1. Import Python libraries

A honey bee (Apis).

Can a machine identify a bee as a honey bee or a bumble bee? These bees have different [behaviors and appearances](#), but given the variety of backgrounds, positions, and image resolutions, it can be a challenge for machines to tell them apart.

Being able to identify bee species from images is a task that ultimately would allow researchers to more quickly and effectively collect field data. Pollinating bees have critical roles in both ecology and agriculture, and diseases like [colony collapse disorder](#) threaten these species. Identifying different species of bees in the wild means that we can better understand the prevalence and growth of these important insects.

A bumble bee (Bombus).

This notebook walks through building a simple deep learning model that can automatically detect honey bees and bumble bees and then loads a pre-trained model for evaluation.

In [2]:

```python
import pickle
from pathlib import Path
from skimage import io

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# import keras library
import keras

# import Sequential from the keras models module
from keras.models import Sequential

# import Dense, Dropout, Flatten, Conv2D, MaxPooling2D from the keras layers module
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
```

```
Using TensorFlow backend.
```

In [3]:

```
%%pass
```

```
%%nose

def test_task_1():
    assert 'keras' in globals(), \
    'Did you forget to import keras?'

def test_task_2():
    assert 'Sequential' in globals(), \
    'Did you forget to import Sequential from keras.models?'

def test_task_3():
    for layer_type in ['Dense', 'Dropout', 'Flatten', 'Conv2D', 'MaxPooling2D']:
        assert layer_type in globals(), \
    'Did you forget to import Dense, Dropout, Flatten, Conv2D, or MaxPooling2D from keras.layers?'
```

Out[3]:

```
3/3 tests passed
```

2. Load image labels

Now that we have all of our imports ready, it is time to look at the labels for our data. We will load our `labels.csv` file into a DataFrame called `labels`, where the index is the image name (e.g. an index of 1036 refers to an image named 1036.jpg) and the `genus` column tells us the bee type. `genus` takes the value of either `0.0` (Apis or honey bee) or `1.0` (Bombus or bumble bee).

In [4]:

```
# load labels.csv from datasets folder using pandas
labels = pd.read_csv('datasets/labels.csv', index_col=0)

# print value counts for genus
print(labels.genus.value_counts())

# assign the genus label values to y
y = labels.genus.values
```

```
0.0    827
1.0    827
Name: genus, dtype: int64
```

In [5]:

```
%%nose

def test_task2_0():
    assert labels.shape == (1654, 1), \
    'Did you remember to set index_col=0 within the pd.read_csv() function?'

def test_task2_1():
    try:
        assert y[13] == 1.0 and len(y) == 1654
    except KeyError:
        assert False, "The data was not assigned to y correctly. Index 13, which doesn't exist in
y currently, should correspond to the value 1.0."
    except AssertionError:
        assert False, "Did you assign labels.genus.values to y?"
```
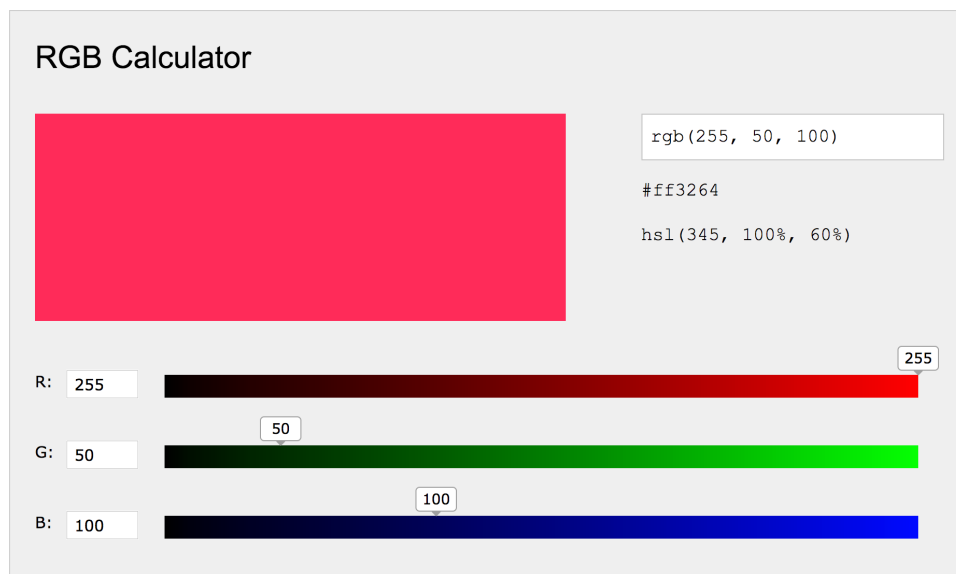
Out[5]:

```
2/2 tests passed
```

3. Examine RGB values in an image matrix

Image data can be represented as a matrix. The width of the matrix is the width of the image, the height of the matrix is the height of the image, and the depth of the matrix is the number of channels. Most image formats have three color channels: red, green, and blue.

For each pixel in an image, there is a value for every channel. The combination of the three values corresponds to the color, as per the RGB color model. Values for each color can range from 0 to 255, so a purely blue pixel would show up as (0, 0, 255).

RGB Calculator

<table>
<tr><td></td><td>rgb(255, 50, 100)</td></tr>
<tr><td></td><td>#ff3264</td></tr>
<tr><td></td><td>hsl(345, 100%, 60%)</td></tr>
</table>

R: 255   255

G: 50

B: 100

Let's explore the data for a sample image.

In [6]:

```python
# load an image and explore
example_image = io.imread('datasets/{}.jpg'.format(labels.index[0]))

# show image
plt.imshow(example_image)

# print shape
print('Image has shape:', example_image.shape)

# print color channel values for top left pixel
print('RGB values for the top left pixel are:', example_image[0, 0, :])
```

```
Image has shape: (50, 50, 3)
RGB values for the top left pixel are: [127 108  95]
```


In [7]:
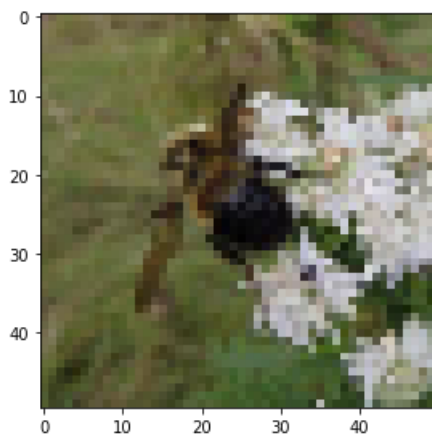
```python
%%nose
import numpy

def test_task3_0():
    assert 'example_image' in globals() and example_image.shape == (50, 50, 3) and
example_image.max() == 236, \
    'Did you load the image using io.imread and assign it to example_image?'
```

Out[7]:

```
1/1 tests passed
```

4. Normalize image data

Now we need to normalize our image data. Normalization is a general term that means changing the scale of our data so it is consistent.

In this case, we want each feature to have a similar range so our neural network can learn effectively across all the features. As explained in the sklearn docs, "If a feature has a variance that is orders of magnitude larger than others, it might dominate the objective function and make the estimator unable to learn from other features correctly as expected."

We will scale our data so that it has a mean of 0 and standard deviation of 1. We'll use sklearn's `StandardScaler` to do the math for us, which entails taking each value, subtracting the mean, and then dividing by the standard deviation. We need to do this for each color channel (i.e. each feature) individually.

In [8]:

```python
# initialize standard scaler
ss = StandardScaler()

image_list = []
for i in labels.index:
    # load image
    img = io.imread('datasets/{}.jpg'.format(i)).astype(np.float64)

    # for each channel, apply standard scaler's fit_transform method
    for channel in range(img.shape[2]):
        img[:, :, channel] = ss.fit_transform(img[:, :, channel])

    # append to list of all images
    image_list.append(img)
# convert image list to single array
X = np.array(image_list)

# print shape of X
print(X.shape)
```

```
(1654, 50, 50, 3)
```

In [9]:

```python
%%nose

# last_value = _

def test_task4_0():
    assert 'StandardScaler' in str(globals()['ss']), \
    'Did you assign StandardScaler() to ss?'

def test_task4_1():
    assert X.shape == (1654, 50, 50, 3), \
    'Did you call np.array on image_list to stack all the images into a matrix and assign it to X?
'

# def test_task4_2():
#     assert last_value == (1654, 50, 50, 3), \
#     'Did you print X.shape?'
```

Out[9]:

```
2/2 tests passed
```

5. Split into train, test, and evaluation sets

Now that we have our big image data matrix, `X`, as well as our labels, `y`, we can split our data into train, test, and evaluation sets. To do this, we'll first allocate 20% of the data into our evaluation, or holdout, set. This is data that the model never sees during training and will be used to score our trained model.

We will then split the remaining data, 60/40, into train and test sets just like in supervised machine learning models. We will pass both the train and test sets into the neural network.

```python
# split out evaluation sets (x_eval and y_eval)
x_interim, x_eval, y_interim, y_eval = train_test_split(X, y, test_size=0.2, random_state=52)

# split remaining data into train and test sets
x_train, x_test, y_train, y_test = train_test_split(x_interim, y_interim, test_size=0.4, random_sta
te=52)

# examine number of samples in train, test, and validation sets
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_eval.shape[0], 'eval samples')
```

```
x_train shape: (793, 50, 50, 3)
793 train samples
530 test samples
331 eval samples
```

```python
%%nose

def test_task5_0():
    assert x_train.shape == (793, 50, 50, 3) and x_test.shape == (530, 50, 50, 3) and x_eval.shape
== (331, 50, 50, 3), \
    '''Did you set test_size=0.2 for the first split into eval and interim sets,
        and then test_size=0.4 for the split into train and test sets?'''

def test_task5_1():
    assert np.all(np.round(x_train[0, 0, 0, :], 2) == [-3.61, -2.38, -1.16]), \
        '''Did you set test_size=0.2 for the first split into eval and interim sets,
        and then test_size=0.4 for the split into train and test sets?'''
```

```
2/2 tests passed
```

6. Model building (part i)

It's time to start building our deep learning model, a convolutional neural network (CNN). CNNs are a specific kind of artificial neural network that is very effective for image classification because they are able to take into account the spatial coherence of the image, i.e., that pixels close to each other are often related.

Building a CNN begins with specifying the model type. In our case, we'll use a Sequential model, which is a linear stack of layers. We'll then add two convolutional layers. To understand convolutional layers, imagine a flashlight being shown over the top left corner of the image and slowly sliding across all the areas of the image, moving across the image in the same way your eyes move across words on a page. Convolutional layers pass a kernel (a sliding window) over the image and perform element-wise matrix multiplication between the kernel values and the pixel values in the image.

```python
# set model constants
num_classes = 1

# define model as Sequential
model = Sequential()

# first convolutional layer with 32 filters
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(50, 50, 3)))

# add a second 2D convolutional layer with 64 filters
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
```

```python
%%nose

def test_task6_0():
```

```
    assert 'num_classes' in globals() and num_classes == 1, \
        'Did you set num_classes equal to 1?'

def test_task6_1():
    assert type(model) == keras.engine.sequential.Sequential, \
        'Did you set model equal to Sequential()?'

def test_task6_2():
    assert (model.layers[1].get_config()['filters'] == 64 and
            model.layers[1].get_config()['kernel_size'] == (3, 3) and
            model.layers[1].get_config()['activation'] == 'relu' and
            type(model.layers[1]) == keras.layers.convolutional.Conv2D), \
        'Did you configure the second convolutional layer as specified in the instructions?'
```

Out[13]:

3/3 tests passed

7. Model building (part ii)

Let's continue building our model. So far our model has two convolutional layers. However, those are not the only layers that we need to perform our task. A complete neural network architecture will have a number of other layers that are designed to play a specific role in the overall functioning of the network. Much deep learning research is about how to structure these layers into coherent systems.

We'll add the following layers:

- `MaxPooling` . This passes a (2, 2) moving window over the image and downscales the image by outputting the maximum value within the window.
- `Conv2D` . This adds a third convolutional layer since deeper models, i.e. models with more convolutional layers, are better able to learn features from images.
- `Dropout` . This prevents the model from overfitting, i.e. perfectly remembering each image, by randomly setting 25% of the input units to 0 at each update during training.
- `Flatten` . As its name suggests, this flattens the output from the convolutional part of the CNN into a one-dimensional feature vector which can be passed into the following fully connected layers.
- `Dense` . Fully connected layer where every input is connected to every output (see image below).
- `Dropout` . Another dropout layer to safeguard against overfitting, this time with a rate of 50%.
- `Dense` . Final layer which calculates the probability the image is either a bumble bee or honey bee.

To take a look at how it all stacks up, we'll print the model summary. Notice that our model has a whopping `3,669,249` paramaters. These are the different weights that the model learns through training and what are used to generate predictions on a new image.
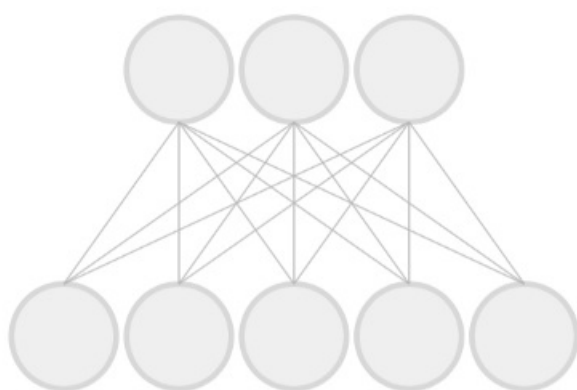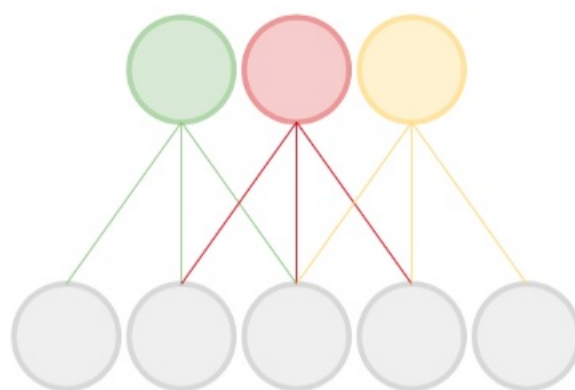
Fully connected layer   Convolutional layer

In [14]:

```
# reduce dimensionality through max pooling
model.add(MaxPooling2D(pool_size=(2, 2)))

# third convolutional layer with 64 filters
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
# add dropout to prevent over fitting
model.add(Dropout(0.25))
# necessary flatten step preceeding dense layer
```

```
model.add(Flatten())
# fully connected layer
model.add(Dense(128, activation='relu'))

# add additional dropout to prevent overfitting
model.add(Dropout(0.5))

# prediction layers
model.add(Dense(num_classes, activation='sigmoid', name='preds'))

# show model summary
model.summary()
```

```
_____
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 48, 48, 32)        896
_____
conv2d_2 (Conv2D)            (None, 46, 46, 64)        18496
_____
max_pooling2d_1 (MaxPooling2 (None, 23, 23, 64)        0
_____
conv2d_3 (Conv2D)            (None, 21, 21, 64)        36928
_____
dropout_1 (Dropout)          (None, 21, 21, 64)        0
_____
flatten_1 (Flatten)          (None, 28224)             0
_____
dense_1 (Dense)              (None, 128)               3612800
_____
dropout_2 (Dropout)          (None, 128)               0
_____
preds (Dense)                (None, 1)                 129
=================================================================
Total params: 3,669,249
Trainable params: 3,669,249
Non-trainable params: 0
_____
```

In [15]:

```
%%nose

def test_task7_0():
    assert (type(model.layers[2]) == keras.layers.pooling.MaxPooling2D and
            model.layers[2].get_config()['pool_size'] == (2, 2)), \
    'Did you add a MaxPooling2D layer with pool size of (2, 2)?'

def test_task7_1():
    assert (type(model.layers[7]) == keras.layers.core.Dropout and
            model.layers[7].get_config()['rate'] == 0.5), \
    'Did you add a Dropout layer with a rate of 0.5`?'

def test_task7_2():
    assert model.layers[8].get_config()['units'] == 1, \
    'Did you pass in num_classes to the final Dense layer?'

def test_task7_3():
    assert model.layers[8].get_config()['activation'] == 'sigmoid', \
    'Did you set sigmoid as the activation in the final Dense layer?'
```

Out[15]:

4/4 tests passed

8. Compile and train model

Now that we've specified the model architecture, we will compile the model for training. For this we need to specify the loss function (what we're trying to minimize), the optimizer (how we want to go about minimizing the loss), and the metric (how we'll judge the performance of the model).

Then, we'll call `.fit` to begin the trainig the process.

> "Neural networks are trained iteratively using optimization techniques like gradient descent. After each cycle of training, an error metric is calculated based on the difference between prediction and target…Each neuron's coefficients (weights) are then adjusted relative to how much they contributed to the total error. This process is repeated iteratively." [ML Cheatsheet](#)

Since training is computationally intensive, we'll do a 'mock' training to get the feel for it, using just the first 10 images in the train and test sets and training for just 5 epochs. Epochs refer to the number of iterations over the data. Typically, neural networks will train for hundreds if not thousands of epochs.

Take a look at the printout for each epoch and note the loss on the train set (`loss`), the accuracy on the train set (`acc`), and loss on the test set (`val_loss`) and the accuracy on the test set (`val_acc`). We'll explore this more in a later step.

In [16]:

```python
model.compile(
    # set the loss as binary_crossentropy
    loss=keras.losses.binary_crossentropy,
    # set the optimizer as stochastic gradient descent
    optimizer=keras.optimizers.SGD(lr=0.001),
    # set the metric as accuracy
    metrics=['accuracy']
)

# mock-train the model using the first ten observations of the train and test sets
model.fit(
    x_train[:10, :, :, :],
    y_train[:10],
    epochs=5,
    verbose=1,
    validation_data=(x_test[:10, :, :, :], y_test[:10])
)
```

```
Train on 10 samples, validate on 10 samples
Epoch 1/5
10/10 [==============================] - 1s 60ms/step - loss: 0.6732 - acc: 0.6000 - val_loss: 0.7
103 - val_acc: 0.5000
Epoch 2/5
10/10 [==============================] - 0s 9ms/step - loss: 0.7900 - acc: 0.4000 - val_loss: 0.71
34 - val_acc: 0.5000
Epoch 3/5
10/10 [==============================] - 0s 9ms/step - loss: 0.6909 - acc: 0.7000 - val_loss: 0.71
57 - val_acc: 0.5000
Epoch 4/5
10/10 [==============================] - 0s 9ms/step - loss: 0.6729 - acc: 0.6000 - val_loss: 0.71
94 - val_acc: 0.5000
Epoch 5/5
10/10 [==============================] - 0s 9ms/step - loss: 0.6372 - acc: 0.5000 - val_loss: 0.72
53 - val_acc: 0.5000
```

Out[16]:

```
<keras.callbacks.History at 0x7f8fb8272ba8>
```

In [17]:

```python
%%nose

def test_task9_0():
    assert 'binary_crossentropy' in str(model.loss), \
    'Did you assign loss to keras.losses.binary_crossentropy?'

def test_task9_1():
    assert model.metrics == ['accuracy'], \
    'Did you specify [accuracy] as the metric?'
```

Out[17]:

```
2/2 tests passed
```

9. Load pre-trained model and score

Now we'll load a pre-trained model that has the architecture we specified above and was trained for 200 epochs on the full train and test sets we created above.

Let's use the `evaluate` method to see how well the model did at classifying bumble bees and honey bees for the test and validation sets. Recall that accuracy is the number of correct predictions divided by the total number of predictions. Given that our classes are balanced, a model that predicts `1.0` for every image would get an accuracy around `0.5`.

Note: it may take a few seconds to load the model. Recall that our model has over 3 million parameters (weights), which are what's being loaded.

In [18]:

```python
# load pre-trained model
pretrained_cnn = keras.models.load_model('datasets/pretrained_model.h5')

# evaluate model on test set
score = pretrained_cnn.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])

print("")

# evaluate model on holdout set
eval_score = pretrained_cnn.evaluate(x_eval, y_eval, verbose=0)
# print loss score
print('Eval loss:', eval_score[0])
# print accuracy score
print('Eval accuracy:', eval_score[1])
```

```
Test loss: 0.6423929142502119
Test accuracy: 0.664150944070996

Eval loss: 0.654895001667864
Eval accuracy: 0.649546827434413
```

In [19]:

```python
%%nose

def test_task9_0():
    assert [round(s, 4) for s in eval_score] == [0.6549, 0.6495], \
    'Did you calculate the eval loss and accuracy using eval_score =
pretrained_cnn.evaluate(x_eval, y_eval, verbose=0)?'
```

Out[19]:

```
1/1 tests passed
```

10. Visualize model training history

In addition to scoring the final iteration of the pre-trained model as we just did, we can also see the evolution of scores throughout training thanks to the `History` object. We'll use the `pickle` library to load the model history and then plot it.

Notice how the accuracy improves over time, eventually leveling off. Correspondingly, the loss decreases over time. Plots like these can help diagnose overfitting. If we had seen an upward curve in the validation loss as times goes on (a U shape in the plot), we'd suspect that the model was starting to memorize the test set and would not generalize well to new data.

In [20]:

```python
# load history
with open('datasets/model_history.pkl', 'rb') as f:
    pretrained_cnn_history = pickle.load(f)

# print keys for pretrained_cnn_history dict
print(pretrained_cnn_history.keys())

fig = plt.figure(1)
plt.subplot(211)
# plot the validation accuracy
```

```python
# plot the validation accuracy
plt.plot(pretrained_cnn_history['val_acc'])
plt.title('Validation accuracy and loss')
plt.ylabel('Accuracy')
plt.subplot(212)
# plot the validation loss
plt.plot(pretrained_cnn_history['val_loss'], 'r')
plt.xlabel('Epoch')
plt.ylabel('Loss value');
```

```
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```
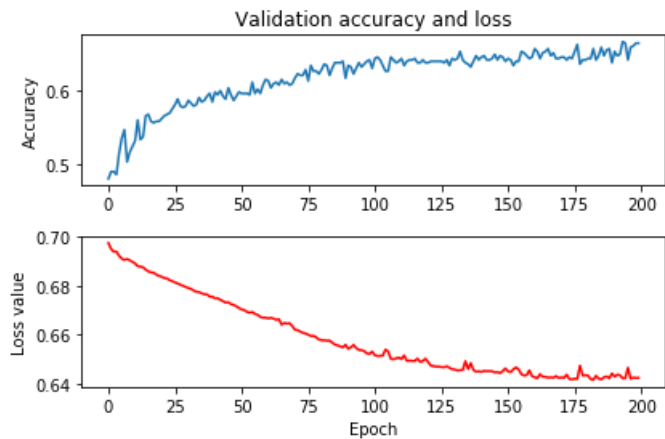

In [21]:

```python
%%nose
import matplotlib

def test_task10_0():
    assert 'fig' in globals() and isinstance(fig, matplotlib.figure.Figure), \
    "Did you plot pretrained_cnn_history['val_acc'] and pretrained_cnn_history['val_loss'] on their appropriate axes?"
```

Out[21]:

```
1/1 tests passed
```

11. Generate predictions

Previously, we calculated an overall score for our pre-trained model on the validation set. To end this notebook, let's access probabilities and class predictions for individual images using the `.predict` and `.predict_classes` methods.

We now have a deep learning model that can be used to identify honey bees and bumble bees in images! The next step is to explore transfer learning, which harnesses the prediction power of models that have been trained on far more images than the mere 1600 in our dataset.

In [22]:

```python
# predicted probabilities for x_eval
y_proba = pretrained_cnn.predict(x_eval)

print("First five probabilities:")
print(y_proba[:5])
print("")

# predicted classes for x_eval
y_pred = pretrained_cnn.predict_classes(x_eval)

print("First five class predictions:")
print(y_pred[:5])
print("")
```

```
First five probabilities:
[[0.6641951 ]
 [0.605251 85]
```

```
    [0.00025100]
    [0.75047696]
    [0.32835406]
    [0.88149595]]

First five class predictions:
[[1]
 [1]
 [1]
 [0]
 [1]]
```

In [23]:

```python
%%nose

def test_task11_0():
    assert round(float(y_proba[0][0]), 2) == 0.66, \
    'Did you predict probabilities using pretrained_cnn.predict(x_eval)?'

def test_task11_1():
    assert y_pred[3][0] == 0, \
    'Did you predict classes using pretrained_cnn.predict_classes(x_eval)?'
```

Out[23]:

2/2 tests passed