

```
from IPython.display import Image, display
display(Image(filename='Lecture 5.png'))
```



Transformer Models and Optimization Methods

Lecture 05

Prepared by

Dr. Hager Saleh

Google drive

<https://drive.google.com/drive/my-drive>

```
# Suppress warnings
import warnings
warnings.simplefilter("ignore")

# Standard library
import os
import time
import copy

# Numerical and data processing
import numpy as np
import pandas as pd

# PyTorch core
import torch
from torch import nn
from torch.utils.data import DataLoader
import torch.optim as optim

# Torchvision for datasets, models, and transforms
```

```

import torchvision
from torchvision import datasets, transforms as T, models
from torchvision.models import vit_b_32

# Timm for vision transformer models and loss functions
import timm
from timm.loss import LabelSmoothingCrossEntropy

# Visualization
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay

# Evaluation metrics
from sklearn.metrics import (
    classification_report,
    confusion_matrix,
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    cohen_kappa_score,
    matthews_corrcoef
)

# Progress bar
from tqdm import tqdm

```

Transformer

A Transformer is a deep learning model introduced in the paper "Attention is All You Need" (Vaswani et al., 2017).

Key Components of the Transformer Model

- Self-Attention Mechanism
The Transformer attends to all input tokens simultaneously.
The Scaled Dot-Product Attention mechanism allows the model to focus on the most relevant parts of the input.
- Multi-Head Attention (MHA)
The model applies multiple self-attention mechanisms in parallel.
Each head learns different aspects of the relationships between tokens.

Types of Transformer models

1. Vision Transformer (ViT)
 - Introduced by Google in 2020.
 - Uses patch embedding: Instead of convolution, it splits an image into fixed-size patches (e.g., 16×16 pixels) and treats them as tokens.

- Applies self-attention mechanisms over these tokens to capture global dependencies.
2. Swin Transformer (Shifted Window Transformer)
- Introduced by Microsoft in 2021.
 - Improves ViT by using a hierarchical structure and shifted window attention to reduce computational complexity.
 - Instead of using global self-attention on all patches, it processes local windows and shifts them between layers to capture both local and global information.

Data augmentation

```
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((224, 224)),
            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((224, 224)),
            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)

def get_classes(data_dir):
    all_data = datasets.ImageFolder(data_dir)
```

```

    return all_data.classes

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')

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

def evaluate_model(model, dataloader):
    y_pred, y_true = [], []
    with torch.no_grad():
        for inputs, labels in tqdm(dataloader):
            inputs = inputs.to(device)
            preds = torch.argmax(model(inputs), axis=1).cpu().numpy()
            y_pred.extend(preds)
            y_true.extend(labels.numpy())
    return y_true, y_pred

```

ViT model

```
model = vit_b_32(pretrained=True)
n_inputs = model.heads.head.in_features

model.heads.head = nn.Sequential(
    nn.Linear(n_inputs, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, len(classes))
)
model = model.to(device)
print(model.heads.head )

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

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

Sequential(
  (0): Linear(in_features=768, out_features=512, bias=True)
  (1): ReLU()
  (2): Dropout(p=0.3, inplace=False)
  (3): Linear(in_features=512, out_features=4, bias=True)
)

ViT_model=train_model(model, criterion, optimizer, scheduler,
num_epochs=20)

Epoch 1/20
-----
train Loss: 1.2614 Acc: 0.3798
val Loss: 1.1816 Acc: 0.4171
Epoch 2/20
-----
train Loss: 1.2203 Acc: 0.4293
val Loss: 1.1923 Acc: 0.4100
Epoch 3/20
-----
train Loss: 1.1507 Acc: 0.4622
val Loss: 1.3099 Acc: 0.4360
Epoch 4/20
-----
train Loss: 1.1231 Acc: 0.4676
val Loss: 1.0962 Acc: 0.4810
Epoch 5/20
-----
```

```
train Loss: 1.0139 Acc: 0.5314
val Loss: 0.9479 Acc: 0.5332
Epoch 6/20
-----
train Loss: 0.9351 Acc: 0.5802
val Loss: 0.8927 Acc: 0.6019
Epoch 7/20
-----
train Loss: 0.9160 Acc: 0.6029
val Loss: 0.8663 Acc: 0.6043
Epoch 8/20
-----
train Loss: 0.8278 Acc: 0.6226
val Loss: 0.8746 Acc: 0.5948
Epoch 9/20
-----
train Loss: 0.7778 Acc: 0.6565
val Loss: 0.7311 Acc: 0.6872
Epoch 10/20
-----
train Loss: 0.7613 Acc: 0.6531
val Loss: 0.7767 Acc: 0.6682
Epoch 11/20
-----
train Loss: 0.7553 Acc: 0.6711
val Loss: 0.7350 Acc: 0.6754
Epoch 12/20
-----
train Loss: 0.7120 Acc: 0.6779
val Loss: 0.8139 Acc: 0.6185
Epoch 13/20
-----
train Loss: 0.7251 Acc: 0.6850
val Loss: 0.6749 Acc: 0.7062
Epoch 14/20
-----
train Loss: 0.6877 Acc: 0.6941
val Loss: 0.6583 Acc: 0.7133
Epoch 15/20
-----
train Loss: 0.7454 Acc: 0.6762
val Loss: 0.6571 Acc: 0.6943
Epoch 16/20
-----
train Loss: 0.7008 Acc: 0.6880
val Loss: 0.7340 Acc: 0.6896
Epoch 17/20
-----
train Loss: 0.6879 Acc: 0.7074
```

```
val Loss: 0.6610 Acc: 0.7014
```

```
Epoch 18/20
```

```
-----
```

```
train Loss: 0.6320 Acc: 0.7270
```

```
val Loss: 0.6222 Acc: 0.7322
```

```
Epoch 19/20
```

```
-----
```

```
train Loss: 0.6526 Acc: 0.7107
```

```
val Loss: 0.6646 Acc: 0.7014
```

```
Epoch 20/20
```

```
-----
```

```
train Loss: 0.6451 Acc: 0.7216
```

```
val Loss: 0.6707 Acc: 0.6991
```

```
# Evaluate best model
```

```
y_true, y_pred= evaluate_model(ViT_model, test_loader)
```

```
print("Classification Report:")
```

```
print(classification_report(y_true, y_pred))
```

```
100%|██████████| 27/27 [00:00<00:00, 27.84it/s]
```

```
Classification Report:
```

	precision	recall	f1-score	support
0	0.85	0.53	0.65	208
1	0.98	0.93	0.95	220
2	0.62	0.53	0.58	202
3	0.58	0.89	0.70	216
accuracy			0.73	846
macro avg	0.76	0.72	0.72	846
weighted avg	0.76	0.73	0.72	846

```
disp =
```

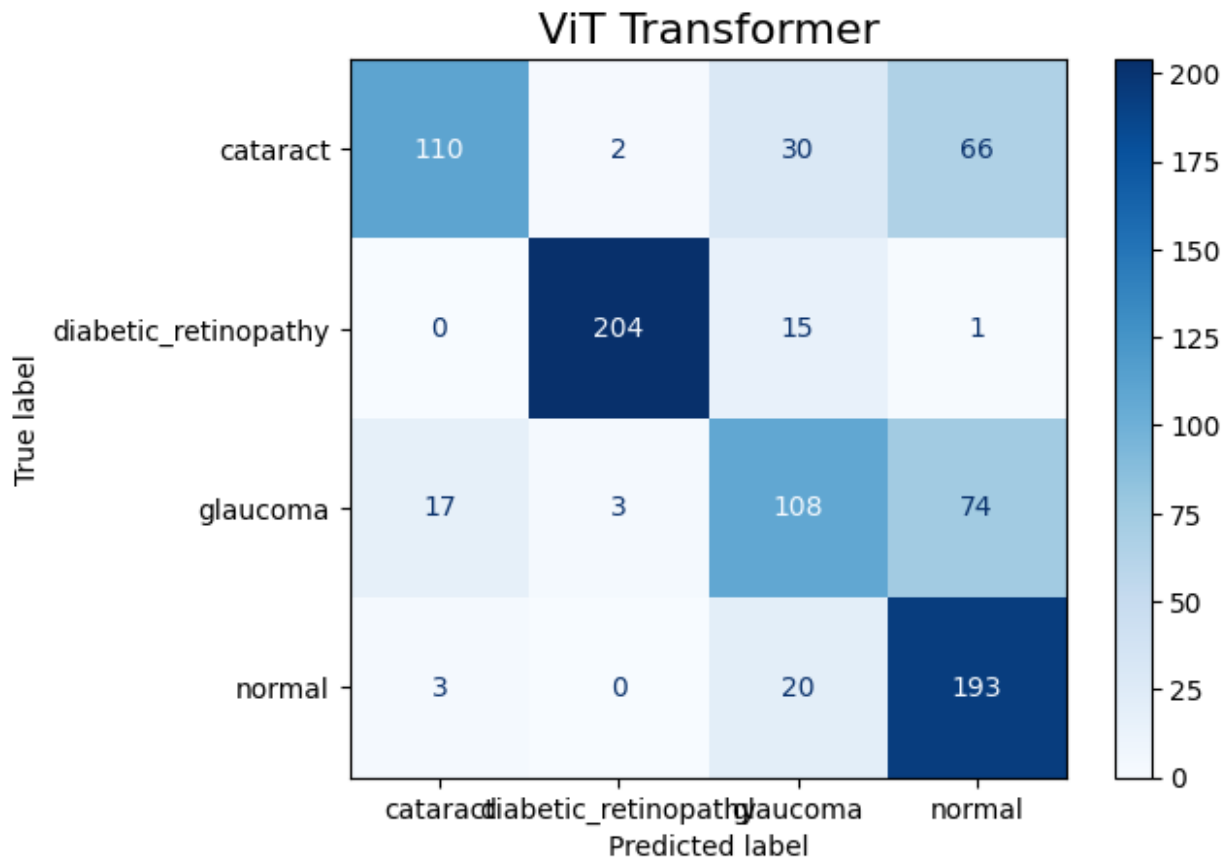
```
ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_true,
```

```
y_pred), display_labels=classes)
```

```
disp.plot(cmap="Blues", values_format='')
```

```
plt.title("ViT Transformer", fontsize=16)
```

```
plt.show()
```



```
results = pd.DataFrame([{'Accuracy': round(accuracy_score(y_true,
y_pred) * 100, 3), 'Precision': round(precision_score(y_true, y_pred,
average='weighted') * 100, 3), 'Recall': round(recall_score(y_true,
y_pred, average='weighted') * 100, 3), 'F1 Score':
round(f1_score(y_true, y_pred, average='weighted') * 100, 3), 'Kappa
Statistic': round(cohen_kappa_score(y_true, y_pred) * 100, 3), 'MCC':
round(matthews_corrcoef(y_true, y_pred) * 100, 3)}])
results
```

	Accuracy	Precision	Recall	F1 Score	Kappa Statistic	MCC
0	72.695	75.846	72.695	72.407	63.513	64.797

Swin Model

```
HUB_URL = "SharanSMenon/swin-transformer-hub:main"
MODEL_NAME = "swin_tiny_patch4_window7_224"
model = torch.hub.load(HUB_URL, MODEL_NAME, pretrained=True)
model = model.to(device)

criterion = CrossEntropyLoss().to(device)
optimizer = optim.AdamW(model.parameters(), lr=1e-4)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.97)
```

```
Swin_model = train_model(model, criterion, optimizer, scheduler,  
num_epochs=5)
```

```
Using cache found in /home/hager/.cache/torch/hub/SharanSMenon_swin-  
transformer-hub_main
```

```
Epoch 0/4
```

```
-----
```

```
100%|██████████| 24/24 [04:37<00:00, 11.55s/it]
```

```
Train Loss: 2.5203 Acc: 0.5381
```

```
Epoch 1/4
```

```
-----
```

```
100%|██████████| 24/24 [04:29<00:00, 11.23s/it]
```

```
Train Loss: 1.5091 Acc: 0.7979
```

```
Epoch 2/4
```

```
-----
```

```
100%|██████████| 24/24 [04:57<00:00, 12.39s/it]
```

```
Train Loss: 1.3686 Acc: 0.8569
```

```
Epoch 3/4
```

```
-----
```

```
100%|██████████| 24/24 [04:48<00:00, 12.01s/it]
```

```
Train Loss: 1.2709 Acc: 0.9034
```

```
Epoch 4/4
```

```
-----
```

```
100%|██████████| 24/24 [04:50<00:00, 12.09s/it]
```

```
Train Loss: 1.2288 Acc: 0.9237
```

```
Training complete in 23m 43s
```

```
Best Train Acc: 0.9237
```

```
# Evaluate best model
```

```
y_true, y_pred= evaluate_model(Swin_model, test_loader)
```

```
print("Classification Report:")
```

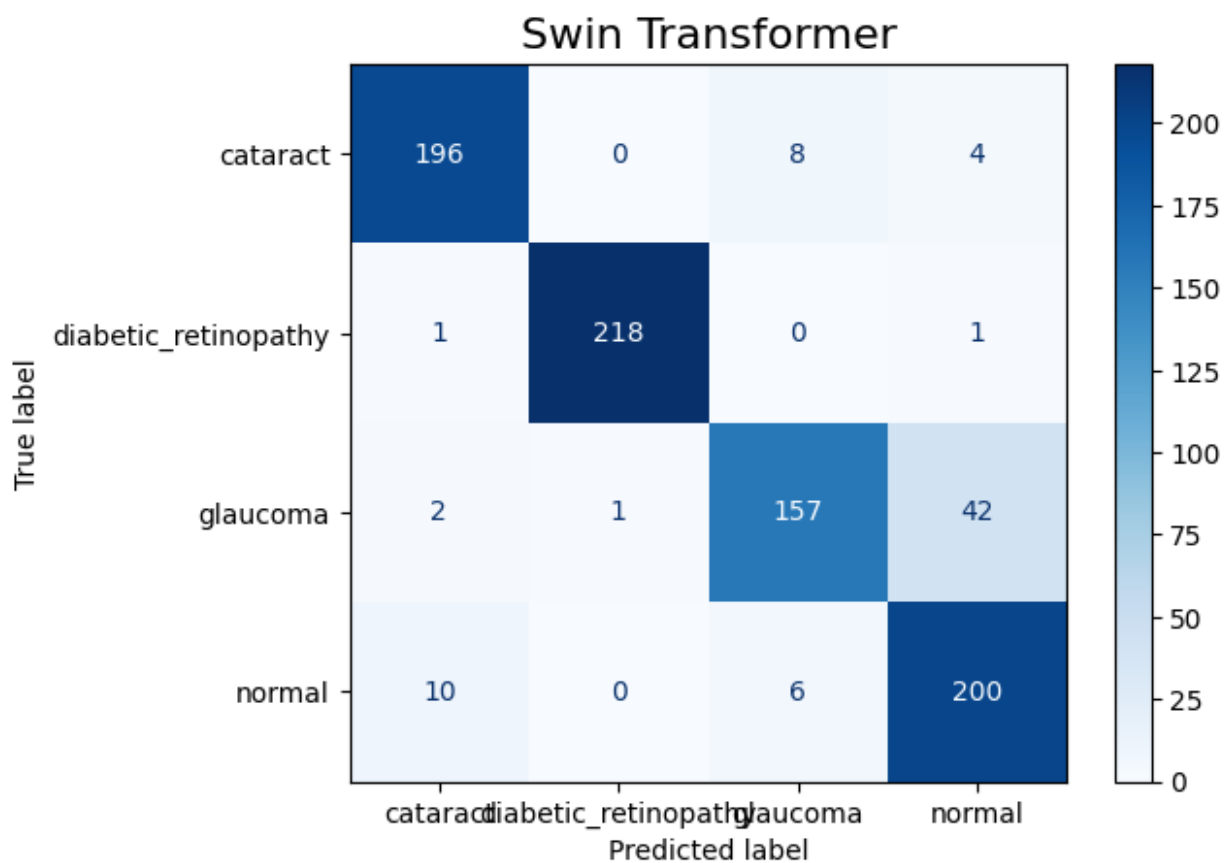
```
print(classification_report(y_true, y_pred))
```

100%|██████████| 27/27 [00:12<00:00, 2.19it/s]

Classification Report:

	precision	recall	f1-score	support
0	0.94	0.94	0.94	208
1	1.00	0.99	0.99	220
2	0.92	0.78	0.84	202
3	0.81	0.93	0.86	216
accuracy			0.91	846
macro avg	0.92	0.91	0.91	846
weighted avg	0.92	0.91	0.91	846

```
disp =  
ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_true,  
y_pred), display_labels=classes)  
disp.plot(cmap="Blues", values_format='')  
plt.title("Swin Transformer", fontsize=16)  
plt.show()
```



```
results = pd.DataFrame([{'Accuracy': round(accuracy_score(y_true,
y_pred) * 100, 3), 'Precision': round(precision_score(y_true, y_pred,
average='weighted') * 100, 3), 'Recall': round(recall_score(y_true,
y_pred, average='weighted') * 100, 3), 'F1 Score':
round(f1_score(y_true, y_pred, average='weighted') * 100, 3), 'Kappa
Statistic': round(cohen_kappa_score(y_true, y_pred) * 100, 3), 'MCC':
round(matthews_corrcoef(y_true, y_pred) * 100, 3) }])
results
```

	Accuracy	Precision	Recall	F1 Score	Kappa Statistic	MCC
0	91.135	91.539	91.135	91.097	88.166	88.325

Optimization models

Hyperparameter tuning optimization methods are techniques used to find the best combination of hyperparameters (e.g., learning rate, number of layers, dropout rate) that maximize model performance (accuracy, F1-score, etc.) on validation data.

Optuna is an open-source hyperparameter optimization framework designed for automated machine learning (AutoML). It helps you efficiently find the best hyperparameters for your machine learning models using intelligent search algorithms.

Common Use Cases:

- Tuning deep learning models (e.g., LSTM, CNN, Transformers).
- Tuning gradient boosting models (e.g., XGBoost, LightGBM).
- Hyperparameter tuning in time series forecasting, NLP, and CV.

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

# Assume dataloaders, dataset_sizes, device, and num_classes are
already defined

def define_model(trial):
    # Suggest hyperparameters
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-2)
    weight_decay = trial.suggest_loguniform("weight_decay", 1e-6, 1e-
2)
    step_size = trial.suggest_int("step_size", 2, 5)
    gamma = trial.suggest_uniform("gamma", 0.85, 0.99)

    # Load EfficientNet-B0
    model = models.efficientnet_b0(pretrained=True)
    model.classifier[1] = nn.Linear(model.classifier[1].in_features,
num_classes)
```

```

    model = model.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=lr,
weight_decay=weight_decay)
    scheduler = optim.lr_scheduler.StepLR(optimizer,
step_size=step_size, gamma=gamma)

    return model, criterion, optimizer, scheduler

best_model = None

def train_model_optuna(trial):
    global best_model # Access the global best_model

    model, criterion, optimizer, scheduler = define_model(trial)
    num_epochs = 10
    best_val_acc = 0.0

    for epoch in range(num_epochs):
        for phase in ['train', 'val']:
            model.train() if phase == 'train' else model.eval()
            running_loss, running_corrects = 0.0, 0

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

                optimizer.zero_grad()
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)

                    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:
                            loss += 0.4 * criterion(aux_outputs,
labels)
                    else:
                        main_outputs = outputs
                        loss = criterion(main_outputs, labels)

                _, preds = torch.max(main_outputs, 1)

```

```

        if phase == 'train':
            loss.backward()
            optimizer.step()

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

    if phase == 'train':
        scheduler.step()

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

    if phase == 'val':
        if epoch_acc.item() > best_val_acc:
            best_val_acc = epoch_acc.item()
            best_model = model  # Save current best model

    trial.report(best_val_acc, epoch)
    if trial.should_prune():
        raise optuna.exceptions.TrialPruned()

    return best_val_acc

# Run Optuna
study = optuna.create_study(direction="maximize")
study.optimize(train_model_optuna, n_trials=20)

# Retrieve the best model after study
final_model = best_model

```

```

[I 2025-05-26 10:41:30,478] A new study created in memory with name:
no-name-8c9ba102-eafb-4bef-b8a3-19fd83baee4d
[I 2025-05-26 10:42:34,959] Trial 0 finished with value:
0.9691943127962086 and parameters: {'lr': 0.0002758982582152921,
'weight_decay': 1.9560707774116736e-05, 'step_size': 3, 'gamma':
0.9451259188092642}. Best is trial 0 with value: 0.9691943127962086.
[I 2025-05-26 10:43:37,656] Trial 1 finished with value:
0.9620853080568721 and parameters: {'lr': 0.0009059617023418751,
'weight_decay': 0.00014007320878336595, 'step_size': 2, 'gamma':
0.9192726127637376}. Best is trial 0 with value: 0.9691943127962086.
[I 2025-05-26 10:44:40,430] Trial 2 finished with value:
0.9573459715639812 and parameters: {'lr': 7.939780770773538e-05,
'weight_decay': 2.1004837933986802e-06, 'step_size': 5, 'gamma':
0.8718149583142932}. Best is trial 0 with value: 0.9691943127962086.
[I 2025-05-26 10:45:42,813] Trial 3 finished with value:
0.9526066350710901 and parameters: {'lr': 0.0005103643441997632,
'weight_decay': 0.00013296468297823908, 'step_size': 3, 'gamma':

```

```
0.8725048576951162}. Best is trial 0 with value: 0.9691943127962086.
[I 2025-05-26 10:46:45,340] Trial 4 finished with value:
0.9312796208530807 and parameters: {'lr': 3.367585304900838e-05,
'weight_decay': 0.00031536235330922326, 'step_size': 3, 'gamma':
0.8680932889782157}. Best is trial 0 with value: 0.9691943127962086.
[I 2025-05-26 10:47:30,080] Trial 5 pruned.
[I 2025-05-26 10:47:35,573] Trial 6 pruned.
[I 2025-05-26 10:47:55,717] Trial 7 pruned.
[I 2025-05-26 10:48:34,757] Trial 8 pruned.
[I 2025-05-26 10:49:40,143] Trial 9 finished with value:
0.9597156398104266 and parameters: {'lr': 0.00018353687459217194,
'weight_decay': 0.0007071236420498208, 'step_size': 4, 'gamma':
0.8926537572880979}. Best is trial 0 with value: 0.9691943127962086.
[I 2025-05-26 10:49:45,603] Trial 10 pruned.
[I 2025-05-26 10:50:05,228] Trial 11 pruned.
[I 2025-05-26 10:50:10,591] Trial 12 pruned.
[I 2025-05-26 10:50:23,662] Trial 13 pruned.
[I 2025-05-26 10:50:55,314] Trial 14 pruned.
[I 2025-05-26 10:51:01,960] Trial 15 pruned.
[I 2025-05-26 10:51:08,586] Trial 16 pruned.
[I 2025-05-26 10:51:13,943] Trial 17 pruned.
[I 2025-05-26 10:51:33,541] Trial 18 pruned.
[I 2025-05-26 10:51:51,908] Trial 19 pruned.
```

```
y_true, y_pred= evaluate_model(final_model, test_loader)
```

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

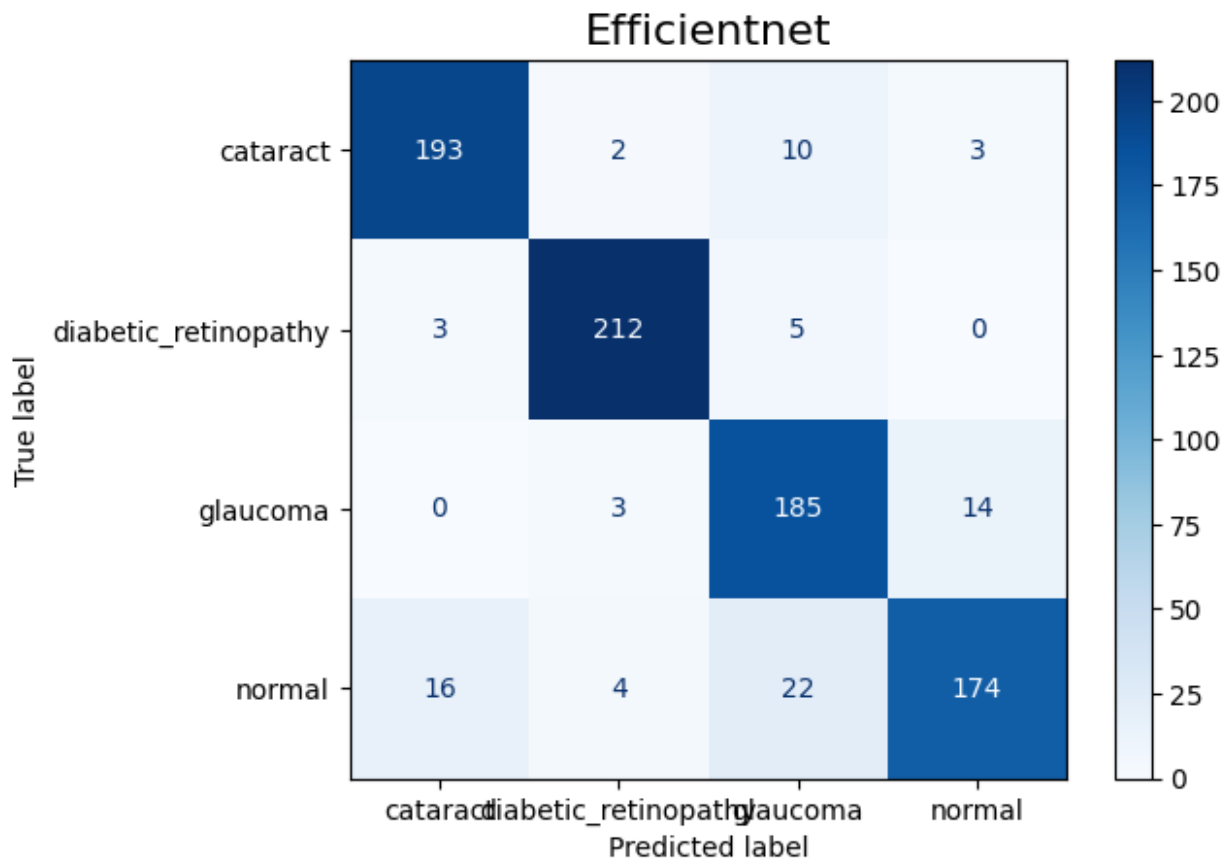
```
100%|██████████| 27/27 [00:01<00:00, 24.34it/s]
```

```
Classification Report:
```

	precision	recall	f1-score	support
0	0.91	0.93	0.92	208
1	0.96	0.96	0.96	220
2	0.83	0.92	0.87	202
3	0.91	0.81	0.86	216
accuracy			0.90	846
macro avg	0.90	0.90	0.90	846
weighted avg	0.90	0.90	0.90	846

```
disp =
ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_true,
y_pred), display_labels=classes)
disp.plot(cmap="Blues", values_format='')
```

```
plt.title("Efficientnet", fontsize=16)
plt.show()
```



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

def define_model(trial):
    # Suggest hyperparameters
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-2)
    weight_decay = trial.suggest_loguniform("weight_decay", 1e-6, 1e-2)

    step_size = trial.suggest_int("step_size", 2, 5)
    gamma = trial.suggest_uniform("gamma", 0.85, 0.99)

    model = models.resnet18(pretrained=True)

    # 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)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=lr,
weight_decay=weight_decay)
    scheduler = optim.lr_scheduler.StepLR(optimizer,
step_size=step_size, gamma=gamma)

    return model, criterion, optimizer, scheduler

best_model = None

def train_model_optuna(trial):
    global best_model # Access the global best_model

    model, criterion, optimizer, scheduler = define_model(trial)
    num_epochs = 10
    best_val_acc = 0.0

    for epoch in range(num_epochs):
        for phase in ['train', 'val']:
            model.train() if phase == 'train' else model.eval()
            running_loss, running_corrects = 0.0, 0

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

                optimizer.zero_grad()
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)

                    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:
                            loss += 0.4 * criterion(aux_outputs,
labels)
                    else:
                        main_outputs = outputs
                        loss = criterion(main_outputs, labels)

```

```

        _, preds = torch.max(main_outputs, 1)

        if phase == 'train':
            loss.backward()
            optimizer.step()

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

    if phase == 'train':
        scheduler.step()

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

    if phase == 'val':
        if epoch_acc.item() > best_val_acc:
            best_val_acc = epoch_acc.item()
            best_model = model # Save current best model

    trial.report(best_val_acc, epoch)
    if trial.should_prune():
        raise optuna.exceptions.TrialPruned()

    return best_val_acc

# Run Optuna
study = optuna.create_study(direction="maximize")
study.optimize(train_model_optuna, n_trials=3)

# Retrieve the best model after study
final_model = best_model

```

```

[I 2025-05-26 11:04:58,004] A new study created in memory with name:
no-name-78fe8f0a-2ce3-4fb3-9c2d-7142178da5f0
[I 2025-05-26 11:05:39,739] Trial 0 finished with value:
0.9454976303317536 and parameters: {'lr': 0.00013133942707508814,
'weight_decay': 9.164801053054393e-06, 'step_size': 2, 'gamma':
0.9509783210299451}. Best is trial 0 with value: 0.9454976303317536.
[I 2025-05-26 11:06:18,215] Trial 1 finished with value:
0.9431279620853081 and parameters: {'lr': 0.00014124489978893846,
'weight_decay': 0.00027224653067919175, 'step_size': 2, 'gamma':
0.9851011698178557}. Best is trial 0 with value: 0.9454976303317536.
[I 2025-05-26 11:06:57,709] Trial 2 finished with value:
0.9407582938388627 and parameters: {'lr': 1.2433091996828278e-05,
'weight_decay': 0.0001924848390598402, 'step_size': 2, 'gamma':
0.9852980486827497}. Best is trial 0 with value: 0.9454976303317536.

```

```
y_true, y_pred= evaluate_model(final_model, test_loader)
```

```
print("Classification Report:")
```

```
print(classification_report(y_true, y_pred))
```

```
100%|██████████| 27/27 [00:00<00:00, 31.70it/s]
```

Classification Report:

	precision	recall	f1-score	support
0	0.96	0.92	0.94	208
1	1.00	1.00	1.00	220
2	0.88	0.90	0.89	202
3	0.89	0.91	0.90	216
accuracy			0.93	846
macro avg	0.93	0.93	0.93	846
weighted avg	0.93	0.93	0.93	846

```
disp =
```

```
ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_true,  
y_pred), display_labels=classes)
```

```
disp.plot(cmap="Blues", values_format='')
```

```
plt.title("Resnet18", fontsize=16)
```

```
plt.show()
```

