Pneumonia Detection Using CNN

# Introduction:

* The code focuses on building a deep learning model to classify chest X-ray images as either normal or indicating pneumonia.
* It utilizes a dataset of chest X-ray images consisting of training, testing, and validation sets.
* The goal is to train a Convolutional Neural Network (CNN) model to accurately classify the X-ray images.
* Key steps in the code include data loading, data preprocessing, model creation, model training, and evaluation.
* The dataset is loaded using file paths and organized into DataFrames with class labels.
* Data augmentation techniques are applied to increase the diversity of the training data.
* A CNN model architecture is defined and compiled with appropriate optimization and loss functions.
* The model is trained using the training data and evaluated on the validation and test sets.
* Evaluation metrics such as accuracy, loss, confusion matrix, and classification report are computed.
* The learning curves are visualized to understand the model's performance over epochs.

# Data Description:

* The dataset used in this project consists of chest X-ray images for the classification of pneumonia and normal cases.
* The dataset is organized into three folders: train, test, and val.
* Within each folder, there are subfolders representing each image category: Pneumonia and Normal.
* The dataset contains a total of 5,863 X-ray images in JPEG format.
* The two main categories in the dataset are Pneumonia and Normal.
* The data was collected from retrospective cohorts of pediatric patients, aged one to five years old, from Guangzhou Women and Children’s Medical Center in Guangzhou.
* The chest X-ray images were obtained as part of routine clinical care for the patients.
* Prior to the analysis of the images, a quality control process was performed to remove low-quality or unreadable scans.
* The diagnoses for the images were then graded by two expert physicians before being used to train the AI system.
* To ensure the accuracy of the dataset, the evaluation set was checked by a third expert to account for any grading errors.
* The dataset used in this code is available from Mendeley Data (https://data.mendeley.com/datasets/rscbjbr9sj/2) and is licensed under CC BY 4.0.
* The dataset can be cited using the following reference: <https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia>

# Step by step explanation of code:

**1. Import the necessary libraries:**

```

import matplotlib.pyplot as plt

import seaborn as sns

import cv2

import os

import numpy as np

import pandas as pd

from PIL import Image

from glob import glob

import tensorflow as tf

```

* The code imports various libraries for data manipulation, visualization, image processing, and deep learning.

**2. Set the main path of the dataset:**

```

main\_path = "/kaggle/input/chest-xray-pneumonia/chest\_xray"

```

* The `main\_path` variable is set to the directory path containing the chest X-ray images.

**3. Load the dataset:**

```

train\_normal = glob(os.path.join(main\_path, "train", "NORMAL", "\*.jpeg"))

train\_pneumonia = glob(os.path.join(main\_path, "train", "PNEUMONIA", "\*.jpeg"))

test\_normal = glob(os.path.join(main\_path, "test", "NORMAL", "\*.jpeg"))

test\_pneumonia = glob(os.path.join(main\_path, "test", "PNEUMONIA", "\*.jpeg"))

val\_normal = glob(os.path.join(main\_path, "val", "NORMAL", "\*.jpeg"))

val\_pneumonia = glob(os.path.join(main\_path, "val", "PNEUMONIA", "\*.jpeg"))

df\_train = pd.DataFrame({

'class': ['Normal'] \* len(train\_normal) + ['Pneumonia'] \* len(train\_pneumonia),

'image': train\_normal + train\_pneumonia

})

df\_test = pd.DataFrame({

'class': ['Normal'] \* len(test\_normal) + ['Pneumonia'] \* len(test\_pneumonia),

'image': test\_normal + test\_pneumonia

})

df\_val = pd.DataFrame({

'class': ['Normal'] \* len(val\_normal) + ['Pneumonia'] \* len(val\_pneumonia),

'image': val\_normal + val\_pneumonia

})

```

* The code uses the `glob` function from the `glob` module to retrieve the file paths of the images in the training, testing, and validation sets.
* DataFrames (`df\_train`, `df\_test`, `df\_val`) are created to store the image paths and corresponding class labels.

**4. Concatenate the training and validation sets:**

```

df\_train = pd.concat([df\_train, df\_val], axis=0)

df\_train = df\_train.reset\_index(drop=True)

```

* The code concatenates the training and validation DataFrames to increase the training data.
* The index of the combined DataFrame is reset for proper indexing.

**5. Visualize the target distribution:**

```

sns.set\_palette("bright")

sns.set\_style("whitegrid")

plt.figure(figsize=(4, 3))

sns.countplot(data=df\_train, x='class')

plt.title('Target Distribution - Training Set')

plt.xlabel('Class')

plt.ylabel('Count')

plt.show()

plt.figure(figsize=(4, 3))

sns.countplot(data=df\_test, x='class')

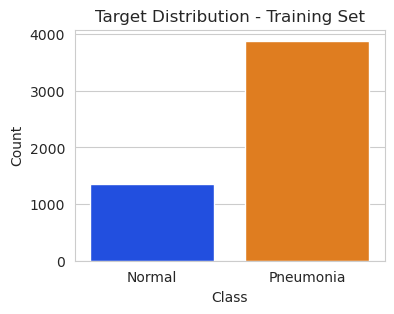
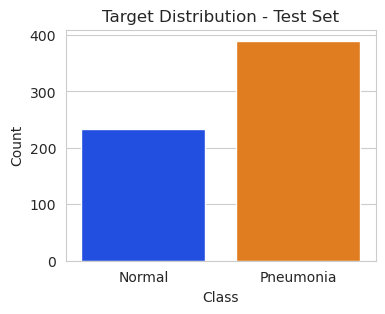
plt.title('Target Distribution - Test Set')

plt.xlabel('Class')

plt.ylabel('Count')

plt.show()

```

* The code uses seaborn's `countplot` to visualize the distribution of classes (Normal and Pneumonia) in the training and test sets.
*  

**6. Display example images:**

```

image\_path1 = df\_train['image'][1]

try:

image1 = Image.open(image\_path1)

plt.imshow(image1, cmap='gray')

plt.title('Normal Image')

plt.axis('off')

plt.show()

except (IOError, OSError) as e:

print(f"Error opening image: {e}")

image\_path2 = df\_train['image'][5110]

try:

image2 = Image.open(image\_path2)

plt.imshow(image2, cmap='gray')

plt.title('Pneumonia Image')

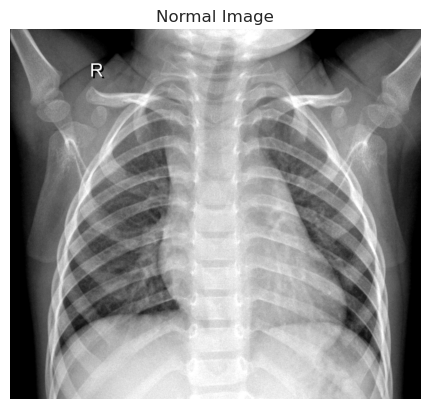
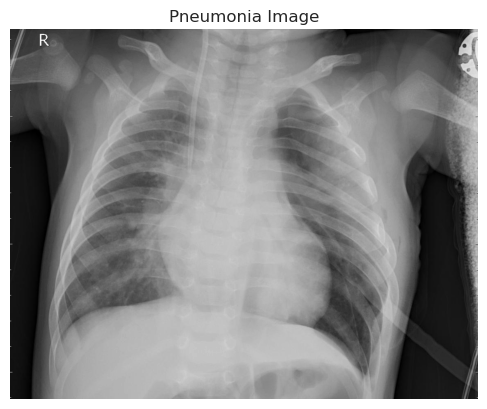
plt.axis('off')

plt.show()

except (IOError, OSError) as e:

print(f"Error opening image: {e}")

```

* The code selects two example images (one normal and one pneumonia) from the training set.
* It uses the PIL library to open and display the images using `imshow`.
*  

**7. Split the training set into train and validation sets:**

```

from sklearn.model\_selection import train\_test\_split

train\_df, val\_df = train\_test\_split(df\_train, test\_size=0.20, random\_state=42, stratify=df\_train['class'])

```

* The code uses `train\_test\_split` from sklearn.model\_selection to split the combined training set into train and validation sets.
* The split is performed based on a specified test size, random state, and stratification using the class labels.

**8. Create data generators:**

```

from tensorflow.keras.preprocessing.image import ImageDataGenerator

train\_datagen = ImageDataGenerator(

rescale=1./255,

shear\_range=0.2,

zoom\_range=0.2,

rotation\_range=0.1,

width\_shift\_range=0.1,

height\_shift\_range=0.1,

horizontal\_flip=True

)

val\_datagen = ImageDataGenerator(rescale=1./255)

```

* The code creates two `ImageDataGenerator` objects: `train\_datagen` and `val\_datagen`.
* The `train\_datagen` object performs data augmentation by rescaling pixel values, applying shear, zoom, rotation, shifting, and flipping operations.
* The `val\_datagen` object only rescales the pixel values.

**9. Generate data flows:**

```

ds\_train = train\_datagen.flow\_from\_dataframe(

dataframe=train\_df,

x\_col='image',

y\_col='class',

class\_mode='binary',

target\_size=(224, 224),

batch\_size=32

)

ds\_val = val\_datagen.flow\_from\_dataframe(

dataframe=val\_df,

x\_col='image',

y\_col='class',

class\_mode='binary',

target\_size=(224, 224),

batch\_size=32

)

ds\_test = val\_datagen.flow\_from\_dataframe(

dataframe=df\_test,

x\_col='image',

y\_col='class',

class\_mode='binary',

target\_size=(224, 224),

batch\_size=1

)

```

* The code generates data flows from the data frames using the `flow\_from\_dataframe` method of the data generators.
* The `ds\_train`, `ds\_val`, and `ds\_test` objects contain the batches of data with specified target sizes, batch sizes, and class modes.

**10. Define the model architecture:**

```

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, BatchNormalization, MaxPooling2D, Flatten, Dense, Dropout

def create\_model():

model = Sequential()

model.add(Conv2D(32, (3, 3), strides=1, padding='valid', activation='relu', input\_shape=(224, 224, 3)))

model.add(BatchNormalization())

model.add(MaxPooling2D((2, 2), strides=2, padding='valid'))

model.add(Conv2D(64, (3, 3), strides=1, padding='valid', activation='relu'))

model.add(Dropout(0.1))

model.add(BatchNormalization())

model.add(MaxPooling2D((2, 2), strides=2, padding='valid'))

model.add(Conv2D(64, (3, 3), strides=1, padding='valid', activation='relu'))

model.add(BatchNormalization())

model.add(MaxPooling2D((2, 2), strides=2, padding='valid'))

model.add(Conv2D(128, (3, 3), strides=1, padding='valid', activation='relu'))

model.add(Dropout(0.2))

model.add(BatchNormalization())

model.add(MaxPooling2D((2, 2), strides=2, padding='valid'))

model.add(Conv2D(256, (3, 3), strides=1, padding='valid', activation='relu'))

model.add(Dropout(0.2))

model.add(BatchNormalization())

model.add(MaxPooling2D((2, 2), strides=2, padding='valid'))

model.add(Flatten())

model.add(Dense(units=256, activation='relu'))

model.add(Dropout(0.2))

model.add(Dense(units=128, activation='relu'))

model.add(Dropout(0.2))

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

return model

model = create\_model()

```

* The code defines a convolutional neural network (CNN) model using the `Sequential` class from `tensorflow.keras.models`.
* The model architecture includes several convolutional layers, batch normalization, max pooling, dropout, and dense layers.
* The output layer uses sigmoid activation for binary classification.

**11. Compile the model:**

```

from tensorflow.keras.optimizers import Adam

model.compile(

optimizer=Adam(learning\_rate=3e-5),

loss='binary\_crossentropy',

metrics=['binary\_accuracy']

)

```

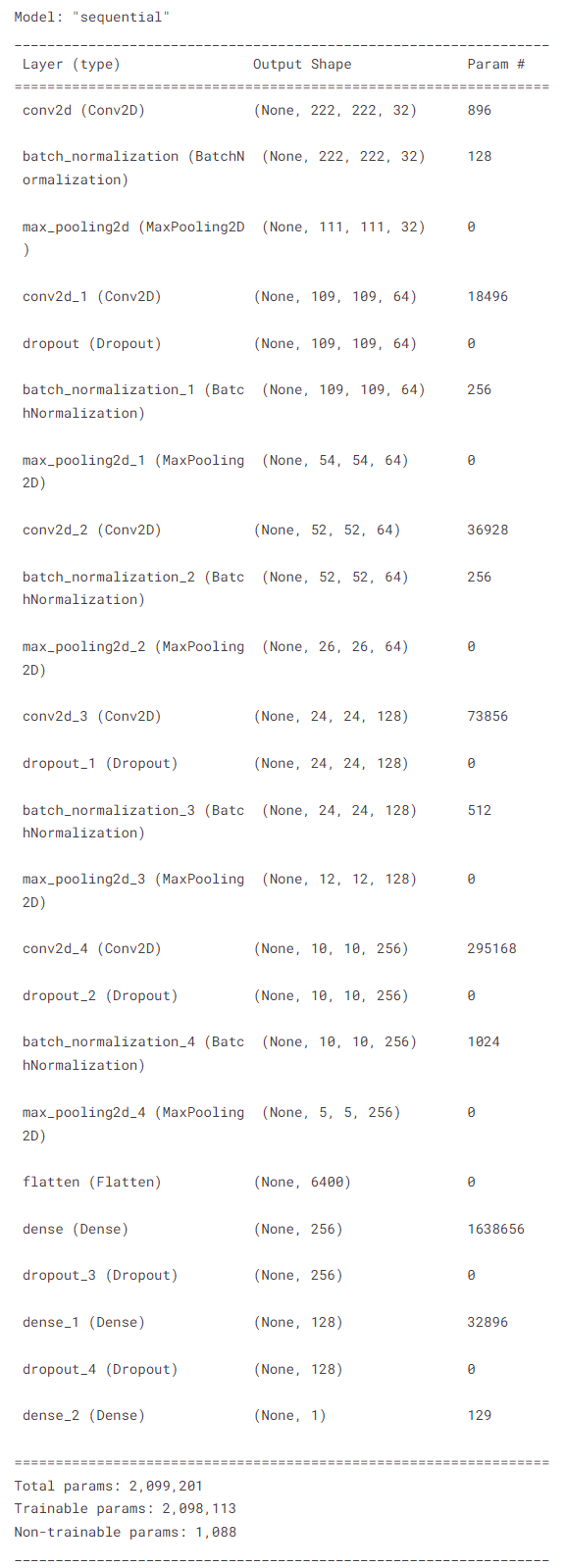
* The code compiles the model using the Adam optimizer with a specific learning rate, binary cross-entropy loss, and binary accuracy metric.

**12. Print the model summary:**

```

model.summary()

```

* The code prints a summary of the model's architecture, including the layer types, output shapes, and the number of parameters.
* 

**13. Set up callbacks for training:**

```

from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, min\_delta=1e-7, restore\_best\_weights=True)

plateau = ReduceLROnPlateau(monitor='val\_loss', factor=0.2, patience=2, min\_delta=1e-7, cooldown=0, verbose=1)

```

* The code defines two callbacks: EarlyStopping and ReduceLROnPlateau.
* EarlyStopping stops the training process if the validation loss does not improve for a certain number of epochs.
* ReduceLROnPlateau reduces the learning rate if the validation loss plateaus.

**14. Train the model:**

```

history = model.fit(

ds\_train,

batch\_size=32,

epochs=50,

validation\_data=ds\_val,

callbacks=[early\_stopping, plateau],

steps\_per\_epoch=(len(train\_df) / 32),

validation\_steps=(len(val\_df) / 32)

)

```

* The code trains the model on the training data using the `fit` method.
* It specifies the training and validation data flows, batch size, number of epochs, and includes the defined callbacks.
* The `steps\_per\_epoch` and `validation\_steps` parameters ensure that the model processes the entire dataset in each epoch.

**15. Visualize the learning curves:**

```

plt.plot(history.history['binary\_accuracy'], color='red', label='Train')

plt.plot(history.history['val\_binary\_accuracy'], color='blue', label='Validation')

plt.legend()

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.title('Learning Curve (Accuracy)')

plt.show()

plt.plot(history.history['loss'], color='red', label='Train')

plt.plot(history.history['val\_loss'], color='blue', label='Validation')

plt.legend()

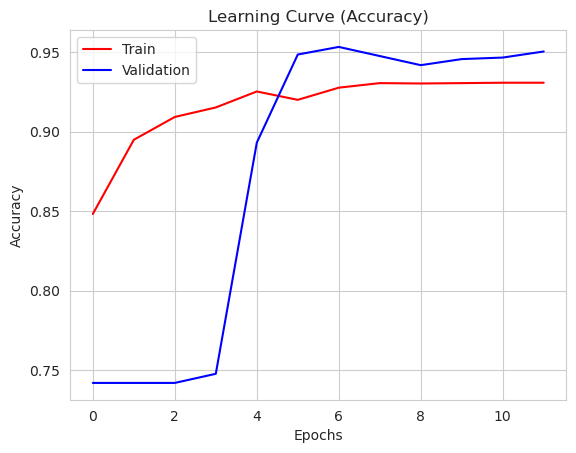
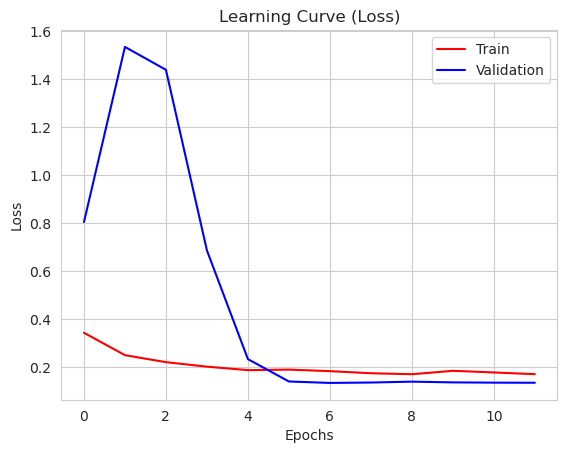
plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Learning Curve (Loss)')

plt.show()

```

* The code plots the training and validation accuracy and loss values over epochs using `matplotlib.pyplot`.
* 
* 

**16. Evaluate the model:**

```

val\_loss, val\_accuracy = model.evaluate(ds\_val, steps=int(len(ds\_val) / 32), verbose=0)

test\_loss, test\_accuracy = model.evaluate(ds\_test, steps=len(ds\_test), verbose=0)

```

* The code evaluates the model's performance on the validation and test datasets using the `evaluate` method on the respective data flows.
* The evaluation results are stored in `val\_loss`, `val\_accuracy`, `test\_loss`, and `test\_accuracy` variables.

**17. Get predictions and evaluate performance:**

```

import numpy as np

from sklearn.metrics import confusion\_matrix, classification\_report

test\_labels = ds\_test.labels

predicted\_labels = model.predict(ds\_test)

predicted\_labels = np.round(predicted\_labels).flatten()

cm = confusion\_matrix(test\_labels, predicted\_labels)

classification\_rep = classification\_report(test\_labels, predicted\_labels)

print("Validation Loss:", val\_loss)

print("Validation Accuracy:", val\_accuracy)

print("Test Loss:", test\_loss)

print("Test Accuracy:", test\_accuracy)

print("Confusion Matrix:")

print(cm)

print("Classification Report:")

print(classification\_rep)

```

* The code retrieves the true labels and predicted labels from the test data flow.
* It flattens the predicted labels to a 1D array.
* The code calculates the confusion matrix and classification report using functions from sklearn.metrics.
* Finally, it prints the evaluation metrics, including validation loss, validation accuracy, test loss, test accuracy, confusion matrix, and classification report.
* 