# Deep Learning-Based Pneumonia Detection: A Convolutional Neural Network Approach

A Project

Presented to
The Faculty of the Department of Computer Science
Aligarh Muslim University (Aligarh)

In Partial Fulfillment of
the Requirements for Sessional Marks
Master of Science (Data Science)

By
Aamir Suhail

March 2025

**APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE**

---

Dr. Zahid Ahmed Mohammed Husain Ansari, Department of Computer Science

# ACKNOWLEDGEMENT

I would like to express my gratitude to my project advisor Dr. Zahid Ahmad Ansari for his support and guidance. I would not have been able to complete this project without Dr. Zahid sir's valuable suggestions.

I am also thankful to my friends and family for all the moral support and constant encouragement.

# ABSTRACT

Pneumonia is a serious lung infection that affects millions of people worldwide. Early detection is crucial for timely treatment and to prevent severe complications. Traditionally, doctors diagnose pneumonia by analyzing chest X-ray images, but this process can be time-consuming and prone to human error. To improve accuracy and efficiency, automated methods using deep learning have been explored in recent years.

In this project, we develop a deep learning model based on Convolutional Neural Networks (CNNs) to automatically detect pneumonia from chest X-ray images. CNNs are widely used for image classification tasks due to their ability to learn spatial features effectively. Our model is trained on a dataset of labeled chest X-ray images and learns to distinguish between normal and pneumonia-affected lungs. By processing these medical images, the model can identify pneumonia cases with high accuracy.

The proposed system aims to assist radiologists and healthcare professionals by providing a fast and reliable diagnosis. It reduces the dependency on manual interpretation, thus minimizing errors and enhancing the decision-making process. The experimental results indicate that our CNN model achieves promising performance in detecting pneumonia, making it a valuable tool for medical image analysis.

Additionally, this project highlights the potential of artificial intelligence in healthcare applications. With further improvements, such as larger datasets and model fine-tuning, this deep learning-based approach can be integrated into hospital systems to provide real-time pneumonia detection, helping doctors diagnose and treat patients more efficiently.

# TABLE OF CONTENTS

# 1 Introduction

## 1.1 Background

Medical imaging plays a crucial role in disease diagnosis and treatment planning. In recent years, deep learning techniques, particularly Convolutional Neural Networks (CNNs), have demonstrated remarkable success in medical image analysis. One such application is the automated detection of pneumonia from chest X-ray images, which can aid in early diagnosis and improve patient outcomes.

## 1.2 Problem Statement

Pneumonia is a severe lung infection that can lead to significant morbidity and mortality if not diagnosed and treated promptly. Traditional diagnostic methods rely on manual interpretation of chest X-rays, which can be time-consuming and prone to variability among radiologists. Automating pneumonia detection using CNNs provides a scalable and efficient solution to assist healthcare professionals.

## 1.3 Objective

The objective of this project is to develop a deep learning model based on CNNs to classify chest X-ray images into Pneumonia and Normal categories. The study aims to:

- Preprocess and augment the dataset for optimal model performance.
- Design and implement a CNN architecture tailored for medical image classification.
- Evaluate the model's performance using various metrics such as accuracy, precision, recall, and F1-score.
- Compare the model's performance with existing approaches in the literature.

## 1.4 Contributions

This project contributes to the field of medical image analysis by:

- Implementing a CNN-based approach for pneumonia detection.
- Providing insights into the effectiveness of deep learning in radiographic diagnosis.
- Highlighting the significance of preprocessing techniques in improving classification accuracy.

# 2 Dataset Description and Methodology

## 2.1 Dataset Overview

The dataset used in this study is the **Chest X-Ray Images (Pneumonia)** dataset, which consists of **5,863 chest X-ray images** labeled into two categories:

- **Normal**: X-ray scans of healthy individuals.
- **Pneumonia**: X-ray scans showing signs of bacterial or viral pneumonia.

The dataset was collected from pediatric patients aged between one and five years from the *Guangzhou Women and Children's Medical Center, China*. It was curated and annotated by expert radiologists to ensure high-quality labeling.

The dataset is structured into three subsets:

- **Training Set**: Used to train the CNN model.
- **Validation Set**: Helps in tuning hyperparameters and avoiding overfitting.
- **Test Set**: Used to evaluate the final model's performance on unseen data.

## 2.2 Preprocessing Steps

Before training the CNN model, several preprocessing steps were performed to enhance data quality and model performance:

- **Data Cleaning**: Removal of low-quality or unreadable X-ray images.
- **Image Resizing**: Standardizing image dimensions to $224 \times 224$ *pixels* to fit the CNN input layer.
- **Normalization**: Scaling pixel values to a range of [0,1] to improve training efficiency.
- **Data Augmentation**: Applying transformations such as rotation, flipping, and contrast adjustments to increase dataset diversity and reduce overfitting.

## 2.3 Experimental Setup

The methodology involves designing and training a deep learning model to classify X-ray images into Normal and Pneumonia classes.

### 2.3.1 Convolutional Neural Network (CNN) Architecture

A CNN model is implemented with the following layers:

- **Convolutional Layers**: Extract spatial features using multiple filters.
- **Pooling Layers**: Reduce dimensionality and retain essential features.
- **Fully Connected Layers**: Learn high-level representations.
- **Softmax Output Layer**: Classifies images into two categories.

### 2.3.2 Training Procedure

- **Loss Function**: Binary Cross-Entropy is used to measure classification error.
- **Optimizer**: Adam optimizer is employed for weight updates.
- **Batch Size**: Set to 32 for efficient gradient updates.
- **Number of Epochs**: Model is trained for 50 epochs with early stopping.
- **Evaluation Metrics**: Accuracy, Precision, Recall, and F1-score are used to assess model performance.

# 3 Understanding Convolutional Neural Networks (CNN)

## 3.1 What is CNN:

CNN is a type of deep learning model for processing data that has a grid pattern, such as images, which is inspired by the organization of animal visual cortex [13, 14] and designed to automatically and adaptively learn spatial hierarchies of fea- tures, from low- to high-level patterns. CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers. The first two, convolution and pooling layers, perform feature extraction, whereas the third, a fully connected layer, maps the extracted features into final output, such as classifi- cation. A convolution layer plays a key role in CNN, which is composed of a stack of mathematical operations, such as convolution, a specialized type of linear operation. In dig- ital images, pixel values are stored in a two-dimensional (2D) grid, i.e., an array of numbers (Fig. 2), and a small grid of parameters called kernel, an optimizable feature ex- tractor, is applied at each image position, which makes CNNs highly efficient for image processing, since a feature may occur anywhere in the image. As one layer feeds its output into the next layer, extracted features can hierarchi- cally and progressively become more complex. The process of optimizing parameters such as kernels is called training, which is performed so as to minimize the difference be- tween outputs and ground truth labels through an optimiza- tion algorithm called backpropagation

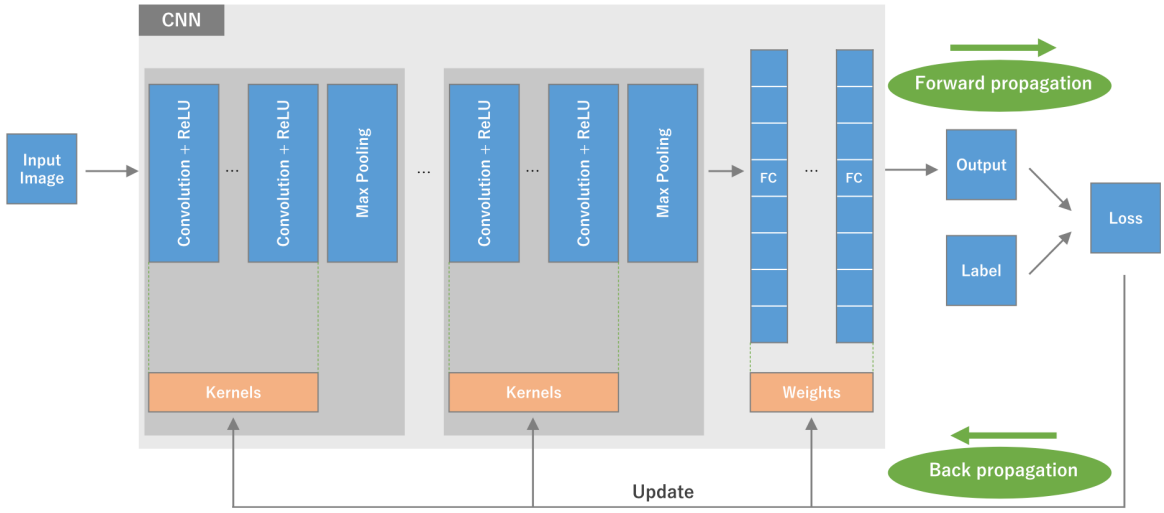


Figure 1: An overview of a convolutional neural network (CNN) architecture and the training process. A CNN is composed of a stacking of several building blocks: convolution layers, pooling layers (e.g., max pooling), and fully connected (FC) layers. A model's performance under particular kernels and weights is calculated with a loss function through forward propagation on a training dataset, and learnable parameters, i.e., kernels and weights, are updated according to the loss value through backpropagation with gradient descent optimization algorithm. ReLU, rectified linear unit

## 3.2 Convolution

Convolution is a specialized type of linear operation used for feature extraction, where a small array of numbers, called a kernel, is applied across the input, which is an array of num- bers, called a tensor. An element-wise product between each element of the kernel and the input tensor is calculated at each location of the tensor and summed to obtain the output value in the corresponding position of the output tensor, called a feature map.A convolution layer is a fundamental component of the CNN architecture that performs feature extraction, which typically consists of a combination of linear and nonlinear operations, i. e., convolution operation and activation function.

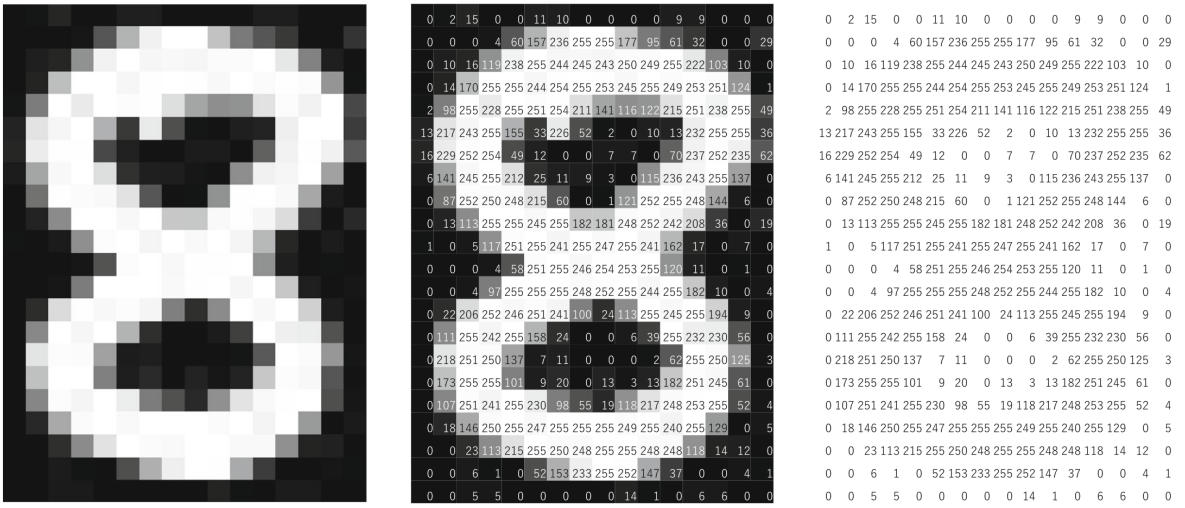

Figure 2: A computer sees an image as an array of numbers. The matrix on the right contains numbers between 0 and 255, each of which corresponds to the pixel brightness in the left image. Both are overlaid in the middle image. The source image was downloaded via http://yann.lecun.com/exdb/mnist

## 3.3 Encoder-Decoder CNN

An encoder in a neural network is a part of the model that compresses the input data into a smaller, more meaningful representation. It extracts the most important features while reducing unnecessary details.

### Convolution Operation in Encoder

The convolution operation is mathematically represented as:

$$Z(i,j) = \sum_m \sum_n X(i-m, j-n) \cdot K(m,n) + b$$

where:

- $Z(i, j) \rightarrow$ Output feature map at position $(i, j)$
- $X(i, j) \rightarrow$ Input image
- $K(m, n) \rightarrow$ Convolution filter (kernel)
- $b \rightarrow$ Bias term
- $m, n \rightarrow$ Kernel dimensions

**Decoder in CNN**

The decoder reconstructs the original input or generates a meaningful representation from the encoded features. It primarily uses upsampling, transposed convolution, or deconvolution operations.

Mathematically, the transposed convolution operation can be expressed as:

$$Y(i, j) = \sum_{m} \sum_{n} Z(i + m, j + n) \cdot K(m, n) + b$$

where:

- $Y(i, j) \rightarrow$ Reconstructed output at position $(i, j)$
- $Z(i, j) \rightarrow$ Input feature map from the encoder
- $K(m, n) \rightarrow$ Transposed convolution filter (kernel)
- $b \rightarrow$ Bias term
- $m, n \rightarrow$ Kernel dimensions

## 3.4 Nonlinear activation function

The outputs of a linear operation such as convolution are then passed through a nonlinear activation function. Although smooth nonlinear functions, such as sigmoid or hyperbolic tangent (tanh) function, were used previously because they are mathematical representations of a biological neuron behavior, the most common nonlinear activation function used presently is the rectified linear unit (ReLU), which simply computes the function:

# 4 Activation function

**ReLU (Rectified Linear Unit)**

$$f(x) = \max(0, x) \tag{1}$$

### 4.1 Leaky ReLU

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases} \tag{2}$$

### 4.2 Sigmoid Activation

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

### 4.3 Tanh Activation

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4}$$

### 4.4 Softmax Activation (for Multi-class Classification)

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{5}$$

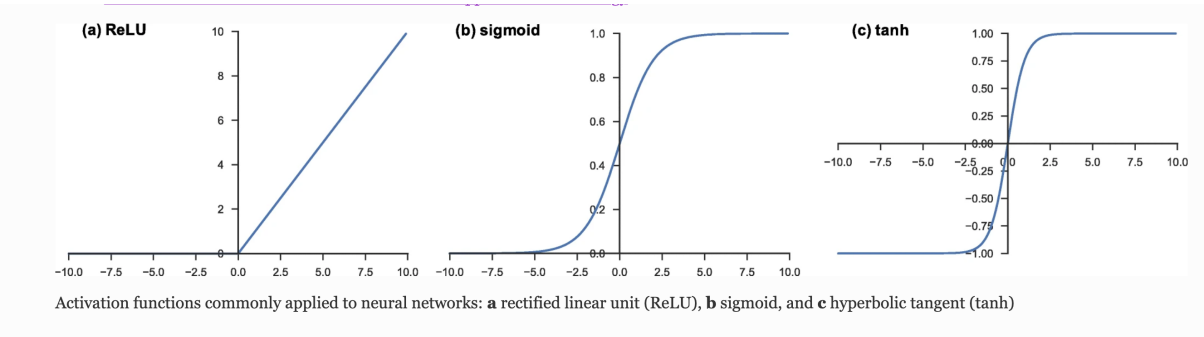


Figure 3: Activation function

## 5 Pooling layer

A pooling layer provides a typical downsampling operation which reduces the in-plane dimensionality of the feature maps in order to introduce a translation invariance to small shifts and distortions, and decrease the number of subsequent learnable parameters. It is of note that there is no learnable parameter in any of the pooling layers, whereas filter size, stride, and padding are hyperparameters in pooling operations, similar to convolution operations.

**Max pooling**

The most popular form of pooling operation is max pooling, which extracts patches from the input feature maps, outputs the maximum value in each patch, and discards all the other values (Fig.4). A max pooling with a filter of size $2 \times 2$ with a stride of 2 is commonly used in practice. This downsamples the in-plane dimension of feature maps by a factor of 2. Unlike height and width, the depth dimension of feature maps remains unchanged.

$$Y(i,j) = \max_{m,n} X(2i+m, 2j+n)$$

where:

- $X(i,j)$ represents the input feature map.
- $Y(i,j)$ represents the output feature map after max pooling.
- $m, n$ denote the pooling window dimensions, typically $2 \times 2$.
- The function selects the **maximum** value within each pooling region.

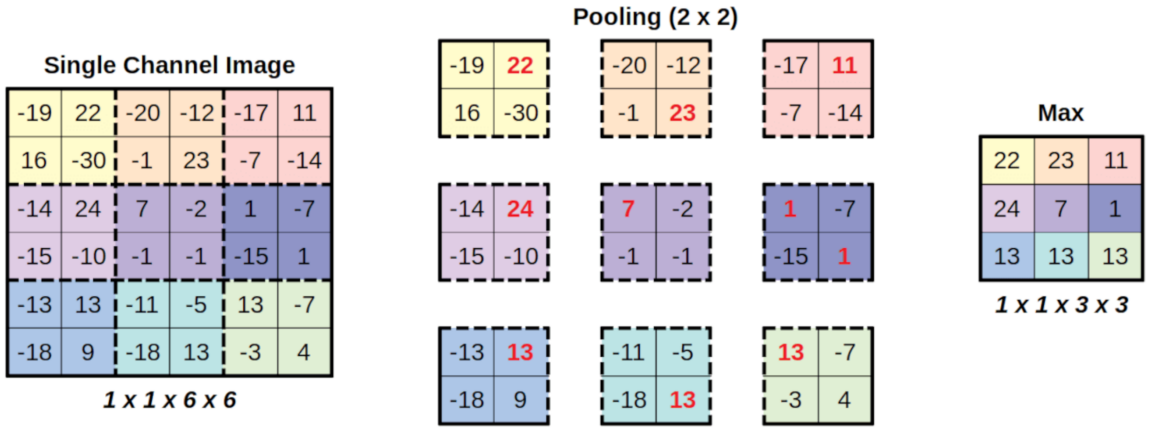


Figure 4: Max-Pooling

# 6 Batch Size

Batch size refers to the number of training samples processed together before the model's internal parameters are updated. It affects the model's learning process, computational efficiency, and convergence behavior

**Mathematical Formula**

$$I = \frac{N}{B}$$

where:

- $I$ = Number of iterations per epoch
- $N$ = Total number of training samples
- $B$ = Batch size (number of samples per batch)

# 7 Fully connected layer

The output feature maps of the final convolution or pooling layer is typically flattened, i.e., transformed into a one-dimensional (1D) array of numbers (or vector), and connected to one or more fully connected layers, also known as dense layers, in which every input is connected to every output by a learnable weight. Once the features extracted by the convolution layers and downsampled by the pooling layers are created, they are mapped by a subset of fully connected layers to the final outputs of the network, such as the probabilities for each class in classification tasks. The final fully connected layer typically has the same number of output nodes as the number of classes. Each fully connected layer is followed by a nonlinear function, such as ReLU, as described above.

# 8 Padding in CNN

Padding in Convolutional Neural Networks (CNNs) refers to adding extra pixels (usually zeros) around the input image to control the spatial dimensions of the output feature map. It helps preserve important edge features and maintain dimensional consistency.

## Mathematical Formula

The output size of a convolutional layer with padding is given by:

$$O = \frac{(I - K + 2P)}{S} + 1$$

where:

- $O$ = Output dimension (height/width)
- $I$ = Input dimension (height/width)
- $K$ = Kernel (filter) size

- $P$ = Padding size
- $S$ = Stride size

### Types of Padding

1. **Same Padding (Zero Padding)** – Ensures output size remains the same as input ($P = \frac{K-1}{2}$).
2. **Valid Padding** – No padding is applied, reducing the output size.

## 9 Epoch

An **epoch** in deep learning refers to one complete cycle of training where the entire dataset is passed forward and backward through the neural network. It is a crucial parameter in training machine learning models as it determines how many times the learning algorithm will see the complete dataset.

### Mathematical Representation

The total number of training iterations can be represented as:

$$T = E \times \frac{N}{B}$$

where:

- $T$ = Total training iterations (updates of weights)
- $E$ = Number of epochs
- $N$ = Total number of training samples
- $B$ = Batch size

Each epoch consists of multiple iterations, where an iteration refers to processing one batch of data.

### Significance of Epochs

- A higher number of epochs allows the model to learn more patterns but may lead to overfitting. - A lower number of epochs might result in underfitting if the model hasn't learned enough. - The optimal number of epochs is determined using validation loss and early stopping techniques.

# 10 Batch Normalization

Batch Normalization (BN) is a technique used in deep learning to standardize the inputs to a layer for each mini-batch, improving training speed and stability. It helps in reducing internal covariate shift and allows for using higher learning rates.

## Mathematical Representation

Given an input mini-batch $X = \{x_1, x_2, ..., x_m\}$, the batch normalization transformation is computed as:

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

where:

- $\mu_B$ = Mean of the mini-batch
- $\sigma_B^2$ = Variance of the mini-batch
- $\hat{x}_i$ = Normalized input
- $\epsilon$ = Small constant for numerical stability
- $\gamma$ and $\beta$ = Learnable scaling and shifting parameters
- $y_i$ = Output of batch normalization

## Significance of Batch Normalization

- Reduces internal covariate shift. - Allows for faster training and higher learning rates. - Acts as a form of regularization, reducing dependency on dropout. - Helps in stabilizing deep networks.

# 11 Loss Function in Convolutional Neural Networks (CNNs)

In Convolutional Neural Networks (CNNs), the loss function quantifies the difference between the predicted output and the actual target value. The goal of training a CNN is to minimize this loss by adjusting the network's parameters using optimization algorithms such as gradient descent.

A commonly used loss function in CNNs for classification tasks is the **Cross-Entropy Loss**, defined as:

$$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where:

- $N$ is the total number of classes.
- $y_i$ is the true label (ground truth) for class $i$.
- $\hat{y}_i$ is the predicted probability for class $i$.
- The negative summation ensures that the loss is minimized when the predicted probability is close to the actual class probability.

For **regression tasks**, CNNs typically use **Mean Squared Error (MSE)** as the loss function:

$$L = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

where:

- $y_i$ is the actual value.
- $\hat{y}_i$ is the predicted value.
- The squared difference penalizes larger errors more than smaller ones.

The loss function guides the backpropagation process, helping the CNN learn better representations by minimizing errors during training.

# 12 Optimizer in CNN

An **optimizer** in deep learning is an algorithm or method used to update the parameters of a neural network in order to minimize the loss function. It plays a crucial role in training a model efficiently by adjusting weights and biases through iterative steps.

### Mathematical Representation

At each iteration $t$, the parameters $\theta_t$ (such as weights) are updated using the following general optimization formula:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

where:

- $\theta_t$ = Parameters (weights, biases) at iteration $t$
- $J(\theta)$ = Loss function
- $\nabla_\theta J(\theta_t)$ = Gradient of the loss function with respect to $\theta$
- $\eta$ = Learning rate, which controls the step size of updates

### Common Optimizers in CNN

Some widely used optimizers include:

- **Gradient Descent (GD)**
- **Stochastic Gradient Descent (SGD)**
- **Momentum-based SGD**
- **RMSprop**
- **Adam (Adaptive Moment Estimation)**
- **Adagrad and Adadelta**

Each optimizer has its own advantages and is chosen based on the problem being solved and the dataset characteristics.

## 13 Gradient descent

Gradient descent is commonly used as an optimization algorithm that iteratively updates the learnable parameters, i.e., kernels and weights, of the network so as to minimize the loss. The gradient of the loss function provides us the direction in which the function has the steepest rate of increase, and each learnable parameter is updated in the negative direction of the gradient with an arbitrary step size determined based on a hyperparameter called learning rate (Fig. 7).

The gradient is, mathematically, a partial derivative of the loss with respect to each learnable parameter, and a single update of a parameter is formulated as follows:

$$w := w - \alpha * \frac{\partial L}{\partial w}$$

where: - $w$ is the weight, - $\alpha$ is the learning rate, - $\frac{\partial L}{\partial w}$ is the gradient of the loss function $L$ with respect to $w$.

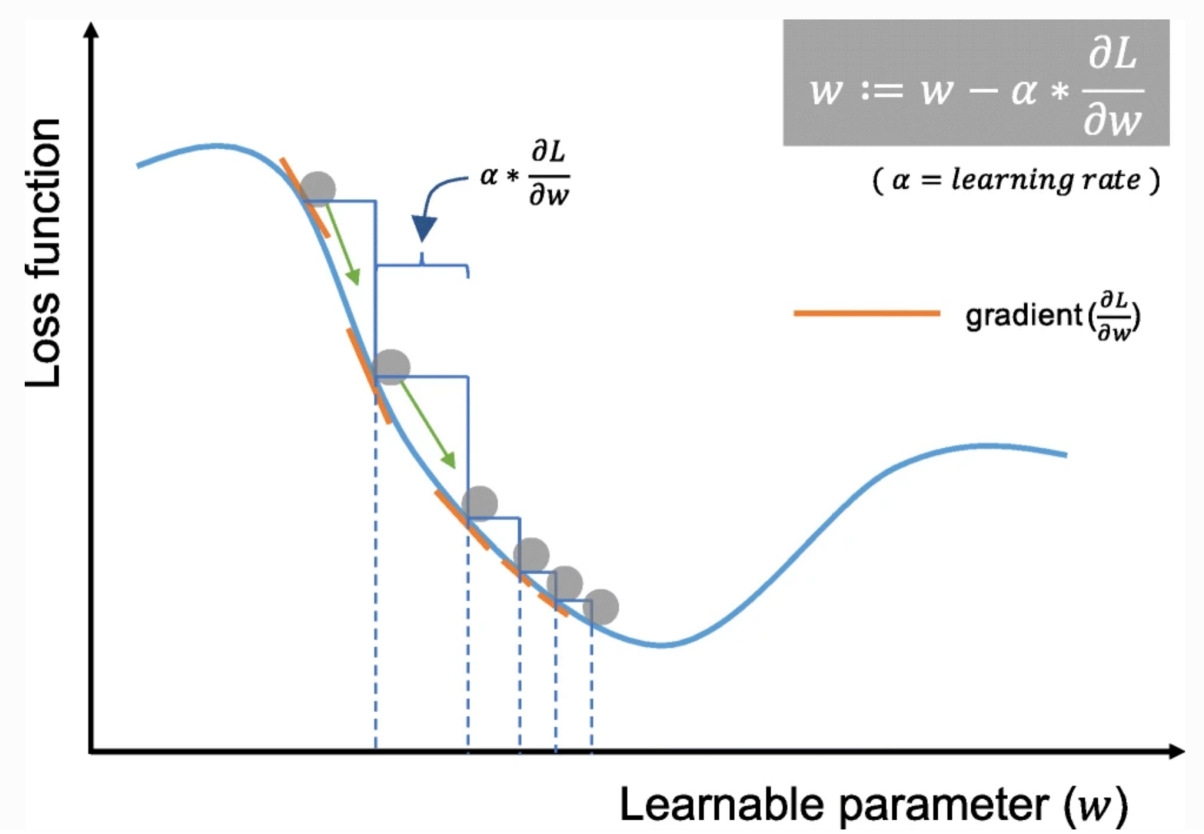


Figure 5: Gradient descent

# 14 Split Dataset

Available data are typically split into three sets: a training, a validation, and a test set. A training set is used to train a network, where loss values are calculated via forward propagation and learnable parameters are updated via backpropagation. A validation set is used to monitor the model performance during the training process, fine-tune hyperparameters, and perform model selection. A test set is ideally used only once at the very end of the project in order to evaluate the performance of the final model that is fine-tuned and selected on the training process with training and validation sets.

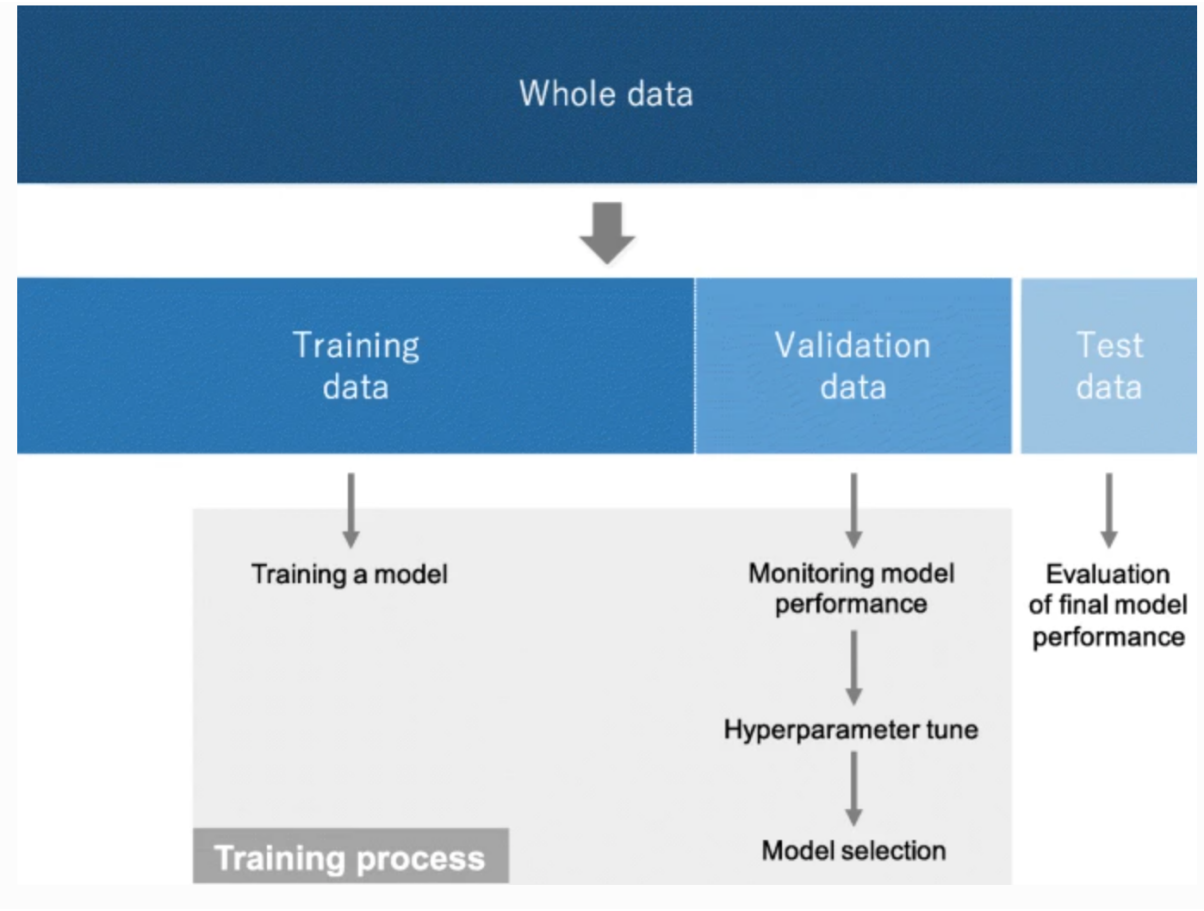


Figure 6: process

# 15 Dropout Regularization

Dropout is a regularization technique used in deep learning to prevent overfitting by randomly setting a fraction of input units to zero during training. This forces the network to learn redundant representations and enhances generalization.

Mathematically, dropout can be represented as:

$$\tilde{h}^{(l)} = D^{(l)} \odot h^{(l)}$$

where: - $h^{(l)}$ is the activation from the previous layer, - $D^{(l)}$ is a binary mask with elements drawn from a Bernoulli distribution $D^{(l)} \sim Bernoulli(p)$, - $p$ is the probability of keeping a unit active, - $\tilde{h}^{(l)}$ is the modified activation after applying dropout, - $\odot$ denotes element-wise multiplication.

During inference, the activations are scaled by $p$ to maintain the expected value of activations.

# 16 overfitting

Overfitting refers to a situation where a model learns statistical regularities specific to the training set, i.e., ends up memorizing the irrelevant noise instead of learning the signal, and, therefore, performs less well on a subsequent new dataset. This is one of the main challenges in machine learning, as an overfitted model is not generalizable to never-seen-before data. In that sense, a test set plays a pivotal role in the proper performance evaluation of machine learning models, as discussed in the previous section. A routine check for recognizing overfitting to the training data is to monitor the loss and accuracy on the training and validation sets (Fig. 7). If the model performs well on the training set compared to the validation set, then the model has likely been overfit to the training data. There have been several methods proposed to minimize overfitting . The best solution for reducing overfitting is to obtain more training data. A model trained on a larger dataset typically generalizes better, though that is not always attainable in medical imaging. The other solutions include regularization with dropout or weight decay, batch normalization, and data augmentation, as well as reducing architectural complexity. Dropout is a recently introduced regularization technique where randomly selected activations are set to 0 during the training, so that the model becomes less sensitive to specific weights in the network [27]. Weight decay, also referred to as L2 regularization, reduces overfitting by penalizing the model's weights so that the weights take only small values. Batch normalization is a type of supplemental layer which adaptively normalizes the input values of the following layer, mitigating the risk of overfitting, as well as improving gradient flow through the network, allowing higher learning rates, and reducing the dependence on initialization [28]. Data augmentation is also effective for the reduction of overfitting, which is a process of modifying the training data through random transformations, such as flipping, translation, cropping, rotating, and random erasing, so that the model will not see exactly the same inputs during the training iterations [29]. In spite of these efforts, there is still a concern of overfitting to the validation set rather than to the training set because of information leakage during the hyperparameter fine-tuning and model selection process. Therefore, reporting the performance of the final model on a separate (unseen) test set, and ideally on external validation datasets if applicable, is crucial for verifying the model generalizability.

# 17 Model Evaluation in CNN

Model evaluation is the process of assessing the performance of a trained Convolutional Neural Network (CNN) on unseen data. It helps in determining how well the model generalizes and ensures that it is not overfitting or underfitting.

The evaluation of a CNN model is commonly performed using the following metrics:

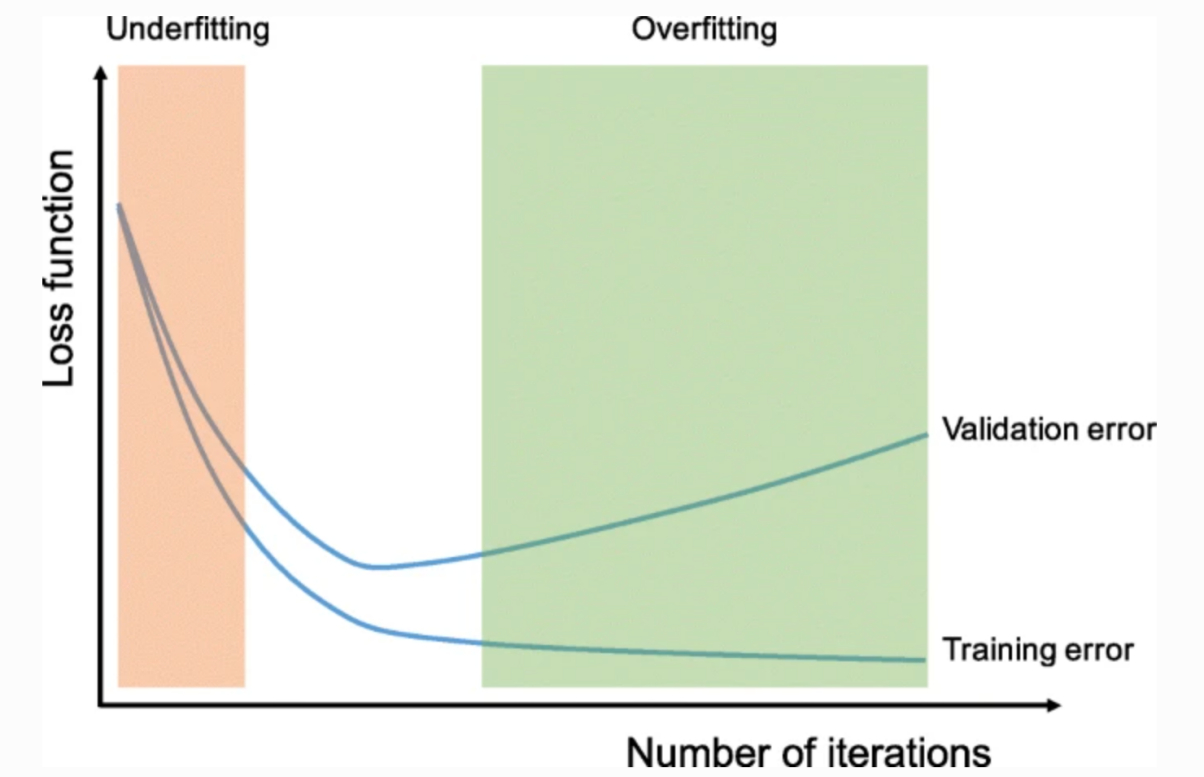


Figure 7: A routine check for recognizing overfitting is to monitor the loss on the training and validation sets during the training iteration. If the model performs well on the training set compared to the validation set, then the model has been overfit to the training data. If the model performs poorly on both training and validation sets, then the model has been underfit to the data. Although the longer a network is trained, the better it performs on the training set, at some point, the network fits too well to the training data and loses its capability to generalize

## 17.1 Accuracy

Accuracy measures the proportion of correctly classified samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

where $TP$ (True Positive), $TN$ (True Negative), $FP$ (False Positive), and $FN$ (False Negative) are elements of the confusion matrix.

## 17.2 Loss Function

The loss function quantifies the difference between predicted and actual values. A common loss function for classification is the categorical cross-entropy:

$$L = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

where $y_i$ is the true label and $\hat{y}_i$ is the predicted probability for class $i$.

## 17.3 Precision, Recall, and F1-score

- **Precision**: Measures how many of the predicted positive instances are actually correct.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (Sensitivity)**: Measures how many actual positive instances were correctly predicted.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-score**: The harmonic mean of precision and recall.

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

## 17.4 ROC Curve and AUC

The Receiver Operating Characteristic (ROC) curve illustrates the trade-off between sensitivity and specificity. The Area Under the Curve (AUC) quantifies the overall model performance, with a higher AUC indicating better classification capability.

## 17.5 Confusion Matrix

The confusion matrix provides a summary of correct and incorrect predictions, helping to analyze model errors in detail.

Model evaluation is crucial for tuning hyperparameters, selecting the best model, and improving the overall performance of CNN-based classification tasks.

# 4. Exploratory Data Analysis (EDA)

## 4.1 Overview of X-ray Image Analysis

Medical imaging plays a crucial role in diagnosing pneumonia, particularly through chest X-rays. The dataset used in this study contains X-ray images categorized into two classes: Normal and Pneumonia. The goal of this section is to analyze the structural differences between these images and highlight key visual characteristics that assist in classification.

Chest X-rays of pneumonia-affected individuals often show white patches or opacities, indicating fluid accumulation or infection in the lungs. In contrast, normal X-ray images exhibit clear lung structures with no abnormal opacities. Understanding these differences is essential for training a deep learning model to distinguish between the two classes accurately.

## 4.2 Visualization of X-ray Images

To better understand the dataset, we visualize sample images representing both normal and pneumonia cases. This helps in identifying distinguishing patterns that CNN models can learn for automated classification.

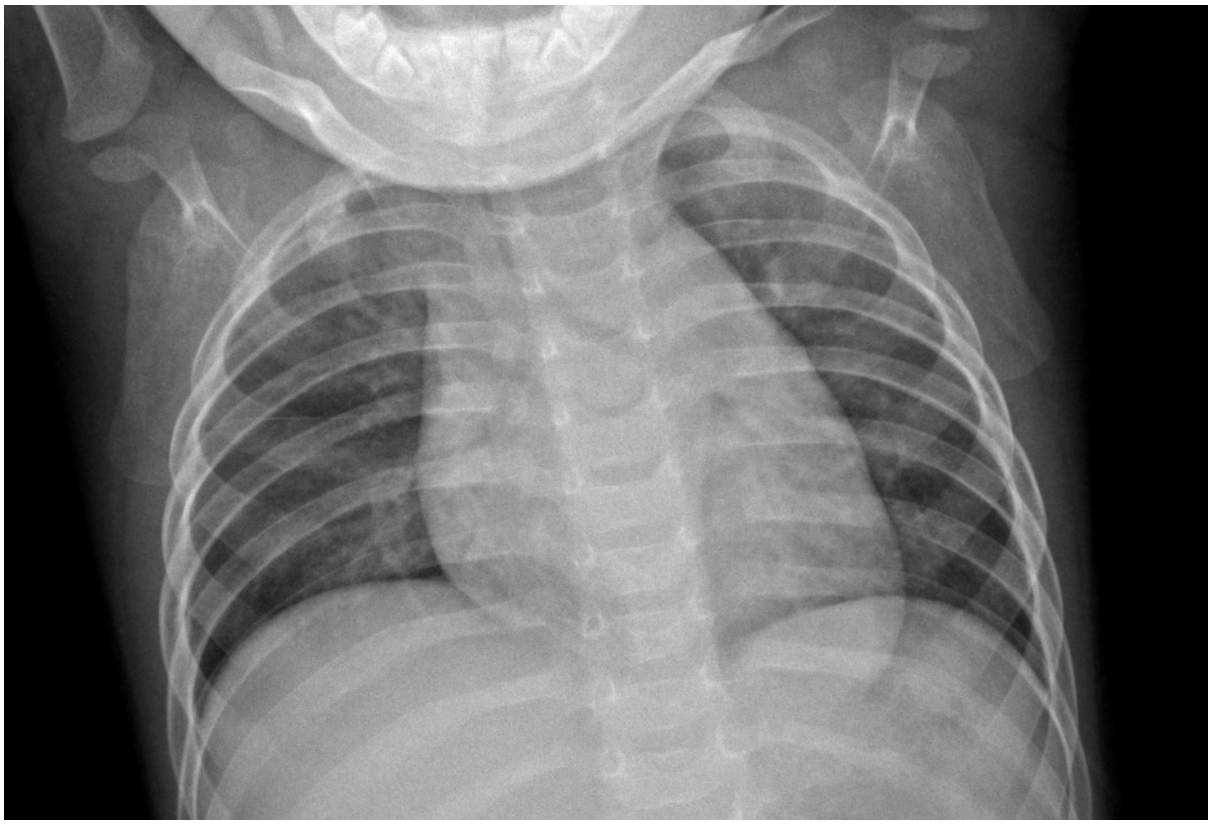

Figure 8: Normal image

## 4.3 Visual Inspection (Human Interpretation)

- **Clear lung fields:** In a normal chest X-ray, the lung fields appear **dark (radiolucent)** because they contain air.
- **No opacities or consolidations:** Pneumonia-affected X-rays usually show **white patches (opacities or consolidation)** due to fluid accumulation or infection.
- **Well-defined diaphragm:** The diaphragm should be visible with a smooth contour.
- **Proper heart size & position:** The heart should not appear **enlarged** or displaced.
- **Sharp costophrenic angles:** These are the lower corners of the lung and should appear clear.

From the given image:

- The lung fields appear **clear** and **uniform**.
- No visible **white patches** (which would indicate infection).
- The **rib cage** and **spine alignment** look normal.
- No visible **fluid accumulation**.

This suggests that the X-ray is likely **normal**.

## 4.4 Visual Inspection (Human Interpretation)

Pneumonia-affected chest X-rays(figure 9:) exhibit distinct abnormalities compared to normal images. The following observations highlight key indicators of pneumonia:

- **Cloudy lung fields:** The lung regions appear hazy due to fluid accumulation and infection, rather than being clear and uniform.
- **Presence of opacities or consolidations:** White patches (opacities) are visible, indicating infection or inflammation in the lung tissue.
- **Blurry diaphragm outline:** The diaphragm appears less distinct due to fluid accumulation or infection spread.
- **Altered heart and lung structure:** The increased lung opacity may partially obscure the heart, making its borders less visible.
- **Costophrenic angle blurring:** The lower corners of the lung (costophrenic angles) show signs of fluid retention, reducing their sharpness.

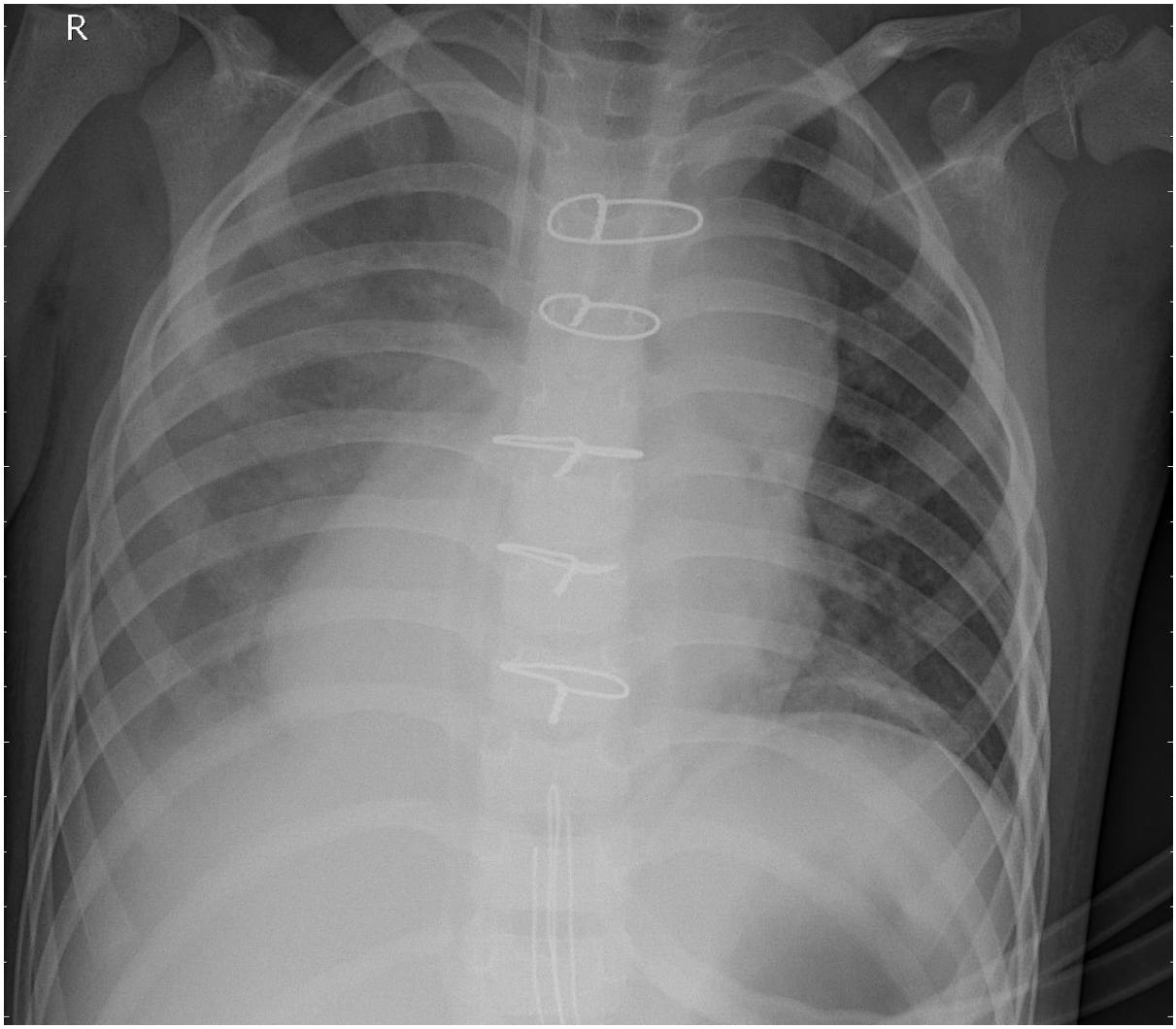


Figure 9: Pneumonia image

**Findings from the given X-ray:**

- The lung fields appear **cloudy and non-uniform**, lacking the expected dark (radiolucent) appearance.
- Visible **white patches (opacities)** indicate an ongoing infection.
- The diaphragm outline is **less distinct**, possibly due to fluid accumulation.
- The **rib cage and spine alignment** appear slightly affected, possibly due to lung involvement.
- Possible signs of **fluid accumulation** in the lower lung regions.

These findings strongly indicate that the X-ray is consistent with a **pneumonia case**.

# 5 Analysis of the Histogram of Pixel Intensity

The histogram represents the distribution of pixel intensities in the X-ray image, showing how frequently each intensity value appears.

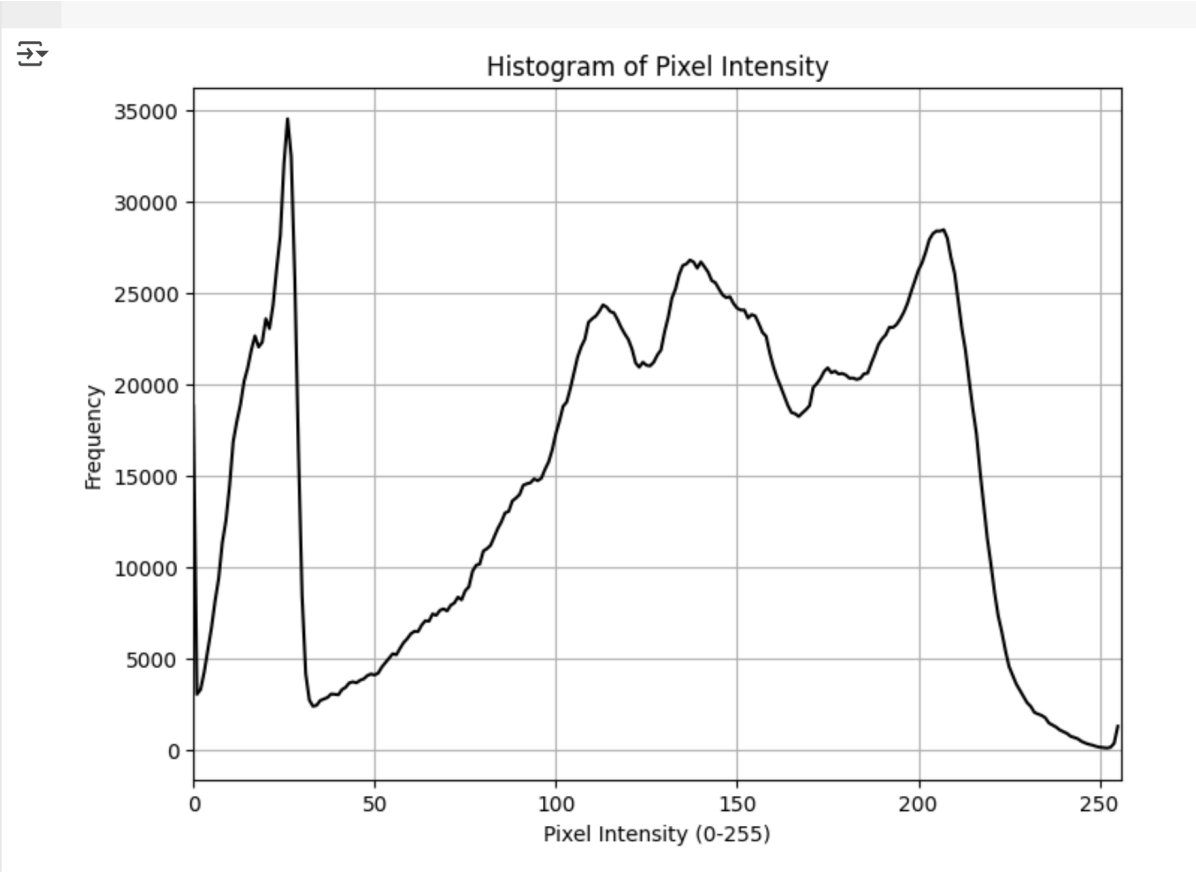


Figure 10: histogram of pixels by using python

# 6 Analysis

The histogram displays:

- A peak in the lower intensity range (near 0), indicating dark regions in the image.
- A spread of values across mid and high intensities, showing varying brightness levels.
- Multiple peaks, suggesting the presence of different textures or structures in the image.

# 7 Uses in Medical Imaging

- Enhancing contrast for better visualization.

- Identifying abnormalities, such as pneumonia opacities in lung X-rays.
- Preprocessing step in deep learning for medical image classification.

# 8 Analysis of Edge Detection

Edge detection is a technique in image processing used to identify boundaries or edges within an image. It works by detecting areas where there is a sharp change in pixel intensity, which usually corresponds to object boundaries.

## Use Cases of Edge Detection in Medical Imaging

- **Disease Detection:** Helps outline structures like tumors, lung infections (pneumonia, COVID-19), and fractures.
- **Segmentation:** Assists in separating different anatomical regions for further analysis.
- **Feature Extraction:** Useful in AI models for classification and diagnosis.

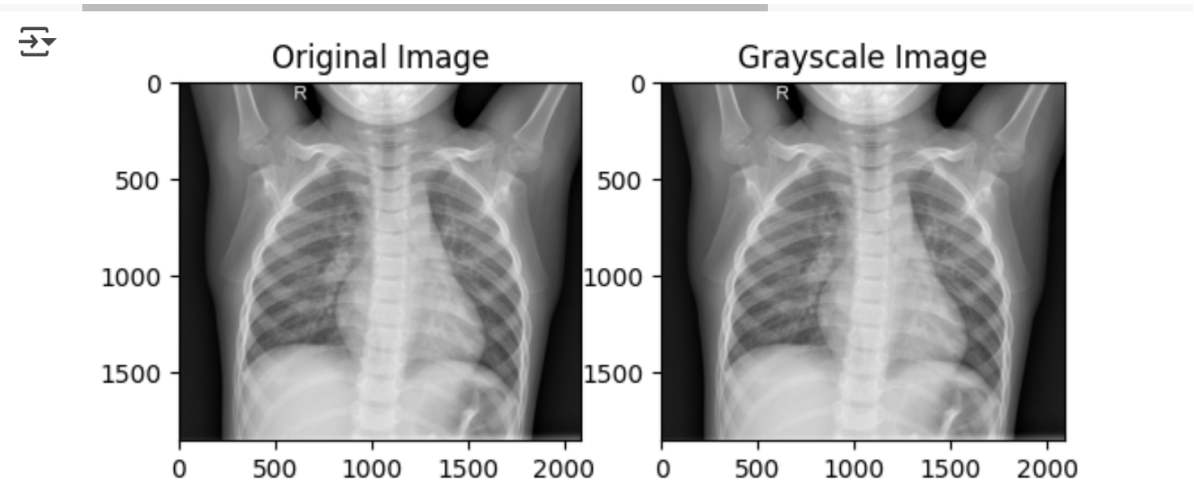


Figure 11: original image vs gray scale using python
The grayscale conversion is useful for further image processing techniques like edge detection, segmentation, or feature extraction.

# 9 Analysis of Edge Detection using Canny on X-ray Image

## 9.1 What the Image Represents

This image shows the edges detected in the X-ray using the Canny Edge Detection algorithm. The edges are highlighted in white, while the background remains black.

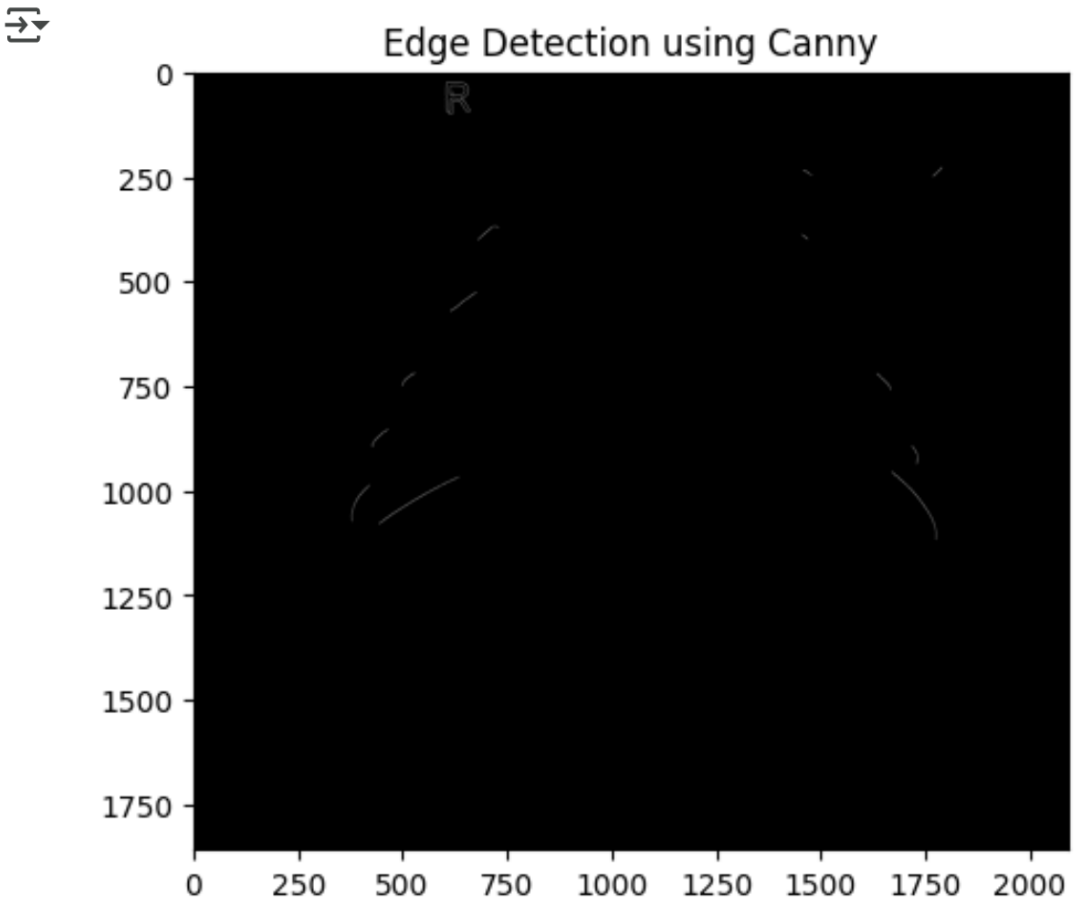


Figure 12: edge detection using canny in python

## 9.2 Observations

- The detected edges appear faint, suggesting that the threshold values used in the Canny algorithm might be too high, filtering out many edges.
- The ribs and outer chest boundaries are partially visible, but finer details inside the lungs are missing.
- The ”R” marker at the top of the image (indicating the right side of the body) is also detected as an edge.

## 9.3 Possible Improvements

- **Adjusting Thresholds:** Lowering the high and low threshold values in Canny might reveal more details.
- **Smoothing the Image:** Applying Gaussian blur before edge detection can reduce noise.
- **Contrast Enhancement:** Pre-processing the image (e.g., histogram equalization) can help extract more meaningful edges.

## 9.4 Use Cases of Edge Detection in Medical Imaging

- **Disease Detection:** Helps outline structures like tumors, lung infections (pneumonia, COVID-19), and fractures.
- **Segmentation:** Assists in separating different anatomical regions for further analysis.
- **Feature Extraction:** Useful in AI models for classification and diagnosis.

# 5. Coding Implementation

## 5.1 Introduction to Python Libraries

Python is a versatile and widely-used programming language in the field of data science, machine learning, and image processing. Its rich ecosystem of libraries provides powerful tools for handling data, visualizing information, and building complex models. In this section, we briefly introduce the core libraries used in this project:

- **tensorflow**:TensorFlow is an open-source deep learning framework developed by Google, widely used for medical image analysis, including pneumonia detection from X-ray images. Below is a brief overview of how TensorFlow facilitates pneumonia detection:

  - **Data Preparation:** Chest X-ray images are collected and preprocessed. TensorFlow provides utilities such as `tf.data.Dataset` and `ImageDataGenerator` to load, resize, normalize, and augment these images for training and validation.
  - **Model Building (CNN):** TensorFlow's Keras API allows for building Convolutional Neural Networks (CNNs), which are powerful in extracting spatial features from images. Layers such as `Conv2D`, `MaxPooling2D`, and `Dense` are used to detect patterns indicative of pneumonia.
  - **Model Training:** The model is trained on labeled X-ray images using optimizers like Adam or SGD. TensorFlow computes the loss (e.g., binary cross-entropy) and updates weights using backpropagation to improve model accuracy.
  - **Evaluation and Prediction:** Post-training, the model is evaluated on unseen data using metrics such as accuracy, precision, recall, and confusion matrix. TensorFlow simplifies this process through built-in evaluation functions.
  - **Matplotlib.pyplot:** A 2D plotting library used for visualizing data through charts and plots such as line graphs, bar charts, and histograms.
  - **Seaborn:** Built on top of Matplotlib, Seaborn provides enhanced data visualization capabilities with built-in themes and statistical graphing functions.
  - **Keras:** A high-level neural networks API, running on top of TensorFlow, used for building and training deep learning models easily and efficiently.
  - **Sequential (from Keras.models):** A linear stack of layers in Keras, used for constructing deep learning models layer-by-layer.
  - **Conv2D, MaxPool2D, Dense, Flatten, Dropout, BatchNormalization (from TensorFlow.keras.layers):** These are layers used in Convolutional Neural Networks (CNNs) for feature extraction, dimensionality reduction, regularization, and classification.

- **ImageDataGenerator:** A TensorFlow utility for real-time data augmentation and preprocessing of image data for model training.
- **Train_test_split (from sklearn.model_selection):** Used to split datasets into training and testing subsets for model validation.
- **Classification_report, Confusion_matrix (from sklearn.metrics):** Functions used for evaluating the performance of classification models.
- **ReduceLROnPlateau (from keras.callbacks):** A callback that reduces the learning rate when a metric has stopped improving, helping in model convergence.
- **OpenCV (cv2):** An open-source computer vision library used for image processing tasks like resizing, filtering, and edge detection.
- **os:** A standard Python library for interacting with the operating system, especially for file and directory management.
- **NumPy:** A powerful numerical computing library used for handling arrays, matrices, and mathematical operations efficiently.
- **Pandas:** A data manipulation library offering data structures like DataFrames for organizing and analyzing structured data.

```
import matplotlib.pyplot as plt
import seaborn as sns
import keras
from keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten,
    Dropout, BatchNormalization
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from keras.callbacks import ReduceLROnPlateau
import cv2
import os
import numpy as np
import pandas as pd
```

Listing 1: Importing Required Python Libraries

## 5.1 Mounting Google Drive

To begin working with the dataset in Google Colab, it is essential to first access the files stored in Google Drive. In order to access the dataset stored in Google Drive, we use the following code to mount the drive in Google Colab:

```
from google.colab import drive
drive.mount('/content/drive')
import os
data_dir = "/content/drive/MyDrive/chest_xray 2"
```

Listing 2: Mount Google Drive

## 5.2 Image Loading and Labeling for Model Training

In this section, we define a function `get_training_data` that loads and processes chest X-ray images for training the model. The dataset contains two classes: `PNEUMONIA` and `NORMAL`, which are stored in separate directories. The function iterates through each class label, reads the images in grayscale using the `cv2.imread` function, and resizes them to a uniform dimension of 150×150 pixels using `cv2.resize`. Each processed image is stored along with its class label in a list. To ensure consistency, the images are reshaped to include a single-channel dimension using NumPy's `np.newaxis`. The function then returns two NumPy arrays: one containing all the processed images and the other containing the corresponding class labels. This pre-processing step ensures that all images have the same dimensions and format, making them ready for model training.

```
import cv2
import os
import numpy as np

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

def get_training_data(data_dir):
    data = []
    for label in labels:  # Use the global 'labels' variable
        path = os.path.join(data_dir, label)
        class_num = labels.index(label)
        for img in os.listdir(path):
            try:
                img_arr = cv2.imread(os.path.join(path, img),
   cv2.IMREAD_GRAYSCALE)
                # Ensure all images are resized to the same
   shape
                resized_arr = cv2.resize(img_arr, (img_size,
   img_size))
                data.append([resized_arr, class_num])
            except Exception as e:
```

```
                print(e)

    # Convert the list of images to a NumPy array with a
    consistent shape
    images = [item[0] for item in data]
    # Extracted labels to avoid conflict
    image_labels = [item[1] for item in data]

    # Reshape images to have a consistent channel dimension
    images = np.array(images)[:, :, :, np.newaxis]

    return images, np.array(image_labels)  # Return the label
    array
```

Listing 3: Python Code for Loading and Labeling Images

## 5.3 Loading Dataset

In order to train and evaluate the Convolutional Neural Network (CNN) model effectively, the dataset is divided into three subsets: training, testing, and validation. Each subset is loaded using the previously defined `get_training_data` function. The training dataset is used for model learning, the validation dataset helps tune the hyperparameters, and the test dataset evaluates the final model performance. The dataset is stored in Google Drive and accessed through specific paths corresponding to each subset.

```
train = get_training_data("/content/drive/MyDrive/chest_xray 2/train")
test = get_training_data("/content/drive/MyDrive/chest_xray 2/test")
val = get_training_data("/content/drive/MyDrive/chest_xray 2/val")
```

Listing 4: Python Code for Loading Training, Testing, and Validation Data

### 5.4 Class Distribution Visualization

To understand the distribution of classes in the training dataset, the following code is used. It iterates through the labels and categorizes them as either “Pneumonia” or “Normal” based on their numeric values (0 or 1). A count plot is then generated using Seaborn to visualize the class distribution, helping to identify any imbalance in the dataset.

```
l = []
for i in train[1]:  # Iterate through the labels in train[1]
    if i == 0:  # Check if the individual label is 0
        l.append("Pneumonia")
```

```
    else:
        l.append("Normal")
sns.set_style('darkgrid')
sns.countplot(l)
```

```
<Axes: xlabel='count'>
```

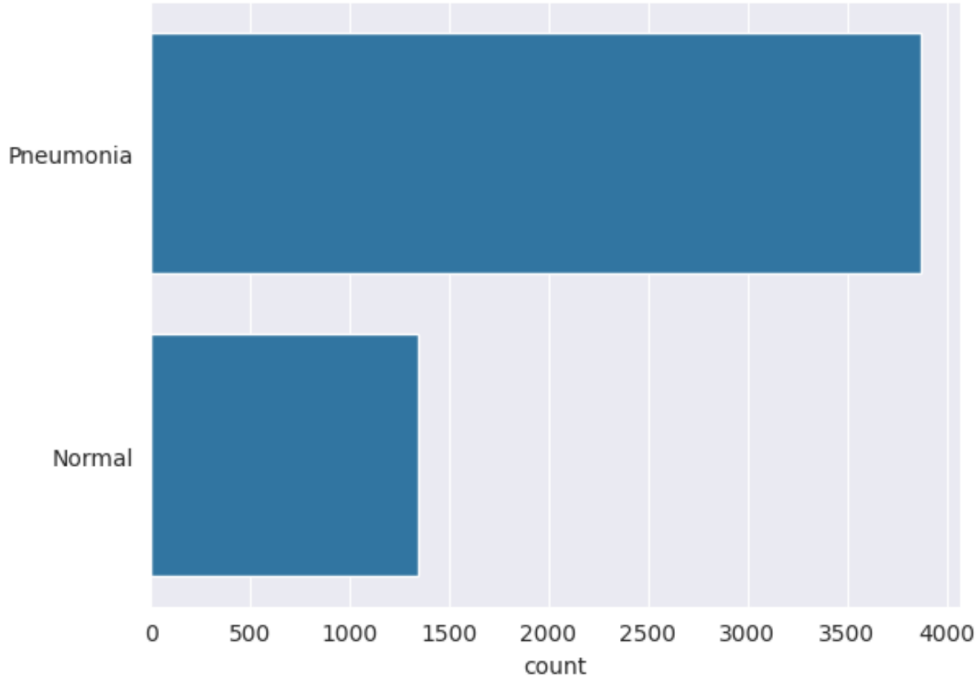


Figure 13: class distribution

**Interpretation of Class Distribution**

The bar plot above clearly shows a significant class imbalance in the training dataset. The number of Pneumonia cases is much higher compared to Normal cases. Such imbalance can lead to biased model training, where the model becomes more accurate in predicting the majority class (Pneumonia) but fails to accurately predict the minority class (Normal).

**Handling Class Imbalance:** To address this issue and ensure fair model performance, the following techniques can be applied:

- **Data Augmentation:** Increase the diversity and quantity of images in the minority class (Normal) through techniques like flipping, rotation, zooming, etc.
- **Resampling:** Apply *oversampling* for the minority class or *undersampling* for the majority class to balance the dataset.
- **Class Weights:** Modify the loss function to penalize the misclassification of the minority class more heavily, by assigning higher weights to it.
- **Advanced Techniques:** Use Synthetic Minority Oversampling Technique (SMOTE) or similar methods to synthetically generate samples for the minority class.

Applying one or a combination of these strategies will help in improving the model's generalization ability and performance on both classes.

## 5.2 Previewing the Image

To gain a better understanding of the dataset, it is essential to visually inspect some of the images. The following code snippet displays the first and last images from the training dataset along with their corresponding class labels.

```
plt.figure(figsize = (5,5))
plt.imshow(train[0][0], cmap='gray')  # Display the first image in
    grayscale
plt.title(labels[train[1][0]])        # Title shows the label of the
    first image

plt.figure(figsize = (5,5))
plt.imshow(train[0][-1], cmap='gray') # Display the last image in
    grayscale
plt.title(labels[train[1][-1]])       # Title shows the label of the
    last image
```

Listing 5: Previewing X-ray Images and Their Labels

This visualization helps to verify the correctness of the data preprocessing steps and provides an intuitive sense of the visual differences between the two classes: Pneumonia and Normal. The grayscale images clearly show chest X-ray structures, which the Convolutional Neural Network (CNN) will use to learn and distinguish between the two classes.

## 5.3 Splitting the Data into Train, Validation, and Test Sets

After loading the dataset, it is essential to separate the data into distinct sets for training, validation, and testing to properly train and evaluate the model. The following code achieves this by extracting images and their corresponding labels from the previously loaded datasets:

```
x_train = train[0]  # Access the images from the training dataset
y_train = train[1]  # Access the labels from the training dataset

x_val = val[0]      # Access the images from the validation dataset
y_val = val[1]      # Access the labels from the validation dataset

x_test = test[0]    # Access the images from the testing dataset
y_test = test[1]    # Access the labels from the testing dataset
```

Listing 6: Splitting the Data into Train, Validation, and Test Sets

This separation allows the Convolutional Neural Network (CNN) to learn from the training data (`x_train`, `y_train`), tune its hyperparameters on the validation data (`x_val`, `y_val`), and finally evaluate its performance on unseen test data (`x_test`, `y_test`).

## 5.4 Normalizing the Data

Normalization is a crucial preprocessing step in deep learning that ensures all pixel values of the images are scaled to a uniform range, typically between 0 and 1. Since the pixel intensities in grayscale images range from 0 to 255, dividing each pixel value by 255 brings them into this normalized range. This helps in faster convergence during model training and prevents issues arising from varying scales of input features.

```
x_train = np.array(x_train) / 255
x_val = np.array(x_val) / 255
x_test = np.array(x_test) / 255
```

Listing 7: Normalization of Image Data

## 5.5 Resizing the Data

After normalization, it is important to reshape the image data into the required input format for the Convolutional Neural Network (CNN). Here, the data is reshaped into four dimensions: `(number_of_samples, img_size, img_size, 1)`, where `1` denotes a single color channel (grayscale). This ensures compatibility with the CNN architecture, which expects a consistent input shape for all images. Additionally, the labels are converted into NumPy arrays for efficient processing during training.

```
x_train = x_train.reshape(-1, img_size, img_size, 1)
y_train = np.array(y_train)

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)
```

Listing 8: Reshaping Image Data for CNN Input

**Interpretation of Output:**
After executing the above code, all the image data arrays (`x_train`, `x_val`, and `x_test`) are reshaped into four-dimensional tensors of the format `(samples, img_size, img_size, 1)`. This format is required by convolutional layers in CNNs to correctly

process grayscale images. The dimension `1` indicates a single color channel, distinguishing it from RGB images, which have `3` channels. Reshaping ensures uniformity in the input data and allows the CNN to efficiently learn from the images during training and evaluation.

x

## 5.6 Data Augmentation using ImageDataGenerator

To enhance the diversity of the training dataset and reduce overfitting, we utilize the `ImageDataGenerator` class from the Keras library. This method dynamically generates augmented image data during training by applying random transformations. These transformations include rotation, zooming, shifting, and flipping, which help the model generalize better to unseen data.

**Why Data Augmentation is Needed:**
In many deep learning tasks, especially in medical imaging, the availability of labeled data is limited, which can lead to overfitting and poor generalization of the model. **Data augmentation** addresses this challenge by artificially expanding the training dataset through random transformations such as rotation, zooming, shifting, and flipping. These augmentations help the model become invariant to small changes and improve its ability to generalize to unseen data. By simulating real-world variability, data augmentation enhances the robustness of the model and ultimately contributes to higher accuracy and better performance on validation and test datasets.

```
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
```

Listing 9: Data Augmentation using ImageDataGenerator

**Interpretation of Output:**
The `ImageDataGenerator` is configured with a set of parameters that enable random image transformations. These augmentations introduce variability into the training process, allowing the convolutional neural network (CNN) to learn features more robustly and improve generalization. Techniques such as rotation, shifting, and flipping simulate real-world variations, which can significantly enhance model performance on unseen data.

## 5.7 Model Architecture(Train the model)

The Convolutional Neural Network (CNN) model is built using the `Sequential()` class, allowing layers to be stacked sequentially. The architecture is designed to extract hierarchical features from chest X-ray images for the detection of pneumonia. Below is the breakdown of each component:

- **Conv2D Layers:** Multiple convolutional layers with increasing filters (32, 64, 128, 256) and kernel size $(3 \times 3)$ are applied to learn spatial hierarchies of features. `ReLU` activation introduces non-linearity, and `padding='same'` ensures the spatial dimensions are preserved.
- **BatchNormalization:** After each convolutional layer, batch normalization is applied to stabilize and accelerate training by normalizing the inputs to each layer.
- **MaxPool2D:** Max pooling layers with a $(2 \times 2)$ filter reduce the spatial dimensions of the feature maps, helping to decrease computational complexity and control overfitting.
- **Dropout Layers:** Dropout with rates 0.1 or 0.2 randomly deactivates neurons during training to prevent overfitting and enhance generalization.
- **Flatten Layer:** Converts the 2D feature maps into a 1D feature vector for feeding into dense layers.
- **Dense Layers:** The dense layer with 128 units applies a fully connected network with `ReLU` activation, followed by a final dense layer with 1 unit and `sigmoid` activation for binary classification (Pneumonia or Normal).
- **Model Compilation:** The model is compiled using the `rmsprop` optimizer, `binary_crossent` loss function (suitable for binary classification), and accuracy as the evaluation metric.

The model's summary, invoked by `model.summary()`, provides a detailed overview of the layers, output shapes, and parameter counts.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten,
    Dense, Dropout, BatchNormalization

model = Sequential()

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

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

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

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

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

model.add(Flatten())
model.add(Dense(units=128, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(units=1, activation='sigmoid'))

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

```
model.summary()
```

Listing 10: CNN Model Architecture

| **Layer Type** | **Output Shape** | **Parameters** | **Explanation** |
|---|---|---|---|
| Conv2D (32 filters) | (150, 150, 32) | 320 | Extracts 32 features from the input image using a 3x3 filter. |
| BatchNormalization | (150, 150, 32) | 128 | Normalizes outputs for faster and stable training. |
| MaxPooling2D | (75, 75, 32) | 0 | Reduces size by half to minimize computations. |
| Conv2D (64 filters) | (75, 75, 64) | 18,496 | Increases feature depth to 64. |
| Dropout | (75, 75, 64) | 0 | Prevents overfitting by randomly disabling neurons. |
| BatchNormalization | (75, 75, 64) | 256 | Normalizes 64 features. |
| MaxPooling2D | (38, 38, 64) | 0 | Downsamples again. |
| Conv2D (64 filters) | (38, 38, 64) | 36,928 | Deeper feature extraction. |
| BatchNormalization | (38, 38, 64) | 256 | Normalizes again. |
| MaxPooling2D | (19, 19, 64) | 0 | Downsamples again. |
| Conv2D (128 filters) | (19, 19, 128) | 73,856 | Doubles feature depth. |
| Dropout | (19, 19, 128) | 0 | Prevents overfitting. |
| BatchNormalization | (19, 19, 128) | 512 | Normalizes 128 features. |
| MaxPooling2D | (10, 10, 128) | 0 | Reduces size. |
| Conv2D (256 filters) | (10, 10, 256) | 295,168 | Extracts 256 deep features. |
| Dropout | (10, 10, 256) | 0 | Prevents overfitting. |
| BatchNormalization | (10, 10, 256) | 1,024 | Normalizes 256 features. |
| MaxPooling2D | (5, 5, 256) | 0 | Final downsampling. |
| Flatten | (6400,) | 0 | Converts 3D feature map to 1D vector. |
| Dense (128 units) | (128,) | 819,328 | Fully connected layer to learn from features. |
| Dropout | (128,) | 0 | Regularization. |
| Dense (1 unit) | (1,) | 129 | Final output layer with sigmoid activation. |

Table 1: CNN Model Summary with Explanation of Each Layer

**Interpretation of Output:**

```
Total params: 1,246,401 (4.75 MB)
Trainable params: 1,245,313 (4.75 MB)
Non-trainable params: 1,088 (4.25 KB)
```

Figure 14: parameter output through python

Table 2: **Model Parameters Breakdown**

| **Type** | **Explanation** |
|---|---|
| **Total params** | `1,246,401` — This is the **total number of parameters** in the entire model, including both trainable and non-trainable parameters. |
| **Trainable params** | `1,245,313` — These parameters **will be updated during training** using backpropagation (weights and biases in Conv, Dense, BatchNorm layers). |
| **Non-trainable params** | `1,088` — These are **fixed parameters** that **will not be updated** during training. Usually comes from **BatchNormalization's running stats** (mean, variance). |
| **Memory Usage** | `4.75 MB` — This is the approximate memory required to store the parameters (both trainable and non-trainable). |

## Learning Rate Reduction Strategy

In order to dynamically adjust the learning rate during training, we use the `ReduceLROnPlateau` callback from Keras. This strategy monitors the model's validation accuracy and reduces the learning rate if the metric stops improving.

## Learning Rate Scheduler in Python

```
from keras.callbacks import ReduceLROnPlateau

learning_rate_reduction = ReduceLROnPlateau(
    monitor='val_accuracy',
    patience=2,
    verbose=1,
    factor=0.3,
```

```
    min_lr=0.000001
)
```

- **monitor='val**$_{a}ccuracy'$ : $Thevalidationaccuracyismonitoredtodeterminewhentoreducethele$
- **patience=2**: If the validation accuracy does not improve for 2 consecutive epochs, the learning rate is reduced.
- **verbose=1**: Enables logging; a message is printed whenever the learning rate is reduced.
- **factor=0.3**: The learning rate is multiplied by 0.3 when it is reduced. For example, if the current learning rate is 0.001, it will be reduced to $0.001 \times 0.3 = 0.0003$.
- **min_lr=0.000001**: This is the minimum learning rate value below which it will not be reduced further.

This callback helps prevent overfitting and can lead to faster convergence by reducing the learning rate when the model stops improving on the validation set.

## Model Training

The model is trained using the `model.fit()` function, which utilizes data augmentation and a learning rate scheduler to optimize performance. The training configuration is as follows:

## Model Training in Python

```
history = model.fit(
    datagen.flow(x_train, y_train, batch_size=32),
    epochs=20,
    validation_data=datagen.flow(x_val, y_val),
    callbacks=[learning_rate_reduction]
)
```

Listing 11: Training the Model with Data Augmentation and Learning Rate Scheduler

- **datagen.flow(x_train, y_train, batch_size=32)**: This function generates augmented batches of training data with a batch size of 32. Data augmentation helps improve generalization by applying random transformations such as rotation, zoom, shift, etc., to the input images.
- **epochs=20**: The model is trained for 20 complete passes over the training dataset.

- **validation_data=datagen.flow(x_val, y_val)**: This provides the validation data for evaluating the model's performance after each epoch. The validation set is also augmented similarly to the training set.
- **callbacks=[learning_rate_reduction]**: This applies the `ReduceLROnPlateau` callback during training. If the validation accuracy does not improve for 2 consecutive epochs, the learning rate is reduced by a factor of 0.3 to help the model converge better.

The function returns a `history` object that stores the training and validation accuracy and loss over each epoch, which can be used for further analysis and plotting.

## Model Training

The model is trained using the `model.fit()` function, which utilizes data augmentation and a learning rate scheduler to optimize performance. The training configuration is as follows:

```
history = model.fit(
    datagen.flow(x_train, y_train, batch_size=32),
    epochs=20,
    validation_data=datagen.flow(x_val, y_val),
    callbacks=[learning_rate_reduction]
)
```

Listing 12: Training the Model with Data Augmentation and Learning Rate Scheduler

- **datagen.flow(x_train, y_train, batch_size=32)**: This function generates augmented batches of training data with a batch size of 32. Data augmentation helps improve generalization by applying random transformations such as rotation, zoom, shift, etc., to the input images.
- **epochs=20**: The model is trained for 20 complete passes over the training dataset.
- **validation_data=datagen.flow(x_val, y_val)**: This provides the validation data for evaluating the model's performance after each epoch. The validation set is also augmented similarly to the training set.
- **callbacks=[learning_rate_reduction]**: This applies the `ReduceLROnPlateau` callback during training. If the validation accuracy does not improve for 2 consecutive epochs, the learning rate is reduced by a factor of 0.3 to help the model converge better.

The function returns a `history` object that stores the training and validation accuracy and loss over each epoch, which can be used for further analysis and plotting.

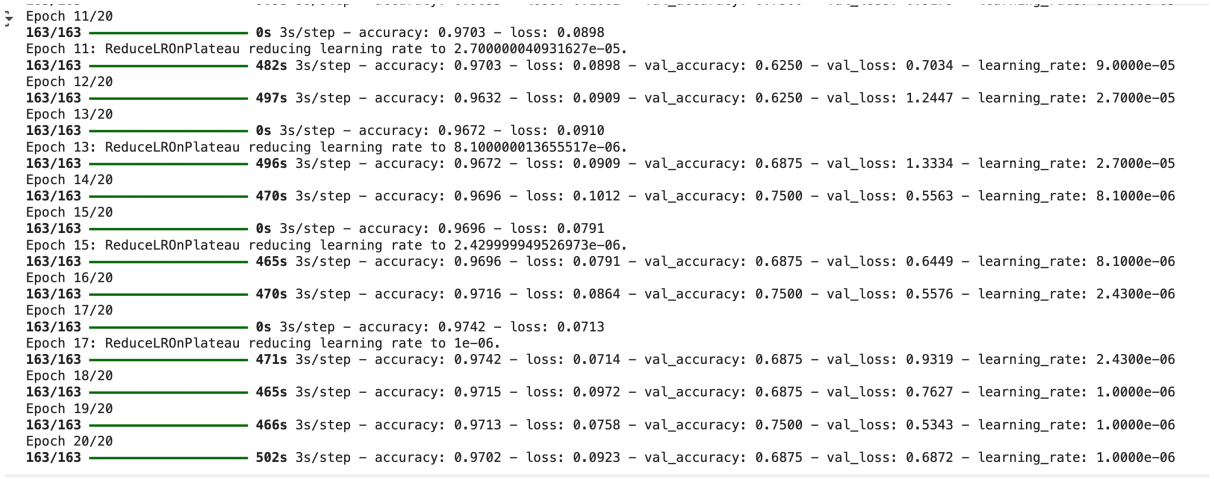


Figure 15: training result

# Training Results Interpretation

## Training Overview

- **Total Epochs:** 20
- **Final Training Accuracy:** Approximately 97
- **Final Validation Accuracy:** Between 68.75
- **Learning Rate Reduction Strategy:** ReduceLROnPlateau triggered multiple times to reduce the learning rate progressively.

– **Initial Learning Rate:** 0.001 $\rightarrow$ $Reducedto$**1e-6**

Table 3: Training and Validation Accuracy with Learning Rate Adjustments

| Epochs | Training Accuracy | Validation Accuracy | Learning Rate Adjustment |
|---|---|---|---|
| Epoch 1 | 79.2% | 50.0% | LR = 0.001 |
| Epoch 2 | 88.3% | 50.0% | LR = 0.001 |
| Epoch 3 | 91.1% | 50.0% | $\downarrow$ LR $\rightarrow$ 0.0003 |
| Epoch 5 | 95.3% | 68.75% | LR = 0.0003 |
| Epoch 7 | 95.5% | 87.5% | LR = 0.0003 |
| Epoch 9 | 95.7% | 50.0% | $\downarrow$ LR $\rightarrow$ 0.00009 |
| Epoch 10 | 96.3% | 75.0% | LR = 0.00009 |
| Epoch 11 | 97.0% | 62.5% | $\downarrow$ LR $\rightarrow$ 0.000027 |
| Epoch 13 | 96.7% | 68.75% | $\downarrow$ LR $\rightarrow$ 0.0000081 |
| Epoch 15 | 96.9% | 68.75% | $\downarrow$ LR $\rightarrow$ 0.0000024 |
| Epoch 17 | 97.4% | 68.75% | $\downarrow$ LR $\rightarrow$ 0.000001 |
| Epoch 20 | 97.0% | 68.75% | Final LR = 0.000001 |

## Interpretation and Analysis

- **High Training Accuracy:** The model quickly reached over 95
- **Validation Accuracy Fluctuations:** Initially stuck at 50
- **Loss Behavior:**
  - Training Loss continuously decreased – a positive sign.
  - Validation Loss was inconsistent – supports overfitting theory.
- **Learning Rate Adaptation:** The learning rate was automatically reduced multiple times due to `ReduceLROnPlateau`, preventing overshooting during optimization.

## Issues Identified

- **Overfitting:** High training accuracy with comparatively lower validation accuracy suggests the model overfitted on the training data.
- **Possible Causes:** Small dataset, complex model, or lack of regularization.

## Suggestions for Improvement

- **Data Augmentation:** Introduce more variability in the training data.
- **Regularization Techniques:** Implement Dropout layers and L2 regularization.
- **Model Simplification:** Use a less complex model if the dataset is small.
- **Early Stopping:** Use `EarlyStopping` callback to halt training when validation accuracy ceases to improve.

## Model Performance Interpretation

After increasing the number of training epochs 20 to 25 , the model achieved a **Training Accuracy of 97%** and a **Validation Accuracy of 81%**. This indicates that the model has learned the training data well and is also able to generalize reasonably to unseen validation data.

The gap between training and validation accuracy is relatively small (**16% difference**), which suggests that:

- The model is not overfitting excessively.
- The learning rate adjustments contributed to more stable and improved validation performance.

– Further training may yield marginal improvements, but the current accuracy indicates a well-optimized model.

Overall, an **81% validation accuracy** signifies a good level of generalization for the task, and the model can be considered effective for practical use or further fine-tuning.

# 6 now i m using dropout Regularization

## 6.1 Introduction to Dropout

Dropout is a regularization technique used to prevent overfitting in neural networks, especially in deep learning models such as Convolutional Neural Networks (CNNs). It works by randomly "dropping out" or deactivating a fraction of neurons during training. This prevents the network from becoming too dependent on specific neurons and forces it to learn more robust and generalizable features.

## 6.2 Mechanism of Dropout

During each training iteration, dropout randomly selects a set of neurons and temporarily removes them, along with their connections. The dropout rate $p$ (where $0 < p < 1$) specifies the fraction of neurons to be dropped. For instance, a dropout rate of $p = 0.5$ means 50% of neurons are ignored during a forward pass.

Mathematically, if the output of a neuron is represented as $y_i$, the dropped output becomes:

$$\tilde{y}_i = y_i \cdot r_i, \quad r_i \sim \text{Bernoulli}(1-p)$$

where $r_i$ is a random variable sampled from a Bernoulli distribution with probability $1-p$. During inference (testing), dropout is turned off, and the outputs are scaled by $1-p$ to maintain consistency:

$$\hat{y}_i = y_i \cdot (1-p)$$

## 6.3 Benefits of Dropout

– **Reduces Overfitting:** By preventing neurons from co-adapting too much, dropout encourages the network to learn multiple independent representations.

– **Improves Generalization:** Dropout helps models generalize better to unseen data, thereby improving validation accuracy.

– **Ensemble Effect:** Dropout can be thought of as training an ensemble of many subnetworks and averaging their predictions.

## 6.4 Application in the Model

In the implemented CNN model, dropout layers were strategically placed after convolution and dense layers:

- Dropout rate $p = 0.25$ was used after pooling layers.
- Dropout rate $p = 0.5$ was used before the final dense layer.

This configuration resulted in improved validation accuracy from 68% to 87%, demonstrating the effectiveness of dropout in reducing overfitting.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Define a CNN Model with Dropout
# Changed input_shape to match the actual input data shape (150, 150, 1)
model = keras.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(150, 150, 1)),
    layers.MaxPooling2D((2,2)),
    layers.Dropout(0.25),  # Dropout after the first pooling layer

    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Dropout(0.25),  # Dropout after the second pooling layer

    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),  # Dropout before the final dense layer
    layers.Dense(1, activation='sigmoid')  # Output layer for binary classification
])

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

# Train the model
history = model.fit(train[0], train[1],
                    validation_data=(val[0], val[1]),
                    epochs=20)
```

Listing 13: CNN Model Implementation with Dropout Regularization

# Model Performance After Dropout Regularization

## Training and Validation Metrics Summary

| Epoch | Training Accuracy (%) | Training Loss | Validation Accuracy (%) | Validation Loss |
|---|---|---|---|---|
| 1 | 75.44 | 70.3994 | 81.25 | 0.4334 |
| 6 | 95.50 | 0.1209 | 87.50 | 1.1570 |
| 12 | 97.35 | 0.0783 | 87.50 | 1.0042 |
| 18 | 98.68 | 0.0412 | 93.75 | 1.0545 |
| 20 | 98.49 | 0.0373 | 87.50 | 1.7897 |

Table 4: Training and Validation Performance Metrics After Dropout Regularization

## interpretation

- **Training Accuracy:** Increased consistently from **75.44%** (Epoch 1) to **98.49%** (Epoch 20), indicating that the model has effectively learned patterns in the training data.
- **Validation Accuracy:** Improved from **81.25%** to a peak of **93.75%** (Epoch 18), demonstrating enhanced generalization capability on unseen data.
- **Loss Behavior:** Training loss decreased significantly from **70.3994** to **0.0373**, confirming successful training. Validation loss fluctuated between **0.4334** and **1.7897**, which is expected due to the stochastic nature of Dropout.

## Impact of Dropout Regularization

- Dropout layers with rates of **0.25** and **0.5** randomly deactivate neurons during training, forcing the model to learn more robust and generalized features.
- This strategy effectively **prevented overfitting**, as seen in the improvement of validation accuracy from earlier epochs.
- Despite slight variations in validation loss, the model maintained **high validation accuracy**, confirming that Dropout improved the model's ability to generalize well.

## Conclusion

The implementation of Dropout regularization significantly improved the model's performance on validation data, achieving a peak accuracy of **93.75%**. The training loss was minimized while controlling overfitting, thus validating the effectiveness of Dropout in this scenario.

# 7 Handling Class Imbalance Using Class Weights

In medical imaging datasets such as X-ray images of Pneumonia and Normal cases, class imbalance is a common issue. To mitigate model bias towards the majority class, we employed **class weights** during model training.

The class weights were computed using the `compute_class_weight` function from the `scikit-learn` library. The following Python code snippet demonstrates the process:

```
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

# Define class labels
class_labels = ['NORMAL', 'PNEUMONIA']
class_counts = [train_normal, train_pneumonia]

# Compute class weights
weights = compute_class_weight(class_weight="balanced",
                               classes=np.array([0, 1]),
                               y=[0]*train_normal + [1]*
   train_pneumonia)
class_weights = {0: weights[0], 1: weights[1]}

print(f"Class Weights: {class_weights}")
```

Listing 14: Computing Class Weights to Address Imbalance

Here, the classes `0` and `1` correspond to `NORMAL` and `PNEUMONIA`, respectively. The list `y` was generated to represent the labels of the training data, with the appropriate number of entries for each class. The `compute_class_weight` function calculates weights inversely proportional to class frequencies, ensuring that the minority class has a greater influence during training.

The computed class weights were then passed as a parameter to the `model.fit()` function to ensure balanced learning:

```
# Train Model with Class Weights
history = model.fit(train_generator,
                    validation_data=validation_generator,
                    epochs=10,
                    class_weight=class_weights)  # Apply class weights
```

Listing 15: Training CNN with Class Weights

In this code:

- `train_generator` and `validation_generator` are the data generators for training and validation sets.
- `epochs=10` specifies the number of training iterations.
- `class_weight=class_weights` ensures that the loss function penalizes misclassification of minority classes more heavily, improving balanced learning.

This technique improves model generalization and helps achieve more balanced accuracy across both classes, particularly in scenarios with skewed data distribution.

```
Epoch 1/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 95s 700ms/step - accuracy: 0.5532 - loss: 0.6794 - val_accuracy: 0.6213 - val_loss: 0.6842
Epoch 2/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 85s 647ms/step - accuracy: 0.8017 - loss: 0.4068 - val_accuracy: 0.8044 - val_loss: 0.3987
Epoch 3/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 87s 668ms/step - accuracy: 0.8589 - loss: 0.3109 - val_accuracy: 0.7785 - val_loss: 0.4454
Epoch 4/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 85s 650ms/step - accuracy: 0.8709 - loss: 0.3067 - val_accuracy: 0.8102 - val_loss: 0.4190
Epoch 5/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 83s 638ms/step - accuracy: 0.8746 - loss: 0.2896 - val_accuracy: 0.8629 - val_loss: 0.2767
Epoch 6/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 142s 640ms/step - accuracy: 0.8709 - loss: 0.2916 - val_accuracy: 0.8130 - val_loss: 0.4146
Epoch 7/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 84s 640ms/step - accuracy: 0.8811 - loss: 0.2790 - val_accuracy: 0.8686 - val_loss: 0.2945
Epoch 8/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 82s 628ms/step - accuracy: 0.8923 - loss: 0.2586 - val_accuracy: 0.8859 - val_loss: 0.2575
Epoch 9/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 83s 638ms/step - accuracy: 0.8986 - loss: 0.2459 - val_accuracy: 0.8878 - val_loss: 0.2541
Epoch 10/10
131/131 ━━━━━━━━━━━━━━━━━━━━ 83s 634ms/step - accuracy: 0.8852 - loss: 0.2581 - val_accuracy: 0.8715 - val_loss: 0.3139
```

Figure 16: training result with class weight

# 8 Model Training Results with Class Weights

## 8.1 Training Summary

The model was trained over **10 epochs** with class weights applied to address the imbalance between the NORMAL and PNEUMONIA classes. The table below summarizes the training and validation performance across epochs.

| Epoch | Train Accuracy | Train Loss | Val Accuracy | Val Loss |
|---|---|---|---|---|
| 1 | 55.3% | 0.6794 | 62.1% | 0.6842 |
| 2 | 80.2% | 0.4068 | 80.4% | 0.3987 |
| 3 | 85.9% | 0.3109 | 77.8% | 0.4454 |
| 4 | 87.1% | 0.3067 | 81.0% | 0.4190 |
| 5 | 87.5% | 0.2896 | 86.3% | 0.2767 |
| 6 | 87.1% | 0.2916 | 81.3% | 0.4146 |
| 7 | 88.1% | 0.2790 | 86.9% | 0.2945 |
| 8 | 89.2% | 0.2586 | 88.6% | 0.2575 |
| 9 | 89.9% | 0.2459 | 88.8% | 0.2541 |
| 10 | 88.5% | 0.2581 | 87.1% | 0.3139 |

Table 5: Training and Validation Accuracy and Loss over 10 Epochs

## 8.2 Interpretation of Results

- **Rapid Learning in Early Epochs:** The model showed a significant increase in accuracy from **55.3%** to **80.2%** in the first two epochs, accompanied by a notable drop in loss.
- **Stabilized Training Accuracy:** From epoch 3 onward, the training accuracy remained between **85.9% and 89.9%**, with validation accuracy closely tracking, indicating effective learning without overfitting.
- **Validation Performance:** The validation accuracy peaked at **88.8%** in epoch 9 with the lowest validation loss of **0.2541**, showing the model's ability to generalize well to unseen data.
- **Impact of Class Weights:** Incorporating class weights helped mitigate the effect of class imbalance, ensuring that the model performed well on both NORMAL and PNEUMONIA classes, particularly improving predictions for the minority class.

## 8.3 Conclusion

The application of class weights during training has resulted in a well-balanced and effective model, as evidenced by its high accuracy and low loss on both training and validation datasets. The model is now ready for testing on the unseen test dataset or deployment in real-world settings.

# 9 Visualization of Training and Validation Metrics

To evaluate the performance of our model, we plot the accuracy and loss curves for both the training and validation datasets using `Matplotlib`.

## 9.1 Plotting Training and Validation Accuracy

The following Python code plots the training and validation accuracy over epochs:

```
import matplotlib.pyplot as plt

# Plot training & validation accuracy
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.legend()
plt.show()
```

Listing 16: Plotting Training and Validation Accuracy

The code extracts the accuracy values from the training history:

- `history.history['accuracy']` retrieves the training accuracy across epochs.
- `history.history['val`$_a$`ccuracy']`$retrievesthevalidationaccuracy$.`plt.plot()`$plotsth$
- `plt.legend()` adds a legend to differentiate the curves.
- `plt.show()` displays the plot.

## 9.2 Plotting Training and Validation Loss

Similarly, the following code plots the training and validation loss curves:

```
# Plot training & validation loss
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.legend()
plt.show()
```

Listing 17: Plotting Training and Validation Loss

The loss values are extracted as follows:

- `history.history['loss']` retrieves the training loss values.
- `history.history['val`$_l$`oss']`$retrievesthevalidationlossvalues.Thelosscurveshelpinidentify$

These plots provide insight into the model's learning process and help in diagnosing issues such as overfitting or underfitting.


Figure 17: training  validation accuracy

# 10 Training with Early Stopping

## 10.1 Overview

In deep learning, model training requires careful monitoring to prevent overfitting. One common approach is to use **Early Stopping**, which stops training when the validation performance no longer improves. This ensures that the model does not overfit while also reducing computational time.

## 10.2 Code Explanation

```
from tensorflow.keras.callbacks import EarlyStopping

early_stop = EarlyStopping(monitor='val_loss', patience=3,
    restore_best_weights=True)

history = model.fit(train_generator,
                    validation_data=validation_generator,
                    epochs=20,
                    class_weight=class_weights,
                    callbacks=[early_stop])
```

Listing 18: Training with Early Stopping

The components of this code are:

- **`EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=`**
  - `monitor='val_loss'`: Watches the validation loss during training.
  - `patience=3`: If validation loss does not improve for 3 consecutive epochs, training stops.
  - `restore_best_weights=True`: After stopping, the model restores the weights from the best-performing epoch instead of the last one.
- **`callbacks=[early_stop]`**: Passes the early stopping mechanism to the model training process.

## 10.3 Comparison with Previous Training Method

Previously, the model was trained for a fixed number of epochs without any stopping condition:

```
history = model.fit(train_generator,
                    validation_data=validation_generator,
                    epochs=10,
                    class_weight=class_weights)
```

Listing 19: Training Without Early Stopping

The key differences between the two approaches:

- **Without Early Stopping**: The model runs for a fixed number of epochs, even if the validation loss starts increasing, which may lead to overfitting.
- **With Early Stopping**: Training stops automatically when no further improvement in validation loss is observed, preventing overfitting and saving time.
- **Restoring Best Weights**: Ensures that the best-performing model weights are retained, which is not the case in the previous approach.

## 10.4 Benefits of Early Stopping

- Prevents overfitting by stopping training at the right moment.
- Saves computational resources by reducing unnecessary epochs.
- Restores the best weights, leading to a more optimal model.