Image Classifier Project Colab

January 24, 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 [1]: # Colab settings
        from google.colab import drive
        drive.mount('/content/drive/')
        !pip3 install torch==0.4.0 torchvision
        !cd '/content/drive/My Drive/app/'
        !wget -c https://raw.githubusercontent.com/udacity/deep-learning-v2-pytorch/master/int
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call drive.mount("/c
Requirement already satisfied: torch==0.4.0 in /usr/local/lib/python3.6/dist-packages (0.4.0)
```

```
Requirement already satisfied: torchvision in /usr/local/lib/python3.6/dist-packages (0.2.1)
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from torchvision)
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.6/dist-packages (from to
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from torchvisio
--2019-01-02 14:30:18--  https://raw.githubusercontent.com/udacity/deep-learning-v2-pytorch/mas
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.13
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... conne
HTTP request sent, awaiting response... 416 Range Not Satisfiable

    The file is already fully retrieved; nothing to do.
```

```
In [2]: # we need pillow version 5.3.0
        # we will uninstall the older version first
        !pip uninstall -y Pillow
        # install the new one
        !pip install Pillow==5.3.0
        # import the new one
        import PIL
        print('pillow version = '+PIL.PILLOW_VERSION)
        # this should print 5.3.0. If it doesn't, then restart your runtime:
        # Menu > Runtime > Restart Runtime
```

```
Uninstalling Pillow-5.3.0:
  Successfully uninstalled Pillow-5.3.0
Collecting Pillow==5.3.0
  Using cached https://files.pythonhosted.org/packages/62/94/5430ebaa83f91cc7a9f687ff5238e2616
Installing collected packages: Pillow
Successfully installed Pillow-5.3.0
pillow version = 5.3.0
```

```
In [0]: # Imports here
        import torch
        from torch import nn, optim
        import torch.nn.functional as F
        import numpy as np
        import torchvision
        from torchvision import transforms, datasets, models
        from collections import OrderedDict
        import matplotlib.pyplot as plt
        import helper
        from PIL import Image
        import seaborn
        import os, random
        import json
```

1.1 Load the data

Here you'll use `torchvision` to load the data (documentation). You can download the data here. The dataset is split into two parts, training and validation. 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. If you use a pre-trained network, you'll also need to make sure the input data is resized to 224x224 pixels as required by the networks.

The validation set is 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 available from `torchvision` were trained on the ImageNet dataset where each color channel was normalized separately. For both 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 [0]: # Download the required data files

        !wget -cq https://github.com/udacity/pytorch_challenge/raw/master/cat_to_name.json
        !wget -cq https://s3.amazonaws.com/content.udacity-data.com/courses/nd188/flower_data.z

        !rm -r flower_data || true
        !unzip -qq flower_data.zip

In [0]: data_dir = 'flower_data'
        train_dir = data_dir + '/train'
        valid_dir = data_dir + '/valid'

In [0]: # TODO: Define your transforms for the training and validation sets
        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])])
        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
        train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
        test_data = datasets.ImageFolder(valid_dir, transform=test_transforms)

        # TODO: Using the image datasets and the trainforms, define the dataloaders
        trainloader = torch.utils.data.DataLoader(train_data, batch_size=16, shuffle=True)
        testloader = torch.utils.data.DataLoader(test_data, batch_size=16)
```

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 [0]: 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. 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 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.

```
In [8]: # TODO: Build and train your network
        model = models.vgg16(pretrained=True)
        model

Out[8]: 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)
  )
)
```

In [0]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

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

n_input = 25088
n_middle1 = 12000
n_middle2 = 6000
n_middle3 = 1000
```

```
        n_output = 102

        classifier = nn.Sequential(OrderedDict([
                                  ('fc1', nn.Linear(n_input, n_middle1)),
                                  ('relu', nn.ReLU()),
                                  ('dropout', nn.Dropout(p=0.2)),
                                  ('fc2', nn.Linear(n_middle1, n_middle2)),
                                  ('relu', nn.ReLU()),
                                  ('dropout', nn.Dropout(p=0.2)),
                                  ('fc3', nn.Linear(n_middle2, n_middle3)),
                                  ('relu', nn.ReLU()),
                                  ('dropout', nn.Dropout(p=0.2)),
                                  ('fc4', nn.Linear(n_middle3, n_output)),
                                  ('output', nn.LogSoftmax(dim=1))
                                  ]))

        model.classifier = classifier
        criterion = nn.NLLLoss()
```

In [21]:
```
# Only train the classifier parameters, feature parameters are frozen
optimizer = optim.Adam(model.classifier.parameters(), lr=0.00005)

print('device =',device)
model.to(device)
```

```
device = cuda:0
```

Out[21]:
```
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=12000, bias=True)
    (relu): ReLU()
    (dropout): Dropout(p=0.2)
    (fc2): Linear(in_features=12000, out_features=6000, bias=True)
    (fc3): Linear(in_features=6000, out_features=1000, bias=True)
    (fc4): Linear(in_features=1000, out_features=102, bias=True)
    (output): LogSoftmax()
  )
)
```

In [22]:
```
epochs = 1
steps = 0
running_loss = 0
print_every = 5
for epoch in range(epochs):
    for inputs, labels in trainloader:
        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:
            test_loss = 0
            accuracy = 0
            model.eval()
```

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

                            test_loss += batch_loss.item()

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

                    print(f"Epoch {epoch+1}/{epochs}.. "
                          f"Train loss: {running_loss/print_every:.3f}.. "
                          f"Test loss: {test_loss/len(testloader):.3f}.. "
                          f"Test accuracy: {accuracy/len(testloader):.3f}")
                    running_loss = 0
                    model.train()
```

```
Epoch 1/1.. Train loss: 0.461.. Test loss: 0.457.. Test accuracy: 0.906
Epoch 1/1.. Train loss: 0.488.. Test loss: 0.433.. Test accuracy: 0.911
Epoch 1/1.. Train loss: 0.510.. Test loss: 0.449.. Test accuracy: 0.915
Epoch 1/1.. Train loss: 0.508.. Test loss: 0.476.. Test accuracy: 0.913
Epoch 1/1.. Train loss: 0.513.. Test loss: 0.496.. Test accuracy: 0.909
Epoch 1/1.. Train loss: 0.363.. Test loss: 0.499.. Test accuracy: 0.912
Epoch 1/1.. Train loss: 0.578.. Test loss: 0.442.. Test accuracy: 0.915
Epoch 1/1.. Train loss: 0.546.. Test loss: 0.412.. Test accuracy: 0.916
Epoch 1/1.. Train loss: 0.479.. Test loss: 0.393.. Test accuracy: 0.922
Epoch 1/1.. Train loss: 0.427.. Test loss: 0.387.. Test accuracy: 0.923
Epoch 1/1.. Train loss: 0.282.. Test loss: 0.406.. Test accuracy: 0.917
Epoch 1/1.. Train loss: 0.698.. Test loss: 0.437.. Test accuracy: 0.916
Epoch 1/1.. Train loss: 0.322.. Test loss: 0.464.. Test accuracy: 0.913
Epoch 1/1.. Train loss: 0.567.. Test loss: 0.469.. Test accuracy: 0.913
Epoch 1/1.. Train loss: 0.428.. Test loss: 0.480.. Test accuracy: 0.913
Epoch 1/1.. Train loss: 0.498.. Test loss: 0.484.. Test accuracy: 0.910
Epoch 1/1.. Train loss: 0.254.. Test loss: 0.463.. Test accuracy: 0.913
Epoch 1/1.. Train loss: 0.494.. Test loss: 0.443.. Test accuracy: 0.918
Epoch 1/1.. Train loss: 0.269.. Test loss: 0.436.. Test accuracy: 0.921
Epoch 1/1.. Train loss: 0.546.. Test loss: 0.429.. Test accuracy: 0.917
Epoch 1/1.. Train loss: 0.442.. Test loss: 0.418.. Test accuracy: 0.916
Epoch 1/1.. Train loss: 0.446.. Test loss: 0.424.. Test accuracy: 0.919
Epoch 1/1.. Train loss: 0.461.. Test loss: 0.434.. Test accuracy: 0.919
Epoch 1/1.. Train loss: 0.462.. Test loss: 0.438.. Test accuracy: 0.917
Epoch 1/1.. Train loss: 0.470.. Test loss: 0.439.. Test accuracy: 0.913
Epoch 1/1.. Train loss: 0.594.. Test loss: 0.429.. Test accuracy: 0.925
Epoch 1/1.. Train loss: 0.571.. Test loss: 0.433.. Test accuracy: 0.925
```

```
Epoch 1/1.. Train loss: 0.447.. Test loss: 0.441.. Test accuracy: 0.922
Epoch 1/1.. Train loss: 0.508.. Test loss: 0.447.. Test accuracy: 0.917
Epoch 1/1.. Train loss: 0.515.. Test loss: 0.441.. Test accuracy: 0.911
Epoch 1/1.. Train loss: 0.237.. Test loss: 0.438.. Test accuracy: 0.913
Epoch 1/1.. Train loss: 0.367.. Test loss: 0.440.. Test accuracy: 0.918
Epoch 1/1.. Train loss: 0.567.. Test loss: 0.432.. Test accuracy: 0.915
Epoch 1/1.. Train loss: 0.606.. Test loss: 0.421.. Test accuracy: 0.922
Epoch 1/1.. Train loss: 0.347.. Test loss: 0.425.. Test accuracy: 0.918
Epoch 1/1.. Train loss: 0.543.. Test loss: 0.437.. Test accuracy: 0.916
Epoch 1/1.. Train loss: 0.374.. Test loss: 0.454.. Test accuracy: 0.910
Epoch 1/1.. Train loss: 0.301.. Test loss: 0.453.. Test accuracy: 0.910
Epoch 1/1.. Train loss: 0.698.. Test loss: 0.443.. Test accuracy: 0.909
Epoch 1/1.. Train loss: 0.304.. Test loss: 0.428.. Test accuracy: 0.919
Epoch 1/1.. Train loss: 0.484.. Test loss: 0.426.. Test accuracy: 0.921
Epoch 1/1.. Train loss: 0.374.. Test loss: 0.434.. Test accuracy: 0.918
Epoch 1/1.. Train loss: 0.459.. Test loss: 0.437.. Test accuracy: 0.927
Epoch 1/1.. Train loss: 0.522.. Test loss: 0.424.. Test accuracy: 0.923
Epoch 1/1.. Train loss: 0.966.. Test loss: 0.416.. Test accuracy: 0.924
Epoch 1/1.. Train loss: 0.304.. Test loss: 0.422.. Test accuracy: 0.928
Epoch 1/1.. Train loss: 0.330.. Test loss: 0.432.. Test accuracy: 0.924
Epoch 1/1.. Train loss: 0.480.. Test loss: 0.434.. Test accuracy: 0.922
Epoch 1/1.. Train loss: 0.673.. Test loss: 0.428.. Test accuracy: 0.923
Epoch 1/1.. Train loss: 0.449.. Test loss: 0.414.. Test accuracy: 0.929
Epoch 1/1.. Train loss: 0.626.. Test loss: 0.411.. Test accuracy: 0.931
Epoch 1/1.. Train loss: 0.602.. Test loss: 0.413.. Test accuracy: 0.930
Epoch 1/1.. Train loss: 0.655.. Test loss: 0.417.. Test accuracy: 0.923
Epoch 1/1.. Train loss: 0.630.. Test loss: 0.412.. Test accuracy: 0.928
Epoch 1/1.. Train loss: 0.657.. Test loss: 0.414.. Test accuracy: 0.927
Epoch 1/1.. Train loss: 0.488.. Test loss: 0.440.. Test accuracy: 0.923
Epoch 1/1.. Train loss: 0.304.. Test loss: 0.464.. Test accuracy: 0.915
Epoch 1/1.. Train loss: 0.539.. Test loss: 0.471.. Test accuracy: 0.919
Epoch 1/1.. Train loss: 0.617.. Test loss: 0.431.. Test accuracy: 0.919
Epoch 1/1.. Train loss: 0.679.. Test loss: 0.420.. Test accuracy: 0.933
Epoch 1/1.. Train loss: 0.590.. Test loss: 0.411.. Test accuracy: 0.929
Epoch 1/1.. Train loss: 0.524.. Test loss: 0.442.. Test accuracy: 0.924
Epoch 1/1.. Train loss: 0.544.. Test loss: 0.477.. Test accuracy: 0.916
Epoch 1/1.. Train loss: 0.598.. Test loss: 0.483.. Test accuracy: 0.911
Epoch 1/1.. Train loss: 0.675.. Test loss: 0.476.. Test accuracy: 0.909
Epoch 1/1.. Train loss: 0.617.. Test loss: 0.474.. Test accuracy: 0.905
Epoch 1/1.. Train loss: 0.625.. Test loss: 0.483.. Test accuracy: 0.903
Epoch 1/1.. Train loss: 0.382.. Test loss: 0.490.. Test accuracy: 0.905
Epoch 1/1.. Train loss: 0.363.. Test loss: 0.460.. Test accuracy: 0.910
Epoch 1/1.. Train loss: 0.493.. Test loss: 0.444.. Test accuracy: 0.917
Epoch 1/1.. Train loss: 0.488.. Test loss: 0.429.. Test accuracy: 0.917
Epoch 1/1.. Train loss: 0.238.. Test loss: 0.408.. Test accuracy: 0.922
Epoch 1/1.. Train loss: 0.681.. Test loss: 0.392.. Test accuracy: 0.924
Epoch 1/1.. Train loss: 0.367.. Test loss: 0.392.. Test accuracy: 0.927
Epoch 1/1.. Train loss: 0.344.. Test loss: 0.387.. Test accuracy: 0.929
```

```
Epoch 1/1.. Train loss: 0.671.. Test loss: 0.387.. Test accuracy: 0.927
Epoch 1/1.. Train loss: 0.286.. Test loss: 0.383.. Test accuracy: 0.928
Epoch 1/1.. Train loss: 0.762.. Test loss: 0.374.. Test accuracy: 0.928
Epoch 1/1.. Train loss: 0.639.. Test loss: 0.366.. Test accuracy: 0.930
Epoch 1/1.. Train loss: 0.492.. Test loss: 0.381.. Test accuracy: 0.925
Epoch 1/1.. Train loss: 0.379.. Test loss: 0.409.. Test accuracy: 0.917
Epoch 1/1.. Train loss: 0.458.. Test loss: 0.427.. Test accuracy: 0.915
```

2.1 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 [0]: # TODO: Save the checkpoint
        model.to('cpu')
        checkpoint = {'input_size': n_input,
                      'output_size': n_output,
                      'hidden_layers': [n_middle1, n_middle2, n_middle3],
                      'opt_state': optimizer.state_dict,
                      'n_epochs': epochs,
                      'class_to_idx': train_data.class_to_idx,
                      'state_dict': model.state_dict()}

        torch.save(checkpoint, '/content/drive/My Drive/app/checkpoint.pth')
```

```
In [0]: # TODO: Write a function that loads a checkpoint and rebuilds the model
        def load_checkpoint(filepath):
            checkpoint = torch.load(filepath)

            model = models.vgg16(pretrained=True)

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


            model.class_to_idx = checkpoint['class_to_idx']

            # Create the classifier
```

```
    classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(checkpoint['input_size'], checkpoint['hidd
                          ('relu', nn.ReLU()),
                          ('dropout', nn.Dropout(p=0.2)),
                          ('fc2', nn.Linear(checkpoint['hidden_layers'][0], checkpoint
                          ('relu', nn.ReLU()),
                          ('dropout', nn.Dropout(p=0.2)),
                          ('fc3', nn.Linear(checkpoint['hidden_layers'][1], checkpoint
                          ('relu', nn.ReLU()),
                          ('dropout', nn.Dropout(p=0.2)),
                          ('fc4', nn.Linear(checkpoint['hidden_layers'][2], checkpoint
                          ('output', nn.LogSoftmax(dim=1))
                          ]))



    # Put the classifier on the pretrained network
    model.classifier = classifier

    model.load_state_dict(checkpoint['state_dict'], strict=False)

    return model
```

2.2 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 [0]: model = load_checkpoint('/content/drive/My Drive/app/checkpoint.pth')
```

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.

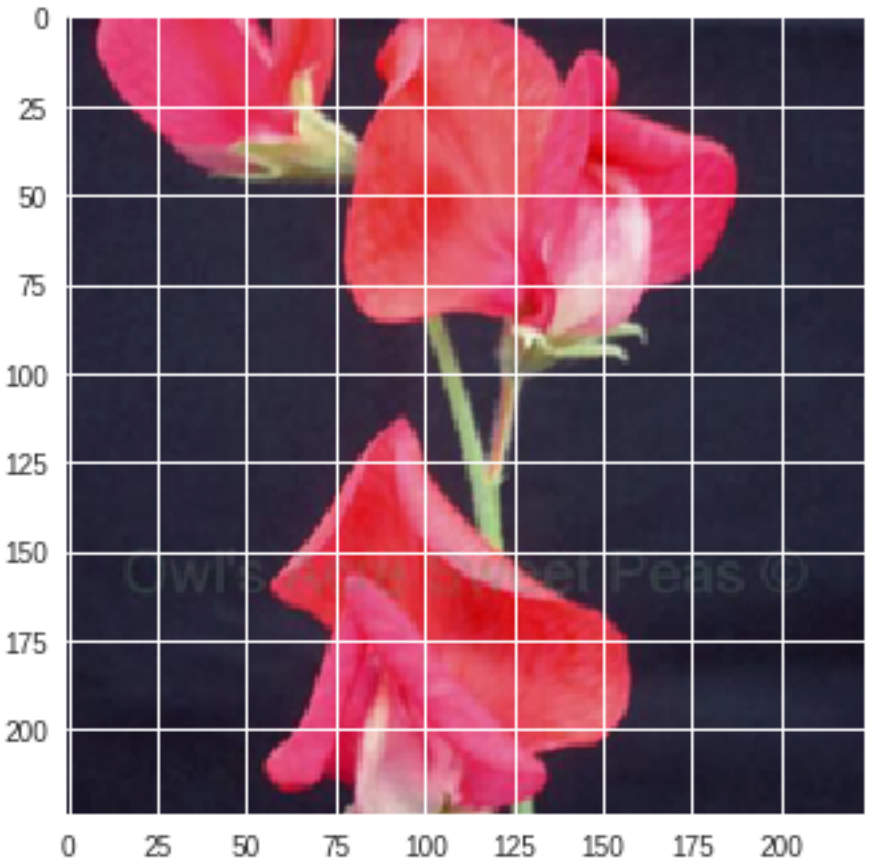
```
In [0]: def process_image(image):
            ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
                returns an Numpy array
            '''

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

            # Resize
            if img.size[0] > img.size[1]: #if width > height
                img.thumbnail((10000, 256)) #constrain the height to be 256, width maximum (wi
            else:
                img.thumbnail((256, 10000)) #otherwise constrain width

            # Crop center 224x224
            left_margin = (img.width-224)/2
            bottom_margin = (img.height-224)/2
            right_margin = left_margin + 224
            top_margin = bottom_margin + 224

            img = img.crop((left_margin, bottom_margin, right_margin,
                            top_margin))

            # Normalize
            img = np.array(img)/255 #color in [0,1] instead of [0,255]
            mean = np.array([0.485, 0.456, 0.406]) #provided mean
            std = np.array([0.229, 0.224, 0.225]) #provided std
            img = (img - mean)/std
```

```
        # Move color channels to first dimension as expected by PyTorch
        img = img.transpose((2, 0, 1))

        return img
```

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

```
In [0]: 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.transpose((1, 2, 0))

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

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

            ax.imshow(image)

            return ax
```

```
In [15]: # function check
         image_path = 'flower_data/train/4/image_05629.jpg'
         img_test = process_image(image_path)
         imshow(img_test);
```



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 2.1). 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 [0]: def predict(image_path, model, top_num=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
    # Process image
    img_pred = process_image(image_path)

    # Numpy -> Tensor
    image_tensor = torch.from_numpy(img_pred).type(torch.FloatTensor)

    # Add batch of size 1 to image
    model_input = image_tensor.unsqueeze(0)

    # Probs
    probs = torch.exp(model.forward(model_input))

    # Top probs
    top_probs, top_labs = probs.topk(top_num)
    top_probs = top_probs.detach().numpy().tolist()[0]
    top_labs = top_labs.detach().numpy().tolist()[0]

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

    top_labels = [idx_to_class[lab] for lab in top_labs]
    top_flowers = [cat_to_name[idx_to_class[lab]] for lab in top_labs]

    return top_probs, top_labels, top_flowers
```

3.3 Sanity Checking

Now that you can use a trained model for predictions, check to make sure it makes sense. Even if the validation 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:

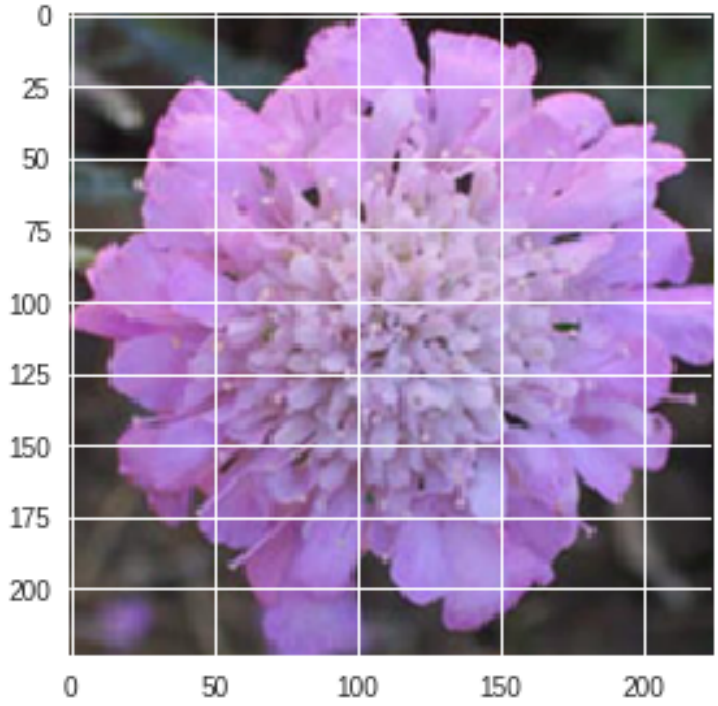
You can convert from the class integer encoding to actual flower names with the `cat_to_name.json` file (should have been loaded earlier in the notebook). To show a PyTorch tensor as an image, use the `imshow` function defined above.

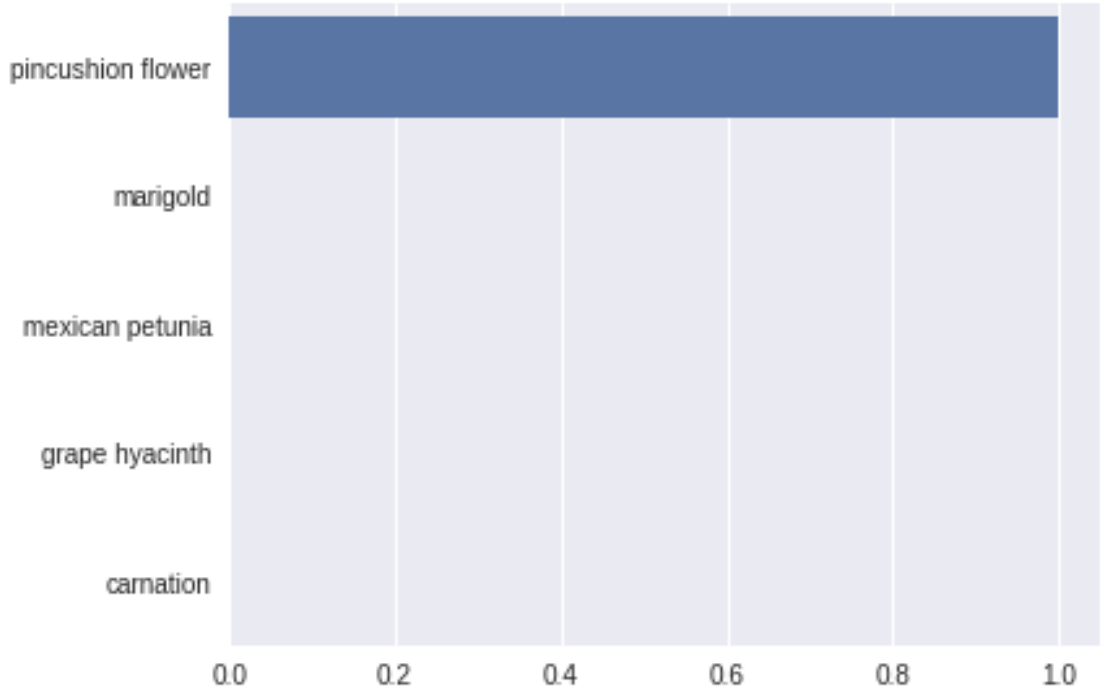
```
In [0]: # TODO: Display an image along with the top 5 classes
        def plot_solution(image_path, model):
            # Set up plot
            plt.figure(figsize = (6,10))
            ax = plt.subplot(2,1,1)

            # Set up title
            flower_num = image_path.split('/')[2]
            title_ = cat_to_name[flower_num]
```

```
            print('This flower is a '+title_+', folder number is '+str(flower_num))

            # Plot flower
            img = process_image(image_path)
            imshow(img, ax);

            # Make prediction
            probs, labs, flowers = predict(image_path, model)

            # Plot bar chart
            plt.subplot(2,1,2)
            seaborn.barplot(x=probs, y=flowers, color=seaborn.color_palette()[0]);
            plt.show()
```

```
In [18]: image_path = 'flower_data/train/22/image_05355.jpg'
         plot_solution(image_path, model)
```

```
This flower is a pincushion flower, folder number is 22


/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:1428: FutureWarning: remove_na is
  stat_data = remove_na(group_data)
```





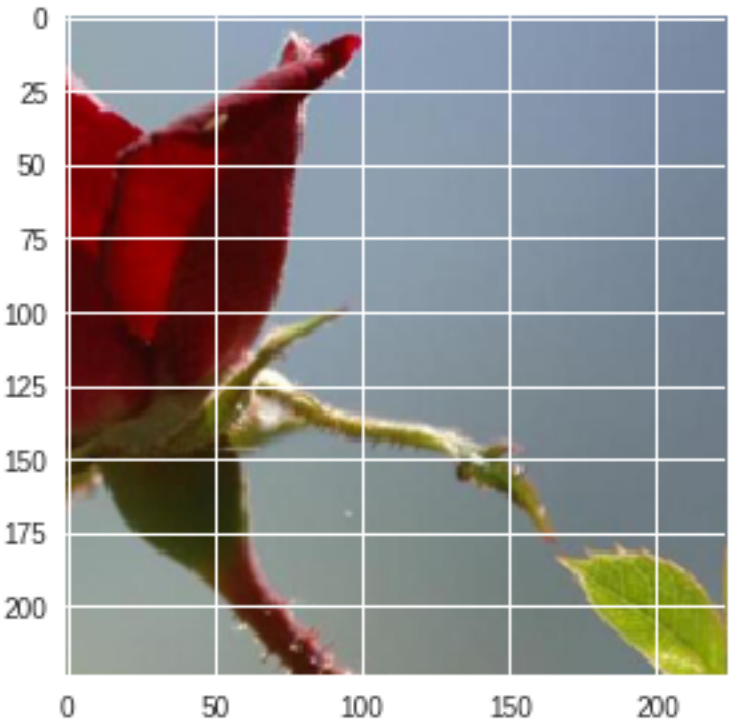

```
In [0]: def scan_folder(parent):
            current_path = parent
            num_folders = len(os.listdir(parent))
```

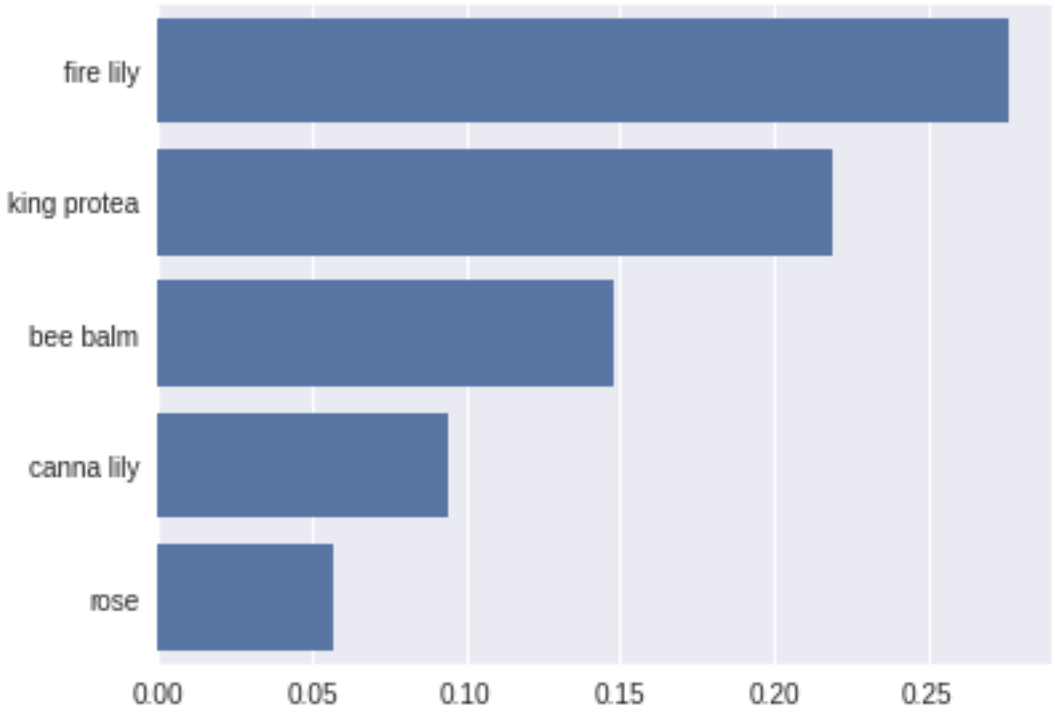
```
            # iterate over all the files in directory 'parent'
            count = 3 if num_folders >= 3 else num_folders
            for file_name in random.sample(os.listdir(parent), count):
                if file_name.endswith(".jpg"):
                    # if it's a jpg file, print its name (or do whatever you want)
                    # print(file_name)
                    whole_path = current_path + '/' + file_name
                    plot_solution(whole_path, model)
                else:
                    current_path = "".join((parent, "/", file_name))
                    if os.path.isdir(current_path):
                        # if we're checking a sub-directory, recall this method
                        print('Folder changed to: {}'.format(current_path))
                        scan_folder(current_path)
```

```
In [20]: scan_folder('flower_data/valid')  # Insert parent direcotry's path
```

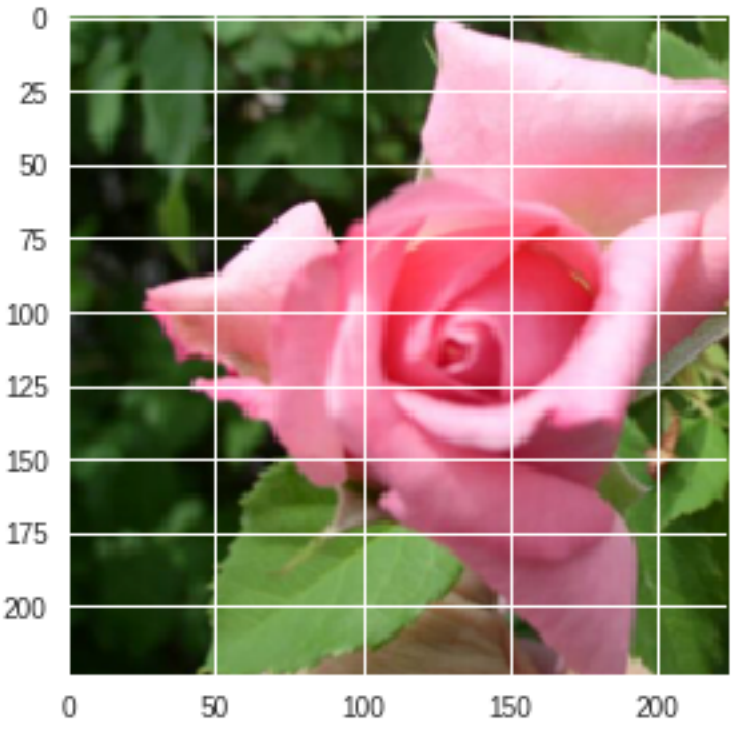
```
Folder changed to: flower_data/valid/74
This flower is a rose, folder number is 74
```

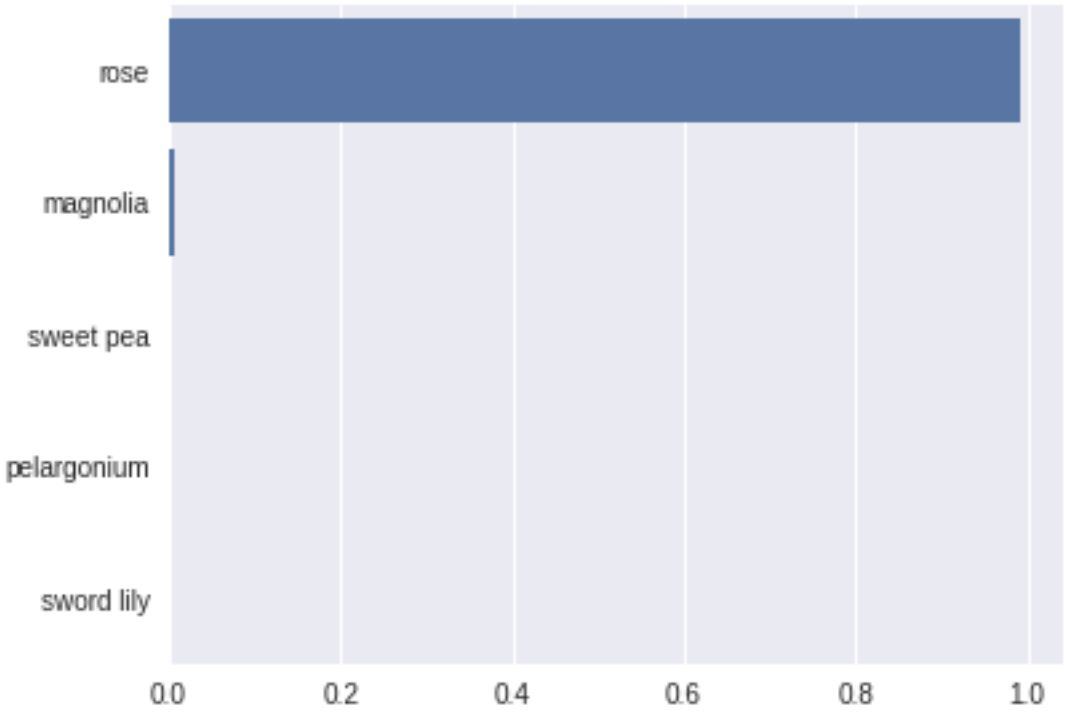
```
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:1428: FutureWarning: remove_na is
  stat_data = remove_na(group_data)
```



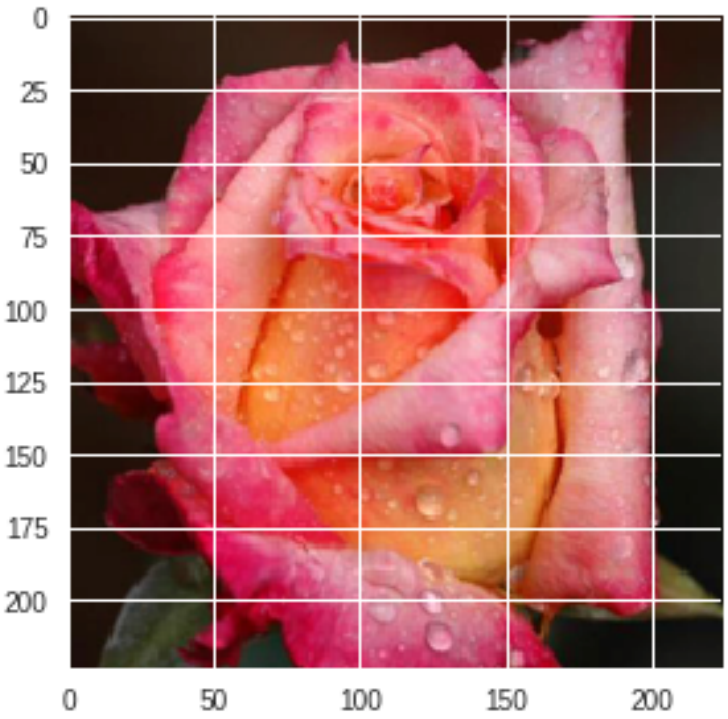


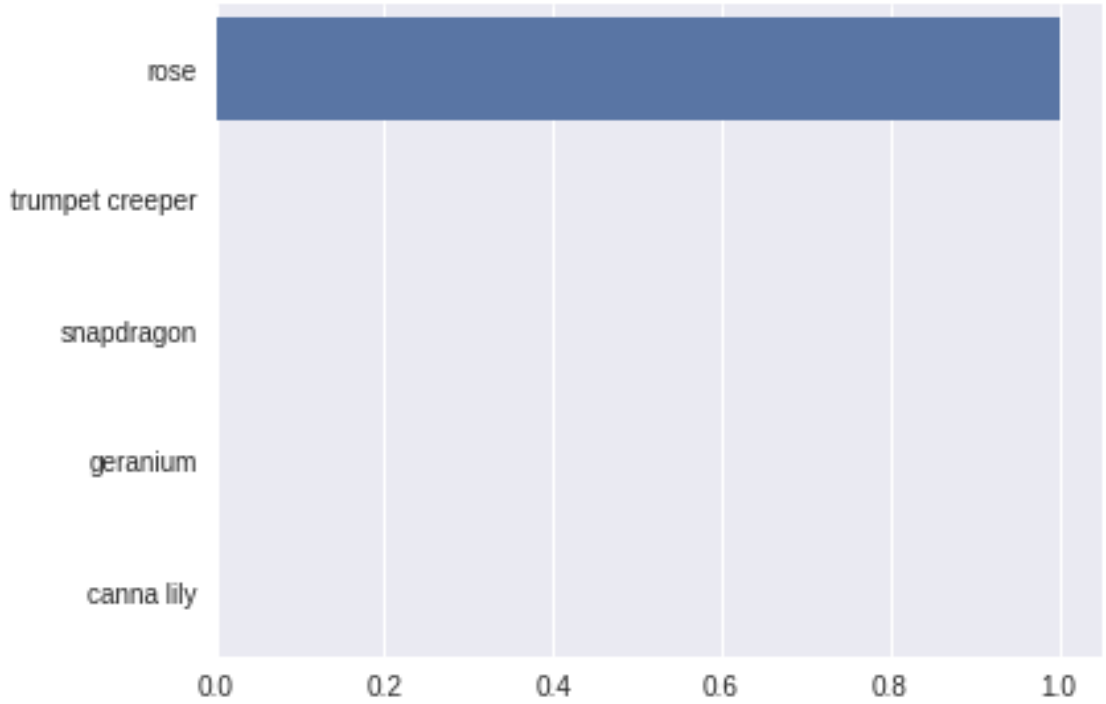

```
This flower is a rose, folder number is 74
```



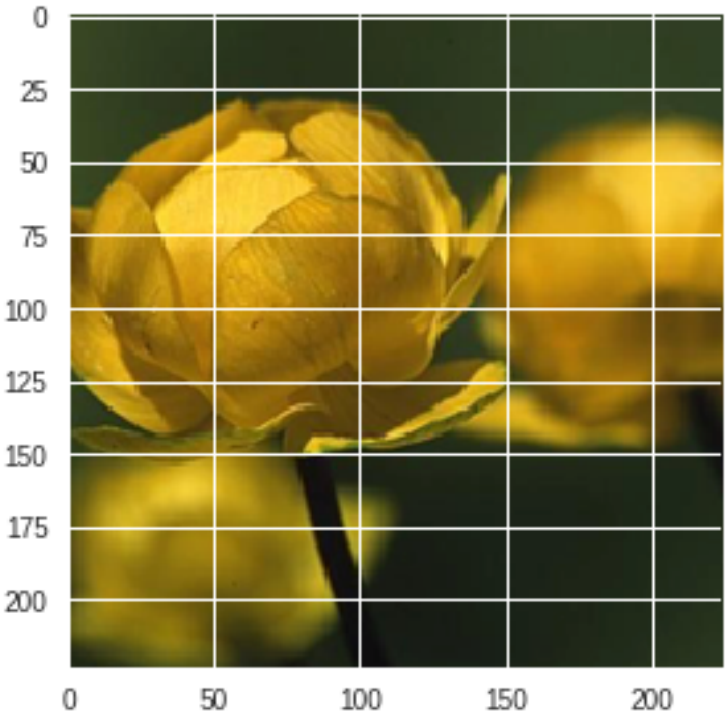


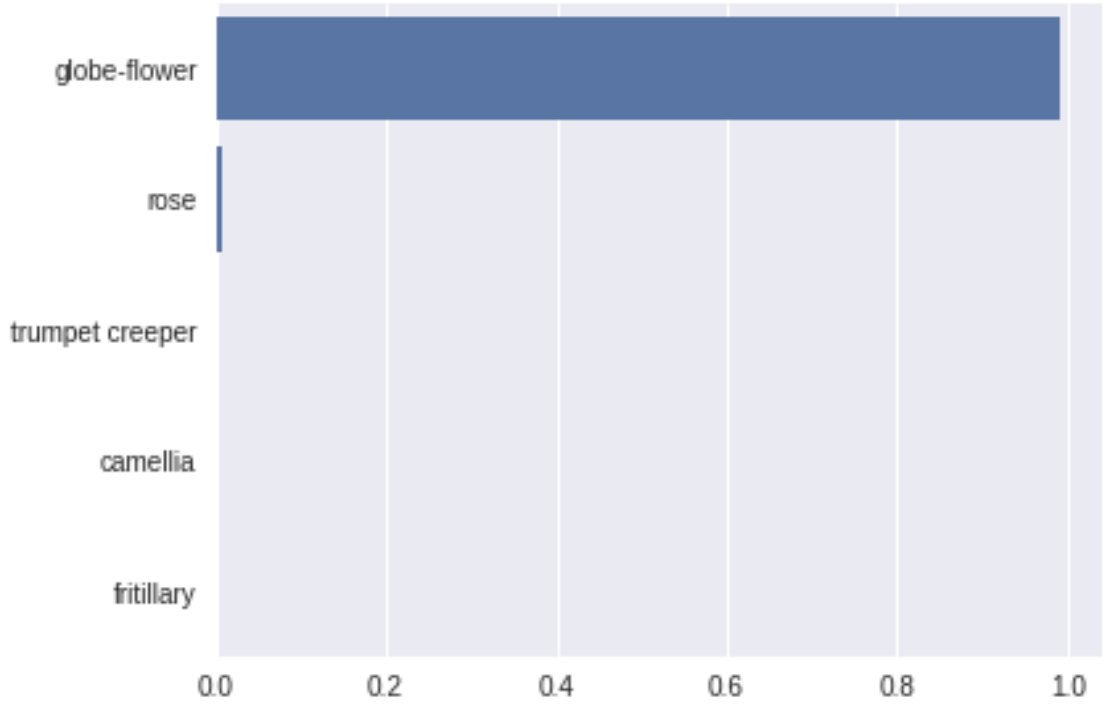

```
This flower is a rose, folder number is 74
```



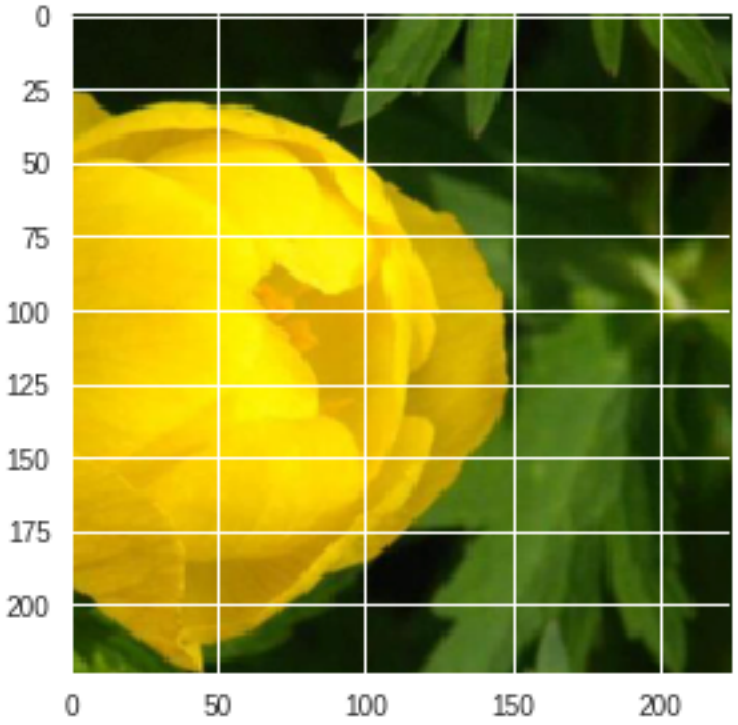


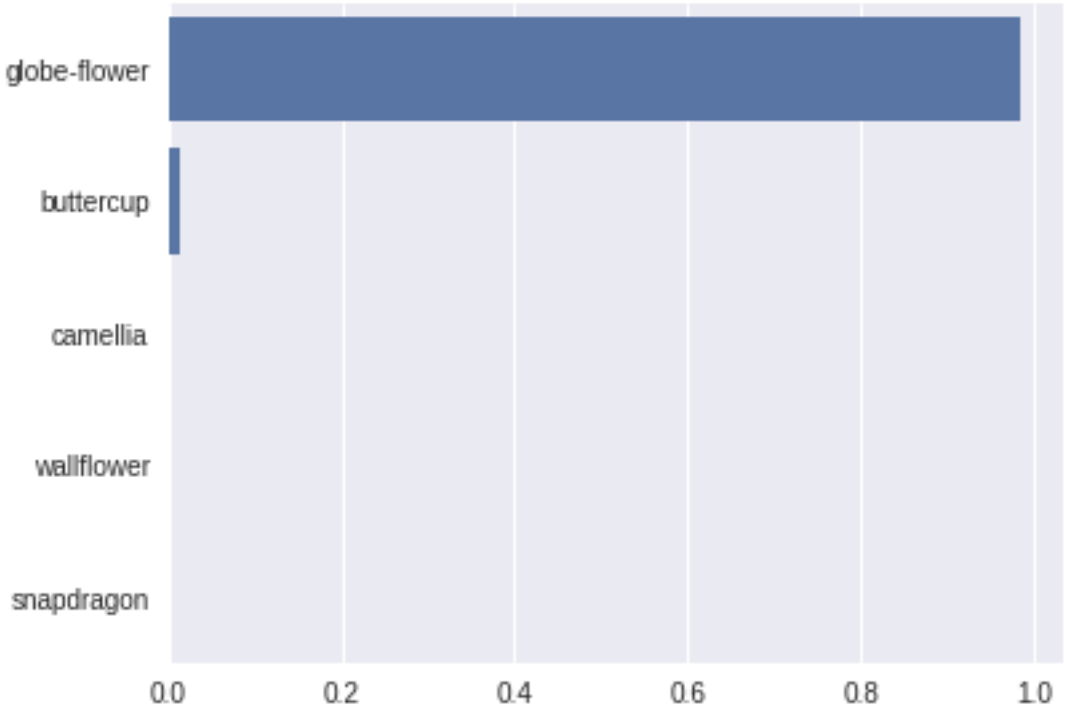

```
Folder changed to: flower_data/valid/16
This flower is a globe-flower, folder number is 16
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200

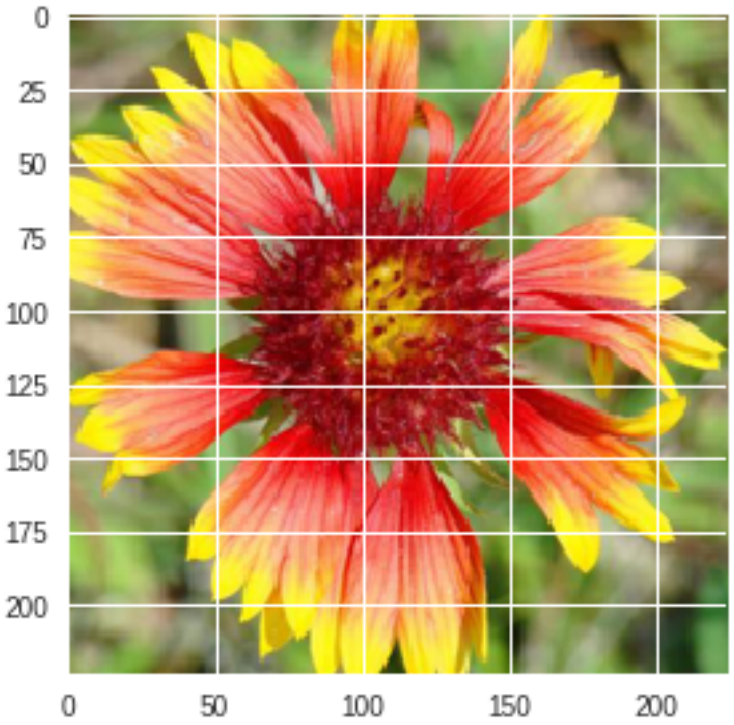

globe-flower
rose
trumpet creeper
camellia
fritillary
0.0
0.2
0.4
0.6
0.8
1.0

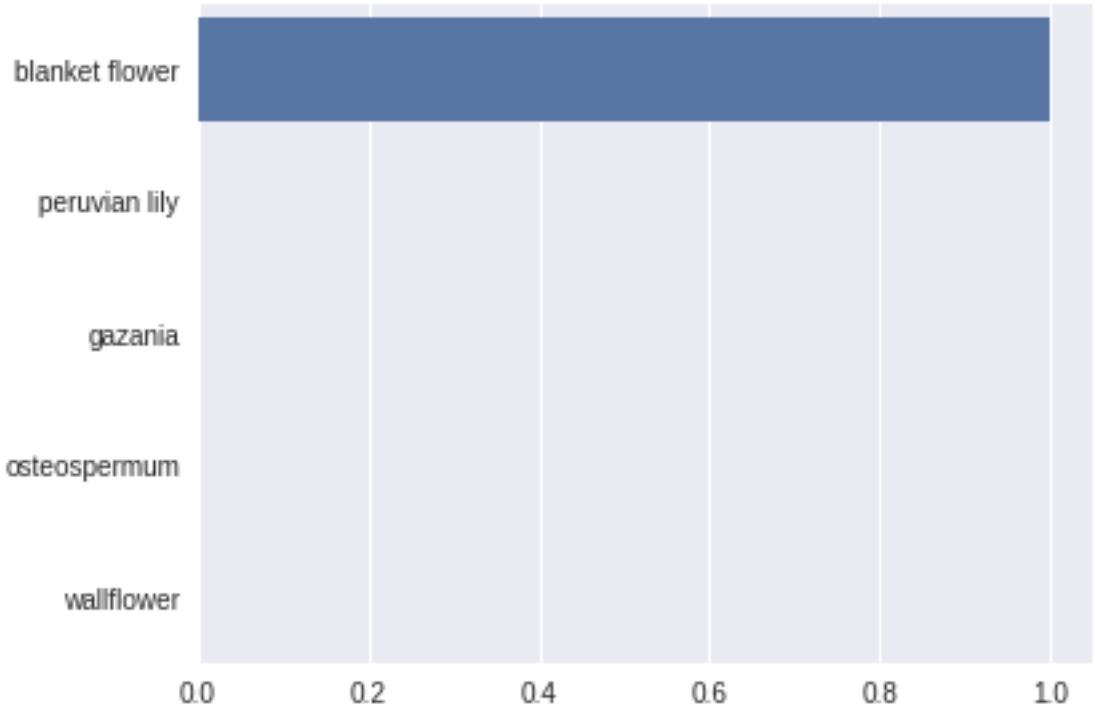
```
This flower is a globe-flower, folder number is 16
```



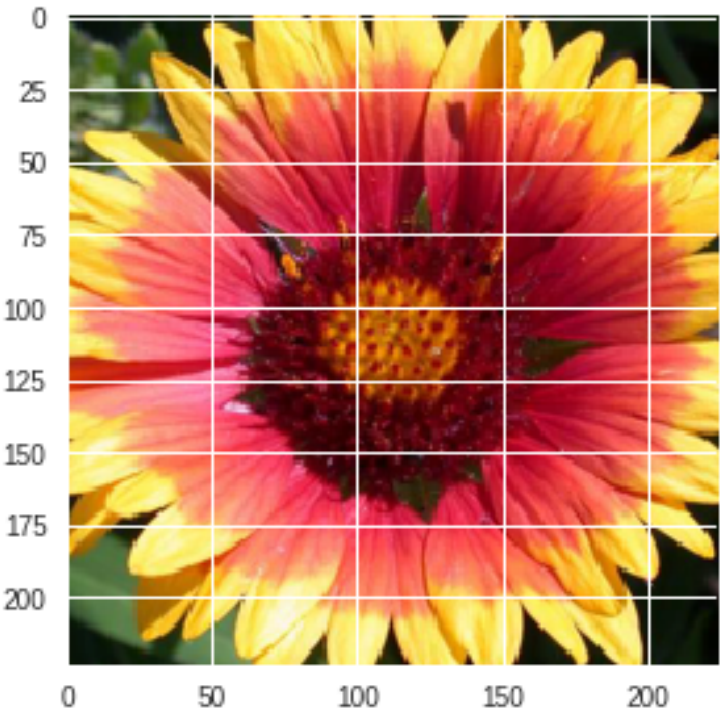


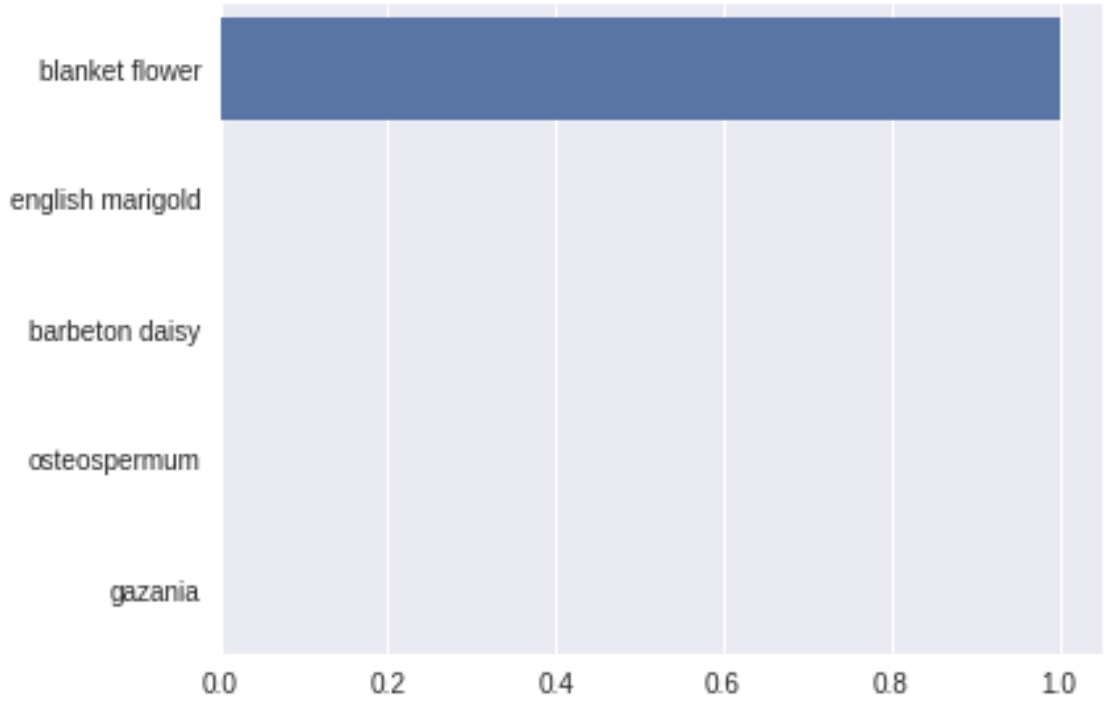
```
Folder changed to: flower_data/valid/100
This flower is a blanket flower, folder number is 100
```



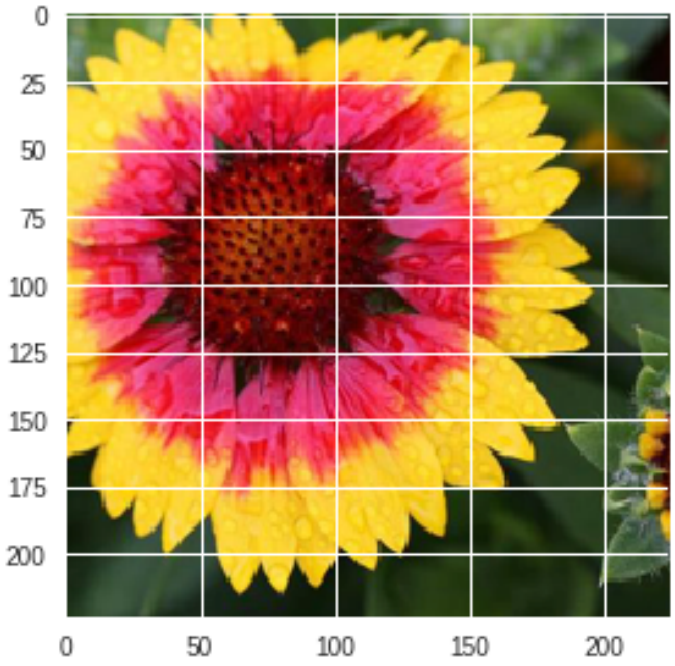


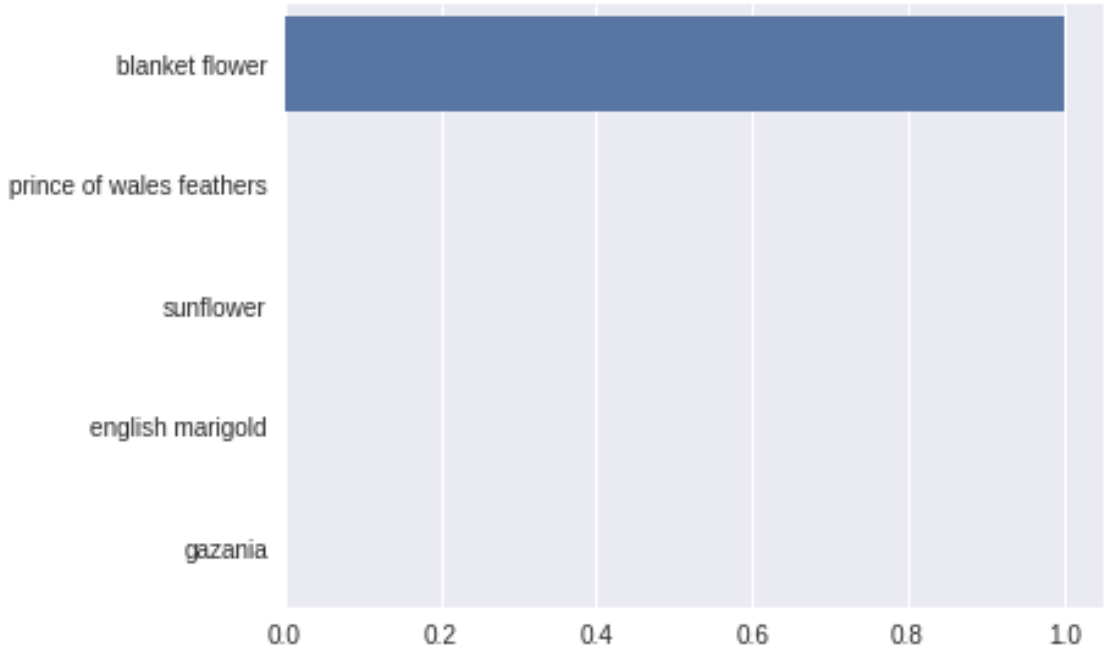
```
This flower is a blanket flower, folder number is 100
```





```
This flower is a blanket flower, folder number is 100
```






```
In [0]:
```