

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
library(ggplot2)
library(gridExtra)
library(imager)
```

```

## Loading required package: magrittr
##
## Attaching package: 'imager'
## The following object is masked from 'package:magrittr':
##     add
## The following objects are masked from 'package:stats':
##     convolve, spectrum
## The following object is masked from 'package:graphics':
##     frame
## The following object is masked from 'package:base':
##     save.image
library(jpeg)
library(magick)

## Linking to ImageMagick 6.9.12.93
## Enabled features: cairo, fontconfig, freetype, heic, lcms, pango, raw, rsvg, webp
## Disabled features: fftw, ghostscript, x11
library(EBImage)

##
## Attaching package: 'EBImage'
## The following objects are masked from 'package:imager':
##     channel, dilate, display, erode, resize, watershed
library(grid)

##
## Attaching package: 'grid'
## The following object is masked from 'package:imager':
##     depth
library(dplyr)

##
## Attaching package: 'dplyr'
## The following object is masked from 'package:EBImage':
##     combine
## The following object is masked from 'package:imager':
##     where
## The following object is masked from 'package:gridExtra':
##     combine

```

```

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

library(abind)

##
## Attaching package: 'abind'

## The following object is masked from 'package:EBImage':
##
##     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:

[“\[https://drive.google.com/file/d/14H_FiIWf12ONOJ_G4vvzNDDGvY7Ccqtm/view?usp=sharing\]\(https://drive.google.com/file/d/14H_FiIWf12ONOJ_G4vvzNDDGvY7Ccqtm/view?usp=sharing\)”](https://drive.google.com/file/d/14H_FiIWf12ONOJ_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 <- "data/lab3_chest_xray.zip"

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

# 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 causes the air sacs in the lungs to become inflamed and may fill with fluid or pus. The benefits of automatically identifying pneumonia from X-rays are that it can be detected and treated earlier, and it also improves diagnostic accuracy and consistency. It can also reduce the workload of radiologists and optimize resource utilization, especially in areas with limited medical resources, helping to quickly provide accurate diagnosis and treatment.

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.

```
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/IM-0091-0001.jpeg"
## [2] "lab3_chest_xray/test/NORMAL/IM-0102-0001.jpeg"
## [3] "lab3_chest_xray/test/NORMAL/NORMAL2-IM-0066-0001.jpeg"
## [4] "lab3_chest_xray/test/PNEUMONIA/person139_bacteria_666.jpeg"
## [5] "lab3_chest_xray/test/PNEUMONIA/person1616_virus_2802.jpeg"
## [6] "lab3_chest_xray/train/NORMAL/IM-0145-0001.jpeg"
## [7] "lab3_chest_xray/train/NORMAL/IM-0199-0001.jpeg"
## [8] "lab3_chest_xray/train/NORMAL/IM-0301-0001.jpeg"
## [9] "lab3_chest_xray/train/NORMAL/IM-0340-0001.jpeg"
## [10] "lab3_chest_xray/train/NORMAL/IM-0354-0001.jpeg"
## [11] "lab3_chest_xray/train/NORMAL/IM-0519-0001-0002.jpeg"
## [12] "lab3_chest_xray/train/NORMAL/IM-0527-0001.jpeg"
## [13] "lab3_chest_xray/train/NORMAL/IM-0552-0001.jpeg"
## [14] "lab3_chest_xray/train/NORMAL/IM-0599-0001.jpeg"
## [15] "lab3_chest_xray/train/NORMAL/IM-0649-0001.jpeg"
## [16] "lab3_chest_xray/train/PNEUMONIA/person1089_virus_1808.jpeg"
## [17] "lab3_chest_xray/train/PNEUMONIA/person1096_bacteria_3037.jpeg"
## [18] "lab3_chest_xray/train/PNEUMONIA/person1130_bacteria_3072.jpeg"
## [19] "lab3_chest_xray/train/PNEUMONIA/person1171_bacteria_3118.jpeg"
## [20] "lab3_chest_xray/validate/NORMAL2-IM-1437-0001.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?

From these file names, we can see some details. These file names include two important information: 1. Is the patient sick: “NORMAL” in the path means that the patient is not sick, and “PNEUMONIA” means that the patient has pneumonia 2. Pathogen type: The file name contains “bacteria” or “virus”, indicating the type of pathogen that causes pneumonia. 3. Image source: The file name contains “person” and a unique number, indicating that each image is from a different individual.

The difficulty lies in: not only to identify whether the patient has pneumonia, but also to identify what pathogen caused the pneumonia, such as bacteria and viruses. This means that different types of pneumonia may appear differently on X-rays.

The dataset folder is assumed to contain three subfolders: train, test, and validate. Each of these subfolders further contains two folders representing the two classes or labels in our classification task: “normal” and “pneumonia”. The code provides a way to explore the dataset directory structure and obtain information about the number of images and their distribution across different classes. This information can be helpful in understanding the dataset and planning the subsequent steps of the image classification task.

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

```

train_pneumonia_dir <- file.path(base_dir, "train", "PNEUMONIA")
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
cat("Total Pneumonia Images:", length(c(train_pn, test_pn, val_pn)), "\n")

## Total Pneumonia Images: 608
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.

```

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-0381-0001.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) {

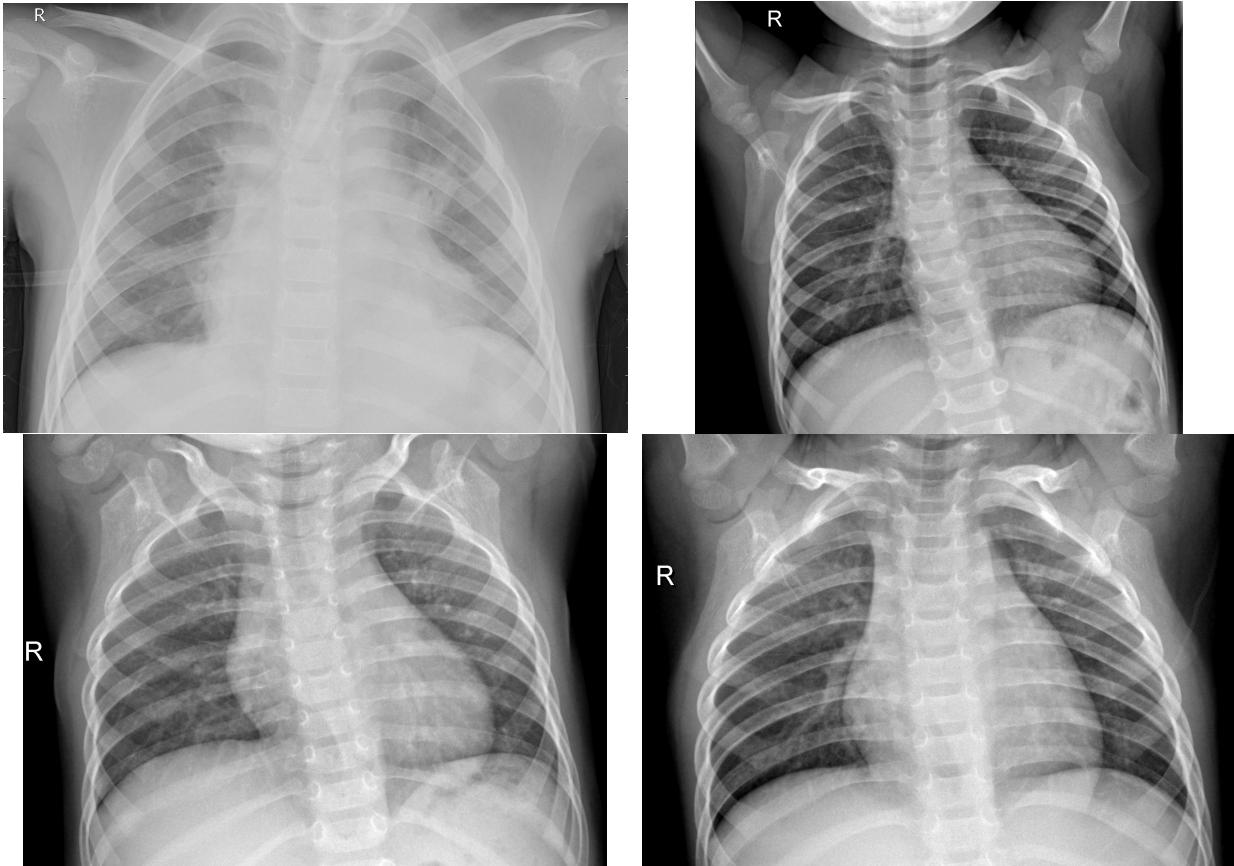
  image <- readImage(shuffled_train_dataset[i])

  # 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?

I think we may need to normalize several properties:

Brightness and Contrast:

X-rays may vary in brightness and contrast.

Image Size and Resolution:

X-rays may have different sizes and resolutions. .

Orientation and Alignment:

Make sure all X-ray images have consistent orientation and alignment.

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)
```

Pneumonia



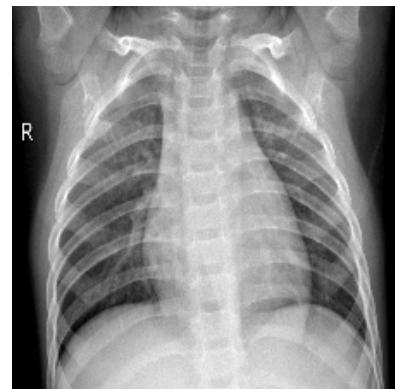
Pneumonia



Pneumonia



Pneumonia

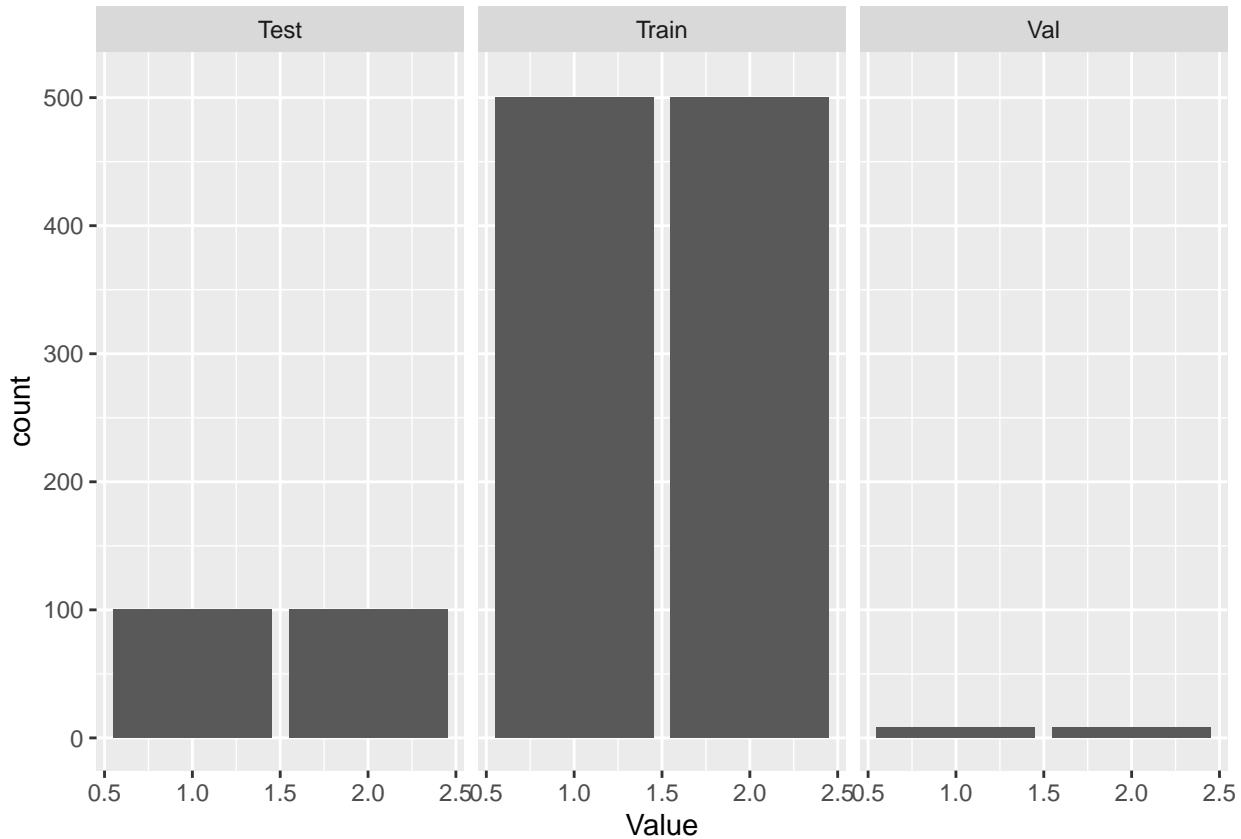


Let's count and display the number of images in each dataset to examine the distribution of labels in the data. This will help us understand the ratio of "normal" and "pneumonia" labels in the dataset.

```
# 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)
)

# 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?

If the dataset is unbalanced, as far as I know, there are two ways to do it:

1. **Undersampling:** Reduce the number of majority class samples to make it close to the number of minority class samples. Although this may lose some information, it can balance the dataset.
2. **Oversampling:** Increase the number of minority class samples to make it close to the number of majority class samples. Common methods include duplicating minority class samples or using SMOTE to generate synthetic minority class 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
```

```

print(dim(test_X))

## [1] 200 224 224    1
print(length(test_y))

## [1] 200
print(dim(val_X))

## [1] 16 224 224    1
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.

```

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.

```

# 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.

```
install_torch()
# 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
```

create 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 task
  },

  forward = function(x) {
    x %>%
      self$conv1() %>% # N * 1 * 224 * 224
      nnf_relu() %>% # N * 32 * 222 * 222
      self$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 classifier)
    }
)

```

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$records$metrics$train[[1]]$acc)
  cat("Valid metrics: Loss: ", fitted$records$metrics$valid[[1]]$loss, " - Acc: ", fitted$records$metrics$valid[[1]]$acc)
  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: 1.030052 - Acc: 0.502
## Valid metrics: Loss: 0.6931065 - Acc: 0.5
##
## Epoch 2 / 3
## Train metrics: Loss: 1.58929 - Acc: 0.527
## Valid metrics: Loss: 0.6938493 - Acc: 0.5
##
## Epoch 3 / 3
## Train metrics: Loss: 1.179784 - Acc: 0.511
## Valid metrics: Loss: 0.6941429 - Acc: 0.5

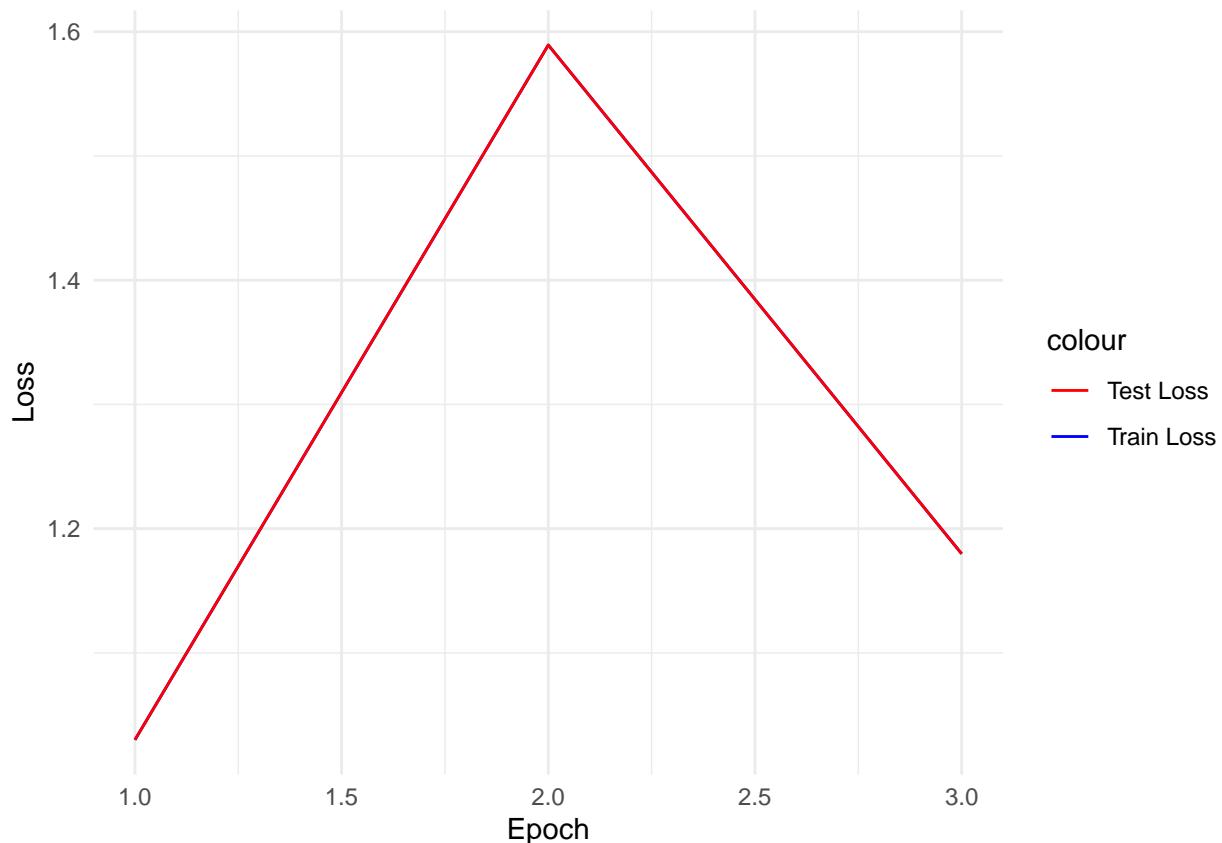
```

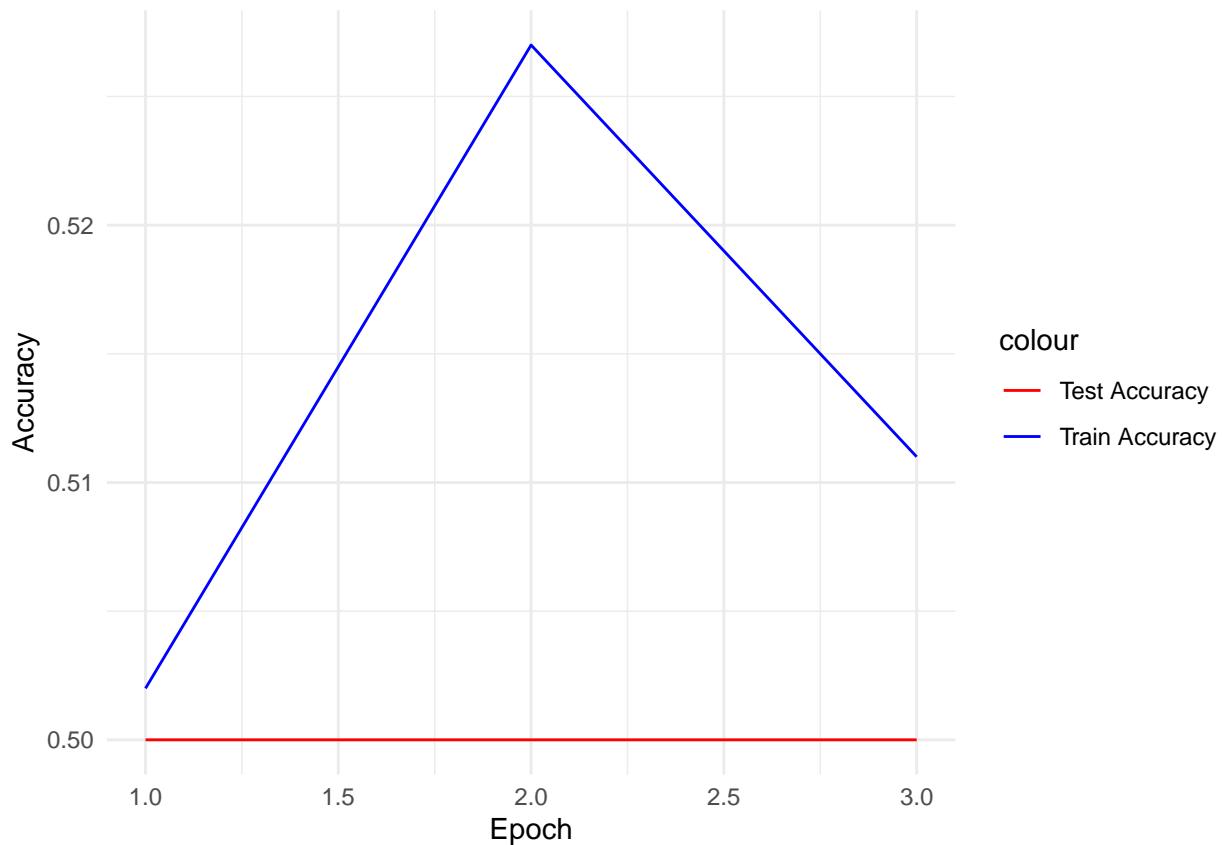
Plot learning curves

```
# 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)
```





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?

From the provided training and test accuracy, it seems that the model is not successfully classifying the X-rays into pneumonia and normal:

1. **Training and validation accuracy:** Both the training accuracy and validation accuracy are around 50%.
2. **Loss:** Although the training loss value is decreasing, the validation loss value remains the same, indicating that the performance of the model on the validation set has not improved. .

Possible reasons:

1. **Dataset is too small:**
2. **Insufficient model complexity:**
3. **Insufficient features**
4. **Insufficient training time**

It seems that the model is not training effectively the model is struggling to learn from the data and is not generalizing well to unseen examples. It suggests that the model may be too complex or the training process needs adjustments.

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

I think the following points can be used for improvement:

1. Improve the model architecture

- **Use pre-trained CNN models:** You can try to use pre-trained CNN models such as VGG, ResNet, etc. for transfer learning.
- **Increase model complexity:** Add more layers or nodes to the current model to capture more complex features.

2. Increase training data

- **Data enhancement:** Perform random transformations on images during training, such as rotation, cropping, scaling, etc., to increase data diversity and prevent overfitting.

3. Adjust hyperparameters

- **Learning rate, batch size:** Adjust the model's hyperparameters, such as learning rate, batch size, etc., through grid search or random search to find the optimal configuration.
- **Number of training rounds:** Increase the number of training rounds appropriately to ensure that the model is fully trained, but prevent overfitting.

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.

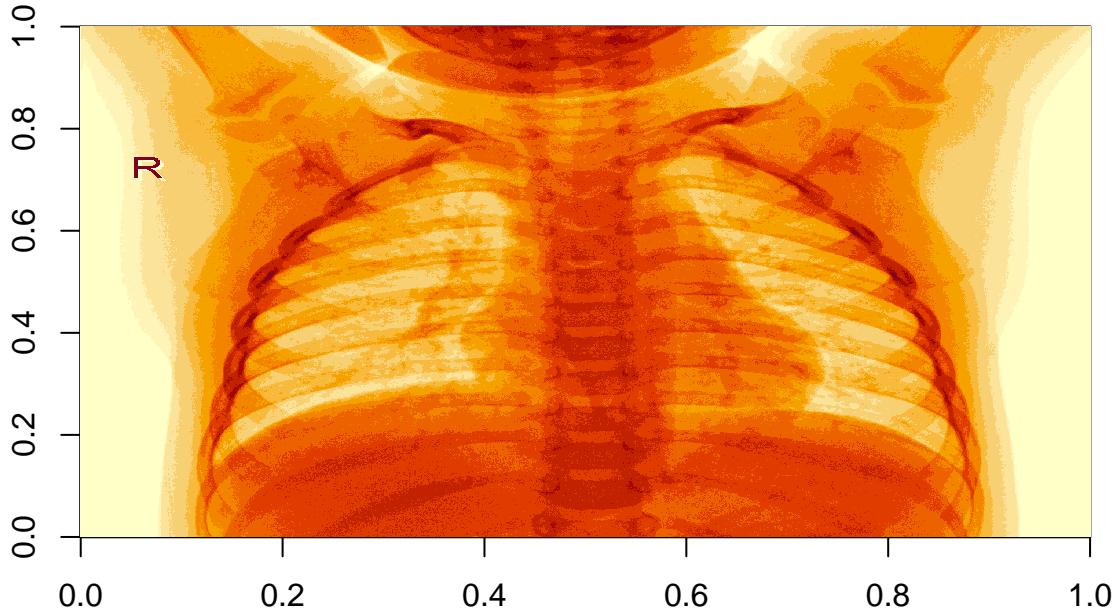
By applying data augmentation, we can create new training samples with slight modifications, such as random rotations, translations, flips, zooms, or changes in brightness and contrast. These modifications can mimic real-world variations and increase the model's ability to handle different scenarios.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

Some potential disadvantages and limitations of data augmentation that I think of:

1. **Increased computational cost:** Data augmentation often generates transformed images dynamically during training, which increases the computational and storage burden.
2. **Over-enhancement:** The generated images may deviate from the distribution of the original images, causing the model to learn inaccurate features.
3. **Degradation of data quality:** Excessive or unreasonable image transformations may degrade the quality of the image, making it more difficult for the model to learn useful features.
4. **False positives and false negatives:** In some cases, data augmentation may introduce spurious features that make the model perform poorly on real-world test data. For example, random flipping or rotation may produce features that do not match actual pneumonia images, leading to misclassification.

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.

Torchvision 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 Large Scale Visual Recognition Challenge 2012 (ILSVRC 2012), also known as ImageNet for short, is a large-scale visual recognition challenge:

1. Dataset size:

The ImageNet dataset contains more than 15 million annotated high-resolution images covering 22,000 categories. In ILSVRC 2012, a portion of the data was used, with a total of 1,000 categories, each containing hundreds to thousands of images.

2. ImageNet helps us:

3. **Pre-trained model:** Use the model pre-trained on ImageNet for transfer learning.

- 4. Feature extraction:** Models trained on large-scale datasets can extract more useful features, which can be reused in different image classification tasks to improve the accuracy and robustness of the model.

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

According to my research, MobileNet is a lightweight convolutional neural network architecture that is particularly suitable for running in resource-constrained environments, such as mobile devices and embedded systems. The main advantages are:

Efficient calculation: MobileNet uses depthwise separable convolutions, which significantly reduces model parameters and calculation amount. This means models run faster both during training and inference, while requiring less computing resources.

Few parameters: Compared with other high-performance networks, the number of parameters of MobileNet is greatly reduced. This allows it to maintain good performance while having lower memory and storage requirements.

Flexibility: MobileNet introduces two key hyperparameters - width multiplier and resolution multiplier. These parameters can be adjusted to trade off the accuracy and efficiency of the model.

Advantages of transfer learning: MobileNet pre-trained on large-scale data sets such as ImageNet can be well applied to transfer learning and quickly adapt to specific tasks, such as the X-ray classification we did this time.

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

Standard MobileNetV1 has approximately 4.2 million parameters under default settings, and by adjusting the width multiplier and resolution multiplier, the number of parameters can be further reduced to 1.3 million or even lower. In comparison, ResNet-50 has approximately 23.5 million parameters and InceptionV3 has 23 million parameters. This huge difference in the number of parameters allows MobileNet to significantly reduce computing resources and memory requirements while maintaining reasonable performance, making it ideal for lightweight applications.

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

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

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

```
to_three_channels <- function(img) {
  img <- torch_unsqueeze(img, dim = 1)
  img <- torch_cat(list(img, img, img), dim = 1)
  return(img)
}

ImageDataset_augment <- dataset(
  name = "ImageDataset",
  initialize = function(X, y, transform = NULL) {
    self$transform <- transform
    self$data <- X
    self$labels <- y
  },
  .getitem = function(i) {
    x <- self$data[i,,,]
    x <- self$transform(x)
    y <- self$labels[i]

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

train_transforms <- function(img) {
  img <- torch_squeeze(torch_tensor(img))
  img <- to_three_channels(img)
  img <- img %>%
    transform_color_jitter() %>%
    transform_random_horizontal_flip() %>%
    transform_random_vertical_flip(p = 0.5)

  return(img)
}

test_transforms <- function(img) {
  img <- torch_squeeze(torch_tensor(img))
  img <- to_three_channels(img)

  return(img)
}

train_dataset <- ImageDataset_augment(train_X, train_y, transform = train_transforms)
test_dataset <- ImageDataset_augment(test_X, test_y, transform = test_transforms)
val_dataset <- ImageDataset_augment(val_X, val_y, transform = test_transforms)

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

```

pretrained_mobilenet <- model_mobilenet_v2(pretrained = TRUE)

net <- nn_module(
  "Net",

  initialize = function() {
    self$features <- pretrained_mobilenet$features
    in_features <- pretrained_mobilenet$classifier[[2]]$in_features

    self$dropout1 <- nn_dropout(0.25)
    self$dropout2 <- nn_dropout(0.5)
    self$fc1 <- nn_linear(in_features, 128)
    self$fc2 <- nn_linear(128, 2)
  },

  forward = function(x) {
    x %>%
      self$features() %>%
      nnf_adaptive_avg_pool2d(c(1, 1)) %>%
      torch_flatten(start_dim = 2) %>%
      self$dropout1() %>%
      self$fc1() %>%
      nnf_relu() %>%
      self$dropout2() %>%
      self$fc2()
  }
)

print(net)

## <Net> object generator
##   Inherits from: <inherit>
##   Public:
##     .classes: Net nn_module
##     initialize: function ()
##     forward: function (x)
##     clone: function (deep = FALSE, ..., replace_values = TRUE)
##   Private:
##     .__clone_r6__: function (deep = FALSE)
##   Parent env: <environment: 0x7f97395d3448>
##   Locked objects: FALSE
##   Locked class: FALSE
##   Portable: TRUE
num_epochs <- 5

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

for (epoch in 1:num_epochs) {
  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)

  cat("Epoch ", epoch, "/", num_epochs, "\n")
  cat("Train metrics: Loss: ", fitted$records$metrics$train[[1]]$loss, " - Acc: ", fitted$records$metrics$train[[1]]$acc)
  cat("Valid metrics: Loss: ", fitted$records$metrics$valid[[1]]$loss, " - Acc: ", fitted$records$metrics$valid[[1]]$acc)
  cat("\n")

  train_loss[epoch] <- fitted$records$metrics$train[[1]]$loss
  train_acc[epoch] <- fitted$records$metrics$train[[1]]$acc
  test_loss[epoch] <- fitted$records$metrics$valid[[1]]$loss
  test_acc[epoch] <- fitted$records$metrics$valid[[1]]$acc
}

## Epoch 1 / 5
## Train metrics: Loss: 0.7253737 - Acc: 0.484
## Valid metrics: Loss: 0.6929915 - Acc: 0.5
##
## Epoch 2 / 5
## Train metrics: Loss: 0.7216554 - Acc: 0.479
## Valid metrics: Loss: 0.6935251 - Acc: 0.5
##
## Epoch 3 / 5
## Train metrics: Loss: 0.7056873 - Acc: 0.479
## Valid metrics: Loss: 0.6931075 - Acc: 0.5
##
## Epoch 4 / 5
## Train metrics: Loss: 0.6985701 - Acc: 0.505
## Valid metrics: Loss: 0.6908732 - Acc: 0.51
##
## Epoch 5 / 5
## Train metrics: Loss: 0.7057576 - Acc: 0.483
## Valid metrics: Loss: 0.6914119 - Acc: 0.555

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()

acc_df <- data.frame(
  Epoch = 1:num_epochs,
  Train_Accuracy = train_acc,

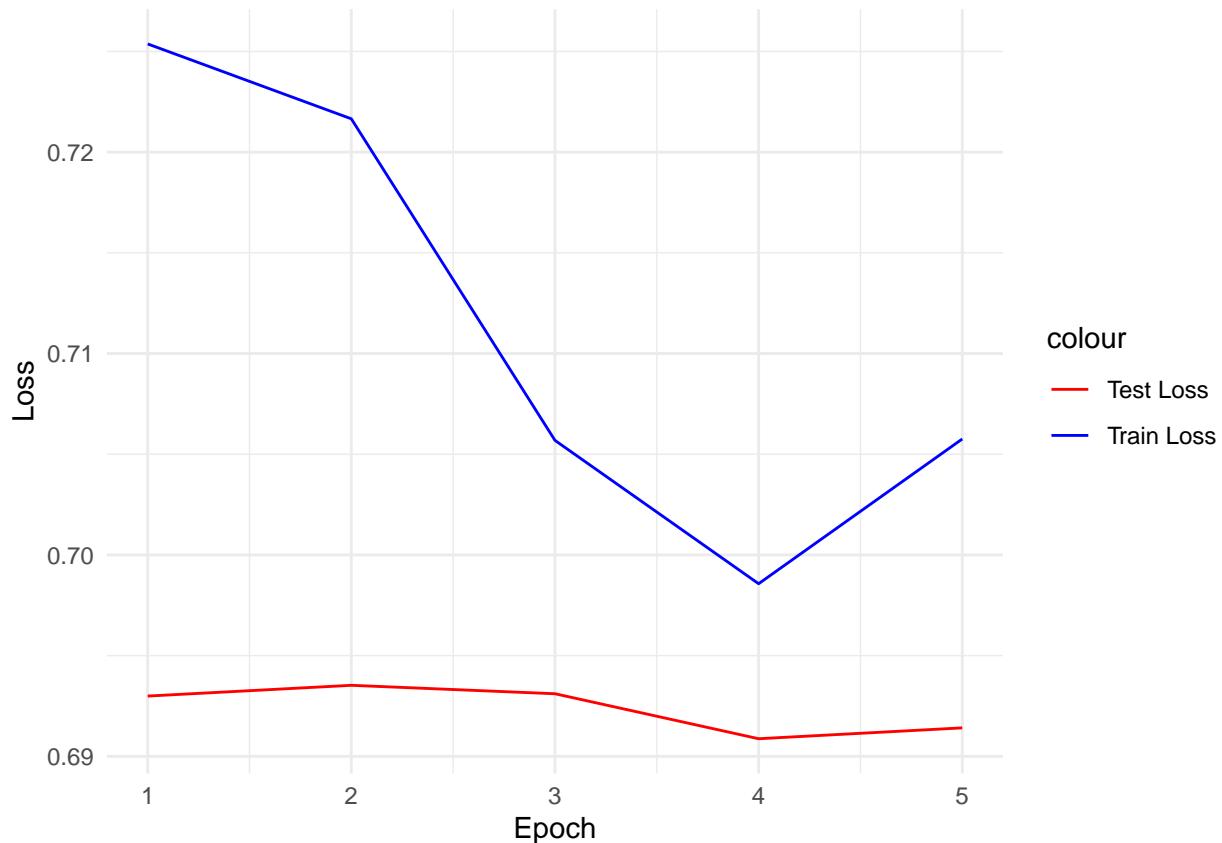
```

```

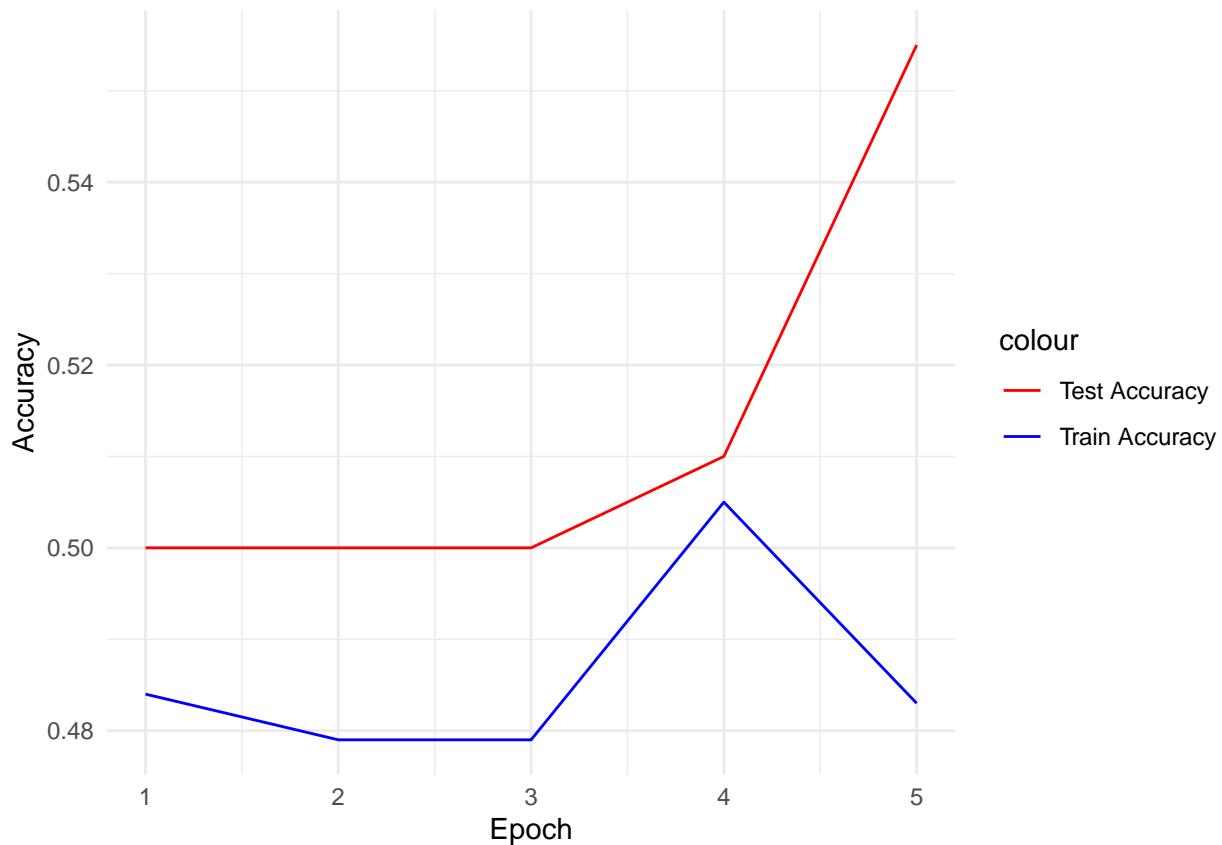
    Test_Accuracy = test_acc
)
acc_plot <- ggplot(data = acc_df) +
  geom_line(aes(x = Epoch, y = Train_Accuracy, color = "Train Accuracy")) +
  geom_line(aes(x = Epoch, y = Test_Accuracy, color = "Test Accuracy")) +
  labs(x = "Epoch", y = "Accuracy") +
  scale_color_manual(values = c("Train Accuracy" = "blue", "Test Accuracy" = "red")) +
  theme_minimal()

print(loss_plot)

```



```
print(acc_plot)
```



```
# 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
```