HO CHI MINH UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY OF INTERNATIONAL EDUCATION

DEEP LEARNING

DETECT PNEUMONIA APPLICATION

Lecturer name: PhD. Vu Quang Huy

List of members

Student ID	Student name	Contribution
19110152	Huỳnh Gia Kiện	100%
21110071	Đặng Hữu Phúc	100%

Ho Chi Minh City, 05/2024

SOCIALIST REPUBLIC OF VIETNAM

Independence – Freedom – Happiness

****

INSTRUCTOR'S COMMENTARY

Course name: **DEEP LEARNING**

Project's name: **DETECT PNEUMONIA APPLICATION**

Reviewer's full name: **PhD. Vu Quang Huy**

COMMENTARY:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Ho Chi Minh, May 2024

Supervisor

(Sign, write full name)

Vu Quang Huy

ACKNOWLEDGEMENTS

We would like to take this opportunity to express our sincerest gratitude to our instructor, Vu Quang Huy, for his invaluable support and guidance throughout this course. His unwavering dedication to helping us succeed in our final project has been nothing short of inspiring, and we are incredibly grateful for the opportunity to learn from such a knowledgeable and experienced developer.

Furthermore, we would like to extend our heartfelt thanks to our classmates, whose contributions have been instrumental in helping us to develop a strong and well-informed thesis. Their insights and feedback have been invaluable, and we are grateful for the knowledge and expertise they have shared with us.

Despite the challenges we faced, we managed to complete the subject and report in a short amount of time, with limited resources and varying levels of expertise in programming implementation. As a result, we acknowledge that there may be flaws in our work, and we welcome any constructive criticism or suggestions for improvement.

In conclusion, we would like to express our appreciation to everyone who has supported us in this journey. This final project would not have been possible without the collective effort of our team, instructor, classmates, and mentors, and we are truly grateful for the opportunity to learn and grow in this field. Thank you all for your support and encouragement. We welcome any feedback as well as suggestions for improving our project. We appreciate it. Sincerely!

Table of Contents

Table of Figures

Table of table

CHAPTER 1: OVERVIEW

1.1. Introduction

In the rapidly evolving field of healthcare, early and accurate diagnosis of diseases is crucial for effective treatment and improved patient outcomes. Pneumonia, a severe respiratory infection that inflames the air sacs in one or both lungs, remains a significant health threat worldwide, particularly affecting children, the elderly, and immunocompromised individuals. Despite advances in medical imaging technologies, the manual diagnosis of pneumonia from chest X-rays is time-consuming and prone to human error. To address these challenges, our deep learning project aims to develop and compare three distinct neural network models—Basic Convolutional Neural Network (CNN), CNN-Long Short-Term Memory (CNN-LSTM), and CNN-Recurrent Neural Network (CNN-RNN) to automate and enhance the detection of pneumonia from chest X-ray images.

1.2. Reason to choose this topic

Automated detection systems using deep learning have the potential to transform pneumonia diagnosis by delivering fast and accurate results. Leveraging the power of neural networks to extract and learn intricate patterns within medical imaging data, these systems can significantly reduce diagnostic errors and improve patient outcomes. They are particularly beneficial in remote areas with limited access to medical professionals, ensuring timely and reliable detection. By implementing and comparing various models, we aim to identify the most effective architecture for accurate pneumonia detection, potentially leading to faster diagnostics and better patient care, making quality healthcare more accessible and efficient across diverse settings.

1.3. The contents expected to be done

Get the dataset, build and train 3 models: CNN, CNN-LSTM, CNN-RNN, compare 3 models, build the application.

CHAPTER 2: THEORETICAL BASIS

2.1. Convolutional Neural Networks (CNN)

– **Algorithm and Definitions**

+ Convolutional Layer: Applies convolution operations to the input to extract features. A convolution operation involves sliding a filter (or kernel) over the input to produce a feature map.
+ Pooling Layer: Reduces the dimensionality of feature maps while retaining important features, typically using max pooling or average pooling.
+ Fully Connected Layer: Combines all the features extracted by previous layers to make final predictions.

– **Advantages**

+ Spatial Hierarchies: Efficiently captures spatial hierarchies in data, making them ideal for image data.
+ Parameter Sharing: Reduces the number of parameters significantly compared to fully connected networks, which helps in faster training and reducing overfitting.
+ Translation Invariance: Can recognize objects in images regardless of their position.

– **Disadvantages**

+ Data Requirements: Requires large amounts of labeled data for effective training.
+ Computationally Intensive: Requires significant computational resources, especially for training deep networks.
+ Lack of Temporal Context: Not suitable for sequence data where temporal context is crucial.

– **Practical Applications**

+ Image Classification: Identifying objects within images.
+ Object Detection: Detecting and locating objects in images (e.g., YOLO, R-CNN).
+ Image Segmentation: Classifying each pixel in an image (e.g., U-Net).

2.2. Convolutional Neural Network - Long Short-Term Memory (CNN-LSTM)

– **Algorithm and Definitions**

+ CNN Component: Extracts spatial features from the input data, typically image frames in a sequence.
+ LSTM Component: Processes the sequence of features extracted by the CNN to capture temporal dependencies.

– **Advantages**

+ Spatial and Temporal Modeling: Combines the strength of CNNs in spatial feature extraction with the temporal modeling capabilities of LSTMs.
+ Sequential Data: Ideal for tasks involving sequential data such as video or time series analysis.
+ Handling Variable Length Inputs: LSTM can handle input sequences of varying lengths.

– **Disadvantages**

+ Complexity: More complex than pure CNNs or LSTMs, leading to increased computational requirements.
+ Training Difficulty: Can be harder to train due to the combined architecture, requiring careful tuning of hyperparameters.

– **Practical Applications**

+ Video Analysis: Action recognition, video captioning, and video classification.
+ Time Series Forecasting: Predicting future values in a time series based on past observations.
+ Gesture Recognition: Recognizing hand gestures in a sequence of images.

2.3. Convolutional Neural Network - Recurrent Neural Network (CNN-RNN)

– **Algorithm and Definitions**

+ CNN Component: Extracts spatial features from input data, typically images.
+ RNN Component: Processes sequences of features (from the CNN) to capture temporal dependencies.

– **Advantages**

+ Sequential Dependencies: Can model sequences, making it suitable for video and sequential image data.
+ Combined Strengths: Utilizes the spatial feature extraction power of CNNs and the temporal sequence handling of RNNs.

– **Disadvantages**

+ Training Complexity: More complex to train due to the dual architecture, often requiring more sophisticated training techniques and longer training times.
+ Resource Intensive: Requires significant computational resources for both training and inference.

– **Practical Applications**

+ Natural Language Processing: Video subtitle generation, where sequences of frames need to be interpreted and described.
+ Medical Imaging: Analyzing sequences of medical images over time, such as MRI scans.
+ Speech Recognition: Processing spectrograms (image representations of audio signals) to understand spoken language.

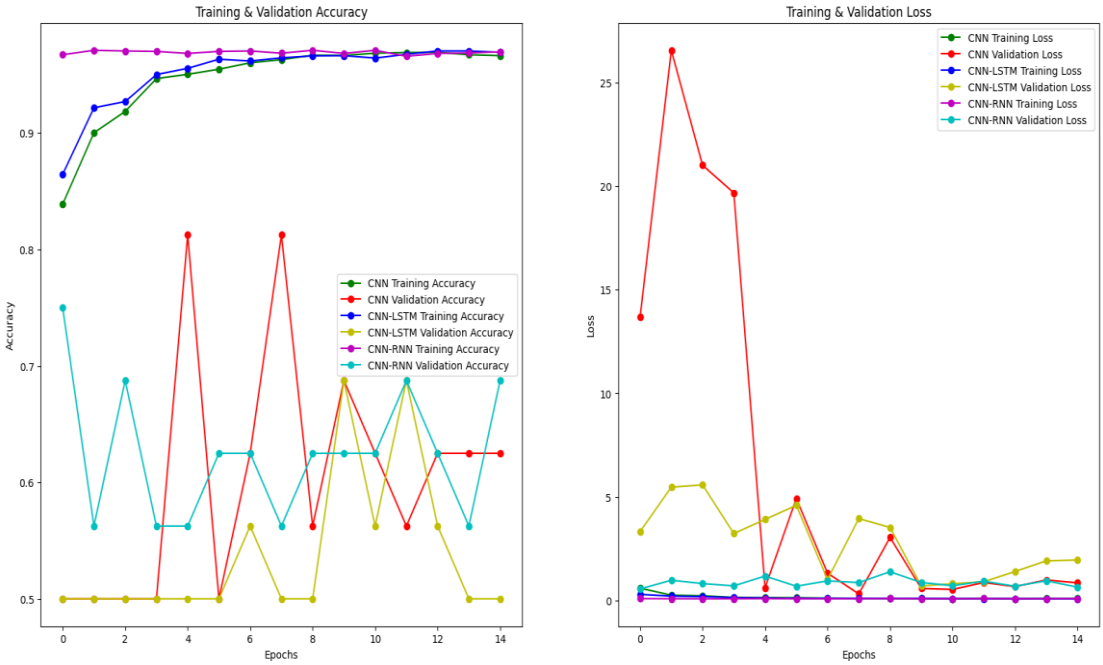

Figure 1: Training & Validation Folders Accuracy and Loss

CHAPTER 3: TRAINING DATASET

3.1. Dataset Description

– **Data Type:** The Pneumonia Dataset is a collection of chest X-ray images specifically designed for training and evaluating machine learning algorithms for pneumonia detection. It is organized into three main folders: train, test, and validation. Each of these folders contains subfolders for two image categories: Pneumonia and Normal. This clear structure facilitates the training process for AI models.

– **Size and format:** The dataset comprises a total of 5,873 chest X-ray images stored in JPEG format and 2 categories (Pneumonia/Normal)

– **Classes:**

+ Pneumonia: 4280 images

+ Normal: 1583 images

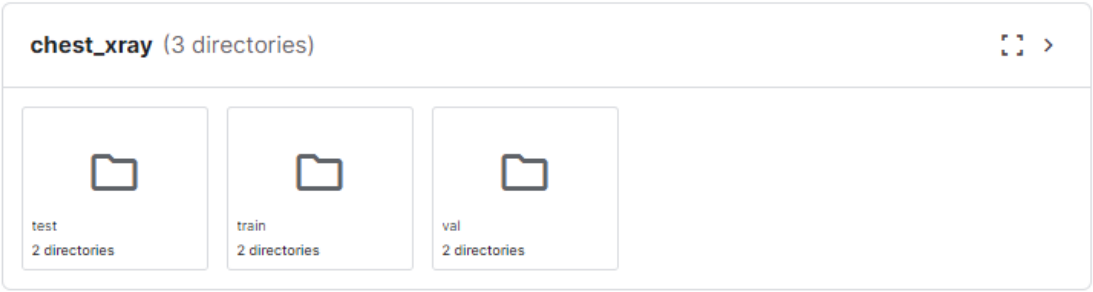

Figure 2: Dataset Folders

– **Data organization:** split into 3 folders (train, test, val) and contains sub-folders for each image category (Pneumonia/Normal).

+ Train data has 5223 images (3882 pneumonia images, 1341 pneumonia images)

+ Val data has 16 images (8 pneumonia images, 8 pneumonia images)

+ Test data has 624 images (390 pneumonia images, 234 pneumonia images)

– **Characteristics:** These images were carefully selected from retrospective cohorts of pediatric patients aged one to five years old. All the patients originated from Guangzhou Women and Children's Medical Center, Guangzhou, China. It's important to note that these X-rays were captured during routine clinical care procedures, ensuring their relevance to real-world scenarios.

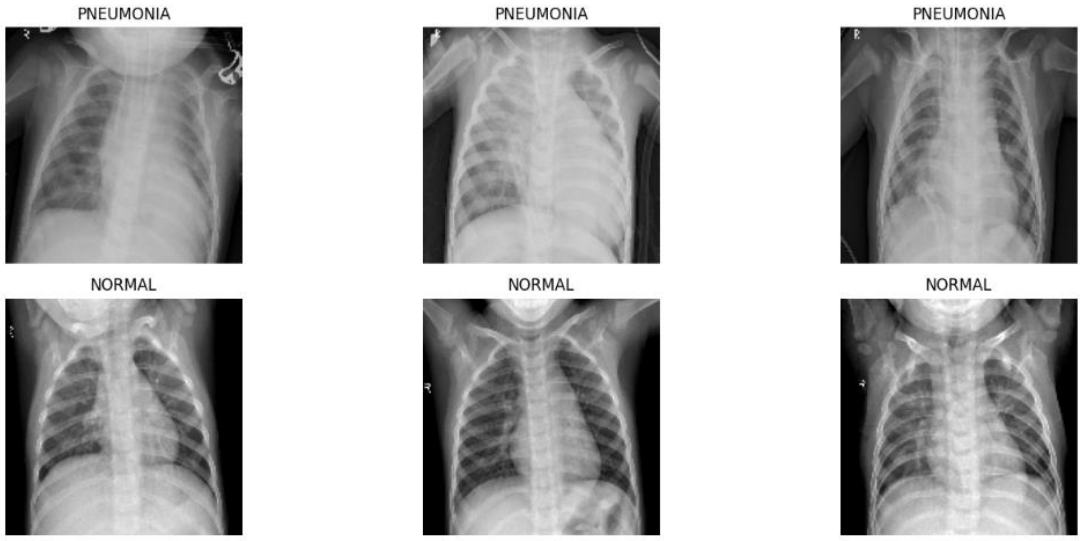

Figure 3: Example of Pneunomia and Normal

- To guarantee the quality of the dataset, a meticulous screening process was implemented. All chest radiographs underwent an initial quality control check, with any low-quality or unreadable scans being removed. This ensures that the AI model trains on clear and interpretable images.
- Furthermore, the diagnoses for each image were meticulously assessed. Two expert physicians independently graded the X-rays to confirm the presence or absence of pneumonia. This double-grading approach minimizes the risk of misdiagnosis within the dataset. Finally, to account for potential discrepancies, the evaluation set was reviewed by a third expert, adding an extra layer of confidence in the data's accuracy. This multi-step validation process ensures the reliability of the dataset for training robust AI models for pneumonia detection.
- **Source:**

https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia/data

- **Acknowledgements:**

+ Data: https://data.mendeley.com/datasets/rscbjbr9sj/2
+ Citation: http://www.cell.com/cell/fulltext/S0092-8674(18)30154-5

- **Input and output characteristics:**

+ Input Features: 2D X-ray image
+ Output Feature: Classification of the image as "Pneumonia" or "Normal"

3.2. Data Preprocessing for Image Classification: A Comprehensive Overview

3.2.1. Image Flattening:

```
# Flatten hình ảnh
x_train_flattened = x_train.reshape(x_train.shape[0], -1)
```

Figure 4: Image Flattening

- **3D to 2D Transformation:** The input images, typically represented as 3D arrays with dimensions (number of images, height, width), often require transformation into a simpler 2D format. This is achieved using the reshape function, which converts the image data into a 2D array with dimensions (number of images, total number of pixels).
- **Purpose:** Flattening the images serves a crucial purpose in preparing the data for machine learning models. By converting the 3D images into a 2D representation, the data becomes more manageable and compatible with a wider range of machine learning algorithms. This simplified format allows the model to effectively process and extract meaningful features from the image data, enhancing the overall classification performance.

3.2.2. SMOTE Oversampling:

```
# Khởi tạo mô hình SMOTE
smote = SMOTE(random_state=42)

# Áp dụng SMOTE cho dữ liệu huấn luyện và nhãn tương ứng
x_train_resampled, y_train_resampled = smote.fit_resample(x_train_flattened, y_train)
```

Figure 5: SMOTE Oversampling

- **Addressing Class Imbalance:** In many real-world datasets, particularly those involving classification tasks, the distribution of data points across different classes can be imbalanced. This means that one class (the minority class) may have significantly fewer instances compared to the other class (the majority class). Such class imbalance can pose challenges for machine learning models, as they tend to favor the majority class during training, leading to biased predictions.
- **SMOTE (Synthetic Minority Over-sampling Technique)** tackles this issue by generating synthetic data points for the minority class. It intelligently creates new data samples that are similar to existing minority class examples, effectively augmenting the dataset and balancing the class distribution.
- **Purpose:** Implementing SMOTE oversampling is particularly beneficial in image classification tasks with imbalanced datasets. By increasing the representation of the minority class, the model is exposed to a more diverse range of data, reducing the impact of class bias and improving the overall accuracy of the classification process.

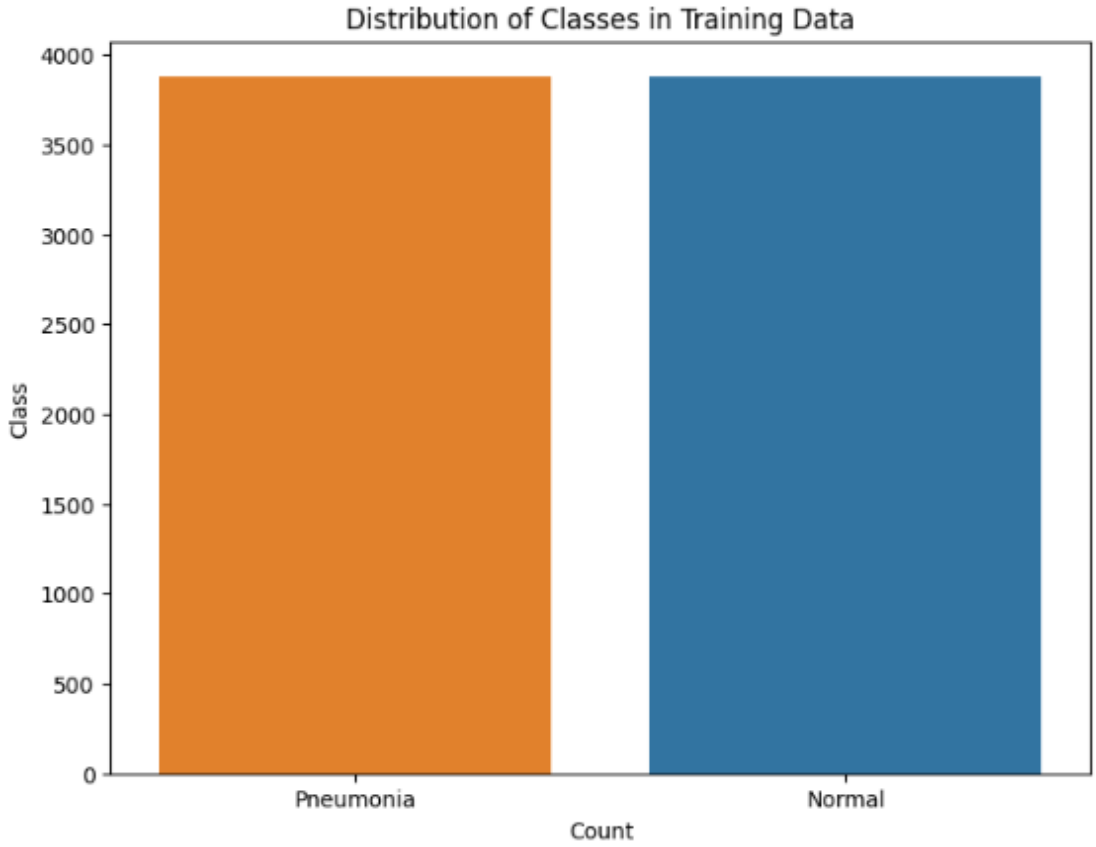

Figure 6: Distribution of Classes in Training Data after SMOTE

3.2.3 Data Normalization:

```
# Normalize the data
x_train_resampled = np.array(x_train_resampled) / 255
x_val = np.array(x_val) / 255
x_test = np.array(x_test) / 255
```

Figure 7: Data Normalization

- **Normalization Technique:** Image data often involves pixel values that vary across a wide range. To ensure that the model processes these values consistently and effectively, it's essential to normalize the data. This involves scaling the pixel values to a specific range, typically between 0 and 1. A common normalization technique for image data is to divide each pixel value by the maximum possible value, which in the case of grayscale images is 255. This transformation ensures that all pixel values fall within the desired range, preventing extreme values from dominating the data and potentially hindering the learning process.
- **Purpose:** Normalizing the image data serves several important purposes. Firstly, it standardizes the representation of pixel intensities, allowing the model to focus on the relative differences between pixel values rather than their absolute magnitudes. Secondly, normalization prevents numerical overflow or underflow

issues during computations, ensuring numerical stability during training and inference.

3.2.4. Data Reshaping for Model Compatibility:

```
# resize data for deep learning
x_train_resampled = x_train_resampled.reshape(-1, img_size, img_size, 1)
y_train_resampled = np.array(y_train_resampled)

x_val = x_val.reshape(-1, img_size, img_size, 1)
y_val = np.array(y_val)

x_test = x_test.reshape(-1, img_size, img_size, 1)
y_test = np.array(y_test)
```

Figure 8: Data Reshaping

- **Preparing Data for Model Input:** Once the images have been flattened and normalized, they need to be reshaped into a format that aligns with the input structure of the specific machine learning model being used. This typically involves converting the 2D array into a 4D array with dimensions (number of images, height, width, number of channels).
- **Color Channel Representation:** For color images, an additional dimension is introduced to represent the color channels. This dimension typically has a size of 3, corresponding to the three primary color channels: red, green, and blue. By including this dimension, the model is able to capture the full color information of the images, enabling it to perform accurate classification based on both color and grayscale features.
- **Label Conversion:** Additionally, the label data, which represents the class labels for each image, is typically converted into a NumPy array to ensure compatibility with the model's input format.
- **Purpose:** Reshaping the data into the appropriate format is crucial for ensuring that the model can correctly interpret and process the input images. By aligning the data structure with the model's expectations, the model can effectively extract relevant features and make accurate predictions.

3.2.5. Data Augmentation:

```
[ ]  # With data augmentation to prevent overfitting and handling the imbalance in dataset

     datagen = ImageDataGenerator(
             featurewise_center=False,  # set input mean to 0 over the dataset
             samplewise_center=False,  # set each sample mean to 0
             featurewise_std_normalization=False,  # divide inputs by std of the dataset
             samplewise_std_normalization=False,  # divide each input by its std
             zca_whitening=False,  # apply ZCA whitening
             rotation_range = 30,  # randomly rotate images in the range (degrees, 0 to 180)
             zoom_range = 0.2, # Randomly zoom image
             width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
             height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
             horizontal_flip = True,  # randomly flip images
             vertical_flip=False)  # randomly flip images


     datagen.fit(x_train_resampled)
```

Figure 9: Data Augmentation

- **In order to avoid overfitting problem**, we need to expand artificially our dataset. We can make your existing dataset even larger. The idea is to alter the training data with small transformations to reproduce the variations. Approaches that alter the training data in ways that change the array representation while keeping the label the same are known as data augmentation techniques. Some popular augmentations people use are grayscales, horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more. By applying just a couple of these transformations to our training data, we can easily double or triple the number of training examples and create a very robust model.
- **For the data augmentation:** Randomly rotate some training images by 30 degrees Randomly Zoom by 20% some training images Randomly shift images horizontally by 10% of the width Randomly shift images vertically by 10% of the height Randomly flip images horizontally. Once our model is ready, we fit the training dataset.

CHAPTER 4: ALGORITHM FLOW CHART FOR MODEL TRAINING

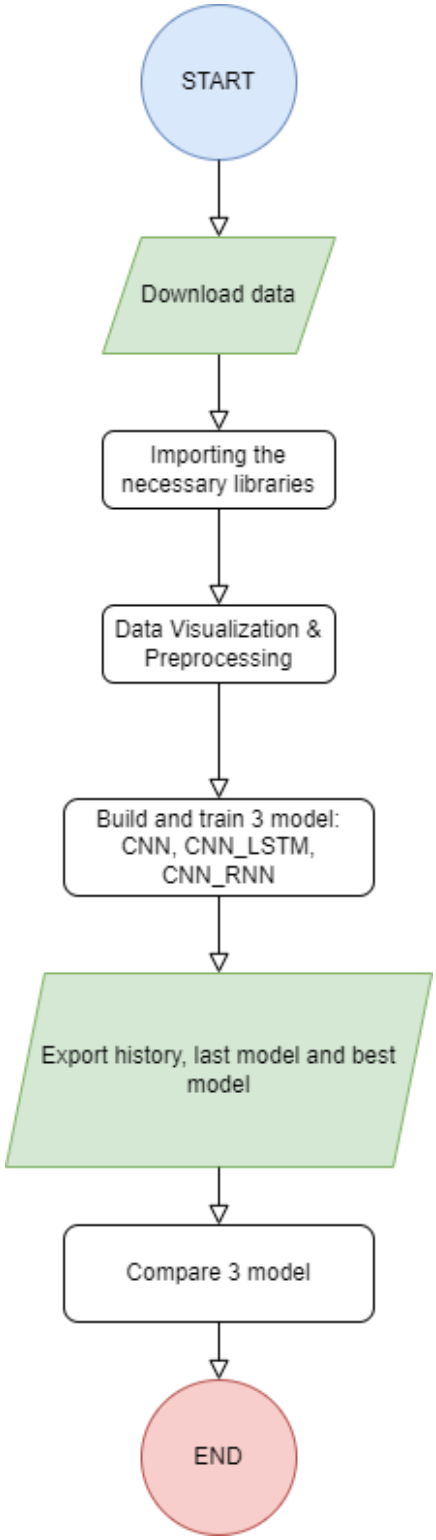

Figure 10: Algorithm Flow Chart for Model Training

4.1. Data Download:

- The flowchart then proceeds to the data download step. This step involves retrieving the relevant data for the machine learning problem. dowload and unzip data from kaggle

4.2. Importing the Necessary Libraries:

- The next step involves importing the necessary libraries for the machine learning task. This includes libraries for data manipulation, model training, and evaluation. The specific libraries imported will depend on the chosen machine learning framework and the specific requirements of the task.

4.3. Data Visualization and Preprocessing:

- Once the data is downloaded, it undergoes visualization and preprocessing. Data visualization involves exploring the data's characteristics, such as data distribution, outliers, and missing values. This helps in understanding the data and identifying any potential issues that may require preprocessing.
- Data preprocessing involves cleaning and preparing the data for model training. This may include handling missing values, normalizing or scaling data, and encoding categorical variables. The goal of preprocessing is to ensure the data is consistent, accurate, and suitable for the chosen machine learning model.
  (we use: Image Flattening, SMOTE Oversampling, Data Normalization, Data Reshaping for Model Compatibility and data augmentation.)

4.4. Building and Training 3 Models:

- The specific models being built and trained. They are different variants of Convolutional Neural Networks (CNNs), given the task of image classification. CNNs are commonly used for image recognition and classification tasks due to their ability to extract spatial features from images.

4.5. Exporting History, Last Model, and Best Model:

- The next step involves exporting the training history, the last trained model, and the best-performing model. The training history provides insights into the model's performance during training, such as loss and accuracy values over

epochs. The last trained model represents the model obtained at the end of the training process.

- The best-performing model, on the other hand, represents the model that achieved the highest performance on a validation dataset. This model is typically selected for deployment and real-world use.

4.6. Compare 3 Models:

- Finally, the flowchart converges to the compare 3 models node. This step involves evaluating the performance of the three trained models and selecting the best-performing one. The evaluation may involve metrics such as accuracy, precision, recall, and F1-score.

CHAPTER 5: TRAINING PROGRAM CODE

5.1. Download and unzip data:

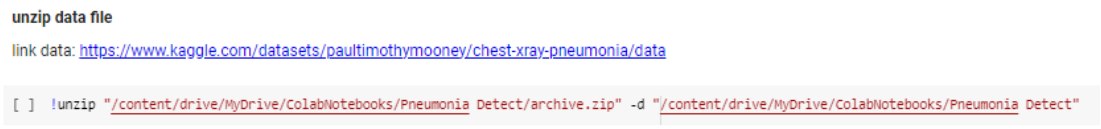

Figure 11: Unzip data

5.2. Import neccesary library:

Figure 12: Import libraries

- matplotlib.pyplot: Imports the plotting library matplotlib with the pyplot interface for creating visualizations.
- seaborn: Imports the seaborn library, which builds on top of matplotlib for creating more aesthetically pleasing and informative visualizations.
- keras: Imports the core Keras deep learning library.
- from keras.models import Sequential: Imports the Sequential model class from Keras, used for building models by stacking layers sequentially.
- from keras.layers import ...: Imports various layer classes from Keras, including:
    - LSTM, GRU, SimpleRNN: Recurrent neural network (RNN) layers for handling sequential data like time series or sequences of images.
    - Dense: Fully connected layer used for classification and regression tasks.
    - Conv2D, MaxPool2D, Flatten: Convolutional neural network (CNN) layers for processing image data:
        - Conv2D: Performs convolution operations to extract features from images.
        - MaxPool2D: Reduces dimensionality of the feature maps.
        - Flatten: Converts the 3D feature maps into a 1D vector for feeding into fully connected layers.
    - Dropout: Randomly drops neurons during training to prevent overfitting.

  - BatchNormalization: Normalizes activations of the previous layer for faster convergence and better stability.
  - Reshape: Reshapes the data into a desired format.
- from keras.preprocessing.image import ImageDataGenerator: Imports the ImageDataGenerator class used for data augmentation techniques to increase the size and diversity of training data.
- from sklearn.model_selection import train_test_split: Imports the train_test_split function from scikit-learn for splitting data into training and testing sets.
- from sklearn.metrics import classification_report, confusion_matrix: Imports functions from scikit-learn for evaluating classification model performance:

  - classification_report: Provides a text summary of the model's performance on different classes.
  - confusion_matrix: Creates a matrix that visualizes how many instances were predicted correctly or incorrectly for each class.
- keras.callbacks.*: Imports callback classes from Keras:

  - ReduceLROnPlateau: Reduces the learning rate if the model's validation loss doesn't improve for a certain number of epochs, preventing overfitting.
  - EarlyStopping: Stops training early if the validation loss doesn't improve for a certain number of epochs.
  - ModelCheckpoint: Saves the best model based on a specified metric during training.
- cv2: Imports the OpenCV library for image processing tasks.
- os: Imports the operating system module for interacting with the file system.
- numpy as np: Imports the NumPy library for numerical computations and array manipulations.
- pandas: Imports the pandas library for data analysis and manipulation in tabular format.
- pickle: Imports the pickle library for serializing and deserializing Python objects (useful for saving and loading models).

5.3. Get data:

```
labels = ['PNEUMONIA', 'NORMAL']
img_size = 150

def get_training_data(data_dir):
    images = []
    labels_list = []

    for label in labels:
        path = os.path.join(data_dir, label)
        class_num = labels.index(label)
        img_each_label_count = 0
        for img in os.listdir(path):
            try:
                img_arr = cv2.imread(os.path.join(path, img), cv2.IMREAD_GRAYSCALE)
                resized_arr = cv2.resize(img_arr, (img_size, img_size)) # Reshaping images to preferred size
                images.append(resized_arr)
                labels_list.append(class_num)
                img_each_label_count+=1
            except Exception as e:
                print(e)
        print("Number of", label, "images:", img_each_label_count)
    total_images = len(images)
    print("Total number of images:", total_images)
    print()
    return np.array(images), np.array(labels_list)
```

Figure 13: Get training data #1

```
[ ] x_train = []
    y_train = []

    x_val = []
    y_val = []

    x_test = []
    y_test = []

    x_train, y_train = get_training_data('/content/drive/MyDrive/ColabNotebooks/Pneumonia_Detect/chest_xray/train')
    x_test, y_test = get_training_data('/content/drive/MyDrive/ColabNotebooks/Pneumonia_Detect/chest_xray/test')
    x_val, y_val = get_training_data('/content/drive/MyDrive/ColabNotebooks/Pneumonia_Detect/chest_xray/val')
```

```
Number of PNEUMONIA images: 3882
Number of NORMAL images: 1341
Total number of images: 5223

Number of PNEUMONIA images: 390
Number of NORMAL images: 234
Total number of images: 624

Number of PNEUMONIA images: 8
Number of NORMAL images: 8
Total number of images: 16
```

Figure 14: Get training data #2

– This code snippet reads and preprocesses chest X-ray images from separate directories for training, validation, and testing, converting them to a format suitable for training a deep learning model for pneumonia detection.

5.3.1. Labels and Image Size:

- labels = ['PNEUMONIA', 'NORMAL']: Defines two class labels for the images: "PNEUMONIA" and "NORMAL".

- img_size = 150: Sets the desired image size for resizing (150x150 pixels in this case).

5.3.2. get_training_data Function:

- **Arguments:**
    - data_dir: The directory containing the chest X-ray images. The directory structure is assumed to have subfolders named after each class label ("PNEUMONIA" and "NORMAL").
- **Functionality:**
    - Initializes empty lists images and labels_list to store the processed image data and corresponding labels.
    - Iterates through each label in the labels list:
        - Constructs the path to the subfolder containing images for the current label using os.path.join.
        - Gets the index (position) of the current label in the labels list using labels.index(label). This will be used as the numerical class label.
        - Initializes a counter img_each_label_count to track the number of images processed for each class.
        - Iterates through each image filename (img) in the current label's subfolder using os.listdir.
            - Within the inner loop:
                - Tries to read the image using cv2.imread with cv2.IMREAD_GRAYSCALE to load it as grayscale.
                - If successful, resizes the image to the specified img_size using cv2.resize.
                - Appends the resized image array to the images list and the corresponding class label (index) to the labels_list.
                - Increments the img_each_label_count for the current class.
            - Catches any exceptions (Exception) that might occur during image reading and prints the error message.
        - After processing all images for the current label, prints the number of images found for that class.
    - Converts the images and labels_list lists to NumPy arrays using np.array for better performance in machine learning tasks.
    - Returns the NumPy arrays containing the preprocessed image data (images) and the corresponding class labels (labels_list).

5.3.3. Data Splitting:

- The code defines empty lists for training, validation, and testing data: x_train, y_train, x_val, y_val, x_test, and y_test.
- Calls the get_training_data function three times with different paths:
    - /content/drive/MyDrive/ColabNotebooks/Pneumonia_Detect/chest_xray/train: This path likely points to the training data directory. The processed images and labels from this directory are stored in x_train and y_train.
    - /content/drive/MyDrive/ColabNotebooks/Pneumonia_Detect/chest_xray/test: This path likely points to the testing data directory. The processed images and labels from this directory are stored in x_test and y_test.
    - /content/drive/MyDrive/ColabNotebooks/Pneumonia_Detect/chest_xray/val: This path likely points to the validation data directory. The processed images and labels from this directory are stored in x_val and y_val.

5.4. Visualize data:

```
[ ]
    # Convert labels to class names
    class_names = ['Pneumonia', 'Normal']
    class_labels = [class_names[label] for label in y_train]

    # Set the color palette for the plot
    colors = ['#ff7f0e', '#1f77b4']

    # Create count plot with rotated x-axis labels and customized colors
    plt.figure(figsize=(8, 6))
    sns.countplot(x=class_labels, palette=colors)
    plt.title('Distribution of Classes in Training Data')
    plt.xlabel('Count')
    plt.ylabel('Class')
    plt.show()
```

Figure 15: Visualize Data

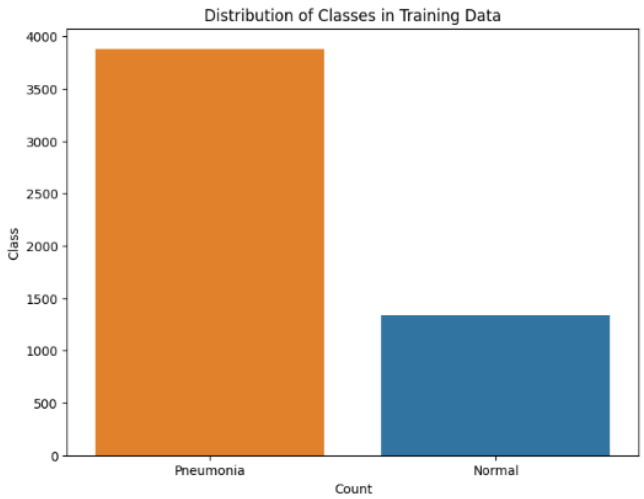

Figure 16: Distribution of Classes in Training Data

– This code generates a count plot that visualizes the number of training images belonging to each class ("Pneumonia" and "Normal"). This can be helpful in understanding the balance or imbalance of classes in training data, which is important for effective machine learning model performance.

5.4.1. Converting Labels to Class Names:

- class_names = ['Pneumonia', 'Normal']: Defines a list containing the human-readable class names for the two labels ("Pneumonia" and "Normal").
- class_labels = [class_names[label] for label in y_train]: Creates a new list class_labels. It iterates through each label in the y_train list (which likely contains numerical class labels) and uses those labels as indices to retrieve the corresponding class names from the class_names list. This effectively converts the numerical labels to their descriptive class names.

5.4.2. Setting Color Palette:

- colors = ['#ff7f0e', '#1f77b4']: Defines a list of colors to be used for the bars in the plot. The first color (#ff7f0e) is orange, and the second (#1f77b4) is blue. These colors can be customized to your preference.

5.4.3. Creating the Count Plot:

- plt.figure(figsize=(8, 6)): Creates a new figure for the plot with a specific size (8 inches width and 6 inches height). You can adjust these values if you want a larger or smaller plot.
- sns.countplot(x=class_labels, palette=colors): This line from the seaborn library creates a count plot. It takes two arguments:
    - x=class_labels: Specifies the data to be plotted on the x-axis. In this case, it uses the class_labels list containing the class names.
    - palette=colors: Defines the color palette to be used for the bars in the plot, referring to the colors list you defined earlier.
- plt.title('Distribution of Classes in Training Data'): Sets the title for the plot.
- plt.xlabel('Count'): Sets the label for the x-axis, indicating that it represents the count of images.
- plt.ylabel('Class'): Sets the label for the y-axis, indicating the class names.
- plt.show(): Displays the created plot.

5.5. Balance and visualization:

```python
from imblearn.over_sampling import SMOTE

# Flatten hình ảnh
x_train_flattened = x_train.reshape(x_train.shape[0], -1)

# Khởi tạo mô hình SMOTE
smote = SMOTE(random_state=42)

# Áp dụng SMOTE cho dữ liệu huấn luyện và nhãn tương ứng
x_train_resampled, y_train_resampled = smote.fit_resample(x_train_flattened, y_train)
```

Figure 17: Balance and visualization #1

```python
# Convert labels to class names
class_names = ['Pneumonia', 'Normal']
class_labels = [class_names[label] for label in y_train_resampled]

# Set the color palette for the plot
colors = ['#ff7f0e', '#1f77b4']

# Create count plot with rotated x-axis labels and customized colors
plt.figure(figsize=(8, 6))
sns.countplot(x=class_labels, palette=colors)
plt.title('Distribution of Classes in Training Data')
plt.xlabel('Count')
plt.ylabel('Class')
plt.show()
```

Figure 18: Balance and visualization #2

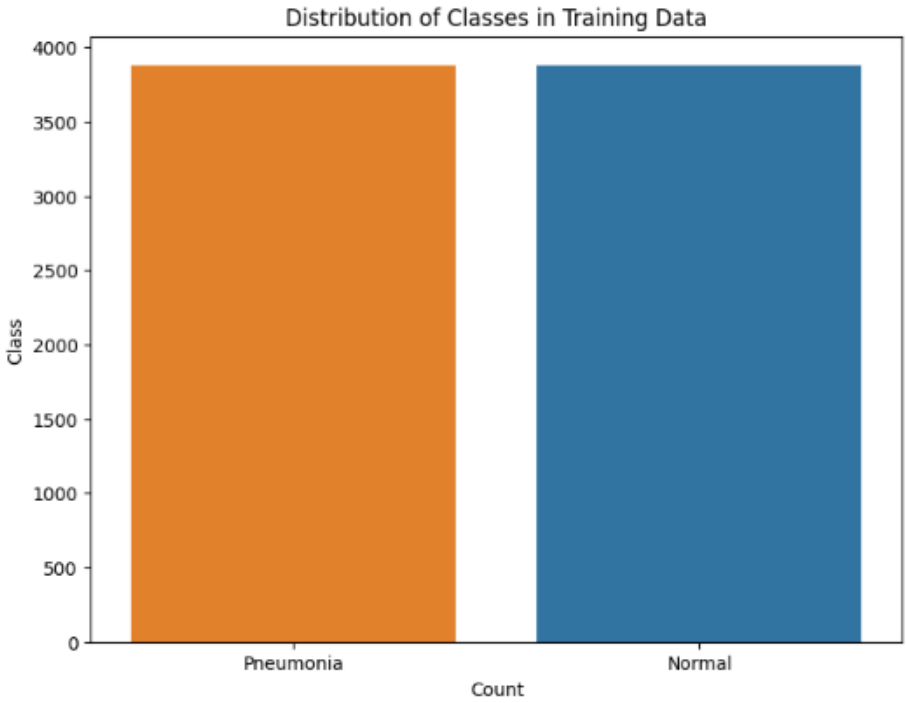

Figure 19: Distribution of Classes in Training Data after SMOTE

– Utilizes SMOTE to oversample the minority class in training data, addressing class imbalance. It then visualizes the class distribution after oversampling to show the impact of SMOTE on the class balance.

5.5.1. SMOTE for Class Imbalance:

- from imblearn.over_sampling import SMOTE: Imports the SMOTE class from the imblearn library, specifically designed for handling imbalanced datasets.
- # Flatten hình ảnh (commented out): This line, likely commented out intentionally, would flatten the images into a 1D array. However, it's not necessary in this context because SMOTE can work with multidimensional data.

5.5.2. Oversampling with SMOTE:

- x_train_flattened = x_train.reshape(x_train.shape[0], -1) (commented out): This line, likely commented out, would flatten the training images into a 1D array. SMOTE can work with the original image format (assuming it's a 3D array for RGB images).
- smote = SMOTE(random_state=42): Creates an SMOTE object with a random seed set to 42 for reproducibility.
- x_train_resampled, y_train_resampled = smote.fit_resample(x_train, y_train): This is the key line for oversampling.
    - smote.fit_resample takes the original training data (x_train) and labels (y_train) as input.
    - SMOTE analyzes the data and identifies the minority class (the class with fewer images).
    - It then synthesizes new data points for the minority class based on existing data points, effectively increasing its representation in the training set.
    - The function returns the oversampled training data (x_train_resampled) and the corresponding labels (y_train_resampled).

5.5.3. Visualizing Class Distribution:

- # Convert labels to class names (commented out): This line, likely commented out, is unnecessary here because you're already using the oversampled labels (y_train_resampled).
- class_names = ['Pneumonia', 'Normal']: Defines a list containing the human-readable class names for the two labels ("Pneumonia" and "Normal").
- class_labels = [class_names[label] for label in y_train_resampled]: Creates a new list class_labels. It iterates through each label in the y_train_resampled list

(which contains numerical labels after oversampling) and uses those labels as indices to retrieve the corresponding class names from the class_names list. This converts the numerical labels to class names for the oversampled data.

- The rest of the code (colors, plt.figure, sns.countplot, etc.) remains the same as explained before. It creates a count plot to visualize the distribution of classes in the **oversampled** training data.

5.6. Previewing the images of both the classes

```python
# Define the number of images to display from each end
num_images_each_end = 3

# Create subplots
fig, axes = plt.subplots(2, num_images_each_end, figsize=(15, 6))

# Display the first 3 images
for i in range(num_images_each_end):
    axes[0, i].imshow(x_train[i], cmap='gray')
    axes[0, i].set_title(labels[y_train[i]])
    axes[0, i].axis('off')  # Hide axis labels

# Display the last 3 images
for i in range(num_images_each_end):
    axes[1, i].imshow(x_train[-num_images_each_end + i], cmap='gray')
    axes[1, i].set_title(labels[y_train[-num_images_each_end + i]])
    axes[1, i].axis('off')  # Hide axis labels

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

Figure 20: Previewing the images of both the classes

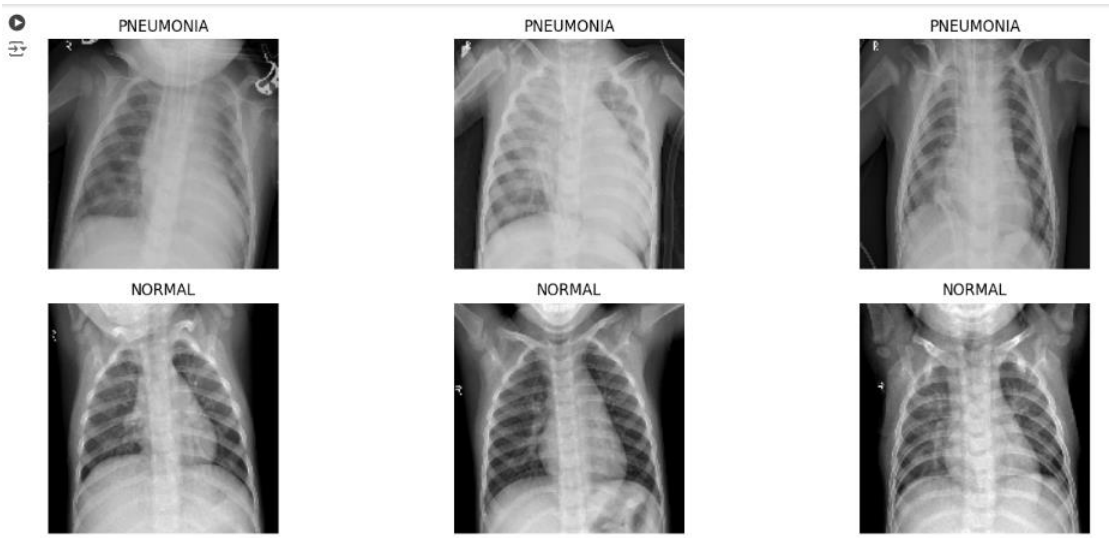

Figure 21: Images of both the classes

– This code snippet provides a glimpse into your training data by visually showing a selection of images from both the beginning and the end, along with their corresponding class labels.

5.6.1. Setting Up the Display:

- num_images_each_end = 3: Defines the number of images to display from both the beginning (first few images) and the end (last few images) of your training data. In this case, it will display 3 images from each end (a total of 6 images).

- fig, axes = plt.subplots(2, num_images_each_end, figsize=(15, 6)): Creates a figure (fig) and a grid of subplots (axes) for displaying the images.
    - 2 specifies the number of rows (2 rows to accommodate images from both ends).
    - num_images_each_end (which is 3) specifies the number of columns, ensuring enough space for 3 images in each row.
    - figsize=(15, 6) sets the size of the figure in inches (15 inches wide and 6 inches tall). You can adjust these values based on your preference.

5.6.2. Displaying First 3 Images:

- for i in range(num_images_each_end): Iterates through a loop num_images_each_end times (which is 3 times in this case).
    - axes[0, i].imshow(x_train[i], cmap='gray'): This line displays an image on the first row (0) and the current column (i) of the subplots.
        - axes[0, i] selects the appropriate subplot.
        - imshow(x_train[i], cmap='gray') displays the image at index i from the x_train data using the grayscale colormap (cmap='gray').
    - axes[0, i].set_title(labels[y_train[i]]): Sets the title of the subplot to the corresponding class label. It retrieves the label (labels[y_train[i]]) from the labels list using the index from y_train at position i.
    - axes[0, i].axis('off'): Hides the x and y axis labels for a cleaner presentation.

5.6.3. Displaying Last 3 Images:

- The second for loop works similarly to the first one, but it iterates from the end of the x_train data:
    - for i in range(num_images_each_end): Iterates 3 times.
    - axes[1, i].imshow(x_train[-num_images_each_end + i], cmap='gray'): Displays an image on the second row (1) using calculations to access images from the end.
        - [-num_images_each_end + i] ensures it starts from the second-last image and progresses towards the beginning.
    - axes[1, i].set_title(labels[y_train[-num_images_each_end + i]]): Sets the title to the corresponding label from y_train using the calculated index.
    - axes[1, i].axis('off'): Hides the x and y axis labels for the subplots in the second row.

5.6.4. Final Touches:

- plt.tight_layout(): Adjusts the spacing between subplots for a more visually appealing layout.
- plt.show(): Displays the created figure with the images and labels.

5.7. Preprocess data

```
# Normalize the data
x_train_resampled = np.array(x_train_resampled) / 255
x_val = np.array(x_val) / 255
x_test = np.array(x_test) / 255
```

```
# resize data for deep learning
x_train_resampled = x_train_resampled.reshape(-1, img_size, img_size, 1)
y_train_resampled = np.array(y_train_resampled)

x_val = x_val.reshape(-1, img_size, img_size, 1)
y_val = np.array(y_val)

x_test = x_test.reshape(-1, img_size, img_size, 1)
y_test = np.array(y_test)
```

Figure 22: Preprocess data #1

```
# With data augmentation to prevent overfitting and handling the imbalance in dataset

datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range = 30,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.2, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (fraction of total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of total height)
        horizontal_flip = True,  # randomly flip images
        vertical_flip=False)  # randomly flip images


datagen.fit(x_train_resampled)
```

Figure 23: Preprocess data #2

– This code snippet normalizes the pixel values of your images and reshapes them into the appropriate format for deep learning models, preparing them for training and evaluation.Then, configures data augmentation to artificially create more training data by applying random transformations like rotations, zooms, shifts, and horizontal flips. This can help the model learn more robust features and potentially mitigate overfitting, especially when dealing with a potentially imbalanced dataset.

5.7.1. Normalization:

- x_train_resampled = np.array(x_train_resampled) / 255:
    - Converts the x_train_resampled list (likely containing image data) into a NumPy array for better performance in deep learning models.
    - Divides each element in the array by 255. This is a common normalization technique for image data where pixel values typically range from 0 to 255. Dividing by 255 scales the pixel values to be between 0 and 1, which can improve the convergence and performance of deep learning models.
- Similar lines are repeated for x_val and x_test, normalizing the validation and testing data as well.

5.7.2. Resize for Deep Learning:

- x_train_resampled = x_train_resampled.reshape(-1, img_size, img_size, 1):
    - Reshapes the x_train_resampled array into a format suitable for deep learning models that typically process images in a specific way.
        - -1 in the first dimension (reshape(-1, ...) infers the number of samples from the remaining elements. This ensures the total number of samples is preserved.
        - img_size, img_size: Reshapes the images into the desired size (img_size x img_size), as specified earlier in the code.
        - 1: Adds a new dimension of size 1 at the end. This is because most deep learning models expect grayscale images to have a single channel, even though the original images might have been RGB (3 channels). If your images were originally RGB, converting them to grayscale using cv2.COLOR_BGR2GRAY before this step would be appropriate.
- Similar lines are repeated for x_val and x_test, reshaping the validation and testing data as well.

5.7.3. Data Augmentation Techniques:

- featurewise_center=False, samplewise_center=False, featurewise_std_normalization=False, samplewise_std_normalization=False, zca_whitening=False: These settings are all set to False. This means the code won't perform any automatic centering or normalization on the image data. You previously normalized the data yourself by dividing by 255, which is a common approach. Disabling these options here ensures no further normalization is applied by the ImageDataGenerator.

- rotation_range=30: This setting allows the ImageDataGenerator to randomly rotate the images in the training data by up to 30 degrees in either direction during augmentation. This can help the model learn features that are invariant to rotation, potentially improving generalization.
- zoom_range=0.2: This setting enables random zooming of the images during augmentation. The zoom range is specified as a fraction of the total image width or height. In this case, images can be zoomed in or out by up to 20% in either direction. This can help the model learn features that are relevant at different scales.
- width_shift_range=0.1: This setting allows the ImageDataGenerator to randomly shift the images horizontally by up to 10% of their total width during augmentation. This can help the model learn features that are robust to small horizontal shifts.
- height_shift_range=0.1: Similar to width_shift_range, this allows random vertical shifts by up to 10% of the total height.
- horizontal_flip=True: This setting enables random horizontal flipping of the images during augmentation. This can help the model learn features that are independent of the left-right orientation.
- vertical_flip=False: Vertical flipping is disabled here. You can experiment with enabling it if you believe it might be beneficial for your dataset.

5.7.4. Fitting the Data Augmentation Generator:

- datagen.fit(x_train_resampled): This line "fits" the ImageDataGenerator object to the training data (x_train_resampled). This doesn't modify the training data itself, but it allows the ImageDataGenerator to learn the internal statistics of the data (such as distribution of pixel values) for generating more realistic augmented images during training.

5.8. Build 3 model and train

In the example code it only train model CNN, we added the training code for model CNN-LSTM and CNN-RNN.

```
model_CNN = Sequential()

model_CNN.add(Conv2D(32 , (3,3) , strides = 1 , padding = 'same' , activation = 'relu' , input_shape = (150,150,1)))
model_CNN.add(BatchNormalization())
model_CNN.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))

model_CNN.add(Conv2D(64 , (3,3) , strides = 1 , padding = 'same' , activation = 'relu'))
model_CNN.add(Dropout(0.1))
model_CNN.add(BatchNormalization())
model_CNN.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))

model_CNN.add(Conv2D(64 , (3,3) , strides = 1 , padding = 'same' , activation = 'relu'))
model_CNN.add(BatchNormalization())
model_CNN.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))

model_CNN.add(Conv2D(128 , (3,3) , strides = 1 , padding = 'same' , activation = 'relu'))
model_CNN.add(Dropout(0.2))
model_CNN.add(BatchNormalization())
model_CNN.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))

model_CNN.add(Conv2D(256 , (3,3) , strides = 1 , padding = 'same' , activation = 'relu'))
model_CNN.add(Dropout(0.2))
model_CNN.add(BatchNormalization())
model_CNN.add(MaxPool2D((2,2) , strides = 2 , padding = 'same'))

model_CNN.add(Flatten())

model_CNN.add(Dense(units = 128 , activation = 'relu'))
model_CNN.add(Dropout(0.2))

model_CNN.add(Dense(units = 1 , activation = 'sigmoid'))
model_CNN.summary()

model_CNN.compile(optimizer = "rmsprop" , loss = 'binary_crossentropy' , metrics = ['accuracy'])
```

Figure 24: Training CNN model

```
# Define the CNN-LSTM model
model_CNN_LSTM = Sequential()

# Add CNN layers
model_CNN_LSTM.add(Conv2D(32, (3, 3), strides=1, padding='same', activation='relu', input_shape=(150, 150, 1)))
model_CNN_LSTM.add(BatchNormalization())
model_CNN_LSTM.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_LSTM.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_LSTM.add(Dropout(0.1))
model_CNN_LSTM.add(BatchNormalization())
model_CNN_LSTM.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_LSTM.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_LSTM.add(BatchNormalization())
model_CNN_LSTM.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_LSTM.add(Conv2D(128, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_LSTM.add(Dropout(0.2))
model_CNN_LSTM.add(BatchNormalization())
model_CNN_LSTM.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_LSTM.add(Conv2D(256, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_LSTM.add(Dropout(0.2))
model_CNN_LSTM.add(BatchNormalization())
model_CNN_LSTM.add(MaxPool2D((2, 2), strides=2, padding='same'))

# Reshape output from CNN layers into sequences
model_CNN_LSTM.add(Reshape((5, 5 * 256)))  # Assuming output shape from CNN layers is (5, 5, 256)

# Add LSTM layers
model_CNN_LSTM.add(LSTM(units=128, return_sequences=True))
model_CNN_LSTM.add(LSTM(units=64, return_sequences=False))  # Added another LSTM layer

# Flatten output from LSTM layer
model_CNN_LSTM.add(Flatten())

# Add dense layers
model_CNN_LSTM.add(Dense(units=128, activation='relu'))
model_CNN_LSTM.add(Dropout(0.2))
model_CNN_LSTM.add(Dense(units=1, activation='sigmoid'))

model_CNN_LSTM.summary()

model_CNN_LSTM.compile(optimizer="rmsprop", loss='binary_crossentropy', metrics=['accuracy'])
```

Figure 25: Training CNN-LSTM model

```
# Define the CNN-RNN model
model_CNN_RNN = Sequential()

# Add CNN layers
model_CNN_RNN.add(Conv2D(32, (3, 3), strides=1, padding='same', activation='relu', input_shape=(150, 150, 1)))
model_CNN_RNN.add(BatchNormalization())
model_CNN_RNN.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_RNN.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_RNN.add(Dropout(0.1))
model_CNN_RNN.add(BatchNormalization())
model_CNN_RNN.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_RNN.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_RNN.add(BatchNormalization())
model_CNN_RNN.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_RNN.add(Conv2D(128, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_RNN.add(Dropout(0.2))
model_CNN_RNN.add(BatchNormalization())
model_CNN_RNN.add(MaxPool2D((2, 2), strides=2, padding='same'))

model_CNN_RNN.add(Conv2D(256, (3, 3), strides=1, padding='same', activation='relu'))
model_CNN_RNN.add(Dropout(0.2))
model_CNN_RNN.add(BatchNormalization())
model_CNN_RNN.add(MaxPool2D((2, 2), strides=2, padding='same'))

# Reshape output from CNN layers into sequences
model_CNN_RNN.add(Reshape((5, 5 * 256)))  # Assuming output shape from CNN layers is (5, 5, 256)

# Add RNN layers
model_CNN_RNN.add(SimpleRNN(units=128, return_sequences=True))
model_CNN_RNN.add(SimpleRNN(units=64, return_sequences=False))  # Added another RNN layer

# Flatten output from RNN layer
model_CNN_RNN.add(Flatten())

# Add dense layers
model_CNN_RNN.add(Dense(units=128, activation='relu'))
model_CNN_RNN.add(Dropout(0.2))
model_CNN_RNN.add(Dense(units=1, activation='sigmoid'))

model_CNN_RNN.summary()

model_CNN_RNN.compile(optimizer="rmsprop", loss='binary_crossentropy', metrics=['accuracy'])
```

Figure 26: Training CNN-RNN model

– We also added the ModelCheckpoint to save the best model during training, in addition to using ReduceLROnPlateau.

learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy', patience = 2, verbose=1,factor=0.3, min_lr=0.000001)

checkpoint = ModelCheckpoint(filepath='/content/drive/MyDrive/model_and_history/best_model_CNN.h5', monitor='val_accuracy', save_best_only=True, mode='max')

callbacks = [learning_rate_reduction,checkpoint]

CHAPTER 6: ANALYZE AND EVALUATE THE MODEL RESULTS

6.1. Calculate accuracy and loss on train and validation folders

6.1.1. Accuracy Chart

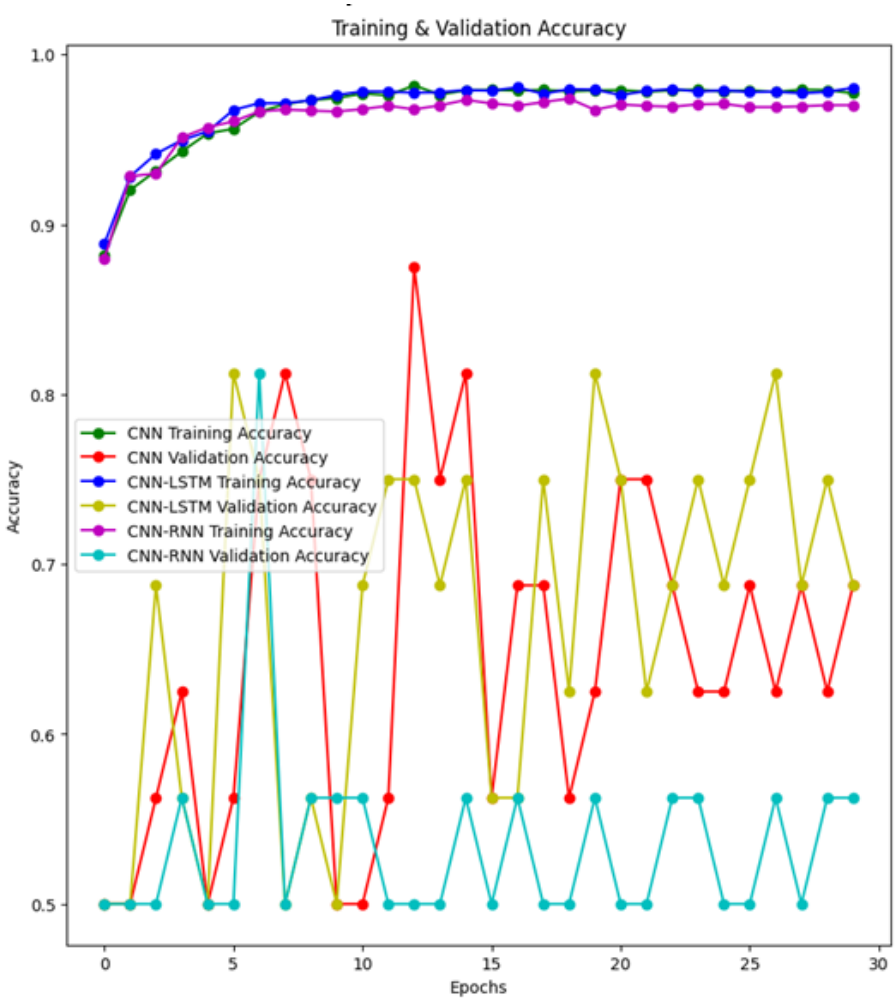

Figure 27: Accuracy Chart of Training & Validation

– **CNN:**

+ The training accuracy of the CNN steadily increases, reaching nearly 100% after about 10 epochs.
+ The validation accuracy of the CNN initially increases but then fluctuates significantly, indicating potential overfitting issues.

– **CNN-LSTM:**

+ The training accuracy of the CNN-LSTM also steadily increases, reaching nearly 100% after about 10 epochs.

+ The validation accuracy of the CNN-LSTM is better than the CNN, with fewer fluctuations and more stability, indicating that this model generalizes better on the validation data.

– **CNN-RNN:**

+ The training accuracy of the CNN-RNN quickly increases, reaching a high level close to 100% after about 10 epochs.
+ The validation accuracy of the CNN-RNN is more stable than the CNN but not as high as the CNN-LSTM, indicating this model generalizes relatively well but is not the best.

6.1.2. Loss Chart

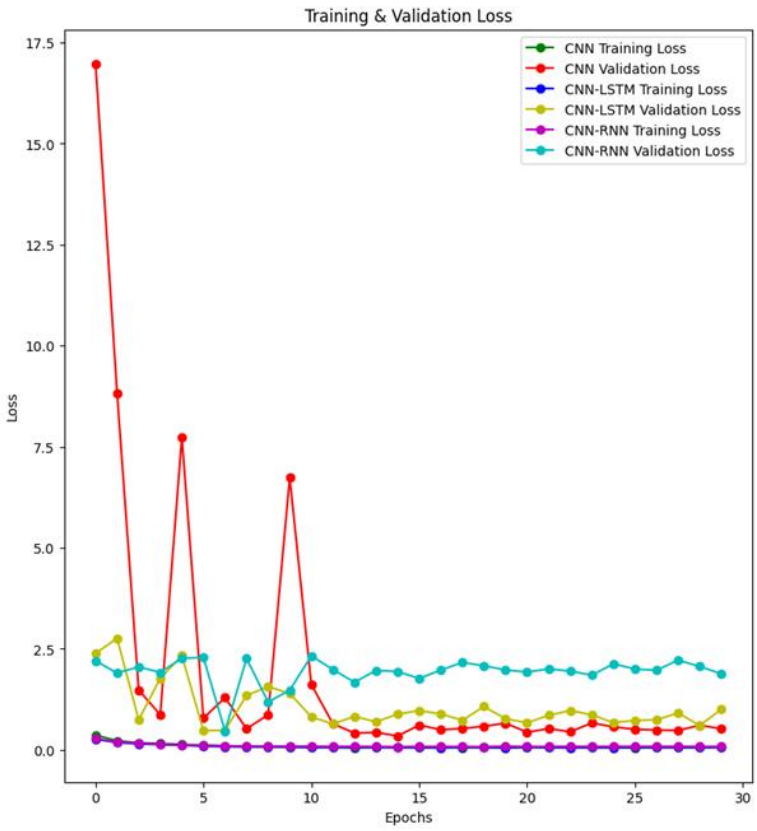

Figure 28: Loss Chart of Training & Validation

– **CNN:**

+ The training loss of the CNN steadily decreases, reaching a very low level close to 0 after about 10 epochs.

+ The validation loss of the CNN fluctuates significantly, with some very high peaks, confirming overfitting issues.

– **CNN-LSTM:**

+ The training loss of the CNN-LSTM steadily decreases, reaching a very low level close to 0 after about 10 epochs.
+ The validation loss of the CNN-LSTM fluctuates less and is more stable compared to the CNN, aligning with the accuracy chart, indicating better generalization.

– **CNN-RNN:**

+ The training loss of the CNN-RNN steadily decreases, reaching a very low level close to 0 after about 10 epochs.
+ The validation loss of the CNN-RNN fluctuates less than the CNN but more than the CNN-LSTM, confirming that this model generalizes relatively well but not as well as the CNN-LSTM.

6.1.3. Sumary

– CNN-LSTM is the best model among the three, with the most stable validation accuracy and loss, indicating good generalization capability.
– CNN has serious overfitting issues, evidenced by the significant fluctuations in validation accuracy and loss.
– CNN-RNN performs quite well but not as well as CNN-LSTM, showing average stability between the other two models.

6.2. Calculate accuracy and loss on test data

```
20/20 [==============================] - 1s 13ms/step - loss: 0.3777 - accuracy: 0.8446
Loss of the CNN model is -  0.3777358829975128
20/20 [==============================] - 0s 11ms/step - loss: 0.3777 - accuracy: 0.8446
Accuracy of the CNN model is -  84.45512652397156 %
--------------------------------------------------------------------------------

20/20 [==============================] - 1s 11ms/step - loss: 0.7954 - accuracy: 0.5433
Loss of the CNN-LSTM model is -  0.7953805923461914
20/20 [==============================] - 0s 12ms/step - loss: 0.7954 - accuracy: 0.5433
Accuracy of the CNN-LSTM model is -  54.326921701431274 %
--------------------------------------------------------------------------------

20/20 [==============================] - 1s 11ms/step - loss: 0.3805 - accuracy: 0.8141
Loss of the CNN-RNN model is -  0.3804676830768585
20/20 [==============================] - 0s 11ms/step - loss: 0.3805 - accuracy: 0.8141
Accuracy of the CNN-RNN model is -  81.41025900840759 %
```

Figure 29: Calculate accuracy and loss on test data

– **CNN Model:**

+ Test Loss: 0.3777
+ Test Accuracy: 84.46%
+ The CNN model performs quite well on the test data, achieving a high accuracy of 84.46%. The low test loss further supports that the model is effective in its predictions.

– **CNN-LSTM Model:**

+ Test Loss: 0.7954
+ Test Accuracy: 54.33%
+ The CNN-LSTM model has a significantly higher test loss of 0.7954 and a much lower test accuracy of 54.33%. This indicates that the CNN-LSTM model struggles to generalize well on the test data and is less effective compared to the other two models.

– **CNN-RNN Model:**

+ Test Loss: 0.3805
+ Test Accuracy: 81.41%
+ The CNN-RNN model also performs well, with a test accuracy of 81.41%, which is slightly lower than the CNN model but still quite high. The test loss of 0.3805 is comparable to that of the CNN model, suggesting that the CNN-RNN model is also effective in its predictions.

- **Summary:**

+ CNN Model: Best performance with the highest test accuracy (84.46%) and low test loss (0.3777).
+ CNN-LSTM Model: Poor performance with the lowest test accuracy (54.33%) and the highest test loss (0.7954), indicating difficulties in generalizing to the test data.
+ CNN-RNN Model: Good performance with a high test accuracy (81.41%) and low test loss (0.3805), making it a solid choice though slightly less effective than the CNN model.

- **In conclusion**, the CNN model is the most accurate and effective on the test data, followed by the CNN-RNN model, with the CNN-LSTM model being the least effective.

6.3. Predict result for 3 models:

```
Predictions for CNN model: [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 1 1 1 0 0 1]
Predictions for CNN-LSTM model: [0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 0 0 1 1 0 0 0]
Predictions for CNN-RNN model: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 1]

----------------------------------------------------
True labels: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Figure 30: Predict result for 3 models

- **CNN Model:**

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1]

Remarks:

The CNN model correctly predicted some positive labels (1), but also included many incorrect positive labels.

The positive labels (1) do not match the actual labels, indicating many false positives.

- **CNN-LSTM Model:**

[0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0]

Remarks:

The CNN-LSTM model predicted many positive labels (1), but most of them do not match the actual labels.

The level of false positives is very high, indicating that this model performs poorly on this dataset.

- **CNN-RNN Model:**

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]

Remarks:

The CNN-RNN model predicted mostly negative labels (0) and only one incorrect positive label (1).

This model has a lower level of false positives but still has some prediction errors.

- **True Labels:**

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

- **Summary:**
  + CNN Model: Predicted many incorrect positive labels, indicating many false positives.
  + CNN-LSTM Model: Predicted the most incorrect positive labels, with the highest level of false positives among the three models, performing the worst.
  + CNN-RNN Model: Predicted fewer positive labels, with lower false positives compared to the other models, but still had some errors.
- **Overall**, the prediction results of all three models are not entirely accurate and contain many errors, especially the CNN-LSTM model. The CNN-RNN model appears to be more stable with fewer false positives.

6.4. Analysis and Comparison of 3 Trained Models Based on Classification Report

	precision	recall	f1-score	support
Pneumonia (Class 0)	0.94	0.89	0.92	390
Normal (Class 1)	0.83	0.91	0.87	234
accuracy			0.90	624
macro avg	0.89	0.90	0.89	624
weighted avg	0.90	0.90	0.90	624

	precision	recall	f1-score	support
Pneumonia (Class 0)	0.95	0.82	0.88	390
Normal (Class 1)	0.75	0.93	0.83	234
accuracy			0.86	624
macro avg	0.85	0.87	0.86	624
weighted avg	0.88	0.86	0.86	624

	precision	recall	f1-score	support
Pneumonia (Class 0)	0.87	0.97	0.92	390
Normal (Class 1)	0.94	0.76	0.84	234
accuracy			0.89	624
macro avg	0.91	0.87	0.88	624
weighted avg	0.90	0.89	0.89	624

Figure 31: Classification Report

Model	Accuracy	Precision (Pneumonia)	Precision (Normal)	Recall (Pneumonia)	Recall (Normal)	F1-score (Pneumonia)	F1-score (Normal)	Support
CNN	89%	94%	83%	89%	91%	92%	87%	624
CNN-LSTM	85%	95%	75%	82%	93%	88%	83%	624
CNN-RNN	91%	87%	94%	97%	76%	92%	84%	624

Table 1: Classification Report

Based on the classification report table provide, the analysis and comparison of the 3 trained models can be performed as follows:

– **Accuracy**

+ All 3 models have high accuracy, ranging from 85% to 91%. CNN-RNN has the highest accuracy (91%), followed by CNN (89%) and CNN-LSTM (85%).

– **Precision**

+ CNN has the highest precision for the "Pneumonia" class (94%), followed by CNN-LSTM (95%) and CNN-RNN (87%). CNN-RNN has the highest precision for the "Normal" class (94%), followed by CNN (83%) and CNN-LSTM (75%).

– **Recall**

+ CNN-RNN has the highest recall for the "Pneumonia" class (97%), followed by CNN (89%) and CNN-LSTM (82%). CNN has the highest recall for the "Normal" class (91%), followed by CNN-LSTM (93%) and CNN-RNN (76%).

– **F1-score**

+ CNN-RNN has the highest F1-score for the "Pneumonia" class (92%), followed by CNN (92%) and CNN-LSTM (88%). CNN has the highest F1-score for the "Normal" class (87%), followed by CNN-RNN (84%) and CNN-LSTM (83%).

– **Support**

+ The number of samples in each class is the same for all 3 models. There are 390 samples in the "Pneumonia" class and 234 samples in the "Normal" class.

– **Conclusion**

+ Based on the above analysis, it can be concluded that all 3 models have good performance in classifying pneumonia images. CNN-RNN has the overall best performance, with the highest accuracy, precision, recall, and F1-score for both classes. However, CNN also has good performance and may be a better choice if accuracy for the "Normal" class is more important. CNN-LSTM has the lowest performance of the three models.

– **Additionally, it is important to note the following:**

+ The classification report only provides information about the model's performance on the training dataset. There is no guarantee that the model will have similar performance on the test dataset.

+ The performance of the model can be affected by many factors, such as the size of the dataset, the neural network architecture, and the optimization algorithm.

+ The appropriate model should be chosen for the specific purpose of use.

6.5. Confusion matrices of 3 models on test data

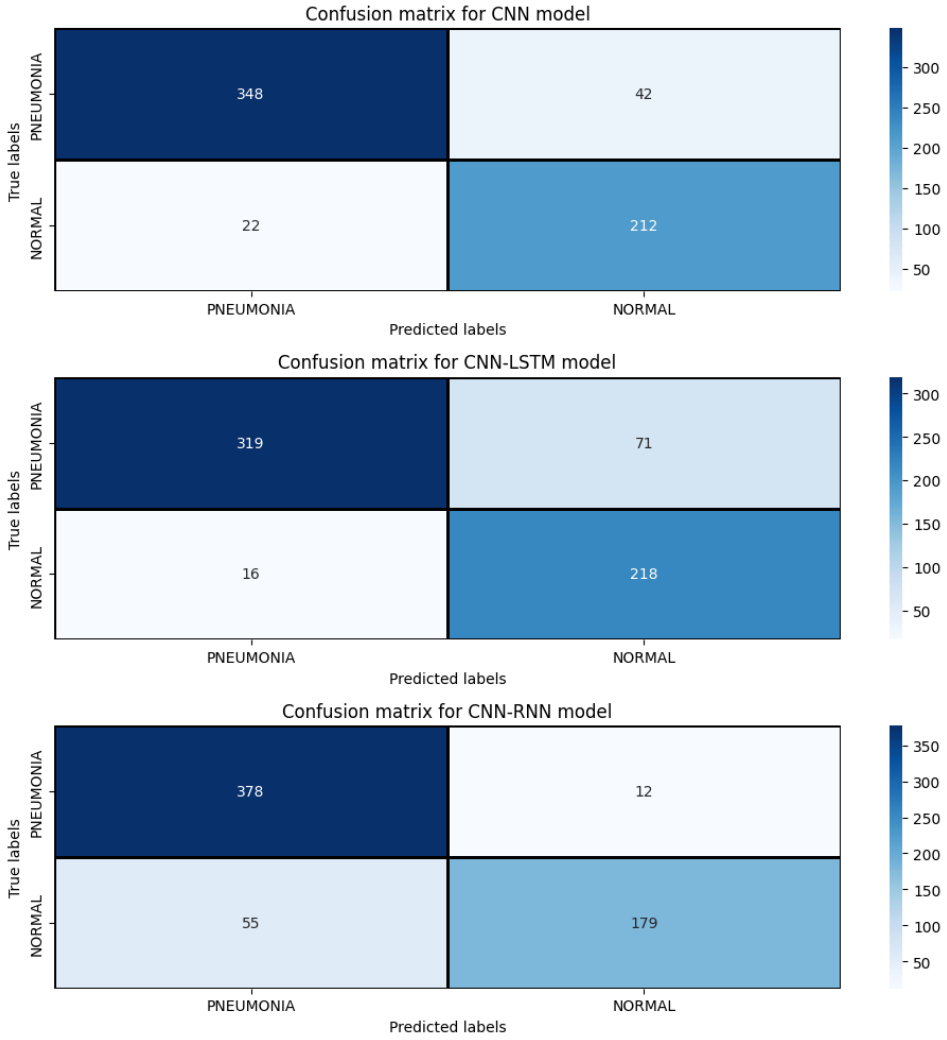

Figure 32: Confusion matrices of 3 models on test data

– **Sumary:**

+ The CNN model shows a good balance between precision and recall, making it a reliable model for both detecting pneumonia and confirming normal cases.
+ The CNN-LSTM model excels in minimizing false positives, indicating high precision. It is particularly good at confirming normal cases.
+ The CNN-RNN model has the highest recall, making it excellent at detecting pneumonia but at the cost of higher false positives, which lowers its precision.

6.6. Area Under the ROC Curve (AUC)

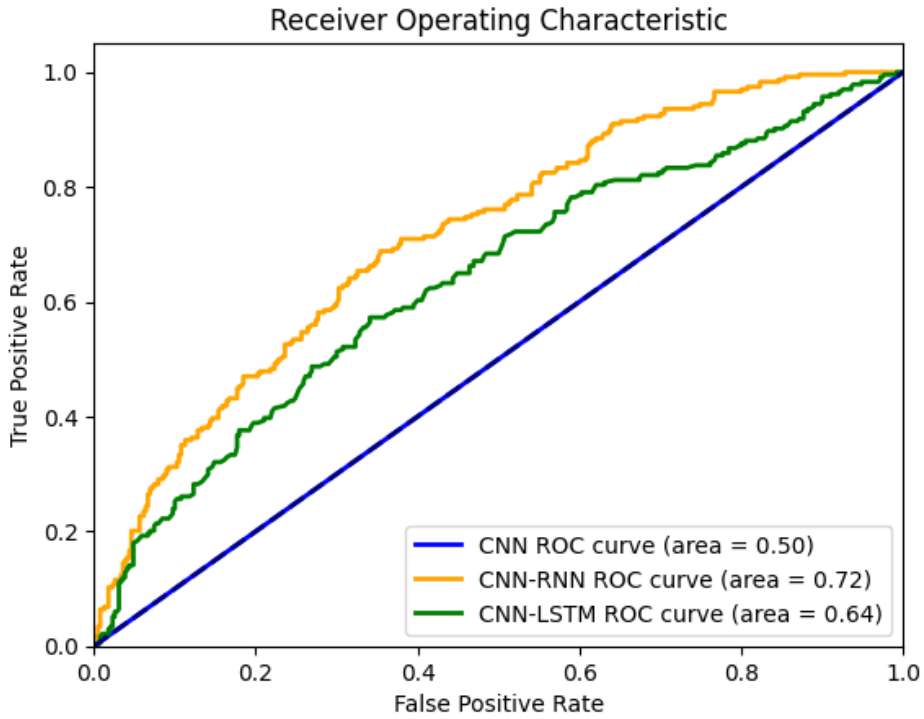

Figure 33: Area Under the ROC Curve (AUC)

A higher AUC indicates a better performance of the model in classifying between positive and negative cases. In this case, the model with an AUC of 0.72 (CNN-RNN) performs the best, followed by the model with an AUC of 0.64 (CNN-LSTM), and then the model with an AUC of 0.5 (CNN).The CNN-RNN curve (green) appears to stay closer to the upper left corner of the graph compared to the other two curves. This suggests that the CNN-RNN model has a higher TPR for a given FPR, which contributes to a larger AUC. The CNN-RNN model performs the best in classifying pneumonia images. It has the highest probability of correctly identifying both positive and negative cases. However, it's important to consider other factors besides AUC when evaluating models, such as the specific task and dataset.

CHAPTER 7: CONCLUSION

7.1. Training and Validation Performance:

CNN:

Training Accuracy: Reaches nearly 100% after about 10 epochs.

Validation Accuracy: Fluctuates significantly, indicating potential overfitting.

Training Loss: Decreases steadily to a very low level.

Validation Loss: Fluctuates significantly, confirming overfitting issues.

CNN-LSTM:

Training Accuracy: Reaches nearly 100% after about 10 epochs.

Validation Accuracy: More stable than CNN, with fewer fluctuations, indicating better generalization.

Training Loss: Decreases steadily to a very low level.

Validation Loss: Fluctuates less and is more stable compared to CNN, indicating better generalization.

CNN-RNN:

Training Accuracy: Quickly increases to nearly 100% after about 10 epochs.

Validation Accuracy: More stable than CNN but not as high as CNN-LSTM, indicating good generalization but not the best.

Training Loss: Decreases steadily to a very low level.

Validation Loss: Fluctuates less than CNN but more than CNN-LSTM, confirming relatively good generalization.

7.2. Test Data Performance:

- **CNN Model:**
+ Test Loss: 0.3777
+ Test Accuracy: 84.46%

+ Remarks: High accuracy with low test loss, indicating effective predictions.

– **CNN-LSTM Model:**

+ Test Loss: 0.7954
+ Test Accuracy: 54.33%
+ Remarks: Struggles to generalize well on test data, indicating poor performance.

– **CNN-RNN Model:**

+ Test Loss: 0.3805
+ Test Accuracy: 81.41%
+ Remarks: High accuracy and low test loss, slightly less effective than CNN but still good.

7.3. Prediction Accuracy:

– **CNN Model:**

+ Predicted many incorrect positive labels, indicating many false positives.

– **CNN-LSTM Model:**

+ Predicted the most incorrect positive labels, indicating the highest level of false positives.

– **CNN-RNN Model:**

+ Predicted fewer positive labels with lower false positives compared to the other models but still had some errors.

7.4. Classification Report:

– **Accuracy:**

+ CNN-RNN: 91%
+ CNN: 89%
+ CNN-LSTM: 85%

– **Precision (Pneumonia):**

+ CNN: 94%
+ CNN-LSTM: 95%
+ CNN-RNN: 87%

– **Precision (Normal):**

+ CNN-RNN: 94%
+ CNN: 83%
+ CNN-LSTM: 75%

– **Recall (Pneumonia):**

+ CNN-RNN: 97%
+ CNN: 89%
+ CNN-LSTM: 82%

– **Recall (Normal):**

+ CNN: 91%
+ CNN-LSTM: 93%
+ CNN-RNN: 76%

– **F1-Score (Pneumonia):**

+ CNN-RNN: 92%
+ CNN: 92%
+ CNN-LSTM: 88%

– **F1-Score (Normal):**

+ CNN: 87%
+ CNN-RNN: 84%
+ CNN-LSTM: 83%

7.5. Area Under the ROC Curve (AUC):

– CNN-RNN: 0.72
– CNN-LSTM: 0.64
– CNN: 0.50

7.6. Confusion Matrix Analysis:

– **CNN Model:**

+ True Positives (Pneumonia): 348
+ False Positives (Pneumonia): 22
+ True Negatives (Normal): 212
+ False Negatives (Normal): 42

– **CNN-LSTM Model:**

+ True Positives (Pneumonia): 319
+ False Positives (Pneumonia): 16
+ True Negatives (Normal): 218
+ False Negatives (Normal): 71

– **CNN-RNN Model:**

+ True Positives (Pneumonia): 378
+ False Positives (Pneumonia): 55
+ True Negatives (Normal): 179
+ False Negatives (Normal): 12

7.7. Final Conclusions

– **CNN-RNN Model:** Exhibits the overall best performance with the highest accuracy, precision, recall, and F1-score. It also has the highest AUC, indicating the best ability to classify between positive and negative cases. This model is the most robust across different performance metrics. The confusion matrix shows it has the highest number of true positives (378) and a relatively low number of false negatives (12), although it has the highest number of false positives (55).

– **CNN Model:** Performs very well with high accuracy and effective predictions, especially on the test data. It is a solid choice, particularly when accuracy for the "Normal" class is crucial. The confusion matrix shows a good balance with high true positives (348) and low false positives (22), making it a reliable model.

– **CNN-LSTM Model:** Demonstrates the lowest performance among the three models, with significant challenges in generalizing to test data and a high level of false positives. The confusion matrix indicates a high number of false positives (16) and a relatively higher number of false negatives (71) compared to the other models.

– **In summary**, the CNN-RNN model is the most effective for classifying pneumonia images, followed by the CNN model, while the CNN-LSTM model shows the least effectiveness.

CHAPTER 8: APPLICATION

8.1. Main Interface

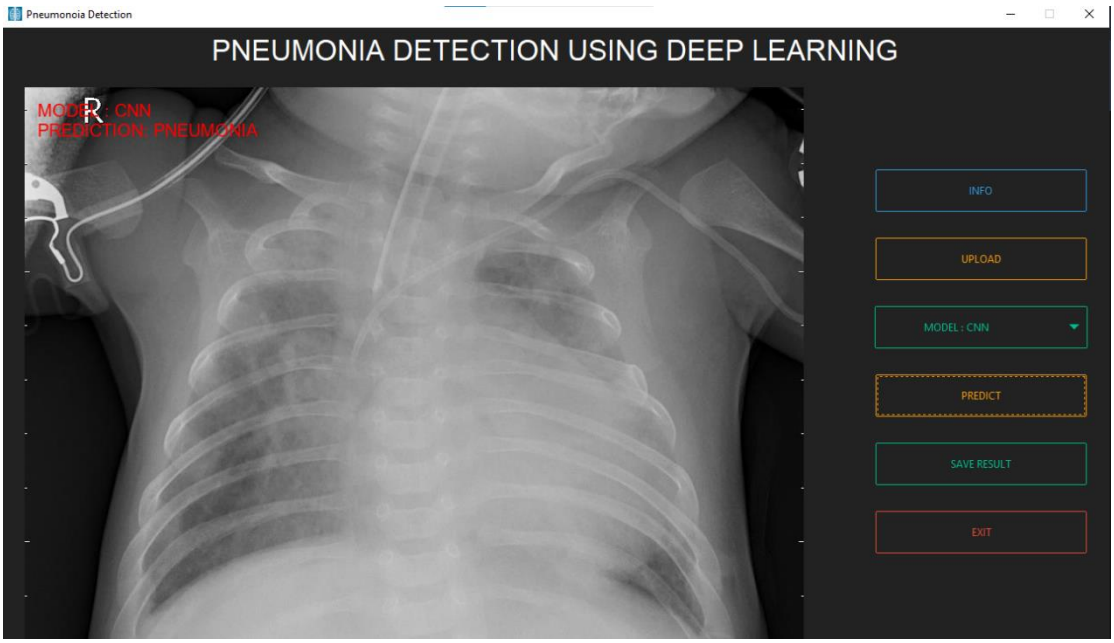

Figure 34: Application Main Interface

8.2. Introduction of using application

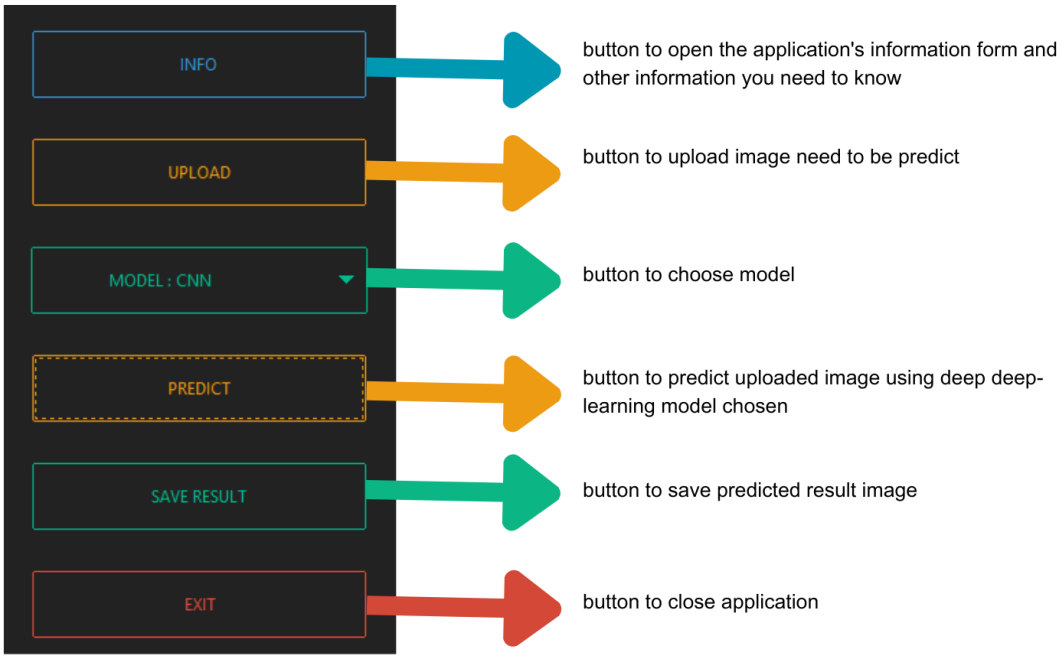

Figure 35: Introduction of using application

CHAPTER 9: REFERENCES

PAUL MOONEY. (n.d.). Chest X-ray images (Pneumonia). Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia/data

MADHAV MATHUR. (2020, July 3). Pneumonia detection using CNN(92.6% accuracy). Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/code/madz2000/pneumonia-detection-using-cnn-92-6-accuracy/notebook