

```
In [1]: 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_s
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
```

```
In [ ]: from PIL import Image
import os

input_root = 'Downloads/dataset/'
output_root = 'Downloads/output/'
resize_size = (256, 256)

os.makedirs(output_root, exist_ok=True)

for class_name in os.listdir(input_root):
    class_path = os.path.join(input_root, class_name)
    if not os.path.isdir(class_path) or class_name == 'masks':
        continue
    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',
            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_im
        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_imag
        except Exception as e:
            print(f"Failed to process {input_image_path}: {e}")
```

```
In [3]: import os, shutil
from sklearn.model_selection import train_test_split
original_dataset_dir = 'Downloads/output'
base_dir = 'Downloads/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_di
    train_images, temp_images = train_test_split(all_images, test_size=(1
```

```

validation_images, test_images = train_test_split(temp_images, test_s
for images, folder_name in zip([train_images, validation_images, test
    dest_dir = os.path.join(dest_base_dir, folder_name, os.path.basen
    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)

```

```

In [4]: import os
for split in ['train', 'validation', 'test']:
    print(f"=== {split} ===")
    split_path = os.path.join('Downloads/new dataset', split)
    for c in os.listdir(split_path):
        cpath = os.path.join(split_path, c)
        if os.path.isdir(cpath):
            print(f" {c}: {len(os.listdir(cpath))}")

```

```

=== train ===
cataract: 726
glaucoma: 704
diabetic_retinopathy: 768
normal: 751
=== validation ===
cataract: 104
glaucoma: 101
diabetic_retinopathy: 110
normal: 107
=== test ===
cataract: 208
glaucoma: 202
diabetic_retinopathy: 220
normal: 216

```

```

In [7]: def get_classes(data_dir):
        all_data = datasets.ImageFolder(data_dir)
        return all_data.classes
classes = get_classes("Downloads/new dataset/test")
print(classes, len(classes))

```

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

```

In [13]: 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.
            ])
            train_data = datasets.ImageFolder(os.path.join(data_dir, "train/")
            train_loader = DataLoader(train_data, batch_size=batch_size, shuf
            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.
    ])
    val_data = datasets.ImageFolder(os.path.join(data_dir, "validation
    test_data = datasets.ImageFolder(os.path.join(data_dir, "test/"),
    val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=
    test_loader = DataLoader(test_data, batch_size=batch_size, shuffl
    return val_loader, test_loader, len(val_data), len(test_data)

# Dataset paths and loaders
dataset_path = "Downloads/new dataset"
batch_size = 32
(train_loader, train_data_len) = get_data_loaders(dataset_path, batch_siz
(val_loader, test_loader, valid_data_len, test_data_len) = get_data_loade
classes = get_classes("Downloads/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('mps' if torch.backends.mps.is_available() else 'cp

```

```

In [20]: # Load the pre-trained 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_clas
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, gamm
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, "lo
                        main_outputs = outputs.logits if hasattr(outputs,

```

```

        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
            loss += 0.4 * criterion(aux_outputs, labels)
        else:
            main_outputs = outputs
            loss = criterion(main_outputs, labels)
        _, preds = torch.max(main_outputs, 1) # Extract predicted class
        # 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.float() / dataset_sizes[phase]
    print(f"{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}")

return model

```

In [22]: `model = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs=10)`

Epoch 1/2

train Loss: 0.3770 Acc: 0.8650

val Loss: 0.2505 Acc: 0.9100

Epoch 2/2

train Loss: 0.2864 Acc: 0.8979

val Loss: 0.2865 Acc: 0.8886

In [25]:

```

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()

```

In [24]:

```

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)
y_proba.extend(probabilities.cpu().numpy())
# 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("efficientnet_b0", y_true, y_pred, classes)
```

```
100% |██████████| ██████████  
██████████ | 27/27 [00:25<00:00, 1.07it/s]
```

Accuracy: 0.8971631205673759

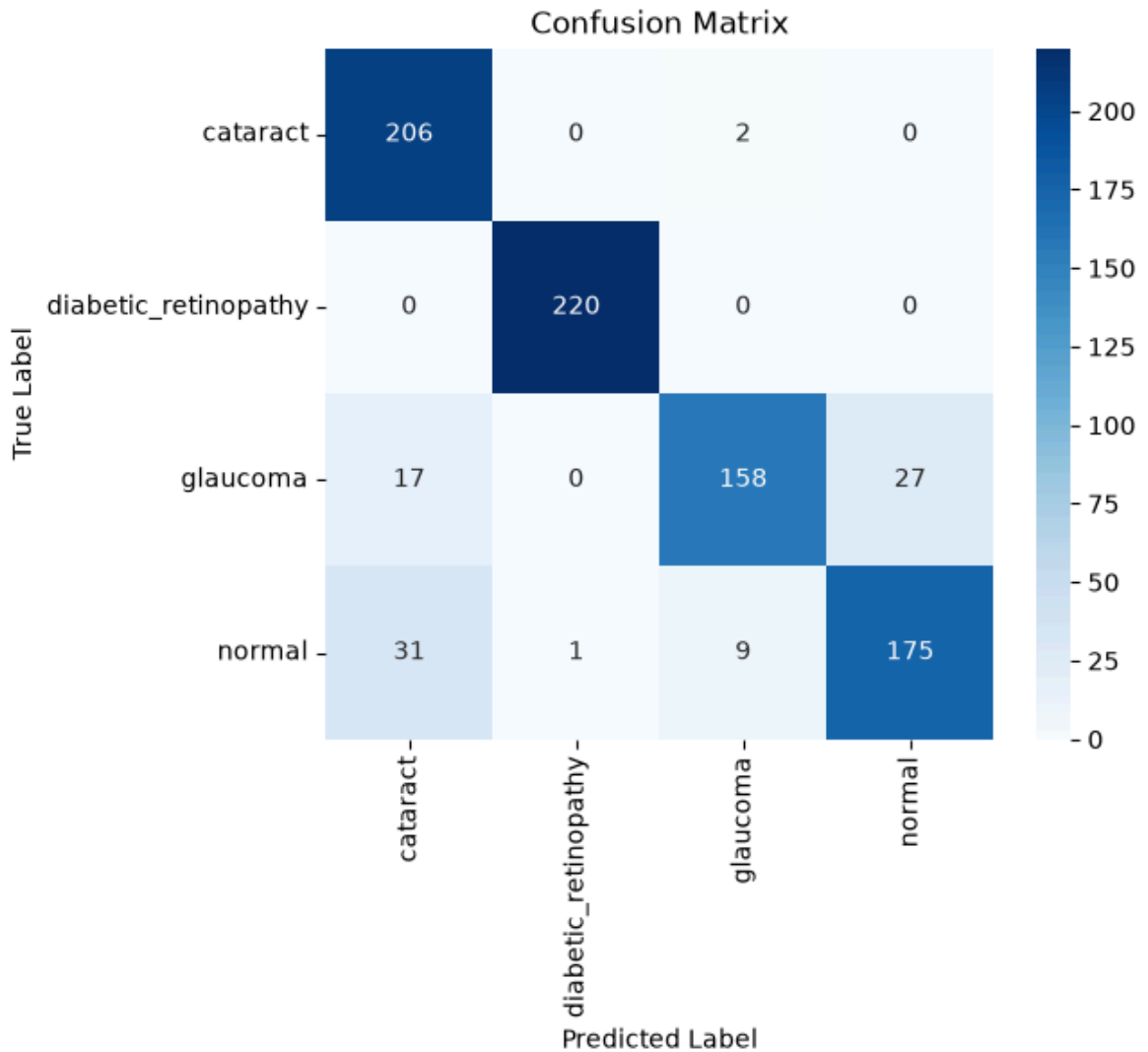
```
Precision : 0.9026929339833439
```

Recall: 0.8971631205673759

F1-score: 0.8958693423770205

Classification Report:

	precision	recall	f1-score	support
0	0.81	0.99	0.89	208
1	1.00	1.00	1.00	220
2	0.93	0.78	0.85	202
3	0.87	0.81	0.84	216
accuracy			0.90	846
macro avg	0.90	0.90	0.89	846
weighted avg	0.90	0.90	0.90	846



```
In [27]: 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=0.1)
model = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs=10)
```

Epoch 1/2

train Loss: 1.1730 Acc: 0.4697

val Loss: 0.9162 Acc: 0.5758

Epoch 2/2

train Loss: 0.8879 Acc: 0.5907

val Loss: 2.1057 Acc: 0.5071

```
In [28]: y_true = []
y_proba = []
```

```
y_pred = []
with torch.no_grad():
    for x, y in tqdm(test_loader):
        output = model(x.to(device))
        y_true.extend(y.numpy())
        probabilities = torch.softmax(output, dim=1)
        y_proba.extend(probabilities.cpu().numpy())
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
        y_pred.extend(pred)
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 [00:23<00:00, 1.15it/s]
```

Accuracy: 0.4988179669030733

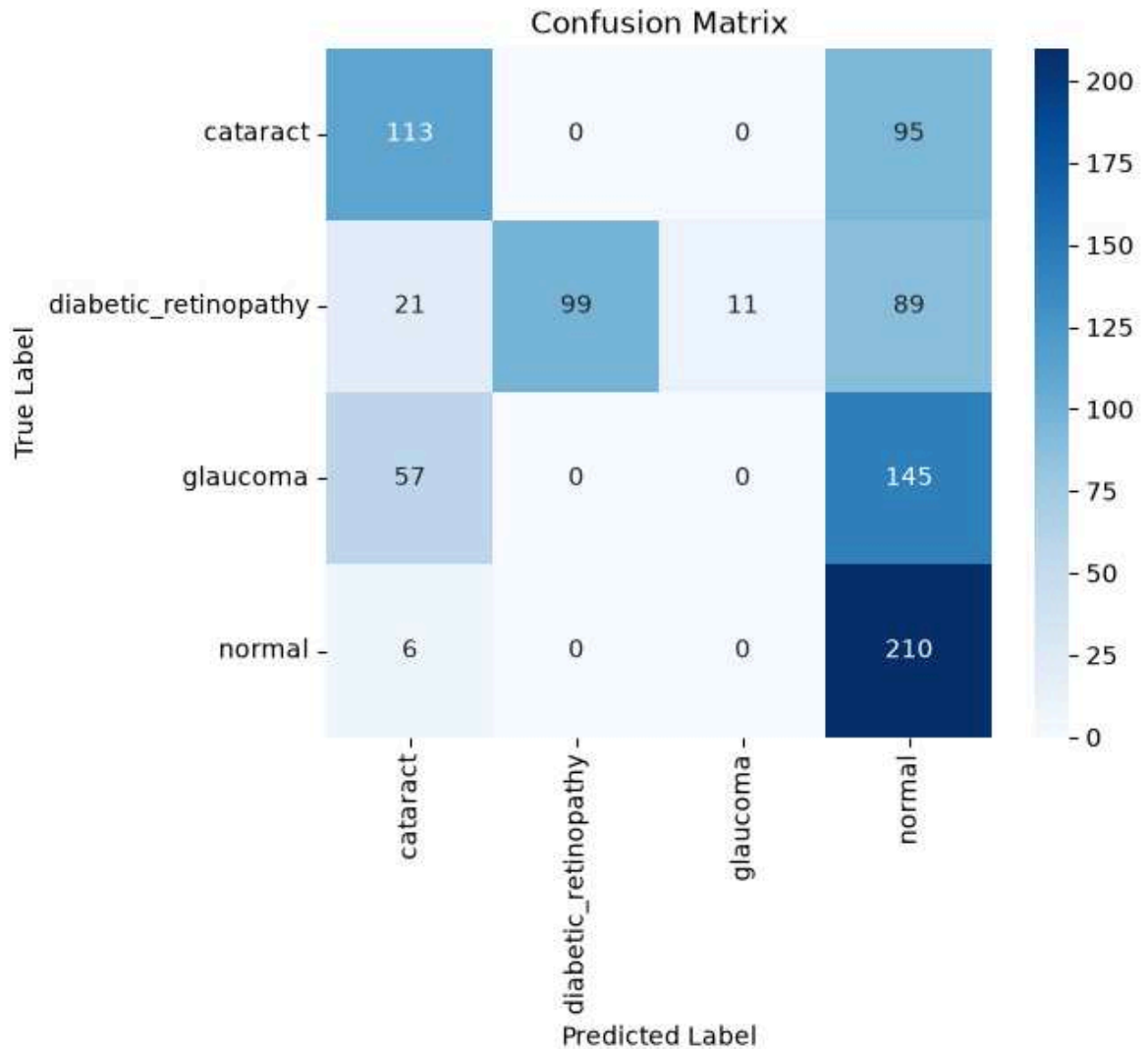
Precision : 0.5005502231984573

Recall: 0.4988179669030733

F1-score: 0.4406380618649541

Classification Report:

	precision	recall	f1-score	support
0	0.57	0.54	0.56	208
1	1.00	0.45	0.62	220
2	0.00	0.00	0.00	202
3	0.39	0.97	0.56	216
accuracy			0.50	846
macro avg	0.49	0.49	0.43	846
weighted avg	0.50	0.50	0.44	846



```
In [29]: 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=0.1)
model = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs=10)
```

Epoch 1/1

train Loss: 1.1544 Acc: 0.4619
val Loss: 1.0753 Acc: 0.5877

```
In [30]: y_true = []
y_proba = []
y_pred = []
with torch.no_grad():
    for x, y in tqdm(test_loader):
        output = model(x.to(device))
        y_true.extend(y.numpy())
        probabilities = torch.softmax(output, dim=1)
        y_proba.extend(probabilities.cpu().numpy())
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
```

```

        y_pred.extend(pred)
y_true = np.array(y_true)
y_pred = np.array(y_pred)
evaluate_model("Densenet121", y_true, y_pred, classes)

```

```
100%|██████████| 27/27 [00:30<00:00, 1.12s/it]
```

Accuracy: 0.5401891252955082

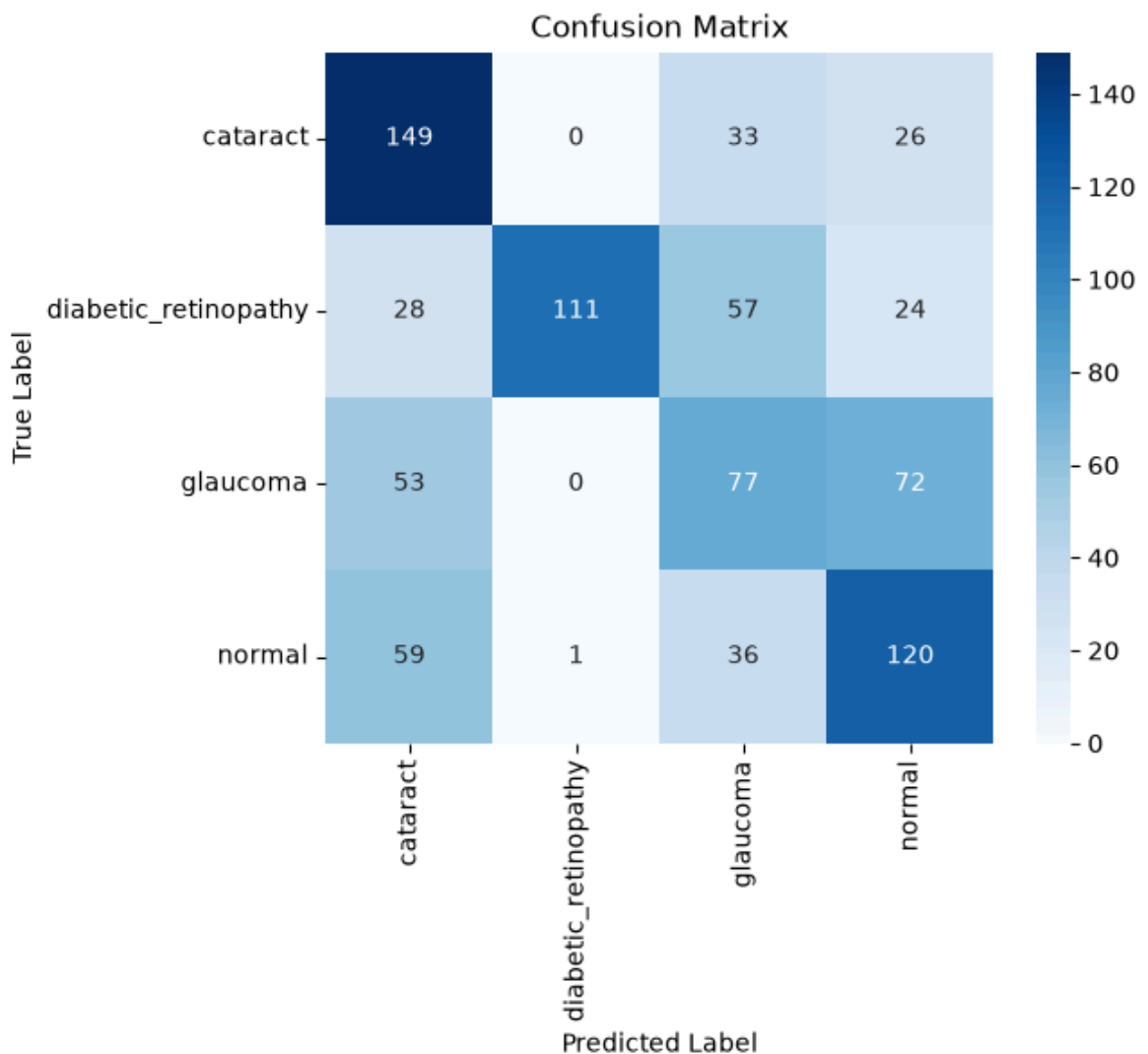
Precision : 0.6016579152277485

Recall: 0.5401891252955082

F1-score: 0.5458893392324281

Classification Report:

	precision	recall	f1-score	support
0	0.52	0.72	0.60	208
1	0.99	0.50	0.67	220
2	0.38	0.38	0.38	202
3	0.50	0.56	0.52	216
accuracy			0.54	846
macro avg	0.60	0.54	0.54	846
weighted avg	0.60	0.54	0.55	846



```
In [33]: # 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)

device_vgg = torch.device('cpu')
model = model.to(device_vgg)

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

# Learning rate scheduler
exp_lr_scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)

device_backup = device
device = device_vgg
model = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs=100)
device = device_backup
```

Epoch 1/1

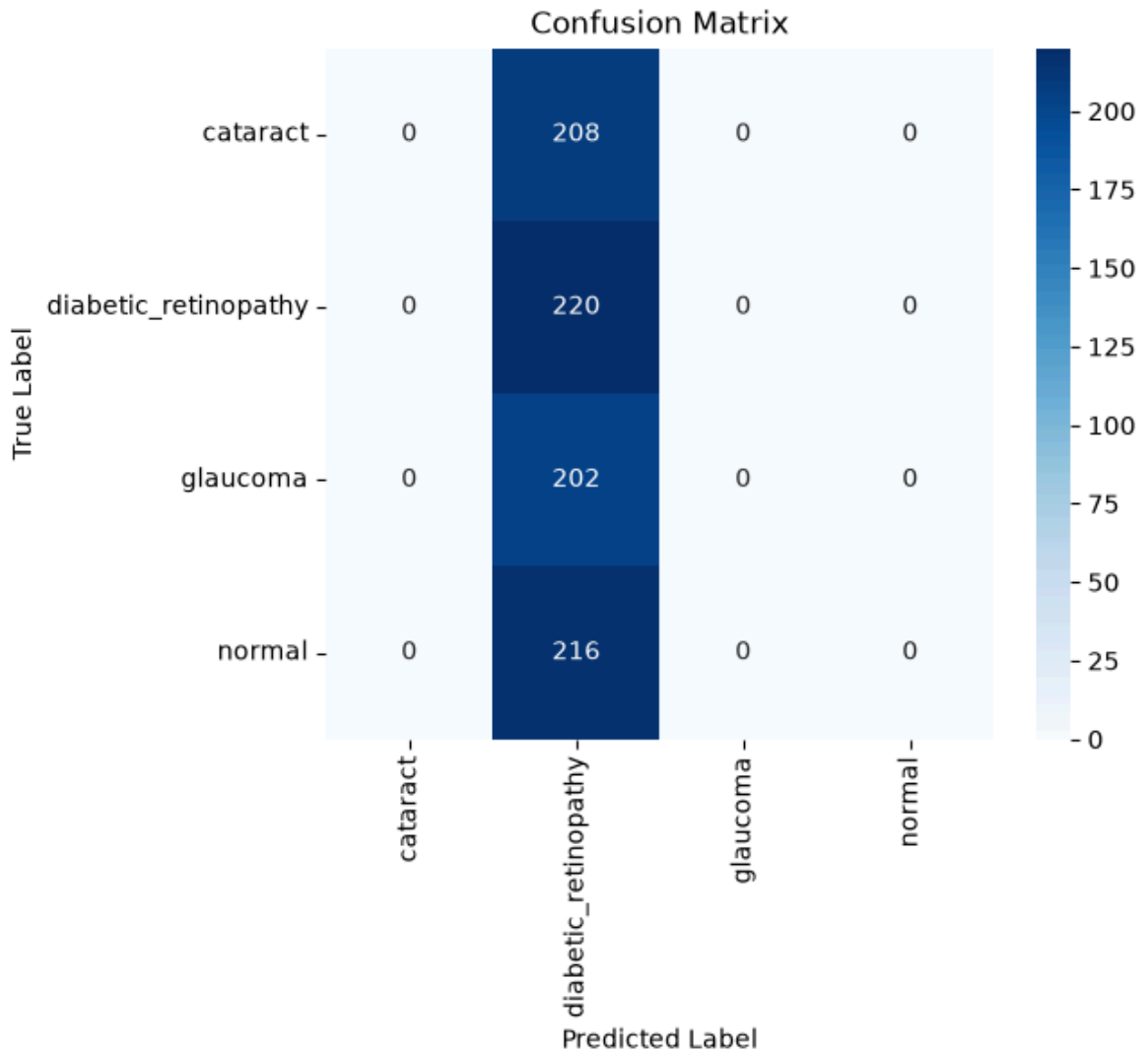
```
train Loss: 3.6523 Acc: 0.2462
val Loss: 1.3859 Acc: 0.2607
```

```
In [34]: model = model.to('cpu')
y_true = []
y_proba = []
y_pred = []
with torch.no_grad():
    for x, y in tqdm(test_loader):
        output = model(x.to('cpu'))
        y_true.extend(y.numpy())
        probabilities = torch.softmax(output, dim=1)
        y_proba.extend(probabilities.cpu().numpy())
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
        y_pred.extend(pred)
y_true = np.array(y_true)
y_pred = np.array(y_pred)
evaluate_model("vgg16", y_true, y_pred, classes)
```

```
100% | ████████████████████████████████████████████████████████████
████████████████████ | 27/27 [03:10<00:00, 7.07s/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



```
In [35]: device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')

# 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=0.1)
model = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs=10)
```

Epoch 1/1

train Loss: 1.6230 Acc: 0.4880
val Loss: 1.5272 Acc: 0.4502

```
In [36]: y_true = []
y_proba = []
y_pred = []
with torch.no_grad():
    for x, y in tqdm(test_loader):
```

```

        output = model(x.to(device))
        y_true.extend(y.numpy())
        probabilities = torch.softmax(output, dim=1)
        y_proba.extend(probabilities.cpu().numpy())
        pred = torch.argmax(probabilities, axis=1).cpu().numpy()
        y_pred.extend(pred)
y_true = np.array(y_true)
y_pred = np.array(y_pred)
evaluate_model("inception_v3", y_true, y_pred, classes)

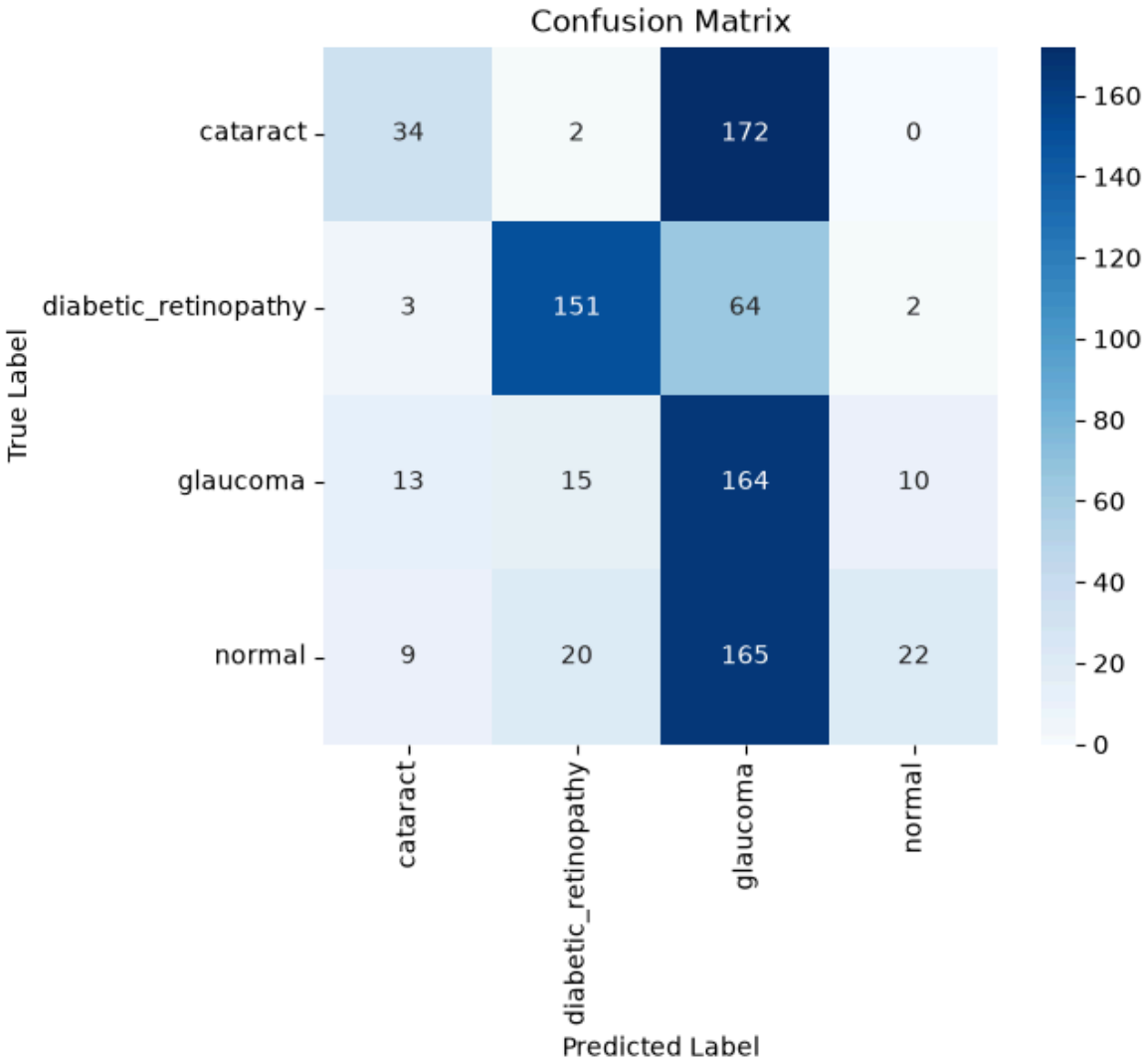
```

```
100%|███████████| 27/27 [00:29<00:00, 1.08s/it]
```

```
Accuracy: 0.4385342789598109
Precision : 0.5850648565486674
Recall: 0.4385342789598109
F1-score: 0.40214684589844935
```

Classification Report:

	precision	recall	f1-score	support
0	0.58	0.16	0.25	208
1	0.80	0.69	0.74	220
2	0.29	0.81	0.43	202
3	0.65	0.10	0.18	216
accuracy			0.44	846
macro avg	0.58	0.44	0.40	846
weighted avg	0.59	0.44	0.40	846



```
In [ ]:
```