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 (http://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html) 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.

Acknowledgement

Special thanks to *Ioannis Breier*, a super mentor and *Juan Delgado*, a phenomenal teacher

It would not have been possible to complete this course and this final project without your guidance and assistance. Thanks you for the privilege to learn from you, you are simply wonderful !!!

Start from the next cell below for classifier training

Override default architecture name, learning rate, dropout rate, epochs in the cell "*set args*"

Training in CPU mode is possible but not practical, please run training in GPU mode

tip : if utilizing Udacity lab environment, save all your changes prior clicking GPU "Enable/Disble" button to avoid loss of unsaved changes

Start from the cell "Workspace Init" if you want to start with prediction using a saved checkpoint

In [1]:

```
# Imports here
%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 PIL import Image

from datetime import datetime
import json
import os
import glob

from workspace_utils import active_session
```

Load the data

Here you'll use `torchvision` to load the data (documentation (http://pytorch.org/docs/0.3.0/torchvision/index.html)). The data should be included alongside this notebook, otherwise you can download it here (https://s3.amazonaws.com/content.udacity-data.com/nd089/flower_data.tar.gz). 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'
savedir = 'chksav'
```

In [3]:

```
# TODO: Define your transforms for the training, validation, and testing sets
norm = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                            std=[0.229, 0.224, 0.225])

data_transforms = {'train': transforms.Compose([transforms.RandomRotation(30),
                                                transforms.RandomResizedCrop(224),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.ToTensor(),
                                                norm]),
                   'valid': transforms.Compose([transforms.Resize(256),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               norm]),
                   'test': transforms.Compose([transforms.Resize(256),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               norm])
                  }

# TODO: Load the datasets with ImageFolder
image_datasets = {k: datasets.ImageFolder(os.path.join(data_dir, k), transform=data_transforms[k])
                  for k in ['train','valid','test']}

# TODO: Using the image datasets and the trainforms, define the dataloaders
dataloaders = {k: torch.utils.data.DataLoader(image_datasets[k], batch_size=64, shuffle=True)
               for k in ['train','valid','test']}
```

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 (https://docs.python.org/2/library/json.html)](https://docs.python.org/2/library/json.html). This will give you a dictionary mapping the integer encoded categories to the actual names of the flowers.

In [4]:

```
import json

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 (https://review.udacity.com/#!/rubrics/1663/view)](https://review.udacity.com/#!/rubrics/1663/view) for guidance on successfully completing this section. Things you'll need to do:

- Load a [pre-trained network (http://pytorch.org/docs/master/torchvision/models.html)](http://pytorch.org/docs/master/torchvision/models.html) (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]:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
device
```

Out[5]:

```
device(type='cuda', index=0)
```

In [6]:

```
device.type
```

Out[6]:

```
'cuda'
```

In [7]:

```
# set up an arg object to store my stuff
class myArgs(dict):
    pass

args = myArgs()
```

set args

set your desired architecture, learning rate, drop out rate, epochs, interval for printing results during training

In [8]:

```
# set args attributes
# supported architectures : resnet18, densenet121, densenet161, vgg16
args.z_arch = 'resnet18'
args.z_hid = None
args.z_lrate = 0.001
args.z_dpout = 0.3
args.z_epochs = 10
args.z_print = 40

# list all my z_ attributes in args
#print('Args list:\n',[i for i in dir(args) if i.startswith('z_')])
```

In [9]:

```
# TODO: Build and train your network
def build_classifier(model, args):

    in_size = {
        'densenet121': 1024,
        'densenet161': 2208,
        'vgg16': 25088,
        }

    hid_size = {
        'densenet121': [500],
        'densenet161': [1000, 500],
        'vgg16': [4096, 4096,1000],
        }

    if args.z_dpout:
        p = args.z_dpout
    else:
        p = 0.5

    output_size = len(dataloaders['train'].dataset.classes)
    relu = nn.ReLU()
    dropout = nn.Dropout(p)
    output = nn.LogSoftmax(dim=1)

    if args.z_hid:
        h_list = args.z_hid.split(',')
        h_list = list(map(int, h_list)) # convert list from string to int
    else:
        h_list = hid_size[args.z_arch]

    h_layers = [nn.Linear(in_size[args.z_arch], h_list[0])]
    h_layers.append(relu)
    if args.z_arch[:3] == 'vgg':
        h_layers.append(dropout)

    if len(h_list) > 1:
        h_sz = zip(h_list[:-1], h_list[1:])
        for h1,h2 in h_sz:
            h_layers.append(nn.Linear(h1, h2))
            h_layers.append(relu)
            if args.z_arch[:3] == 'vgg':
                h_layers.append(dropout)

    last = nn.Linear(h_list[-1], output_size)
    h_layers.append(last)
    h_layers.append(output)

    print(h_layers)
    model.classifier = nn.Sequential(*h_layers)

    return model
```

In [10]:

```
# Build a classifier

# load a pre-trained model and override with own classifier
model = models.__dict__[args.z_arch](pretrained=True)

print('\nmodel architecture:', args.z_arch, '\n')

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

if args.z_arch == 'resnet18':
    model.fc = nn.Linear(model.fc.in_features, len(dataloaders['train'].dataset.classes))
    print('model.fc: ', model.fc)
else:
    model = build_classifier(model, args)
    print('\nmodel.classifier', model.classifier)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root/.torch/models/resnet1
8-5c106cde.pth
100%|██████████| 46827520/46827520 [00:02<00:00, 15774044.15it/s]

model architecture: resnet18

model.fc:  Linear(in_features=512, out_features=102, bias=True)
```

In [11]:

```
def validate(model, dataloaders, criterion):
    valid_loss = 0
    accuracy = 0

    for images, labels in iter(dataloaders['valid']):
        images, labels = images.to(device), labels.to(device)

        output = model.forward(images)
        valid_loss += criterion(output, labels).item()
        ps = torch.exp(output)
        equality = (labels.data == ps.max(dim=1)[1])
        accuracy += equality.type(torch.FloatTensor).mean()

    return valid_loss, accuracy
```

In [12]:

```
# training
def train(model, dataloaders, optimizer, criterion, epochs=2, print_freq=20, lr=0.001):
    if torch.cuda.is_available():
        print('*** training classifier in GPU mode ...\n')
    else:
        print('*** training classifier in CPU mode ...\n')

    model.to(device)
    start_time = datetime.now()

    print('epochs:', epochs, ', print_freq:', print_freq, ', lr:', lr, '\n')

    steps = 0


    for e in range(epochs):
        model.train()
        running_loss = 0
        for images, labels in iter(dataloaders['train']):
            steps +=1

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

            optimizer.zero_grad()

            output = model.forward(images)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            if steps % print_freq == 0:
                model.eval()

                with torch.no_grad():
                    valid_loss, accuracy = validate(model, dataloaders, criterion)

                print('Epoch: {}/{}..'.format(e+1, epochs),
                      'Training Loss: {:.3f}..'.format(running_loss/print_freq),
                      'Validation Loss: {:.3f}..'.format(valid_loss/len(dataloaders['valid'])),
                      'Validation Accuracy: {:.3f}%'.format(accuracy/len(dataloaders['valid']) * 100)
                     )
                running_loss = 0

                model.train()

    elapsed = datetime.now() - start_time

    print('\n*** classifier training done ! \nElapsed time[hh:mm:ss.ms]: {}'.format(elapsed))

    return model
```

In [13]:

```
# start training model
print('*** model architecture:', args.z_arch, '\n')
if args.z_arch == 'resnet18':
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.fc.parameters(), lr=args.z_lrate)
else:
    criterion = nn.NLLLoss()
    optimizer = optim.Adam(model.classifier.parameters(), lr=args.z_lrate)

with active_session():
    model = train(model, dataloaders, optimizer, criterion, args.z_epochs, args.z_print, args.z_lrate)
```

```
*** model architecture: resnet18

*** training classifier in GPU mode ...

epochs: 10 , print_freq: 40 , lr: 0.001

Epoch: 1/10.. Training Loss: 4.158.. Validation Loss: 3.258.. Validation Accuracy: 33.865%
Epoch: 1/10.. Training Loss: 3.043.. Validation Loss: 2.286.. Validation Accuracy: 54.889%
Epoch: 2/10.. Training Loss: 0.902.. Validation Loss: 1.642.. Validation Accuracy: 72.091%
Epoch: 2/10.. Training Loss: 1.826.. Validation Loss: 1.283.. Validation Accuracy: 81.048%
Epoch: 2/10.. Training Loss: 1.526.. Validation Loss: 1.029.. Validation Accuracy: 84.928%
Epoch: 3/10.. Training Loss: 1.114.. Validation Loss: 0.887.. Validation Accuracy: 86.337%
Epoch: 3/10.. Training Loss: 1.172.. Validation Loss: 0.765.. Validation Accuracy: 88.587%
Epoch: 4/10.. Training Loss: 0.257.. Validation Loss: 0.664.. Validation Accuracy: 89.240%
Epoch: 4/10.. Training Loss: 0.982.. Validation Loss: 0.611.. Validation Accuracy: 90.250%
Epoch: 4/10.. Training Loss: 0.895.. Validation Loss: 0.581.. Validation Accuracy: 90.216%
Epoch: 5/10.. Training Loss: 0.563.. Validation Loss: 0.503.. Validation Accuracy: 91.365%
Epoch: 5/10.. Training Loss: 0.792.. Validation Loss: 0.478.. Validation Accuracy: 92.413%
Epoch: 6/10.. Training Loss: 0.087.. Validation Loss: 0.458.. Validation Accuracy: 92.173%
Epoch: 6/10.. Training Loss: 0.718.. Validation Loss: 0.434.. Validation Accuracy: 91.365%
Epoch: 6/10.. Training Loss: 0.696.. Validation Loss: 0.412.. Validation Accuracy: 92.721%
Epoch: 7/10.. Training Loss: 0.362.. Validation Loss: 0.388.. Validation Accuracy: 92.962%
Epoch: 7/10.. Training Loss: 0.636.. Validation Loss: 0.381.. Validation Accuracy: 93.188%
Epoch: 7/10.. Training Loss: 0.685.. Validation Loss: 0.368.. Validation Accuracy: 92.688%
Epoch: 8/10.. Training Loss: 0.609.. Validation Loss: 0.368.. Validation Accuracy: 91.899%
Epoch: 8/10.. Training Loss: 0.594.. Validation Loss: 0.345.. Validation Accuracy: 92.981%
Epoch: 9/10.. Training Loss: 0.215.. Validation Loss: 0.323.. Validation Accuracy: 93.529%
Epoch: 9/10.. Training Loss: 0.559.. Validation Loss: 0.331.. Validation Accuracy: 92.207%
Epoch: 9/10.. Training Loss: 0.573.. Validation Loss: 0.335.. Validation Accuracy: 93.529%
Epoch: 10/10.. Training Loss: 0.447.. Validation Loss: 0.318.. Validation Accuracy: 93.135%
Epoch: 10/10.. Training Loss: 0.563.. Validation Loss: 0.311.. Validation Accuracy: 93.529%

*** classifier training done !
Elapsed time[hh:mm:ss.ms]: 0:16:30.486419
```

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 [14]:

```
# TODO: Do validation on the test set
def test(model, dataloaders, criterion, arch):
    print('*** model architecture:', arch, '\n')
    print('*** validating testset ...\n')
    model.cpu()
    model.eval()

    test_loss = 0
    total = 0
    match = 0

    start_time = datetime.now()

    with torch.no_grad():
        for images, labels in iter(dataloaders['test']):
            model, images, labels = model.to(device), images.to(device), labels.to(device)

            output = model.forward(images)
            test_loss += criterion(output, labels).item()
            total += images.shape[0]
            equality = labels.data == torch.max(output, 1)[1]
            match += equality.sum().item()

    model.test_accuracy = match/total * 100
    print('Test Loss: {:.3f}'.format(test_loss/len(dataloaders['test'])),
          'Test Accuracy: {:.2f}%'.format(model.test_accuracy))

    elapsed = datetime.now() - start_time
    print('\n*** test validation done ! \nElapsed time[hh:mm:ss.ms]: {}'.format(elapsed))
```

In [15]:

```
with active_session():
    test(model, dataloaders, criterion, args.z_arch)
```

```
*** model architecture: resnet18

*** validating testset ...

Test Loss: 0.386 Test Accuracy: 90.11%

*** test validation done !
Elapsed time[hh:mm:ss.ms]: 0:00:09.746072
```

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 [16]:

```
# TODO: Save the checkpoint
model = model.cpu() # back to CPU mode post training
model.class_to_idx = dataloaders['train'].dataset.class_to_idx

checkpoint = {'arch': args.z_arch,
              'state_dict': model.state_dict(),
              'class_to_idx' : model.class_to_idx
             }

if args.z_arch == 'resnet18':
    checkpoint['fc'] = model.fc
else:
    checkpoint['classifier'] = model.classifier

if not os.path.isdir(savedir):
        os.makedirs(savedir)

chkpt = datetime.now().strftime('%Y%m%d_%H%M%S') + '_' + args.z_arch + '.pth'
checkpt = os.path.join(savedir, chkpt)

torch.save(checkpoint, checkpt)
print('*** checkpoint: ', chkpt,  ', saved to: ', os.path.dirname(checkpt))
```

```
*** checkpoint:  20181006_220138_resnet18.pth , saved to:  chksav
```

Workspace Init

If there has been a switch from a GPU to CPU workspace on Udacity lab platform or if you start with prediction using a saved checkpoint without executing cells from the top to run training first, then

start from the next cell below to run prediction with a saved checkpoint

In [17]:

```
# Initialize workspace

try:
    args
except NameError:
    # re-import required packages
    %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 PIL import Image

    from datetime import datetime
    import json
    import os
    import glob

    from workspace_utils import active_session

    # set up required file directories
    data_dir = 'flowers'
    train_dir = data_dir + '/train'
    valid_dir = data_dir + '/valid'
    test_dir = data_dir + '/test'
    savedir = 'chksav'

    # re-define your transforms for the training, validation, and testing sets

    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])

    data_transforms = {'train': transforms.Compose([transforms.RandomRotation(30),
                                                    transforms.RandomResizedCrop(224),
                                                    transforms.RandomHorizontalFlip(),
                                                    transforms.ToTensor(),
                                                    normalize]),
                       'valid': transforms.Compose([transforms.Resize(256),
                                                   transforms.CenterCrop(224),
                                                   transforms.ToTensor(),
                                                   normalize]),
                       'test': transforms.Compose([transforms.Resize(256),
                                                   transforms.CenterCrop(224),
                                                   transforms.ToTensor(),
                                                   normalize])
                      }

    image_datasets = {k: datasets.ImageFolder(os.path.join(data_dir, k), transform=data_transforms[k])
                  for k in ['train','valid','test']}

    dataloaders = {k: torch.utils.data.DataLoader(image_datasets[k], batch_size=64, shuffle=True)
               for k in ['train','valid','test']}

    # re-load json class name mapper
    with open('cat_to_name.json', 'r') as f:
        cat_to_name = json.load(f)

    # set device
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
    print('device type:', device.type)

    # Re-instantiate args after exiting GPU mode
    class myArgs(dict):
        pass

    print('re-instantiate args...')
    args = myArgs()

    # get the last saved checkpoint
    if len(glob.glob(savedir+'/*.pth')) > 0 :
        checkpt = max(glob.glob(savedir+'/*.pth'), key=os.path.getctime)
        print('checkpoint:', checkpt, ' located')
    else:
        checkpt = None
        print('\n*** no saved checkpoint found !!!\n')
```

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 [18]:

```
# TODO: Write a function that loads a checkpoint and rebuilds the model
def load_checkpoint(filepath):

    # if no longer in GPU, force all tensors to be on CPU
    if device.type == 'cpu':
        checkpoint = torch.load(filepath, map_location=lambda storage, loc: storage)
    else:
        checkpoint = torch.load(filepath)

    model = models.__dict__[checkpoint['arch']](pretrained=True)

    args.z_arch = checkpoint['arch']
    #print('checkpoint[fc]:', checkpoint['fc'])

    if args.z_arch == 'resnet18':
        #model.fc = nn.Linear(model.fc.in_features, len(dataloaders['train'].dataset.classes))
        model.fc = checkpoint['fc']
        #print(model.fc)
    else:
        model.classifier = checkpoint['classifier']

    model.class_to_idx = checkpoint['class_to_idx']
    model.load_state_dict(checkpoint['state_dict'])

    return model, args
```

In [19]:
```
# load checkpoint
if checkpt:
    model, args = load_checkpoint(checkpt)
    # check result
    print('model architecture:', args.z_arch, '\n')
    if args.z_arch == 'resnet18':
        print('model.fc:\n', model.fc)
    else:
        print('model.classifier:\n', model.classifier)
else:
    print('\n*** stop !!! no saved checkpoint found \n')


# list all my z_ attributes in args
#print('Args list:\n',[i for i in dir(args) if i.startswith('z_')])
```

```
model architecture: resnet18

model.fc:
 Linear(in_features=512, out_features=102, bias=True)
```

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 (https://pillow.readthedocs.io/en/latest/reference/Image.html)). 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` (http://pillow.readthedocs.io/en/3.1.x/reference/Image.html#PIL.Image.Image.thumbnail) or `resize` (http://pillow.readthedocs.io/en/3.1.x/reference/Image.html#PIL.Image.Image.thumbnail) 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` (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.ndarray.transpose.html). The color channel needs to be first and retain the order of the other two dimensions.

In [20]:

```
def process_image(image):
    ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
        returns an Numpy array
    '''

    # TODO: Process a PIL image for use in a PyTorch model

    pil_img=image

    sz = image.size
    h = min(image.size)
    w = max(image.size)
    #print('size:',sz, ', h:',h, ', w:',w)

    # calculate ratio_aspect using original height & width
    # chosen h is 256, ratio aspect for adjusted w is original w/h
    ratio_aspect = w/h

    # get indices of short and long sides
    x = image.size.index(min(image.size))
    y = image.size.index(max(image.size))

    # calc new size with short side 256 pixels keeping ratio aspect
    new_sz = [0, 0]
    new_sz[x] = 256
    new_sz[y] = int(new_sz[x] * ratio_aspect)

    #print('new_sz:',new_sz, '\npre resized img:', pil_img)

    # resize base on short side of 256 pixels
    pil_img=image.resize(new_sz)
    #print('post resized image:', pil_img)

    # crop out the center 224x224 portion
    wid, hgt = new_sz
    #print('wid:', wid, ', hgt:', hgt)

    # calc left, top, right, bottom margin pos
    l_margin = (wid - 224)/2
    t_margin = (hgt - 224)/2
    r_margin = (wid + 224 )/2
    b_margin = (hgt + 224)/2

    #print('left:',l_margin, ', top:',t_margin, ', right:',r_margin, ', bottom:',b_margin)

    # crop the image
    pil_img=pil_img.crop((l_margin, t_margin, r_margin, b_margin))
    #print('cropped img:', pil_img)

    # convert to np array for normalization purpose
    np_img = np.array(pil_img)

    print('np_img.shape',np_img.shape)

    np_img = np_img/255
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    np_img = (np_img - mean)/std

    # transpose to get color channel to 1st pos
    np_img = np_img.transpose((2, 0, 1))

    return np_img
```

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 [21]:

```
def imshow(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)

    ax.imshow(image)

    return ax
```

In [22]:

```
# confingure args to randomly select an image file, its class and path
def pick_a_pic(dset_dir, dset_type, args):
    args.z_imgcls = np.random.choice(dataloaders[dset_type].dataset.classes)
    args.z_rndimg = np.random.choice(os.listdir(dset_dir + '/' + args.z_imgcls))
    args.z_rndimgpth = dset_dir + '/' + args.z_imgcls + '/' + args.z_rndimg
    return args
```

In [23]:

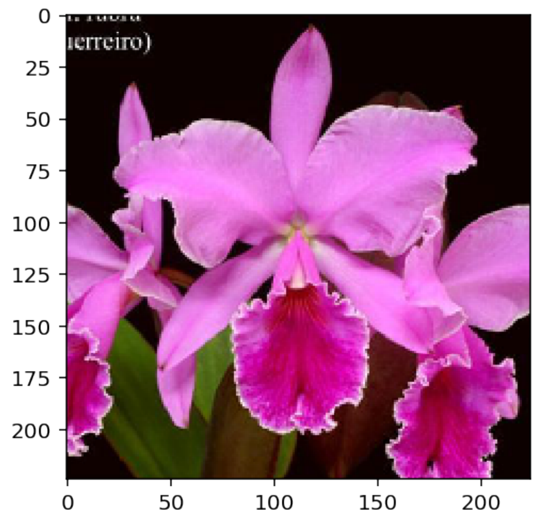
```
# pick an image from test set
args = pick_a_pic(test_dir,'test', args)

print('class:', args.z_imgcls,', image:', args.z_rndimg, '\npath:', args.z_rndimgpth,'\n')
print('args parameters:\n',[i for i in dir(args) if i.startswith('z_')], '\n')
```

```
class: 36 , image: image_04334.jpg
path: flowers/test/36/image_04334.jpg

args parameters:
 ['z_arch', 'z_dpout', 'z_epochs', 'z_hid', 'z_imgcls', 'z_lrate', 'z_print', 'z_rndimg', 'z_rndimgpth'
]
```

In [24]:

```
# Pass a pic from test set to image_process to convert into torch FloatTensor
with Image.open(args.z_rndimgpth) as image:
    np_img = process_image(image)
    imshow(np_img)
```

```
np_img.shape (224, 224, 3)
```



In [25]:
```
# Restore image to original config
with Image.open(args.z_rndimgpth) as image:
    plt.imshow(image)
```

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)` (http://pytorch.org/docs/master/torch.html#torch.topk). 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 [26]:

```
def predict(image_path, cat_to_name, arch, model, topk=5):
    ''' Predict the class (or classes) of an image using a trained deep learning model.
    '''

    # TODO: Implement the code to predict the class from an image file
    model.cpu()
    model.eval()

    pil_img = Image.open(image_path)
    image = process_image(pil_img)
    image = torch.FloatTensor(image)

    model, image = model.to(device), image.to(device)

    print('\nori image.shape:', image.shape)
    image.unsqueeze_(0) # add a new dimension in pos 0
    print('new image.shape:', image.shape, '\n')

    output = model.forward(image)
    #print('output:\n', output.data, '\n')

    # get the top k classes of prob
    if arch == 'resnet18':
        ps = F.softmax(output, dim=1).data[0]
    else:
        ps = torch.exp(output).data[0]

    #print('ps:\n', ps, '\n')
    topk_prob, topk_idx = ps.topk(topk)

    # bring back to cpu and convert to numpy
    topk_probs = topk_prob.cpu().numpy()
    topk_idxs = topk_idx.cpu().numpy()

    # map topk_idx to classes in model.class_to_idx
    idx_class={i: k for k, i in model.class_to_idx.items()}
    topk_classes = [idx_class[i] for i in topk_idxs]

    # map class to class name
    topk_names = [cat_to_name[i] for i in topk_classes]

    print('*** Top ', topk, ' classes ***')
    print('class names:   ', topk_names)
    print('classes:       ', topk_classes)
    print('probabilities: ', topk_probs)

    return topk_classes, topk_names, topk_probs
```

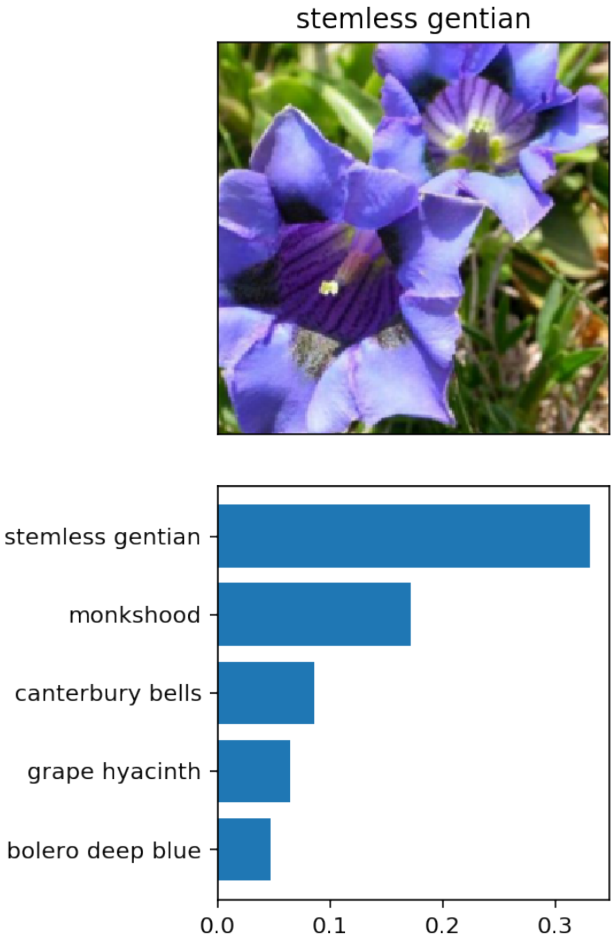
In [27]:

```
# Call predict() to predict the class (or classes) of an image
args.z_topk = 5
#args.z_imgpath = test_dir + '/10/image_07090.jpg'
args = pick_a_pic(test_dir,'test', args)
print('image path:', args.z_rndimgpth, '\n')

with active_session():
    start_time = datetime.now()

    topk_classes, topk_names, topk_probs = predict(args.z_rndimgpth, cat_to_name, args.z_arch, model)

    elapsed = datetime.now() - start_time
    print('\n*** predict elapsed time[hh:mm:ss.ms]: {}'.format(elapsed))
```

```
image path: flowers/test/38/image_05799.jpg

np_img.shape (224, 224, 3)

ori image.shape: torch.Size([3, 224, 224])
new image.shape: torch.Size([1, 3, 224, 224])

*** Top  5  classes ***
class names:    ['great masterwort', 'bee balm', 'gaura', 'pincushion flower', 'clematis']
classes:        ['38', '92', '57', '22', '82']
probabilities:  [  9.99165416e-01   2.37751039e-04   2.36056134e-04   3.35714176e-05
   2.96454928e-05]

*** predict elapsed time[hh:mm:ss.ms]: 0:00:00.053370
```

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.

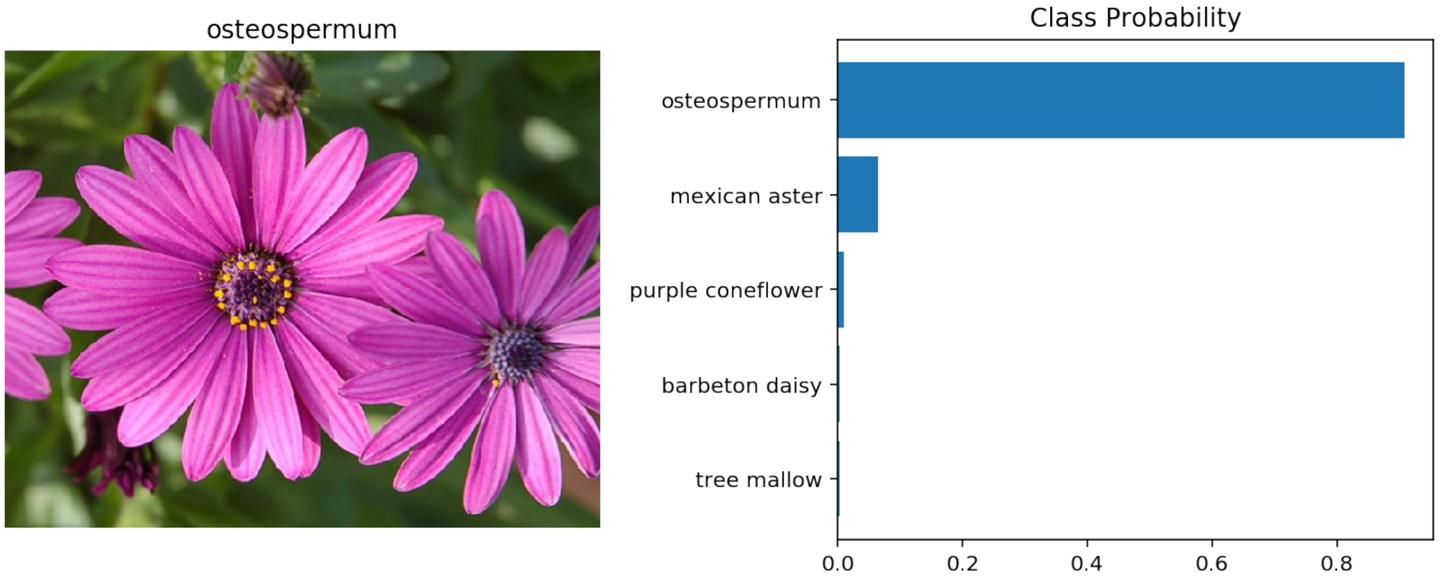
In [28]:

```
# TODO: Display an image along with the top 5 classes
def show_classifier(model, imgcls, imgpth, arch, cat_to_name):

    topk_classes, topk_names, topk_probs = predict(imgpth, cat_to_name, arch, model)

    img = Image.open(imgpth)

    # get img name
    img_name = topk_names[0] #cat_to_name[imgcls]

    fig, (ax1, ax2) = plt.subplots(figsize=(10,4), ncols=2)
    ax1.set_title(img_name)
    ax1.imshow(img)
    ax1.axis('off')

    y_pos = np.arange(len(topk_probs))
    ax2.barh(y_pos, topk_probs)
    ax2.set_yticks(y_pos)
    ax2.set_yticklabels(topk_names)
    ax2.invert_yaxis()
    #ax2.set_xlim(0, 1.1)
    ax2.set_title('Class Probability')

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

In [29]:

```
# display a randomly selected image with its top 5 classes probabilities
args = pick_a_pic(test_dir,'test', args)
print('image path:', args.z_rndimgpth, '\n')
show_classifier(model, args.z_imgcls, args.z_rndimgpth, args.z_arch, cat_to_name)
```

```
image path: flowers/test/66/image_05549.jpg

np_img.shape (224, 224, 3)

ori image.shape: torch.Size([3, 224, 224])
new image.shape: torch.Size([1, 3, 224, 224])

*** Top  5  classes ***
class names:    ['osteospermum', 'mexican aster', 'purple coneflower', 'barbeton daisy', 'tree mallow']
classes:        ['66', '34', '17', '41', '86']
probabilities:  [ 0.90905267  0.06390259  0.00937185  0.0035635   0.00237345]
```



References

AIPNP Neural Network Lesson 4

[Part 5 - Inference and Validation (https://youtu.be/coBbbrGZXI0)](https://youtu.be/coBbbrGZXI0)

[Part 6 - Saving and Loading Models (https://youtu.be/HiTih59dCWQ)](https://youtu.be/HiTih59dCWQ)

[Part 7 - Loading Data Sets with Torchvision (https://youtu.be/hFu7GTfRWks)](https://youtu.be/hFu7GTfRWks)

[Part 8 - Transfer Learning (https://youtu.be/3eqn5sgCOsY)](https://youtu.be/3eqn5sgCOsY)

cat_to_name.json

helper.py

workspace_utils.py

Pytorch

[Pytorch 0.4.1 Documentation (https://pytorch.org/docs/stable/index.html)](https://pytorch.org/docs/stable/index.html)

[Pytorch 0.4.1 tutorial (https://pytorch.org/tutorials/index.html)](https://pytorch.org/tutorials/index.html)

[Iterate through custom dataset (https://discuss.pytorch.org/t/trying-to-iterate-through-my-custom-dataset/1909)](https://discuss.pytorch.org/t/trying-to-iterate-through-my-custom-dataset/1909)

[Problem loading model trained on GPU (https://discuss.pytorch.org/t/problem-loading-model-trained-on-gpu/17745)](https://discuss.pytorch.org/t/problem-loading-model-trained-on-gpu/17745)

Python & Miscellaneous

[Objects and classes in Python (http://jfine-python-classes.readthedocs.io/en/latest/construct.html)](http://jfine-python-classes.readthedocs.io/en/latest/construct.html)

[find the latest file in a folder (https://stackoverflow.com/questions/39327032/how-to-get-the-latest-file-in-a-folder-using-python)](https://stackoverflow.com/questions/39327032/how-to-get-the-latest-file-in-a-folder-using-python)

[How to get the aspect ratio of an image? (https://math.stackexchange.com/questions/180804/how-to-get-the-aspect-ratio-of-an-image)](https://math.stackexchange.com/questions/180804/how-to-get-the-aspect-ratio-of-an-image)

[Activation functions and its types (https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f)](https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f)

[argparse (https://docs.python.org/dev/library/argparse.html#)](https://docs.python.org/dev/library/argparse.html#)