## Path

```
path='/kaggle/input/brian-tumor-dataset'
```

## Importing Libraries

```
import os

import random

#import Torch
import torch
import torch.nn as nnx
import torch.optim as optimx
from torchvision import datasets, models, transforms
from torch.optim.lr_scheduler import ReduceLROnPlateau
from tqdm import tqdm
from torch.utils.data import random_split, DataLoader

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score, recall_score, precision_score, confusion_matrix, roc_auc_score, roc_curve
import matplotlib.pyplot as pltx
import numpy as npx
import seaborn as snsx
import pandas as pdx
from PIL import Image

import cv2
import timm
```

Check Availability of Cuda

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```

Show hidden output

## Spliting Data in Brain Tumor and Healthy

```
yes_dir = os.path.join(path, 'Brain Tumor Data Set', 'Brain Tumor Data Set','Brain Tumor')
no_dir = os.path.join(path, 'Brain Tumor Data Set', 'Brain Tumor Data Set','Healthy')

# List all files
yes_files = [os.path.join(yes_dir, f) for f in os.listdir(yes_dir) if os.path.isfile(os.path.join(yes_dir, f))]
no_files = [os.path.join(no_dir, f) for f in os.listdir(no_dir) if os.path.isfile(os.path.join(no_dir, f))]

# Select a subset of files
num_files_to_select = 1500

import random

selected_yes_files = random.sample(yes_files, min(num_files_to_select, len(yes_files)))
selected_no_files = random.sample(no_files, min(num_files_to_select, len(no_files)))
selected_files=selected_yes_files+selected_no_files
print(f"Selected {len(selected_yes_files)} 'yes' files")
print(f"Selected {len(selected_no_files)} 'no' files")
```

Show hidden output

```
file_types = ['Brain Tumor (Yes)', 'Healthy (No)']
counts = [len(selected_yes_files), len(selected_no_files)]

pltx.figure(figsize=(8, 5))
pltx.bar(file_types, counts, color=['skyblue', 'lightgreen'])
pltx.title("Distribution of Selected Brain Tumor and Healthy Files")
pltx.xlabel("File Type")
pltx.ylabel("Number of Files")
pltx.ylim(0,  len(selected_files)* 1.1) # Set y-axis limit slightly above max count
pltx.grid(axis='y', linestyle='--', alpha=0.7)

# Add labels on the bars
```

```
for i, count in enumerate(counts):
    pltx.text(i, count + 20, str(count), ha='center', va='bottom')

pltx.show()
```

Show hidden output

## Visualization

```
# Define the image transformation (resizing)
data_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

# Load a few images and apply the transformation
num_images_to_visualize = 5
sample_files = random.sample(selected_files, num_images_to_visualize)

pltx.figure(figsize=(15, 5))
for i, file_path in enumerate(sample_files):
    image = Image.open(file_path).convert('RGB')
    transformed_image = data_transform(image)

    # Convert the tensor back to a PIL Image for displaying
    transformed_image_display = transforms.ToPILImage()(transformed_image)

    pltx.subplot(1, num_images_to_visualize, i + 1)
    pltx.imshow(transformed_image_display)
    pltx.title(f"Resized Image {i+1}")
    pltx.axis('off')
pltx.tight_layout()
pltx.show()
```

Show hidden output

```
# Visualize a single image as a histogram of pixel values
if sample_files:
    image = Image.open(sample_files[0]).convert('RGB')
    transformed_image = data_transform(image)

    # Convert tensor to numpy array for histogram
    img_np = transformed_image.numpy()

    pltx.figure(figsize=(10, 5))

    # Plot histogram for each channel (R, G, B)
    colors = ['red', 'green', 'blue']
    for channel, color in enumerate(colors):
        pltx.hist(img_np[channel].flatten(), bins=50, color=color, alpha=0.5, label=f'{color.capitalize()} Channel')

    pltx.title("Pixel Value Histogram of a Resized Image")
    pltx.xlabel("Pixel Value")
    pltx.ylabel("Frequency")
    pltx.legend()
    pltx.show()
```

Show hidden output

```
# Visualize a heatmap of pixel values
if sample_files:
    image = Image.open(sample_files[1]).convert('RGB')
    transformed_image = data_transform(image)

    # Calculate the average across color channels for a grayscale heatmap
    avg_image = torch.mean(transformed_image, dim=0)

    pltx.figure(figsize=(6, 6))
    pltx.imshow(avg_image.numpy(), cmap='viridis') # Use a colormap like 'viridis' or 'plasma'
    pltx.title("Heatmap of Pixel Intensities (Average)")
    pltx.colorbar(label='Average Pixel Intensity')
    pltx.axis('off')
    pltx.show()
```

Show hidden output

## Train, Validate and Test Dataframe splits

```
# Split the data into train, validation, and test sets
train_df, test_df = train_test_split(selected_files, train_size = 0.95, random_state = 0)
train_df,valid_df = train_test_split(train_df, train_size=0.9,random_state = 0)


print(f"\nNumber of samples in training set: {len(train_df)}")
print(f"Number of samples in validation set: {len(valid_df)}")
print(f"Number of samples in test set: {len(test_df)}")

# Calculate percentages
total_samples = len(selected_files)
train_percent = (len(train_df) / total_samples) * 100
valid_percent = (len(valid_df) / total_samples) * 100
test_percent = (len(test_df) / total_samples) * 100

print(f"\nNumber of samples in training set: {len(train_df)} ({train_percent:.2f}%)")
print(f"Number of samples in validation set: {len(valid_df)} ({valid_percent:.2f}%)")
print(f"Number of samples in test set: {len(test_df)} ({test_percent:.2f}%)")
```

Show hidden output

Distribution of classes in Train, Validate and Test Sets

```
# Add a pie chart showing the percentage distribution of the splits (train, validation, test)
split_sizes = [len(train_df), len(valid_df), len(test_df)]
split_labels = [f'Train ({len(train_df)})', f'Validation ({len(valid_df)})', f'Test ({len(test_df)})']
colors = ['gold', 'lightcoral', 'lightskyblue']
explode = (0.1, 0.1, 0.1)

fig_shifted = pltx.figure(figsize=(10, 10))
ax_shifted = fig_shifted.add_axes([0.4, 0.3, 0.4, 0.4])

ax_shifted.pie(split_sizes, explode=explode, labels=split_labels, colors=colors, autopct='%1.1f%%',
               shadow=True, startangle=140, wedgeprops={'edgecolor': 'white'})
ax_shifted.axis('equal')
ax_shifted.set_title("Data Split Distribution (Train, Validation, Test)", y=1.05) # Adjust title position

pltx.show()

```

Show hidden output

```
# Function to count 'yes' and 'no' files in a list of file paths
def count_yes_no(file_list, yes_dir, no_dir):
  yes_count = 0
  no_count = 0
  for file_path in file_list:
    if file_path.startswith(yes_dir):
      yes_count += 1
    elif file_path.startswith(no_dir):
      no_count += 1
  return yes_count, no_count

# Count for training set
train_yes_count, train_no_count = count_yes_no(train_df, yes_dir, no_dir)
print(f"\nTraining Set: Yes = {train_yes_count}, No = {train_no_count}")

# Count for validation set
valid_yes_count, valid_no_count = count_yes_no(valid_df, yes_dir, no_dir)
print(f"Validation Set: Yes = {valid_yes_count}, No = {valid_no_count}")

# Count for test set
test_yes_count, test_no_count = count_yes_no(test_df, yes_dir, no_dir)
print(f"Test Set: Yes = {test_yes_count}, No = {test_no_count}")

```

Show hidden output

## Transforming the data

```
# Define the custom CLAHE transform class
class ApplyCLAHE:
    """Apply CLAHE (Contrast Limited Adaptive Histogram Equalization) to an image."""
    def __init__(self, clipLimit=2.0, tileGridSize=(8,8)):
```

```
        """
        Args:
            clipLimit (float): Threshold for contrast limiting.
            tileGridSize (tuple): Size of the grid for histogram equalization.
        """
        # Initialize CLAHE in the constructor
        self.clahe = cv2.createCLAHE(clipLimit=clipLimit, tileGridSize=tileGridSize)

    def __call__(self, img):
        """
        Applies CLAHE to the input image.
        Args:
            img (PIL Image): Image to be processed.

        Returns:
            PIL Image: Processed image with CLAHE applied.
        """
        # Convert PIL Image to NumPy array
        img_np = npx.array(img)
        if len(img_np.shape) == 3 and img_np.shape[2] == 3:
            clahe_img_np = npx.zeros_like(img_np)
            for i in range(img_np.shape[-1]):
                channel = img_np[:, :, i]
                clahe_img_np[:, :, i] = self.clahe.apply(channel)
        elif len(img_np.shape) == 2 or (len(img_np.shape) == 3 and img_np.shape[2] == 1):
             if len(img_np.shape) == 3:
                 img_np = img_np[:, :, 0]
             clahe_img_np = self.clahe.apply(img_np)
             if len(img_np.shape) == 3 and img_np.shape[2] == 1:
                 clahe_img_np = npx.expand_dims(clahe_img_np, axis=-1)
        else:
             print("Warning: Image format not supported for CLAHE. Returning original image.")
             return img # Or raise an error
        return Image.fromarray(clahe_img_np)
data_transform_with_clahe = transforms.Compose([
    ApplyCLAHE(clipLimit=2.0, tileGridSize=(8,8)), # Instantiate with desired parameters
    transforms.Resize((224, 224)) ,
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
if sample_files:
    sample_image_path = sample_files[0]
    original_image = Image.open(sample_image_path).convert('RGB') # Ensure RGB for consistency

    # Apply the combined transform
    transformed_image_with_clahe = data_transform_with_clahe(original_image)
    transformed_image_display = transforms.ToPILImage()(transformed_image_with_clahe)


    pltx.figure(figsize=(6, 3))
    pltx.subplot(1, 2, 1)
    pltx.imshow(original_image)
    pltx.title("Original Image")
    pltx.axis('off')

    pltx.subplot(1, 2, 2)
    if transformed_image_display.mode != 'RGB':
         transformed_image_display = transformed_image_display.convert('RGB')

    pltx.imshow(transformed_image_display)
    pltx.title("CLAHE Applied (Normalized)")
    pltx.axis('off')
    pltx.show()
```

Show hidden output

## Creating Data Loader

```
class TumorDataset(torch.utils.data.Dataset):
    def __init__(self, file_list, yes_dir, transform=None):
        self.file_list = file_list
        self.yes_dir = yes_dir
        self.transform = transform

    def __len__(self):
        return len(self.file_list)

    def __getitem__(self, idx):
        img_path = self.file_list[idx]
        image = Image.open(img_path).convert('RGB') # Ensure RGB
```

```python
        label = 1 if img_path.startswith(self.yes_dir) else 0

        if self.transform:
            image = self.transform(image)

        return image, label

train_dataset = TumorDataset(train_df, yes_dir, transform=data_transform_with_clahe)
valid_dataset = TumorDataset(valid_df, yes_dir, transform=data_transform_with_clahe)
test_dataset = TumorDataset(test_df, yes_dir, transform=data_transform_with_clahe)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=2)
valid_loader = DataLoader(valid_dataset, batch_size=32, shuffle=False, num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False, num_workers=2)
```

## Base Class for training and testing

```python
class BaseTrainer:
    def __init__(self, model, train_loader, val_loader, test_loader,
                 criterion, optimizer, scheduler=None, device='cuda',
                 early_stopping_patience=5, model_name='best_model.pt'):
        self.model = model.to(device)
        self.train_loader = train_loader
        self.val_loader = val_loader
        self.test_loader = test_loader
        self.criterion = criterion
        self.optimizer = optimizer
        self.scheduler = scheduler
        self.device = device
        self.early_stopping_patience = early_stopping_patience
        self.model_name = model_name

        # History
        self.train_losses, self.val_losses = [], []
        self.train_accuracies, self.val_accuracies = [], []

    def train(self, num_epochs):
        best_val_loss = npx.inf
        patience_counter = 0

        for epoch in range(num_epochs):
            self.model.train()
            train_loss, train_preds, train_targets = 0.0, [], []
            for inputs, labels in self.train_loader:
                inputs, labels = inputs.to(self.device), labels.to(self.device)
                self.optimizer.zero_grad()

                outputs = self.model(inputs)
                loss = self.criterion(outputs, labels)
                loss.backward()
                self.optimizer.step()

                train_loss += loss.item()
                preds = torch.argmax(outputs, dim=1)
                train_preds.extend(preds.cpu().numpy())
                train_targets.extend(labels.cpu().numpy())

            epoch_train_loss = train_loss / len(self.train_loader)
            epoch_train_acc = accuracy_score(train_targets, train_preds)
            self.train_losses.append(epoch_train_loss)
            self.train_accuracies.append(epoch_train_acc)

            # Validation
            val_loss, val_acc = self.evaluate(self.val_loader)
            self.val_losses.append(val_loss)
            self.val_accuracies.append(val_acc)

            print(f"Epoch [{epoch+1}/{num_epochs}] "
                  f"Train Loss: {epoch_train_loss:.4f} Acc: {epoch_train_acc:.4f} | "
                  f"Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}")

            if self.scheduler:
                self.scheduler.step(val_loss)

            # Early stopping
            if val_loss < best_val_loss:
                best_val_loss = val_loss
                patience_counter = 0
                torch.save(self.model.state_dict(), self.model_name)
```

```python
            else:
                patience_counter += 1
                if patience_counter >= self.early_stopping_patience:
                    print("Early stopping triggered.")
                    break

        self.plot_accuracy_loss()

    def evaluate(self, loader):
        self.model.eval()
        loss_total, preds_all, labels_all = 0.0, [], []

        with torch.no_grad():
            for inputs, labels in loader:
                inputs, labels = inputs.to(self.device), labels.to(self.device)
                outputs = self.model(inputs)
                loss = self.criterion(outputs, labels)

                loss_total += loss.item()
                preds = torch.argmax(outputs, dim=1)
                preds_all.extend(preds.cpu().numpy())
                labels_all.extend(labels.cpu().numpy())

        loss_avg = loss_total / len(loader)
        acc = accuracy_score(labels_all, preds_all)
        return loss_avg, acc

    def test(self):
        self.model.load_state_dict(torch.load(self.model_name))
        self.model.eval()
        preds_all, labels_all, probs_all = [], [], []

        with torch.no_grad():
            for inputs, labels in self.test_loader:
                inputs, labels = inputs.to(self.device), labels.to(self.device)
                outputs = self.model(inputs)

                probs = torch.softmax(outputs, dim=1)
                preds = torch.argmax(probs, dim=1)

                preds_all.extend(preds.cpu().numpy())
                labels_all.extend(labels.cpu().numpy())
                probs_all.extend(probs[:, 1].cpu().numpy())  # Assumes binary classification

        self.print_metrics(labels_all, preds_all)
        self.plot_confusion_matrix(labels_all, preds_all)
        self.plot_roc_curve(labels_all, probs_all)

    def save_model(self, path=None):
        """Save the current model state dict."""
        if path is None:
            path = self.model_name
        torch.save(self.model.state_dict(), path)
        print(f"Model saved to {path}")

    def load_model(self, path=None):
        """Load the model state dict from file."""
        if path is None:
            path = self.model_name
        self.model.load_state_dict(torch.load(path, map_location=self.device))
        self.model.to(self.device)
        self.model.eval()
        print(f"Model loaded from {path}")

    def print_metrics(self, y_true, y_pred):
        print(f"Accuracy:  {accuracy_score(y_true, y_pred):.4f}")
        print(f"Precision: {precision_score(y_true, y_pred):.4f}")
        print(f"Recall:    {recall_score(y_true, y_pred):.4f}")
        print(f"F1 Score:  {f1_score(y_true, y_pred):.4f}")

    def plot_confusion_matrix(self, y_true, y_pred):
        cm = confusion_matrix(y_true, y_pred)
        pltx.figure(figsize=(4, 4))
        pltx.imshow(cm, cmap='Blues')
        pltx.title("Confusion Matrix")
        pltx.colorbar()
        pltx.xlabel("Predicted")
        pltx.ylabel("True")
        for i in range(len(cm)):
            for j in range(len(cm[0])):
                pltx.text(j, i, cm[i, j], ha='center', va='center', color='black')
        pltx.tight_layout()
```

```
        pltx.show()

    def plot_roc_curve(self, y_true, y_prob):
        fpr, tpr, _ = roc_curve(y_true, y_prob)
        auc = roc_auc_score(y_true, y_prob)
        pltx.figure()
        pltx.plot(fpr, tpr, label=f"AUC = {auc:.2f}")
        pltx.plot([0, 1], [0, 1], linestyle='--')
        pltx.xlabel('False Positive Rate')
        pltx.ylabel('True Positive Rate')
        pltx.title('ROC Curve')
        pltx.legend()
        pltx.grid()
        pltx.tight_layout()
        pltx.show()

    def plot_accuracy_loss(self):
        pltx.figure(figsize=(10, 4))
        pltx.subplot(1, 2, 1)
        pltx.plot(self.train_losses, label='Train Loss')
        pltx.plot(self.val_losses, label='Val Loss')
        pltx.legend()
        pltx.title("Loss over Epochs")
        pltx.xlabel("Epoch")
        pltx.ylabel("Loss")

        pltx.subplot(1, 2, 2)
        pltx.plot(self.train_accuracies, label='Train Accuracy')
        pltx.plot(self.val_accuracies, label='Val Accuracy')
        pltx.legend()
        pltx.title("Accuracy over Epochs")
        pltx.xlabel("Epoch")
        pltx.ylabel("Accuracy")

        pltx.tight_layout()
        pltx.show()

```

## Resnet18

```
class Resnet18(nnx.Module):
    def __init__(self, num_classes=2):
        super(Resnet18, self).__init__()
        self.model = models.resnet18(weights=None)  # offline safe
        num_ftrs = self.model.fc.in_features
        self.model.fc = nnx.Linear(num_ftrs, num_classes)

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

```

```
# Initialize the model
resnet18_model = Resnet18(num_classes=2)

# Define loss function, optimizer, and scheduler
criterion = nnx.CrossEntropyLoss()
optimizer = optimx.Adam(resnet18_model.parameters(), lr=0.001)
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3)

# Initialize and train the trainer
resnet18_trainer = BaseTrainer(model=resnet18_model,
                      train_loader=train_loader,
                      val_loader=valid_loader,
                      test_loader=test_loader,
                      criterion=criterion,
                      optimizer=optimizer,
                      scheduler=scheduler,
                      device=device,
                      early_stopping_patience=5,
                      model_name='resnet18_best_model.pt')

# Train the model for a specified number of epochs
num_epochs = 20  # You can adjust this number
resnet18_trainer.train(num_epochs)
resnet18_trainer.save_model('/kaggle/working/resnet_saved_model.pth')

resnet18_trainer.load_model()
resnet18_trainer.test()
```

Show hidden output

## DenseNet121

```
class DenseNetTrainer(BaseTrainer):
    def __init__(self, model_name='densenet121', num_classes=2, **kwargs):
        densenet_model = timm.create_model(model_name, pretrained=False, num_classes=num_classes)
        num_ftrs = densenet_model.classifier.in_features
        densenet_model.classifier = nnx.Linear(num_ftrs, num_classes)
        optimizer = optimx.Adam(densenet_model.parameters(), lr=0.001)
        criterion = nnx.CrossEntropyLoss()
        super().__init__(densenet_model, criterion=criterion, optimizer=optimizer, **kwargs)

```

```
densenet_trainer = DenseNetTrainer(
    train_loader=train_loader,
    val_loader=valid_loader,
    test_loader=test_loader,
    device=device,
    early_stopping_patience=5,

)

num_epochs = 20
densenet_trainer.train(num_epochs)
densenet_trainer.save_model('/kaggle/working/hybrid_saved_model.pth')
densenet_trainer.load_model()
densenet_trainer.test()
```

Show hidden output

## Hybrid: Resnet18 & DenseNet121

```
class HybridModel(nnx.Module):
    def __init__(self, num_classes=2):
        super(HybridModel, self).__init__()
        self.resnet = timm.create_model('resnet18', pretrained=True, num_classes=0, global_pool='')
        self.densenet = timm.create_model('densenet121', pretrained=True, num_classes=0, global_pool='')

        # Get feature sizes
        self.resnet_out = self.resnet.num_features
        self.densenet_out = self.densenet.num_features
        total_features = self.resnet_out + self.densenet_out
        self.global_pool = nnx.AdaptiveAvgPool2d(1)
        self.classifier = nnx.Sequential(
            nnx.Flatten(),
            nnx.Linear(total_features, 256),
            nnx.ReLU(),
            nnx.Dropout(0.5),
            nnx.Linear(256, num_classes)
        )

    def forward(self, x):
        r = self.global_pool(self.resnet(x))  # [B, resnet_out, 1, 1]
        d = self.global_pool(self.densenet(x))  # [B, densenet_out, 1, 1]
        concat = torch.cat([r, d], dim=1)  # [B, total_features, 1, 1]
        return self.classifier(concat)

```

```
hybrid_model = HybridModel(num_classes=2)

# Define loss function, optimizer, and scheduler for the Hybrid Model
criterion = nnx.CrossEntropyLoss()
optimizer = optimx.Adam(hybrid_model.parameters(), lr=0.001)  # Fine-tune the classifier layer
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=3)

# Initialize and train the trainer for the Hybrid Model
hybrid_trainer = BaseTrainer(model=hybrid_model,
                             train_loader=train_loader,
                             val_loader=valid_loader,
                             test_loader=test_loader,
                             criterion=criterion,
                             optimizer=optimizer,
                             scheduler=scheduler,
                             device=device,
```

```
                             early_stopping_patience=5,
                             model_name='hybrid_best_model.pt')

# Train the Hybrid model
num_epochs = 20  # You can adjust this number
hybrid_trainer.train(num_epochs)
hybrid_trainer.save_model('/kaggle/working/hybrid_saved_model.pth')

hybrid_trainer.load_model()
hybrid_trainer.test()
```

Show hidden output

```
import matplotlib.pyplot as plt

def plot_comparative_bars(trainers, labels):
    # Collect final epoch metrics
    train_accuracies = [t.train_accuracies[-1] for t in trainers]
    val_accuracies   = [t.val_accuracies[-1] for t in trainers]
    train_losses     = [t.train_losses[-1] for t in trainers]
    val_losses       = [t.val_losses[-1] for t in trainers]

    x = range(len(labels))
    bar_width = 0.6

    fig, axs = plt.subplots(2, 2, figsize=(12, 8))

    # Training Accuracy
    axs[0, 0].bar(x, train_accuracies, color='skyblue', width=bar_width)
    axs[0, 0].set_title('Final Training Accuracy')
    axs[0, 0].set_xticks(x)
    axs[0, 0].set_xticklabels(labels)
    axs[0, 0].set_ylim(0, 1)

    # Validation Accuracy
    axs[0, 1].bar(x, val_accuracies, color='lightgreen', width=bar_width)
    axs[0, 1].set_title('Final Validation Accuracy')
    axs[0, 1].set_xticks(x)
    axs[0, 1].set_xticklabels(labels)
    axs[0, 1].set_ylim(0, 1)

    # Training Loss
    axs[1, 0].bar(x, train_losses, color='salmon', width=bar_width)
    axs[1, 0].set_title('Final Training Loss')
    axs[1, 0].set_xticks(x)
    axs[1, 0].set_xticklabels(labels)

    # Validation Loss
    axs[1, 1].bar(x, val_losses, color='orange', width=bar_width)
    axs[1, 1].set_title('Final Validation Loss')
    axs[1, 1].set_xticks(x)
    axs[1, 1].set_xticklabels(labels)

    plt.tight_layout()
    plt.show()

```

```
# Call after all models have been trained
plot_comparative_bars(
    trainers=[resnet18_trainer, densenet_trainer, hybrid_trainer],
    labels=["Resnet 18", "DenseNet 121", "Hybrid"]
)

```

Show hidden output