Pneumonia Detection Using Deep Convolutional Neural Networks

By:

Poorya MohammadiNasab

M.Sc. Student in Artificial Intelligence

Iran University of Science and Technology (IUST)

Tehran, Tehran, Iran

pooryamohammadinasab@gmail.com

Spring 2023

Contents

Abstract

Background: This study presents a deep learning project conducted for Pneumonia Detection using the PyTorch framework. The project focuses on leveraging the ResNet18 architecture and the RSNA Pneumonia Detection challenge dataset to achieve accurate disease detection. The primary objective of this research is to investigate the effectiveness of the applied deep learning techniques in detecting pneumonia cases. **Method:** The ResNet18 architecture was chosen due to its proven performance in various computer vision tasks. The model was trained and evaluated on the RSNA Pneumonia Detection dataset, which is widely recognized for its complexity and diversity. The dataset consists of a large number of annotated chest X-ray images, providing valuable training samples for deep learning models. To assess the model's performance and gain insights into its inner workings, artificial intelligence interpretability techniques were employed. By applying these techniques, the researchers were able to explore how the model made decisions and identify the features it considered important for pneumonia detection. This approach enhanced the interpretability and transparency of the deep learning model, addressing the black-box issue often associated with deep learning algorithms. **Results:** The results of this project demonstrated promising performance in accurately detecting pneumonia cases. The deep learning model achieved notable accuracy rates, showcasing its potential as a reliable tool for pneumonia detection. The interpretability techniques further provided valuable insights into the decision-making process of the model, assisting medical professionals in understanding and trusting the model's predictions. The full implementation (based on PyTorch) and the trained networks are available at github.com/Pooryamn/Pneumonia-Detection.

Keywords: Pneumonia Detection, Medical Image Analysis, Deep Convolutional Neural Networks (DCNN), Explainable artificial intelligence (XAI)

1. Introduction

Pneumonia is a prevalent and potentially life-threatening respiratory infection that affects millions of individuals worldwide [1]. It is characterized by the inflammation and infection of the lung tissue, primarily caused by bacteria, viruses, or fungi [2]. Pneumonia can affect individuals of all ages, although the highest incidence rates are observed among young children, older adults, and individuals with weakened immune systems [3]. The clinical manifestations of pneumonia vary depending on the causative agent and the severity of the infection. Common symptoms include cough, fever, chest pain, shortness of breath, fatigue, and, in severe cases, respiratory distress [2]. Prompt diagnosis and treatment are crucial in managing pneumonia to prevent complications, such as lung abscesses, respiratory failure, and even death [4].

Traditionally, pneumonia diagnosis relies on a combination of clinical assessment, physical examination, and imaging techniques, most commonly chest X-rays [5]. These diagnostic methods, while effective, have inherent limitations, such as subjectivity in interpretation, variability in expertise, and time-consuming manual analysis [6]. This has led to the exploration of utilizing artificial intelligence (AI) and machine learning algorithms to enhance pneumonia detection and improve diagnostic accuracy [7]. Recent advancements in deep learning techniques have shown promising results in medical image analysis, including the detection and classification of pneumonia from chest X-rays [8][9]. Deep learning models, trained on large annotated datasets, can learn complex patterns and features, enabling automated detection and classification of pneumonia with high accuracy [8].

One notable dataset used for pneumonia detection is the RSNA Pneumonia Detection Dataset, provided by the Radiological Society of North America (RSNA) [10]. This dataset comprises a large number of chest X-ray images, annotated by expert radiologists, ensuring high-quality data for model training and evaluation [10]. The RSNA Pneumonia Detection Dataset poses significant challenges due to the subtle radiographic features associated with pneumonia and the presence of various confounding factors, such as overlapping structures and other lung abnormalities [11]. Numerous studies have explored the application of deep learning models for pneumonia detection and achieved promising results [8][9]. For instance, research conducted by Rajpurkar et al. (2017) developed a deep convolutional neural network (CNN) model that outperformed radiologists in detecting pneumonia from chest X-rays [8]. Similarly, studies by Yao et al. (2019) and Zhu et al. (2020) demonstrated the efficacy of deep learning models in accurately identifying pneumonia cases [9][12].

In light of the potential benefits offered by AI-based approaches in pneumonia diagnosis, this study aims to investigate the effectiveness of deep learning techniques, specifically utilizing the ResNet18 architecture, in detecting pneumonia from chest X-rays [13]. The primary objective is to evaluate the performance of the deep learning model trained on the RSNA Pneumonia Detection Dataset and assess its accuracy in pneumonia detection. Furthermore, to enhance the interpretability and transparency of the deep learning model, artificial intelligence interpretability techniques will be employed. These techniques enable researchers to examine the inner workings of the model, identify the features driving its decision-making process, and contribute to a better understanding of the underlying mechanisms employed by the model in pneumonia detection [14].

By improving the accuracy and efficiency of pneumonia diagnosis, AI-based approaches have the potential to assist healthcare professionals in making timely and informed treatment decisions. They can also aid in reducing diagnostic errors and optimizing healthcare resource utilization. However, it is important to ensure the robustness, reliability, and interpretability of these AI models before integrating them into clinical practice [15]. In conclusion, this study aims to

contribute to the growing body of literature on AI-enabled pneumonia detection. By investigating the performance of a deep learning model utilizing the ResNet18 architecture on the challenging RSNA Pneumonia Detection Dataset, along with the application of interpretability techniques, this research seeks to enhance the accuracy and transparency of pneumonia diagnosis. The insights gained from this study may have significant implications for improving healthcare outcomes and optimizing pneumonia management strategies.

2. Method

2.1 Dataset

The RSNA Pneumonia Detection Dataset [10], provided by the Radiological Society of North America (RSNA), is a comprehensive dataset widely used in the development and evaluation of deep learning models for pneumonia detection from chest X-rays. This dataset plays a crucial role in advancing the field of medical image analysis and improving pneumonia diagnosis. The RSNA Pneumonia Detection Dataset consists of a large collection of chest X-ray images, accompanied by expert annotations, making it a valuable resource for training and evaluating AI models. The images in the dataset cover a diverse range of patient demographics, including various age groups and both genders. This diversity helps ensure the robustness and generalizability of the developed models.

The dataset is particularly challenging due to several factors. Firstly, pneumonia can exhibit subtle radiographic features, making it difficult to detect, even for experienced radiologists. The RSNA dataset includes cases with varying degrees of opacity and infiltration, representing the different manifestations of pneumonia, further enhancing the complexity of the task. Additionally, the dataset contains confounding factors commonly encountered in clinical practice, such as overlapping structures, other lung abnormalities, medical devices, and surgical hardware. These factors mimic real-world scenarios and add realism to the training process, enabling models to learn and adapt to complex clinical situations.

To ensure the accuracy and reliability of the dataset, expert radiologists have meticulously annotated each image for the presence of pneumonia. The annotations classify the image as either "normal" or "pneumonia" and provide precise localization information when pneumonia is present. These annotations serve as ground truth labels for training and evaluating AI models, enabling the development of accurate and robust pneumonia detection algorithms.Researchers and data scientists have utilized the RSNA Pneumonia Detection Dataset to train deep learning models, such as convolutional neural networks (CNNs), to automatically detect and classify pneumonia from chest X-ray images. These models leverage the large-scale annotated dataset to learn complex patterns and features associated with pneumonia, enabling accurate and automated detection.

The availability of the RSNA Pneumonia Detection Dataset has facilitated comparative studies and benchmarking of different AI algorithms. It has also provided a standardized evaluation platform for researchers to assess the performance, generalizability, and interpretability of their models. This has contributed to the advancement of AI-based pneumonia detection systems and has the potential to improve clinical decision-making and patient outcomes.

The distribution of instances in the RSNA Pneumonia Detection Dataset is as follows: out of a total of 26,684 X-ray images, the data is unbalanced, with 20,672 images classified as without pneumonia and 6,012 images classified as having pneumonia. This distribution indicates that the dataset

contains a larger proportion of images without pneumonia compared to those with pneumonia. The majority of instances (20,672 images) depict cases where pneumonia is not present. These images serve as negative examples, representing normal lung conditions or other non-pneumonia abnormalities. These instances play a crucial role in training AI models to accurately distinguish between normal and abnormal chest X-ray findings, reducing the risk of false positives.

2.2 Preprocessing

The preprocessing steps applied to the dataset include the following:

1) **Resizing Images**: The original images in the dataset have dimensions of 1024x1024 pixels. To make the images compatible with the desired input size of the model, they are resized to 224x224 pixels. Resizing the images reduces their resolution but allows for faster processing and conserves memory.

2) **Standardizing Pixel Values**: The pixel values of the resized images are then standardized, ensuring that they fall within the interval [0,1]. This normalization step is crucial for ensuring consistent and uniform data representation across the dataset. Standardizing the pixel values simplifies further processing and enhances the model's ability to learn from the data effectively.

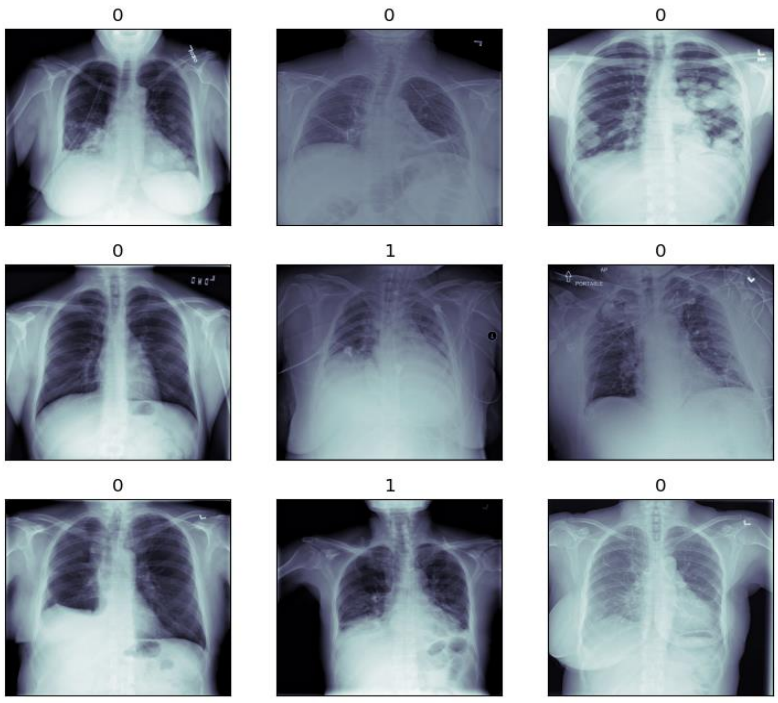
3**) Dataset Split**: The dataset is split into two subsets: a training set and a validation set. **24,000** images are allocated for the training set, while **2,684** images are set aside for validation. The training set is used to train the AI model, while the validation set is used to evaluate the model's performance and fine-tune its parameters.

4) **Folder Organization**: To facilitate data management, the preprocessed images are stored in separate folders based on their class labels. Images labeled as "no pneumonia" are placed in a folder labeled as 0, while images labeled as "pneumonia" are stored in a folder labeled as 1. This organization allows for easy access to the images during the training and evaluation stages.

5) **Data Augmentation**: Data augmentation techniques are applied to enhance the diversity and variability of the dataset. This step involves applying **random rotations**, **random translations**, **random scale adjustments**, and **random resized crops** to the images. These transformations introduce slight modifications to the images, creating new variations while preserving the fundamental characteristics of the original data. Data augmentation helps prevent overfitting by exposing the AI model to a wider range of image variations, improving its generalization capabilities.

```
from pathlib import Path
import pydicom
import numpy as np
import cv2
import pandas as pd
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
from google.colab import drive
drive.mount('/content/drive')

# read labels from .csv file
labels =
pd.read_csv('/content/drive/Shareddrives/Gdrive/Dataset/Pneumonia_Detection/st
age_2_train_labels.csv')
```

```
# remove duplicates
labels = labels.drop_duplicates('patientId')
ROOT_PATH =
Path("/content/drive/Shareddrives/Gdrive/Dataset/Pneumonia_Detection/stage_2_t
rain_images")
SAVE_PATH =
Path("/content/drive/Shareddrives/Gdrive/Dataset/Pneumonia_Detection/Processed
")

# visualize some data
fig, axis = plt.subplots(3, 3, figsize=(9,9))
c = 0
for i in range(3):
    for j in range(3):
        # get file name from labels
        patient_id = labels.patientId.iloc[c]
        # create file name path
        dcm_path = ROOT_PATH/patient_id
        dcm_path = dcm_path.with_suffix('.dcm')
        # read image
        dcm = pydicom.read_file(dcm_path).pixel_array
        # read label
        label = labels['Target'].iloc[c]
        # visualize
        axis[i][j].imshow(dcm, cmap='bone')
        axis[i][j].set_title(label)
        axis[i][j].set_xticks([])
        axis[i][j].set_yticks([])
        c += 1
```



```
# calculate standart deviation and mean of images for standardization
sums, sums_squared = 0, 0

for c, patient_id in enumerate(tqdm(labels.patientId)):

    # get file name from labels
    patient_id = labels.patientId.iloc[c]

    # create file name path
    dcm_path = ROOT_PATH/patient_id
    dcm_path = dcm_path.with_suffix('.dcm')

    # read image
    dcm = pydicom.read_file(dcm_path).pixel_array / 255 # standardization

    dcm_array = cv2.resize(dcm, (224,224)).astype(np.float16)

    label = labels.Target.iloc[c]

    # images before 24000 th are belongs to training set
    train_or_val = 'train' if c < 24000 else 'val'

    current_save_path = SAVE_PATH/train_or_val/str(label)
    current_save_path.mkdir(parents=True, exist_ok=True)

    # save it
    np.save(current_save_path/patient_id, dcm_array)

    # calculate sums and sums_square
    normalizer = 224 * 224

    if train_or_val == 'train':
        # validationset must not have any effect on training set
        sums += np.sum(dcm_array) / normalizer
        sums_squared += (dcm_array ** 2).sum() / normalizer

drive.flush_and_unmount()
```

2.3 Loading data

To train a deep learning model effectively, it is essential to properly load and prepare the data. Loading data involves the process of acquiring the preprocessed dataset and organizing it into a format that can be readily fed into the model during the training phase. This critical step sets the foundation for training the model by ensuring that the data is accessible, correctly structured, and efficiently utilized. In this section, we will discuss the steps involved in loading the preprocessed data, including reading the images, applying necessary data transformations, and creating data loaders. By understanding how to load the data effectively, we can seamlessly integrate it with the deep model, enabling us to capitalize on the full potential of the dataset for training the model to make accurate predictions and uncover meaningful insights.

We employ transformers to apply specific operations and manipulations to the training and validation data. These transformers help prepare the data by converting it into a suitable format for training the deep model. The `**train_transforms**` represent a set of transformations to be applied exclusively to the training data. In this case, we have the following transformations:

1. `**transforms.ToTensor()**`: This transformation converts the image into a tensor format. Tensors are the primary data structure used in deep learning frameworks, allowing efficient mathematical computations on the image data.

2. `**transforms.normalize(0.49, 0.248)**`: This transformation normalizes the tensor by subtracting the mean of 0.49 and dividing by the standard deviation of 0.248. Normalization is crucial for ensuring that the input data has a consistent and standardized range, which aids in stabilizing the training process.

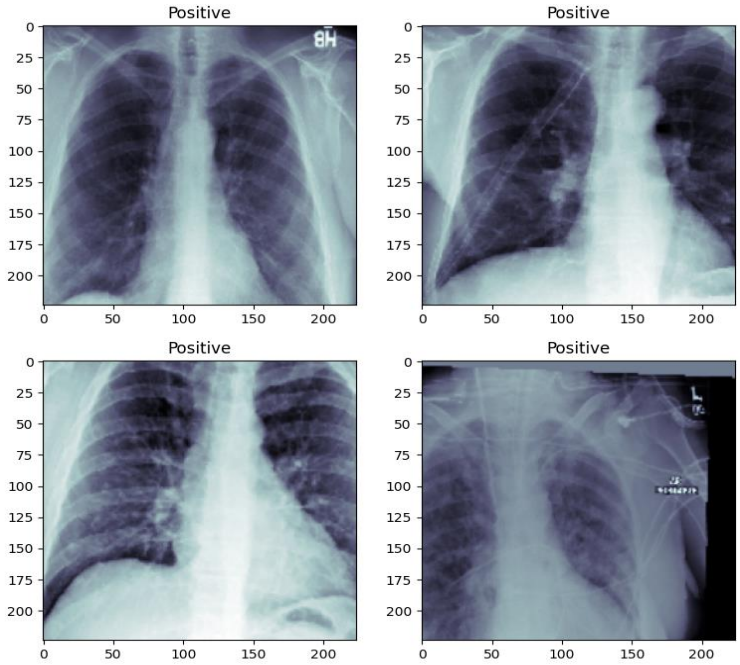
3. `**transforms.RandomAffine(degrees=(-5, 5), translate=(0, 0.05), scale=(0.9, 1.1))**`: This transformation applies a random affine transformation to the image. It involves rotating the image by a random angle between -5 to 5 degrees, translating it randomly by up to 5% in both the x and y directions, and scaling it randomly between 0.9 to 1.1 times its original size. These random transformations provide data augmentation, helping to increase the diversity of the training data and improve the model's generalization capability.

4. `**transforms.RandomResizedCrop((224, 224), scale=(0.35, 1))**`: This transformation applies a random resized crop to the image. It crops the image to a size of 224x224 pixels and randomly scales it between 0.35 to 1 times its original size. Random resized cropping enables the model to learn from various image scales and focuses on different regions, enhancing its ability to handle objects of various sizes.

On the other hand, the `**val_transforms**` represent the transformations applied to the validation data, which may differ from the training data to some extent. In this case, the `val_transforms` include the same transformations as `train_transforms`, except for the random affine and random resized crop transformations. This exclusion ensures that the validation data is processed consistently and does not undergo any random variations that could affect the evaluation of the trained model.

```
# install libraries
!pip install torchmetrics --quiet
!pip install pytorch_lightning --quiet
!pip install tqdm -quiet

import torch # model creation
import torchvision # data loaders
from torchvision import transforms # data augmentation and normalization
import torchmetrics # easy metric computation
import pytorch_lightning as pl
from pytorch_lightning.callbacks import ModelCheckpoint # frequently store
weights
from pytorch_lightning.loggers import TensorBoardLogger # logging in
TensorBoard
from tqdm.notebook import tqdm
import numpy as np
import matplotlib.pyplot as plt
```

```
# load drive
from google.colab import drive
drive.mount('/content/drive')

def load_file(path):
    # Load the file as a numpy array
    # Convert the data to float32 type
    return np.load(path).astype(np.float32)

train_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(0.49, 0.248),
    transforms.RandomAffine(degrees=(-5, 5), translate=(0, 0.05), scale=(0.9,
1.1)),
    transforms.RandomResizedCrop((224, 224), scale=(0.35, 1))
])

val_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(0.49, 0.248)
])

train_dataset =
torchvision.datasets.DatasetFolder('/content/drive/Shareddrives/Gdrive/Dataset
/Pneumonia_Detection/Processed/train/', loader = load_file, extensions='npy',
transform=train_transforms)

val_dataset =
torchvision.datasets.DatasetFolder('/content/drive/Shareddrives/Gdrive/Dataset
/Pneumonia_Detection/Processed/val/', loader = load_file, extensions='npy',
transform=val_transforms)

# visualize some samples to see the effect of augmentation
fig, axis = plt.subplots(2, 2, figsize=(9,9))

for i in range(2):
    for j in range(2):

        random_idx = np.random.randint(0, 24000)

        x_ray, label = train_dataset[random_idx]

        axis[i][j].imshow(x_ray[0], cmap='bone')
        label_str = 'Positive' if label == 0 else 'Negative'
        axis[i][j].set_title(label_str)
```




```
# set hyper parameters
batch_size = 512
num_workers = 2

# create data loaders
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=True)

val_loader = torch.utils.data.DataLoader(val_dataset,
                                         batch_size=batch_size,
                                         num_workers=num_workers,
                                         shuffle=False)

# check the distribution of data
np.unique(train_dataset.targets, return_counts=True)
```

2.4 Model Creation

2.4.1 ResNet Architecture

ResNet-18 is a CNN architecture that is widely used for various computer vision tasks, including image classification and object detection. It was introduced by Kaiming He et al. in the paper "Deep Residual Learning for Image Recognition" in 2016. The main innovation of ResNet-18 lies in the use of **residual connections**, also known as **skip connections** or shortcut connections. These connections enable the network to learn residual mappings, which significantly ease the training process for very deep networks. By introducing skip connections, ResNet-18 effectively mitigates the vanishing gradient problem associated with deep networks, allowing gradients to flow more directly during training.

ResNet-18 architecture consists of a stack of convolutional layers, followed by a global average pooling layer and a fully connected layer for classification. The convolutional layers are organized into several residual blocks, each containing multiple convolutional layers and a shortcut connection. The shortcut connection adds the original input of the block to the output feature maps of the block, effectively bypassing the convolutional layers. This way, the network can learn the residual between the input and the desired output. Each residual block in ResNet-18 architecture typically follows a "**building block**" structure, which can be a combination of convolutional layers, batch normalization, and activation functions like ReLU. The number of residual blocks and the number of filters in each block increase as we move deeper into the network, enabling the model to learn increasingly complex and abstract features.

ResNet-18 has proven to be highly effective and efficient in image classification tasks, outperforming previous CNN architectures on benchmark datasets. Its architecture has also served as the foundation for subsequent variants such as ResNet-34, ResNet-50, and so on, which further increase the depth and capacity of the network. Figure 1 illustrates the ResNet architecture.

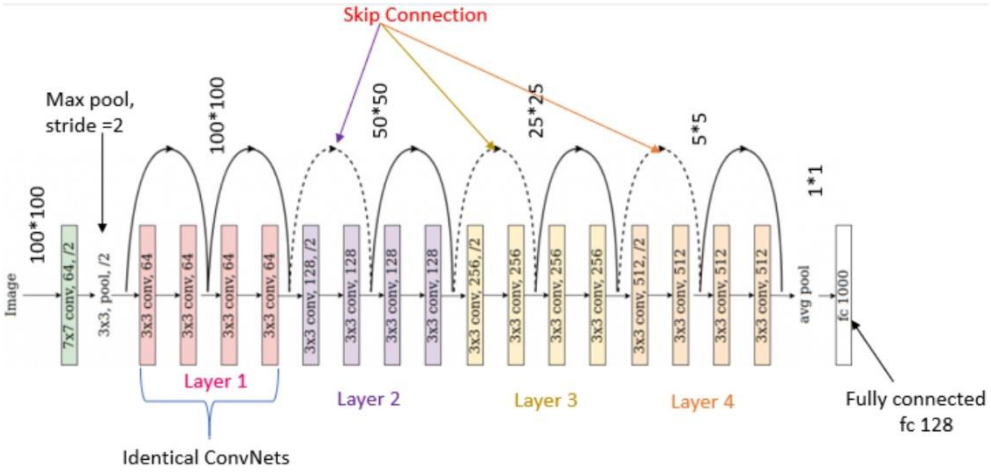


Figure 1 Original ResNet-18 Architecture

2.4.2 Pytorch Lightning

PyTorch Lightning is a lightweight PyTorch wrapper that simplifies the process of organizing, structuring, and scaling PyTorch code for training deep learning models. It provides a high-level interface and a set of abstractions that abstract away boilerplate code and make it easier to write clean and readable code. PyTorch Lightning encourages best practices in deep learning by providing a clear separation between model code and training code. It helps in reducing code duplication, simplifying the training loop, and making experiments more reproducible. It also provides built-in support for distributed training, multi-GPU training, and TPU training, making it easier to scale models across different hardware configurations. Some of the key features and benefits of PyTorch Lightning are:

1. Simplified training loop: PyTorch Lightning abstracts away the training loop, allowing you to focus on defining the model and its logic.

2. Cleaner code: With PyTorch Lightning, you can organize your code into modular components, making it more readable and maintainable.

3. Reproducibility: PyTorch Lightning takes care of setting random seeds and managing deterministic behavior, ensuring greater reproducibility across different runs.

4. Automatic checkpointing: PyTorch Lightning automatically handles saving and loading model checkpoints, making it easier to resume training from a specific point.

5. TensorBoard integration: It provides seamless integration with TensorBoard for visualizing training metrics and model graphs.

6. Distributed training: PyTorch Lightning simplifies the process of training models on multiple GPUs or across distributed systems.

7. Easy experimentation: With PyTorch Lightning, it's easier to modify hyperparameters, metrics, and training configurations to quickly experiment and iterate on your models.

2.4.3 Define the modified ResNet-18

We define a **PyTorch Lightning** Module for a Pneumonia classification model based on the ResNet-18 architecture. Let us break down the code and explain its different components. The class `**PneumoniaModel**` inherits from `**pl.LightningModule**`, which is the base class for Lightning modules, providing useful functionalities for training, validation, and testing. In the constructor `__init__`, the ResNet-18 model from torchvision is instantiated by calling `**torchvision.models.resnet18()**`. Afterwards, some **modifications** are made to adapt the ResNet-18 model to the specific problem of Pneumonia classification. In this case, the input channels of the model need to be changed from 3 (RGB) to 1 (grayscale) since the Pneumonia dataset likely contains grayscale images. Then we replace the convolutional layer of the model to accept single-channel inputs. Similarly, the **fully connected layer** (classifier) of the model is modified to have a single output unit (out_features=1), as the Pneumonia classification task is binary (pneumonia or not pneumonia).

An **Adam optimizer** is created for optimizing the model parameters with a learning rate of 1e-4. The binary cross-entropy with logits loss function (`torch.nn.BCEWithLogitsLoss`) is defined as the loss function. The `pos_weight` parameter is set to a tensor containing the value 3, which can be used to give **more weight to the positive class** during training. Two metrics, `train_acc` and `val_acc`, are defined using the `torchmetrics.Accuracy` class. These metrics will be used to track the accuracy during training and validation.

The `**forward**` method is responsible for performing the forward pass through the model. It takes the input data and returns the predictions. The `training_step` function defines the logic for a single training step. It takes a batch of input data (`x_ray`) and labels, performs the forward pass, **calculates the loss** using the defined loss function, logs the loss and accuracy metrics using the `self.log` method, and returns the loss. The `**on_train_epoch_end**` function is called at the end of each training epoch. It uses the `self.train_acc` metric to compute and log the overall training accuracy for the epoch. The `validation_step` function defines the logic for a single validation step. Similar to the `training_step`, it takes a batch of input data and labels, performs the forward pass, calculates the loss, and logs the metrics for validation. The `on_validation_epoch_end` function is called at the end of each validation epoch to compute and log the overall validation accuracy using the `self.val_acc` metric. The `configure_optimizers` function specifies the optimizer to be used for training, in this case, the `self.optimizer` created earlier.

```
class PneumoniaModel(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.model = torchvision.models.resnet18()
        self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7),
                                   stride=(2, 2), padding=(3, 3), bias=False)
        self.model.fc = torch.nn.Linear(in_features=512, out_features=1,
                                        bias=True)
        # create the optimizer
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=1e-4)
        # loss function
        self.loss_fn=torch.nn.BCEWithLogitsLoss(pos_weight=torch.tensor([3]))
        # create the metric
        self.train_acc = torchmetrics.Accuracy(task='binary')
        self.val_acc = torchmetrics.Accuracy(task='binary')

    def forward(self, data):
        '''This function is responsible for computing the forward pass'''
        pred = self.model(data)
        return pred

    def training_step(self, batch, batch_idx):
        x_ray, label = batch
        label = label.float()
        pred = self(x_ray)[:, 0]
        loss = self.loss_fn(pred, label)
        self.log('Train Loss', loss)
        self.log('Step Train Acc', self.train_acc(torch.sigmoid(pred),
                                   label.int()))

        return loss

    def on_train_epoch_end(self, outs):
        self.log('Train ACC', self.train_acc.compute())

    def validation_step(self, batch, batch_idx):

        x_ray, label = batch
        label = label.float()
        pred = self(x_ray)[:, 0]
        loss = self.loss_fn(pred, label)
        self.log('Val Loss', loss)
        self.log('Step Val Acc', self.val_acc(torch.sigmoid(pred),
                                              label.int()))
    def on_validation_epoch_end(self):
        self.log('Val ACC', self.val_acc.compute())
    def configure_optimizers(self):
        return [self.optimizer]
```

2.5 Model Training

This section involves creating a model, defining a checkpoint callback, creating a trainer object, and fitting the model using the trainer.

1. Creating the model:

It starts by creating an instance of the `**PneumoniaModel**` class. This assumes that the `PneumoniaModel` class is defined elsewhere and contains the implementation of your pneumonia detection model.

2. Creating the checkpoint callback:

The `**ModelCheckpoint**` callback is created with specific configurations. `**monitor='Val ACC'**` sets the metric to monitor for determining the best models during training. In this case, it is monitoring the validation accuracy. `**save_top_k=10**` specifies to save the top 10 best models based on the monitored metric. `**mode='max'**` indicates that a higher metric value is considered better.

3. Creating the trainer:

The `Trainer` object is created with various configurations. `**accelerator="auto"**` automatically selects the appropriate accelerator (e.g., GPU, TPU) if available or falls back to CPU. `**logger=TensorBoardLogger(...)**` sets up a logger to log training metrics and visualize them in TensorBoard. The `**save_dir**` parameter specifies the directory to save the log files. `**log_every_n_steps=1**` specifies that logs should be generated for every training step. `**callbacks=checkpoint_callback**` assigns the created checkpoint callback to the trainer. `**max_epochs=35**` sets the maximum number of training epochs.

4. Fitting the model:

The `**trainer.fit**` method is called to start the training process. `**model**` is the instance of the `**PneumoniaModel**` created earlier. `**train_loader**` and `**val_loader**` are assumed to be PyTorch data loaders providing the training and validation data, respectively.

5. Saving the checkpoint:

After the fitting process completes, the trained model's checkpoint is saved using `**trainer.save_checkpoint**`. The checkpoint file is saved with the name `**'Pneumonia_Detetion_Resnet18_Epoch35.ckpt'**`.

```
# create model
model = PneumoniaModel()
# create Checkpoint callbacks
checkpoint_callback = ModelCheckpoint(
    monitor= 'Val ACC',
    save_top_k= 10,
    mode = 'max')
# create the trainer
trainer = pl.Trainer(accelerator="auto", logger=TensorBoardLogger(
    save_dir='/content/drive/Shareddrives/Gdrive/Dataset/Pneumonia_Detection/l
ogs'), log_every_n_steps= 1, callbacks= checkpoint_callback, max_epochs=35)

trainer.fit(model, train_loader, val_loader)
trainer.save_checkpoint('Pneumonia_Detetion_Resnet18_Epoch35.ckpt')
```

3. Results

3.1 Metrics

3.1.1 ACCURACY

Accuracy is a commonly used metric to evaluate the performance of a machine learning model, particularly for binary classification problems. It measures how often the model correctly predicts the class labels for the given dataset. In a binary classification problem, there are two possible classes, often referred to as the positive class (e.g., class 1) and the negative class (e.g., class 0). The accuracy of a model is determined by the number of correct predictions (both true positives and true negatives) divided by the total number of predictions. The formula for accuracy in a binary classification problem is:

$$Accuracy = \frac{TP + TN}{Total\ number\ of\ predictions}$$

True positives (TP): The model correctly predicts instances of the positive class.

True negatives (TN): The model correctly predicts instances of the negative class.

Total number of predictions: The sum of true positives, false positives, true negatives, and false negatives.

3.1.2 PRECISION

In machine learning, precision is a performance metric used to evaluate the accuracy of a classification model in a binary problem. It measures the proportion of correctly predicted positive instances out of the total instances predicted as positive. Here's the formula for precision in a binary problem:

$$Precision = \frac{TP}{TP + FP}$$

To calculate precision, you divide the number of true positives by the sum of true positives and false positives. The result is a value ranging from 0 to 1, where 1 indicates perfect precision (all positive predictions are correct) and 0 indicates no precision (all positive predictions are incorrect). Precision is a valuable metric, especially in scenarios where you want to minimize false positive predictions.

3.1.3 RECALL

Recall (also known as sensitivity or true positive rate) is a performance metric used to evaluate the ability of a classification model to identify all positive instances correctly in a binary problem. It measures the proportion of correctly predicted positive instances out of the actual positive instances. Here's the formula for recall in a binary problem:

$$Recall = \frac{TP}{TP + FN}$$

To calculate recall, you divide the number of true positives by the sum of true positives and false negatives. The resulting value also ranges from 0 to 1, where 1 indicates perfect recall (all positive instances are correctly identified) and 0 indicates no recall (all positive instances are missed). Recall is an important metric, especially in scenarios where you want to minimize false negatives. For

example, in a medical diagnosis task, you want to ensure that all instances of a certain disease are correctly identified to avoid missing potential cases.

3.2 Model Evaluation

1. Checking device availability and setting device:

- The code checks if a CUDA-compatible GPU is available using `torch.cuda.is_available()`.

- If GPU is available, it sets the device to 'cuda:0'; otherwise, it sets the device to 'cpu'.

- The selected device is then printed.

2. Loading the checkpoint:

- The code loads the pretrained model from the checkpoint file using `PneumoniaModel.load_from_checkpoint`.

- The checkpoint file path is '/kaggle/working/Pneumonia_Detetion_Resnet18_Epoch35.ckpt'.

3. Setting model evaluation mode and device:

- The model's evaluation mode is enabled using `model.eval()` to disable certain operations like dropout.

- The model is then moved to the selected device using `model.to(device)` to ensure computations are performed there.

4. Making predictions:

- Predictions are made for each data point in the `val_dataset` using a loop.

- The loop iterates over the data and label pairs in `val_dataset`.

- Each data point is sent to the device and converted to float using `.to(device).float()`.

- The data is unsqueezed to add a batch dimension using `.unsqueeze(0)`.

- The model's forward pass is conducted on the data, passing it through `model(data)`. The output is a prediction tensor.

- The prediction tensor is passed through a sigmoid function using `torch.sigmoid` to obtain the predicted probability.

- The predicted probability tensor is appended to the `preds` list, and the label is appended to the `labels` list.

5. Converting predictions and labels to tensors:

- The `preds` list is converted to a tensor using `torch.tensor(preds)`.

- The `labels` list is converted to a tensor of integer values using `torch.tensor(labels).int()`.

6. Computing evaluation metrics:

- Several evaluation metrics are computed using the `torchmetrics` library.

- `torchmetrics.Accuracy`, `torchmetrics.Precision`, `torchmetrics.Recall`, and `torchmetrics.ConfusionMatrix` are used to compute accuracy, precision, recall, and confusion matrix, respectively.

- The metrics are computed by calling each metric with the predicted probabilities tensor (`preds`) and the labels tensor (`labels`).

- The computed metrics are stored in separate variables (`acc`, `precision`, `recall`, `cm`).

7. Printing the evaluation results:

- The calculated evaluation metrics (`acc`, `precision`, `recall`, `cm`) are printed using `print()`.

- Each metric is printed on a separate line with an appropriate label.

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)

# load the ckpt
model =
PneumoniaModel.load_from_checkpoint('/kaggle/working/Pneumonia_Detetion_Resnet
18_Epoch35.ckpt')

model.eval()
model.to(device)

preds = []
labels = []
with torch.no_grad():
    for data, label in tqdm(val_dataset):
        data = data.to(device).float().unsqueeze(0)
        pred = torch.sigmoid(model(data)[0].cpu())

        preds.append(pred)
        labels.append(label)

preds = torch.tensor(preds)
labels = torch.tensor(labels).int()

# compute accuracy
acc = torchmetrics.Accuracy(task='binary')(preds, labels)
precision = torchmetrics.Precision(task='binary')(preds, labels)
recall = torchmetrics.Recall(task='binary')(preds, labels)
cm = torchmetrics.ConfusionMatrix(num_classes=2,task='binary')(preds, labels)

print(f'Accuracy: {acc}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'CM:\n{cm}')
```

3.3 Discussion

The accuracy of the model is calculated to be approximately 82.56%. However, accuracy alone might not give the complete picture, especially in imbalanced datasets. The precision of the model is calculated to be around 58.73%. A lower precision suggests that the model has a higher chance of predicting false positives. The recall of your model is approximately 76.20%. A higher recall implies that the model is capturing a higher number of true positives. The confusion matrix provides further insights into the model's performance. It visualizes the predicted labels against the actual labels. From the provided confusion matrix, we can derive the following:

- **True Positives (TP):** 461 cases were correctly classified as pneumonia.
- **True Negatives (TN):** 1755 cases were correctly classified as non-pneumonia.
- **False Positives (FP):** 324 cases were incorrectly classified as pneumonia. These are the cases where the model predicted pneumonia, but the actual condition was non-pneumonia.
- **False Negatives (FN):** 144 cases were incorrectly classified as non-pneumonia. These are the cases where the model predicted non-pneumonia, but the actual condition was pneumonia.

4. Interpretation of Model

In this section we implement the concept of Class Activation Mapping (CAM) to interpret the predictions of a deep model for pneumonia detection. Let's break down the code and understand its logic step by step:

Model Definition: PneumoniaModel is a PyTorch Lightning module that extends the LightningModule class. It serves as the main model for pneumonia detection. In the initialization (__init__) method, the model is defined based on ResNet-18 architecture (self.model = torchvision.models.resnet18()). Modification is made to the first convolution layer of the ResNet-18 model to handle grayscale images (self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)). The last fully connected layer (self.model.fc) is replaced with a new linear layer to output a single value (assume probability or logit for pneumonia).

Feature Extraction: self.feature_map is defined as a sequential module consisting of all layers of the model up to the penultimate layer (list(self.model.children())[:-2]). In the forward method, the input data is passed through the self.feature_map module to obtain the feature map. An adaptive average pooling operation is applied to obtain a fixed-size feature representation (avg_pool_output) regardless of the input image size. The feature representation is flattened (avg_output_flattened) and passed through the model's fully connected layer (self.model.fc) to obtain a prediction (pred).

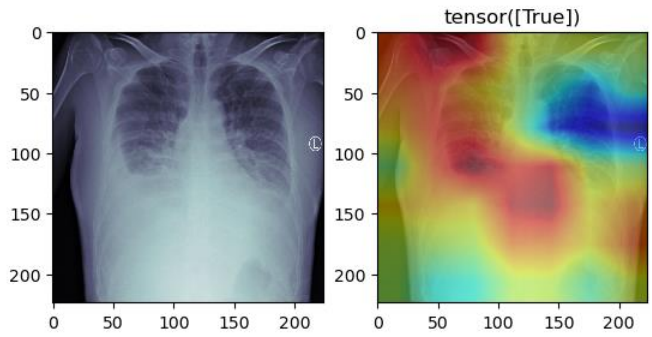
CAM Calculation: The cam function takes the trained model (model) and an input image (img) as parameters to calculate the Class Activation Map. Inside the function, the forward pass is performed using the input image, and the output features are obtained (features). The features are reshaped into a 2D tensor of size (512, 49) to match the dimensions required for matrix multiplication (torch.matmul) with the weights of the fully connected layer. The weights (weight) of the fully connected layer are extracted (list(model.model.fc.parameters())[0][0].detach()) and multiplied with the reshaped features to obtain the class activation map (cam). The CAM is then reshaped to match the dimensions of the original input image (7x7) to visualize the importance of different regions.

Visualization: The visualize function takes the input image (img), class activation map (cam), and prediction (pred) as parameters to visualize the CAM overlay on the input image. The input image and class activation map are resized to the same dimensions using transforms.functional.resize. A figure with two subplots is created using plt.subplots. The original input image is plotted on the left

axis (axis[0]) and the image with the CAM overlay is plotted on the right axis (axis[1]). The CAM is plotted with transparency (alpha=0.5) and a color map (cmap='jet') to highlight the regions contributing to the prediction. The title of the plot indicates the prediction value (pred > 0.5) for binary classification.

```
class PneumoniaModel(pl.LightningModule):
    def __init__(self):
        super().__init__()

        self.model = torchvision.models.resnet18()
        self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7),
stride=(2, 2), padding=(3, 3), bias=False)
        self.model.fc = torch.nn.Linear(in_features=512, out_features=1)

        self.feature_map = torch.nn.Sequential(*list(self.model.children())[:-
2])



    def forward(self, data):
        feature_map = self.feature_map(data)
        avg_pool_output = torch.nn.functional.adaptive_avg_pool2d(input =
feature_map, output_size=(1,1))
        avg_output_flattened = torch.flatten(avg_pool_output)
        pred = self.model.fc(avg_output_flattened)
        return pred, feature_map

model =
PneumoniaModel.load_from_checkpoint('/kaggle/working/Pneumonia_Detetion_Resnet
18_Epoch35.ckpt', strict=False)
model.eval();

def cam(model, img):
    # class activation map
    with torch.no_grad():
        pred, features = model(img.unsqueeze(0))

    features = features.reshape((512,49))

    weight_params = list(model.model.fc.parameters())[0]
    weight = weight_params[0].detach()

    cam = torch.matmul(weight, features)

    cam_img = cam.reshape(7,7).cpu()

    return cam_img, torch.sigmoid(pred)


def visualize(img, cam, pred):
```

```
    img = img[0]
    cam = transforms.functional.resize(cam.unsqueeze(0), (224,224))[0]

    fix, axis = plt.subplots(1,2)

    axis[0].imshow(img, cmap='bone')
    axis[1].imshow(img, cmap='bone')
    axis[1].imshow(cam, alpha=0.5, cmap='jet')

    plt.title(pred > 0.5)


img = val_dataset[170][0]
actvation_map, pred = cam(model, img)
visualize(img, actvation_map, pred)
```



References

1. World Health Organization. Pneumonia. Retrieved from https://www.who.int/health-topics/pneumonia

2. Mayo Clinic. Pneumonia. Retrieved from https://www.mayoclinic.org/diseases-conditions/pneumonia/symptoms-causes/syc-20354204

3. Jain, S., & Williams, DJ. Pneumonia in Children. In: StatPearls [Internet]. Treasure Island (FL): StatPearls Publishing; 2022. Retrieved from https://www.ncbi.nlm.nih.gov/books/NBK459171/

4. Torres, A., et al. (2019). Treatment of hospital-acquired pneumonia and ventilator-associated pneumonia: a comparison of 2 international guidelines. Chest, 155(4), 1110–1118. doi: 10.1016/j.chest.2018.12.016

5. Metlay, JP., Waterer, GW., Long, AC., Anzueto, A., Brozek, J., Crothers, KA., Cooley, LA., Dean, NC., Fine, MJ., Flanders, SA., Griffin, MR., Metersky, ML., Musher, DM., Restrepo, MI., Whitney, CG. (2019). Diagnosis and Treatment of Adults with Community-acquired Pneumonia: An Official Clinical Practice Guideline of the American Thoracic Society and Infectious Diseases Society of America. Am J Respir Crit Care Med, 200(7), e45–e67. doi: 10.1164/rccm.201908-1581ST

6. Self, WH., Courchesne, M., McNaughton, CD., Casey, JD., Grijalva, CG., Waldrop, G., Cecco, S., Stothert, JC., Jain, S., Anderson, DJ., Edwards, KM., et al. (2017). Validating Clinical Score for Identifying Low-Risk Adults in the Emergency Department with Community-Acquired Pneumonia. J Emerg Med, 53(5), 699-708. doi: 10.1016/j.jemermed.2017.06.053

7. Rajpurkar, P., Irvin, J., Zhu, K., et al. (2017). CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning. arXiv preprint arXiv:1711.05225.

8. Yao, L., Dong, L., Zhang, H., et al. (2019). A Deep Learning System to Screen Novel Coronavirus Disease 2019 Pneumonia. Engineering, 6(10), 1122-1129.

9. Zhu, D., Zhang, H., Zhang, D., et al. (2020). Diagnosis of Common Thoracic Diseases Based on Radiographic Images: A Deep Learning Method. Journal of Digital Imaging, 33(2), 537-548.

10. Radiological Society of North America (RSNA). RSNA Pneumonia Detection Challenge. Retrieved from https://www.kaggle.com/c/rsna-pneumonia-detection-challenge

11. Chartrand, G., Cheng, PM., Vorontsov, E., et al. (2018). Deep learning: a primer for radiologists. Radiographics, 37(7), 2113-2131.

12. Zhu, T., Liu, Y., Wang, J., et al. (2020). Detection of Pneumonia by Deep Learning in Chest Radiographs and CT Scans: A Systematic Review and Meta-Analysis. In: Health Information Science Lecture Notes in Computer Science, 12440.

13. He, K., Zhang, X., Ren, S., et al. (2016). Deep Residual Learning for Image Recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 770-778.

14. Samek, W., Binder, A., Montavon, G., et al. (2017). Evaluating the visualization of what a Deep Neural Network has learned. IEEE Transactions on Neural Networks and Learning Systems, 28(11), 2660-2673.

15. Obermeyer, Z., & Emanuel, EJ. (2016). Predicting the Future - Big Data, Machine Learning, and Clinical Medicine. The New England Journal of Medicine, 375(13), 1216-1219.