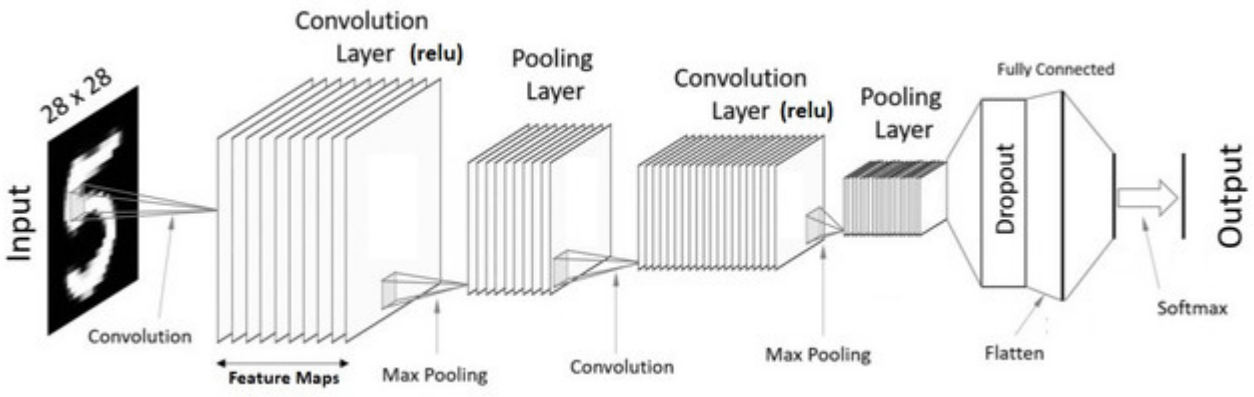
- The code above defines data generators for training, validation, and test sets using the `ImageDataGenerator` class from keras. These generators are essential for loading and preprocessing image data efficiently during model training and evaluation. Each generator is configured to load images from their respective directories (`train_dir` , `validation_dir` , and `test_dir`). The `flow_from_directory` method ensures that images are loaded in batches, allowing for efficient memory usage. Additionally, the `target_size` parameter specifies the dimensions to which the input images will be resized for consistency across the dataset. The `class_mode` parameter is set to `'binary'` indicating a binary classification problem.

Modeling

In the modeling phase, we transition from data exploration and data preprocessing to the development and training of two distinct convolutional neural network (CNN) architectures tailored to our binary image classification task of pneumonia diagnosis from X-ray images. Our first model, the baseline CNN, will serve as a foundational architecture, while the second model will leverage the ResNet50V2 architecture, known for its depth and performance. Leveraging insights from exploratory data analysis (EDA), we aim to optimize these models for image size, class distribution, and regularization techniques to mitigate overfitting. Through rigorous experimentation and evaluation, we seek to identify the most effective model configurations, utilizing metrics such as accuracy and loss to guide refinement and ensure robust, high-performing classifiers capable of accurately discerning `normal` from `pneumonia` cases in X-ray images.

(1) Convolutional Neutral Network (CNN) - Baseline Model

Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for image classification tasks. In a CNN, the input image is passed through a series of convolutional layers, each comprising filters that extract features such as edges, textures, and shapes. These features are then aggregated and processed by subsequent layers, including pooling layers for downsampling and fully connected layers for classification. The baseline CNN model typically consists of convolutional layers followed by pooling layers to extract and aggregate features, and fully connected layers for classification. Through training on labeled image data, CNNs learn to automatically detect patterns and features in images, enabling accurate classification of unseen images. Below is a schematic of the same..



(https://ibb.co/kV1j9p)

In [10]:
```
# defining a function for training a model, validating on the validation set
# and evaluating its performance on the test set

def train_and_evaluate_model(model, train_generator, validation_generator, test_generator, epochs=30):
    """
    Train the model, evaluate on validation and test sets, plot training history (loss and accuracy),
    and plot train vs test accuracy and train vs test loss.
    """
    # Train the model
    history = model.fit(
        train_generator,
        epochs=epochs,
        validation_data=validation_generator,
        verbose=1
    )

    # Evaluate on validation set
    validation_loss, validation_accuracy = model.evaluate(validation_generator, verbose=0)
    print("Validation Loss:", validation_loss)
    print("Validation Accuracy:", validation_accuracy)

    # Evaluate on test set
    test_loss, test_accuracy = model.evaluate(test_generator, verbose=0)
    print("Test Loss:", test_loss)
    print("Test Accuracy:", test_accuracy)

    # Plot training history (loss and accuracy)
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.legend()

    plt.show()

    # Plot train vs test accuracy
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.axhline(y=test_accuracy, color='r', linestyle='--', label='Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Train vs Test Accuracy')
    plt.legend()
    plt.show()

    # Plot train vs test loss
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.axhline(y=test_loss, color='r', linestyle='--', label='Test Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Train vs Test Loss')
    plt.legend()
    plt.show()

    return history
```

In [11]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

def cnn_model(input_shape):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(128, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    return model

# Define input shape based on your image dimensions (e.g., (height, width, channels))
input_shape = (224, 224, 3)

# Create the baseline CNN model
baseline_model = cnn_model(input_shape)

# Compile the model
baseline_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

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

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 128)	0
flatten (Flatten)	(None, 86528)	0
dense (Dense)	(None, 128)	11,075,712
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 1)	129

Total params: 11,169,089 (42.61 MB)

Trainable params: 11,169,089 (42.61 MB)

Non-trainable params: 0 (0.00 B)

- The baseline model architecture comprises a total of 11,169,089 parameters, with all parameters being trainable. The model's size is approximately 42.61 MB. There are no non-trainable parameters in this architecture, indicating that all parameters are updated during the training process.

In [12]:
```
import datetime
start = datetime.datetime.now()
```

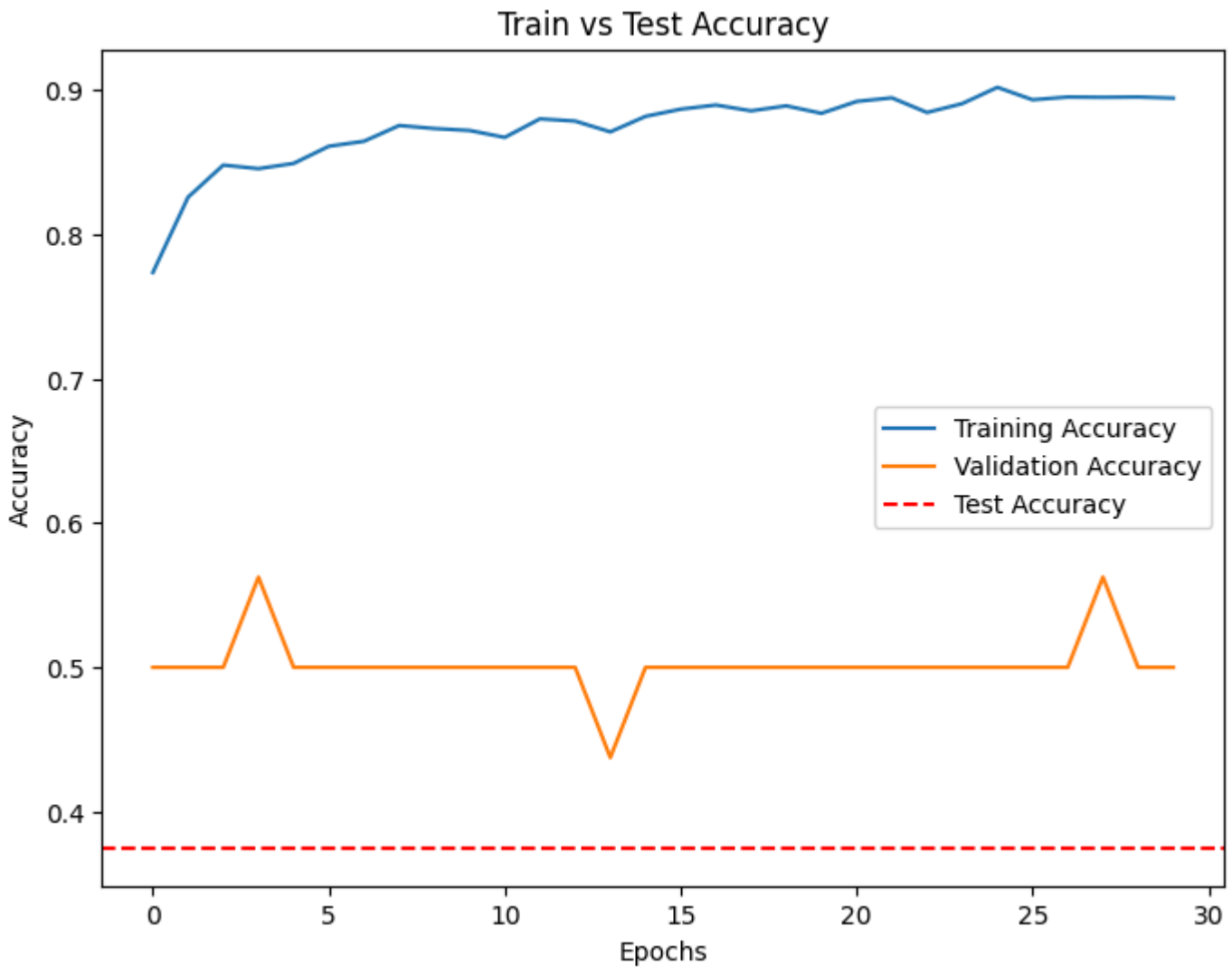
In [13]:
```
train_and_evaluate_model(baseline_model, train, validation, test, epochs=30)
```

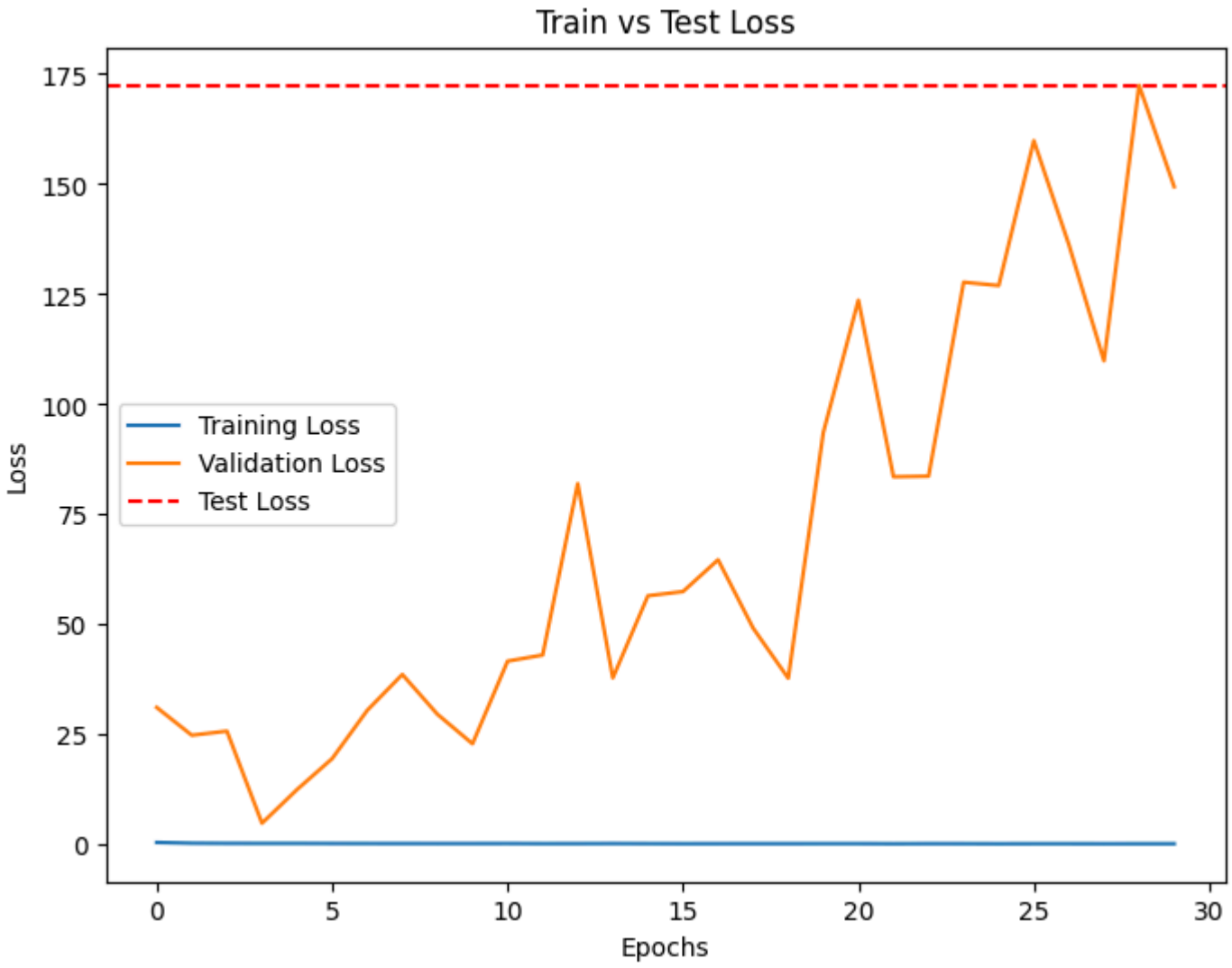
```
Epoch 1/30

2024-03-21 11:56:38.412163: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49284: 7.84
353, expected 6.92193
2024-03-21 11:56:38.412222: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49286: 7.76
792, expected 6.84632
2024-03-21 11:56:38.412236: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49287: 6.15
631, expected 5.23472
2024-03-21 11:56:38.412249: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49288: 6.71
859, expected 5.797
2024-03-21 11:56:38.412269: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49289: 7.43
913, expected 6.51753
2024-03-21 11:56:38.412281: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49290: 7.17
83, expected 6.25671
2024-03-21 11:56:38.412291: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49291: 6.06
838, expected 5.14679
2024-03-21 11:56:38.412301: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49292: 6.24
725, expected 5.32565
2024-03-21 11:56:38.412313: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49293: 6.19
403, expected 5.27243
2024-03-21 11:56:38.412324: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49294: 7.10
498, expected 6.18338
2024-03-21 11:56:38.434182: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:705] Results mismatch betwe
en different convolution algorithms. This is likely a bug/unexpected loss of precision in cudnn.
(f32[32,32,222,222]{3,2,1,0}, u8[0]{0}) custom-call(f32[32,3,224,224]{3,2,1,0}, f32[32,3,3,3]{3,2,1,0}, f32[32]{0}),
window={size=3x3}, dim_labels=bf01_oi01->bf01, custom_call_target="__cudnn$convBiasActivationForward", backend_config
={"conv_result_scale":1,"activation_mode":"kRelu","side_input_scale":0,"leakyrelu_alpha":0} for eng20{k2=2,k4=1,k5=1,
k6=0,k7=0} vs eng15{k5=1,k6=0,k7=1,k10=1}
2024-03-21 11:56:38.434223: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:270] Device: Tesla P100-PCI
E-16GB
2024-03-21 11:56:38.434236: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:271] Platform: Compute Capa
bility 6.0
2024-03-21 11:56:38.434248: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:272] Driver: 12020 (535.12
9.3)
2024-03-21 11:56:38.434263: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:273] Runtime: <undefined>
2024-03-21 11:56:38.434285: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:280] cudnn version: 8.9.0
2024-03-21 11:56:39.166799: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49284: 7.84
353, expected 6.92193
2024-03-21 11:56:39.166853: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49286: 7.76
792, expected 6.84632
2024-03-21 11:56:39.166870: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49287: 6.15
631, expected 5.23472
2024-03-21 11:56:39.166887: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49288: 6.71
859, expected 5.797
2024-03-21 11:56:39.166901: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49289: 7.43
913, expected 6.51753
2024-03-21 11:56:39.166930: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49290: 7.17
83, expected 6.25671
2024-03-21 11:56:39.166941: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49291: 6.06
838, expected 5.14679
2024-03-21 11:56:39.166952: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49292: 6.24
725, expected 5.32565
2024-03-21 11:56:39.166963: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49293: 6.19
403, expected 5.27243
2024-03-21 11:56:39.166974: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 49294: 7.10
498, expected 6.18338
2024-03-21 11:56:39.189494: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:705] Results mismatch betwe
en different convolution algorithms. This is likely a bug/unexpected loss of precision in cudnn.
(f32[32,32,222,222]{3,2,1,0}, u8[0]{0}) custom-call(f32[32,3,224,224]{3,2,1,0}, f32[32,3,3,3]{3,2,1,0}, f32[32]{0}),
window={size=3x3}, dim_labels=bf01_oi01->bf01, custom_call_target="__cudnn$convBiasActivationForward", backend_config
={"conv_result_scale":1,"activation_mode":"kRelu","side_input_scale":0,"leakyrelu_alpha":0} for eng20{k2=2,k4=1,k5=1,
k6=0,k7=0} vs eng15{k5=1,k6=0,k7=1,k10=1}
2024-03-21 11:56:39.189540: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:270] Device: Tesla P100-PCI
E-16GB
2024-03-21 11:56:39.189554: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:271] Platform: Compute Capa
bility 6.0
2024-03-21 11:56:39.189567: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:272] Driver: 12020 (535.12
9.3)
2024-03-21 11:56:39.189581: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:273] Runtime: <undefined>
2024-03-21 11:56:39.189603: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:280] cudnn version: 8.9.0

  1/163 ━━━━━━━━━━━━━━━━━━━━ 38:33 14s/step - accuracy: 0.6875 - loss: 0.6437

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1711022204.539738     108 device_compiler.h:186] Compiled cluster using XLA!  This line is logged at most
once for the lifetime of the process.

162/163 ━━━━━━━━━━━━━━━━━━━━ 0s 560ms/step - accuracy: 0.7429 - loss: 0.8185
```

```
2024-03-21 11:58:15.151848: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 11: 3.4831,
expected 2.95098
2024-03-21 11:58:15.151906: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 28: 4.1217
3, expected 3.58961
2024-03-21 11:58:15.151916: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 62: 4.2714
8, expected 3.73936
2024-03-21 11:58:15.151924: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 66: 3.8020
7, expected 3.26995
2024-03-21 11:58:15.151932: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 83: 3.9544
9, expected 3.42237
2024-03-21 11:58:15.151940: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 85: 4.1469
8, expected 3.61486
2024-03-21 11:58:15.151948: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 180: 3.9983
1, expected 3.46619
2024-03-21 11:58:15.151955: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 181: 3.8535
2, expected 3.3214
2024-03-21 11:58:15.151963: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 198: 3.8483
8, expected 3.31626
2024-03-21 11:58:15.151970: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 233: 4.1787
7, expected 3.64665
2024-03-21 11:58:15.151986: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:705] Results mismatch betwe
en different convolution algorithms. This is likely a bug/unexpected loss of precision in cudnn.
(f32[2,32,222,222]{3,2,1,0}, u8[0]{0}) custom-call(f32[2,3,224,224]{3,2,1,0}, f32[32,3,3,3]{3,2,1,0}, f32[32]{0}), wi
ndow={size=3x3}, dim_labels=bf01_oi01->bf01, custom_call_target="__cudnn$convBiasActivationForward", backend_config=
{"conv_result_scale":1,"activation_mode":"kRelu","side_input_scale":0,"leakyrelu_alpha":0} for eng20{k2=2,k4=1,k5=1,k
6=0,k7=0} vs eng15{k5=1,k6=0,k7=1,k10=1}
2024-03-21 11:58:15.151993: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:270] Device: Tesla P100-PCI
E-16GB
2024-03-21 11:58:15.152009: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:271] Platform: Compute Capa
bility 6.0
2024-03-21 11:58:15.152015: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:272] Driver: 12020 (535.12
9.3)
2024-03-21 11:58:15.152022: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:273] Runtime: <undefined>
2024-03-21 11:58:15.152032: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:280] cudnn version: 8.9.0
2024-03-21 11:58:15.202170: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 11: 3.4831,
expected 2.95098
2024-03-21 11:58:15.202216: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 28: 4.1217
3, expected 3.58961
2024-03-21 11:58:15.202225: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 62: 4.2714
8, expected 3.73936
2024-03-21 11:58:15.202233: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 66: 3.8020
7, expected 3.26995
2024-03-21 11:58:15.202241: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 83: 3.9544
9, expected 3.42237
2024-03-21 11:58:15.202249: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 85: 4.1469
8, expected 3.61486
2024-03-21 11:58:15.202257: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 180: 3.9983
1, expected 3.46619
2024-03-21 11:58:15.202265: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 181: 3.8535
2, expected 3.3214
2024-03-21 11:58:15.202272: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 198: 3.8483
8, expected 3.31626
2024-03-21 11:58:15.202280: E external/local_xla/xla/service/gpu/buffer_comparator.cc:1137] Difference at 233: 4.1787
7, expected 3.64665
2024-03-21 11:58:15.202294: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:705] Results mismatch betwe
en different convolution algorithms. This is likely a bug/unexpected loss of precision in cudnn.
(f32[2,32,222,222]{3,2,1,0}, u8[0]{0}) custom-call(f32[2,3,224,224]{3,2,1,0}, f32[32,3,3,3]{3,2,1,0}, f32[32]{0}), wi
ndow={size=3x3}, dim_labels=bf01_oi01->bf01, custom_call_target="__cudnn$convBiasActivationForward", backend_config=
{"conv_result_scale":1,"activation_mode":"kRelu","side_input_scale":0,"leakyrelu_alpha":0} for eng20{k2=2,k4=1,k5=1,k
6=0,k7=0} vs eng15{k5=1,k6=0,k7=1,k10=1}
2024-03-21 11:58:15.202304: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:270] Device: Tesla P100-PCI
E-16GB
2024-03-21 11:58:15.202315: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:271] Platform: Compute Capa
bility 6.0
2024-03-21 11:58:15.202326: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:272] Driver: 12020 (535.12
9.3)
2024-03-21 11:58:15.202344: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:273] Runtime: <undefined>
2024-03-21 11:58:15.202358: E external/local_xla/xla/service/gpu/conv_algorithm_picker.cc:280] cudnn version: 8.9.0
```

```
163/163 ━━━━━━━━━━━━━━━━━━━━ 106s 564ms/step - accuracy: 0.7433 - loss: 0.8152 - val_accuracy: 0.5000 - val_loss: 31.
1900
Epoch 2/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 94s 553ms/step - accuracy: 0.8101 - loss: 0.4054 - val_accuracy: 0.5000 - val_loss: 24.8
801
Epoch 3/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 559ms/step - accuracy: 0.8426 - loss: 0.3572 - val_accuracy: 0.5000 - val_loss: 25.8
220
Epoch 4/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 556ms/step - accuracy: 0.8453 - loss: 0.3371 - val_accuracy: 0.5625 - val_loss: 4.91
89
Epoch 5/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 142s 560ms/step - accuracy: 0.8451 - loss: 0.3383 - val_accuracy: 0.5000 - val_loss: 12.
5605
Epoch 6/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 557ms/step - accuracy: 0.8551 - loss: 0.3206 - val_accuracy: 0.5000 - val_loss: 19.5
797
Epoch 7/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 557ms/step - accuracy: 0.8657 - loss: 0.3065 - val_accuracy: 0.5000 - val_loss: 30.5
356
Epoch 8/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 556ms/step - accuracy: 0.8754 - loss: 0.2945 - val_accuracy: 0.5000 - val_loss: 38.6
800
Epoch 9/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 556ms/step - accuracy: 0.8728 - loss: 0.2866 - val_accuracy: 0.5000 - val_loss: 29.6
506
Epoch 10/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 94s 553ms/step - accuracy: 0.8609 - loss: 0.3091 - val_accuracy: 0.5000 - val_loss: 22.9
504
Epoch 11/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 554ms/step - accuracy: 0.8690 - loss: 0.3067 - val_accuracy: 0.5000 - val_loss: 41.6
978
Epoch 12/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.8749 - loss: 0.2802 - val_accuracy: 0.5000 - val_loss: 43.0
979
Epoch 13/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 554ms/step - accuracy: 0.8849 - loss: 0.2655 - val_accuracy: 0.5000 - val_loss: 82.0
108
Epoch 14/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 97s 570ms/step - accuracy: 0.8683 - loss: 0.3110 - val_accuracy: 0.4375 - val_loss: 37.8
978
Epoch 15/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 562ms/step - accuracy: 0.8865 - loss: 0.2644 - val_accuracy: 0.5000 - val_loss: 56.5
410
Epoch 16/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.8909 - loss: 0.2561 - val_accuracy: 0.5000 - val_loss: 57.5
008
Epoch 17/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 560ms/step - accuracy: 0.8836 - loss: 0.2818 - val_accuracy: 0.5000 - val_loss: 64.7
203
Epoch 18/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 555ms/step - accuracy: 0.8828 - loss: 0.2699 - val_accuracy: 0.5000 - val_loss: 49.1
908
Epoch 19/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 558ms/step - accuracy: 0.8786 - loss: 0.2768 - val_accuracy: 0.5000 - val_loss: 37.8
299
Epoch 20/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 559ms/step - accuracy: 0.8751 - loss: 0.2913 - val_accuracy: 0.5000 - val_loss: 93.6
136
Epoch 21/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 565ms/step - accuracy: 0.8883 - loss: 0.2908 - val_accuracy: 0.5000 - val_loss: 123.
6608
Epoch 22/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 557ms/step - accuracy: 0.8934 - loss: 0.2617 - val_accuracy: 0.5000 - val_loss: 83.5
632
Epoch 23/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 555ms/step - accuracy: 0.8811 - loss: 0.2737 - val_accuracy: 0.5000 - val_loss: 83.6
938
Epoch 24/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 559ms/step - accuracy: 0.8737 - loss: 0.2880 - val_accuracy: 0.5000 - val_loss: 127.
7297
Epoch 25/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 559ms/step - accuracy: 0.9054 - loss: 0.2318 - val_accuracy: 0.5000 - val_loss: 126.
9766
Epoch 26/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 94s 555ms/step - accuracy: 0.9033 - loss: 0.2418 - val_accuracy: 0.5000 - val_loss: 159.
8766
Epoch 27/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 98s 576ms/step - accuracy: 0.8993 - loss: 0.2460 - val_accuracy: 0.5000 - val_loss: 136.
2623
Epoch 28/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 556ms/step - accuracy: 0.9009 - loss: 0.2340 - val_accuracy: 0.5625 - val_loss: 109.
8880
Epoch 29/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 560ms/step - accuracy: 0.8888 - loss: 0.2610 - val_accuracy: 0.5000 - val_loss: 172.
4390
Epoch 30/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.9022 - loss: 0.2426 - val_accuracy: 0.5000 - val_loss: 149.
```

```
4193
Validation Loss: 256.3694152832031
Validation Accuracy: 0.5
Test Loss: 172.31935119628906
Test Accuracy: 0.375
```



Out[13]: <keras.src.callbacks.history.History at 0x7dc57c3d88e0>

- The baseline model's test accuracy, currently at 38%, indicates substantial room for improvement. Moreover, signs of overfitting are evident. Therefore, refining the model architecture and incorporating regularization techniques are crucial steps to enhance performance and mitigate overfitting.

In [14]:
```
end = datetime.datetime.now()
```

In [15]:
```
elapsed = end - start
print('Time elapsed', elapsed)
```

```
Time elapsed 0:48:40.248647
```

The baseline model took about 49mins to complete the training process.

(2) Tuned Baseline Model (CNN)

In [87]:
```
from kerastuner.tuners import RandomSearch
from kerastuner.engine.hyperparameters import HyperParameters

# defining a function to find the best hyperparams and build a new model architecture

def tune_model(hp):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(128, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    # Tune the learning rate
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=hp_learning_rate),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Define input shape based on your image dimensions (e.g., (height, width, channels))
input_shape = (224, 224, 3)

# Instantiate the tuner
tuner = RandomSearch(
    tune_model,
    objective='val_accuracy',
    max_trials=3,  # you can increase this value for more trials
    executions_per_trial=1,
    directory='hyperparameter_tuning',
    project_name='tuned_baseline_cnn'
)

# Define data generators or use existing ones (train_generator, validation_generator, test_generator)

# Start the search for the best hyperparameters
tuner.search(train, validation_data=validation, epochs=10)

# Get the best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"Best learning rate: {best_hps.get('learning_rate')}")

# Build the model with the best hyperparameters
best_model = tuner.get_best_models(num_models=1)[0]
```

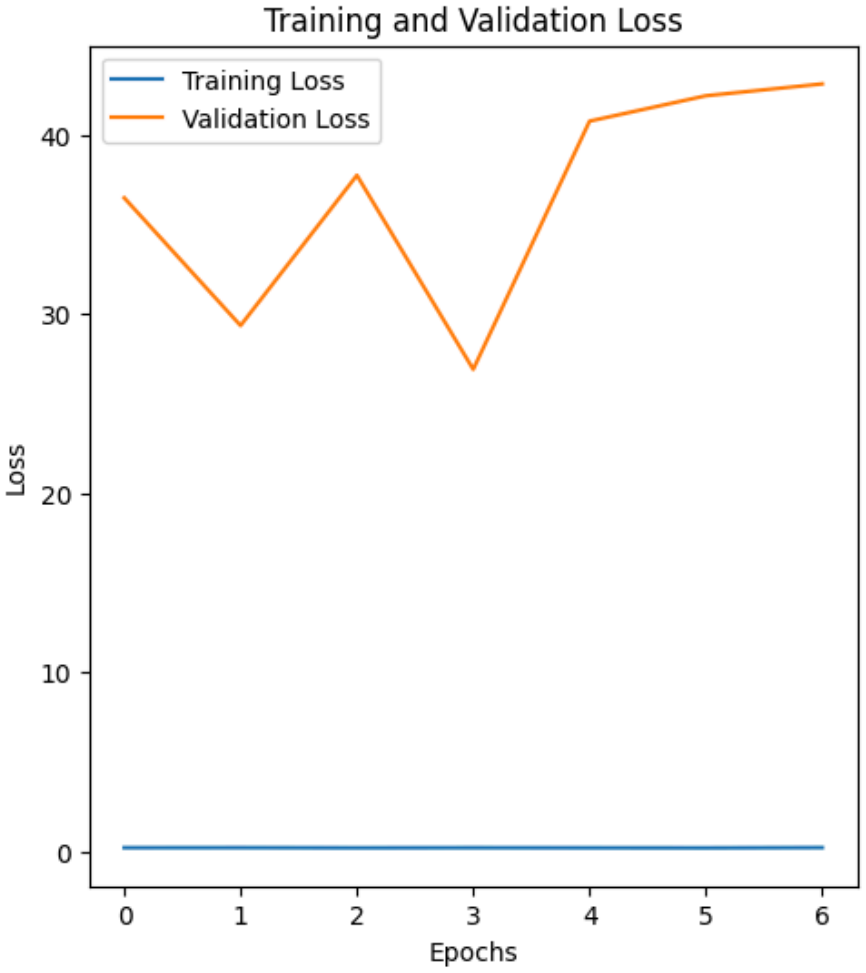
```
Reloading Tuner from hyperparameter_tuning/tuned_baseline_cnn/tuner0.json
Best learning rate: 0.001
```

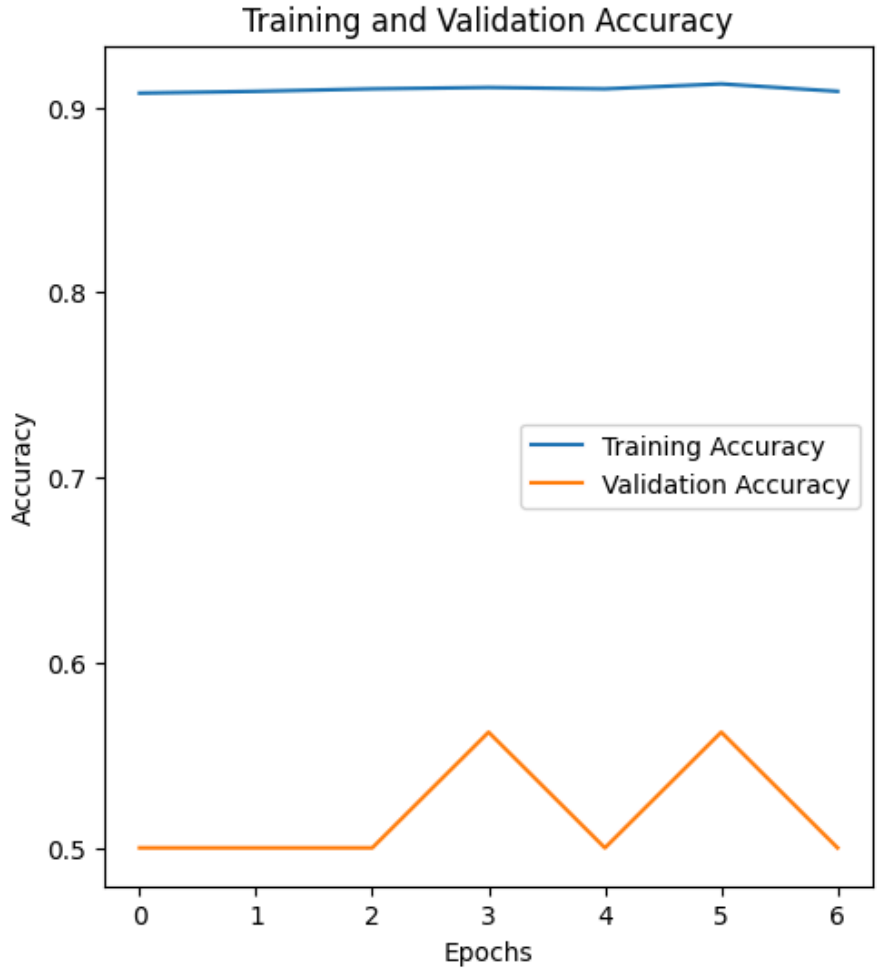
- It appears that the best learning rate is the default rate (0.001). We shall explore earlystopping technique, which is a regularization measure help to minimize overfitting.

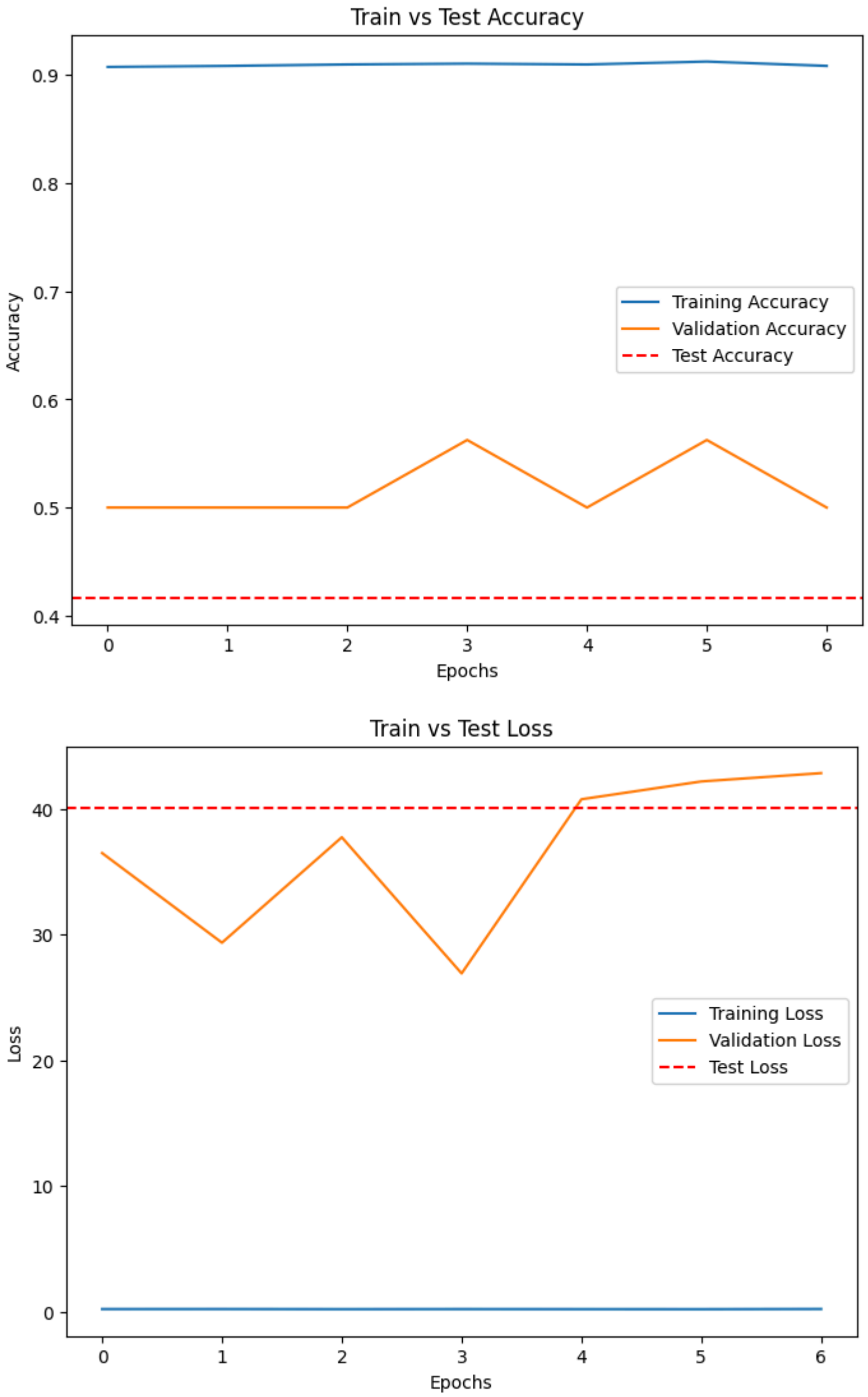
In [94]:
```
#redifining our train_and_evaluate_model function to incorporate early stopping

from tensorflow.keras.callbacks import EarlyStopping

def train_and_evaluate_model_es(model, train_generator, validation_generator, test_generator, epochs=30):
    """
    Train the model, evaluate on validation and test sets, plot training history (loss and accuracy),
    and plot train vs test accuracy and train vs test loss.
    """
    # Define early stopping callback
    early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

    # Train the model
    history = model.fit(
        train_generator,
        epochs=epochs,
        validation_data=validation_generator,
        callbacks=[early_stopping],  # Pass the early stopping callback
        verbose=1
    )

    # Evaluate on validation set
    validation_loss, validation_accuracy = model.evaluate(validation_generator, verbose=0)
    print("Validation Loss:", validation_loss)
    print("Validation Accuracy:", validation_accuracy)

    # Evaluate on test set
    test_loss, test_accuracy = model.evaluate(test_generator, verbose=0)
    print("Test Loss:", test_loss)
    print("Test Accuracy:", test_accuracy)

    # Plot training history (loss and accuracy)
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.legend()

    plt.show()

    # Plot train vs test accuracy
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.axhline(y=test_accuracy, color='r', linestyle='--', label='Test Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Train vs Test Accuracy')
    plt.legend()
    plt.show()

    # Plot train vs test loss
    plt.figure(figsize=(8, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.axhline(y=test_loss, color='r', linestyle='--', label='Test Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Train vs Test Loss')
    plt.legend()
    plt.show()

    return history
```

In [95]:
```
train_and_evaluate_model_es(baseline_model, train, validation, test)
```

```
Epoch 1/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 104s 604ms/step - accuracy: 0.9102 - loss: 0.2253 - val_accuracy: 0.5000 - val_loss: 36.
4714
Epoch 2/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 97s 568ms/step - accuracy: 0.9082 - loss: 0.2215 - val_accuracy: 0.5000 - val_loss: 29.3
474
Epoch 3/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 97s 573ms/step - accuracy: 0.9108 - loss: 0.2222 - val_accuracy: 0.5000 - val_loss: 37.7
282
Epoch 4/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.9099 - loss: 0.2168 - val_accuracy: 0.5625 - val_loss: 26.9
053
Epoch 5/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 562ms/step - accuracy: 0.9136 - loss: 0.2144 - val_accuracy: 0.5000 - val_loss: 40.7
493
Epoch 6/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 560ms/step - accuracy: 0.9094 - loss: 0.2210 - val_accuracy: 0.5625 - val_loss: 42.1
655
Epoch 7/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 559ms/step - accuracy: 0.9116 - loss: 0.2344 - val_accuracy: 0.5000 - val_loss: 42.8
272
Validation Loss: 25.11384391784668
Validation Accuracy: 0.5625
Test Loss: 40.0745735168457
Test Accuracy: 0.4166666567325592
```







Out[95]: <keras.src.callbacks.history.History at 0x7f2d0473ba30>

- By incorporating early stopping regularization technique, the model's accuracy slightly improved to 42% while effectively minimizing overfitting.

(3) Different Architecture Model

The test accuracy on the tuned baseline model is still low (42%), so we shall experiment with a more complex CNN model by adding an additional dense layer to the original architecture, while still incorporating early stopping.

In [97]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

def new_cnn_model(input_shape):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=input_shape),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(128, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(256, activation='relu'),
        Dropout(0.5),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(1, activation='sigmoid')
    ])
    return model

# Define input shape based on your image dimensions (e.g., (height, width, channels))
input_shape = (224, 224, 3)

# Create the new CNN model
new_model = new_cnn_model(input_shape)

# Compile the model
new_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

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

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d_6 (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_7 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_7 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_8 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_8 (MaxPooling2D)	(None, 26, 26, 128)	0
flatten_2 (Flatten)	(None, 86528)	0
dense_4 (Dense)	(None, 256)	22,151,424
dropout_2 (Dropout)	(None, 256)	0
dense_5 (Dense)	(None, 128)	32,896
dropout_3 (Dropout)	(None, 128)	0
dense_6 (Dense)	(None, 1)	129

Total params: 22,277,697 (84.98 MB)

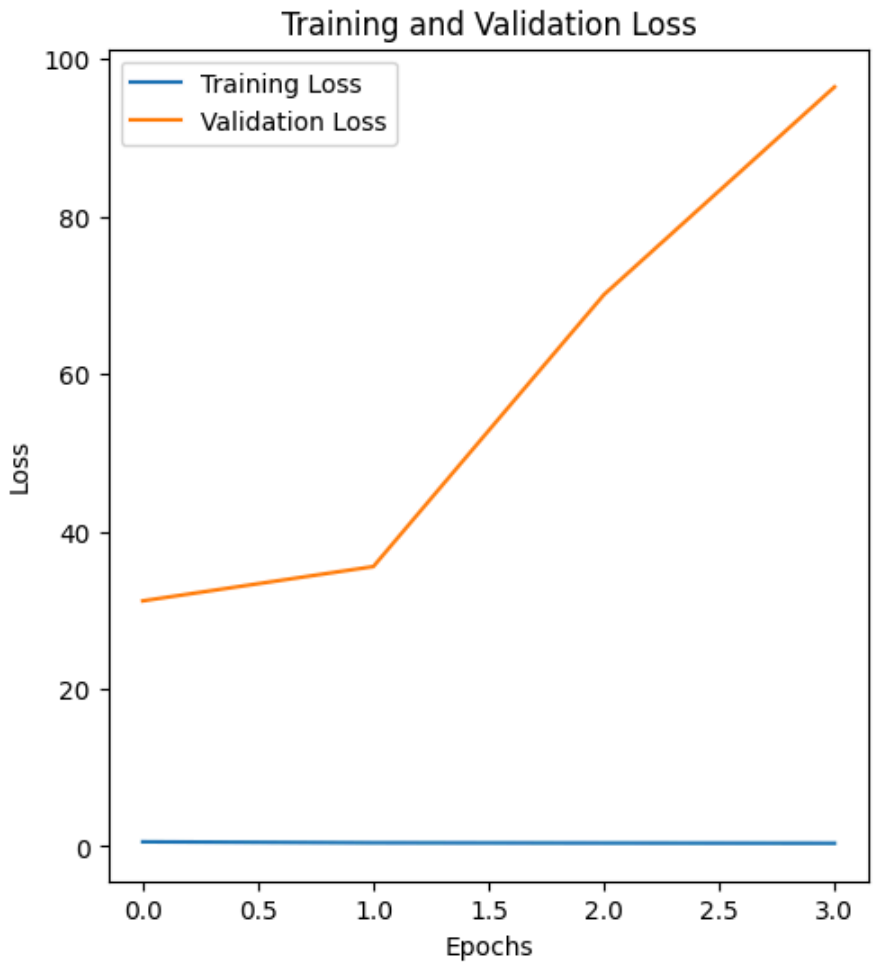
Trainable params: 22,277,697 (84.98 MB)

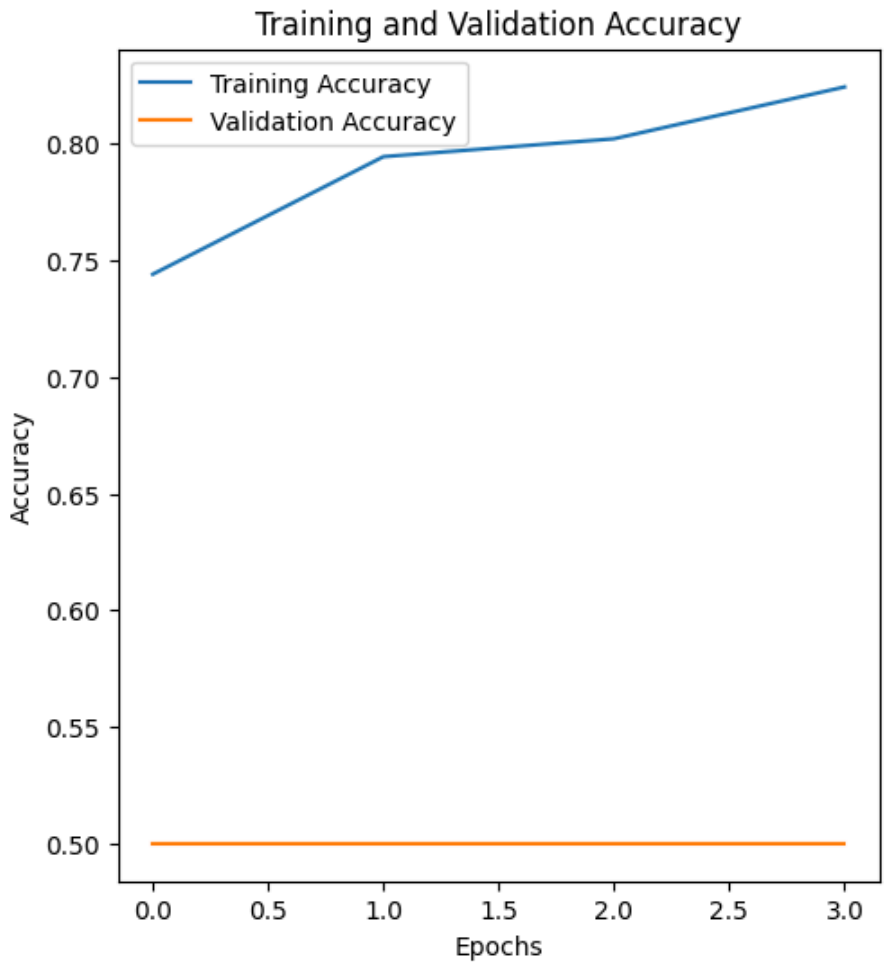
Non-trainable params: 0 (0.00 B)

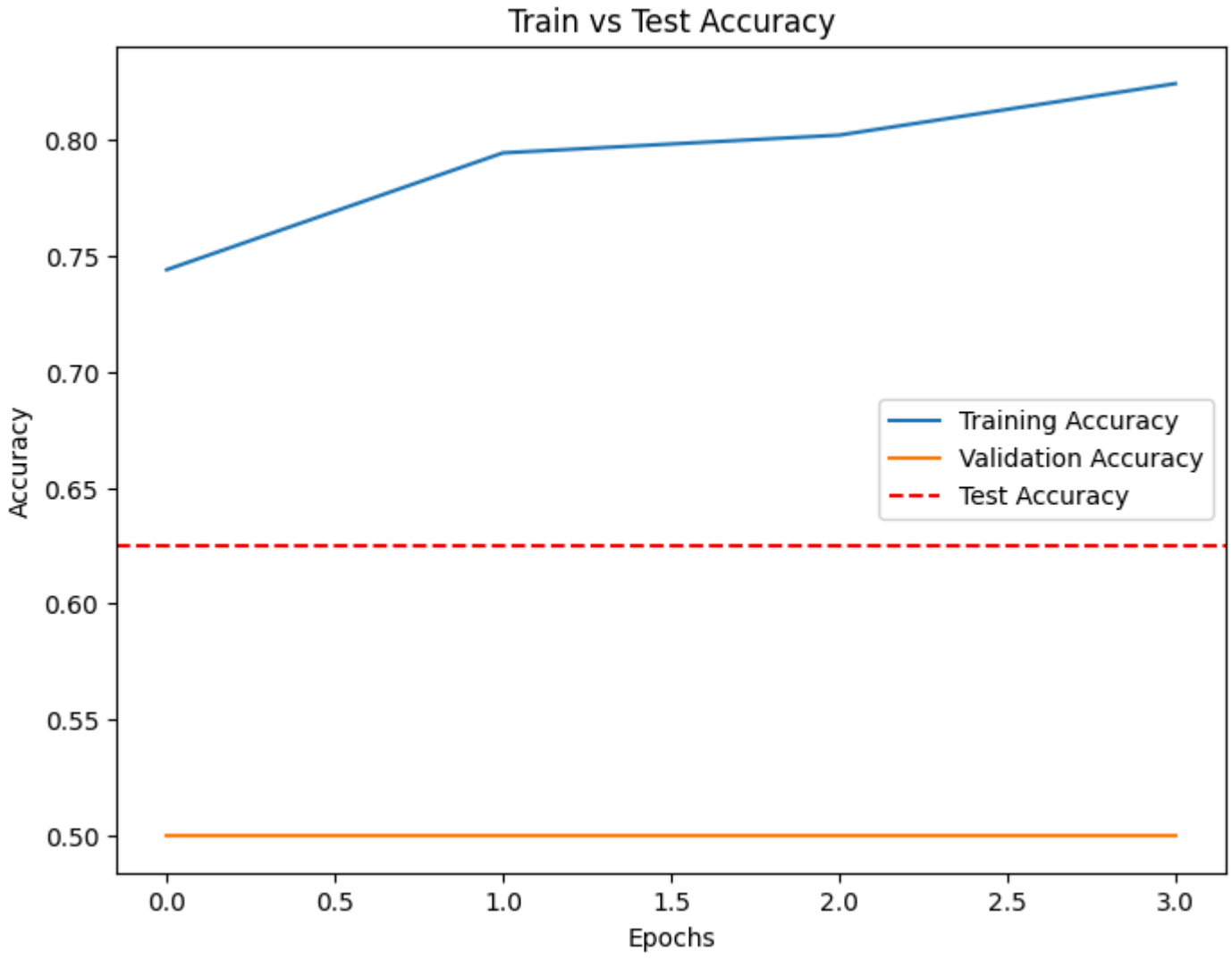
- This model's architecture comprises a total of 22,277,697 parameters, with all parameters being trainable. The model's size is approximately 84.98 MB. There are no non-trainable parameters in this architecture, indicating that all parameters are updated during the training process.

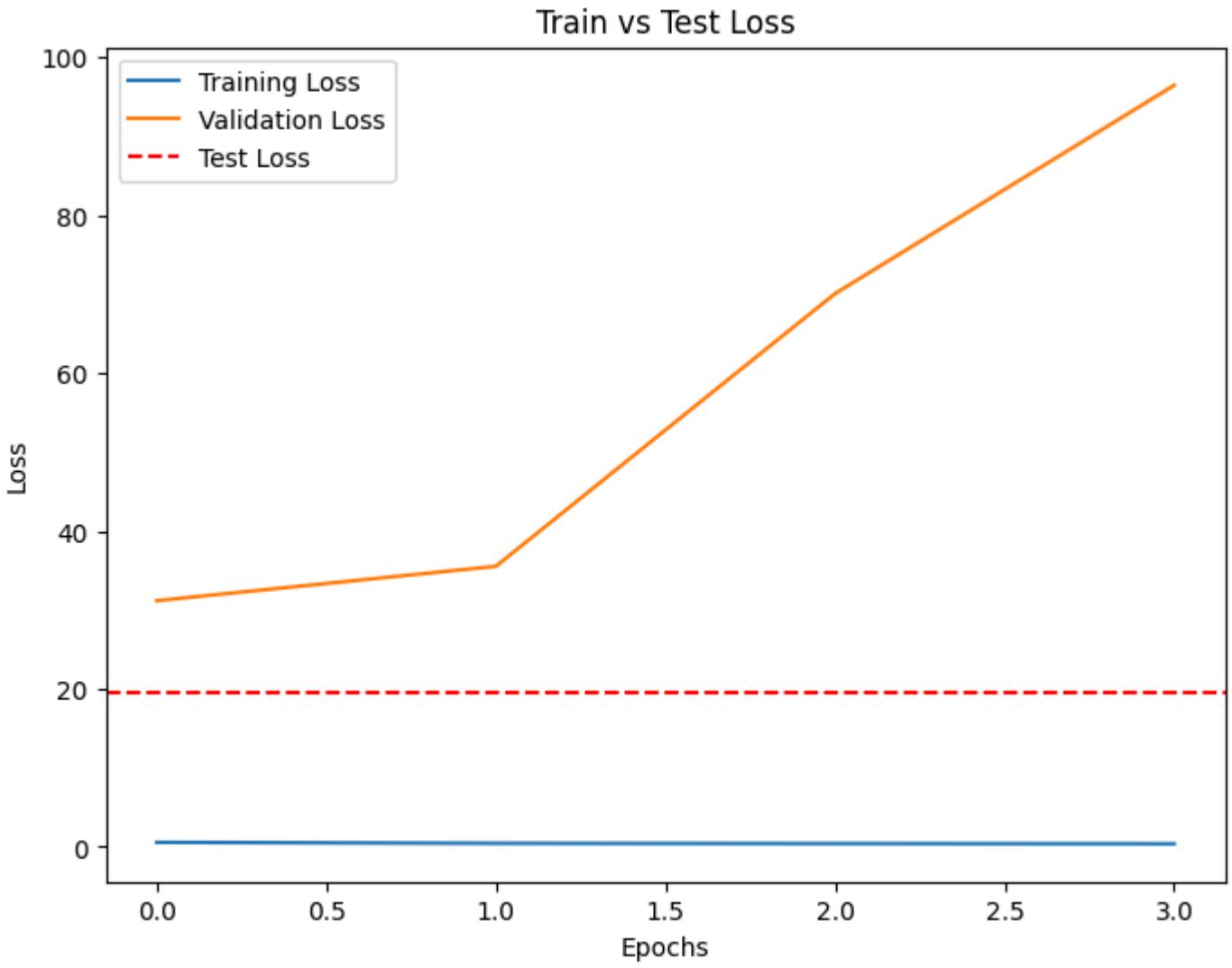
In [98]:
```
train_and_evaluate_model_es(new_model, train, validation, test)
```

```
Epoch 1/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 108s 599ms/step - accuracy: 0.7159 - loss: 0.7201 - val_accuracy: 0.5000 - val_loss: 31.
1775
Epoch 2/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 97s 567ms/step - accuracy: 0.7972 - loss: 0.4512 - val_accuracy: 0.5000 - val_loss: 35.5
361
Epoch 3/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 97s 569ms/step - accuracy: 0.8083 - loss: 0.4043 - val_accuracy: 0.5000 - val_loss: 70.0
781
Epoch 4/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 97s 565ms/step - accuracy: 0.8100 - loss: 0.3988 - val_accuracy: 0.5000 - val_loss: 96.4
741
Validation Loss: 30.517587661743164
Validation Accuracy: 0.5
Test Loss: 19.448657989501953
Test Accuracy: 0.625
```









Out[98]: `<keras.src.callbacks.history.History at 0x7f2de0b80760>`

- By employing a more complex architecture, significant improvements were observed in the test accuracy, reaching 62%, alongside notable reductions in overfitting.

(4) ResNet50V2 Model

In this section, we explore the ResNet50V2 model, a powerful convolutional neural network architecture renowned for its depth and performance in various computer vision tasks. ResNet50V2 builds upon the original ResNet architecture, incorporating skip connections and residual blocks to mitigate the vanishing gradient problem and enable training of significantly deeper networks. By leveraging pre-trained weights on large datasets like ImageNet, ResNet50V2 offers a robust feature extraction framework that can be fine-tuned for specific image classification tasks with relatively small datasets. We investigate the architecture, training process, and performance of ResNet50V2 on our dataset, aiming to harness its capabilities for accurate and efficient pneumonia classification. Documentation here (https://keras.io/api/applications/resnet/#resnet50v2-function).

In [16]:
```
from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D

# function for importing the ResNET50v2 model, defining architecture and compilation

def resnet_model(input_shape):
    # Load pre-trained ResNet50V2 model
    resnet_model = ResNet50V2(weights='imagenet', include_top=False, input_shape=input_shape)

    # Freeze all layers of the pre-trained model
    for layer in resnet_model.layers:
        layer.trainable = False

    # Create a Sequential model
    model = Sequential(name='ResNet50V2')

    # Add the pre-trained ResNet50V2 model to the Sequential model
    model.add(resnet_model)

    # Add global average pooling layer to reduce parameters
    model.add(GlobalAveragePooling2D())

    # Add a fully connected layer with fewer neurons
    model.add(Dense(64, activation='relu'))

    # Add dropout layer
    model.add(Dropout(0.5))

    # Add output layer
    model.add(Dense(1, activation='sigmoid'))

    return model

# Define input shape based on your image dimensions (e.g., (height, width, channels))
input_shape = (224, 224, 3)

# Create the ResNet50V2 model
resnet_model = resnet_model(input_shape)

# Compile the model
resnet_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

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

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50v2_weights_tf_dim_ordering_tf_kernels_notop.h5 (https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50v2_weights_tf_dim_ordering_tf_kernels_notop.h5)
94668760/94668760 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

Model: "ResNet50V2"
```

Layer (type)	Output Shape	Param #
resnet50v2 (Functional)	?	23,564,800
global_average_pooling2d (GlobalAveragePooling2D)	?	0 (unbuilt)
dense_2 (Dense)	?	0 (unbuilt)
dropout_1 (Dropout)	?	0
dense_3 (Dense)	?	0 (unbuilt)

```
Total params: 23,564,800 (89.89 MB)

Trainable params: 0 (0.00 B)

Non-trainable params: 23,564,800 (89.89 MB)
```

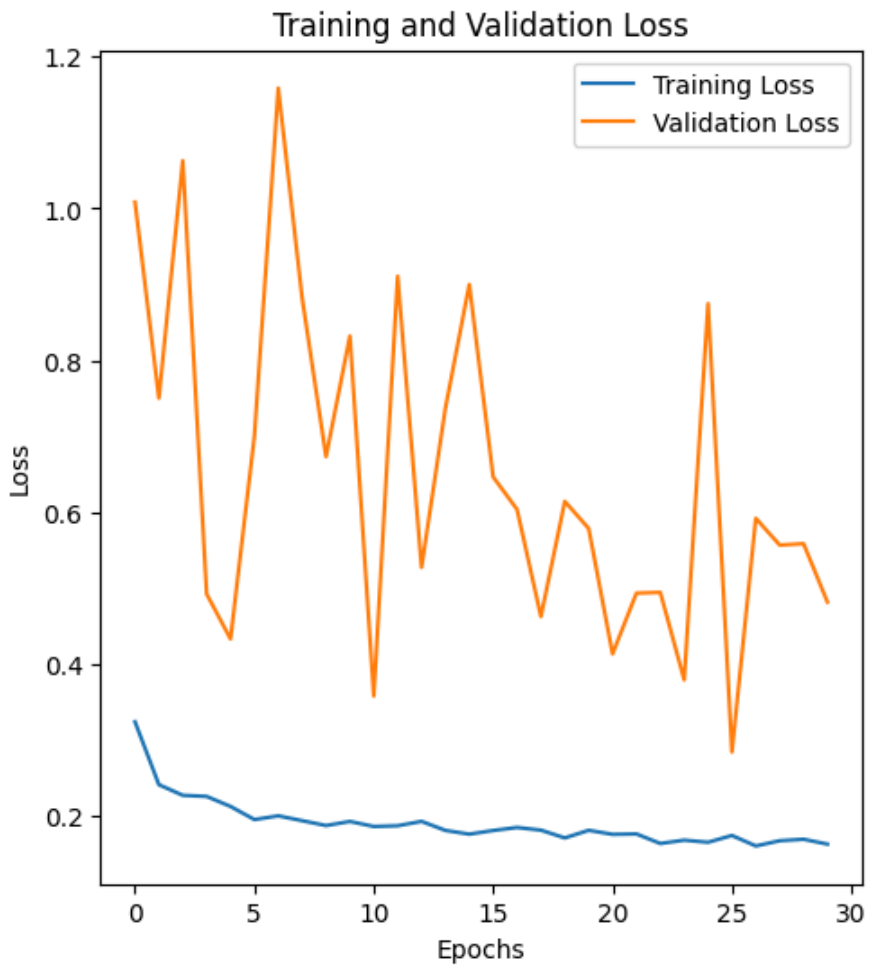
- The ResNET50v2 model comprises a total of 23,564,800 parameters, with all parameters being trainable. The model's size is approximately 89.89 MB. There are no non-trainable parameters in this architecture, indicating that all parameters are updated during the training process.

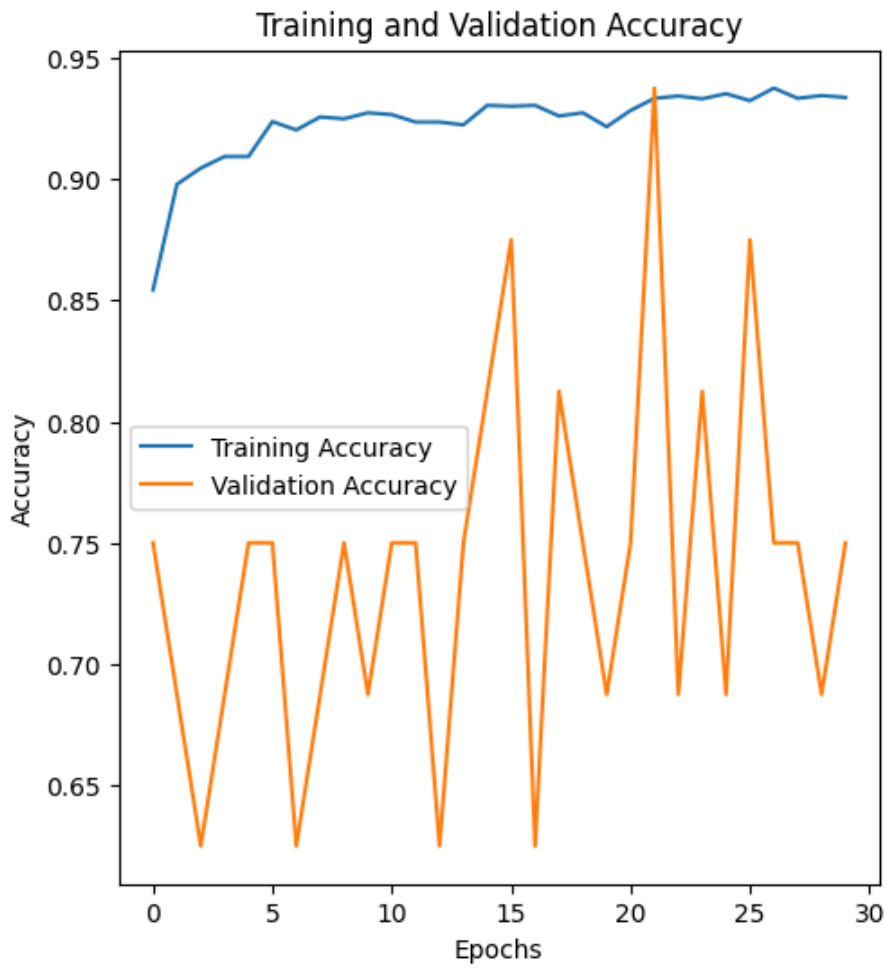
In [17]:
```
# timing model training

start = datetime.datetime.now()
```

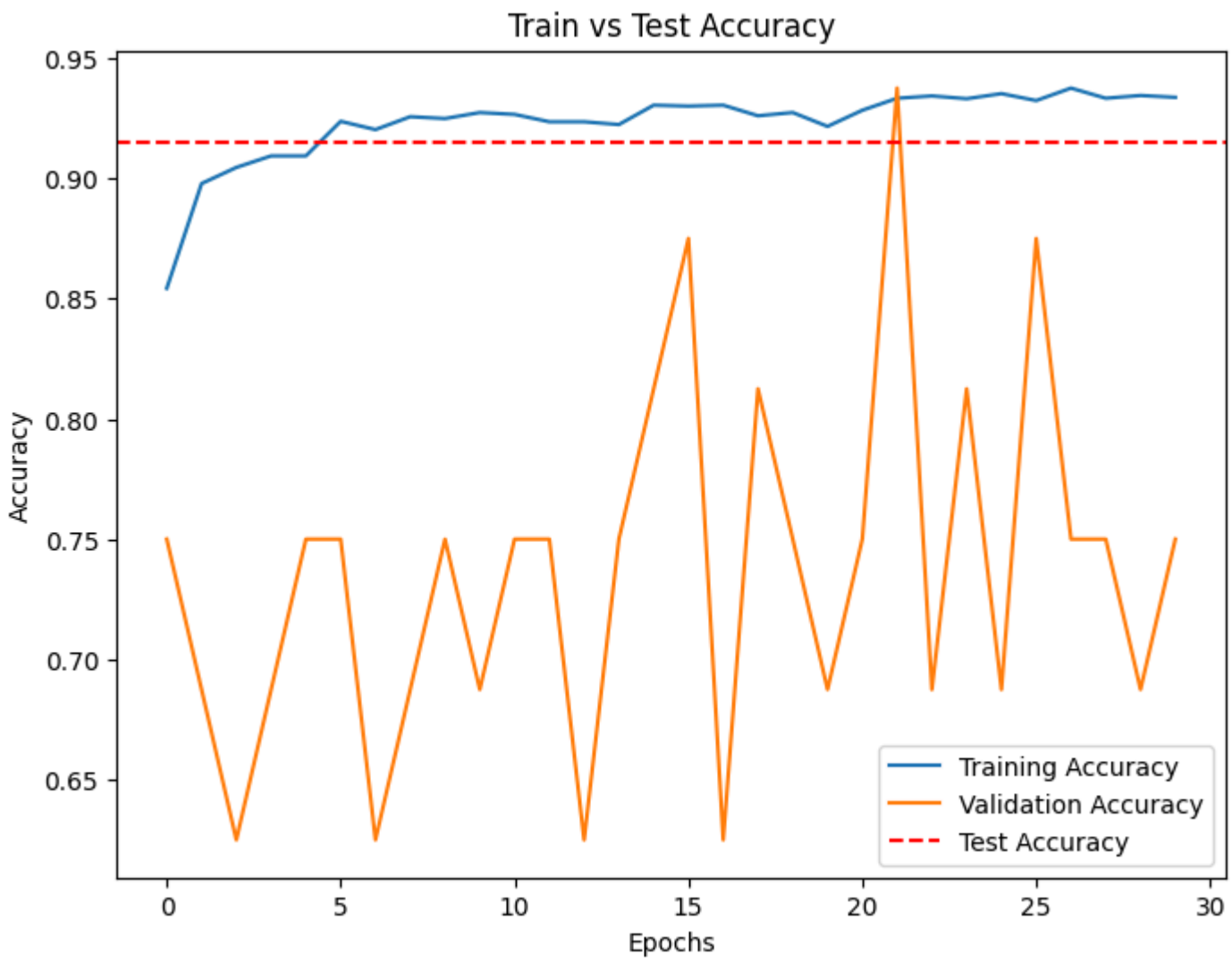
In [18]:
```
train_and_evaluate_model(resnet_model, train, validation, test, epochs=30)
```

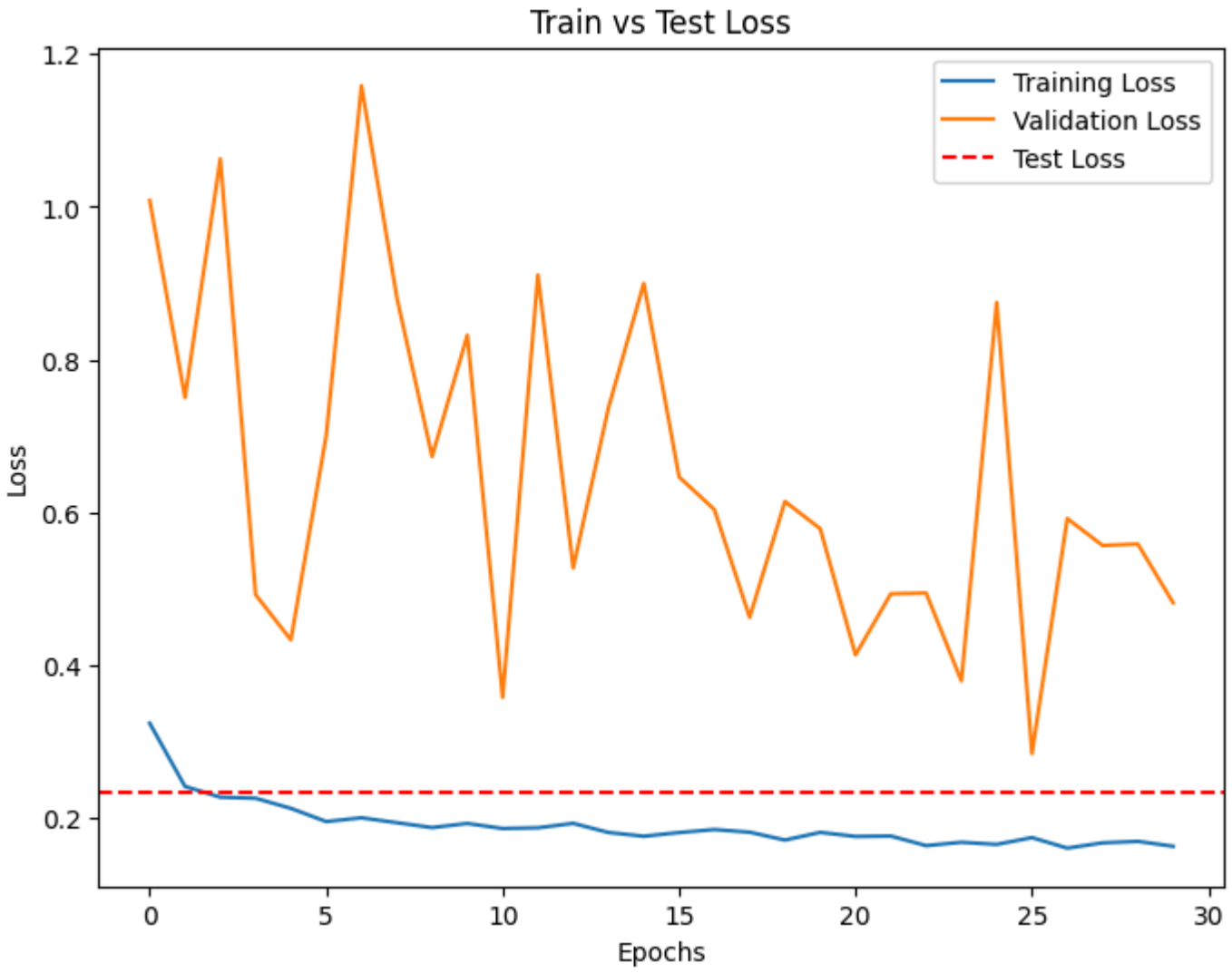
```
Epoch 1/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 115s 585ms/step - accuracy: 0.8093 - loss: 0.4031 - val_accuracy: 0.7500 - val_loss: 1.0
086
Epoch 2/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 557ms/step - accuracy: 0.8913 - loss: 0.2530 - val_accuracy: 0.6875 - val_loss: 0.75
04
Epoch 3/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 559ms/step - accuracy: 0.9029 - loss: 0.2329 - val_accuracy: 0.6250 - val_loss: 1.06
30
Epoch 4/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 562ms/step - accuracy: 0.9116 - loss: 0.2166 - val_accuracy: 0.6875 - val_loss: 0.49
25
Epoch 5/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 562ms/step - accuracy: 0.9070 - loss: 0.2257 - val_accuracy: 0.7500 - val_loss: 0.43
33
Epoch 6/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.9228 - loss: 0.1981 - val_accuracy: 0.7500 - val_loss: 0.70
05
Epoch 7/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 565ms/step - accuracy: 0.9291 - loss: 0.1839 - val_accuracy: 0.6250 - val_loss: 1.15
86
Epoch 8/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 98s 577ms/step - accuracy: 0.9247 - loss: 0.1950 - val_accuracy: 0.6875 - val_loss: 0.88
20
Epoch 9/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 564ms/step - accuracy: 0.9250 - loss: 0.1882 - val_accuracy: 0.7500 - val_loss: 0.67
32
Epoch 10/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.9261 - loss: 0.1954 - val_accuracy: 0.6875 - val_loss: 0.83
21
Epoch 11/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.9258 - loss: 0.1860 - val_accuracy: 0.7500 - val_loss: 0.35
81
Epoch 12/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 559ms/step - accuracy: 0.9199 - loss: 0.1888 - val_accuracy: 0.7500 - val_loss: 0.91
08
Epoch 13/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 564ms/step - accuracy: 0.9328 - loss: 0.1773 - val_accuracy: 0.6250 - val_loss: 0.52
75
Epoch 14/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 560ms/step - accuracy: 0.9216 - loss: 0.1825 - val_accuracy: 0.7500 - val_loss: 0.73
73
Epoch 15/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 559ms/step - accuracy: 0.9291 - loss: 0.1748 - val_accuracy: 0.8125 - val_loss: 0.90
00
Epoch 16/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 561ms/step - accuracy: 0.9325 - loss: 0.1672 - val_accuracy: 0.8750 - val_loss: 0.64
66
Epoch 17/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 563ms/step - accuracy: 0.9275 - loss: 0.1920 - val_accuracy: 0.6250 - val_loss: 0.60
38
Epoch 18/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 564ms/step - accuracy: 0.9242 - loss: 0.1836 - val_accuracy: 0.8125 - val_loss: 0.46
27
Epoch 19/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 561ms/step - accuracy: 0.9304 - loss: 0.1636 - val_accuracy: 0.7500 - val_loss: 0.61
44
Epoch 20/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 96s 563ms/step - accuracy: 0.9176 - loss: 0.1915 - val_accuracy: 0.6875 - val_loss: 0.57
88
Epoch 21/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 556ms/step - accuracy: 0.9234 - loss: 0.1855 - val_accuracy: 0.7500 - val_loss: 0.41
36
Epoch 22/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 557ms/step - accuracy: 0.9358 - loss: 0.1654 - val_accuracy: 0.9375 - val_loss: 0.49
36
Epoch 23/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 554ms/step - accuracy: 0.9362 - loss: 0.1620 - val_accuracy: 0.6875 - val_loss: 0.49
46
Epoch 24/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 94s 555ms/step - accuracy: 0.9330 - loss: 0.1593 - val_accuracy: 0.8125 - val_loss: 0.37
95
Epoch 25/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 557ms/step - accuracy: 0.9296 - loss: 0.1757 - val_accuracy: 0.6875 - val_loss: 0.87
48
Epoch 26/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 560ms/step - accuracy: 0.9331 - loss: 0.1667 - val_accuracy: 0.8750 - val_loss: 0.28
44
Epoch 27/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 558ms/step - accuracy: 0.9399 - loss: 0.1598 - val_accuracy: 0.7500 - val_loss: 0.59
21
Epoch 28/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 94s 553ms/step - accuracy: 0.9327 - loss: 0.1699 - val_accuracy: 0.7500 - val_loss: 0.55
69
Epoch 29/30
163/163 ━━━━━━━━━━━━━━━━━━━━ 95s 557ms/step - accuracy: 0.9356 - loss: 0.1714 - val_accuracy: 0.6875 - val_loss: 0.55
87
Epoch 30/30
```

```
163/163 ━━━━━━━━━━━━━━━━━━━━ 94s 551ms/step - accuracy: 0.9260 - loss: 0.1815 - val_accuracy: 0.7500 - val_loss: 0.48
18
Validation Loss: 0.2682477533817291
Validation Accuracy: 0.875
Test Loss: 0.23323379456996918
Test Accuracy: 0.9150640964508057
```









Out[18]: <keras.src.callbacks.history.History at 0x7dc558604cd0>

- The model demonstrated exceptional accuracy at 92%, accompanied by minimal signs of overfitting, as illustrated in the above plots.

In [19]:
```
end = datetime.datetime.now()
```

In [20]:
```
elapsed = end - start
print('Time elapsed', elapsed)
```

```
Time elapsed 0:48:08.395083
```

The ResNET50V2 model took around 48 mins to complete training, which is slightly less than the baseline CNN model.

Model Comparison

Sn. No.	Model Name	Model Architecture	Hyperparameters	Training Time	Validation Accuracy	Test Accuracy	Test Loss
1	Baseline Model	CNN with original architecture	Adam optimizer, learning rate=0.001	48.40 mins	0.50	0.38	172.32
2	Tuned Baseline Model	CNN with original architecture	Adam optimizer, learning rate=0.001, early stopping	26.45 mins	0.56	0.42	40.07
3	Different Architecture Model	CNN with modified architecture	Adam optimizer, learning rate=0.001, early stopping	10.25 mins	0.45	0.63	19.45
4	ResNet50v2 Model	Pretrained ResNet50v2	Adam optimizer, learning rate=0.0001	48 mins	0.88	0.92	0.23

We can derive the following insights from the model comparison table:

- The Baseline Model achieved the lowest validation and test accuracies, indicating that the original CNN architecture might not be sufficiently complex for the task at hand.
- The Tuned Baseline Model showed improvement in validation and test accuracies compared to the Baseline Model. The addition of early stopping helped prevent overfitting, resulting in better generalization performance.
- The Different Architecture Model, despite having a shorter training time, achieved the highest test accuracy among the CNN models. This suggests that modifying the architecture led to a more effective model, even with fewer training epochs.
- The ResNet50v2 Model, which utilized a pretrained ResNet50v2 architecture, demonstrated the highest validation and test accuracies. Pretrained models often outperform handcrafted architectures when trained on large datasets like ImageNet, as they leverage learned features.

Conclusion

Based on the results of our model performance comparison, several key insights emerge:

1. **Baseline Model vs. Tuned Baseline Model:** The incorporation of hyperparameter tuning and early stopping significantly improved the performance of the baseline CNN model. This is particularly crucial in the context of deep learning for pneumonia detection, where accurate and reliable diagnosis is paramount for timely treatment and patient outcomes.
2. **Different Architecture Model:** Exploring alternative architectures led to mixed results. While the different architecture model showed a notable improvement in test accuracy compared to the baseline model, it also exhibited higher training times.
3. **ResNet50v2 Model:** Leveraging a pretrained ResNet50v2 model yielded the highest validation and test accuracies among all models evaluated. This underscores the potential of pretrained models in accelerating the development of robust and accurate diagnostic tools for pneumonia detection, demonstrating the value of leveraging transfer learning techniques in medical imaging applications.
4. **Training Times:** The ResNet50v2 model required the longest training time, which could be a consideration in scenarios where computational resources are limited or when faster model iteration is desired. However, the superior accuracy of the ResNet50v2 model highlights its potential as a powerful tool in the arsenal of deep learning methods for pneumonia detection, particularly in settings where high performance is prioritized over training time.

In conclusion, the findings of this study contribute to the ongoing efforts in leveraging deep learning for pneumonia detection, showcasing the efficacy of various techniques such as hyperparameter tuning, architectural exploration, and transfer learning. Moving forward, continued research

References

- Kermany, Daniel; Zhang, Kang; Goldbaum, Michael (2018), “Large Dataset of Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images”, Mendeley Data, V3, doi: 10.17632/rscbjbr9sj.3
- Salehi M, Mohammadi R, Ghaffari H, Sadighi N, Reiazi R. Automated detection of pneumonia cases using deep transfer learning with paediatric chest X-ray images. Br J Radiol. 2021 May 1;94(1121):20201263. doi: 10.1259/bjr.20201263. Epub 2021 Apr 16. PMID: 33861150; PMCID: PMC8506182.