Keras tutorial - the Happy House

Welcome to the first assignment of week 2. In this assignment, you will:

1. Learn to use Keras, a high-level neural networks API (programming framework), written in Python and capable of running on top of several lower-level frameworks including TensorFlow and CNTK.
2. See how you can in a couple of hours build a deep learning algorithm.

Why are we using Keras? Keras was developed to enable deep learning engineers to build and experiment with different models very quickly. Just as TensorFlow is a higher-level framework than Python, Keras is an even higher-level framework and provides additional abstractions. Being able to go from idea to result with the least possible delay is key to finding good models. However, Keras is more restrictive than the lower-level frameworks, so there are some very complex models that you can implement in TensorFlow but not (without more difficulty) in Keras. That being said, Keras will work fine for many common models.

In this exercise, you'll work on the "Happy House" problem, which we'll explain below. Let's load the required packages and solve the problem of the Happy House!

In [19]:
```
import numpy as np
from keras import layers
from keras.layers import Input, Dense, Activation, ZeroPadding2D, BatchNormali
zation, Flatten, Conv2D
from keras.layers import AveragePooling2D, MaxPooling2D, Dropout, GlobalMaxPoo
ling2D, GlobalAveragePooling2D
from keras.models import Model
from keras.preprocessing import image
from keras.utils import layer_utils
from keras.utils.data_utils import get_file
from keras.applications.imagenet_utils import preprocess_input
import pydot
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model
from kt_utils import *

import keras.backend as K
K.set_image_data_format('channels_last')
import matplotlib.pyplot as plt
from matplotlib.pyplot import imshow

%matplotlib inline
```

Note: As you can see, we've imported a lot of functions from Keras. You can use them easily just by calling them directly in the notebook. Ex: `X = Input(...)` or `X = ZeroPadding2D(...)`.

1 - The Happy House

For your next vacation, you decided to spend a week with five of your friends from school. It is a very convenient house with many things to do nearby. But the most important benefit is that everybody has commited to be happy when they are in the house. So anyone wanting to enter the house must prove their current state of happiness.

Figure 1 : **the Happy House**

As a deep learning expert, to make sure the "Happy" rule is strictly applied, you are going to build an algorithm which that uses pictures from the front door camera to check if the person is happy or not. The door should open only if the person is happy.
You have gathered pictures of your friends and yourself, taken by the front-door camera. The dataset is labbeled.

Happy House Members

Labels

$$y = \begin{cases} 0 \longrightarrow \text{“not happy”} \\ 1 \longrightarrow \text{“happy”} \end{cases}$$

Run the following code to normalize the dataset and learn about its shapes.

In [20]:
```
X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset
()

# Normalize image vectors
X_train = X_train_orig/255.
X_test = X_test_orig/255.

# Reshape
Y_train = Y_train_orig.T
Y_test = Y_test_orig.T

print ("number of training examples = " + str(X_train.shape[0]))
print ("number of test examples = " + str(X_test.shape[0]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))
```

```
number of training examples = 600
number of test examples = 150
X_train shape: (600, 64, 64, 3)
Y_train shape: (600, 1)
X_test shape: (150, 64, 64, 3)
Y_test shape: (150, 1)
```

Details of the "Happy" dataset:

- Images are of shape (64,64,3)
- Training: 600 pictures
- Test: 150 pictures

It is now time to solve the "Happy" Challenge.

2 - Building a model in Keras

Keras is very good for rapid prototyping. In just a short time you will be able to build a model that achieves outstanding results.

Here is an example of a model in Keras:

```
    def model(input_shape):
        # Define the input placeholder as a tensor with shape input_shape. Think of thi
    s as your input image!
        X_input = Input(input_shape)

        # Zero-Padding: pads the border of X_input with zeroes
        X = ZeroPadding2D((3, 3))(X_input)

        # CONV -> BN -> RELU Block applied to X
        X = Conv2D(32, (7, 7), strides = (1, 1), name = 'conv0')(X)
        X = BatchNormalization(axis = 3, name = 'bn0')(X)
        X = Activation('relu')(X)

        # MAXPOOL
        X = MaxPooling2D((2, 2), name='max_pool')(X)

        # FLATTEN X (means convert it to a vector) + FULLYCONNECTED
        X = Flatten()(X)
        X = Dense(1, activation='sigmoid', name='fc')(X)

        # Create model. This creates your Keras model instance, you'll use this instanc
    e to train/test the model.
        model = Model(inputs = X_input, outputs = X, name='HappyModel')

        return model
```

Note that Keras uses a different convention with variable names than we've previously used with numpy and TensorFlow. In particular, rather than creating and assigning a new variable on each step of forward propagation such as `X`, `Z1`, `A1`, `Z2`, `A2`, etc. for the computations for the different layers, in Keras code each line above just reassigns `X` to a new value using `X = ...`. In other words, during each step of forward propagation, we are just writing the latest value in the commputation into the same variable `X`. The only exception was `X_input`, which we kept separate and did not overwrite, since we needed it at the end to create the Keras model instance (`model = Model(inputs = X_input, ...)` above).

Exercise: Implement a `HappyModel()`. This assignment is more open-ended than most. We suggest that you start by implementing a model using the architecture we suggest, and run through the rest of this assignment using that as your initial model. But after that, come back and take initiative to try out other model architectures. For example, you might take inspiration from the model above, but then vary the network architecture and hyperparameters however you wish. You can also use other functions such as `AveragePooling2D()`, `GlobalMaxPooling2D()`, `Dropout()`.

Note: You have to be careful with your data's shapes. Use what you've learned in the videos to make sure your convolutional, pooling and fully-connected layers are adapted to the volumes you're applying it to.

In [21]:
```
# GRADED FUNCTION: HappyModel

def HappyModel(input_shape):
    """
    Implementation of the HappyModel.

    Arguments:
    input_shape -- shape of the images of the dataset

    Returns:
    model -- a Model() instance in Keras
    """

    ### START CODE HERE ###
    # Feel free to use the suggested outline in the text above to get started, and run through the whole
    # exercise (including the later portions of this notebook) once. The come back also try out other
    # network architectures as well.
    X_input = Input(input_shape)

    X = ZeroPadding2D((3, 3))(X_input)

    # CONV -> BN -> RELU Block applied to X
    X = Conv2D(32, (7, 7), strides = (1, 1), name = 'conv0')(X)
    X = BatchNormalization(axis = 3, name = 'bn0')(X)
    X = Activation('relu')(X)

    X = MaxPooling2D((2, 2), name='max_pool')(X)

    X = Flatten()(X)
    X = Dense(1, activation='sigmoid', name='fc')(X)

    model = Model(inputs = X_input, outputs = X, name='HappyModel')

    ### END CODE HERE ###

    return model
```

You have now built a function to describe your model. To train and test this model, there are four steps in Keras:

1. Create the model by calling the function above
2. Compile the model by calling `model.compile(optimizer = "...", loss = "...", metrics = ["accuracy"])`
3. Train the model on train data by calling `model.fit(x = ..., y = ..., epochs = ..., batch_size = ...)`
4. Test the model on test data by calling `model.evaluate(x = ..., y = ...)`

If you want to know more about `model.compile()`, `model.fit()`, `model.evaluate()` and their arguments, refer to the official Keras documentation (https://keras.io/models/model/).

Exercise: Implement step 1, i.e. create the model.

In [22]:
```
### START CODE HERE ### (1 line)
happyModel = HappyModel(X_train.shape[1:])
### END CODE HERE ###
```

Exercise: Implement step 2, i.e. compile the model to configure the learning process. Choose the 3 arguments of `compile()` wisely. Hint: the Happy Challenge is a binary classification problem.

In [23]:
```
### START CODE HERE ### (1 line)
happyModel.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
### END CODE HERE ###
```

Exercise: Implement step 3, i.e. train the model. Choose the number of epochs and the batch size.

In [24]:
```
### START CODE HERE ### (1 line)
happyModel.fit(x = X_train, y = Y_train, epochs = 50, batch_size = 64)
### END CODE HERE ###
```

```
Epoch 1/50
600/600 [==============================] - 14s - loss: 4.7555 - acc: 0.4650

Epoch 2/50
600/600 [==============================] - 14s - loss: 1.8421 - acc: 0.5983

Epoch 3/50
600/600 [==============================] - 14s - loss: 1.1138 - acc: 0.6967

Epoch 4/50
600/600 [==============================] - 14s - loss: 0.7533 - acc: 0.7500

Epoch 5/50
600/600 [==============================] - 14s - loss: 0.3131 - acc: 0.8600

Epoch 6/50
600/600 [==============================] - 14s - loss: 0.2384 - acc: 0.9000

Epoch 7/50
600/600 [==============================] - 14s - loss: 0.1668 - acc: 0.9333

Epoch 8/50
600/600 [==============================] - 14s - loss: 0.1490 - acc: 0.9383

Epoch 9/50
600/600 [==============================] - 15s - loss: 0.1273 - acc: 0.9517

Epoch 10/50
600/600 [==============================] - 16s - loss: 0.0988 - acc: 0.9667

Epoch 11/50
600/600 [==============================] - 15s - loss: 0.0876 - acc: 0.9717

Epoch 12/50
600/600 [==============================] - 15s - loss: 0.0835 - acc: 0.9800

Epoch 13/50
600/600 [==============================] - 16s - loss: 0.0797 - acc: 0.9783

Epoch 14/50
600/600 [==============================] - 15s - loss: 0.0762 - acc: 0.9817

Epoch 15/50
600/600 [==============================] - 14s - loss: 0.0755 - acc: 0.9800

Epoch 16/50
600/600 [==============================] - 14s - loss: 0.0646 - acc: 0.9833

Epoch 17/50
600/600 [==============================] - 15s - loss: 0.0594 - acc: 0.9883

Epoch 18/50
600/600 [==============================] - 15s - loss: 0.0543 - acc: 0.9883

Epoch 19/50
600/600 [==============================] - 14s - loss: 0.0537 - acc: 0.9867
```

```
Epoch 20/50
600/600 [==============================] - 14s - loss: 0.0553 - acc: 0.9817

Epoch 21/50
600/600 [==============================] - 14s - loss: 0.0451 - acc: 0.9933

Epoch 22/50
600/600 [==============================] - 14s - loss: 0.0514 - acc: 0.9900

Epoch 23/50
600/600 [==============================] - 14s - loss: 0.0601 - acc: 0.9883

Epoch 24/50
600/600 [==============================] - 14s - loss: 0.0515 - acc: 0.9883

Epoch 25/50
600/600 [==============================] - 15s - loss: 0.0413 - acc: 0.9883

Epoch 26/50
600/600 [==============================] - 14s - loss: 0.0394 - acc: 0.9883

Epoch 27/50
600/600 [==============================] - 13s - loss: 0.0353 - acc: 0.9900

Epoch 28/50
600/600 [==============================] - 13s - loss: 0.0411 - acc: 0.9900

Epoch 29/50
600/600 [==============================] - 13s - loss: 0.0376 - acc: 0.9917

Epoch 30/50
600/600 [==============================] - 13s - loss: 0.0388 - acc: 0.9900

Epoch 31/50
600/600 [==============================] - 14s - loss: 0.0396 - acc: 0.9883

Epoch 32/50
600/600 [==============================] - 14s - loss: 0.0402 - acc: 0.9900

Epoch 33/50
600/600 [==============================] - 14s - loss: 0.0318 - acc: 0.9900

Epoch 34/50
600/600 [==============================] - 13s - loss: 0.0287 - acc: 0.9933

Epoch 35/50
600/600 [==============================] - 14s - loss: 0.0260 - acc: 0.9917

Epoch 36/50
600/600 [==============================] - 13s - loss: 0.0270 - acc: 0.9933

Epoch 37/50
600/600 [==============================] - 13s - loss: 0.0205 - acc: 0.9967

Epoch 38/50
600/600 [==============================] - 13s - loss: 0.0229 - acc: 0.9933
```

```
Epoch 39/50
600/600 [==============================] - 13s - loss: 0.0231 - acc: 0.9933

Epoch 40/50
600/600 [==============================] - 13s - loss: 0.0227 - acc: 0.9933

Epoch 41/50
600/600 [==============================] - 13s - loss: 0.0222 - acc: 0.9917

Epoch 42/50
600/600 [==============================] - 13s - loss: 0.0223 - acc: 0.9950

Epoch 43/50
600/600 [==============================] - 13s - loss: 0.0161 - acc: 0.9983

Epoch 44/50
600/600 [==============================] - 13s - loss: 0.0180 - acc: 0.9967

Epoch 45/50
600/600 [==============================] - 13s - loss: 0.0178 - acc: 0.9967

Epoch 46/50
600/600 [==============================] - 13s - loss: 0.0160 - acc: 0.9950

Epoch 47/50
600/600 [==============================] - 13s - loss: 0.0185 - acc: 0.9950

Epoch 48/50
600/600 [==============================] - 14s - loss: 0.0163 - acc: 0.9950

Epoch 49/50
600/600 [==============================] - 14s - loss: 0.0155 - acc: 0.9967

Epoch 50/50
600/600 [==============================] - 13s - loss: 0.0155 - acc: 0.9950
```

Out[24]: `<keras.callbacks.History at 0x7fee39a56f28>`

Note that if you run `fit()` again, the `model` will continue to train with the parameters it has already learnt instead of reinitializing them.

Exercise: Implement step 4, i.e. test/evaluate the model.

In [25]:
```
### START CODE HERE ### (1 line)
preds = happyModel.evaluate(x = X_test, y = Y_test)
### END CODE HERE ###
print()
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

```
150/150 [==============================] - 1s

Loss = 0.138111003439
Test Accuracy = 0.939999997616
```

If your `happyModel()` function worked, you should have observed much better than random-guessing (50%) accuracy on the train and test sets.

To give you a point of comparison, our model gets around **95% test accuracy in 40 epochs** (and 99% train accuracy) with a mini batch size of 16 and "adam" optimizer. But our model gets decent accuracy after just 2-5 epochs, so if you're comparing different models you can also train a variety of models on just a few epochs and see how they compare.

If you have not yet achieved a very good accuracy (let's say more than 80%), here're some things you can play around with to try to achieve it:

- Try using blocks of CONV->BATCHNORM->RELU such as:

  ```
  X = Conv2D(32, (3, 3), strides = (1, 1), name = 'conv0')(X)
  X = BatchNormalization(axis = 3, name = 'bn0')(X)
  X = Activation('relu')(X)
  ```

  until your height and width dimensions are quite low and your number of channels quite large (≈32 for example). You are encoding useful information in a volume with a lot of channels. You can then flatten the volume and use a fully-connected layer.
- You can use MAXPOOL after such blocks. It will help you lower the dimension in height and width.
- Change your optimizer. We find Adam works well.
- If the model is struggling to run and you get memory issues, lower your batch_size (12 is usually a good compromise)
- Run on more epochs, until you see the train accuracy plateauing.

Even if you have achieved a good accuracy, please feel free to keep playing with your model to try to get even better results.

Note: If you perform hyperparameter tuning on your model, the test set actually becomes a dev set, and your model might end up overfitting to the test (dev) set. But just for the purpose of this assignment, we won't worry about that here.

3 - Conclusion

Congratulations, you have solved the Happy House challenge!

Now, you just need to link this model to the front-door camera of your house. We unfortunately won't go into the details of how to do that here.

What we would like you to remember from this assignment:

- Keras is a tool we recommend for rapid prototyping. It allows you to quickly try out different model architectures. Are there any applications of deep learning to your daily life that you'd like to implement using Keras?
- Remember how to code a model in Keras and the four steps leading to the evaluation of your model on the test set. Create->Compile->Fit/Train->Evaluate/Test.

4 - Test with your own image (Optional)

Congratulations on finishing this assignment. You can now take a picture of your face and see if you could enter the Happy House. To do that:

1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.
2. Add your image to this Jupyter Notebook's directory, in the "images" folder
3. Write your image's name in the following code
4. Run the code and check if the algorithm is right (0 is unhappy, 1 is happy)!

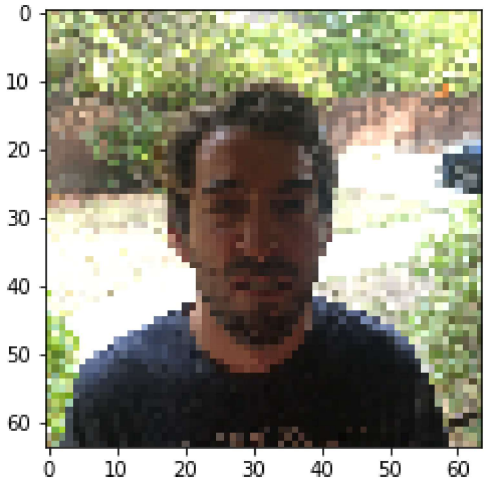
The training/test sets were quite similar; for example, all the pictures were taken against the same background (since a front door camera is always mounted in the same position). This makes the problem easier, but a model trained on this data may or may not work on your own data. But feel free to give it a try!

In [26]:
```
### START CODE HERE ###
img_path = 'images/my_image.jpg'
### END CODE HERE ###
img = image.load_img(img_path, target_size=(64, 64))
imshow(img)

x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

print(happyModel.predict(x))
```

```
[[ 0.]]
```



5 - Other useful functions in Keras (Optional)

Two other basic features of Keras that you'll find useful are:

- `model.summary()`: prints the details of your layers in a table with the sizes of its inputs/outputs
- `plot_model()`: plots your graph in a nice layout. You can even save it as ".png" using SVG() if you'd like to share it on social media ;). It is saved in "File" then "Open..." in the upper bar of the notebook.

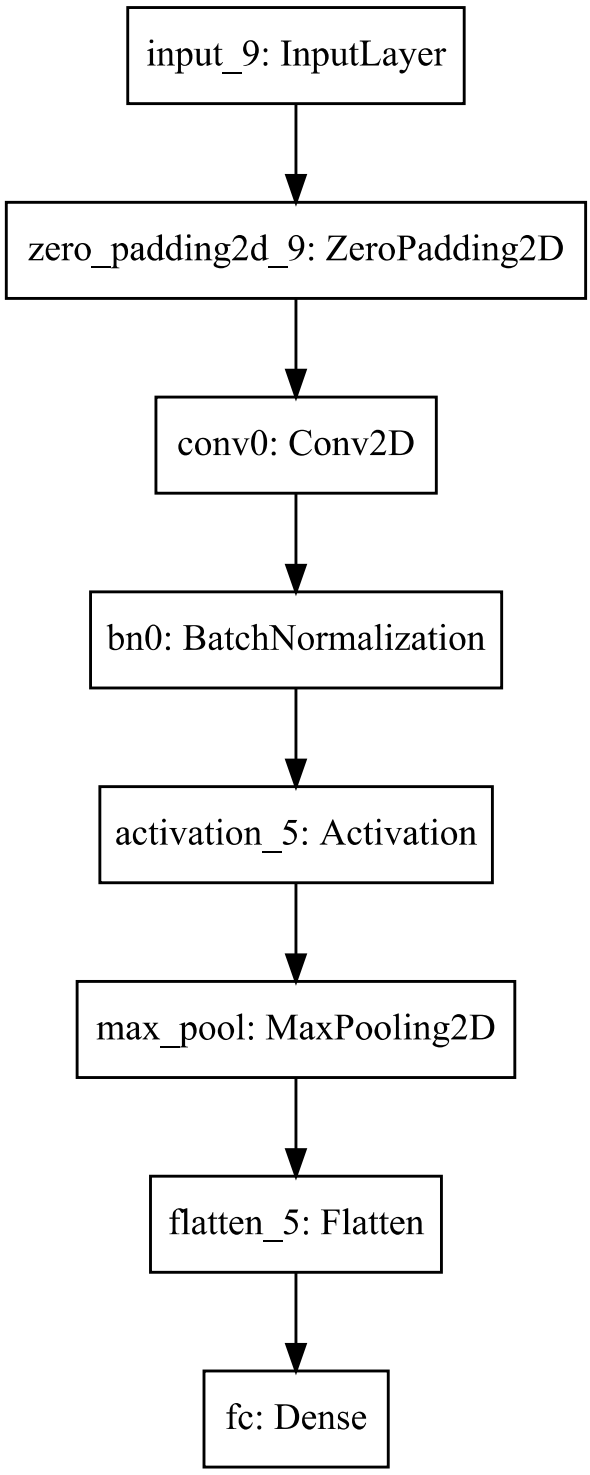
Run the following code.

In [27]:
```
happyModel.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         (None, 64, 64, 3)         0
_________________________________________________________________
zero_padding2d_9 (ZeroPaddin (None, 70, 70, 3)         0
_________________________________________________________________
conv0 (Conv2D)               (None, 64, 64, 32)        4736
_________________________________________________________________
bn0 (BatchNormalization)     (None, 64, 64, 32)        128
_________________________________________________________________
activation_5 (Activation)    (None, 64, 64, 32)        0
_________________________________________________________________
max_pool (MaxPooling2D)      (None, 32, 32, 32)        0
_________________________________________________________________
flatten_5 (Flatten)          (None, 32768)             0
_________________________________________________________________
fc (Dense)                   (None, 1)                 32769
=================================================================
Total params: 37,633
Trainable params: 37,569
Non-trainable params: 64
_________________________________________________________________
```

In [28]:
```
plot_model(happyModel, to_file='HappyModel.png')
SVG(model_to_dot(happyModel).create(prog='dot', format='svg'))
```

Out[28]:



In [ ]: