Practical 3 - Medical Images

Alexandra Del Favero-Campbell

2024-06-03

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 6.9.12.98
## Enabled features: cairo, freetype, fftw, ghostscript, heic, lcms, pango, raw, rsvg, webp
## Disabled features: fontconfig, 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 (https://www.xquartz.org/) - 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 (https://www.cell.com/cell/fulltext/S0092-8674(18)30154-5)) 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 (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 <- "C:/Users/ajdel/Desktop/Alex/PhD Stuff/CSCI6410/Practicals/Practical 3/lab3_chest_xr
ay.zip"

# Specify the destination directory to extract the files
destination_dir <- "C:/Users/ajdel/Desktop/Alex/PhD Stuff/CSCI6410/Practicals/Practical 3/lab3_c
hest_xray"

# Create the destination directory if it doesn't exist
dir.create(destination_dir, showWarnings = FALSE, recursive = TRUE)

# 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?

To put simply, pneumonia is an infection of the lungs that cause the air sacs in one or both of a patient's lungs to become inflamed. It can be caused by a variety of organisms, including bacteria, viruses, and fungi. This infection can cause the air sacs to fill with fluid or a sort of pus-like material, which can cause the patient to cough with phlegm or pus, as well as display symptoms such as fever, chills, and difficulty breathing (see Figures 1 and 2 for how this looks on a chest X-ray). The application of being able to identify pneumonia from X-rays automatically is particularly useful, especially in pediatric medicine, as pediatric pneumonia is consistently estimated as one of the single leading causes of childhood mortality (Rudan et al., 2008), killing about 2 million children under the age of 5 globally and killing more children than malaria, measles, and HIV/AIDS combined (Adegbola, 2012). Furthermore, certain types of pneumonia (e.g., bacterial pneumonia) require an urgent referral for immediate antibiotic treatment, which means that it is crucial to have an accurate and timely diagnosis. Using radiological chest imaging is part of the routine procedure to help differentiate between the types of pneumonia, which requires radiologic interpretation by a doctor. However, rapid radiologic interpretation of these images is not always readily available, especially in regions with low resource availability where pediatric pneumonia tends to have the highest incidence and highest mortality rates.

Therefore, there are many benefits of potentially being able to identify pneumonia from X-rays automatically, with one of the key points being the great potential for early detection and timely treatment. As this is very important for improving patient outcomes and avoiding mortality rates, especially in severe cases, this would be one of the biggest benefits of this potential opportunity. Other benefits include better resource optimization and increases in consistency and efficiency. With such a technological opportunity in place, medical centers and hospitals could use this to better optimize what resources they have, as well as help prioritize cases that need urgent. Furthermore, such a system could potentially process more chest X-rays more quickly and consistently than a human could, reducing the workload of radiologists in areas where they are a shortage of radiologists and/or other medical professionals and helping support the provision of critical care and diagnosis. Thus, there are great benefits to an automated system like the one described in the Kermany et al. (2018) paper.

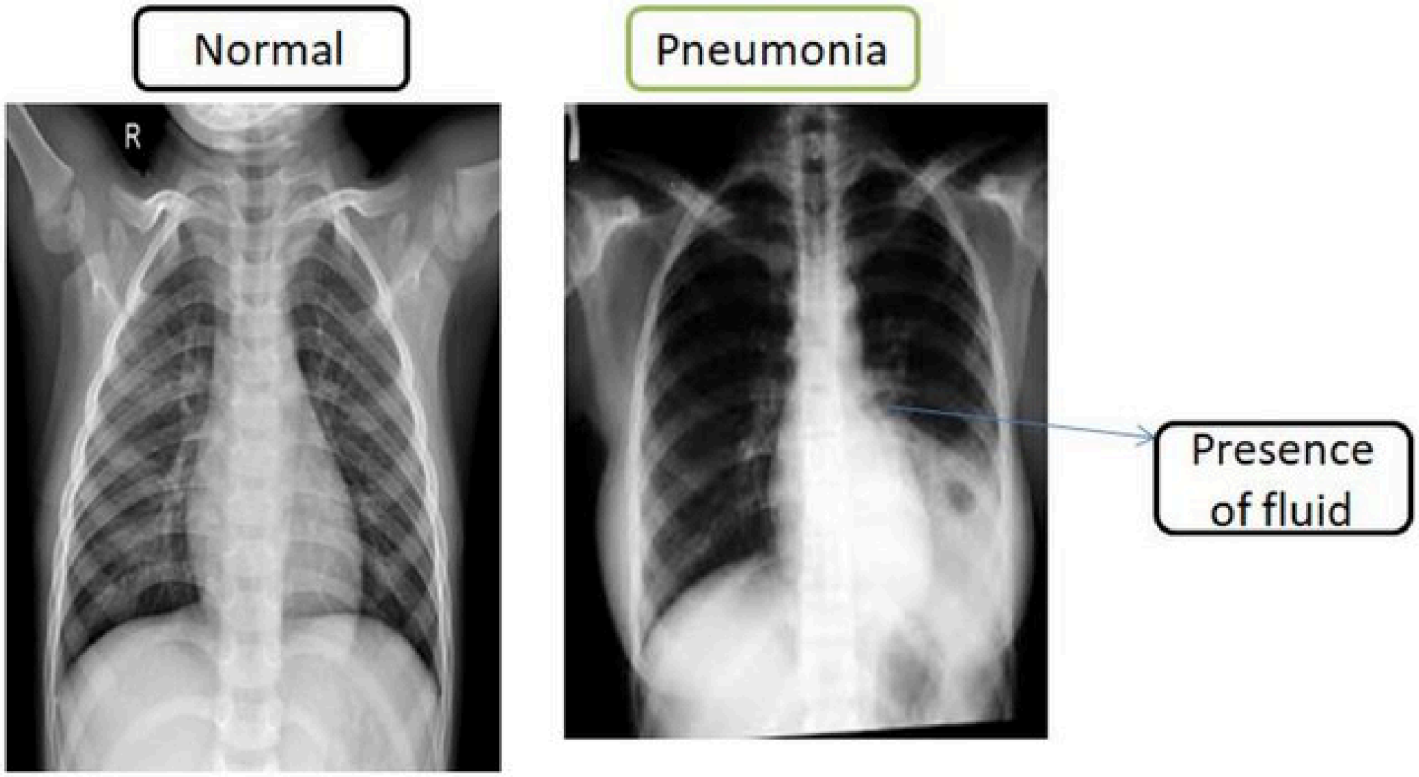


Figure 1. Images showing difference between a normal chest X-ray (left) compared to a chest X-ray of a patient with pneumonia (right) (Singh and Tripathi, 2022). As we can see in the X-ray on the right-hand side, there is some presence of fluid in the lungs that is characteristic of pneumonia.

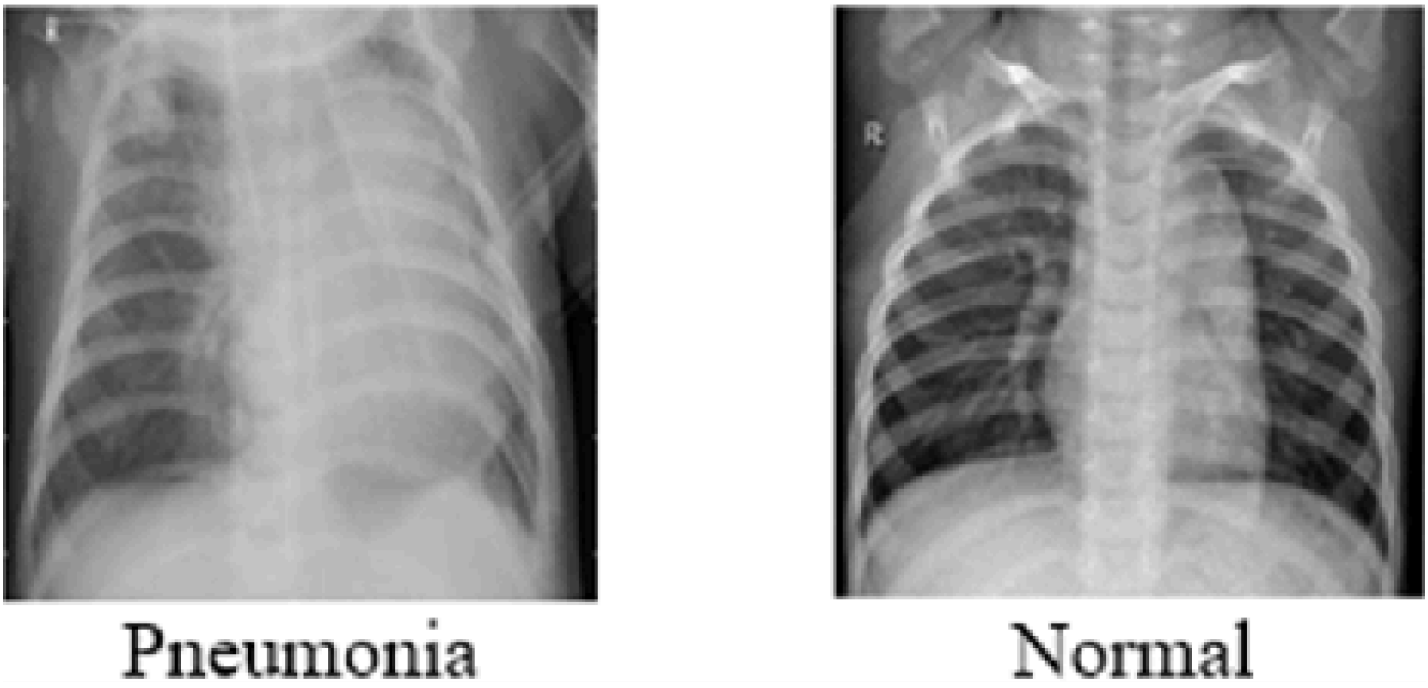


Figure 2. Images showing difference in normal pediatric X-ray of lungs (right) compared to pediatric X-ray of lungs that is positive for pneumonia (left) (Labhane et al., 2020).

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/lab3_chest_xray/test/NORMAL/IM-0103-0001.jpeg"
##  [2] "lab3_chest_xray/lab3_chest_xray/test/PNEUMONIA/person1629_virus_2823.jpeg"
##  [3] "lab3_chest_xray/lab3_chest_xray/test/PNEUMONIA/person1631_virus_2826.jpeg"
##  [4] "lab3_chest_xray/lab3_chest_xray/test/PNEUMONIA/person81_bacteria_397.jpeg"
##  [5] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0172-0001.jpeg"
##  [6] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0297-0001.jpeg"
##  [7] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0354-0001.jpeg"
##  [8] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0451-0001.jpeg"
##  [9] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0501-0001-0002.jpeg"
## [10] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0527-0001.jpeg"
## [11] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0582-0001.jpeg"
## [12] "lab3_chest_xray/lab3_chest_xray/train/NORMAL/IM-0614-0001.jpeg"
## [13] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1035_virus_1729.jpeg"
## [14] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1040_virus_1735.jpeg"
## [15] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1049_bacteria_2983.jpeg"
## [16] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1080_bacteria_3020.jpeg"
## [17] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1105_bacteria_3046.jpeg"
## [18] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1123_virus_1848.jpeg"
## [19] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1158_virus_1940.jpeg"
## [20] "lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1235_virus_2095.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?

Yes, these filenames tell us exactly what type of pneumonia the patient had and a unique identifier (e.g., a patient ID number and/or image number). For example, “person1216_virus_2062.jpeg” potentially indicates patient number 1216 and that this patient had viral pneumonia. This differs from the filenames from the “NORMAL” folder, which give us no information about the patient ID number or anything other identifiable information. In addition, it looks like some of the “PNEUMONIA” folder images may come from the same patient ID number and are noted as two different types of pneumonia. This is the case it seems for “person1057”, for example (e.g, “person1057_bacteria_2991.jpeg” and “person1057_virus_1756.jpeg”). What this also tells us is that there is variability in the characteristics of what pneumonia looks like in chest X-rays. It seems, from the way the files are documented that viral and bacterial infections present differently on X-ray imaging, which potentially has an effect on the patterns that could be recognized by a model that is meant to train to identify pneumonia. This leads us to believe that the model will need to handle heterogeneity in X-ray images, which will certainly increase the complexity of the prediction model that we want to create.

From what we know about how the files are named, we can tell a few things that may make predicting pneumonia a bit more challenging than we originally thought. For example, the fact that there are multiple causes for pneumonia that require differing treatment plans indicates that it is critical to annotate our images with necessary detail. In this case, we need to annotate whether it is bacterial pneumonia or viral pneumonia that a patient has. The heterogeneity of what pneumonia looks like on a chest X-ray based on what is causing it will potentially make it more challenging for the model to learn a consistent pattern that consistently indicates a correct diagnosis of

pneumonia, requiring the learning of multiple different patterns and features. This is very important for training a robust model; without such details, the model may struggle to effectively and accurately differentiate between different types/causes of pneumonia. Another thing that is important for us to keep in mind as we have different types of pneumonia annotated is we need to make sure that there are no imbalances in the amounts of cases, as this could lead to model overfitting and affect the model's overall generalizability.

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/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_norma
l)), "\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 train set
cat("file name: ", shuffled_train_dataset[5], "\nlabel: ", shuffled_train_labels[5])
```

```
## file name:  lab3_chest_xray/lab3_chest_xray/train/PNEUMONIA/person1000_bacteria_2931.jpeg
## label:  pneumonia
```

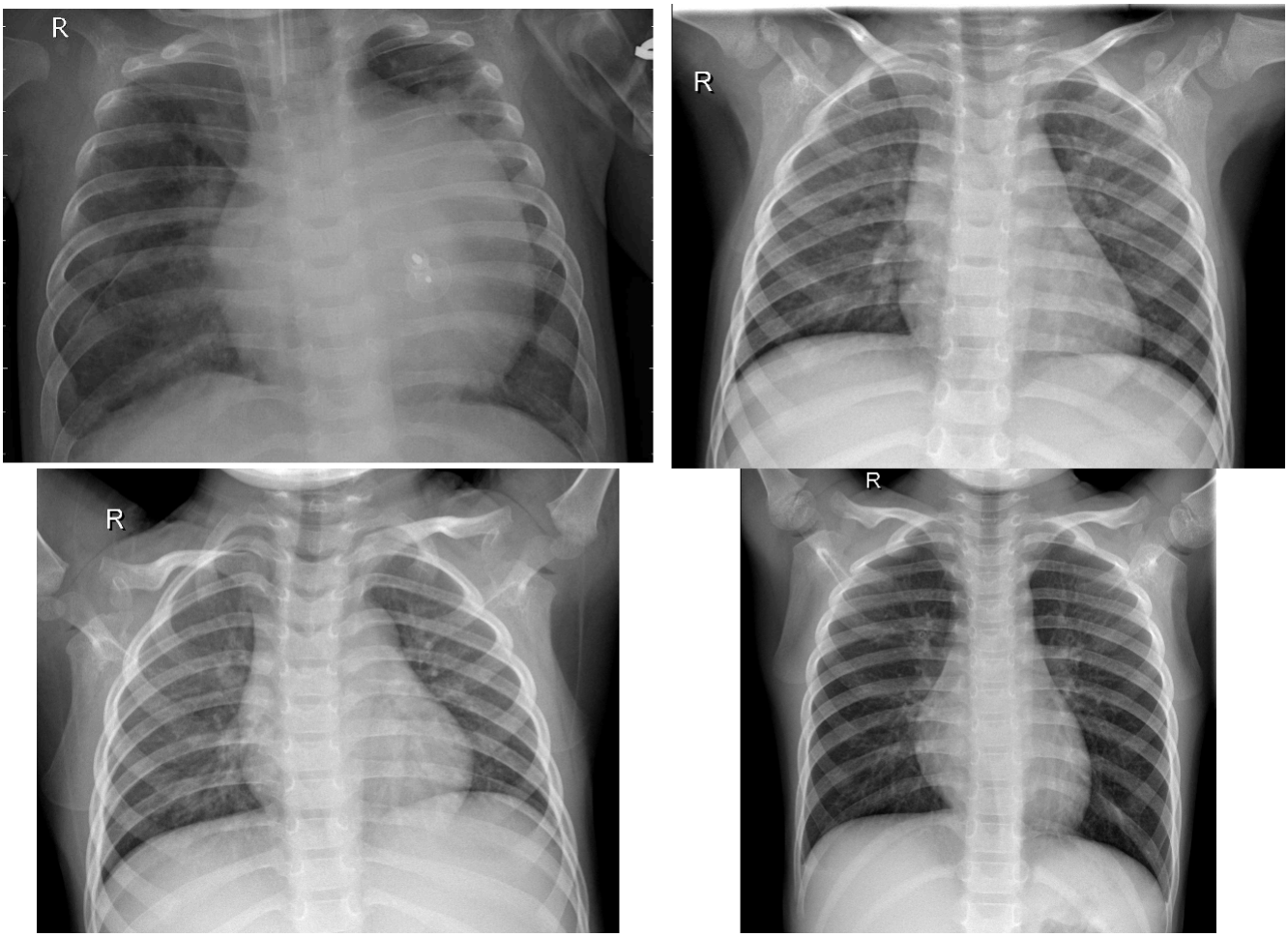
```
#showing a file name from test set
cat("file name: ", shuffled_test_dataset[5], "\nlabel: ", shuffled_test_labels[5])
```

```
## file name:  lab3_chest_xray/lab3_chest_xray/test/PNEUMONIA/person100_bacteria_482.jpeg
## label:  pneumonia
```

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 4 images do you see any attribute of the images that we may have to normalise before training a model?

Generally, before even looking at these images, there are a few things that we should consider that should be normalized in images before trying to train a model on them. Firstly, we need to normalize and adjust the brightness and contrast exposure settings so that the intensity values can help better standardize the appearance of each image. This is important because X-ray images can very often vary in contrast and brightness due to differing exposure settings used at different sittings, different machinery, and at different medical centers. Secondly, we need to standardize our image sizing. the size dimensions of different images may vary, so making sure to resize images to one standard size that is used uniformly throughout should help with normalizing images before training. Thirdly, orientation of images should be standardized. We need to make sure that all images have a consistent orientation that is used to help the model learn structures and features more effectively. If the orientation of body structures and images are not all the same in the images used for training, it will make it harder for the model to learn what is an important feature and what is not. Fourthly, we should try our best to enhance the quality of the images used to the best of our ability before they are used for training. Of course, people (especially children) may move a bit during their chest X-rays being taken. This can cause noise and decrease the visual quality of the images used. Therefore, including a sort of preprocessing step that helps decrease noise and enhances the quality of all of the images in a standardized way would be very helpful for normalization. Finally, as X-rays usually use a certain range of pixel intensity values for better identification of certain structures in the body, adjusting these values in a standardized way that is visually acceptable and makes the diagnosis more visually identifiable would be useful for normalization before training.

Now, based on the four images we received, I would say that the images do not look like they are taken all at the same zoom level and perhaps we need to adjust the contrast of some of them to match the other images better. We can see the patient’s shoulders and/or chins in some of the images are in different positions and in others, we

cannot see the same extent of their shoulders and/or their chins. We can also see that things like tubes and medical devices can be seen in some images, but are not present in others. Thus, we need to find a way to normalize the images to be able to remove unnecessary structures from the X-ray and have clear imaging at the same exact (or close enough) perspective and positioning as all of the other images. We also notice that some of the chest images are not centered, so this will also need to be normalized so that all images have the portion of the chest centered for better viewing. Another example that we can see that could require some normalizing is change in contrast as some of the images can look pretty different from the other images. This could be because there is a bit of a difference in the X-ray contrast settings, or perhaps the patient moved a bit during the X-ray. Nevertheless, we must find a way to better normalize these images before training a model.

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.

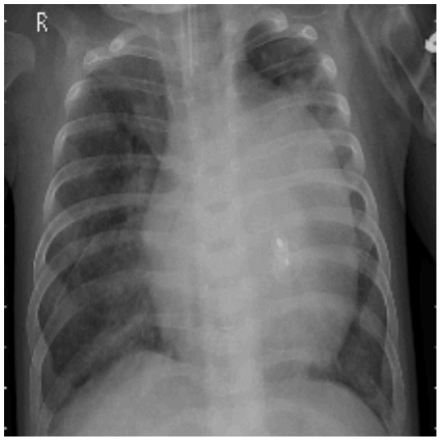
Furthermore, when working with the Torch package in R, it is essential to ensure that labels start from 1 instead of 0. In binary classification problems, the labels should be 1 and 2, representing the two classes. This adjustment is necessary to avoid errors when using the labels as indices for the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

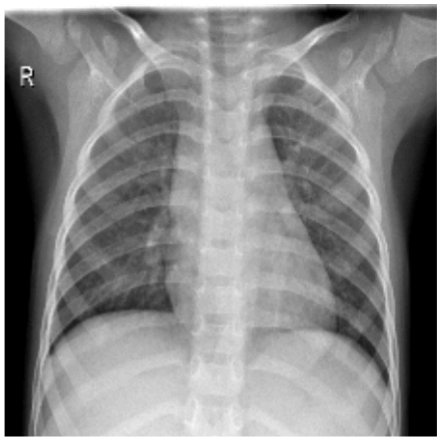
Now let's have an other look at images after doing pre processing.

```
# 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

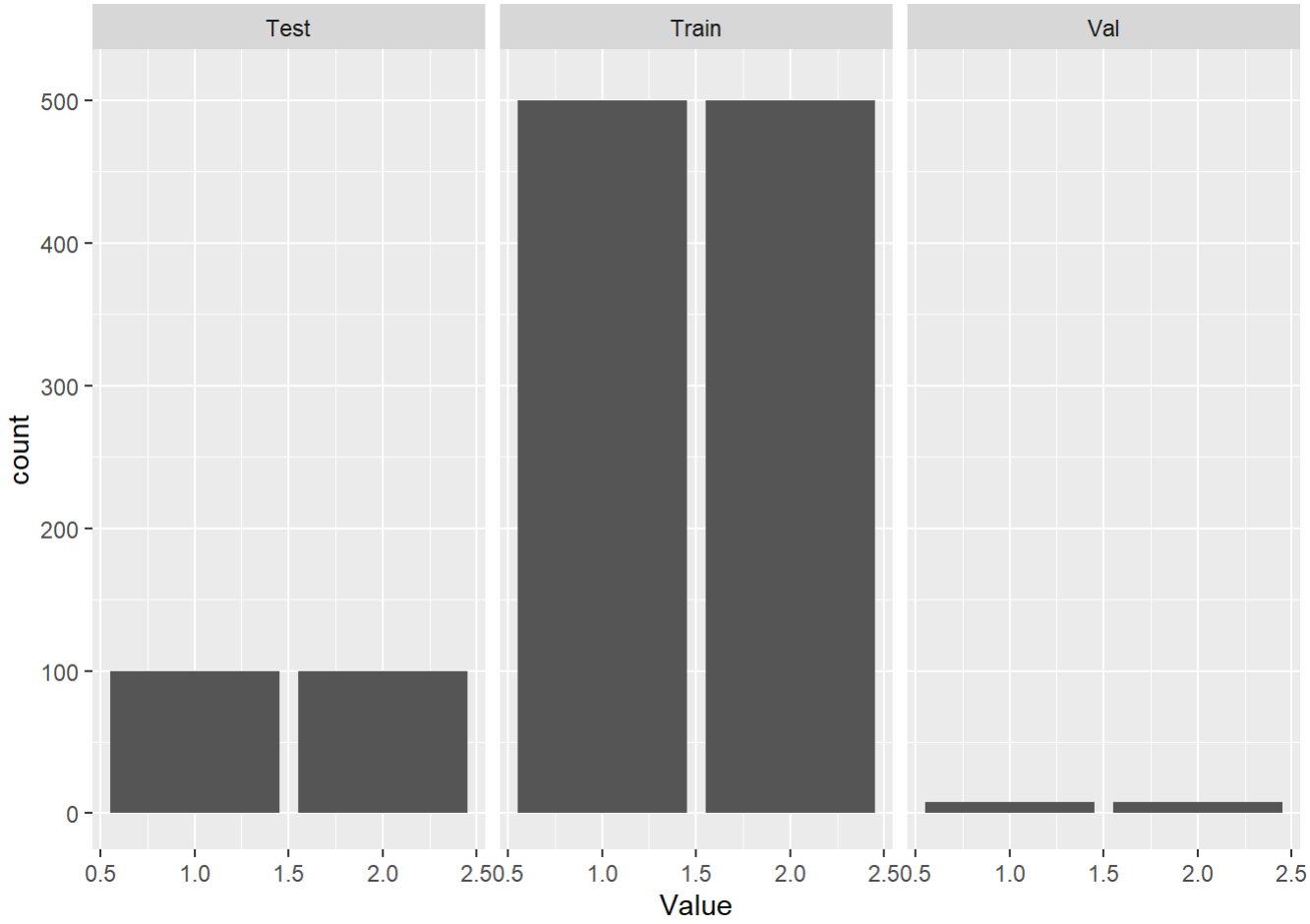


Pneumonia

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 was not balanced, it would be hard to train a machine learning model that is robust enough for what we want it to do. This is because thoroughly imbalanced datasets can lead to biased overfitting, for example for one particular sub-type. This means it can be biased toward performing well on the majority sub-type (i.e., the sub-type that it has more images for) but performs very poorly on the sub-type(s) that it has way less images of. There are some techniques that we could potentially use to help with an imbalanced dataset situation. For example, we could adjust our class weightings, potentially giving more importance to the minority classes during the training process of the modeling. This would help the model to pay a bit more attention on the underrepresented sub-type. We could also try doing some sort of image augmentation, such as rotating, flipping, cropping, or adding noise to the existing images that we already have to help generate more training data for the model and potentially make the model more generalizable. Lastly, we could try introducing a resampling technique, such as oversampling the minority sub-type or undersampling the majority sub-type. These are just a few options of techniques that could be useful. There are likely many more.

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.

```
# 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	128	-
(ReLU)		
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 classificat
ion task
  },

  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 c
lassification task)
  }
)
```

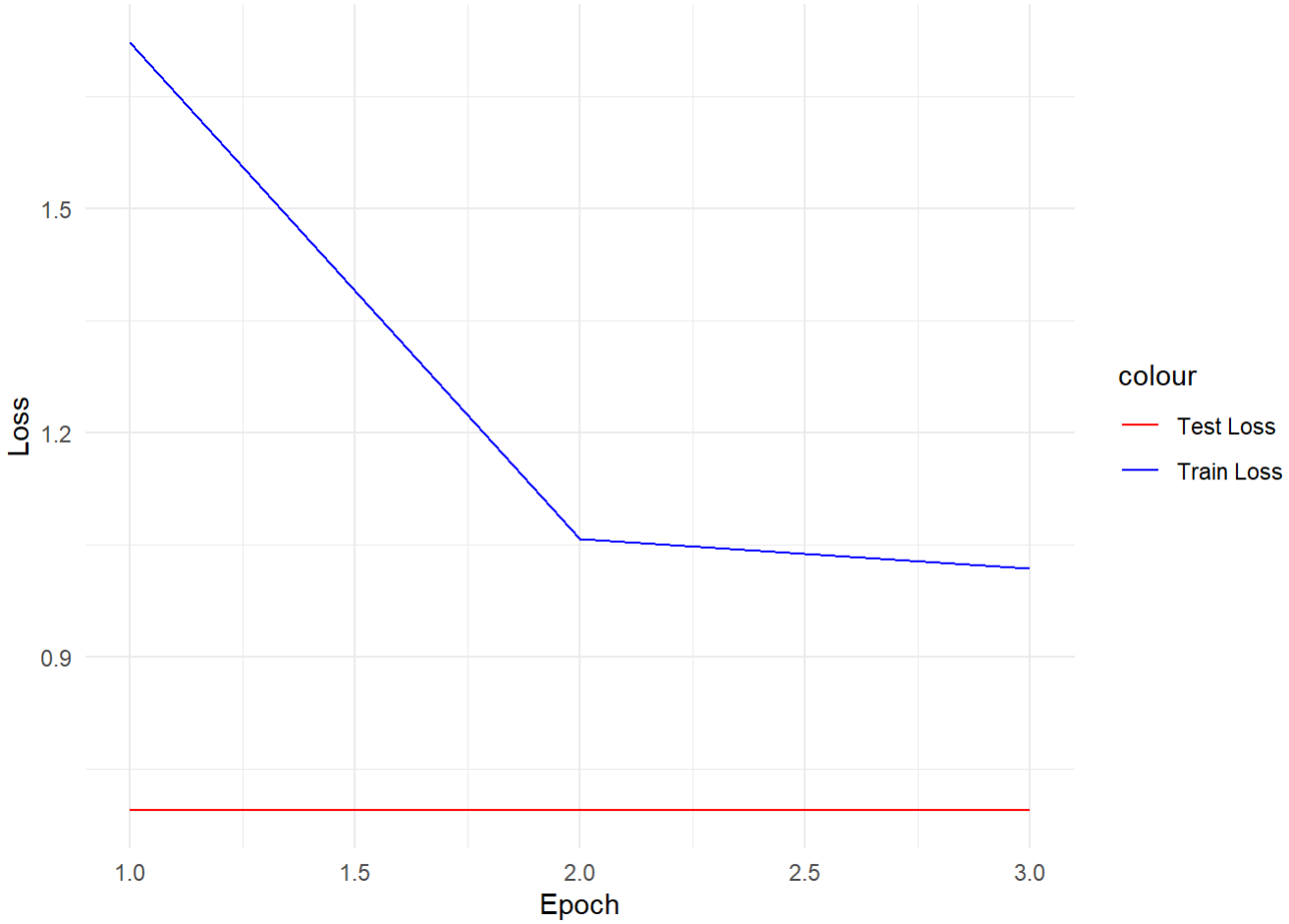
Train model

```
# 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$record
s$metrics$train[[1]]$acc, "\n")
  cat("Valid metrics: Loss: ", fitted$records$metrics$valid[[1]]$loss, " - Acc: ", fitted$record
s$metrics$valid[[1]]$acc, "\n")
  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$valid[[1]]$loss
  test_acc[epoch] <- fitted$records$metrics$valid[[1]]$acc
}
```

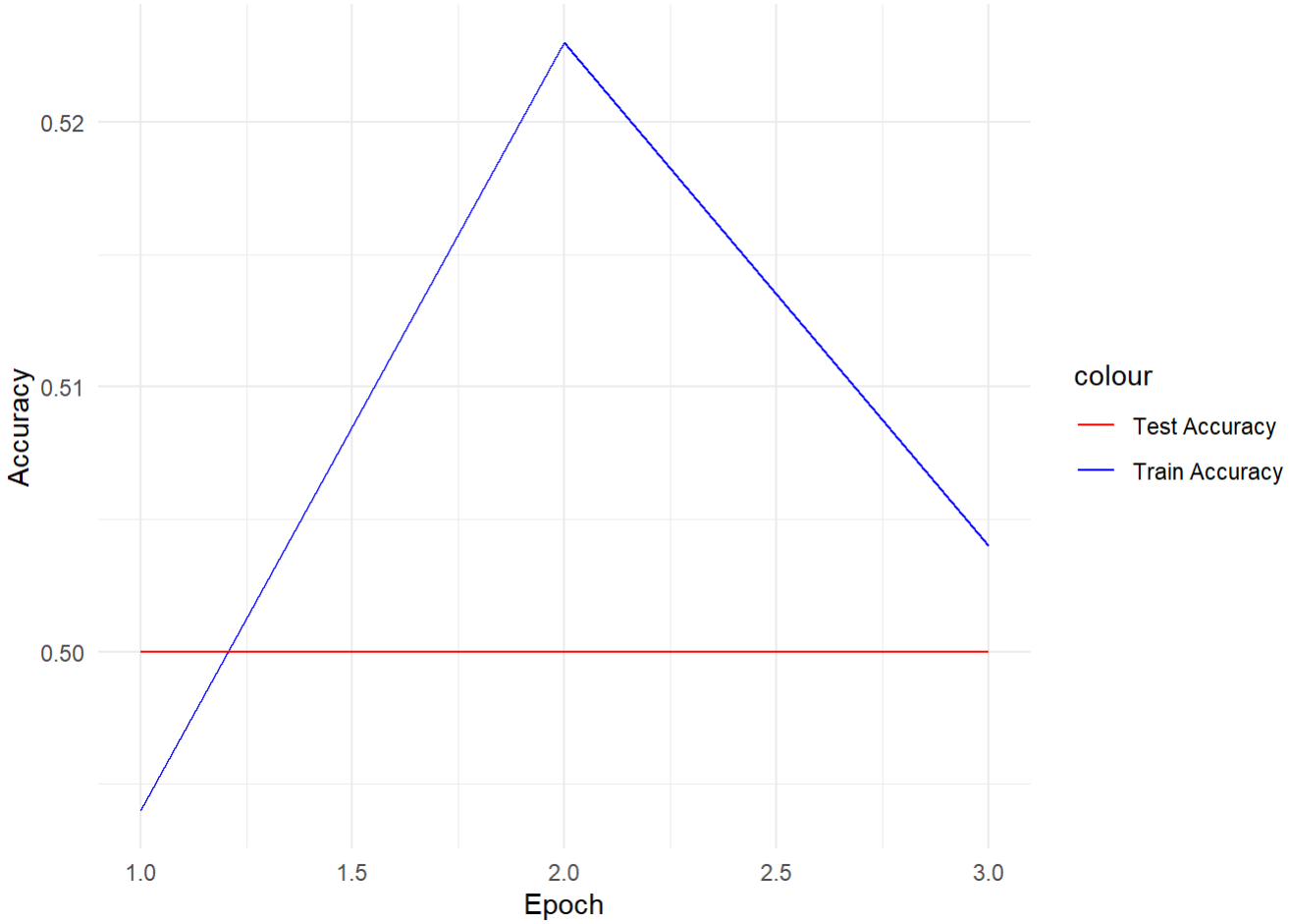
```
## Epoch  1 / 3
## Train metrics: Loss:  1.721979  - Acc:  0.494
## Valid metrics: Loss:  0.6957034  - Acc:  0.5
##
## Epoch  2 / 3
## Train metrics: Loss:  1.057964  - Acc:  0.523
## Valid metrics: Loss:  0.6935953  - Acc:  0.5
##
## Epoch  3 / 3
## Train metrics: Loss:  1.018346  - Acc:  0.504
## Valid metrics: Loss:  0.6940457  - 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?

Based on the plot results, the training loss slightly, but not significantly, fluctuates over the number of epochs, while the test loss remains relatively constant across all epochs. In terms of the accuracy plot, the training accuracy seems to fluctuate. The test accuracy seems to remain constant at 0.5 across all epochs.

As we are looking at a binary classification type of problem (i.e., distinguishing between whether an X-ray image is normal or whether the image shows a patient that has pneumonia), an accuracy of 0.5 is equivalent to just randomly guessing whether the image is of a normal patient or of a patient with pneumonia. The fact that the test accuracy seems to remain constant at 0.5 suggests that the model is basically making random guesses. This is further corroborated by the fact that the test loss also remains constant across all epochs, meaning that the model is not really learning to properly distinguish between the two classes. Finally, the changes in the training accuracy fluctuate, albeit minimally, and do not really reflect a great learning curve.

Overall, it does not really seem like our model is effectively managing to effectively classify our chest X-ray images into normal vs. pneumonia cases. There are a few things that could potentially contribute and be related to this issue. For example, we did do some pre-processing of the image data before training and we did check for any imbalances in our dataset. However, there still could be some things, such as orientation of the patient's body, noisiness, or even medical devices still visible in the images, that could still be making the model struggle to learn to distinguish features between the classes. This is at least true for some of the images that we tried to normalize and pre-process. Moreover, perhaps the complexity of our model does not match the architecture of the model we are trying to use. It could be that perhaps our model is just too overly complex; this could cause some overfitting of the training data, which would affect the model's generalizability. Some other things that could be contributing to

poor model performance could be suboptimal parameters, such as the number of epochs used. In our model, we only ran three epochs, which might be insufficient enough for this model to learn effectively. These are just a few things that could contribute to our model's poor performance and reasons why they may be contributing.

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 are your suggestions to solve this problem? How could we improve this model?

We could perform a combination of things to try to help solve this problem and improve our model. For example, a combination of better data pre-processing, better parameter tuning, optimization techniques, and architecture adjustments could potentially improve our model and enhance its ability to learn more effectively and generalize better to unseen examples. Some examples that have been discussed previously that could be used to improve data pre-processing are better normalization and standardization of images to help the model converge faster and potentially increase performance, as well as performing some data augmentation. By performing some data augmentation techniques, we could artificially increase the size of our training dataset, which could in turn help improve the model's ability to generalize to unseen examples. Other suggestions of examples that could help improve the model are to maybe experiment with different optimization algorithms or introduce some cross-validation into our methodology to ensure the consistency of the model's performance across different data subsets. Finally, increasing the number of epochs used could allow the model a bit more time to learn from the data, and potentially experimenting with different model architectures to find one that better fits the problem could be very useful here.

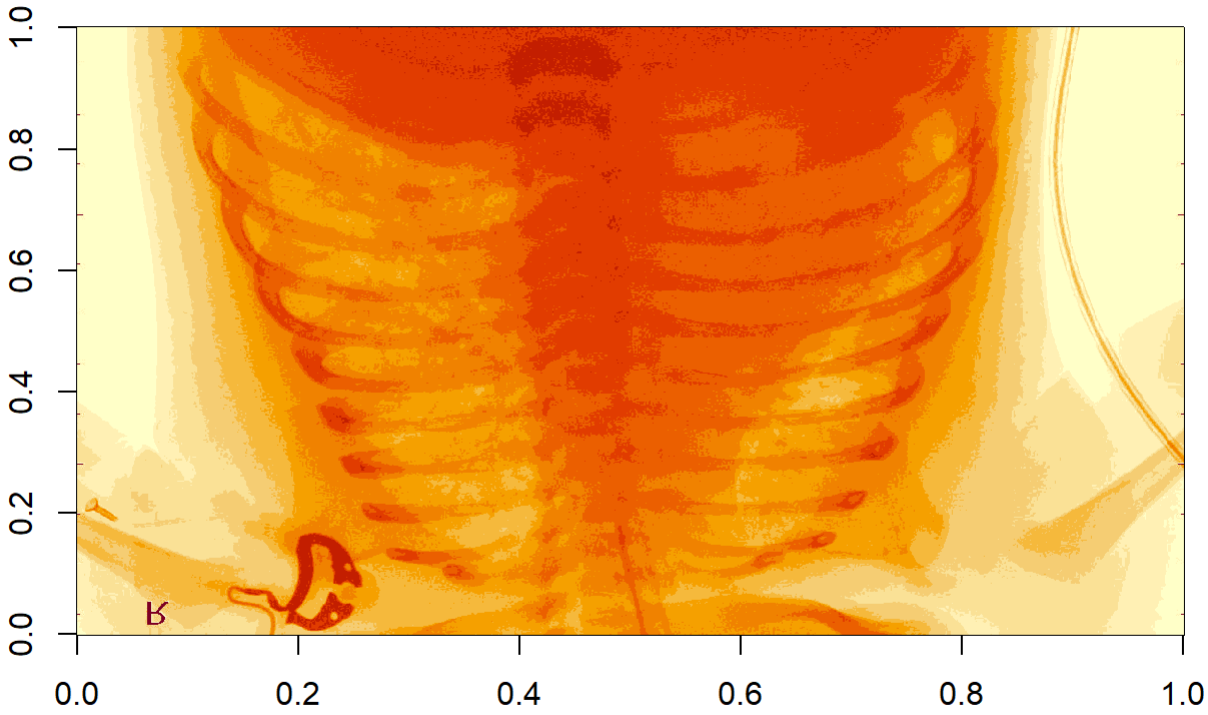
Data Augmentation

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?

There are several potential drawbacks to incorporating data augmentation into our model-building methodology. For example, although adding in more data into our training dataset can be very beneficial for generalizability and model performance, we are also actively increasing the size of our training dataset. This means that more data will need to be processed during each epoch, which can lead to much longer training times and can require much more computational resource power. In addition, although we use augmentation to help prevent overfitting, we still run the risk of our model being overfitted to the augmented data patterns, especially if the transformations we use are not heterogeneous enough. This could potentially limit the generalizability of our model to unseen examples. Another example of a drawback to using data augmentation is that, again, it increases the complexity of implementation, which may require an expert touch for fine-tuning code that may not be readily available to everyone. Finally, although data augmentation is used to create variations of our existing data, we also need to be aware that it has the potential to also create non-realistic variations, potentially introducing noise and creating variations that are not representative of real-world scenarios. We have to keep in mind too that too much data augmentation might not actually provide any further improvement to the model and could actually decrease the model's overall performance if we too excessively alter our original image data. These are just a few drawbacks and disadvantages of data augmentation.

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, publicly available visual database that contains annotated images that are composed to be used in multiple computer vision tasks. It contains over 14 million images in high resolution and millions of annotated images that are organized in accordance to a hierarchy established by WordNet. WordNet helps link words into semantic relations that can be used in natural language processing and computational linguistics (see Figure below) (Ünal and Başçiftçi, 2022). It is useful and often used in visual object recognition software research and can be useful in machine learning techniques, such as transfer learning. In fact, it is one of the largest resources available to the public for training deep learning models in image recognition tasks. The ImageNet Large Scale Visual Recognition Challenge 2012 is a competition that had the goal of estimating the content of images with the purpose of retrieval and automatic annotation using a subset of labeled ImageNet data as training. The competition was set to use 10,000,000 labeled images depicting 10,000+ object categories for its training. This competition now occurs annually and has become a benchmark for evaluating the performance of algorithms that use large-scale visual recognition. The competition requires participants to perform image classification, object detection, and image segmentation. The full original ImageNet dataset (commonly referred to as ImageNet 21-K) contains 14,197,122 images and 21,841 synsets or classes that have been indexed. ImageNet-1K (which is also referred in the literature as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012–2017 image classification and localization dataset that involved 1,000 classes) contains 1,281,167 training images, 50,000 images for validation, and 100,000 images for testing.

ImageNet may help us for what we want to do because transfer learning may be a technique that may potentially benefit our model's training and overall performance. If we could use pre-trained models that have already been training from ImageNet to recognize certain structures, such as structures that are a part of the chest (e.g., the rib

cage, the lungs, the heart, shoulders, etc.), this could definitely enhance the overall performance of chest X-ray classification as it would already know the basic structures of what a chest has already and it could then focus on specific features that are found in pneumonia patients. In addition, using ImageNet and transfer learning to adapt an already pre-trained model to use and fine-tune to our specific dataset could be more effective than having us train our model entirely from scratch. This could dramatically reduce the amount of labeled data that we would need to use, as well as potentially reduce the computational resources that could be required. There are certainly other reasons why using something like ImageNet could help us, but these are just a few examples.

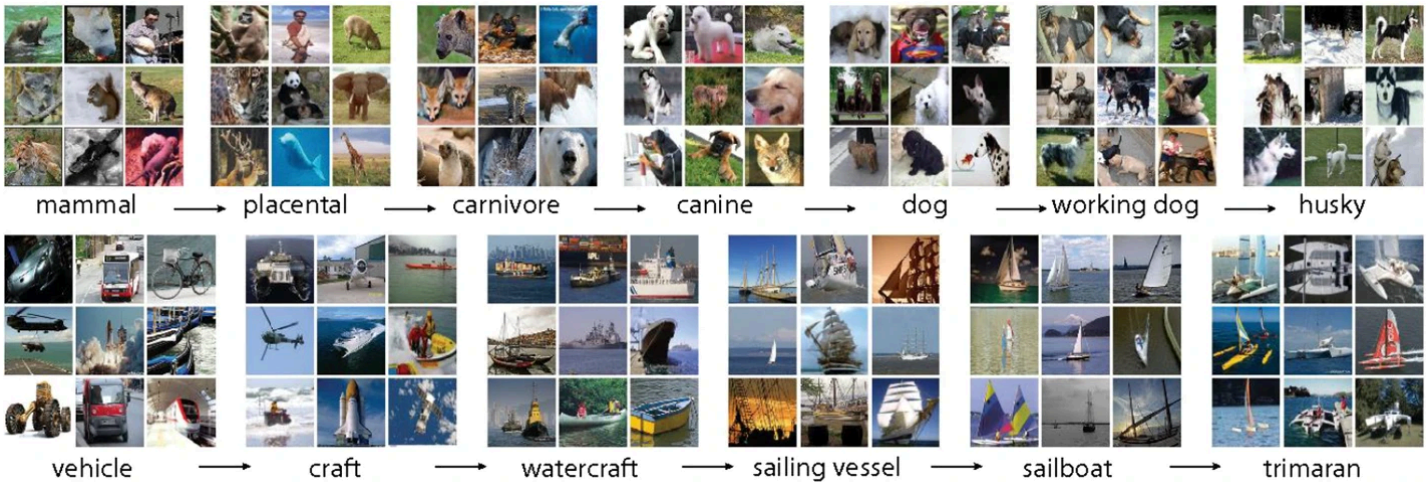


Figure 3. An example showing how the ImageNet database is organized into a WordNet hierarchy; WordNet basically links words with semantic relations so that they can be used for different research work, including computational linguistics and language processing (Ünal and Başçiftçi, 2022).

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

First of all, MobileNet architecture is pre-trained on the ImageNet dataset, which means that it comes with already extensive pre-training before we even use it for our own dataset. This would allow us to have more robust feature extraction capabilities that could potentially generalize well for use in our binary image classification. We would also, potentially, not have to augment our data as much, as well as not have to use so many computational resources. In addition, the architecture of MobileNet seems flexible enough and easily adaptable to different input sizes and shapes, making it potentially easily customizable for our chest X-ray image dataset. The fact that we can adjust its size and complexity, such as by adjusting the width or resolution multipliers to control for computational cost and/or parameter numbers, depending on what we need to use it for means that we can fit what we want for our practical assignment perfectly. This also means that we can create a model that performs the necessary task in a more timely manner. This could be particularly beneficial if, for example, we wanted to deploy this model to be used in areas with limited computational resources, but that have access to a smartphone. It could allow, for example, for very rural regions or developing countries that have very limited access to medical specialists to be able to tell in a timely manner whether a patient has pneumonia or not without needing to have a large and expensive computer with high computing power to run for hours to give them an answer.

Moreover, the fact that the use of this architecture has been demonstrated to be effective for use in various medical imaging capacities (e.g., Lu et al., 2020; Zhang et al., 2023), including for classifying chest X-rays (e.g., Tangudu et al., 2022), means that it could potentially work for us and could help improve our performance results. In other words, it has the ability to better generalize from pre-trained weights and using that on new medical imaging data like ours could make it a strong candidate to help us improve the performance of our classification model. It also is designed to be a time-efficient, lightweight neural network, using depthwise separable

convolutions to reduce the number of parameters and computational costs. Finally, MobileNet also is able to support various types of regularization techniques, including batch normalization, which would help us ensure the prevention of overfitting our model and help improve its generalizability on our test set. As we are using a relatively small dataset (compared to what ImageNet has to offer), which is common when it comes to medical image analyses, the potential to be able to use such regularization techniques are highly useful to improve our model's performance.

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

In general, under default settings (i.e., a width multiplier of 1.0 and a resolution multiplier of 1.0), a MobileNet model has about 4.2 million parameters. However, by adjusting the width multiplier and/or resolution multiplier, the number of parameters can be increased or decreased accordingly. In comparison to better performing networks available in Torch, such as VGG-16 and GoogleNet (A.K.A. Inception-v1), MobileNet generally has fewer parameters. For instance, VGG-16 has approximately 138 million parameters, while GoogleNet has around 6.8 million. Thus, in terms of the number of parameters, MobileNet is much lighter, especially as compared to these better-performing networks available on Torch. Although it has less parameters, this allows it to be more efficient and suitable for deployment in certain areas and in certain devices that have resource limitations, such as smartphones.

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 classif
ication task
mobilenet
```

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

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

For this, the different network architecture that we will try is using the MobileNet architecture. We will compare running the model using the network architecture used previously in this practical using 3 epochs to using the MobileNet architecture using 3 epochs. We will then also compare it to using the MobileNet architecture using 20 epochs (i.e., the very last plots at the end of this practical document).

Based on both sets of MobileNet plots, we notice that training loss gradually decreases and, at a certain point, stabilizes, which tells us that, over several epochs, the model learns from the training data pretty well. However, in both cases, we see significant fluctuation in the test loss part of the plot, which likely tells us that there could be some overfitting and instability going on in our model. In terms of the accuracy plots, the training accuracy in both cases increase significantly quickly and tends to remain very high. When looking at the test accuracy portion of the plots, we see that test accuracy basically remains pretty constant at 0.5, which lets us know that our model does not seem to generalize well and could confirm overfitting.

Overall, when comparing all of these plots from different network architectures, we notice that there is likely some overfitting going on in both network architecture models and the test accuracy pretty consistently remains flat at about 0.5 across all of the scenarios we have depicted in this practical. This, again, further indicates that these models we have created do not generalize well to unseen examples. So, I would say in the grand scheme of things, none of these models performed any better than the other, especially in terms of generalizability. However, if we are to consider things like training performance, I would say perhaps the MobileNet model that is trained for 20 epochs performs better as it achieves almost a near-perfect training accuracy even though it completely fails at generalization.

```
# Load necessary packages
library(torch)
library(torchvision)
library(luz)
library(imager)

# Define base directory
base_dir <- "lab3_chest_xray/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_norma
l)), "\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
```

```
# Combine paths and create labels
train_image_paths <- c(train_pn, train_normal)
train_labels <- c(rep(1, length(train_pn)), rep(0, length(train_normal)))

test_image_paths <- c(test_pn, test_normal)
test_labels <- c(rep(1, length(test_pn)), rep(0, length(test_normal)))

val_image_paths <- c(val_pn, val_normal)
val_labels <- c(rep(1, length(val_pn)), rep(0, length(val_normal)))

# Adjust labels to start from 1
train_labels <- train_labels + 1
test_labels <- test_labels + 1
val_labels <- val_labels + 1

# Function to preprocess the images
process_images <- function(image_paths) {
  img_size <- 224  # Desired image size

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

  # Loop through each image path
  for (image_path in image_paths) {
    # Read the image
    img <- load.image(image_path)

    # Normalize the image
    img_normalized <- img / 255

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

    # Convert the image to an array and store in the list
    X <- c(X, list(as.array(img_resized)))
  }

  # Convert the list to a tensor
  X <- torch_tensor(array(unlist(X), dim = c(length(X), 3, img_size, img_size)))

  return(X)
}

# Preprocess images
train_X <- process_images(train_image_paths)
test_X <- process_images(test_image_paths)
val_X <- process_images(val_image_paths)

train_y <- torch_tensor(train_labels, dtype = torch_long())
test_y <- torch_tensor(test_labels, dtype = torch_long())
val_y <- torch_tensor(val_labels, dtype = torch_long())
```

```r
library(torch)
library(torchvision)
library(luz)
library(ggplot2)

# Define a custom dataset class
ImageDataset <- dataset(
  name = "ImageDataset",
  initialize = function(X, y) {
    self$data <- X
    self$labels <- y
  },
  .getitem = function(i) {
    list(x = self$data[i,,,], y = self$labels[i])
  },
  .length = function() {
    dim(self$data)[1]
  }
)

# Create dataset instances
train_dataset <- ImageDataset(train_X, train_y)
test_dataset <- ImageDataset(test_X, test_y)
val_dataset <- ImageDataset(val_X, val_y)

# Create dataloaders
train_dataloader <- dataloader(train_dataset, batch_size = 16, shuffle = TRUE)
test_dataloader <- dataloader(test_dataset, batch_size = 16)
val_dataloader <- dataloader(val_dataset, batch_size = 16)

# Define the MobileNet model
mobilenet_model <- model_mobilenet_v2(pretrained = TRUE)
mobilenet_model$classifier <- nn_sequential(
  nn_linear(mobilenet_model$last_channel, 2)
)

# 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)

# Loss function and optimizer
criterion <- nn_cross_entropy_loss()
optimizer <- optim_adam(mobilenet_model$parameters, lr = 0.001)

# Training loop
for (epoch in 1:num_epochs) {
  mobilenet_model$train()  # Ensure model is in training mode
  epoch_train_loss <- 0
```

```
  epoch_train_correct <- 0
  epoch_train_total <- 0
  for (batch in enumerate(train_dataloader)) {
    optimizer$zero_grad()

    output <- mobilenet_model(batch[[1]])
    loss <- criterion(output, batch[[2]])
    loss$backward()
    optimizer$step()

    # Compute training accuracy
    predicted <- torch_argmax(output, dim = 2)
    epoch_train_total <- epoch_train_total + batch[[2]]$size(1)
    epoch_train_correct <- epoch_train_correct + (predicted == batch[[2]])$sum()$item()

    epoch_train_loss <- epoch_train_loss + loss$item()
  }

  # Store training loss and accuracy
  train_loss[epoch] <- epoch_train_loss / length(train_dataloader)
  train_acc[epoch] <- epoch_train_correct / epoch_train_total

  # Evaluation on test set
  mobilenet_model$eval()
  epoch_test_loss <- 0
  epoch_test_correct <- 0
  epoch_test_total <- 0
  for (batch in enumerate(test_dataloader)) {
    output <- mobilenet_model(batch[[1]])
    loss <- criterion(output, batch[[2]])

    # Compute test accuracy
    predicted <- torch_argmax(output, dim = 2)
    epoch_test_total <- epoch_test_total + batch[[2]]$size(1)
    epoch_test_correct <- epoch_test_correct + (predicted == batch[[2]])$sum()$item()

    epoch_test_loss <- epoch_test_loss + loss$item()
  }

  # Store test loss and accuracy
  test_loss[epoch] <- epoch_test_loss / length(test_dataloader)
  test_acc[epoch] <- epoch_test_correct / epoch_test_total

  cat(sprintf("Epoch: %d, Train Loss: %.4f, Train Acc: %.4f, Test Loss: %.4f, Test Acc: %.4f\n",
              epoch, train_loss[epoch], train_acc[epoch], test_loss[epoch], test_acc[epoch]))
}
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 1, Train Loss: 0.6041, Train Acc: 0.6980, Test Loss: 4.6976, Test Acc: 0.5000
```

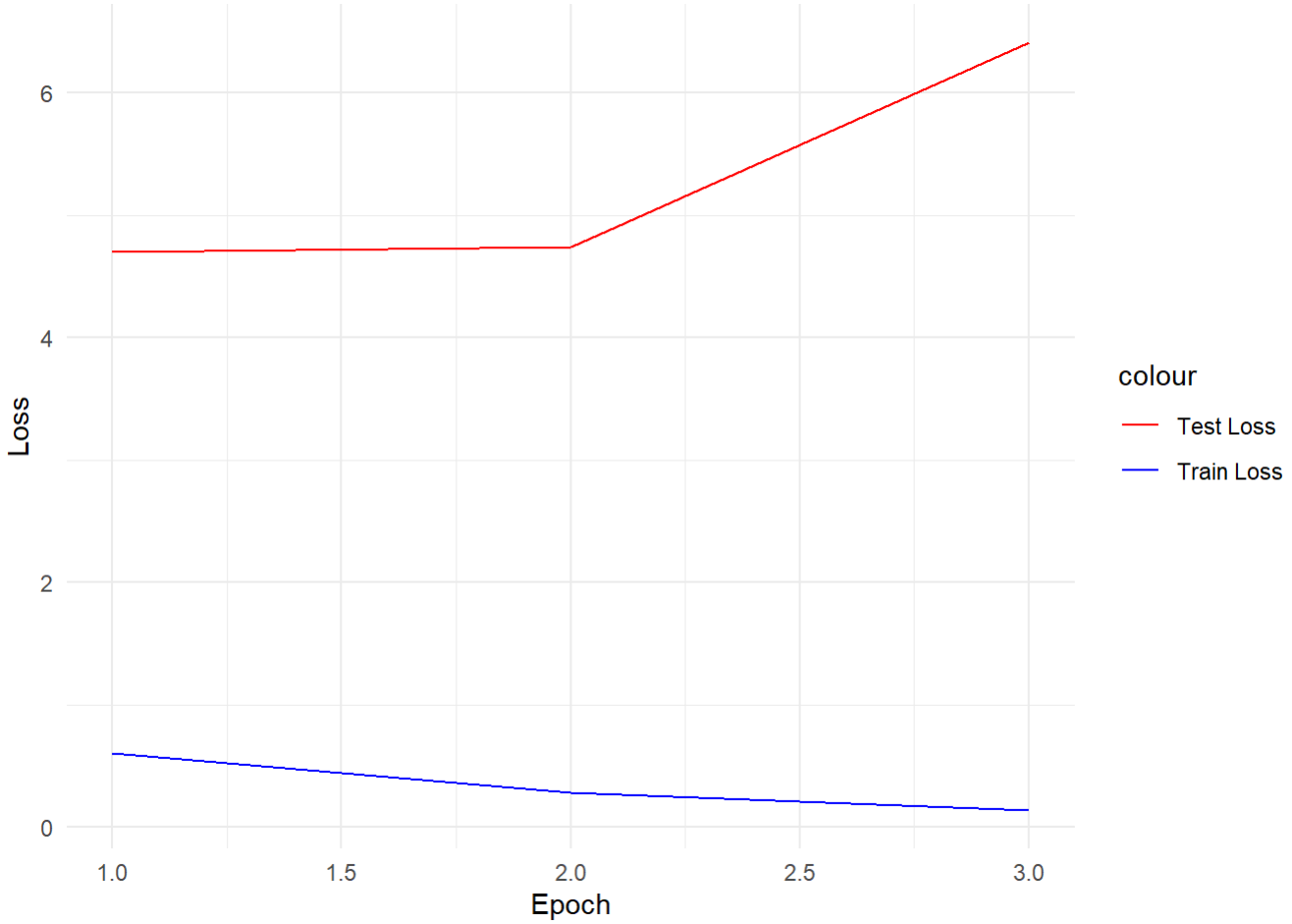
```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 2, Train Loss: 0.2802, Train Acc: 0.8920, Test Loss: 4.7443, Test Acc: 0.5000
```

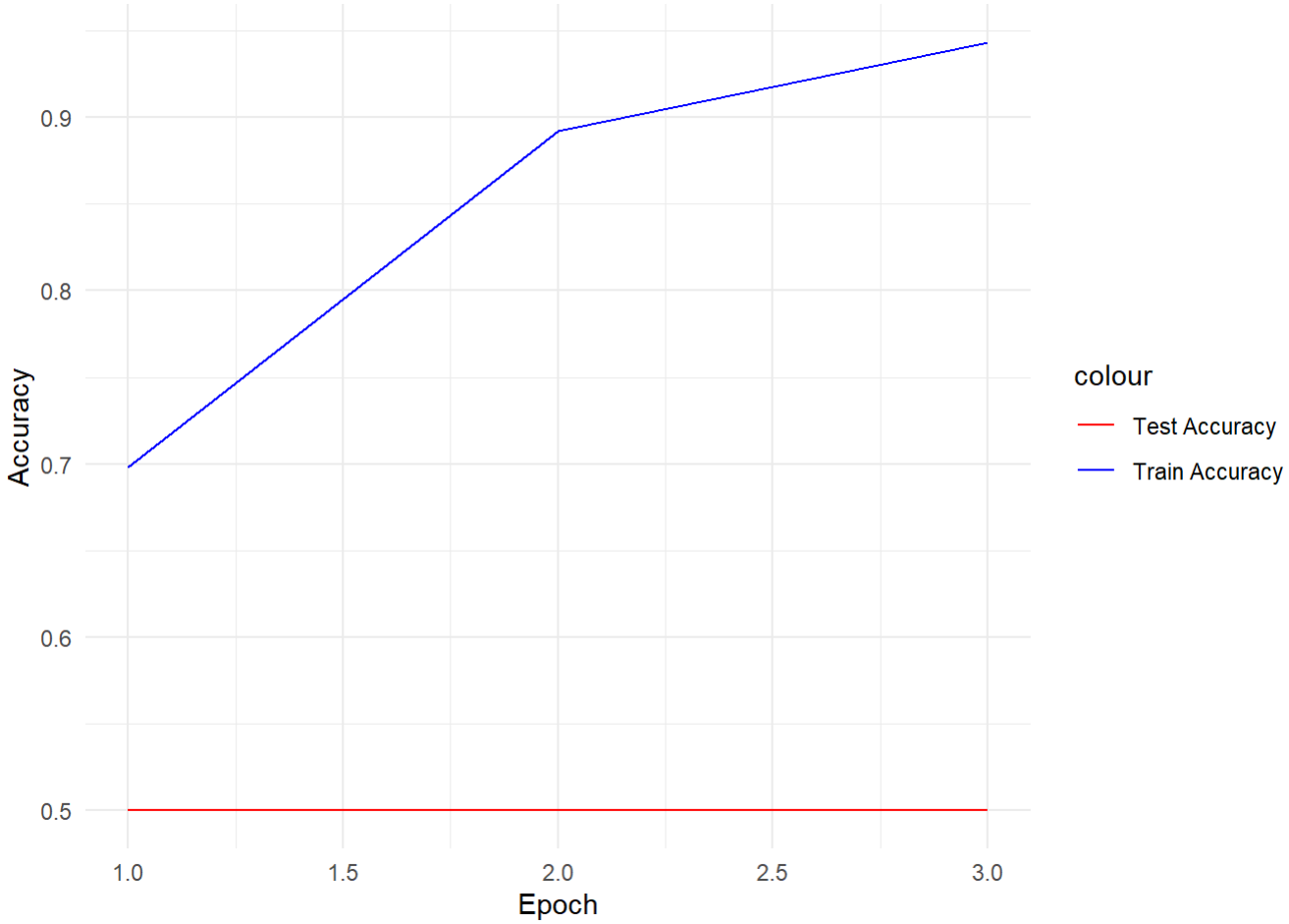
```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 3, Train Loss: 0.1358, Train Acc: 0.9430, Test Loss: 6.4101, Test Acc: 0.5000
```

```
# Plot learning curves
metrics_df <- data.frame(
  Epoch = 1:num_epochs,
  Train_Loss = train_loss,
  Test_Loss = test_loss,
  Train_Accuracy = train_acc,
  Test_Accuracy = test_acc
)

# Plot the train and test loss
loss_plot <- ggplot(data = metrics_df, aes(x = Epoch)) +
  geom_line(aes(y = Train_Loss, color = "Train Loss")) +
  geom_line(aes(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_plot <- ggplot(data = metrics_df, aes(x = Epoch)) +
  geom_line(aes(y = Train_Accuracy, color = "Train Accuracy")) +
  geom_line(aes(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)
```



#With 20 epochs used

```r
# Load necessary packages
library(torch)
library(torchvision)
library(luz)
library(imager)

# Define base directory
base_dir <- "lab3_chest_xray/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_norma
l)), "\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
```

```
# Combine paths and create labels
train_image_paths <- c(train_pn, train_normal)
train_labels <- c(rep(1, length(train_pn)), rep(0, length(train_normal)))

test_image_paths <- c(test_pn, test_normal)
test_labels <- c(rep(1, length(test_pn)), rep(0, length(test_normal)))

val_image_paths <- c(val_pn, val_normal)
val_labels <- c(rep(1, length(val_pn)), rep(0, length(val_normal)))

# Adjust labels to start from 1
train_labels <- train_labels + 1
test_labels <- test_labels + 1
val_labels <- val_labels + 1

# Function to preprocess the images
process_images <- function(image_paths) {
  img_size <- 224  # Desired image size

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

  # Loop through each image path
  for (image_path in image_paths) {
    # Read the image
    img <- load.image(image_path)

    # Normalize the image
    img_normalized <- img / 255

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

    # Convert the image to an array and store in the list
    X <- c(X, list(as.array(img_resized)))
  }

  # Convert the list to a tensor
  X <- torch_tensor(array(unlist(X), dim = c(length(X), 3, img_size, img_size)))

  return(X)
}

# Preprocess images
train_X <- process_images(train_image_paths)
test_X <- process_images(test_image_paths)
val_X <- process_images(val_image_paths)

train_y <- torch_tensor(train_labels, dtype = torch_long())
test_y <- torch_tensor(test_labels, dtype = torch_long())
val_y <- torch_tensor(val_labels, dtype = torch_long())
```

```r
library(torch)
library(torchvision)
library(luz)
library(ggplot2)

# Define a custom dataset class
ImageDataset <- dataset(
  name = "ImageDataset",
  initialize = function(X, y) {
    self$data <- X
    self$labels <- y
  },
  .getitem = function(i) {
    list(x = self$data[i,,,], y = self$labels[i])
  },
  .length = function() {
    dim(self$data)[1]
  }
)

# Create dataset instances
train_dataset <- ImageDataset(train_X, train_y)
test_dataset <- ImageDataset(test_X, test_y)
val_dataset <- ImageDataset(val_X, val_y)

# Create dataloaders
train_dataloader <- dataloader(train_dataset, batch_size = 16, shuffle = TRUE)
test_dataloader <- dataloader(test_dataset, batch_size = 16)
val_dataloader <- dataloader(val_dataset, batch_size = 16)

# Define the MobileNet model
mobilenet_model <- model_mobilenet_v2(pretrained = TRUE)
mobilenet_model$classifier <- nn_sequential(
  nn_linear(mobilenet_model$last_channel, 2)
)

# Set the number of epochs
num_epochs <- 20

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

# Loss function and optimizer
criterion <- nn_cross_entropy_loss()
optimizer <- optim_adam(mobilenet_model$parameters, lr = 0.001)

# Training loop
for (epoch in 1:num_epochs) {
  mobilenet_model$train()  # Ensure model is in training mode
  epoch_train_loss <- 0
```

```
  epoch_train_correct <- 0
  epoch_train_total <- 0
  for (batch in enumerate(train_dataloader)) {
    optimizer$zero_grad()

    output <- mobilenet_model(batch[[1]])
    loss <- criterion(output, batch[[2]])
    loss$backward()
    optimizer$step()

    # Compute training accuracy
    predicted <- torch_argmax(output, dim = 2)
    epoch_train_total <- epoch_train_total + batch[[2]]$size(1)
    epoch_train_correct <- epoch_train_correct + (predicted == batch[[2]])$sum()$item()

    epoch_train_loss <- epoch_train_loss + loss$item()
  }

  # Store training loss and accuracy
  train_loss[epoch] <- epoch_train_loss / length(train_dataloader)
  train_acc[epoch] <- epoch_train_correct / epoch_train_total

  # Evaluation on test set
  mobilenet_model$eval()
  epoch_test_loss <- 0
  epoch_test_correct <- 0
  epoch_test_total <- 0
  for (batch in enumerate(test_dataloader)) {
    output <- mobilenet_model(batch[[1]])
    loss <- criterion(output, batch[[2]])

    # Compute test accuracy
    predicted <- torch_argmax(output, dim = 2)
    epoch_test_total <- epoch_test_total + batch[[2]]$size(1)
    epoch_test_correct <- epoch_test_correct + (predicted == batch[[2]])$sum()$item()

    epoch_test_loss <- epoch_test_loss + loss$item()
  }

  # Store test loss and accuracy
  test_loss[epoch] <- epoch_test_loss / length(test_dataloader)
  test_acc[epoch] <- epoch_test_correct / epoch_test_total

  cat(sprintf("Epoch: %d, Train Loss: %.4f, Train Acc: %.4f, Test Loss: %.4f, Test Acc: %.4f\n",
              epoch, train_loss[epoch], train_acc[epoch], test_loss[epoch], test_acc[epoch]))
}
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 1, Train Loss: 0.6644, Train Acc: 0.6340, Test Loss: 0.7481, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 2, Train Loss: 0.2401, Train Acc: 0.9150, Test Loss: 2.3233, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 3, Train Loss: 0.1466, Train Acc: 0.9460, Test Loss: 0.8368, Test Acc: 0.5050
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 4, Train Loss: 0.0909, Train Acc: 0.9670, Test Loss: 1.4764, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 5, Train Loss: 0.0865, Train Acc: 0.9620, Test Loss: 1.5163, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 6, Train Loss: 0.0301, Train Acc: 0.9910, Test Loss: 3.4934, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 7, Train Loss: 0.0644, Train Acc: 0.9730, Test Loss: 1.4112, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 8, Train Loss: 0.0549, Train Acc: 0.9830, Test Loss: 3.6180, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 9, Train Loss: 0.0159, Train Acc: 0.9960, Test Loss: 4.8526, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 10, Train Loss: 0.0043, Train Acc: 0.9990, Test Loss: 5.6406, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 11, Train Loss: 0.0249, Train Acc: 0.9930, Test Loss: 11.2709, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 12, Train Loss: 0.0240, Train Acc: 0.9950, Test Loss: 7.1607, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 13, Train Loss: 0.0088, Train Acc: 0.9990, Test Loss: 6.6032, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 14, Train Loss: 0.0041, Train Acc: 0.9970, Test Loss: 5.8244, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 15, Train Loss: 0.0050, Train Acc: 0.9980, Test Loss: 0.7479, Test Acc: 0.5150
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 16, Train Loss: 0.0682, Train Acc: 0.9770, Test Loss: 11.5710, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 17, Train Loss: 0.0526, Train Acc: 0.9840, Test Loss: 15.1703, Test Acc: 0.5000
```

```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 18, Train Loss: 0.0062, Train Acc: 0.9980, Test Loss: 10.3240, Test Acc: 0.5000
```

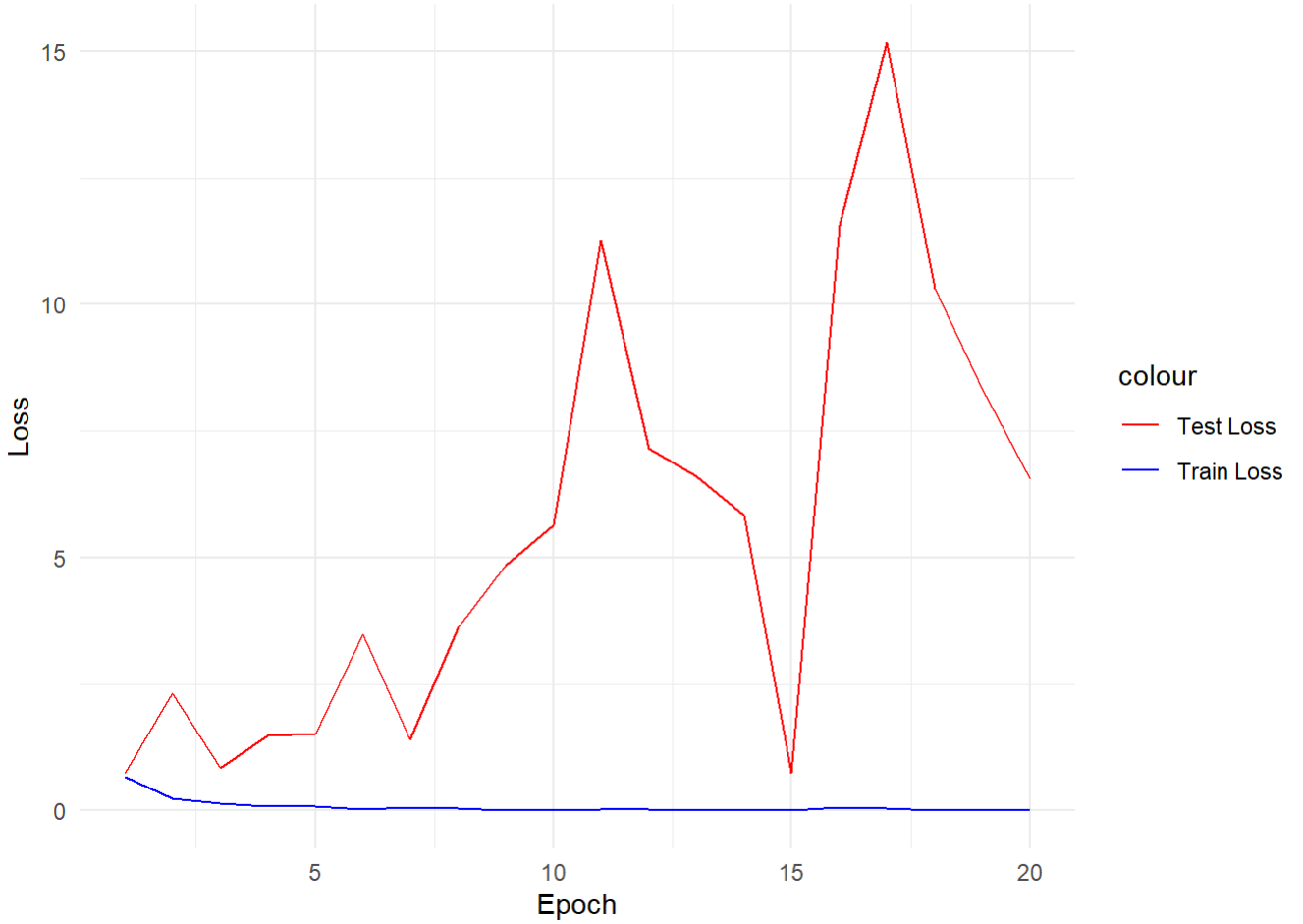
```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 19, Train Loss: 0.0019, Train Acc: 1.0000, Test Loss: 8.3479, Test Acc: 0.5000
```

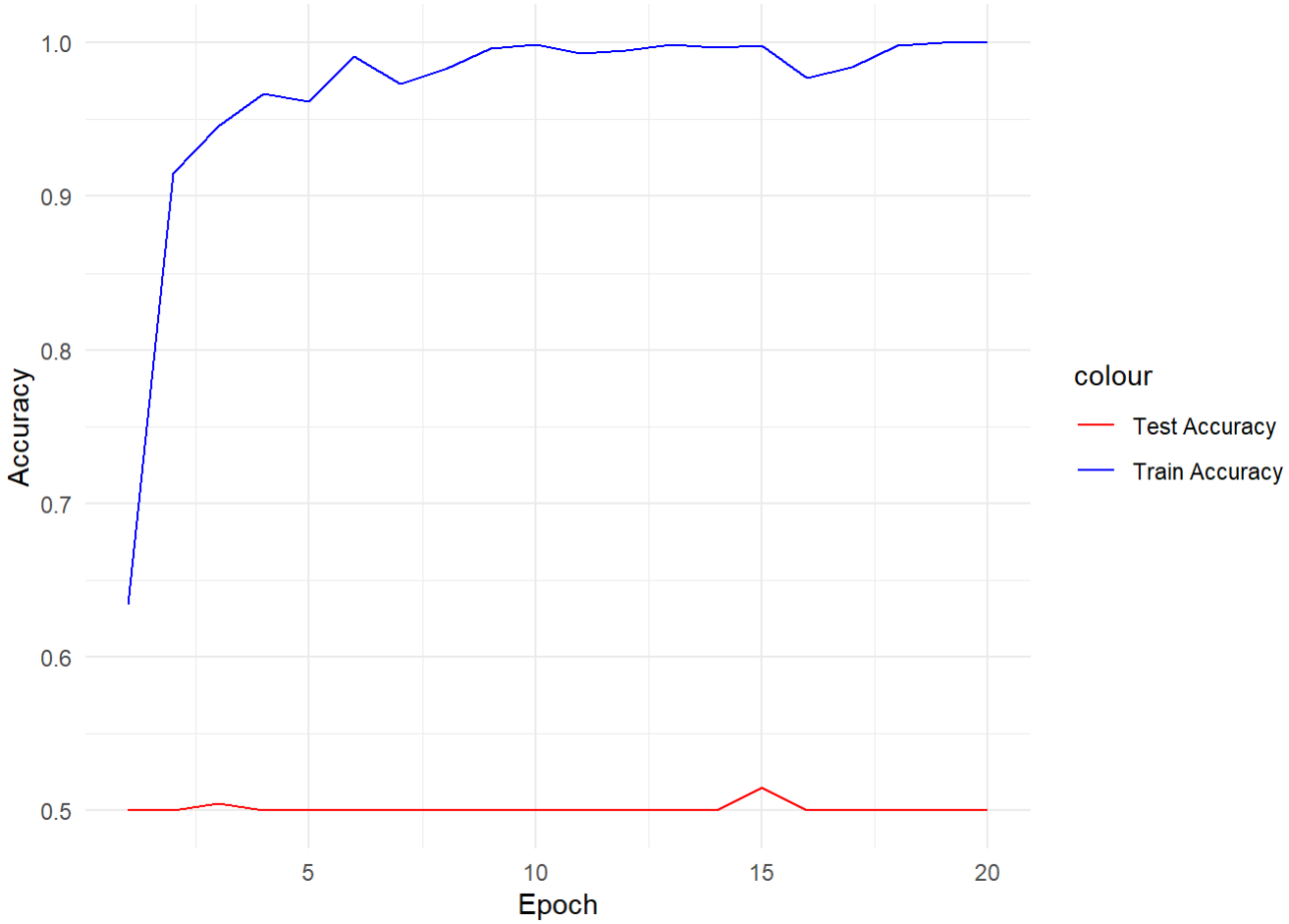
```
## Warning: The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
## The `enumerate` construct is deprecated in favor of the `coro::loop` syntax.
## • See https://github.com/mlverse/torch/issues/558 for more information.
```

```
## Epoch: 20, Train Loss: 0.0010, Train Acc: 1.0000, Test Loss: 6.5485, Test Acc: 0.5000
```

```
# Plot learning curves
metrics_df <- data.frame(
  Epoch = 1:num_epochs,
  Train_Loss = train_loss,
  Test_Loss = test_loss,
  Train_Accuracy = train_acc,
  Test_Accuracy = test_acc
)

# Plot the train and test loss
loss_plot <- ggplot(data = metrics_df, aes(x = Epoch)) +
  geom_line(aes(y = Train_Loss, color = "Train Loss")) +
  geom_line(aes(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_plot <- ggplot(data = metrics_df, aes(x = Epoch)) +
  geom_line(aes(y = Train_Accuracy, color = "Train Accuracy")) +
  geom_line(aes(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://medium.com/@kemalgunay/getting-started-with-image-preprocessing-in-r-52c7d153b381) https://cran.r-project.org/web/packages/magick/vignettes/intro.html (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://www.datanovia.com/en/blog/easy-image-processing-in-r-using-the-magick-package/) https://dahtah.github.io/imager/imager.html (https://dahtah.github.io/imager/imager.html) https://rdrr.io/github/mlverse/torchvision/api/ (https://rdrr.io/github/mlverse/torchvision/api/) https://github.com/brandonyph/Torch-for-R-CNN-Example (https://github.com/brandonyph/Torch-for-R-CNN-Example)

References:

Adegbola, R. A. (2012). Childhood pneumonia as a global health priority and the strategic interest of the Bill & Melinda Gates Foundation. *Clinical infectious diseases*, *54*(suppl_2), S89-S92.

Kermany, D. S., Goldbaum, M., Cai, W., Valentim, C. C., Liang, H., Baxter, S. L., … & Zhang, K. (2018). Identifying medical diagnoses and treatable diseases by image-based deep learning. *cell*, *172*(5), 1122-1131.

Labhane, G., Pansare, R., Maheshwari, S., Tiwari, R., & Shukla, A. (2020, February). Detection of pediatric pneumonia from chest X-ray images using CNN and transfer learning. In *2020 3rd international conference on emerging technologies in computer engineering: machine learning and internet of things (ICETCE)* (pp. 85-92). IEEE.

Lu, S. Y., Wang, S. H., & Zhang, Y. D. (2020). A classification method for brain MRI via MobileNet and feedforward network with random weights. *Pattern Recognition Letters*, *140*, 252-260.

Rudan, I., Boschi-Pinto, C., Biloglav, Z., Mulholland, K., & Campbell, H. (2008). Epidemiology and etiology of childhood pneumonia. *Bulletin of the world health organization*, *86*, 408-416B.

Singh, S., & Tripathi, B. K. (2022). Pneumonia classification using quaternion deep learning. *Multimedia Tools and Applications*, *81*(2), 1743-1764.

Tangudu, V. S. K., Kakarla, J., & Venkateswarlu, I. B. (2022). COVID-19 detection from chest x-ray using MobileNet and residual separable convolution block. *Soft Computing*, *26*(5), 2197-2208.

Ünal, H. T., & Başçiftçi, F. (2022). Evolutionary design of neural network architectures: a review of three decades of research. *Artificial Intelligence Review*, *55*(3), 1723-1802.

Zhang, L., Xu, R., & Zhao, J. (2023). Learning technology for detection and grading of cancer tissue using tumour ultrasound images 1. *Journal of X-Ray Science and Technology*, (Preprint), 1-15.