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)](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.

In [1]:
```
#numpys
import numpy as np
import matplotlib.pyplot as plt
#torchvision
import torchvision
from torchvision import datasets, transforms, models
#torches
import torch
from torch import nn, optim
from torch.autograd import Variable
import torch.nn.functional as F
#normal
from collections import OrderedDict
import time
import random, os
#PIL
from PIL import Image
#json
import json
```

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'
```

In [3]:

```
# TODO: Define your transforms for the training, validation, and testing sets
#throughout this project, in the parts i wasn't sure how to do, I turned to thi
s link which aided me: https://github.com/shamjam/Create-Your-Own-Image-Classif
ier/blob/master/Image%20Classifier%20Project.ipynb
#data_transforms = the training, validation, and testing sets

#defining the transformations for training
training_transforms = transforms.Compose([transforms.RandomRotation(30),
                                          transforms.RandomResizedCrop(224),
                                          transforms.RandomHorizontalFlip(),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456,
0.406],
                                                               [0.229, 0.224,
0.225])])

#defining the transformations for validation
validation_transforms = transforms.Compose([transforms.Resize(256),
                                            transforms.CenterCrop(224),
                                            transforms.ToTensor(),
                                            transforms.Normalize([0.485, 0.456,
0.406],
                                                                 [0.229, 0.224,
0.225])])

#defining the transformations for testing
testing_transforms = transforms.Compose([transforms.Resize(256),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.485, 0.456, 0.
406],
                                                              [0.229, 0.224, 0.
225])])



# TODO: Load the datasets with ImageFolder


#loading traing transformation into train_dir
training_data = datasets.ImageFolder(train_dir, transform=training_transforms)
#loading validation transformations into valid_dir
validation_data = datasets.ImageFolder(valid_dir, transform=validation_transfor
ms)
#loading testing transformations into test_dir
testing_data = datasets.ImageFolder(test_dir ,transform = testing_transforms)


# TODO: Using the image datasets and the trainforms, define the dataloaders

#defining the train loader
trainingloader = torch.utils.data.DataLoader(training_data, batch_size=64, shuf
fle=True)
#defining the validation loader
validationloader = torch.utils.data.DataLoader(validation_data, batch_size =32,
shuffle = True)
#defining the testing loader
testingloader = torch.utils.data.DataLoader(testing_data, batch_size = 20, shuf
fle = True)
```

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. 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.

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.

In [8]:
```python
# TODO: Build and train your network

#i chose this from the library
model = models.vgg16(pretrained=True)

#parameters in the model the require_grad is set false
for param in model.parameters():
    param.requires_grad = False

#import
from collections import OrderedDict
#by using the sequential, it saves time
model.classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(25088, 2048)),
                          ('relu', nn.ReLU()),
                          ('fc2', nn.Linear(2048, 256)),
                          ('relu', nn.ReLU()),
                          ('fc3', nn.Linear(256, 102)),
                          ('output', nn.LogSoftmax(dim=1))
                          ]))
#print the model
print(model)
#cuda type
model = model.to('cuda')
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
  )
  (classifier): Sequential(
    (fc1): Linear(in_features=25088, out_features=2048, bias=True)
    (relu): ReLU()
    (fc2): Linear(in_features=2048, out_features=256, bias=True)
    (fc3): Linear(in_features=256, out_features=102, bias=True)
    (output): LogSoftmax()
  )
)
```

In [11]:
```
# Train the classifier layers using backpropagation using the pre-trained netwo
rk to get the features
# Track the loss and accuracy on the validation set to determine the best hyper
parameters

#declare epoch,steps,runningloss, and printevery
epochs = 3
steps = 0
running_loss = 0
print_every = 5

#for in loop to try all of them
for epoch in range(epochs):
    for inputs, labels in trainingloader:
        steps += 1
        # Move input and label tensors to the default device

        inputs, labels = inputs.to('cuda'), labels.to('cuda')

        optimizer.zero_grad()

        #Forward pass
        logps = model.forward(inputs)
        loss = criterion(logps, labels)
        # Backward pass
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        if steps % print_every == 0:
            valid_loss = 0
            accuracy = 0
            model.eval()
            with torch.no_grad():
                for inputs, labels in validationloader:
                    inputs, labels = inputs.to('cuda'), labels.to('cuda')

                    logps = model.forward(inputs)
                    batch_loss = criterion(logps, labels)
                    valid_loss += batch_loss.item()

                    # Calculate accuracy
                    ps = torch.exp(logps)
                    top_p, top_class = ps.topk(1, dim=1)
                    equals = top_class == labels.view(*top_class.shape)
                    accuracy += torch.mean(equals.type(torch.FloatTensor)).item
()

            print(f"Epoch {epoch+1}/{epochs}.. "
                  f"Loss: {running_loss/print_every:.3f}.. "
                  f"Validation Loss: {valid_loss/len(validationloader):.3f}.. "
                  f"Accuracy: {accuracy/len(validationloader):.3f}")
            running_loss = 0
            #training the actual thing
            model.train()
```

```
Epoch 1/3.. Loss: 4.774.. Validation Loss: 4.124.. Accuracy: 0.150
Epoch 1/3.. Loss: 3.981.. Validation Loss: 3.520.. Accuracy: 0.244
Epoch 1/3.. Loss: 3.352.. Validation Loss: 3.015.. Accuracy: 0.390
Epoch 1/3.. Loss: 3.083.. Validation Loss: 2.568.. Accuracy: 0.436
Epoch 1/3.. Loss: 2.828.. Validation Loss: 2.144.. Accuracy: 0.495
Epoch 1/3.. Loss: 2.553.. Validation Loss: 1.960.. Accuracy: 0.484
Epoch 1/3.. Loss: 2.278.. Validation Loss: 1.712.. Accuracy: 0.572
Epoch 1/3.. Loss: 1.992.. Validation Loss: 1.716.. Accuracy: 0.561
Epoch 1/3.. Loss: 2.017.. Validation Loss: 1.419.. Accuracy: 0.621
Epoch 1/3.. Loss: 1.821.. Validation Loss: 1.321.. Accuracy: 0.641
Epoch 1/3.. Loss: 1.607.. Validation Loss: 1.240.. Accuracy: 0.652
Epoch 1/3.. Loss: 1.651.. Validation Loss: 1.100.. Accuracy: 0.681
Epoch 1/3.. Loss: 1.518.. Validation Loss: 1.015.. Accuracy: 0.733
Epoch 1/3.. Loss: 1.396.. Validation Loss: 1.003.. Accuracy: 0.725
Epoch 1/3.. Loss: 1.412.. Validation Loss: 0.900.. Accuracy: 0.752
Epoch 1/3.. Loss: 1.314.. Validation Loss: 1.006.. Accuracy: 0.730
Epoch 1/3.. Loss: 1.380.. Validation Loss: 0.907.. Accuracy: 0.752
Epoch 1/3.. Loss: 1.398.. Validation Loss: 0.966.. Accuracy: 0.756
Epoch 1/3.. Loss: 1.402.. Validation Loss: 0.863.. Accuracy: 0.754
Epoch 1/3.. Loss: 1.113.. Validation Loss: 0.823.. Accuracy: 0.745
Epoch 2/3.. Loss: 1.107.. Validation Loss: 0.801.. Accuracy: 0.771
Epoch 2/3.. Loss: 1.091.. Validation Loss: 0.825.. Accuracy: 0.775
Epoch 2/3.. Loss: 1.042.. Validation Loss: 0.830.. Accuracy: 0.772
Epoch 2/3.. Loss: 1.127.. Validation Loss: 0.922.. Accuracy: 0.756
Epoch 2/3.. Loss: 1.047.. Validation Loss: 0.825.. Accuracy: 0.759
Epoch 2/3.. Loss: 1.165.. Validation Loss: 0.675.. Accuracy: 0.812
Epoch 2/3.. Loss: 1.062.. Validation Loss: 0.891.. Accuracy: 0.780
Epoch 2/3.. Loss: 1.202.. Validation Loss: 0.679.. Accuracy: 0.812
Epoch 2/3.. Loss: 0.932.. Validation Loss: 0.676.. Accuracy: 0.805
Epoch 2/3.. Loss: 1.266.. Validation Loss: 0.640.. Accuracy: 0.814
Epoch 2/3.. Loss: 1.042.. Validation Loss: 0.815.. Accuracy: 0.773
Epoch 2/3.. Loss: 1.033.. Validation Loss: 0.682.. Accuracy: 0.796
Epoch 2/3.. Loss: 1.054.. Validation Loss: 0.607.. Accuracy: 0.820
Epoch 2/3.. Loss: 1.121.. Validation Loss: 0.712.. Accuracy: 0.798
Epoch 2/3.. Loss: 1.049.. Validation Loss: 0.594.. Accuracy: 0.825
Epoch 2/3.. Loss: 0.809.. Validation Loss: 0.639.. Accuracy: 0.821
Epoch 2/3.. Loss: 1.040.. Validation Loss: 0.682.. Accuracy: 0.819
Epoch 2/3.. Loss: 0.951.. Validation Loss: 0.700.. Accuracy: 0.799
Epoch 2/3.. Loss: 0.937.. Validation Loss: 0.645.. Accuracy: 0.833
Epoch 2/3.. Loss: 0.695.. Validation Loss: 0.574.. Accuracy: 0.834
Epoch 2/3.. Loss: 0.908.. Validation Loss: 0.574.. Accuracy: 0.839
Epoch 3/3.. Loss: 0.957.. Validation Loss: 0.666.. Accuracy: 0.808
Epoch 3/3.. Loss: 0.711.. Validation Loss: 0.593.. Accuracy: 0.839
Epoch 3/3.. Loss: 0.962.. Validation Loss: 0.658.. Accuracy: 0.820
Epoch 3/3.. Loss: 0.889.. Validation Loss: 0.669.. Accuracy: 0.820
Epoch 3/3.. Loss: 0.757.. Validation Loss: 0.538.. Accuracy: 0.864
Epoch 3/3.. Loss: 0.726.. Validation Loss: 0.586.. Accuracy: 0.847
Epoch 3/3.. Loss: 0.842.. Validation Loss: 0.604.. Accuracy: 0.853
Epoch 3/3.. Loss: 1.018.. Validation Loss: 0.593.. Accuracy: 0.849
Epoch 3/3.. Loss: 0.839.. Validation Loss: 0.630.. Accuracy: 0.837
Epoch 3/3.. Loss: 0.838.. Validation Loss: 0.628.. Accuracy: 0.823
Epoch 3/3.. Loss: 0.957.. Validation Loss: 0.649.. Accuracy: 0.824
Epoch 3/3.. Loss: 0.813.. Validation Loss: 0.556.. Accuracy: 0.851
Epoch 3/3.. Loss: 0.647.. Validation Loss: 0.538.. Accuracy: 0.857
Epoch 3/3.. Loss: 0.925.. Validation Loss: 0.574.. Accuracy: 0.841
Epoch 3/3.. Loss: 0.791.. Validation Loss: 0.609.. Accuracy: 0.831
Epoch 3/3.. Loss: 0.766.. Validation Loss: 0.625.. Accuracy: 0.841
Epoch 3/3.. Loss: 0.917.. Validation Loss: 0.518.. Accuracy: 0.870
Epoch 3/3.. Loss: 0.737.. Validation Loss: 0.614.. Accuracy: 0.834
Epoch 3/3.. Loss: 0.816.. Validation Loss: 0.531.. Accuracy: 0.861
Epoch 3/3.. Loss: 0.812.. Validation Loss: 0.574.. Accuracy: 0.845
```

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 [15]:

```
# TODO: Do validation on the test set
test_loss = 0
accuracy = 0
model.to('cuda')
#labling to cuda and calculating the accuracy
with torch.no_grad():
    for inputs, labels in testingloader:
        inputs, labels = inputs.to('cuda'), labels.to('cuda')
        logps = model.forward(inputs)
        batch_loss = criterion(logps, labels)

        test_loss += batch_loss.item()

        # Calculate accuracy
        ps = torch.exp(logps)
        top_p, top_class = ps.topk(1, dim=1)
        equals = top_class == labels.view(*top_class.shape)
        accuracy += torch.mean(equals.type(torch.FloatTensor)).item()
#this is the accuracy
print(f"Test accuracy: {accuracy/len(testingloader):.3f}")
```

```
Test accuracy: 0.819
```

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 [31]:
```
# TODO: Save the checkpoint
model.class_to_idx = training_data.class_to_idx #saving mapping between predict
ed class and class name,
#second variable is a class name in numeric

#creating dictionary
checkpoint = {'classifier': model.classifier,
              'state_dict': model.state_dict (),
              'mapping':    model.class_to_idx
             }
#saving the checkpoint
torch.save (checkpoint, 'project_checkpoint.pth')
```

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 [34]:
```
# TODO: Write a function that loads a checkpoint and rebuilds the model
#method for loading
def loading_model (file_path):
    checkpoint = torch.load (file_path) #loading checkpoint from a file
    model = models.vgg16(pretrained=True)


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

    for param in model.parameters():
        param.requires_grad = False #turning off tuning of the model

    return model
```

In [35]:
```python
#calling
model_verify = loading_model ('project_checkpoint.pth')
model_verify
```

Out[35]:
```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
lse)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
  )
  (classifier): Sequential(
    (fc1): Linear(in_features=25088, out_features=2048, bias=True)
    (relu): ReLU()
    (fc2): Linear(in_features=2048, out_features=256, bias=True)
    (fc3): Linear(in_features=256, out_features=102, bias=True)
    (output): LogSoftmax()
  )
)
```

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 [36]:
```
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
    #this is to process a pil img
    img_pil = Image.open(image)
    img_transforms = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.22
5])
    ])
    #transofrmation
    image = img_transforms(img_pil)

    return image
```

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

In [39]:
```
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 dis
played
    image = np.clip(image, 0, 1)

    ax.imshow(image)

    return ax
```

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 [40]:
```
def predict(image_path, model, topk=5):
    ''' Predict the class (or classes) of an image using a trained deep learnin
g model.
    '''
    #cuda
    model.to('cuda')
    model.eval()
    img = process_image(image_path)
    img = img.numpy()
    img = torch.from_numpy(np.array([img])).float()

    with torch.no_grad():
        output = model.forward(img.cuda())
    #calculating the probability
    probability = torch.exp(output).data
    #return
    return probability.topk(topk)
```

In [41]:
```
img = "flowers/test/10/image_07090.jpg"
probability, classes = predict(img, model)
#printing both
print (probability)
print (classes)
```

```
tensor([[ 0.9924,  0.0044,  0.0031,  0.0000,  0.0000]], device='cuda:0')
tensor([[  1,  17,  24,  94,   8]], device='cuda:0')
```

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 [42]:

```
# TODO: Display an image along with the top 5 classes
plt.rcParams["figure.figsize"] = (10,10)
plt.subplot(211)

index = 1
path = test_dir + '/1/image_06743.jpg'

probabilities = predict(path, model)
image = process_image(path)

#graph setup
axs = imshow(image, ax = plt)
axs.axis('off')
axs.title(cat_to_name[str(index)])
axs.show()

#graph by probablilies and concat to name
a = np.array(probabilities[0][0])
b = [cat_to_name[str(index+1)] for index in np.array(probabilities[1][0])]

N=float(len(b))
fig,ax = plt.subplots(figsize=(10,5))
width = 0.5
tickLocations = np.arange(N)

#creating the graph
ax.bar(tickLocations, a, width, linewidth=4.0, align = 'center')
ax.set_xticks(ticks = tickLocations)
ax.set_xticklabels(b)
ax.set_xlim(min(tickLocations)-0.6,max(tickLocations)+0.6)
ax.set_yticks([0.2,0.4,0.6,0.8,1,1.2])
ax.set_ylim((0,1))
ax.yaxis.grid(True)
#show plot
plt.show()
```

pink primrose