Assignment 3

Rashik Mahmud Orchi-B00968298

16/06/2024

```
rm(list=ls())
```

As we found for natural language processing, R is a little bit limited when it comes to deep neural networks. Unfortunately, these networks are the current gold-standard methods for medical image analysis. The ML/DL R packages that do exist provide interfaces (APIs) to libraries built in other languages like C/C++ and Java. This means there are a few "non-reproducible" installation and data gathering steps to run first. It also means there are a few "compromises" to get a practical that will A) be runnable B) actually complete within the allotted time and C) even vaguely works. Doing this on a full real dataset would likely follow a similar process (admittedly with more intensive training and more expressive architectures). Google colab running an R kernel hasn't proven a very effective alternative thus far either.

Install packages

In this practical, we will be performing image classification on real medical images using the Torch deep learning library and several packages to manage and preprocess the images. We will utilize the following packages for our task:

`gridExtra`: This package provides functions for arranging multiple plots in a grid layout, allowing us to visualize and compare our results effectively.

`jpeg`: This package enables us to read and write JPEG images, which is a commonly used image format.

`imager`: This package offers a range of image processing functionalities, such as loading, saving, resizing, and transforming images.

`magick`: This package provides an interface to the ImageMagick library, allowing us to manipulate and process images using a wide variety of operations.

To ensure that the required packages are available, we can install them using the `install.packages()` function. Additionally, some packages may require additional dependencies to be installed. In such cases, we can use the `BiocManager::install()` function from the Bioconductor project to install the necessary dependencies.

Here is an example code snippet to install the required packages: you can uncomment the following code to install required packages

```
# Install necessary packages
#install.packages("gridExtra")
#install.packages("jpeg")
#install.packages("imager")
#install.packages("magick")

# Install Bioconductor package (if not already installed)
```

```
#if (!require("BiocManager", quietly = TRUE))
#    install.packages("BiocManager")

# Install EBImage package from Bioconductor
#BiocManager::install("EBImage")
#install.packages("abind")

# Install torch, torchvision, and luz
#install.packages("torch")
#install.packages("torchvision")
#install.packages("luz")
```

```
# load packages
knitr::opts_chunk$set(echo = TRUE)
suppressMessages({library(ggplot2)
library(gridExtra)
library(imager)
library(jpeg)
library(magick)
library(EBImage)
library(grid)
library(dplyr)
library(abind)
})
```

If using Mac OSX: imager needs Xquartz - If using homebrew: `brew install --cask xquartz`

Diagnosing Pneumonia from Chest X-Rays

Parsing the data

Today, we are going to look at a series of chest X-rays from children (from this paper) and see if we can accurately diagnose pneumonia from them.

Chest X-ray images (anterior-posterior) were selected from retrospective cohorts of pediatric patients of one to five years old from Guangzhou Women and Children's Medical Center, Guangzhou. All chest X-ray imaging was performed as part of patients' routine clinical care. For the analysis of chest x-ray images, all chest radiographs were initially screened for quality control by removing all low quality or unreadable scans. The diagnoses for the images were then graded by two expert physicians before being cleared for inclusion. In order to account for any grading errors, the evaluation set was also checked by a third expert.

We can download, unzip, then inspect the format of this dataset as follows:

"https://drive.google.com/file/d/14H_FilWf12ONOJ_G4vvzNDDGvY7Ccqtm/view?usp=sharing"

you can **unzip** the file in a data directory, using the following code: *uncomment it*

```
# Specify the path to the zip file
zip_file <- "lab3_chest_xray.zip"

# Specify the destination directory to extract the files
destination_dir <- "data"
```

```r
# Extract the zip file
unzip(zip_file, exdir = destination_dir)
```

As with every data analysis, the first thing we need to do is learn about our data. If we are diagnosing pneumonia, we should use the internet to better understand what pneumonia actually is and what types of pneumonia exist.

0__ What is pneumonia and what is the point/benefit of being able to identify it from X-rays automatically?

Pneumonia is an infection that inflames the air sacs in one or both lungs (Mayo Clinic,2020). These air sacs, called alveoli, fill with fluid or pus (infected material), making it difficult to breathe. Pneumonia can be caused by bacteria, viruses, or fungi, and it's a serious illness that can be life-threatening, particularly for young children and older adults(WHO,2024).

The benefit of automatically identifying pneumonia from X-rays are discussed below:

1. **Enhanced Diagnostic Accuracy and Speed:** Automated systems for identifying pneumonia from X-rays can significantly improve diagnostic accuracy and speed. These systems utilize advanced machine learning algorithms to analyze X-ray images and detect signs of pneumonia with high precision. This reduces the risk of misdiagnosis and ensures timely treatment, which is crucial for patient outcomes (Rajpurkar et al., 2017).

2. **Early Detection and Intervention:** Early detection of pneumonia is crucial for effective treatment and reducing complications. Automated systems can identify subtle signs of pneumonia that may be missed by the human eye, facilitating earlier intervention. This is particularly beneficial in emergency settings where rapid decision-making is required (Annarumma et al., 2019).

3. **Consistency and Objectivity:** Human interpretation of X-rays can be subjective and vary between radiologists. Automated systems provide consistent and objective analysis, reducing variability in diagnoses. This consistency is vital for ensuring that all patients receive the same standard of care regardless of where or by whom they are treated (Lakhani & Sundaram, 2017).

4. **Increased Efficiency and Reduced Workload:** Radiologists often face heavy workloads and time constraints, leading to potential delays in diagnosis. Automated detection systems can handle large volumes of X-rays quickly and efficiently, alleviating the burden on radiologists. This allows healthcare providers to manage their workload more effectively and focus on more complex cases (Esteva et al., 2017).

5. **Accessibility in Resource-Limited Settings:** In many parts of the world, there is a shortage of trained radiologists. Automated pneumonia detection systems can be deployed in resource-limited settings to assist healthcare providers in diagnosing pneumonia. This can improve access to quality healthcare and reduce health disparities (Qin et al., 2018).

Exploring dataset

In the provided code snippet, we are exploring the dataset directory structure for a chest x-ray image classification task.

Make sure to put lab3_chest_xray directory in your R project directory.

```r
data_folder = "lab3_chest_xray"

files <- list.files(data_folder, full.names=TRUE, recursive=TRUE)
sort(sample(files, 20))
```

```
##  [1] "lab3_chest_xray/test/NORMAL/NORMAL2-IM-0030-0001.jpeg"
##  [2] "lab3_chest_xray/test/NORMAL/NORMAL2-IM-0060-0001.jpeg"
##  [3] "lab3_chest_xray/test/PNEUMONIA/person147_bacteria_706.jpeg"
##  [4] "lab3_chest_xray/train/NORMAL/IM-0156-0001.jpeg"
##  [5] "lab3_chest_xray/train/NORMAL/IM-0235-0001.jpeg"
##  [6] "lab3_chest_xray/train/NORMAL/IM-0351-0001.jpeg"
##  [7] "lab3_chest_xray/train/NORMAL/IM-0437-0001-0002.jpeg"
##  [8] "lab3_chest_xray/train/NORMAL/IM-0557-0001.jpeg"
##  [9] "lab3_chest_xray/train/NORMAL/IM-0575-0001.jpeg"
## [10] "lab3_chest_xray/train/PNEUMONIA/person1028_bacteria_2959.jpeg"
## [11] "lab3_chest_xray/train/PNEUMONIA/person1048_bacteria_2982.jpeg"
## [12] "lab3_chest_xray/train/PNEUMONIA/person1049_virus_1746.jpeg"
## [13] "lab3_chest_xray/train/PNEUMONIA/person1069_virus_1772.jpeg"
## [14] "lab3_chest_xray/train/PNEUMONIA/person1076_bacteria_3016.jpeg"
## [15] "lab3_chest_xray/train/PNEUMONIA/person1107_virus_1831.jpeg"
## [16] "lab3_chest_xray/train/PNEUMONIA/person1138_virus_1879.jpeg"
## [17] "lab3_chest_xray/train/PNEUMONIA/person1151_virus_1928.jpeg"
## [18] "lab3_chest_xray/train/PNEUMONIA/person120_virus_226.jpeg"
## [19] "lab3_chest_xray/train/PNEUMONIA/person1200_virus_2042.jpeg"
## [20] "lab3_chest_xray/train/PNEUMONIA/person124_virus_249.jpeg"
```

In the output above, you can see the filenames for all the chest X-rays. Look carefully at the filenames for the X-rays showing pneumonia (i.e., those in `{train,test}/PNEUMONIA`).

1__ Do these filenames tell you anything about pneumonia? Why might this make predicting pneumonia more challenging?

The file names of the chest X-ray images do provide some clues about the contents and context of the images, particularly regarding the presence or absence of pneumonia. However, it provides us with an additional critical information regarding the cause of pneumonia such as bacteria or virus, indicating bacterial or viral pneumonia, respectively.

Here's why it might make predicting pneumonia more challenging:

1. Pneumonia can present differently depending on the type (bacterial vs. viral) and the severity of the infection. This variability can be seen in the filenames that differentiate between "bacteria" and "virus".Hence, the model needs to be robust enough to recognize these different manifestations of pneumonia, which can have varying appearances on X-rays.
2. The file names suggest that the data set contains both bacterial and viral pneumonia, which might require the model to distinguish not just between normal and pneumonia but also between different types of pneumonia.This adds complexity to the classification task, as the model needs to learn to differentiate between subcategories within the pneumonia class.
3. The file names suggest that there might be a different number of images in the "NORMAL" and "PNEUMONIA" categories which could lead to class imbalance. As as result it would the result a biased model where the algorithm becomes better at identifying the more frequent class (e.g., normal) and less accurate at identifying the less frequent class (e.g., pneumonia).
4. Images may come from different machines or hospitals, leading to variability in image quality and characteristics.

```r
# Exploring dataset
base_dir <- "lab3_chest_xray"

train_pneumonia_dir <- file.path(base_dir, "train", "PNEUMONIA")
```

```r
train_normal_dir <- file.path(base_dir, "train", "NORMAL")

test_pneumonia_dir <- file.path(base_dir, "test", "PNEUMONIA")
test_normal_dir <- file.path(base_dir, "test", "NORMAL")

val_normal_dir <- file.path(base_dir, "validate", "NORMAL")
val_pneumonia_dir <- file.path(base_dir, "validate", "PNEUMONIA")

train_pn <- list.files(train_pneumonia_dir, full.names = TRUE)
train_normal <- list.files(train_normal_dir, full.names = TRUE)

test_normal <- list.files(test_normal_dir, full.names = TRUE)
test_pn <- list.files(test_pneumonia_dir, full.names = TRUE)

val_pn <- list.files(val_pneumonia_dir, full.names = TRUE)
val_normal <- list.files(val_normal_dir, full.names = TRUE)

cat("Total Images:", length(c(train_pn, train_normal, test_normal, test_pn, val_pn, val_normal)), "\n")
```

```
## Total Images: 1216
```

```r
cat("Total Pneumonia Images:", length(c(train_pn, test_pn, val_pn)), "\n")
```

```
## Total Pneumonia Images: 608
```

```r
cat("Total Normal Images:", length(c(train_normal, test_normal, val_normal)), "\n")
```

```
## Total Normal Images: 608
```

Creating training datasets

The provided code segment focuses on creating datasets for training, testing, and validation, as well as assigning labels to the corresponding datasets. Additionally, the code shuffles the data to introduce randomness in the order of the samples. Here's a breakdown of the code:

`train_dataset`: Combines the lists `train_pn` and `train_normal` to create a single dataset for training.

`train_labels`: Creates a vector of labels for the training dataset by repeating "pneumonia" for the length of `train_pn` and "normal" for the length of `train_normal`.

`shuffled_train_dataset`, `shuffled_train_labels`: Extracts the shuffled training dataset and labels from the shuffled `train_data` data frame.

By creating datasets and assigning labels, this code prepares the data for subsequent steps, such as model training and evaluation. The shuffling of the data ensures that the samples are presented in a random order during training, which can help prevent any biases or patterns that may exist in the original dataset.

```r
train_dataset <- c(train_pn, train_normal)
train_labels <- c(rep("pneumonia", length(train_pn)), rep("normal", length(train_normal)))

test_dataset <- c(test_pn, test_normal)
test_labels <- c(rep("pneumonia", length(test_pn)), rep("normal", length(test_normal)))
```

```
val_dataset <- c(val_pn, val_normal)
val_labels <- c(rep("pneumonia", length(val_pn)), rep("normal", length(val_normal)))

# Create a data frame with the dataset and labels
train_data <- data.frame(dataset = train_dataset, label = train_labels)
test_data <- data.frame(dataset = test_dataset, label = test_labels)
val_data <- data.frame(dataset = val_dataset, label = val_labels)

# Shuffle the data frame
train_data <- train_data[sample(nrow(train_data)), ]
test_data <- test_data[sample(nrow(test_data)), ]
val_data <- val_data[sample(nrow(val_data)), ]

# Extract the shuffled dataset and labels
shuffled_train_dataset <- train_data$dataset
shuffled_train_labels <- train_data$label

shuffled_test_dataset <- test_data$dataset
shuffled_test_labels <- test_data$label

shuffled_val_dataset <- val_data$dataset
shuffled_val_labels <- val_data$label
```

```
#showing a file name from test set
cat("file name: ", shuffled_train_dataset[5], "\nlabel: ", shuffled_train_labels[5])
```

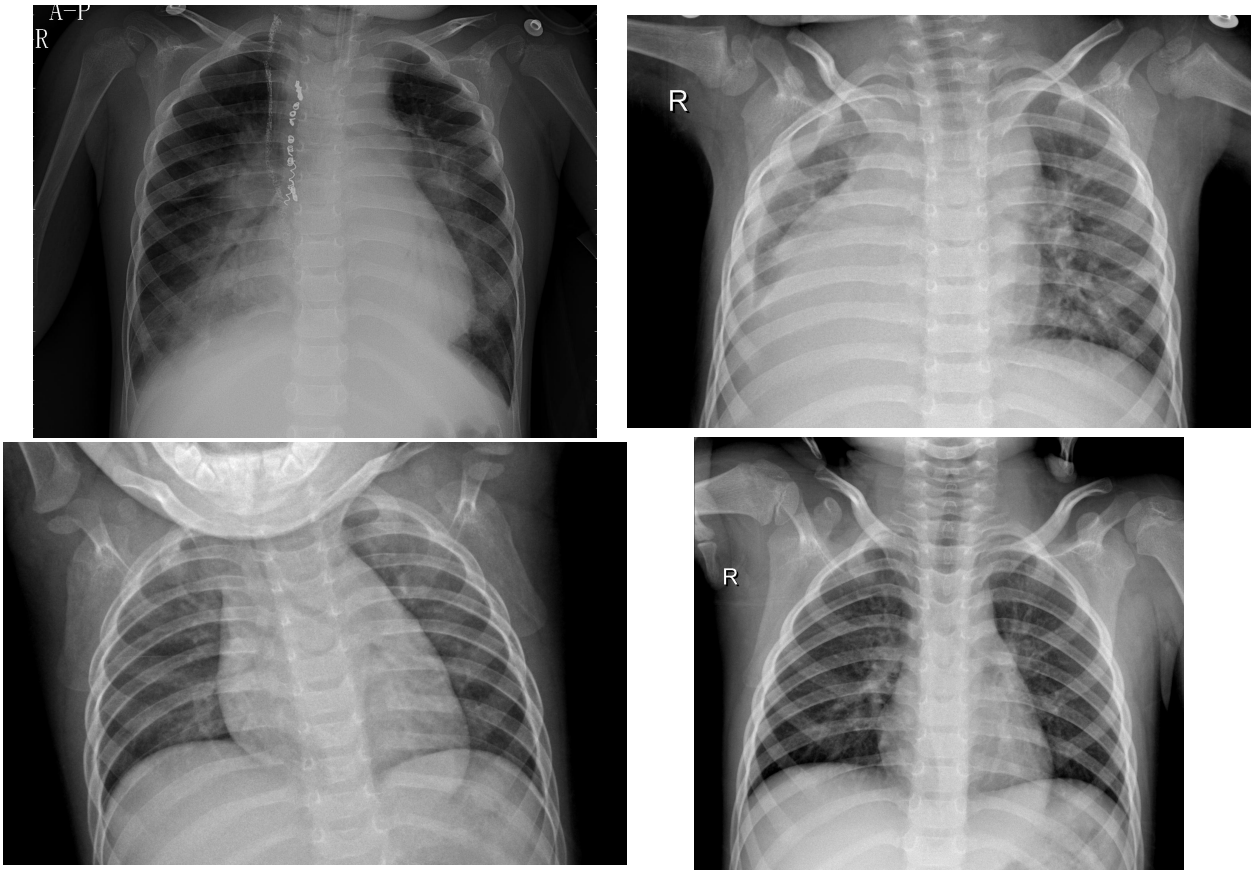
```
## file name:  lab3_chest_xray/train/NORMAL/IM-0600-0001.jpeg
## label:  normal
```

Data Visualization

Let's inspect a couple of these files as it is always worth looking directly at data.

```
# Create a list to store the ggplot objects
plots <- list()

# Iterate through the images and labels
for (i in 1:4) {

  image <- readImage(shuffled_train_dataset[i])

  # Create a ggplot object for the image with the corresponding label
  plot <- ggplot() +
    theme_void() +
    annotation_custom(
      rasterGrob(image, interpolate = TRUE),
      xmin = -Inf, xmax = Inf, ymin = -Inf, ymax = Inf
    )

  # Add the ggplot object to the list
  plots[[i]] <- plot
```

```
}

# Arrange the plots in a 2x2 grid
grid.arrange(grobs = plots, nrow = 2, ncol = 2)
```



2__ From looking at only 3 images do you see any attribute of the images that we may have to normalise before training a model? From examining the provided images, there are a few attributes that should be normalized before training a model:

1. Brightness and Contrast: The images have varying levels of brightness and contrast. Normalizing these aspects can help ensure that the model does not become biased towards certain lighting conditions.

2. Size and Scale: The images should be resized to a consistent size, ensuring that the scale of anatomical features is similar across all images. This helps the model focus on relevant features rather than differences in image dimensions.
3. Orientation:All images should have a consistent orientation. From the sample images, we can see that the images are not aligned with a consistent orientation. Instead, they are rotated at some angle from one another. Therefore, they should be corrected to match the standard orientation.

4. Presence of Artifacts: Some images may contain medical devices or other artifacts that can introduce noise. Preprocessing should aim to minimize the impact of such artifacts.

Data Pre-processing

The provided code segment presents a function called `process_images` that performs several preprocessing steps on the images. Here's an explanation of each preprocessing step and its purpose:

1. Loading the "imager" library: This line imports the "imager" library, which provides functions for image processing.
2. Setting the desired image size: The variable `img_size` is initialized to 224, indicating the desired size (both width and height) of the processed images. This step ensures that all images are resized to a consistent size.
3. Initializing an empty list for processed images: The variable `X` is initialized as an empty list that will store the processed images.
4. Looping through each image path: The function iterates over each image path in the `shuffled_dataset` input, which represents the paths to the shuffled images.
5. Loading the image: The `imager::load.image()` function is used to read and load the image from its file path.
6. Normalizing the image: The loaded image is divided by 255 to normalize its pixel values. This step scales the pixel values between 0 and 1, which is a common practice in image processing and deep learning.
7. Resizing the image: The `resize()` function from the "imager" library is employed to resize the normalized image to the desired `img_size`. Resizing the images to a consistent size is important for ensuring compatibility with the subsequent steps of the deep learning pipeline.
8. Appending the processed image to the list: The processed image, stored in the variable `img_resized`, is added to the `X` list using the `c()` function and the `list()` function. The resulting `X` list will contain all the processed images.
9. Returning the processed images: Finally, the `X` list, which now holds the processed images, is returned as the output of the `process_images` function.

These preprocessing steps are commonly performed in image classification tasks to prepare the images for training a deep learning model. Normalizing the pixel values and resizing the images ensure that they are in a consistent format and range, which facilitates the learning process of the model. Additionally, resizing the images to a fixed size allows for efficient batch processing and ensures that all images have the same dimensions, enabling them to be fed into the model's input layer.

```
process_images <- function(shuffled_dataset) {

  img_size <- 224  # Desired image size

  # Initialize an empty list to store processed images
  X <- list()

  # Loop through each image path in shuffled_train_dataset
  for (image_path in shuffled_dataset) {
    # Read the image
    img <- imager::load.image(image_path)

    # Normalize the image
    img_normalized <- img / 255
```

```
    # Resize the image
    img_resized <- resize(img_normalized, img_size, img_size)

    # Append the processed image to the list
    X <- c(X, list(img_resized))
  }

  return(X)
}
```

In this section, by using the process_images function we will do preprocessing on the training, testing, and validation images.

Then we will encode the labels: The ifelse() function is utilized to encode the labels. If a label in shuffled_train_labels, shuffled_test_labels, or shuffled_val_labels is "normal," it is assigned a value of 1. Otherwise, if the label is "pneumonia," it is assigned a value of 2. This encoding scheme allows for easier handling of the labels in subsequent steps.

Finally, We Convert labels to integer type: The labels are converted to the integer data type using the as.integer() function. This ensures that the labels are represented as integers, which is the expected format for the target tensor when using the nn_cross_entropy_loss function.

It is important to note that when using the nn_cross_entropy_loss function in R, the target tensor is expected to have a "long" data type, which is equivalent to the integer type in R. Hence, the labels need to be converted to integers.

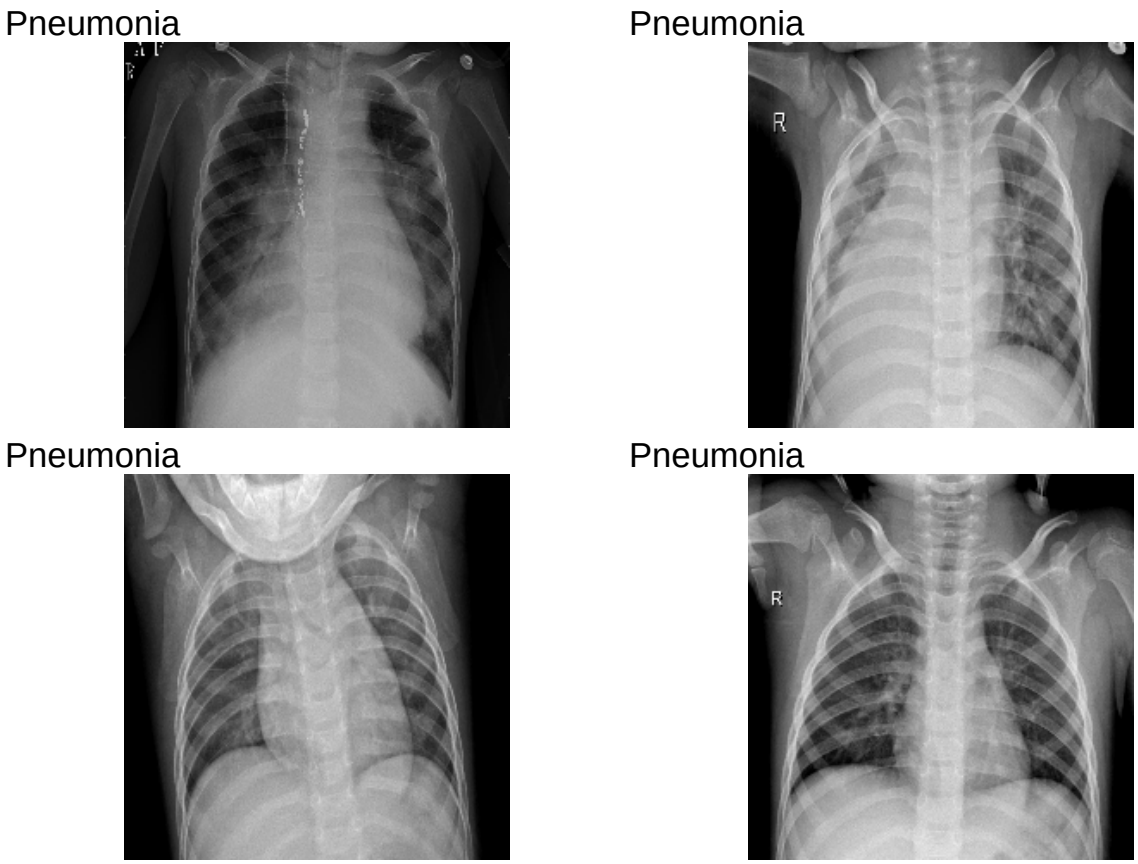
Furthermore, when working with the Torch package in R, it is essential to ensure that labels start from 1 instead of 0. In binary classification problems, the labels should be 1 and 2, representing the two classes. This adjustment is necessary to avoid errors when using the labels as indices for the output tensor.

```
train_X <- process_images(shuffled_train_dataset)
test_X <- process_images(shuffled_test_dataset)
val_X <- process_images(shuffled_val_dataset)

train_y <- ifelse(shuffled_train_labels == "normal", 1, 2)
test_y <- ifelse(shuffled_test_labels == "normal", 1, 2)
val_y <- ifelse(shuffled_val_labels == "normal", 1, 2)

train_y <- as.integer(train_y)
test_y <- as.integer(test_y)
val_y <- as.integer(val_y)
```

Now let's have an other look at images after doing pre processing.

```
# Create a list to store the ggplot objects
plots <- list()

# Iterate through the images and labels
for (i in 1:4) {
  if (train_y[i] == 0) {
    label <- "Normal"
  } else {
    label <- "Pneumonia"
  }
```

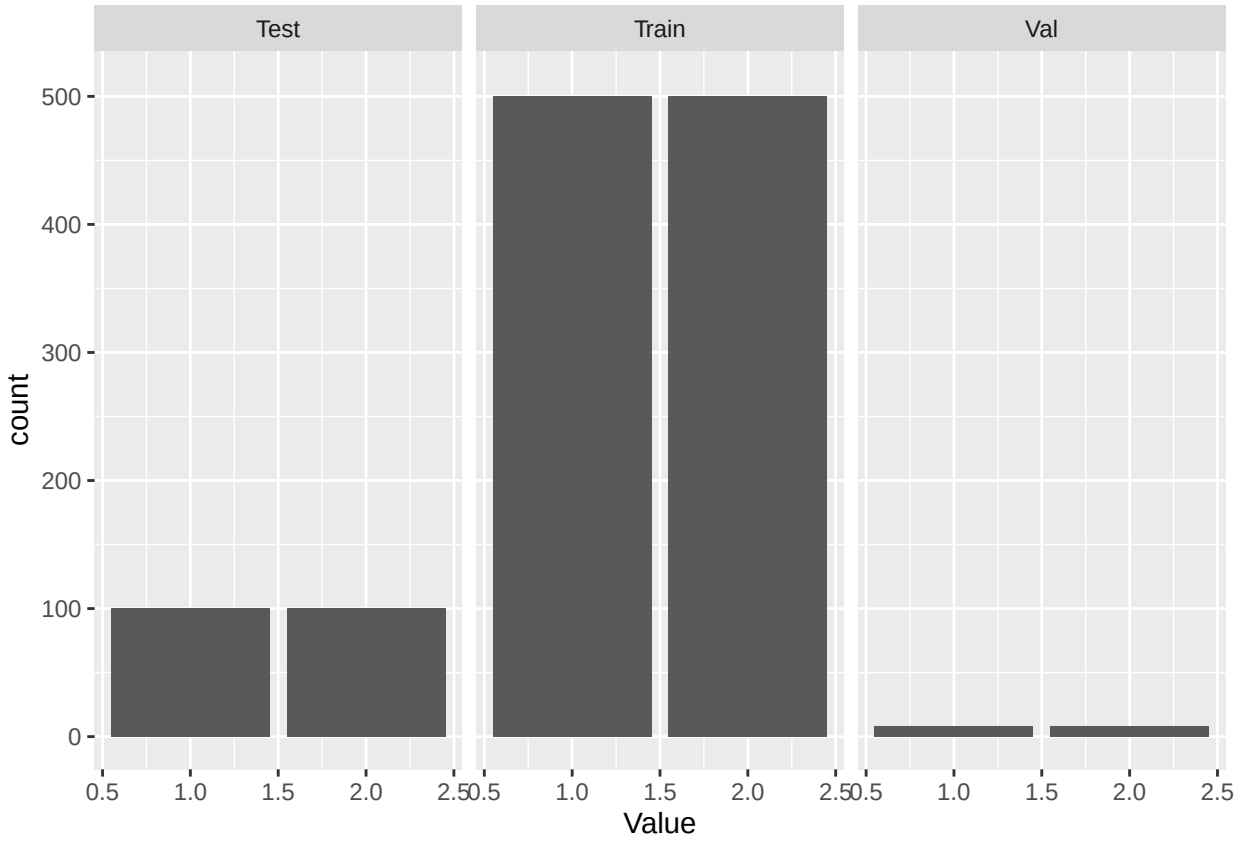
```
  # Create a ggplot object for the image with the corresponding label
  plot <- ggplot() +
    theme_void() +
    ggtitle(label) +
    annotation_custom(
      rasterGrob(train_X[[i]], interpolate = TRUE),
      xmin = -Inf, xmax = Inf, ymin = -Inf, ymax = Inf
    )

  # Add the ggplot object to the list
  plots[[i]] <- plot
}

# Arrange the plots in a 2x2 grid
grid.arrange(grobs = plots, nrow = 2, ncol = 2)
```



Let's count and display the number of images in each dataset to examine the distribution of labels in the data. This will help us understand the ratio of "normal" and "pneumonia" labels in the dataset.

```
# Combine train, test, and val vectors into a single data frame
df <- data.frame(
  Data = rep(c("Train", "Test", "Val"), times = c(length(train_y), length(test_y), length(val_y))),
  Value = c(train_y, test_y, val_y)
)
```

```r
# Create a single bar plot with facets
fig <- ggplot(df, aes(x = Value)) +
  geom_bar() +
  ylim(0, 510) +
  facet_wrap(~Data, ncol = 3)

# Arrange the plot
grid.arrange(fig, nrow = 1)
```



As you can see, the provided dataset is completely balanced in all sets. **3_** if the dataset was not balanced, what kind of techniques could be useful? When dealing with an imbalanced dataset, where the number of samples in different classes is not equal, several techniques can be applied to handle the imbalance and improve model performance. Here are some common methods:

1. **Resampling Techniques**

a. Oversampling:

i) Synthetic Minority Over-sampling Technique (SMOTE): Generates synthetic samples for the minority class by interpolating between existing samples.

ii) Random Oversampling: Duplicates existing samples in the minority class to increase its representation.

b. Under sampling:

i) Random Undersampling: Removes samples from the majority class to balance the dataset.

ii) Cluster Centroids: Reduces the number of samples by clustering and using the cluster centroids as representatives.

2. **Algorithmic Approaches**

a. Cost-sensitive Learning: Modify the learning algorithm to pay more attention to the minority class by assigning a higher cost to misclassifying minority class samples. Many machine learning libraries, like Scikit-learn, allow setting class weights for this purpose.

b. Anomaly Detection Models: Treat the minority class as anomalies or outliers and use anomaly detection algorithms.

3. **Ensemble Methods**

a. Balanced Random Forest: Use balanced random forests, where each decision tree is trained on a balanced subset of the data.

b. Boosting Algorithms: Algorithms like AdaBoost, Gradient Boosting, and XGBoost can be adjusted to handle class imbalance by focusing more on the harder-to-classify samples.

4. **Data Augmentation:** Applying transformations (e.g., rotations, translations, flips) to images in the minority class to create additional training samples.

Training

In the next step, we need to reshape the dataset to ensure it is in the appropriate format for feeding into deep learning models. Reshaping involves modifying the structure and dimensions of the data to match the expected input shape of the model.

```
train_X <- array(data = unlist(train_X), dim = c(1000, 224, 224, 1))
test_X <- array(data = unlist(test_X), dim = c(200, 224, 224, 1))
val_X <- array(data = unlist(val_X), dim = c(16, 224, 224, 1))
```

```
print(dim(train_X))
```

```
## [1] 1000  224  224    1
```

```
print(length(train_y))
```

```
## [1] 1000
```

```r
print(dim(test_X))
```

```
## [1] 200 224 224   1
```

```r
print(length(test_y))
```

```
## [1] 200
```

```r
print(dim(val_X))
```

```
## [1]  16 224 224   1
```

```r
print(length(val_y))
```

```
## [1] 16
```

The last dimension of an image represents the color channels, typically RGB (Red, Green, Blue) in color images or grayscale in black and white images. In our case, since we are working with black and white images, there is only one channel, hence the value 1.

However, the deep learning framework expects the input tensor to have a specific shape, with the color channels as the second dimension. The desired shape is (batch_size x channels x height x width), but our current data has the shape (batch_size x height x width x channels).

To rearrange the dimensions of our data to match the expected shape, we use the aperm function in R. The aperm function allows us to permute the dimensions of an array. In this case, we are permuting the dimensions of train_X to change the order of the dimensions, so that the channels dimension becomes the second dimension.

```r
train_X <- aperm(train_X, c(1,4,2,3))
test_X <- aperm(test_X, c(1,4,2,3))
val_X <- aperm(val_X,c(1,4,2,3))

dim(train_X)
```

```
## [1] 1000    1  224  224
```

In order to train a Convolutional Neural Network (CNN), we will utilize the torch package in R. Torch is a powerful deep learning library that provides a wide range of functions and tools for building and training neural networks.

CNNs are particularly effective for image-related tasks due to their ability to capture local patterns and spatial relationships within the data. Torch provides a high-level interface to define and train CNN models in R.

By using torch, we can leverage its extensive collection of pre-built layers, loss functions, and optimization algorithms to construct our CNN architecture. We can define the network structure, specifying the number and size of convolutional layers, pooling layers, and fully connected layers.

```r
# Load the torch package
library(torch)
library(torchvision)
library(luz)
```

In this code, we are defining a custom dataset class called "ImageDataset" to encapsulate our training, testing, and validation data. The purpose of the dataset class is to provide a structured representation of our data that can be easily consumed by deep learning models.

The "ImageDataset" class has three main functions: 1. "initialize": This function is called when creating an instance of the dataset class. It takes the input data (X) and labels (y) as arguments and stores them as tensors. 2. ".getitem": This function is responsible for retrieving a single sample and its corresponding label from the dataset. Given an index (i), it returns the i-th sample and label as tensors. 3. ".length": This function returns the total number of samples in the dataset.

After defining the dataset class, we create instances of it for our training, testing, and validation data: "train_dataset", "test_dataset", and "val_dataset". We pass the respective input data and labels to each dataset instance.

Next, we create dataloader objects for each dataset. A dataloader is an abstraction that allows us to efficiently load and iterate over the data in batches during the training process. The batch size (16 in this case) determines the number of samples that will be processed together in each iteration. It helps in optimizing memory usage and can speed up the training process by leveraging parallel computation.

Finally, the code visualizes the size of the first batch by calling "batch[[1]]$size()". This can be useful for understanding the dimensions of the data and ensuring that the input shapes are consistent with the network architecture.

```r
# Define a custom dataset class
ImageDataset <- dataset(
  name = "ImageDataset",
  initialize = function(X, y) {
    # Store the data as tensors
    self$data <- torch_tensor(X)
    self$labels <- torch_tensor(y)
  },
  .getitem = function(i) {
    # Return a single sample and label
    x <- self$data[i,,,]
    y <- self$labels[i]
    list(x = x, y = y)
  },
  .length = function() {
    # Return the number of samples
    dim(self$data)[1]
  }
)

# Create a dataset object from your data
train_dataset <- ImageDataset(train_X, train_y)
test_dataset <- ImageDataset(test_X, test_y)
val_dataset <- ImageDataset(val_X, val_y)

# Create a dataloader object from your dataset
train_dataloader <- dataloader(train_dataset, batch_size = 16)
```

```
test_dataloader <- dataloader(test_dataset, batch_size = 16)
val_dataloader <- dataloader(val_dataset, batch_size = 16)

# Iterate over batches of data
batch = train_dataloader$.iter()$.next()

# Visualize the first batch size
batch[[1]]$size()
```

```
## [1]  16   1 224 224
```

creat the CNN model

The input image has one channel and a size of 224 x 224 pixels. The first convolutional layer has 32 filters with a kernel size of 3 x 3 and a stride of 1. The output of this layer has a size of 32 x 222 x 222. The second convolutional layer has 64 filters with the same kernel size and stride. The output of this layer has a size of 64 x 220 x 220. The max pooling layer has a kernel size of 2 x 2 and reduces the spatial dimensions by half. The output of this layer has a size of 64 x 110 x 110. The dropout layer randomly sets some elements to zero with a probability of 0.25. The flatten layer reshapes the output into a vector with a length of 774400. The first fully connected layer has 128 neurons and applies a ReLU activation function. The second dropout layer randomly sets some elements to zero with a probability of 0.5. The second fully connected layer has 2 neurons and produces the final output for the classification task.

Input Image: 1 channel, 224 x 224

Layer Type	Output Size	Parameters
Conv2D	32 x 222 x 222	32 x 3 x 3 (weights)
Conv2D	64 x 220 x 220	64 x 3 x 3 (weights)
MaxPooling2D	64 x 110 x 110	2 x 2 (kernel size)
Dropout	64 x 110 x 110	0.25 (dropout rate)
Flatten	774400	-
FullyConnected (ReLU)	128	-
Dropout	128	0.5 (dropout rate)
FullyConnected	2	-

```
net <- nn_module(
  "Net",

  initialize = function() {
    self$conv1 <- nn_conv2d(1, 32, 3, 1)
    self$conv2 <- nn_conv2d(32, 64, 3, 1)
    self$dropout1 <- nn_dropout2d(0.25)
    self$dropout2 <- nn_dropout2d(0.5)
    self$fc1 <- nn_linear(774400, 128)  # Adjust the input size based on your image dimensions
    self$fc2 <- nn_linear(128, 2)              # Change the output size to match your classification tas
```

```
  },

  forward = function(x) {
    x %>%                                       # N * 1 * 224 * 224
      self$conv1() %>%                          # N * 32 * 222 * 222
      nnf_relu() %>%
      self$conv2() %>%                          # N * 64 * 220 * 220
      nnf_relu() %>%
      nnf_max_pool2d(2) %>%                     # N * 64 * 110 * 110
      self$dropout1() %>%
      torch_flatten(start_dim = 2) %>%          # N * 64 * 110 * 110 --> N * 774400
      self$fc1() %>%                            # N * 128
      nnf_relu() %>%
      self$dropout2() %>%
      self$fc2()                                # N * 2 (change the output size to match your classifi
  }
)
```

Train model

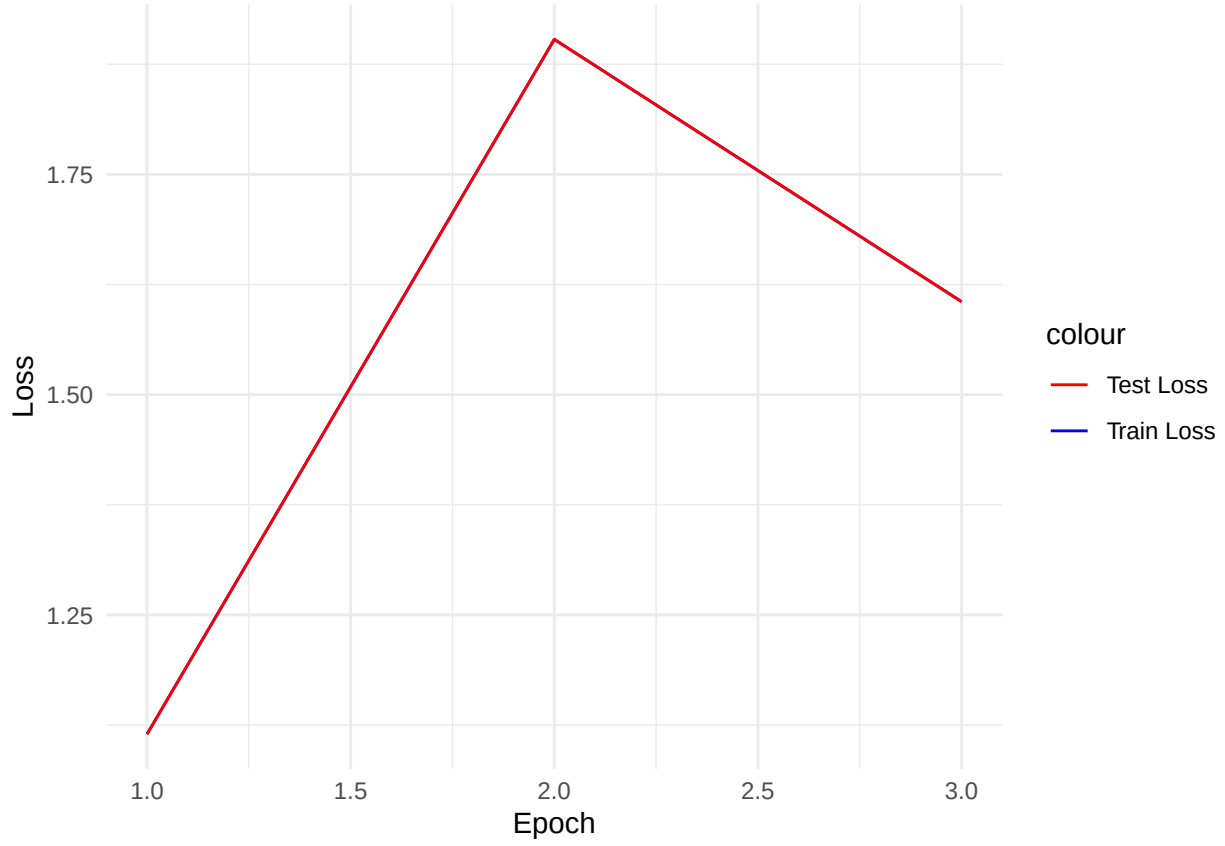
```
# Set the number of epochs
num_epochs <- 3

train_loss <- numeric(num_epochs)
train_acc <- numeric(num_epochs)
test_loss <- numeric(num_epochs)
test_acc <- numeric(num_epochs)

# Loop through the epochs
for (epoch in 1:num_epochs) {
  # Perform training and validation for each epoch
  fitted <- net %>%
    setup(
      loss = nn_cross_entropy_loss(),
      optimizer = optim_adam,
      metrics = list(
        luz_metric_accuracy()
      )
    ) %>%
    fit(train_dataloader, epochs = 1, valid_data = test_dataloader)

  # Print the metrics for the current epoch
  cat("Epoch ", epoch, "/", num_epochs, "\n")
  cat("Train metrics: Loss: ", fitted$records$metrics$train[[1]]$loss, " - Acc: ", fitted$records$metri
  cat("Valid metrics: Loss: ", fitted$records$metrics$valid[[1]]$loss, " - Acc: ", fitted$records$metri
  cat("\n")

  # Store the loss and accuracy values
  train_loss[epoch] <- fitted$records$metrics$train[[1]]$loss
  train_acc[epoch] <- fitted$records$metrics$train[[1]]$acc
  test_loss[epoch] <- fitted$records$metrics$train[[1]]$loss
```

```
  test_acc[epoch] <- fitted$records$metrics$valid[[1]]$acc
}
```

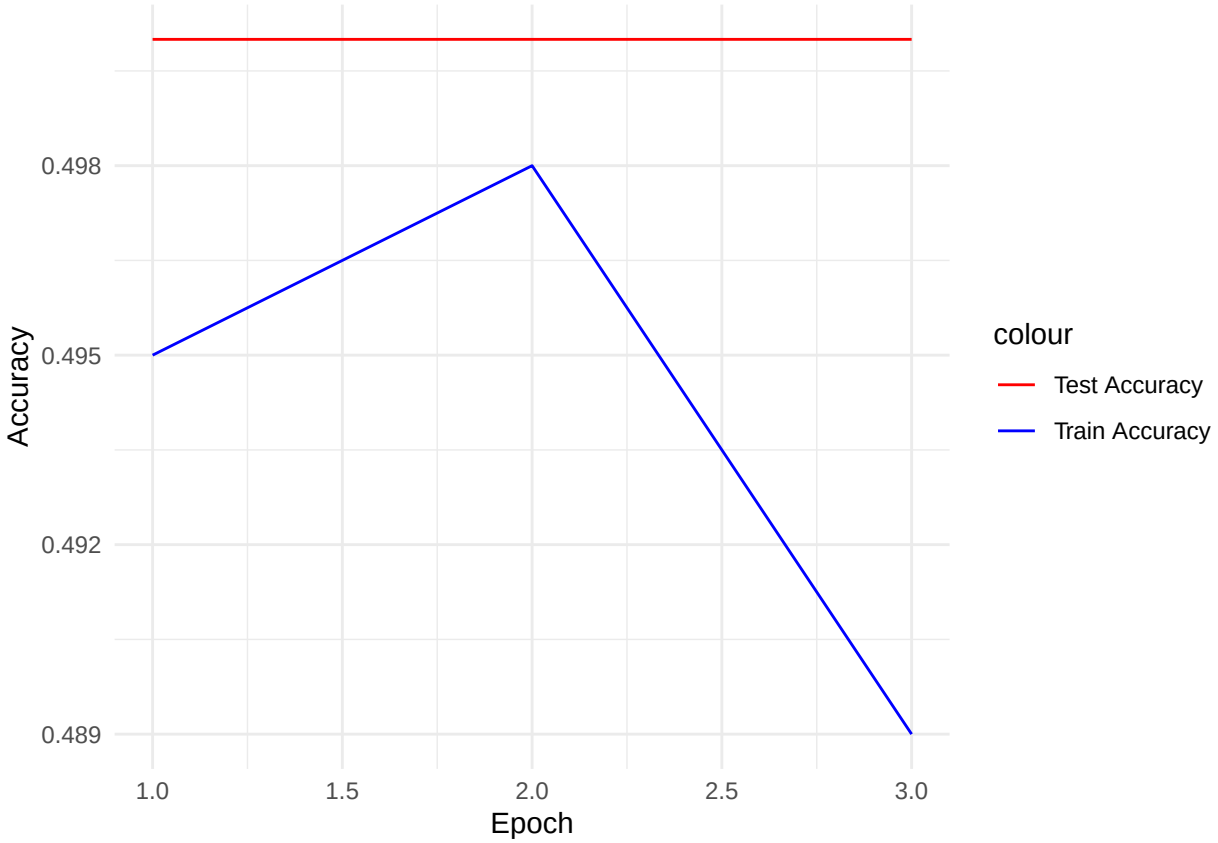
```
## Epoch  1 / 3
## Train metrics: Loss:  1.114643  - Acc:  0.495
## Valid metrics: Loss:  0.6929638  - Acc:  0.5
##
## Epoch  2 / 3
## Train metrics: Loss:  1.90326  - Acc:  0.498
## Valid metrics: Loss:  0.6944985  - Acc:  0.5
##
## Epoch  3 / 3
## Train metrics: Loss:  1.605521  - Acc:  0.489
## Valid metrics: Loss:  0.6933644  - Acc:  0.5
```

Plot learning curves

```
# Plot the train and test loss
loss_df <- data.frame(
  Epoch = 1:num_epochs,
  Train_Loss = train_loss,
  Test_Loss = test_loss
)
loss_plot <- ggplot(data = loss_df) +
  geom_line(aes(x = Epoch, y = Train_Loss, color = "Train Loss")) +
  geom_line(aes(x = Epoch, y = Test_Loss, color = "Test Loss")) +
  labs(x = "Epoch", y = "Loss") +
  scale_color_manual(values = c("Train Loss" = "blue", "Test Loss" = "red")) +
  theme_minimal()

# Plot the train and test accuracy
acc_df <- data.frame(
  Epoch = 1:num_epochs,
  Train_Accuracy = train_acc,
  Test_Accuracy = test_acc
)
acc_plot <- ggplot(data = acc_df) +
  geom_line(aes(x = Epoch, y = Train_Accuracy, color = "Train Accuracy")) +
  geom_line(aes(x = Epoch, y = Test_Accuracy, color = "Test Accuracy")) +
  labs(x = "Epoch", y = "Accuracy") +
  scale_color_manual(values = c("Train Accuracy" = "blue", "Test Accuracy" = "red")) +
  theme_minimal()

# Print the plots
print(loss_plot)
```




```
print(acc_plot)
```



4__ Based on the training and test accuracy, is this model actually managing to classify X-rays into pneumonia vs normal? What do you think contributes to this? why?

Both the training and validation accuracy hover around 50%, which is equivalent to random guessing.This indicates that the model is not learning effectively to distinguish between pneumonia and normal cases.

Several issues could be contributing to the poor performance of the model such as: 1. The images may not be properly normalized. X-ray images can vary in intensity, and normalization helps in standardizing these variations.
2. Lack of sufficient data augmentation might cause the model to overfit or not generalize well. 3. Even though the dataset appears balanced (608 pneumonia images and 608 normal images), the challenge might still lie in the complexity and variations within the pneumonia cases (e.g., different types of pneumonia: bacterial, viral). 4. The model architecture might not be complex enough to capture the underlying patterns in the X-ray images. A deeper network or one with more parameters might be necessary. 5. The learning rate might be too high or too low, preventing the model from converging to an optimal solution. 6. The number of epochs might be insufficient for the model to learn effectively. 7. X-ray images require effective feature extraction methods. 8. The current method is not extracting relevant features.

5__ what is your suggestions to solve this problem? How could we improve this model?

Few suggestions to improve to model are:

1. Adjusting the learning rate to find an optimal value.A high learning rate might cause instability, while a low learning rate can slow down learning.
2. Using a learning rate scheduler to adjust the learning rate dynamically during training.
3. Apply data augmentation techniques to increase the variability in the training dataset, helping the model generalize better. Techniques include rotations, flips, zooms, and shifts.

4. Using a more complex architecture if the current model is too simple.
5. Fine-tuning a pre-trained model (such as ResNet, VGG) on your dataset, which often yields better performance.
6. Checking for class imbalance in the category of the pneumonia whether caused by bateria or virus.If there is class imbalance, then using techniques such as oversampling the minority class, undersampling the majority class, or using class weights to balance the loss function.

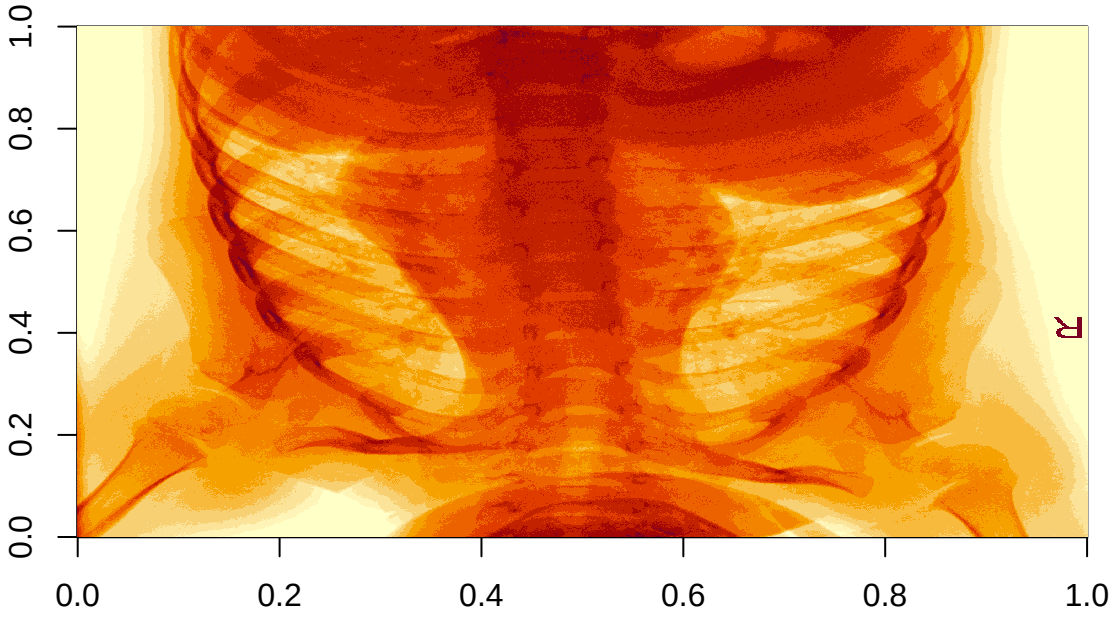
Data Augmentation

Data augmentation is one of the solutions to consider when facing training difficulties. Data augmentation involves applying various transformations or modifications to the existing training data, effectively expanding the dataset and introducing additional variations. This technique can help improve model performance and generalization by providing the model with more diverse examples to learn from.

By applying data augmentation, we can create new training samples with slight modifications, such as random rotations, translations, flips, zooms, or changes in brightness and contrast. These modifications can mimic real-world variations and increase the model's ability to handle different scenarios. Data augmentation increased dataset size, improved generalization, and reduced overfitting.

Below is an example that demonstrates how we can augment the data.

```
img <- readImage(shuffled_train_dataset[5])

T_img <- torch_squeeze(torch_tensor(img)) %>%
         # Randomly change the brightness, contrast and saturation of an image
         transform_color_jitter() %>%
         # Horizontally flip an image randomly with a given probability
         transform_random_horizontal_flip() %>%
         # Vertically flip an image randomly with a given probability
         transform_random_vertical_flip(p = 0.5)

image(as.array(T_img))
```



We can also add the data transformations to our dataloader:

```
# Define a custom dataset class with transformations
ImageDataset_augment <- dataset(
  name = "ImageDataset",
  initialize = function(X, y, transform = NULL) {
    self$transform <- transform
    self$data <- X
    self$labels <- y
  },
  .getitem = function(i) {
    # Return a single sample and label
    x <- self$data[i,,,]
    x <- self$transform(x)
    y <- self$labels[i]

    list(x = x, y = y)
  },
  .length = function() {
    dim(self$data)[1]
  }
)

# Define the transformations for training data
train_transforms <- function(img) {
  img <- torch_squeeze(torch_tensor(img)) %>%
```

```
    transform_color_jitter() %>%
    transform_random_horizontal_flip() %>%
    transform_random_vertical_flip(p = 0.5) %>%
    torch_unsqueeze(dim = 1)

  return(img)
}

# Apply the transformations to your training dataset
train_dataset <- ImageDataset_augment(train_X, train_y, transform = train_transforms)
test_dataset <- ImageDataset(test_X, test_y)
val_dataset <- ImageDataset(val_X, val_y)

# Create a dataloader for training
train_dataloader <- dataloader(train_dataset, batch_size = 16)
test_dataloader <- dataloader(test_dataset, batch_size = 16)
val_dataloader <- dataloader(val_dataset, batch_size = 16)

# Iterate over batches of data
batch = train_dataloader$.iter()$.next()

# Visualize the first batch size
batch[[1]]$size()
```

```
## [1]  16   1 224 224
```

6__ What are the potential drawbacks or disadvantages of data augmentation? Data augmentation is a powerful technique, but it's not without limitations. Based on (Hallaj,2023), Here are some potential drawbacks to consider:

1. **Increased Computational Cost:** Training a model on a larger dataset ,even an augmented dataset requires more computational resources. This can be a significant drawback for complex models or when dealing with limited computational power.

2. **Overfitting to Augmented Data:** While data augmentation aims to reduce overfitting on the original data, using it excessively can lead to a different kind of overfitting. If the transformations applied to the data are too aggressive or unrealistic, the model might become overly specialized in recognizing these augmented patterns and fail to generalize well to unseen real-world data.

3. **Introducing Noise:** Data augmentation techniques can introduce noise into the data if not applied carefully. For instance, blurring an image might introduce artifacts that the model misinterprets as real features. This noise can lead to decreased model performance on the test set .

4. **Limited Effectiveness for Certain Tasks:** Data augmentation might not be very effective for tasks where the data is already quite diverse. For example, handwritten digits naturally exhibit some variation in rotation and scale due to human writing styles. In such cases, applying rotations and scaling for data augmentation might not significantly increase the model's ability to recognize these variations because the model might already be exposed to them in the original dataset.

Mobile net

Transfer learning is another technique in machine learning where knowledge gained from solving one problem is applied to a different but related problem. It involves leveraging pre-trained models that have been trained on large-scale datasets and have learned general features. By utilizing transfer learning, models can benefit from the knowledge and representations learned from these pre-trained models.

Torchvison provides versions of all the integrated architectures that have already been trained on the ImageNet dataset.

7__ What is ImageNet aka "ImageNet Large Scale Visual Recognition Challenge 2012"? How many images and classes does it involve? Why might this help us? ImageNet is a large visual database designed for use in visual object recognition software research. It contains millions of images that have been hand-annotated to indicate the objects they depict, and these annotations span a wide variety of object categories.(IMAGENET,2024)

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual computer vision competition. ILSVRC 2012 was a pivotal year for the challenge, as it marked a significant advancement in the field of deep learning and computer vision.

Details of ImageNet and ILSVRC 2012(Farrajota,2017) are:
Images: The ImageNet dataset used for the ILSVRC 2012 challenge contains over 1.2 million training images, 50,000 validation images, and 150,000 testing images. Classes: There are 1,000 different object categories or classes in the dataset, ranging from animals and plants to everyday objects.

For our project on classifying pneumonia and normal patients using deep neural networks, leveraging pre-trained models from ImageNet offers several advantages:

1. **Transfer Learning:** The pre-trained models on the ImageNet dataset can be fine-tuned for specific tasks such as pneumonia classification with fewer labeled images, leading to better performance and faster training times (Yosinski et al., 2014).By applying transfer learning, we can adapt the learned features from ImageNet to for our project, improving the model's ability to recognize patterns specific to pneumonia in chest X-rays.

2. **Efficiency:** Models trained on ImageNet learn hierarchical feature representations, from low-level edges and textures to high-level concepts. These features are useful for medical image analysis, where distinguishing subtle differences in texture and shape is crucial (Razavian et al., 2014).Using pre-trained models reduces the computational resources and time required to train deep networks from scratch.

3. **Accuracy:** Fine-tuning these models on your specific dataset can yield high accuracy in distinguishing between pneumonia and normal cases due to the robust feature extraction capabilities developed from the large and diverse ImageNet dataset.

MobileNet is a pre-trained model that can be effectively utilized in transfer learning scenarios. Do some research about this model.

8__ Why do you think using this architecture in this practical assignment can help to improve the results? Hint: See MobileNet publication

Based on (Howard et al., 2017) for the following reasons MobileNet can improve results in Pneumonia Classification :

Transfer Learning: MobileNet is often used as a pre-trained model on large datasets like ImageNet. By leveraging transfer learning, the pre-trained weights can be fine-tuned on the pneumonia dataset. This will allow the model to benefit from the features learned on a large, diverse dataset while adapting to the specific task of pneumonia detection.The ability to use pre-trained weights from ImageNet will allow the model to start with a strong baseline, requiring fewer training examples to achieve good performance.

Reduced Complexity: MobileNet utilizes depthwise separable convolutions, which factorize a standard convolution into a depthwise convolution and a pointwise convolution. The depthwise separable convolutions significantly reduce the number of parameters and computational complexity. This makes it easier to train the model on smaller datasets without the risk of overfitting.Hence a better result is expected by using this architecture since limited dataset is common problem in healthcare.

Efficiency and Accuracy: MobileNet is optimized for speed and efficiency, which means it can process chest X-ray images quickly even on less powerful hardware.Reduced computational requirements make it feasible to train and fine-tune the model even with limited hardware resources. Despite its efficiency, MobileNet maintains high accuracy, making it suitable for medical image classification tasks.

9__ How many parameters does this network have? How does this compare to better performing networks available in torch?

The pre-trained MobleNet model tuned for binary classification has 2,524,930 parameters

In the following you can see an example of how we can load pre-trained models in our codes.

```
# Load the pre-trained MobileNet model

## doesn't work, looking at the source (https://rdrr.io/github/mlverse/torchvision/api/)
mobilenet <- torchvision::model_mobilenet_v2(pretrained = TRUE)

#source("lab3_chunk_24.R")

#run all code in source to load functions
mobilenet<-model_mobilenet_v2(pretrained = TRUE)

# Modify the last fully connected layer to match your classification task

#in_features <- mobilenet$classifier$in_features
mobilenet$classifier <- nn_linear(224*224*3, 2)   # Adjust the output size based on your classification
mobilenet
```

```
## An ‘nn_module‘ containing 2,524,930 parameters.
##
## -- Modules ---------------------------------------------------------------
## * features: <nn_sequential> #2,223,872 parameters
## * classifier: <nn_linear> #301,058 parameters
```

Other networks available in torch are:

ResNet:
ResNet-50: Approximately 25.6 million parameters.
ResNet-101: Approximately 44.5 million parameters.
ResNet-152: Approximately 60.2 million parameters.
VGG:
VGG-16: Approximately 138 million parameters.
VGG-19: Approximately 144 million parameters.
EfficientNet:
EfficientNet-B0: Approximately 5.3 million parameters.
EfficientNet-B7: Approximately 66 million parameters.

MobileNets comparison and rationale for using in pneumonia classification is discussed below:

1. Other Networks like ResNet and VGG have significantly more parameters, offering better performance but at the cost of increased computational requirements.

2. MobileNet is significantly more lightweight compared to other architectures, making it ideal for deployment on resource-constrained devices and for scenarios where computational resources are limited.

3. Due to the reduced number of parameters and efficient design, MobileNet allows for faster inference times, which is crucial in real-time applications such as medical diagnosis.

4. While it has fewer parameters, MobileNet still performs adequately for many tasks, especially when combined with transfer learning techniques. For example, it can be fine-tuned on our data set to achieve high accuracy.

10_ Using the provided materials in this practical, train a different network architecture.Does this perform better?

References

1. Annarumma, M., Withey, S. J., Bakewell, R. J., Pesce, E., Goh, V., & Montana, G. (2019). Automated Triaging of Adult Chest Radiographs with Deep Artificial Neural Networks. *Radiology*, 291(1), 196-202.

2. Esteva, A., Robicquet, A., Ramsundar, B., Kuleshov, V., DePristo, M., Chou, K., ... & Dean, J. (2017). A Guide to Deep Learning in Healthcare. *Nature Medicine*, 25(1), 24-29.

3. Farrajota, M.

(2017) ILSVRC2012 - Imagenet Large Scale Visual Recognition Challenge 2012, ILSVRC2012 - Imagenet Large Scale Visual Recognition Challenge 2012 - dbcollection 0.2.6 documentation. Available at: https://dbcollection.readthedocs.io/en/latest/datasets/imagenet.html (Accessed: 16 June 2024).

4. Hallaj, P. (2023). Data Augmentation: Benefits and Disadvantages. *Medium*. Available at: https://medium.com/@pouyahallaj/data-augmentation-benefits-and-disadvantages-38d8201aead (Accessed: 16 June 2024).

5. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., ... & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*. Retrieved from https://arxiv.org/abs/1704.04861.

6. Lakhani, P., & Sundaram, B. (2017). Deep Learning at Chest Radiography: Automated Classification of Pulmonary Tuberculosis by Using Convolutional Neural Networks. *Radiology*, 284(2), 574-582.

7. IMAGENET (2024) About ImageNET, ImageNet. Available at: https://www.image-net.org/about.php (Accessed: 16 June 2024).

8. Mayo Clinic (2020). Pneumonia. *Mayo Clinic*. Available at: https://www.mayoclinic.org/diseases-conditions/pneumonia/symptoms-causes/syc-20354204 (Accessed: 13 June 2024).

9. Qin, C., Yao, D., Shi, Y., & Song, Z. (2018). Computer-Aided Detection in Chest Radiography Based on Artificial Intelligence: A Survey. *BioMedical Engineering Online*, 17(1), 113.

10. Rajpurkar, P., Irvin, J., Ball, R. L., Zhu, K., Yang, B., Mehta, H., ... & Ng, A. Y. (2017). Deep Learning for Chest Radiograph Diagnosis: A Retrospective Comparison of the CheXNeXt Algorithm to Practicing Radiologists. *PLoS Medicine*, 15(11), e1002686.

11. Razavian, A. S., Azizpour, H., Sullivan, J., & Carlsson, S. (2014). CNN Features Off-the-Shelf: An Astounding Baseline for Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 806-813.

12. WHO (2024). Pneumonia. *World Health Organization*. Available at: https://www.who.int/health-topics/pneumonia#tab=tab_1 (Accessed: 13 June 2024).

13. Yosinski, J., Clune, J., Bengio, Y., & Lipson, H. (2014). How Transferable Are Features in Deep Neural Networks? In *Advances in Neural Information Processing Systems (NIPS)*, 3320-3328.

Useful links https://medium.com/@kemalgunay/getting-started-with-image-preprocessing-in-r-52c7d153b381 https://cran.r-project.org/web/packages/magick/vignettes/intro.html https://www.datanovia.com/en/blog/easy-image-processing-in-r-using-the-magick-package/ https://dahtah.github.io/imager/imager.html https://rdrr.io/github/mlverse/torchvision/api/ https://github.com/brandonyph/Torch-for-R-CNN-Example