

```
from IPython.display import Image, display
display(Image(filename='Lecture 3.png'))
```



MIDOCEAN
UNIVERSITY

Training and Evaluating CNN Models Using PyTorch

Lecture 03

Prepared by

Dr. Hager Saleh

```
import os
import numpy as np
import torch
from torchvision import datasets, transforms as T, models
from torch import nn, optim
from torch.utils.data import DataLoader
import warnings
warnings.filterwarnings("ignore")
from tqdm import tqdm
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score, classification_report
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import roc_curve
import matplotlib.pyplot as plt
from sklearn.metrics import auc
```

What is PyTorch?

PyTorch is an open-source machine learning framework developed by Facebook's AI Research lab (FAIR). It is widely used for:

- Deep learning
- Computer vision
- Natural language processing (NLP)
- Time series modeling
- Scientific research

PyTorch provides a flexible and dynamic platform to build, train, and deploy deep neural networks, especially popular in research and production.

Image augmentation and preprocessing in Pytorch

T.Compose is a method from torchvision.transforms that is used to chain multiple image transformations together. This is commonly used in PyTorch image preprocessing pipelines.

T.RandomHorizontalFlip()

Randomly flips the image horizontally with a default probability of 0.5.

T.RandomVerticalFlip()

Randomly flips the image vertically with a default probability of 0.5.

T.RandomRotation(10)

Rotates the image randomly within a ± 10 degree range.

T.Resize((299, 299))

Resizes the image to 299×299 pixels. This is often used to match the input size expected by models like InceptionV3.

T.ToTensor()

Converts a PIL Image or NumPy ndarray into a PyTorch Tensor.

Scales pixel values from [0, 255] to [0.0, 1.0].

T.Normalize(mean, std)

Normalizes the image tensor using the specified mean and standard deviation for each RGB channel.

These values are typically used for models pretrained on ImageNet.

What is DataLoader in PyTorch?

In PyTorch, DataLoader is a utility class provided by `torch.utils.data` that loads data efficiently and feeds it into your model during training or evaluation.

It's commonly used to:

- Load data in mini-batches
- Shuffle the dataset
- Enable parallel data loading using multiple worker processes
- Automatically handle batching, shuffling, and sampling

Download data from Kaggle

- Create a Kaggle Account (if you don't have one): Go to <https://www.kaggle.com> and sign up/log in.
- Go this link: <https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia> and download the dataset
- This dataset is about Chest X-Ray Images (Pneumonia) consists of two classes: Benign, Malignant, and Normal.

Image augmentation and preprocessing

```
def get_classes(data_dir):  
    """Get class names from dataset directory."""  
    all_data = datasets.ImageFolder(data_dir)  
    return all_data.classes  
  
def get_data_loaders(data_dir, batch_size, train=False):  
    """Prepare data loaders for training, validation, and testing."""  
    if train:  
        # Data augmentation for training  
        transform = T.Compose([  
            T.RandomHorizontalFlip(),  
            T.RandomVerticalFlip(),  
            T.RandomRotation(10),  
            T.Resize((299, 299)),  
            T.ToTensor(),  
            T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,  
0.225])  
        ])  
        train_data = datasets.ImageFolder(os.path.join(data_dir,  
"train/"), transform=transform)  
        train_loader = DataLoader(train_data, batch_size=batch_size,  
shuffle=True, num_workers=4)  
        return train_loader, len(train_data)  
    else:  
        # Validation and test data transformations  
        transform = T.Compose([
```

```

        T.Resize((299, 299)),
        T.ToTensor(),
        T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]))
    ])
    val_data = datasets.ImageFolder(os.path.join(data_dir,
"val/"), transform=transform)
    test_data = datasets.ImageFolder(os.path.join(data_dir,
"test/"), transform=transform)
    val_loader = DataLoader(val_data, batch_size=batch_size,
shuffle=False, num_workers=4)
    test_loader = DataLoader(test_data, batch_size=batch_size,
shuffle=False, num_workers=4)
    return val_loader, test_loader, len(val_data), len(test_data)

# Dataset paths and loaders
dataset_path = "chest_xray"
batch_size = 32

(train_loader, train_data_len) = get_data_loaders(dataset_path,
batch_size, train=True)
(val_loader, test_loader, valid_data_len, test_data_len) =
get_data_loaders(dataset_path, batch_size, train=False)

classes = get_classes("chest_xray/train")
num_classes = len(classes)

dataloaders = {"train": train_loader, "val": val_loader}
dataset_sizes = {"train": train_data_len, "val": valid_data_len}

# Device setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

```

efficientnet_b0

```

# Load the pre-trained Inception-v3 model
model = models.efficientnet_b0(pretrained=True)

# Update the fully connected layer for the number of classes
model.classifier[1] = nn.Linear(model.classifier[1].in_features,
num_classes)
model = model.to(device)

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler

```

```
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.97)
```

Downloading:

```
"https://download.pytorch.org/models/efficientnet_b0_rwightman-7f5810bc.pth" to C:\Users\compumarts/.cache\torch\hub\checkpoints\efficientnet_b0_rwightman-7f5810bc.pth
100%|██████████| 20.5M/20.5M [00:00<00:00, 22.2MB/s]
```

```
def train_model(model, criterion, optimizer, scheduler,
num_epochs=20):
    """Train and validate the model."""
    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train() # Set model to training mode
            else:
                model.eval() # Set model to evaluation mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward pass
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)

                    # Handle auxiliary logits for Inception-v3
                    if isinstance(outputs, tuple) or hasattr(outputs,
"logits"):
                        main_outputs = outputs.logits if
hasattr(outputs, "logits") else outputs[0]
                        aux_outputs = outputs.aux_logits if
hasattr(outputs, "aux_logits") else None
                        loss = criterion(main_outputs, labels)
                        if aux_outputs is not None: # Add auxiliary
loss if present
                            loss += 0.4 * criterion(aux_outputs,
labels)
```

```

        else:
            main_outputs = outputs
            loss = criterion(main_outputs, labels)

            _, preds = torch.max(main_outputs, 1) # Extract
            predictions from logits

            # Backward pass and optimization in training phase
            if phase == 'train':
                loss.backward()
                optimizer.step()

            # Statistics
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

            # Step the scheduler in the training phase
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() /
dataset_sizes[phase]

            print(f"{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}")

        return model

```

Train the model

```

model = train_model(model, criterion, optimizer, exp_lr_scheduler,
num_epochs=2)

```

Epoch 1/2

train Loss: 0.1356 Acc: 0.9523

val Loss: 0.2557 Acc: 0.8125

Epoch 2/2

train Loss: 0.0787 Acc: 0.9712

val Loss: 0.2781 Acc: 0.8125

```

def evaluate_model(model_name, y_true, y_pred, y_proba):

```

```

    labels=['NORMAL', 'PNEUMONIA']

```

```

    accuracy = accuracy_score(y_true, y_pred)

```

```

    precision = precision_score(y_true, y_pred, average='weighted')

```

```

    recall = recall_score(y_true, y_pred, average='weighted')

```

```

f1 = f1_score(y_true, y_pred, average='weighted')
cm = confusion_matrix(y_true, y_pred)

print("Accuracy:", accuracy)
print("Precision :", precision)
print("Recall:", recall)
print("F1-score:" , f1)

print("Classification Report:")
print(classification_report(y_true, y_pred))

# Plot confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=labels, yticklabels=labels)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# If binary classification, take probabilities of the positive
class
if y_proba.shape[1] == 2: # For binary classification
    y_proba = y_proba[:, 1] # Take probabilities for the positive
class

# Calculate ROC curve
fpr, tpr, thresholds = roc_curve(y_true, y_proba)
roc_auc = auc(fpr, tpr)

# Plot ROC curve
plt.figure()
plt.plot(fpr, tpr, color='blue', lw=2, label='ROC curve (area =
{:.2f})'.format(roc_auc))
plt.plot([0, 1], [0, 1], color='red', lw=2, linestyle='--') #
Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc='lower right')
plt.grid()
plt.show()

y_true = []
y_proba = []

```

```

y_pred = []

# Disable gradient calculation
with torch.no_grad():
    for x, y in tqdm(test_loader):
        # Forward pass
        output = model(x.to(device))

        # Store true labels
        y_true.extend(y.numpy())

        # Get predicted probabilities (softmax output)
        probabilities = torch.softmax(output, dim=1) # Assuming it's
a multi-class classification
        y_proba.extend(probabilities.cpu().numpy()) # Store
probabilities

        # Get predicted classes (argmax for final class prediction)
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
        y_pred.extend(pred)

# Convert to numpy arrays for further processing
y_true = np.array(y_true)
y_proba = np.array(y_proba)
y_pred = np.array(y_pred)

```

100%|██████████| 20/20 [00:41<00:00, 2.06s/it]

evaluate_model("efficientnet_b0", y_true, y_pred, y_proba)

Accuracy: 0.8685897435897436

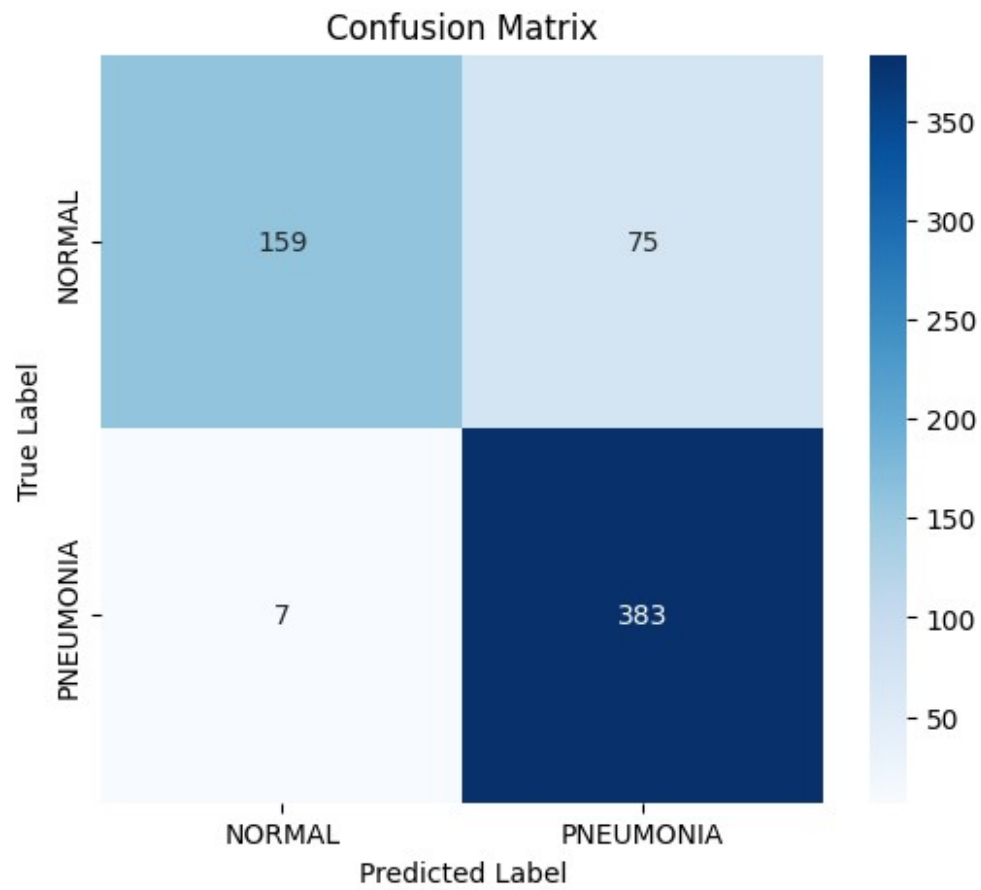
Precision : 0.8818395854158994

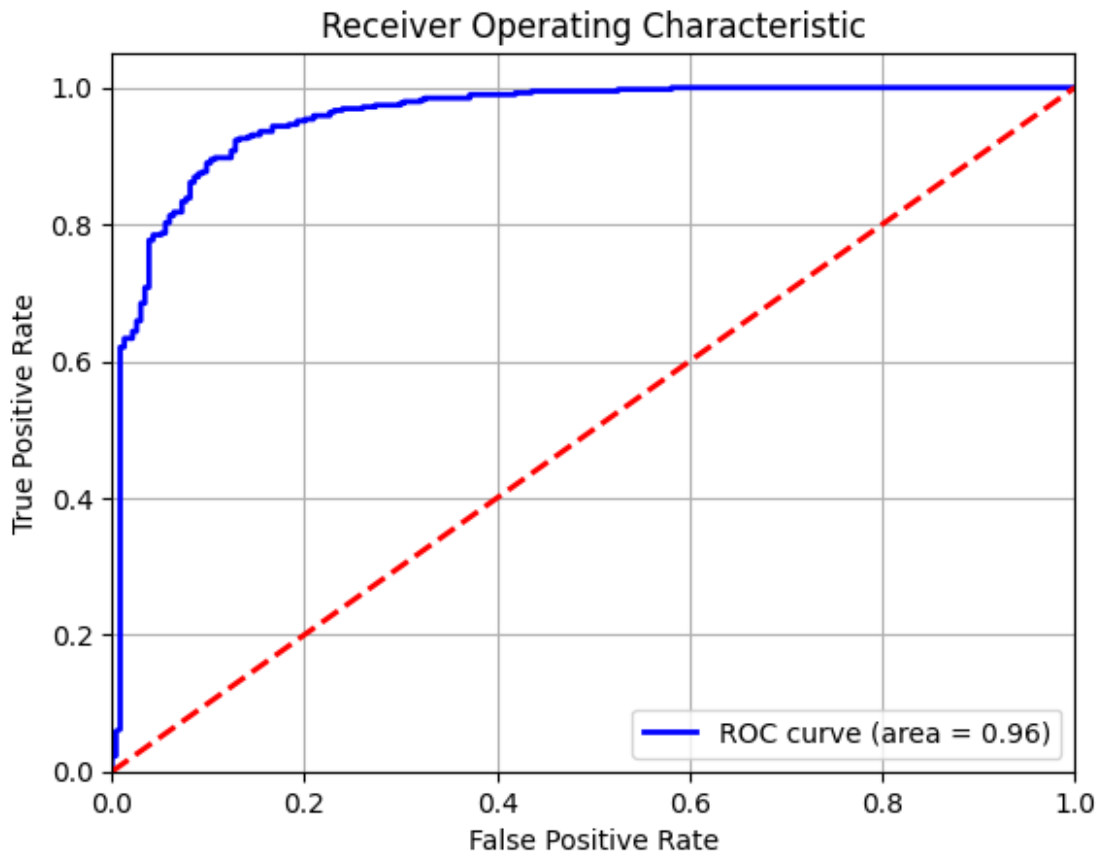
Recall: 0.8685897435897436

F1-score: 0.862688679245283

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.68	0.80	234
1	0.84	0.98	0.90	390
accuracy			0.87	624
macro avg	0.90	0.83	0.85	624
weighted avg	0.88	0.87	0.86	624





Resnet18

```
from torchvision import models
import torch.nn as nn
import torch.optim as optim

# Initialize ResNet-18 from scratch (no pretrained weights)
model = models.resnet18(pretrained=False)

# Replace the final fully connected layer to match the number of
# output classes
model.fc = nn.Linear(model.fc.in_features, num_classes)
model = model.to(device)

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.97)
```

```

model = train_model(model, criterion, optimizer, exp_lr_scheduler,
num_epochs=2)

Epoch 1/2
-----
train Loss: 0.2827 Acc: 0.8829
val Loss: 1.0846 Acc: 0.5625
Epoch 2/2
-----
train Loss: 0.1807 Acc: 0.9291
val Loss: 2.3690 Acc: 0.5625

y_true = []
y_proba = []
y_pred = []

# Disable gradient calculation
with torch.no_grad():
    for x, y in tqdm(test_loader):
        # Forward pass
        output = model(x.to(device))

        # Store true labels
        y_true.extend(y.numpy())

        # Get predicted probabilities (softmax output)
        probabilities = torch.softmax(output, dim=1) # Assuming it's
a multi-class classification
        y_proba.extend(probabilities.cpu().numpy()) # Store
probabilities

        # Get predicted classes (argmax for final class prediction)
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
        y_pred.extend(pred)

# Convert to numpy arrays for further processing
y_true = np.array(y_true)
y_proba = np.array(y_proba)
y_pred = np.array(y_pred)

evaluate_model("Resnet18", y_true, y_pred, y_proba)

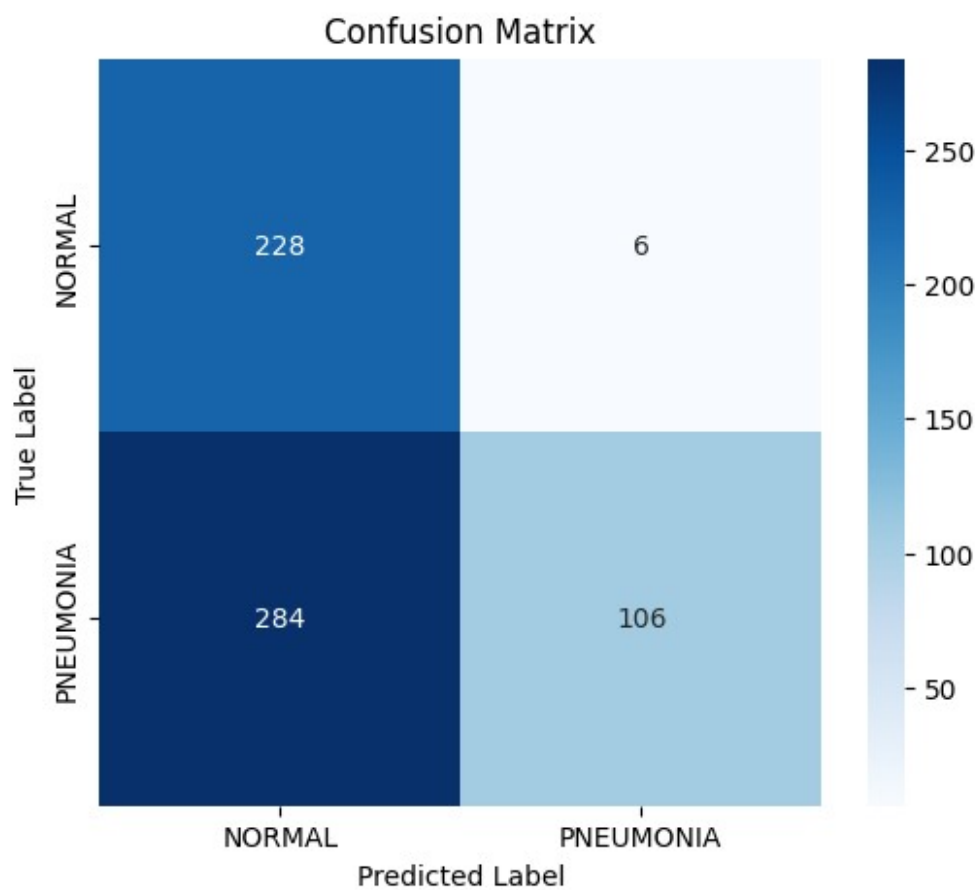
100%|██████████| 20/20 [00:15<00:00, 1.30it/s]

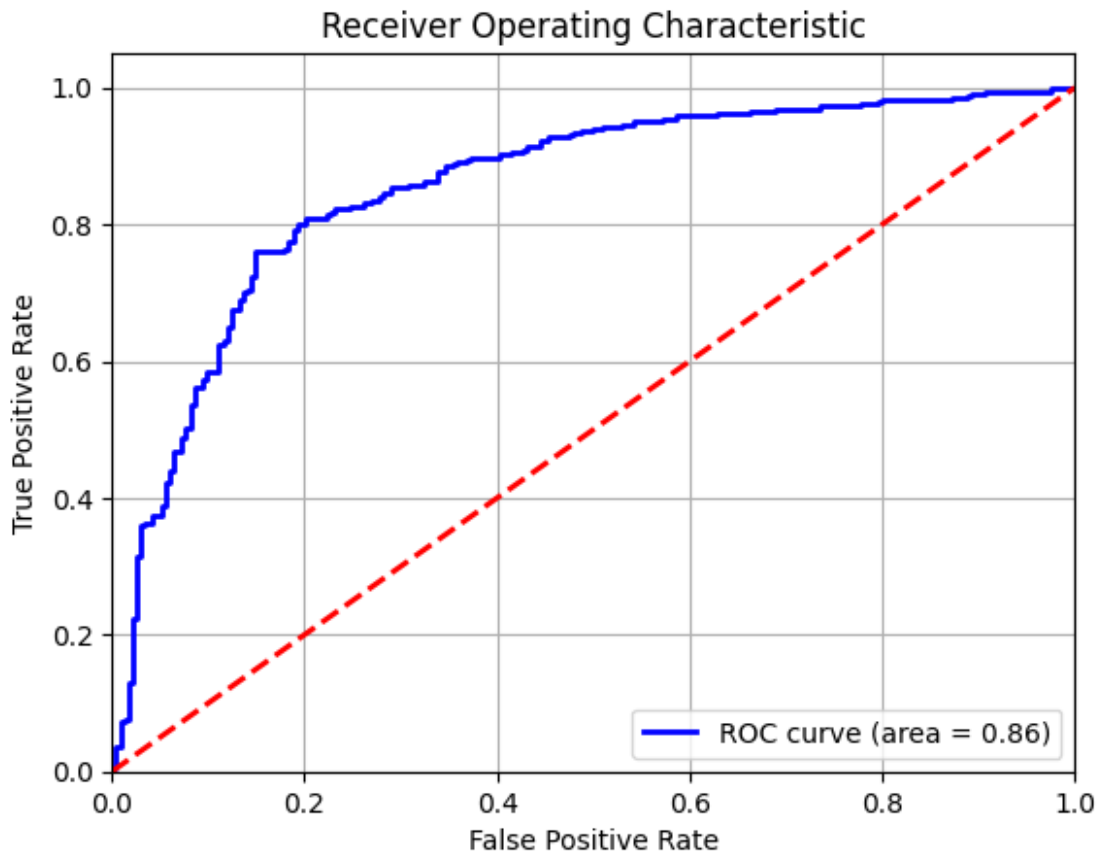
Accuracy: 0.5352564102564102
Precision : 0.7585100446428571
Recall: 0.5352564102564102
F1-score: 0.49316674321480836
Classification Report:

```

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

	0	0.45	0.97	0.61	234
	1	0.95	0.27	0.42	390
accuracy				0.54	624
macro avg		0.70	0.62	0.52	624
weighted avg		0.76	0.54	0.49	624





Densenet121

```
model = models.densenet121(pretrained=False)

# Replace the classifier to match the number of output classes
model.classifier = nn.Linear(model.classifier.in_features,
num_classes)
model = model.to(device)

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.97)

model = train_model(model, criterion, optimizer, exp_lr_scheduler,
num_epochs=1)

Epoch 1/1
-----
```

```
train Loss: 0.2449 Acc: 0.9015
val Loss: 0.9264 Acc: 0.5000
```

```
y_true = []
y_proba = []
y_pred = []
```

```
# Disable gradient calculation
```

```
with torch.no_grad():
```

```
    for x, y in tqdm(test_loader):
```

```
        # Forward pass
```

```
        output = model(x.to(device))
```

```
        # Store true labels
```

```
        y_true.extend(y.numpy())
```

```
        # Get predicted probabilities (softmax output)
```

```
        probabilities = torch.softmax(output, dim=1) # Assuming it's
```

```
a multi-class classification
```

```
        y_proba.extend(probabilities.cpu().numpy()) # Store
probabilities
```

```
        # Get predicted classes (argmax for final class prediction)
```

```
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
```

```
        y_pred.extend(pred)
```

```
# Convert to numpy arrays for further processing
```

```
y_true = np.array(y_true)
```

```
y_proba = np.array(y_proba)
```

```
y_pred = np.array(y_pred)
```

```
evaluate_model("Densenet121", y_true, y_pred, y_proba)
```

```
100%|██████████| 20/20 [00:38<00:00, 1.93s/it]
```

```
Accuracy: 0.8413461538461539
```

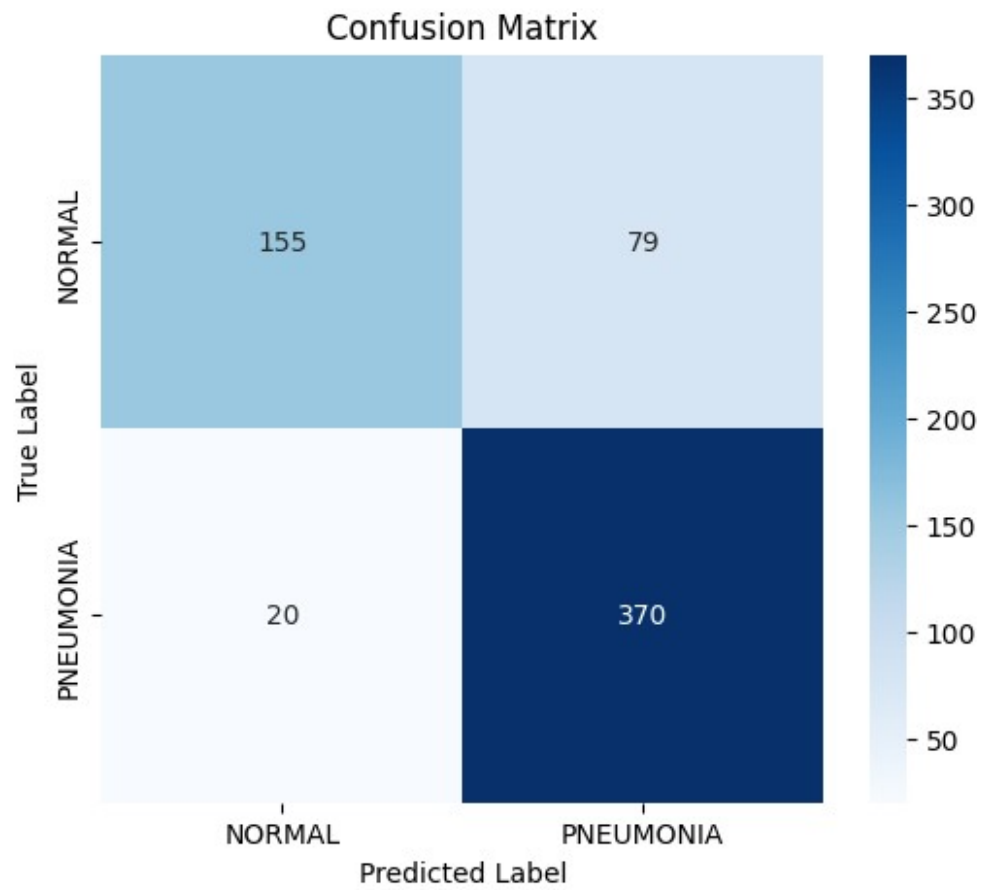
```
Precision : 0.8471762647152402
```

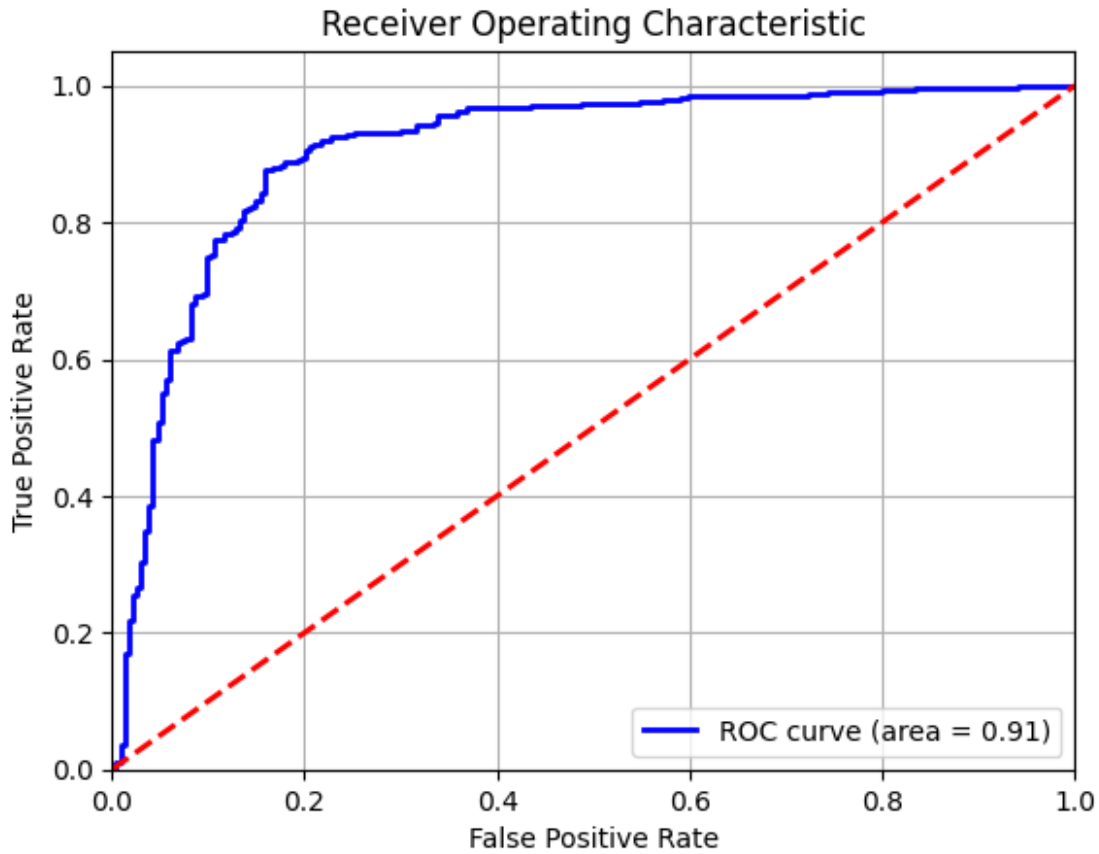
```
Recall: 0.8413461538461539
```

```
F1-score: 0.8354813187197473
```

```
Classification Report:
```

	precision	recall	f1-score	support
0	0.89	0.66	0.76	234
1	0.82	0.95	0.88	390
accuracy			0.84	624
macro avg	0.85	0.81	0.82	624
weighted avg	0.85	0.84	0.84	624





vgg16

```
# Initialize VGG16 from scratch (no pretrained weights)
model = models.vgg16(pretrained=False)

# Replace the classifier to match the number of output classes
model.classifier[6] = nn.Linear(model.classifier[6].in_features,
num_classes)
model = model.to(device)

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Learning rate scheduler
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.97)

model = train_model(model, criterion, optimizer, exp_lr_scheduler,
num_epochs=1)
```

```

Epoch 1/1
-----
train Loss: 3.0593 Acc: 0.7327
val Loss: 0.8355 Acc: 0.5000

y_true = []
y_proba = []
y_pred = []

# Disable gradient calculation
with torch.no_grad():
    for x, y in tqdm(test_loader):
        # Forward pass
        output = model(x.to(device))

        # Store true labels
        y_true.extend(y.numpy())

        # Get predicted probabilities (softmax output)
        probabilities = torch.softmax(output, dim=1) # Assuming it's
a multi-class classification
        y_proba.extend(probabilities.cpu().numpy()) # Store
probabilities

        # Get predicted classes (argmax for final class prediction)
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
        y_pred.extend(pred)

# Convert to numpy arrays for further processing
y_true = np.array(y_true)
y_proba = np.array(y_proba)
y_pred = np.array(y_pred)

evaluate_model("vgg16", y_true, y_pred, y_proba)

100%|██████████| 20/20 [01:06<00:00, 3.34s/it]

Accuracy: 0.625
Precision : 0.390625
Recall: 0.625
F1-score: 0.4807692307692308
Classification Report:

```

	precision	recall	f1-score	support
0	0.00	0.00	0.00	234
1	0.62	1.00	0.77	390
accuracy			0.62	624
macro avg	0.31	0.50	0.38	624
weighted avg	0.39	0.62	0.48	624

