Developing an AI application

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, you'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. We'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

We'll lead you through each part which you'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.

```
# Package Imports: All the necessary packages and modules are imported
in the first cell of the notebook
import argparse
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, models
import json
from PIL import Image
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

```
import numpy as np
import os
```

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.

```
data_dir = '/kaggle/input/102-flower-categories'
train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

# Training data : torchvision transforms are used to augment the
training data with random scaling, rotations, mirroring, and/or
cropping
# TODO: Define your transforms for the training, validation, and
testing sets
# Define data augmentation and normalization transformations
train_transforms = transforms.Compose([
    transforms.RandomResizedCrop(224),  # Random cropping and resizing
    transforms.RandomHorizontalFlip(),  # Random horizontal flip
    transforms.RandomRotation(30),       # Random rotation (degrees)
    transforms.ColorJitter(brightness=0.2, contrast=0.2,
saturation=0.2, hue=0.2),  # Color jitter
    transforms.ToTensor(),               # Convert to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])   # Image normalization
])

# For validation and testing, you typically don't perform data
augmentation
valid_test_transforms = transforms.Compose([
    transforms.Resize(256),         # Resize to 256x256
```

```
    transforms.CenterCrop(224),     # Center crop to 224x224
    transforms.ToTensor(),          # Convert to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])  # Image normalization
])

# Data loading: The data for each set (train, validation, test) is
loaded with torchvision's ImageFolder
# TODO: Load the datasets with ImageFolder
image_datasets = {
    'train' : datasets.ImageFolder(train_dir,
transform=train_transforms),
    'valid' : datasets.ImageFolder(valid_dir,
transform=valid_test_transforms),
    'test' : datasets.ImageFolder(test_dir,
transform=valid_test_transforms)
}

# Data batching: The data for each set is loaded with torchvision's
DataLoader
# TODO: Using the image datasets and the trainforms, define the
dataloaders
dataloaders = {
    'train': DataLoader(image_datasets['train'], batch_size=32,
shuffle=True),
    'valid': DataLoader(image_datasets['valid'], batch_size=32),
    'test': DataLoader(image_datasets['test'], 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.

```
with open('/kaggle/input/integer-encoded-categories/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. 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.

One last important tip if you're using the workspace to run your code: To avoid having your workspace disconnect during the long-running tasks in this notebook, please read in the earlier page in this lesson called Intro to GPU Workspaces about Keeping Your Session Active. You'll want to include code from the workspace_utils.py module.

Note for Workspace users: If your network is over 1 GB when saved as a checkpoint, there might be issues with saving backups in your workspace. Typically this happens with wide dense layers after the convolutional layers. If your saved checkpoint is larger than 1 GB (you can open a terminal and check with `ls -lh`), you should reduce the size of your hidden layers and train again.

```
# Implement a function for the validation
def validation(model, testloader, criterion,device):
    test_loss = 0
    accuracy = 0

    for inputs, labels in testloader:

        inputs, labels = inputs.to(device), labels.to(device)

        output = model.forward(inputs)
        test_loss += criterion(output, labels).item()

        ps = torch.exp(output)
        equality = (labels.data == ps.max(dim=1)[1])
        accuracy += equality.type(torch.FloatTensor).mean()

    return test_loss, accuracy
```

```
# TODO: Build and train your network
# Pretrained Network: A pretrained network such as VGG16 is loaded
from torchvision.models and the parameters are frozen
# Load a pre-trained network (VGG in this case)
model = models.vgg16(weights='DEFAULT')

# Freeze parameters so we don't backpropagate through them
for param in model.parameters():
    param.requires_grad = False

# Feedforward Classifier: A new feedforward network is defined for use
as a classifier using the features as input
# Define a new feed-forward classifier
classifier = nn.Sequential(
    nn.Linear(25088, 4096),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(4096, 102),
    nn.LogSoftmax(dim=1)
)

# Replace the classifier of the pre-trained model with our new
classifier
model.classifier = classifier

# Use GPU if available, otherwise use CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

epochs = 5
learning_rate = 0.001

# Define the loss function and optimizer
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)

#  trainng model
print("Training process initializing .....\n")
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in dataloaders['train']:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()

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

        running_loss += loss.item()
```

```
    model.eval()
    with torch.no_grad():
#     Testing Accuracy     The network's accuracy is measured on the
test data
            valid_loss, accuracy = validation(model,
dataloaders['valid'], criterion,device)

# Validation Loss and Accuracy: During training, the validation loss
and accuracy are displayed
    print("Epoch: {}/{} | ".format(epoch+1, epochs),
                "Training Loss: {:.4f} |
".format(running_loss/len(dataloaders['train'])),
                "Validation Loss: {:.4f} |
".format(valid_loss/len(dataloaders['valid'])),
                "Validation Accuracy:
{:.4f}".format(accuracy/len(dataloaders['valid'])))

    running_loss = 0
    model.train()

print("\nTraining process is now complete!!")
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth"
to /root/.cache/torch/hub/checkpoints/vgg16-397923af.pth
100%|██████████| 528M/528M [00:02<00:00, 201MB/s]

Training process initializing .....

Epoch: 1/5 |  Training Loss: 3.8296 |  Validation Loss: 1.5498 |
Validation Accuracy: 0.5565
Epoch: 2/5 |  Training Loss: 2.1219 |  Validation Loss: 1.1611 |
Validation Accuracy: 0.6850
Epoch: 3/5 |  Training Loss: 1.8705 |  Validation Loss: 0.9339 |
Validation Accuracy: 0.7460
Epoch: 4/5 |  Training Loss: 1.7825 |  Validation Loss: 0.9321 |
Validation Accuracy: 0.7694
Epoch: 5/5 |  Training Loss: 1.7368 |  Validation Loss: 0.7304 |
Validation Accuracy: 0.7986

Training process is now complete!!
```

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.

```
# TODO: Do validation on the test set
# Initialize variables for evaluation metrics
total_correct = 0
total_samples = 0

# Test the model on the test dataset
with torch.no_grad():
    for inputs, labels in dataloaders['test']:
        inputs, labels = inputs.to(device), labels.to(device)  # Move
data to the appropriate device
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)

        total_samples += labels.size(0)
        total_correct += (predicted == labels).sum().item()

# Calculate accuracy
accuracy = (total_correct / total_samples) * 100
print(f'Test Accuracy: {accuracy:.2f}%')
```

```
Test Accuracy: 69.84%
```

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.

```
# TODO: Save the checkpoint
# Saving the model: The trained model is saved as a checkpoint along
with associated hyperparameters and the class_to_idx dictionary
checkpoint_path = 'flower_classifier_checkpoint.pth'
torch.save({
    'model_architecture': "VGG16",
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'num_epochs': epochs,
    'learning_rate': learning_rate,
}, checkpoint_path)
```

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.

```
# TODO: Write a function that loads a checkpoint and rebuilds the
model
# Loading checkpoints: There is a function that successfully loads a
checkpoint and rebuilds the model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
checkpoint_path = '/kaggle/working/flower_classifier_checkpoint.pth'
checkpoint = torch.load(checkpoint_path,map_location=device)
arch = checkpoint['model_architecture']
loaded_model = models.vgg16(weights=None)
loaded_model.classifier = nn.Sequential(
            nn.Linear(25088, 4096),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(4096, 102),
            nn.LogSoftmax(dim=1)
            )
# Loading state dictionary
loaded_model.load_state_dict(checkpoint['model_state_dict'])
```

```
<All keys matched successfully>
```

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.

```
# TODO: Process a PIL image for use in a PyTorch model
# Image Processing: The process_image function successfully converts a
PIL image into an object that can be used as input to a trained model
def process_image(image_path):
    ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
        returns a PyTorch tensor
    '''
    # Load the image using PIL
    pil_image = Image.open(image_path)

    # Define the image transforms
    transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
    ])

    # Apply the transforms to the image
    pil_image = transform(pil_image)

    return pil_image

image_path =
'/kaggle/input/102-flower-categories/test/30/image_03481.jpg'
output_image = process_image(image_path)
output_image
```

```
tensor([[[-0.8507, -0.6965, -0.5253,  ..., -0.0972,  0.8276,  1.1358],
         [-0.7993, -0.6794, -0.5082,  ...,  0.2111,  0.8789,  1.1529],
         [-0.7993, -0.6623, -0.4911,  ...,  0.5193,  0.9646,  1.2043],
         ...,
         [ 1.0673,  1.0844,  1.2728,  ..., -1.4500, -1.6213, -1.7240],
```

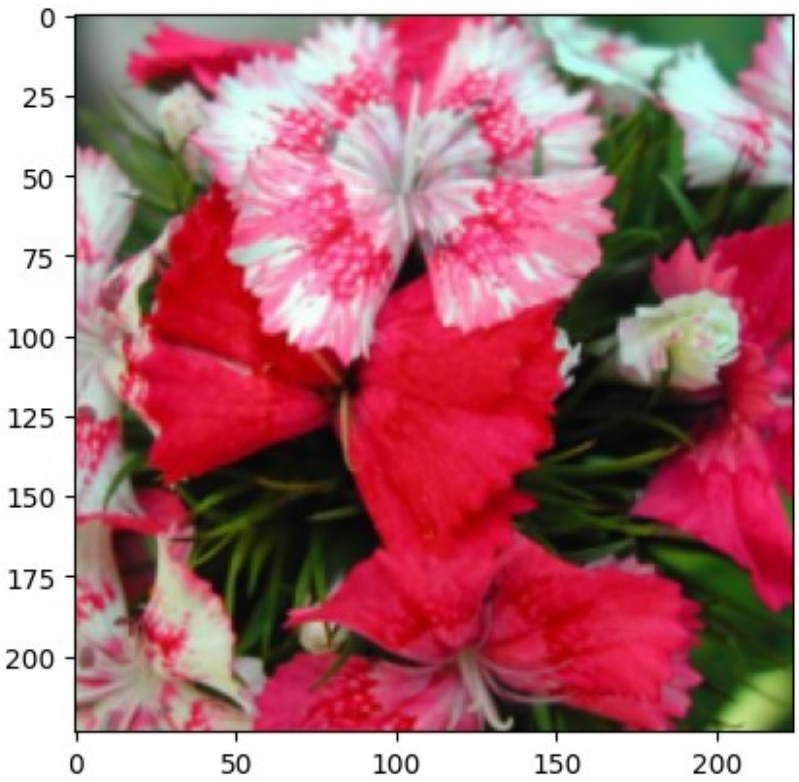
```
         [ 1.1015,  1.1187,  1.3413,  ..., -1.5870, -1.7069, -1.7412],
         [ 1.3070,  1.3070,  1.2899,  ..., -1.6898, -1.7240, -
1.7412]],

        [[-0.5126, -0.3901, -0.2150,  ...,  0.6078,  1.5007,  1.6933],
         [-0.5301, -0.4076, -0.2325,  ...,  0.7829,  1.6583,  1.9909],
         [-0.5301, -0.3901, -0.2150,  ...,  1.0455,  1.7283,  2.0959],
         ...,
         [ 1.0455,  1.0805,  0.8704,  ..., -0.8978, -1.0378, -1.0728],
         [ 1.1331,  1.2381,  1.0980,  ..., -1.0553, -1.1429, -1.1429],
         [ 1.2906,  1.3081,  1.0805,  ..., -1.1779, -1.1954, -
1.1604]],

        [[-0.3753, -0.2358, -0.0615,  ...,  0.5834,  1.5942,  1.9080],
         [-0.3578, -0.2358, -0.0615,  ...,  0.8622,  1.6988,  2.0474],
         [-0.3578, -0.2184, -0.0441,  ...,  1.2282,  1.8383,  2.1520],
         ...,
         [ 0.9145,  0.9668,  0.9319,  ..., -1.3687, -1.4559, -1.4733],
         [ 1.0365,  1.1237,  1.1237,  ..., -1.3687, -1.4733, -1.4559],
         [ 1.2282,  1.2457,  1.1062,  ..., -1.4559, -1.4907, -
1.4733]]])
```

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).

```
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
```

```
# Process the image using process_image function
redrawn_image = process_image(image_path)

# Display the processed image using imshow function
imshow(redrawn_image)
```

```
<Axes: >
```



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']

# TODO: Implement the code to predict the class from an image file
# Class Prediction: The predict function successfully takes the path
to an image and a checkpoint, then returns the top K most probably
classes for that image
def predict(image_path, model, topk=5):
    ''' Predict the class (or classes) of an image using a trained
deep learning model '''
    # Load and process the image
    processed_image = process_image(image_path).unsqueeze(0)
    # Move the image tensor to the same device as the model (CPU or
GPU)
    device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")
    model.to(device)
    processed_image = processed_image.to(device)

    # Set the model to evaluation mode
    model.eval()

    # Calculate the class probabilities
    with torch.no_grad():
        output = model(processed_image)
        probabilities = torch.exp(output)
        top_probs, top_indices = probabilities.topk(topk)

    # Convert the probabilities and indices to numpy arrays
    top_probs = top_probs.cpu().numpy().squeeze()
    top_indices = top_indices.cpu().numpy().squeeze()

    return top_probs, top_indices

image_path =
'/kaggle/input/102-flower-categories/test/30/image_03481.jpg'

top_probs, top_indices = predict(image_path,model)

cat_file = '/kaggle/input/integer-encoded-categories/cat_to_name.json'
with open(cat_file, 'r') as f:
  cat_to_name = json.load(f)

def find_key_by_value(value):
    class_to_idx = {'1': 0, '10': 1, '100': 2, '101': 3, '102': 4,
'11': 5, '12': 6, '13': 7, '14': 8, '15': 9, '16': 10, '17': 11, '18':
12, '19': 13, '2': 14, '20': 15, '21': 16, '22': 17, '23': 18, '24':
19, '25': 20, '26': 21, '27': 22, '28': 23, '29': 24, '3': 25, '30':
```

```
26, '31': 27, '32': 28, '33': 29, '34': 30, '35': 31, '36': 32, '37':
33, '38': 34, '39': 35, '4': 36, '40': 37, '41': 38, '42': 39, '43':
40, '44': 41, '45': 42, '46': 43, '47': 44, '48': 45, '49': 46, '5':
47, '50': 48, '51': 49, '52': 50,'53': 51, '54': 52, '55': 53, '56':
54, '57': 55, '58': 56, '59': 57, '6': 58, '60': 59, '61': 60, '62':
61, '63': 62, '64': 63, '65': 64, '66': 65, '67': 66, '68': 67, '69':
68, '7': 69, '70': 70, '71': 71, '72': 72, '73': 73, '74': 74, '75':
75, '76': 76, '77': 77, '78': 78, '79': 79, '8': 80, '80': 81, '81':
82, '82': 83, '83': 84, '84': 85, '85': 86, '86': 87, '87': 88, '88':
89, '89': 90, '9': 91, '90': 92, '91': 93, '92': 94, '93': 95, '94':
96, '95': 97, '96': 98, '97': 99, '98': 100, '99': 101}
    for key, val in class_to_idx.items():
        if val == value:
            return key

# Top 5 indices
for i,(prob, idx) in enumerate(zip(top_probs, top_indices)):
        actual_idx = find_key_by_value(idx)
        flower_name = cat_to_name[actual_idx]
        print(f"Predicted index {idx} Actrual index: {actual_idx}
Class: {flower_name} - Probability: {prob:.3f}")
        top_indices[i] = actual_idx
```

```
Predicted index 26 Actrual index: 30 Class: sweet william -
Probability: 0.338
Predicted index 72 Actrual index: 72 Class: azalea - Probability:
0.291
Predicted index 12 Actrual index: 18 Class: peruvian lily -
Probability: 0.134
Predicted index 49 Actrual index: 51 Class: petunia - Probability:
0.046
Predicted index 83 Actrual index: 82 Class: clematis - Probability:
0.043
```

Sanity Checking

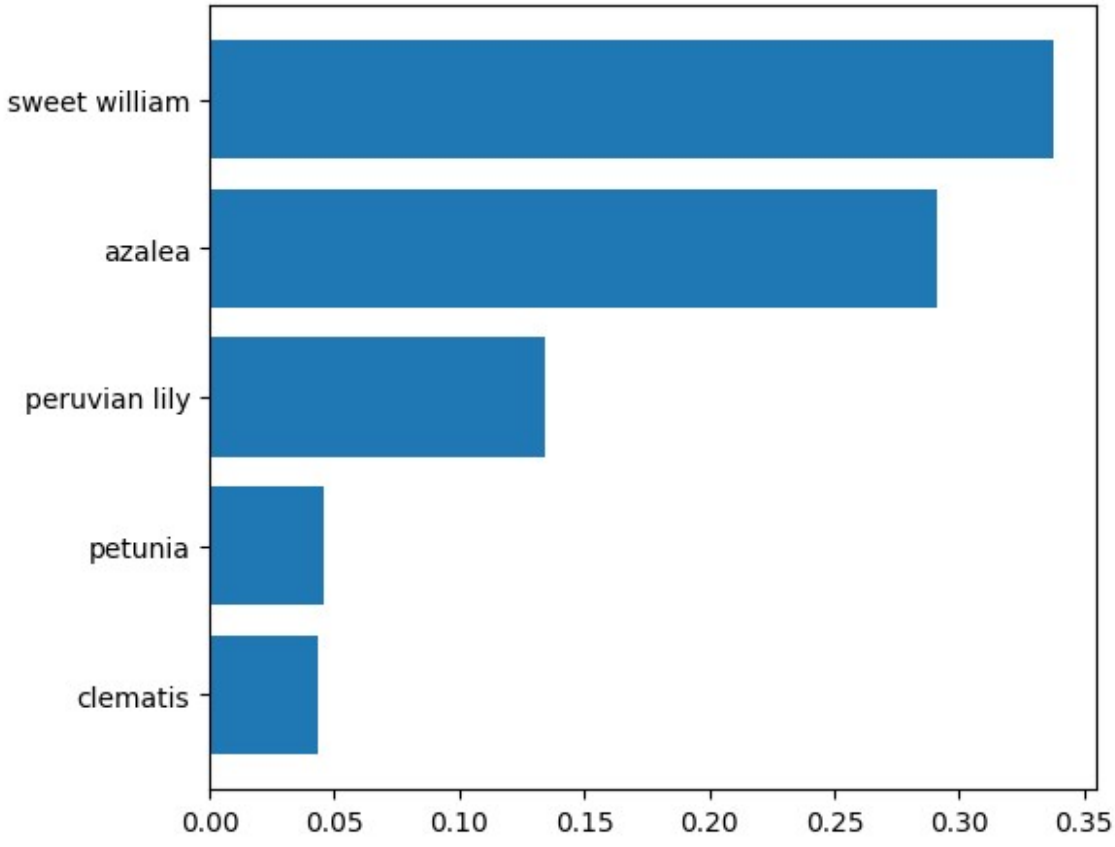
Now that you can use a trained model for predictions, check to make sure it makes sense. Even if the testing accuracy is high, it's always good to check that there aren't obvious bugs. Use `matplotlib` to plot the probabilities for the top 5 classes as a bar graph, along with the input image. It should look like this:

You can convert from the class integer encoding to actual flower names with the `cat_to_name.json` file (should have been loaded earlier in the notebook). To show a PyTorch tensor as an image, use the `imshow` function defined above.

```
# TODO: Display an image along with the top 5 classes
# Sanity Checking with matplotlib: A matplotlib figure is created
```

```
displaying an image and its associated top 5 most probable classes
with actual flower names
# Get the corresponding class names for the top labels
top_class_names = [cat_to_name[str(idx.item())] for idx in
top_indices.squeeze() ]

# Load and process the image
processed_image = process_image(image_path)

# Get the original image
original_image = Image.open(image_path)

# Plot the image
fig, (ax1, ax2) = plt.subplots(figsize=(6, 9),ncols=1, nrows=2)

# Show the original image
ax1.imshow(original_image)
ax1.set_title(top_class_names[0])
ax1.axis('off')

# Plot the top k probabilities as a bar graph
ax2.barh(np.arange(len(top_probs)), top_probs)
ax2.set_yticks(np.arange(len(top_probs)))
ax2.set_yticklabels(top_class_names)
ax2.invert_yaxis()

plt.tight_layout()
plt.show()
```


sweet william


sweet william
azalea
peruvian lily
petunia
clematis
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35

Testing model on test dataset

use pre_process function to normalize image and then predicting class

previous accuracy 69.84%

accuracy with pre-processing 75%

```
prediction = {}
incorrect_prediction = []
image_count = 0
# Iterate through subfolders and images inside them
test_dataset_path = '/kaggle/input/102-flower-categories/test'
for subfolder_name in os.listdir(test_dataset_path):
    subfolder_path = os.path.join(test_dataset_path, subfolder_name)
    no_of_images = 0
    prediction[subfolder_name] = 0
    for image in os.listdir(subfolder_path):
        image_path = '/'.join([subfolder_path,image])
        top_probs, top_indices = predict(image_path,loaded_model)
        no_of_images += 1
        if subfolder_name == find_key_by_value(top_indices[0]):
            prediction[subfolder_name] += 1
        else:

incorrect_prediction.append('/'.join([subfolder_name,image]))
        image_count += 1
    print(f'{subfolder_name} {cat_to_name[subfolder_name]} correct:
{prediction[subfolder_name]}/{no_of_images}')
```

```
7 moon orchid correct: 3/6
47 marigold correct: 3/3
17 purple coneflower correct: 9/9
81 frangipani correct: 13/13
19 balloon flower correct: 6/7
22 pincushion flower correct: 4/4
2 hard-leaved pocket orchid correct: 5/5
35 alpine sea holly correct: 6/6
92 bee balm correct: 7/11
50 common dandelion correct: 7/8
23 fritillary correct: 6/7
87 magnolia correct: 4/6
10 globe thistle correct: 2/3
5 english marigold correct: 2/4
61 cautleya spicata correct: 8/8
36 ruby-lipped cattleya correct: 4/7
20 giant white arum lily correct: 1/3
45 bolero deep blue correct: 3/3
60 pink-yellow dahlia correct: 10/10
```

```
27 prince of wales feathers correct: 1/3
64 silverbush correct: 3/5
41 barbeton daisy correct: 12/14
89 watercress correct: 14/15
39 siam tulip correct: 5/5
32 garden phlox correct: 3/6
98 mexican petunia correct: 3/4
25 grape hyacinth correct: 3/5
42 daffodil correct: 3/4
52 wild pansy correct: 7/8
75 thorn apple correct: 12/13
8 bird of paradise correct: 9/10
38 great masterwort correct: 8/8
12 colt's foot correct: 6/9
94 foxglove correct: 14/16
55 pelargonium correct: 4/7
49 oxeye daisy correct: 3/3
31 carnation correct: 0/2
62 japanese anemone correct: 3/4
53 primula correct: 4/14
101 trumpet creeper correct: 2/4
70 tree poppy correct: 2/4
34 mexican aster correct: 4/5
18 peruvian lily correct: 3/6
79 toad lily correct: 3/3
85 desert-rose correct: 4/10
88 cyclamen correct: 12/13
65 californian poppy correct: 5/7
67 spring crocus correct: 4/4
78 lotus lotus correct: 13/14
28 stemless gentian correct: 4/6
66 osteospermum correct: 3/4
56 bishop of llandaff correct: 6/8
72 azalea correct: 8/11
16 globe-flower correct: 1/3
13 king protea correct: 6/6
99 bromelia correct: 6/7
26 corn poppy correct: 1/5
74 rose correct: 8/14
15 yellow iris correct: 2/4
3 canterbury bells correct: 0/2
90 canna lily correct: 5/14
69 windflower correct: 2/3
77 passion flower correct: 25/25
102 blackberry lily correct: 6/6
86 tree mallow correct: 4/5
95 bougainvillea correct: 7/14
43 sword lily correct: 12/16
91 hippeastrum correct: 6/8
```

```
71 gazania correct: 9/9
1 pink primrose correct: 1/5
58 geranium correct: 11/14
59 orange dahlia correct: 5/7
97 mallow correct: 3/5
30 sweet william correct: 8/14
14 spear thistle correct: 1/3
76 morning glory correct: 4/4
84 columbine correct: 0/10
4 sweet pea correct: 4/6
83 hibiscus correct: 9/14
82 clematis correct: 13/17
57 gaura correct: 9/11
9 monkshood correct: 2/2
96 camellia correct: 5/9
46 wallflower correct: 21/21
21 fire lily correct: 2/2
44 poinsettia correct: 11/11
40 lenten rose correct: 4/8
80 anthurium correct: 10/11
6 tiger lily correct: 7/9
11 snapdragon correct: 6/9
68 bearded iris correct: 2/3
63 black-eyed susan correct: 4/4
37 cape flower correct: 8/8
51 petunia correct: 15/24
33 love in the mist correct: 5/8
100 blanket flower correct: 6/8
54 sunflower correct: 4/4
48 buttercup correct: 4/5
29 artichoke correct: 9/9
24 red ginger correct: 2/2
73 water lily correct: 25/28
93 ball moss correct: 2/6

# Calculate accuracy after defining pre-process function
acc = ((image_count-len(incorrect_prediction))/ image_count) * 100
print(f'Test Accuracy: {acc:.2f}%')

Test Accuracy: 75.09%
```