# Image Classifier Project

July 14, 2019

## 1 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.

```
In [35]: # 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 workspace_utils import keep_awake
from workspace_utils import active_session
from PIL import Image
import json
from matplotlib.ticker import FormatStrFormatter
```

### 1.1 Load the data

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

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

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

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

In [3]:
```
# TODO: Define your transforms for the training, validation, and testing sets
data_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])])

validation_transforms = transforms.Compose([transforms.Resize(255)
                                            ,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(255)
                                      ,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 = datasets.ImageFolder(train_dir, transform=data_transforms)
        validation_datasets = datasets.ImageFolder(valid_dir, transform=validation_transforms)
        test_datasets = datasets.ImageFolder(test_dir, transform=test_transforms)

        # TODO: Using the image datasets and the trainforms, define the dataloaders
        dataloaders = torch.utils.data.DataLoader(image_datasets, batch_size=64, shuffle=True)
        validationloaders = torch.utils.data.DataLoader(validation_datasets, batch_size=64)
        testloaders = torch.utils.data.DataLoader(test_datasets, batch_size=64)
```

### 1.1.1 Label mapping

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

```
In [4]: import json

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

## 2 Building and training the classifier

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

We're going to leave this part up to you. Refer to the rubric for guidance on successfully completing this section. Things you'll need to do:

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

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

When training make sure you're updating only the weights of the feed-forward network. You should be able to get the validation accuracy above 70% if you build everything right. Make sure

to try different hyperparameters (learning rate, units in the classifier, epochs, etc) to find the best model. Save those hyperparameters to use as default values in the next part of the project.

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

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

In [5]:
```
# TODO: Build and train your network
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [6]:
```
model = models.vgg19(pretrained=True)
# model
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.torch/models/vgg
100%|| 574673361/574673361 [00:27<00:00, 20993927.63it/s]
```

In [32]:
```
# Free parameters so we don't backdrop through them
def classifier_setup(structure='vgg19', dropout=0.2, fc2=6272, fc3=784, lr=0.001):
    for param in model.parameters():
        param.requires_grad = False

    from collections import OrderedDict
    classifier = nn.Sequential(OrderedDict([
        ('fc1', nn.Linear(25088, fc2))
        ,('relu1', nn.ReLU())
        ,('dropout', nn.Dropout(0.2))
        ,('fc2', nn.Linear(6272, fc3))
        ,('relu2', nn.ReLU())
        ,('dropout', nn.Dropout(0.2))
        ,('fc3', nn.Linear(fc3, 102))
        ,('output', nn.LogSoftmax(dim=1))
    ]))

    model.classifier = classifier
    return model

model = classifier_setup()
```

In [15]:
```
# define loss function
criterion = nn.NLLLoss()

# Only train the classifier parameters, feature parametets are frozen
optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)
```

```
model.to(device);
```

In [22]:
```
# training the network
epochs = 10
steps = 0
running_loss = 0
print_every = 5

with active_session():
    for epoch in range(epochs):
        for ii, (inputs, labels) in enumerate(dataloaders):
            steps += 1
            # Move input and label tensors to the default device
            inputs, labels = inputs.to(device), labels.to(device)

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

            running_loss += loss.item()

            if steps % print_every == 0:
                vloss = 0
                accuracy = 0
                model.eval()

                with torch.no_grad():
                    for ii, (inputs, labels) in enumerate(validationloaders):
                        inputs, labels = inputs.to(device), labels.to(device)
                        logps = model.forward(inputs)
                        batch_loss = criterion(logps, labels)

                        vloss += batch_loss.item()

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

                print(f"Epoch {epoch+1}/{epochs}.. "
                      f"Train loss: {running_loss/print_every:.3f}.. "
                      f"Validation loss: {vloss/len(validationloaders):.3f}.. "
                      f"Validation accuracy: {accuracy/len(validationloaders):.3f}")
```

```
                    running_loss = 0
                    model.train()
```

```
Epoch 1/10.. Train loss: 1.429.. Validation loss: 0.859.. Validation accuracy: 0.764
Epoch 1/10.. Train loss: 1.374.. Validation loss: 0.843.. Validation accuracy: 0.763
Epoch 1/10.. Train loss: 1.320.. Validation loss: 0.835.. Validation accuracy: 0.763
Epoch 1/10.. Train loss: 1.235.. Validation loss: 0.825.. Validation accuracy: 0.778
Epoch 1/10.. Train loss: 1.141.. Validation loss: 0.819.. Validation accuracy: 0.779
Epoch 1/10.. Train loss: 1.422.. Validation loss: 0.814.. Validation accuracy: 0.778
Epoch 1/10.. Train loss: 1.393.. Validation loss: 0.810.. Validation accuracy: 0.777
Epoch 1/10.. Train loss: 1.273.. Validation loss: 0.824.. Validation accuracy: 0.775
Epoch 1/10.. Train loss: 1.527.. Validation loss: 0.823.. Validation accuracy: 0.782
Epoch 1/10.. Train loss: 1.409.. Validation loss: 0.827.. Validation accuracy: 0.793
Epoch 1/10.. Train loss: 1.273.. Validation loss: 0.840.. Validation accuracy: 0.773
Epoch 1/10.. Train loss: 1.367.. Validation loss: 0.827.. Validation accuracy: 0.776
Epoch 1/10.. Train loss: 1.322.. Validation loss: 0.817.. Validation accuracy: 0.777
Epoch 1/10.. Train loss: 1.597.. Validation loss: 0.817.. Validation accuracy: 0.767
Epoch 1/10.. Train loss: 1.515.. Validation loss: 0.816.. Validation accuracy: 0.771
Epoch 1/10.. Train loss: 1.364.. Validation loss: 0.823.. Validation accuracy: 0.772
Epoch 1/10.. Train loss: 1.650.. Validation loss: 0.820.. Validation accuracy: 0.767
Epoch 1/10.. Train loss: 1.408.. Validation loss: 0.802.. Validation accuracy: 0.771
Epoch 1/10.. Train loss: 1.456.. Validation loss: 0.788.. Validation accuracy: 0.789
Epoch 1/10.. Train loss: 1.478.. Validation loss: 0.779.. Validation accuracy: 0.776
Epoch 2/10.. Train loss: 1.285.. Validation loss: 0.781.. Validation accuracy: 0.771
Epoch 2/10.. Train loss: 1.516.. Validation loss: 0.765.. Validation accuracy: 0.780
Epoch 2/10.. Train loss: 1.412.. Validation loss: 0.748.. Validation accuracy: 0.789
Epoch 2/10.. Train loss: 1.140.. Validation loss: 0.740.. Validation accuracy: 0.798
Epoch 2/10.. Train loss: 1.360.. Validation loss: 0.733.. Validation accuracy: 0.802
Epoch 2/10.. Train loss: 1.304.. Validation loss: 0.721.. Validation accuracy: 0.801
Epoch 2/10.. Train loss: 1.352.. Validation loss: 0.710.. Validation accuracy: 0.806
Epoch 2/10.. Train loss: 1.313.. Validation loss: 0.717.. Validation accuracy: 0.806
Epoch 2/10.. Train loss: 1.521.. Validation loss: 0.737.. Validation accuracy: 0.795
Epoch 2/10.. Train loss: 1.106.. Validation loss: 0.754.. Validation accuracy: 0.796
Epoch 2/10.. Train loss: 1.219.. Validation loss: 0.768.. Validation accuracy: 0.782
Epoch 2/10.. Train loss: 1.204.. Validation loss: 0.758.. Validation accuracy: 0.782
Epoch 2/10.. Train loss: 1.125.. Validation loss: 0.738.. Validation accuracy: 0.790
Epoch 2/10.. Train loss: 1.416.. Validation loss: 0.728.. Validation accuracy: 0.794
Epoch 2/10.. Train loss: 1.241.. Validation loss: 0.738.. Validation accuracy: 0.799
Epoch 2/10.. Train loss: 1.230.. Validation loss: 0.727.. Validation accuracy: 0.801
Epoch 2/10.. Train loss: 1.383.. Validation loss: 0.731.. Validation accuracy: 0.800
Epoch 2/10.. Train loss: 1.339.. Validation loss: 0.717.. Validation accuracy: 0.799
Epoch 2/10.. Train loss: 1.321.. Validation loss: 0.712.. Validation accuracy: 0.800
Epoch 2/10.. Train loss: 1.429.. Validation loss: 0.715.. Validation accuracy: 0.809
Epoch 2/10.. Train loss: 1.278.. Validation loss: 0.706.. Validation accuracy: 0.812
Epoch 3/10.. Train loss: 1.367.. Validation loss: 0.704.. Validation accuracy: 0.802
Epoch 3/10.. Train loss: 1.217.. Validation loss: 0.711.. Validation accuracy: 0.803
Epoch 3/10.. Train loss: 1.228.. Validation loss: 0.724.. Validation accuracy: 0.791
Epoch 3/10.. Train loss: 1.213.. Validation loss: 0.723.. Validation accuracy: 0.799
```

```
Epoch 3/10.. Train loss: 1.247.. Validation loss: 0.700.. Validation accuracy: 0.805
Epoch 3/10.. Train loss: 1.174.. Validation loss: 0.683.. Validation accuracy: 0.801
Epoch 3/10.. Train loss: 1.206.. Validation loss: 0.703.. Validation accuracy: 0.805
Epoch 3/10.. Train loss: 1.280.. Validation loss: 0.720.. Validation accuracy: 0.799
Epoch 3/10.. Train loss: 1.343.. Validation loss: 0.707.. Validation accuracy: 0.804
Epoch 3/10.. Train loss: 1.237.. Validation loss: 0.686.. Validation accuracy: 0.824
Epoch 3/10.. Train loss: 1.128.. Validation loss: 0.683.. Validation accuracy: 0.824
Epoch 3/10.. Train loss: 1.186.. Validation loss: 0.683.. Validation accuracy: 0.807
Epoch 3/10.. Train loss: 1.119.. Validation loss: 0.677.. Validation accuracy: 0.816
Epoch 3/10.. Train loss: 1.535.. Validation loss: 0.681.. Validation accuracy: 0.810
Epoch 3/10.. Train loss: 1.169.. Validation loss: 0.703.. Validation accuracy: 0.810
Epoch 3/10.. Train loss: 1.030.. Validation loss: 0.724.. Validation accuracy: 0.792
Epoch 3/10.. Train loss: 1.317.. Validation loss: 0.731.. Validation accuracy: 0.795
Epoch 3/10.. Train loss: 1.322.. Validation loss: 0.710.. Validation accuracy: 0.813
Epoch 3/10.. Train loss: 1.255.. Validation loss: 0.702.. Validation accuracy: 0.816
Epoch 3/10.. Train loss: 0.948.. Validation loss: 0.712.. Validation accuracy: 0.805
Epoch 4/10.. Train loss: 1.330.. Validation loss: 0.701.. Validation accuracy: 0.812
Epoch 4/10.. Train loss: 1.189.. Validation loss: 0.706.. Validation accuracy: 0.805
Epoch 4/10.. Train loss: 1.176.. Validation loss: 0.698.. Validation accuracy: 0.810
Epoch 4/10.. Train loss: 1.041.. Validation loss: 0.703.. Validation accuracy: 0.803
Epoch 4/10.. Train loss: 1.236.. Validation loss: 0.712.. Validation accuracy: 0.793
Epoch 4/10.. Train loss: 0.961.. Validation loss: 0.713.. Validation accuracy: 0.808
Epoch 4/10.. Train loss: 1.229.. Validation loss: 0.690.. Validation accuracy: 0.804
Epoch 4/10.. Train loss: 1.246.. Validation loss: 0.676.. Validation accuracy: 0.802
Epoch 4/10.. Train loss: 1.043.. Validation loss: 0.657.. Validation accuracy: 0.809
Epoch 4/10.. Train loss: 1.179.. Validation loss: 0.665.. Validation accuracy: 0.814
Epoch 4/10.. Train loss: 0.969.. Validation loss: 0.670.. Validation accuracy: 0.812
Epoch 4/10.. Train loss: 1.109.. Validation loss: 0.694.. Validation accuracy: 0.802
Epoch 4/10.. Train loss: 1.217.. Validation loss: 0.722.. Validation accuracy: 0.802
Epoch 4/10.. Train loss: 1.261.. Validation loss: 0.717.. Validation accuracy: 0.792
Epoch 4/10.. Train loss: 1.159.. Validation loss: 0.701.. Validation accuracy: 0.796
Epoch 4/10.. Train loss: 1.193.. Validation loss: 0.679.. Validation accuracy: 0.812
Epoch 4/10.. Train loss: 1.235.. Validation loss: 0.679.. Validation accuracy: 0.817
Epoch 4/10.. Train loss: 1.104.. Validation loss: 0.690.. Validation accuracy: 0.804
Epoch 4/10.. Train loss: 1.096.. Validation loss: 0.678.. Validation accuracy: 0.812
Epoch 4/10.. Train loss: 1.147.. Validation loss: 0.663.. Validation accuracy: 0.820
Epoch 4/10.. Train loss: 1.334.. Validation loss: 0.653.. Validation accuracy: 0.821
Epoch 5/10.. Train loss: 1.161.. Validation loss: 0.666.. Validation accuracy: 0.808
Epoch 5/10.. Train loss: 1.063.. Validation loss: 0.685.. Validation accuracy: 0.805
Epoch 5/10.. Train loss: 1.024.. Validation loss: 0.704.. Validation accuracy: 0.806
Epoch 5/10.. Train loss: 1.160.. Validation loss: 0.688.. Validation accuracy: 0.814
Epoch 5/10.. Train loss: 1.280.. Validation loss: 0.662.. Validation accuracy: 0.828
Epoch 5/10.. Train loss: 1.052.. Validation loss: 0.656.. Validation accuracy: 0.834
Epoch 5/10.. Train loss: 1.088.. Validation loss: 0.657.. Validation accuracy: 0.838
Epoch 5/10.. Train loss: 1.008.. Validation loss: 0.647.. Validation accuracy: 0.838
Epoch 5/10.. Train loss: 1.085.. Validation loss: 0.648.. Validation accuracy: 0.829
Epoch 5/10.. Train loss: 1.056.. Validation loss: 0.641.. Validation accuracy: 0.828
Epoch 5/10.. Train loss: 1.154.. Validation loss: 0.635.. Validation accuracy: 0.824
```

```
Epoch 5/10.. Train loss: 1.077.. Validation loss: 0.645.. Validation accuracy: 0.824
Epoch 5/10.. Train loss: 1.116.. Validation loss: 0.660.. Validation accuracy: 0.823
Epoch 5/10.. Train loss: 1.205.. Validation loss: 0.660.. Validation accuracy: 0.821
Epoch 5/10.. Train loss: 1.310.. Validation loss: 0.673.. Validation accuracy: 0.822
Epoch 5/10.. Train loss: 1.193.. Validation loss: 0.655.. Validation accuracy: 0.828
Epoch 5/10.. Train loss: 0.972.. Validation loss: 0.644.. Validation accuracy: 0.829
Epoch 5/10.. Train loss: 1.324.. Validation loss: 0.641.. Validation accuracy: 0.811
Epoch 5/10.. Train loss: 1.126.. Validation loss: 0.663.. Validation accuracy: 0.815
Epoch 5/10.. Train loss: 1.037.. Validation loss: 0.662.. Validation accuracy: 0.823
Epoch 5/10.. Train loss: 1.130.. Validation loss: 0.676.. Validation accuracy: 0.818
Epoch 6/10.. Train loss: 1.191.. Validation loss: 0.666.. Validation accuracy: 0.815
Epoch 6/10.. Train loss: 1.126.. Validation loss: 0.657.. Validation accuracy: 0.813
Epoch 6/10.. Train loss: 1.177.. Validation loss: 0.662.. Validation accuracy: 0.810
Epoch 6/10.. Train loss: 1.186.. Validation loss: 0.654.. Validation accuracy: 0.817
Epoch 6/10.. Train loss: 1.075.. Validation loss: 0.653.. Validation accuracy: 0.822
Epoch 6/10.. Train loss: 1.232.. Validation loss: 0.645.. Validation accuracy: 0.821
Epoch 6/10.. Train loss: 1.172.. Validation loss: 0.640.. Validation accuracy: 0.827
Epoch 6/10.. Train loss: 1.004.. Validation loss: 0.633.. Validation accuracy: 0.828
Epoch 6/10.. Train loss: 0.981.. Validation loss: 0.628.. Validation accuracy: 0.834
Epoch 6/10.. Train loss: 1.004.. Validation loss: 0.646.. Validation accuracy: 0.820
Epoch 6/10.. Train loss: 1.029.. Validation loss: 0.669.. Validation accuracy: 0.811
Epoch 6/10.. Train loss: 1.062.. Validation loss: 0.640.. Validation accuracy: 0.828
Epoch 6/10.. Train loss: 1.050.. Validation loss: 0.626.. Validation accuracy: 0.837
Epoch 6/10.. Train loss: 1.160.. Validation loss: 0.605.. Validation accuracy: 0.842
Epoch 6/10.. Train loss: 1.142.. Validation loss: 0.607.. Validation accuracy: 0.834
Epoch 6/10.. Train loss: 1.029.. Validation loss: 0.641.. Validation accuracy: 0.821
Epoch 6/10.. Train loss: 1.197.. Validation loss: 0.635.. Validation accuracy: 0.825
Epoch 6/10.. Train loss: 0.950.. Validation loss: 0.627.. Validation accuracy: 0.831
Epoch 6/10.. Train loss: 1.001.. Validation loss: 0.631.. Validation accuracy: 0.825
Epoch 6/10.. Train loss: 1.038.. Validation loss: 0.636.. Validation accuracy: 0.827
Epoch 7/10.. Train loss: 0.881.. Validation loss: 0.645.. Validation accuracy: 0.828
Epoch 7/10.. Train loss: 1.101.. Validation loss: 0.658.. Validation accuracy: 0.823
Epoch 7/10.. Train loss: 1.011.. Validation loss: 0.658.. Validation accuracy: 0.814
Epoch 7/10.. Train loss: 1.076.. Validation loss: 0.653.. Validation accuracy: 0.809
Epoch 7/10.. Train loss: 0.964.. Validation loss: 0.653.. Validation accuracy: 0.810
Epoch 7/10.. Train loss: 1.053.. Validation loss: 0.643.. Validation accuracy: 0.816
Epoch 7/10.. Train loss: 1.061.. Validation loss: 0.620.. Validation accuracy: 0.809
Epoch 7/10.. Train loss: 1.002.. Validation loss: 0.605.. Validation accuracy: 0.823
Epoch 7/10.. Train loss: 0.926.. Validation loss: 0.608.. Validation accuracy: 0.831
Epoch 7/10.. Train loss: 1.352.. Validation loss: 0.616.. Validation accuracy: 0.826
Epoch 7/10.. Train loss: 0.981.. Validation loss: 0.639.. Validation accuracy: 0.831
Epoch 7/10.. Train loss: 1.133.. Validation loss: 0.654.. Validation accuracy: 0.816
Epoch 7/10.. Train loss: 1.137.. Validation loss: 0.660.. Validation accuracy: 0.805
Epoch 7/10.. Train loss: 1.025.. Validation loss: 0.642.. Validation accuracy: 0.822
Epoch 7/10.. Train loss: 1.100.. Validation loss: 0.614.. Validation accuracy: 0.832
Epoch 7/10.. Train loss: 1.129.. Validation loss: 0.589.. Validation accuracy: 0.841
Epoch 7/10.. Train loss: 1.057.. Validation loss: 0.589.. Validation accuracy: 0.839
Epoch 7/10.. Train loss: 1.127.. Validation loss: 0.601.. Validation accuracy: 0.827
```

```
Epoch 7/10.. Train loss: 1.124.. Validation loss: 0.627.. Validation accuracy: 0.825
Epoch 7/10.. Train loss: 1.093.. Validation loss: 0.622.. Validation accuracy: 0.827
Epoch 7/10.. Train loss: 1.005.. Validation loss: 0.614.. Validation accuracy: 0.827
Epoch 8/10.. Train loss: 1.088.. Validation loss: 0.628.. Validation accuracy: 0.823
Epoch 8/10.. Train loss: 0.812.. Validation loss: 0.623.. Validation accuracy: 0.830
Epoch 8/10.. Train loss: 1.119.. Validation loss: 0.602.. Validation accuracy: 0.822
Epoch 8/10.. Train loss: 0.985.. Validation loss: 0.583.. Validation accuracy: 0.829
Epoch 8/10.. Train loss: 1.099.. Validation loss: 0.575.. Validation accuracy: 0.839
Epoch 8/10.. Train loss: 0.971.. Validation loss: 0.588.. Validation accuracy: 0.837
Epoch 8/10.. Train loss: 0.966.. Validation loss: 0.600.. Validation accuracy: 0.826
Epoch 8/10.. Train loss: 1.100.. Validation loss: 0.590.. Validation accuracy: 0.834
Epoch 8/10.. Train loss: 0.839.. Validation loss: 0.580.. Validation accuracy: 0.831
Epoch 8/10.. Train loss: 0.927.. Validation loss: 0.589.. Validation accuracy: 0.833
Epoch 8/10.. Train loss: 1.002.. Validation loss: 0.583.. Validation accuracy: 0.840
Epoch 8/10.. Train loss: 1.016.. Validation loss: 0.572.. Validation accuracy: 0.848
Epoch 8/10.. Train loss: 1.101.. Validation loss: 0.566.. Validation accuracy: 0.847
Epoch 8/10.. Train loss: 1.134.. Validation loss: 0.577.. Validation accuracy: 0.848
Epoch 8/10.. Train loss: 1.113.. Validation loss: 0.603.. Validation accuracy: 0.841
Epoch 8/10.. Train loss: 1.021.. Validation loss: 0.631.. Validation accuracy: 0.822
Epoch 8/10.. Train loss: 1.095.. Validation loss: 0.619.. Validation accuracy: 0.825
Epoch 8/10.. Train loss: 1.086.. Validation loss: 0.593.. Validation accuracy: 0.831
Epoch 8/10.. Train loss: 0.989.. Validation loss: 0.585.. Validation accuracy: 0.828
Epoch 8/10.. Train loss: 0.808.. Validation loss: 0.586.. Validation accuracy: 0.832
Epoch 9/10.. Train loss: 1.004.. Validation loss: 0.597.. Validation accuracy: 0.824
Epoch 9/10.. Train loss: 0.974.. Validation loss: 0.618.. Validation accuracy: 0.815
Epoch 9/10.. Train loss: 0.913.. Validation loss: 0.612.. Validation accuracy: 0.821
Epoch 9/10.. Train loss: 1.003.. Validation loss: 0.604.. Validation accuracy: 0.825
Epoch 9/10.. Train loss: 1.038.. Validation loss: 0.599.. Validation accuracy: 0.837
Epoch 9/10.. Train loss: 1.084.. Validation loss: 0.584.. Validation accuracy: 0.831
Epoch 9/10.. Train loss: 1.199.. Validation loss: 0.580.. Validation accuracy: 0.842
Epoch 9/10.. Train loss: 0.940.. Validation loss: 0.580.. Validation accuracy: 0.838
Epoch 9/10.. Train loss: 1.126.. Validation loss: 0.583.. Validation accuracy: 0.839
Epoch 9/10.. Train loss: 0.971.. Validation loss: 0.577.. Validation accuracy: 0.836
Epoch 9/10.. Train loss: 1.075.. Validation loss: 0.570.. Validation accuracy: 0.837
Epoch 9/10.. Train loss: 0.846.. Validation loss: 0.558.. Validation accuracy: 0.840
Epoch 9/10.. Train loss: 0.870.. Validation loss: 0.561.. Validation accuracy: 0.849
Epoch 9/10.. Train loss: 0.947.. Validation loss: 0.571.. Validation accuracy: 0.848
Epoch 9/10.. Train loss: 1.013.. Validation loss: 0.556.. Validation accuracy: 0.846
Epoch 9/10.. Train loss: 1.087.. Validation loss: 0.553.. Validation accuracy: 0.853
Epoch 9/10.. Train loss: 1.011.. Validation loss: 0.579.. Validation accuracy: 0.839
Epoch 9/10.. Train loss: 1.126.. Validation loss: 0.595.. Validation accuracy: 0.841
Epoch 9/10.. Train loss: 1.028.. Validation loss: 0.589.. Validation accuracy: 0.849
Epoch 9/10.. Train loss: 1.082.. Validation loss: 0.577.. Validation accuracy: 0.844
Epoch 9/10.. Train loss: 0.899.. Validation loss: 0.571.. Validation accuracy: 0.850
Epoch 10/10.. Train loss: 1.047.. Validation loss: 0.580.. Validation accuracy: 0.846
Epoch 10/10.. Train loss: 0.981.. Validation loss: 0.594.. Validation accuracy: 0.831
Epoch 10/10.. Train loss: 1.133.. Validation loss: 0.597.. Validation accuracy: 0.842
Epoch 10/10.. Train loss: 0.926.. Validation loss: 0.610.. Validation accuracy: 0.829
```

```
Epoch 10/10.. Train loss: 0.915.. Validation loss: 0.619.. Validation accuracy: 0.830
Epoch 10/10.. Train loss: 0.971.. Validation loss: 0.617.. Validation accuracy: 0.837
Epoch 10/10.. Train loss: 0.993.. Validation loss: 0.600.. Validation accuracy: 0.839
Epoch 10/10.. Train loss: 0.954.. Validation loss: 0.596.. Validation accuracy: 0.843
Epoch 10/10.. Train loss: 1.115.. Validation loss: 0.603.. Validation accuracy: 0.842
Epoch 10/10.. Train loss: 1.017.. Validation loss: 0.623.. Validation accuracy: 0.828
Epoch 10/10.. Train loss: 0.966.. Validation loss: 0.622.. Validation accuracy: 0.827
Epoch 10/10.. Train loss: 0.941.. Validation loss: 0.596.. Validation accuracy: 0.831
Epoch 10/10.. Train loss: 0.867.. Validation loss: 0.569.. Validation accuracy: 0.840
Epoch 10/10.. Train loss: 0.943.. Validation loss: 0.565.. Validation accuracy: 0.840
Epoch 10/10.. Train loss: 1.050.. Validation loss: 0.570.. Validation accuracy: 0.839
Epoch 10/10.. Train loss: 1.078.. Validation loss: 0.565.. Validation accuracy: 0.849
Epoch 10/10.. Train loss: 1.196.. Validation loss: 0.565.. Validation accuracy: 0.845
Epoch 10/10.. Train loss: 0.994.. Validation loss: 0.577.. Validation accuracy: 0.853
Epoch 10/10.. Train loss: 0.983.. Validation loss: 0.572.. Validation accuracy: 0.855
Epoch 10/10.. Train loss: 0.954.. Validation loss: 0.569.. Validation accuracy: 0.841
Epoch 10/10.. Train loss: 0.847.. Validation loss: 0.577.. Validation accuracy: 0.834
```

In [23]:

```
hello
```

In [ ]:

In [ ]:

In [ ]:

### 2.1 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 [26]:
```
# TODO: Do validation on the test set
correct = 0
total = 0
model.to(device)

with torch.no_grad():
    for data in testloaders:
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        output = model(inputs)
        _, predicted = torch.max(output.data, 1)
```

```
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

                print(f'Accuracy on test images: {100 * correct/total} %')
```

```
Accuracy on test images: 62.5 %
Accuracy on test images: 71.09375 %
Accuracy on test images: 71.35416666666667 %
Accuracy on test images: 69.140625 %
Accuracy on test images: 73.125 %
Accuracy on test images: 72.39583333333333 %
Accuracy on test images: 74.33035714285714 %
Accuracy on test images: 75.390625 %
Accuracy on test images: 77.08333333333333 %
Accuracy on test images: 77.8125 %
Accuracy on test images: 77.13068181818181 %
Accuracy on test images: 76.171875 %
Accuracy on test images: 75.7020757020757 %
```

## 2.2 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 [28]: # TODO: Save the checkpoint
         model.class_to_idx = image_datasets.class_to_idx
         model.cpu
         torch.save({'structure': 'vgg19'
                    ,'fc2': 6272
                    ,'fc3': 784
                    ,'state_dict': model.state_dict()
                    ,'class_to_idx': model.class_to_idx}
                    ,'checkpoint.pth')
```

## 2.3 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
         def load_model(path):
             checkpoint = torch.load(path)
             structure = checkpoint['structure']
             fc2 = checkpoint['fc2']
             fc3 = checkpoint['fc3']
             model = classifier_setup()
             model.class_to_idx = checkpoint['class_to_idx']
             model.load_state_dict(checkpoint['state_dict'])

         load_model('checkpoint.pth')
         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): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 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): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (33): ReLU(inplace)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (fc1): Linear(in_features=25088, out_features=6272, bias=True)
    (relu1): ReLU()
    (dropout): Dropout(p=0.2)
    (fc2): Linear(in_features=6272, out_features=784, bias=True)
    (relu2): ReLU()
    (fc3): Linear(in_features=784, out_features=102, bias=True)
    (output): LogSoftmax()
  )
)
```

# 3 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.

## 3.1 Image Preprocessing

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

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

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

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

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

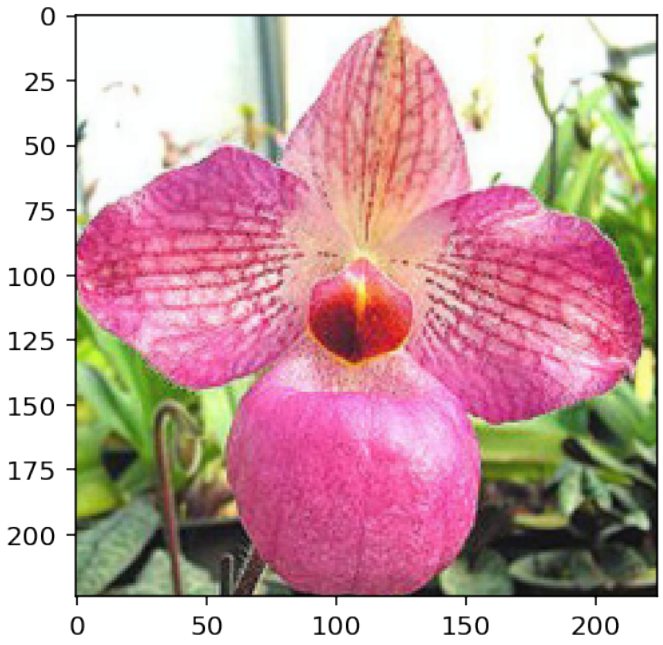
```
In [36]: def process_image(image):
             ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
                 returns an Numpy array
             '''

             # TODO: Process a PIL image for use in a PyTorch model
             image_pil = Image.open(image)

             transform = transforms.Compose([
                 transforms.Resize(256)
                 ,transforms.CenterCrop(224)
                 ,transforms.ToTensor()
                 ,transforms.Normalize([0.485, 0.456, 0.406]
                                       ,[0.229, 0.224, 0.225])
             ])

             image_tensor = transform(image_pil)

             return image_tensor

         image = data_dir + '/test' + '/1/' + 'image_06760.jpg'
         image = process_image(image)
         print(image)
```

```
tensor([[[-1.5357, -1.6213, -1.6213,  ..., -0.2684, -0.2171, -0.1828],
         [-1.5870, -1.6555, -1.6727,  ..., -0.3198, -0.2513, -0.1828],
         [-1.6384, -1.6898, -1.7069,  ..., -0.3712, -0.2856, -0.2513],
         ...,
         [-1.1418, -1.2788, -1.2617,  ..., -0.4739, -0.4739, -0.4397],
         [-1.0390, -1.1418, -1.2103,  ..., -0.4397, -0.4054, -0.3883],
         [-0.9534, -0.9705, -1.0048,  ..., -0.4226, -0.3369, -0.3541]],

        [[-1.8782, -1.8957, -1.8957,  ..., -1.6331, -1.6155, -1.5805],
         [-1.8957, -1.9132, -1.9132,  ..., -1.6856, -1.6681, -1.6331],
         [-1.9307, -1.9307, -1.9307,  ..., -1.7031, -1.7031, -1.6681],
         ...,
         [-0.6352, -0.7402, -0.7402,  ..., -1.0903, -1.0728, -1.0553],
         [-0.5651, -0.6352, -0.6527,  ..., -1.0378, -1.0203, -1.0028],
         [-0.4776, -0.4426, -0.4426,  ..., -1.0203, -0.9503, -0.9678]],

        [[-1.7696, -1.7696, -1.7522,  ..., -1.7347, -1.7347, -1.7347],
         [-1.7696, -1.7696, -1.7522,  ..., -1.7522, -1.7522, -1.7347],
         [-1.7870, -1.7870, -1.7696,  ..., -1.7696, -1.7696, -1.7522],
         ...,
```

```
        [-1.8044, -1.8044, -1.7870,  ..., -1.6650, -1.6650, -1.6476],
        [-1.7696, -1.8044, -1.8044,  ..., -1.6127, -1.6127, -1.6127],
        [-1.7522, -1.7522, -1.7696,  ..., -1.5779, -1.5256, -1.5604]]])
```

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 displayed
             image = np.clip(image, 0, 1)

             ax.imshow(image)

             return ax

         imshow(process_image("flowers/test/2/image_05109.jpg"))

Out[39]: <matplotlib.axes._subplots.AxesSubplot at 0x7f878f75a7f0>
```




### 3.2 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 (Section 2.2). 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 [41]: model.class_to_idx = image_datasets.class_to_idx
         ctx = model.class_to_idx

         def predict(image_path, 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.to(device)
            image_torch = process_image(image_path)
            image_torch = image_torch.unsqueeze_(0)
            image_torch = image_torch.float()

            with torch.no_grad():
                output = model.forward(image_torch.cuda())

            probability = F.softmax(output.data, dim=1)

            return probability.topk(topk)

        image = data_dir + '/test' + '/1/' + 'image_06760.jpg'
        val1, val2 = predict(image, model)
        print(val1)
        print(val2)
```

```
tensor([[ 0.5224,  0.1235,  0.1077,  0.0902,  0.0619]], device='cuda:0')
tensor([[ 13,  53,  49,   0,  99]], device='cuda:0')
```

### 3.3 Sanity Checking

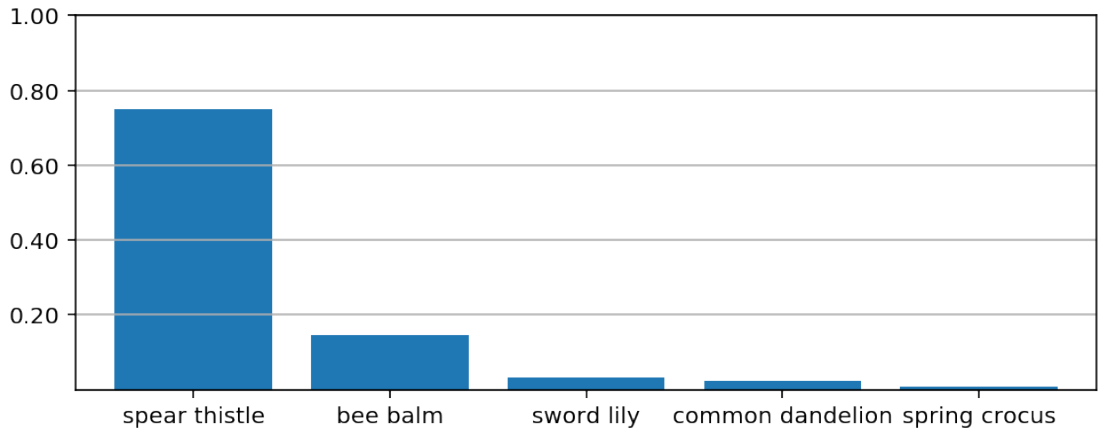
Now that you can use a trained model for predictions, check to make sure it makes sense. Even if the testing accuracy is high, it's always good to check that there aren't obvious bugs. Use `matplotlib` to plot the probabilities for the top 5 classes as a bar graph, along with the input image. It should 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 [52]: # TODO: Display an image along with the top 5 classes
         def sanity_check():
             plt.rcParams["figure.figsize"] = (10, 5)
             plt.subplot(211)

             index = 1
             path = test_dir + '/19/image_06197.jpg'

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


             axs = imshow(image, ax = plt)
             axs.axis('off')
             axs.title(cat_to_name[str(index)])
```

```
    axs.show()


    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=(8,3))
    width = 0.8
    tickLocations = np.arange(N)
    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)
    ax.yaxis.set_major_formatter(FormatStrFormatter('%.2f'))

    plt.show()
```

In [55]:
```
sanity_check()
```


1.00
0.80
0.60
0.40
0.20
spear thistle
bee balm
sword lily
common dandelion
spring crocus

In [ ]:

In [ ]: