Keras_Tutorial_v2a

November 11, 2020

1 Keras tutorial - Emotion Detection in Images of Faces

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 would still implement in TensorFlow rather than in Keras.
- That being said, Keras will work fine for many common models.

1.1 Updates

If you were working on the notebook before this update...

- The current notebook is version "v2a".
- You can find your original work saved in the notebook with the previous version name ("v2").
- To view the file directory, go to the menu "File->Open", and this will open a new tab that shows the file directory.

List of updates

- Changed back-story of model to "emotion detection" from "happy house."
- Cleaned/organized wording of instructions and commentary.
- Added instructions on how to set `input_shape`
- Added explanation of "objects as functions" syntax.
- Clarified explanation of variable naming convention.
- Added hints for steps 1,2,3,4

1.2 Load packages

- In this exercise, you'll work on the "Emotion detection" model, which we'll explain below.
- Let's load the required packages.

```
In [12]: import numpy as np
         from keras import layers
         from keras.layers import Input, Dense, Activation, ZeroPadding2D, BatchNor
         from keras.layers import AveragePooling2D, MaxPooling2D, Dropout, GlobalMa
         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

         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 by calling them directly in your code. Ex: `X = Input(...)` or `X = ZeroPadding2D(...)`.

In other words, unlike TensorFlow, you don't have to create the graph and then make a separate `sess.run()` call to evaluate those variables.

1.3 1 - Emotion Tracking

- A nearby community health clinic is helping the local residents monitor their mental health.

- As part of their study, they are asking volunteers to record their emotions throughout the day.
- To help the participants more easily track their emotions, you are asked to create an app that will classify their emotions based on some pictures that the volunteers will take of their facial expressions.
- As a proof-of-concept, you first train your model to detect if someone's emotion is classified as "happy" or "not happy."

To build and train this model, you have gathered pictures of some volunteers in a nearby neighborhood. The dataset is labeled.

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

```
In [5]: X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_datase

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

1.4 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):
    """
    input_shape: The height, width and channels as a tuple.
        Note that this does not include the 'batch' as a dimension.
        If you have a batch like 'X_train',
        then you can provide the input_shape using
        X_train.shape[1:]
    """

    # Define the input placeholder as a tensor with shape input_shape. Think of thi
    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
    model = Model(inputs = X_input, outputs = X, name='HappyModel')

    return model
```

Variable naming convention

- Note that Keras uses a different convention with variable names than we've previously used with numpy and TensorFlow.
- Instead of creating unique variable names for each step and each layer, such as

```
X = ...
Z1 = ...
A1 = ...
```

- Keras re-uses and overwrites the same variable at each step:

```
X = ...
X = ...
X = ...
```

- The exception is `X_input`, which we kept separate since it's needed later.

Objects as functions

- Notice how there are two pairs of parentheses in each statement. For example:

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

- The first is a constructor call which creates an object (ZeroPadding2D).
- In Python, objects can be called as functions. Search for 'python object as function and you can read this blog post Python Pandemonium. See the section titled "Objects as functions."
- The single line is equivalent to this:

```
ZP = ZeroPadding2D((3, 3)) # ZP is an object that can be called as a function
X = ZP(X_input)
```

Exercise: Implement a `HappyModel()`.
This assignment is more open-ended than most. * Start by implementing a model using the architecture we suggest, and run through the rest of this assignment using that as your initial model. * Later, come back and 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: 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 [10]:
```
# GRADED FUNCTION: HappyModel

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

    Arguments:
    input_shape -- shape of the images of the dataset
        (height, width, channels) as a tuple.
        Note that this does not include the 'batch' as a dimension.
        If you have a batch like 'X_train',
        then you can provide the input_shape using
        X_train.shape[1:]
    """
    """
    Returns:
    model -- a Model() instance in Keras
    """

    ### START CODE HERE ###

    # Define the input placeholder as a tensor with shape input_shape. Thi
    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 thi
    model = Model(inputs = X_input, outputs = X, name='HappyModel')
    # Feel free to use the suggested outline in the text above to get star
    # exercise (including the later portions of this notebook) once. The c
    # network architectures as well.


    ### 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.

Step 1: create the model. **Hint**:
The `input_shape` parameter is a tuple (height, width, channels). It excludes the batch number. Try `X_train.shape[1:]` as the `input_shape`.

```
In [11]: ### START CODE HERE ### (1 line)
         happyModel = HappyModel(X_train.shape[1:])
         ### END CODE HERE ###
```

Step 2: compile the model **Hint**:
Optimizers you can try include `'adam'`, `'sgd'` or others. See the documentation for optimizers The "happiness detection" is a binary classification problem. The loss function that you can use is `'binary_cross_entropy'`. Note that `'categorical_cross_entropy'` won't work with your data set as its formatted, because the data is an array of 0 or 1 rather than two arrays (one for each category). Documentation for losses

```
In [19]: ### START CODE HERE ### (1 line)
         happyModel.compile(optimizer='adam',loss='binary_crossentropy',metrics=['a
         ### END CODE HERE ###
```

Step 3: train the model **Hint**:
Use the `'X_train','Y_train'` variables. Use integers for the epochs and batch_size

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

```
In [20]: ### START CODE HERE ### (1 line)
         happyModel.fit(X_train,Y_train,epochs=50,batch_size=25,verbose=2,validatio
         ### END CODE HERE ###

Train on 600 samples, validate on 150 samples
Epoch 1/50
15s - loss: 1.9779 - acc: 0.6033 - val_loss: 0.6023 - val_acc: 0.6400
Epoch 2/50
15s - loss: 0.2906 - acc: 0.8933 - val_loss: 0.9428 - val_acc: 0.5600
Epoch 3/50
14s - loss: 0.1838 - acc: 0.9317 - val_loss: 0.8980 - val_acc: 0.5667
Epoch 4/50
15s - loss: 0.1147 - acc: 0.9617 - val_loss: 0.7084 - val_acc: 0.6000
Epoch 5/50
15s - loss: 0.1264 - acc: 0.9483 - val_loss: 0.7367 - val_acc: 0.6000
Epoch 6/50
14s - loss: 0.0811 - acc: 0.9700 - val_loss: 0.6431 - val_acc: 0.6000
Epoch 7/50
15s - loss: 0.1020 - acc: 0.9550 - val_loss: 0.4183 - val_acc: 0.7467
Epoch 8/50
15s - loss: 0.0889 - acc: 0.9700 - val_loss: 0.3798 - val_acc: 0.7600
Epoch 9/50
15s - loss: 0.0760 - acc: 0.9833 - val_loss: 0.2911 - val_acc: 0.8867
Epoch 10/50
15s - loss: 0.0831 - acc: 0.9750 - val_loss: 0.1762 - val_acc: 0.9667
Epoch 11/50
15s - loss: 0.0655 - acc: 0.9767 - val_loss: 0.7111 - val_acc: 0.6400
Epoch 12/50
15s - loss: 0.0542 - acc: 0.9850 - val_loss: 0.3276 - val_acc: 0.8000
Epoch 13/50
14s - loss: 0.0400 - acc: 0.9883 - val_loss: 0.2303 - val_acc: 0.9067
Epoch 14/50
15s - loss: 0.0404 - acc: 0.9883 - val_loss: 0.2562 - val_acc: 0.8667
Epoch 15/50
15s - loss: 0.0500 - acc: 0.9800 - val_loss: 0.1969 - val_acc: 0.9133
Epoch 16/50
15s - loss: 0.0403 - acc: 0.9850 - val_loss: 0.1375 - val_acc: 0.9533
Epoch 17/50
15s - loss: 0.0479 - acc: 0.9867 - val_loss: 0.1341 - val_acc: 0.9667
Epoch 18/50
15s - loss: 0.0748 - acc: 0.9767 - val_loss: 2.4672 - val_acc: 0.5600
Epoch 19/50
15s - loss: 0.0505 - acc: 0.9850 - val_loss: 0.1284 - val_acc: 0.9533
```

```
Epoch 20/50
15s - loss: 0.0474 - acc: 0.9883 - val_loss: 0.9889 - val_acc: 0.6933
Epoch 21/50
15s - loss: 0.0265 - acc: 0.9950 - val_loss: 0.2948 - val_acc: 0.8467
Epoch 22/50
14s - loss: 0.0325 - acc: 0.9867 - val_loss: 0.0972 - val_acc: 0.9733
Epoch 23/50
15s - loss: 0.0320 - acc: 0.9933 - val_loss: 0.1066 - val_acc: 0.9533
Epoch 24/50
15s - loss: 0.1046 - acc: 0.9600 - val_loss: 2.3451 - val_acc: 0.5733
Epoch 25/50
14s - loss: 0.0642 - acc: 0.9717 - val_loss: 1.1919 - val_acc: 0.6933
Epoch 26/50
15s - loss: 0.0371 - acc: 0.9817 - val_loss: 0.7585 - val_acc: 0.7000
Epoch 27/50
15s - loss: 0.0590 - acc: 0.9817 - val_loss: 1.3999 - val_acc: 0.6067
Epoch 28/50
15s - loss: 0.0237 - acc: 0.9933 - val_loss: 0.4457 - val_acc: 0.8200
Epoch 29/50
15s - loss: 0.0326 - acc: 0.9900 - val_loss: 1.1723 - val_acc: 0.6667
Epoch 30/50
15s - loss: 0.0212 - acc: 0.9900 - val_loss: 0.0954 - val_acc: 0.9400
Epoch 31/50
15s - loss: 0.0273 - acc: 0.9900 - val_loss: 0.0996 - val_acc: 0.9667
Epoch 32/50
15s - loss: 0.0727 - acc: 0.9750 - val_loss: 0.6012 - val_acc: 0.8333
Epoch 33/50
15s - loss: 0.0288 - acc: 0.9983 - val_loss: 0.0965 - val_acc: 0.9667
Epoch 34/50
14s - loss: 0.0218 - acc: 0.9933 - val_loss: 0.1375 - val_acc: 0.9333
Epoch 35/50
15s - loss: 0.0132 - acc: 0.9950 - val_loss: 0.0595 - val_acc: 0.9733
Epoch 36/50
15s - loss: 0.0165 - acc: 0.9950 - val_loss: 0.0662 - val_acc: 0.9800
Epoch 37/50
14s - loss: 0.0112 - acc: 0.9967 - val_loss: 0.0829 - val_acc: 0.9667
Epoch 38/50
15s - loss: 0.0172 - acc: 0.9967 - val_loss: 0.1412 - val_acc: 0.9600
Epoch 39/50
15s - loss: 0.0163 - acc: 0.9933 - val_loss: 0.3233 - val_acc: 0.8867
Epoch 40/50
15s - loss: 0.0320 - acc: 0.9883 - val_loss: 0.3415 - val_acc: 0.8867
Epoch 41/50
15s - loss: 0.0122 - acc: 0.9950 - val_loss: 0.6623 - val_acc: 0.7467
Epoch 42/50
15s - loss: 0.0274 - acc: 0.9883 - val_loss: 0.0704 - val_acc: 0.9667
Epoch 43/50
15s - loss: 0.0203 - acc: 0.9933 - val_loss: 0.2687 - val_acc: 0.9133
```

```
Epoch 44/50
15s - loss: 0.0127 - acc: 0.9983 - val_loss: 0.1422 - val_acc: 0.9467
Epoch 45/50
14s - loss: 0.0147 - acc: 0.9950 - val_loss: 0.0811 - val_acc: 0.9667
Epoch 46/50
15s - loss: 0.0263 - acc: 0.9883 - val_loss: 0.2109 - val_acc: 0.9200
Epoch 47/50
14s - loss: 0.0319 - acc: 0.9883 - val_loss: 0.1305 - val_acc: 0.9600
Epoch 48/50
15s - loss: 0.0160 - acc: 0.9933 - val_loss: 0.0785 - val_acc: 0.9733
Epoch 49/50
15s - loss: 0.0103 - acc: 0.9967 - val_loss: 0.0678 - val_acc: 0.9667
Epoch 50/50
15s - loss: 0.0098 - acc: 0.9967 - val_loss: 0.0686 - val_acc: 0.9667


Out[20]: <keras.callbacks.History at 0x7f080841c198>
```

Step 4: evaluate model **Hint**:
Use the `'X_test'` and `'Y_test'` variables to evaluate the model's performance.

```
In [22]: ### START CODE HERE ### (1 line)
         preds = happyModel.evaluate(X_test, Y_test, batch_size=32, verbose=1, samp
         ### END CODE HERE ###
         print()
         print ("Loss = " + str(preds[0]))
         print ("Test Accuracy = " + str(preds[1]))

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

Loss = 0.0685787600279
Test Accuracy = 0.96666667064
```

Expected performance If your `happyModel()` function worked, its accuracy should be better than random guessing (50% accuracy).

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.

Tips for improving your model If you have not yet achieved a very good accuracy (>= 80%), here are some things tips:

- Use 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 can then flatten the volume and use a fully-connected layer.

- 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 you get memory issues, lower your batch_size (e.g. 12)
- Run more epochs until you see the train accuracy no longer improves.

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. Normally, you'll want separate dev and test sets. The dev set is used for parameter tuning, and the test set is used once to estimate the model's performance in production.

1.5 3 - Conclusion

Congratulations, you have created a proof of concept for "happiness detection"!

1.6 Key Points to remember

- Keras is a tool we recommend for rapid prototyping. It allows you to quickly try out different model architectures.
- Remember The four steps in Keras:

1. Create
2. Compile
3. Fit/Train
4. Evaluate/Test

1.7 4 - Test with your own image (Optional)

Congratulations on finishing this assignment. You can now take a picture of your face and see if it can classify whether your expression is "happy" or "not happy". 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 not happy, 1 is happy)!

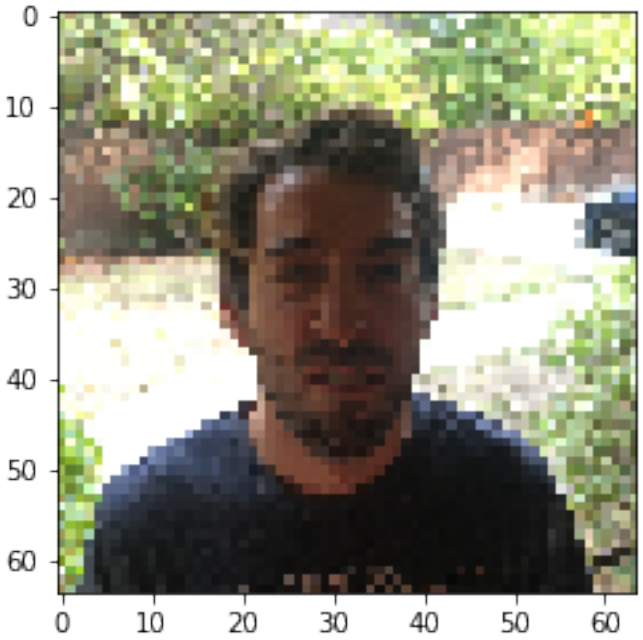
The training/test sets were quite similar; for example, all the pictures were taken against the same background (since a front door camera is always mounted in the same position). This makes the problem easier, but a model trained on this data may or may not work on your own data. But feel free to give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

```
In [23]: ### 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.]]
```



1.8 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.

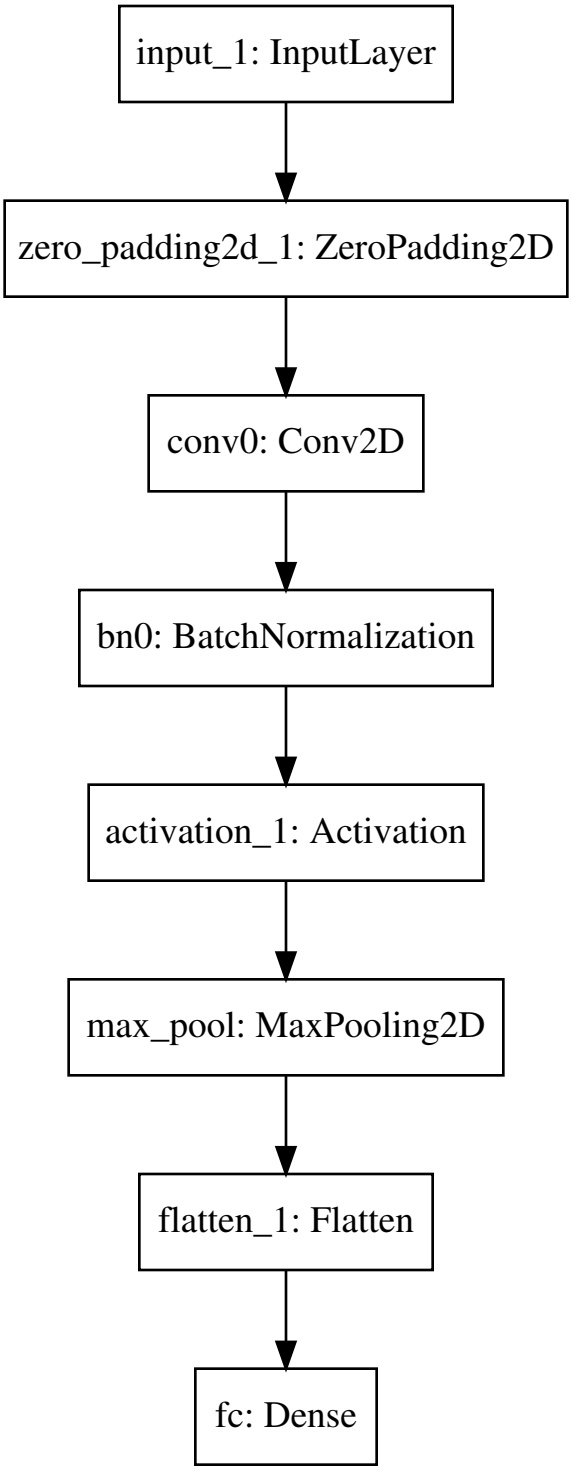
```
In [24]: happyModel.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

```
=================================================================
input_1 (InputLayer)         (None, 64, 64, 3)         0
_________________________________________________________________
zero_padding2d_1 (ZeroPaddin (None, 70, 70, 3)         0
_________________________________________________________________
conv0 (Conv2D)               (None, 64, 64, 32)        4736
_________________________________________________________________
bn0 (BatchNormalization)     (None, 64, 64, 32)        128
_________________________________________________________________
activation_1 (Activation)    (None, 64, 64, 32)        0
_________________________________________________________________
max_pool (MaxPooling2D)      (None, 32, 32, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 32768)             0
_________________________________________________________________
fc (Dense)                   (None, 1)                 32769
=================================================================
Total params: 37,633
Trainable params: 37,569
Non-trainable params: 64
_________________________________________________________________
```

```
In [25]: plot_model(happyModel, to_file='HappyModel.png')
         SVG(model_to_dot(happyModel).create(prog='dot', format='svg'))
```

Out[25]:


input_1: InputLayer
zero_padding2d_1: ZeroPadding2D
conv0: Conv2D
bn0: BatchNormalization
activation_1: Activation
max_pool: MaxPooling2D
flatten_1: Flatten
fc: Dense

In [ ]: