Chest X-Ray Image Classification: Project Report

Zach Wimpee

February 14, 2021

Introduction and Overview

1 Motivation, Problem, and Client

The COVID-19 pandemic has presented seemingly all areas of society with a host of new and unprecedented challenges, many of which are faced in the healthcare field.

For example, while many people living in developed countries are fortunate enough to have relatively easy access to COVID-19 testing, those living in developing or impoverished areas may not be afforded this same luxury. The virus, however, does not discriminate, and therefore additional tools are needed to aid in diagnosing infected patients when available tests are limited or perhaps even nonexistent.

2 Proposed Solution and Approach

Consider the following hypothetical scenario: Dr. Smith runs a clinic for the impoverished in a developing country, and has a new patient exhibiting a variety of symptoms including chest pain, coughing, and fever. Dr. Smith suspects that the patient has pneumonia, but is unsure if it is a viral or a bacterial infection. He would like to rule out the possibility of a COVID-19 infection, but the clinic is still waiting to be resupplied with a shipment of tests. The patient's condition is declining, and the fastest diagnostic tool available is a chest X-ray.

In this hypothetical scenario, Dr. Smith and his patient would benefit from a tool that could determine from an X-ray whether or not the pneumonia is being caused by a COVID-19 infection. The project being proposed here is an exploration into the potential use of neural network machine learning models in developing such a tool.

2.1 Finding a Dataset

A brief search identified a promising dataset to use for this project's development. It is a curated set of chest X-rays for which each image is labeled one of either COVID-19, normal, viral-pneumonia, or bacterial-pneumonia.[1] The dataset has already been checked for duplicates samples and defective images, and therefore minimal cleaning and preprocessing will be required.

[1] https://data.mendeley.com/datasets/9xkhgts2s6/2

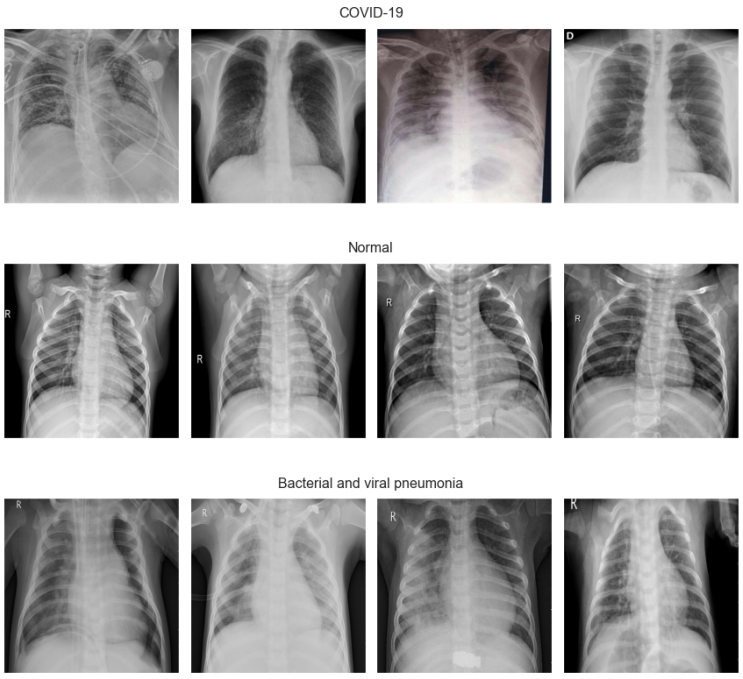


Figure 1: Sample chest X-rays from curated dataset

2.2 Outlining the Deliverables

The final goals for this project are twofold: The first is to design and train a neural network model to classify chest X-rays, and achieve an acceptable level of performance upon test set evaluation. After acquiring a decent model, the final goal will be to implement visualizations that give insight into what the model deems important in its decisions so as to provide model interpretability.

Data Exploration and Preprocessing

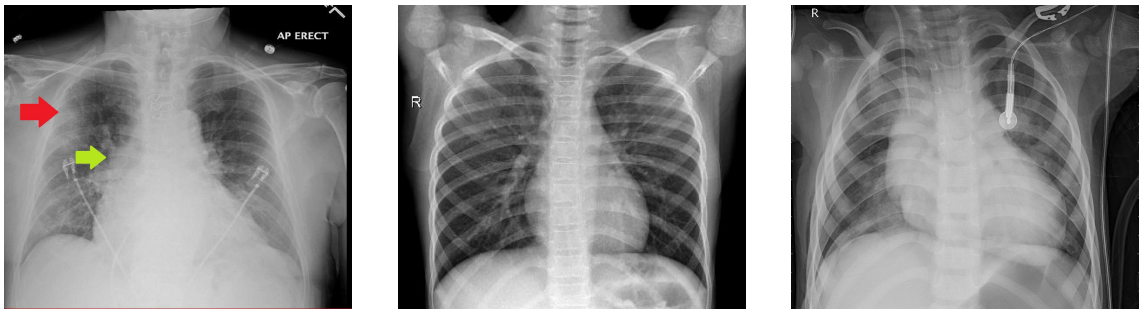
3 Examining the Dataset

The original dataset contains a total of **9208 chest X-ray images...**

- 1281 COVID-19
- 3270 Normal
- 3001 Bacterial Pneumonia
- 1656 Viral Pneumonia

For the purpose of minimizing complexity the viral and bacterial pneumonia samples will be consolidated under the single class label of pneumonia. This reduces the goal of the project to obtaining a model to make predictions between 3 classes instead of 4. While this simplification is less realistic than keeping the distinction between pneumonia cases, it will facilitate the development of a baseline model with maintained focus on distinguishing between the COVID-19 and pneumonia samples.

3.1 Issues and Concerns



(a) COVID-19 (b) Normal (c) Pneumonia

Figure 2: Sample X-ray for each class label

A brief examination of the data reveals several causes for concern that should be noted. **Many of the COVID-19 and pneumonia samples have annotations and text highlighting points of interest, as well as what appears to be pieces of equipment that are not seen in the normal X-rays.** These artifacts pose the threat of artificially inflating the performance metrics for a model that has been trained on these images. This concern will be addressed in two places:

- First during data loading procedures...
  - Data augmentation is applied to training set. See section 5.2
- Then during model testing and evaluation...
  - Assess attribution of input to model predictions using Captum. See section 9

3.2 Generalizing the Preprocessing Files

It should be noted that some of the tools created during the preprocessing steps have potential value for future use.

```
import argparse
from pathlib import Path
from kaggle.api.kaggle_api_extended import KaggleApi
from sklearn.model_selection import train_test_split

def rmtree(root):
    if not root.is_dir():
        print('Folder or directory does not exist.')
        return
    for p in root.iterdir():
        if p.is_dir():
            rmtree(p)
        else:
            p.unlink()
    root.rmdir()

def download_dataset(args):
    api = KaggleApi()
    api.authenticate()
    api.dataset_download_cli(args.dataset, force=args.kforce,
      unzip=args.unzip)

def main(args):
    if args.download:
        if Path(args.src_dir).is_dir():
            rmtree(Path(args.src_dir))
        download_dataset(args)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Dataset creation for Chest X-Ray image
     classification project.")
    parser.add_argument(
        "-u","--dataset", type=str,
        default="unaissait/curated-chest-xray-image-dataset-
     for-covid19",help="dataset url suffix")
    parser.add_argument(
        "--src-dir", type=str,
        default="Curated X-Ray Dataset/", help="path to
     source dataset to be split")
    parser.add_argument(
        "-d","--download", action="store_true",
        help="download the original dataset")
    parser.add_argument(
        "-f","--kforce", type=bool,
        default=False,help="choice to download if original
     dataset already exists")
    parser.add_argument(
        "-z","--unzip", type=bool,
        default=True,help="choice unzip downloaded dataset")
    args = parser.parse_args()
    main(args)
```

To the left is an excerpt of the data wrangling code used to acquire the original dataset that has been simplified in order to illustrate the potential generalization to future use cases.

- *rmtree* function provides an easy way to handle nested directories, which are commonly encountered when downloading datasets.
- *download_dataset* function allows for any Kaggle dataset to be downloaded with control over a variety of options.
- The use of command-line argument parsing makes it possible to use these functions for similar future applications.

4 Splitting Data and Addressing Class Imbalance

The data is now split into disjoint training, validation, and testing sets. 60% will be used to construct the training set, while the validation and testing sets will evenly split the remaining 40% of the data. Although the full dataset has a high level of class imbalance, each class is represented by a relatively large number of samples. Therefore the issue of class imbalance is handled as follows:

1. Get train/validation/test sets such that the distribution of class labels is approximately the same for each split, as shown below.

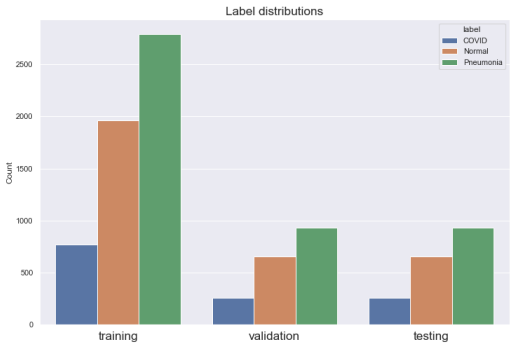


Figure 3: Class label distributions after splitting

2. Compute the accuracy during training steps by taking the class frequencies into account

   - Consider a single training epoch in which label predictions are made for N samples. Then the raw epoch accuracy is given by

$$\textit{Raw Epoch Accuracy} = \frac{1}{N}\sum_{i=1}^{N} \delta_{x_i y_i}$$

     Where,

     - $x_i \equiv$ Predicted label for sample i
     - $y_i \equiv$ True label for sample i
     - $\delta_{x_i y_i} = \begin{cases} 0 & x_i \neq y_i \\ 1 & x_i = y_i \end{cases}$

   - Here we will modify the calculation of epoch accuracy by weighting each prediction by the inverse frequency of the true class labels.

$$\textit{Balanced Epoch Accuracy} = \frac{1}{C}\sum_{j=1}^{C}\sum_{i=1}^{N_j} \frac{\delta_{x_{ji} y_{ji}}}{N_j}$$

     Where,

     - $C \equiv$ Number of classes
     - $N_j \equiv$ Sample counts for class label j, where $j = 1, ..., C$
     - $x_{ji}, y_{ji} \equiv$ Predicted and true labels, respectively, for sample index ji
     - $\delta_{x_{ji} y_{ji}} = \begin{cases} 0 & x_{ji} \neq y_{ji} \\ 1 & x_{ji} = y_{ji} \end{cases}$

Note that this is equivalent to taking the average of the recalls for each class. The code used to accomplish this can be found in Section 7.

3. Use metrics that can encapsulate the imbalance into the scores during model evaluation on test data

    - In addition to both raw and balanced accuracy scores, test data predictions will be evaluated using a variety of performance metrics and techniques:
        - Confusion Matrix
        - Precision-Recall Curves
        - ROC Curves
        - Precision, Recall, and F1-Scores

5 Data Loading

5.1 Transformations

5.2 Augmentation

5.3 Additional Details and Considerations

Model Building

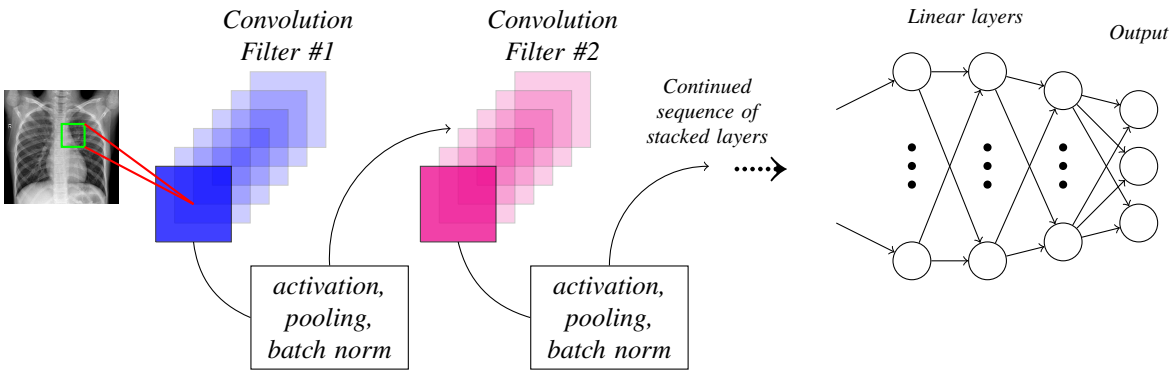
6 Designing Network Architecture

The aim of this project is to not only obtain an accurate image classification model, but to also design the network architecture on which the model will be trained. The following sections outline the intuition behind the most important design choices and provide explicit examples of the code used to implement them.

6.1 Simple Convolutional Network

Begin by considering a simple convolutional neural network architecture where an input image first passes through a sequence of convolution filters, each separated by a combination of non-linear activation functions, pooling filters, and batch normalizations. The output of these stacked layers is then flattened and passed through a few fully connected linear layers (which are separated by non-linearities and batch normalization), and finally a $C-$dimensional output is returned, where C is the number of class labels. A simplified visual representation of such a network is given below, along with the actual code used to construct it.

Figure 4: Simplified CNN architecture



The code to construct such a network is relatively simple:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class LayerStack(nn.Module):
    def __init__(self, top ,layers = None,*args, **kwargs):
        super(LayerStack, self).__init__()

        self.top = top(*args, **kwargs)

        self.layers = layers

        if type(layers) is list:
            self.layers = nn.Sequential(*self.layers)

    def forward(self, x):
        x = self.top(x)
        if self.layers is not None:
            x = self.layers(x)
        return x
```

First we make the neccesarry imports and subclass PyTorch's *nn.Module* to create a new class *LayerStack*:

- First parameter *top* accepts as an argument some PyTorch *nn* layer.
- Optionally can specify list of additional layers to append to the end of the stack using the *layers* parameter.

In the context of our CNN described by Figure 4, 2D convolutions would be passed to the *top* parameter, and *layers* would be get passed an activation function, pooling filter, and batch normalization as an ordered list.

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()

        l1 = [nn.ReLU(),
              nn.MaxPool2d(kernel_size=2,stride=2),
              nn.BatchNorm2d(16)]
        self.conv1 = LayerStack(
            nn.Conv2d,layers=l1,in_channels=3,
            out_channels=16,kernel_size=3,padding=1)

        l2 = [nn.ReLU(),
              nn.MaxPool2d(kernel_size=2,stride=2),
              nn.BatchNorm2d(64)]
        self.conv2 = LayerStack(
            nn.Conv2d,layers=l2,in_channels=16,
            out_channels=64,kernel_size=3,padding=1)

        l3 = [nn.ReLU(),
              nn.MaxPool2d(kernel_size=2,stride=2),
              nn.BatchNorm2d(128)]
        self.conv3 = LayerStack(
            nn.Conv2d,layers=l3,in_channels=64,
            out_channels=128,kernel_size=3,padding=1)

        l4 = [nn.ReLU(),
              nn.MaxPool2d(kernel_size=2,stride=2),
              nn.BatchNorm2d(256)]
        self.conv4 = LayerStack(
            nn.Conv2d,layers=l4,in_channels=128,
            out_channels=256,kernel_size=3,padding=1)

        l5 = [nn.ReLU(),
              nn.MaxPool2d(kernel_size=2,stride=2),
              nn.BatchNorm2d(512)]
        self.conv5 = LayerStack(
            nn.Conv2d,layers=l5,in_channels=256,
            out_channels=512,kernel_size=3,padding=1)

        self.fc1 = LayerStack(
            nn.Linear,layers=[nn.ReLU(),nn.BatchNorm1d(64)],
            in_features=512*7*7,out_features=64)
        self.classifier = LayerStack(
            nn.Linear,in_features=64,out_features=3)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = x.view(-1, 512 * 7 * 7)
        x = self.fc1(x)
        x = self.classifier(x)
        return x
```

6.1.1 Drawbacks and Limitations

Constructing *ConvNet* from instances of *LayerStack* allows for flexibility in expanding both the network's depth and its width. The width can be increased by simply adding additional output channels to each convolution layer, and the depth can be increased by adding more convolution layers. This raises an important question: ***How deep should the network be made?*** Notice that in our case this is equivalent to asking how many instances of *LayerStack* to use.

6.2 Defining CXResnet

Model Training

7 Overview of Training Function

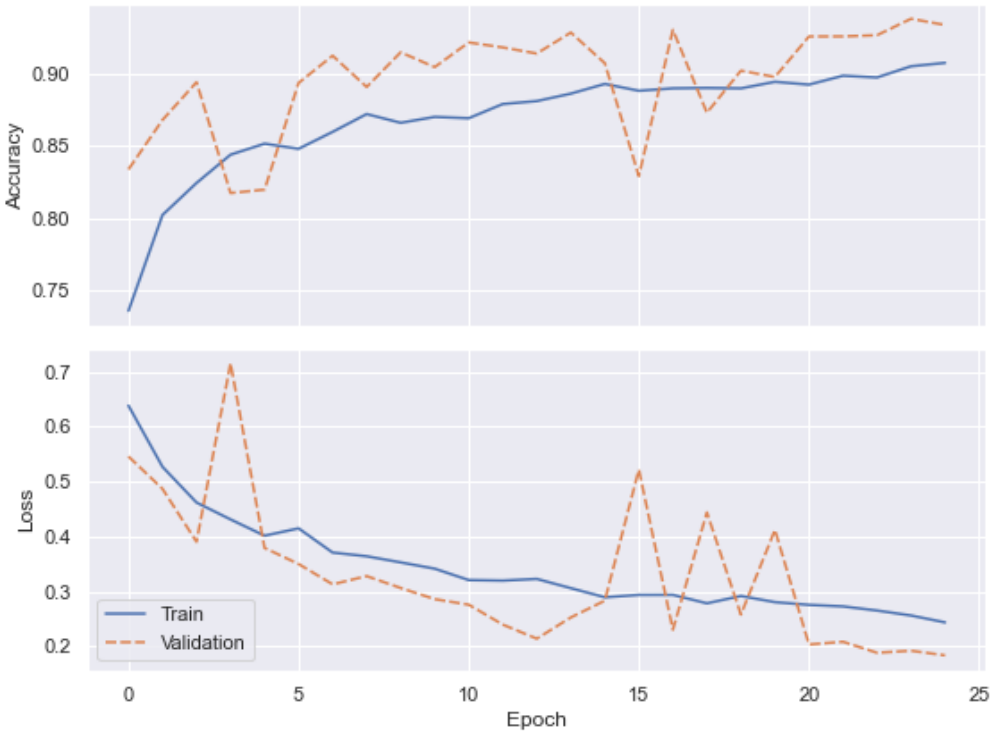


Figure 5: Accuracy and Loss by Epoch

Model Evaluation

8 Testing on Holdout Data

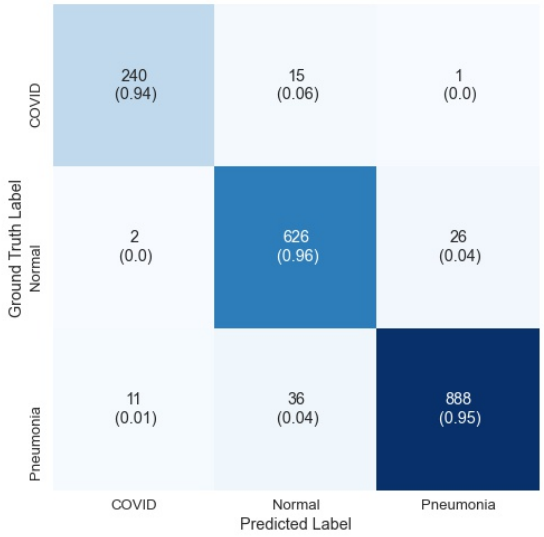


Figure 6: Confusion Matrix

9 Insights and Interpretability

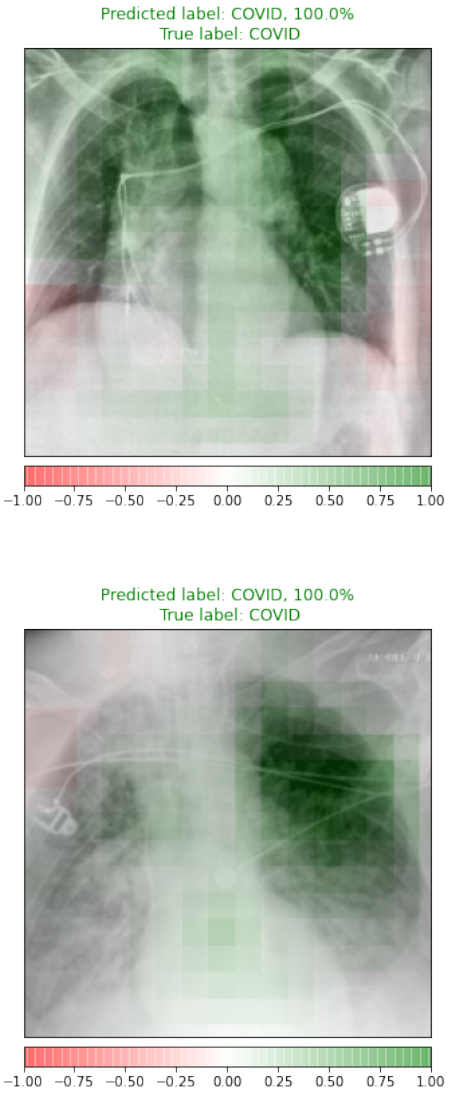


Figure 7: Occlusion attribution heat maps show possible evidence of the model predictions being influenced by equipment present in the image

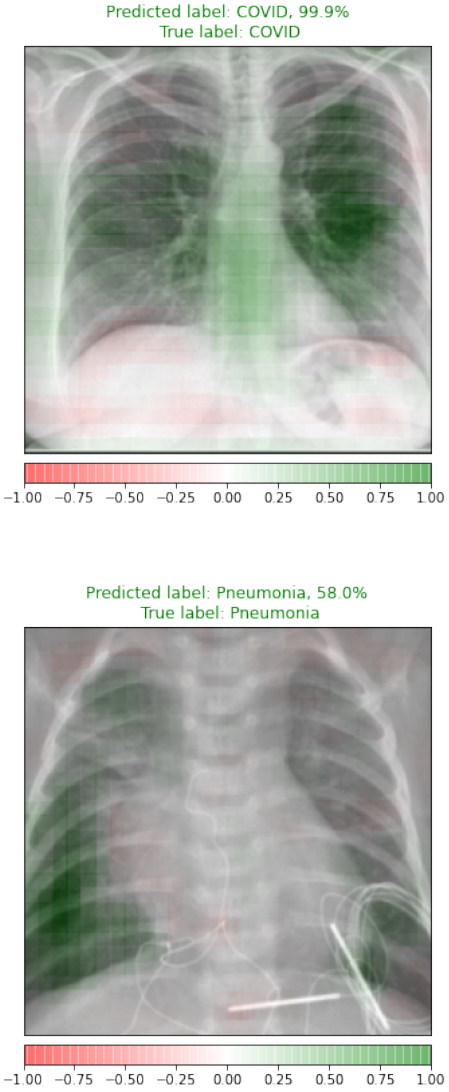


Figure 8: Occlusion attribution heat maps also show evidence that the model could be picking up on visual markers relevant to the correct diagnosis, namely the ability to identify regions exhibiting ground-glass opacification

Results

10 Discussion

11 Closing Remarks

References