In [1]:
```python
import kagglehub

# Download latest version
path = kagglehub.dataset_download("sohansakib75/cotton-4-class")

print("Path to dataset files:", path)
```

```
Path to dataset files: /kaggle/input/cotton-4-class
```

In [2]:
```python
import os
import kagglehub

# Download dataset
path = kagglehub.dataset_download("sohansakib75/cotton-4-class")

print("Dataset root path:", path)

# List folders and files inside
print("Contents inside dataset folder:")
for item in os.listdir(path):
    item_path = os.path.join(path, item)
    if os.path.isdir(item_path):
        print(f" {item}/")
    else:
        print(f" {item}")
```

```
Dataset root path: /kaggle/input/cotton-4-class
Contents inside dataset folder:
 Cotton leaf/
```

In [3]:
```python
import os
import glob

dataset_path = os.path.join(path, "Cotton leaf")

print("Path to Dataset folder:", dataset_path)


for subdir in sorted(os.listdir(dataset_path)):
    subpath = os.path.join(dataset_path, subdir)
    if os.path.isdir(subpath):
        # Count images by common formats
        image_files = glob.glob(os.path.join(subpath, "*.jpg")) + \
                      glob.glob(os.path.join(subpath, "*.jpeg")) + \
                      glob.glob(os.path.join(subpath, "*.png"))
        print(f"{subdir}: {len(image_files)} images")
```

```
Path to Dataset folder: /kaggle/input/cotton-4-class/Cotton leaf
diseased cotton leaf: 346 images
diseased cotton plant: 921 images
fresh cotton leaf: 519 images
fresh cotton plant: 514 images
```

In [4]:
```python
import os
import glob
import matplotlib.pyplot as plt
import random

dataset_path = "/kaggle/input/cotton-4-class/Cotton leaf"

# Classes
classes = sorted(os.listdir(dataset_path))

# Plot 5 images per class
fig, axes = plt.subplots(len(classes), 5, figsize=(15, 12))

for i, cls in enumerate(classes):
    cls_path = os.path.join(dataset_path, cls)
    image_files = glob.glob(os.path.join(cls_path, "*.jpg")) + \
                  glob.glob(os.path.join(cls_path, "*.jpeg")) + \
                  glob.glob(os.path.join(cls_path, "*.png"))

    # Randomly pick 5 images
    sample_files = random.sample(image_files, 5)

    for j, img_path in enumerate(sample_files):
        img = plt.imread(img_path)
        axes[i, j].imshow(img)
        axes[i, j].axis("off")
        if j == 2:  # center column
            axes[i, j].set_title(cls, fontsize=10)

plt.tight_layout()
```

```
plt.show()
```

In [5]:

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import random

# Paths
input_dir = "/kaggle/input/cotton-4-class/Cotton leaf"
output_dir = "/kaggle/working/preprocessed_dataset"
os.makedirs(output_dir, exist_ok=True)

img_size = (224, 224)
num_samples = 5

classes = sorted(os.listdir(input_dir))
fig, axes = plt.subplots(len(classes), num_samples, figsize=(15, 3*len(classes)))

clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))

for i, cls in enumerate(classes):
    cls_path = os.path.join(input_dir, cls)
    if os.path.isdir(cls_path):
        save_cls_path = os.path.join(output_dir, cls)
        os.makedirs(save_cls_path, exist_ok=True)

        files = [f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))]
        sample_files = random.sample(files, min(num_samples, len(files)))

        for j, file in enumerate(files):
            img_path = os.path.join(cls_path, file)
            img = cv2.imread(img_path)

            # 1️⃣ Resize
            img_resized = cv2.resize(img, img_size)

            # 2️⃣ CLAHE on Y channel
            yuv = cv2.cvtColor(img_resized, cv2.COLOR_BGR2YUV)
            yuv[:,:,0] = clahe.apply(yuv[:,:,0])
```

```
            img_clahe = cv2.cvtColor(yuv, cv2.COLOR_YUV2BGR)

            # 3️⃣ Normalize to [0,1]
            img_final = img_clahe.astype(np.float32) / 255.0

            # Save preprocessed image
            save_img = (img_final * 255).astype(np.uint8)
            cv2.imwrite(os.path.join(save_cls_path, file), save_img)

            # Plot sample images
            if file in sample_files:
                axes[i, sample_files.index(file)].imshow(cv2.cvtColor(save_img, cv2.COLOR_BGR2RGB))
                axes[i, sample_files.index(file)].axis("off")
                if sample_files.index(file) == num_samples // 2:
                    axes[i, sample_files.index(file)].set_title(cls, fontsize=12)

plt.tight_layout()
plt.show()
print("\n Preprocessed images saved in:", output_dir)
```


```
Preprocessed images saved in: /kaggle/working/preprocessed_dataset
```

In [6]:

```
import os
import shutil
import random

# Preprocessed dataset path
preprocessed_dir = "/kaggle/working/preprocessed_dataset"

# Split paths
split_base = "/kaggle/working/cotton_split"
train_dir = os.path.join(split_base, "train")
val_dir   = os.path.join(split_base, "val")
test_dir  = os.path.join(split_base, "test")

# Create split folders
for d in [train_dir, val_dir, test_dir]:
    os.makedirs(d, exist_ok=True)

# Split ratios
train_ratio = 0.75
```

```python
val_ratio = 0.1
test_ratio = 0.15

classes = sorted(os.listdir(preprocessed_dir))

for cls in classes:
    cls_path = os.path.join(preprocessed_dir, cls)
    files = [f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))]
    random.shuffle(files)

    n_total = len(files)
    n_train = int(train_ratio * n_total)
    n_val = int(val_ratio * n_total)
    n_test = n_total - n_train - n_val

    splits = {
        train_dir: files[:n_train],
        val_dir: files[n_train:n_train+n_val],
        test_dir: files[n_train+n_val:]
    }

    for split_folder, split_files in splits.items():
        cls_split_path = os.path.join(split_folder, cls)
        os.makedirs(cls_split_path, exist_ok=True)
        for f in split_files:
            shutil.copy(os.path.join(cls_path, f), os.path.join(cls_split_path, f))

print(" Dataset split into 75-10-15 and saved in:", split_base)
```

```
 Dataset split into 75-10-15 and saved in: /kaggle/working/cotton_split
```

In [7]:

```python
import os

split_base = "/kaggle/working/cotton_split"
splits = ["train", "val", "test"]

for split in splits:
    split_path = os.path.join(split_base, split)
    print(f"\n {split.capitalize()} Split:")
    for cls in sorted(os.listdir(split_path)):
        cls_path = os.path.join(split_path, cls)
        num_images = len([f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))])
        print(f"{cls}: {num_images} images")
```

```
 Train Split:
diseased cotton leaf: 316 images
diseased cotton plant: 865 images
fresh cotton leaf: 487 images
fresh cotton plant: 481 images

 Val Split:
diseased cotton leaf: 64 images
diseased cotton plant: 171 images
fresh cotton leaf: 98 images
fresh cotton plant: 100 images

 Test Split:
diseased cotton leaf: 92 images
diseased cotton plant: 263 images
fresh cotton leaf: 145 images
fresh cotton plant: 149 images
```

In [8]:

```python
import os
import cv2
import numpy as np
import random

train_dir = "/kaggle/working/cotton_split/train"
aug_train_dir = "/kaggle/working/cotton_train_aug"
os.makedirs(aug_train_dir, exist_ok=True)

# Augmentation functions
def random_flip(img):
    flip_code = random.choice([-1, 0, 1])
    return cv2.flip(img, flip_code)

def random_rotate(img):
    angle = random.uniform(-25, 25)
    h, w = img.shape[:2]
    M = cv2.getRotationMatrix2D((w//2, h//2), angle, 1)
    return cv2.warpAffine(img, M, (w, h), borderMode=cv2.BORDER_REFLECT)

def random_zoom(img):
    zoom_factor = random.uniform(0.8, 1.2)
```

```python
    h, w = img.shape[:2]
    new_h, new_w = int(h*zoom_factor), int(w*zoom_factor)
    img_resized = cv2.resize(img, (new_w, new_h))
    if zoom_factor < 1:
        pad_h = (h - new_h) // 2
        pad_w = (w - new_w) // 2
        img_padded = cv2.copyMakeBorder(img_resized, pad_h, h-new_h-pad_h,
                                        pad_w, w-new_w-pad_w, cv2.BORDER_REFLECT)
        return img_padded
    else:
        start_h = (new_h - h)//2
        start_w = (new_w - w)//2
        return img_resized[start_h:start_h+h, start_w:start_w+w]

def random_brightness(img):
    factor = random.uniform(0.7, 1.3)
    img = img.astype(np.float32) * factor
    img = np.clip(img, 0, 255).astype(np.uint8)
    return img

augmentations = [random_flip, random_rotate, random_zoom, random_brightness]

# Apply augmentations
classes = sorted(os.listdir(train_dir))
for cls in classes:
    cls_path = os.path.join(train_dir, cls)
    save_cls_path = os.path.join(aug_train_dir, cls)
    os.makedirs(save_cls_path, exist_ok=True)

    for file in os.listdir(cls_path):
        if not file.lower().endswith((".jpg", ".jpeg", ".png")):
            continue
        img_path = os.path.join(cls_path, file)
        img = cv2.imread(img_path)

        # Save original
        cv2.imwrite(os.path.join(save_cls_path, file), img)

        # 3 random augmentations
        for k in range(3):
            aug_img = img.copy()
            aug_funcs = random.sample(augmentations, 2)
            for func in aug_funcs:
                aug_img = func(aug_img)
            filename, ext = os.path.splitext(file)
            aug_name = f"{filename}_aug{k+1}{ext}"
            cv2.imwrite(os.path.join(save_cls_path, aug_name), aug_img)

# Print image count per class
print("\n Image count per class after augmentation:")
for cls in classes:
    cls_path = os.path.join(aug_train_dir, cls)
    count = len([f for f in os.listdir(cls_path) if f.lower().endswith((".jpg", ".jpeg", ".png"))])
    print(f"{cls}: {count} images")

print(f"\n All train images and augmented images saved in: {aug_train_dir}")
```

```
 Image count per class after augmentation:
diseased cotton leaf: 1264 images
diseased cotton plant: 3460 images
fresh cotton leaf: 1948 images
fresh cotton plant: 1924 images

 All train images and augmented images saved in: /kaggle/working/cotton_train_aug
```

In [9]:
```
!pip install timm
```

```
Requirement already satisfied: timm in /usr/local/lib/python3.11/dist-packages (1.0.19)
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages (from timm) (2.6.0+cu124)
Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist-packages (from timm) (0.21.0+cu124)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.11/dist-packages (from timm) (6.0.3)
Requirement already satisfied: huggingface_hub in /usr/local/lib/python3.11/dist-packages (from timm) (1.0.0rc2)
Requirement already satisfied: safetensors in /usr/local/lib/python3.11/dist-packages (from timm) (0.5.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from huggingface_hub->timm)
(3.19.1)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.11/dist-packages (from huggingface_hub
->timm) (2025.9.0)
Requirement already satisfied: packaging>=20.9 in /usr/local/lib/python3.11/dist-packages (from huggingface_hub-
>timm) (25.0)
Requirement already satisfied: httpx<1,>=0.23.0 in /usr/local/lib/python3.11/dist-packages (from huggingface_hub
->timm) (0.28.1)
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.11/dist-packages (from huggingface_hub->ti
mm) (4.67.1)
Requirement already satisfied: typer-slim in /usr/local/lib/python3.11/dist-packages (from huggingface_hub->timm
```

```
) (0.19.2)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.11/dist-packages (from huggi
ngface_hub->timm) (4.15.0)
Requirement already satisfied: hf-xet<2.0.0,>=1.1.3 in /usr/local/lib/python3.11/dist-packages (from huggingface
_hub->timm) (1.1.10)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch->timm) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch->timm) (3.1.6)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from
torch->timm) (12.4.127)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (fr
om torch->timm) (12.4.127)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from
torch->timm) (12.4.127)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python3.11/dist-packages (from torc
h->timm) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /usr/local/lib/python3.11/dist-packages (from tor
ch->timm) (12.4.5.8)
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /usr/local/lib/python3.11/dist-packages (from torc
h->timm) (11.2.1.3)
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /usr/local/lib/python3.11/dist-packages (from t
orch->timm) (10.3.5.147)
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /usr/local/lib/python3.11/dist-packages (from t
orch->timm) (11.6.1.9)
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /usr/local/lib/python3.11/dist-packages (from
torch->timm) (12.3.1.170)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/python3.11/dist-packages (from to
rch->timm) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch->
timm) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from torch
->timm) (12.4.127)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /usr/local/lib/python3.11/dist-packages (from
torch->timm) (12.4.127)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packages (from torch->timm) (3.2.
0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch->timm) (1.13
.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1
->torch->timm) (1.3.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from torchvision->timm) (1.26.4
)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.11/dist-packages (from torchvisio
n->timm) (11.3.0)
Requirement already satisfied: anyio in /usr/local/lib/python3.11/dist-packages (from httpx<1,>=0.23.0->huggingf
ace_hub->timm) (4.11.0)
Requirement already satisfied: certifi in /usr/local/lib/python3.11/dist-packages (from httpx<1,>=0.23.0->huggin
gface_hub->timm) (2025.8.3)
Requirement already satisfied: httpcore==1.* in /usr/local/lib/python3.11/dist-packages (from httpx<1,>=0.23.0->
huggingface_hub->timm) (1.0.9)
Requirement already satisfied: idna in /usr/local/lib/python3.11/dist-packages (from httpx<1,>=0.23.0->huggingfa
ce_hub->timm) (3.10)
Requirement already satisfied: h11>=0.16 in /usr/local/lib/python3.11/dist-packages (from httpcore==1.*->httpx<1
,>=0.23.0->huggingface_hub->timm) (0.16.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch->t
imm) (3.0.2)
Requirement already satisfied: mkl_fft in /usr/local/lib/python3.11/dist-packages (from numpy->torchvision->timm
) (1.3.8)
Requirement already satisfied: mkl_random in /usr/local/lib/python3.11/dist-packages (from numpy->torchvision->t
imm) (1.2.4)
Requirement already satisfied: mkl_umath in /usr/local/lib/python3.11/dist-packages (from numpy->torchvision->ti
mm) (0.1.1)
Requirement already satisfied: mkl in /usr/local/lib/python3.11/dist-packages (from numpy->torchvision->timm) (2
025.2.0)
Requirement already satisfied: tbb4py in /usr/local/lib/python3.11/dist-packages (from numpy->torchvision->timm)
(2022.2.0)
Requirement already satisfied: mkl-service in /usr/local/lib/python3.11/dist-packages (from numpy->torchvision->
timm) (2.4.1)
Requirement already satisfied: click>=8.0.0 in /usr/local/lib/python3.11/dist-packages (from typer-slim->hugging
face_hub->timm) (8.3.0)
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.11/dist-packages (from anyio->httpx<1,>=0.
23.0->huggingface_hub->timm) (1.3.1)
Requirement already satisfied: intel-openmp<2026,>=2024 in /usr/local/lib/python3.11/dist-packages (from mkl->nu
mpy->torchvision->timm) (2024.2.0)
Requirement already satisfied: tbb==2022.* in /usr/local/lib/python3.11/dist-packages (from mkl->numpy->torchvis
ion->timm) (2022.2.0)
Requirement already satisfied: tcmlib==1.* in /usr/local/lib/python3.11/dist-packages (from tbb==2022.*->mkl->nu
mpy->torchvision->timm) (1.4.0)
Requirement already satisfied: intel-cmplr-lib-rt in /usr/local/lib/python3.11/dist-packages (from mkl_umath->nu
mpy->torchvision->timm) (2024.2.0)
Requirement already satisfied: intel-cmplr-lib-ur==2024.2.0 in /usr/local/lib/python3.11/dist-packages (from int
el-openmp<2026,>=2024->mkl->numpy->torchvision->timm) (2024.2.0)
```

In [10]:

```python
import timm
```

```
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schema.py:2225: UnsupportedFieldAttributeWarning: The 'repr' attribute with value False was provided to the `Field()` function, which has no effect in the context it was used. 'repr' is field-specific metadata, and can only be attached to a model field using `Annotated` metadata or by assignment. This may have happened because an `Annotated` type alias using the `type` statement was used, or if the `Field()` function was attached to a single member of a union type.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/pydantic/_internal/_generate_schema.py:2225: UnsupportedFieldAttributeWarning: The 'frozen' attribute with value True was provided to the `Field()` function, which has no effect in the context it was used. 'frozen' is field-specific metadata, and can only be attached to a model field using `Annotated` metadata or by assignment. This may have happened because an `Annotated` type alias using the `type` statement was used, or if the `Field()` function was attached to a single member of a union type.
  warnings.warn(
```

In [11]:

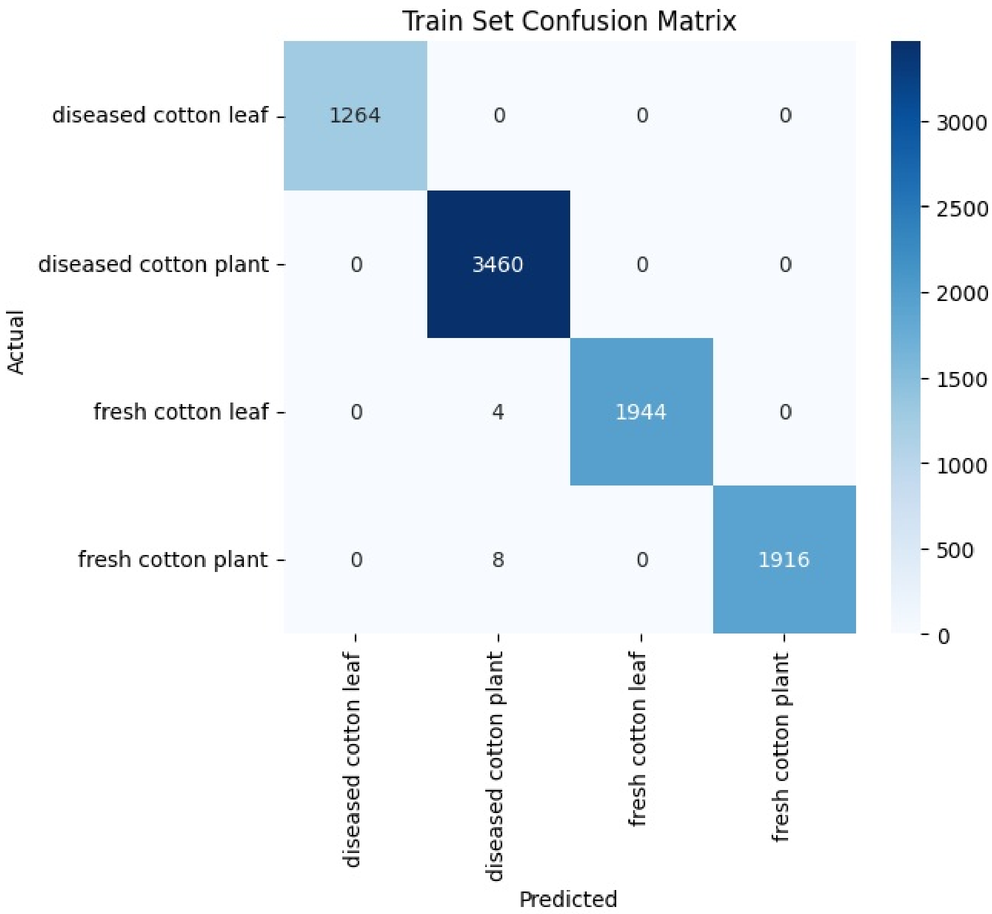
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from timm import create_model
from torch.optim.lr_scheduler import ReduceLROnPlateau

from sklearn.metrics import (
    classification_report, confusion_matrix, matthews_corrcoef,
    roc_auc_score, average_precision_score, roc_curve,
    precision_recall_curve, cohen_kappa_score
)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

# =========================
# Config for Cotton Dataset
# =========================
train_dir = "/kaggle/working/cotton_train_aug"  # Augmented train
val_dir   = "/kaggle/working/cotton_split/val"
test_dir  = "/kaggle/working/cotton_split/test"

batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["diseased cotton leaf", "diseased cotton plant", "fresh cotton leaf", "fresh cotton plant"]

# =========================
# Data Transforms & Dataloaders
# =========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=train_dir, transform=common_tfms)
val_ds   = datasets.ImageFolder(root=val_dir, transform=common_tfms)
test_ds  = datasets.ImageFolder(root=test_dir, transform=common_tfms)

train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=2)
val_loader   = DataLoader(val_ds, batch_size=batch_size, shuffle=False, num_workers=2)
test_loader  = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=2)

# =========================
# Model
# =========================
model = create_model("mobilevit_s", pretrained=True, num_classes=num_classes)
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-4)
scheduler = ReduceLROnPlateau(optimizer, mode="min", patience=2, factor=0.5, verbose=True)

# =========================
# Training Loop with Early Stopping
# =========================
best_val_loss = float("inf")
patience_counter = 0

for epoch in range(num_epochs):
    start_time = time.time()

    # ---- Train ----
    model.train()
```

```
    train_loss, train_correct = 0, 0
    for imgs, labels in train_loader:
        imgs, labels = imgs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * imgs.size(0)
        train_correct += (outputs.argmax(1) == labels).sum().item()

    train_loss /= len(train_loader.dataset)
    train_acc = train_correct / len(train_loader.dataset)

    # ---- Validation ----
    model.eval()
    val_loss, val_correct = 0, 0
    with torch.no_grad():
        for imgs, labels in val_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)

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

    val_loss /= len(val_loader.dataset)
    val_acc = val_correct / len(val_loader.dataset)

    scheduler.step(val_loss)

    elapsed = time.time() - start_time
    print(f"Epoch [{epoch+1}/{num_epochs}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f} "
          f"Time: {elapsed:.2f}s")

    # Early Stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        torch.save(model.state_dict(), "mobilevit_cotton_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as mobilevit_cotton_best.pth")

# ==========================
# Evaluation Functions
# ==========================
def plot_confusion_matrix(cm, classes, title="Confusion Matrix"):
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=classes, yticklabels=classes)
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(title)
    plt.show()


def print_report(y_true, y_pred, y_prob, title="", plot_curves=False):
    print(f"\n--- {title} ---")
    print(classification_report(y_true, y_pred, digits=4))

    cm = confusion_matrix(y_true, y_pred)
    plot_confusion_matrix(cm, class_names, title=f"{title} Confusion Matrix")

    mcc = matthews_corrcoef(y_true, y_pred)
    print(f"MCC: {mcc:.4f}")

    kappa = cohen_kappa_score(y_true, y_pred)
    print(f"Cohen's Kappa: {kappa:.4f}")

    # Class-wise NPV + PPV
    npv_list, ppv_list = [], []
    for i in range(len(cm)):
        TN = np.sum(np.delete(np.delete(cm, i, axis=0), i, axis=1))
        FN = np.sum(cm[i, :]) - cm[i, i]
        FP = np.sum(cm[:, i]) - cm[i, i]
```

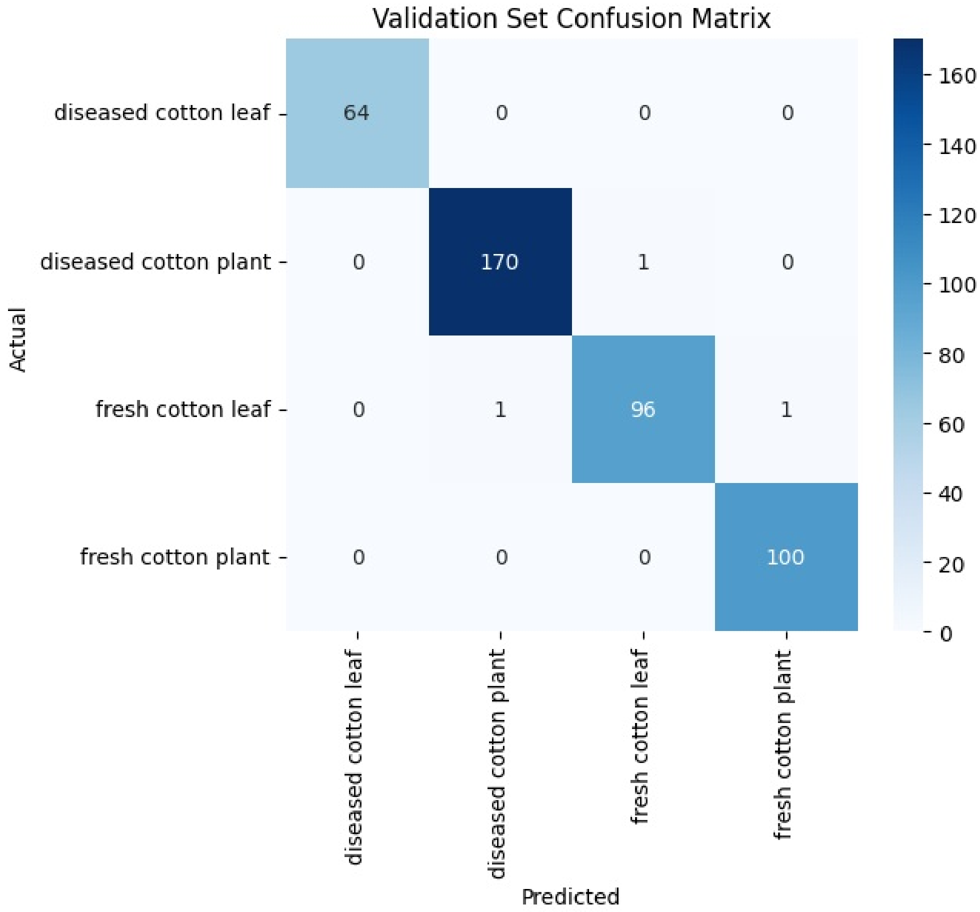
```
        TP = cm[i, i]

        NPV = TN / (TN + FN) if (TN + FN) > 0 else 0
        PPV = TP / (TP + FP) if (TP + FP) > 0 else 0
        npv_list.append(NPV)
        ppv_list.append(PPV)

    print(f"Mean NPV: {np.mean(npv_list):.4f}")
    print(f"Mean PPV (Precision): {np.mean(ppv_list):.4f}")

    if plot_curves:
        y_onehot = np.eye(num_classes)[y_true]
        roc_auc = roc_auc_score(y_onehot, y_prob, average='macro', multi_class='ovr')
        pr_auc = average_precision_score(y_onehot, y_prob, average='macro')
        print(f"ROC AUC: {roc_auc:.4f}, PR AUC: {pr_auc:.4f}")

        # ROC Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            fpr, tpr, _ = roc_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(fpr, tpr, label=f"{class_names[i]} (AUC={roc_auc_score(y_onehot[:,i], y_prob[:,i]):.4f})")
        plt.plot([0,1],[0,1],'k--')
        plt.title(f"{title} ROC Curve")
        plt.xlabel("FPR")
        plt.ylabel("TPR")
        plt.legend()
        plt.show()

        # PR Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            precision, recall, _ = precision_recall_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(recall, precision, label=f"{class_names[i]} (AP={average_precision_score(y_onehot[:,i], y_p
        plt.title(f"{title} Precision-Recall Curve")
        plt.xlabel("Recall")
        plt.ylabel("Precision")
        plt.legend()
        plt.show()

def evaluate_model(model, loader, title="", plot_curves=False):
    model.eval()
    y_true, y_pred, y_prob = [], [], []
    start_time = time.time()

    with torch.no_grad():
        for imgs, labels in loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            probs = torch.softmax(outputs, dim=1)
            preds = outputs.argmax(1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_prob.extend(probs.cpu().numpy())

    infer_time = time.time() - start_time
    y_true, y_pred, y_prob = np.array(y_true), np.array(y_pred), np.array(y_prob)

    print_report(y_true, y_pred, y_prob, title, plot_curves)
    print(f"{title} inference time: {infer_time:.2f} sec")
    return y_true, y_pred, y_prob, infer_time

# =========================
# Final Evaluation
# =========================
model.load_state_dict(torch.load("mobilevit_cotton_best.pth"))

y_true_train, y_pred_train, y_prob_train, train_infer_time = evaluate_model(model, train_loader, "Train Set", pl
y_true_val,   y_pred_val,   y_prob_val,   val_infer_time   = evaluate_model(model, val_loader, "Validation Set",
y_true_test,  y_pred_test,  y_prob_test,  test_infer_time  = evaluate_model(model, test_loader, "Test Set", plot

print("\n================ Summary ================")
print(f"Training inference time: {train_infer_time:.2f} sec")
print(f"Validation inference time: {val_infer_time:.2f} sec")
print(f"Test inference time: {test_infer_time:.2f} sec")
```

```
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is de
precated. Please use get_last_lr() to access the learning rate.
  warnings.warn(
```

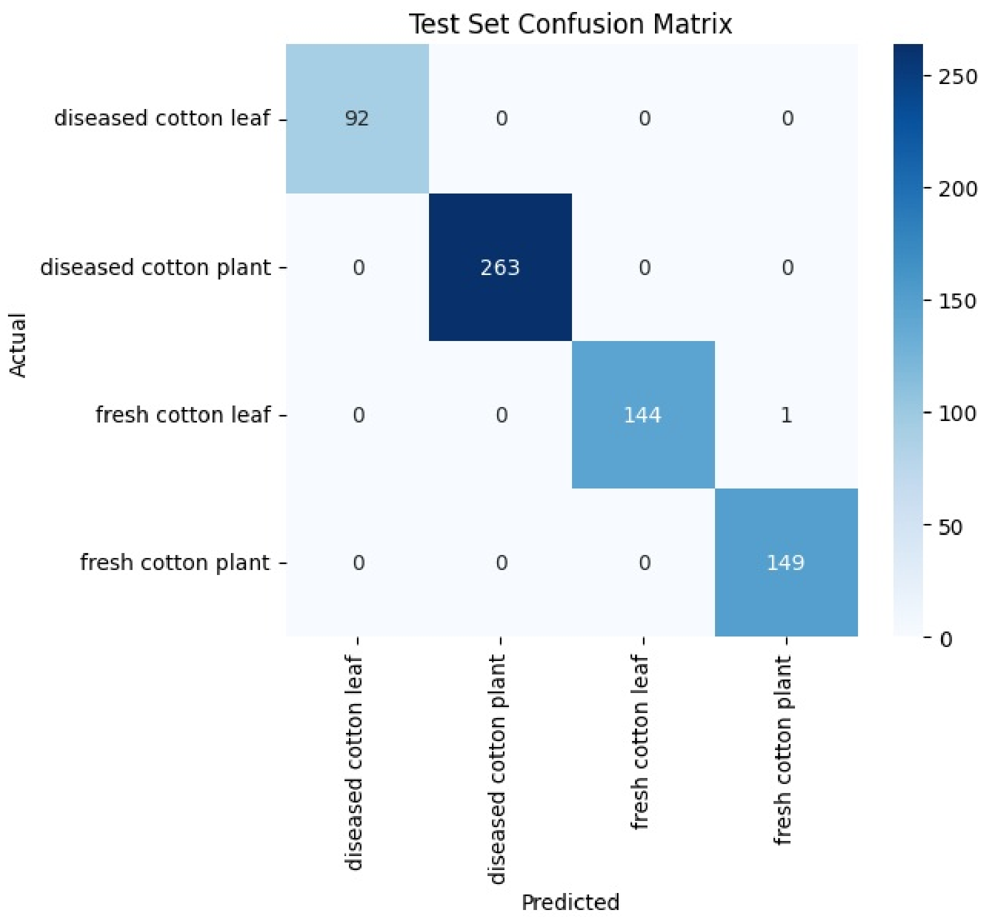
```
Epoch [1/35] Train Loss: 0.4105, Train Acc: 0.9267 Val Loss: 0.0459, Val Acc: 0.9908 Time: 50.33s
Epoch [2/35] Train Loss: 0.0619, Train Acc: 0.9869 Val Loss: 0.0157, Val Acc: 0.9931 Time: 49.89s
Epoch [3/35] Train Loss: 0.0336, Train Acc: 0.9929 Val Loss: 0.0193, Val Acc: 0.9931 Time: 50.00s
Epoch [4/35] Train Loss: 0.0278, Train Acc: 0.9933 Val Loss: 0.0182, Val Acc: 0.9954 Time: 50.18s

Epoch [5/35] Train Loss: 0.0126, Train Acc: 0.9970 Val Loss: 0.0169, Val Acc: 0.9977 Time: 50.06s
Epoch [6/35] Train Loss: 0.0096, Train Acc: 0.9980 Val Loss: 0.0204, Val Acc: 0.9954 Time: 50.04s
Epoch [7/35] Train Loss: 0.0070, Train Acc: 0.9990 Val Loss: 0.0185, Val Acc: 0.9954 Time: 50.21s
Early stopping triggered!
Training finished  Best model saved as mobilevit_cotton_best.pth

--- Train Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000      1264
           1     0.9965    1.0000    0.9983      3460
           2     1.0000    0.9979    0.9990      1948
           3     1.0000    0.9958    0.9979      1924

    accuracy                         0.9986      8596
   macro avg     0.9991    0.9984    0.9988      8596
weighted avg     0.9986    0.9986    0.9986      8596
```



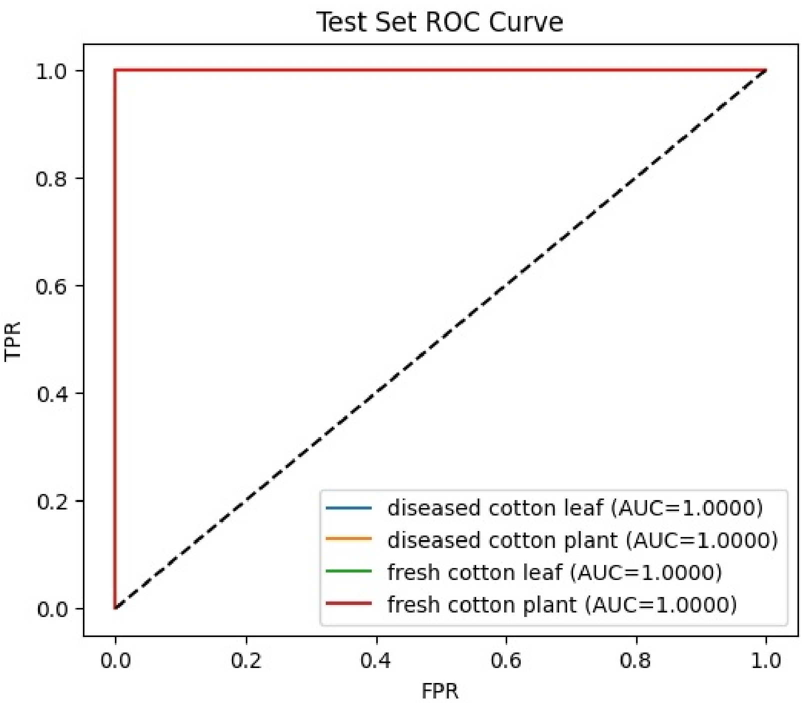

```
MCC: 0.9980
Cohen's Kappa: 0.9980
Mean NPV: 0.9996
Mean PPV (Precision): 0.9991
Train Set inference time: 15.54 sec

--- Validation Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        64
           1     0.9942    0.9942    0.9942       171
           2     0.9897    0.9796    0.9846        98
           3     0.9901    1.0000    0.9950       100

    accuracy                         0.9931       433
   macro avg     0.9935    0.9934    0.9934       433
weighted avg     0.9931    0.9931    0.9931       433
```

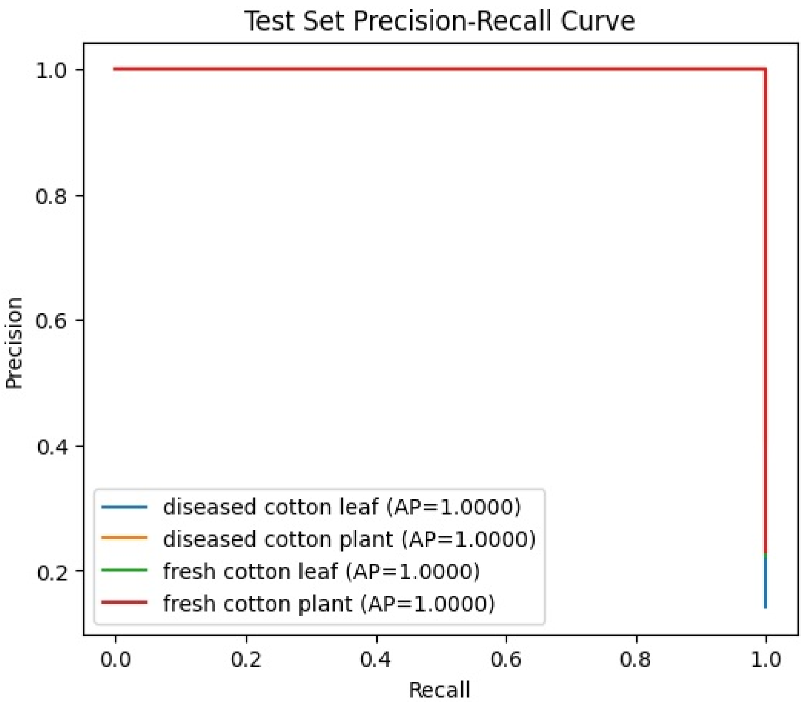



```
MCC: 0.9904
Cohen's Kappa: 0.9903
Mean NPV: 0.9976
Mean PPV (Precision): 0.9935
Validation Set inference time: 0.97 sec

--- Test Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        92
           1     1.0000    1.0000    1.0000       263
           2     1.0000    0.9931    0.9965       145
           3     0.9933    1.0000    0.9967       149

    accuracy                         0.9985       649
   macro avg     0.9983    0.9983    0.9983       649
weighted avg     0.9985    0.9985    0.9985       649
```




```
MCC: 0.9978
Cohen's Kappa: 0.9978
Mean NPV: 0.9995
Mean PPV (Precision): 0.9983
ROC AUC: 1.0000, PR AUC: 1.0000
```






```
Test Set inference time: 1.37 sec

================ Summary ================
Training inference time: 15.54 sec
Validation inference time: 0.97 sec
Test inference time: 1.37 sec
```

In [12]:
```
import torch
```

```python
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from timm import create_model
from torch.optim.lr_scheduler import ReduceLROnPlateau

from sklearn.metrics import (
    classification_report, confusion_matrix, matthews_corrcoef,
    roc_auc_score, average_precision_score, roc_curve,
    precision_recall_curve, cohen_kappa_score
)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

# =========================
# Config for Cotton Dataset
# =========================
train_dir = "/kaggle/working/cotton_train_aug"  # Augmented train
val_dir   = "/kaggle/working/cotton_split/val"
test_dir  = "/kaggle/working/cotton_split/test"

batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["diseased cotton leaf", "diseased cotton plant", "fresh cotton leaf", "fresh cotton plant"]

# =========================
# Data Transforms & Dataloaders
# =========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=train_dir, transform=common_tfms)
val_ds   = datasets.ImageFolder(root=val_dir, transform=common_tfms)
test_ds  = datasets.ImageFolder(root=test_dir, transform=common_tfms)

train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=2)
val_loader   = DataLoader(val_ds, batch_size=batch_size, shuffle=False, num_workers=2)
test_loader  = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=2)

# =========================
# Model
# =========================
model = create_model("tiny_vit_5m_224", pretrained=True, num_classes=num_classes)
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-4)
scheduler = ReduceLROnPlateau(optimizer, mode="min", patience=2, factor=0.5, verbose=True)

# =========================
# Training Loop with Early Stopping
# =========================
best_val_loss = float("inf")
patience_counter = 0

for epoch in range(num_epochs):
    start_time = time.time()

    # ---- Train ----
    model.train()
    train_loss, train_correct = 0, 0
    for imgs, labels in train_loader:
        imgs, labels = imgs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * imgs.size(0)
        train_correct += (outputs.argmax(1) == labels).sum().item()

    train_loss /= len(train_loader.dataset)
```

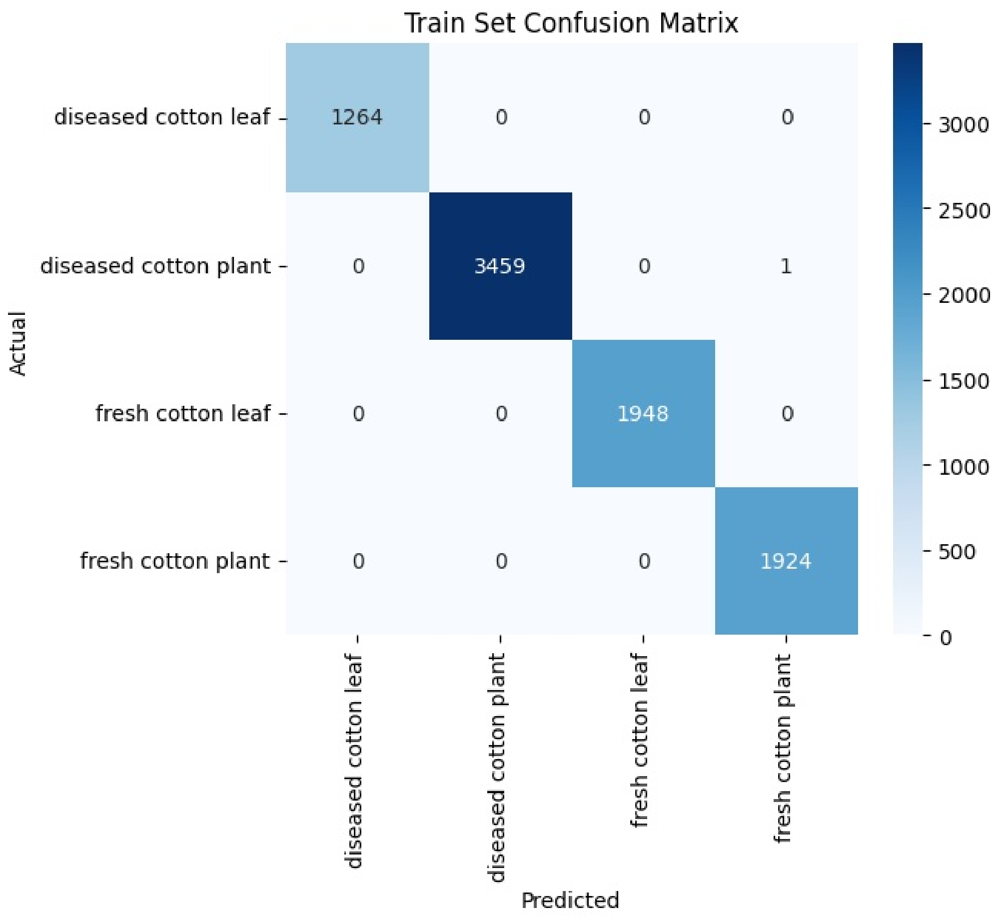
```python
    train_acc = train_correct / len(train_loader.dataset)

    # ---- Validation ----
    model.eval()
    val_loss, val_correct = 0, 0
    with torch.no_grad():
        for imgs, labels in val_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)

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

    val_loss /= len(val_loader.dataset)
    val_acc = val_correct / len(val_loader.dataset)

    scheduler.step(val_loss)

    elapsed = time.time() - start_time
    print(f"Epoch [{epoch+1}/{num_epochs}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f} "
          f"Time: {elapsed:.2f}s")

    # Early Stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        torch.save(model.state_dict(), "tiny_vit_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as tiny_vit_best.pth")

# =========================
# Evaluation Functions
# =========================
def plot_confusion_matrix(cm, classes, title="Confusion Matrix"):
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=classes, yticklabels=classes)
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(title)
    plt.show()


def print_report(y_true, y_pred, y_prob, title="", plot_curves=False):
    print(f"\n--- {title} ---")
    print(classification_report(y_true, y_pred, digits=4))

    cm = confusion_matrix(y_true, y_pred)
    plot_confusion_matrix(cm, class_names, title=f"{title} Confusion Matrix")

    mcc = matthews_corrcoef(y_true, y_pred)
    print(f"MCC: {mcc:.4f}")

    kappa = cohen_kappa_score(y_true, y_pred)
    print(f"Cohen's Kappa: {kappa:.4f}")

    # Class-wise NPV + PPV
    npv_list, ppv_list = [], []
    for i in range(len(cm)):
        TN = np.sum(np.delete(np.delete(cm, i, axis=0), i, axis=1))
        FN = np.sum(cm[i, :]) - cm[i, i]
        FP = np.sum(cm[:, i]) - cm[i, i]
        TP = cm[i, i]

        NPV = TN / (TN + FN) if (TN + FN) > 0 else 0
        PPV = TP / (TP + FP) if (TP + FP) > 0 else 0
        npv_list.append(NPV)
        ppv_list.append(PPV)

    print(f"Mean NPV: {np.mean(npv_list):.4f}")
    print(f"Mean PPV (Precision): {np.mean(ppv_list):.4f}")

    if plot_curves:
        y_onehot = np.eye(num_classes)[y_true]
        roc_auc = roc_auc_score(y_onehot, y_prob, average='macro', multi_class='ovr')
        pr_auc = average_precision_score(y_onehot, y_prob, average='macro')
```

```
        print(f"ROC AUC: {roc_auc:.4f}, PR AUC: {pr_auc:.4f}")

        # ROC Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            fpr, tpr, _ = roc_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(fpr, tpr, label=f"{class_names[i]} (AUC={roc_auc_score(y_onehot[:,i], y_prob[:,i]):.4f})")
        plt.plot([0,1],[0,1],'k--')
        plt.title(f"{title} ROC Curve")
        plt.xlabel("FPR")
        plt.ylabel("TPR")
        plt.legend()
        plt.show()

        # PR Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            precision, recall, _ = precision_recall_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(recall, precision, label=f"{class_names[i]} (AP={average_precision_score(y_onehot[:,i], y_p
        plt.title(f"{title} Precision-Recall Curve")
        plt.xlabel("Recall")
        plt.ylabel("Precision")
        plt.legend()
        plt.show()

def evaluate_model(model, loader, title="", plot_curves=False):
    model.eval()
    y_true, y_pred, y_prob = [], [], []
    start_time = time.time()

    with torch.no_grad():
        for imgs, labels in loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            probs = torch.softmax(outputs, dim=1)
            preds = outputs.argmax(1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_prob.extend(probs.cpu().numpy())

    infer_time = time.time() - start_time
    y_true, y_pred, y_prob = np.array(y_true), np.array(y_pred), np.array(y_prob)

    print_report(y_true, y_pred, y_prob, title, plot_curves)
    print(f"{title} inference time: {infer_time:.2f} sec")
    return y_true, y_pred, y_prob, infer_time

# =========================
# Final Evaluation
# =========================
model.load_state_dict(torch.load("tiny_vit_best.pth"))

y_true_train, y_pred_train, y_prob_train, train_infer_time = evaluate_model(model, train_loader, "Train Set", pl
y_true_val,   y_pred_val,   y_prob_val,   val_infer_time   = evaluate_model(model, val_loader, "Validation Set",
y_true_test,  y_pred_test,  y_prob_test,  test_infer_time  = evaluate_model(model, test_loader, "Test Set", plot

print("\n=============== Summary ===============")
print(f"Training inference time: {train_infer_time:.2f} sec")
print(f"Validation inference time: {val_infer_time:.2f} sec")
print(f"Test inference time: {test_infer_time:.2f} sec")
```

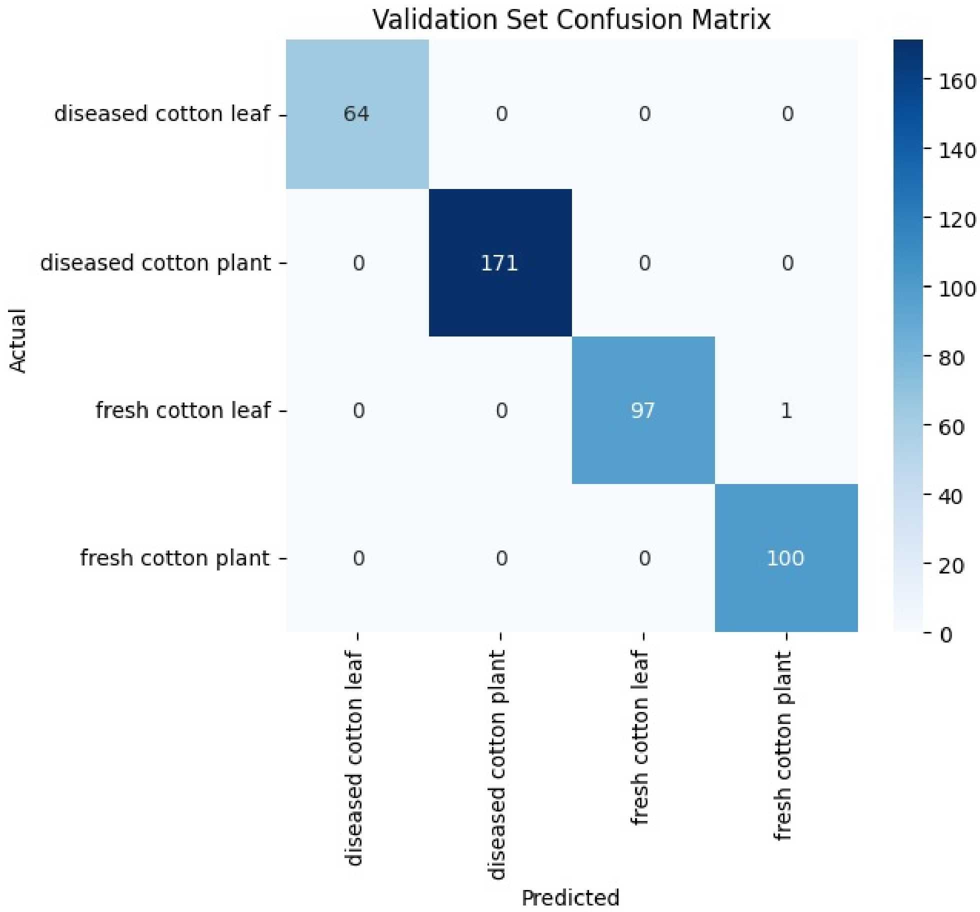
```
model.safetensors:   0%|          | 0.00/48.4M [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is de
precated. Please use get_last_lr() to access the learning rate.
  warnings.warn(
```

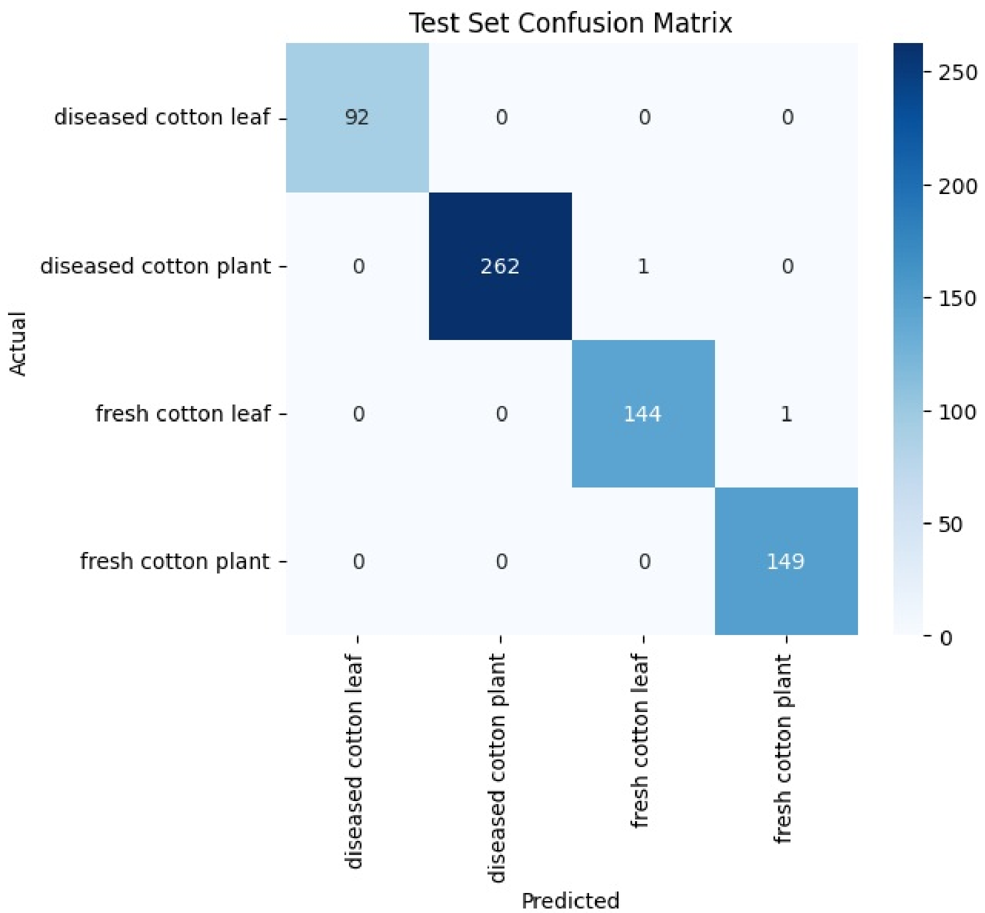
```
Epoch [1/35] Train Loss: 0.2403, Train Acc: 0.9359 Val Loss: 0.0451, Val Acc: 0.9931 Time: 39.18s
Epoch [2/35] Train Loss: 0.0320, Train Acc: 0.9929 Val Loss: 0.0309, Val Acc: 0.9931 Time: 39.02s
Epoch [3/35] Train Loss: 0.0201, Train Acc: 0.9948 Val Loss: 0.0203, Val Acc: 0.9954 Time: 39.10s
Epoch [4/35] Train Loss: 0.0083, Train Acc: 0.9984 Val Loss: 0.0248, Val Acc: 0.9908 Time: 38.87s
Epoch [5/35] Train Loss: 0.0094, Train Acc: 0.9973 Val Loss: 0.0267, Val Acc: 0.9931 Time: 38.88s
Epoch [6/35] Train Loss: 0.0059, Train Acc: 0.9987 Val Loss: 0.0175, Val Acc: 0.9954 Time: 38.98s
Epoch [7/35] Train Loss: 0.0036, Train Acc: 0.9991 Val Loss: 0.0165, Val Acc: 0.9977 Time: 39.24s
Epoch [8/35] Train Loss: 0.0040, Train Acc: 0.9986 Val Loss: 0.0170, Val Acc: 0.9977 Time: 39.03s
Epoch [9/35] Train Loss: 0.0073, Train Acc: 0.9978 Val Loss: 0.0245, Val Acc: 0.9954 Time: 38.90s
Epoch [10/35] Train Loss: 0.0043, Train Acc: 0.9988 Val Loss: 0.0433, Val Acc: 0.9908 Time: 39.07s
Epoch [11/35] Train Loss: 0.0024, Train Acc: 0.9994 Val Loss: 0.0188, Val Acc: 0.9977 Time: 38.99s
Epoch [12/35] Train Loss: 0.0007, Train Acc: 1.0000 Val Loss: 0.0187, Val Acc: 0.9977 Time: 39.02s
Early stopping triggered!
Training finished  Best model saved as tiny_vit_best.pth

--- Train Set ---
              precision    recall  f1-score   support


           0     1.0000    1.0000    1.0000      1264
           1     1.0000    0.9997    0.9999      3460
           2     1.0000    1.0000    1.0000      1948
           3     0.9995    1.0000    0.9997      1924

    accuracy                         0.9999      8596
   macro avg     0.9999    0.9999    0.9999      8596
weighted avg     0.9999    0.9999    0.9999      8596
```



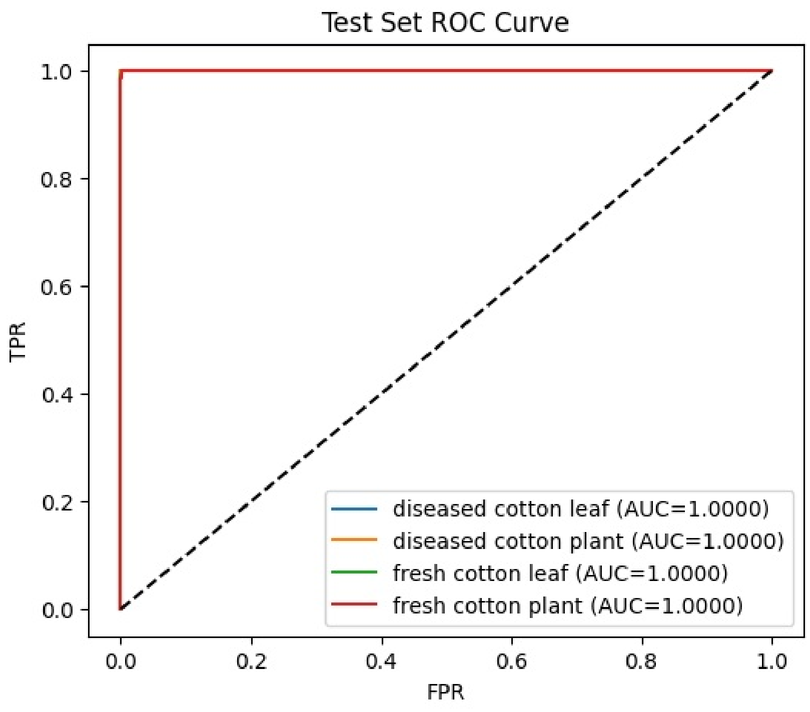

```
MCC: 0.9998
Cohen's Kappa: 0.9998
Mean NPV: 1.0000
Mean PPV (Precision): 0.9999
Train Set inference time: 12.94 sec

--- Validation Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        64
           1     1.0000    1.0000    1.0000       171
           2     1.0000    0.9898    0.9949        98
           3     0.9901    1.0000    0.9950       100

    accuracy                         0.9977       433
   macro avg     0.9975    0.9974    0.9975       433
weighted avg     0.9977    0.9977    0.9977       433
```

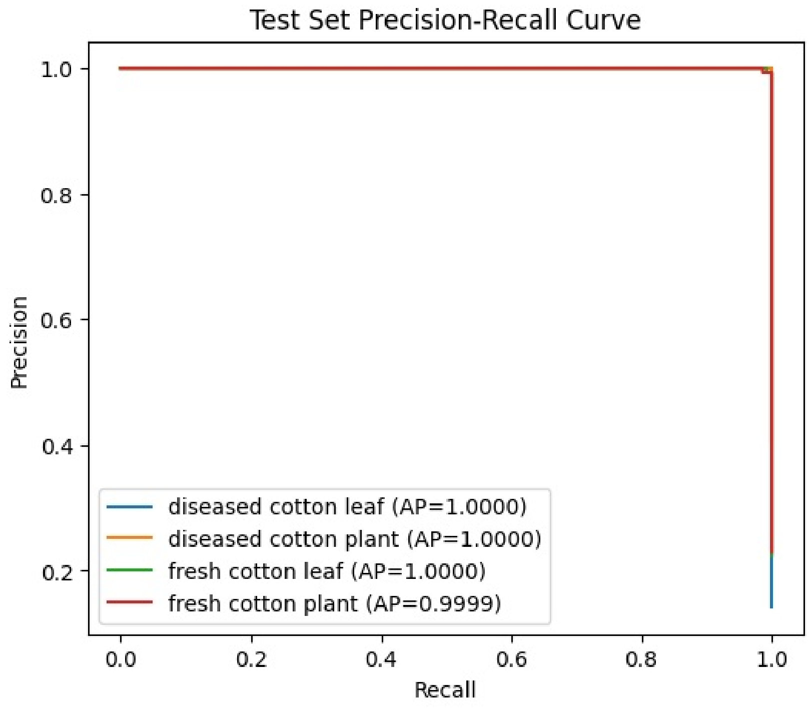



```
MCC: 0.9968
Cohen's Kappa: 0.9968
Mean NPV: 0.9993
Mean PPV (Precision): 0.9975
Validation Set inference time: 0.84 sec

--- Test Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        92
           1     1.0000    0.9962    0.9981       263
           2     0.9931    0.9931    0.9931       145
           3     0.9933    1.0000    0.9967       149

    accuracy                         0.9969       649
   macro avg     0.9966    0.9973    0.9970       649
weighted avg     0.9969    0.9969    0.9969       649
```




```
MCC: 0.9957
Cohen's Kappa: 0.9957
Mean NPV: 0.9989
Mean PPV (Precision): 0.9966
ROC AUC: 1.0000, PR AUC: 1.0000
```






```
Test Set inference time: 1.24 sec

================ Summary ================
Training inference time: 12.94 sec
Validation inference time: 0.84 sec
Test inference time: 1.24 sec
```

In [13]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from timm import create_model
```

```python
from torch.optim.lr_scheduler import ReduceLROnPlateau

from sklearn.metrics import (
    classification_report, confusion_matrix, matthews_corrcoef,
    roc_auc_score, average_precision_score, roc_curve,
    precision_recall_curve, cohen_kappa_score
)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

# =========================
# Config for Cotton Dataset
# =========================
train_dir = "/kaggle/working/cotton_train_aug"  # Augmented train
val_dir   = "/kaggle/working/cotton_split/val"
test_dir  = "/kaggle/working/cotton_split/test"

batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["diseased cotton leaf", "diseased cotton plant", "fresh cotton leaf", "fresh cotton plant"]

# =========================
# Data Transforms & Dataloaders
# =========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=train_dir, transform=common_tfms)
val_ds   = datasets.ImageFolder(root=val_dir, transform=common_tfms)
test_ds  = datasets.ImageFolder(root=test_dir, transform=common_tfms)

train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=2)
val_loader   = DataLoader(val_ds, batch_size=batch_size, shuffle=False, num_workers=2)
test_loader  = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=2)

# =========================
# Model
# =========================
model = create_model("levit_128", pretrained=True, num_classes=num_classes)  # smaller LeViT
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-4)
scheduler = ReduceLROnPlateau(optimizer, mode="min", patience=2, factor=0.5, verbose=True)

# =========================
# Training Loop with Early Stopping
# =========================
best_val_loss = float("inf")
patience_counter = 0

for epoch in range(num_epochs):
    start_time = time.time()

    # ---- Train ----
    model.train()
    train_loss, train_correct = 0, 0
    for imgs, labels in train_loader:
        imgs, labels = imgs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * imgs.size(0)
        train_correct += (outputs.argmax(1) == labels).sum().item()

    train_loss /= len(train_loader.dataset)
    train_acc = train_correct / len(train_loader.dataset)

    # ---- Validation ----
    model.eval()
    val_loss, val_correct = 0, 0
```

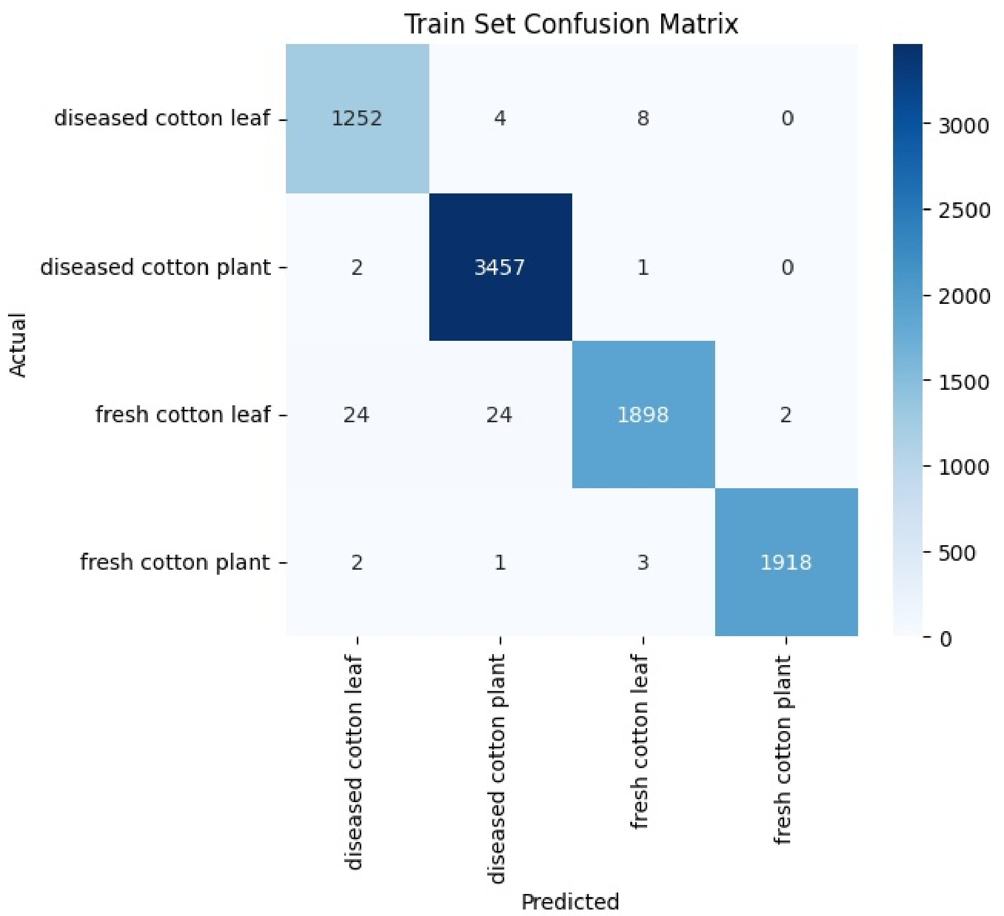
```
    with torch.no_grad():
        for imgs, labels in val_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)

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

    val_loss /= len(val_loader.dataset)
    val_acc = val_correct / len(val_loader.dataset)

    scheduler.step(val_loss)

    elapsed = time.time() - start_time
    print(f"Epoch [{epoch+1}/{num_epochs}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f} "
          f"Time: {elapsed:.2f}s")

    # Early Stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        torch.save(model.state_dict(), "tiny_vit_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as tiny_vit_best.pth")

# =========================
# Evaluation Functions
# =========================
def plot_confusion_matrix(cm, classes, title="Confusion Matrix"):
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=classes, yticklabels=classes)
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(title)
    plt.show()


def print_report(y_true, y_pred, y_prob, title="", plot_curves=False):
    print(f"\n--- {title} ---")
    print(classification_report(y_true, y_pred, digits=4))

    cm = confusion_matrix(y_true, y_pred)
    plot_confusion_matrix(cm, class_names, title=f"{title} Confusion Matrix")

    mcc = matthews_corrcoef(y_true, y_pred)
    print(f"MCC: {mcc:.4f}")

    kappa = cohen_kappa_score(y_true, y_pred)
    print(f"Cohen's Kappa: {kappa:.4f}")

    # Class-wise NPV + PPV
    npv_list, ppv_list = [], []
    for i in range(len(cm)):
        TN = np.sum(np.delete(np.delete(cm, i, axis=0), i, axis=1))
        FN = np.sum(cm[i, :]) - cm[i, i]
        FP = np.sum(cm[:, i]) - cm[i, i]
        TP = cm[i, i]

        NPV = TN / (TN + FN) if (TN + FN) > 0 else 0
        PPV = TP / (TP + FP) if (TP + FP) > 0 else 0
        npv_list.append(NPV)
        ppv_list.append(PPV)

    print(f"Mean NPV: {np.mean(npv_list):.4f}")
    print(f"Mean PPV (Precision): {np.mean(ppv_list):.4f}")

    if plot_curves:
        y_onehot = np.eye(num_classes)[y_true]
        roc_auc = roc_auc_score(y_onehot, y_prob, average='macro', multi_class='ovr')
        pr_auc = average_precision_score(y_onehot, y_prob, average='macro')
        print(f"ROC AUC: {roc_auc:.4f}, PR AUC: {pr_auc:.4f}")

        # ROC Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
```

```python
            fpr, tpr, _ = roc_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(fpr, tpr, label=f"{class_names[i]} (AUC={roc_auc_score(y_onehot[:,i], y_prob[:,i]):.4f})")
        plt.plot([0,1],[0,1],'k--')
        plt.title(f"{title} ROC Curve")
        plt.xlabel("FPR")
        plt.ylabel("TPR")
        plt.legend()
        plt.show()

        # PR Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            precision, recall, _ = precision_recall_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(recall, precision, label=f"{class_names[i]} (AP={average_precision_score(y_onehot[:,i], y_p
        plt.title(f"{title} Precision-Recall Curve")
        plt.xlabel("Recall")
        plt.ylabel("Precision")
        plt.legend()
        plt.show()

def evaluate_model(model, loader, title="", plot_curves=False):
    model.eval()
    y_true, y_pred, y_prob = [], [], []
    start_time = time.time()

    with torch.no_grad():
        for imgs, labels in loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            probs = torch.softmax(outputs, dim=1)
            preds = outputs.argmax(1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_prob.extend(probs.cpu().numpy())

    infer_time = time.time() - start_time
    y_true, y_pred, y_prob = np.array(y_true), np.array(y_pred), np.array(y_prob)

    print_report(y_true, y_pred, y_prob, title, plot_curves)
    print(f"{title} inference time: {infer_time:.2f} sec")
    return y_true, y_pred, y_prob, infer_time

# ==========================
# Final Evaluation
# ==========================
model.load_state_dict(torch.load("tiny_vit_best.pth"))

y_true_train, y_pred_train, y_prob_train, train_infer_time = evaluate_model(model, train_loader, "Train Set", pl
y_true_val,   y_pred_val,   y_prob_val,   val_infer_time   = evaluate_model(model, val_loader, "Validation Set",
y_true_test,  y_pred_test,  y_prob_test,  test_infer_time  = evaluate_model(model, test_loader, "Test Set", plot

print("\n================ Summary ================")
print(f"Training inference time: {train_infer_time:.2f} sec")
print(f"Validation inference time: {val_infer_time:.2f} sec")
print(f"Test inference time: {test_infer_time:.2f} sec")
```

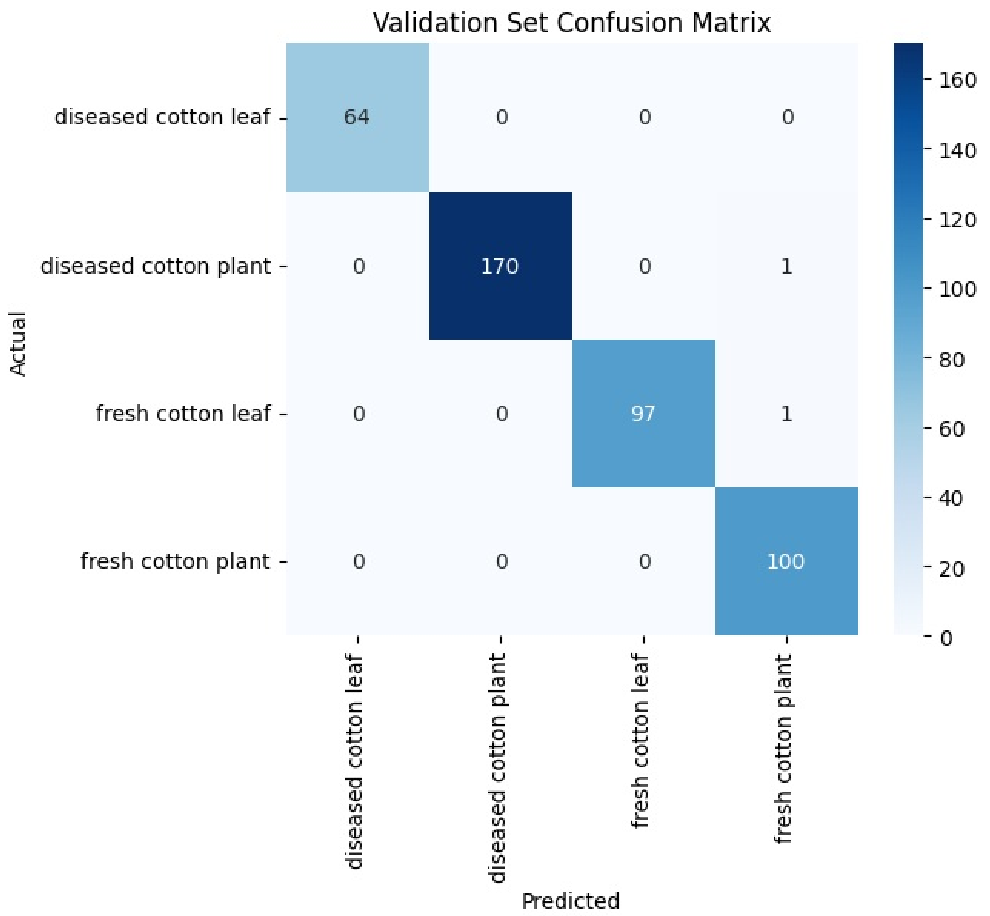
```
model.safetensors:   0%|          | 0.00/37.1M [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is de
precated. Please use get_last_lr() to access the learning rate.
  warnings.warn(
```

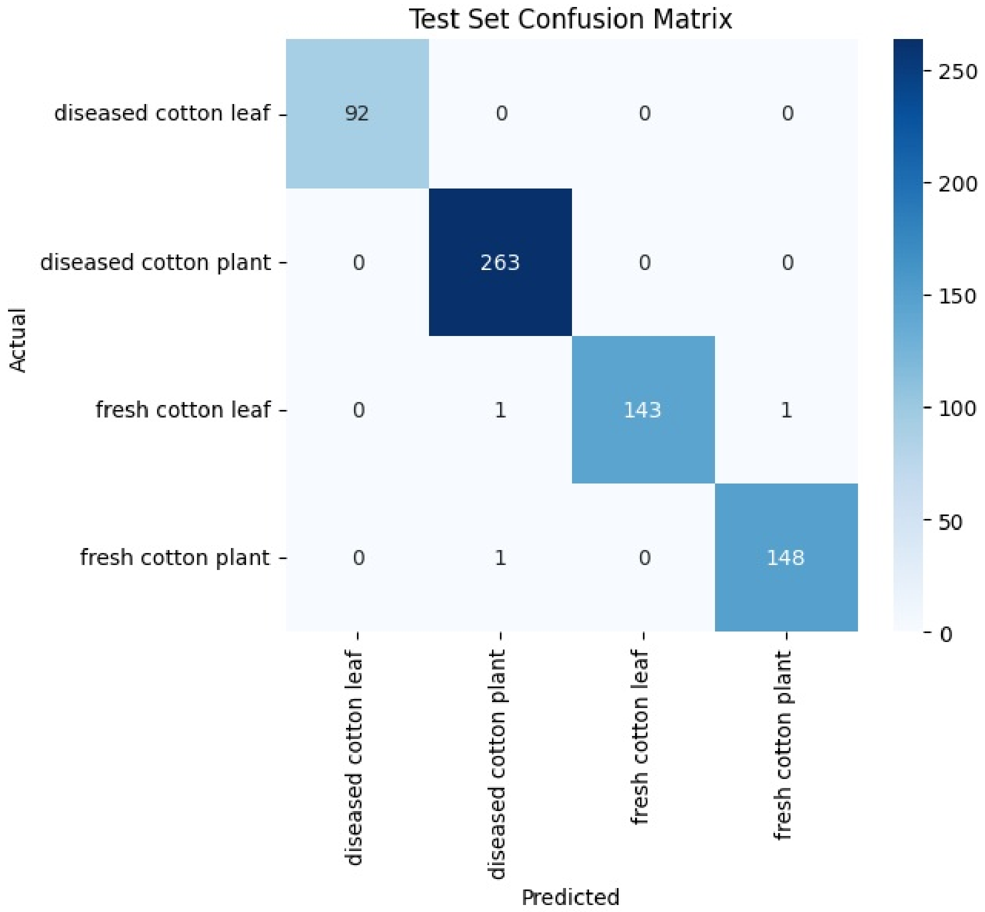
```
Epoch [1/35] Train Loss: 0.6636, Train Acc: 0.8213 Val Loss: 5.8121, Val Acc: 0.9492 Time: 18.83s
Epoch [2/35] Train Loss: 0.1902, Train Acc: 0.9532 Val Loss: 0.2565, Val Acc: 0.9815 Time: 18.88s
Epoch [3/35] Train Loss: 0.1058, Train Acc: 0.9716 Val Loss: 2.9911, Val Acc: 0.9861 Time: 18.77s
Epoch [4/35] Train Loss: 0.0666, Train Acc: 0.9824 Val Loss: 0.0516, Val Acc: 0.9908 Time: 18.82s
Epoch [5/35] Train Loss: 0.0978, Train Acc: 0.9699 Val Loss: 0.0448, Val Acc: 0.9908 Time: 19.02s
Epoch [6/35] Train Loss: 0.0855, Train Acc: 0.9739 Val Loss: 0.7255, Val Acc: 0.9746 Time: 18.82s
Epoch [7/35] Train Loss: 0.0697, Train Acc: 0.9789 Val Loss: 0.0359, Val Acc: 0.9908 Time: 19.11s
Epoch [8/35] Train Loss: 0.0614, Train Acc: 0.9786 Val Loss: 0.0321, Val Acc: 0.9931 Time: 19.05s
Epoch [9/35] Train Loss: 0.0420, Train Acc: 0.9866 Val Loss: 0.3267, Val Acc: 0.9908 Time: 18.97s
Epoch [10/35] Train Loss: 0.0361, Train Acc: 0.9899 Val Loss: 0.0174, Val Acc: 0.9931 Time: 19.16s
Epoch [11/35] Train Loss: 0.0295, Train Acc: 0.9912 Val Loss: 0.0878, Val Acc: 0.9954 Time: 19.18s
Epoch [12/35] Train Loss: 0.0263, Train Acc: 0.9924 Val Loss: 0.0206, Val Acc: 0.9931 Time: 19.30s
Epoch [13/35] Train Loss: 0.0179, Train Acc: 0.9944 Val Loss: 0.0184, Val Acc: 0.9931 Time: 19.09s
Epoch [14/35] Train Loss: 0.0150, Train Acc: 0.9965 Val Loss: 0.0135, Val Acc: 0.9954 Time: 19.38s
Epoch [15/35] Train Loss: 0.0133, Train Acc: 0.9973 Val Loss: 0.0151, Val Acc: 0.9931 Time: 18.97s
Epoch [16/35] Train Loss: 0.0098, Train Acc: 0.9973 Val Loss: 0.0125, Val Acc: 0.9954 Time: 19.18s
Epoch [17/35] Train Loss: 0.0100, Train Acc: 0.9974 Val Loss: 0.0164, Val Acc: 0.9954 Time: 18.76s
Epoch [18/35] Train Loss: 0.0090, Train Acc: 0.9979 Val Loss: 0.0220, Val Acc: 0.9931 Time: 18.66s
Epoch [19/35] Train Loss: 0.0075, Train Acc: 0.9985 Val Loss: 0.0190, Val Acc: 0.9931 Time: 19.26s
Epoch [20/35] Train Loss: 0.0086, Train Acc: 0.9980 Val Loss: 0.0242, Val Acc: 0.9931 Time: 18.91s
Epoch [21/35] Train Loss: 0.0079, Train Acc: 0.9983 Val Loss: 0.0472, Val Acc: 0.9931 Time: 18.98s
Early stopping triggered!

Training finished  Best model saved as tiny_vit_best.pth

--- Train Set ---
              precision    recall  f1-score   support

           0     0.9781    0.9905    0.9843      1264
           1     0.9917    0.9991    0.9954      3460
           2     0.9937    0.9743    0.9839      1948
           3     0.9990    0.9969    0.9979      1924

    accuracy                         0.9917      8596
   macro avg     0.9906    0.9902    0.9904      8596
weighted avg     0.9918    0.9917    0.9917      8596
```



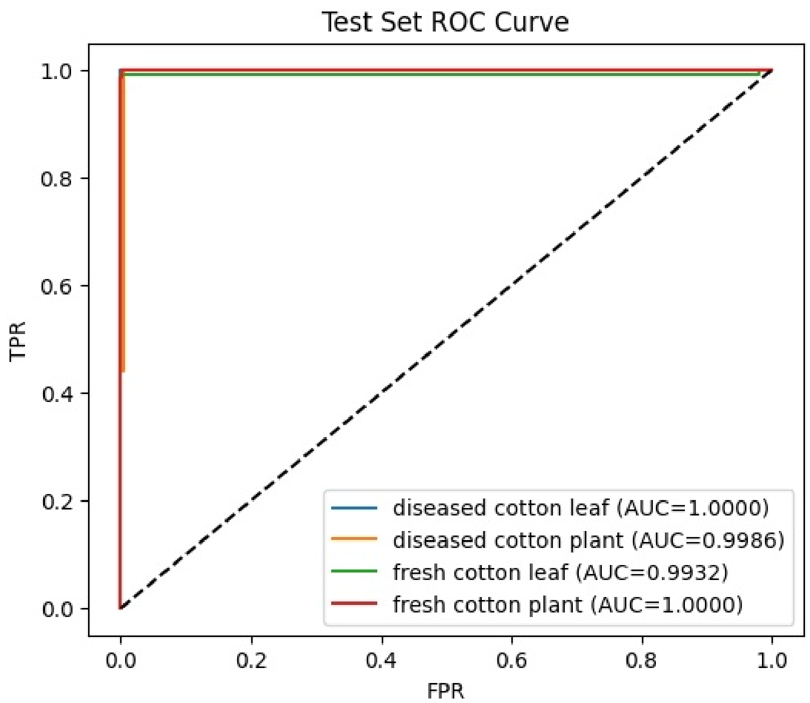
```
MCC: 0.9885
Cohen's Kappa: 0.9884
Mean NPV: 0.9973
Mean PPV (Precision): 0.9906
Train Set inference time: 11.03 sec

--- Validation Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        64
           1     1.0000    0.9942    0.9971       171
           2     1.0000    0.9898    0.9949        98
           3     0.9804    1.0000    0.9901       100

    accuracy                         0.9954       433
   macro avg     0.9951    0.9960    0.9955       433
weighted avg     0.9955    0.9954    0.9954       433
```

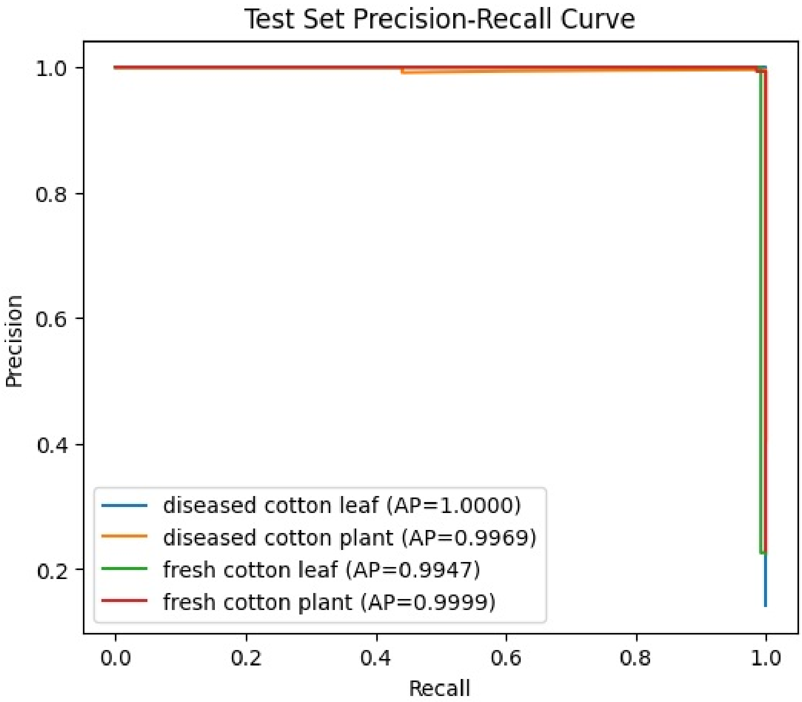



```
MCC: 0.9936
Cohen's Kappa: 0.9936
Mean NPV: 0.9983
Mean PPV (Precision): 0.9951
Validation Set inference time: 0.73 sec

--- Test Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        92
           1     0.9925    1.0000    0.9962       263
           2     1.0000    0.9862    0.9931       145
           3     0.9933    0.9933    0.9933       149

    accuracy                         0.9954       649
   macro avg     0.9964    0.9949    0.9956       649
weighted avg     0.9954    0.9954    0.9954       649
```




```
MCC: 0.9935
Cohen's Kappa: 0.9935
Mean NPV: 0.9985
Mean PPV (Precision): 0.9964
ROC AUC: 0.9979, PR AUC: 0.9979
```






```
Test Set inference time: 1.02 sec

================ Summary ================
Training inference time: 11.03 sec
Validation inference time: 0.73 sec
Test inference time: 1.02 sec
```

In [15]:

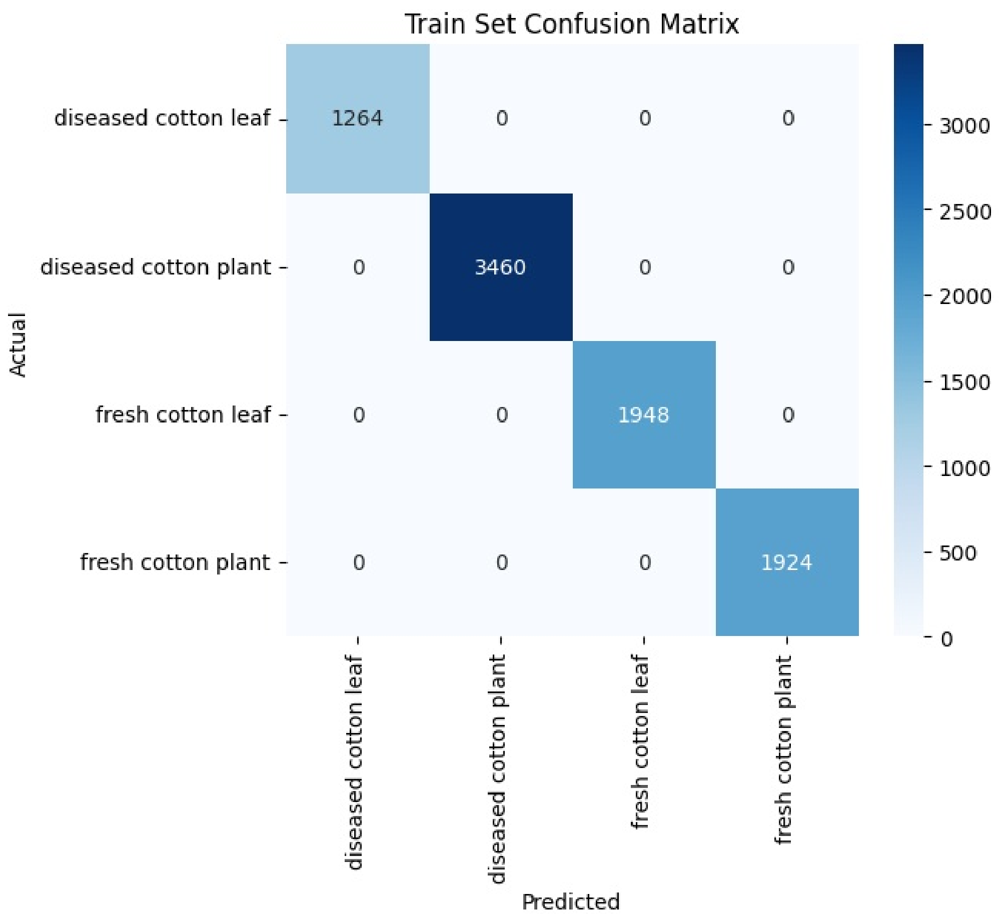
```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from timm import create_model
from torch.optim.lr_scheduler import ReduceLROnPlateau

from sklearn.metrics import (
    classification_report, confusion_matrix, matthews_corrcoef,
    roc_auc_score, average_precision_score, roc_curve,
    precision_recall_curve, cohen_kappa_score
)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

# =========================
# Config for Cotton Dataset
# =========================
train_dir = "/kaggle/working/cotton_train_aug"  # Augmented train
val_dir   = "/kaggle/working/cotton_split/val"
test_dir  = "/kaggle/working/cotton_split/test"

batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["diseased cotton leaf", "diseased cotton plant", "fresh cotton leaf", "fresh cotton plant"]

# =========================
# Data Transforms & Dataloaders
# =========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=train_dir, transform=common_tfms)
val_ds   = datasets.ImageFolder(root=val_dir, transform=common_tfms)
test_ds  = datasets.ImageFolder(root=test_dir, transform=common_tfms)
```

```python
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=2)
val_loader   = DataLoader(val_ds, batch_size=batch_size, shuffle=False, num_workers=2)
test_loader  = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=2)

# ========================
# Model
# ========================
model = create_model("deit_tiny_patch16_224", pretrained=True, num_classes=num_classes)
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-4)
scheduler = ReduceLROnPlateau(optimizer, mode="min", patience=2, factor=0.5, verbose=True)

# ========================
# Training Loop with Early Stopping
# ========================
best_val_loss = float("inf")
patience_counter = 0

for epoch in range(num_epochs):
    start_time = time.time()

    # ---- Train ----
    model.train()
    train_loss, train_correct = 0, 0
    for imgs, labels in train_loader:
        imgs, labels = imgs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * imgs.size(0)
        train_correct += (outputs.argmax(1) == labels).sum().item()

    train_loss /= len(train_loader.dataset)
    train_acc = train_correct / len(train_loader.dataset)

    # ---- Validation ----
    model.eval()
    val_loss, val_correct = 0, 0
    with torch.no_grad():
        for imgs, labels in val_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)

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

    val_loss /= len(val_loader.dataset)
    val_acc = val_correct / len(val_loader.dataset)

    scheduler.step(val_loss)

    elapsed = time.time() - start_time
    print(f"Epoch [{epoch+1}/{num_epochs}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f} "
          f"Time: {elapsed:.2f}s")

    # Early Stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        torch.save(model.state_dict(), "tiny_vit_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as tiny_vit_best.pth")

# ========================
# Evaluation Functions
# ========================
def plot_confusion_matrix(cm, classes, title="Confusion Matrix"):
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=classes, yticklabels=classes)
```

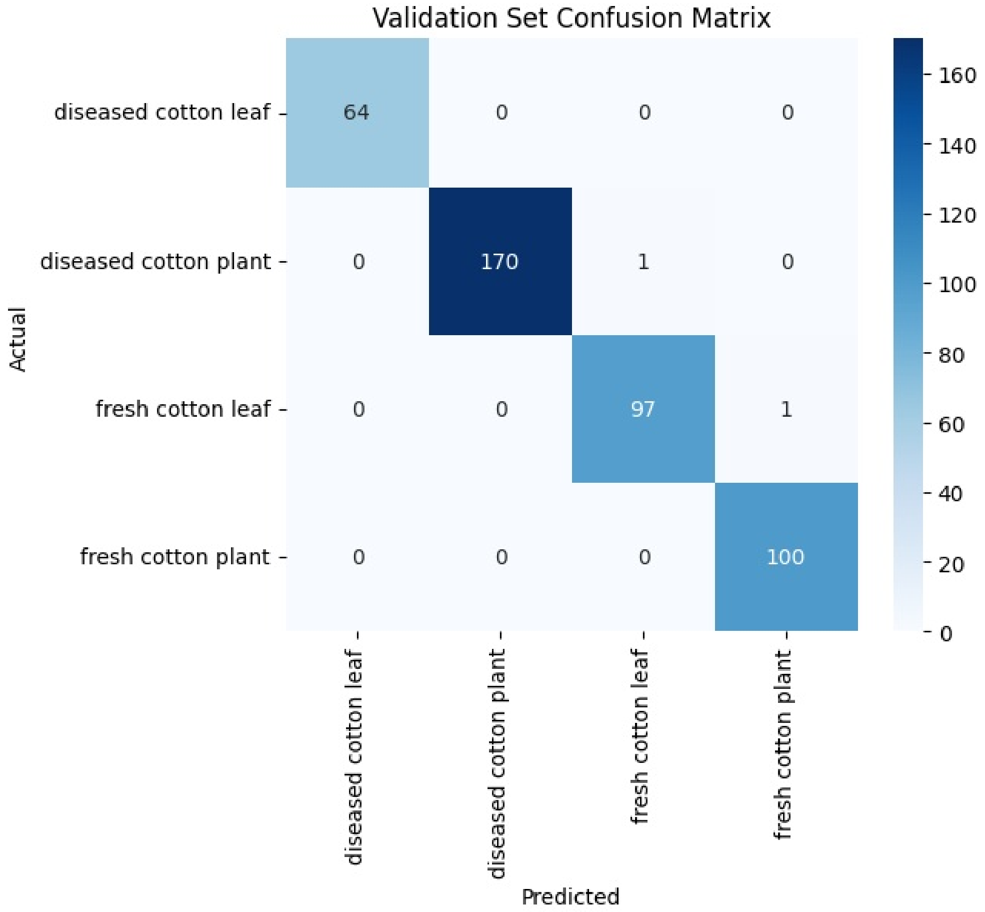
```python
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(title)
    plt.show()


def print_report(y_true, y_pred, y_prob, title="", plot_curves=False):
    print(f"\n--- {title} ---")
    print(classification_report(y_true, y_pred, digits=4))

    cm = confusion_matrix(y_true, y_pred)
    plot_confusion_matrix(cm, class_names, title=f"{title} Confusion Matrix")

    mcc = matthews_corrcoef(y_true, y_pred)
    print(f"MCC: {mcc:.4f}")

    kappa = cohen_kappa_score(y_true, y_pred)
    print(f"Cohen's Kappa: {kappa:.4f}")

    # Class-wise NPV + PPV
    npv_list, ppv_list = [], []
    for i in range(len(cm)):
        TN = np.sum(np.delete(np.delete(cm, i, axis=0), i, axis=1))
        FN = np.sum(cm[i, :]) - cm[i, i]
        FP = np.sum(cm[:, i]) - cm[i, i]
        TP = cm[i, i]

        NPV = TN / (TN + FN) if (TN + FN) > 0 else 0
        PPV = TP / (TP + FP) if (TP + FP) > 0 else 0
        npv_list.append(NPV)
        ppv_list.append(PPV)

    print(f"Mean NPV: {np.mean(npv_list):.4f}")
    print(f"Mean PPV (Precision): {np.mean(ppv_list):.4f}")

    if plot_curves:
        y_onehot = np.eye(num_classes)[y_true]
        roc_auc = roc_auc_score(y_onehot, y_prob, average='macro', multi_class='ovr')
        pr_auc = average_precision_score(y_onehot, y_prob, average='macro')
        print(f"ROC AUC: {roc_auc:.4f}, PR AUC: {pr_auc:.4f}")

        # ROC Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            fpr, tpr, _ = roc_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(fpr, tpr, label=f"{class_names[i]} (AUC={roc_auc_score(y_onehot[:,i], y_prob[:,i]):.4f})")
        plt.plot([0,1],[0,1],'k--')
        plt.title(f"{title} ROC Curve")
        plt.xlabel("FPR")
        plt.ylabel("TPR")
        plt.legend()
        plt.show()

        # PR Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            precision, recall, _ = precision_recall_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(recall, precision, label=f"{class_names[i]} (AP={average_precision_score(y_onehot[:,i], y_p
        plt.title(f"{title} Precision-Recall Curve")
        plt.xlabel("Recall")
        plt.ylabel("Precision")
        plt.legend()
        plt.show()

def evaluate_model(model, loader, title="", plot_curves=False):
    model.eval()
    y_true, y_pred, y_prob = [], [], []
    start_time = time.time()

    with torch.no_grad():
        for imgs, labels in loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            probs = torch.softmax(outputs, dim=1)
            preds = outputs.argmax(1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_prob.extend(probs.cpu().numpy())

    infer_time = time.time() - start_time
    y_true, y_pred, y_prob = np.array(y_true), np.array(y_pred), np.array(y_prob)
```

```
    print_report(y_true, y_pred, y_prob, title, plot_curves)
    print(f"{title} inference time: {infer_time:.2f} sec")
    return y_true, y_pred, y_prob, infer_time

# =========================
# Final Evaluation
# =========================
model.load_state_dict(torch.load("tiny_vit_best.pth"))

y_true_train, y_pred_train, y_prob_train, train_infer_time = evaluate_model(model, train_loader, "Train Set", pl
y_true_val,   y_pred_val,   y_prob_val,   val_infer_time   = evaluate_model(model, val_loader, "Validation Set",
y_true_test,  y_pred_test,  y_prob_test,  test_infer_time  = evaluate_model(model, test_loader, "Test Set", plot

print("\n================ Summary ================")
print(f"Training inference time: {train_infer_time:.2f} sec")
print(f"Validation inference time: {val_infer_time:.2f} sec")
print(f"Test inference time: {test_infer_time:.2f} sec")
```

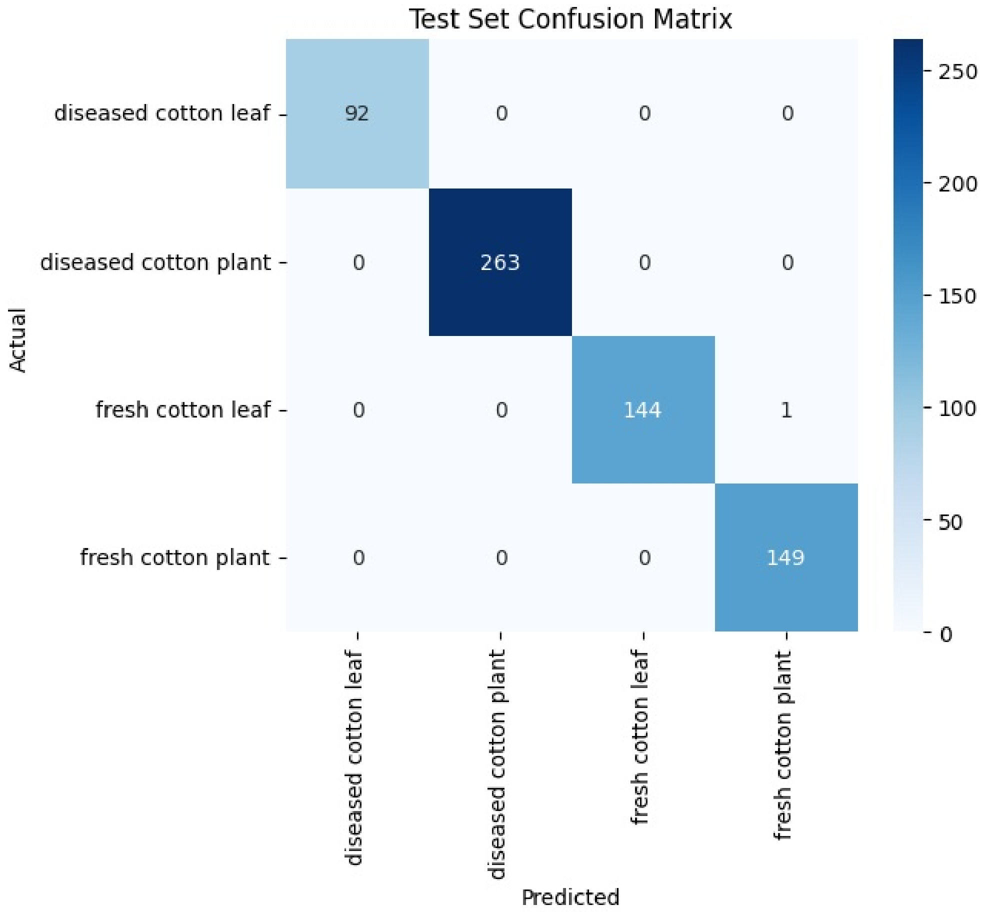
```
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is de
precated. Please use get_last_lr() to access the learning rate.
  warnings.warn(
Epoch [1/35] Train Loss: 0.1376, Train Acc: 0.9522 Val Loss: 0.0358, Val Acc: 0.9931 Time: 24.20s
Epoch [2/35] Train Loss: 0.0242, Train Acc: 0.9928 Val Loss: 0.0451, Val Acc: 0.9885 Time: 24.11s
Epoch [3/35] Train Loss: 0.0210, Train Acc: 0.9935 Val Loss: 0.0170, Val Acc: 0.9977 Time: 24.32s
Epoch [4/35] Train Loss: 0.0174, Train Acc: 0.9938 Val Loss: 0.0421, Val Acc: 0.9861 Time: 24.21s
Epoch [5/35] Train Loss: 0.0178, Train Acc: 0.9943 Val Loss: 0.0171, Val Acc: 0.9954 Time: 24.27s
Epoch [6/35] Train Loss: 0.0090, Train Acc: 0.9974 Val Loss: 0.0497, Val Acc: 0.9838 Time: 24.30s
Epoch [7/35] Train Loss: 0.0050, Train Acc: 0.9985 Val Loss: 0.0147, Val Acc: 0.9954 Time: 24.28s
Epoch [8/35] Train Loss: 0.0016, Train Acc: 0.9995 Val Loss: 0.0172, Val Acc: 0.9977 Time: 24.23s
Epoch [9/35] Train Loss: 0.0003, Train Acc: 1.0000 Val Loss: 0.0178, Val Acc: 0.9977 Time: 24.15s
Epoch [10/35] Train Loss: 0.0003, Train Acc: 1.0000 Val Loss: 0.0179, Val Acc: 0.9977 Time: 24.49s
Epoch [11/35] Train Loss: 0.0002, Train Acc: 1.0000 Val Loss: 0.0180, Val Acc: 0.9977 Time: 24.19s
Epoch [12/35] Train Loss: 0.0002, Train Acc: 1.0000 Val Loss: 0.0181, Val Acc: 0.9977 Time: 24.19s
Early stopping triggered!
Training finished  Best model saved as tiny_vit_best.pth

--- Train Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000      1264
           1     1.0000    1.0000    1.0000      3460
           2     1.0000    1.0000    1.0000      1948
           3     1.0000    1.0000    1.0000      1924

    accuracy                         1.0000      8596
   macro avg     1.0000    1.0000    1.0000      8596
weighted avg     1.0000    1.0000    1.0000      8596
```



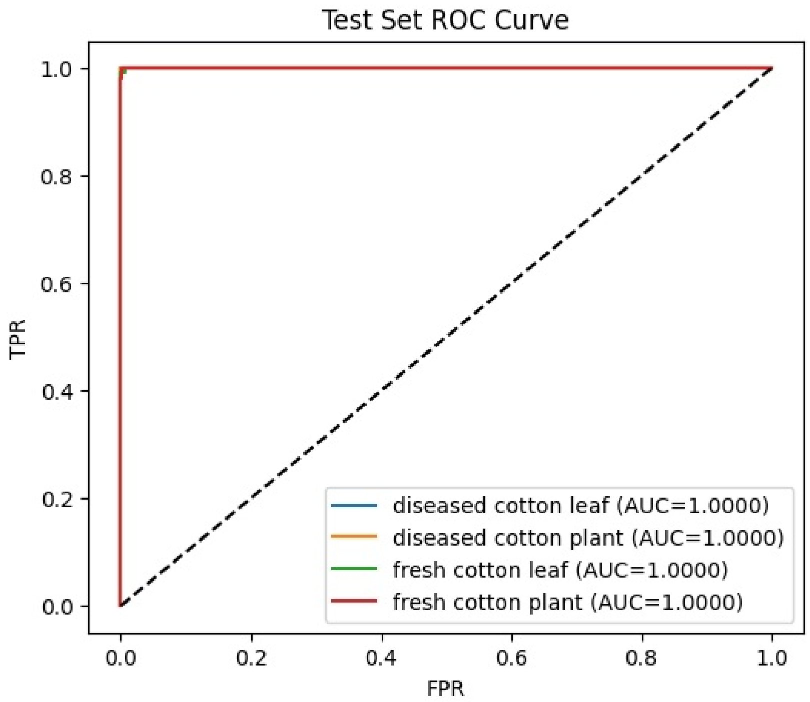
```
MCC: 1.0000
Cohen's Kappa: 1.0000
Mean NPV: 1.0000
Mean PPV (Precision): 1.0000
Train Set inference time: 11.02 sec

--- Validation Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        64
           1     1.0000    0.9942    0.9971       171
           2     0.9898    0.9898    0.9898        98
           3     0.9901    1.0000    0.9950       100

    accuracy                         0.9954       433
   macro avg     0.9950    0.9960    0.9955       433
weighted avg     0.9954    0.9954    0.9954       433
```

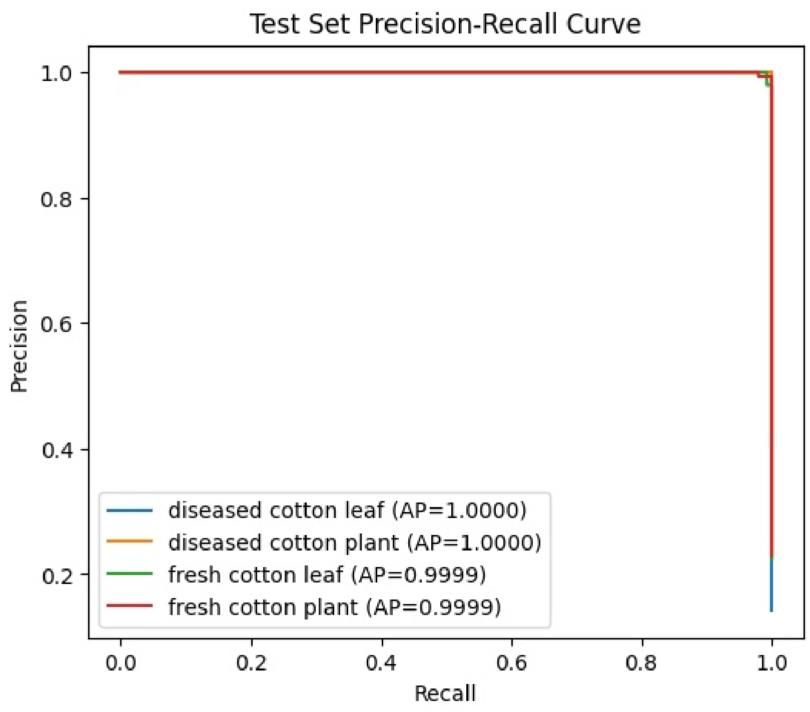



```
MCC: 0.9936
Cohen's Kappa: 0.9936
Mean NPV: 0.9983
Mean PPV (Precision): 0.9950
Validation Set inference time: 0.78 sec

--- Test Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        92
           1     1.0000    1.0000    1.0000       263
           2     1.0000    0.9931    0.9965       145
           3     0.9933    1.0000    0.9967       149

    accuracy                         0.9985       649
   macro avg     0.9983    0.9983    0.9983       649
weighted avg     0.9985    0.9985    0.9985       649
```




```
MCC: 0.9978
Cohen's Kappa: 0.9978
Mean NPV: 0.9995
Mean PPV (Precision): 0.9983
ROC AUC: 1.0000, PR AUC: 0.9999
```






```
Test Set inference time: 1.02 sec

=============== Summary ===============
Training inference time: 11.02 sec
Validation inference time: 0.78 sec
Test inference time: 1.02 sec
```

In [16]:

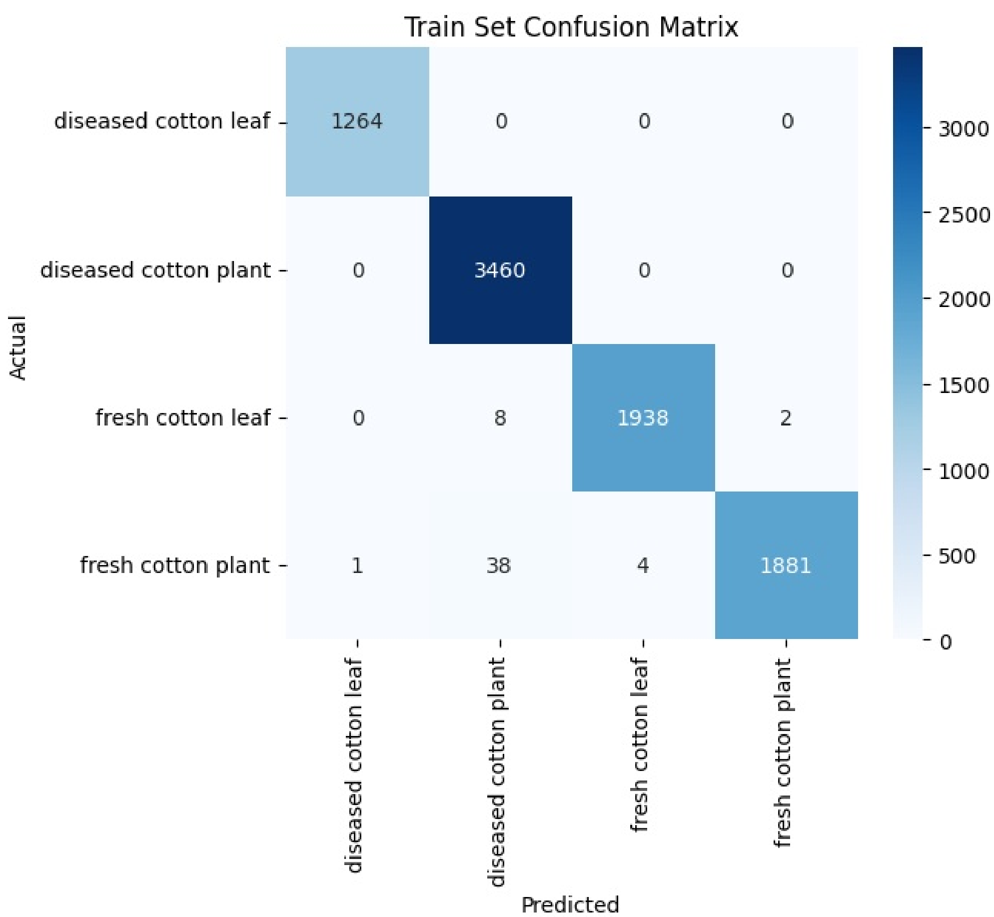
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from timm import create_model
from torch.optim.lr_scheduler import ReduceLROnPlateau

from sklearn.metrics import (
    classification_report, confusion_matrix, matthews_corrcoef,
    roc_auc_score, average_precision_score, roc_curve,
    precision_recall_curve, cohen_kappa_score
)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

# ========================
# Config for Cotton Dataset
# ========================
train_dir = "/kaggle/working/cotton_train_aug"  # Augmented train
val_dir   = "/kaggle/working/cotton_split/val"
test_dir  = "/kaggle/working/cotton_split/test"

batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["diseased cotton leaf", "diseased cotton plant", "fresh cotton leaf", "fresh cotton plant"]

# ========================
# Data Transforms & Dataloaders
# ========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=train_dir, transform=common_tfms)
val_ds   = datasets.ImageFolder(root=val_dir, transform=common_tfms)
test_ds  = datasets.ImageFolder(root=test_dir, transform=common_tfms)
```

```python
train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=2)
val_loader   = DataLoader(val_ds, batch_size=batch_size, shuffle=False, num_workers=2)
test_loader  = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=2)

# ==========================
# Model
# ==========================
model = create_model("swin_tiny_patch4_window7_224", pretrained=True, num_classes=num_classes)
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-4, weight_decay=1e-4)
scheduler = ReduceLROnPlateau(optimizer, mode="min", patience=2, factor=0.5, verbose=True)

# ==========================
# Training Loop with Early Stopping
# ==========================
best_val_loss = float("inf")
patience_counter = 0

for epoch in range(num_epochs):
    start_time = time.time()

    # ---- Train ----
    model.train()
    train_loss, train_correct = 0, 0
    for imgs, labels in train_loader:
        imgs, labels = imgs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * imgs.size(0)
        train_correct += (outputs.argmax(1) == labels).sum().item()

    train_loss /= len(train_loader.dataset)
    train_acc = train_correct / len(train_loader.dataset)

    # ---- Validation ----
    model.eval()
    val_loss, val_correct = 0, 0
    with torch.no_grad():
        for imgs, labels in val_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)

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

    val_loss /= len(val_loader.dataset)
    val_acc = val_correct / len(val_loader.dataset)

    scheduler.step(val_loss)

    elapsed = time.time() - start_time
    print(f"Epoch [{epoch+1}/{num_epochs}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f} "
          f"Time: {elapsed:.2f}s")

    # Early Stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        torch.save(model.state_dict(), "tiny_vit_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as tiny_vit_best.pth")

# ==========================
# Evaluation Functions
# ==========================
def plot_confusion_matrix(cm, classes, title="Confusion Matrix"):
    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=classes, yticklabels=classes)
```

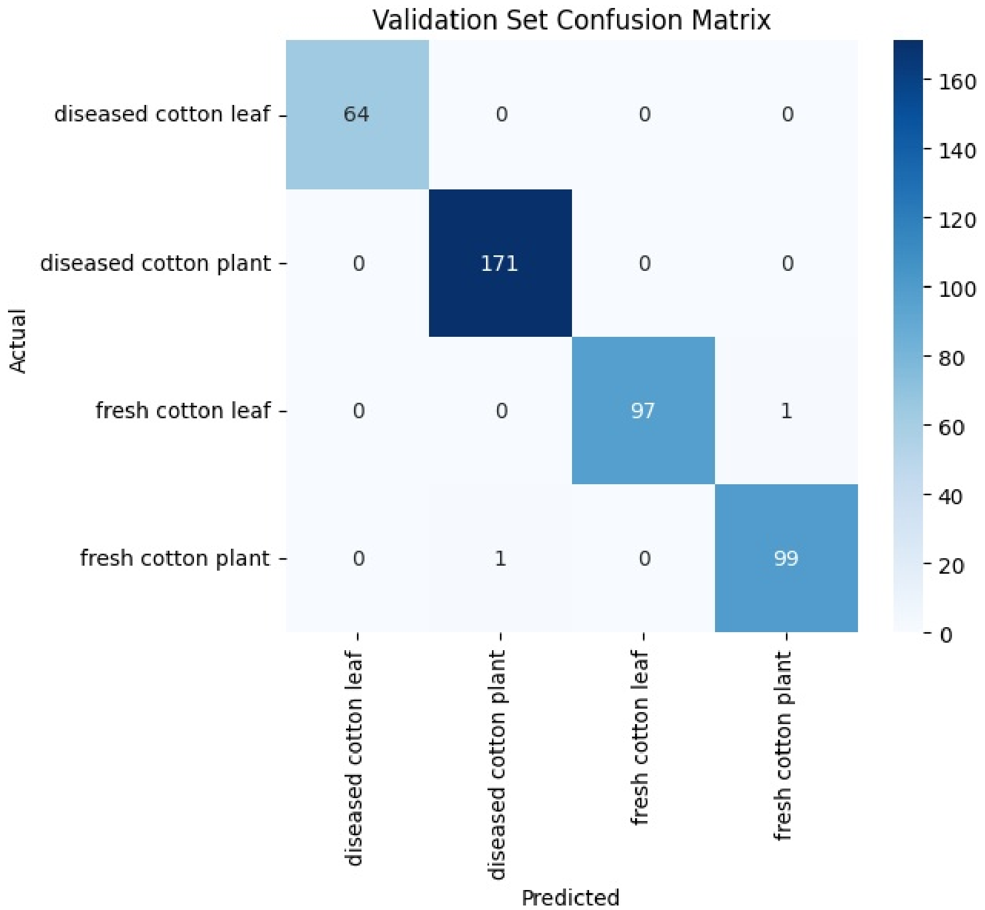
```
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title(title)
    plt.show()


def print_report(y_true, y_pred, y_prob, title="", plot_curves=False):
    print(f"\n--- {title} ---")
    print(classification_report(y_true, y_pred, digits=4))

    cm = confusion_matrix(y_true, y_pred)
    plot_confusion_matrix(cm, class_names, title=f"{title} Confusion Matrix")

    mcc = matthews_corrcoef(y_true, y_pred)
    print(f"MCC: {mcc:.4f}")

    kappa = cohen_kappa_score(y_true, y_pred)
    print(f"Cohen's Kappa: {kappa:.4f}")

    # Class-wise NPV + PPV
    npv_list, ppv_list = [], []
    for i in range(len(cm)):
        TN = np.sum(np.delete(np.delete(cm, i, axis=0), i, axis=1))
        FN = np.sum(cm[i, :]) - cm[i, i]
        FP = np.sum(cm[:, i]) - cm[i, i]
        TP = cm[i, i]

        NPV = TN / (TN + FN) if (TN + FN) > 0 else 0
        PPV = TP / (TP + FP) if (TP + FP) > 0 else 0
        npv_list.append(NPV)
        ppv_list.append(PPV)

    print(f"Mean NPV: {np.mean(npv_list):.4f}")
    print(f"Mean PPV (Precision): {np.mean(ppv_list):.4f}")

    if plot_curves:
        y_onehot = np.eye(num_classes)[y_true]
        roc_auc = roc_auc_score(y_onehot, y_prob, average='macro', multi_class='ovr')
        pr_auc = average_precision_score(y_onehot, y_prob, average='macro')
        print(f"ROC AUC: {roc_auc:.4f}, PR AUC: {pr_auc:.4f}")

        # ROC Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            fpr, tpr, _ = roc_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(fpr, tpr, label=f"{class_names[i]} (AUC={roc_auc_score(y_onehot[:,i], y_prob[:,i]):.4f})")
        plt.plot([0,1],[0,1],'k--')
        plt.title(f"{title} ROC Curve")
        plt.xlabel("FPR")
        plt.ylabel("TPR")
        plt.legend()
        plt.show()

        # PR Curve
        plt.figure(figsize=(6,5))
        for i in range(num_classes):
            precision, recall, _ = precision_recall_curve(y_onehot[:,i], y_prob[:,i])
            plt.plot(recall, precision, label=f"{class_names[i]} (AP={average_precision_score(y_onehot[:,i], y_p
        plt.title(f"{title} Precision-Recall Curve")
        plt.xlabel("Recall")
        plt.ylabel("Precision")
        plt.legend()
        plt.show()

def evaluate_model(model, loader, title="", plot_curves=False):
    model.eval()
    y_true, y_pred, y_prob = [], [], []
    start_time = time.time()

    with torch.no_grad():
        for imgs, labels in loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            probs = torch.softmax(outputs, dim=1)
            preds = outputs.argmax(1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(preds.cpu().numpy())
            y_prob.extend(probs.cpu().numpy())

    infer_time = time.time() - start_time
    y_true, y_pred, y_prob = np.array(y_true), np.array(y_pred), np.array(y_prob)
```

```
    print_report(y_true, y_pred, y_prob, title, plot_curves)
    print(f"{title} inference time: {infer_time:.2f} sec")
    return y_true, y_pred, y_prob, infer_time

# =========================
# Final Evaluation
# =========================
model.load_state_dict(torch.load("tiny_vit_best.pth"))

y_true_train, y_pred_train, y_prob_train, train_infer_time = evaluate_model(model, train_loader, "Train Set", pl
y_true_val,   y_pred_val,   y_prob_val,   val_infer_time   = evaluate_model(model, val_loader, "Validation Set",
y_true_test,  y_pred_test,  y_prob_test,  test_infer_time  = evaluate_model(model, test_loader, "Test Set", plot

print("\n================ Summary ================")
print(f"Training inference time: {train_infer_time:.2f} sec")
print(f"Validation inference time: {val_infer_time:.2f} sec")
print(f"Test inference time: {test_infer_time:.2f} sec")
```

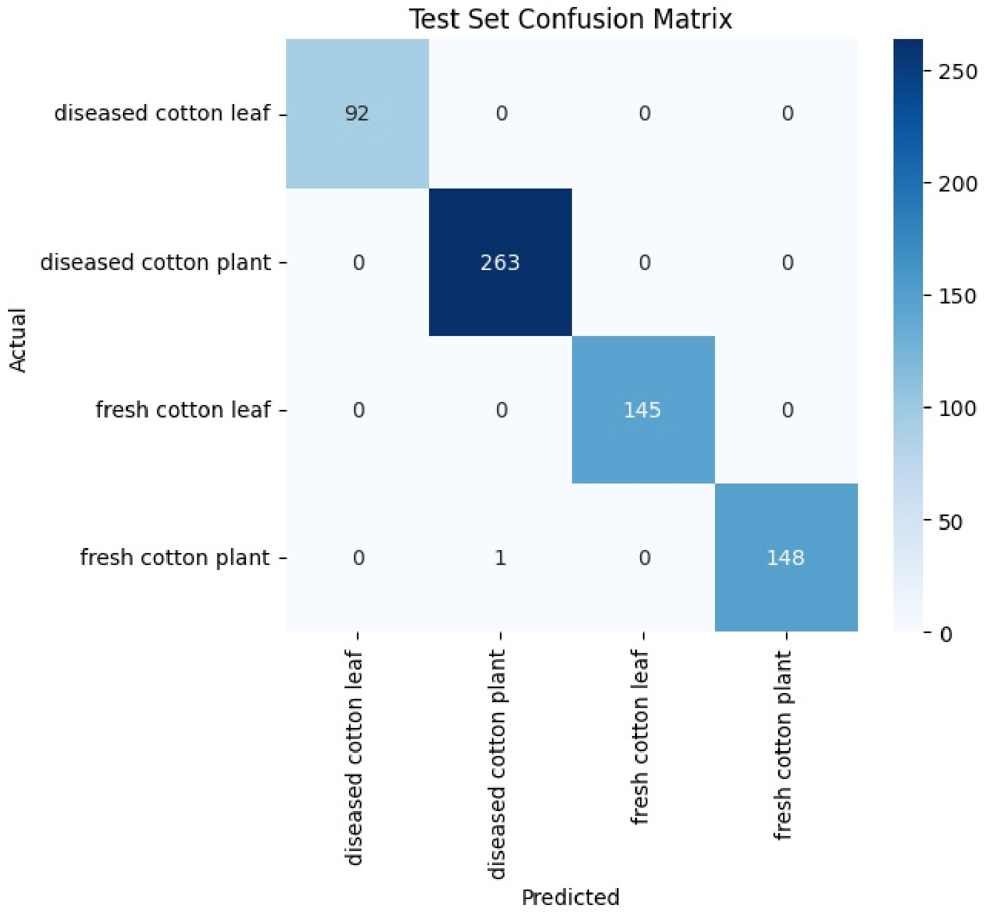
```
model.safetensors:   0%|          | 0.00/114M [00:00<?, ?B/s]
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:62: UserWarning: The verbose parameter is de
precated. Please use get_last_lr() to access the learning rate.
  warnings.warn(
Epoch [1/35] Train Loss: 0.1291, Train Acc: 0.9549 Val Loss: 0.0643, Val Acc: 0.9700 Time: 69.90s
Epoch [2/35] Train Loss: 0.0262, Train Acc: 0.9912 Val Loss: 0.0094, Val Acc: 0.9954 Time: 69.79s
Epoch [3/35] Train Loss: 0.0235, Train Acc: 0.9922 Val Loss: 0.0113, Val Acc: 0.9954 Time: 69.84s
Epoch [4/35] Train Loss: 0.0163, Train Acc: 0.9960 Val Loss: 0.0247, Val Acc: 0.9931 Time: 69.93s
Epoch [5/35] Train Loss: 0.0135, Train Acc: 0.9958 Val Loss: 0.0447, Val Acc: 0.9931 Time: 69.73s
Epoch [6/35] Train Loss: 0.0041, Train Acc: 0.9988 Val Loss: 0.0172, Val Acc: 0.9931 Time: 69.85s
Epoch [7/35] Train Loss: 0.0013, Train Acc: 0.9998 Val Loss: 0.0245, Val Acc: 0.9931 Time: 69.82s
Early stopping triggered!
Training finished  Best model saved as tiny_vit_best.pth

--- Train Set ---
              precision    recall  f1-score   support

           0     0.9992    1.0000    0.9996      1264
           1     0.9869    1.0000    0.9934      3460
           2     0.9979    0.9949    0.9964      1948
           3     0.9989    0.9777    0.9882      1924

    accuracy                         0.9938      8596
   macro avg     0.9957    0.9931    0.9944      8596
weighted avg     0.9939    0.9938    0.9938      8596
```



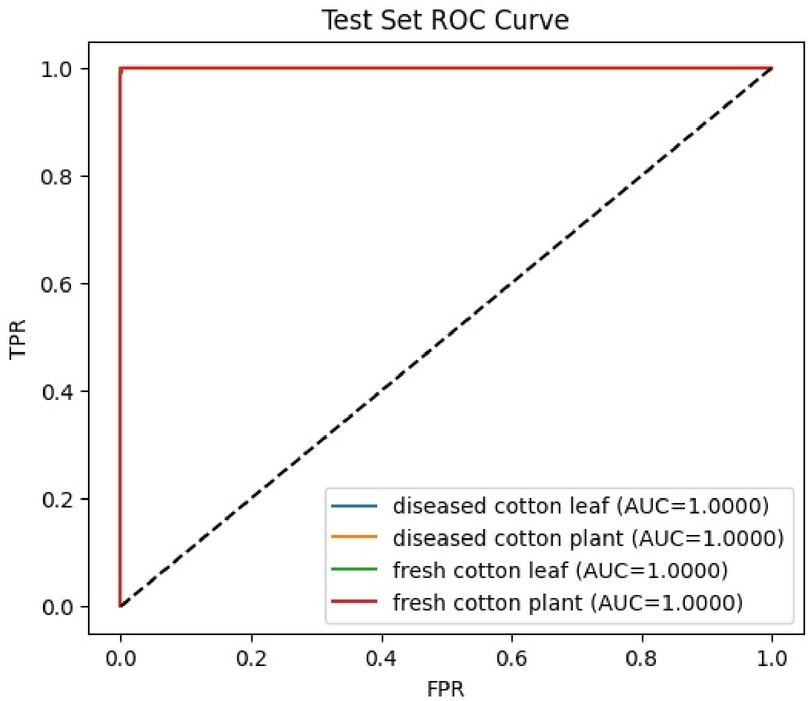
```
MCC: 0.9914
Cohen's Kappa: 0.9914
Mean NPV: 0.9980
Mean PPV (Precision): 0.9957
Train Set inference time: 24.34 sec

--- Validation Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        64
           1     0.9942    1.0000    0.9971       171
           2     1.0000    0.9898    0.9949        98
           3     0.9900    0.9900    0.9900       100

    accuracy                         0.9954       433
   macro avg     0.9960    0.9949    0.9955       433
weighted avg     0.9954    0.9954    0.9954       433
```

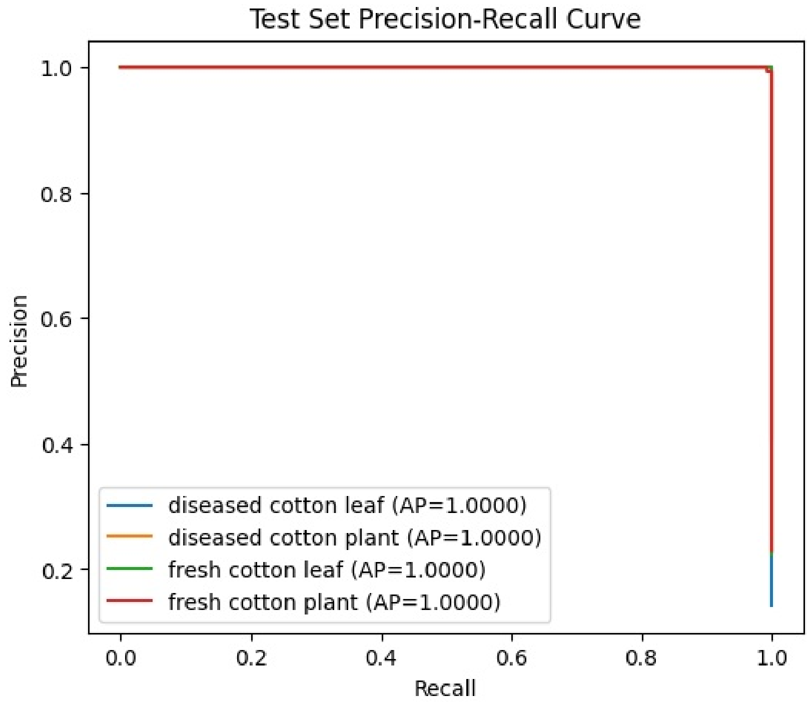



```
MCC: 0.9936
Cohen's Kappa: 0.9936
Mean NPV: 0.9985
Mean PPV (Precision): 0.9960
Validation Set inference time: 1.44 sec

--- Test Set ---
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000        92
           1     0.9962    1.0000    0.9981       263
           2     1.0000    1.0000    1.0000       145
           3     1.0000    0.9933    0.9966       149

    accuracy                         0.9985       649
   macro avg     0.9991    0.9983    0.9987       649
weighted avg     0.9985    0.9985    0.9985       649
```




```
MCC: 0.9978
Cohen's Kappa: 0.9978
Mean NPV: 0.9995
Mean PPV (Precision): 0.9991
ROC AUC: 1.0000, PR AUC: 1.0000
```






```
Test Set inference time: 2.08 sec

================ Summary ================
Training inference time: 24.34 sec
Validation inference time: 1.44 sec
Test inference time: 2.08 sec
```