Sample Project: Using pyTorch to classify flowers by species.

Similar to many machine learning project, my approach includes:

- image processing
- Training classifier
- Deployment of the classifier.

I didn't train my own model, it will take too long, instead I used transform learning by integrate pretrained models. I'll be using this dataset (http://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html) of 102 flower categories.

In [ ]:
```
import os
import json
import numpy as np
import pandas as pd
import torch # using pytorch
from torch import optim
from torchsummary import summary # borrow from https://github.com/sksq96/pytorch-summary to show the model co
nfiguration
from timeit import default_timer as timer

## Computer Vision functions
from torchvision import transforms, datasets
from torchvision import models
from PIL import Image

## Neural network functions
import torch.nn as nn
import torch.nn.functional as F

## data loading functions
from torch.utils.data import DataLoader
from torch.utils.data.sampler import SubsetRandomSampler

## plotting function
import matplotlib.pyplot as plt
%matplotlib inline

## to display the full results instead of the last one.
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
import sys
```

Load the data

In [ ]:
```
batch_size = 32
IMAGE_SIZE = 224
USE_GPU = torch.cuda.is_available()
MODEL_SAVE_FILE = 'XI_Model.pth'

data_dir = './assets/flower_data'
train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
```

In [ ]:

```
# Transforming for the training and validation sets
# Images need to be processed to the right size, and normalized with the mean
# of the whole data size.
training_transforms = transforms.Compose([
        transforms.RandomRotation(45),
        transforms.RandomResizedCrop(IMAGE_SIZE), # adding randomness
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])

validation_transforms = transforms.Compose([
        transforms.Resize(IMAGE_SIZE + 32),
        transforms.CenterCrop(IMAGE_SIZE),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ])

# Load the datasets with ImageFolder function with transform furnction passed
# in
train_dataset = datasets.ImageFolder(train_dir, transform =training_transforms)
valid_dataset = datasets.ImageFolder(valid_dir, transform =validation_transforms)

# Define the dataloaders, which is similar to "generator"
train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size =batch_size, shuffle =True)
valid_dataloader = torch.utils.data.DataLoader(valid_dataset, batch_size =batch_size)

dataloaders = {'train': train_dataloader,
               'valid': valid_dataloader}

dataset_sizes = {'train': len(train_dataset),
                 'valid': len(valid_dataset)}
```

In [ ]:

```
# Loading labels

with open('cat_to_name.json', 'r') as f:
    cat_to_name = json.load(f)

list(cat_to_name.items())[:5]
```

Building and training the classifier

I will use DenseNet model, which I feel should give me a great result.

In [ ]:
```
DenseNet = models.densenet161(pretrained=True)
#print(DenseNet)
# freeze all pretrained model parameters
for param in DenseNet.parameters():
    param.requires_grad_(False)
```

In [ ]:
```
# My classifier:
# modify the last layer by adding two fully connected layer and an log softmax
# layer as output.

from collections import OrderedDict
classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(2208, 1200)),
                          ('relu', nn.ReLU()),
                              ('fc2', nn.Linear(1200, 102)),
                          ('output', nn.LogSoftmax(dim=1))
                          ]))
DenseNet.classifier = classifier
```

In [ ]:
```
# Specify Loss Function and Optimizer
# Criteria NLLLoss which is recommended with Softmax final layer
criterion = nn.NLLLoss()

# Set learning rate and decay learning rate by a factor of 0.1 every 4 epoches
optimizer = optim.Adam(DenseNet.classifier.parameters(), lr=0.001)

from torch.optim import lr_scheduler
scheduler = lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.1)
```

Install needed package for Google colab

In [ ]:
```
!pip install requests
!pip install airtable
```

In [ ]:
```
def train_model(model, criterion, optimizer, scheduler, n_epoch=60):
    ''' train_model
    Model training function.

    Argument:
        model: model archetecture
        criterion: loss function
        optimizer: optimization function
        scheduler: schedular
        n_epoch: number of epoches to train on
    Return: trained model
    '''

    # if torch.cuda.is_available():
    model = model.cuda()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    start_time = timer()

    for epoch in range(n_epoch):
        print('Epoch {}/{}'.format(epoch, n_epoch-1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'valid']:
            if phase == 'train':
                scheduler.step()
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.cuda()
                labels = labels.cuda()

                # zero the parameter gradients
```

```
                optimizer.zero_grad()

                # forward
                # track history if only in train
                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)
            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

            # deep copy the model
            if phase == 'valid' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()
    time_elapsed = timer() - start_time
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

In [ ]:

```
model_ft = train_model(DenseNet, criterion=criterion, optimizer=optimizer, scheduler=scheduler, n_epoch=60)
```

Save the checkpoint

In [ ]:
```
def save_checkpoint(model, save_path, save_cpu=False):
    ''' Same checkpoint to local
    Argument:
        model: model to save
        save_path: local path for save model
        save_cpu: whether to conver model to CPU.
    Return: None
    '''
    if save_cpu:
        model = model.cpu()

    checkpoint = {
        'arch': 'DenseNet',
        'state_dict': model.state_dict()
    }

    torch.save(checkpoint, save_path)
```

In [ ]:
```
save_checkpoint(model_ft, "./DenseNet.pth", save_cpu=True)
```

Loading the checkpoint

At this point it's good to write a function that can load a checkpoint and rebuild the model. That way you can come back to this project and keep working on it without having to retrain the network.

In [ ]:
```
def load_model(checkpoint_path):
    ''' Load trained model from checkpoint
    Argument: checkpoint path
    Return: model
    '''
    from collections import OrderedDict
    chpt = torch.load(checkpoint_path)
    model = models.densenet161(pretrained=True)
    for param in model.parameters():
        param.requires_grad = False

#     model.class_to_idx = chpt['class_to_idx']

    # Create the classifier
    classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(2208, 1200)),
                          ('relu', nn.ReLU()),
                              ('fc2', nn.Linear(1200, 102)),
                          ('output', nn.LogSoftmax(dim=1))
                          ]))
    # Put the classifier on the pretrained network
    model.classifier = classifier

    model.load_state_dict(chpt['state_dict'], strict=False)

    return model
```

In [ ]:
```
load_model("DenseNet1.pth")
```

Deployment for prediction.

In [ ]:
```
def process_image(image):
    ''' Process Image function for prediction
    Argument: an input image
    Return: a processed image
    '''
    image = Image.open(image)

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

    image = image_transforms(image)
    image = np.array(image)
    return image
```

To check your work, the function below converts a PyTorch tensor and displays it in the notebook. If your `process_image` function works, running the output through this function should return the original image (except for the cropped out portions).

In [ ]:
```
def imshow(image, ax=None, title=None):
    """Imshow for Tensor."""
    if ax is None:
        fig, ax = plt.subplots()

    # PyTorch tensors assume the color channel is the first dimension
    # but matplotlib assumes is the third dimension
    image = image.numpy().transpose((1, 2, 0))

    # Undo preprocessing
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    image = std * image + mean

    # Image needs to be clipped between 0 and 1 or it looks like noise when displayed
    image = np.clip(image, 0, 1)

    ax.imshow(image)

    return ax
```

In [ ]:
```
def predict(image_path, model, topk=5):
    ''' Predict the class  of an image using a trained model.

    Argument:
        image_path: image location for classification
        model: trained model.
        topk: show top k classes with highest predicted prob.
    Return:
        probs: probabilities,
        classes: predicted classes
    '''
    model.eval()

    probs = torch.exp(model(img))

    return print(probs, classes)
```