Note: This was a 'guided' project originally rendered in a Jupyter notebook and includes Udacity guidelines throughout the code.

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.

```
## Program/Final Project: Developing an AI Application
## Course: Udacity AI Programming with Python
## Programmer/Student: Phyllis La Monica
## Date: Project started 12August18

## Note: Last Cell -References used for this Project

## Part 1
```

```
# Imports here
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import matplotlib.pyplot as plt
import numpy as np
import time

import torch
from torch import nn
from torch import optim
import torch.nn.functional as F
from torch.autograd import Variable
from torchvision import datasets, transforms, models

from collections import OrderedDict
from PIL import Image

import json
```

Load the data

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

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

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

```
data_dir = 'flowers'
```

```
train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

print('Hello World!')

Hello World!

# TODO: Define your transforms for the training, validation, and
testing sets
#data_transforms =

##.........Define transforms for the training data validation
data, and testing data

train_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])])

valid_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])])

test_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
#image_datasets = DONE
## image_dataset = datasets.ImageFolder(data_dir,
transform=transforms) ##corrected this to _datasets
image_datasets = datasets.ImageFolder(data_dir,
transform=transforms)


##....... Pass transforms here
train_data = datasets.ImageFolder(data_dir + '/train',
transform=train_transforms)
valid_data = datasets.ImageFolder(data_dir + '/valid',
transform=test_transforms)  ##...added for validation
test_data = datasets.ImageFolder(data_dir + '/test',
transform=test_transforms)


# TODO: Using the image datasets and the train forms, define the
dataloaders
#dataloaders = DONE
##.......dataloader = # TODO: use the ImageFolder dataset to
create the DataLoader
dataloader = torch.utils.data.DataLoader(image_datasets,
batch_size=32, shuffle=True)

##...#dataloader = # (see less9) use the ImageFolder datasets to
create the DataLoader, define train/valid/test loaders
trainloader = torch.utils.data.DataLoader(train_data,
batch_size=64, shuffle=True)
validloader = torch.utils.data.DataLoader(valid_data,
batch_size=64)  ##...added for validation
testloader = torch.utils.data.DataLoader(test_data,
batch_size=32)

###model = models.densenet121(pretrained=True)  First USED but
REPLACED due to poor output results.
model = models.vgg16(pretrained=True)
model

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=False)
    (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=False)
    (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=False)
    (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=False)
    (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=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=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](). This will give you a dictionary mapping the integer encoded categories to the actual names of the flowers.

```
#* Load in mapping from category label to category name.
#* For: dictionary mapping integer encoded categories to actual
names of the flowers.

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) 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) (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.

```
## Build and train the classifier.
#* Define a new, untrained feed-forward network as a classifier,
using ReLU activations and dropout
## Lesson ref: inferences/7, build network/8, training/6

## Including nn.Dropout, log-softmax -- use torch.exp at output
for validation and testing loops

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

    #from collections import OrderedDict  -- included in first
cell
    classifier = nn.Sequential(OrderedDict([
                         ('fc1', nn.Linear(25088, 4096)),
                         ('relu1', nn.ReLU()),
                         ('dropout1', nn.Dropout(.25)),  ##
Revised dropout from .5 to .25.
                         ('fc2', nn.Linear(4096, 1000)),
                         ('relu2', nn.ReLU()),
                         ('dropout2', nn.Dropout(.25)),
                         ('fc3', nn.Linear(1000, 102)),
                         ('output', nn.LogSoftmax(dim=1))
                         ]))

model.classifier = classifier
print(model)

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=False)
    (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=False)
    (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=False)
    (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=False)
    (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=False)
  )
  (classifier): Sequential(
    (fc1): Linear(in_features=25088, out_features=4096,
bias=True)
    (relu1): ReLU()
    (dropout1): Dropout(p=0.25)
    (fc2): Linear(in_features=4096, out_features=1000,
bias=True)
```

```
    (relu2): ReLU()
    (dropout2): Dropout(p=0.25)
    (fc3): Linear(in_features=1000, out_features=102, bias=True)
    (output): LogSoftmax()
  )
)

## REF notes from section 6 on training the network
#* Make a forward pass through the network,  using softmax here
so be sure to apply exponent code at at validation.
#*Perform a backward pass through the network with
loss.backward() to calculate the gradients.
#* Take a step with the optimizer to update the weights.

#*By adjusting the hyperparameters (hidden units, learning rate,
etc), you should be able
#* to get the training loss below 0.4.

## Did not get results wanted so.....
##   Revised dropout from .5 to .25, included model.train() for
trainloader data
##   and model.eval for validloader since used dropout.
##   Added ps = torch.exp(outputs) to validation and testing
loops.

##  Per InferenceVal/7: Since employing log-softmax, used the
negative log loss as my criterion, nn.NLLLoss().
criterion = nn.NLLLoss()
##orignal optimizer, optim.SGD(model.parameters(), lr=0.01), not
preferred for vgg16.
optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)
## Tried 0.01, 0.001 better

                                                  ##### not
needed, used to confirm *** do not run.

print("Our model: \n\n", model, '\n')
print("The state dict keys: \n\n", model.state_dict().keys())

Our model:

 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=False)
    (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=False)
    (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=False)
    (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=False)
    (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=False)
```

```
  )
  (classifier): Sequential(
    (fc1): Linear(in_features=25088, out_features=4096,
bias=True)
    (relu1): ReLU()
    (dropout1): Dropout(p=0.25)
    (fc2): Linear(in_features=4096, out_features=1000,
bias=True)
    (relu2): ReLU()
    (dropout2): Dropout(p=0.25)
    (fc3): Linear(in_features=1000, out_features=102, bias=True)
    (output): LogSoftmax()
  )
)

The state dict keys:

 odict_keys(['features.0.weight', 'features.0.bias', 'features.
2.weight', 'features.2.bias', 'features.5.weight', 'features.
5.bias', 'features.7.weight', 'features.7.bias', 'features.
10.weight', 'features.10.bias', 'features.12.weight', 'features.
12.bias', 'features.14.weight', 'features.14.bias', 'features.
17.weight', 'features.17.bias', 'features.19.weight', 'features.
19.bias', 'features.21.weight', 'features.21.bias', 'features.
24.weight', 'features.24.bias', 'features.26.weight', 'features.
26.bias', 'features.28.weight', 'features.28.bias',
'classifier.fc1.weight', 'classifier.fc1.bias',
'classifier.fc2.weight', 'classifier.fc2.bias',
'classifier.fc3.weight', 'classifier.fc3.bias'])

# Use loop; attempt to create a function failed

device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")     ###### training, start  ******
print('Starting training process. Data in trainloader.')

epochs = 3
print_every = 40
steps = 0

model.to('cuda')

for e in range(epochs):
    running_loss = 0

    # using trainloader data: put in training mode
    model.train()

    for ii, (inputs, labels) in enumerate(trainloader):
        steps += 1
        inputs, labels = inputs.to('cuda'), labels.to('cuda')
        optimizer.zero_grad()

        # Forward and backward passes
        outputs = model.forward(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        if steps % print_every == 0:
             print("Epoch: {}/{}... ".format(e+1, epochs),
                          "Loss: {:.4f} Trainloader loop
".format(running_loss/print_every))


        running_loss = 0

print('Completed: Training with model.train.')

Starting training process. Data in trainloader.
Epoch: 1/3...  Loss: 0.0712 Trainloader loop
Epoch: 1/3...  Loss: 0.0492 Trainloader loop
Epoch: 2/3...  Loss: 0.0346 Trainloader loop
Epoch: 2/3...  Loss: 0.0213 Trainloader loop
Epoch: 2/3...  Loss: 0.0430 Trainloader loop
Epoch: 3/3...  Loss: 0.0279 Trainloader loop
Epoch: 3/3...  Loss: 0.0344 Trainloader loop
Completed: Training with model.train.


#### Validation loop

print('Start:  validloader loop.')

correct = 0
total = 0
device = torch.device("cuda:0" if torch.cuda.is_available() else
'cpu')
model.to('cuda')
with torch.no_grad():
    model.eval()
    for data in validloader:
        inputs,labels = data
        inputs, labels = inputs.to('cuda'), labels.to('cuda')
        outputs = model(inputs)
        ps = torch.exp(outputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Number: validation/test images:
{}'.format(len(validloader.dataset)))
print('Accuracy of the network on test images: %d %%' % (100 *
correct / total))
print('')
print('Completed: Accuracy loop for data in validloader.')

Start:  validloader loop.
Number: validation/test images: 819
Accuracy of the network on test images: 81 %

Completed: Accuracy loop for data in validloader.

#* Following: validation on the test set

##**  Here using the 13 aug accuracy loop to complete the
training\validation\testing    ###### test phase ******
# Using testloader data
# Using accuracy loop ; attempt to create function failed
print('Starting accuracy on testloader set.')
correct = 0
total = 0

model.to('cuda')
with torch.no_grad():
     model.eval()
     for inputs, labels in testloader: #for data in testloader:
        inputs, labels = inputs.to('cuda'), labels.to('cuda')
        outputs = model(inputs)
        ps = torch.exp(outputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on test images: %d %%' % (100 *
correct / total))
```

```
print('')
print('Completed: Accuracy loop for data in testloader set.')
```

```
Starting accuracy on testloader set.
Accuracy of the network on test images: 81 %

Completed: Accuracy loop for data in testloader set.
```

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.

```
##  Training & testing network complete. GPU at 40hrs, 56min
##  Reworking of training/valid./testing ended at 38hr, 21 min
(however time spent also on unproductive image sections)

##  Comments...
##  Failed to get success when implemented functions ref8/7.
Error persisted: 'input has less dimensions than expected'.
##  Lost GPU time trying solutions. My initial attempts are
implemented in this program.
##  Therefore stuck with straightforward coding which was first
attempt.  Loss and accuracy acceptable!

##** End of 14 aug revision which addresses using dropout,
model.train and model.eval,
##  implementing validation sequence, failing to implement
functions & reverted to
#   straightforward coding.  Loss and accuracy acceptable!
```

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.

```
# Following: Save the checkpoint

                                                                    ####
checkpoint, create *****

print('Start: save checkpoint.')


model.class_to_idx = train_data.class_to_idx

checkpoint = {'model': models.vgg16(pretrained=True),
              'epochs': 3,
              'input_size': 25088,
              'output_size': 102,
              'learn_rate': 0.001,
              'hidden_layers': [19,992],
             #classifier': model.classifier,      ## Removed to
conserve file size
              'state_dict': model.state_dict(),
              'optimizer': optimizer.state_dict(),
              'class_to_idx': model.class_to_idx
            }

torch.save(checkpoint, 'checkpoint.pth')
print('Completed: save checkpoint.')

Start: save checkpoint.
Completed: save checkpoint.
```

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.

```
print('Start: load checkpoint.')
#### checkpoint, load *****
```

```
def load_checkpoint(filepath):
    checkpoint = torch.load(filepath)
    model.classifer = classifier
    model = models.vgg16(pretrained=True)
    model.class_to_idx = checkpoint('class to idx')

    return model

print(model)
#### checkpoint, test *****
print('')
print('Completed: load checkpoint.')

Start: load checkpoint.
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=False)
    (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=False)
    (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=False)
    (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=False)
    (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=False)
  )
  (classifier): Sequential(
    (fc1): Linear(in_features=25088, out_features=4096,
bias=True)
    (relu1): ReLU()
    (dropout1): Dropout(p=0.25)
    (fc2): Linear(in_features=4096, out_features=1000,
bias=True)
    (relu2): ReLU()
    (dropout2): Dropout(p=0.25)
    (fc3): Linear(in_features=1000, out_features=102, bias=True)
    (output): LogSoftmax()
  )
)

Completed: load checkpoint.

## Checkpoint concluded.
#### checkpoint, concluded *****
```

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.

# #

Following cells are prelim work on resizing the original image to 256 on a side to 224 crop

```
from PIL import Image
# this cell: original image
```

```
image_path ='/home/workspace/aipnd-project/flowers/test/10/
image_07090.jpg'
flw_image = Image.open(image_path)
flw_image
```

```
                                #### this cell resizes from
original to flw_resized 256ht img
#image_path ='/home/workspace/aipnd-project/flowers/test/10/
image_07090.jpg'
from PIL import Image
flw_img = Image.open('./flowers/test/10/image_07090.jpg')

width, height = flw_img.size
print("For Flw_img:  Width:", width,"Height:",height)

ratio =  614 /500     #height/width
res_width = 256 * ratio
```

```
flw_resized = flw_img.resize( ( (int(res_width)),
256) ,Image.ANTIALIAS)
flw_resized  ## aok shape


width, height = flw_resized.size
print("For Flw_resized:  Width:", width,"Height:",height)
flw_resized
```

```
For Flw_img:  Width: 614 Height: 500
For Flw_resized:  Width: 314 Height: 256
```

```
from PIL import Image                                         ####
this cell resizes from ori to flw_resized to cropImg
image_path ='/home/workspace/aipnd-project/flowers/test/10/
image_07090.jpg'
flw_img = Image.open(image_path)

width, height = flw_img.size
print("For Flw_img:  Width:", width,"Height:",height)
```

```
ratio =  width / height     # ratio needed for resize
res_height = 256 * ratio

flw_resized = flw_img.resize( ( (int(res_height)),
256) ,Image.ANTIALIAS)
flw_resized  ## aok shape
##ref for resize: https://stackoverflow.com/questions/29367990/
#what-is-the-difference-between-image-resize-and-image-
thumbnail-in-pillow-python

width, height = flw_resized.size
print("For Flw_resized:  Width:", width,"Height:",height)
flw_resized

## crop happens next

cropW = 112
cropH = 112

left = (width/2 - cropW)
top = (height/2 - cropH)
right = (width/2 + cropW)
bottom = (height/2 + cropH)
print(left, top, right, bottom)

cropImg = flw_resized.crop((left, top, right, bottom))

width, height = cropImg.size
print("For cropImg:  Width:", width,"Height:",height)
cropImg
```

```
For Flw_img:  Width: 614 Height: 500
For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
```

```
##############################   ABOVE CELLS: resized the
original image to 256 on a side to 224 crop
```

Process_Image, Imshow

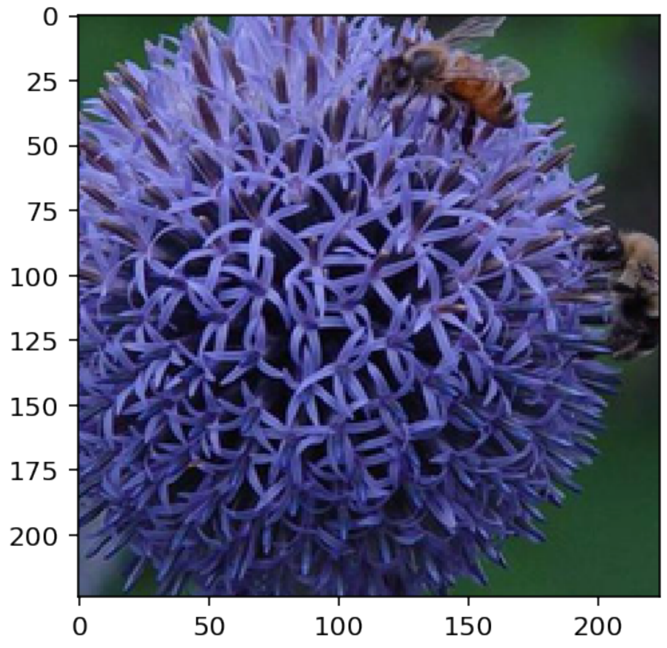
###################### FOLLOWING: process_image and imshow functions and predict function

```
####FINAL .................................................
......................................

########                                              ######
FINAL ....................
################################### ..................  works
25aug FINAL process_image


print('Start: duplicate process_image fct.')           ## ****
process_image function FINAL

def process_image(image):                              ##
Function resizes PIL imported image, creates numpy array

    from PIL import Image

    image_path ='/home/workspace/aipnd-project/flowers/test/10/
image_07090.jpg'
```

```
    flw_img = Image.open(image_path)

    width, height = flw_img.size
   # print("For Flw_img:  Width:", width,"Height:",height)

    ratio =  width/height
    res_width = 256 * ratio

    flw_resized = flw_img.resize( ( (int(res_width)),
256) ,Image.ANTIALIAS)
    flw_resized  ## aok shape

    width, height = flw_resized.size
    print("For Flw_resized:  Width:", width,"Height:",height)
## Confirm size

## Crop: compute measurements
    cropW = 112
    cropH = 112

    left = (width/2 - cropW)
    top = (height/2 - cropH)
    right = (width/2 + cropW)
    bottom = (height/2 + cropH)
    print(left, top, right, bottom)

    flw_img = flw_resized.crop((left, top, right, bottom))
    width, height = flw_img.size
## Confirm size
    print("For cropImg:  Width:", width,"Height:",height)
    print('cropped image:', flw_img )
## Confirm shape by returning neew flw_img

    np_img = np.array(flw_img)/255

## Convert values, use np.array
    np_img

    print('np image:', np_img )
    mean = np.array([0.485,0.456,0.406])
    mean
    print('mean', mean)
    std = np.array([0.229, 0.224, 0.225])
    std
    print('std', std)
    np_img = ( np_img - (mean)) / (std)
## Normalize image

    np_img = np_img.transpose(2,0,1)
## Set image channels


    return np_img

#print('np_img',np_img)
print('End:  process_image fct. FINAL')

process_image('./flowers/test/10/image_07090.jpg')

Start: duplicate process_image fct.
End:  process_image fct. FINAL
For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
cropped image: <PIL.Image.Image image mode=RGB size=224x224 at
0x7F6B64258630>
np image: [[[ 0.13333333  0.27843137  0.14901961]
  [ 0.13333333  0.27843137  0.14901961]
  [ 0.1372549   0.28235294  0.15294118]
  ...,
  [ 0.13333333  0.22352941  0.15294118]
  [ 0.1372549   0.22745098  0.15686275]
  [ 0.1372549   0.22745098  0.16078431]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  ...,
  [ 0.13333333  0.21176471  0.14509804]
  [ 0.1372549   0.21568627  0.14901961]
  [ 0.1372549   0.21568627  0.15294118]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14509804  0.29019608  0.16078431]
  [ 0.14901961  0.29411765  0.16470588]
  ...,
  [ 0.1254902   0.20784314  0.14901961]
  [ 0.1254902   0.21176471  0.14509804]
  [ 0.12941176  0.21176471  0.15294118]]

 ...,
 [[ 0.31764706  0.36862745  0.47058824]
  [ 0.31764706  0.36470588  0.47843137]
  [ 0.31764706  0.35686275  0.48235294]
  ...,
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15686275  0.30196078  0.2       ]]

 [[ 0.31764706  0.36470588  0.44313725]
  [ 0.32156863  0.36470588  0.45098039]
  [ 0.31764706  0.35294118  0.45882353]
  ...,
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]]

 [[ 0.30196078  0.35686275  0.42745098]
  [ 0.30588235  0.35294118  0.43921569]
  [ 0.30196078  0.34509804  0.44313725]
  ...,
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]]]
mean [ 0.485  0.456  0.406]
std [ 0.229  0.224  0.225]

array([[[-1.5356623 , -1.5356623 , -1.51853755, ...,
-1.5356623 ,
         -1.51853755, -1.51853755],
        [-1.50141279, -1.50141279, -1.50141279, ...,
-1.5356623 ,
         -1.51853755, -1.51853755],
        [-1.50141279, -1.48428804, -1.46716328, ...,
-1.56991181,
         -1.56991181, -1.55278705],
        ...,
        [-0.73079887, -0.73079887, -0.73079887, ...,
-1.45003853,
         -1.45003853, -1.43291378],
        [-0.73079887, -0.71367412, -0.73079887, ...,
-1.46716328,
         -1.46716328, -1.46716328],
        [-0.79929789, -0.78217313, -0.79929789, ...,
-1.45003853,
         -1.45003853, -1.45003853]],
       [[-0.79271709, -0.79271709, -0.77521008, ...,
-1.03781513,
         -1.02030812, -1.02030812],
        [-0.75770308, -0.75770308, -0.75770308, ...,
-1.09033613,
         -1.07282913, -1.07282913],
        [-0.75770308, -0.74019608, -0.72268908, ...,
-1.10784314,
         -1.09033613, -1.09033613],
        ...,
        [-0.39005602, -0.40756303, -0.44257703, ...,
-0.70518207,
         -0.70518207, -0.68767507],
        [-0.40756303, -0.40756303, -0.46008403, ...,
-0.68767507,
         -0.68767507, -0.68767507],
        [-0.44257703, -0.46008403, -0.49509804, ...,
-0.67016807,
         -0.67016807, -0.67016807]],

       [[-1.14213508, -1.14213508, -1.12470588, ...,
-1.12470588,
         -1.10727669, -1.08984749],
        [-1.10727669, -1.10727669, -1.10727669, ...,
-1.15956427,
         -1.14213508, -1.12470588],
        [-1.10727669, -1.08984749, -1.0724183 , ...,
-1.14213508,
         -1.15956427, -1.12470588],
        ...,
        [ 0.28705882,  0.32191721,  0.33934641, ...,
-0.95041394,
         -0.95041394, -0.91555556],
        [ 0.16505447,  0.19991285,  0.23477124, ...,
-0.93298475,
         -0.93298475, -0.93298475],
        [ 0.09533769,  0.14762527,  0.16505447, ...,
-0.91555556,
         -0.91555556, -0.91555556]]])

 ################################### $$$$$$$$$$$works$$$$$$$$$
$$$$$$$$$$$$  works 25aug


#### 25aug FINAL imshow  ....................
print('Start imshow function.')
```

```
###################################
image = (process_image('./flowers/test/10/image_07090.jpg'))
##############
torch_image = torch.from_numpy(image)
## Create tensor image
torch_image
print('tensor image', torch_image)

print('Start: image imshow fct.')
def imshow(image, ax=None, title=None):

    if ax is None:
        fig, ax = plt.subplots()
    if ax is None:
        fig, ax = plt.subplots()

    # PyTorch tensors assume the color channel is the first
dimension; use transpose as
    # .... matplotlib assumes is the third dimension
    torch_image = process_image('./flowers/test/10/
image_07090.jpg')
    np_img = torch.from_numpy(torch_image)
## From array to tensor
    np_img = np_img.numpy()
    np_img = np_img.transpose(1,2,0)
## Transpose np_img dimensions

    #print('np shape', np_img)  ##  Use for testing

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

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

    ax.imshow(np_img)
    return ax


###################################
print('Completed: image imshow fct.   FINAL')

imshow(image)
```

```
Start imshow function.
For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
cropped image: <PIL.Image.Image image mode=RGB size=224x224 at
0x7F6B43409908>
np image: [[[ 0.13333333  0.27843137  0.14901961]
  [ 0.13333333  0.27843137  0.14901961]
  [ 0.1372549   0.28235294  0.15294118]
  ...,
  [ 0.13333333  0.22352941  0.15294118]
  [ 0.1372549   0.22745098  0.15686275]
  [ 0.1372549   0.22745098  0.16078431]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  ...,
  [ 0.13333333  0.21176471  0.14509804]
  [ 0.1372549   0.21568627  0.14901961]
  [ 0.1372549   0.21568627  0.15294118]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14509804  0.29019608  0.16078431]
  [ 0.14901961  0.29411765  0.16470588]
  ...,
  [ 0.1254902   0.20784314  0.14901961]
  [ 0.1254902   0.21176471  0.14509804]
  [ 0.12941176  0.21176471  0.15294118]]

 ...,
 [[ 0.31764706  0.36862745  0.47058824]
  [ 0.31764706  0.36470588  0.47843137]
  [ 0.31764706  0.35686275  0.48235294]
  ...,
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15686275  0.30196078  0.2       ]]

 [[ 0.31764706  0.36470588  0.44313725]
  [ 0.32156863  0.36470588  0.45098039]
  [ 0.31764706  0.35294118  0.45882353]
  ...,
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]]

 [[ 0.30196078  0.35686275  0.42745098]
  [ 0.30588235  0.35294118  0.43921569]
  [ 0.30196078  0.34509804  0.44313725]
  ...,
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]]]
mean [ 0.485  0.456  0.406]
std [ 0.229  0.224  0.225]
tensor image tensor([[[-1.5357, -1.5357, -1.5185,  ..., -1.5357,
-1.5185, -1.5185],
         [-1.5014, -1.5014, -1.5014,  ..., -1.5357, -1.5185,
-1.5185],
         [-1.5014, -1.4843, -1.4672,  ..., -1.5699, -1.5699,
-1.5528],
         ...,
         [-0.7308, -0.7308, -0.7308,  ..., -1.4500, -1.4500,
-1.4329],
         [-0.7308, -0.7137, -0.7308,  ..., -1.4672, -1.4672,
-1.4672],
         [-0.7993, -0.7822, -0.7993,  ..., -1.4500, -1.4500,
-1.4500]],

        [[-0.7927, -0.7927, -0.7752,  ..., -1.0378, -1.0203,
-1.0203],
         [-0.7577, -0.7577, -0.7577,  ..., -1.0903, -1.0728,
-1.0728],
         [-0.7577, -0.7402, -0.7227,  ..., -1.1078, -1.0903,
-1.0903],
         ...,
         [-0.3901, -0.4076, -0.4426,  ..., -0.7052, -0.7052,
-0.6877],
         [-0.4076, -0.4076, -0.4601,  ..., -0.6877, -0.6877,
-0.6877],
         [-0.4426, -0.4601, -0.4951,  ..., -0.6702, -0.6702,
-0.6702]],

        [[-1.1421, -1.1421, -1.1247,  ..., -1.1247, -1.1073,
-1.0898],
         [-1.1073, -1.1073, -1.1073,  ..., -1.1596, -1.1421,
-1.1247],
         [-1.1073, -1.0898, -1.0724,  ..., -1.1421, -1.1596,
-1.1247],
         ...,
         [ 0.2871,  0.3219,  0.3393,  ..., -0.9504, -0.9504,
-0.9156],
         [ 0.1651,  0.1999,  0.2348,  ..., -0.9330, -0.9330,
-0.9330],
         [ 0.0953,  0.1476,  0.1651,  ..., -0.9156, -0.9156,
-0.9156]]], dtype=torch.float64)
Start: image imshow fct.
Completed: image imshow fct.   FINAL
For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
cropped image: <PIL.Image.Image image mode=RGB size=224x224 at
0x7F6B43428940>
np image: [[[ 0.13333333  0.27843137  0.14901961]
  [ 0.13333333  0.27843137  0.14901961]
  [ 0.1372549   0.28235294  0.15294118]
  ...,
  [ 0.13333333  0.22352941  0.15294118]
  [ 0.1372549   0.22745098  0.15686275]
  [ 0.1372549   0.22745098  0.16078431]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  ...,
  [ 0.13333333  0.21176471  0.14509804]
  [ 0.1372549   0.21568627  0.14901961]
  [ 0.1372549   0.21568627  0.15294118]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14509804  0.29019608  0.16078431]
  [ 0.14901961  0.29411765  0.16470588]
  ...,
  [ 0.1254902   0.20784314  0.14901961]
  [ 0.1254902   0.21176471  0.14509804]
  [ 0.12941176  0.21176471  0.15294118]]


 ...,
 [[ 0.31764706  0.36862745  0.47058824]
  [ 0.31764706  0.36470588  0.47843137]
  [ 0.31764706  0.35686275  0.48235294]
  ...,
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15686275  0.30196078  0.2       ]]
 [[ 0.31764706  0.36470588  0.44313725]
  [ 0.32156863  0.36470588  0.45098039]
  [ 0.31764706  0.35294118  0.45882353]
  ...,
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]]

 [[ 0.30196078  0.35686275  0.42745098]
  [ 0.30588235  0.35294118  0.43921569]
  [ 0.30196078  0.34509804  0.44313725]
  ...,
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]]]
mean [ 0.485  0.456  0.406]
std [ 0.229  0.224  0.225]

<matplotlib.axes._subplots.AxesSubplot at 0x7f6b43402978>
```



Class Prediction

```
####^^^^^^^^^^^*** 28aug version  == 3rd test WORKS  FINAL
def predict(image, model, topk=5):
    ''' Predict the class (or classes) of an image using a
trained deep learning model.
    '''

    image = torch.from_numpy(process_image(image))
    image = image.unsqueeze(0).float()

    model = model.eval()
   )
    model = model.to('cpu')


    ## Create tensor image
    ## Transfer image to tensor

    with torch.no_grad():
        output = model.forward(image)
        results = torch.exp(output).topk(topk)

    # Calculate the class probabilities for img
    probs, classes = results[0].data.cpu().numpy()[0],
results[1].data.cpu().numpy()[0]

    idx_to_class = {val: key for key, val in
model.class_to_idx.items()}
    ########..........................................
## Convert indices to classes
    classes =  [idx_to_class[classes] for classes in classes]

    return probs, classes


probs, classes = predict(image, model)
print(probs)
print(classes)


print('End 11pm 28 aug predict function.')

For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
cropped image: <PIL.Image.Image image mode=RGB size=224x224 at
0x7F6B434A6A20>
np image: [[[ 0.13333333  0.27843137  0.14901961]
  [ 0.13333333  0.27843137  0.14901961]
  [ 0.1372549   0.28235294  0.15294118]
  ...,
  [ 0.13333333  0.22352941  0.15294118]
  [ 0.1372549   0.22745098  0.15686275]
  [ 0.1372549   0.22745098  0.16078431]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  ...,
  [ 0.13333333  0.21176471  0.14509804]
  [ 0.1372549   0.21568627  0.14901961]
  [ 0.1372549   0.21568627  0.15294118]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14509804  0.29019608  0.16078431]
  [ 0.14901961  0.29411765  0.16470588]
  ...,
  [ 0.1254902   0.20784314  0.14901961]
  [ 0.1254902   0.21176471  0.14509804]
  [ 0.12941176  0.21176471  0.15294118]]

 ...,
 [[ 0.31764706  0.36862745  0.47058824]
  [ 0.31764706  0.36470588  0.47843137]
  [ 0.31764706  0.35686275  0.48235294]
  ...,
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15686275  0.30196078  0.2       ]]

 [[ 0.31764706  0.36470588  0.44313725]
  [ 0.32156863  0.36470588  0.45098039]
  [ 0.31764706  0.35294118  0.45882353]
  ...,
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]]

 [[ 0.30196078  0.35686275  0.42745098]
  [ 0.30588235  0.35294118  0.43921569]
  [ 0.30196078  0.34509804  0.44313725]
  ...,
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]]]
```

```
mean [ 0.485  0.456  0.406]
std [ 0.229  0.224  0.225]
[ 0.01327967  0.01229576  0.01195177  0.01192604  0.01185571]
['27', '87', '37', '102', '99']
End 11pm 28 aug predict function.
```

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).

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']
```

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.

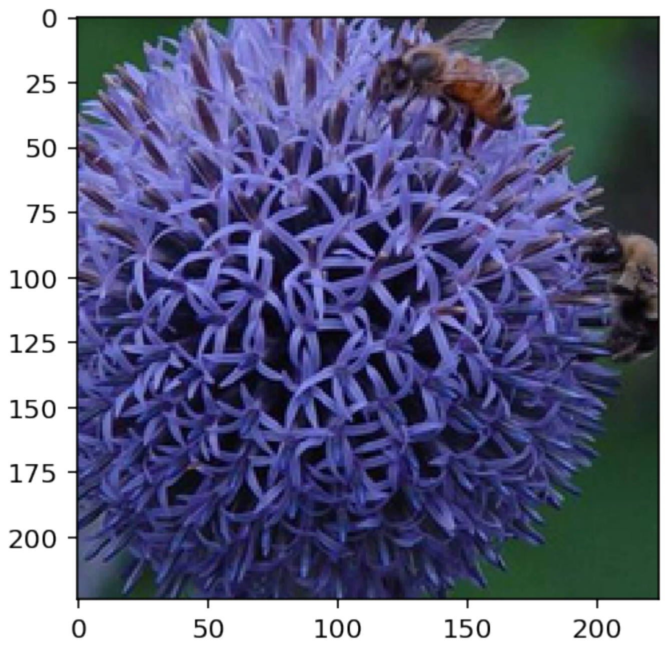
```
# TODO: Display an image along with the top 5 classes

# plot the probabilities for the top 5 classes as a bar graph,
along with the input image

print('Start Sanity Test Function')


def sanity_test(image, model):                              #28 aug
FINAL Sanity Test............
    from PIL import Image


    # Show INPUT image
    image = imshow(process_image(image) )   #('./flowers/test/
10/image_07090.jpg'))


    # Predict topk flowers
    probs, classes = predict( image, model) # Call predict
function
    print('Probabilities',probs)
    print('Classes', classes)

    idx_to_class = {val: key for key, val in         ## Swap
keys in index

model.class_to_idx.items()}
    image = [cat_to_name[lab] for lab in classes]    ## Match
flower img/name
```

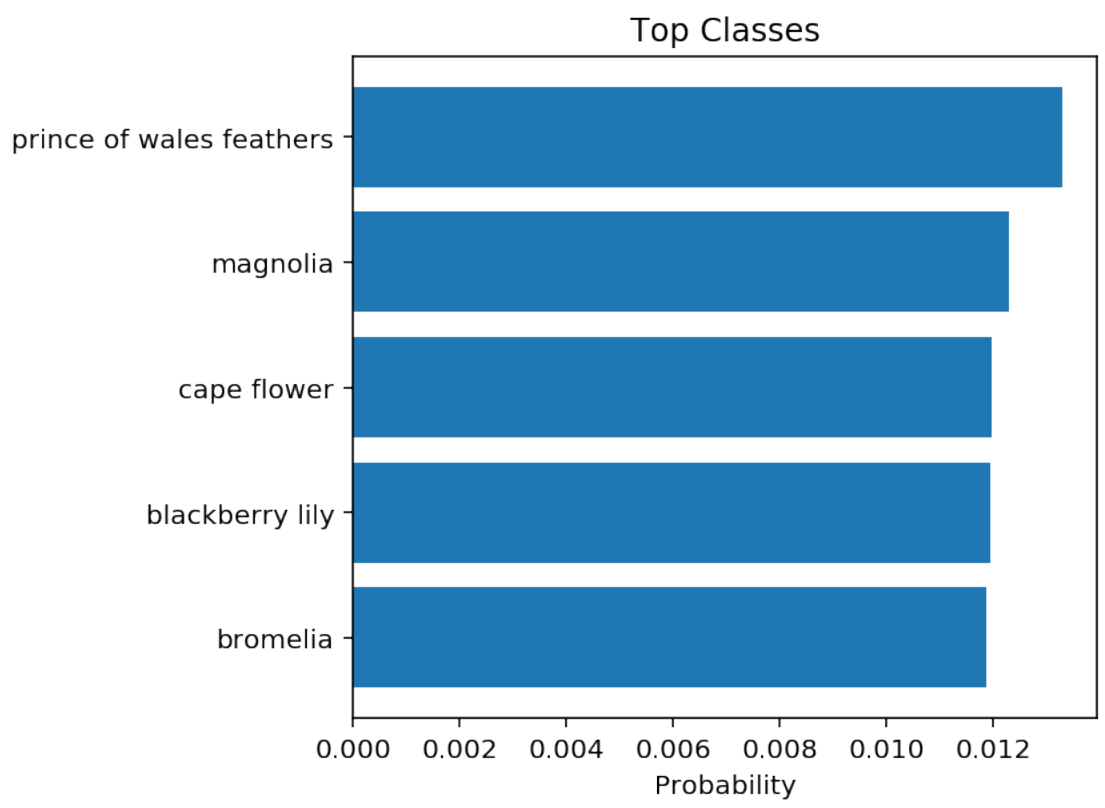
```
    topk = [cat_to_name[i] for i in classes]

    # Create image bar graph for top flowers
    plt.figure(figsize = (5,10))

    ax = plt.subplot(2,1,2)
    y_pos = np.arange(len(topk))

    ax.barh(y_pos, probs)
    ax.set_yticks(y_pos)
    ax.set_yticklabels(topk)
    ax.invert_yaxis()
    ax.set_xlabel('Probability')
    ax.set_title('Top Classes')

    plt.savefig('top_classes')
    plt.show()

    return image, model

sanity_test(image, model)

print('End Sanity Test Function')

Start Sanity Test Function
For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
cropped image: <PIL.Image.Image image mode=RGB size=224x224 at
0x7F6B405ABC50>
np image: [[[ 0.13333333  0.27843137  0.14901961]
  [ 0.13333333  0.27843137  0.14901961]
  [ 0.1372549   0.28235294  0.15294118]
  ...,
  [ 0.13333333  0.22352941  0.15294118]
  [ 0.1372549   0.22745098  0.15686275]
  [ 0.1372549   0.22745098  0.16078431]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  ...,
  [ 0.13333333  0.21176471  0.14509804]
  [ 0.1372549   0.21568627  0.14901961]
  [ 0.1372549   0.21568627  0.15294118]]
 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14509804  0.29019608  0.16078431]
  [ 0.14901961  0.29411765  0.16470588]
  ...,
  [ 0.1254902   0.20784314  0.14901961]
  [ 0.1254902   0.21176471  0.14509804]
  [ 0.12941176  0.21176471  0.15294118]]

 ...,
 [[ 0.31764706  0.36862745  0.47058824]
  [ 0.31764706  0.36470588  0.47843137]
  [ 0.31764706  0.35686275  0.48235294]
  ...,
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15686275  0.30196078  0.2       ]]

 [[ 0.31764706  0.36470588  0.44313725]
  [ 0.32156863  0.36470588  0.45098039]
  [ 0.31764706  0.35294118  0.45882353]
  ...,
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]]

 [[ 0.30196078  0.35686275  0.42745098]
  [ 0.30588235  0.35294118  0.43921569]
  [ 0.30196078  0.34509804  0.44313725]
  ...,
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]]]
mean [ 0.485  0.456  0.406]
std [ 0.229  0.224  0.225]
For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
cropped image: <PIL.Image.Image image mode=RGB size=224x224 at
0x7F6B60808630>
np image: [[[ 0.13333333  0.27843137  0.14901961]
  [ 0.13333333  0.27843137  0.14901961]
  [ 0.1372549   0.28235294  0.15294118]
  ...,
  [ 0.13333333  0.22352941  0.15294118]
  [ 0.1372549   0.22745098  0.15686275]
  [ 0.1372549   0.22745098  0.16078431]]
```

```
 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  ...,
  [ 0.13333333  0.21176471  0.14509804]
  [ 0.1372549   0.21568627  0.14901961]
  [ 0.1372549   0.21568627  0.15294118]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14509804  0.29019608  0.16078431]
  [ 0.14901961  0.29411765  0.16470588]
  ...,
  [ 0.1254902   0.20784314  0.14901961]
  [ 0.1254902   0.21176471  0.14509804]
  [ 0.12941176  0.21176471  0.15294118]]

 ...,
 [[ 0.31764706  0.36862745  0.47058824]
  [ 0.31764706  0.36470588  0.47843137]
  [ 0.31764706  0.35686275  0.48235294]
  ...,
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15686275  0.30196078  0.2       ]]

 [[ 0.31764706  0.36470588  0.44313725]
  [ 0.32156863  0.36470588  0.45098039]
  [ 0.31764706  0.35294118  0.45882353]
  ...,
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]]

 [[ 0.30196078  0.35686275  0.42745098]
  [ 0.30588235  0.35294118  0.43921569]
  [ 0.30196078  0.34509804  0.44313725]
  ...,
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]
  [ 0.15294118  0.30588235  0.2       ]]]
mean [ 0.485  0.456  0.406]
std [ 0.229  0.224  0.225]
For Flw_resized:  Width: 314 Height: 256
45.0 16.0 269.0 240.0
For cropImg:  Width: 224 Height: 224
```

```
cropped image: <PIL.Image.Image image mode=RGB size=224x224 at
0x7F6B60808BE0>
np image: [[[ 0.13333333  0.27843137  0.14901961]
  [ 0.13333333  0.27843137  0.14901961]
  [ 0.1372549   0.28235294  0.15294118]
  ...,
  [ 0.13333333  0.22352941  0.15294118]
  [ 0.1372549   0.22745098  0.15686275]
  [ 0.1372549   0.22745098  0.16078431]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  [ 0.14117647  0.28627451  0.15686275]
  ...,
  [ 0.13333333  0.21176471  0.14509804]
  [ 0.1372549   0.21568627  0.14901961]
  [ 0.1372549   0.21568627  0.15294118]]

 [[ 0.14117647  0.28627451  0.15686275]
  [ 0.14509804  0.29019608  0.16078431]
  [ 0.14901961  0.29411765  0.16470588]
  ...,
  [ 0.1254902   0.20784314  0.14901961]
  [ 0.1254902   0.21176471  0.14509804]
  [ 0.12941176  0.21176471  0.15294118]]

 ...,
 [[ 0.31764706  0.36862745  0.47058824]
  [ 0.31764706  0.36470588  0.47843137]
  [ 0.31764706  0.35686275  0.48235294]
  ...,
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15294118  0.29803922  0.19215686]
  [ 0.15686275  0.30196078  0.2       ]]

 [[ 0.31764706  0.36470588  0.44313725]
  [ 0.32156863  0.36470588  0.45098039]
  [ 0.31764706  0.35294118  0.45882353]
  ...,
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]
  [ 0.14901961  0.30196078  0.19607843]]

 [[ 0.30196078  0.35686275  0.42745098]
  [ 0.30588235  0.35294118  0.43921569]
  [ 0.30196078  0.34509804  0.44313725]
```

```
   ...,
   [ 0.15294118  0.30588235  0.2        ]
   [ 0.15294118  0.30588235  0.2        ]
   [ 0.15294118  0.30588235  0.2        ]]]
mean [ 0.485  0.456  0.406]
std [ 0.229  0.224  0.225]
Probabilities [ 0.01327967  0.01229576  0.01195177  0.01192604
0.01185571]
Classes ['27', '87', '37', '102', '99']
```





```
End Sanity Test Function

####### Following cell includes references for Part 1
process_image, imshow, predict, sanity check.
###... as well as for Part 2.

####### Following includes references for Part 1 process_image,
imshow, predict, sanity check.
###... as well as for Part 2.

##References used for this project include conversations on
various relevant aspects
##  of the project that were posted on the Udacity Slack
channel.
##  Of note: direct feedback from mentors @Ortal and @Partha,
and from Slack contacts
##  Ishan Mishra, Pramod Geddam Saravanan TK, Majd Barchini.
##  Relevant Slack feedback given others that were helpful to me
included replies by Mat (instructor),
##  Abdel Affo, Audrey Tan,

## REFERENCES, in addition to above, for work on process_image
and imshow functions
```

```
## ref: Image.thumbnail: http://pillow.readthedocs.org/en/
latest/reference/Image.html#PIL.Image.
#       Image.thumbnail
## ref: https://medium.com/@josh_2774/deep-learning-with-
pytorch-9574e74d17ad
#       scaling to 255: img color channels int to expected float
## ref: https://pillow.readthedocs.io/en/3.1.x/search.html?
q=image.thumbnail&
#        check_keywords=yes&area=default
#        Image Module
## ref:  https://stackoverflow.com/questions/16646183/crop-an-
image-in-the-centre-using-pil
## ref for resize: https://stackoverflow.com/questions/29367990/
#       what-is-the-difference-between-image-resize-and-image-
thumbnail-in-pillow-python

## REFERENCES for work on Part 2,in addition to above
# ref: for argparse -- Python Software Foundation(US)https://
docs.python.org/2/howto/argparse.html
#                  -- Stackexchange posts relating to argparse
#                  -- YouTube videos introducing argparse
```