Step 0: Import Datasets

Import Dog Dataset

In the code cell below, we import a dataset of dog images. We populate a few variables through the use of the `load_files`function from the scikit-learn library:

- `train_files`, `valid_files`, `test_files` - numpy arrays containing file paths to images
- `train_targets`, `valid_targets`, `test_targets` - numpy arrays containing onehot-encoded classification labels
- `dog_names` - list of string-valued dog breed names for translating labels

In [1]:
```
from sklearn.datasets import load_files
from keras.utils import np_utils
import numpy as np
from glob import glob
```

```
Using TensorFlow backend.
```

In [2]:
```
# define function to load train, test, and validation datasets
def load_dataset(path):
    data = load_files(path)
    dog_files = np.array(data['filenames'])
    dog_targets = np_utils.to_categorical(np.array(data['target']), 133)
    return dog_files, dog_targets
```

In [3]:
```
# load train, test, and validation datasets
train_files, train_targets = load_dataset('dogImages/train')
valid_files, valid_targets = load_dataset('dogImages/valid')
test_files, test_targets = load_dataset('dogImages/test')
```

In [4]:
```
print('There are %d training dog images.' % len(train_files))
print('There are %d validation dog images.' % len(valid_files))
print('There are %d test dog images.'% len(test_files))
```

```
There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```

In [5]:
```
# load list of dog names
dog_names = [item[20:-1] for item in sorted(glob("dogImages/train/*/"))]
# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files,
valid_files, test_files])))
```

```
There are 133 total dog categories.
There are 8351 total dog images.
```

Import Human Dataset

In the code cell below, we import a dataset of human images, where the file paths are stored in the numpy array `human_files`.

In [6]:

```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("lfw/*/*"))
random.shuffle(human_files)

# print statistics about the dataset
print('There are %d total human images.' % len(human_files))
```

```
There are 13233 total human images.
```

Step 1: Detect Humans

We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [7]:

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline
```

In [8]:

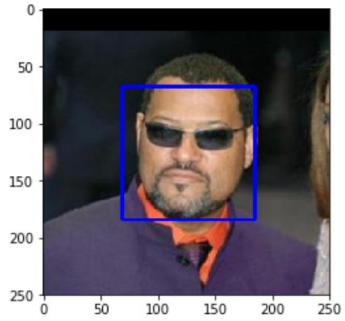
```
# extract pre-trained face detector
face_cascade =
cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')
```

In [9]:

```
# load color (BGR) image
img = cv2.imread(human_files[3])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale`function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

In [10]:

```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

(IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.

- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

In [11]:

```
%matplotlib inline

human_files_short = human_files[:100]
dog_files_short = train_files[:100]
# Do NOT modify the code above this line.
## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.

def faces_counter(given_list, object_type):
    no_detect = []
    detect = []
```

```
    for i in range(len(given_list)):
        if face_detector(given_list[i]):
            detect.append(given_list[i])
        else:
            no_detect.append(given_list[i])

    fig = plt.figure(figsize=(20,5))

    if object_type == 'human':
        print(len(detect), '%% of %s images with a detected face' %
object_type)
        print('No faces have been detected in the following %d photo(s):' %
len(no_detect))
        for i in range(len(no_detect)):
            img = cv2.imread(no_detect[i])
            ax = fig.add_subplot(1, 12, i + 1, xticks=[], yticks=[])
            ax.imshow(img)
    else:
        print('\n',len(detect),'%% of %s images with a detected face' %
object_type)
        print('The faces have been detected in the following %d photo(s):' %
(100-len(no_detect)))
        for i in range(len(detect)):
            img = cv2.imread(detect[i])
            ax = fig.add_subplot(1, len(detect), i + 1, xticks=[], yticks=[])
            ax.imshow(img)

faces_counter(human_files_short,'human')
faces_counter(dog_files_short,'dog')
```

```
99 % of human images with a detected face
No faces have been detected in the following 1 photo(s):


 11 % of dog images with a detected face
The faces have been detected in the following 11 photo(s):
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on each of the datasets.

In [12]:

```
## (Optional) TODO: Report the performance of another
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained ResNet-50 model to detect dogs in images. Our first line of code downloads the ResNet-50 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories. Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

In [13]:

```
from keras.applications.resnet50 import ResNet50

# define ResNet50 model
ResNet50_model = ResNet50(weights='imagenet')
```

Pre-process the Data

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

(nb_samples,rows,columns,channels),(nb_samples,rows,columns,channels),

where `nb_samples` corresponds to the total number of images (or samples), and `rows`, `columns`, and `channels` correspond to the number of rows, columns, and channels for each image, respectively.

The `path_to_tensor` function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

(1,224,224,3).(1,224,224,3).

The `paths_to_tensor` function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

(nb_samples,224,224,3).(nb_samples,224,224,3).

Here, `nb_samples` is the number of samples, or number of images, in the supplied array of image paths. It is best to think of `nb_samples` as the number of 3D tensors (where each 3D tensor corresponds to a different image) in your dataset!

In [14]:

```
from keras.preprocessing import image
from tqdm import tqdm
```

In [15]:

```
def path_to_tensor(img_path):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D
tensor
    return np.expand_dims(x, axis=0)
```

In [16]:

```
def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in
tqdm(img_paths)]
    return np.vstack(list_of_tensors)
```

Making Predictions with ResNet-50

Getting the 4D tensor ready for ResNet-50, and for any other pre-trained model in Keras, requires some additional processing. First, the RGB image is converted to BGR by reordering the channels. All pre-trained models have the additional normalization step that the mean pixel (expressed in RGB as [103.939,116.779,123.68][103.939,116.779,123.68] and calculated from all pixels in all images in ImageNet) must be subtracted from every pixel in each image. This is implemented in the imported function `preprocess_input`. If you're curious, you can check the code for `preprocess_input` here.
Now that we have a way to format our image for supplying to ResNet-50, we are now ready to use the model to extract the predictions. This is accomplished with the `predict` method, which returns an array whose ii-th entry is the model's predicted probability that the image belongs to the ii-th ImageNet category. This is implemented in the `ResNet50_predict_labels`function below.

By taking the argmax of the predicted probability vector, we obtain an integer corresponding to the model's predicted object class, which we can identify with an object category through the use of this dictionary.

In [17]:

```
from keras.applications.resnet50 import preprocess_input, decode_predictions

def ResNet50_predict_labels(img_path):
    # returns prediction vector for image located at img_path
    img = preprocess_input(path_to_tensor(img_path))
    return np.argmax(ResNet50_model.predict(img))
```

Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained ResNet-50 model, we need only check if the `ResNet50_predict_labels` function above returns a value between 151 and 268 (inclusive).

We use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

In [18]:

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    prediction = ResNet50_predict_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

(IMPLEMENTATION) Assess the Dog Detector

Question 3: Use the code cell below to test the performance of your `dog_detector` function.

- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

In [19]:

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
def dogs_counter(given_list, object_type):

    no_detect = []
    detect = []

    for i in range(len(given_list)):
        if dog_detector(given_list[i]):
            detect.append(given_list[i])
        else:
            no_detect.append(given_list[i])

    fig = plt.figure(figsize=(20,5))

    if object_type == 'human':
        print(len(detect), '%% of %s images with a detected dog' % object_type)
        print('Dogs have been detected in the following %d photo(s):' %
len(detect))
        for i in range(len(detect)):
            img = cv2.imread(detect[i])
            ax = fig.add_subplot(1, len(detect), i + 1, xticks=[], yticks=[])
            ax.imshow(img)
    else:
        print('\n',len(detect),'%% of %s images with a detected dog' %
object_type)
        if len(no_detect) !=0:
            print('No dogs have been detected in the following %d photo(s):' %
len(no_detect))
            for i in range(len(no_detect)):
                img = cv2.imread(no_detect[i])
                ax = fig.add_subplot(1, len(no_detect), i + 1, xticks=[],
yticks=[])
                ax.imshow(img)

dogs_counter(human_files_short,'human')
dogs_counter(dog_files_short,'dog')
```

```
1 % of human images with a detected dog
Dogs have been detected in the following 1 photo(s):
 100 % of dog images with a detected dog
<matplotlib.figure.Figure at 0x166b3855be0>
```

In [20]:

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

# pre-process the data for Keras
train_tensors = paths_to_tensor(train_files).astype('float32')/255
valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

```
100%|██████████████████████████████████████████████████████████████████████████████
███| 6680/6680 [01:45<00:00, 63.02it/s]
100%|██████████████████████████████████████████████████████████████████████████████
█████| 835/835 [01:01<00:00, 13.55it/s]
100%|██████████████████████████████████████████████████████████████████████████████
█████| 836/836 [00:19<00:00, 43.16it/s]
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 1%. In Step 5 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

Be careful with adding too many trainable layers! More parameters means longer training, which means you are more likely to need a GPU to accelerate the training process. Thankfully, Keras provides a handy estimate of the time that each epoch is likely to take; you can extrapolate this estimate to figure out how long it will take for your algorithm to train.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.
It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

Pre-process the Data

We rescale the images by dividing every pixel in every image by 255.

(IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
model.summary()
```

We have imported some Python modules to get you started, but feel free to import as many modules as you need. If you end up getting stuck, here's a hint that specifies a model that trains relatively fast on CPU and attains >1% test accuracy in 5 epochs:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 223, 223, 16)	208
max_pooling2d_1 (MaxPooling2	(None, 111, 111, 16)	0
conv2d_2 (Conv2D)	(None, 110, 110, 32)	2080
max_pooling2d_2 (MaxPooling2	(None, 55, 55, 32)	0
conv2d_3 (Conv2D)	(None, 54, 54, 64)	8256
max_pooling2d_3 (MaxPooling2	(None, 27, 27, 64)	0
global_average_pooling2d_1 (	(None, 64)	0
dense_1 (Dense)	(None, 133)	8645

Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. If you chose to use the hinted architecture above, describe why you think that CNN architecture should work well for the image classification task.

Answer: First of all I've read carefully all lessons materials :-) that is why just took the proposed model from the lesson **CNNs for Image Classification with Keras**. With the aim to reduce the calculations I just minimize the fully connected layer's nodes from 500 through 64 and add two **dropout** layers.

The task was to get accuracy more than 1%. And this model even obviously undertrained gave acceptable result with only 10 epochs of training. I'm sure, that there is learning potential and with 30 epochs accuracy should be much better.

In [75]:

```
# Helper for the plotting
# graph properties
def graph_prop(title,ylabel,history,a,v):
    plt.title(title)
    plt.xlabel('Epoch')
    plt.ylabel(ylabel)
    plt.grid(True)
#    plt.xlim(xlimit_min,xlimit_max)
#    plt.ylim(ylimit_min,y_limit_max)
    plt.plot(history.history[a])
    plt.plot(history.history[v])
    plt.legend(['Train', 'Test'], loc='upper left')

# graph the history of model.fit
def show_history_graph(history):

    fig1 = plt.figure()
    ax1 = fig1.add_subplot(1,2,1)
    graph_prop('Model Accuracy','Accuracy',history,'acc','val_acc')
    ax2 = fig1.add_subplot(1,2,2)
    graph_prop('Model Loss','Loss',history,'loss','val_loss')
    plt.subplots_adjust(top=0.92, bottom=0.08, left=0.20, right=1.95,
hspace=2.95, wspace=0.75)
    plt.show()
```

In [69]:

```
# Helpers for the time checking while training CNNs
import keras
import timeit

class EpochTimer(keras.callbacks.Callback):
    train_start = 0
    train_end = 0
    epoch_start = 0
    epoch_end = 0

    def get_time(self):
        return timeit.default_timer()

    def on_train_begin(self, logs={}):
        self.train_start = self.get_time()

    def on_train_end(self, logs={}):
        self.train_end = self.get_time()
        print('Training took {} seconds'.format(self.train_end -
self.train_start))

    def on_epoch_begin(self, epoch, logs={}):
        self.epoch_start = self.get_time()

    def on_epoch_end(self, epoch, logs={}):
        self.epoch_end = self.get_time()
        print('Epoch {} took {} seconds'.format(epoch, self.epoch_end -
self.epoch_start))
```

In [70]:

```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(filters=16, kernel_size=5, strides=2, activation='relu',
input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(133, activation='softmax'))
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 110, 110, 16)      1216
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 55, 55, 16)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 55, 55, 32)        4640
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 27, 27, 32)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 27, 27, 64)        8256
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 13, 13, 64)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 13, 13, 64)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 10816)             0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                692288
_________________________________________________________________
dropout_4 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 133)               8645
=================================================================
Total params: 715,045.0
Trainable params: 715,045.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [72]:

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data, but this is not a requirement.

In [131]:

```
# Helper for the model fitting
def model_fit(model,augmentation,train_data,valid_data):
    if not augmentation:
        print('Training without data augmentation.')
        hist = model.fit(train_data, train_targets,
                         validation_data=(valid_data, valid_targets),
                         epochs=epochs, batch_size=batch_size, verbose=0,
                         callbacks=[checkpointer, epochtimer])
    else:
        print('Training with data augmentation.')
```

```
        datagen = ImageDataGenerator(
            rotation_range=40,
            width_shift_range=0.2,
            height_shift_range=0.2,
            shear_range=0.2,
            zoom_range=0.2,
            horizontal_flip=True,
            fill_mode='nearest')

        datagen.fit(train_tensors)

        # Fit the model on the batches generated by datagen.flow().
        hist = model.fit_generator(datagen.flow(train_data, train_targets,
batch_size=batch_size),
                                   steps_per_epoch=train_tensors.shape[0]
// batch_size,
                                   epochs=epochs,
                                   callbacks=[checkpointer, epochtimer],
                                   validation_data=(valid_data,
valid_targets), verbose=0)
    return hist
```

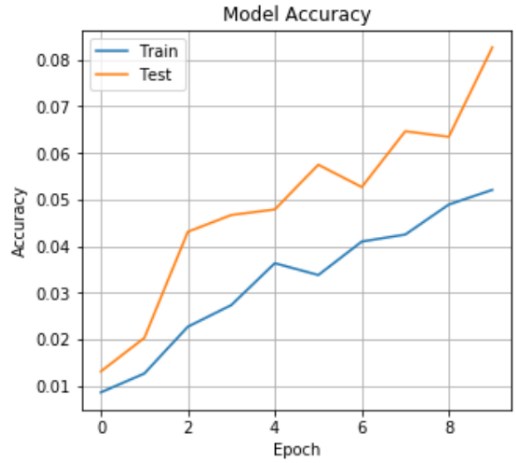
In [103]:

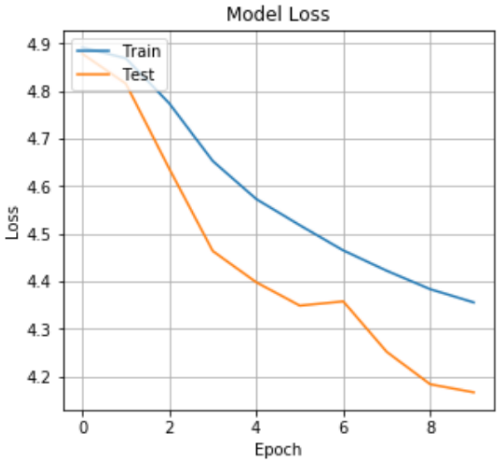
```
from keras.callbacks import ModelCheckpoint
from keras.preprocessing.image import ImageDataGenerator

### TODO: specify the number of epochs that you would like to use to train the
model.

epochs = 10
batch_size = 20

checkpointer =
ModelCheckpoint(filepath='saved_models/weights.best.from_scratch.hdf5',
                               verbose=1, save_best_only=True)
epochtimer = EpochTimer()

hist = model_fit(model,True,train_tensors,valid_tensors)
```

```
Training with data augmentation.
Epoch 00000: val_loss improved from inf to 4.09226, saving model to
saved_models/weights.best.from_scratch.hdf5
Epoch 0 took 183.6523209656425 seconds
Epoch 00001: val_loss did not improve
Epoch 1 took 143.16612761240867 seconds
Epoch 00002: val_loss improved from 4.09226 to 4.08469, saving model to
saved_models/weights.best.from_scratch.hdf5
…..
Epoch 9 took 149.9676940653644 seconds
Training took 1515.3985815245505 seconds
```

In [76]:

```
show_history_graph(hist)
```





Load the Model with the Best Validation Loss

In [77]:

```
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 1%.

In [78]:

```
# get index of predicted dog breed for each image in test set
dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(tensor,
axis=0))) for tensor in test_tensors]

# report test accuracy
test_accuracy =
100*np.sum(np.array(dog_breed_predictions)==np.argmax(test_targets,
axis=1))/len(dog_breed_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 7.4163%
```

Step 4: Use a CNN to Classify Dog Breeds

To reduce training time without sacrificing accuracy, we show you how to train a CNN using transfer learning. In the following step, you will get a chance to use transfer learning to train your own CNN.

Obtain Bottleneck Features

In [79]:

```
bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
train_VGG16 = bottleneck_features['train']
valid_VGG16 = bottleneck_features['valid']
test_VGG16 = bottleneck_features['test']
```

Model Architecture

The model uses the the pre-trained VGG-16 model as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. We only add a global average pooling layer and a fully connected layer, where the latter contains one node for each dog category and is equipped with a softmax.

In [80]:

```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_1 ( (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 133)               68229
=================================================================
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [81]:

```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
```

Train the Model

In [85]:

```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.hdf5',
                               verbose=1, save_best_only=True)

VGG16_model.fit(train_VGG16, train_targets,
          validation_data=(valid_VGG16, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=0)
```

```
Epoch 00000: val_loss improved from inf to 7.38981, saving model to
saved_models/weights.best.VGG16.hdf5
Epoch 00001: val_loss improved from 7.38981 to 7.32302, saving model to
saved_models/weights.best.VGG16.hdf5
Epoch 00002: val_loss improved from 7.32302 to 7.27938, saving model to
saved_models/weights.best.VGG16.hdf5
Epoch 00003: val_loss did not improve
…….
Epoch 00019: val_loss improved from 6.88125 to 6.85164, saving model to
saved_models/weights.best.VGG16.hdf5
```

Out[85]:

```
<keras.callbacks.History at 0x166a7848c18>
```

Load the Model with the Best Validation Loss

In [86]:

```
VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

Test the Model

Now, we can use the CNN to test how well it identifies breed within our test dataset of dog images. We print the test accuracy below.

In [87]:

```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature,
axis=0))) for feature in test_VGG16]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG16_predictions)==np.argmax(test_targets,
axis=1))/len(VGG16_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 50.9569%
```

Predict Dog Breed with the Model

In [88]:

```
from extract_bottleneck_features import *

def VGG16_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_VGG16(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = VGG16_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

Step 5: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

In Step 4, we used transfer learning to create a CNN using VGG-16 bottleneck features. In this section, you must use the bottleneck features from a different pre-trained model. To make things easier for you, we have pre-computed the features for all of the networks that are currently available in Keras:

- VGG-19 bottleneck features
- ResNet-50 bottleneck features
- Inception bottleneck features
- Xception bottleneck features

The files are encoded as such:

```
Dog{network}Data.npz
```

where `{network}`, in the above filename, can be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`. Pick one of the above architectures, download the corresponding bottleneck features, and store the downloaded file in the `bottleneck_features/` folder in the repository.

(IMPLEMENTATION) Obtain Bottleneck Features

In the code block below, extract the bottleneck features corresponding to the train, test, and validation sets by running the following:

```
bottleneck_features =
np.load('bottleneck_features/Dog{network}Data.npz')

train_{network} = bottleneck_features['train']

valid_{network} = bottleneck_features['valid']

test_{network} = bottleneck_features['test']
```

In [110]:

```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features_V3 = np.load('bottleneck_features/DogInceptionV3Data.npz')
train_InceptionV3 = bottleneck_features_V3['train']
valid_InceptionV3 = bottleneck_features_V3['valid']
test_InceptionV3 = bottleneck_features_V3['test']
```

(IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
<your model's name>.summary()
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: The table below shows the accuracies obtained for every Deep Neural Net model used to extract features from FLOWERS17 dataset using different parameter settings. https://gogul09.github.io/software/flower-recognition-deep-learning

We can see that the prediction of the net trained with **Xception** should be better with the same all other conditions. Let's check.

First I decided to work with the Inception-V3 as far as it is trained for the ImageNet Large Visual Recognition Challenge using the data from 2012 and has a great position. I did some reading "Building powerful image classification models using very little data" https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html and took some ideas from there about the model architecture and the data augmentation.

To be sure, that my model is great I plotted the result accuracy and losses after model training. That helped me to place the acceptable number of the nodes in all layers and choose the right batch_size and the epochs number.

Checking the accordance of the test graph with the training graph I decided to use three fully connected layers with the **tanh**activation function and **dropouts** with the different parameter.

During my experimenting with the activation functions, I found the **advanced activations** and tried to use them. The best result was achieved with the following combination.

I had to decrease the LeakyReLU parameter from 0.3 through 0.0007 to get the better performance.

Finally, getting the acceptable model I checked it with the Xception, and we can see below that the result is over 83%!!! Not bad as for me)).

In [968]:

```
### TODO: Define your architecture.
from keras.layers.advanced_activations import LeakyReLU, PReLU

inceptionV3_model = Sequential()
inceptionV3_model.add(GlobalAveragePooling2D(input_shape=train_InceptionV3.shape[1:]))
inceptionV3_model.add(LeakyReLU(alpha=0.0007))# add an advanced activation
inceptionV3_model.add(Dense(384, activation='tanh'))
inceptionV3_model.add(PReLU(alpha_initializer='zero', weights=None))# add an advanced activation
inceptionV3_model.add(Dropout(0.2))
inceptionV3_model.add(Dense(256, activation='tanh'))
inceptionV3_model.add(PReLU(alpha_initializer='zero', weights=None))# add an advanced activation
inceptionV3_model.add(Dropout(0.4))
inceptionV3_model.add(Dense(128, activation='tanh'))
inceptionV3_model.add(PReLU(alpha_initializer='zero', weights=None))# add an advanced activation
inceptionV3_model.add(Dropout(0.6))
inceptionV3_model.add(Dense(133, activation='softmax'))

inceptionV3_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_141 (None, 2048)              0
_________________________________________________________________
leaky_re_lu_151 (LeakyReLU)  (None, 2048)              0
_________________________________________________________________
dense_335 (Dense)            (None, 384)               786816
_________________________________________________________________
p_re_lu_180 (PReLU)          (None, 384)               384
_________________________________________________________________
dropout_69 (Dropout)         (None, 384)               0
_________________________________________________________________
dense_336 (Dense)            (None, 256)               98560
_________________________________________________________________
p_re_lu_181 (PReLU)          (None, 256)               256
_________________________________________________________________
dropout_70 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_337 (Dense)            (None, 128)               32896
_________________________________________________________________
p_re_lu_182 (PReLU)          (None, 128)               128
_________________________________________________________________
dropout_71 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_338 (Dense)            (None, 133)               17157
=================================================================
```

```
Total params: 936,197.0
Trainable params: 936,197.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [969]:

```
### TODO: Compile the model.
inceptionV3_model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data, but this is not a requirement.

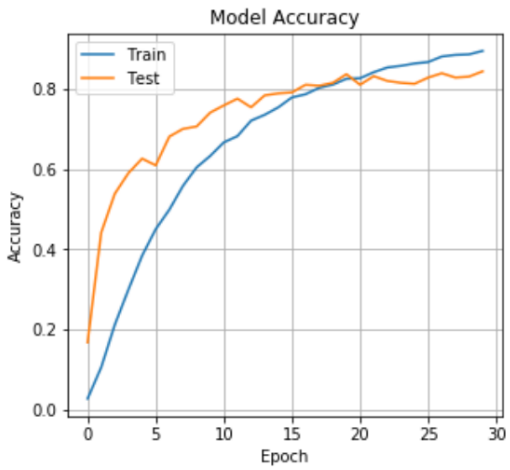
In [970]:

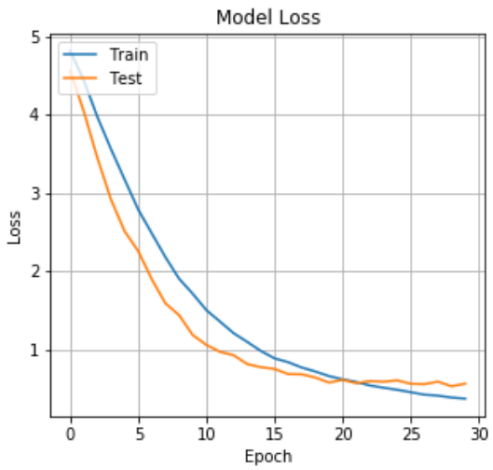
```
epochs = 30
batch_size = 330
checkpointer =
ModelCheckpoint(filepath='saved_models/weights.best.inceptionv3_bneck.hdf5',
                                verbose=1, save_best_only=True)
epochtimer = EpochTimer()
hist_V3 =
model_fit(inceptionV3_model,False,train_InceptionV3,valid_InceptionV3)
```

```
Training without data augmentation.
Epoch 00000: val_loss improved from inf to 4.55914, saving model to
saved_models/weights.best.inceptionv3_bneck.hdf5
Epoch 0 took 9.945023929714807 seconds
.....

Epoch 29 took 3.750745602315874 seconds
Training took 115.29048516748298 seconds
```

In [972]:

```
show_history_graph(hist_V3)
```





(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 60%.

In [974]:

```
### TODO: Calculate classification accuracy on the test dataset.
predictions = [np.argmax(inceptionV3_model.predict(np.expand_dims(feature,
axis=0))) for feature in test_InceptionV3]

# report test accuracy
test_accuracy = 100*np.sum(np.array(predictions)==np.argmax(test_targets,
axis=1))/len(predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 80.3828%
```

In [975]:

```
bottleneck_features_X = np.load('bottleneck_features/DogXceptionData.npz')
train_Xception = bottleneck_features_X['train']
valid_Xception = bottleneck_features_X['valid']
test_Xception = bottleneck_features_X['test']
```

In [985]:

```
model_Xception = Sequential()
model_Xception.add(GlobalAveragePooling2D(input_shape=train_Xception.shape[1:])
)
model_Xception.add(LeakyReLU(alpha=0.0007))# add an advanced activation
model_Xception.add(Dense(384, activation='tanh'))
model_Xception.add(PReLU(alpha_initializer='zero', weights=None))# add an
advanced activation
model_Xception.add(Dropout(0.2))
model_Xception.add(Dense(256, activation='tanh'))
model_Xception.add(PReLU(alpha_initializer='zero', weights=None))# add an
advanced activation
model_Xception.add(Dropout(0.4))
model_Xception.add(Dense(128, activation='tanh'))
model_Xception.add(PReLU(alpha_initializer='zero', weights=None))# add an
advanced activation
model_Xception.add(Dropout(0.6))
model_Xception.add(Dense(133, activation='softmax'))

model_Xception.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_144 (None, 2048)              0
_________________________________________________________________
leaky_re_lu_154 (LeakyReLU)  (None, 2048)              0
_________________________________________________________________
dense_347 (Dense)            (None, 384)               786816
_________________________________________________________________
p_re_lu_189 (PReLU)          (None, 384)               384
_________________________________________________________________
```

```
dropout_78 (Dropout)          (None, 384)               0
_________________________________________________________________
dense_348 (Dense)             (None, 256)               98560
_________________________________________________________________
p_re_lu_190 (PReLU)           (None, 256)               256
_________________________________________________________________
dropout_79 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_349 (Dense)             (None, 128)               32896
_________________________________________________________________
p_re_lu_191 (PReLU)           (None, 128)               128
_________________________________________________________________
dropout_80 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_350 (Dense)             (None, 133)               17157
=================================================================
Total params: 936,197.0
Trainable params: 936,197.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [986]:

```
model_Xception.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
```

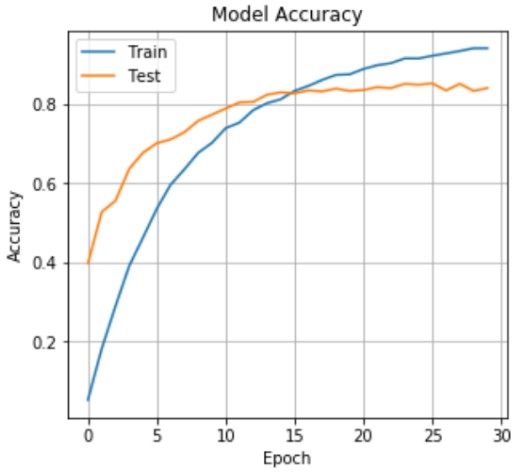
In [987]:

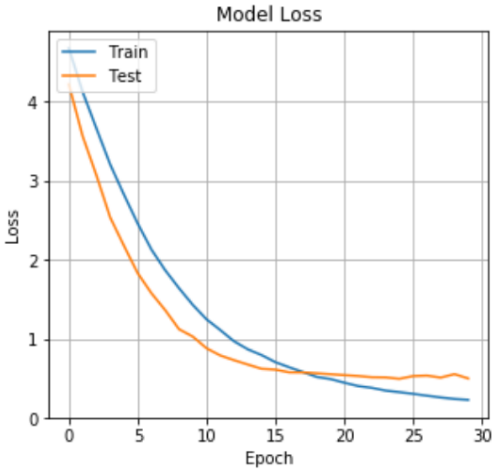
```
epochs = 30
batch_size = 330
checkpointer =
ModelCheckpoint(filepath='saved_models/weights.best.Xception_bneck.hdf5',
                                verbose=1, save_best_only=True)
epochtimer = EpochTimer()
hist_X = model_fit(model_Xception,False,train_Xception,valid_Xception)
```

```
Training without data augmentation.
Epoch 00000: val_loss improved from inf to 4.22130, saving model to
saved_models/weights.best.Xception_bneck.hdf5
Epoch 0 took 21.848149977915455 seconds
……

Epoch 29 took 5.2167124081315706 seconds
Training took 173.55223508083145 seconds
```

In [988]:

```
show_history_graph(hist_X)
```



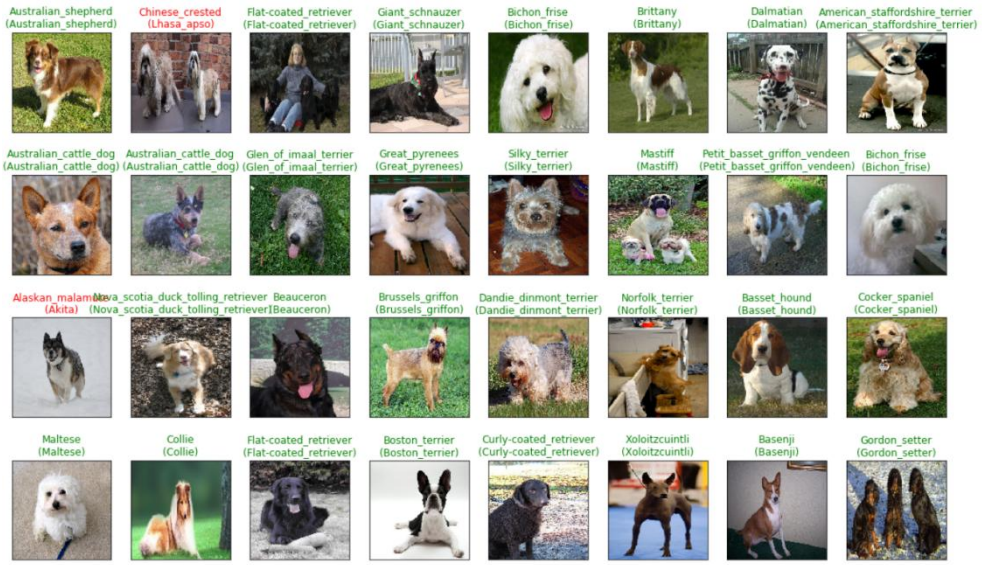



```
inceptionV3_model.load_weights('saved_models/weights.best.Xception_bneck.hdf5')

### TODO: Calculate classification accuracy on the test dataset.
predictions = [np.argmax(model_Xception.predict(np.expand_dims(feature,
axis=0))) for feature in test_Xception]

# report test accuracy
test_accuracy = 100*np.sum(np.array(predictions)==np.argmax(test_targets,
axis=1))/len(predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 83.7321%
```

In [994]:

```
fig = plt.figure(figsize=(20, 12))
for i, idx in enumerate(np.random.choice(test_tensors.shape[0], size=32,
replace=False)):
    ax = fig.add_subplot(4, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(test_tensors[idx]))
    true_idx = np.argmax(test_targets[idx])
    pred_idx = predictions[idx]
    ax.set_title("{}\n({})".format(dog_names[pred_idx], dog_names[true_idx]),
                                color=("green" if pred_idx == true_idx else
"red"))
```



(IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan_hound`, etc) that is predicted by your model.

Similar to the analogous function in Step 5, your function should have three steps:

1. Extract the bottleneck features corresponding to the chosen CNN model.
2. Supply the bottleneck features as input to the model to return the predicted vector. Note that the argmax of this prediction vector gives the index of the predicted dog breed.
3. Use the `dog_names` array defined in Step 0 of this notebook to return the corresponding breed.

The functions to extract the bottleneck features can be found in `extract_bottleneck_features.py`, and they have been imported in an earlier code cell. To obtain the bottleneck features corresponding to your chosen CNN architecture, you need to use the function

```
extract_{network}
```

where `{network}`, in the above filename, should be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`.

In [1019]:

```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
def Xception_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Xception(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = model_Xception.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

Step 6: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- if a **dog** is detected in the image, return the predicted breed.
- if a **human** is detected in the image, return the resembling dog breed.
- if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector`and `dog_detector` functions developed above. You are **required** to use your CNN from Step 5 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

(IMPLEMENTATION) Write your Algorithm

In [1053]:

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
import matplotlib.image as mpimg

def look_at_image(img_path):
    print('Look at this image: {}'.format(img_path))
```

```
    img = mpimg.imread(img_path)
    fig = plt.figure()
    plt.subplot()
    plt.imshow(img)
    plt.axis('off')
    plt.plot()

def object_detector(img_path):
    human_detected = face_detector(img_path)
    dog_detected = dog_detector(img_path)

    if human_detected:
        print('It seems this is a Human')
    elif dog_detected:
        print('To my mind this is a Dog')
    else:
        print('What the hell is this!!!')
    pred = Xception_predict_breed(img_path)
    print('It looks like a {}...'.format(pred))
```

Step 7: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that **you** look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

In [1040]:

```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
def check_image(img_path):
    look_at_image(img_path)
    object_detector(img_path)
```

In [1041]:

```
check_image('dogImages/train/004.Akita/Akita_00221.jpg')
```

```
Look at this image: dogImages/train/004.Akita/Akita_00221.jpg
To my mind this is a Dog
It looks like a Akita...
```