Img-classification

Narges Yarahmadi Gharaei

2023-06-05

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.packages("torch")
# torch::install_torch()
# 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")
# torch::install_torch()
# install.packages("torchvision")
# install.packages("luz")
# install.packages("luz", repos = "https://cran.rstudio.com/")
# install.packages("torchvision", repos = "https://cran.rstudio.com/")
```

```
# load packages
library(ggplot2)
library(torch)
library(torchvision)
library(luz)
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 6.9.12.3
## 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
```

```
# library(torch)
# torch::install_torch()
```

Diagnosing Pneumonia from Chest X-Rays

Parsing the data

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

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

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

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

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

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

I also deleted the keep folder which was mistakenly inside the train/PNEUMONIA.

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

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

Pneumonia is an infection that inflames the air sacs in one or both lungs. It can be caused by bacteria, viruses, or fungi and leads to symptoms such as cough, fever, shortness of breath, and chest pain. Pneumonia can range from mild to severe, and it can be particularly dangerous for young children, older adults, and people with weakened immune systems.

Automatic identification of pneumonia from X-rays offers several benefits. It enables early detection, leading to prompt treatment and improved outcomes. It increases efficiency and speed by automating the process, reducing the need for manual review. Automated systems can enhance accuracy by detecting subtle patterns that may be missed by humans. It optimizes resources by freeing up radiologists' time for complex cases. It also expands access to healthcare in remote areas. However, medical professionals should still make the final diagnosis, with automated systems serving as supportive tools.

Exploring dataset

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

```
data_folder = "data/lab3_chest_xray"

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

```
##  [1] "data/lab3_chest_xray/test/PNEUMONIA/person114_bacteria_546.jpeg"
##  [2] "data/lab3_chest_xray/test/PNEUMONIA/person128_bacteria_607.jpeg"
##  [3] "data/lab3_chest_xray/test/PNEUMONIA/person136_bacteria_648.jpeg"
##  [4] "data/lab3_chest_xray/train/NORMAL/IM-0225-0001.jpeg"
##  [5] "data/lab3_chest_xray/train/NORMAL/IM-0293-0001.jpeg"
##  [6] "data/lab3_chest_xray/train/NORMAL/IM-0295-0001.jpeg"
##  [7] "data/lab3_chest_xray/train/NORMAL/IM-0428-0001.jpeg"
##  [8] "data/lab3_chest_xray/train/NORMAL/IM-0479-0001.jpeg"
##  [9] "data/lab3_chest_xray/train/NORMAL/IM-0501-0001-0001.jpeg"
## [10] "data/lab3_chest_xray/train/NORMAL/IM-0537-0001.jpeg"
## [11] "data/lab3_chest_xray/train/NORMAL/IM-0544-0001.jpeg"
## [12] "data/lab3_chest_xray/train/NORMAL/IM-0570-0001.jpeg"
## [13] "data/lab3_chest_xray/train/PNEUMONIA/person1016_bacteria_2947.jpeg"
## [14] "data/lab3_chest_xray/train/PNEUMONIA/person1137_virus_1876.jpeg"
## [15] "data/lab3_chest_xray/train/PNEUMONIA/person1147_virus_1917.jpeg"
## [16] "data/lab3_chest_xray/train/PNEUMONIA/person1149_virus_1924.jpeg"
## [17] "data/lab3_chest_xray/train/PNEUMONIA/person1164_virus_1956.jpeg"
## [18] "data/lab3_chest_xray/train/PNEUMONIA/person118_virus_224.jpeg"
## [19] "data/lab3_chest_xray/train/PNEUMONIA/person1213_virus_2058.jpeg"
## [20] "data/lab3_chest_xray/train/PNEUMONIA/person1238_bacteria_3194.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?

It is expected that the dataset folder consists of three subfolders: train, test, and validate. Within each of these subfolders, there are two folders representing the classes or labels in our classification task: “normal” and “pneumonia”. The provided code snippet allows us to navigate through the dataset’s directory structure and gather details such as the count of images and their distribution across the different classes. This information is valuable for comprehending the dataset and organizing the subsequent stages of the image classification task. Looking at the filenames for the X-rays showing pneumonia, we can observe that they include information about the patient and the type of infection (e.g., bacteria or virus). For example, some filenames contain terms like “bacteria” or “virus” along with a unique identifier for the patient.

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 <- "data/lab3_chest_xray"
```

```r
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
```

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

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

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

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

Creating training datasets

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

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

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

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

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

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

```r
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
```

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

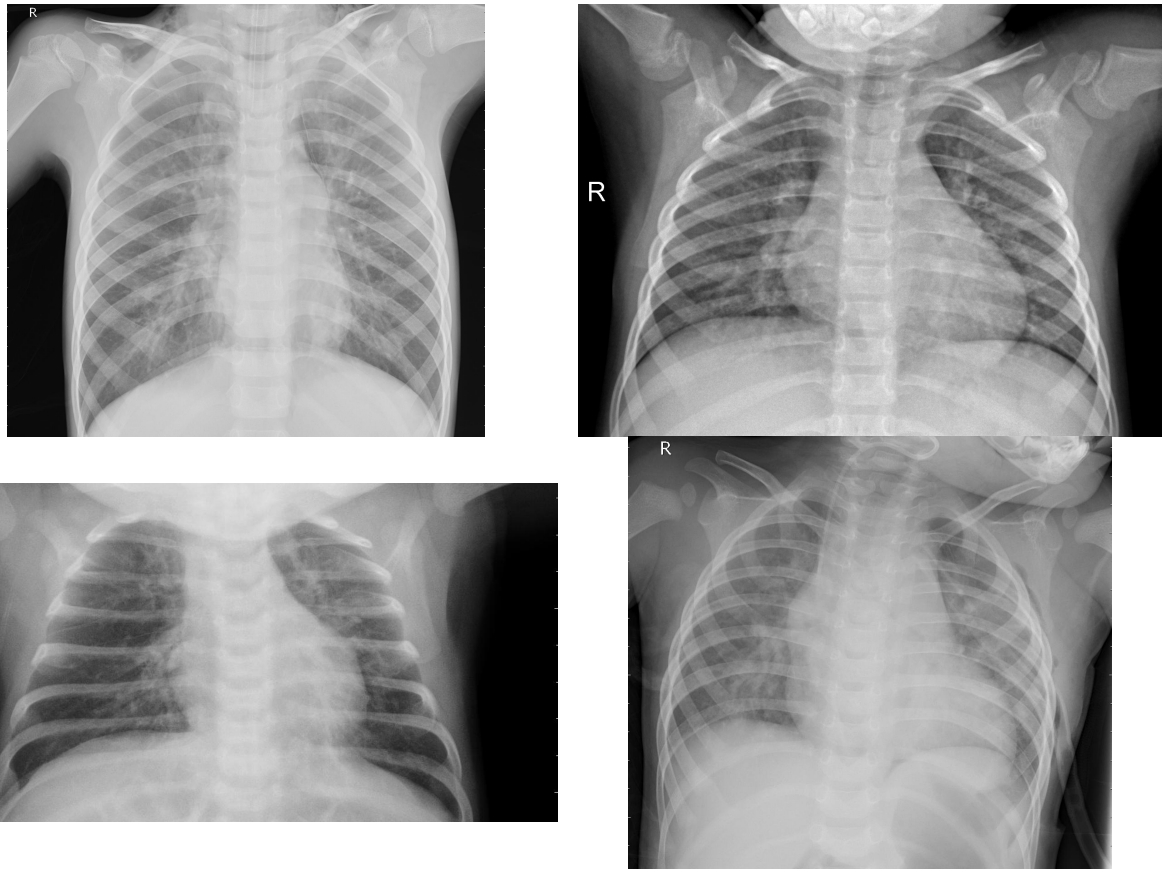
```
## finle name:  data/lab3_chest_xray/train/PNEUMONIA/person105_virus_192.jpeg
## label:  pneumonia
```

Data Visualization

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

```r
# 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
    )
```

```r
  # 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?

yes, i think the images dose not have same size and one other point is, its better to check the values of each cell of each image and bring it into the range(0,1) if it's in range for example (0,255).

Without examining the actual images or having more information about their characteristics, it is challenging to identify specific attributes that may require normalization. However, there are some common attributes in medical image datasets that often benefit from normalization before training a model. Here are a few possibilities:

1. **Image Intensity**: The pixel intensity values in medical images can vary significantly due to differences in acquisition settings or equipment. Normalizing the pixel intensities across images can help to remove such variations and make the data more consistent.
2. **Image Size**: Images in a dataset might have different spatial resolutions or sizes. Rescaling the images to a standardized size can ensure that the model receives consistent input dimensions during training.
3. **Contrast**: Medical images can have varying contrast levels, which can affect the visibility of certain features. Normalizing the contrast by applying techniques such as histogram equalization or adaptive contrast enhancement can enhance the consistency of image features.

4. **Orientation**: In some cases, images may have different orientations or alignments. Ensuring a consistent orientation across all images can help in maintaining uniformity and avoiding biases during training.

5. **Anatomical Variations**: Medical images can exhibit variations in anatomical structures due to patient demographics or imaging protocols. Normalizing the images to a common anatomical reference or accounting for anatomical variations during preprocessing can be beneficial.

It is important to note that the specific normalization techniques will depend on the characteristics of the dataset and the requirements of the model you plan to use. It is recommended to analyze the dataset thoroughly and consult with domain experts to determine the appropriate normalization steps to apply for your particular task.

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

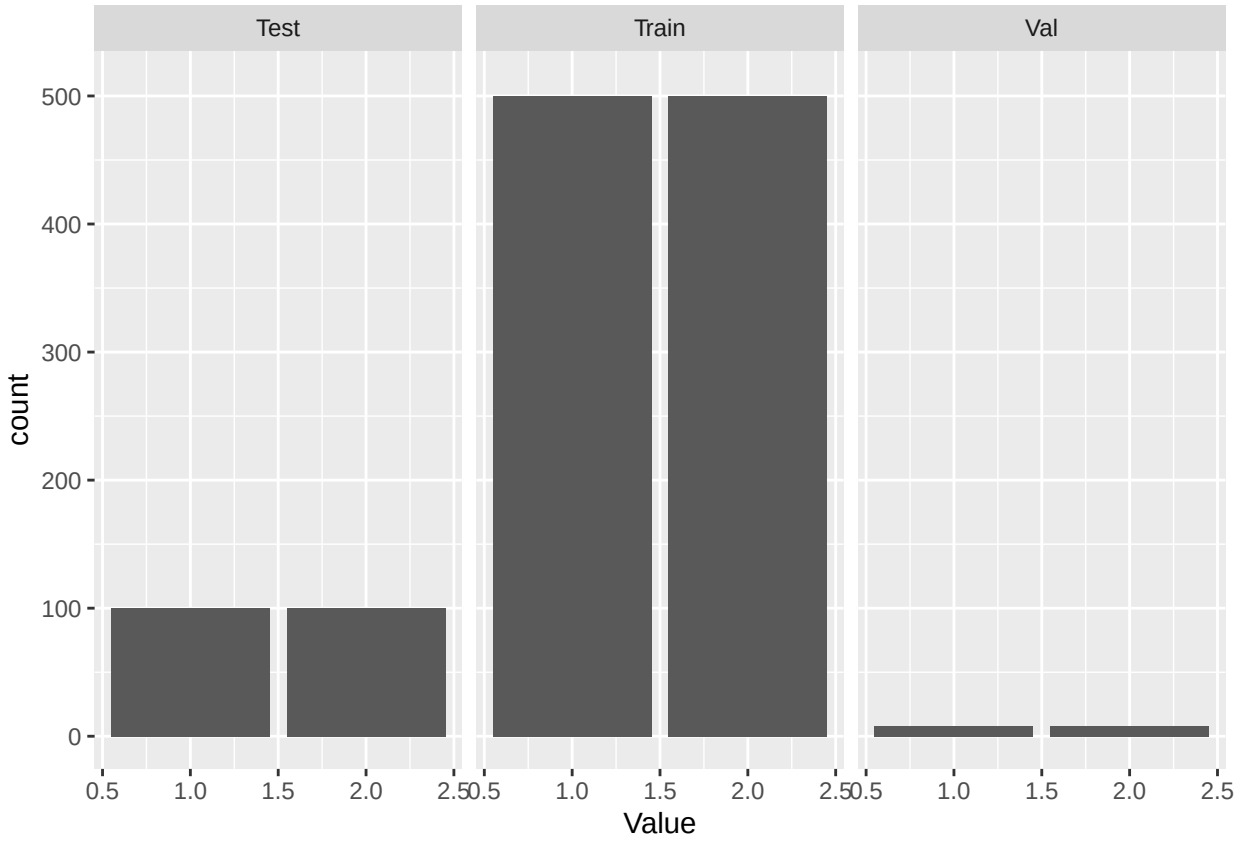
Pneumonia

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 not balanced, meaning that there is a significant difference in the number of samples between different classes, several techniques can be useful to address this issue. Here are some approaches commonly used to handle imbalanced datasets:

1. **Data Resampling**: This technique involves either oversampling the minority class or undersampling the majority class to achieve a more balanced representation of the classes. Oversampling techniques include duplication or generating synthetic samples, while undersampling involves randomly removing samples from the majority class. Both methods aim to equalize the class distribution.

2. **Data Augmentation**: Data augmentation techniques can be applied to increase the number of samples in the minority class by generating new samples through transformations such as rotations, translations, or flips. This helps to diversify the data and create a more balanced dataset.

3. **Class Weighting**: Many machine learning algorithms allow assigning different weights to different classes during training. By assigning higher weights to the minority class, the model gives it more importance, effectively balancing the impact of each class on the training process.

4. **Ensemble Methods**: Ensemble methods, such as bagging or boosting, can be beneficial for imbalanced datasets. These techniques involve combining multiple models to improve performance. By training models on different subsets of the data, ensemble methods can reduce the impact of class imbalance and enhance overall predictive accuracy.

5. **Cost-Sensitive Learning**: Cost-sensitive learning assigns different misclassification costs to different classes. By assigning higher costs to misclassifying the minority class, the model is encouraged to focus more on correctly predicting the minority class instances.

6. **Anomaly Detection**: In some cases, the imbalanced class can be treated as an anomaly or rare event. Anomaly detection techniques can be applied to identify and focus on the minority class instances, treating them as special cases during the modeling process.

It's important to note that the choice of technique depends on the specific problem and dataset characteristics. It's advisable to experiment with different approaches and evaluate their impact on model performance to determine the most effective strategy for handling the imbalanced dataset.

Trainig

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.

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)

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

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

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

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

Train model

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

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

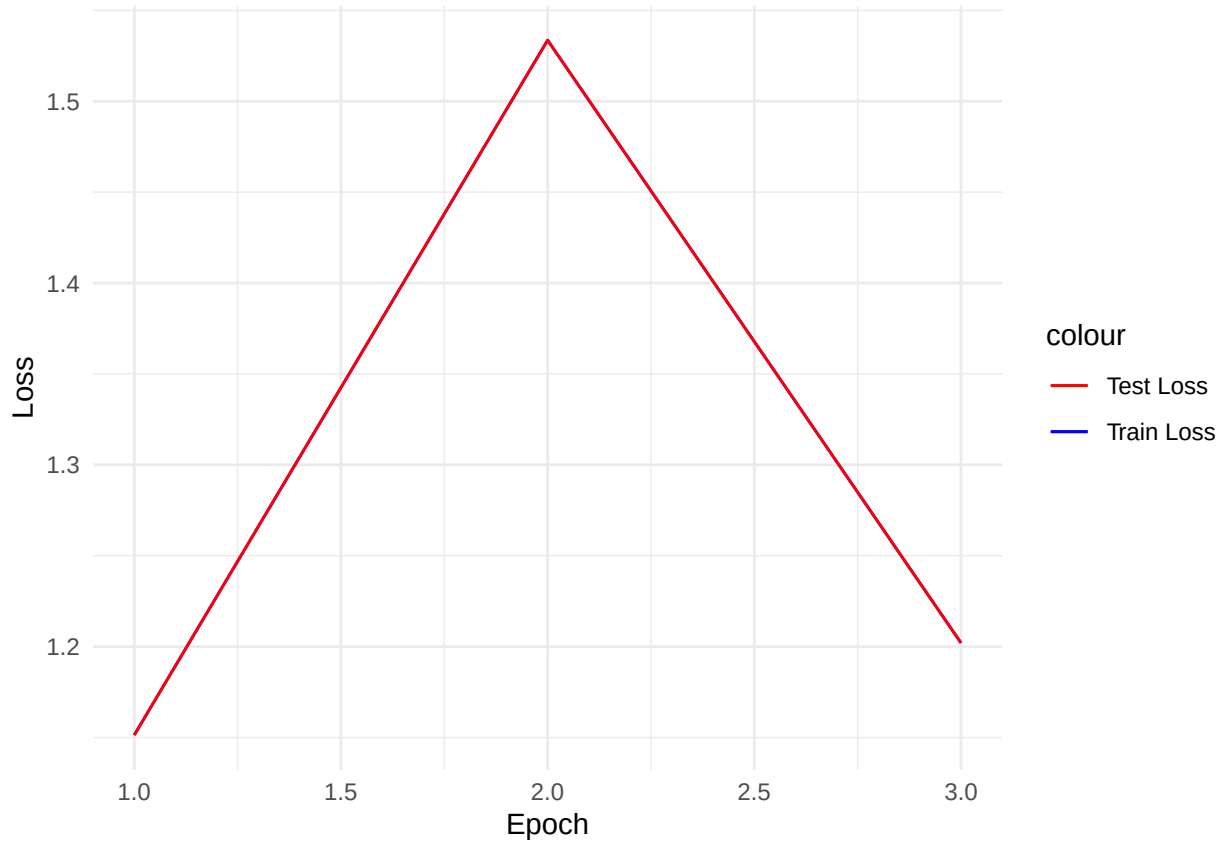
```
# Loop through the epochs
for (epoch in 1:num_epochs) {
  # Perform training and validation for each epoch
  fitted <- net %>%
    setup(
      loss = nn_cross_entropy_loss(),
      optimizer = optim_adam,
      metrics = list(
        luz_metric_accuracy()
      )
    ) %>%
    fit(train_dataloader,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

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

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

```
## Epoch  1 / 3
## Train metrics: Loss:  1.151386  - Acc:  0.515
## Valid metrics: Loss:  0.6931742  - Acc:  0.5
##
## Epoch  2 / 3
## Train metrics: Loss:  1.533564  - Acc:  0.486
## Valid metrics: Loss:  0.693269  - Acc:  0.5
##
## Epoch  3 / 3
## Train metrics: Loss:  1.202049  - Acc:  0.513
## Valid metrics: Loss:  0.6932681  - Acc:  0.5
```

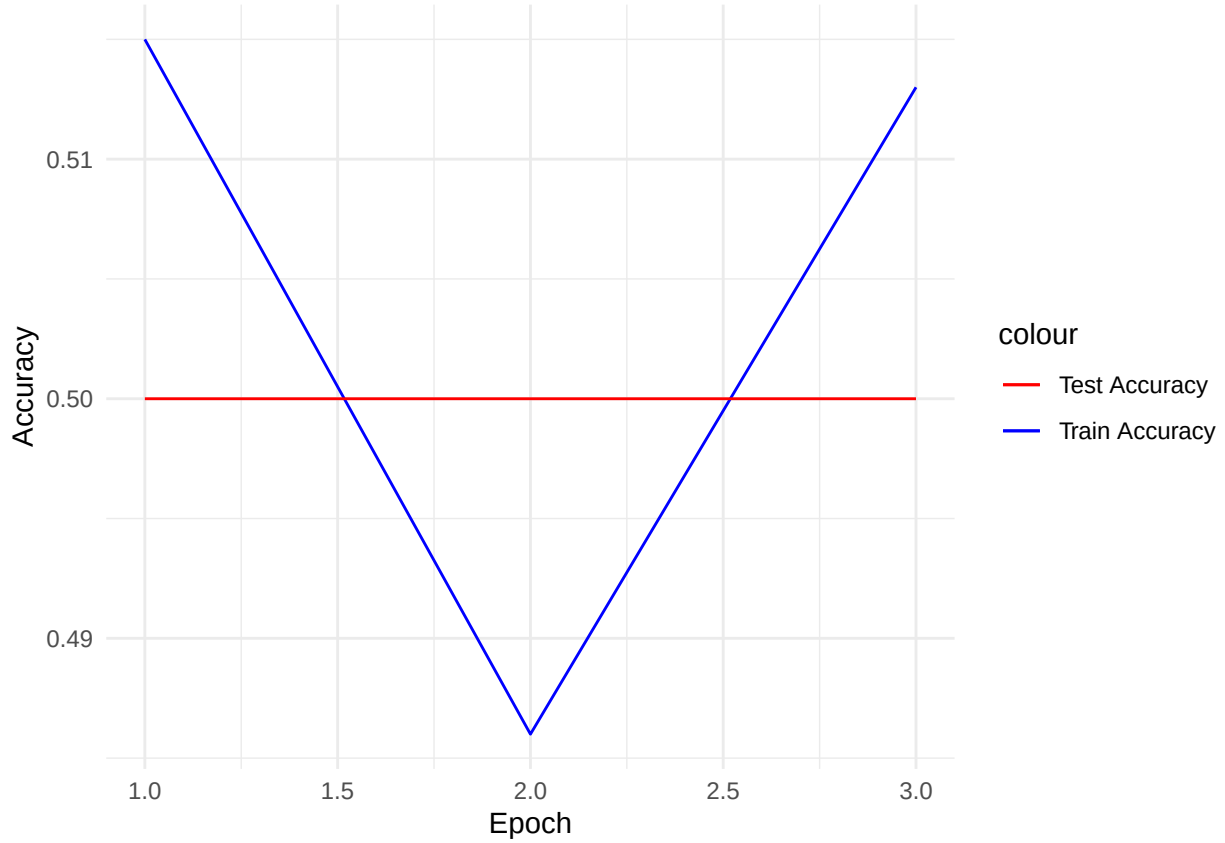
Plot learning curves

```
# 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")) +
```

```r
  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)
```




```r
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 training and test accuracy, it appears that the model is not performing well in classifying X-rays into pneumonia vs normal. The loss plot shows that the model has not significantly reduced the loss, indicating that it needs more training epochs to improve its performance. The accuracy plot also demonstrates a similar trend. The achieved accuracy is comparable to randomly guessing the classes, suggesting that the model is not effectively learning from the data.

In a lighthearted manner, it can be humorously suggested that using a coin toss might be more cost-effective than training this model. The model's accuracy is not significantly better than chance, indicating that it is not capturing the underlying patterns in the data.

Several factors could contribute to these observations. The model may be too complex for the available data, resulting in overfitting or difficulties in generalizing to new examples. Alternatively, the model architecture or hyperparameters may not be suitable for the specific task, requiring adjustments such as changing the training batches or increasing the number of training epochs.

In summary, the model's current performance suggests that it is struggling to effectively learn from the data, and further exploration and modifications may be necessary to improve its classification capabilities.

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. may be this model is too complex for the data or reverse. or the training batches should change or the epoched should be more.

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

I think if we run the model with more epochs and let it explore more would help to increase the accuracy or even if it doesn't help we can try to different architectures for our model, may be a new model with different layers and neurons can act better on this data set. To improve the model's performance, it is suggested

to increase the number of training epochs, allowing the model to explore the data further and potentially increase the accuracy. Additionally, experimenting with different model architectures could be beneficial. Trying out new models with different layers and neurons might provide better results for this dataset. It is worth considering that the current model may be too complex for the available data, and simplifying it or making adjustments to the training process could be necessary.

In addition to the suggestions mentioned, here are some additional recommendations to improve the model:

1. **Data Augmentation**: Applying data augmentation techniques, such as random rotations, flips, or zooms, can artificially increase the diversity of the training data. This can help the model generalize better to unseen examples and improve its performance.
2. **Transfer Learning**: Instead of training a model from scratch, leveraging pre-trained models on a large-scale dataset (e.g., ImageNet) can be beneficial. By utilizing transfer learning, the model can benefit from the learned features and patterns in the pre-trained model, which can help improve its performance on the pneumonia vs normal classification task.
3. **Hyperparameter Tuning**: Experimenting with different hyperparameter settings, such as learning rate, batch size, or optimizer, can help find the optimal configuration for the model. This process can be done using techniques like grid search or random search to systematically explore the hyperparameter space.
4. **Regularization Techniques**: Applying regularization techniques like L1 or L2 regularization, dropout, or batch normalization can help prevent overfitting and improve the model's ability to generalize to unseen data.
5. **Ensemble Learning**: Creating an ensemble of multiple models and combining their predictions can often lead to improved performance. By training several models with different architectures or variations in training data, the ensemble can capture a wider range of patterns and make more accurate predictions.

It is important to iteratively test and evaluate these suggestions to determine which approaches work best for improving the model's performance on the pneumonia vs normal classification task.

Data Augmentation

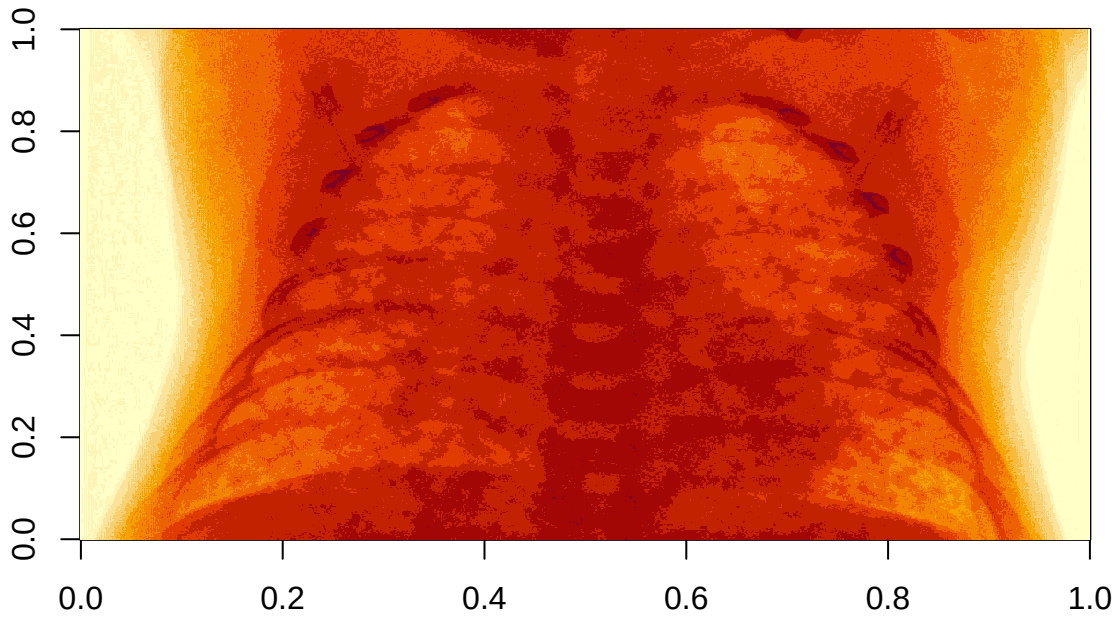
Data augmentation is one of the solutions to consider when facing training difficulties. Data augmentation involves applying various transformations or modifications to the existing training data, effectively expanding the dataset and introducing additional variations. This technique can help improve 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 transofmrations 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
```

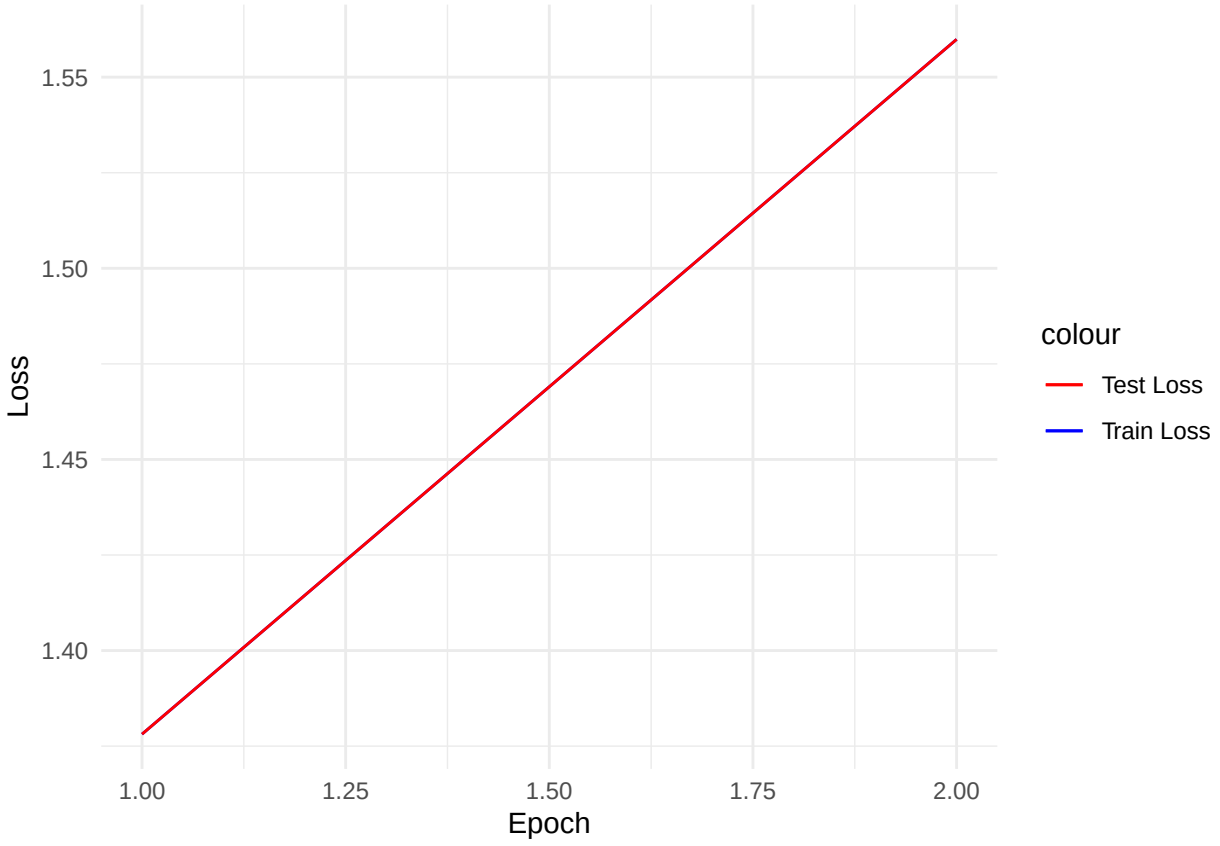
i tried to train the model with the augmented data too as folow:

```
# Set the number of epochs
num_epochs <- 2

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

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

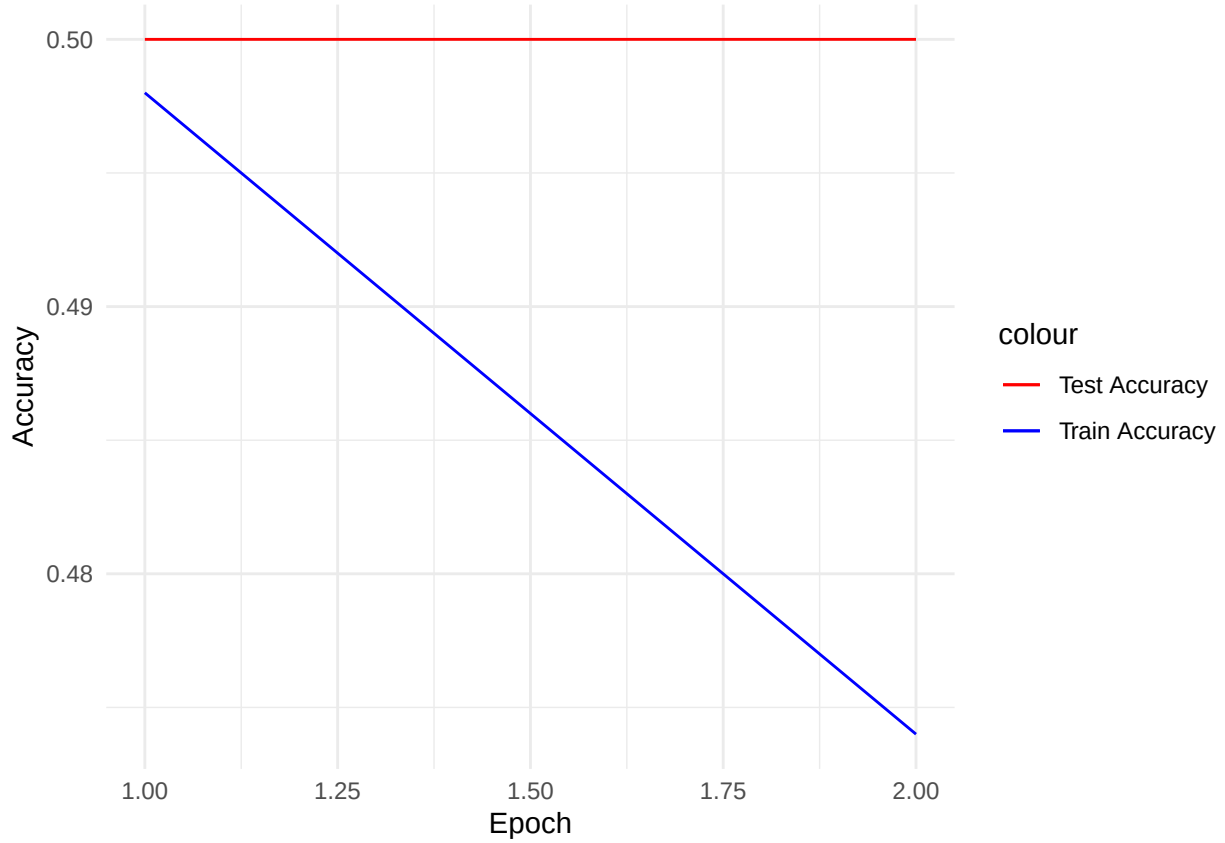
```
  # Print the metrics for the current epoch
  cat("Epoch ", epoch, "/", num_epochs, "\n")
  cat("Train metrics: Loss: ", fitted$records$metrics$train[[1]]$loss, " - Acc: ", fitted$records$metri
  cat("Valid metrics: Loss: ", fitted$records$metrics$valid[[1]]$loss, " - Acc: ", fitted$records$metri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

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

```
## Epoch  1 / 2
## Train metrics: Loss:  1.378133  - Acc:  0.498
## Valid metrics: Loss:  0.6936648  - Acc:  0.5
##
## Epoch  2 / 2
## Train metrics: Loss:  1.559889  - Acc:  0.474
## Valid metrics: Loss:  0.6938958  - Acc:  0.5
```

```
# 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)
```



actually after training just for 2 epoches its hard to judge the model's performance. it must train more to see the result after more epochs based on the trend it takes (for example may be it 20 or 50 epochs) .

6__ What are the potential drawbacks or disadvantages of data augmentation?

practically the using data augmentation has not added lots of improvements to the accuracy. based on that it's not going to make a deep change to the pixels of an image . for example rotating or some thing like it would not make lots of change in the pattern of the image.

While data augmentation offers several benefits, it also has potential drawbacks and disadvantages that should be considered. Some of these drawbacks include:

1.**Overfitting**: Data augmentation can sometimes lead to overfitting, especially if the augmentation techniques are too aggressive or applied excessively. Overfitting occurs when the model becomes too specialized in the augmented data and performs poorly on real-world, unseen data. It is essential to strike a balance between augmentation and the original data to avoid overfitting.

2.**Loss of data fidelity**: Certain data augmentation techniques, such as random cropping, flipping, or rotation, may introduce distortions or alterations that could affect the fidelity of the original data. These transformations can modify the contextual relationships within the data, potentially leading to the model learning incorrect or misleading patterns.

3.**Increased computational requirements**: Data augmentation increases the amount of data that needs to be processed during training. As a result, it requires additional computational resources and time to perform the augmentations on the fly. If the augmentation process is time-consuming, it can slow down the training process, making it less efficient.

4.**Limited diversity in augmented samples**: Data augmentation techniques are often applied to existing data samples, resulting in variations of the original samples. However, this may not capture the full diversity of the underlying data distribution. Augmentation cannot create entirely new, unique data points that

represent different aspects of the problem space. Therefore, the augmented dataset may still lack certain variations that could be important for accurate model training.

5.**Inconsistent augmentation effects**: The impact of data augmentation on model performance can vary depending on the specific problem, dataset, and augmentation techniques used. Some augmentation techniques may work well for one task but not for another. It requires careful experimentation and validation to determine the most effective augmentation strategy for a particular problem, which can be time-consuming and resource-intensive.

6.**Augmentation bias**: If the data augmentation techniques are not carefully chosen or applied, they may introduce biases into the training data. For example, if the augmentation disproportionately amplifies certain characteristics or biases already present in the original data, the model may become biased and make unfair or inaccurate predictions.

It's important to note that while these drawbacks exist, data augmentation remains a valuable technique in machine learning, as it can help improve model generalization, increase dataset size, and enhance robustness to variations in input data. The key lies in understanding the limitations and potential risks associated with data augmentation and using it judiciously in combination with other best practices for model training and evaluation.

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, also known as the "ImageNet Large Scale Visual Recognition Challenge 2012," is a dataset and benchmark in the field of computer vision. It was created by researchers at Stanford University and is widely recognized as a significant milestone in the development of deep learning and image classification algorithms.

The ImageNet dataset consists of over a million labeled images collected from the internet, covering a wide range of objects, scenes, and categories. The dataset was originally organized into 1,000 different classes, with each class representing a distinct object or concept. These classes include various animals, plants, everyday objects, vehicles, and more.

The main purpose of ImageNet and the associated challenge was to advance the state of the art in object recognition algorithms. Participants were tasked with developing models capable of classifying the images into the correct classes. The dataset was notably challenging due to its large scale, high diversity, and the presence of subtle variations and visual complexities.

ImageNet helped propel the development of deep convolutional neural networks (CNNs) by providing a large-scale, labeled dataset for training and evaluation. The size and diversity of the dataset allowed researchers to train deep learning models on a massive scale, leading to breakthroughs in image classification accuracy.

By providing a benchmark dataset and evaluation metric, ImageNet facilitated direct comparisons between different algorithms and models. This standardized evaluation allowed researchers to objectively measure progress and identify the most effective techniques. The ImageNet Challenge became a platform for researchers worldwide to showcase their innovations, leading to rapid advancements in computer vision research.

The success of deep learning models on ImageNet demonstrated their ability to learn hierarchical features and representations from raw image data, surpassing traditional handcrafted feature extraction methods.

This achievement paved the way for deep learning to become the dominant approach in computer vision and sparked advancements in other areas such as object detection, semantic segmentation, and image generation.

Overall, ImageNet's large-scale dataset and challenge played a pivotal role in the advancement of computer vision research. It helped establish deep learning as a powerful technique for image classification tasks and spurred progress in developing more accurate and robust algorithms.

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

Using the MobileNet architecture in a practical assignment can help improve the results for several reasons, as outlined in the MobileNet publication. Here are a few key points:

1.**Efficiency**: MobileNet is specifically designed to be computationally efficient and have a smaller model size compared to traditional deep learning architectures. It achieves this through the use of depth-wise separable convolutions, which significantly reduce the number of parameters and operations required. As a result, MobileNet can run faster on resource-constrained devices, making it suitable for real-time applications or scenarios with limited computational resources.

2.**Speed and latency**: The efficiency of MobileNet translates into faster inference speeds and lower latency, making it advantageous in applications where real-time or near-real-time performance is crucial. By reducing the computational overhead, MobileNet allows for quicker predictions and faster processing of input data.

3.**Accuracy and performance**: Despite its efficiency, MobileNet still achieves competitive accuracy compared to larger and more computationally expensive models. The architecture is carefully designed to strike a balance between model size and accuracy, making it well-suited for tasks where a good trade-off between speed and performance is required. MobileNet has demonstrated strong performance on tasks such as image classification, object detection, and semantic segmentation.

4.**Transfer learning and fine-tuning**: MobileNet has been pre-trained on large-scale datasets such as ImageNet, which helps to capture a general understanding of visual features. Leveraging transfer learning, one can use a pre-trained MobileNet model as a starting point and fine-tune it on a specific task or dataset. This can save significant training time and computational resources while still benefiting from the knowledge learned from the ImageNet dataset.

5.**Mobile and embedded applications**: MobileNet's efficiency and smaller model size make it particularly suitable for deployment on mobile devices and embedded systems. Its lightweight nature allows for on-device inference, reducing the need for frequent data transfers to remote servers and ensuring privacy and faster response times. MobileNet has enabled a wide range of applications on mobile platforms, including image recognition, object detection, and even real-time video processing.

By leveraging the MobileNet architecture in a practical assignment, one can potentially achieve faster inference, reduced latency, improved efficiency, and competitive accuracy, making it a compelling choice for various real-world applications.

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

The number of parameters in a neural network is a key indicator of its complexity and memory requirements. The specific number of parameters in a network depends on various factors such as the network architecture, layer sizes, and connectivity patterns.

Regarding the MobileNet architecture, there are different variations and versions available, each with different complexities. The original MobileNet architecture introduced in the paper "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications" by Howard et al. consists of depth-wise separable convolutions and point-wise convolutions.

The number of parameters in MobileNet can vary based on factors such as the input image size, width multiplier, and resolution multiplier. However, to provide a rough estimate, the original MobileNet architecture with a width multiplier of 1.0 has approximately 4.2 million parameters.

To compare this with better performing networks available in Torch (PyTorch), it's important to note that Torch has a wide range of pre-trained models with varying complexities and performance levels. Some popular high-performance networks available in Torch include ResNet, DenseNet, and EfficientNet.

For example, ResNet variants such as ResNet-50 or ResNet-101 can have tens of millions of parameters, typically ranging from 20 to 40 million parameters. DenseNet models, known for their dense connectivity patterns, can have a similar parameter count as ResNet. EfficientNet, which uses a combination of depth-wise convolutions, squeeze-and-excitation blocks, and compound scaling, can have millions to hundreds of millions of parameters depending on the specific variant (e.g., EfficientNet-B0 to EfficientNet-B7).

Comparing the parameter count of MobileNet (approximately 4.2 million) to these better performing networks available in Torch, it is evident that MobileNet has a significantly smaller number of parameters. This reduced parameter count is one of the reasons why MobileNet is considered efficient, enabling faster inference and making it suitable for resource-constrained environments such as mobile devices and embedded systems.

It's important to note that while MobileNet has fewer parameters compared to some high-performance networks, it achieves a good balance between model size and accuracy, making it well-suited for specific use cases that prioritize computational efficiency and real-time performance.

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
# 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 classificati
```

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

I tried mobilenet but it was really hard to configure it's error and solve it. so i decided to train an other costume architecture.

as i know the knit would run all chunks in order, so for ignoring the output of data augmentation chunk i prefer to rerun the following chunk:

```
# 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
```

Actually i tried to train the data with different model like the modeified model in following.

```
new_net <- nn_module(
  "new_net",

  initialize = function() {
    self$conv1 <- nn_conv2d(1, 32, 3, 1)
    self$conv2 <- nn_conv2d(32, 64, 3, 1)
    self$conv3 <- nn_conv2d(64, 128, 3, 1)
    self$dropout1 <- nn_dropout2d(0.25)
    self$dropout2 <- nn_dropout2d(0.5)
    self$fc1 <- nn_linear(16 * 1520768, 256)
    self$fc2 <- nn_linear(256, 128)
    self$fc3 <- nn_linear(128, 2)
  },

  forward = function(x) {
    x %>%
      self$conv1() %>%
      nnf_relu() %>%
      self$conv2() %>%
      nnf_relu() %>%
      self$conv3() %>%
      nnf_relu() %>%
      nnf_max_pool2d(2) %>%
      self$dropout1() %>%
      torch_flatten(start_dim = 2) %>%
      self$fc1() %>%
      nnf_relu() %>%
      self$dropout2() %>%
```

```
      self$fc2() %>%
      nnf_relu() %>%
      self$fc3()
  }
)
```

But there's a problem when running the above problem, and it takes lots of time to train, and my system crashes while I run it. and it happens because the parameters are more than before, and the model is a bit more complex with respect to the previous one. as the question wants to train with another architecture, I had to use a very simplified one. So to do this, I tried the following model.

```
new_net <- nn_module(
  "new_net",

  initialize = function() {
    self$fc1 <- nn_linear(1 * 224 * 224, 128)  # Adjust the input size based on your image dimensions
    self$fc2 <- nn_linear(128, 2)              # Change the output size to match your classification ta
  },

  forward = function(x) {
    x %>%
      torch_flatten(start_dim = 2) %>%          # N * 1 * 224 * 224 --> N * 50176
      self$fc1() %>%                            # N * 128
      nnf_relu() %>%
      self$fc2()                                # N * 2 (change the output size to match your classifica
  }
)
```

Lets train the very simplified new net above:

```
# 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 <- new_net %>%
    setup(
      loss = nn_cross_entropy_loss(),
      optimizer = optim_adam,
      metrics = list(
        luz_metric_accuracy()
      )
    ) %>%
    fit(train_dataloader, epochs = 1, valid_data = test_dataloader)

  # Print the metrics for the current epoch
  cat("Epoch ", epoch, "/", num_epochs, "\n")
```

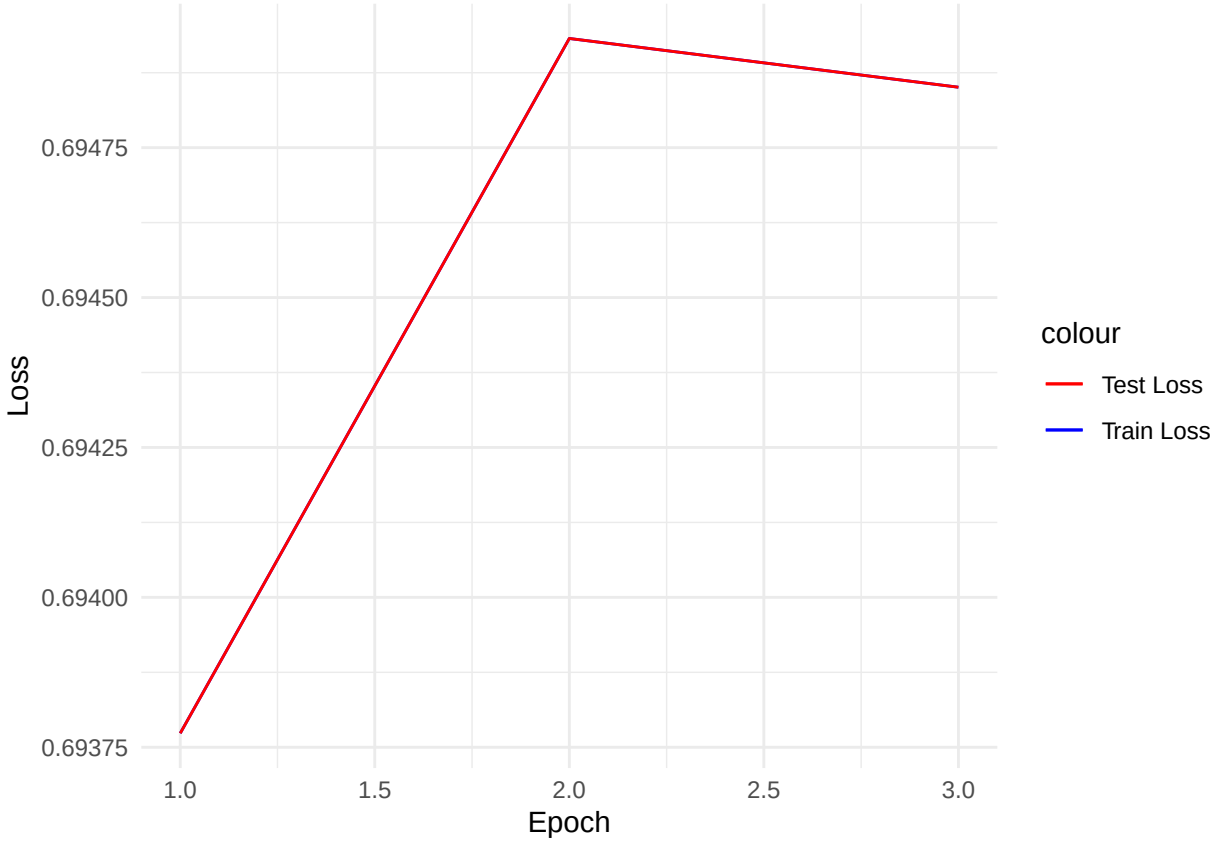
```
  cat("Train metrics: Loss: ", fitted$records$metrics$train[[1]]$loss, " - Acc: ", fitted$records$metri
  cat("Valid metrics: Loss: ", fitted$records$metrics$valid[[1]]$loss, " - Acc: ", fitted$records$metri
  cat("\n")

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

```
## Epoch  1 / 3
## Train metrics: Loss:  0.6937733  - Acc:  0.52
## Valid metrics: Loss:  0.6942505  - Acc:  0.5
##
## Epoch  2 / 3
## Train metrics: Loss:  0.6949321  - Acc:  0.503
## Valid metrics: Loss:  0.6933181  - Acc:  0.5
##
## Epoch  3 / 3
## Train metrics: Loss:  0.6948508  - Acc:  0.494
## Valid metrics: Loss:  0.6931662  - Acc:  0.5
```

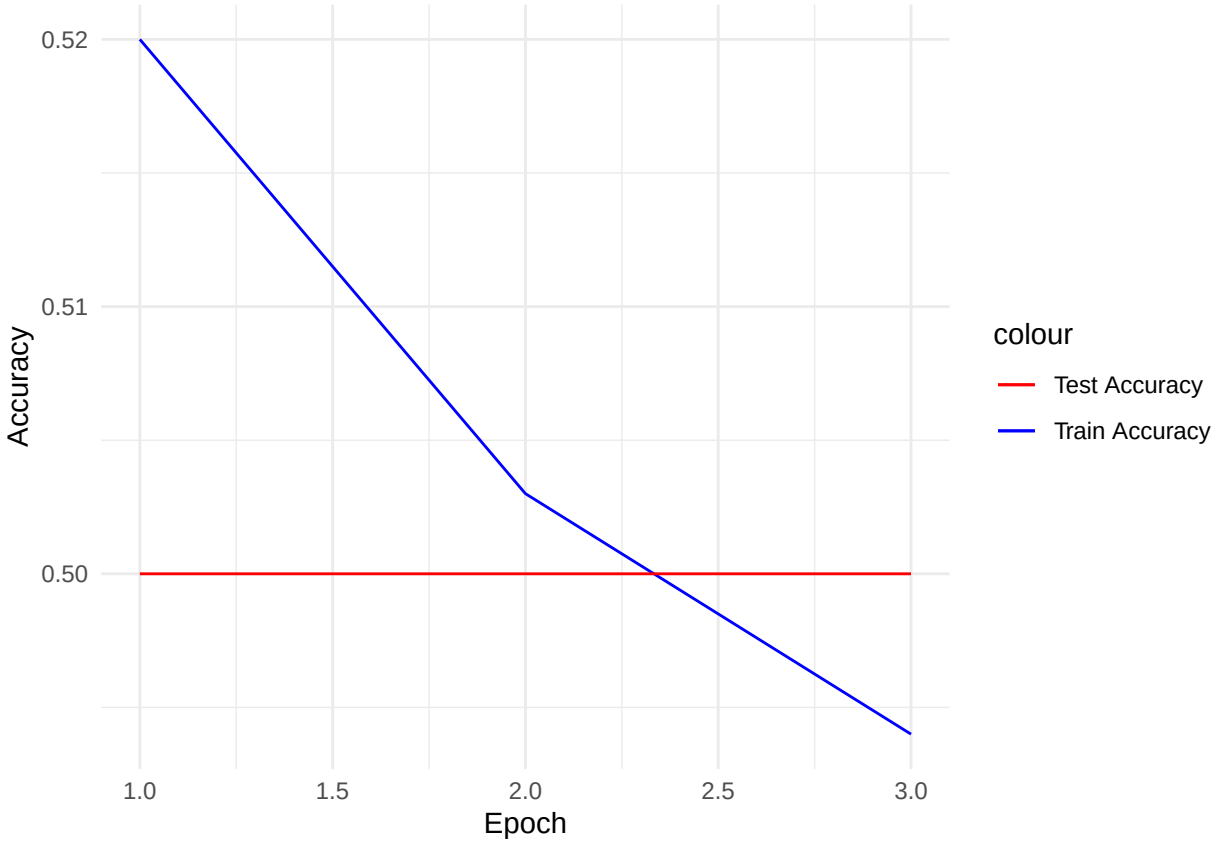
Plot learning curves

```
# 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()
```

```r
# Print the plots
print(loss_plot)
```




```r
print(acc_plot)
```



Or an other model like below, i just added an other linear model in compare to the first model you have provided in top of this file.

```
new_net_two <- nn_module(
  "new_net_two",

  initialize = function() {
    self$conv1 <- nn_conv2d(1, 32, 3, 1)
    self$conv2 <- nn_conv2d(32, 64, 3, 1)
    self$fc1 <- nn_linear(64 * 110 * 110, 256)  # Adjust the input size based on the output size of the
    self$fc2 <- nn_linear(256, 128)
    self$fc3 <- nn_linear(128, 2)                # Change the output size to match your classification
  },

  forward = function(x) {
    x %>%
      self$conv1() %>%
      nnf_relu() %>%
      self$conv2() %>%
      nnf_relu() %>%
      nnf_max_pool2d(2) %>%
      torch_flatten(start_dim = 2) %>%
      self$fc1() %>%
      nnf_relu() %>%
      self$fc2() %>%
      nnf_relu() %>%
```

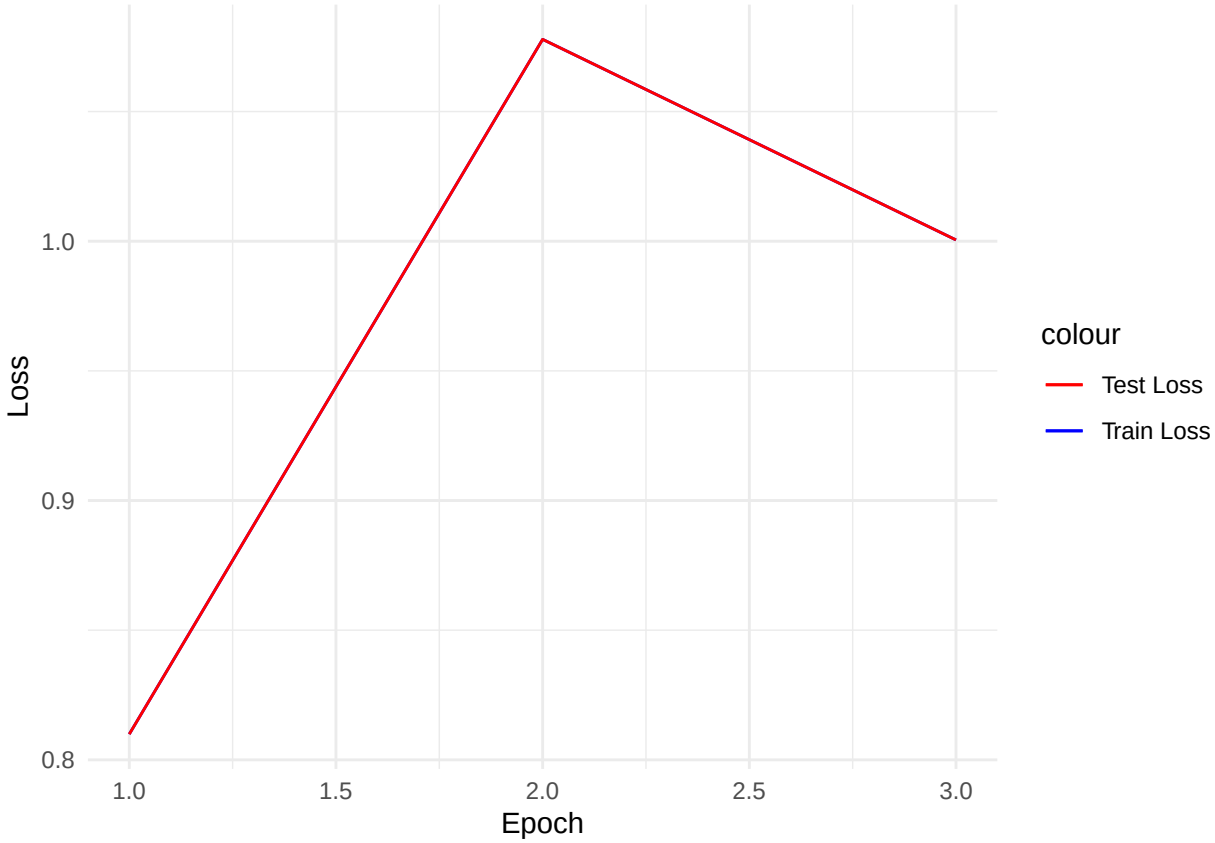
```
      self$fc3()                                    # N * 2 (change the output size to match your classifi
  }
)
```

```
# 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 <- new_net_two %>%
    setup(
      loss = nn_cross_entropy_loss(),
      optimizer = optim_adam,
      metrics = list(
        luz_metric_accuracy()
      )
    ) %>%
    fit(train_dataloader, epochs = 1, valid_data = test_dataloader)

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

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

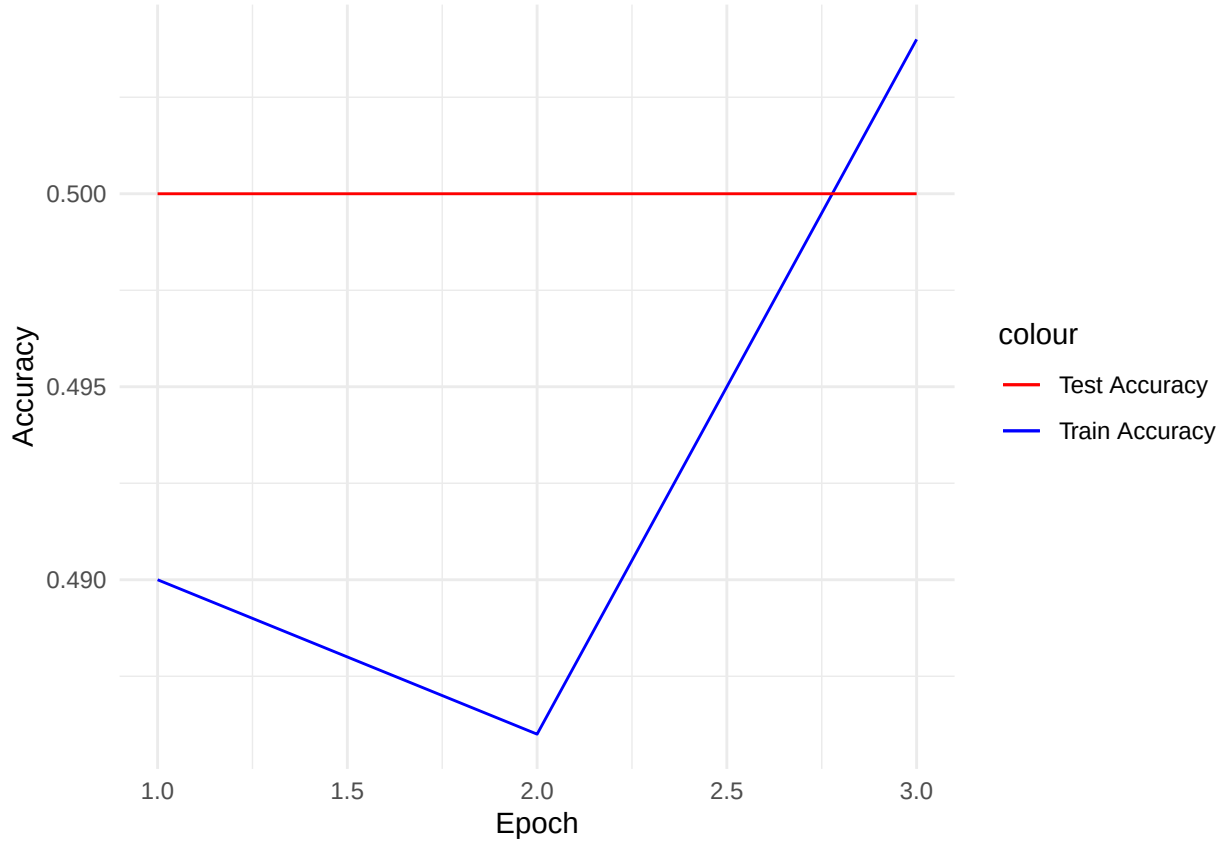
```
## Epoch  1 / 3
## Train metrics: Loss:  0.8099071  - Acc:  0.49
## Valid metrics: Loss:  0.6931592  - Acc:  0.5
##
## Epoch  2 / 3
## Train metrics: Loss:  1.077843  - Acc:  0.486
## Valid metrics: Loss:  0.6933942  - Acc:  0.5
##
## Epoch  3 / 3
## Train metrics: Loss:  1.000499  - Acc:  0.504
## Valid metrics: Loss:  0.6938391  - 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)
```



To answer the performance question, the performance has not increased by these models that I have trained. Still, if you train the model with a more complex architecture, like using transfer learning models such mobilenet or resnet or . . . , on a better system with GPU, then you may see better performances.

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