In [1]:
```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("sohansakib75/pmrambrain")

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

```
Path to dataset files: /kaggle/input/pmrambrain
```

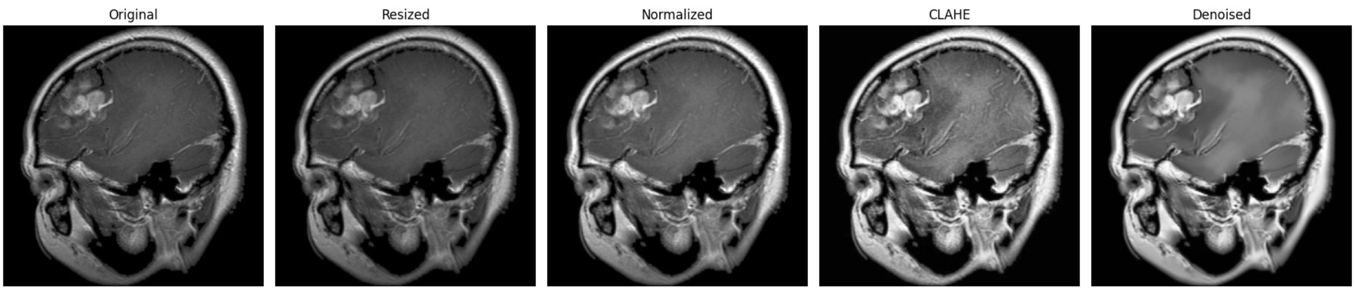
In [2]:
```
import os

for root, dirs, files in os.walk("/kaggle/input/pmrambrain"):
    for d in dirs:
        print(os.path.join(root, d))
```

```
/kaggle/input/pmrambrain/Raw
/kaggle/input/pmrambrain/Raw/512Glioma
/kaggle/input/pmrambrain/Raw/512Meningioma
/kaggle/input/pmrambrain/Raw/512Pituitary
/kaggle/input/pmrambrain/Raw/512Normal
```

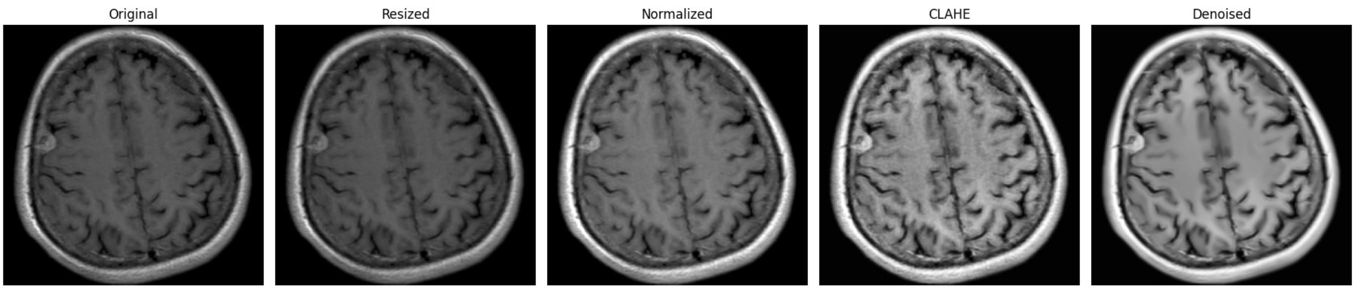
In [3]:
```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.feature import peak_local_max

# Paths and params
base_path = "/kaggle/input/pmrambrain/Raw"
output_base = "/kaggle/working/preprocessed"
class_dirs = ["512Glioma", "512Meningioma", "512Pituitary", "512Normal"]
img_size = (224, 224)

def ensure_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

def save_image(path, img):
    cv2.imwrite(path, img)

def show_images(images, titles):
    plt.figure(figsize=(18, 6))
    for i, (img, title) in enumerate(zip(images, titles)):
        plt.subplot(1, len(images), i+1)
        if len(img.shape) == 2:
            plt.imshow(img, cmap='gray')
        else:
            plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        plt.title(title)
        plt.axis('off')
    plt.tight_layout()
    plt.show()

def preprocess_image(img):
    results = {}

    # Step 1: Resize
    img_resized = cv2.resize(img, img_size)
    results['resized'] = img_resized

    # Step 2: Z-score normalization (grayscale)
    img_gray = cv2.cvtColor(img_resized, cv2.COLOR_BGR2GRAY)
    img_norm = (img_gray - np.mean(img_gray)) / (np.std(img_gray) + 1e-8)
    img_norm = np.clip(img_norm, -3, 3)
    img_norm = ((img_norm - img_norm.min()) / (img_norm.max() - img_norm.min()) * 255).astype(np.uint8)
    results['normalized'] = img_norm

    # Step 3: CLAHE
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
    img_clahe = clahe.apply(img_norm)
    results['clahe'] = img_clahe

    # Step 4: Denoising
    img_denoised = cv2.fastNlMeansDenoising(img_clahe, h=10, templateWindowSize=7, searchWindowSize=21)
    results['denoised'] = img_denoised

    return results

# Visualize and save images for one example per class
for class_name in class_dirs:
    class_path = os.path.join(base_path, class_name)
    save_class_base = os.path.join(output_base, class_name)
    sample_image_name = next((f for f in os.listdir(class_path) if f.lower().endswith(('.png', '.jpg', '.jpeg'))
```

```python
    if sample_image_name:
        img_path = os.path.join(class_path, sample_image_name)
        img = cv2.imread(img_path)  # Read as color BGR
        print(f"\nClass: {class_name}, Image: {sample_image_name}")

        processed_imgs = preprocess_image(img)

        # Show preprocessing steps (excluding watershed color overlay and final grayscale)
        show_images(
            [img, processed_imgs['resized'], processed_imgs['normalized'], processed_imgs['clahe'],
             processed_imgs['denoised']],
            ['Original', 'Resized', 'Normalized', 'CLAHE', 'Denoised']
        )

        # Save all preprocessed images for this class (one sample)
        for step_name, proc_img in processed_imgs.items():
            save_folder = os.path.join(save_class_base, step_name)
            ensure_dir(save_folder)
            save_path = os.path.join(save_folder, os.path.splitext(sample_image_name)[0] + '.png')
            save_image(save_path, proc_img)

# Batch preprocess and save all images (comment out to speed up)
print("\nStarting batch preprocessing & saving all images...")

for class_name in class_dirs:
    class_path = os.path.join(base_path, class_name)
    save_class_base = os.path.join(output_base, class_name)

    for filename in os.listdir(class_path):
        if not filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            continue
        img_path = os.path.join(class_path, filename)
        img = cv2.imread(img_path)  # Read as color BGR

        processed_imgs = preprocess_image(img)
        for step_name, proc_img in processed_imgs.items():
            save_folder = os.path.join(save_class_base, step_name)
            ensure_dir(save_folder)
            save_path = os.path.join(save_folder, os.path.splitext(filename)[0] + '.png')
            save_image(save_path, proc_img)

print("\nAll images preprocessed and saved in /kaggle/working/preprocessed/")
```

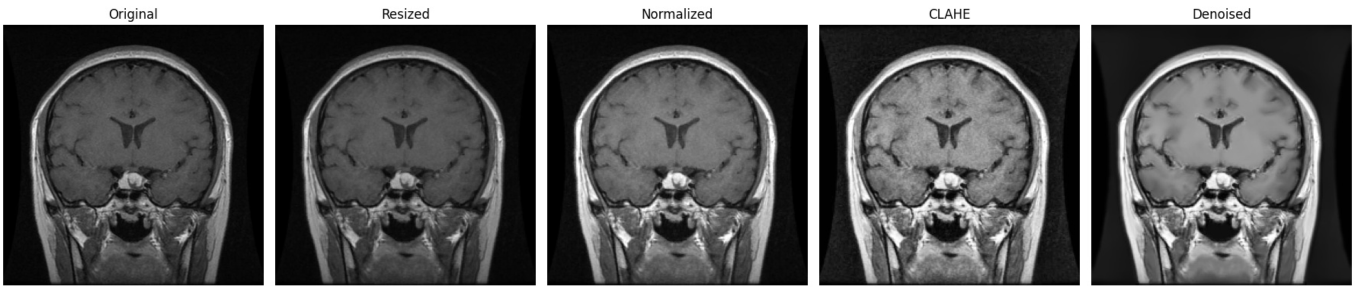
```
Class: 512Glioma, Image: glioma (123).jpg
```



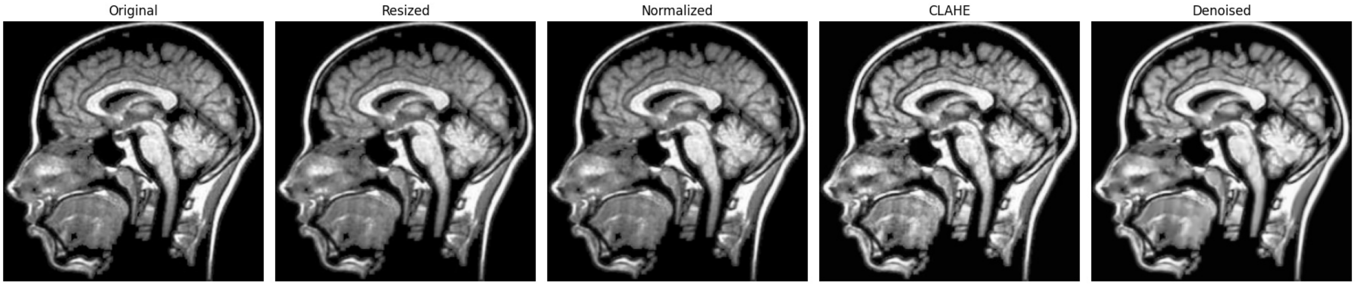

```
Class: 512Meningioma, Image: M_187.jpg
```




```
Class: 512Pituitary, Image: pituitary (205).jpg
```




```
Class: 512Normal, Image: normal (279).jpg
```




```
Starting batch preprocessing & saving all images...

All images preprocessed and saved in /kaggle/working/preprocessed/
```

In [4]:
```python
import os
import cv2
import random

# Paths and params
preprocessed_base = "/kaggle/working/preprocessed"
augmented_base = "/kaggle/working/augmented"
class_dirs = ["512Glioma", "512Meningioma", "512Pituitary", "512Normal"]
target_total = 5500  # approximate total images across all classes

def ensure_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

def save_image(path, img):
    cv2.imwrite(path, img)

def augment_image_flip_rotate(img):
    """Return a dictionary of possible flip and rotation augmentations."""
    aug_dict = {}
    # Horizontal flip
    aug_dict['hflip'] = cv2.flip(img, 1)
    # Vertical flip
    aug_dict['vflip'] = cv2.flip(img, 0)
    # Rotations ±15 degrees
    for angle in [-15, -10, -5, 5, 10, 15]:
        h, w = img.shape[:2]
        M = cv2.getRotationMatrix2D((w//2, h//2), angle, 1)
        aug_dict[f'rot{angle}'] = cv2.warpAffine(img, M, (w, h), borderMode=cv2.BORDER_REFLECT)
    return aug_dict

# Dictionary to hold summary
summary = {}

# Augment only the final preprocessed image (denoised)
for class_name in class_dirs:
    denoised_path = os.path.join(preprocessed_base, class_name, "denoised")
    save_aug_class_path = os.path.join(augmented_base, class_name, "denoised")
    ensure_dir(save_aug_class_path)
    summary[class_name] = 0

    for filename in os.listdir(denoised_path):
        if not filename.lower().endswith('.png'):
            continue
        img_path = os.path.join(denoised_path, filename)
        img = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)

        # Save original
        save_image(os.path.join(save_aug_class_path, filename), img)
        summary[class_name] += 1

        # Decide how many augmentations per image
        aug_dict = augment_image_flip_rotate(img)
        aug_per_image = 2  # choose 2 random augmentations per image
        selected_keys = random.sample(list(aug_dict.keys()), min(aug_per_image, len(aug_dict)))

        for key in selected_keys:
            save_path = os.path.join(save_aug_class_path, f"{os.path.splitext(filename)[0]}_{key}.png")
            save_image(save_path, aug_dict[key])
            summary[class_name] += 1

# Print summary
print("\nAugmentation Summary (including original):")
total_aug = 0
for class_name, count in summary.items():
    print(f"Class: {class_name} -> Total images: {count}")
```

```
    total_aug += count

print(f"\nTotal images across all classes: {total_aug}")
print(f"\nAll augmented images saved in {augmented_base}")
```

```
Augmentation Summary (including original):
Class: 512Glioma -> Total images: 1119
Class: 512Meningioma -> Total images: 1089
Class: 512Pituitary -> Total images: 1119
Class: 512Normal -> Total images: 1188

Total images across all classes: 4515

All augmented images saved in /kaggle/working/augmented
```

In [5]:

```
import os
import cv2
import random
import shutil

# Paths
augmented_base = "/kaggle/working/augmented"
split_base = "/kaggle/working/augmented_split"
class_dirs = ["512Glioma", "512Meningioma", "512Pituitary", "512Normal"]

# Split ratios
train_ratio = 0.75
val_ratio = 0.10
test_ratio = 0.15

def ensure_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

# Create split directories
for split in ['train', 'val', 'test']:
    for class_name in class_dirs:
        ensure_dir(os.path.join(split_base, split, class_name))

# Function to split images
split_summary = {}

for class_name in class_dirs:
    class_aug_path = os.path.join(augmented_base, class_name)

    # Gather all images from all preprocessing steps
    all_imgs = []
    for step_name in os.listdir(class_aug_path):
        step_path = os.path.join(class_aug_path, step_name)
        for fname in os.listdir(step_path):
            if fname.lower().endswith('.png'):
                all_imgs.append(os.path.join(step_path, fname))

    random.shuffle(all_imgs)
    total = len(all_imgs)
    train_end = int(total * train_ratio)
    val_end = train_end + int(total * val_ratio)

    train_imgs = all_imgs[:train_end]
    val_imgs = all_imgs[train_end:val_end]
    test_imgs = all_imgs[val_end:]

    # Copy images to respective folders
    for img_path in train_imgs:
        shutil.copy(img_path, os.path.join(split_base, 'train', class_name))
    for img_path in val_imgs:
        shutil.copy(img_path, os.path.join(split_base, 'val', class_name))
    for img_path in test_imgs:
        shutil.copy(img_path, os.path.join(split_base, 'test', class_name))

    # Save summary for this class
    split_summary[class_name] = {
        'total': total,
        'train': len(train_imgs),
        'val': len(val_imgs),
        'test': len(test_imgs)
    }

# Print summary
print("Augmented Dataset Split Summary:")
for class_name, counts in split_summary.items():
    print(f"{class_name}: Total={counts['total']}, Train={counts['train']}, Val={counts['val']}, Test={counts['t

print(f"\nAll augmented images split and saved in {split_base}")
```

```
Augmented Dataset Split Summary:
512Glioma: Total=1119, Train=839, Val=111, Test=169
512Meningioma: Total=1089, Train=816, Val=108, Test=165
512Pituitary: Total=1119, Train=839, Val=111, Test=169
512Normal: Total=1188, Train=891, Val=118, Test=179

All augmented images split and saved in /kaggle/working/augmented_split
```

In [6]:

```
# ====================================================
# Swin-Tiny Training + Evaluation Pipeline
# ====================================================

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
# ========================
data_dir = "/kaggle/working/augmented_split"
batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4  # Glioma, Meningioma, Pituitary, Normal
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["Glioma", "Meningioma", "Pituitary", "Normal"]

# ========================
# Data (no extra augmentation)
# ========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=f"{data_dir}/train", transform=common_tfms)
val_ds   = datasets.ImageFolder(root=f"{data_dir}/val", transform=common_tfms)
test_ds  = datasets.ImageFolder(root=f"{data_dir}/test", 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("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)
```

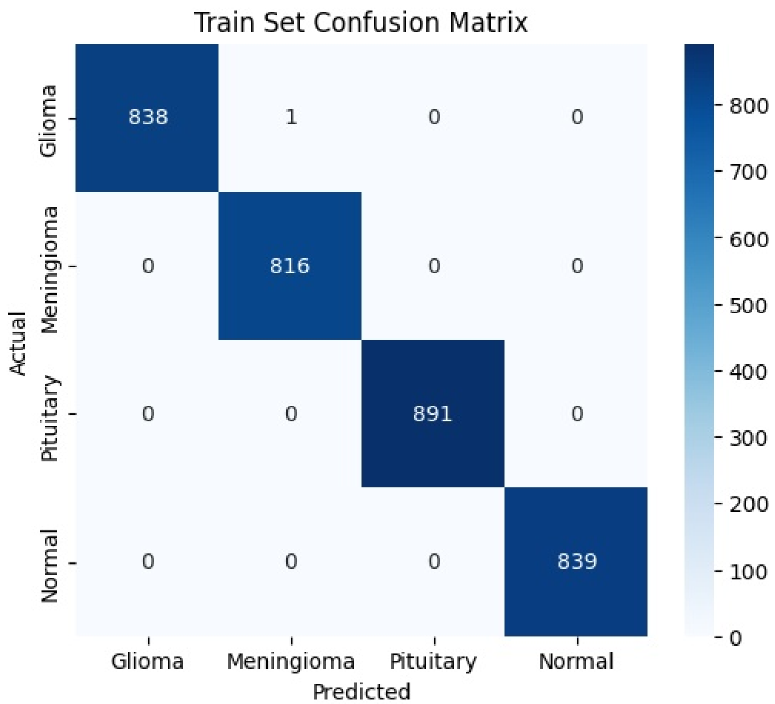
```python
        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        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(), "swin_tiny_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as swin_tiny_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))

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

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

    # Cohen's Kappa
    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]
```

```python
        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:
        # ROC AUC + PR AUC
        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("swin_tiny_best.pth"))

# Only plot curves for Test Set
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")
```

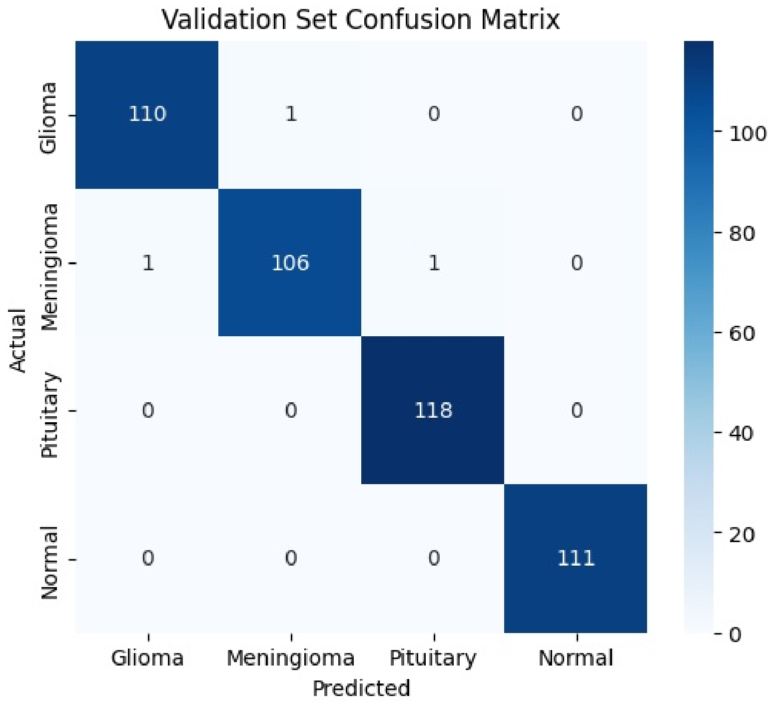
```
model.safetensors:   0%|          | 0.00/114M [00:00<?, ?B/s]
```

```
/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(
```

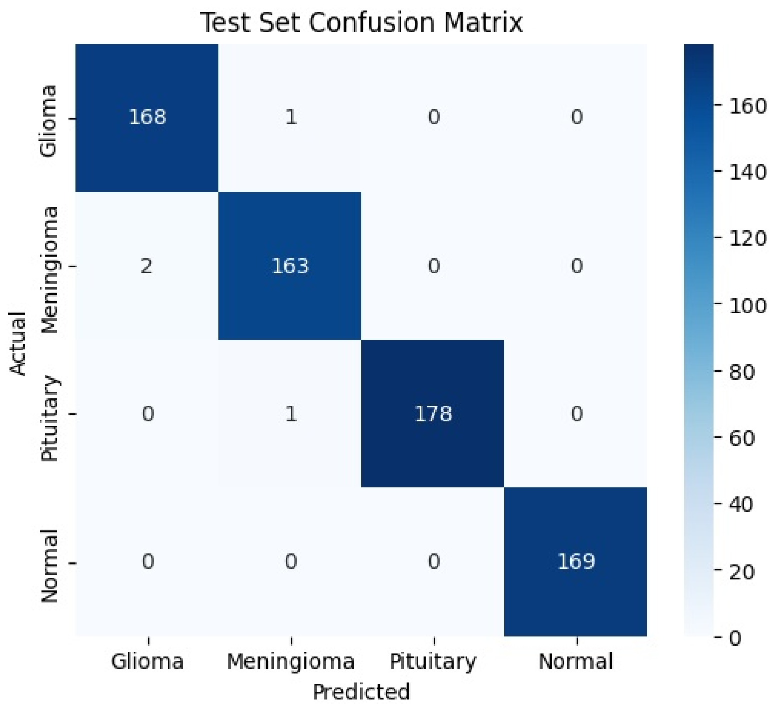
```
Epoch [1/35] Train Loss: 0.3674, Train Acc: 0.8656 Val Loss: 0.1095, Val Acc: 0.9643 Time: 29.94s

Epoch [2/35] Train Loss: 0.0638, Train Acc: 0.9784 Val Loss: 0.2573, Val Acc: 0.9196 Time: 28.66s
Epoch [3/35] Train Loss: 0.0336, Train Acc: 0.9900 Val Loss: 0.0347, Val Acc: 0.9866 Time: 28.73s
Epoch [4/35] Train Loss: 0.0080, Train Acc: 0.9979 Val Loss: 0.0225, Val Acc: 0.9933 Time: 28.71s
Epoch [5/35] Train Loss: 0.0078, Train Acc: 0.9973 Val Loss: 0.0603, Val Acc: 0.9821 Time: 28.80s
Epoch [6/35] Train Loss: 0.0291, Train Acc: 0.9920 Val Loss: 0.0529, Val Acc: 0.9866 Time: 28.74s
Epoch [7/35] Train Loss: 0.0573, Train Acc: 0.9823 Val Loss: 0.0674, Val Acc: 0.9844 Time: 28.73s
Epoch [8/35] Train Loss: 0.0159, Train Acc: 0.9956 Val Loss: 0.0308, Val Acc: 0.9955 Time: 28.71s
Epoch [9/35] Train Loss: 0.0018, Train Acc: 0.9997 Val Loss: 0.0246, Val Acc: 0.9955 Time: 28.71s
Early stopping triggered!
Training finished  Best model saved as swin_tiny_best.pth

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

           0     1.0000    0.9988    0.9994       839
           1     0.9988    1.0000    0.9994       816
           2     1.0000    1.0000    1.0000       891
           3     1.0000    1.0000    1.0000       839

    accuracy                         0.9997      3385
   macro avg     0.9997    0.9997    0.9997      3385
weighted avg     0.9997    0.9997    0.9997      3385
```



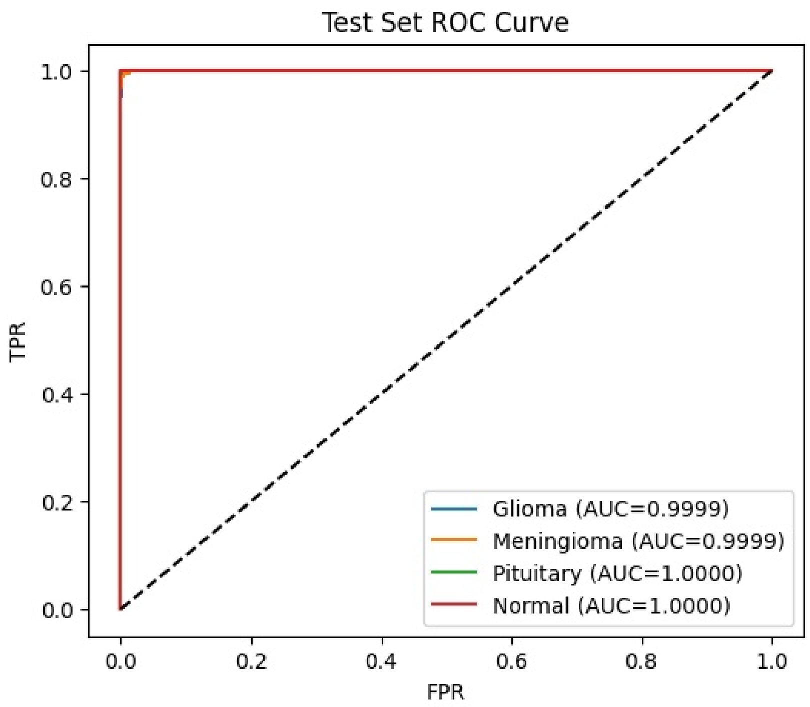

```
MCC: 0.9996
Cohen's Kappa: 0.9996
Mean NPV: 0.9999
Mean PPV (Precision): 0.9997
Train Set inference time: 9.87 sec

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

           0     0.9910    0.9910    0.9910       111
           1     0.9907    0.9815    0.9860       108
           2     0.9916    1.0000    0.9958       118
           3     1.0000    1.0000    1.0000       111

    accuracy                         0.9933       448
   macro avg     0.9933    0.9931    0.9932       448
weighted avg     0.9933    0.9933    0.9933       448
```

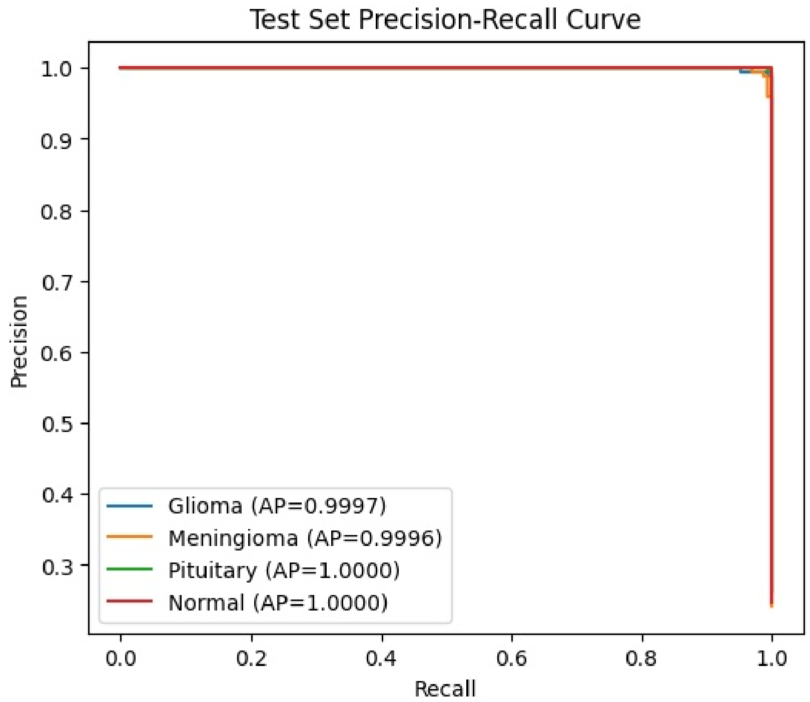



```
MCC: 0.9911
Cohen's Kappa: 0.9911
Mean NPV: 0.9978
Mean PPV (Precision): 0.9933
Validation Set inference time: 1.49 sec

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

           0     0.9882    0.9941    0.9912       169
           1     0.9879    0.9879    0.9879       165
           2     1.0000    0.9944    0.9972       179
           3     1.0000    1.0000    1.0000       169

    accuracy                         0.9941       682
   macro avg     0.9940    0.9941    0.9941       682
weighted avg     0.9942    0.9941    0.9941       682
```




```
MCC: 0.9922
Cohen's Kappa: 0.9922
Mean NPV: 0.9980
Mean PPV (Precision): 0.9940
ROC AUC: 0.9999, PR AUC: 0.9998
```


Test Set ROC Curve
TPR
FPR
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Glioma (AUC=0.9999)
Meningioma (AUC=0.9999)
Pituitary (AUC=1.0000)
Normal (AUC=1.0000)


Test Set Precision-Recall Curve
Precision
Recall
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.0
0.2
0.4
0.6
0.8
1.0
Glioma (AP=0.9997)
Meningioma (AP=0.9996)
Pituitary (AP=1.0000)
Normal (AP=1.0000)

```
Test Set inference time: 2.19 sec

=============== Summary ===============
Training inference time: 9.87 sec
Validation inference time: 1.49 sec
Test inference time: 2.19 sec
```

In [7]:
```python
# ====================================================
# MobileViT Training + Evaluation Pipeline
# ====================================================

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
# ========================
data_dir = "/kaggle/working/augmented_split"
batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4  # Glioma, Meningioma, Pituitary, Normal
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["Glioma", "Meningioma", "Pituitary", "Normal"]

# ========================
# Data (no extra augmentation)
# ========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=f"{data_dir}/train", transform=common_tfms)
val_ds   = datasets.ImageFolder(root=f"{data_dir}/val", transform=common_tfms)
test_ds  = datasets.ImageFolder(root=f"{data_dir}/test", 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 (MobileViT Small)
# Options: "mobilevit_xxs", "mobilevit_xs", "mobilevit_s"
# ========================
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)
```

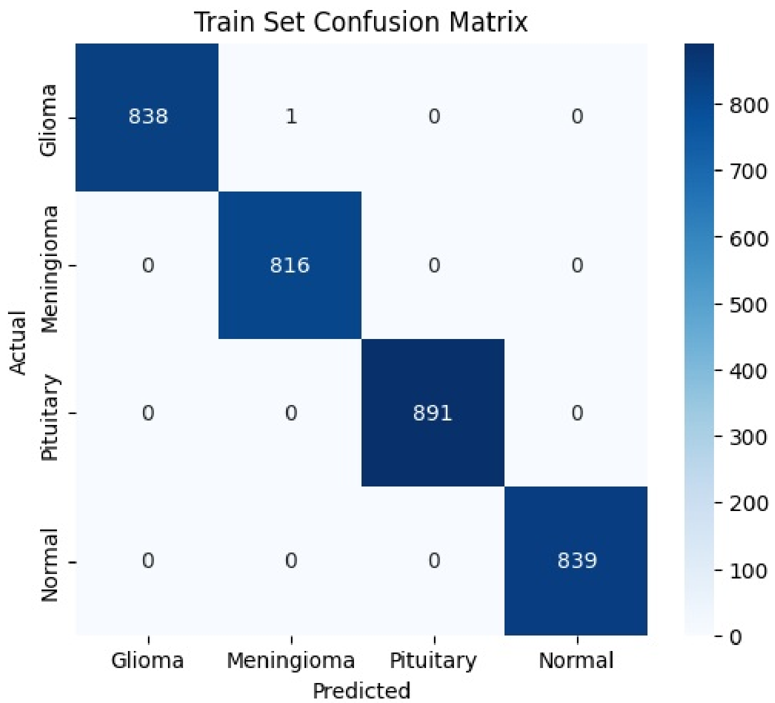
```python
        optimizer.zero_grad()
        outputs = model(imgs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        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_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as mobilevit_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))

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

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

    # Cohen's Kappa
    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:
        # ROC AUC + PR AUC
        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_best.pth"))

# Only plot curves for Test Set
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")
```

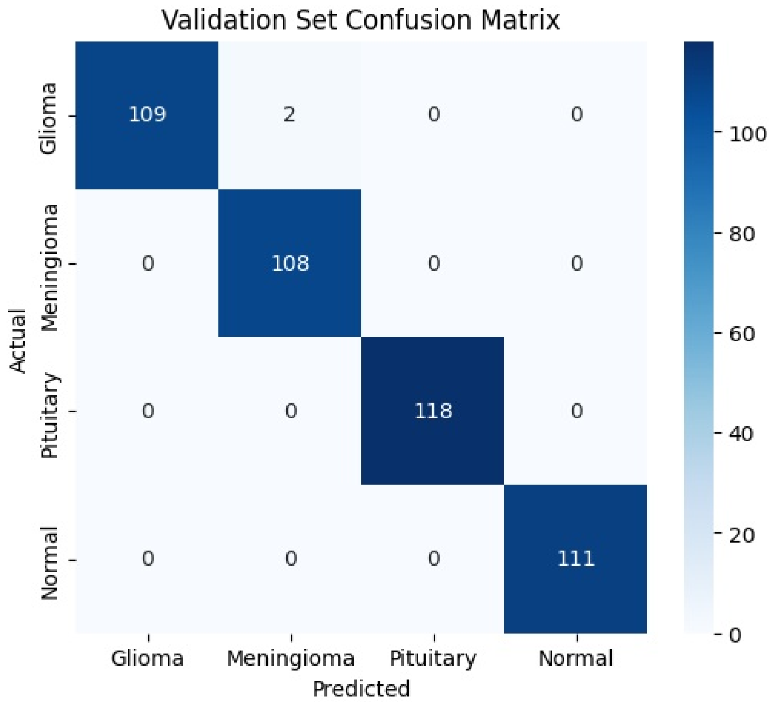
```
model.safetensors:   0%|          | 0.00/22.4M [00:00<?, ?B/s]
```

```
/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(
```

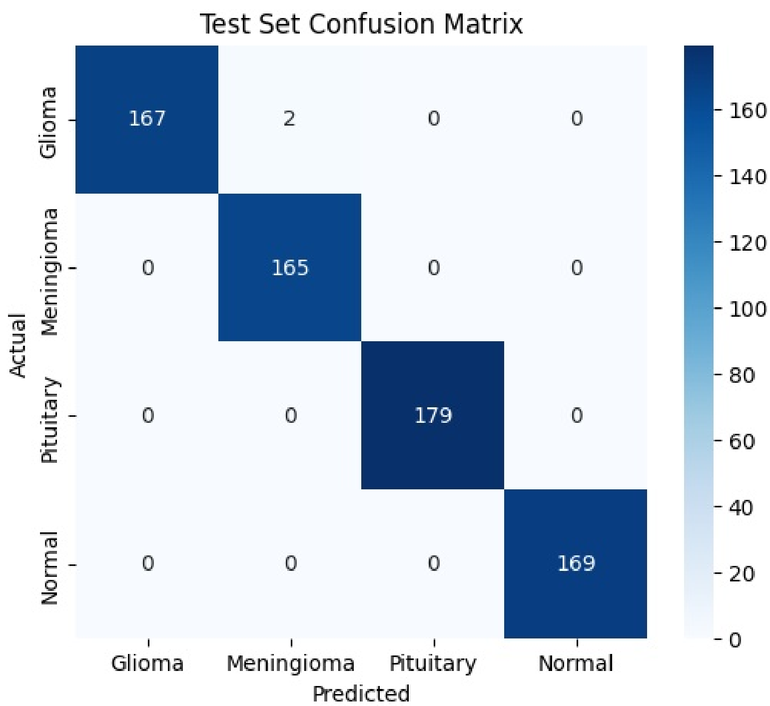
```
Epoch [1/35] Train Loss: 0.8362, Train Acc: 0.8399 Val Loss: 0.3077, Val Acc: 0.9353 Time: 21.01s

Epoch [2/35] Train Loss: 0.2154, Train Acc: 0.9533 Val Loss: 0.0899, Val Acc: 0.9844 Time: 20.64s
Epoch [3/35] Train Loss: 0.1087, Train Acc: 0.9758 Val Loss: 0.0689, Val Acc: 0.9844 Time: 20.76s
Epoch [4/35] Train Loss: 0.0678, Train Acc: 0.9861 Val Loss: 0.0349, Val Acc: 0.9933 Time: 20.66s
Epoch [5/35] Train Loss: 0.0437, Train Acc: 0.9911 Val Loss: 0.0398, Val Acc: 0.9888 Time: 20.64s
Epoch [6/35] Train Loss: 0.0348, Train Acc: 0.9911 Val Loss: 0.0734, Val Acc: 0.9754 Time: 20.65s
Epoch [7/35] Train Loss: 0.0249, Train Acc: 0.9938 Val Loss: 0.1708, Val Acc: 0.9487 Time: 20.59s
Epoch [8/35] Train Loss: 0.0225, Train Acc: 0.9950 Val Loss: 0.0462, Val Acc: 0.9888 Time: 20.74s
Epoch [9/35] Train Loss: 0.0179, Train Acc: 0.9953 Val Loss: 0.0310, Val Acc: 0.9888 Time: 20.68s
Epoch [10/35] Train Loss: 0.0172, Train Acc: 0.9968 Val Loss: 0.0217, Val Acc: 0.9933 Time: 20.82s
Epoch [11/35] Train Loss: 0.0124, Train Acc: 0.9976 Val Loss: 0.0239, Val Acc: 0.9933 Time: 20.63s
Epoch [12/35] Train Loss: 0.0108, Train Acc: 0.9970 Val Loss: 0.0211, Val Acc: 0.9955 Time: 20.76s
Epoch [13/35] Train Loss: 0.0108, Train Acc: 0.9970 Val Loss: 0.0306, Val Acc: 0.9911 Time: 20.83s
Epoch [14/35] Train Loss: 0.0115, Train Acc: 0.9982 Val Loss: 0.0380, Val Acc: 0.9911 Time: 20.73s
Epoch [15/35] Train Loss: 0.0063, Train Acc: 0.9988 Val Loss: 0.0372, Val Acc: 0.9933 Time: 20.74s
Epoch [16/35] Train Loss: 0.0068, Train Acc: 0.9985 Val Loss: 0.0384, Val Acc: 0.9911 Time: 20.74s
Epoch [17/35] Train Loss: 0.0161, Train Acc: 0.9970 Val Loss: 0.0283, Val Acc: 0.9933 Time: 20.73s
Early stopping triggered!
Training finished  Best model saved as mobilevit_best.pth

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

           0     1.0000    0.9988    0.9994       839
           1     0.9988    1.0000    0.9994       816
           2     1.0000    1.0000    1.0000       891
           3     1.0000    1.0000    1.0000       839

    accuracy                         0.9997      3385
   macro avg     0.9997    0.9997    0.9997      3385
weighted avg     0.9997    0.9997    0.9997      3385
```



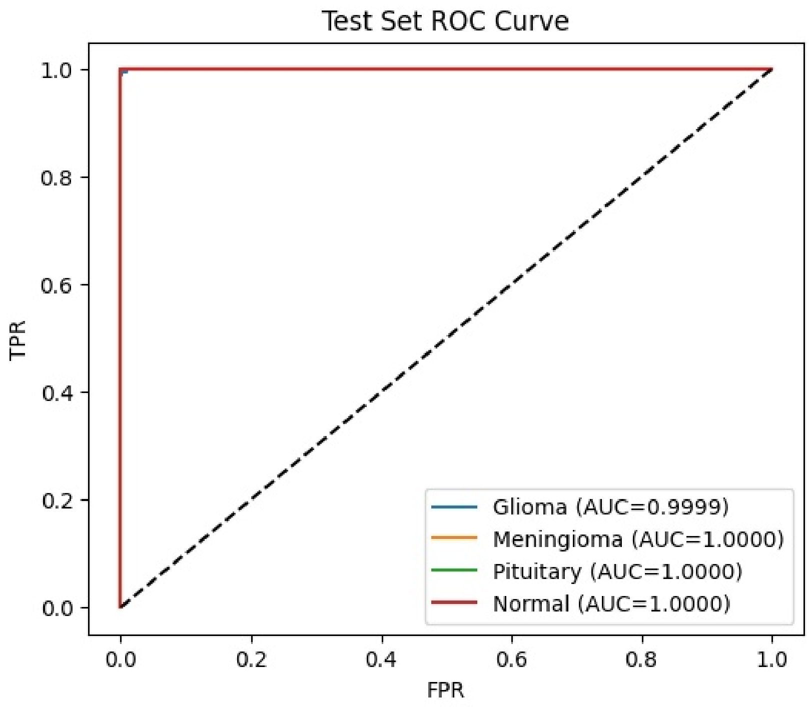

```
MCC: 0.9996
Cohen's Kappa: 0.9996
Mean NPV: 0.9999
Mean PPV (Precision): 0.9997
Train Set inference time: 6.48 sec

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

           0     1.0000    0.9820    0.9909       111
           1     0.9818    1.0000    0.9908       108
           2     1.0000    1.0000    1.0000       118
           3     1.0000    1.0000    1.0000       111

    accuracy                         0.9955       448
   macro avg     0.9955    0.9955    0.9954       448
weighted avg     0.9956    0.9955    0.9955       448
```

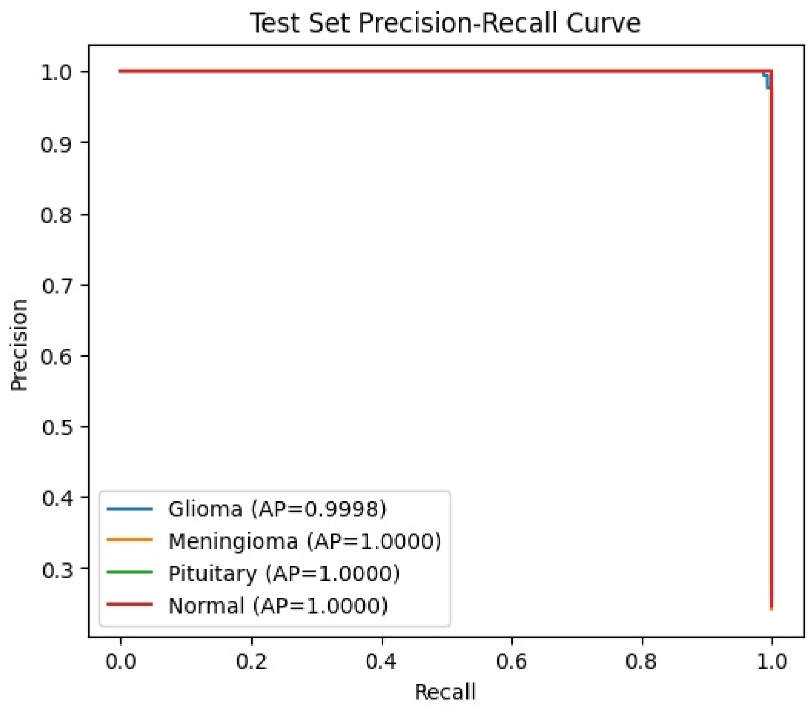



```
MCC: 0.9941
Cohen's Kappa: 0.9940
Mean NPV: 0.9985
Mean PPV (Precision): 0.9955
Validation Set inference time: 1.03 sec

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

           0     1.0000    0.9882    0.9940       169
           1     0.9880    1.0000    0.9940       165
           2     1.0000    1.0000    1.0000       179
           3     1.0000    1.0000    1.0000       169

    accuracy                         0.9971       682
   macro avg     0.9970    0.9970    0.9970       682
weighted avg     0.9971    0.9971    0.9971       682
```




```
MCC: 0.9961
Cohen's Kappa: 0.9961
Mean NPV: 0.9990
Mean PPV (Precision): 0.9970
ROC AUC: 1.0000, PR AUC: 1.0000
```






```
Test Set inference time: 1.56 sec

================ Summary ================
Training inference time: 6.48 sec
Validation inference time: 1.03 sec
Test inference time: 1.56 sec
```

In [8]:

```python
# ====================================================
# TinyViT Training + Evaluation Pipeline
# ====================================================

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 time
```

```python
# ========================
# Config
# ========================
data_dir = "/kaggle/working/augmented_split"
batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4  # Glioma, Meningioma, Pituitary, Normal
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# ========================
# Data (same as before, no new augmentation)
# ========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=f"{data_dir}/train", transform=common_tfms)
val_ds   = datasets.ImageFolder(root=f"{data_dir}/val", transform=common_tfms)
test_ds  = datasets.ImageFolder(root=f"{data_dir}/test", 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 (TinyViT)
# Options: "tiny_vit_5m_224", "tiny_vit_11m_224", "tiny_vit_21m_224"
# ========================
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)
    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
```

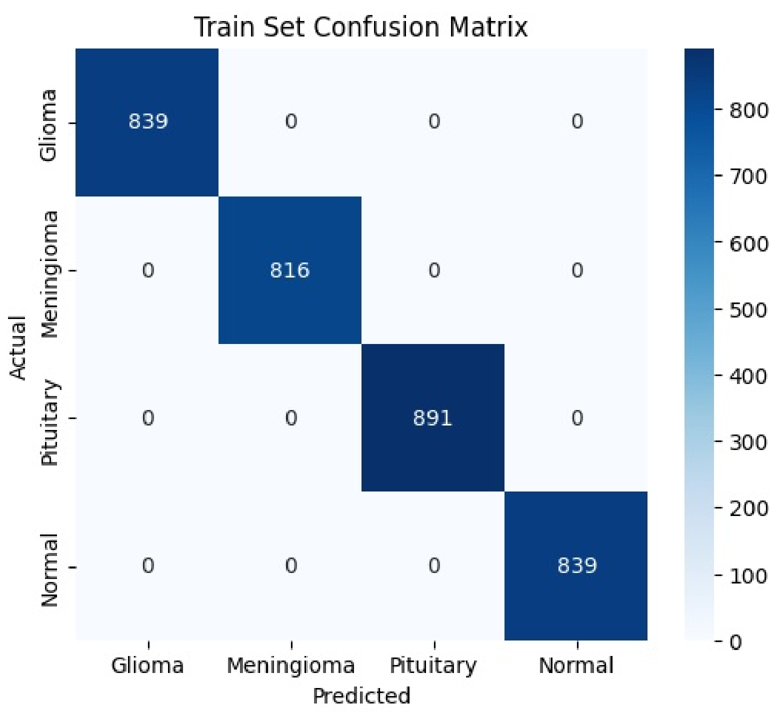
```
    print(f"Epoch [{epoch+1}/{num_epochs}] "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f} "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f} "
          f"Time: {elapsed:.2f}s")

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

print("Training finished  Best model saved as tinyvit_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))

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

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

    # Cohen's Kappa
    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:
        # ROC AUC + PR AUC
        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("tinyvit_best.pth"))

# Only plot curves for Test Set
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")
```

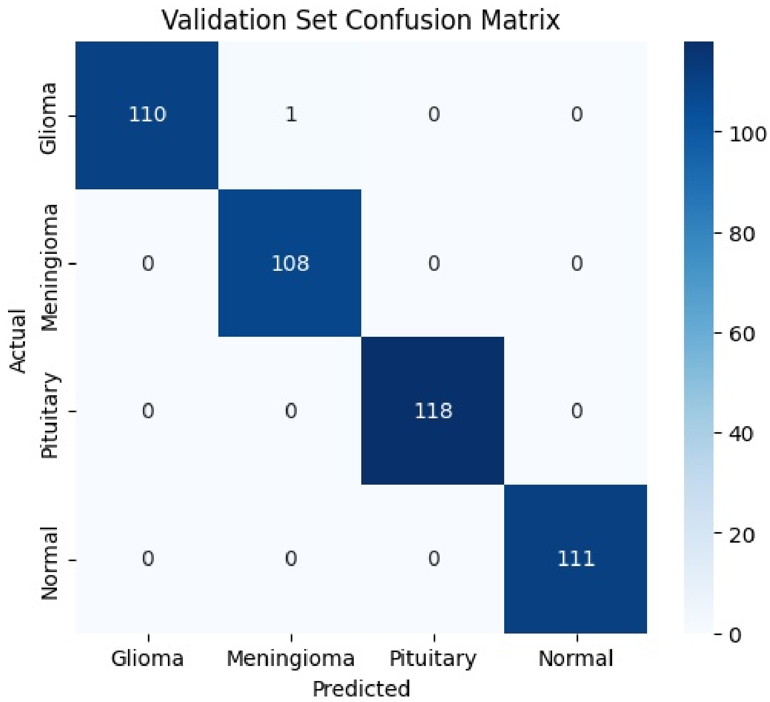
```
model.safetensors:   0%|          | 0.00/48.4M [00:00<?, ?B/s]
```

```
/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(
```

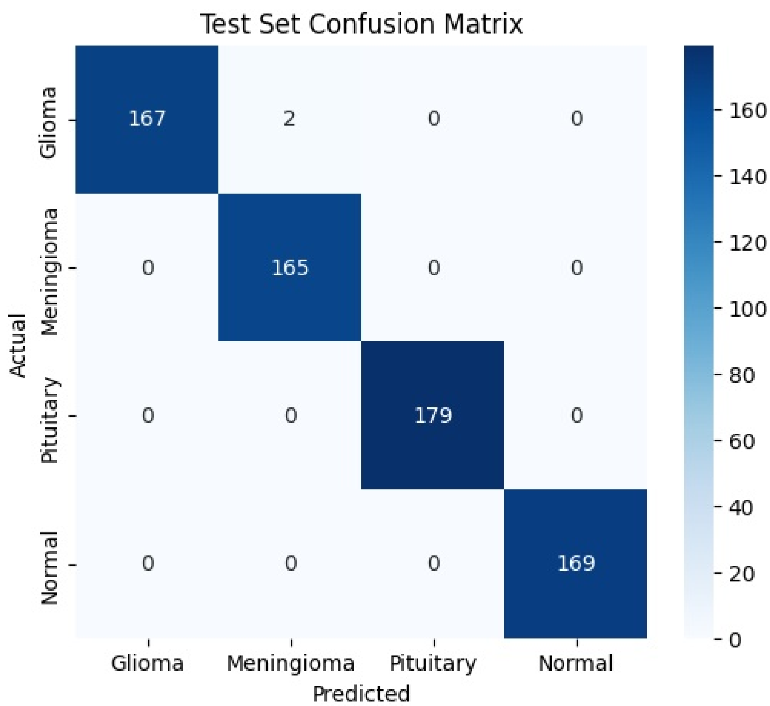
```
Epoch [1/35] Train Loss: 0.5656, Train Acc: 0.8331 Val Loss: 0.1971, Val Acc: 0.9621 Time: 16.38s
Epoch [2/35] Train Loss: 0.1211, Train Acc: 0.9761 Val Loss: 0.0870, Val Acc: 0.9821 Time: 16.20s
Epoch [3/35] Train Loss: 0.0416, Train Acc: 0.9953 Val Loss: 0.0353, Val Acc: 0.9978 Time: 16.26s
Epoch [4/35] Train Loss: 0.0221, Train Acc: 0.9973 Val Loss: 0.0331, Val Acc: 0.9955 Time: 16.16s
Epoch [5/35] Train Loss: 0.0128, Train Acc: 0.9988 Val Loss: 0.0361, Val Acc: 0.9933 Time: 16.33s
Epoch [6/35] Train Loss: 0.0105, Train Acc: 0.9991 Val Loss: 0.0283, Val Acc: 0.9955 Time: 16.17s
Epoch [7/35] Train Loss: 0.0132, Train Acc: 0.9973 Val Loss: 0.0208, Val Acc: 0.9978 Time: 16.18s
Epoch [8/35] Train Loss: 0.0134, Train Acc: 0.9973 Val Loss: 0.0325, Val Acc: 0.9933 Time: 16.21s
Epoch [9/35] Train Loss: 0.0245, Train Acc: 0.9905 Val Loss: 0.0961, Val Acc: 0.9732 Time: 16.30s
Epoch [10/35] Train Loss: 0.0120, Train Acc: 0.9976 Val Loss: 0.0222, Val Acc: 0.9978 Time: 16.04s
Epoch [11/35] Train Loss: 0.0071, Train Acc: 0.9988 Val Loss: 0.0218, Val Acc: 0.9955 Time: 16.13s
Epoch [12/35] Train Loss: 0.0040, Train Acc: 0.9997 Val Loss: 0.0190, Val Acc: 0.9978 Time: 16.24s
Epoch [13/35] Train Loss: 0.0046, Train Acc: 0.9991 Val Loss: 0.0215, Val Acc: 0.9955 Time: 16.17s
Epoch [14/35] Train Loss: 0.0024, Train Acc: 1.0000 Val Loss: 0.0210, Val Acc: 0.9955 Time: 16.10s
Epoch [15/35] Train Loss: 0.0021, Train Acc: 1.0000 Val Loss: 0.0202, Val Acc: 0.9978 Time: 16.11s
Epoch [16/35] Train Loss: 0.0019, Train Acc: 1.0000 Val Loss: 0.0212, Val Acc: 0.9955 Time: 16.18s
Epoch [17/35] Train Loss: 0.0018, Train Acc: 1.0000 Val Loss: 0.0196, Val Acc: 0.9978 Time: 16.08s
Early stopping triggered!
Training finished  Best model saved as tinyvit_best.pth

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

           0     1.0000    1.0000    1.0000       839
           1     1.0000    1.0000    1.0000       816
           2     1.0000    1.0000    1.0000       891
           3     1.0000    1.0000    1.0000       839

    accuracy                         1.0000      3385
   macro avg     1.0000    1.0000    1.0000      3385
weighted avg     1.0000    1.0000    1.0000      3385
```



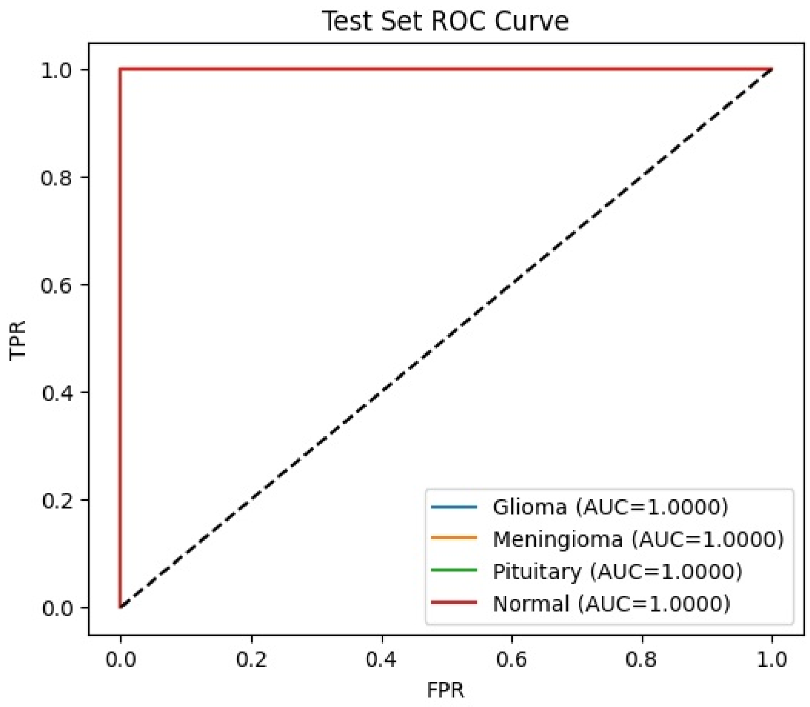

```
MCC: 1.0000
Cohen's Kappa: 1.0000
Mean NPV: 1.0000
Mean PPV (Precision): 1.0000
Train Set inference time: 5.46 sec

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

           0     1.0000    0.9910    0.9955       111
           1     0.9908    1.0000    0.9954       108
           2     1.0000    1.0000    1.0000       118
           3     1.0000    1.0000    1.0000       111

    accuracy                         0.9978       448
   macro avg     0.9977    0.9977    0.9977       448
weighted avg     0.9978    0.9978    0.9978       448
```

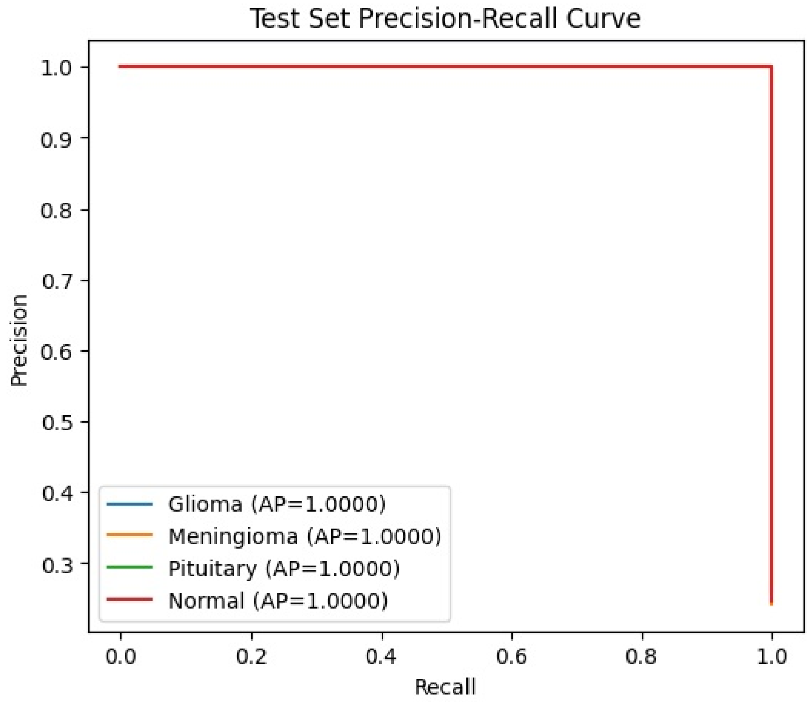



```
MCC: 0.9970
Cohen's Kappa: 0.9970
Mean NPV: 0.9993
Mean PPV (Precision): 0.9977
Validation Set inference time: 0.90 sec

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

           0     1.0000    0.9882    0.9940       169
           1     0.9880    1.0000    0.9940       165
           2     1.0000    1.0000    1.0000       179
           3     1.0000    1.0000    1.0000       169

    accuracy                         0.9971       682
   macro avg     0.9970    0.9970    0.9970       682
weighted avg     0.9971    0.9971    0.9971       682
```




```
MCC: 0.9961
Cohen's Kappa: 0.9961
Mean NPV: 0.9990
Mean PPV (Precision): 0.9970
ROC AUC: 1.0000, PR AUC: 1.0000
```






```
Test Set inference time: 1.27 sec

================ Summary ================
Training inference time: 5.46 sec
Validation inference time: 0.90 sec
Test inference time: 1.27 sec
```

In [11]:

```
# ====================================================
# LeViT Training + Evaluation Pipeline
# ====================================================

import os
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
```

```python
import time

# ========================
# Config
# ========================
data_dir = "/kaggle/working/augmented_split"
batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4  # Glioma, Meningioma, Pituitary, Normal
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class_names = ["Glioma", "Meningioma", "Pituitary", "Normal"]

# ========================
# Data (no extra augmentation)
# ========================
common_tfms = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.5]*3, [0.5]*3)
])

train_ds = datasets.ImageFolder(root=f"{data_dir}/train", transform=common_tfms)
val_ds   = datasets.ImageFolder(root=f"{data_dir}/val", transform=common_tfms)
test_ds  = datasets.ImageFolder(root=f"{data_dir}/test", 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 (LeViT)
# Options: "levit_128s", "levit_128", "levit_192", "levit_256"
# ========================
model = create_model("levit_128s", 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)
```

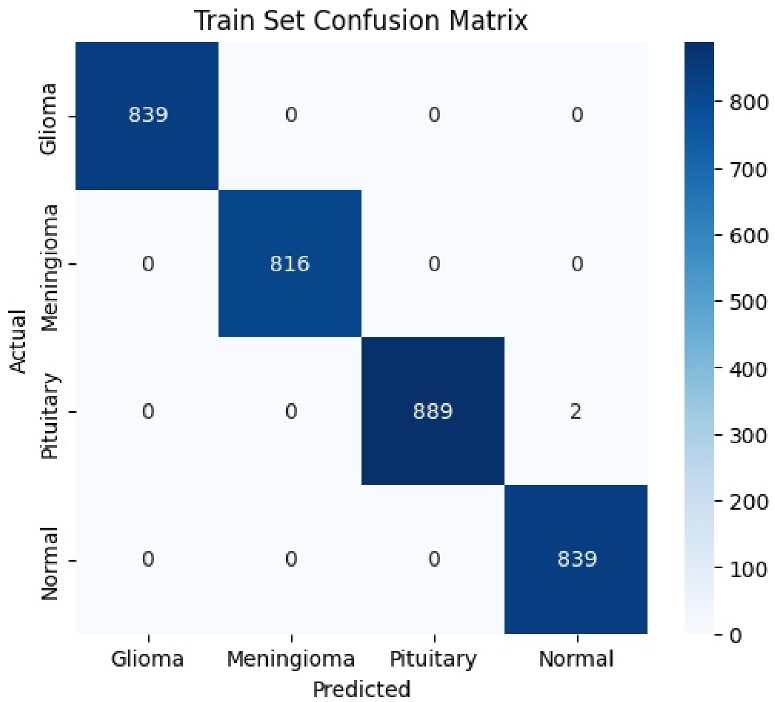
```python
    scheduler.step(val_loss)

    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(), "levit_best.pth")
    else:
        patience_counter += 1
        if patience_counter >= patience:
            print("Early stopping triggered!")
            break

print("Training finished  Best model saved as levit_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, target_names=class_names, digits=4))

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

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

    # Cohen's Kappa
    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:
        # ROC AUC + PR AUC
        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()
```

```python
        # --- 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("levit_best.pth"))

# Only plot curves for Test Set
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")
```

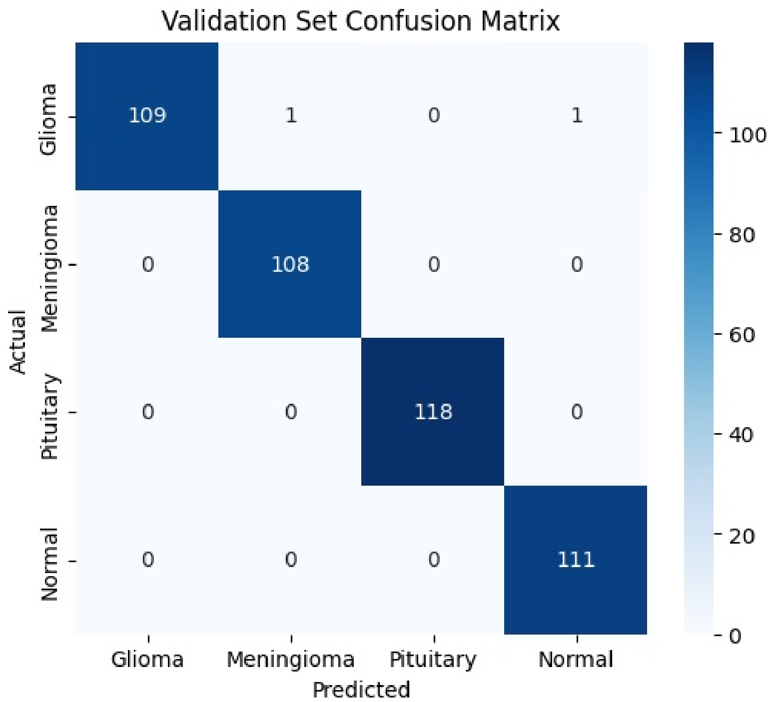
```
model.safetensors:   0%|          | 0.00/31.3M [00:00<?, ?B/s]
```

```
/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: 0.9599, Train Acc: 0.7288 Val Loss: 0.5414, Val Acc: 0.8661 Time: 6.76s
Epoch [2/35] Train Loss: 0.4157, Train Acc: 0.9019 Val Loss: 1.0715, Val Acc: 0.9286 Time: 6.76s
Epoch [3/35] Train Loss: 0.2169, Train Acc: 0.9518 Val Loss: 0.1574, Val Acc: 0.9665 Time: 7.51s
Epoch [4/35] Train Loss: 0.1163, Train Acc: 0.9808 Val Loss: 0.1213, Val Acc: 0.9732 Time: 7.53s
Epoch [5/35] Train Loss: 0.0676, Train Acc: 0.9935 Val Loss: 0.0763, Val Acc: 0.9866 Time: 7.55s
Epoch [6/35] Train Loss: 0.0447, Train Acc: 0.9950 Val Loss: 0.0612, Val Acc: 0.9911 Time: 7.40s
Epoch [7/35] Train Loss: 0.0341, Train Acc: 0.9962 Val Loss: 0.0730, Val Acc: 0.9777 Time: 7.75s
Epoch [8/35] Train Loss: 0.0295, Train Acc: 0.9970 Val Loss: 0.0783, Val Acc: 0.9844 Time: 7.35s
Epoch [9/35] Train Loss: 0.0197, Train Acc: 0.9979 Val Loss: 0.0656, Val Acc: 0.9866 Time: 7.46s
Epoch [10/35] Train Loss: 0.0209, Train Acc: 0.9965 Val Loss: 0.0349, Val Acc: 0.9866 Time: 7.40s
Epoch [11/35] Train Loss: 0.0144, Train Acc: 0.9988 Val Loss: 0.0428, Val Acc: 0.9866 Time: 6.76s
Epoch [12/35] Train Loss: 0.0132, Train Acc: 0.9985 Val Loss: 0.0442, Val Acc: 0.9844 Time: 7.07s
Epoch [13/35] Train Loss: 0.0132, Train Acc: 0.9985 Val Loss: 0.0426, Val Acc: 0.9888 Time: 7.29s
Epoch [14/35] Train Loss: 0.0118, Train Acc: 0.9988 Val Loss: 0.0305, Val Acc: 0.9955 Time: 7.09s
Epoch [15/35] Train Loss: 0.0082, Train Acc: 0.9994 Val Loss: 0.0372, Val Acc: 0.9888 Time: 7.23s
Epoch [16/35] Train Loss: 0.0086, Train Acc: 0.9997 Val Loss: 0.0428, Val Acc: 0.9866 Time: 7.46s
Epoch [17/35] Train Loss: 0.0104, Train Acc: 0.9988 Val Loss: 0.0333, Val Acc: 0.9888 Time: 7.01s
Epoch [18/35] Train Loss: 0.0078, Train Acc: 1.0000 Val Loss: 0.0389, Val Acc: 0.9866 Time: 6.78s
Epoch [19/35] Train Loss: 0.0089, Train Acc: 0.9994 Val Loss: 0.0455, Val Acc: 0.9866 Time: 7.12s
Early stopping triggered!
Training finished  Best model saved as levit_best.pth

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

      Glioma     1.0000    1.0000    1.0000       839
  Meningioma     1.0000    1.0000    1.0000       816
   Pituitary     1.0000    0.9978    0.9989       891
      Normal     0.9976    1.0000    0.9988       839


    accuracy                         0.9994      3385
   macro avg     0.9994    0.9994    0.9994      3385
weighted avg     0.9994    0.9994    0.9994      3385
```



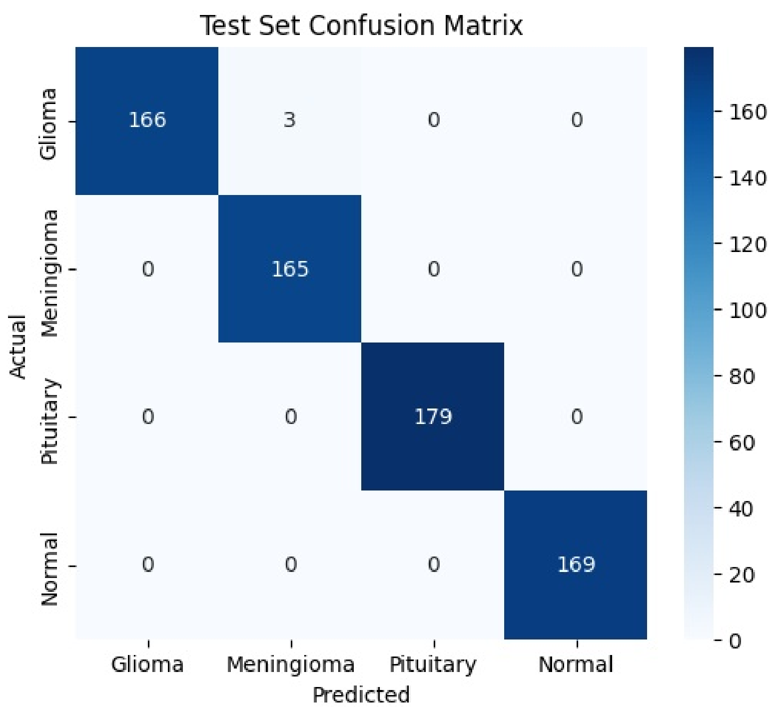

```
MCC: 0.9992
Cohen's Kappa: 0.9992
Mean NPV: 0.9998
Mean PPV (Precision): 0.9994
Train Set inference time: 4.81 sec

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

      Glioma     1.0000    0.9820    0.9909       111
  Meningioma     0.9908    1.0000    0.9954       108
   Pituitary     1.0000    1.0000    1.0000       118
      Normal     0.9911    1.0000    0.9955       111

    accuracy                         0.9955       448
   macro avg     0.9955    0.9955    0.9955       448
weighted avg     0.9956    0.9955    0.9955       448
```



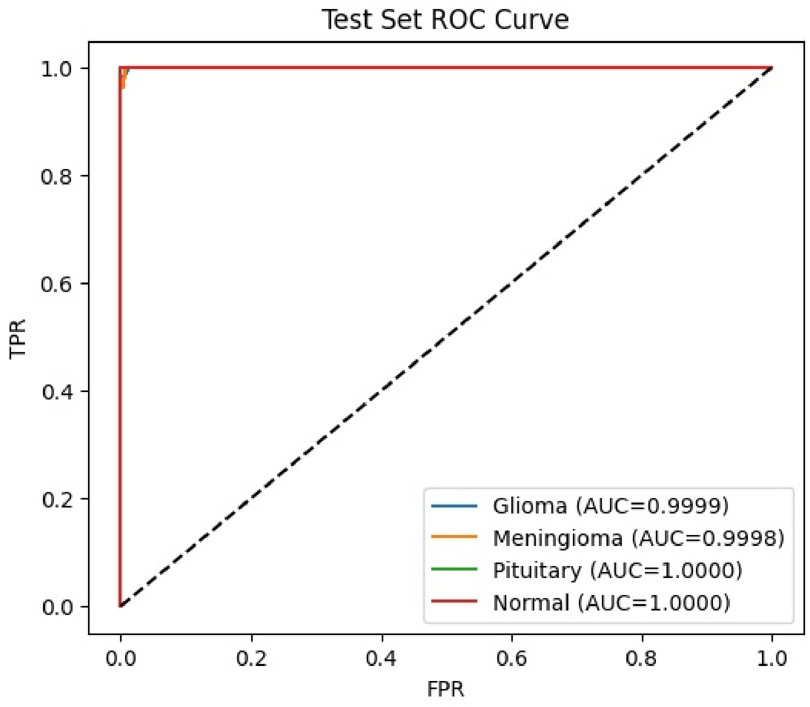

```
MCC: 0.9941
Cohen's Kappa: 0.9940
Mean NPV: 0.9985
Mean PPV (Precision): 0.9955
Validation Set inference time: 0.78 sec
```

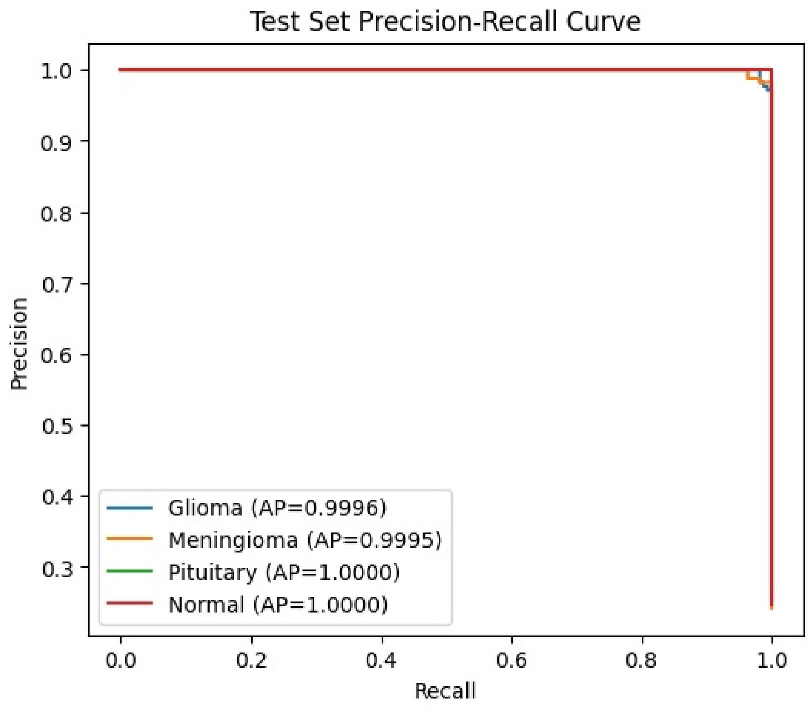
--- Test Set ---

	precision	recall	f1-score	support
Glioma	1.0000	0.9822	0.9910	169
Meningioma	0.9821	1.0000	0.9910	165
Pituitary	1.0000	1.0000	1.0000	179
Normal	1.0000	1.0000	1.0000	169
accuracy			0.9956	682
macro avg	0.9955	0.9956	0.9955	682
weighted avg	0.9957	0.9956	0.9956	682




```
MCC: 0.9942
Cohen's Kappa: 0.9941
Mean NPV: 0.9985
Mean PPV (Precision): 0.9955
ROC AUC: 0.9999, PR AUC: 0.9998
```






```
Test Set inference time: 1.09 sec

================ Summary ================
Training inference time: 4.81 sec
Validation inference time: 0.78 sec
Test inference time: 1.09 sec
```

In [39]:

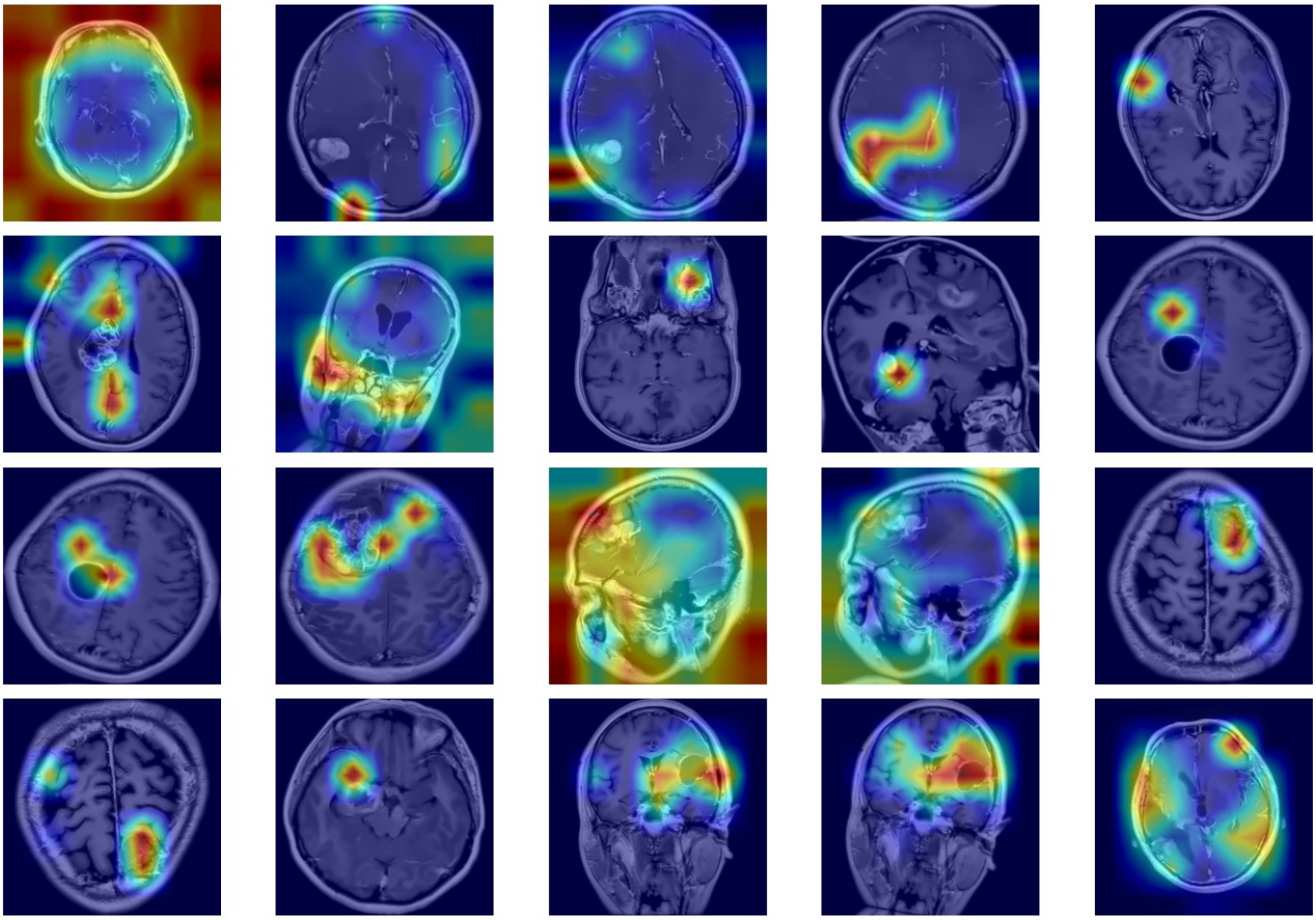
```
# ====================================================
# Full TinyViT Training + Grad-CAM Visualization
# ====================================================

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

import numpy as np
import matplotlib.pyplot as plt
import cv2
import time

# ========================
# Grad-CAM Class
```

```python
# ========================
class GradCAM:
    def __init__(self, model, target_layer):
        self.model = model
        self.target_layer = target_layer
        self.gradients = None
        self.activations = None
        self._register_hooks()

    def _register_hooks(self):
        def forward_hook(module, input, output):
            self.activations = output.detach()
        def backward_hook(module, grad_input, grad_output):
            self.gradients = grad_output[0].detach()

        self.target_layer.register_forward_hook(forward_hook)
        self.target_layer.register_full_backward_hook(backward_hook)

    def __call__(self, x, class_idx=None):
        self.model.eval()
        x = x.to(next(self.model.parameters()).device)
        output = self.model(x)

        if class_idx is None:
            class_idx = output.argmax(dim=1)[0]

        self.model.zero_grad()
        loss = output[0, class_idx]
        loss.backward(retain_graph=True)

        weights = self.gradients.mean(dim=(2, 3), keepdim=True)
        cam = (weights * self.activations).sum(dim=1, keepdim=True)
        cam = torch.nn.functional.relu(cam)
        cam = torch.nn.functional.interpolate(cam, size=(x.size(2), x.size(3)), mode='bilinear', align_corners=F
        cam = cam.squeeze().cpu().numpy()
        cam = (cam - cam.min()) / (cam.max() - cam.min() + 1e-8)
        return cam

# ========================
# Config
# ========================
data_dir = "/kaggle/working/augmented_split"
batch_size = 32
num_epochs = 35
patience = 5
num_classes = 4  # Glioma, Meningioma, Pituitary, Normal
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

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

train_ds = datasets.ImageFolder(root=f"{data_dir}/train", transform=common_tfms)
val_ds   = datasets.ImageFolder(root=f"{data_dir}/val", transform=common_tfms)
test_ds  = datasets.ImageFolder(root=f"{data_dir}/test", 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 = 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_loss /= len(train_loader.dataset)

    # ---- Validation ----
    model.eval()
    val_loss = 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_loss /= len(val_loader.dataset)
    scheduler.step(val_loss)

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

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

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

# ========================
# Grad-CAM Visualization
# ========================
target_layer = model.stages[-1].blocks[-1].local_conv
grad_cam = GradCAM(model, target_layer)

# Collect first 20 test images
imgs_collected = []
labels_collected = []
for imgs, labels in test_loader:
    for i in range(len(imgs)):
        imgs_collected.append(imgs[i])
        labels_collected.append(labels[i].item())
        if len(imgs_collected) >= 20:
            break
    if len(imgs_collected) >= 20:
        break

# Plot Grad-CAM
plt.figure(figsize=(15, 10))
for idx in range(20):
    img = imgs_collected[idx].unsqueeze(0)
    label = labels_collected[idx]

    mask = grad_cam(img, class_idx=label)

    plt.subplot(4, 5, idx+1)
    img_np = img[0].permute(1,2,0).cpu().numpy()
    img_np = (img_np - img_np.min()) / (img_np.max() - img_np.min())
    heatmap = cv2.applyColorMap(np.uint8(255*mask), cv2.COLORMAP_JET)
    heatmap = cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB) / 255.0
    cam_img = 0.5*heatmap + 0.5*img_np

    plt.imshow(cam_img)
    plt.axis('off')

plt.tight_layout()
```

```
plt.show()
```

```
/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: 0.5339 Val Loss: 0.2788 Time: 16.08s
Epoch [2/35] Train Loss: 0.1072 Val Loss: 0.0554 Time: 16.03s
Epoch [3/35] Train Loss: 0.0434 Val Loss: 0.1607 Time: 16.04s
Epoch [4/35] Train Loss: 0.0299 Val Loss: 0.0509 Time: 16.04s
Epoch [5/35] Train Loss: 0.0134 Val Loss: 0.0233 Time: 16.09s
Epoch [6/35] Train Loss: 0.0295 Val Loss: 0.0263 Time: 16.15s
Epoch [7/35] Train Loss: 0.0109 Val Loss: 0.0301 Time: 15.98s
Epoch [8/35] Train Loss: 0.0101 Val Loss: 0.0245 Time: 16.12s
Epoch [9/35] Train Loss: 0.0070 Val Loss: 0.0184 Time: 16.13s
Epoch [10/35] Train Loss: 0.0040 Val Loss: 0.0179 Time: 16.02s
Epoch [11/35] Train Loss: 0.0036 Val Loss: 0.0190 Time: 16.11s
Epoch [12/35] Train Loss: 0.0030 Val Loss: 0.0211 Time: 16.01s
Epoch [13/35] Train Loss: 0.0028 Val Loss: 0.0207 Time: 16.13s
Epoch [14/35] Train Loss: 0.0022 Val Loss: 0.0188 Time: 16.13s
Epoch [15/35] Train Loss: 0.0023 Val Loss: 0.0187 Time: 16.07s
Early stopping triggered!
Training finished  Best model saved as tinyvit_best.pth
```



In [7]:

```
# ====================================================
# TinyViT + 5-Fold Cross Validation
# ====================================================

# ------------------------
# Imports
# ------------------------
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
from timm import create_model
from torch.optim.lr_scheduler import ReduceLROnPlateau

import numpy as np
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# ------------------------
# Config
# ------------------------
data_dir = "/kaggle/working/augmented_split"
batch_size = 32
```

```python
num_epochs = 35
patience = 5
num_classes = 4   # Glioma, Meningioma, Pituitary, Normal
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

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

# Test dataset (kept separate for final evaluation)
test_ds = datasets.ImageFolder(root=f"{data_dir}/test", transform=common_tfms)
test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=2)

# For CV (use only train split here)
full_ds = datasets.ImageFolder(root=f"{data_dir}/train", transform=common_tfms)
X = np.arange(len(full_ds))
y = [label for _, label in full_ds.samples]

# ------------------------
# Stratified 5-Fold CV
# ------------------------
k_folds = 5
skf = StratifiedKFold(n_splits=k_folds, shuffle=True, random_state=42)

fold_metrics = {"accuracy": [], "precision": [], "recall": [], "f1": []}

for fold, (train_idx, val_idx) in enumerate(skf.split(X, y), 1):
    print(f"\n========== Fold {fold} ==========")

    # Subset datasets
    train_subset = Subset(full_ds, train_idx)
    val_subset   = Subset(full_ds, val_idx)

    train_loader = DataLoader(train_subset, batch_size=batch_size, shuffle=True, num_workers=2)
    val_loader   = DataLoader(val_subset, batch_size=batch_size, shuffle=False, num_workers=2)

    # Reinitialize model for each fold
    model = create_model("tiny_vit_5m_224", pretrained=True, num_classes=num_classes).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=False)

    best_val_loss = float("inf")
    patience_counter = 0

    # ------------------------
    # Training Loop
    # ------------------------
    for epoch in range(num_epochs):
        # --- Train ---
        model.train()
        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()

        # --- Validation ---
        model.eval()
        val_loss, y_true, y_pred = 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)
                preds = outputs.argmax(1)
                y_true.extend(labels.cpu().numpy())
                y_pred.extend(preds.cpu().numpy())

        val_loss /= len(val_loader.dataset)
        scheduler.step(val_loss)

        # Early stopping
        if val_loss < best_val_loss:
            best_val_loss = val_loss
```

```
            patience_counter = 0
            best_y_true, best_y_pred = y_true[:], y_pred[:]
        else:
            patience_counter += 1
            if patience_counter >= patience:
                break

    # ------------------------
    # Metrics for this fold
    # ------------------------
    acc  = accuracy_score(best_y_true, best_y_pred)
    prec = precision_score(best_y_true, best_y_pred, average="macro", zero_division=0)
    rec  = recall_score(best_y_true, best_y_pred, average="macro", zero_division=0)
    f1   = f1_score(best_y_true, best_y_pred, average="macro", zero_division=0)

    fold_metrics["accuracy"].append(acc)
    fold_metrics["precision"].append(prec)
    fold_metrics["recall"].append(rec)
    fold_metrics["f1"].append(f1)

    print(f"Fold {fold} - Acc: {acc:.4f}, Prec: {prec:.4f}, Recall: {rec:.4f}, F1: {f1:.4f}")

# ------------------------
# Summary (mean ± std)
# ------------------------
print("\n===== Cross-Validation Results =====")
for metric, values in fold_metrics.items():
    mean, std = np.mean(values), np.std(values)
    print(f"{metric.capitalize()}: {mean:.4f} ± {std:.4f}")
```

```
========== Fold 1 ==========
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(
Fold 1 - Acc: 0.9970, Prec: 0.9970, Recall: 0.9971, F1: 0.9970

========== Fold 2 ==========
/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(
Fold 2 - Acc: 0.9970, Prec: 0.9970, Recall: 0.9970, F1: 0.9970

========== Fold 3 ==========
/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(
Fold 3 - Acc: 0.9970, Prec: 0.9970, Recall: 0.9970, F1: 0.9970

========== Fold 4 ==========
/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(
Fold 4 - Acc: 1.0000, Prec: 1.0000, Recall: 1.0000, F1: 1.0000

========== Fold 5 ==========
/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(
Fold 5 - Acc: 0.9985, Prec: 0.9985, Recall: 0.9985, F1: 0.9985

===== Cross-Validation Results =====
Accuracy: 0.9979 ± 0.0012
Precision: 0.9979 ± 0.0012
Recall: 0.9979 ± 0.0012
F1: 0.9979 ± 0.0012
```