

```
from IPython.display import Image, display
display(Image(filename='4.png'))
```



MIDOCEAN
UNIVERSITY

Reducing Images Size and Training Models

Lecture 04

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
```

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/gunavenkatdoddi/eye-diseases-classification/data> and download the dataset

Reduce image size

```
from PIL import Image
import os

input_root = 'dataset/'
output_root = 'output/'
resize_size = (256, 256) # Change size as needed

# Create output root if it doesn't exist
os.makedirs(output_root, exist_ok=True)

# Iterate through each class folder
for class_name in os.listdir(input_root):
    class_path = os.path.join(input_root, class_name)

    # Skip if it's not a directory or if it's the 'masks' folder
    if not os.path.isdir(class_path) or class_name == 'masks':
        continue

    # Create output path for each class
    output_class_path = os.path.join(output_root, class_name)
    os.makedirs(output_class_path, exist_ok=True)

    for image_name in os.listdir(class_path):
        if image_name.lower().endswith(('.png', '.jpg', '.jpeg',
        '.bmp', '.tiff')):
            input_image_path = os.path.join(class_path, image_name)
            output_image_name = os.path.splitext(image_name)[0] +
            '.jpg'
            output_image_path = os.path.join(output_class_path,
            output_image_name)

            try:
                with Image.open(input_image_path) as img:
                    img = img.convert('RGB')
                    img = img.resize(resize_size)
                    img.save(output_image_path, 'JPEG')
                    print(f"Processed: {input_image_path} ->
                    {output_image_path}")
            except Exception as e:
                print(f"Failed to process {input_image_path}: {e}")
```

- The original dataset size is 750 MB
- After reduce size is 25MB

Splitting dataset

```
import os, shutil
from sklearn.model_selection import train_test_split
```

```

original_dataset_dir = 'output'
base_dir = 'new dataset'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

for dir in [train_dir, validation_dir, test_dir]:
    os.makedirs(dir, exist_ok=True)

train_ratio, validation_ratio, test_ratio = 0.7, 0.1, 0.2

def split_and_copy_files(class_dir, dest_base_dir):
    all_images = [os.path.join(class_dir, f) for f in
os.listdir(class_dir) if os.path.isfile(os.path.join(class_dir, f))]
    train_images, temp_images = train_test_split(all_images,
test_size=(1 - train_ratio), random_state=42)
    validation_images, test_images = train_test_split(temp_images,
test_size=test_ratio/(validation_ratio + test_ratio), random_state=42)

    for images, folder_name in zip([train_images, validation_images,
test_images], ['train', 'validation', 'test']):
        dest_dir = os.path.join(dest_base_dir, folder_name,
os.path.basename(class_dir))
        os.makedirs(dest_dir, exist_ok=True)
        for file in images:
            shutil.copy(file, dest_dir)

# Apply to all class subfolders
for class_dir in os.listdir(original_dataset_dir):
    full_class_dir = os.path.join(original_dataset_dir, class_dir)
    if os.path.isdir(full_class_dir):
        split_and_copy_files(full_class_dir, base_dir)

```

Getting name of classes in dataset

```

def get_classes(data_dir):
    all_data = datasets.ImageFolder(data_dir)
    return all_data.classes

classes = get_classes("new dataset/test")
print(classes, len(classes))

['cataract', 'diabetic_retinopathy', 'glaucoma', 'normal'] 4

```

Image augmentation and preprocessing

```

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,
"validation/"), 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 = "new dataset"
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("new dataset/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)

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: 0.5239 Acc: 0.8091

val Loss: 0.3973 Acc: 0.8531

Epoch 2/2

```
train Loss: 0.3263 Acc: 0.8806
```

```
val Loss: 0.2715 Acc: 0.9052
```

```
def evaluate_model(model_name, y_true, y_pred, classes):

    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=classes, yticklabels=classes)
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title("Confusion Matrix")
    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%|██████████| 27/27 [00:01<00:00, 16.46it/s]

```
evaluate_model("efficientnet_b0", y_true, y_pred, classes)
```

Accuracy: 0.8971631205673759

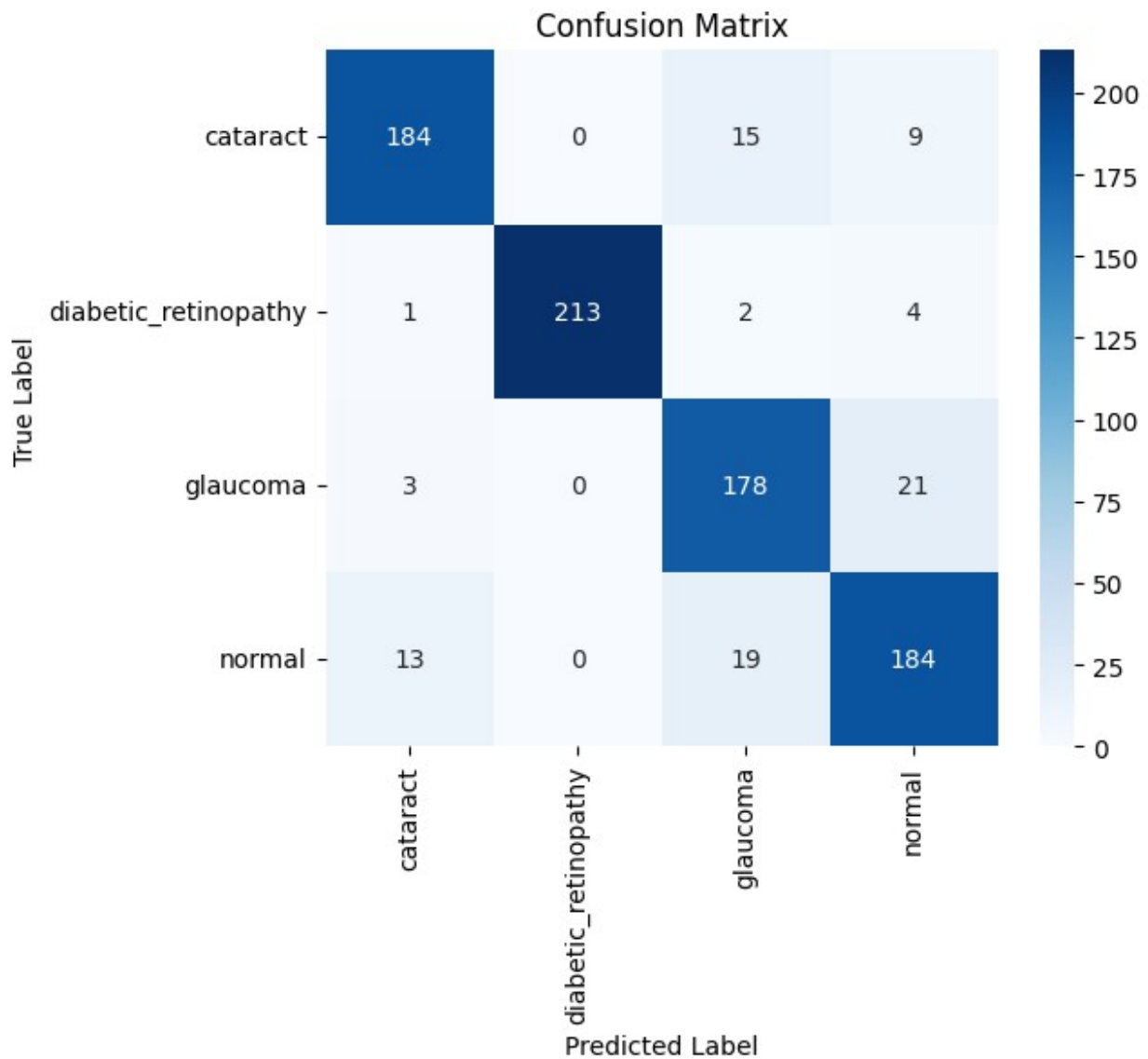
Precision : 0.8992181777698517

Recall: 0.8971631205673759

F1-score: 0.897884185125578

Classification Report:

	precision	recall	f1-score	support
0	0.92	0.88	0.90	208
1	1.00	0.97	0.98	220
2	0.83	0.88	0.86	202
3	0.84	0.85	0.85	216
accuracy			0.90	846
macro avg	0.90	0.90	0.90	846
weighted avg	0.90	0.90	0.90	846



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: 1.1432 Acc: 0.4934
val Loss: 1.3655 Acc: 0.4408
Epoch 2/2
-----
train Loss: 0.8761 Acc: 0.5931
val Loss: 1.6173 Acc: 0.4573

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, classes)

100%|██████████| 27/27 [00:01<00:00, 26.78it/s]

```

Accuracy: 0.48108747044917255

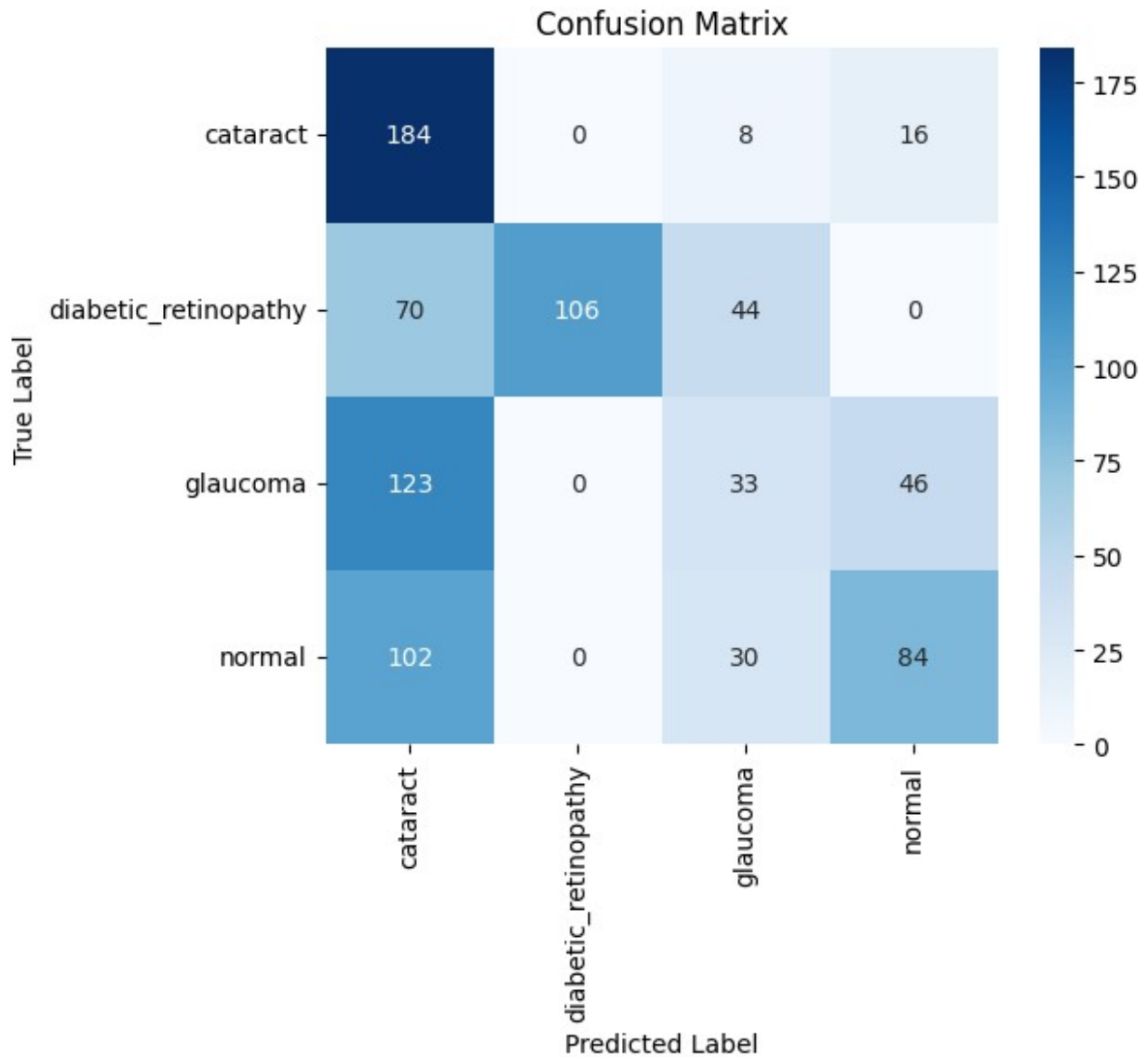
Precision : 0.5699042331273958

Recall: 0.48108747044917255

F1-score: 0.4690131433680105

Classification Report:

	precision	recall	f1-score	support
0	0.38	0.88	0.54	208
1	1.00	0.48	0.65	220
2	0.29	0.16	0.21	202
3	0.58	0.39	0.46	216
accuracy			0.48	846
macro avg	0.56	0.48	0.46	846
weighted avg	0.57	0.48	0.47	846



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: 1.0974 Acc: 0.6758
val Loss: 0.8770 Acc: 0.6280

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_pred = np.array(y_pred)

evaluate_model("Densenet121", y_true, y_pred, classes)

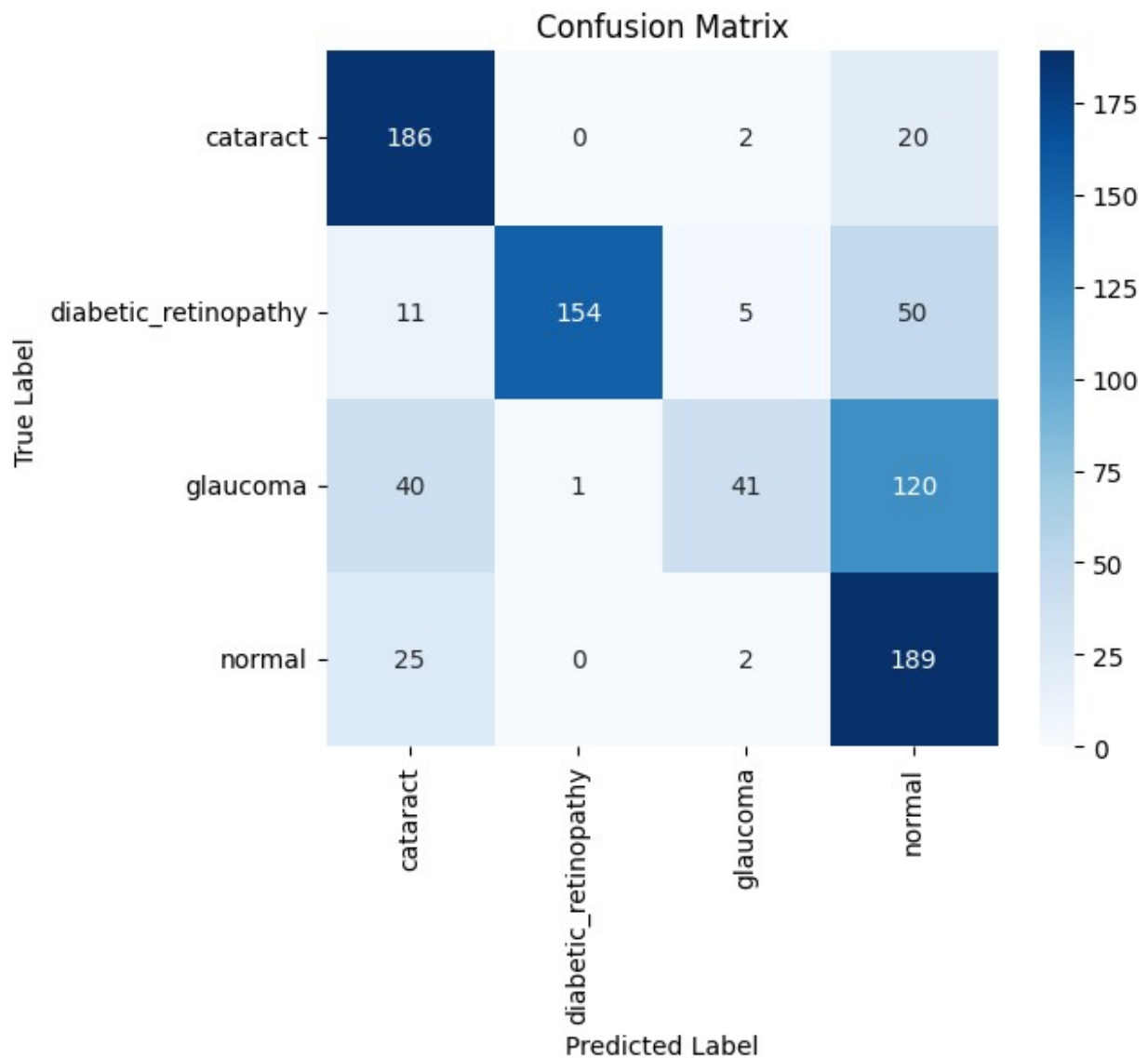
100%|██████████| 27/27 [00:01<00:00, 14.95it/s]

Accuracy: 0.6737588652482269
Precision : 0.7560281405159944
Recall: 0.6737588652482269
F1-score: 0.6480813341717486
Classification Report:

```

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

0	0.71	0.89	0.79	208
1	0.99	0.70	0.82	220
2	0.82	0.20	0.33	202
3	0.50	0.88	0.64	216
accuracy			0.67	846
macro avg	0.76	0.67	0.64	846
weighted avg	0.76	0.67	0.65	846



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: 6.1135 Acc: 0.2597
val Loss: 1.3860 Acc: 0.2607

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_pred = np.array(y_pred)

```

```
evaluate_model("vgg16", y_true, y_pred, classes)
```

```
100%|██████████| 27/27 [01:26<00:00, 3.19s/it]
```

Accuracy: 0.26004728132387706

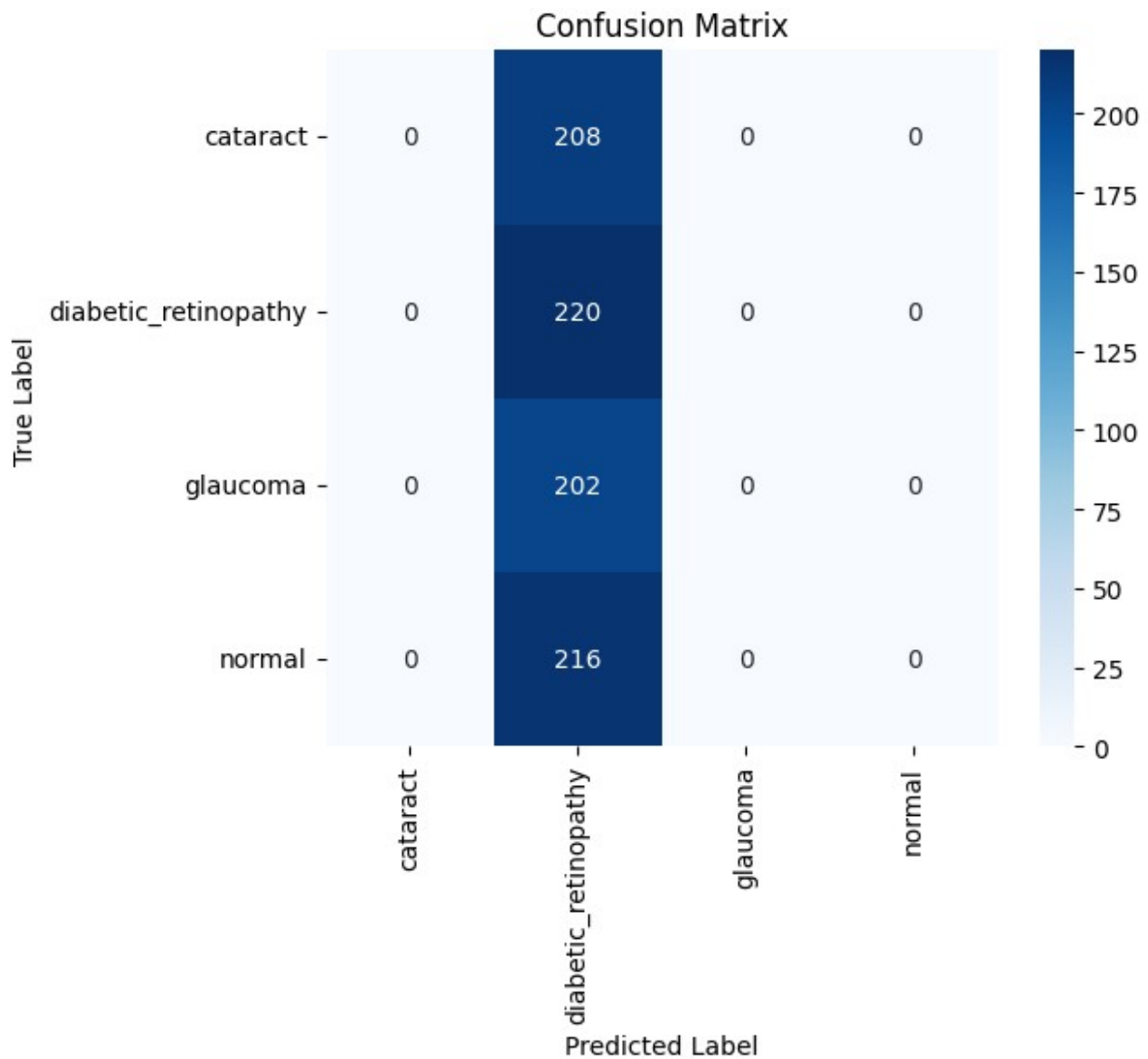
Precision : 0.06762458852393965

Recall: 0.26004728132387706

F1-score: 0.10733658891417064

Classification Report:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	208
1	0.26	1.00	0.41	220
2	0.00	0.00	0.00	202
3	0.00	0.00	0.00	216
accuracy			0.26	846
macro avg	0.07	0.25	0.10	846
weighted avg	0.07	0.26	0.11	846



```
# Load InceptionV3 model without pretraining
model = models.inception_v3(pretrained=False, aux_logits=True)

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

# Also replace the auxiliary classifier's fully connected layer (if
# used)
if model.aux_logits:
    model.AuxLogits.fc = nn.Linear(model.AuxLogits.fc.in_features,
    num_classes)

model = model.to(device)
```

```

# Loss function and optimizer
criterion = nn.CrossEntropyLoss().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: 1.5465 Acc: 0.5175
val Loss: 1.1176 Acc: 0.5261

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_pred = np.array(y_pred)

evaluate_model("inception_v3", y_true, y_pred, classes)

100%|██████████| 27/27 [00:01<00:00, 13.83it/s]

Accuracy: 0.5295508274231678
Precision : 0.518034187041779
Recall: 0.5295508274231678

```

F1-score: 0.48800395431167026

Classification Report:

	precision	recall	f1-score	support
0	0.61	0.77	0.68	208
1	0.56	0.80	0.66	220
2	0.51	0.12	0.20	202
3	0.39	0.40	0.40	216
accuracy			0.53	846
macro avg	0.52	0.52	0.48	846
weighted avg	0.52	0.53	0.49	846

