Tomato disease classifier

This notebook is about building a tomato disease classifier that can be used to identify the possible disseases of your tomato using an image

```
# # downloaded the dataset:
# !curl -L -o /content/tomato-disease-multiple-sources.zip
https://www.kaggle.com/api/v1/datasets/download/cookiefinder/tomato-
disease-multiple-sources

# # Unzip the file
# !unzip /content/tomato-disease-multiple-sources.zip -d
/content/tomato-disease-dataset

# # Remove the zip file
# !rm /content/tomato-disease-multiple-sources.zip

# import os

# # Change the working directory
# os.chdir('/content/tomato-disease-dataset')

# # Verify the change
# print(os.getcwd())

#importing required libraries for the job

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import transforms, models
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import ImageFolder
import torchvision.transforms as T

import matplotlib.pyplot as plt
import numpy as np


# Check if any GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# If a CUDA device is available, use the first available GPU
if torch.cuda.is_available():
    device = torch.device("cuda:0")  # Using the first available GPU
at index 0
```

```python
else:
    device = torch.device("cpu")

print(f"Using device: {device}")
```

```
Using device: cuda:0
```

```python
global IMAGE_SIZE, BATCH_SIZE, CHANNELS, NUM_CLASSES, EPOCH
IMAGE_SIZE = 256
BATCH_SIZE = 16
CHANNELS = 3
EPOCH = 80

#defining the path of the dateset
data_path = '/kaggle/input/tomato-disease-multiple-sources/train'
#/content/tomato-disease-dataset/train

transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),  # Making sure that
the size is fixed for all
    transforms.ToTensor(), # Convert to tensor
])

#Load dataset from directory
dataset = ImageFolder(root=data_path, transform=transform)

# Create DataLoader with batching
data_loader = DataLoader(dataset, batch_size= BATCH_SIZE,
shuffle=True,
                          num_workers=4,               # Number of
workers for data loading
                          pin_memory=True,             # Use pinned
memory for GPU transfer
                          prefetch_factor=2,            # Number of
batches to prefetch per workers
                          persistent_workers=True      # Keep workers
alive between epochs
                         )



CLASSES = dataset.classes
NUM_CLASSES = len(CLASSES)
print(NUM_CLASSES)

# View class names (subfolder names)
print("Class Names:")
for class_name in dataset.classes:
    print(class_name)

# View class-to-index mapping
```

```
print("\nClass-to-Index Mapping:")
for class_name, index in dataset.class_to_idx.items():
    print(f"{index}: {class_name}")

# View total number of samples
print("\nNumber of images:", len(dataset))

# View a single sample (image and label)
img1, label1 = dataset[0]  # Access the first sample
print("Image shape:", img1.shape)  # Shape of the image tensor
print("Label:", label1)  # Integer label corresponding to the class
```

```
11
Class Names:
Bacterial_spot
Early_blight
Late_blight
Leaf_Mold
Septoria_leaf_spot
Spider_mites Two-spotted_spider_mite
Target_Spot
Tomato_Yellow_Leaf_Curl_Virus
Tomato_mosaic_virus
healthy
powdery_mildew

Class-to-Index Mapping:
0: Bacterial_spot
1: Early_blight
2: Late_blight
3: Leaf_Mold
4: Septoria_leaf_spot
5: Spider_mites Two-spotted_spider_mite
6: Target_Spot
7: Tomato_Yellow_Leaf_Curl_Virus
8: Tomato_mosaic_virus
9: healthy
10: powdery_mildew

Number of images: 25851
Image shape: torch.Size([3, 256, 256])
Label: 0
```

```
num = len(data_loader)
print(num)
```

```
1616
```

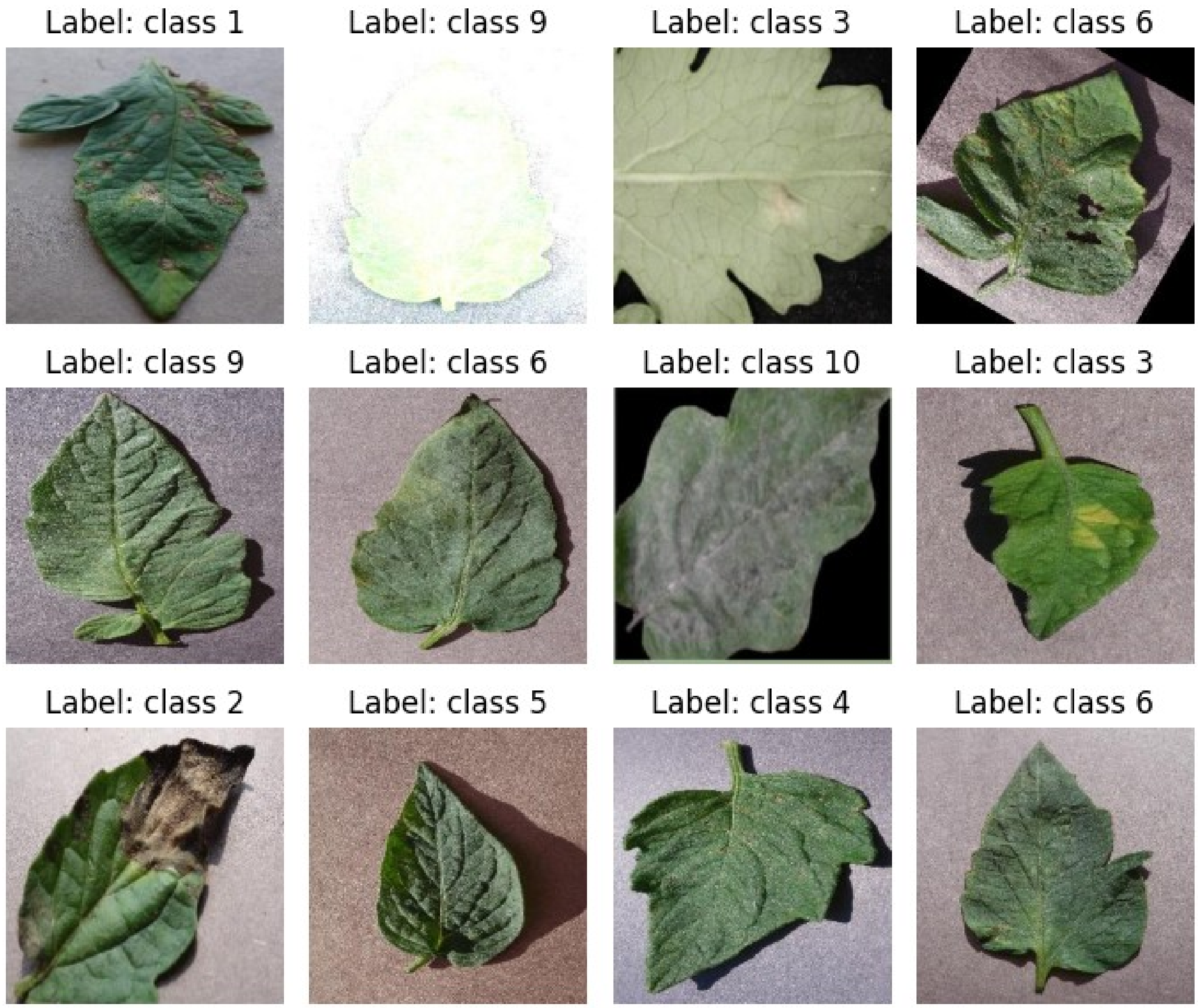
```
print(num*32) #Matches the number of batches we have created and the number of images we have got
```

```
51712
```

```python
for images, labels in data_loader:
    print(images.shape)
    print(labels.numpy())
    break
```

```
torch.Size([16, 3, 256, 256])
[ 9  0  7  9  9  8  4  5  7 10  0  6  7  9  0  2]
```

```python
#for a singular image
for images, labels in data_loader:
  fig, axes = plt.subplots(3, 4, figsize=(7, 6))

  # Plot the first 12 images
  for i in range(12):
      row, col = divmod(i, 4)
      ax = axes[row, col]
      ax.imshow(images[i].permute(1, 2, 0))  # Change from C x H x W
to H x W x C)
      ax.set_title(f"Label: class {labels[i].item()}")
      ax.axis('off')  # Turn off axes

  plt.tight_layout()
  plt.show()
  break
```



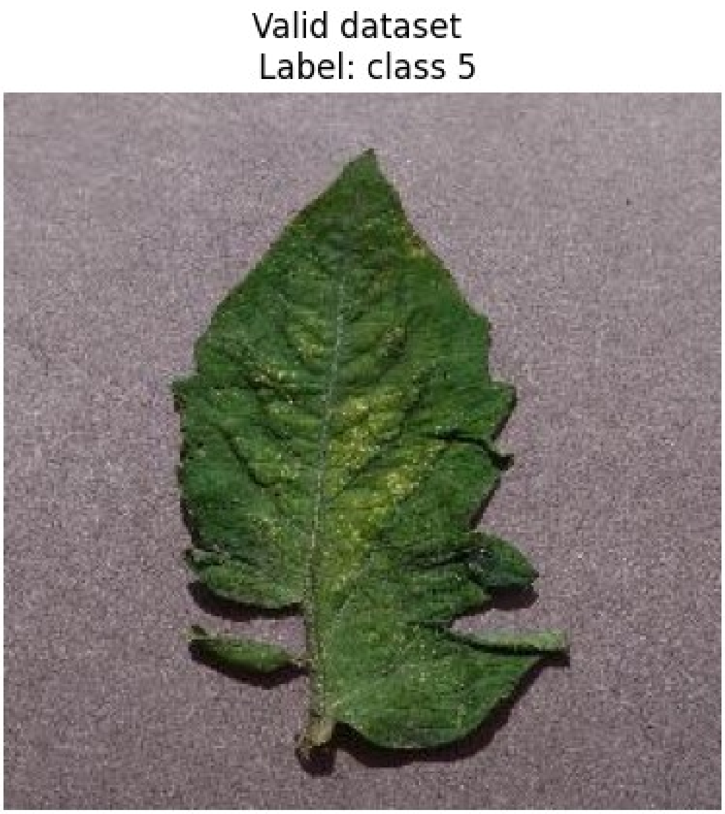

```
#defining the path of the valid and test dateset
valid_data_path =
'/kaggle/input/tomato-disease-multiple-sources/valid'
#'/content/tomato-disease-dataset/valid'

# Define the transformation
valid_transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),  # Making sure that
the size is fixed for all
    transforms.ToTensor(),                         # Convert to tensor
])

# Load dataset from directory
valid_dataset = ImageFolder(root=valid_data_path,
transform=valid_transform)

# Define the split sizes
valid_size = int(0.5 * len(valid_dataset))  # 50% for validation
```

```
test_size = len(valid_dataset) - valid_size  # Remaining for test

# Split the dataset
valid_subset, test_subset = random_split(valid_dataset, [valid_size,
test_size])

# Create DataLoaders for validation and test sets
# Set a seed for reproducibility
torch.manual_seed(42)
valid_loader = DataLoader(valid_subset, batch_size= BATCH_SIZE,
shuffle=True,
                           num_workers=4,              # Number of
workers for data loading
                           pin_memory=True,            # Use pinned
memory for GPU transfer
                           prefetch_factor=2,           # Number of
batches to prefetch per workers
                           persistent_workers=True      # Keep workers
alive between epochs
                         )



# Set a seed for reproducibility
torch.manual_seed(42)
test_loader = DataLoader(test_subset,  batch_size= BATCH_SIZE,
shuffle=True,
                           num_workers=4,              # Number of
workers for data loading
                           pin_memory=True,            # Use pinned
memory for GPU transfer
                           prefetch_factor=2,           # Number of
batches to prefetch per workers
                           persistent_workers=True      # Keep workers
alive between epochs
                         )




# Check the size of SETS
print(f"Training set size: {len(data_loader)}")
print(f"Validation set size: {len(valid_loader)}")
print(f"Test set size: {len(test_loader)}")
```

```
Training set size: 1616
Validation set size: 209
Test set size: 209
```

```
#Checking whether the datasets are fine
for images, labels in valid_loader:
```

```
    print(images[0].shape)
    print(labels[0].numpy())
    plt.imshow(images[0].permute(1, 2, 0))  # Change from C x H x W to
H x W x C)
    plt.title(f"Valid dataset \n Label: class {labels[0]}")
    plt.axis('off')
    plt.show()
    break

torch.Size([3, 256, 256])
5
```




```
for images, labels in test_loader:
    print(images[0].shape)
    print(labels[0].numpy())
    plt.imshow(images[0].permute(1, 2, 0))  # Change from C x H x W to
H x W x C)
    plt.title(f"Test Dataset \nLabel: class {labels[0]}")
    plt.axis('off')
    plt.show()
    break

torch.Size([3, 256, 256])
7
```


```
# For scaling and resizing incase external images are used
class ScaleAndResizeLayer(nn.Module):
    def __init__(self, size=(IMAGE_SIZE, IMAGE_SIZE),
scale=1.0/255.0):
        super(ScaleAndResizeLayer, self).__init__()
        self.resize = T.Resize(size)  # Resize to the target size
        self.scale = scale           # Scale factor

    def forward(self, x):
        # Resize and scale
        x = self.resize(x)
        x = x * self.scale  # Scale the pixel values
        return x

#Augmenting the input the data for better modelling
class AugmentationLayer(nn.Module):
    def __init__(self, image_size=(IMAGE_SIZE, IMAGE_SIZE)):
        super(AugmentationLayer, self).__init__()
        self.augmentations = T.Compose([
            T.RandomHorizontalFlip(p=0.5),            # 50% chance of
horizontal flip
            T.RandomRotation(degrees=30),            # Random rotation
within ±30 degrees
            T.Resize(image_size),                    # Resize to the
```

```
target size
            T.RandomCrop(image_size),                # Optional:
Random crop
            T.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.1),  # Random color jitter
        ])

    def forward(self, x):
        # Apply augmentations
        return self.augmentations(x)

#Building the CNN -based model now


class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.model = nn.Sequential(
            # Preprocessing and augmentation layers
            ScaleAndResizeLayer(),                                 #
Scaling and Resizing layer
            AugmentationLayer(),                                   #
Augmentation layer

            # First convolution block
            nn.Conv2d(3, 32, kernel_size=(3, 3), stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Additional convolution blocks
            nn.Conv2d(32, 64, kernel_size=(3, 3), stride=1,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 64, kernel_size=(3, 3), stride=1,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 64, kernel_size=(3, 3), stride=1,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 64, kernel_size=(3, 3), stride=1,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
```

```
            nn.Conv2d(64, 64, kernel_size=(3, 3), stride=1,
padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Flatten layer
            nn.Flatten(),

            # Fully connected (dense) layer
            nn.Linear(64 * 4 * 4, 64),  # Adjust input features based
on the final dimensions
            nn.ReLU(),

            # Final classification layer
            nn.Linear(64, NUM_CLASSES),  # Final layer with number of
neurons equal to number of classes
            nn.Softmax(dim=1)              # Softmax for multi-class
classification
        )

    def forward(self, x):
        assert x.shape[1:] == (3, 256, 256), f"Expected (3, 256, 256),
got {x.shape[1:]}"  # Check input shape
        return self.model(x)



!pip install torchsummary # Install the torchsummary package

#Printing a summary of the built model

from torchsummary import summary

model = MyModel()
model.to(device)
summary(model, input_size=(3, 256, 256))  # (Channels, Height, Width)
```

```
Requirement already satisfied: torchsummary in
/usr/local/lib/python3.10/dist-packages (1.5.1)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Resize-1          [-1, 3, 256, 256]               0
ScaleAndResizeLayer-2          [-1, 3, 256, 256]               0
 AugmentationLayer-3          [-1, 3, 256, 256]               0
            Conv2d-4         [-1, 32, 256, 256]             896
              ReLU-5         [-1, 32, 256, 256]               0
         MaxPool2d-6         [-1, 32, 128, 128]               0
            Conv2d-7         [-1, 64, 128, 128]          18,496
              ReLU-8         [-1, 64, 128, 128]               0
```

```
        MaxPool2d-9          [-1, 64, 64, 64]               0
           Conv2d-10          [-1, 64, 64, 64]          36,928
             ReLU-11          [-1, 64, 64, 64]               0
        MaxPool2d-12          [-1, 64, 32, 32]               0
           Conv2d-13          [-1, 64, 32, 32]          36,928
             ReLU-14          [-1, 64, 32, 32]               0
        MaxPool2d-15          [-1, 64, 16, 16]               0
           Conv2d-16          [-1, 64, 16, 16]          36,928
             ReLU-17          [-1, 64, 16, 16]               0
        MaxPool2d-18            [-1, 64, 8, 8]               0
           Conv2d-19            [-1, 64, 8, 8]          36,928
             ReLU-20            [-1, 64, 8, 8]               0
        MaxPool2d-21            [-1, 64, 4, 4]               0
          Flatten-22                [-1, 1024]               0
           Linear-23                  [-1, 64]          65,600
             ReLU-24                  [-1, 64]               0
           Linear-25                  [-1, 11]             715
          Softmax-26                  [-1, 11]               0
================================================================
Total params: 233,419
Trainable params: 233,419
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 64.49
Params size (MB): 0.89
Estimated Total Size (MB): 66.13
----------------------------------------------------------------

'''
This model in it self when trained results in very low accuracies.
Hence, I thought of initiationg with pre trained weights to have a
better chgen of having a good accuracy

'''


# def train_model(model, train_loader, val_loader, criterion,
optimizer, num_epochs, device):
#     """
#     Trains and validates a PyTorch model, similar to model.fit in
Keras.

#     Args:
#         model: The PyTorch model to train.
#         train_loader: DataLoader for training data.
#         val_loader: DataLoader for validation data.
#         criterion: Loss function.
#         optimizer: Optimizer.
#         num_epochs: Number of epochs to train.
#         device: Device to run the model on ('cpu' or 'cuda').
```

```
#     Returns:
#         Trained model.
#     """
#     for epoch in range(num_epochs):
#         # Training phase
#         model.train()  # Set model to training mode
#         train_loss = 0.0
#         train_correct = 0
#         train_total = 0

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

#             # Zero the parameter gradients
#             optimizer.zero_grad()

#             # Forward pass
#             outputs = model(inputs)

#             # Compute loss
#             loss = criterion(outputs, labels)

#             # Backward pass and optimization
#             loss.backward()
#             optimizer.step()

#             # Accumulate loss and metrics
#             train_loss += loss.item() * inputs.size(0)
#             train_correct += (outputs.argmax(1) ==
labels).sum().item()
#             train_total += labels.size(0)

#         # Calculate average training loss and accuracy
#         avg_train_loss = train_loss / train_total
#         train_accuracy = train_correct / train_total

#         # Validation phase
#         model.eval()  # Set model to evaluation mode
#         val_loss = 0.0
#         val_correct = 0
#         val_total = 0

#         with torch.no_grad():
#             for inputs, labels in val_loader:
#                 inputs, labels = inputs.to(device),
labels.to(device)

#                 # Forward pass
#                 outputs = model(inputs)
```

```
#                   # Compute loss
#                   loss = criterion(outputs, labels)

#                   # Accumulate loss and metrics
#                   val_loss += loss.item() * inputs.size(0)
#                   val_correct += (outputs.argmax(1) ==
labels).sum().item()
#                   val_total += labels.size(0)

#         # Calculate average validation loss and accuracy
#         avg_val_loss = val_loss / val_total
#         val_accuracy = val_correct / val_total

#         # Print progress for the current epoch
#         print(f"Epoch {epoch+1}/{num_epochs}")
#         print(f"  Train Loss: {avg_train_loss:.4f}, Train Accuracy:
{train_accuracy:.4f}")
#         print(f"  Val Loss: {avg_val_loss:.4f}, Val Accuracy:
{val_accuracy:.4f}")

#     return model
```

```
'\nThis model in it self when trained results in very low accuracies.\
nHence, I thought of initiationg with pre trained weights to have a
better chgen of having a good accuracy\n\n'
```

```
from tqdm.notebook import tqdm
import gc
from datetime import datetime
import warnings
warnings.filterwarnings('ignore')

class EarlyStopping:
    def __init__(self, patience=7, min_delta=0,
restore_best_weights=True):
        self.patience = patience
        self.min_delta = min_delta
        self.restore_best_weights = restore_best_weights
        self.best_model = None
        self.best_loss = None
        self.counter = 0
        self.status = ""

    def __call__(self, val_loss, model):
        if self.best_loss is None:
            self.best_loss = val_loss
            self.best_model = model.state_dict().copy()
        elif val_loss > self.best_loss - self.min_delta:
```

```
            self.counter += 1
            self.status = f'EarlyStopping counter: {self.counter} out
of {self.patience}'
            if self.counter >= self.patience:
                self.status = f'EarlyStopping triggered after
{self.counter} epochs'
                return True
        else:
            self.best_loss = val_loss
            self.best_model = model.state_dict().copy()
            self.counter = 0
            self.status = 'Improved'
        return False

def train_model(model, train_loader, val_loader, criterion, optimizer,
num_epochs, device,
                scheduler=None, early_stopping_patience=5):
    """
    Trains and validates a PyTorch model with Colab-optimized
features.

    Args:
        model: The PyTorch model to train
        train_loader: DataLoader for training data
        val_loader: DataLoader for validation data
        criterion: Loss function
        optimizer: Optimizer
        num_epochs: Number of epochs to train
        device: Device to run the model on ('cpu' or 'cuda')
        scheduler: Optional learning rate scheduler
        early_stopping_patience: Number of epochs to wait before early
stopping

    Returns:
        Trained model and dictionary containing training history
    """
    # Initialize early stopping
    early_stopping = EarlyStopping(patience=early_stopping_patience)

    # Initialize history dictionary
    history = {
        'train_loss': [], 'val_loss': [],
        'train_acc': [], 'val_acc': []
    }

    # Get the start time
    start_time = datetime.now()

    print(f"Training started at {start_time.strftime('%H:%M:%S')}")
    print(f"Using device: {device}")
```

```
    try:
        for epoch in range(num_epochs):
            # Training phase
            model.train()
            train_loss = 0.0
            train_correct = 0
            train_total = 0

            # Use tqdm for progress bar
            train_pbar = tqdm(train_loader, desc=f'Epoch
{epoch+1}/{num_epochs} [Train]')

            for inputs, labels in train_pbar:
                inputs, labels = inputs.to(device), labels.to(device)

                # Clear gradients
                optimizer.zero_grad()

                # Forward pass
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                # Backward pass and optimization
                loss.backward()
                optimizer.step()

                # Accumulate metrics
                train_loss += loss.item() * inputs.size(0)
                train_correct += (outputs.argmax(1) ==
labels).sum().item()
                train_total += labels.size(0)

                # Update progress bar
                train_pbar.set_postfix({
                    'loss': f'{loss.item():.4f}',
                    'acc': f'{train_correct/train_total:.4f}'
                })

            # Validation phase
            model.eval()
            val_loss = 0.0
            val_correct = 0
            val_total = 0

            val_pbar = tqdm(val_loader, desc=f'Epoch
{epoch+1}/{num_epochs} [Valid]')

            with torch.no_grad():
                for inputs, labels in val_pbar:
```

```
                    inputs, labels = inputs.to(device),
labels.to(device)

                    outputs = model(inputs)
                    loss = criterion(outputs, labels)

                    val_loss += loss.item() * inputs.size(0)
                    val_correct += (outputs.argmax(1) ==
labels).sum().item()
                    val_total += labels.size(0)

                    val_pbar.set_postfix({
                        'loss': f'{loss.item():.4f}',
                        'acc': f'{val_correct/val_total:.4f}'
                    })

            # Calculate epoch metrics
            avg_train_loss = train_loss / train_total
            avg_val_loss = val_loss / val_total
            train_accuracy = train_correct / train_total
            val_accuracy = val_correct / val_total

            # Update history
            history['train_loss'].append(avg_train_loss)
            history['val_loss'].append(avg_val_loss)
            history['train_acc'].append(train_accuracy)
            history['val_acc'].append(val_accuracy)

            # Print epoch summary
            print(f"\nEpoch {epoch+1}/{num_epochs} Summary:")
            print(f"Train Loss: {avg_train_loss:.4f}, Train Acc:
{train_accuracy:.4f}")
            print(f"Val Loss: {avg_val_loss:.4f}, Val Acc:
{val_accuracy:.4f}")

            # Learning rate scheduler step
            if scheduler is not None:
                scheduler.step()
                print(f"Learning Rate: {optimizer.param_groups[0]
['lr']:.6f}")

            # Early stopping check
            if early_stopping(avg_val_loss, model):
                print("Early stopping triggered!")
                model.load_state_dict(early_stopping.best_model)
                break

            # Clear GPU memory
            torch.cuda.empty_cache()
            gc.collect()
```

```
    except KeyboardInterrupt:
        print("\nTraining interrupted by user!")
        if early_stopping.best_model is not None:
            print("Loading best model weights...")
            model.load_state_dict(early_stopping.best_model)

    # Calculate training time
    training_time = datetime.now() - start_time
    print(f"\nTraining completed in {training_time}")

    return model, history

# Initialize model, criterion, and optimizer with improved settings
for Colab
def initialize_training(model, learning_rate=1e-4):
    """
    Initialize training components with Colab-optimized settings
    """
    # Move model to GPU if available
    device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
    model = model.to(device)

    # Use mixed precision for faster training
    scaler = torch.cuda.amp.GradScaler()

    # Loss function with label smoothing for better generalization
    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

    # AdamW optimizer with weight decay
    optimizer = torch.optim.AdamW(model.parameters(),
                                 lr=learning_rate,
                                 weight_decay=0.01)

    # Cosine annealing scheduler with warm restarts
    scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
        optimizer,
        T_0=10,  # Reset LR every 10 epochs
        T_mult=2,  # Double the reset interval after each restart
        eta_min=1e-6  # Minimum learning rate
    )

    return device, criterion, optimizer, scheduler, scaler

class PretrainedClassifier(nn.Module):
    def __init__(self, num_classes, model_name='resnet50',
freeze_backbone=True):
        """
        Initialize model with pre-trained backbone
```

```
        Args:
            num_classes: Number of classes to predict
            model_name: Name of the pre-trained model to use
            freeze_backbone: Whether to freeze the backbone layers
        """
        super(PretrainedClassifier, self).__init__()

        # Available pretrained models
        self.available_models = {
            'resnet18': models.resnet18,
            'resnet50': models.resnet50,
            'efficientnet_b0': models.efficientnet_b0,
            'mobilenet_v3_large': models.mobilenet_v3_large,
            'convnext_small': models.convnext_small
        }

        if model_name not in self.available_models:
            raise ValueError(f"Model {model_name} not available.
Choose from: {list(self.available_models.keys())}")

        # Load pretrained model
        self.backbone = self.available_models[model_name]
(pretrained=True)

        # Freeze backbone if specified
        if freeze_backbone:
            for param in self.backbone.parameters():
                param.requires_grad = False

        # Replace the final layer based on model type
        if model_name.startswith('resnet'):
            in_features = self.backbone.fc.in_features
            self.backbone.fc = nn.Sequential(
                nn.Linear(in_features, 512),
                nn.BatchNorm1d(512),
                nn.ReLU(),
                nn.Dropout(0.3),
                nn.Linear(512, num_classes)
            )
        elif model_name.startswith('efficientnet'):
            in_features = self.backbone.classifier[-1].in_features
            self.backbone.classifier = nn.Sequential(
                nn.Linear(in_features, 512),
                nn.BatchNorm1d(512),
                nn.ReLU(),
                nn.Dropout(0.3),
                nn.Linear(512, num_classes)
            )
        elif model_name.startswith('mobilenet'):
```

```
            in_features = self.backbone.classifier[-1].in_features
            self.backbone.classifier = nn.Sequential(
                nn.Linear(in_features, 512),
                nn.BatchNorm1d(512),
                nn.ReLU(),
                nn.Dropout(0.3),
                nn.Linear(512, num_classes)
            )
        elif model_name.startswith('convnext'):
            in_features = self.backbone.classifier[-1].in_features
            self.backbone.classifier = nn.Sequential(
                nn.Linear(in_features, 512),
                nn.LayerNorm(512),
                nn.GELU(),
                nn.Dropout(0.3),
                nn.Linear(512, num_classes)
            )

    def forward(self, x):
        return self.backbone(x)

def initialize_training_pretrained(model, num_epochs, train_loader,
learning_rate=1e-4):
    """
    Initialize training components optimized for fine-tuning
    """
    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
    model = model.to(device)

    # Different learning rates for pretrained and new layers
    params = [
        {'params': [p for n, p in model.named_parameters() if
'backbone' not in n or 'fc' in n or 'classifier' in n],
         'lr': learning_rate},
        {'params': [p for n, p in model.named_parameters() if
'backbone' in n and 'fc' not in n and 'classifier' not in n],
         'lr': learning_rate/10}
    ]

    # Optimizer with weight decay
    optimizer = torch.optim.AdamW(params, weight_decay=0.01)

    # Loss function with label smoothing
    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)

    # Learning rate scheduler
    scheduler = torch.optim.lr_scheduler.OneCycleLR(
        optimizer,
        max_lr=[learning_rate, learning_rate/10],
```

```python
        epochs=num_epochs,
        steps_per_epoch=len(train_loader),
        pct_start=0.3,
        anneal_strategy='cos'
    )

    # Gradient scaler for mixed precision training
    scaler = torch.cuda.amp.GradScaler()

    return device, criterion, optimizer, scheduler, scaler

# Creating a model with pretrained weights

model = PretrainedClassifier(
    num_classes= NUM_CLASSES,
    model_name='resnet50',
    freeze_backbone=True    # freeze pretrained weights initially
)

# Initialize training components
device, criterion, optimizer, scheduler, scaler =
initialize_training_pretrained(
    model,
    num_epochs=EPOCH,
    train_loader=data_loader,
    learning_rate=1e-4
)



# Train as before
trained_model, history = train_model(
    model=model,
    train_loader=data_loader,
    val_loader=valid_loader,
    criterion=criterion,
    optimizer=optimizer,
    num_epochs=EPOCH,
    device=device,
    scheduler=scheduler,
    early_stopping_patience=5
)
```

```
Downloading: "https://download.pytorch.org/models/resnet50-
0676ba61.pth" to /root/.cache/torch/hub/checkpoints/resnet50-
0676ba61.pth
100%|██████████| 97.8M/97.8M [00:00<00:00, 223MB/s]

Training started at 03:22:11
Using device: cuda
```

{"model_id":"25cee0437c1a402ca0d300b7c26c4179","version_major":2,"version_minor":0}

{"model_id":"9f740369012442f2b83dcd683f4b756c","version_major":2,"version_minor":0}

```
Epoch 1/80 Summary:
Train Loss: 1.8778, Train Acc: 0.4621
Val Loss: 1.4052, Val Acc: 0.7316
Learning Rate: 0.000004
```

{"model_id":"c1927e10bd5c48e7893dd12e7ca40133","version_major":2,"version_minor":0}

{"model_id":"a1f9e6589c6b438693a9c0a40cd2b341","version_major":2,"version_minor":0}

```
Epoch 2/80 Summary:
Train Loss: 1.3769, Train Acc: 0.7012
Val Loss: 1.1670, Val Acc: 0.7944
Learning Rate: 0.000004
```

{"model_id":"8a636a48a10c441687929c470ebac982","version_major":2,"version_minor":0}

{"model_id":"0fdbe911b4754fff81f9841ab6e5f49f","version_major":2,"version_minor":0}

```
Epoch 3/80 Summary:
Train Loss: 1.2182, Train Acc: 0.7473
Val Loss: 1.0679, Val Acc: 0.8181
Learning Rate: 0.000004
```

{"model_id":"fef27562bbe54321a9384283df1a9eae","version_major":2,"version_minor":0}

{"model_id":"27366917de7c471db4598a0f9b86135c","version_major":2,"version_minor":0}

```
Epoch 4/80 Summary:
Train Loss: 1.1421, Train Acc: 0.7719
Val Loss: 1.0161, Val Acc: 0.8303
Learning Rate: 0.000004
```

{"model_id":"234b1c6f77d74e77af2ddc7dc914e947","version_major":2,"version_minor":0}

{"model_id":"a380cf99236041d381ba085cab7af22d","version_major":2,"version_minor":0}

```
Epoch 5/80 Summary:
Train Loss: 1.0902, Train Acc: 0.7907
Val Loss: 0.9650, Val Acc: 0.8534
Learning Rate: 0.000004
```

{"model_id":"f989e4808f9745d4bd577764f93c9029","version_major":2,"version_minor":0}

{"model_id":"4238798d500948e6a87c307257656bb3","version_major":2,"version_minor":0}

```
Epoch 6/80 Summary:
Train Loss: 1.0541, Train Acc: 0.8029
Val Loss: 0.9376, Val Acc: 0.8606
Learning Rate: 0.000004
```

{"model_id":"5b195ee52d8a44a19830f802d0c5a238","version_major":2,"version_minor":0}

{"model_id":"5c6d815c9a144edeab049029a7ba75db","version_major":2,"version_minor":0}

```
Epoch 7/80 Summary:
Train Loss: 1.0258, Train Acc: 0.8164
Val Loss: 0.9186, Val Acc: 0.8698
Learning Rate: 0.000004
```

{"model_id":"949eb9ae41894eb08a639cc2c08b0605","version_major":2,"version_minor":0}

{"model_id":"1520aa8487374ec4b284291a8cd9c197","version_major":2,"version_minor":0}

```
Epoch 8/80 Summary:
Train Loss: 1.0080, Train Acc: 0.8214
Val Loss: 0.9115, Val Acc: 0.8645
Learning Rate: 0.000004
```

{"model_id":"0c2a92b7a6ab477682d75e03382e490d","version_major":2,"version_minor":0}

{"model_id":"e2498e0abea04bf0bb8450d60d90a650","version_major":2,"version_minor":0}

```
Epoch 9/80 Summary:
Train Loss: 0.9883, Train Acc: 0.8296
```

```
Val Loss: 0.8950, Val Acc: 0.8749
Learning Rate: 0.000004
```
{"model_id":"13e27e318bba463d9abae259b667b323","version_major":2,"version_minor":0}

{"model_id":"bd436358739b4d2c936b41caf70c9d2f","version_major":2,"version_minor":0}

```
Epoch 10/80 Summary:
Train Loss: 0.9719, Train Acc: 0.8359
Val Loss: 0.8853, Val Acc: 0.8752
Learning Rate: 0.000004
```
{"model_id":"2affad513ba245db9d1e429586f7bda6","version_major":2,"version_minor":0}

{"model_id":"5166222c17184a09a7441c61b60fab4a","version_major":2,"version_minor":0}

```
Epoch 11/80 Summary:
Train Loss: 0.9597, Train Acc: 0.8405
Val Loss: 0.8676, Val Acc: 0.8845
Learning Rate: 0.000004
```
{"model_id":"11d62a72acbf4df7aa34f8bc922866a8","version_major":2,"version_minor":0}

{"model_id":"7ba1363c2fe64dd98dcb90060c9d7486","version_major":2,"version_minor":0}

```
Epoch 12/80 Summary:
Train Loss: 0.9488, Train Acc: 0.8485
Val Loss: 0.8627, Val Acc: 0.8839
Learning Rate: 0.000004
```
{"model_id":"0d33c207d26a4bd582661ea78d4b38d6","version_major":2,"version_minor":0}

{"model_id":"93a891579a8044ea90de75649e6c8b45","version_major":2,"version_minor":0}

```
Epoch 13/80 Summary:
Train Loss: 0.9386, Train Acc: 0.8534
Val Loss: 0.8524, Val Acc: 0.8941
Learning Rate: 0.000004
```
{"model_id":"9f266626022d4db8a342b0358a58b550","version_major":2,"version_minor":0}

{"model_id":"6834344fd907450e868b9fec5c58b165","version_major":2,"version_minor":0}

```
Epoch 14/80 Summary:
Train Loss: 0.9247, Train Acc: 0.8566
Val Loss: 0.8513, Val Acc: 0.8923
Learning Rate: 0.000004
```

{"model_id":"ee76c4847b3b431e9951cae71e1a7948","version_major":2,"version_minor":0}

{"model_id":"41d58d1aae7c418788f9b496c2fd1391","version_major":2,"version_minor":0}

```
Epoch 15/80 Summary:
Train Loss: 0.9176, Train Acc: 0.8626
Val Loss: 0.8471, Val Acc: 0.8911
Learning Rate: 0.000004
```

{"model_id":"a56a454b0c8e4cce88c99d5640812bb7","version_major":2,"version_minor":0}

{"model_id":"ad7db578e5194e6292dfcd0150935a94","version_major":2,"version_minor":0}

```
Epoch 16/80 Summary:
Train Loss: 0.9079, Train Acc: 0.8648
Val Loss: 0.8266, Val Acc: 0.9078
Learning Rate: 0.000004
```

{"model_id":"ebb78448a9024f92880e1992ff3136d3","version_major":2,"version_minor":0}

{"model_id":"1798b87da8b743b68f6c8d317487f840","version_major":2,"version_minor":0}

```
Epoch 17/80 Summary:
Train Loss: 0.9006, Train Acc: 0.8696
Val Loss: 0.8227, Val Acc: 0.9045
Learning Rate: 0.000004
```

{"model_id":"5db04f172704436ba73be12f09ab4111","version_major":2,"version_minor":0}

{"model_id":"2fa82a6934ba4f92ba963579295d5bd6","version_major":2,"version_minor":0}

```
Epoch 18/80 Summary:
```

```
Train Loss: 0.8942, Train Acc: 0.8711
Val Loss: 0.8184, Val Acc: 0.9057
Learning Rate: 0.000004
```

{"model_id":"86d4049ec7584ab4865132d7a1a1869c","version_major":2,"version_minor":0}

{"model_id":"a228801065c744579aa1e2f682fe9ac7","version_major":2,"version_minor":0}

```
Epoch 19/80 Summary:
Train Loss: 0.8869, Train Acc: 0.8755
Val Loss: 0.8119, Val Acc: 0.9102
Learning Rate: 0.000004
```

{"model_id":"d218102c514c4b58abddff785eb897d3","version_major":2,"version_minor":0}

{"model_id":"5a2b447ce0aa40be8c3e58e6b945757c","version_major":2,"version_minor":0}

```
Epoch 20/80 Summary:
Train Loss: 0.8803, Train Acc: 0.8770
Val Loss: 0.8128, Val Acc: 0.9081
Learning Rate: 0.000004
```

{"model_id":"2403c13611054606bd9e3bc42a917879","version_major":2,"version_minor":0}

{"model_id":"4499a7f8ce48474bb579944373b05dde","version_major":2,"version_minor":0}

```
Epoch 21/80 Summary:
Train Loss: 0.8763, Train Acc: 0.8807
Val Loss: 0.8016, Val Acc: 0.9192
Learning Rate: 0.000004
```

{"model_id":"a4d43e90de354552ad6d726520019150","version_major":2,"version_minor":0}

{"model_id":"e61d7589cd9c46dcbe8f258c727f1791","version_major":2,"version_minor":0}

```
Epoch 22/80 Summary:
Train Loss: 0.8699, Train Acc: 0.8843
Val Loss: 0.7988, Val Acc: 0.9174
Learning Rate: 0.000004
```

{"model_id":"c21d236987fa41158ed423bdd369518c","version_major":2,"version_minor":0}

{"model_id":"54122c1e449341b2ad5b3812ab25f811","version_major":2,"version_minor":0}

```
Epoch 23/80 Summary:
Train Loss: 0.8652, Train Acc: 0.8857
Val Loss: 0.8032, Val Acc: 0.9147
Learning Rate: 0.000004
```

{"model_id":"337a9f42bdcc4e8e849a185b3b584071","version_major":2,"version_minor":0}

{"model_id":"15306839d0de42738f4db0899ef9b07e","version_major":2,"version_minor":0}

```
Epoch 24/80 Summary:
Train Loss: 0.8603, Train Acc: 0.8882
Val Loss: 0.7888, Val Acc: 0.9234
Learning Rate: 0.000004
```

{"model_id":"3044797ae2394a0d957c27002c94d8b3","version_major":2,"version_minor":0}

{"model_id":"ffc11afd3ca2436b90e7b19c5e9803ef","version_major":2,"version_minor":0}

```
Epoch 25/80 Summary:
Train Loss: 0.8536, Train Acc: 0.8921
Val Loss: 0.7936, Val Acc: 0.9183
Learning Rate: 0.000004
```

{"model_id":"df0c9dfdf4c2409e86b73c5f4ef00fd1","version_major":2,"version_minor":0}

{"model_id":"dddbf2cf245944f29e6326d2e40dad77","version_major":2,"version_minor":0}

```
Epoch 26/80 Summary:
Train Loss: 0.8475, Train Acc: 0.8939
Val Loss: 0.7835, Val Acc: 0.9222
Learning Rate: 0.000004
```

{"model_id":"70293ef685854e89b93f782c41c67b4a","version_major":2,"version_minor":0}

{"model_id":"df1ea72514184271a9611c0b2bc1614d","version_major":2,"version_minor":0}

```
Epoch 27/80 Summary:
Train Loss: 0.8446, Train Acc: 0.8950
Val Loss: 0.7771, Val Acc: 0.9252
Learning Rate: 0.000004
```

{"model_id":"200e8130387d413badd9ef79d2ede817","version_major":2,"version_minor":0}

{"model_id":"5b8747619e194e8aa431e1a352a065e4","version_major":2,"version_minor":0}

```
Epoch 28/80 Summary:
Train Loss: 0.8385, Train Acc: 0.8982
Val Loss: 0.7754, Val Acc: 0.9294
Learning Rate: 0.000004
```

{"model_id":"707098f6f37a431c95b99af304772138","version_major":2,"version_minor":0}

{"model_id":"908f70e776f945d3bbe8c605860dd8ba","version_major":2,"version_minor":0}

```
Epoch 29/80 Summary:
Train Loss: 0.8320, Train Acc: 0.9036
Val Loss: 0.7764, Val Acc: 0.9279
Learning Rate: 0.000004
```

{"model_id":"85d5035811bd425082a21d9d4d45246c","version_major":2,"version_minor":0}

{"model_id":"e8256a00bde548f7a15cd52b1cf4f706","version_major":2,"version_minor":0}

```
Epoch 30/80 Summary:
Train Loss: 0.8326, Train Acc: 0.9026
Val Loss: 0.7805, Val Acc: 0.9252
Learning Rate: 0.000004
```

{"model_id":"c66aa0db3ee14e7aa0096e8d74f82433","version_major":2,"version_minor":0}

{"model_id":"16aa6ea4bfc14022b1a4e7bb85cc726a","version_major":2,"version_minor":0}

```
Epoch 31/80 Summary:
Train Loss: 0.8255, Train Acc: 0.9026
```

```
Val Loss: 0.7639, Val Acc: 0.9303
Learning Rate: 0.000004
```

{"model_id":"a81d35e4adf14fd48a060c7842e4d22a","version_major":2,"version_minor":0}

{"model_id":"de119b736a784490b8b3289a75048b35","version_major":2,"version_minor":0}

```
Epoch 32/80 Summary:
Train Loss: 0.8212, Train Acc: 0.9079
Val Loss: 0.7623, Val Acc: 0.9342
Learning Rate: 0.000004
```

{"model_id":"b552a195dd6345a58edb2da033b075a2","version_major":2,"version_minor":0}

{"model_id":"856f7b3464aa4d148ea77adf786fde9b","version_major":2,"version_minor":0}

```
Epoch 33/80 Summary:
Train Loss: 0.8194, Train Acc: 0.9071
Val Loss: 0.7634, Val Acc: 0.9303
Learning Rate: 0.000004
```

{"model_id":"7f05e8c62dc042c28c875a2ba219b38d","version_major":2,"version_minor":0}

{"model_id":"3fff37b812bb4830a3ffb0f7d60fd1d8","version_major":2,"version_minor":0}

```
Epoch 34/80 Summary:
Train Loss: 0.8144, Train Acc: 0.9106
Val Loss: 0.7597, Val Acc: 0.9345
Learning Rate: 0.000004
```

{"model_id":"b8f8f49d0c4e4d9b9d201963dc79d9be","version_major":2,"version_minor":0}

{"model_id":"b0a410a79de544d0820a36c17a810bc5","version_major":2,"version_minor":0}

```
Epoch 35/80 Summary:
Train Loss: 0.8112, Train Acc: 0.9118
Val Loss: 0.7573, Val Acc: 0.9330
Learning Rate: 0.000004
```

{"model_id":"566474a53d2946c69e16304cbc41b2df","version_major":2,"version_minor":0}

```
{"model_id":"eb60f3efaccd4d6593ebde86bf9c30f9","version_major":2,"version_minor":0}


Epoch 36/80 Summary:
Train Loss: 0.8098, Train Acc: 0.9139
Val Loss: 0.7585, Val Acc: 0.9366
Learning Rate: 0.000004

{"model_id":"9f28fb7cc5fb493195bff64ac2686d5c","version_major":2,"version_minor":0}

{"model_id":"fca4a45152ba4f668f7fa75f862864e0","version_major":2,"version_minor":0}


Epoch 37/80 Summary:
Train Loss: 0.8065, Train Acc: 0.9163
Val Loss: 0.7578, Val Acc: 0.9348
Learning Rate: 0.000004

{"model_id":"e61e9f1d106c4603aed5a71380bacc83","version_major":2,"version_minor":0}

{"model_id":"ba50470181ea41fa9632cad3ff0c84e0","version_major":2,"version_minor":0}


Epoch 38/80 Summary:
Train Loss: 0.8036, Train Acc: 0.9145
Val Loss: 0.7539, Val Acc: 0.9321
Learning Rate: 0.000004

{"model_id":"d059d49bfe3b45beb79d9855042b43da","version_major":2,"version_minor":0}

{"model_id":"cc08aedd34354d47afb2088b6f4ce9d1","version_major":2,"version_minor":0}


Epoch 39/80 Summary:
Train Loss: 0.8003, Train Acc: 0.9179
Val Loss: 0.7427, Val Acc: 0.9417
Learning Rate: 0.000004

{"model_id":"b248b441408649e9afb45e1fec579e68","version_major":2,"version_minor":0}

{"model_id":"f4bfb695f4064aa1a45f0a744f350e04","version_major":2,"version_minor":0}


Epoch 40/80 Summary:
```

```
Train Loss: 0.7918, Train Acc: 0.9197
Val Loss: 0.7438, Val Acc: 0.9405
Learning Rate: 0.000004
```

{"model_id":"052301b4a8b8471383f8f2006580715c","version_major":2,"version_minor":0}

{"model_id":"8469ba6e73f74b8b900d2ff821e04f81","version_major":2,"version_minor":0}

```
Epoch 41/80 Summary:
Train Loss: 0.7899, Train Acc: 0.9221
Val Loss: 0.7408, Val Acc: 0.9405
Learning Rate: 0.000004
```

{"model_id":"c9a5f75db1a948e580e49dbe4de2195c","version_major":2,"version_minor":0}

{"model_id":"e884047877c8482597539e87104c9643","version_major":2,"version_minor":0}

```
Epoch 42/80 Summary:
Train Loss: 0.7924, Train Acc: 0.9206
Val Loss: 0.7481, Val Acc: 0.9381
Learning Rate: 0.000004
```

{"model_id":"b2bc8537f333483392e6deb22493b13d","version_major":2,"version_minor":0}

{"model_id":"fd62c2cf0efd405ba800933ed49cd7d1","version_major":2,"version_minor":0}

```
Epoch 43/80 Summary:
Train Loss: 0.7848, Train Acc: 0.9238
Val Loss: 0.7416, Val Acc: 0.9405
Learning Rate: 0.000004
```

{"model_id":"d08ef04136a4464b95b12e69c45a28c3","version_major":2,"version_minor":0}

{"model_id":"c0c7db8ae05c43428bf163fcca248574","version_major":2,"version_minor":0}

```
Epoch 44/80 Summary:
Train Loss: 0.7852, Train Acc: 0.9262
Val Loss: 0.7391, Val Acc: 0.9425
Learning Rate: 0.000004
```

{"model_id":"2f3f097263c14c53b7314299d4cad7e1","version_major":2,"version_minor":0}

{"model_id":"e28caf6ca2d04dca9f1a667b4fe27824","version_major":2,"version_minor":0}

```
Epoch 45/80 Summary:
Train Loss: 0.7795, Train Acc: 0.9263
Val Loss: 0.7306, Val Acc: 0.9449
Learning Rate: 0.000004
```

{"model_id":"75fc03b80c1b484c9e1a2079b89fdcca","version_major":2,"version_minor":0}

{"model_id":"e20e5d31ea264e63b87976d6269ad10c","version_major":2,"version_minor":0}

```
Epoch 46/80 Summary:
Train Loss: 0.7793, Train Acc: 0.9287
Val Loss: 0.7392, Val Acc: 0.9405
Learning Rate: 0.000004
```

{"model_id":"fc7cafd041ee498981b1e98361c09772","version_major":2,"version_minor":0}

{"model_id":"0c77b8f7b89c4d9c8a64ee28dfaac72e","version_major":2,"version_minor":0}

```
Epoch 47/80 Summary:
Train Loss: 0.7778, Train Acc: 0.9292
Val Loss: 0.7347, Val Acc: 0.9446
Learning Rate: 0.000004
```

{"model_id":"8db375046bdc4b189d819097b66ea541","version_major":2,"version_minor":0}

{"model_id":"0e5530168db1499d99262bc06ba147b7","version_major":2,"version_minor":0}

```
Epoch 48/80 Summary:
Train Loss: 0.7775, Train Acc: 0.9301
Val Loss: 0.7284, Val Acc: 0.9476
Learning Rate: 0.000004
```

{"model_id":"388fce329eb34032869eb9f2896a3404","version_major":2,"version_minor":0}

{"model_id":"7463b1f8983c477787b25f72e1fbf2e3","version_major":2,"version_minor":0}

```
Epoch 49/80 Summary:
Train Loss: 0.7720, Train Acc: 0.9318
Val Loss: 0.7315, Val Acc: 0.9431
Learning Rate: 0.000004
```

{"model_id":"59f7be67f6e64f7cad571af568b019cd","version_major":2,"version_minor":0}

{"model_id":"44e292ae464644369eae2616264b9e43","version_major":2,"version_minor":0}

```
Epoch 50/80 Summary:
Train Loss: 0.7699, Train Acc: 0.9328
Val Loss: 0.7240, Val Acc: 0.9494
Learning Rate: 0.000004
```

{"model_id":"e5e8f13b17c24ebc9436687a7e40cfe1","version_major":2,"version_minor":0}

{"model_id":"35b190a86611468aa01bb3a34d7b986c","version_major":2,"version_minor":0}

```
Epoch 51/80 Summary:
Train Loss: 0.7657, Train Acc: 0.9354
Val Loss: 0.7241, Val Acc: 0.9524
Learning Rate: 0.000004
```

{"model_id":"2b4b8938e1224ce88f051130fb0e0ec8","version_major":2,"version_minor":0}

{"model_id":"534f1350660142babdb388439ff5ae19","version_major":2,"version_minor":0}

```
Epoch 52/80 Summary:
Train Loss: 0.7643, Train Acc: 0.9356
Val Loss: 0.7205, Val Acc: 0.9467
Learning Rate: 0.000004
```

{"model_id":"b46706f8ce4a43508dbd1e6e49436db5","version_major":2,"version_minor":0}

{"model_id":"977302b6ef1d4966be2510c83df1f865","version_major":2,"version_minor":0}

```
Epoch 53/80 Summary:
Train Loss: 0.7658, Train Acc: 0.9329
```

```
Val Loss: 0.7216, Val Acc: 0.9515
Learning Rate: 0.000004
```

{"model_id":"e8418024c58b4c3cb80416f991413f52","version_major":2,"version_minor":0}

{"model_id":"7e005a68731b46dca93b9decda74743e","version_major":2,"version_minor":0}

```
Epoch 54/80 Summary:
Train Loss: 0.7562, Train Acc: 0.9398
Val Loss: 0.7271, Val Acc: 0.9443
Learning Rate: 0.000004
```

{"model_id":"e11ebd55a25847eda26a36d879dec02b","version_major":2,"version_minor":0}

{"model_id":"6569e3fe246748aca5522a484b96b368","version_major":2,"version_minor":0}

```
Epoch 55/80 Summary:
Train Loss: 0.7553, Train Acc: 0.9412
Val Loss: 0.7183, Val Acc: 0.9518
Learning Rate: 0.000004
```

{"model_id":"dcf9a43ef7c64229894944be5135eb73","version_major":2,"version_minor":0}

{"model_id":"39edabe8172f415183dbf3a967dc42c2","version_major":2,"version_minor":0}

```
Epoch 56/80 Summary:
Train Loss: 0.7571, Train Acc: 0.9383
Val Loss: 0.7168, Val Acc: 0.9542
Learning Rate: 0.000004
```

{"model_id":"0eb974f4754747ff8cdc1dbe58b7c158","version_major":2,"version_minor":0}

{"model_id":"fde83b2748804801825220abb028823e","version_major":2,"version_minor":0}

```
Epoch 57/80 Summary:
Train Loss: 0.7561, Train Acc: 0.9380
Val Loss: 0.7167, Val Acc: 0.9527
Learning Rate: 0.000004
```

{"model_id":"3b1b3c43dfe745158f06773eb055713f","version_major":2,"version_minor":0}

```
{"model_id":"920f7fa776f341f484ddf900f21e5599","version_major":2,"version_minor":0}


Epoch 58/80 Summary:
Train Loss: 0.7509, Train Acc: 0.9409
Val Loss: 0.7106, Val Acc: 0.9551
Learning Rate: 0.000004

{"model_id":"82c8e22fcaa34a28aa8db35b028eca87","version_major":2,"version_minor":0}

{"model_id":"77aa22d88f1f49a09cfade023fdd584b","version_major":2,"version_minor":0}


Epoch 59/80 Summary:
Train Loss: 0.7511, Train Acc: 0.9421
Val Loss: 0.7133, Val Acc: 0.9506
Learning Rate: 0.000004

{"model_id":"45781552321e43cf97e6ea5e8d22e147","version_major":2,"version_minor":0}

{"model_id":"83b7f3005397445cade064a581ccf598","version_major":2,"version_minor":0}


Epoch 60/80 Summary:
Train Loss: 0.7472, Train Acc: 0.9436
Val Loss: 0.7092, Val Acc: 0.9584
Learning Rate: 0.000004

{"model_id":"b3f3a9f1f6c94c89b09a8ffe29ea390c","version_major":2,"version_minor":0}

{"model_id":"664a1749c5754c639eb70ff800135138","version_major":2,"version_minor":0}


Epoch 61/80 Summary:
Train Loss: 0.7467, Train Acc: 0.9425
Val Loss: 0.7063, Val Acc: 0.9545
Learning Rate: 0.000004

{"model_id":"cd8cb7ee63974a519fd3d6c670ccbe88","version_major":2,"version_minor":0}

{"model_id":"e18f812c56cc446d926e4e416aa69bc6","version_major":2,"version_minor":0}


Epoch 62/80 Summary:
```

```
Train Loss: 0.7416, Train Acc: 0.9470
Val Loss: 0.7105, Val Acc: 0.9530
Learning Rate: 0.000004
```

{"model_id":"3d9b8a7888484daa97755b626f26938b","version_major":2,"version_minor":0}

{"model_id":"843b7c69f7044da18f92e1195256059a","version_major":2,"version_minor":0}

```
Epoch 63/80 Summary:
Train Loss: 0.7450, Train Acc: 0.9450
Val Loss: 0.7116, Val Acc: 0.9527
Learning Rate: 0.000004
```

{"model_id":"9b7c268136be4c6aa0b423c1b3f808ac","version_major":2,"version_minor":0}

{"model_id":"f5806c2d6ba248ec9369a8db71239735","version_major":2,"version_minor":0}

```
Epoch 64/80 Summary:
Train Loss: 0.7399, Train Acc: 0.9482
Val Loss: 0.7075, Val Acc: 0.9536
Learning Rate: 0.000004
```

{"model_id":"b9ca4ef473d44464be754ea0f3198b6a","version_major":2,"version_minor":0}

{"model_id":"ee1de921698842659674eef151f2d12b","version_major":2,"version_minor":0}

```
Epoch 65/80 Summary:
Train Loss: 0.7384, Train Acc: 0.9487
Val Loss: 0.7080, Val Acc: 0.9539
Learning Rate: 0.000004
```

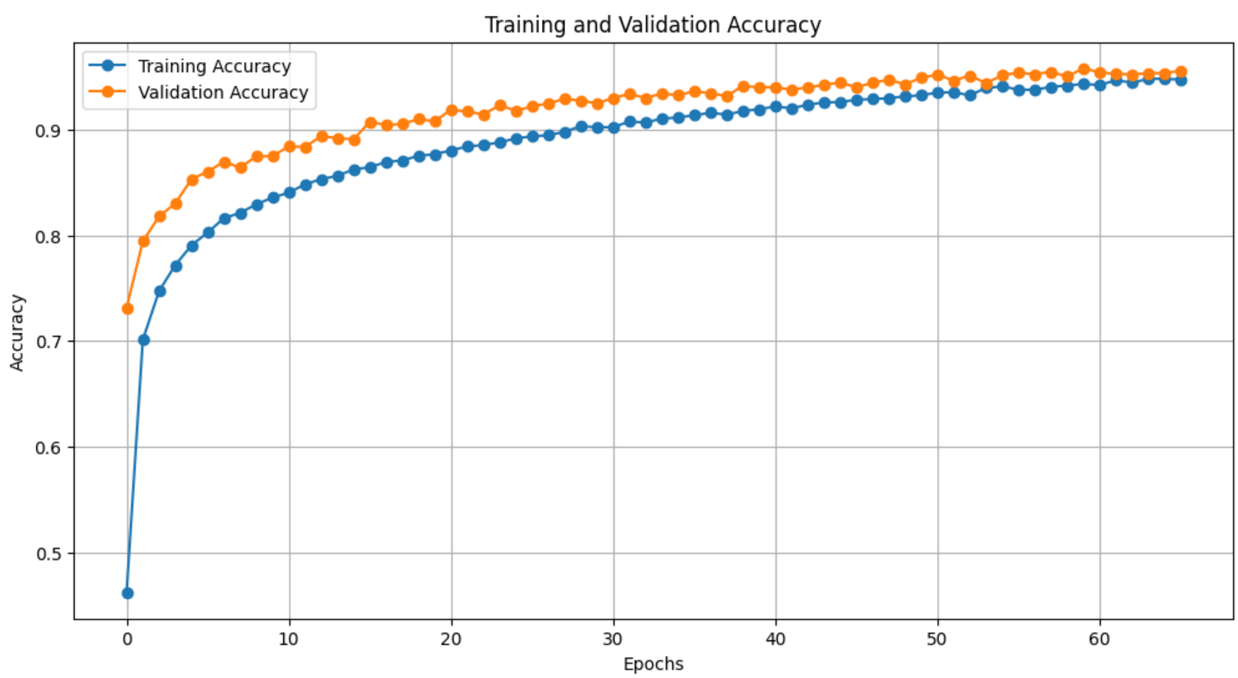
{"model_id":"c4b1cbc3061643aa8b7d455ba7996604","version_major":2,"version_minor":0}

{"model_id":"de271fb5d1124fa69ff3e23e14b4c48c","version_major":2,"version_minor":0}

```
Epoch 66/80 Summary:
Train Loss: 0.7391, Train Acc: 0.9475
Val Loss: 0.7076, Val Acc: 0.9560
Learning Rate: 0.000004
Early stopping triggered!
```

```
Training completed in 2:11:16.915932
```

```python
# Plot training and validation accuracy
plt.figure(figsize=(12, 6))
plt.plot(history["train_acc"], label="Training Accuracy", marker="o")
plt.plot(history["val_acc"], label="Validation Accuracy", marker="o")
plt.title("Training and Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.grid()
plt.show()

# Plot training and validation loss
plt.figure(figsize=(12, 6))
plt.plot(history["train_loss"], label="Training Loss", marker="o")
plt.plot(history["val_loss"], label="Validation Loss", marker="o")
plt.title("Training and Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid()
plt.show()
```




```
def evaluate_model(model, data_loader, device):
    """
    Evaluates the accuracy of the model on the given DataLoader.

    Args:
        model: Trained PyTorch model.
        data_loader: DataLoader for the dataset to evaluate.
        device: Device to run the model on ('cpu' or 'cuda').

    Returns:
        accuracy: Accuracy of the model on the dataset.
    """
    model.eval()  # Set the model to evaluation mode
    correct = 0
    total = 0

    with torch.no_grad():  # Disable gradient calculation for
evaluation
        for inputs, labels in data_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            # Forward pass
            outputs = model(inputs)

            # Predictions
            predicted = outputs.argmax(dim=1)

            # Count correctly classified samples
            correct += (predicted == labels).sum().item()
```

```
            total += labels.size(0)

    # Calculate accuracy
    accuracy = correct / total
    return accuracy

# Evaluate the model
true_accuracy = evaluate_model(trained_model, test_loader, device)
print(f"Test Accuracy: {true_accuracy:.4f}")
```

```
Test Accuracy: 0.9545
```

```
def visualize_prediction(model, data, device, class_names):
    """
    Displays a test image along with its actual and predicted class
probabilities.

    Args:
        model: Trained PyTorch model.
        data: A single (image, label) tuple from the dataset or
DataLoader.
        device: The device to run the model on ('cpu' or 'cuda').
        class_names: List of class names corresponding to the labels.
    """
    model.eval()  # Set the model to evaluation mode

    # Unpack the image and label
    img, label = data
    img, label = img.to(device), label.to(device)

    # Add batch dimension to the image if needed
    img_batch = img.unsqueeze(0)

    # Predict the class of the image
    with torch.no_grad():
        outputs = model(img_batch)
        probabilities = torch.nn.functional.softmax(outputs, dim=1)
        _, predicted = torch.max(outputs, 1)

    # Convert the image tensor to a NumPy array for visualization
    img_np = img.cpu().numpy().transpose((1, 2, 0))  # Convert to HWC
format
    img_np = np.clip(img_np, 0, 1)  # Ensure pixel values are in range
[0, 1]

    # Get actual and predicted class names and probabilities
    actual_class = class_names[label.item()]
    predicted_class = class_names[predicted.item()]
    predicted_prob = probabilities[0, predicted.item()].item() * 100
```

```
    # Plot the image with predicted and actual class
    plt.figure(figsize=(6, 6))
    plt.imshow(img_np)
    plt.title(f"Actual: {actual_class} \nPredicted: {predicted_class}
({predicted_prob:.2f}%)", fontsize=14)
    plt.axis('off')
    plt.show()

# Visualizing a single image from the test_loader
for img, label in test_loader:  # Get one batch of data
    visualize_prediction(trained_model, (img[0], label[0]), device,
CLASSES)  # Visualize the first image
    break
```

Actual: Septoria_leaf_spot
Predicted: Septoria_leaf_spot (97.30%)

```
#Downloading the model's state_dict
torch.save(trained_model.state_dict(),
```

```
"/kaggle/working/trained_model.pth")

#Also downloading the full model just in case
torch.save(trained_model,
"/kaggle/working/trained_model_complete.pth")
```