Name: Bola-Matanmi Samiat

Batch Code: LISUM11:30

Submission Date: 31st July 2022

The chosen simple data for this task was an animal dataset gotten from Kaggle. This dataset consists of three classes of animals namely cat, dog and pandas. Each class consists of 1000 images of various sizes.

Below consists of the steps taken to deploy the trained classifier:

Step 1:

Importing all necessary libraries needed for the training of the model.

```
In [111]: import os
          import numpy as np
          import matplotlib.pyplot as plt
          import torchvision
          import torch
          import torch.nn as nn
          import torch.nn.functional as F
          from torchvision.datasets import ImageFolder
          from torch.utils.data import Dataset, DataLoader
          from sklearn.model_selection import train_test_split
          from torchvision import transforms
          from collections import Counter
          from pathlib import Path
          from PIL import Image
          import torchmetrics
```

Step 2:

Reading the directory of the images

```
In [39]: path = []
         labels = []

         path_to_dataset = "Data/animals/"
         os.listdir(path_to_dataset)
         for file in os.listdir(path_to_dataset):
             path_dir = Path(os.path.join(path_to_dataset, file))
             for image in os.listdir(path_dir):
                 image_path = Path(os.path.join(path_dir, image))
                 path.append(image_path)
                 labels.append(image_path.parent.stem)
             #     os.remove(os.path.join(path_to_dataset, '.DS_Store'))

             print(path_dir)

         Data/animals/dogs
         Data/animals/cats
         Data/animals/panda
```

Step 3:

Splitting of the dataset into train and validation dataset using the scikit-learn train_test_split() function.

Split dataset

```
In [42]: x_train, x_test, y_train, y_test = train_test_split(path, labels, test_size=0.2, random_state=123)
```

```
In [44]: print(len(x_test))
```

```
600
```

Step 4:

Preparing data augmentation for both train and validation dataset.

Data Augumentation

```
In [50]: train_augument = transforms.Compose([
             transforms.ColorJitter(0.1, 0.1, 0.1),
             transforms.RandomRotation(30),
             transforms.RandomHorizontalFlip(1),
             transforms.RandomVerticalFlip(0.1),
             transforms.Resize(224),
             transforms.CenterCrop(224),
             transforms.ToTensor()
         ])

         test_augument = transforms.Compose([
             transforms.Resize(224),
             transforms.CenterCrop(224),
             transforms.ToTensor()
         ])
```

Step 4

Created a custom data class called Animal_dataset.

```
In [88]: class Animal_Dataset:
             def __init__(self, path, labels, augumentations):
                 self.path = path
                 self.labels = labels
                 self.augumentations = augumentations
                 self.classes = {
                     'dogs': 0,
                     'cats': 1,
                     'panda': 2
                 }

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

             def __getitem__(self, idx):

                 sample_data = Image.open(self.path[idx]).convert(mode='RGB')
                 sample_data = self.augumentations(sample_data)

                 label = self.labels[idx]
                 if label == 'dogs':
                     label = 0
                 elif label == 'cats':
                     label = 1
                 elif label == 'panda':
                     label = 2

                 return sample_data, label
```

```
In [89]: train_dataset = Animal_Dataset(x_train, y_train, train_augument)
         test_dataset = Animal_Dataset(x_test, y_test, test_augument)
```

Step5:

Loading the dataset using the torch.utils.data class.

DataLoaders

```
In [114]: train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)
          test_dataloader = DataLoader(test_dataset, batch_size=64)
```

Step 6:

Developing a custom model

Model

In [94]:
```
class CNNModel(nn.Module):

    def __init__(self, in_features=3, num_classes=3):
        self.in_features = in_features
        self.num_classes = num_classes

        super(CNNModel, self).__init__()

        self.conv_block = nn.Sequential(
            nn.Conv2d(self.in_features,32,3),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(32,64,3),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(64,128,3),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(128,256,3),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(256,256,3),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        self.linear_block = nn.Sequential(
            nn.Linear(256*5*5, 1024),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(1024, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256,64),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(64,self.num_classes)
        )

    def forward(self,x):
        x = self.conv_block(x)
        x = torch.flatten(x,1)
        x = self.linear_block(x)
        return x
```

In [105]:
```
print(CNNModel())
```

```
CNNModel(
  (conv_block): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
    (13): ReLU()
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (linear_block): Sequential(
    (0): Linear(in_features=6400, out_features=1024, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=1024, out_features=256, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.3, inplace=False)
    (6): Linear(in_features=256, out_features=64, bias=True)
```

Step 7:

Creating a training loop

Training

In [117]:
```
class Trainer:

    def __init__(self, model, loaders, config):
        self.model = model
        self.train_loader, self.val_loader = loaders
        self.config = config

        self.loss_fn = nn.CrossEntropyLoss()
        self.optim1 = torch.optim.SGD(self.model.parameters(),
                                      lr=self.config['lr'],
                                      momentum=self.config['sgd_momentum']
                                     )
        self.optim2 = torch.optim.Adam(self.model.parameters(),
                                       lr = self.config['lr']
                                       )


        self.optim = self.optim2

        if self.config['scheduler']:
            self.scheduler = torch.optim.lr_scheduler.StepLR(optimizer=self.optim,step_size=2, gamma=0.2)

        self.metrics = torchmetrics.MetricCollection([
            torchmetrics.Accuracy(average='macro', num_classes=3),
            torchmetrics.Precision(average='macro', num_classes=3),
            torchmetrics.Recall(average='macro', num_classes=3),
            torchmetrics.F1Score(average='macro', num_classes=3)
        ]).to(self.config['device'])
        self.train_metrics = self.metrics.clone(prefix='train_')
        self.val_metrics = self.metrics.clone(prefix='val_')

        self.train_logs = []
        self.val_logs = []

    def logger(self, epoch, metrics, loss, mode):
```

```
    def logger(self, epoch, metrics, loss, mode):

        metrics = {metric:metrics[metric].cpu().item() for metric in metrics}

        log = {
            f'epoch_{epoch}': {
                'loss': loss,
                'metrics': metrics
            }
        }
        if mode == 'train':
            self.train_logs.append(log)
        else:
            self.val_logs.append(log)

    def print_per_epoch(self, epoch):
        print(f"\n\n{'-'*30}EPOCH {epoch+1}/{self.config['epochs']}{'-'*30}")
        train_loss = self.train_logs[epoch][f'epoch_{epoch}']['loss']
        train_acc = self.train_logs[epoch][f'epoch_{epoch}']['metrics']['train_Accuracy']
        val_loss = self.val_logs[epoch][f'epoch_{epoch}']['loss']
        val_acc = self.val_logs[epoch][f'epoch_{epoch}']['metrics']['val_Accuracy']
        print(f"Train -> LOSS: {train_loss} | ACC: {train_acc}")
        print(f"Validation -> LOSS: {val_loss} | ACC: {val_acc}")

    def train_one_epoch(self, epoch):

        running_loss = 0

        for x,y in self.train_loader:

            self.optim.zero_grad()

            x = x.to(self.config['device'])
            y = y.to(self.config['device'])

            preds = self.model(x)

            loss = self.loss_fn(preds, y)
            running_loss += loss.item()
            loss.backward()
```

```
            self.optim.step()

            self.train_metrics(torch.argmax(preds, dim=1), y)

            del x,y,preds,loss

        if self.config['scheduler']:
            self.scheduler.step()

        metrics = self.train_metrics.compute()
        train_loss = running_loss / len(self.train_loader)

        self.logger(epoch,metrics,train_loss,'train')

        del metrics

    @torch.no_grad()
    def valid_one_epoch(self, epoch):

        running_loss = 0

        for x,y in self.val_loader:

            x = x.to(self.config['device'])
            y = y.to(self.config['device'])

            preds = self.model(x)
            loss = self.loss_fn(preds, y)
            running_loss += loss.item()

            self.val_metrics(torch.argmax(preds, dim=1), y)

            del x,y,preds,loss

        metrics = self.val_metrics.compute()
        val_loss = running_loss / len(self.val_loader)
        self.logger(epoch,metrics,val_loss,'val')

        del metrics
```

```
    def clear(self):
        gc.collect()
        torch.cuda.empty_cache()


    def fit(self):

        for epoch in range(self.config['epochs']):

            self.model.train()
            self.train_one_epoch(epoch)

#             self.clear()

            self.model.eval()
            self.valid_one_epoch(epoch)

#             self.clear()

            # reset metrics
            self.train_metrics.reset()
            self.val_metrics.reset()

            # print metrics
            self.print_per_epoch(epoch)
```

Step 8:

Training the model

Train with Scheduler

In [127]:
```
config = {
    'lr': 1e-3,
    'epochs': 30,
    'sgd_momentum': 0.8,
    'scheduler': True,
    'sch_step_size': 2,
    'sch_gamma': 0.2,
    'device': 'cuda' if torch.cuda.is_available() else 'cpu'
}
print(config['device'])
model = CNNModel().to(device=config['device'])
trainer = Trainer(model, (train_dataloader, test_dataloader), config)
```

```
cpu
```

In [128]:
```
trainer.fit()
```

```
--------------------------------EPOCH 27/30--------------------------------
Train -> LOSS: 0.8173727565690091 | ACC: 0.5752115249633789
Validation -> LOSS: 0.7553458333015441 | ACC: 0.5788576602935791


--------------------------------EPOCH 28/30--------------------------------
Train -> LOSS: 0.8059799671173096 | ACC: 0.5715314149856567
Validation -> LOSS: 0.7553458333015441 | ACC: 0.5788576602935791


--------------------------------EPOCH 29/30--------------------------------
Train -> LOSS: 0.8099471158102939 | ACC: 0.5831961631774902
Validation -> LOSS: 0.7553458333015441 | ACC: 0.5788576602935791


--------------------------------EPOCH 30/30--------------------------------
Train -> LOSS: 0.8133960005484129 | ACC: 0.5848942399024963
Validation -> LOSS: 0.7553458333015441 | ACC: 0.5788576602935791
```

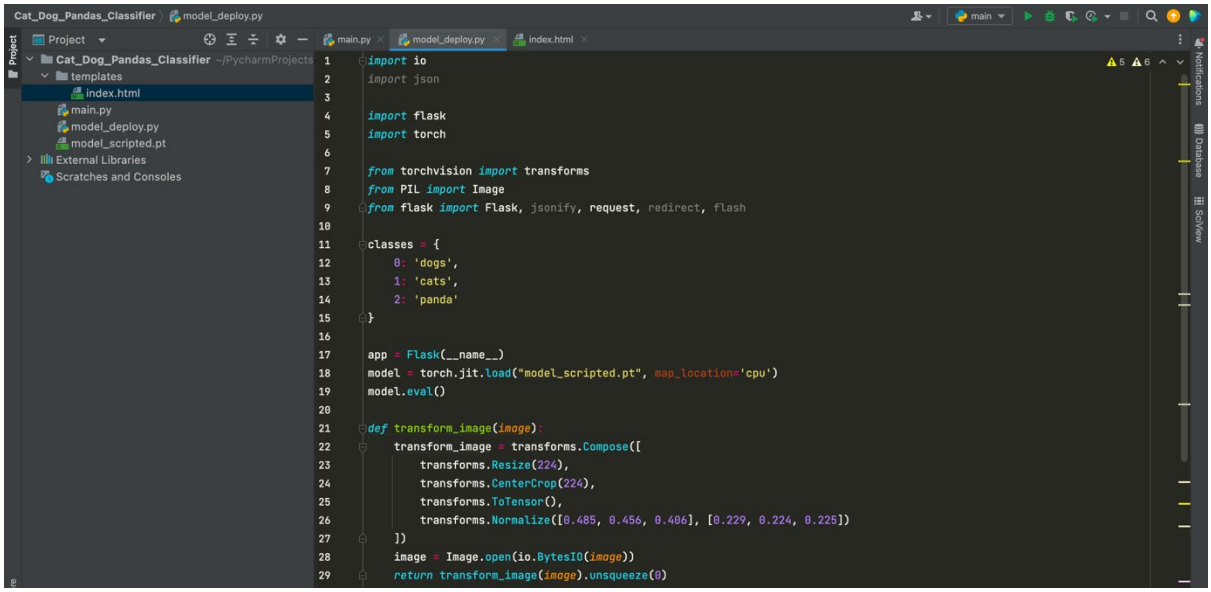
Step 9:

Saving the model

In [120]:
```
model_scripted = torch.jit.script(model)
model_scripted.save('model_scripted.pt')
```

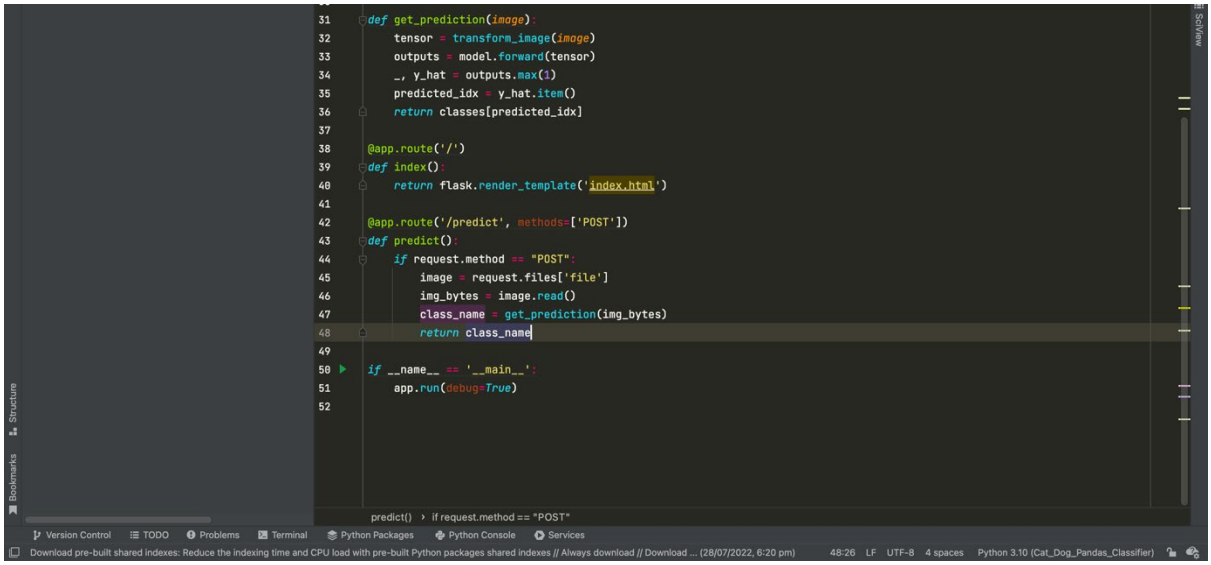
Step 10:

Loading the saved model using pytorch, applying transform to the incoming image for inference.



Step 11:

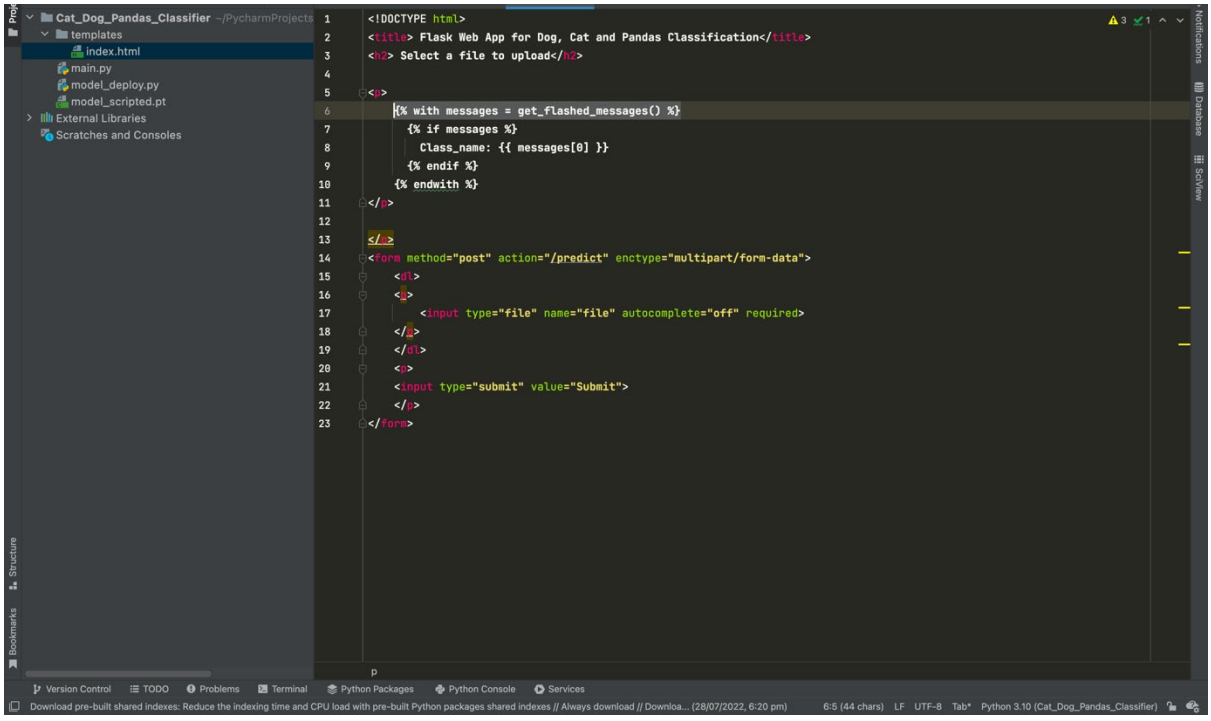
Creating the index and the predict api route in order to create a web app for the classification problem.



```
def get_prediction(image):
    tensor = transform_image(image)
    outputs = model.forward(tensor)
    _, y_hat = outputs.max(1)
    predicted_idx = y_hat.item()
    return classes[predicted_idx]

@app.route('/')
def index():
    return flask.render_template('index.html')

@app.route('/predict', methods=['POST'])
def predict():
    if request.method == "POST":
        image = request.files['file']
        img_bytes = image.read()
        class_name = get_prediction(img_bytes)
        return class_name

if __name__ == '__main__':
    app.run(debug=True)
```

Step 12:

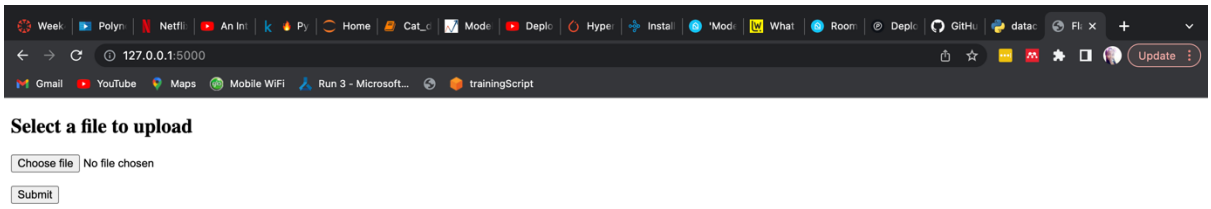
Building a template for the uploading of images for classification, This template is called “index.html”.



```
<!DOCTYPE html>
<title> Flask Web App for Dog, Cat and Pandas Classification</title>
<h2> Select a file to upload</h2>

<p>
    {% with messages = get_flashed_messages() %}
      {% if messages %}
        Class_name: {{ messages[0] }}
      {% endif %}
    {% endwith %}
</p>

</ >
<form method="post" action="/predict" enctype="multipart/form-data">
    <dl>
    <p>
        <input type="file" name="file" autocomplete="off" required>
    </p>
    </dl>
    <p>
    <input type="submit" value="Submit">
    </p>
</form>
```

Step 13:

Testing of the model deployment



Output