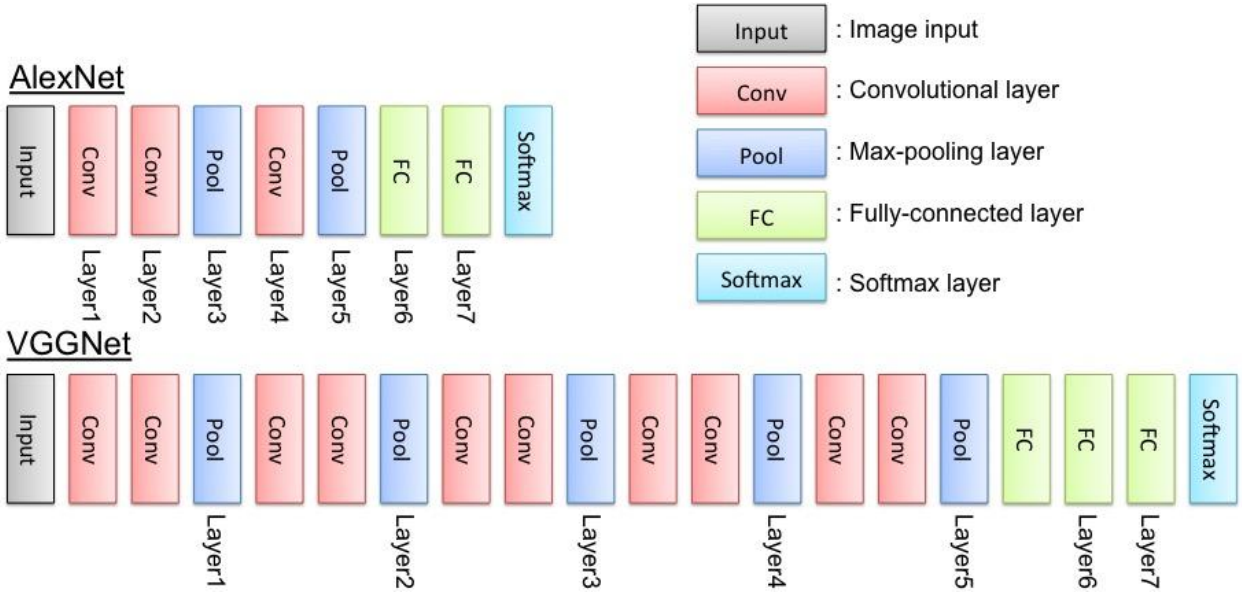
Transfer Learning

Most of the time you won't want to train a whole convolutional network yourself. Modern ConvNets training on huge datasets like ImageNet take weeks on multiple GPUs. Instead, most people use a pretrained network either as a fixed feature extractor, or as an initial network to fine tune. In this notebook, you'll be using VGGNet trained on the ImageNet dataset as a feature extractor. Below is a diagram of the VGGNet architecture.



VGGNet is great because it's simple and has great performance, coming in second in the ImageNet competition. The idea here is that we keep all the convolutional layers, but replace the final fully connected layers with our own classifier. This way we can use VGGNet as a feature extractor for our images then easily train a simple classifier on top of that. What we'll do is take the first fully connected layer with 4096 units, including thresholding with ReLUs. We can use those values as a code for each image, then build a classifier on top of those codes.

You can read more about transfer learning from the CS231n course notes.

Pretrained VGGNet

We'll be using a pretrained network from https://github.com/machrisaa/tensorflow-vgg.

This is a really nice implementation of VGGNet, quite easy to work with. The network has already been trained and the parameters are available from this link.

In [3]:

```
from urllib.request import urlretrieve
from os.path import isfile, isdir
from tqdm import tqdm

vgg_dir = 'tensorflow_vgg/'
# Make sure vgg exists
if not isdir(vgg_dir):
    raise Exception("VGG directory doesn't exist!")

class DLProgress(tqdm):
    last_block = 0

    def hook(self, block_num=1, block_size=1, total_size=None):
        self.total = total_size
```

```
        self.update((block_num - self.last_block) * block_size)
        self.last_block = block_num

if not isfile(vgg_dir + "vgg16.npy"):
    with DLProgress(unit='B', unit_scale=True, miniters=1, desc='VGG16 Parameters') as pbar:
        urlretrieve(
            'https://s3.amazonaws.com/content.udacity-data.com/nd101/vgg16.npy',
            vgg_dir + 'vgg16.npy',
            pbar.hook)
else:
    print("Parameter file already exists!")
```

```
Parameter file already exists!
```

Flower power

Here we'll be using VGGNet to classify images of flowers. To get the flower dataset, run the cell below. This dataset comes from the TensorFlow inception tutorial.

In [4]:

```
import tarfile

dataset_folder_path = 'flower_photos'

class DLProgress(tqdm):
    last_block = 0

    def hook(self, block_num=1, block_size=1, total_size=None):
        self.total = total_size
        self.update((block_num - self.last_block) * block_size)
        self.last_block = block_num

if not isfile('flower_photos.tar.gz'):
    with DLProgress(unit='B', unit_scale=True, miniters=1, desc='Flowers Dataset') as pbar:
        urlretrieve(
            'http://download.tensorflow.org/example_images/flower_photos.tgz',
            'flower_photos.tar.gz',
            pbar.hook)

if not isdir(dataset_folder_path):
    with tarfile.open('flower_photos.tar.gz') as tar:
        tar.extractall()
        tar.close()
```

ConvNet Codes

Below, we'll run through all the images in our dataset and get codes for each of them. That is, we'll run the images through the VGGNet convolutional layers and record the values of the first fully connected layer. We can then write these to a file for later when we build our own classifier.
Here we're using the `vgg16` module from `tensorflow_vgg`. The network takes images of size $244 \times 224 \times 3$ 244×224×3 as input. Then it has 5 sets of convolutional layers. The network implemented here has this structure (copied from the source code:

```
    self.conv1_1 = self.conv_layer(bgr, "conv1_1")
    self.conv1_2 = self.conv_layer(self.conv1_1, "conv1_2")
    self.pool1   = self.max_pool(self.conv1_2, 'pool1')

    self.conv2_1 = self.conv_layer(self.pool1, "conv2_1")
    self.conv2_2 = self.conv_layer(self.conv2_1, "conv2_2")
    self.pool2   = self.max_pool(self.conv2_2, 'pool2')

    self.conv3_1 = self.conv_layer(self.pool2, "conv3_1")
    self.conv3_2 = self.conv_layer(self.conv3_1, "conv3_2")
    self.conv3_3 = self.conv_layer(self.conv3_2, "conv3_3")
    self.pool3   = self.max_pool(self.conv3_3, 'pool3')

    self.conv4_1 = self.conv_layer(self.pool3, "conv4_1")
    self.conv4_2 = self.conv_layer(self.conv4_1, "conv4_2")
    self.conv4_3 = self.conv_layer(self.conv4_2, "conv4_3")
    self.pool4   = self.max_pool(self.conv4_3, 'pool4')

    self.conv5_1 = self.conv_layer(self.pool4, "conv5_1")
    self.conv5_2 = self.conv_layer(self.conv5_1, "conv5_2")
    self.conv5_3 = self.conv_layer(self.conv5_2, "conv5_3")
    self.pool5   = self.max_pool(self.conv5_3, 'pool5')

    self.fc6     = self.fc_layer(self.pool5, "fc6")
    self.relu6   = tf.nn.relu(self.fc6)
```

So what we want are the values of the first fully connected layer, after being ReLUd (`self.relu6`). To build the network, we use

```
    with tf.Session() as sess:

        vgg = vgg16.Vgg16()

        input_ = tf.placeholder(tf.float32, [None, 224, 224, 3])

        with tf.name_scope("content_vgg"):

            vgg.build(input_)
```

This creates the `vgg` object, then builds the graph with `vgg.build(input_)`. Then to get the values from the layer,

```
    feed_dict = {input_: images}

    codes = sess.run(vgg.relu6, feed_dict=feed_dict)
```

In [5]:

```
import os

import numpy as np
import tensorflow as tf

from tensorflow_vgg import vgg16
from tensorflow_vgg import utils
```

In [6]:

```
data_dir = 'flower_photos/'
contents = os.listdir(data_dir)
classes = [each for each in contents if os.path.isdir(data_dir + each)]
```

Below I'm running images through the VGG network in batches.

In [7]:

```
# Set the batch size higher if you can fit in in your GPU memory
batch_size = 10
codes_list = []
labels = []
batch = []

codes = None

with tf.Session() as sess:
    vgg = vgg16.Vgg16()
    input_ = tf.placeholder(tf.float32, [None, 224, 224, 3])

    with tf.name_scope("content_vgg"):
        vgg.build(input_)

    for each in classes:
        print("Starting {} images".format(each))
        class_path = data_dir + each
        files = os.listdir(class_path)

        for ii, file in enumerate(files, 1):
            # Add images to the current batch
            # utils.load_image crops the input images for us, from the center
            img = utils.load_image(os.path.join(class_path, file))
            batch.append(img.reshape((1, 224, 224, 3)))
            labels.append(each)

            # Running the batch through the network to get the codes
            if ii % batch_size == 0 or ii == len(files):
                images = np.concatenate(batch)

                feed_dict = {input_: images}
                codes_batch = sess.run(vgg.relu6, feed_dict=feed_dict)

                # Here I'm building an array of the codes
                if codes is None:
                    codes = codes_batch
                else:
                    codes = np.concatenate((codes, codes_batch))

                # Reset to start building the next batch
                batch = []
                print('{} images processed'.format(ii))
```

```
C:\Users\VadymSerpak\transfer-learning\tensorflow_vgg\vgg16.npy
npy file loaded
build model started
build model finished: 0s
Starting daisy images
10 images processed
……
799 images processed
```

In [8]:

```
# write codes to file
with open('codes', 'w') as f:
    codes.tofile(f)

# write labels to file
import csv
with open('labels', 'w') as f:
    writer = csv.writer(f, delimiter='\n')
    writer.writerow(labels)
```

Building the Classifier

Now that we have codes for all the images, we can build a simple classifier on top of them. The codes behave just like normal input into a simple neural network. Below I'm going to have you do most of the work.

In [9]:

```
# read codes and labels from file
import csv

with open('labels') as f:
    reader = csv.reader(f, delimiter='\n')
    labels = np.array([each for each in reader if len(each) > 0]).squeeze()
with open('codes') as f:
    codes = np.fromfile(f, dtype=np.float32)
    codes = codes.reshape((len(labels), -1))
```

Data prep

As usual, now we need to one-hot encode our labels and create validation/test sets. First up, creating our labels!

Exercise: From scikit-learn, use LabelBinarizer to create one-hot encoded vectors from the labels.

In [10]:

```
from sklearn.preprocessing import LabelBinarizer

lb = LabelBinarizer()
lb.fit(labels)

labels_vecs = lb.transform(labels)
```

Now you'll want to create your training, validation, and test sets. An important thing to note here is that our labels and data aren't randomized yet. We'll want to shuffle our data so the validation and test sets contain data from all classes. Otherwise, you could end up with testing sets that are all one class. Typically, you'll also want to make sure that each smaller set has the same the distribution of classes as it is for the whole data set. The easiest way to accomplish both these goals is to use `StratifiedShuffleSplit` from scikit-learn.

You can create the splitter like so:

```
    ss = StratifiedShuffleSplit(n_splits=1, test_size=0.2)
```

Then split the data with

```
    splitter = ss.split(x, y)
```

`ss.split` returns a generator of indices. You can pass the indices into the arrays to get the split sets. The fact that it's a generator means you either need to iterate over it, or use `next(splitter)` to get the indices. Be sure to read the documentation and the user guide.
Exercise: Use StratifiedShuffleSplit to split the codes and labels into training, validation, and test sets.

In [11]:

```
from sklearn.model_selection import StratifiedShuffleSplit

ss = StratifiedShuffleSplit(n_splits=1, test_size=0.2)

train_idx, val_idx = next(ss.split(codes, labels))

half_val_len = int(len(val_idx)/2)
val_idx, test_idx = val_idx[:half_val_len], val_idx[half_val_len:]

train_x, train_y = codes[train_idx], labels_vecs[train_idx]
val_x, val_y = codes[val_idx], labels_vecs[val_idx]
test_x, test_y = codes[test_idx], labels_vecs[test_idx]
```

In [12]:

```
print("Train shapes (x, y):", train_x.shape, train_y.shape)
print("Validation shapes (x, y):", val_x.shape, val_y.shape)
print("Test shapes (x, y):", test_x.shape, test_y.shape)
```

```
Train shapes (x, y): (2936, 4096) (2936, 5)
Validation shapes (x, y): (367, 4096) (367, 5)
Test shapes (x, y): (367, 4096) (367, 5)
```

Classifier layers

Once you have the convolutional codes, you just need to build a classfier from some fully connected layers. You use the codes as the inputs and the image labels as targets. Otherwise the classifier is a typical neural network.

> **Exercise:** With the codes and labels loaded, build the classifier. Consider the codes as your inputs, each of them are 4096D vectors. You'll want to use a hidden layer and an output layer as your classifier. Remember that the output layer needs to have one unit for each class and a softmax activation function. Use the cross entropy to calculate the cost.

In [13]:

```
inputs_ = tf.placeholder(tf.float32, shape=[None, codes.shape[1]])
labels_ = tf.placeholder(tf.int64, shape=[None, labels_vecs.shape[1]])

fc = tf.contrib.layers.fully_connected(inputs_, 256)

logits = tf.contrib.layers.fully_connected(fc, labels_vecs.shape[1],
activation_fn=None)
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=labels_,
logits=logits)
cost = tf.reduce_mean(cross_entropy)

optimizer = tf.train.AdamOptimizer().minimize(cost)

predicted = tf.nn.softmax(logits)
correct_pred = tf.equal(tf.argmax(predicted, 1), tf.argmax(labels_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Batches!

Here is just a simple way to do batches. I've written it so that it includes all the data. Sometimes you'll throw out some data at the end to make sure you have full batches. Here I just extend the last batch to include the remaining data.

In [14]:

```
def get_batches(x, y, n_batches=10):
    """ Return a generator that yields batches from arrays x and y. """
    batch_size = len(x)//n_batches

    for ii in range(0, n_batches*batch_size, batch_size):
        # If we're not on the last batch, grab data with size batch_size
        if ii != (n_batches-1)*batch_size:
            X, Y = x[ii: ii+batch_size], y[ii: ii+batch_size]
        # On the last batch, grab the rest of the data
        else:
            X, Y = x[ii:], y[ii:]
        # I love generators
        yield X, Y
```

Training

Here, we'll train the network.

> **Exercise:** So far we've been providing the training code for you. Here, I'm going to give you a bit more of a challenge and have you write the code to train the network. Of course, you'll be able to see my solution if you need help.

In [15]:

```
epochs = 10
iteration = 0
saver = tf.train.Saver()
with tf.Session() as sess:

    sess.run(tf.global_variables_initializer())
    for e in range(epochs):
        for x, y in get_batches(train_x, train_y):
            feed = {inputs_: x,
                    labels_: y}
            loss, _ = sess.run([cost, optimizer], feed_dict=feed)
            print("Epoch: {}/{}".format(e+1, epochs),
                  "Iteration: {}".format(iteration),
                  "Training loss: {:.5f}".format(loss))
            iteration += 1

            if iteration % 5 == 0:
                feed = {inputs_: val_x,
                        labels_: val_y}
                val_acc = sess.run(accuracy, feed_dict=feed)
                print("Epoch: {}/{}".format(e, epochs),
                      "Iteration: {}".format(iteration),
                      "Validation Acc: {:.4f}".format(val_acc))
    saver.save(sess, "checkpoints/flowers.ckpt")
```

```
Epoch: 1/10 Iteration: 0 Training loss: 5.80969
……
Epoch: 9/10 Iteration: 100 Validation Acc: 0.8801
```

Testing

Below you see the test accuracy. You can also see the predictions returned for images.

In [16]:

```
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('checkpoints'))

    feed = {inputs_: test_x,
            labels_: test_y}
    test_acc = sess.run(accuracy, feed_dict=feed)
    print("Test accuracy: {:.4f}".format(test_acc))
```

```
Test accuracy: 0.9019
```

In [17]:

```
%matplotlib inline

import matplotlib.pyplot as plt
from scipy.ndimage import imread
```

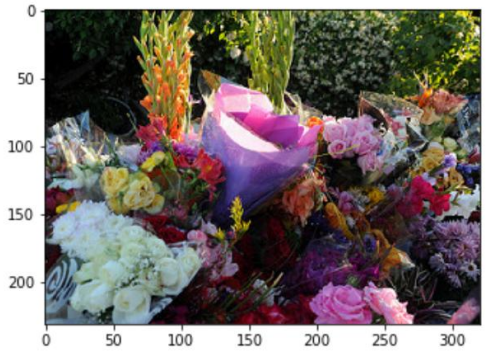
Below, feel free to choose images and see how the trained classifier predicts the flowers in them.

In [18]:

```
test_img_path = 'flower_photos/roses/10894627425_ec76bbc757_n.jpg'
test_img = imread(test_img_path)
plt.imshow(test_img)
```

Out[18]:

```
<matplotlib.image.AxesImage at 0x22b80d74278>
```



In [19]:

```
# Run this cell if you don't have a vgg graph built
with tf.Session() as sess:
    input_ = tf.placeholder(tf.float32, [None, 224, 224, 3])
    vgg = vgg16.Vgg16()
    vgg.build(input_)
```

```
C:\Users\VadymSerpak\transfer-learning\tensorflow_vgg\vgg16.npy
npy file loaded
build model started
build model finished: 0s
```

In [20]:

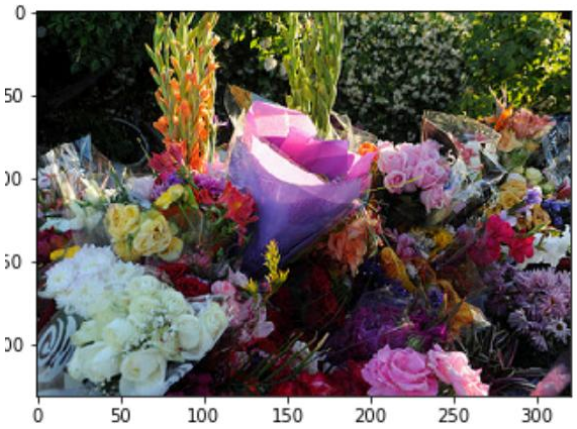
```
with tf.Session() as sess:
    img = utils.load_image(test_img_path)
    img = img.reshape((1, 224, 224, 3))

    feed_dict = {input_: img}
    code = sess.run(vgg.relu6, feed_dict=feed_dict)

saver = tf.train.Saver()
with tf.Session() as sess:
    saver.restore(sess, tf.train.latest_checkpoint('checkpoints'))

    feed = {inputs_: code}
    prediction = sess.run(predicted, feed_dict=feed).squeeze()
```

In [21]:

```
plt.imshow(test_img)
```

Out[21]:

```
<matplotlib.image.AxesImage at 0x22b811c9f98>
```



In [22]:

```
plt.barh(np.arange(5), prediction)
_ = plt.yticks(np.arange(5), lb.classes_)
```