COMS W4705 - Homework 4

Image Captioning with Conditioned LSTM Generators

Yassine Benajiba yb2235@cs.columbia.edu

Follow the instructions in this notebook step-by step. Much of the code is provided, but some sections are marked with todo.

Specifically, you will build the following components:

- **Create matrices of image representations using an off-the-shelf image encoder.**
- **Read and preprocess the image captions.**
- **Write a generator function that returns one training instance (input/output sequence pair) at a time.**
- **Train an LSTM language generator on the caption data.**
- **Write a decoder function for the language generator.**
- **Add the image input to write an LSTM caption generator.**
- **Implement beam search for the image caption generator.**

Please submit a copy of this notebook only, including all outputs. Do not submit any of the data files.

Getting Started

First, run the following commands to make sure you have all required packages.

In [1]:

```
import os
from collections import defaultdict
import numpy as np
import PIL
from matplotlib import pyplot as plt
%matplotlib inline

from keras import Sequential, Model
from keras.layers import Embedding, LSTM, Dense, Input, Bidirectional, RepeatVector, Con
catenate, Activation
from keras.activations import softmax
from tensorflow.keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences

from keras.applications.inception_v3 import InceptionV3

from tensorflow.keras.optimizers import Adam

from google.colab import drive
```

Access to the flickr8k data

We will use the flickr8k data set, described here in more detail:

> **M. Hodosh, P. Young and J. Hockenmaier (2013) "Framing Image Description as a Ranking Task: Data, Models and Evaluation Metrics", Journal of Artificial Intelligence Research, Volume 47, pages 853-899 http://www.jair.org/papers/paper3994.html when discussing our results**

I have uploaded all the data and model files you'll need to my GDrive and you can access the folder here: https://drive.google.com/drive/folders/1i9Iun4h3EN1vSd1A1woez0mXJ9vRjFIT?usp=sharing

Google Drive does not allow to copy a folder, so you'll need to download the whole folder and then upload it again to your own drive. Please assign the name you chose for this folder to the variable `my_data_dir` in the

again to your own drive. Please assign the name you chose for this folder to the variable `my_data_dir` in the next cell.

N.B.: Usage of this data is limited to this homework assignment. If you would like to experiment with the data set beyond this course, I suggest that you submit your owndownload request here: https://forms.illinois.edu/sec/1713398

In [2]:

```
#this is where you put the name of your data folder.
#Please make sure it's correct because it'll be used in many places later.
my_data_dir="NLP_HW4_Data"
```

Mounting your GDrive so you can access the files from Colab

In [3]:

```
#running this command will generate a message that will ask you to click on a link where you'll obtain your GDrive auth code.
#copy paste that code in the text box that will appear below
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

Please look at the 'Files' tab on the left side and make sure you can see the 'hw5_data' folder that you have in your GDrive.

Part I: Image Encodings (14 pts)

The files Flickr_8k.trainImages.txt Flickr_8k.devImages.txt Flickr_8k.testImages.txt, contain a list of training, development, and test images, respectively. Let's load these lists.

In [4]:

```
def load_image_list(filename):
    with open(filename,'r') as image_list_f:
        return [line.strip() for line in image_list_f]
```

In [5]:

```
train_list = load_image_list('/content/gdrive/My Drive/'+my_data_dir+'/Flickr_8k.trainImages.txt')
dev_list = load_image_list('/content/gdrive/My Drive/'+my_data_dir+'/Flickr_8k.devImages.txt')
test_list = load_image_list('/content/gdrive/My Drive/'+my_data_dir+'/Flickr_8k.testImages.txt')
```

Let's see how many images there are

In [6]:

```
len(train_list), len(dev_list), len(test_list)
```

Out[6]:

```
(6000, 1000, 1000)
```

Each entry is an image filename.

In [7]:

```
dev_list[20]
```

Out[7]:

```
'3693961165_9d6c333d5b.jpg'
```

The images are located in a subdirectory.

In [8]:

```
cd gdrive/MyDrive/NLP_HW4_Data/
```

```
/content/gdrive/MyDrive/NLP_HW4_Data
```

In [10]:

```
IMG_PATH = "Flickr8k_Dataset"
```

We can use PIL to open the image and matplotlib to display it.

In [11]:

```
image = PIL.Image.open(os.path.join(IMG_PATH, dev_list[20]))
image
```

Out[11]:

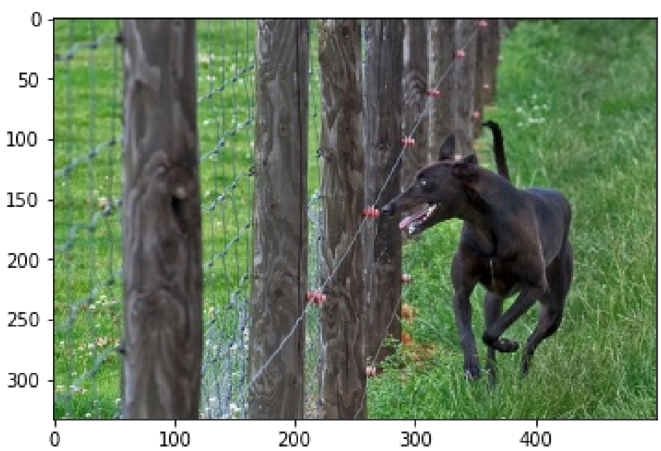
if you can't see the image, try

In [12]:

```
plt.imshow(image)
```

Out[12]:

```
<matplotlib.image.AxesImage at 0x7fd0b7a2a2d0>
```



We are going to use an off-the-shelf pre-trained image encoder, the Inception V3 network. The model is a

version of a convolution neural network for object detection. Here is more detail about this model (not required for this project):

> Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826). https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Szegedy_Rethinking_the_Inception_CVPR_2016_pape

The model requires that input images are presented as 299x299 pixels, with 3 color channels (RGB). The individual RGB values need to range between 0 and 1.0. The flickr images don't fit.

In [13]:

```
np.asarray(image).shape
```

Out[13]:

```
(333, 500, 3)
```

The values range from 0 to 255.

In [14]:

```
np.asarray(image)
```

Out[14]:

```
array([[[118, 161,  89],
        [120, 164,  89],
        [111, 157,  82],
        ...,
        [ 68, 106,  65],
        [ 64, 102,  61],
        [ 65, 104,  60]],

       [[125, 168,  96],
        [121, 164,  92],
        [119, 165,  90],
        ...,
        [ 72, 115,  72],
        [ 65, 108,  65],
        [ 72, 115,  70]],

       [[129, 175, 102],
        [123, 169,  96],
        [115, 161,  88],
        ...,
        [ 88, 129,  87],
        [ 75, 116,  72],
        [ 75, 116,  72]],

       ...,

       [[ 41, 118,  46],
        [ 36, 113,  41],
        [ 45, 111,  49],
        ...,
        [ 23,  77,  15],
        [ 60, 114,  62],
        [ 19,  59,   0]],

       [[100, 158,  97],
        [ 38, 100,  37],
        [ 46, 117,  51],
        ...,
        [ 25,  54,   8],
        [ 88, 112,  76],
        [ 65, 106,  48]],
```

```
        [[ 89, 148,  84],
         [ 44, 112,  35],
         [ 71, 130,  72],
         ...,
         [152, 188, 142],
         [113, 151, 110],
         [ 94, 138,  75]]], dtype=uint8)
```

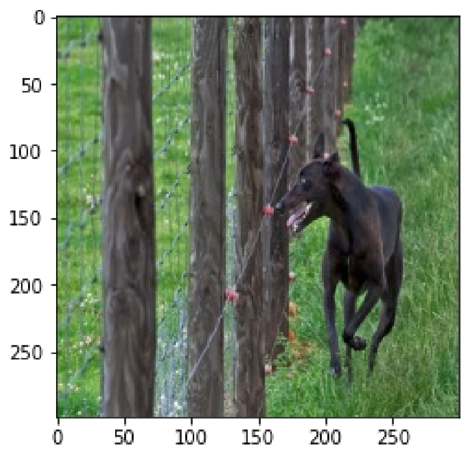
We can use PIL to resize the image and then divide every value by 255.

In [15]:

```
new_image = np.asarray(image.resize((299,299))) / 255.0
plt.imshow(new_image)
```

Out[15]:

```
<matplotlib.image.AxesImage at 0x7fd0b75210d0>
```



In [16]:

```
new_image.shape
```

Out[16]:

```
(299, 299, 3)
```

Let's put this all in a function for convenience.

In [17]:

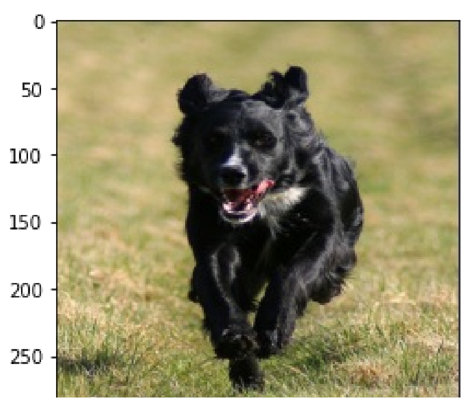
```
def get_image(image_name):
    image = PIL.Image.open(os.path.join(IMG_PATH, image_name))
    return np.asarray(image.resize((299,299))) / 255.0
```

In [18]:

```
plt.imshow(get_image(dev_list[25]))
```

Out[18]:

```
<matplotlib.image.AxesImage at 0x7fd0b748bfd0>
```



Next, we load the pre-trained Inception model.

In [19]:

```
img_model = InceptionV3(weights='imagenet') # This will download the weight files for you and might take a while.
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels.h5
96116736/96112376 [==============================] - 1s 0us/step
96124928/96112376 [==============================] - 1s 0us/step
```

In [20]:

```
img_model.summary() # this is quite a complex model.
```

```
Model: "inception_v3"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to

==================================================================================================
 input_1 (InputLayer)           [(None, 299, 299, 3  0           []
                                )]


 conv2d (Conv2D)                (None, 149, 149, 32  864         ['input_1[0][0]']
                                )


 batch_normalization (BatchNorm  (None, 149, 149, 32  96         ['conv2d[0][0]']
 alization)                     )


 activation (Activation)        (None, 149, 149, 32  0           ['batch_normalization[0][0]']
                                )


 conv2d_1 (Conv2D)              (None, 147, 147, 32  9216        ['activation[0][0]']
                                )


 batch_normalization_1 (BatchNo  (None, 147, 147, 32  96         ['conv2d_1[0][0]']
 rmalization)                   )


 activation_1 (Activation)      (None, 147, 147, 32  0           ['batch_normalization_1[0][0]']
                                )


 conv2d_2 (Conv2D)              (None, 147, 147, 64  18432       ['activation_1[0][0]']
                                )
```

```
 batch_normalization_2 (BatchNo  (None, 147, 147, 64  192         ['conv2d_2[0][0]']
 rmalization)                   )

 activation_2 (Activation)      (None, 147, 147, 64  0           ['batch_normalization_2[
0][0]']
                                )

 max_pooling2d (MaxPooling2D)   (None, 73, 73, 64)   0           ['activation_2[0][0]']

 conv2d_3 (Conv2D)              (None, 73, 73, 80)   5120        ['max_pooling2d[0][0]']

 batch_normalization_3 (BatchNo  (None, 73, 73, 80)  240         ['conv2d_3[0][0]']
 rmalization)

 activation_3 (Activation)      (None, 73, 73, 80)   0           ['batch_normalization_3[
0][0]']
 conv2d_4 (Conv2D)              (None, 71, 71, 192)  138240      ['activation_3[0][0]']

 batch_normalization_4 (BatchNo  (None, 71, 71, 192)  576        ['conv2d_4[0][0]']
 rmalization)

 activation_4 (Activation)      (None, 71, 71, 192)  0           ['batch_normalization_4[
0][0]']
 max_pooling2d_1 (MaxPooling2D)  (None, 35, 35, 192)  0          ['activation_4[0][0]']

 conv2d_8 (Conv2D)              (None, 35, 35, 64)   12288       ['max_pooling2d_1[0][0]'
]
 batch_normalization_8 (BatchNo  (None, 35, 35, 64)  192         ['conv2d_8[0][0]']
 rmalization)

 activation_8 (Activation)      (None, 35, 35, 64)   0           ['batch_normalization_8[
0][0]']
 conv2d_6 (Conv2D)              (None, 35, 35, 48)   9216        ['max_pooling2d_1[0][0]'
]
 conv2d_9 (Conv2D)              (None, 35, 35, 96)   55296       ['activation_8[0][0]']
```

```
 batch_normalization_6 (BatchNo  (None, 35, 35, 48)  144         ['conv2d_6[0][0]']

 rmalization)


 batch_normalization_9 (BatchNo  (None, 35, 35, 96)  288         ['conv2d_9[0][0]']

 rmalization)


 activation_6 (Activation)      (None, 35, 35, 48)   0           ['batch_normalization_6[
0][0]']

 activation_9 (Activation)      (None, 35, 35, 96)   0           ['batch_normalization_9[
0][0]']

 average_pooling2d (AveragePool  (None, 35, 35, 192)  0          ['max_pooling2d_1[0][0]'
]
 ing2D)


 conv2d_5 (Conv2D)              (None, 35, 35, 64)   12288       ['max_pooling2d_1[0][0]'
]

 conv2d_7 (Conv2D)              (None, 35, 35, 64)   76800       ['activation_6[0][0]']


 conv2d_10 (Conv2D)             (None, 35, 35, 96)   82944       ['activation_9[0][0]']


 conv2d_11 (Conv2D)             (None, 35, 35, 32)   6144        ['average_pooling2d[0][0
]']

 batch_normalization_5 (BatchNo  (None, 35, 35, 64)  192         ['conv2d_5[0][0]']

 rmalization)


 batch_normalization_7 (BatchNo  (None, 35, 35, 64)  192         ['conv2d_7[0][0]']

 rmalization)


 batch_normalization_10 (BatchN  (None, 35, 35, 96)  288         ['conv2d_10[0][0]']

 ormalization)


 batch_normalization_11 (BatchN  (None, 35, 35, 32)  96          ['conv2d_11[0][0]']

 ormalization)


 activation_5 (Activation)      (None, 35, 35, 64)   0           ['batch_normalization_5[
0][0]']

 activation_7 (Activation)      (None, 35, 35, 64)   0           ['batch_normalization_7[
```

```
0][0]']

 activation_10 (Activation)     (None, 35, 35, 96)   0           ['batch_normalization_10
[0][0]']

 activation_11 (Activation)     (None, 35, 35, 32)   0           ['batch_normalization_11
[0][0]']

 mixed0 (Concatenate)           (None, 35, 35, 256)  0           ['activation_5[0][0]',
                                                                  'activation_7[0][0]',
                                                                  'activation_10[0][0]'
,
                                                                  'activation_11[0][0]'
]

 conv2d_15 (Conv2D)             (None, 35, 35, 64)   16384       ['mixed0[0][0]']

 batch_normalization_15 (BatchN  (None, 35, 35, 64)  192         ['conv2d_15[0][0]']
 ormalization)

 activation_15 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_15
[0][0]']

 conv2d_13 (Conv2D)             (None, 35, 35, 48)   12288       ['mixed0[0][0]']

 conv2d_16 (Conv2D)             (None, 35, 35, 96)   55296       ['activation_15[0][0]']

 batch_normalization_13 (BatchN  (None, 35, 35, 48)  144         ['conv2d_13[0][0]']
 ormalization)

 batch_normalization_16 (BatchN  (None, 35, 35, 96)  288         ['conv2d_16[0][0]']
 ormalization)

 activation_13 (Activation)     (None, 35, 35, 48)   0           ['batch_normalization_13
[0][0]']

 activation_16 (Activation)     (None, 35, 35, 96)   0           ['batch_normalization_16
[0][0]']

 average_pooling2d_1 (AveragePo  (None, 35, 35, 256)  0          ['mixed0[0][0]']
 oling2D)

 conv2d_12 (Conv2D)             (None, 35, 35, 64)   16384       ['mixed0[0][0]']
```

```
 conv2d_14 (Conv2D)             (None, 35, 35, 64)   76800       ['activation_13[0][0]']


 conv2d_17 (Conv2D)             (None, 35, 35, 96)   82944       ['activation_16[0][0]']


 conv2d_18 (Conv2D)             (None, 35, 35, 64)   16384       ['average_pooling2d_1[0]
[0]']

 batch_normalization_12 (BatchN  (None, 35, 35, 64)  192         ['conv2d_12[0][0]']

 ormalization)


 batch_normalization_14 (BatchN  (None, 35, 35, 64)  192         ['conv2d_14[0][0]']

 ormalization)


 batch_normalization_17 (BatchN  (None, 35, 35, 96)  288         ['conv2d_17[0][0]']

 ormalization)


 batch_normalization_18 (BatchN  (None, 35, 35, 64)  192         ['conv2d_18[0][0]']

 ormalization)


 activation_12 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_12
[0][0]']

 activation_14 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_14
[0][0]']

 activation_17 (Activation)     (None, 35, 35, 96)   0           ['batch_normalization_17
[0][0]']

 activation_18 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_18
[0][0]']

 mixed1 (Concatenate)           (None, 35, 35, 288)  0           ['activation_12[0][0]',
                                                                  'activation_14[0][0]'
,
                                                                  'activation_17[0][0]'
,
                                                                  'activation_18[0][0]'
]

 conv2d_22 (Conv2D)             (None, 35, 35, 64)   18432       ['mixed1[0][0]']


 batch_normalization_22 (BatchN  (None, 35, 35, 64)  192         ['conv2d_22[0][0]']

 ormalization)
```

```
 activation_22 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_22
[0][0]']

 conv2d_20 (Conv2D)             (None, 35, 35, 48)   13824       ['mixed1[0][0]']


 conv2d_23 (Conv2D)             (None, 35, 35, 96)   55296       ['activation_22[0][0]']


 batch_normalization_20 (BatchN  (None, 35, 35, 48)  144         ['conv2d_20[0][0]']

 ormalization)


 batch_normalization_23 (BatchN  (None, 35, 35, 96)  288         ['conv2d_23[0][0]']

 ormalization)


 activation_20 (Activation)     (None, 35, 35, 48)   0           ['batch_normalization_20
[0][0]']

 activation_23 (Activation)     (None, 35, 35, 96)   0           ['batch_normalization_23
[0][0]']

 average_pooling2d_2 (AveragePo  (None, 35, 35, 288)  0          ['mixed1[0][0]']

 oling2D)


 conv2d_19 (Conv2D)             (None, 35, 35, 64)   18432       ['mixed1[0][0]']


 conv2d_21 (Conv2D)             (None, 35, 35, 64)   76800       ['activation_20[0][0]']


 conv2d_24 (Conv2D)             (None, 35, 35, 96)   82944       ['activation_23[0][0]']


 conv2d_25 (Conv2D)             (None, 35, 35, 64)   18432       ['average_pooling2d_2[0]
[0]']

 batch_normalization_19 (BatchN  (None, 35, 35, 64)  192         ['conv2d_19[0][0]']

 ormalization)


 batch_normalization_21 (BatchN  (None, 35, 35, 64)  192         ['conv2d_21[0][0]']

 ormalization)


 batch_normalization_24 (BatchN  (None, 35, 35, 96)  288         ['conv2d_24[0][0]']

 ormalization)
```

```
 batch_normalization_25 (BatchN  (None, 35, 35, 64)  192         ['conv2d_25[0][0]']

 ormalization)


 activation_19 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_19
[0][0]']

 activation_21 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_21
[0][0]']

 activation_24 (Activation)     (None, 35, 35, 96)   0           ['batch_normalization_24
[0][0]']

 activation_25 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_25
[0][0]']

 mixed2 (Concatenate)           (None, 35, 35, 288)  0           ['activation_19[0][0]',

                                                                  'activation_21[0][0]'
,
                                                                  'activation_24[0][0]'
,
                                                                  'activation_25[0][0]'
]

 conv2d_27 (Conv2D)             (None, 35, 35, 64)   18432       ['mixed2[0][0]']


 batch_normalization_27 (BatchN  (None, 35, 35, 64)  192         ['conv2d_27[0][0]']

 ormalization)


 activation_27 (Activation)     (None, 35, 35, 64)   0           ['batch_normalization_27
[0][0]']

 conv2d_28 (Conv2D)             (None, 35, 35, 96)   55296       ['activation_27[0][0]']


 batch_normalization_28 (BatchN  (None, 35, 35, 96)  288         ['conv2d_28[0][0]']

 ormalization)


 activation_28 (Activation)     (None, 35, 35, 96)   0           ['batch_normalization_28
[0][0]']

 conv2d_26 (Conv2D)             (None, 17, 17, 384)  995328      ['mixed2[0][0]']


 conv2d_29 (Conv2D)             (None, 17, 17, 96)   82944       ['activation_28[0][0]']


 batch_normalization_26 (BatchN  (None, 17, 17, 384)  1152       ['conv2d_26[0][0]']

 ormalization)
```

```
 batch_normalization_29 (BatchN  (None, 17, 17, 96)  288         ['conv2d_29[0][0]']
 ormalization)

 activation_26 (Activation)     (None, 17, 17, 384)  0           ['batch_normalization_26
[0][0]']

 activation_29 (Activation)     (None, 17, 17, 96)   0           ['batch_normalization_29
[0][0]']

 max_pooling2d_2 (MaxPooling2D)  (None, 17, 17, 288)  0          ['mixed2[0][0]']

 mixed3 (Concatenate)           (None, 17, 17, 768)  0           ['activation_26[0][0]',
                                                                  'activation_29[0][0]'
,
                                                                  'max_pooling2d_2[0][0
]']

 conv2d_34 (Conv2D)             (None, 17, 17, 128)  98304       ['mixed3[0][0]']

 batch_normalization_34 (BatchN  (None, 17, 17, 128)  384        ['conv2d_34[0][0]']
 ormalization)

 activation_34 (Activation)     (None, 17, 17, 128)  0           ['batch_normalization_34
[0][0]']

 conv2d_35 (Conv2D)             (None, 17, 17, 128)  114688      ['activation_34[0][0]']

 batch_normalization_35 (BatchN  (None, 17, 17, 128)  384        ['conv2d_35[0][0]']
 ormalization)

 activation_35 (Activation)     (None, 17, 17, 128)  0           ['batch_normalization_35
[0][0]']

 conv2d_31 (Conv2D)             (None, 17, 17, 128)  98304       ['mixed3[0][0]']

 conv2d_36 (Conv2D)             (None, 17, 17, 128)  114688      ['activation_35[0][0]']

 batch_normalization_31 (BatchN  (None, 17, 17, 128)  384        ['conv2d_31[0][0]']
 ormalization)

 batch_normalization_36 (BatchN  (None, 17, 17, 128)  384        ['conv2d_36[0][0]']
```

```
 ormalization)


 activation_31 (Activation)     (None, 17, 17, 128)  0           ['batch_normalization_31
[0][0]']

 activation_36 (Activation)     (None, 17, 17, 128)  0           ['batch_normalization_36
[0][0]']

 conv2d_32 (Conv2D)             (None, 17, 17, 128)  114688      ['activation_31[0][0]']


 conv2d_37 (Conv2D)             (None, 17, 17, 128)  114688      ['activation_36[0][0]']


 batch_normalization_32 (BatchN  (None, 17, 17, 128)  384        ['conv2d_32[0][0]']
 ormalization)


 batch_normalization_37 (BatchN  (None, 17, 17, 128)  384        ['conv2d_37[0][0]']
 ormalization)


 activation_32 (Activation)     (None, 17, 17, 128)  0           ['batch_normalization_32
[0][0]']

 activation_37 (Activation)     (None, 17, 17, 128)  0           ['batch_normalization_37
[0][0]']

 average_pooling2d_3 (AveragePo  (None, 17, 17, 768)  0          ['mixed3[0][0]']
 oling2D)


 conv2d_30 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed3[0][0]']


 conv2d_33 (Conv2D)             (None, 17, 17, 192)  172032      ['activation_32[0][0]']


 conv2d_38 (Conv2D)             (None, 17, 17, 192)  172032      ['activation_37[0][0]']


 conv2d_39 (Conv2D)             (None, 17, 17, 192)  147456      ['average_pooling2d_3[0]
[0]']

 batch_normalization_30 (BatchN  (None, 17, 17, 192)  576        ['conv2d_30[0][0]']
 ormalization)


 batch_normalization_33 (BatchN  (None, 17, 17, 192)  576        ['conv2d_33[0][0]']
 ormalization)
```

```
 batch_normalization_38 (BatchN  (None, 17, 17, 192)  576         ['conv2d_38[0][0]']

 ormalization)


 batch_normalization_39 (BatchN  (None, 17, 17, 192)  576         ['conv2d_39[0][0]']

 ormalization)


 activation_30 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_30
[0][0]']

 activation_33 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_33
[0][0]']

 activation_38 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_38
[0][0]']

 activation_39 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_39
[0][0]']

 mixed4 (Concatenate)           (None, 17, 17, 768)  0           ['activation_30[0][0]',

                                                                  'activation_33[0][0]'
,
                                                                  'activation_38[0][0]'
,
                                                                  'activation_39[0][0]'
]

 conv2d_44 (Conv2D)             (None, 17, 17, 160)  122880      ['mixed4[0][0]']


 batch_normalization_44 (BatchN  (None, 17, 17, 160)  480         ['conv2d_44[0][0]']

 ormalization)


 activation_44 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_44
[0][0]']

 conv2d_45 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_44[0][0]']


 batch_normalization_45 (BatchN  (None, 17, 17, 160)  480         ['conv2d_45[0][0]']

 ormalization)


 activation_45 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_45
[0][0]']

 conv2d_41 (Conv2D)             (None, 17, 17, 160)  122880      ['mixed4[0][0]']
```

```
 conv2d_46 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_45[0][0]']


 batch_normalization_41 (BatchN  (None, 17, 17, 160)  480        ['conv2d_41[0][0]']

 ormalization)


 batch_normalization_46 (BatchN  (None, 17, 17, 160)  480        ['conv2d_46[0][0]']

 ormalization)


 activation_41 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_41
[0][0]']

 activation_46 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_46
[0][0]']

 conv2d_42 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_41[0][0]']


 conv2d_47 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_46[0][0]']


 batch_normalization_42 (BatchN  (None, 17, 17, 160)  480        ['conv2d_42[0][0]']

 ormalization)


 batch_normalization_47 (BatchN  (None, 17, 17, 160)  480        ['conv2d_47[0][0]']

 ormalization)


 activation_42 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_42
[0][0]']

 activation_47 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_47
[0][0]']

 average_pooling2d_4 (AveragePo  (None, 17, 17, 768)  0          ['mixed4[0][0]']

 oling2D)


 conv2d_40 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed4[0][0]']


 conv2d_43 (Conv2D)             (None, 17, 17, 192)  215040      ['activation_42[0][0]']


 conv2d_48 (Conv2D)             (None, 17, 17, 192)  215040      ['activation_47[0][0]']


 conv2d_49 (Conv2D)             (None, 17, 17, 192)  147456      ['average_pooling2d_4[0]
```

```
[0]']

 batch_normalization_40 (BatchN  (None, 17, 17, 192)  576         ['conv2d_40[0][0]']
 ormalization)

 batch_normalization_43 (BatchN  (None, 17, 17, 192)  576         ['conv2d_43[0][0]']
 ormalization)

 batch_normalization_48 (BatchN  (None, 17, 17, 192)  576         ['conv2d_48[0][0]']
 ormalization)

 batch_normalization_49 (BatchN  (None, 17, 17, 192)  576         ['conv2d_49[0][0]']
 ormalization)

 activation_40 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_40
[0][0]']

 activation_43 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_43
[0][0]']

 activation_48 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_48
[0][0]']

 activation_49 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_49
[0][0]']

 mixed5 (Concatenate)           (None, 17, 17, 768)  0           ['activation_40[0][0]',
                                                                  'activation_43[0][0]'
,
                                                                  'activation_48[0][0]'
,
                                                                  'activation_49[0][0]'
]

 conv2d_54 (Conv2D)             (None, 17, 17, 160)  122880      ['mixed5[0][0]']

 batch_normalization_54 (BatchN  (None, 17, 17, 160)  480         ['conv2d_54[0][0]']
 ormalization)

 activation_54 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_54
[0][0]']

 conv2d_55 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_54[0][0]']

 batch_normalization_55 (BatchN  (None, 17, 17, 160)  480         ['conv2d_55[0][0]']
```

```
 ormalization)


 activation_55 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_55
[0][0]']

 conv2d_51 (Conv2D)             (None, 17, 17, 160)  122880      ['mixed5[0][0]']


 conv2d_56 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_55[0][0]']


 batch_normalization_51 (BatchN  (None, 17, 17, 160)  480         ['conv2d_51[0][0]']

 ormalization)


 batch_normalization_56 (BatchN  (None, 17, 17, 160)  480         ['conv2d_56[0][0]']

 ormalization)


 activation_51 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_51
[0][0]']

 activation_56 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_56
[0][0]']

 conv2d_52 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_51[0][0]']


 conv2d_57 (Conv2D)             (None, 17, 17, 160)  179200      ['activation_56[0][0]']


 batch_normalization_52 (BatchN  (None, 17, 17, 160)  480         ['conv2d_52[0][0]']

 ormalization)


 batch_normalization_57 (BatchN  (None, 17, 17, 160)  480         ['conv2d_57[0][0]']

 ormalization)


 activation_52 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_52
[0][0]']

 activation_57 (Activation)     (None, 17, 17, 160)  0           ['batch_normalization_57
[0][0]']

 average_pooling2d_5 (AveragePo  (None, 17, 17, 768)  0           ['mixed5[0][0]']

 oling2D)


 conv2d_50 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed5[0][0]']
```

```
 conv2d_53 (Conv2D)             (None, 17, 17, 192)  215040      ['activation_52[0][0]']


 conv2d_58 (Conv2D)             (None, 17, 17, 192)  215040      ['activation_57[0][0]']


 conv2d_59 (Conv2D)             (None, 17, 17, 192)  147456      ['average_pooling2d_5[0]
[0]']

 batch_normalization_50 (BatchN  (None, 17, 17, 192)  576        ['conv2d_50[0][0]']

 ormalization)


 batch_normalization_53 (BatchN  (None, 17, 17, 192)  576        ['conv2d_53[0][0]']

 ormalization)


 batch_normalization_58 (BatchN  (None, 17, 17, 192)  576        ['conv2d_58[0][0]']

 ormalization)


 batch_normalization_59 (BatchN  (None, 17, 17, 192)  576        ['conv2d_59[0][0]']

 ormalization)


 activation_50 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_50
[0][0]']

 activation_53 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_53
[0][0]']

 activation_58 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_58
[0][0]']

 activation_59 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_59
[0][0]']

 mixed6 (Concatenate)           (None, 17, 17, 768)  0           ['activation_50[0][0]',

                                                                  'activation_53[0][0]'
,
                                                                  'activation_58[0][0]'
,
                                                                  'activation_59[0][0]'
]

 conv2d_64 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed6[0][0]']


 batch_normalization_64 (BatchN  (None, 17, 17, 192)  576        ['conv2d_64[0][0]']

 ormalization)
```

```
 activation_64 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_64
[0][0]']

 conv2d_65 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_64[0][0]']

 batch_normalization_65 (BatchN  (None, 17, 17, 192)  576        ['conv2d_65[0][0]']
 ormalization)

 activation_65 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_65
[0][0]']

 conv2d_61 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed6[0][0]']

 conv2d_66 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_65[0][0]']

 batch_normalization_61 (BatchN  (None, 17, 17, 192)  576        ['conv2d_61[0][0]']
 ormalization)

 batch_normalization_66 (BatchN  (None, 17, 17, 192)  576        ['conv2d_66[0][0]']
 ormalization)

 activation_61 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_61
[0][0]']

 activation_66 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_66
[0][0]']

 conv2d_62 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_61[0][0]']

 conv2d_67 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_66[0][0]']

 batch_normalization_62 (BatchN  (None, 17, 17, 192)  576        ['conv2d_62[0][0]']
 ormalization)

 batch_normalization_67 (BatchN  (None, 17, 17, 192)  576        ['conv2d_67[0][0]']
 ormalization)

 activation_62 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_62
[0][0]']
```

```
 activation_67 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_67
[0][0]']

 average_pooling2d_6 (AveragePo  (None, 17, 17, 768)  0           ['mixed6[0][0]']
 oling2D)

 conv2d_60 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed6[0][0]']

 conv2d_63 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_62[0][0]']

 conv2d_68 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_67[0][0]']

 conv2d_69 (Conv2D)             (None, 17, 17, 192)  147456      ['average_pooling2d_6[0]
[0]']

 batch_normalization_60 (BatchN  (None, 17, 17, 192)  576         ['conv2d_60[0][0]']
 ormalization)

 batch_normalization_63 (BatchN  (None, 17, 17, 192)  576         ['conv2d_63[0][0]']
 ormalization)

 batch_normalization_68 (BatchN  (None, 17, 17, 192)  576         ['conv2d_68[0][0]']
 ormalization)

 batch_normalization_69 (BatchN  (None, 17, 17, 192)  576         ['conv2d_69[0][0]']
 ormalization)

 activation_60 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_60
[0][0]']

 activation_63 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_63
[0][0]']

 activation_68 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_68
[0][0]']

 activation_69 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_69
[0][0]']

 mixed7 (Concatenate)           (None, 17, 17, 768)  0           ['activation_60[0][0]',
                                                                  'activation_63[0][0]'
,
                                                                  'activation_68[0][0]'
```

```
,
                                                                 'activation_69[0][0]'
]

 conv2d_72 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed7[0][0]']

 batch_normalization_72 (BatchN  (None, 17, 17, 192)  576        ['conv2d_72[0][0]']
 ormalization)

 activation_72 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_72
[0][0]']

 conv2d_73 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_72[0][0]']

 batch_normalization_73 (BatchN  (None, 17, 17, 192)  576        ['conv2d_73[0][0]']
 ormalization)

 activation_73 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_73
[0][0]']

 conv2d_70 (Conv2D)             (None, 17, 17, 192)  147456      ['mixed7[0][0]']

 conv2d_74 (Conv2D)             (None, 17, 17, 192)  258048      ['activation_73[0][0]']

 batch_normalization_70 (BatchN  (None, 17, 17, 192)  576        ['conv2d_70[0][0]']
 ormalization)

 batch_normalization_74 (BatchN  (None, 17, 17, 192)  576        ['conv2d_74[0][0]']
 ormalization)

 activation_70 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_70
[0][0]']

 activation_74 (Activation)     (None, 17, 17, 192)  0           ['batch_normalization_74
[0][0]']

 conv2d_71 (Conv2D)             (None, 8, 8, 320)    552960      ['activation_70[0][0]']

 conv2d_75 (Conv2D)             (None, 8, 8, 192)    331776      ['activation_74[0][0]']

 batch_normalization_71 (BatchN  (None, 8, 8, 320)   960         ['conv2d_71[0][0]']
 ormalization)
```

```
 batch_normalization_75 (BatchN  (None, 8, 8, 192)   576         ['conv2d_75[0][0]']
 ormalization)


 activation_71 (Activation)     (None, 8, 8, 320)    0           ['batch_normalization_71
[0][0]']

 activation_75 (Activation)     (None, 8, 8, 192)    0           ['batch_normalization_75
[0][0]']

 max_pooling2d_3 (MaxPooling2D)  (None, 8, 8, 768)   0           ['mixed7[0][0]']


 mixed8 (Concatenate)           (None, 8, 8, 1280)   0           ['activation_71[0][0]',
                                                                  'activation_75[0][0]'
,
                                                                  'max_pooling2d_3[0][0
]']

 conv2d_80 (Conv2D)             (None, 8, 8, 448)    573440      ['mixed8[0][0]']


 batch_normalization_80 (BatchN  (None, 8, 8, 448)   1344        ['conv2d_80[0][0]']
 ormalization)


 activation_80 (Activation)     (None, 8, 8, 448)    0           ['batch_normalization_80
[0][0]']

 conv2d_77 (Conv2D)             (None, 8, 8, 384)    491520      ['mixed8[0][0]']


 conv2d_81 (Conv2D)             (None, 8, 8, 384)    1548288     ['activation_80[0][0]']


 batch_normalization_77 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_77[0][0]']
 ormalization)


 batch_normalization_81 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_81[0][0]']
 ormalization)


 activation_77 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_77
[0][0]']

 activation_81 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_81
[0][0]']

 conv2d_78 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_77[0][0]']
```

```
 conv2d_79 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_77[0][0]']

 conv2d_82 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_81[0][0]']

 conv2d_83 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_81[0][0]']

 average_pooling2d_7 (AveragePo  (None, 8, 8, 1280)  0           ['mixed8[0][0]']
 oling2D)

 conv2d_76 (Conv2D)             (None, 8, 8, 320)    409600      ['mixed8[0][0]']

 batch_normalization_78 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_78[0][0]']
 ormalization)

 batch_normalization_79 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_79[0][0]']
 ormalization)

 batch_normalization_82 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_82[0][0]']
 ormalization)

 batch_normalization_83 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_83[0][0]']
 ormalization)

 conv2d_84 (Conv2D)             (None, 8, 8, 192)    245760      ['average_pooling2d_7[0]
[0]']

 batch_normalization_76 (BatchN  (None, 8, 8, 320)   960         ['conv2d_76[0][0]']
 ormalization)

 activation_78 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_78
[0][0]']

 activation_79 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_79
[0][0]']

 activation_82 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_82
[0][0]']

 activation_83 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_83
```

```
[0][0]']

 batch_normalization_84 (BatchN  (None, 8, 8, 192)   576         ['conv2d_84[0][0]']
 ormalization)


 activation_76 (Activation)     (None, 8, 8, 320)    0           ['batch_normalization_76
[0][0]']

 mixed9_0 (Concatenate)         (None, 8, 8, 768)    0           ['activation_78[0][0]',
                                                                  'activation_79[0][0]'
]

 concatenate (Concatenate)      (None, 8, 8, 768)    0           ['activation_82[0][0]',
                                                                  'activation_83[0][0]'
]

 activation_84 (Activation)     (None, 8, 8, 192)    0           ['batch_normalization_84
[0][0]']

 mixed9 (Concatenate)           (None, 8, 8, 2048)   0           ['activation_76[0][0]',
                                                                  'mixed9_0[0][0]',
                                                                  'concatenate[0][0]',
                                                                  'activation_84[0][0]'
]

 conv2d_89 (Conv2D)             (None, 8, 8, 448)    917504      ['mixed9[0][0]']


 batch_normalization_89 (BatchN  (None, 8, 8, 448)   1344        ['conv2d_89[0][0]']
 ormalization)


 activation_89 (Activation)     (None, 8, 8, 448)    0           ['batch_normalization_89
[0][0]']

 conv2d_86 (Conv2D)             (None, 8, 8, 384)    786432      ['mixed9[0][0]']


 conv2d_90 (Conv2D)             (None, 8, 8, 384)    1548288     ['activation_89[0][0]']


 batch_normalization_86 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_86[0][0]']
 ormalization)


 batch_normalization_90 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_90[0][0]']
 ormalization)
```

```
 activation_86 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_86
[0][0]']

 activation_90 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_90
[0][0]']

 conv2d_87 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_86[0][0]']

 conv2d_88 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_86[0][0]']

 conv2d_91 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_90[0][0]']

 conv2d_92 (Conv2D)             (None, 8, 8, 384)    442368      ['activation_90[0][0]']

 average_pooling2d_8 (AveragePo  (None, 8, 8, 2048)  0           ['mixed9[0][0]']
 oling2D)

 conv2d_85 (Conv2D)             (None, 8, 8, 320)    655360      ['mixed9[0][0]']

 batch_normalization_87 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_87[0][0]']
 ormalization)

 batch_normalization_88 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_88[0][0]']
 ormalization)

 batch_normalization_91 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_91[0][0]']
 ormalization)

 batch_normalization_92 (BatchN  (None, 8, 8, 384)   1152        ['conv2d_92[0][0]']
 ormalization)

 conv2d_93 (Conv2D)             (None, 8, 8, 192)    393216      ['average_pooling2d_8[0]
[0]']

 batch_normalization_85 (BatchN  (None, 8, 8, 320)   960         ['conv2d_85[0][0]']
 ormalization)

 activation_87 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_87
[0][0]']
```

```
 activation_88 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_88
[0][0]']

 activation_91 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_91
[0][0]']

 activation_92 (Activation)     (None, 8, 8, 384)    0           ['batch_normalization_92
[0][0]']

 batch_normalization_93 (BatchN  (None, 8, 8, 192)   576         ['conv2d_93[0][0]']

 ormalization)


 activation_85 (Activation)     (None, 8, 8, 320)    0           ['batch_normalization_85
[0][0]']

 mixed9_1 (Concatenate)         (None, 8, 8, 768)    0           ['activation_87[0][0]',

                                                                  'activation_88[0][0]'
]

 concatenate_1 (Concatenate)    (None, 8, 8, 768)    0           ['activation_91[0][0]',

                                                                  'activation_92[0][0]'
]

 activation_93 (Activation)     (None, 8, 8, 192)    0           ['batch_normalization_93
[0][0]']

 mixed10 (Concatenate)          (None, 8, 8, 2048)   0           ['activation_85[0][0]',

                                                                  'mixed9_1[0][0]',

                                                                  'concatenate_1[0][0]'
,
                                                                  'activation_93[0][0]'
]

 avg_pool (GlobalAveragePooling  (None, 2048)        0           ['mixed10[0][0]']

 2D)


 predictions (Dense)            (None, 1000)         2049000     ['avg_pool[0][0]']

==========================================================================================
=========
Total params: 23,851,784
Trainable params: 23,817,352
Non-trainable params: 34,432
__________________________________________________________________________________________
_________
```

This is a prediction model,so the output is typically a softmax-activated vector representing 1000 possible object types. Because we are interested in an encoded representation of the image we are just going to use the second-to-last layer as a source of image encodings. Each image will be encoded as a vector of size 2048.

We will use the following hack: hook up the input into a new Keras model and use the penultimate layer of the existing model as output.

In [21]:

```
new_input = img_model.input
new_output = img_model.layers[-2].output
img_encoder = Model(new_input, new_output) # This is the final Keras image encoder model
we will use.
```

Let's try the encoder.

In [22]:

```
encoded_image = img_encoder.predict(np.array([new_image]))
```

In [23]:

```
encoded_image
```

Out[23]:

```
array([[0.63806605, 0.4887302 , 0.05526248, ..., 0.6425564 , 0.29595238,
        0.4900436 ]], dtype=float32)
```

TODO: We will need to create encodings for all images and store them in one big matrix (one for each dataset, train, dev, test). We can then save the matrices so that we never have to touch the bulky image data again.

To save memory (but slow the process down a little bit) we will read in the images lazily using a generator. We will encounter generators again later when we train the LSTM. If you are unfamiliar with generators, take a look at this page: https://wiki.python.org/moin/Generators

Write the following generator function, which should return one image at a time. `img_list` is a list of image file names (i.e. the train, dev, or test set). The return value should be a numpy array of shape (1,299,299,3).

In [24]:

```
def img_generator(img_list):
    n = 0
    while n<len(img_list):
      image_new = get_image(img_list[n])
      n=n+1
      yield image_new.reshape((1,299,299,3))
```

Now we can encode all images (this takes a few minutes).

In [25]:

```
enc_train = img_encoder.predict_generator(img_generator(train_list), steps=len(train_list
), verbose=1)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.predi
ct_generator` is deprecated and will be removed in a future version. Please use `Model.pr
edict`, which supports generators.
  """Entry point for launching an IPython kernel.
```

```
6000/6000 [==============================] - 173s 29ms/step
```

In [26]:

```
enc_train[11]
```

Out[26]:

```
array([0.26818565, 1.0321662 , 0.58516157, ..., 1.2316744 , 0.17969333,
       0.22405314], dtype=float32)
```

In [27]:

```
enc_dev = img_encoder.predict_generator(img_generator(dev_list), steps=len(dev_list), ver
bose=1)
```

```
   3/1000 [..............................] - ETA: 35s
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.predi
ct_generator` is deprecated and will be removed in a future version. Please use `Model.pr
edict`, which supports generators.
  """Entry point for launching an IPython kernel.
```

```
1000/1000 [==============================] - 29s 29ms/step
```

In [28]:

```
enc_test = img_encoder.predict_generator(img_generator(test_list), steps=len(test_list),
verbose=1)
```

```
   3/1000 [..............................] - ETA: 35s
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.predi
ct_generator` is deprecated and will be removed in a future version. Please use `Model.pr
edict`, which supports generators.
  """Entry point for launching an IPython kernel.
```

```
1000/1000 [==============================] - 29s 29ms/step
```

It's a good idea to save the resulting matrices, so we do not have to run the encoder again.

In [31]:

```
np.save("/content/gdrive/MyDrive/"+my_data_dir+"/outputs/encoded_images_train.npy", enc_t
rain)
np.save("/content/gdrive/MyDrive/"+my_data_dir+"/outputs/encoded_images_dev.npy", enc_dev
)
np.save("/content/gdrive/MyDrive/"+my_data_dir+"/outputs/encoded_images_test.npy", enc_te
st)
```

Part II Text (Caption) Data Preparation (14 pts)

Next, we need to load the image captions and generate training data for the generator model.

Reading image descriptions

TODO: Write the following function that reads the image descriptions from the file `filename` and returns a dictionary in the following format. Take a look at the file `Flickr8k.token.txt` for the format of the input file. The keys of the dictionary should be image filenames. Each value should be a list of 5 captions. Each caption should be a list of tokens.

The captions in the file are already tokenized, so you can just split them at white spaces. You should convert each token to lower case. You should then pad each caption with a START token on the left and an END token on the right.

In [32]:

```
def read_image_descriptions(filename):
    image_descriptions = defaultdict(list)

    f = open(filename)

    for fi in f:
      line_li=fi.split("\t")
      file_name= line_li[0].split("#")[0]
      image_caption=line_li[1]
      caption_li=image_caption.split()
      caption_li.insert(0,"<START>" )
      caption_li.append("<END>")
      image_descriptions[file_name].append(caption_li)
```

```
    return image_descriptions
```

In [33]:

```
descriptions = read_image_descriptions("/content/gdrive/MyDrive/"+my_data_dir+"/Flickr8k.
token.txt")
```

In [34]:

```
print(descriptions[dev_list[0]])
```

```
[['<START>', 'the', 'boy', 'laying', 'face', 'down', 'on', 'a', 'skateboard', 'is', 'bein
g', 'pushed', 'along', 'the', 'ground', 'by', 'another', 'boy', '.', '<END>'], ['<START>'
, 'Two', 'girls', 'play', 'on', 'a', 'skateboard', 'in', 'a', 'courtyard', '.', '<END>'],
['<START>', 'Two', 'people', 'play', 'on', 'a', 'long', 'skateboard', '.', '<END>'], ['<S
TART>', 'Two', 'small', 'children', 'in', 'red', 'shirts', 'playing', 'on', 'a', 'skatebo
ard', '.', '<END>'], ['<START>', 'two', 'young', 'children', 'on', 'a', 'skateboard', 'go
ing', 'across', 'a', 'sidewalk', '<END>']]
```

Running the previous cell should print:

```
[['<START>', 'the', 'boy', 'laying', 'face', 'down', 'on', 'a', 'skateboard', 'is',
'being', 'pushed', 'along', 'the', 'ground', 'by', 'another', 'boy', '.', '<END>'],
['<START>', 'two', 'girls', 'play', 'on', 'a', 'skateboard', 'in', 'a', 'courtyard', '.',
'<END>'], ['<START>', 'two', 'people', 'play', 'on', 'a', 'long', 'skateboard', '.',
'<END>'], ['<START>', 'two', 'small', 'children', 'in', 'red', 'shirts', 'playing', 'on',
'a', 'skateboard', '.', '<END>'], ['<START>', 'two', 'young', 'children', 'on', 'a',
'skateboard', 'going', 'across', 'a', 'sidewalk', '<END>']]
```

Creating Word Indices

Next, we need to create a lookup table from the training data mapping words to integer indices, so we can encode input and output sequences using numeric representations. TODO create the dictionaries id_to_word and word_to_id, which should map tokens to numeric ids and numeric ids to tokens.
Hint: Create a set of tokens in the training data first, then convert the set into a list and sort it. This way if you run the code multiple times, you will always get the same dictionaries.

In [35]:

```
id_to_word = defaultdict()
word_to_id = defaultdict()
words=[]
image_descriptions = read_image_descriptions("/content/gdrive/MyDrive/"+my_data_dir+"/Fli
ckr8k.token.txt")
```

In [36]:

```
for captions in descriptions.values():
    for caption in captions:
      for word in caption:
        words.append(word)
```

In [37]:

```
words_list = list(set(words))
words_list.sort()
n = len(words_list)
for i in range(n):
  id_to_word[i] = words_list[i]
  word_to_id[words_list[i]] = i
```

In [38]:

```
word_to_id['dog'] # should print an integer
```

Out[38]:

Out[38]:

3376

In [39]:

```
id_to_word[1985] # should print a token
```

Out[39]:

'blows'

Note that we do not need an UNK word token because we are generating. The generated text will only contain tokens seen at training time.

Part III Basic Decoder Model (24 pts)

For now, we will just train a model for text generation without conditioning the generator on the image input.

There are different ways to do this and our approach will be slightly different from the generator discussed in class.

The core idea here is that the Keras recurrent layers (including LSTM) create an "unrolled" RNN. Each time-step is represented as a different unit, but the weights for these units are shared. We are going to use the constant MAX_LEN to refer to the maximum length of a sequence, which turns out to be 40 words in this data set (including START and END).

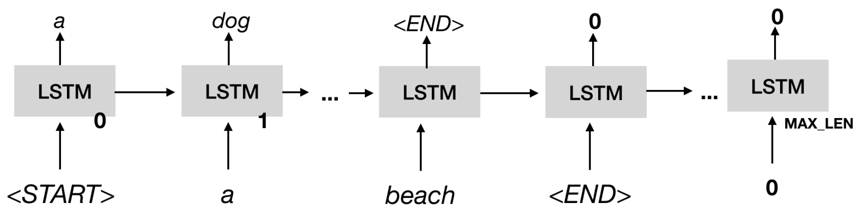
In [40]:

```
max(len(description) for image_id in train_list for description in descriptions[image_id]
)
```

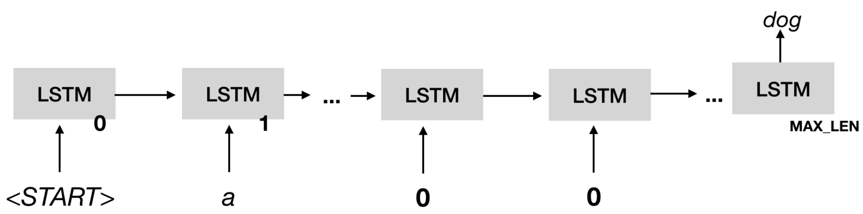
Out[40]:

40

In class, we discussed LSTM generators as transducers that map each word in the input sequence to the next word.



Instead, we will use the model to predict one word at a time, given a partial sequence. For example, given the sequence ["START","a"], the model might predict "dog" as the most likely word. We are basically using the LSTM to encode the input sequence up to this point.



To train the model, we will convert each description into a set of input output pairs as follows. For example, consider the sequence

`['<START>', 'a', 'black', 'dog', '.', '<END>']`

We would train the model using the following input/output pairs

i	input	output
0	[START]	a
1	[START , a]	black
2	[START , a , black]	dog
3	[START , a , black , dog]	END

Here is the model in Keras Keras. Note that we are using a Bidirectional LSTM, which encodes the sequence from both directions and then predicts the output. Also note the `return_sequence=False` parameter, which causes the LSTM to return a single output instead of one output per state.

Note also that we use an embedding layer for the input words. The weights are shared between all units of the unrolled LSTM. We will train these embeddings with the model.

In [41]:

```
MAX_LEN = 40
EMBEDDING_DIM=300
vocab_size = len(word_to_id)

# Text input
text_input = Input(shape=(MAX_LEN,))
embedding = Embedding(vocab_size, EMBEDDING_DIM, input_length=MAX_LEN)(text_input)
x = Bidirectional(LSTM(512, return_sequences=False))(embedding)
pred = Dense(vocab_size, activation='softmax')(x)
model = Model(inputs=[text_input],outputs=pred)
model.compile(loss='categorical_crossentropy', optimizer='RMSprop', metrics=['accuracy']
)

model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 40)]              0         
                                                                 
 embedding (Embedding)       (None, 40, 300)           2889600   
                                                                 
 bidirectional (Bidirectiona  (None, 1024)             3330048   
 l)                                                              
                                                                 
 dense (Dense)               (None, 9632)              9872800   
                                                                 
=================================================================
Total params: 16,092,448
Trainable params: 16,092,448
Non-trainable params: 0
_________________________________________________________________
```

In [42]:

```
print(vocab_size)
```

```
9632
```

The model input is a numpy ndarray (a tensor) of size `(batch_size, MAX_LEN)`. Each row is a vector of size MAX_LEN in which each entry is an integer representing a word (according to the `word_to_id` dictionary). If the input sequence is shorter than MAX_LEN, the remaining entries should be padded with 0.

For each input example, the model returns a softmax activated vector (a probability distribution) over possible output words. The model output is a numpy ndarray of size `(batch_size, vocab_size)`. vocab_size is the number of vocabulary words.

Creating a Generator for the Training Data

TODO:

We could simply create one large numpy ndarray for all the training data. Because we have a lot of training instances (each training sentence will produce up to MAX_LEN input/output pairs, one for each word), it is better to produce the training examples *lazily*, i.e. in batches using a generator (recall the image generator in part I).

Write the function `text_training_generator` below, that takes as a paramater the batch_size and returns an `(input, output)` pair. `input` is a `(batch_size, MAX_LEN)` ndarray of partial input sequences, `output` contains the next words predicted for each partial input sequence, encoded as a `(batch_size, vocab_size)` ndarray.

Each time the next() function is called on the generator instance, it should return a new batch of the *training* data. You can use `train_list` as a list of training images. A batch may contain input/output examples extracted from different descriptions or even from different images.

You can just refer back to the variables you have defined above, including `descriptions`, `train_list`, `vocab_size`, etc.

Hint: To prevent issues with having to reset the generator for each epoch and to make sure the generator can always return exactly `batch_size` input/output pairs in each step, wrap your code into a `while True:` loop. This way, when you reach the end of the training data, you will just continue adding training data from the beginning into the batch.

In [43]:

```
def text_training_generator(batch_size=128):
    ndarray_ip = []
    ndarray_op = []

    while(True):
      for f in train_list:
        for caption in descriptions[f]:
          n = len(caption)
          for i in range(n-1):

            count = 0
            sentence_generator_ip_num = np.zeros(40)
            sentence_generator_ip = caption[:i+1]
            op = caption[i+1]
            op_index = word_to_id[op]

            for word in sentence_generator_ip:
              sentence_generator_ip_num[count]=word_to_id[word]
              count += 1

            ndarray_ip.append(sentence_generator_ip_num)

            op_one_hot_encoded = to_categorical(op_index, vocab_size)
            ndarray_op.append(op_one_hot_encoded)

            if(len(ndarray_ip)==128):
              ndarray_ip = np.asarray(ndarray_ip)
              ndarray_op = np.asarray(ndarray_op)
              reshaped_ip= np.reshape(ndarray_ip,(batch_size, 40))
              reshaped_op= np.reshape(ndarray_op,(batch_size, vocab_size))
              yield (reshaped_ip, reshaped_op)

              ndarray_ip=[]
              ndarray_op=[]
```

Training the Model

We will use the `fit_generator` method of the model to train the model. fit_generator needs to know how

many iterator steps there are per epoch.

Because there are len(train_list) training samples with up to `MAX_LEN` words, an upper bound for the number of total training instances is `len(train_list)*MAX_LEN`. Because the generator returns these in batches, the number of steps is len(train_list) * MAX_LEN // batch_size

In [44]:

```
batch_size = 128
generator = text_training_generator(batch_size)
steps = len(train_list) * MAX_LEN // batch_size
```

In [45]:

```
model.fit_generator(generator, steps_per_epoch=steps, verbose=True, epochs=15)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.fit_g
enerator` is deprecated and will be removed in a future version. Please use `Model.fit`,
which supports generators.
  """Entry point for launching an IPython kernel.
```

```
Epoch 1/15
1875/1875 [==============================] - 201s 105ms/step - loss: 4.2973 - accuracy: 0
.2907
Epoch 2/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.7278 - accuracy: 0
.3544
Epoch 3/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.5509 - accuracy: 0
.3705
Epoch 4/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.4439 - accuracy: 0
.3795
Epoch 5/15
1875/1875 [==============================] - 197s 105ms/step - loss: 3.4050 - accuracy: 0
.3850
Epoch 6/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.3240 - accuracy: 0
.3913
Epoch 7/15
1875/1875 [==============================] - 197s 105ms/step - loss: 3.3204 - accuracy: 0
.3942
Epoch 8/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.3297 - accuracy: 0
.3967
Epoch 9/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.3134 - accuracy: 0
.3979
Epoch 10/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.3486 - accuracy: 0
.3974
Epoch 11/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.3522 - accuracy: 0
.3992
Epoch 12/15
1875/1875 [==============================] - 196s 104ms/step - loss: 4.4811 - accuracy: 0
.3958
Epoch 13/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.4118 - accuracy: 0
.3985
Epoch 14/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.4940 - accuracy: 0
.4013
Epoch 15/15
1875/1875 [==============================] - 196s 105ms/step - loss: 3.4235 - accuracy: 0
.4034
```

Out[45]:

```
<keras.callbacks.History at 0x7fcfcc005910>
```

In [ ]:

In [ ]:

```
model.save('trained_new.hdf5')
```

In [ ]:

```
model.load_weights('trained_new.hdf5')
```

Continue to train the model until you reach an accuracy of at least 40%.

Greedy Decoder

TODO Next, you will write a decoder. The decoder should start with the sequence `["<START>"]`, use the model to predict the most likely word, append the word to the sequence and then continue until `"<END>"` is predicted or the sequence reaches `MAX_LEN` words.

In [ ]:

```
def decoder():
  len = 1
  input = np.zeros(40)
  input[0] = word_to_id["<START>"]

  prediction = "<START>"
  generated_caption = ["<START>"]

  while(len<40 and prediction != "<END>" ):
    output = model.predict(np.array([input]))
    index = np.where(output == np.amax(output))
    input[len]= index[1][0]
    prediction = id_to_word[index[1][0]]
    generated_caption.append (prediction)

    len+=1
  return generated_caption
```

In [ ]:

```
print(decoder())
```

```
['<START>', 'A', 'man', 'in', 'a', 'hat', 'and', 'a', 'hat', 'is', 'standing', 'on', 'a',
'rock', '.', '<END>']
```

This simple decoder will of course always predict the same sequence (and it's not necessarily a good one).

Modify the decoder as follows. Instead of choosing the most likely word in each step, sample the next word from the distribution (i.e. the softmax activated output) returned by the model. Take a look at the np.random.multinomial function to do this.

In [ ]:

```
def sample_decoder():
  len =1
  input = np.zeros(40)
  input[0] = word_to_id["<START>"]
  generated_caption = ["<START>"]
  prediction = "<START>"

  while(len<40 and prediction != "<END>"):

    output = model.predict(np.array([input]))
    output = output.tolist()[0]

    norm_output = []
    for i in output:
      norm_output.append(i/sum(output))

    multi = np.random.multinomial(10, norm_output, size=None)
```

```
    index = np.where(multi == np.amax(multi))

    input[len] = index[0][0]
    prediction = id_to_word[index[0][0]]
    generated_caption.append(prediction)
    len+=1
  return generated_caption
```

You should now be able to see some interesting output that looks a lot like flickr8k image captions -- only that the captions are generated randomly without any image input.

In [ ]:

```
for i in range(10):
    print(sample_decoder())
```

```
['<START>', 'A', 'black', 'and', 'white', 'dog', 'is', 'running', 'in', 'the', 'snow', '.
', '<END>']
['<START>', 'A', 'man', 'in', 'a', 'red', 'coat', 'is', 'rock', 'climbing', '.', '<END>']
['<START>', 'A', 'girl', 'in', 'a', 'pink', 'dress', 'is', 'playing', 'in', 'a', 'park',
'.', '<END>']
['<START>', 'A', 'man', 'in', 'a', 'white', 'shirt', 'and', 'black', 'pants', 'and', 'a',
'jacket', 'and', 'black', 'vest', 'is', 'a', 'sign', '.', '<END>']
['<START>', 'A', 'man', 'in', 'a', 'red', 'shirt', 'and', 'sunglasses', ',', 'with', 'a',
'stick', 'in', 'his', 'mouth', ',', '.', '<END>']
['<START>', 'A', 'man', 'in', 'a', 'black', 'hat', 'and', 'hat', ',', 'and', 'a', 'hat',
',', 'all', 'a', 'hat', ',', 'is', 'are', 'holding', 'a', 'a', 'a', 'a', 'a', 'a', 'a', '
backpack', '.', '<END>']
['<START>', 'A', 'girl', 'in', 'a', 'pink', 'are', 'on', 'a', 'trampoline', '.', '<END>']
['<START>', 'A', 'man', 'in', 'a', 'red', 'hat', 'and', 'black', 'hat', ',', 'holding', '
a', 'camera', '.', '<END>']
['<START>', 'A', 'black', 'and', 'white', 'dog', 'is', 'running', 'in', 'the', 'snow', '.
', '<END>']
['<START>', 'A', 'black', 'and', 'white', 'dog', 'is', 'jumping', 'over', 'a', 'log', 'in
', 'a', 'field', '.', '<END>']
```

Part III - Conditioning on the Image (24 pts)

We will now extend the model to condition the next word not only on the partial sequence, but also on the encoded image.

We will project the 2048-dimensional image encoding to a 300-dimensional hidden layer. We then concatenate this vector with each embedded input word, before applying the LSTM.

Here is what the Keras model looks like:

In [ ]:

```
MAX_LEN = 40
EMBEDDING_DIM=300
IMAGE_ENC_DIM=300

# Image input
img_input = Input(shape=(2048,))
img_enc = Dense(300, activation="relu") (img_input)
images = RepeatVector(MAX_LEN)(img_enc)

# Text input
text_input = Input(shape=(MAX_LEN,))
embedding = Embedding(vocab_size, EMBEDDING_DIM, input_length=MAX_LEN)(text_input)
x = Concatenate()([images,embedding])
y = Bidirectional(LSTM(256, return_sequences=False))(x)
pred = Dense(vocab_size, activation='softmax')(y)
model = Model(inputs=[img_input,text_input],outputs=pred)
model.compile(loss='categorical_crossentropy', optimizer="RMSProp", metrics=['accuracy']
)

model.summary()
```

```
Model: "model_2"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to                     
==================================================================================================
 input_3 (InputLayer)           [(None, 2048)]       0           []                               
                                                                                                  
 dense_1 (Dense)                (None, 300)          614700      ['input_3[0][0]']                
                                                                                                  
 input_4 (InputLayer)           [(None, 40)]         0           []                               
                                                                                                  
 repeat_vector (RepeatVector)   (None, 40, 300)      0           ['dense_1[0][0]']                
                                                                                                  
 embedding_1 (Embedding)        (None, 40, 300)      2889600     ['input_4[0][0]']                
                                                                                                  
 concatenate_2 (Concatenate)    (None, 40, 600)      0           ['repeat_vector[0][0]',          
                                                                  'embedding_1[0][0]']            
                                                                                                  
 bidirectional_1 (Bidirectional  (None, 512)         1755136     ['concatenate_2[0][0]']          
 )                                                                                                
                                                                                                  
 dense_2 (Dense)                (None, 9632)         4941216     ['bidirectional_1[0][0]
']                                                                                                  
                                                                                                  
==================================================================================================
Total params: 10,200,652
Trainable params: 10,200,652
Non-trainable params: 0
__________________________________________________________________________________________________
```

The model now takes two inputs:

1. **a `(batch_size, 2048)` ndarray of image encodings.**
2. **a `(batch_size, MAX_LEN)` ndarray of partial input sequences.**

And one output as before: a `(batch_size, vocab_size)` ndarray of predicted word distributions.

TODO: Modify the training data generator to include the image with each input/output pair. Your generator needs to return an object of the following format: `([image_inputs, text_inputs], next_words)`. Where each element is an ndarray of the type described above.

You need to find the image encoding that belongs to each image. You can use the fact that the index of the image in `train_list` is the same as the index in enc_train and enc_dev.

If you have previously saved the image encodings, you can load them from disk:

In [ ]:

```
enc_train = np.load("/content/gdrive/My Drive/"+my_data_dir+"/outputs/encoded_images_trai
n.npy")
enc_dev = np.load("/content/gdrive/My Drive/"+my_data_dir+"/outputs/encoded_images_dev.np
y")
```

In [ ]:

```
def training_generator(batch_size=128):
  ndarray_ip_img = []
  ndarray_ip = []
  ndarray_op = []

  while(True):
    for f in range(len(train_list)):
      fname = train_list[f]
      encoding = enc_train[f]
      for caption in descriptions[fname]:
        n = len(caption)
        for i in range(n-1):
          count = 0
          sentence_generator_ip_num = np.zeros(40)
          sentence_generator_ip = caption[:i+1]
          op = caption[i+1]
          op_index = word_to_id[op]
          op_numeric_oh_encoded = to_categorical(op_index, vocab_size)


          for word in sentence_generator_ip:
              sentence_generator_ip_num[count]=word_to_id[word]
              count += 1

          ndarray_ip_img.append(encoding)
          ndarray_ip.append(sentence_generator_ip_num)
          ndarray_op.append(op_numeric_oh_encoded)

          if(len(ndarray_ip)==128):
            ndarray_ip_img = np.asarray(ndarray_ip_img)
            ndarray_ip = np.asarray(ndarray_ip)
            ndarray_op = np.asarray(ndarray_op)
            reshaped_ip_img = np.reshape(ndarray_ip_img, (batch_size,2048))
            reshaped_ip = np.reshape(ndarray_ip,(batch_size, 40))
            reshaped_op = np.reshape(ndarray_op,(batch_size, vocab_size))
            yield ([reshaped_ip_img, reshaped_ip], reshaped_op)

            ndarray_ip = []
            ndarray_op = []
            ndarray_ip_img = []
```

You should now be able to train the model as before:

In [ ]:

```
batch_size = 128
generator = training_generator(batch_size)
steps = len(train_list) * MAX_LEN // batch_size
```

In [ ]:

```
model.fit_generator(generator, steps_per_epoch=steps, verbose=True, epochs=20)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.fit_g
enerator` is deprecated and will be removed in a future version. Please use `Model.fit`,
which supports generators.
  """Entry point for launching an IPython kernel.
```

```
Epoch 1/20
1875/1875 [==============================] - 135s 70ms/step - loss: 4.4888 - accuracy: 0.
2590
Epoch 2/20
1875/1875 [==============================] - 131s 70ms/step - loss: 3.6950 - accuracy: 0.
3577
```

```
Epoch 3/20
1875/1875 [==============================] - 131s 70ms/step - loss: 3.4934 - accuracy: 0.
3795
Epoch 4/20
1875/1875 [==============================] - 131s 70ms/step - loss: 3.3746 - accuracy: 0.
3901
Epoch 5/20
1875/1875 [==============================] - 131s 70ms/step - loss: 3.3138 - accuracy: 0.
3969
Epoch 6/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2688 - accuracy: 0.
4009
Epoch 7/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2387 - accuracy: 0.
4035
Epoch 8/20
1875/1875 [==============================] - 132s 71ms/step - loss: 3.2348 - accuracy: 0.
4084
Epoch 9/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2721 - accuracy: 0.
4080
Epoch 10/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.3050 - accuracy: 0.
4069
Epoch 11/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2920 - accuracy: 0.
4094
Epoch 12/20
1875/1875 [==============================] - 131s 70ms/step - loss: 3.2556 - accuracy: 0.
4129
Epoch 13/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2406 - accuracy: 0.
4147
Epoch 14/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2200 - accuracy: 0.
4191
Epoch 15/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2393 - accuracy: 0.
4169
Epoch 16/20
1875/1875 [==============================] - 131s 70ms/step - loss: 3.2442 - accuracy: 0.
4175
Epoch 17/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2427 - accuracy: 0.
4218
Epoch 18/20
1875/1875 [==============================] - 132s 70ms/step - loss: 3.2054 - accuracy: 0.
4245
Epoch 19/20
1875/1875 [==============================] - 132s 71ms/step - loss: 3.2278 - accuracy: 0.
4239
Epoch 20/20
1875/1875 [==============================] - 132s 71ms/step - loss: 3.2454 - accuracy: 0.
4264
```

Out[ ]:

```
<keras.callbacks.History at 0x7f27768c39d0>
```

Again, continue to train the model until you hit an accuracy of about 40%. This may take a while. I strongly encourage you to experiment with cloud GPUs using the GCP voucher for the class.

You can save your model weights to disk and continue at a later time.

In [ ]:

```
model.save_weights("/content/gdrive/My Drive/"+my_data_dir+"/outputs/model_new.h5")
```

to load the model:

In [ ]:

```
model.load_weights("/content/gdrive/My Drive/"+my_data_dir+"/outputs/model_new.h5")
```

TODO: Now we are ready to actually generate image captions using the trained model. Modify the simple greedy decoder you wrote for the text-only generator, so that it takes an encoded image (a vector of length 2048) as input, and returns a sequence.

In [ ]:

```
def image_decoder(enc_image):
  len = 1
  input = np.zeros(40)
  input[0] = word_to_id["<START>"]
  prediction = "<START>"
  generated_caption = ["<START>"]

  while(len<40 and prediction != "<END>" ):

    output = model.predict([np.array([enc_image]),np.array([input])])
    index = np.where(output == np.amax(output))
    input[len]= index[1][0]
    prediction = id_to_word[index[1][0]]
    generated_caption.append (prediction)

    len+=1
  return generated_caption
```

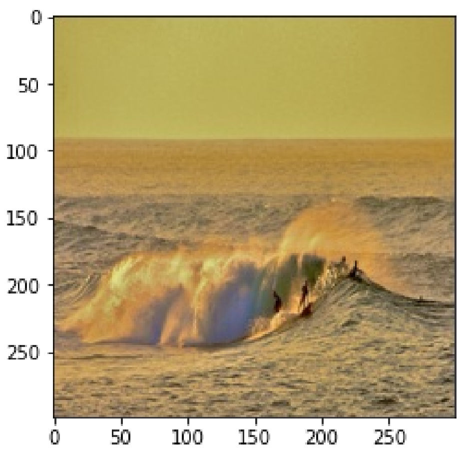
As a sanity check, you should now be able to reproduce (approximately) captions for the training images.

In [ ]:

```
plt.imshow(get_image(train_list[80]))
image_decoder(enc_train[80])
```

Out[ ]:

```
['<START>',
 'A',
 'man',
 'in',
 'a',
 'black',
 'wetsuit',
 'is',
 'in',
 'the',
 'water',
 '.',
 '<END>']
```



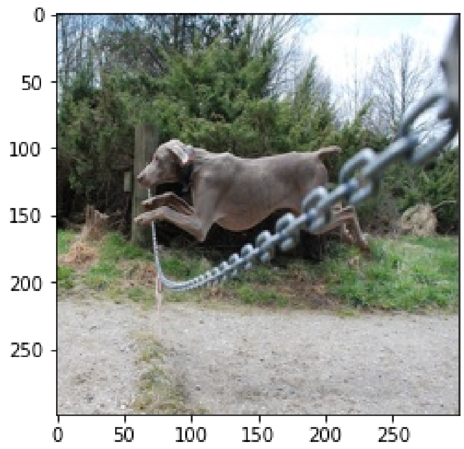
You should also be able to apply the model to dev images and get reasonable captions:

In [ ]:

```
plt.imshow(get_image(dev_list[97]))
image_decoder(enc_dev[97])
```

Out[ ]:

```
['<START>', 'A', 'dog', 'is', 'running', 'on', 'the', 'grass', '.', '<END>']
```



For this assignment we will not perform a formal evaluation.

Feel free to experiment with the parameters of the model or continue training the model. At some point, the model will overfit and will no longer produce good descriptions for the dev images.

Part IV - Beam Search Decoder (24 pts)

TODO Modify the simple greedy decoder for the caption generator to use beam search. Instead of always selecting the most probable word, use a *beam*, which contains the n highest-scoring sequences so far and their total probability (i.e. the product of all word probabilities). I recommend that you use a list of `(probability, sequence)` tuples. After each time-step, prune the list to include only the n most probable sequences.

Then, for each sequence, compute the n most likely successor words. Append the word to produce n new sequences and compute their score. This way, you create a new list of n*n candidates.

Prune this list to the best n as before and continue until `MAX_LEN` words have been generated.

Note that you cannot use the occurence of the "<END>" tag to terminate generation, because the tag may occur in different positions for different entries in the beam.

Once `MAX_LEN` has been reached, return the most likely sequence out of the current n.

In [ ]:

```
def img_beam_decoder(n, image_enc):

    input = [word_to_id["<START>"]]
    array = [[input, 0.0]]

    while len(array[0][0]) < 40:

        temp = []
        for seq in array:

            captions_num = pad_sequences([seq[0]], maxlen=40, padding ='post')
            encoded_image_np = np.asarray([image_enc]).reshape((1,2048))
            captions_num_np = np.asarray([captions_num]).reshape((1,40))
            probability = model.predict([encoded_image_np, captions_num_np])
            prob_seq = np.argsort(probability[0])[-n:]

            for i in prob_seq:
                p = seq[1]
```

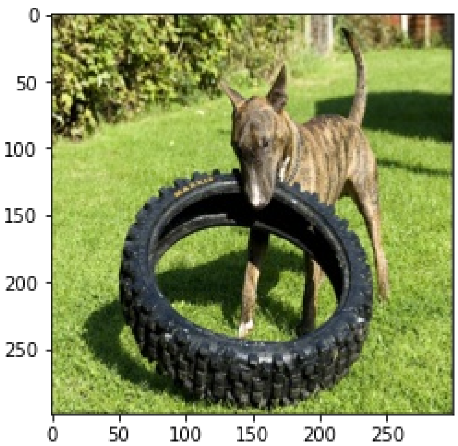
```
                p += probability[0][i]
                next_caption = seq[0][:]
                next_caption.append(i)
                temp.append([next_caption, p])

        array = temp
        array = sorted(array, reverse=False, key= lambda l: l[1])
        array = array[-n:]

    array = array[-1][0]

    temporary_caption=[]
    for i in array:
      temporary_caption.append(id_to_word[i])

    final_caption = []
    for i in temporary_caption:
        if i != "<END>":
            final_caption.append(i)
        else:
            break

    final_caption.append("<END>")
    return final_caption
```

In [ ]:

```
plt.imshow(get_image(dev_list[80]))
img_beam_decoder(3, enc_dev[80])
```

Out[ ]:

```
['<START>', 'A', 'dog', 'jumps', 'over', 'a', 'hurdle', '.', '<END>']
```

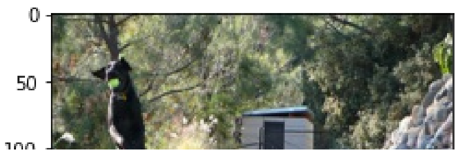


TODO Finally, before you submit this assignment, please show 5 development images, each with 1) their greedy output, 2) beam search at n=3 3) beam search at n=5.

In [ ]:

```
plt.imshow(get_image(dev_list[3]))
print("Greedy Output: ", image_decoder(enc_dev[3]))
print("Beam Search at n=3: ", img_beam_decoder(3, enc_dev[3]))
print("Beam Search at n=5:", img_beam_decoder(5, enc_dev[3]))
```

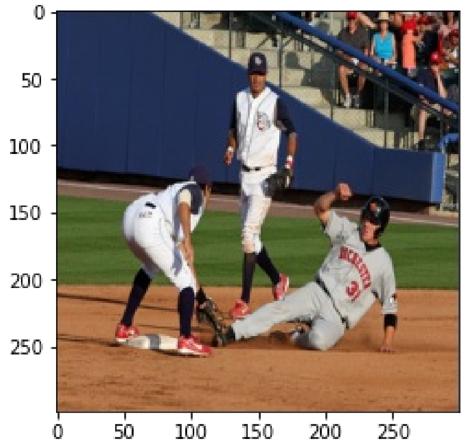
```
Greedy Output:  ['<START>', 'A', 'dog', 'is', 'running', 'in', 'the', 'grass', '.', '<END
>']
Beam Search at n=3:  ['<START>', 'A', 'dog', 'and', 'a', 'dog', 'are', 'in', 'the', 'gras
s', '.', '<END>']
Beam Search at n=5: ['<START>', 'A', 'brown', 'dog', 'and', 'a', 'black', 'and', 'white',
'dog', 'are', 'running', 'in', 'the', 'grass', '.', '<END>']
```



In [ ]:

```
plt.imshow(get_image(dev_list[100]))
print("Greedy Output: ", image_decoder(enc_dev[100]))
print("Beam Search at n=3: ", img_beam_decoder(3, enc_dev[100]))
print("Beam Search at n=5:", img_beam_decoder(5, enc_dev[100]))
```

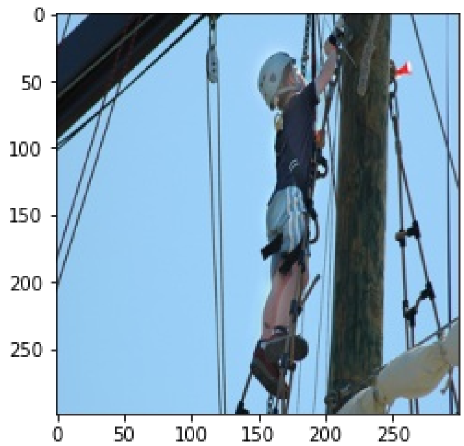
```
Greedy Output:  ['<START>', 'A', 'man', 'in', 'a', 'blue', 'shirt', 'and', 'a', 'white',
'shirt', 'is', 'in', 'a', 'field', '.', '<END>']
Beam Search at n=3:  ['<START>', 'A', 'man', 'in', 'a', 'blue', 'shirt', 'and', 'a', 'man
', 'in', 'a', 'blue', 'shirt', '.', '<END>']
Beam Search at n=5: ['<START>', 'A', 'man', 'in', 'a', 'blue', 'shirt', 'and', 'a', 'man'
, 'in', 'a', 'blue', 'shirt', '.', '<END>']
```



In [ ]:

```
plt.imshow(get_image(dev_list[57]))
print("Greedy Output: ", image_decoder(enc_dev[57]))
print("Beam Search at n=3: ", img_beam_decoder(3, enc_dev[57]))
print("Beam Search at n=5:", img_beam_decoder(5, enc_dev[57]))
```

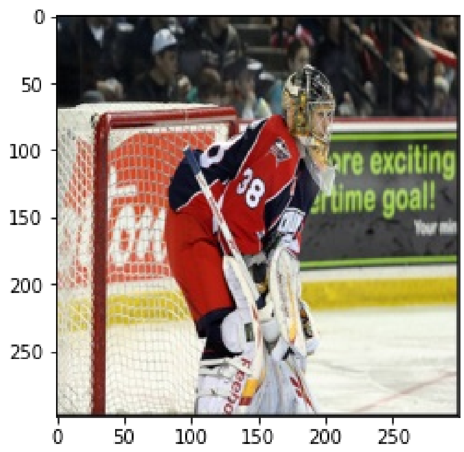
```
Greedy Output:  ['<START>', 'A', 'man', 'in', 'a', 'blue', 'shirt', 'and', 'a', 'white',
'shirt', 'is', 'in', 'a', 'field', '.', '<END>']
Beam Search at n=3:  ['<START>', 'A', 'boy', 'in', 'a', 'blue', 'shirt', 'is', 'in', 'the
', 'air', '.', '<END>']
Beam Search at n=5: ['<START>', 'A', 'boy', 'in', 'a', 'blue', 'shirt', 'is', 'in', 'the'
, 'air', '.', '<END>']
```



In [ ]:

```
plt.imshow(get_image(dev_list[200]))
print("Greedy Output: ", image_decoder(enc_dev[200]))
print("Beam Search at n=3: ", img_beam_decoder(3, enc_dev[200]))
print("Beam Search at n=5:", img_beam_decoder(5, enc_dev[200]))
```

```
Greedy Output:  ['<START>', 'A', 'man', 'in', 'a', 'white', 'shirt', 'and', 'white', 'sho
rts', 'is', 'in', 'a', 'field', '.', '<END>']
Beam Search at n=3:  ['<START>', 'A', 'group', 'of', 'football', 'players', 'in', 'a', 'f
ield', '.', '<END>']
Beam Search at n=5: ['<START>', 'A', 'football', 'player', 'in', 'a', 'white', 'uniform',
'.', '<END>']
```



In [ ]:

```
plt.imshow(get_image(dev_list[60]))
print("Greedy Output: ", image_decoder(enc_dev[60]))
print("Beam Search at n=3: ", img_beam_decoder(3, enc_dev[60]))
print("Beam Search at n=5:", img_beam_decoder(5, enc_dev[60]))
```

```
Greedy Output:  ['<START>', 'A', 'dog', 'is', 'running', 'in', 'the', 'water', '.', '<END
>']
Beam Search at n=3:  ['<START>', 'A', 'brown', 'dog', 'and', 'a', 'black', 'and', 'white'
, 'dog', 'are', 'in', 'the', 'water', '.', '<END>']
Beam Search at n=5: ['<START>', 'A', 'brown', 'dog', 'and', 'a', 'black', 'and', 'white',
'dog', 'are', 'in', 'the', 'water', '.', '<END>']
```