Developing an AI application

RIHAD VARIAWA

07-12-2018

Going forward, AI algorithms will be incorporated into more and more everyday applications. For example, you might want to include an image classifier in a smart phone app. To do this, you'd use a deep learning model trained on hundreds of thousands of images as part of the overall application architecture. A large part of software development in the future will be using these types of models as common parts of applications.

In this project, I'll train an image classifier to recognize different species of flowers. You can imagine using something like this in a phone app that tells you the name of the flower your camera is looking at. In practice you'd train this classifier, then export it for use in your application. I'll be using this dataset of 102 flower categories, you can see a few examples below.

The project is broken down into multiple steps:

- Load and preprocess the image dataset
- Train the image classifier on your dataset
- Use the trained classifier to predict image content

I'll lead you through each part which I'll implement in Python.

When you've completed this project, you'll have an application that can be trained on any set of labeled images. Here your network will be learning about flowers and end up as a command line application. But, what you do with your new skills depends on your imagination and effort in building a dataset. For example, imagine an app where you take a picture of a car, it tells you what the make and model is, then looks up information about it. Go build your own dataset and make something new.

First up is importing the packages you'll need. It's good practice to keep all the imports at the beginning of your code. As you work through this notebook and find you need to import a package, make sure to add the import up here.

In [1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import matplotlib.pyplot as plt

import torch
import numpy as np
from torch import nn
from torch import optim
import torch.nn.functional as F
from torchvision import datasets, transforms, models
from workspace_utils import active_session
import time
from collections import OrderedDict # use dict, but we have to keep the order
from PIL import Image
import json
```

Load the data

Here you'll use `torchvision` to load the data (documentation). The data should be included alongside this notebook, otherwise you can download it here. The dataset is split into three parts, training, validation, and testing. For the training, you'll want to apply transformations such as random scaling, cropping, and flipping. This will help the network generalize leading to better performance. You'll also need to make sure the input data is resized to 224x224 pixels as required by the pre-trained networks.

The validation and testing sets are used to measure the model's performance on data it hasn't seen yet. For this you don't want any scaling or rotation transformations, but you'll need to resize then crop the images to the appropriate size.

The pre-trained networks you'll use were trained on the ImageNet dataset where each color channel was normalized separately. For all three sets you'll need to normalize the means and standard deviations of the images to what the network expects. For the means, it's `[0.485, 0.456, 0.406]` and for the standard deviations `[0.229, 0.224, 0.225]`, calculated from the ImageNet images. These values will shift each color channel to be centered at 0 and range from -1 to 1.

In [2]:
```
data_dir = 'flowers'
train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'
```

In [7]:
```
# Define transforms for the training, validation, and testing sets, using data augumentations on training set,
# Inception_v3 has input size 299x299
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(299),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

validation_transforms = transforms.Compose([transforms.Resize(299),
                                            transforms.CenterCrop(299),
                                            transforms.ToTensor(),
                                            transforms.Normalize([0.485, 0.456, 0.406],
                                                                 [0.229, 0.224, 0.225])])


test_transforms = transforms.Compose([transforms.Resize(299),
                                      transforms.CenterCrop(299),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

# Load the datasets with ImageFolder
train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
validation_data = datasets.ImageFolder(valid_dir, transform=validation_transforms)
test_data = datasets.ImageFolder(test_dir, transform=test_transforms)

# Using the image datasets and the trainforms, define the dataloaders
trainloader = torch.utils.data.DataLoader(train_data, batch_size=64, shuffle=True)
validloader = torch.utils.data.DataLoader(validation_data, batch_size= 32)
testloader = torch.utils.data.DataLoader(test_data, batch_size= 32)
```

Label mapping

You'll also need to load in a mapping from category label to category name. You can find this in the file `cat_to_name.json`. It's a JSON object which you can read in with the `json` module. This will give you a dictionary mapping the integer encoded categories to the actual names of the flowers.

In [15]:
```
with open('cat_to_name.json', 'r') as f:
    cat_to_name = json.load(f)
```

Building and training the classifier

Now that the data is ready, it's time to build and train the classifier. As usual, you should use one of the pretrained models from `torchvision.models` to get the image features. Build and train a new feed-forward classifier using those features.

We're going to leave this part up to you. If you want to talk through it with someone, chat with your fellow students! You can also ask questions on the forums or join the instructors in office hours.

Refer to the rubric for guidance on successfully completing this section. Things you'll need to do:

- Load a pre-trained network (If you need a starting point, the VGG networks work great and are straightforward to use)
- Define a new, untrained feed-forward network as a classifier, using ReLU activations and dropout
- Train the classifier layers using backpropagation using the pre-trained network to get the features
- Track the loss and accuracy on the validation set to determine the best hyperparameters

We've left a cell open for you below, but use as many as you need. Our advice is to break the problem up into smaller parts you can run separately. Check that each part is doing what you expect, then move on to the next. You'll likely find that as you work through each part, you'll need to go back and modify your previous code. This is totally normal!

When training make sure you're updating only the weights of the feed-forward network. You should be able to get the validation accuracy above 70% if you build everything right. Make sure to try different hyperparameters (learning rate, units in the classifier, epochs, etc) to find the best model. Save those hyperparameters to use as default values in the next part of the project.

In [5]:
```
# Build Inception network
inception = models.inception_v3(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/inception_v3_google-1a9a5a14.pth" to /root/.torch/models/inception_v3_google-1a9a5a14.pth
100%|██████████| 108857766/108857766 [00:02<00:00, 41898537.19it/s]
```

The last layer of the inception network, (fc): Linear(in_features=2048, out_features=1000, bias=True), therefore the inputs of the feedforward network is 2048

In [6]:
```
# Freeze parameters so we don't backprop through them
for param in inception.parameters():
    param.requires_grad = False


classifier = nn.Sequential(OrderedDict([
                                         ('fc1', nn.Linear(2048, 500)),
                                         ('relu1', nn.ReLU()),
                                         ('dropout1', nn.Dropout(0.1)),
                                         ('fc2', nn.Linear(500, 102)),
                                         ('output', nn.LogSoftmax(dim=1))
                                        ]))
# Attach the feedforward neural network
inception.fc = classifier
```

Define criterion and loss

In [7]:
```
criterion = nn.NLLLoss()
# Only train the classifier parameters, feature parameters are frozen
adam = optim.Adam(inception.fc.parameters(), lr=0.001)
# Important: Send model to use gpu cuda
inception = inception.to('cuda')
```

Build function to calculate the loss and accuracy of the validation set on a single batch

In [8]:
```
def evaluate_performance_batch(model,batch, criterion, device = 'cuda'):
    with torch.no_grad():
        images, labels = tuple(map(lambda x: x.to(device), batch))
        predictions = model.forward(images)
        _, predict = torch.max(predictions, 1)

        correct = (predict == labels).sum().item()
        total  = len(labels)

    return correct, total
```

Build function to calculate the loss and accuracy of the validation set

In [9]:
```
def evaluate_performance(model, dataloader,criterion, device = 'cuda'):
    performance = [evaluate_performance_batch(model, i, criterion) for i in iter(dataloader)]
    correct, total = list(map(sum, zip(*performance)))
    return correct/total
```

Build function to tranin the neural network

In [10]:
```
def train_model(model, trainloader, validloader, epochs, print_every, criterion, optimizer, device = 'cuda'):

    for e in range(epochs):
        model.train()
        running_loss = 0 # the loss for every batch

        for i, train_batch in enumerate(trainloader): # minibatch training

            # send the inputs labels to the tensors that uses the specified devices
            inputs, labels = tuple(map(lambda x: x.to(device), train_batch))
            optimizer.zero_grad() # clear out previous gradients, avoids accumulations

            # Forward and backward passes
            predictions = model.forward(inputs)

            loss = criterion(predictions[0], labels)
            loss.backward()
            optimizer.step()

            # calculate the total loss for 1 epoch of training
            running_loss += loss.item()

            # print the loss every .. batches
            if i % print_every == 0:
                model.eval() # set to evaluation mode
                train_accuracy = evaluate_performance(model, trainloader, criterion)
                validate_accuracy = evaluate_performance(model, validloader, criterion)
                print("Epoch: {}/{}... :".format(e+1, epochs),
                      "Loss: {:.4f},".format(running_loss/print_every),
                      "Training Accuracy:{: .4f} %,".format(train_accuracy * 100),
                      "Validation Accuracy:{: .4f} %".format(validate_accuracy * 100)
                     )
                running_loss = 0
                model.train()
```

In [11]:
```
start = time.time()
with active_session():
    train_model(inception, trainloader, validloader, 15, 50, criterion, adam, device = 'cuda')
end = time.time()
print('Training time lapsed:', end - start, 's')
```

```
Epoch: 1/15... : Loss: 0.0925, Training Accuracy: 3.1288 %, Validation Accuracy: 3.4230 %
Epoch: 1/15... : Loss: 4.2734, Training Accuracy: 24.8474 %, Validation Accuracy: 26.1614 %
Epoch: 1/15... : Loss: 3.3331, Training Accuracy: 43.9713 %, Validation Accuracy: 45.4768 %
Epoch: 2/15... : Loss: 0.0521, Training Accuracy: 45.5281 %, Validation Accuracy: 47.3105 %
Epoch: 2/15... : Loss: 2.5228, Training Accuracy: 56.0134 %, Validation Accuracy: 59.0465 %
Epoch: 2/15... : Loss: 2.0230, Training Accuracy: 61.3248 %, Validation Accuracy: 63.5697 %
Epoch: 3/15... : Loss: 0.0327, Training Accuracy: 62.9731 %, Validation Accuracy: 62.7139 %
Epoch: 3/15... : Loss: 1.7305, Training Accuracy: 65.9799 %, Validation Accuracy: 68.2152 %
Epoch: 3/15... : Loss: 1.6081, Training Accuracy: 70.3602 %, Validation Accuracy: 72.2494 %
Epoch: 4/15... : Loss: 0.0246, Training Accuracy: 70.5433 %, Validation Accuracy: 72.2494 %
Epoch: 4/15... : Loss: 1.4438, Training Accuracy: 71.2912 %, Validation Accuracy: 73.3496 %
Epoch: 4/15... : Loss: 1.4034, Training Accuracy: 74.7711 %, Validation Accuracy: 76.6504 %
Epoch: 5/15... : Loss: 0.0321, Training Accuracy: 75.8852 %, Validation Accuracy: 77.8729 %
Epoch: 5/15... : Loss: 1.2899, Training Accuracy: 76.8620 %, Validation Accuracy: 80.0733 %
Epoch: 5/15... : Loss: 1.2660, Training Accuracy: 77.1215 %, Validation Accuracy: 79.3399 %
Epoch: 6/15... : Loss: 0.0245, Training Accuracy: 76.1294 %, Validation Accuracy: 77.3839 %
Epoch: 6/15... : Loss: 1.2428, Training Accuracy: 78.9988 %, Validation Accuracy: 80.5623 %
Epoch: 6/15... : Loss: 1.1950, Training Accuracy: 79.7924 %, Validation Accuracy: 82.5183 %
Epoch: 7/15... : Loss: 0.0266, Training Accuracy: 80.0519 %, Validation Accuracy: 82.2738 %
Epoch: 7/15... : Loss: 1.1748, Training Accuracy: 80.4335 %, Validation Accuracy: 82.2738 %
Epoch: 7/15... : Loss: 1.1639, Training Accuracy: 80.1129 %, Validation Accuracy: 84.1076 %
Epoch: 8/15... : Loss: 0.0216, Training Accuracy: 80.1129 %, Validation Accuracy: 82.5183 %
Epoch: 8/15... : Loss: 1.1408, Training Accuracy: 81.3645 %, Validation Accuracy: 82.8851 %
Epoch: 8/15... : Loss: 1.0852, Training Accuracy: 81.2882 %, Validation Accuracy: 81.4181 %
Epoch: 9/15... : Loss: 0.0217, Training Accuracy: 81.5934 %, Validation Accuracy: 83.6186 %
Epoch: 9/15... : Loss: 1.0224, Training Accuracy: 81.5476 %, Validation Accuracy: 84.1076 %
Epoch: 9/15... : Loss: 1.0719, Training Accuracy: 81.7766 %, Validation Accuracy: 83.9853 %
Epoch: 10/15... : Loss: 0.0149, Training Accuracy: 82.0360 %, Validation Accuracy: 84.9633 %
Epoch: 10/15... : Loss: 0.9796, Training Accuracy: 81.9597 %, Validation Accuracy: 83.1296 %
Epoch: 10/15... : Loss: 1.0293, Training Accuracy: 83.3486 %, Validation Accuracy: 84.8411 %
Epoch: 11/15... : Loss: 0.0188, Training Accuracy: 84.0965 %, Validation Accuracy: 85.6968 %
Epoch: 11/15... : Loss: 1.0163, Training Accuracy: 83.0128 %, Validation Accuracy: 84.4743 %
Epoch: 11/15... : Loss: 1.0363, Training Accuracy: 82.7686 %, Validation Accuracy: 85.2078 %
Epoch: 12/15... : Loss: 0.0260, Training Accuracy: 83.7607 %, Validation Accuracy: 85.0856 %
Epoch: 12/15... : Loss: 0.9740, Training Accuracy: 83.6386 %, Validation Accuracy: 85.9413 %
Epoch: 12/15... : Loss: 1.0162, Training Accuracy: 83.7302 %, Validation Accuracy: 83.8631 %
Epoch: 13/15... : Loss: 0.0206, Training Accuracy: 84.1728 %, Validation Accuracy: 83.9853 %
Epoch: 13/15... : Loss: 0.9534, Training Accuracy: 84.6459 %, Validation Accuracy: 85.8191 %
Epoch: 13/15... : Loss: 0.9911, Training Accuracy: 84.0812 %, Validation Accuracy: 86.9193 %
Epoch: 14/15... : Loss: 0.0206, Training Accuracy: 84.0507 %, Validation Accuracy: 86.4303 %
Epoch: 14/15... : Loss: 0.9785, Training Accuracy: 85.2106 %, Validation Accuracy: 85.2078 %
Epoch: 14/15... : Loss: 0.9902, Training Accuracy: 84.4170 %, Validation Accuracy: 85.4523 %
Epoch: 15/15... : Loss: 0.0192, Training Accuracy: 83.9286 %, Validation Accuracy: 84.8411 %
Epoch: 15/15... : Loss: 0.9125, Training Accuracy: 85.3022 %, Validation Accuracy: 86.4303 %
Epoch: 15/15... : Loss: 0.9841, Training Accuracy: 85.1496 %, Validation Accuracy: 85.2078 %
Training time lapsed: 10200.958985805511 s
```

Testing your network

It's good practice to test your trained network on test data, images the network has never seen either in training or validation. This will give you a good estimate for the model's performance on completely new images. Run the test images through the network and measure the accuracy, the same way you did validation. You should be able to reach around 70% accuracy on the test set if the model has been trained well.

In [12]:
```
# TODO: Do validation on the test set
inception.eval()
test_accuracy = evaluate_performance(inception, testloader, criterion)
print ("Test Accuracy:{:.4f} %,".format(test_accuracy * 100) )
```

```
Test Accuracy:83.7607 %,
```

Save the checkpoint

Now that your network is trained, save the model so you can load it later for making predictions. You probably want to save other things such as the mapping of classes to indices which you get from one of the image datasets: `image_datasets['train'].class_to_idx`. You can attach this to the model as an attribute which makes inference easier later on.

`model.class_to_idx = image_datasets['train'].class_to_idx`

Remember that you'll want to completely rebuild the model later so you can use it for inference. Make sure to include any information you need in the checkpoint. If you want to load the model and keep training, you'll want to save the number of epochs as well as the optimizer state, `optimizer.state_dict`. You'll likely want to use this trained model in the next part of the project, so best to save it now.

In [13]:
```
# Save the checkpoint
with active_session():
    check_point_file = 'inception_checkpoint.pth'
    inception.class_to_idx =  train_data.class_to_idx

    checkpoint_dict = {
        'architecture': 'inception_v3',
        'class_to_idx': inception.class_to_idx,
        'input_units': 2048,
        'hidden_units': 500,
        'dropout_prob': 0.1,
        'state_dict': inception.state_dict()
    }
    torch.save(checkpoint_dict, check_point_file)
```

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 [2]:
```
# Write a function that loads a checkpoint and rebuilds the model
def load_model_checkpoint(path, device = 'cuda'):
    checkpoint = torch.load(path, map_location={'cuda:0': 'cpu'})
    model = models.inception_v3(pretrained=True) # static, will change in application
    classifier = nn.Sequential(OrderedDict([
                                         ('fc1', nn.Linear(checkpoint['input_units'], checkpoint['hidden_units'])),
                                         ('relu1', nn.ReLU()),
                                         ('dropout1', nn.Dropout(checkpoint['dropout_prob'])),
                                         ('fc3', nn.Linear(checkpoint['hidden_units'], 102)),
                                         ('output', nn.LogSoftmax(dim=1))
                                        ]))
    # Attach the feedforward neural network
    model.fc = classifier
    model.load_state_dict(checkpoint['state_dict'])
    model.class_to_idx = checkpoint['class_to_idx']
    model.to(device)
    return model
```

Load model

In [3]:
```
model = load_model_checkpoint('inception_v3_checkpoint.pth')
```

Inference for classification

Now you'll write a function to use a trained network for inference. That is, you'll pass an image into the network and predict the class of the flower in the image. Write a function called `predict` that takes an image and a model, then returns the top K most likely classes along with the probabilities. It should look like

```
probs, classes = predict(image_path, model)
print(probs)
print(classes)
> [ 0.01558163  0.01541934  0.01452626  0.01443549  0.01407339]
> ['70', '3', '45', '62', '55']
```

First you'll need to handle processing the input image such that it can be used in your network.

Image Preprocessing

You'll want to use `PIL` to load the image (documentation). It's best to write a function that preprocesses the image so it can be used as input for the model. This function should process the images in the same manner used for training.

First, resize the images where the shortest side is 256 pixels, keeping the aspect ratio. This can be done with the `thumbnail` or `resize` methods. Then you'll need to crop out the center 224x224 portion of the image.

Color channels of images are typically encoded as integers 0-255, but the model expected floats 0-1. You'll need to convert the values. It's easiest with a Numpy array, which you can get from a PIL image like so `np_image = np.array(pil_image)`.

As before, the network expects the images to be normalized in a specific way. For the means, it's `[0.485, 0.456, 0.406]` and for the standard deviations `[0.229, 0.224, 0.225]`. You'll want to subtract the means from each color channel, then divide by the standard deviation.

And finally, PyTorch expects the color channel to be the first dimension but it's the third dimension in the PIL image and Numpy array. You can reorder dimensions using `ndarray.transpose`. The color channel needs to be first and retain the order of the other two dimensions.

In [4]:
```
def process_image(image):
    ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
        returns an Numpy array
    '''
    # Process a PIL image for use in a PyTorch model
    im = Image.open(image)

    # resize image to 320 on the shortest side
    size = (320, 320)
    im.thumbnail(size)

    # crop out 299 portion in the center
    width, height = im.size
    left = (width - 299)/2
    top = (height - 299)/2
    right = (width + 299)/2
    bottom = (height + 299)/2
    im = im.crop((left, top, right, bottom))

    # normalize image
    np_image = np.array(im)
    im_mean = np.array([0.485, 0.456, 0.406])
    im_sd = np.array([0.229, 0.224, 0.225])
    np_image = (np_image/255 - im_mean)/im_sd

    # transpose the image
    np_image = np_image.T
    return np_image
```

In [5]:
```
image_path = 'flowers/valid/1/image_06739.jpg'
processed_image = process_image(image_path)
```

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 [6]:
```
def imshow2(image, ax=None, title=None):
    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.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)

    #plt.suptitle(title)
    ax.imshow(image)

    return ax
```

In [16]:
```
imshow2(processed_image, title= cat_to_name['1'])
```

Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x7f19f851f278>

Class Prediction

Once you can get images in the correct format, it's time to write a function for making predictions with your model. A common practice is to predict the top 5 or so (usually called top-K) most probable classes. You'll want to calculate the class probabilities then find the K largest values.

To get the top K largest values in a tensor use `x.topk(k)`. This method returns both the highest `k` probabilities and the indices of those probabilities corresponding to the classes. You need to convert from these indices to the actual class labels using `class_to_idx` which hopefully you added to the model or from an `ImageFolder` you used to load the data (see here). Make sure to invert the dictionary so you get a mapping from index to class as well.

Again, this method should take a path to an image and a model checkpoint, then return the probabilities and classes.

```
probs, classes = predict(image_path, model)
print(probs)
print(classes)
> [ 0.01558163  0.01541934  0.01452626  0.01443549  0.01407339]
> ['70', '3', '45', '62', '55']
```

In [13]:
```
def predict(image_path, model, device = 'cuda', topk=5):
    ''' Predict the class (or classes) of an image using a trained deep learning model.
    '''
    # Implement the code to predict the class from an image file
    model.eval()
```