SAT 5165 Final Project

Title: Spark-Based Deep Convolutional Neural Network for Pneumonia Detection in Chest X-Rays

Github Link: https://github.com/omgette/finalProject

Team Members
● Olivia Gette
● Andrea Wroblewski

Background

The primary goal of this project is to build a pipeline for pneumonia detection in chest X-ray images using Apache Spark for distributed data preprocessing and a custom Convolutional Neural Network (CNN) for classification. Pneumonia is one of the most common and dangerous respiratory illnesses worldwide, and early detection is essential for improving patient outcomes. Chest X-rays are widely available in clinical settings, making automated classification models a valuable tool for supporting diagnostic workflows.

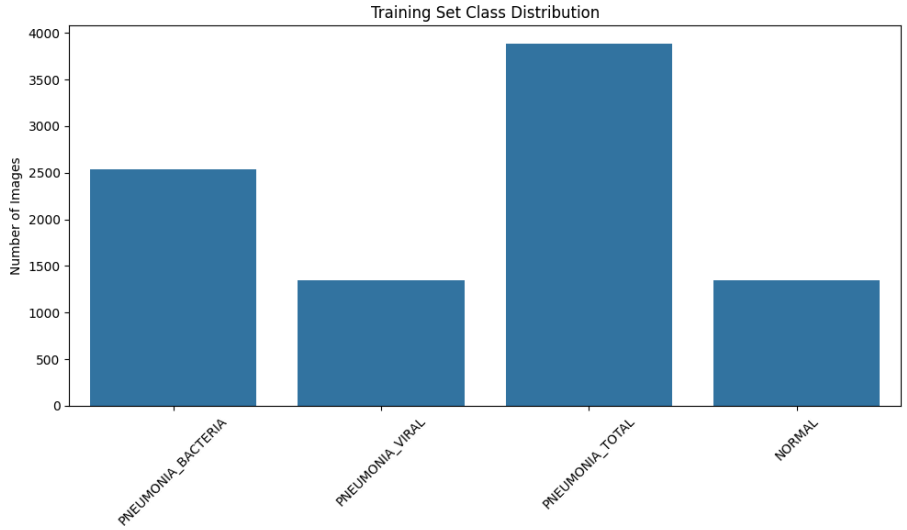
We used the publicly available Chest X-Ray Pneumonia dataset from Kaggle, which contains 5,856 labeled X-ray images across three clinical categories: Normal, Bacterial Pneumonia, and Viral Pneumonia, although we end up classifying into two classes (Normal vs Pneumonia) instead of three. Medical imaging datasets are often large and stored across many directories and subfolders, making consistent ingestion, labeling, and organization a challenge. To address this, we use Apache Spark to construct a distributed preprocessing pipeline capable of scanning the entire dataset, extracting label information, and building a unified manifest of image paths and their corresponding class indices.

To handle the scale of the dataset, all preprocessing is executed on a four-VM Spark cluster consisting of one Master Node and three Worker Nodes. This distributed setup enables parallel directory scanning, faster metadata extraction, and efficient splitting of the dataset into training, validation, and test sets. The Spark output is saved to HDFS, allowing consistent access from downstream training scripts and ensuring reproducibility.

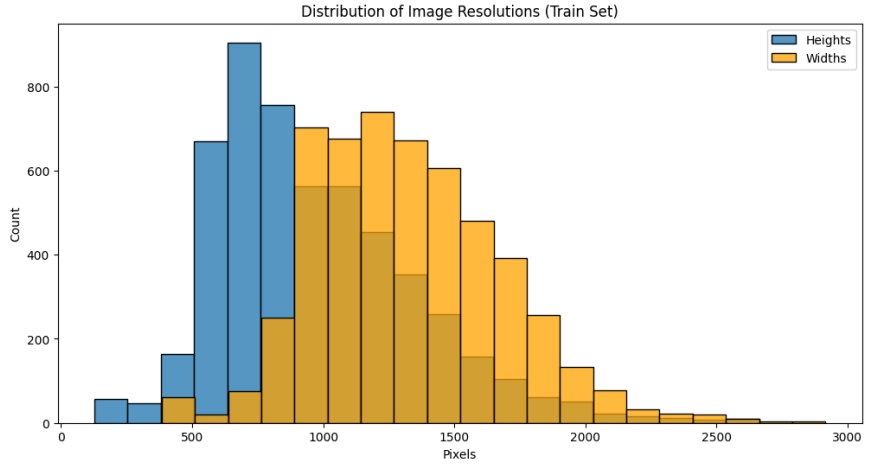
Overall, this project demonstrates how Spark can be combined with deep learning to create a scalable medical imaging pipeline. By distributing preprocessing across multiple VMs and centralizing training on a reproducible manifest, we show how large imaging datasets can be handled efficiently even in CPU-only environments. The final model provides automated pneumonia classification and highlights the potential of distributed computing in clinical decision support applications.

Exploratory Data Analysis

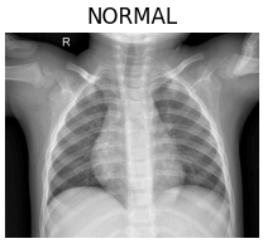
We conducted an exploratory analysis of the Chest X-Ray Pneumonia dataset (Kaggle), which contains 5,856 pediatric chest X-ray images. The dataset is highly imbalanced, with pneumonia cases appearing roughly three times more often than normal cases (training folder: 3,883 pneumonia vs 1,349 normal). This imbalance means we need to carefully choose our evaluation metrics and potentially add some class-balancing techniques during training. A barplot was created to visualize this imbalance:

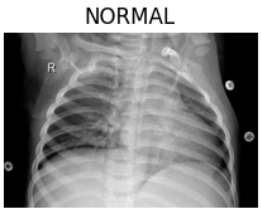


The original images show substantial variability in resolution, as confirmed by a histogram of image dimensions. This justified our preprocessing choice to resize all images to 128×128 for consistency across the model pipeline.



When examining overall pixel statistics, we found that the mean pixel intensity was similar across classes (around 122-123). However, pneumonia images exhibited notably higher intensity variance, reflecting the presence of bright, opaque lung infiltrates commonly associated with infection. These findings support the clinical differences observed in X-ray pathology and provide context for how the model may learn to distinguish between classes. An example of this difference is shown below:





Data Engineering and HDFS Preprocessing Pipeline:

Before training the model, all image files from the Kaggle Chest X-Ray Pneumonia dataset had to be prepared, organized, and distributed across the Hadoop cluster for preprocessing. The dataset was first downloaded onto the Master VM, where the directory structure for storing the raw images was created.

```
[sat3812@master ~]$ mkdir -p /home/sat3812/data

mv ~/Downloads/archive.zip /home/sat3812/data/
```

The zip file was then extracted to the local /home/sat3812/data/ direction on the Master VM. Next the raw images were uploaded to HDFS, allowing the dataset to be processed distributedly across the 4 VM cluster.
An HDFS directory was created to store the images:

```
hdfs dfs -mkdir -p /user/sat3812/cnn/raw
```

Once uploaded, the dataset became accessible to all nodes in the cluster (Master, Worker1, Worker2, and Worker3). Storing the data in HDFS allows:

- **Distributed access:** Each worker node could read image paths in parallel.
- **Fault tolerance:** Files were automatically replicated across nodes.
- **Scalability:** Enabled Spark to preprocess large collections of radiographs efficiently.

Preparing the dataset for distributed preprocessing in Spark.

Distributed Preprocessing with Spark

To prepare the dataset for model training, a PySpark job was executed on the Hadoop cluster to perform distributed preprocessing and create a manifest file for all images.
The script began by defining the HDFS locations for the raw Kaggle dataset and the output manifest:

```
HDFS_ROOT = "hdfs://master:9000/user/sat3812/cnn/raw/chest_xray"
# Output location for the processed manifest
OUTPUT_MANIFEST = "hdfs://master:9000/user/sat3812/cnn/xray_manifest_final"
```

A Spark session was created and configured to run on the cluster:

```
spark = SparkSession.builder.appName("XrayEDAAndSplit").getOrCreate()
spark.sparkContext.setLogLevel("WARN")
print("Spark Session Initialized.")
```

All image files in the Kaggle directory structure were loaded from HDFS using Spark's binaryFile source with recursive lookup:

```
df = (
    spark.read.format("binaryFile")
        .option("recursiveFileLookup", "true")
        .load(HDFS_ROOT)
)
```

From each file path, two pieces of metadata were extracted:

- label: Normal or Pneumonia
- orig_split: The Kaggle train and test split
- This was completed by using regular expressions:

```
df = df.withColumn(
    "label",
    F.regexp_extract("path", r"/(NORMAL|PNEUMONIA)/", 1)
).withColumn(
    "orig_split",
    F.regexp_extract("path", r"/(train|test)/", 1)
)
df = df.filter("label != '' AND orig_split != ''")
```

Class distributions were examined to confirm all the images were labeled correctly.

```
--- Initial Class Distribution ---
+----------+---------+-----+
|orig_split|    label|count|
+----------+---------+-----+
|      test|   NORMAL|  234|
|      test|PNEUMONIA|  390|
|     train|   NORMAL| 1349|
|     train|PNEUMONIA| 3883|
+----------+---------+-----+


--- Final Project Split Distribution --
+-----+---------+-----+
|split|    label|count|
+-----+---------+-----+
| test|   NORMAL|  234|
| test|PNEUMONIA|  390|
|train|   NORMAL| 1079|
|train|PNEUMONIA| 3106|
|  val|   NORMAL|  270|
|  val|PNEUMONIA|  777|
+-----+---------+-----+
```

The first table Initial Class Distribution shows the number of normal and pneumonia images in the original Kaggle train and test folders. After performing the stratified 80/20 train/validation split, a second group-by was used to check the final distribution of images across the train, validation, and test sets. Final Project Split Distribution, show the final count and that class proportions were preserved and that the original Kaggle test set remained unchanged.

After verifying that the final train/val/test splits were balanced, we saved the manifest to HDFS using:

```
(final_manifest_df
    .select("path", "label", "split", "label_idx")
    .write
    .mode("overwrite")
    .option("header", "true")
    .csv(OUTPUT_MANIFEST)
)
```

The manifest was downloaded to the master VM for CNN training.

The preprocessing was performed across the entire Hadoop Cluster. This confirms that recursive file scanning, label extraction, and stratified splitting were executed in a fully distributed manner across the cluster.

▾ Workers (4)

Worker Id	Address	State	Cores	Memory	Resources
worker-20251130132208-192.168.13.201-45867	192.168.13.201:45867	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	
worker-20251130132227-192.168.13.122-35659	192.168.13.122:35659	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	
worker-20251130132442-192.168.13.200-36999	192.168.13.200:36999	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	
worker-20251207113840-192.168.13.123-37945	192.168.13.123:37945	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	

▾ Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20251207140505-0001 (kill)	XrayEDAAndSplit	32	1024.0 MiB		2025/12/07 14:05:05	sat3812	RUNNING	1.1 min

CNN Model:

After completing all distributed preprocessing steps in Spark and downloading the final manifest to the Master VM, the next stage of the project focused on developing and training a CNN to classify chest X-ray images as normal or pneumonia. We updated the model based on your recommendations:

- Do not use max pool:
    - Instead of using max pooling, strided convolutions were used for downsampling to preserve spatial information while reducing feature map size.
- Add a Conv2d (inch, inch) where inch and ouch are same:
    - Convolution layers with equal input and output channels (Conv2d(inch, inch)) were inserted throughout the network.
- Use augmentations (random flips, rotations, shift and wrap, random crops):
    - Random horizontal flips to account for left/right positioning differences
    - Random rotations (±15 degrees) to simulate slight rotational variance in X-rays
    - Random affine transformations, including small translations and zooming, to mimic patient positioning differences

    - Random cropping after resizing, which encourages the model to focus on different areas of the lung field
    - Grayscale normalization to stabilize intensity distribution across the dataset
- Use L1 and L2 regularizers
    - L2 regularization (implemented through the weight_decay parameter in the Adam optimizer) penalizes large weight values and helps reduce model complexity.
    - L1 regularization was added manually by computing the sum of the absolute values of all trainable weights and adding the resulting penalty term to the loss function.
- Use dropout:
    - Two dropout layers were added:
        - Dropout(p = 0.5) before the first fully connected layer
        - Dropout(p = 0.5) before the final classification layer

Next, a helper function was created to read all manifest files generated by Spark into a single, clean pandas DataFrame for use in PyTorch.

```python
def load_manifest(manifest_dir: str):
    """Load and concatenate all CSV parts from the local manifest directory."""
    # Uses glob to find all 'part-*.csv' files downloaded from HDFS
    pattern = os.path.join(manifest_dir, "part-*.csv")
    files = [f for f in glob.glob(pattern) if not os.path.basename(f).startswith("_")]
    if not files:
        raise FileNotFoundError(f"No CSV files found in {manifest_dir}")

    # Read and combine the dataframes
    dfs = [pd.read_csv(f) for f in files]
    df = pd.concat(dfs, ignore_index=True)

    # Simple check for required columns created by Spark
    required_cols = {"path", "label", "split", "label_idx"}
    missing = required_cols - set(df.columns)
    if missing:
        raise ValueError(f"Manifest missing columns: {missing}")
    return df
```

The load_manifest:

- Uses glob to find all part-*.csv files in the manifest directory that were downloaded from HDFS.
- Reads each CSV into a pandas DataFrame and concatenates them into one combined table.
- Checks that the required columns created by Spark are present:
    - path – full HDFS image path
    - label – class name
    - split – dataset partition
    - label_idx – binary label
- Returns a clean manifest DataFrame.

Next, function hdfs_path_to_local is created, it:

```python
def hdfs_path_to_local(path: str):
    """
    Converts a Spark-generated HDFS path into a local filesystem path
    by replacing the HDFS prefix with the LOCAL_DATA_ROOT.
    """
    marker = "/chest_xray/"
    idx = path.find(marker)
    if idx == -1:
        raise ValueError(f"Could not find '{marker}' in path: {path}")

    # Extracts the relative path starting from 'chest_xray/'
    rel_path = path[idx + 1:]
    # Joins the local root with the relative path to form the final local path
    local_path = os.path.join(LOCAL_DATA_ROOT, rel_path)
    return local_path
```

- Takes the HDFS-style image path stored in the Spark manifest.
- Searches the path for the folder marker /chest_xray/
- Extracts the part of the path starting at chest_xray/.
- Joins that relative path to local dataset root:
    - LOCAL_DATA_ROOT = "/home/sat3812/data".
- Produces the correct local filesystem path where the actual image is stored.
- Returns this local path so PIL and PyTorch can open the file.

ManifestImageDataset:

- Takes the combined manifest DataFrame and filters it by split.
- Converts every HDFS path to a local filesystem path using hdfs_path_to_local()
- Stores:
    - self.paths: list of image file paths
    - self.labels: list of numeric labels
- In __getitem__(idx):
    - Opens the image with PIL
    - Converts it to grayscale
    - Applies the transform (train augmentation or eval transform)
    - Returns: image_tensor, label_idx
- Makes it possible for PyTorch DataLoaders to generate mini-batches of preprocessed images.

Data Transformations and DataLoader Setup

Train Transform (Augmentation and Preprocessing):

To reduce overfitting and increase dataset variability, the training transform included the full set of augmentations recommended in class:

```
train_transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE + 32, IMAGE_SIZE + 32)),
    transforms.RandomCrop(IMAGE_SIZE),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.RandomAffine(
        degrees=0,
        translate=(0.1, 0.1),
        scale=(0.9, 1.1)
    ),
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5]),
])
```

- Resize image slightly larger than 128×128
- RandomCrop(128×128)
- RandomHorizontalFlip
- RandomRotation (±15°) for rotational variance
- RandomAffine shifts and zoom
- Convert to grayscale tensor
- Normalize (mean = 0.5, std = 0.5)

Validation and test images must remain consistent and unbiased:

```
eval_transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.Grayscale(num_output_channels=1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5]),
])
```

- Resize to 128×128
- Convert to grayscale
- Convert to tensor
- Normalize (mean = 0.5, std = 0.5)

DataLoaders:

PyTorch DataLoaders were created for each split:

- Batch size: 32
- Shuffle: enabled for training only
- num_workers = 0
- train_loader, val_loader, test_loader feed images into the CNN in mini-batches

```
train_dataset = ManifestImageDataset(manifest_df, split="train", transform=train_transform)
val_dataset   = ManifestImageDataset(manifest_df, split="val",   transform=eval_transform)
test_dataset  = ManifestImageDataset(manifest_df, split="test",  transform=eval_transform)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS)
val_loader   = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)
test_loader  = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)
```

Model Architecture: CNN

Feature Extractor (self.features):

- Block 1: Initial feature extraction (1 to 16 to 32 channels):

```
# Block 1: Initial feature extraction
nn.Conv2d(1, 16, kernel_size=3, padding=1, stride=1),
nn.BatchNorm2d(16),
nn.ReLU(inplace=True),

nn.Conv2d(16, 16, kernel_size=3, padding=1, stride=1),
nn.BatchNorm2d(16),
nn.ReLU(inplace=True),

nn.Conv2d(16, 32, kernel_size=3, padding=1, stride=2),
nn.BatchNorm2d(32),
nn.ReLU(inplace=True),
```

    - Conv2d(1, 16, 3×3, stride=1, padding=1)
        - First layer, learns low-level edges and textures.
    - BatchNorm2d(16) & ReLU
    - Conv2d(16, 16, 3×3, stride=1, padding=1)
    - BatchNorm2d(16) & ReLU
    - Conv2d(16, 32, 3×3, stride=2, padding=1)
        - Strided convolution used for downsampling (instead of MaxPool).

- Block 2: Deeper mid-level features (32 to 64 to 128 channels):

```
# Block 2
nn.Conv2d(32, 64, kernel_size=3, padding=1, stride=1),
nn.BatchNorm2d(64),
nn.ReLU(inplace=True),

nn.Conv2d(64, 64, kernel_size=3, padding=1, stride=1),
nn.BatchNorm2d(64),
nn.ReLU(inplace=True),

nn.Conv2d(64, 128, kernel_size=3, padding=1, stride=2),
nn.BatchNorm2d(128),
nn.ReLU(inplace=True),
```

    - Conv2d(32, 64, 3×3, stride=1, padding=1)
    - BatchNorm2d(64) & ReLU
    - Conv2d(64, 64, 3×3, stride=1, padding=1)
    - BatchNorm2d(64) & ReLU
    - Conv2d(64, 128, 3×3, stride=2, padding=1)
        - Second strided convolution for downsampling.
- Block 3: High-level features (128 to 256 channels):

```
# Block 3
nn.Conv2d(128, 128, kernel_size=3, padding=1, stride=1),
nn.BatchNorm2d(128),
nn.ReLU(inplace=True),

nn.Conv2d(128, 128, kernel_size=3, padding=1, stride=1),
nn.BatchNorm2d(128),
nn.ReLU(inplace=True),

nn.Conv2d(128, 256, kernel_size=3, padding=1, stride=2),
nn.BatchNorm2d(256),
nn.ReLU(inplace=True),
```

    - Conv2d(128, 128, 3×3, stride=1, padding=1)
    - BatchNorm2d(128) & ReLU
    - Conv2d(128, 128, 3×3, stride=1, padding=1)
    - BatchNorm2d(128) & ReLU
    - Conv2d(128, 256, 3×3, stride=2, padding=1)

- Global pooling:

```
self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
```

    - AdaptiveAvgPool2d((1, 1))
        - Converts the final feature map into a 256-dim vector

Classifier (self.classifier):

- Flatten():
  - flattens 256×1×1 to a 256-dim vector.
- Dropout(p=0.5):
  - Regularization
- Linear(256 to 128) & ReLU
- Dropout(p=0.5)
- Linear(128 to num_classes):
  - Final logits for 2 classes.

```
def forward(self, x):
    x = self.features(x)
    x = self.global_pool(x)
    x = self.classifier(x)
    return x
```

Forward Pass:

- x = self.features(x):
  - Convolutional blocks
  - Strided downsampling.
- x = self.global_pool(x):
  - Adaptive average pooling.
- x = self.classifier(x):
  - Dropout
  - Dense layers
  - Class logits

Training Configuration and Regularization:

- Device and Hyperparameters
  - Training ran on the CPU of the master VM.
  - BATCH_SIZE = 32 – balances speed with limited VM memory.
  - NUM_EPOCHS = 10 – upper limit; early stopping used to avoid overfitting.
  - LR = 1e-3 – learning rate for the optimizer.

```
BATCH_SIZE = 32
NUM_EPOCHS = 10
LR = 1e-3
L2_WEIGHT_DECAY = 1e-4
L1_LAMBDA = 1e-6
NUM_WORKERS = 0
IMAGE_SIZE = 128
num_classes = 2
```

```
criterion = nn.CrossEntropyLoss()
# Optimizer implements L2 regularization via weight_decay
optimizer = optim.Adam(model.parameters(), lr=LR, weight_decay=L2_WEIGHT_DECAY)
```

- Loss Function:
    - CrossEntropyLoss
- Optimizer & L2 Regularization:
    - Adam optimizer:
        - weight_decay = L2_WEIGHT_DECAY
            - 1e-4 adds L2 regularization.
- L1 Regularization:
    - Computes the sum of absolute values of all trainable weights.

```
loss = criterion(outputs, targets)
loss = loss + compute_l1_penalty(model, L1_LAMBDA)
```

Training Loop and Validation Monitoring:

- For each epoch:

```
for epoch in range(1, NUM_EPOCHS + 1):
    train_one_epoch(model, train_loader, optimizer, criterion, epoch)
    _, val_acc = evaluate(model, val_loader, criterion, split_name="Val", class_names=class_names)
```

    - train_one_epoch: Runs a full pass over the training DataLoader and prints the training loss.
    - evaluate: computes validation loss and accuracy.
- Validation accuracy is tracked in best_val_acc.
    - When a new best validation accuracy is observed, the model is saved.

```
if val_acc > best_val_acc:
    best_val_acc = val_acc
    torch.save(model.state_dict(), "best_pneumonia_cnn_manifest.pt")
    print(f"New best model saved (Val Acc = {val_acc:.4f})")
```

Early Stopping:

Although the loop is allowed to run up to 10 epochs, the best model is the one with the highest validation accuracy (Epoch 4 in this project).

```
Epoch 1 | Train Loss: 0.3384 | Train Acc: 0.6616
Val Loss: 0.5350 | Val Acc: 0.8042
New best model saved (Val Acc = 0.8042)
Epoch 2 | Train Loss: 0.2873 | Train Acc: 0.7127
Val Loss: 0.9149 | Val Acc: 0.7240
Epoch 3 | Train Loss: 0.2458 | Train Acc: 0.7542
Val Loss: 0.1962 | Val Acc: 0.9169
New best model saved (Val Acc = 0.9169)
Epoch 4 | Train Loss: 0.2417 | Train Acc: 0.7583
Val Loss: 0.1626 | Val Acc: 0.9465
New best model saved (Val Acc = 0.9465)
Epoch 5 | Train Loss: 0.2331 | Train Acc: 0.7669
Val Loss: 0.2512 | Val Acc: 0.8930
```

In the original training script, the plan was to reload the best validation checkpoint at the end of training and immediately evaluate it on the held-out test set inside the same file. However, due to issues on the VM, this final evaluation block did not run. To keep the evaluation logic simple and reproducible, we moved the test-set evaluation into a stand-alone script eval_only.py.

The new script reuses the same components defined in cnn_final.py and performs the evaluation:

- Load the manifest and build the test set
    - Calls load_manifest(MANIFEST_DIR) to read the Spark-generated manifest.
    - Extracts class names from the label column.
    - Creates a ManifestImageDataset for split="test" using the same eval_transform used during validation.
    - Wraps it in a DataLoader with batch_size=32, shuffle=False, num_workers=0.

- Recreate the CNN architecture
    - Instantiates PneumoniaCNN(num_classes=len(class_names)) and moves it to the device (CPU).
- Load the best saved checkpoint.
    - Loads the weights from best_pneumonia_cnn_manifest.pt

```
model.load_state_dict(torch.load("best_pneumonia_cnn_manifest.pt", map_location=DEVICE))
```

    - Uses CrossEntropyLoss as the loss function

```
criterion = torch.nn.CrossEntropyLoss()
```

- Run final test evaluation on unseen test data:
    - Calls the existing evaluate function with split_name="Test".

```
evaluate(model, test_loader, criterion, split_name="Test", class_names=class_names)
```

        - This computes:
            - Test Loss
            - Accuracy
            - Confusion matrix
            - Classification report

Results:

Training was initially configured to run for 10 epochs; however, the model began to show signs of overfitting after Epoch 4. The validation accuracy peaked at 94.65% during Epoch 4, but in Epoch 5 the validation accuracy dropped to 89.30%, even though the training accuracy continued to improve.

This behavior indicates overfitting:

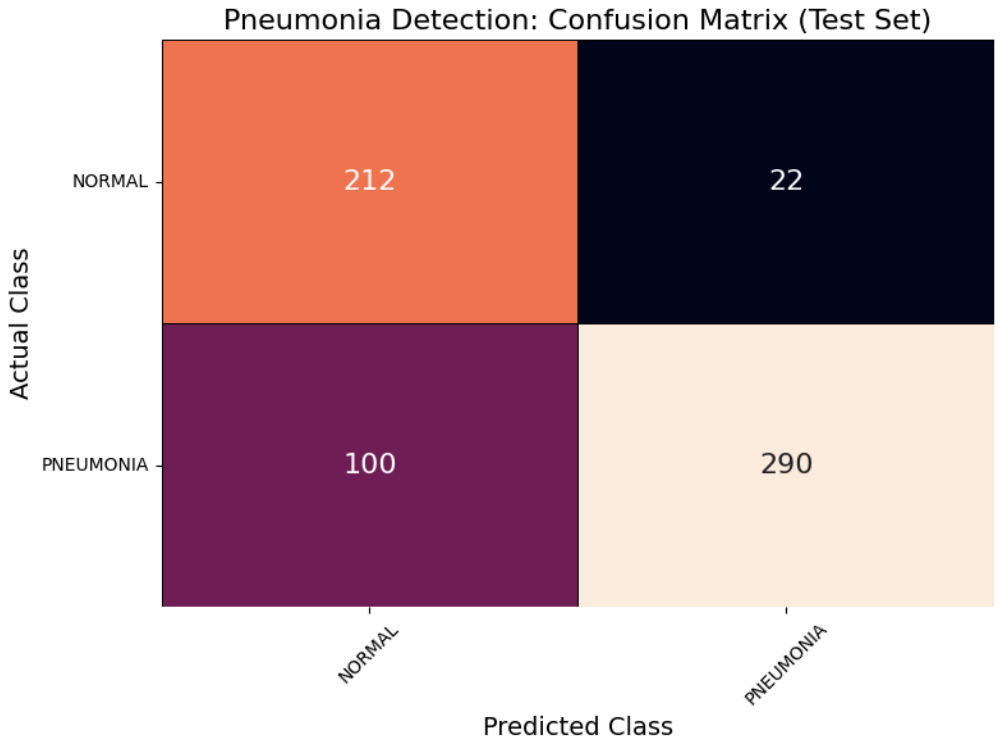
- Training accuracy increases.
- Validation accuracy decreases.
- Validation loss increases.

Summary of Epochs					
Epoch	**Train Loss**	**Train Acc**	**Val Loss**	**Val Acc**	**Action**
1	0.3384	0.6616	0.5350	0.8042	Best model saved
2	0.2873	0.7127	0.9149	0.7240	—
3	0.2458	0.7542	0.1962	0.9169	Best model saved
4	0.2417	0.7583	0.1626	0.9465	Best model saved
5	0.2331	0.7669	0.2512	0.8930	Validation accuracy drops
Terminated to prevent overfitting					

Because Epoch 4 produced the highest validation accuracy, the training process was stopped after Epoch 5, and the Epoch 4 checkpoint (best_pneumonia_cnn_manifest.pt) was selected for final evaluation. This ensured the model used for testing was the best-performing and least overfit version.

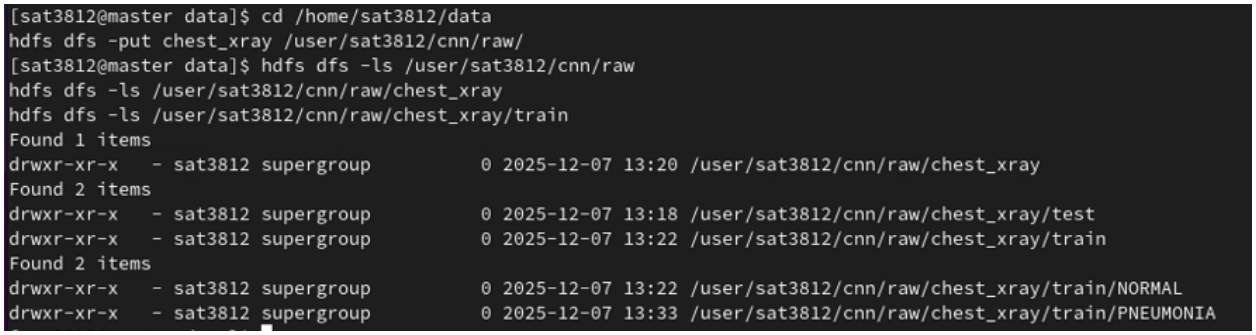
```
Using device: cpu
Loaded 624 samples for split='test'

--- Running Final Test Evaluation ---
Test Loss: 0.4222 | Test Acc: 0.8045

--- Test Set Metrics ---
Confusion Matrix:
[[212  22]
 [100 290]]

Classification Report (Precision, Recall, F1-score):
              precision    recall  f1-score   support

      NORMAL       0.68      0.91      0.78       234
   PNEUMONIA       0.93      0.74      0.83       390

    accuracy                           0.80       624
   macro avg       0.80      0.82      0.80       624
weighted avg       0.84      0.80      0.81       624
```

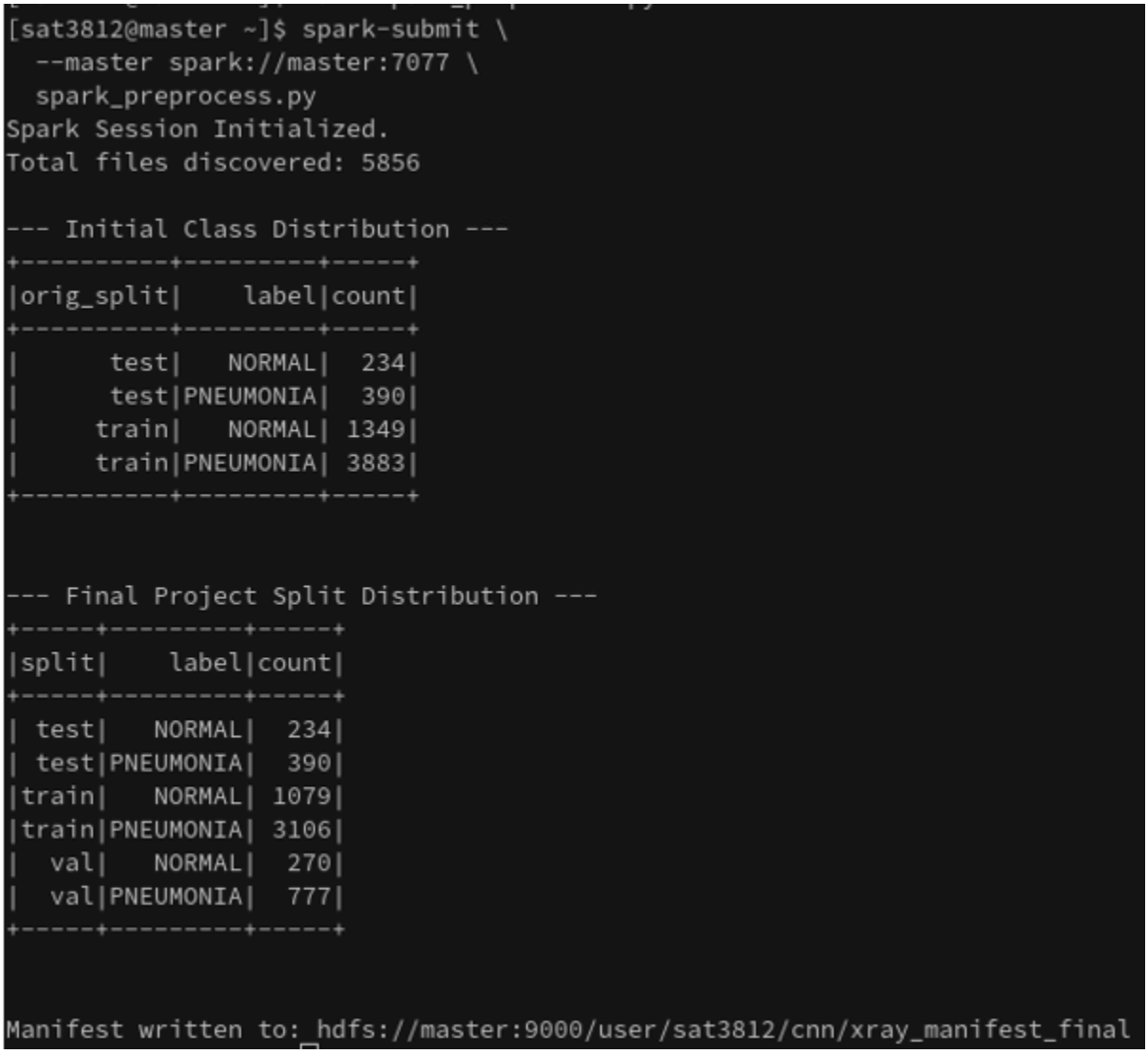
The final model achieved an overall Test Accuracy of 80.45% , with a Test Loss of 0.4222. The weighted average F1-score was 0.81, indicating a balance between precision and recall across both classes.



The results indicate a trade-off where the model sacrifices Pneumonia Recall (missing 100 cases) to achieve a high Pneumonia Precision (high certainty when it does predict the disease) and high Normal Recall (correctly identifying most healthy patients).

Additional Screen Shots:





Spark Master at spark://master:7077

URL: spark://master:7077
Alive Workers: 4
Cores in use: 32 Total, 32 Used
Memory in use: 16.0 GiB Total, 4.0 GiB Used
Resources in use:
Applications: 1 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (4)

Worker Id	Address	State	Cores	Memory	Resources
worker-20251130132208-192.168.13.201-45867	192.168.13.201:45867	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	
worker-20251130132227-192.168.13.122-35659	192.168.13.122:35659	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	
worker-20251130132442-192.168.13.200-36999	192.168.13.200:36999	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	
worker-20251207113840-192.168.13.123-37945	192.168.13.123:37945	ALIVE	8 (8 Used)	4.0 GiB (1024.0 MiB Used)	

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20251207140505-0001 (kill)	XrayEDAAndSplit	32	1024.0 MiB		2025/12/07 14:05:05	sat3812	RUNNING	1.1 min

```
Using device: cpu
Loading manifest from /home/sat3812/xray_manifest_final ...
Classes: ['NORMAL', 'PNEUMONIA']
Loaded 4185 samples for split='train'
Loaded 1047 samples for split='val'
Loaded 624 samples for split='test'
```

```
Epoch 1 | Train Loss: 0.3384 | Train Acc: 0.6616
Val Loss: 0.5350 | Val Acc: 0.8042
New best model saved (Val Acc = 0.8042)
Epoch 2 | Train Loss: 0.2873 | Train Acc: 0.7127
Val Loss: 0.9149 | Val Acc: 0.7240
Epoch 3 | Train Loss: 0.2458 | Train Acc: 0.7542
Val Loss: 0.1962 | Val Acc: 0.9169
New best model saved (Val Acc = 0.9169)
Epoch 4 | Train Loss: 0.2417 | Train Acc: 0.7583
Val Loss: 0.1626 | Val Acc: 0.9465
New best model saved (Val Acc = 0.9465)
Epoch 5 | Train Loss: 0.2331 | Train Acc: 0.7669
Val Loss: 0.2512 | Val Acc: 0.8930
```

Discussion and Conclusion:

This project demonstrates how distributed preprocessing with Apache Spark can support deep learning workflows in medical imaging, even on CPU-only virtual machines. Using Spark allowed us to efficiently traverse thousands of chest X-ray files, extract labels, shuffle the dataset, and create clean train/validation/test splits stored in HDFS. This distributed pipeline reduced the overhead of file I/O and ensured a reproducible structure for downstream training.

Our exploratory data analysis revealed that the dataset is heavily imbalanced, with pneumonia cases outnumbering normal cases by roughly 3 to 1. The images also varied significantly in resolution, confirming the need for uniform resizing before model training. Although mean pixel intensity was similar across classes, pneumonia images exhibited higher variance, suggesting more structural irregularity such as bright or opaque regions, patterns better captured by a convolutional network rather than simple intensity-based features.

The custom CNN architecture, which relied on strided convolutions for downsampling and incorporated batch normalization, dropout, and adaptive average pooling, was designed to balance representational power with computational limitations. During training, the model quickly began to overfit: validation accuracy peaked around the fourth epoch even as training accuracy continued to improve. This behavior, common in medical imaging with limited datasets, justified the use of early stopping and highlighted the importance of regularization and augmentation. The final model achieved a test accuracy of approximately 80%, indicating that it was able to learn meaningful features but still struggled with generalization, likely due to class imbalance, dataset size, and the lack of GPU resources for deeper or more expressive architectures.

Overall, this project taught us a lot about integrating distributed data processing with deep learning in a constrained compute environment. It shows that Spark can reliably handle the preprocessing demands of large imaging datasets and that even a relatively lightweight CNN can extract clinically relevant features from chest X-rays. Future work could focus on expanding the dataset, using pre-trained models, improving class balance strategies, and validating performance across more diverse imaging sources to enhance robustness and clinical applicability.