[1]:

```python
import os
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms, datasets, models
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, confusion_matrix
import seaborn as sns
from PIL import Image
from torch.cuda.amp import GradScaler, autocast
```

[2]:

```python
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

[2]:

```
'cuda'
```

[3]:

```python
train_dir = "/kaggle/input/chest-xrays-bacterial-viral-pneumonia-normal/train_images/train_images"
image_path = "/kaggle/input/chest-xrays-bacterial-viral-pneumonia-normal"
df = pd.read_csv("/kaggle/input/chest-xrays-bacterial-viral-pneumonia-normal/labels_train.csv")
```

[4]:

```python
from sklearn.model_selection import train_test_split

# Split the dataframe into train and test datasets (80% train, 20% test)
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42, stratify=df['class_id'])

# Check the size of the splits
print(f"Train set size: {len(train_df)}")
print(f"Test set size: {len(test_df)}")
```

```
Train set size: 3737
Test set size: 935
```

[5]:

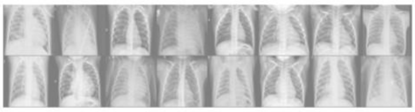
```python
class ChestXRayDataset(Dataset):
    def __init__(self, df, image_dir, transform=None):
        self.df = df
        self.image_dir = image_dir
        self.transform = transform

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        img_name = os.path.join(self.image_dir, self.df.iloc[idx, 0])  # assuming the first column is the image file name
        image = Image.open(img_name).convert('L')  # Load image in grayscale ('L' mode) -> 1 channel
        label = self.df.iloc[idx, 1]  # assuming the second column is the label

        if self.transform:
            image = self.transform(image)

        return image, label
```

[6]:

```python
transform = transforms.Compose([
    transforms.Resize((256, 256)),  # Resize to 256x256
    transforms.ToTensor(),  # Convert to tensor (will be 1 channel for grayscale)
    transforms.Normalize(mean=[0.5], std=[0.5])  # Normalize for 1 channel (grayscale)
])

# Load datasets
train_dataset = ChestXRayDataset(df=train_df, image_dir=train_dir, transform=transform)
test_dataset = ChestXRayDataset(df=test_df, image_dir=train_dir, transform=transform)

# DataLoader
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

[7]:

```
# Function to display images
def imshow(img):
    img = img * 0.5 + 0.5  # Undo normalization
    npimg = img.numpy()

    if npimg.shape[0] == 1:  # Grayscale image
        npimg = npimg.squeeze(0)  # Remove channel dimension
        plt.imshow(npimg, cmap="gray")  # Display grayscale image
    else:  # RGB image
        plt.imshow(np.transpose(npimg, (1, 2, 0)))  # Display RGB image

    plt.axis("off")  # Hide axes
    plt.show()

# Display a batch of images
dataiter = iter(train_loader)
images, labels = next(dataiter)

images, labels = images[:16], labels[:16]  # Display only the first 16 images

imshow(torchvision.utils.make_grid(images, nrow=8, padding=2, normalize=True))

# Print the labels
print("Labels:", labels.numpy())
```



```
Labels: [1 2 2 1 0 0 0 0 2 1 1 0 2 0 2 1]
```

[8]:

```
class CNNLSTM(nn.Module):
    def __init__(self, num_classes=3):
        super(CNNLSTM, self).__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(64)  # Batch Normalization
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.dropout1 = nn.Dropout(0.3)  # Dropout Layer

        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.dropout2 = nn.Dropout(0.4)  # Higher dropout after deeper layers

        # LSTM layer
        self.lstm_hidden_size = 128
        self.lstm = nn.LSTM(input_size=256, hidden_size=self.lstm_hidden_size, batch_first=True)

        # Fully connected layer
        self.fc = nn.Linear(self.lstm_hidden_size, num_classes)

    def forward(self, x):
        batch_size = x.size(0)

        # Convolutional layers
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.maxpool1(x)
        x = self.dropout1(x)

        x = F.relu(self.bn3(self.conv3(x)))
        x = self.maxpool2(x)
        x = self.dropout2(x)

        # Get the shape of the output after convolutions and pooling
        _, c, h, w = x.size()  # batch_size, channels, height, width
```

```python
        # Reshape the tensor to (batch_size, sequence_length, input_size)
        # Here, height * width is the sequence length, and the channels are the feature size
        x = x.view(batch_size, h * w, c)  # (batch_size, sequence_length, input_size)

        # Pass through LSTM
        x, _ = self.lstm(x)

        # We take the output from the last time step
        x = x[:, -1, :]

        # Fully connected layer to output class predictions
        x = self.fc(x)
        return x

# Instantiate the model
model = CNNLSTM(num_classes=3)

# Move model to device (GPU or CPU)
model.to(device)  # Ensure model is on GPU
```

```
CNNLSTM(
  (conv1): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout1): Dropout(p=0.3, inplace=False)
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout2): Dropout(p=0.4, inplace=False)
  (lstm): LSTM(256, 128, batch_first=True)
  (fc): Linear(in_features=128, out_features=3, bias=True)
)
```

[9]:

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

[10]:

```python
scaler = GradScaler()
```

```
<ipython-input-10-9f6a838c5572>:1: FutureWarning: `torch.cuda.amp.GradScaler(args...)` is deprecated. Please use `torch.amp.GradScaler('cuda', args...)` instead.
  scaler = GradScaler()
```

[13]:

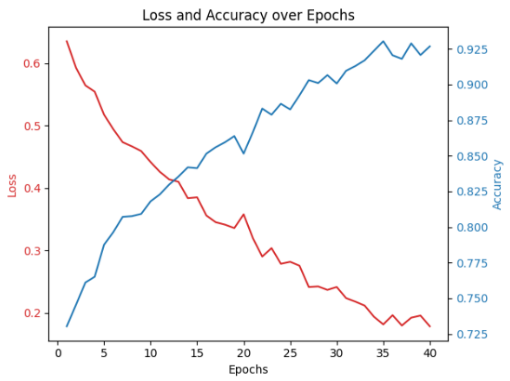
```python
num_epochs = 40

for epoch in range(num_epochs):
    print(f"\nEpoch {epoch+1}/{num_epochs}")

    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for batch_idx, (inputs, labels) in enumerate(train_loader):
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        # Enable mixed precision training
        with autocast():
            outputs = model(inputs)
            loss = criterion(outputs, labels)

        # Scale loss for mixed precision
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        running_loss += loss.item()
```

```python
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

        # Print every 10 batches to reduce log spam
        if batch_idx % 10 == 0:
            print(f"Batch {batch_idx}, Loss: {loss.item():.4f}, Accuracy: {correct / total:.4f}")

    # Compute and print epoch loss and accuracy
    epoch_loss = running_loss / len(train_loader)
    epoch_accuracy = correct / total
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Accuracy: {epoch_accuracy:.4f}")
```

```
Epoch 1/40
<ipython-input-13-5639f261be6d>:17: FutureWarning: `torch.cuda.amp.autocast(args...)` is deprecated. Please use `torch.amp.autocast('cuda', args...)` instead.
  with autocast():
Batch 0, Loss: 0.7014, Accuracy: 0.7344
Batch 10, Loss: 0.6879, Accuracy: 0.7301
Batch 20, Loss: 0.5852, Accuracy: 0.7307
Batch 30, Loss: 0.4637, Accuracy: 0.7248
Batch 40, Loss: 0.6324, Accuracy: 0.7268
Batch 50, Loss: 0.6299, Accuracy: 0.7261
Epoch 1/40, Loss: 0.6351, Accuracy: 0.7303

Epoch 2/40
Batch 0, Loss: 0.6281, Accuracy: 0.6875
Batch 10, Loss: 0.5934, Accuracy: 0.7500
Batch 20, Loss: 0.6821, Accuracy: 0.7426
Batch 30, Loss: 0.5435, Accuracy: 0.7424
Batch 40, Loss: 0.4549, Accuracy: 0.7435
Batch 50, Loss: 0.5493, Accuracy: 0.7433
Epoch 2/40, Loss: 0.5924, Accuracy: 0.7455

Epoch 3/40
Batch 0, Loss: 0.5236, Accuracy: 0.7344
Batch 10, Loss: 0.5922, Accuracy: 0.7543
Batch 20, Loss: 0.7260, Accuracy: 0.7470
Batch 30, Loss: 0.6818, Accuracy: 0.7505
Batch 40, Loss: 0.5200, Accuracy: 0.7595
Batch 50, Loss: 0.5216, Accuracy: 0.7577
Epoch 3/40, Loss: 0.5644, Accuracy: 0.7610
```

```
Batch 0, Loss: 0.2657, Accuracy: 0.9062
Batch 10, Loss: 0.1956, Accuracy: 0.9176
Batch 20, Loss: 0.2919, Accuracy: 0.9152
Batch 30, Loss: 0.2287, Accuracy: 0.9158
Batch 40, Loss: 0.1277, Accuracy: 0.9177
Batch 50, Loss: 0.2184, Accuracy: 0.9210
Epoch 36/40, Loss: 0.1963, Accuracy: 0.9205

Epoch 37/40
Batch 0, Loss: 0.1831, Accuracy: 0.9375
Batch 10, Loss: 0.2368, Accuracy: 0.9347
Batch 20, Loss: 0.1586, Accuracy: 0.9390
Batch 30, Loss: 0.1188, Accuracy: 0.9350
Batch 40, Loss: 0.2311, Accuracy: 0.9303
Batch 50, Loss: 0.3067, Accuracy: 0.9305
Epoch 37/40, Loss: 0.1876, Accuracy: 0.9310

Epoch 38/40
Batch 0, Loss: 0.1971, Accuracy: 0.9375
Batch 10, Loss: 0.1219, Accuracy: 0.9403
Batch 20, Loss: 0.1087, Accuracy: 0.9375
Batch 30, Loss: 0.0737, Accuracy: 0.9360
Batch 40, Loss: 0.1547, Accuracy: 0.9360
Batch 50, Loss: 0.1727, Accuracy: 0.9354
Epoch 38/40, Loss: 0.1713, Accuracy: 0.9347

Epoch 39/40
Batch 0, Loss: 0.1253, Accuracy: 0.9531
Batch 10, Loss: 0.1404, Accuracy: 0.9361
Batch 20, Loss: 0.1560, Accuracy: 0.9435
Batch 30, Loss: 0.2380, Accuracy: 0.9435
Batch 40, Loss: 0.2069, Accuracy: 0.9463
Batch 50, Loss: 0.0961, Accuracy: 0.9445
Epoch 39/40, Loss: 0.1600, Accuracy: 0.9414

Epoch 40/40
Batch 0, Loss: 0.2524, Accuracy: 0.8750
Batch 10, Loss: 0.1182, Accuracy: 0.9247
Batch 20, Loss: 0.1301, Accuracy: 0.9382
Batch 30, Loss: 0.2331, Accuracy: 0.9430
Batch 40, Loss: 0.1288, Accuracy: 0.9386
Batch 50, Loss: 0.1029, Accuracy: 0.9393
Epoch 40/40, Loss: 0.1609, Accuracy: 0.9393
```

```python
import matplotlib.pyplot as plt

# Sample loss and accuracy data from your provided epochs
epochs = list(range(1, 41))
loss = [
    0.6351, 0.5924, 0.5644, 0.5543, 0.5176, 0.4942, 0.4734, 0.4667, 0.4591, 0.4416,
    0.4259, 0.4138, 0.4100, 0.3837, 0.3851, 0.3559, 0.3452, 0.3413, 0.3358, 0.3579,
    0.3195, 0.2902, 0.3038, 0.2786, 0.2818, 0.2756, 0.2414, 0.2424, 0.2368, 0.2415,
    0.2237, 0.2180, 0.2117, 0.1938, 0.1815, 0.1963, 0.1797, 0.1923, 0.1957, 0.1785
]
accuracy = [
    0.7303, 0.7455, 0.7610, 0.7651, 0.7875, 0.7964, 0.8071, 0.8076, 0.8092, 0.8180,
    0.8229, 0.8298, 0.8354, 0.8419, 0.8413, 0.8515, 0.8560, 0.8595, 0.8638, 0.8515,
    0.8667, 0.8831, 0.8788, 0.8865, 0.8825, 0.8924, 0.9031, 0.9010, 0.9066, 0.9007,
    0.9096, 0.9130, 0.9170, 0.9237, 0.9304, 0.9205, 0.9180, 0.9289, 0.9207, 0.9268
]

# Create a figure with two subplots
fig, ax1 = plt.subplots()

# Plotting loss on the left y-axis
ax1.set_xlabel('Epochs')
ax1.set_ylabel('Loss', color='tab:red')
ax1.plot(epochs, loss, color='tab:red', label='Loss')
ax1.tick_params(axis='y', labelcolor='tab:red')

# Create a second y-axis to plot accuracy
ax2 = ax1.twinx()
ax2.set_ylabel('Accuracy', color='tab:blue')
ax2.plot(epochs, accuracy, color='tab:blue', label='Accuracy')
ax2.tick_params(axis='y', labelcolor='tab:blue')

# Add a title
plt.title('Loss and Accuracy over Epochs')

# Show the plot
plt.tight_layout()
plt.show()
```



[14]:

```python
def evaluate_accuracy(model, test_loader, device):
    model.eval()  # Set the model to evaluation mode
    correct = 0
    total = 0

    with torch.no_grad():  # Disable gradient calculations for inference
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            # Forward pass through the model
            outputs = model(inputs)

            # Get the predicted class (index with the highest output)
            _, predicted = torch.max(outputs, 1)

            total += labels.size(0)  # Total number of samples
            correct += (predicted == labels).sum().item()  # Count correct predictions

    # Calculate accuracy
    accuracy = correct / total * 100  # Accuracy in percentage
    return accuracy

# Example usage
test_accuracy = evaluate_accuracy(model, test_loader, device)
print(f"Test Accuracy: {test_accuracy:.2f}%")
```

```
Test Accuracy: 77.43%
```

[19]:

```python
import torch
from sklearn.metrics import classification_report

def generate_classification_report(model, test_loader, device):
    model.eval()  # Set the model to evaluation mode

    all_preds = []
    all_labels = []

    with torch.no_grad():  # Disable gradient calculation during inference
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            # Forward pass
            outputs = model(inputs)

            # Get the predicted class (index with the highest output)
            _, predicted = torch.max(outputs, 1)

            # Store the predicted and true labels
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Generate classification report
    report = classification_report(all_labels, all_preds, target_names=['Class 0', 'Class 1', 'Class 2'], output_dict=True)

    return report, all_labels, all_preds

# Example usage
classification_report_result, all_labels, all_preds = generate_classification_report(model, test_loader, device)

# Print the classification report
print("Classification Report:")
print(classification_report_result)

# If you want to format it as a nice string report, do this:
formatted_report = classification_report(all_labels, all_preds, target_names=['Class 0', 'Class 1', 'Class 2'])
print("\nFormatted Classification Report:")
print(formatted_report)
```

```
Classification Report:
{'Class 0': {'precision': 0.9087136929460581, 'recall': 0.8902439024390244, 'f1-score': 0.8993839835728953, 'support': 246}, 'Class 1': {'precision': 0.8093023255813954, 'recall':
0.7767857142857143, 'f1-score': 0.7927107061503417, 'support': 448}, 'Class 2': {'precision': 0.5946969696969697, 'recall': 0.6514522821576764, 'f1-score': 0.6217821782178219, 'supp
ort': 241}, 'accuracy': 0.774331550802139, 'macro avg': {'precision': 0.770904329408141, 'recall': 0.7728272996274717, 'f1-score': 0.7712922893136863, 'support': 935}, 'weighted av
g': {'precision': 0.780142224622636, 'recall': 0.774331550802139, 'f1-score': 0.7767191029569844, 'support': 935}}

Formatted Classification Report:
              precision    recall  f1-score   support

     Class 0       0.91      0.89      0.90       246
     Class 1       0.81      0.78      0.79       448
     Class 2       0.59      0.65      0.62       241

    accuracy                           0.77       935
   macro avg       0.77      0.77      0.77       935
weighted avg       0.78      0.77      0.78       935
```

[16]:

```python
# To save the model to kaggle working Directory
model_path = "/kaggle/working/cnn_lstm_model_full.pth"

# Save the entire model
torch.save(model, model_path)

print(f"Entire model saved to {model_path}")
```

```
Entire model saved to /kaggle/working/cnn_lstm_model_full.pth
```

[ ]:

```python
# To load the entire model (architecture + parameters)
model = torch.load("/kaggle/working/cnn_lstm_model_full.pth")

# Move the model to the device (CPU or GPU)
model.to(device)

# Set the model to evaluation mode
model.eval()

print("Entire model loaded successfully!")
```