Problem Statement 02: Surgical Smoke Detection

Team: **B1TMINDS**

Importing required Libraries:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
import os
from PIL import Image
import shutil
import random
import csv
```

Dataset preparation:

Unzipping the Dataset files.

```
! unzip /content/drive/MyDrive/PS2\ Train.zip
```

The main folder after unzipping is '**PS2 Train**'. Inside that there are several video folders.From the video folders, the samples ending with 0 and 1 have images in which no smoke and smoke are to observed respectively. In order to separate the images into twi different classes, two folders are made named '**class0**' and '**class1**'

```
# Source base directory
base_dir = "/content/PS2 Train"

# Destination directories
destination_dir_0 = "/content/data/class0"
destination_dir_1 = "/content/data/class1"

# Ensure destination directories exist
os.makedirs(destination_dir_0, exist_ok=True)
```

```
os.makedirs(destination_dir_1, exist_ok=True)

# Initialize file index
file_index = 0

# Iterate over video directories
for videodir in os.listdir(base_dir):
    video_path = os.path.join(base_dir, videodir)
    if os.path.isdir(video_path):
        # Iterate over subfolders in each video directory
        for foldername in os.listdir(video_path):
            folder_path = os.path.join(video_path,
foldername)
            if os.path.isdir(folder_path):
                # Check if folder name ends with '0' or
'1'
                if foldername.endswith('0'):
                    destination_dir = destination_dir_0
                elif foldername.endswith('1'):
                    destination_dir = destination_dir_1
                else:
                    continue  # Skip folders that don't
end with '0' or '1'

                # Iterate over files in the folder
                for filename in os.listdir(folder_path):
                    file_path = os.path.join(folder_path,
filename)
                    if os.path.isfile(file_path):
                        # Construct destination path with
incremented file index
                        destination_path =
os.path.join(destination_dir, f"{file_index}.jpeg")

                        # Copy file from source to
destination
                        shutil.copyfile(file_path,
destination_path)

                        # Increment file index
                        file_index += 1
```

```
    ]),
}
```

Dataset preprocessing:

Data augmentation

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
[0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
[0.229, 0.224, 0.225])
    ]),
}
```

The source_dir contains the path to the folder of 'data' directory in which 'class0' and 'class1' folders are present.
We are making 3 more directories inside data folder named as train, val, test.
The data is split into training set, validation set and testing set.
We have used only **10% of the data to train our model** as the dataset is quite huge and we didn't have the required time and resources.

```
# Define paths
source_dir = "/content/data"
train_dir = "/content/data_/train"
val_dir = "/content/data_/val"
test_dir = "/content/data_/test"

# Create train, val, test directories
os.makedirs(train_dir, exist_ok=True)
os.makedirs(val_dir, exist_ok=True)
os.makedirs(test_dir, exist_ok=True)
```

```
# Define the ratio for using only 10% of the data
data_ratio = 0.1

# Iterate through each class folder
for class_folder in os.listdir(source_dir):
    class_path = os.path.join(source_dir, class_folder)
    if os.path.isdir(class_path):
        # List all images in the class folder
        images = os.listdir(class_path)
        # Shuffle the images
        random.shuffle(images)
        # Calculate the number of images to use based on
the data_ratio
        num_images_to_use = int(len(images) * data_ratio)

        # Select only 10% of the images randomly
        selected_images = random.sample(images,
num_images_to_use)

        # Split the selected images into train, val, and
test sets
        train_split = int(0.7 * num_images_to_use)
        val_split = int(0.2 * num_images_to_use)

        train_images = selected_images[:train_split]
        val_images =
selected_images[train_split:train_split + val_split]
        test_images = selected_images[train_split +
val_split:]

        # Move selected images to train directory
        for image in train_images:
            src = os.path.join(class_path, image)
            dst = os.path.join(train_dir, class_folder,
image)
            os.makedirs(os.path.dirname(dst),
exist_ok=True)
            shutil.copy(src, dst)

        # Move selected images to val directory
        for image in val_images:
```

```
            src = os.path.join(class_path, image)
            dst = os.path.join(val_dir, class_folder,
image)
            os.makedirs(os.path.dirname(dst),
exist_ok=True)
            shutil.copy(src, dst)

        # Move selected images to test directory
        for image in test_images:
            src = os.path.join(class_path, image)
            dst = os.path.join(test_dir, class_folder,
image)
            os.makedirs(os.path.dirname(dst),
exist_ok=True)
            shutil.copy(src, dst)

print("Data splitting completed successfully.”)
```

Model Training and Evaluation:

Defining the data directory and creating data loaders for training and validation sets using PyTorch.

```
data_dir = '/content/data_'
```

```
image_datasets = {x:
datasets.ImageFolder(os.path.join(data_dir, x),
data_transforms[x]) for x in ['train', 'val']}
```

```
dataloaders = {x:
torch.utils.data.DataLoader(image_datasets[x],
batch_size=4, shuffle=True, num_workers=4) for x in
['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in
['train', 'val']}
print(dataset_sizes)

class_names = image_datasets['train'].classes
```

Model Initialisation -

Loading the pre-trained ResNet-18 model and freeze layers except the final classification layer.

```
model = models.resnet18(pretrained=True)

for name, param in model.named_parameters():
    if "fc" in name:
        param.requires_grad = True
    else:
        param.requires_grad = False
```

Training Setup-

Define the loss function (CrossEntropyLoss) and optimizer (SGD) for model training.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001,
momentum=0.9)

device = torch.device("cuda:0" if
torch.cuda.is_available() else "cpu")
model = model.to(device)
```

Training Loop-

```
# Define the number of epochs
num_epochs = 5

# Iterate over epochs
for epoch in range(num_epochs):
    print(f'Epoch {epoch + 1}/{num_epochs}')
    print('-' * 10)

    # Iterate over training and validation phases
    for phase in ['train', 'val']:
        if phase == 'train':
            model.train()  # Set model to training mode
```

```
        else:
            model.eval()   # Set model to evaluation mode

        running_loss = 0.0
        running_corrects = 0

        # Iterate over data.
        for i, (inputs, labels) in
enumerate(dataloaders[phase], 1):
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Zero the parameter gradients only in the
training phase
            optimizer.zero_grad()

            # Forward pass, track history if only in
training phase
            with torch.set_grad_enabled(phase ==
'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)

                # Backward + optimize only if in training
phase
                if phase == 'train':
                    loss.backward()
                    optimizer.step()

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

            # Print mini-batch statistics if in training
phase
            if phase == 'train':
                print(f'{phase} Epoch [{epoch + 1}/
{num_epochs}], '
                      f'Mini-batch [{i}/
{len(dataloaders[phase])}], '
```

```
                        f'Loss: {loss.item():.4f}, '
                        f'Acc: {(torch.sum(preds ==
labels.data).double() / labels.size(0)):.4f}')

        # Calculate epoch statistics
        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() /
dataset_sizes[phase]

        # Print epoch statistics
        print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}')

print("Training complete!")
```

Loading and preprocessing the unseen image(test set)**:**

```
# Load and preprocess the unseen image

preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229,
0.224, 0.225])
])
```

For Class0

```
class0_test_path = '/content/data_/test/class0'
images0 = os.listdir(class0_test_path)
predicted_0_labels = []
for image in images0:
  image_path = "/content/data_/test/class0/"+image
  input_tensor = preprocess(Image.open(image_path))
  input_batch = input_tensor.unsqueeze(0)
  # Perform inference
  with torch.no_grad():
      output = model(input_batch)

  # Get the predicted class
```

```
  _, predicted_class = output.max(1)

  # Map the predicted class to the class name
  class_names = ['class0', 'class1']  # Make sure these
class names match your training data
  predicted_class_name =
class_names[predicted_class.item()]

  predicted_0_labels.append(predicted_class_name)
```

For Class1

```
class1_test_path = '/content/data_/test/class1'
images1 = os.listdir(class1_test_path)
predicted_1_labels = []
for image in images1:
  image_path = "/content/data_/test/class1/"+image
  input_tensor = preprocess(Image.open(image_path))
  input_batch = input_tensor.unsqueeze(0)
  # Perform inference
  with torch.no_grad():
      output = model(input_batch)

  # Get the predicted class
  _, predicted_class = output.max(1)

  # Map the predicted class to the class name
  class_names = ['class0', 'class1']  # Make sure these
class names match your training data
  predicted_class_name =
class_names[predicted_class.item()]

  predicted_1_labels.append(predicted_class_name)
```

Dealing with the final TEST dataset:

Unzipping the provided dataset

```
!unzip /content/drive/MyDrive/PS2_test.zip
```

Preparing the dataset for evaluation.
We are iterating the 10 frames from each sample folders and feeding it to the model for classification.
Here we have defined a threshold of 5 i.e. if at least 5 frames from a sample are classified to 1 (smoke) then the whole sample is labels as 1.

```
PS2_test_path = '/content/PS2_test'
activation_threshold = 5
sample_predictions=[]
sample_names=[]
videos = os.listdir(PS2_test_path)
for video in videos:
  video_path = PS2_test_path+"/"+video
  samples = os.listdir(video_path)
  for sample in samples:
    sample_path = video_path+"/"+sample
    images = os.listdir(sample_path)
    images_prediction=[]
    for image in images:
      image_path = sample_path+"/"+image
      input_tensor = preprocess(Image.open(image_path))
      input_batch = input_tensor.unsqueeze(0)
      # Perform inference
      with torch.no_grad():
          output = model(input_batch)

      # Get the predicted class
      _, predicted_class = output.max(1)

      # Map the predicted class to the class name
      class_names = ['class0', 'class1']  # Make sure
these class names match your training data
      predicted_class_name =
class_names[predicted_class.item()]

      images_prediction.append(predicted_class_name)
    smoke_count = images_prediction.count("class1")
    if smoke_count>=activation_threshold:
```

```
      sample_prediction = 1
    else:
      sample_prediction = 0
    sample_predictions.append(sample_prediction)
    sample_name = video+"_"+sample
    sample_names.append(sample_name)
```

Storing the final results into a CSV file

```
with open("output.csv", "w", newline="") as csvfile:
  writer = csv.writer(csvfile)
  writer.writerow(["sample_name", "label"])
  for sample_name, label in zip(sample_names,
sample_predictions):
    writer.writerow([sample_name, label])
```