Introduction to Deep Learning, Harvard Summer School 2021

Abstract

Image captioning refers to generating a textual description of a given image. It is a challenging Deep Learning problem which combines both Computer Vision and Natural Language Processing techniques. Image captioning has applications in various domains, such as helping visually impaired people better understand the content of images, usage in self-driving cars, virtual assistants, image indexing, social media, etc. Successful captioning requires not only recognizing the objects in the image, but also recognizing the interactions between them. It is also important to produce sentences which are valid both syntactically and semantically.

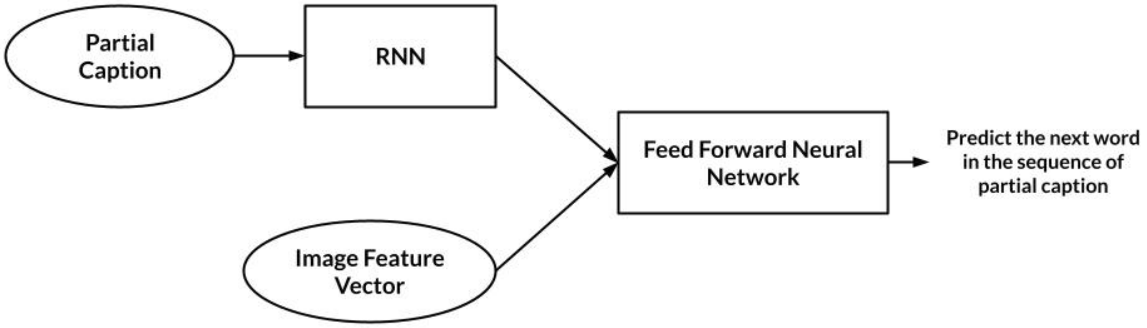
In this project, using Flickr8k dataset, we develop a model for generating english captions for given images.We implement an Encoder/Decoder architecture, where we encode both images and text captions and feed the decoder with the encoded forms. We use a CNN architecture to encode images and the RNN/LSTM as the 'language model' to encode the text sequences of varying length. We therefore create a merge architecture (more information on types of architectures for image captioning can be found here) in order not to introduce the image data in the RNN/LSTM and be able to train the part of the neural network that handles images and the part that handles language separately, using images and sentences from separate training sets.

We manage to produce valid results, especially for images originated from the dataset, but also for completely unknown ones. However, we observe that in many cases our models produce mistaken captions; mistakes mostly refer to incorrectly describing details of the image.

1. Our Approach

There are several known approaches to the problem, such as [Andrej Karpathy's architecture](), [Google's architecture]() and [Microsoft's architecture]().

1.1. Model Architecture

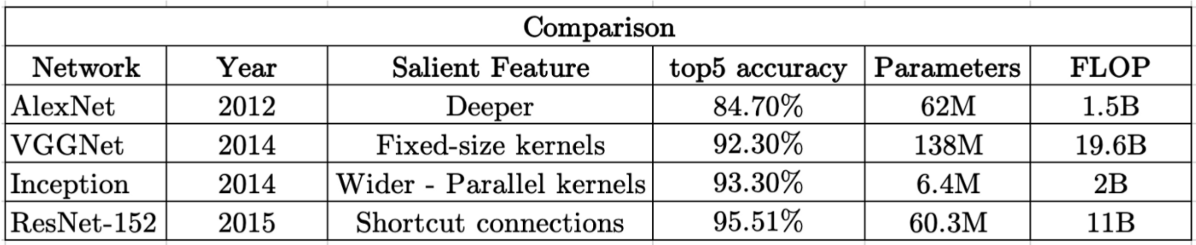


Partial Caption
RNN
Feed Forward Neural Network
Predict the next word in the sequence of partial caption
Image Feature Vector

Our model's architecture.

Here, we implement an Encoder/Decoder model. We encode both images and text captions and feed the decoder with the encoded forms. We use a CNN architecture to encode images and the RNN/LSTM as the 'language model' to encode the text sequences of varying length. We therefore create a merge architecture (more information on types of architectures for image captioning can be found here) in order not to introduce the image data in the RNN/LSTM and be able to train the part of the neural network that handles images and the part that handles language separately, using images and sentences from separate training sets.

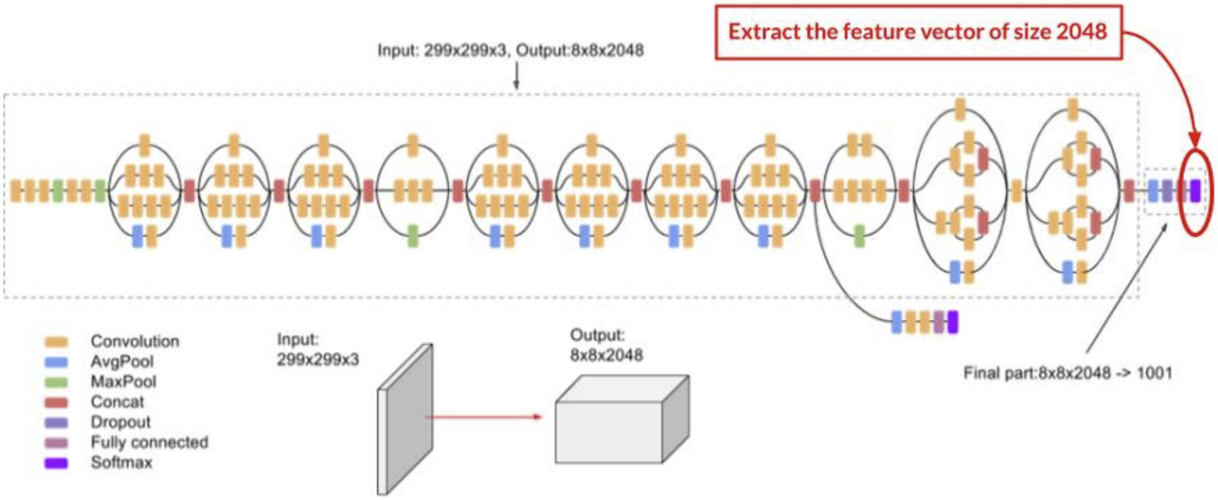
1.2. Encoding Images



Comparison					
Network	Year	Salient Feature	top5 accuracy	Parameters	FLOP
AlexNet	2012	Deeper	84.70%	62M	1.5B
VGGNet	2014	Fixed-size kernels	92.30%	138M	19.6B
Inception	2014	Wider - Parallel kernels	93.30%	6.4M	2B
ResNet-152	2015	Shortcut connections	95.51%	60.3M	11B

Comparison between AlexNet, VGGNet, ResNet and Inception models.

Encoding images means to convert every image into a fixed sized feature vector which can then be fed as input to the neural network. To extract image features we use *transfer learning*. We can choose among different known pre-trained models, such as VGG-16, InceptionV3, ResNet, etc. In particular, we use the pre-trained weights of the InceptionV3 model, which is faster to train as it has the least number of training parameters in comparison to the others and also outperforms them. The InceptionV3 model was trained on the Imagenet dataset to perform image classification on 1000 different classes of images. For the scope of our project we only extract the feature vector of size 2048 from the last layer of the model, before the softmax layer which is used for the classification.


Extract the feature vector of size 2048
Input: 299x299x3, Output:8x8x2048
Convolution
AvgPool
MaxPool
Concat
Dropout
Fully connected
Softmax
Input:
299x299x3
Output:
8x8x2048
Final part:8x8x2048 -> 1001

InceptionV3 architecture.

1.3. Encoding Text Captions

To encode our text sequence we will map every word to a 200-dimensional vector using Glove embeddings. Word vectors map words to a vector space, where similar words are clustered together and different words are separated. Glove does not just rely on the local context of words but it can derive semantic relationships between words.

2. Datasets

Flickr8k (containing 8k images), Flickr30k (containing 30k images) and MS COCO (containing 180k images) are widely used datasets for image captioning. In our approach, we use Flickr8k dataset, where each image is associated with five different captions. By associating each image with multiple, independently produced sentences, the dataset captures some of the linguistic variety that can be used to describe the same image. Flickr8k is a good dataset to begin with image captioning, as it can be easily trained and achieve good results.

Flickr8k has the following structure:

- Flick8k/
  - Flicker8k_Dataset/ → contains the 8000 images
  - Flickr8k_text/
    - Flickr8k.token.txt → contains the image id along with the 5 captions
    - Flickr_8k.trainImages.txt → contains the 6000 training image id's
    - Flickr_8k.testImages.txt → contains the 1000 test image id's
    - Flickr_8k.devImages.txt → contains the 1000 dev image id's

1. *A child in a pink dress is climbing up a set of stairs in an entry way.*
2. *A girl going into a wooden building.*
3. *A little girl climbing into a wooden playhouse.*
4. *A little girl climbing the stairs to her playhouse.*
5. *A little girl in a pink dress going into a wooden cabin.*

A sample from Flickr8k dataset. Each image is associated with five captions.

3. Installation and Configuration

For our experiments we used Google Colaboratory and Keras library. The models were trained using GPUs provided by Google Colaboratory. No special configuration is needed, except for having a Google account. At first, we downloaded the dataset to our Google Drive using the following commands in the Colab notebook:

```
!wget -P "drive/My Drive/IntroDL/Final Project" https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_Dataset.zip

!wget -P "drive/My Drive/IntroDL/Final Project" https://github.com/jbrownlee/Datasets/releases/download/Flickr8k/Flickr8k_text.zip

!mkdir "drive/My Drive/IntroDL/Final Project/Flickr8k_Dataset"
!unzip "drive/My Drive/IntroDL/Final Project/Flickr8k_Dataset.zip" -d "drive/My Drive/IntroDL/Final Project/Flickr8k_Dataset"

!mkdir "drive/My Drive/IntroDL/Final Project/Flickr8k_text"
!unzip -qq "drive/My Drive/IntroDL/Final Project/Flickr8k_text.zip" -d "drive/My Drive/IntroDL/Final Project/Flickr8k_text"
```

We also downloaded Glove embeddings to our Drive using the following commands in the Colab notebook:

```
!wget -P "drive/My Drive/IntroDL/Final Project"
http://nlp.stanford.edu/data/glove.6B.zip

!unzip -qq '/content/drive/MyDrive/IntroDL/Final Project/glove.6B.zip' -d
'/content/drive/MyDrive/IntroDL/Final Project/GloVe'
```

4. Preprocessing

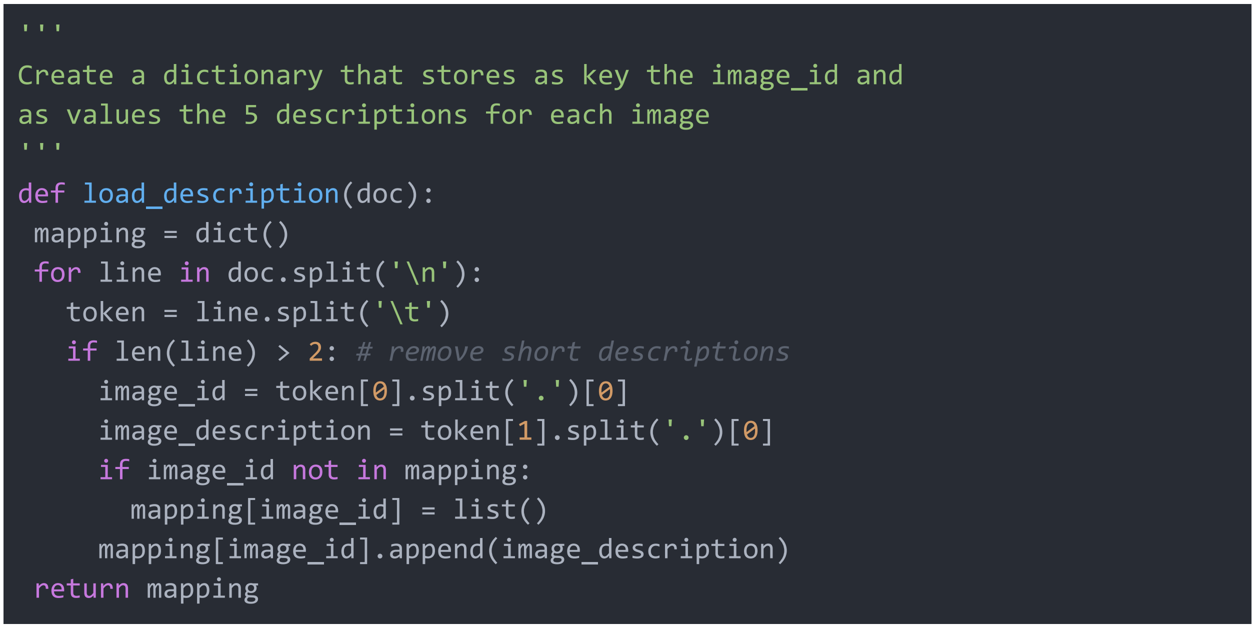
4.1. Captions Preprocessing

If we print a part of our corpus from *Flickr8k.token.txt*, we observe that each line has the following format:

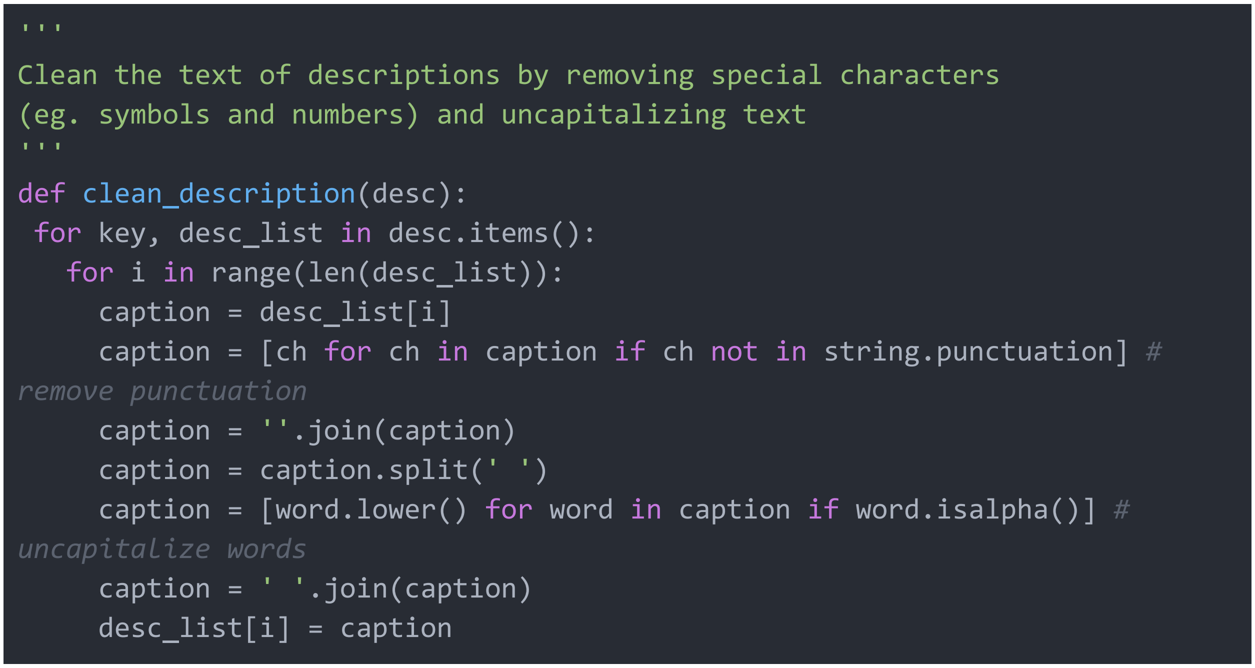
$$\boxed{< image\ id > \#i \quad < caption >, \quad where\ i \in [0,\ 4]}$$

```
                    **** Print a part of the corpus: ****

1000268201_693b08cb0e.jpg#0   A child in a pink dress is climbing up a set
of stairs in an entry way.

1000268201_693b08cb0e.jpg#1   A girl going into a wooden building.

1000268201_693b08cb0e.jpg#2   A little girl climbing into a wooden
playhouse.

1000268201_693b08cb0e.jpg#3   A little girl climbing the stairs to her
playhouse.

1000268201_693b08cb0e.jpg#4   A little girl in a pink dress going into a
wooden cabin.
```

Firstly, we need to create a dictionary that stores as key the image_id and as values the five descriptions for each image. Then, clean the text of descriptions by removing special characters (eg. symbols and numbers) and uncapitalizing text. And also, we create a vocabulary of all unique words existing in our corpus and find that we have 8774 words in our vocabulary.



```
'''
Create a dictionary that stores as key the image_id and
as values the 5 descriptions for each image
'''
def load_description(doc):
 mapping = dict()
 for line in doc.split('\n'):
   token = line.split('\t')
   if len(line) > 2: # remove short descriptions
     image_id = token[0].split('.')[0]
     image_description = token[1].split('.')[0]
     if image_id not in mapping:
       mapping[image_id] = list()
     mapping[image_id].append(image_description)
 return mapping
```

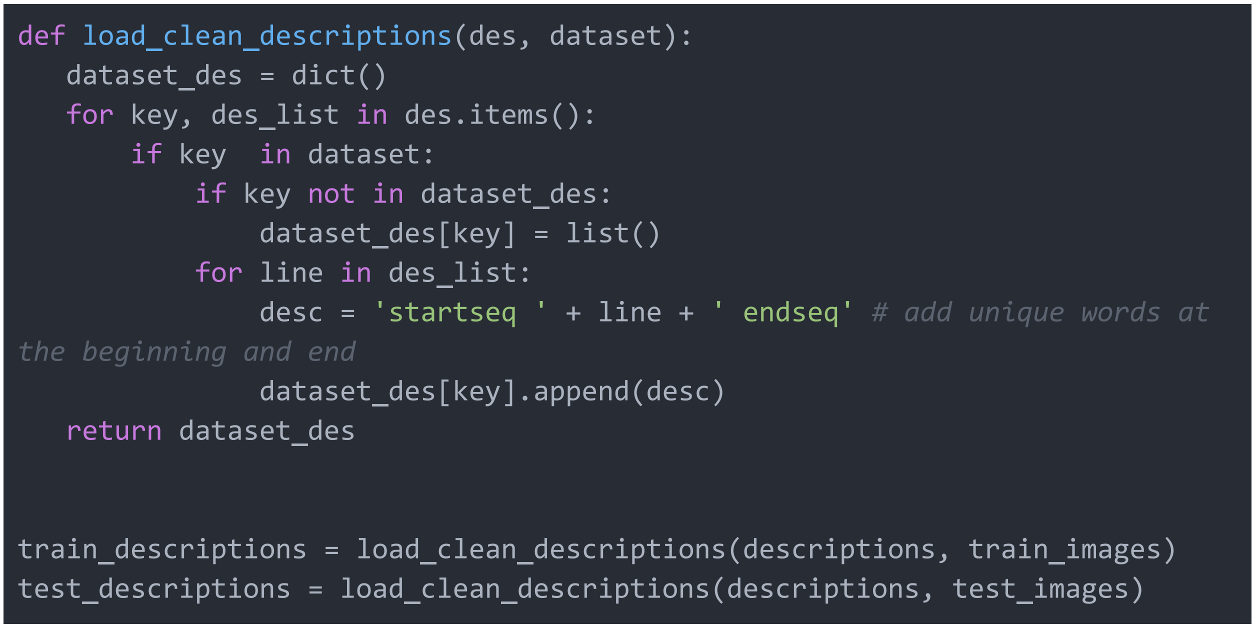


```
'''
Clean the text of descriptions by removing special characters
(eg. symbols and numbers) and uncapitalizing text
'''
def clean_description(desc):
 for key, desc_list in desc.items():
   for i in range(len(desc_list)):
     caption = desc_list[i]
     caption = [ch for ch in caption if ch not in string.punctuation] #
remove punctuation
     caption = ''.join(caption)
     caption = caption.split(' ')
     caption = [word.lower() for word in caption if word.isalpha()] #
uncapitalize words
     caption = ' '.join(caption)
     desc_list[i] = caption
```

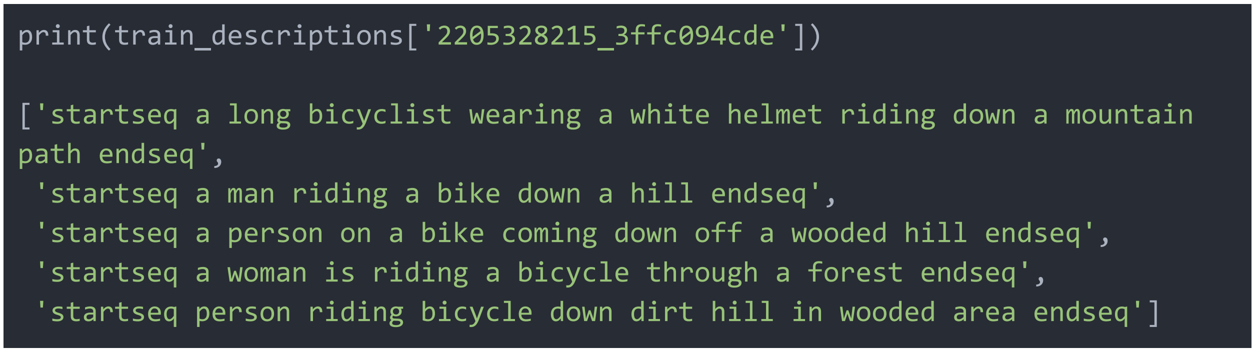
```
'''
Create a vocabulary of all unique words existing in our corpus
'''
def create_vocabulary(desc):
 vocabulary = set()
 for key in desc.keys():
   for line in desc[key]:
     vocabulary.update(line.split())
 return vocabulary
```

The next step is to create a list of ids for all train and test images and then create an empty dictionary and map the images to their descriptions using image id as key and a list of descriptions as its value. However, we add two tokens (***'startseq'***, ***'endseq'***) in every description (the utility of these tokens is explained later in this report). The dictionaries for train and test images are called *train_descriptions* and *test_descriptions* respectively. An instance of the dictionaries can be seen below:

```
'''
Create a list of ids of all train and test images and
then create an empty dictionary and map the images
to their descriptions using image id as key and
a list of descriptions as its value.
'''

train_images = open(train_images_path, 'r', encoding =
'utf-8').read().split("\n")
train_images = [image.split('.')[0] for image in train_images] # throw the
'.jpg' suffix from each image id
train_images = train_images[:-1] # throw the last element which is an empty
string

test_images = open(test_images_path, 'r', encoding =
'utf-8').read().split("\n")
test_images = [image.split('.')[0] for image in test_images] # throw the
'.jpg' suffix from each image id
test_images = test_images[:-1] # throw the last element which is an empty
string

print('There are {} train images and {} test
images'.format(len(train_images), len(test_images)))
```

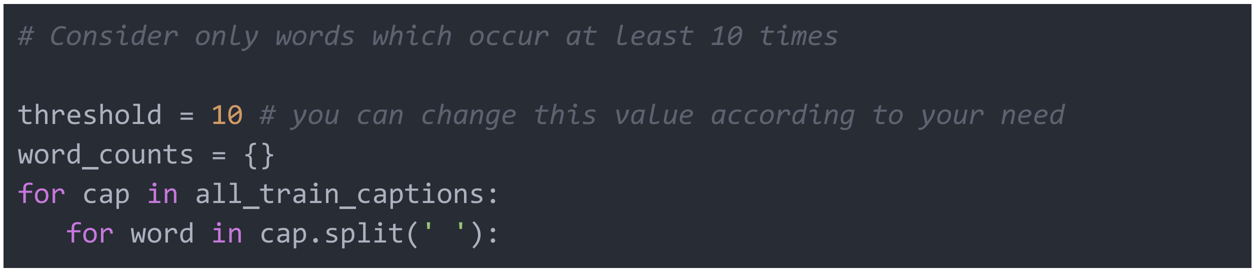


```
def load_clean_descriptions(des, dataset):
   dataset_des = dict()
   for key, des_list in des.items():
       if key  in dataset:
           if key not in dataset_des:
               dataset_des[key] = list()
           for line in des_list:
               desc = 'startseq ' + line + ' endseq' # add unique words at
the beginning and end
               dataset_des[key].append(desc)
   return dataset_des


train_descriptions = load_clean_descriptions(descriptions, train_images)
test_descriptions = load_clean_descriptions(descriptions, test_images)
```



```
print(train_descriptions['2205328215_3ffc094cde'])

['startseq a long bicyclist wearing a white helmet riding down a mountain
path endseq',
 'startseq a man riding a bike down a hill endseq',
 'startseq a person on a bike coming down off a wooded hill endseq',
 'startseq a woman is riding a bicycle through a forest endseq',
 'startseq person riding bicycle down dirt hill in wooded area endseq']
```

Previously, we created a vocabulary of 8774 words. However, we notice that many of these words will appear only a few times in the corpus. For training our model we want to throw these words from our vocabulary and keep only those words which appear at least 10 times in the entire corpus. In this way, the model becomes more robust to outliers and makes less mistaken predictions. Indeed, after considering only words which occur at least 10 times, we have a vocabulary with 1653 words.



```
# Consider only words which occur at least 10 times

threshold = 10 # you can change this value according to your need
word_counts = {}
for cap in all_train_captions:
   for word in cap.split(' '):
       word_counts[word] = word_counts.get(word, 0) + 1
 voc = [word for word in word_counts if word_counts[word] >= threshold]

print('Vocabulary = %d' % (len(voc)))
```

In order to use Glove embeddings, we first need to represent every unique word in the vocabulary by an integer. For that, we create two dictionaries (***wordtoix*** and ***ixtoword***) to map words to an index and vice versa. For all the 1654 unique words in our vocabulary, we create an embedding matrix which will be loaded into the model before training. Each one of the words in our 38-word long caption will map to a 200-dimension vector using Glove.

```
# word mapping to integers
ixtoword = {}
wordtoix = {}
 ix = 1
for word in voc:
   wordtoix[word] = ix
   ixtoword[ix] = word
   ix += 1

vocab_size = len(ixtoword) + 1
```

We also need to find out what is the *max_length* of a caption in our corpus, since our model needs to know in advance where to stop creating words for a caption.

```
# find the maximum length of a description in a dataset
max_length = max(len(des.split()) for des in all_train_captions)
print(max_length)
```

Finally, we extract embeddings matrix of size (1654, 200).

```
embeddings_index = {}
with open(glove_path + 'glove.6B.200d.txt') as f:
   for line in f:
       word, coefs = line.split(maxsplit=1)
       coefs = np.fromstring(coefs, "f", sep=" ")
       embeddings_index[word] = coefs

print("Found {} word vectors.".format(len(embeddings_index)))
```

```
embedding_dim = 200
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, i in wordtoix.items():
   emb_vec = embeddings_index.get(word)
   if emb_vec is not None:
       embedding_matrix[i] = emb_vec
print(embedding_matrix.shape)
```

4.2. Image Preprocessing

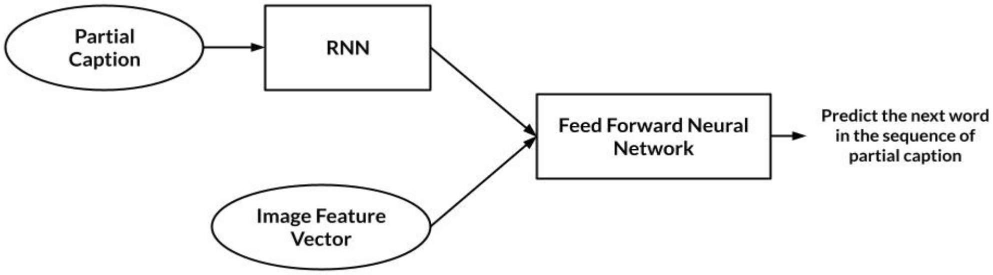
As we described in Section 2.2, we use the InceptionV3 model to extract feature vectors from images. To do so, we need to reshape our images to 299x299 and then feed them to the model to extract feature vectors.

***Notice**: This process using Colab's GPU lasts almost an hour. So, we propose to save the encoded images in a pickle file and reload them when later fine-tuning the model.*

```
def preprocess_img(img_path):
   # inception v3 accept images of size 299 * 299 * 3
   img = load_img(img_path, target_size = (299, 299))
   x = img_to_array(img)
   x = np.expand_dims(x, axis = 0) # Add one more dimension
   x = preprocess_input(x)
   return x

 def encode(image):
   image = preprocess_img(images_path + image + '.jpg')
   vec = model.predict(image)
   vec = np.reshape(vec, (vec.shape[1]))
   return vec

base_model = InceptionV3(weights = 'imagenet')
model = Model(base_model.input, base_model.layers[-2].output)

# Run the encode function on all train and test images and store the
feature vectors in a list
encoding_train = {}
for img in train_images:
 encoding_train[img] = encode(img)

encoding_test = {}
for img in test_images:
   encoding_test[img] = encode(img)

# save the file
pickle.dump(encoding_train, open("drive/My Drive/IntroDL/Final
Project/encoding_train.pkl","wb"))
pickle.dump(encoding_test, open("drive/My Drive/IntroDL/Final
Project/encoding_test.pkl","wb"))
```

5. Model Definition and Model Training

5.1. The Idea Behind the Model


Partial Caption
RNN
Feed Forward Neural Network
Predict the next word in the sequence of partial caption
Image Feature Vector

Our model's architecture.

We repeat here the figure with the model's architecture. As we said in Section 2, we create a merge model where we combine the image feature vector and the partial caption as input and expect the next word in the caption as output. Therefore our model has three main stages:

1. Processing the sequence from the text
2. Extracting the feature vector from the image
3. Decoding the output using softmax by concatenating the above two layers

Having this in mind, we need to frame the image captioning problem as a *supervised learning* problem. For that, we need to understand what's the desired input and output of the model. The input will be the feature vector of the image and a partial caption. The desired output will be the next word in the caption. Each caption has now the following form:

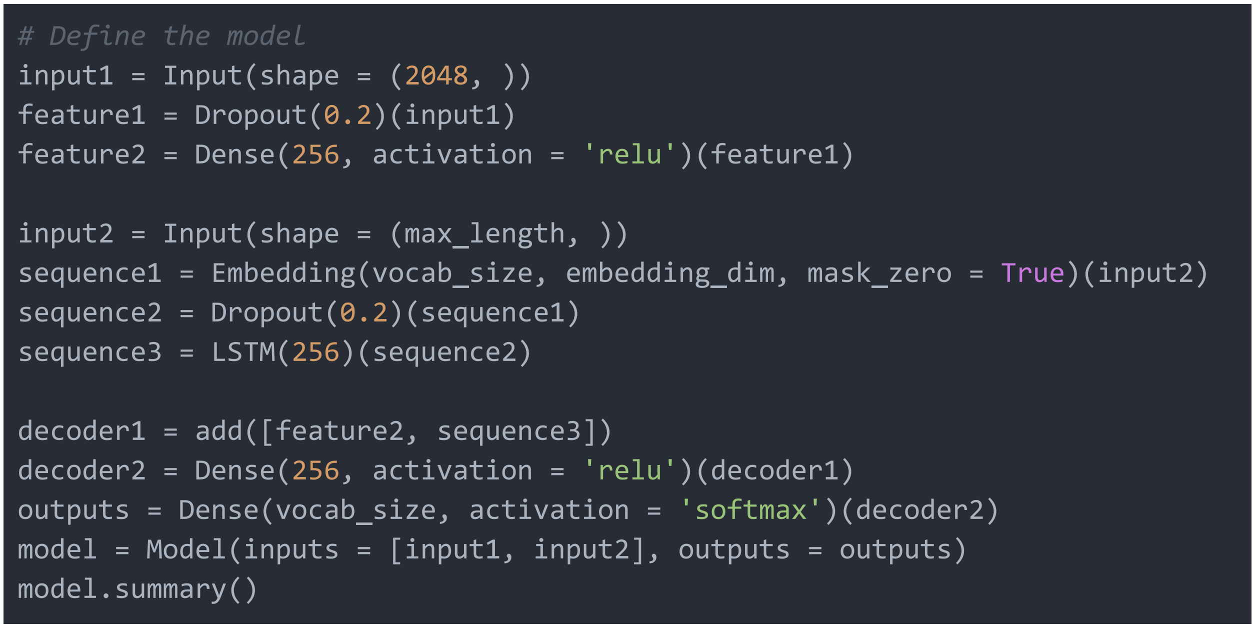
'startseq' + caption + 'endseq'

The token ***'startseq'*** will act as our first word when the feature vector of the image is fed to the decoder. The model will stop predicting words either when the token ***'endseq'*** appears or when we have predicted all words from the train dictionary. Let's see a table visualizing the described process for the image associated with the caption *"a girl going into a wooden building"*:

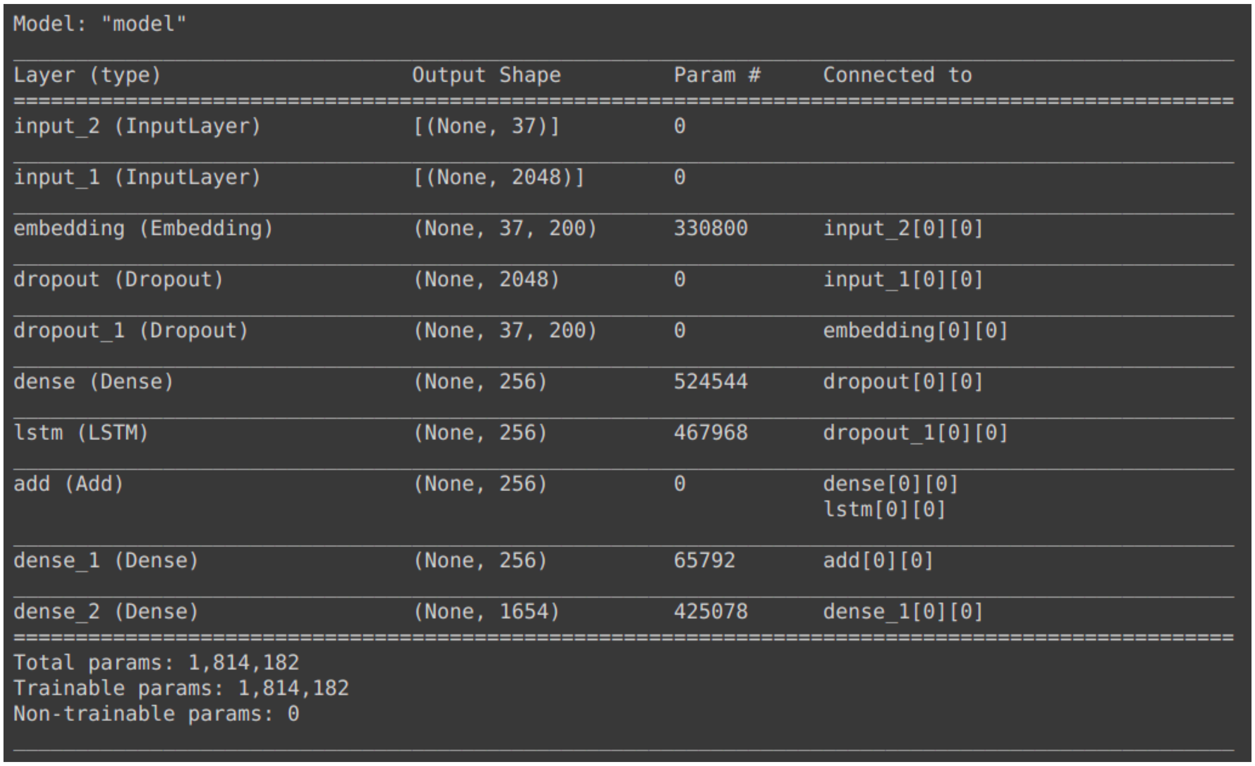
Input		Output
Feature Vector	***Partial Caption***	***Target Word***
img_1	*<startseq>*	a
img_1	*<startseq>* a	girl
img_1	*<startseq>* a girl	going
img_1	*<startseq>* a girl going	into
img_1	*<startseq>* a girl going into	a
img_1	*<startseq>* a girl going into a	wooden
img_1	*<startseq>* a girl going into a wooden	building
img_1	*<startseq>* a girl going into a wooden building	*<endseq>*

5.2. The Model's Architecture

Below, we show the Keras implementation of the model, as well as the model's summary:



```
# Define the model
input1 = Input(shape = (2048, ))
feature1 = Dropout(0.2)(input1)
feature2 = Dense(256, activation = 'relu')(feature1)

input2 = Input(shape = (max_length, ))
sequence1 = Embedding(vocab_size, embedding_dim, mask_zero = True)(input2)
sequence2 = Dropout(0.2)(sequence1)
sequence3 = LSTM(256)(sequence2)

decoder1 = add([feature2, sequence3])
decoder2 = Dense(256, activation = 'relu')(decoder1)
outputs = Dense(vocab_size, activation = 'softmax')(decoder2)
model = Model(inputs = [input1, input2], outputs = outputs)
model.summary()
```



```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to                     
==================================================================================================
input_2 (InputLayer)            [(None, 37)]         0                                            
__________________________________________________________________________________________________
input_1 (InputLayer)            [(None, 2048)]       0                                            
__________________________________________________________________________________________________
embedding (Embedding)           (None, 37, 200)      330800      input_2[0][0]                    
__________________________________________________________________________________________________
dropout (Dropout)               (None, 2048)         0           input_1[0][0]                    
__________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 37, 200)      0           embedding[0][0]                  
__________________________________________________________________________________________________
dense (Dense)                   (None, 256)          524544      dropout[0][0]                    
__________________________________________________________________________________________________
lstm (LSTM)                     (None, 256)          467968      dropout_1[0][0]                  
__________________________________________________________________________________________________
add (Add)                       (None, 256)          0           dense[0][0]                      
                                                                 lstm[0][0]                       
__________________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          65792       add[0][0]                        
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 1654)         425078      dense_1[0][0]                    
==================================================================================================
Total params: 1,814,182
Trainable params: 1,814,182
Non-trainable params: 0
__________________________________________________________________________________________________
```

Input_3 is the partial caption of *max_length*=34 which is fed into the embedding layer. This is where the words are mapped to the 200-dimensional Glove embedding. It is followed by a dropout of 0.5 to avoid overfitting. This is then fed into the LSTM for processing the sequence. ***Input_2*** is the feature vector of the image extracted using the InceptionV3 network. It is followed by a dropout of 0.5 to avoid overfitting and then fed into a Fully Connected layer. Both the Image model and the Caption model are then concatenated and fed into another Fully Connected layer. The output layer is a softmax layer, which generates the probability distribution across all 1654 words in the vocabulary.

5.3. The Model's Training

Before training the model we freeze the weights of the Glove embedding layer, as we do not want to retrain them. We compile the model using the ***'categorical_crossentropy'*** loss function, as we want a probability vector as output and we also use the ***'adam'*** optimizer. Since our dataset has 6000 images and 30000 captions we will create a generator function that can train the data in batches. This is happening because loading the whole dataset into the RAM will decelerate the system's performance. We finally train our model for 30 epochs with a batch size of 128.

```
model.layers[2].set_weights([embedding_matrix])
model.layers[2].trainable = False
model.compile(loss = 'categorical_crossentropy', optimizer = 'adam')
# model.fit([X1, X2], y, epochs = 50, batch_size = 256)
# you can increase the number of epochs for better results
```

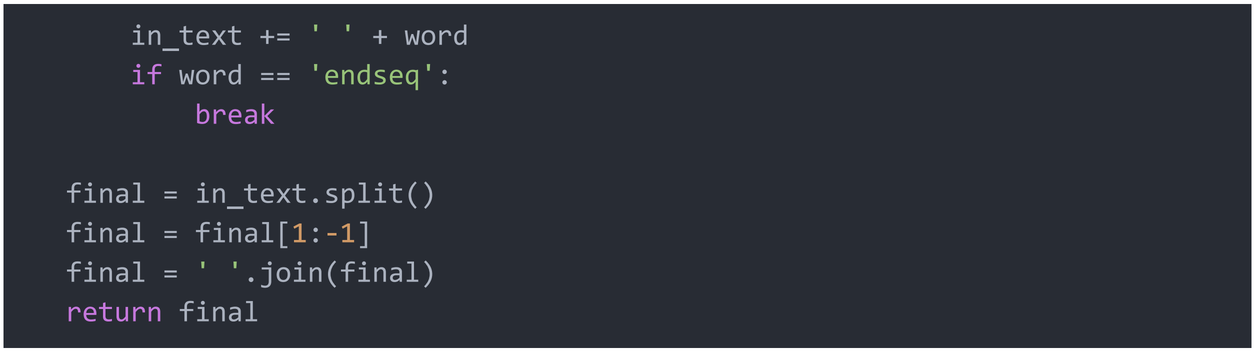
```
def data_generator(descriptions, photos, wordtoix, max_length,
num_photos_per_batch):
    X1, X2, y = list(), list(), list()
    n = 0
    # loop for ever over images
    while 1:
        for key, desc_list in descriptions.items():
            n += 1
            # retrieve the photo feature
            photo = photos[key]
            for desc in desc_list:
                # encode the sequence
                seq = [wordtoix[word] for word in desc.split(' ') if word in
wordtoix]
                # split one sequence into multiple X, y pairs
                for i in range(1, len(seq)):
                    # split into input and output pair
                    in_seq, out_seq = seq[:i], seq[i]
                    # pad input sequence
                    in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
                    # encode output sequence
                    out_seq = to_categorical([out_seq],
num_classes=vocab_size)[0]
                    # store
                    X1.append(photo)
                    X2.append(in_seq)
                    y.append(out_seq)

            if n == num_photos_per_batch:
                yield ([array(X1), array(X2)], array(y))
                X1, X2, y = list(), list(), list()
                n=0
```

```
epochs = 30
batch_size = 128
steps = len(train_descriptions) // batch_size
train_features = encoding_train

generator = data_generator(train_descriptions, train_features, wordtoix,
max_length, batch_size)
history = model.fit(generator, epochs=epochs, steps_per_epoch=steps,
verbose=1)

model.save('drive/My Drive/IntroDL/Final
Project/model_{}.h5'.format(epochs))  # creates a HDF5 file with the model
```

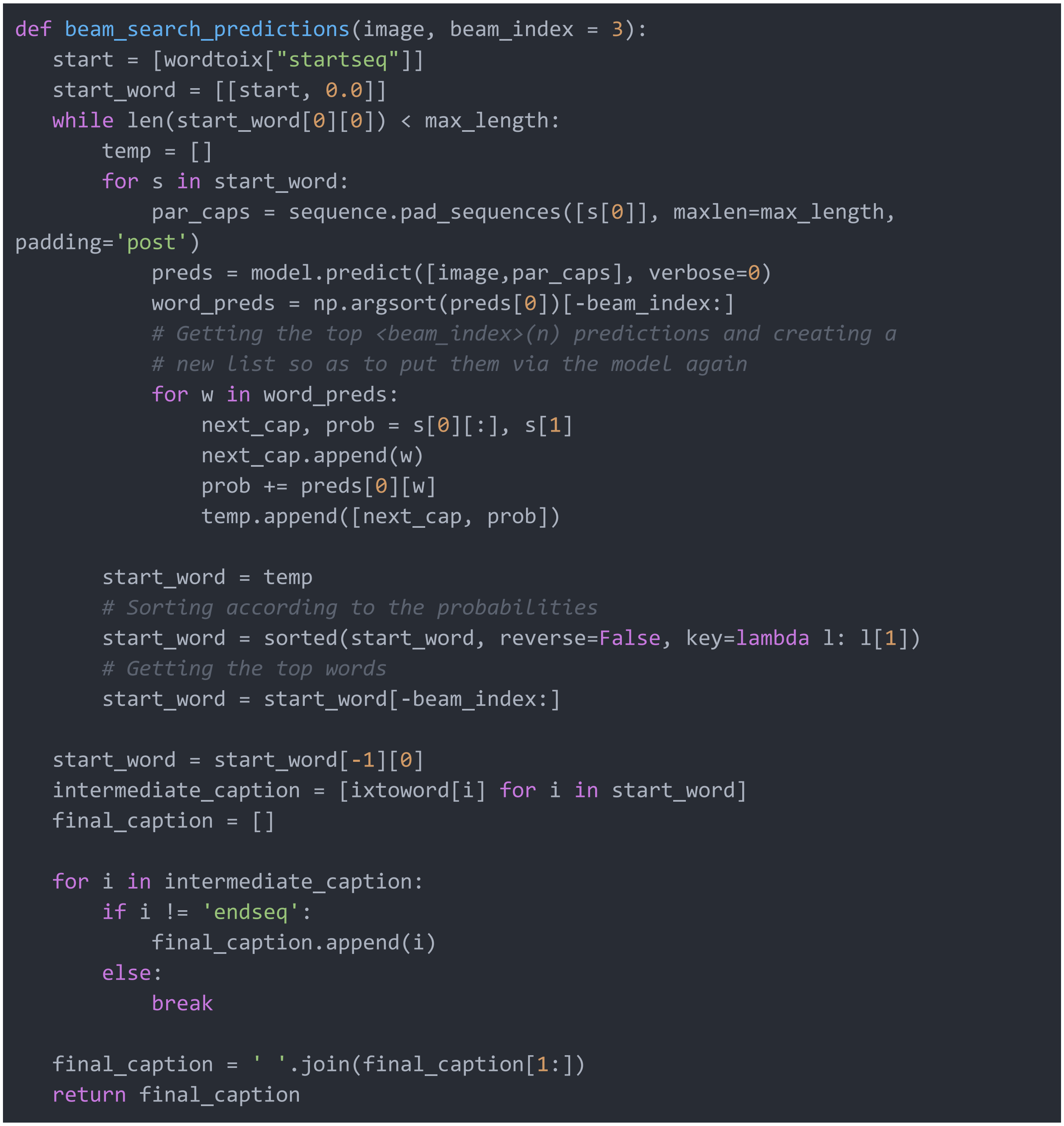
6. Generating Captions

As described above, the model receives a feature vector of an image together with a partial caption and generates the next word of that caption. When we insert a new image in the model, the first input is the feature vector together with the ***‘startseq’*** token. The model outputs a 1654-long vector, which is the probability distribution of all words in the vocabulary. For this reason, we should use a search algorithm to select the word with the maximum probability. We implement both ***Greedy Search*** (we select the word with the maximum probability) and ***Beam Search***. Finally, note that the model will stop predicting words either when the token ***‘endseq’*** appears or when we have predicted all words from the train dictionary. If any of the above conditions is met, we break the loop and report the generated caption as the output of the model for the given image. Let’s see an example of generating captions for an unknown image.

```
def greedySearch(photo):
   in_text = 'startseq'
   for i in range(max_length):
       sequence = [wordtoix[w] for w in in_text.split() if w in wordtoix]
       sequence = pad_sequences([sequence], maxlen=max_length)
       yhat = model.predict([photo,sequence], verbose=0)
       yhat = np.argmax(yhat)
       word = ixtoword[yhat]
```

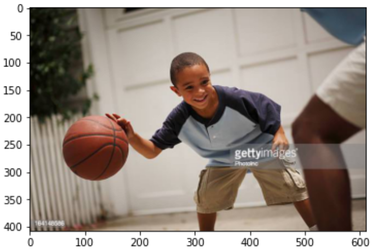
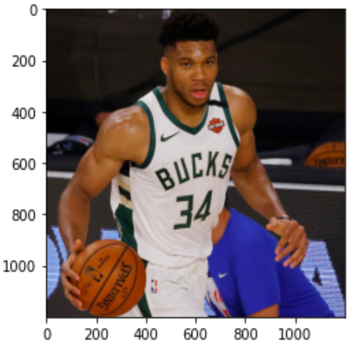


```python
        in_text += ' ' + word
        if word == 'endseq':
            break

    final = in_text.split()
    final = final[1:-1]
    final = ' '.join(final)
    return final
```



```python
def beam_search_predictions(image, beam_index = 3):
    start = [wordtoix["startseq"]]
    start_word = [[start, 0.0]]
    while len(start_word[0][0]) < max_length:
        temp = []
        for s in start_word:
            par_caps = sequence.pad_sequences([s[0]], maxlen=max_length,
padding='post')
            preds = model.predict([image,par_caps], verbose=0)
            word_preds = np.argsort(preds[0])[-beam_index:]
            # Getting the top <beam_index>(n) predictions and creating a
            # new list so as to put them via the model again
            for w in word_preds:
                next_cap, prob = s[0][:], s[1]
                next_cap.append(w)
                prob += preds[0][w]
                temp.append([next_cap, prob])

        start_word = temp
        # Sorting according to the probabilities
        start_word = sorted(start_word, reverse=False, key=lambda l: l[1])
        # Getting the top words
        start_word = start_word[-beam_index:]

    start_word = start_word[-1][0]
    intermediate_caption = [ixtoword[i] for i in start_word]
    final_caption = []

    for i in intermediate_caption:
        if i != 'endseq':
            final_caption.append(i)
        else:
            break

    final_caption = ' '.join(final_caption[1:])
    return final_caption
```

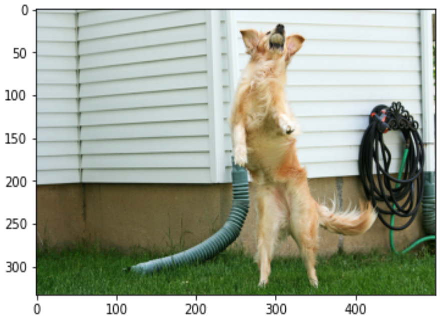
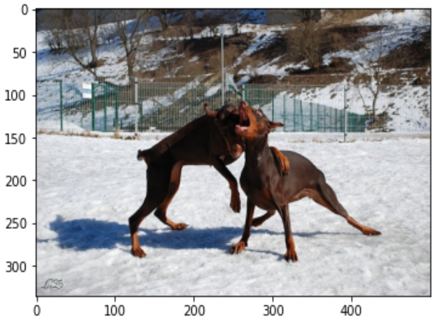
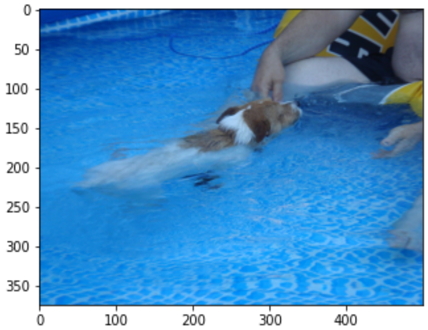
We present captions for Greedy Search (*1st caption*) and Beam Search with K=3, 5, 7, 10 (*2nd, 3rd, 4th, 5th caption respectively*):

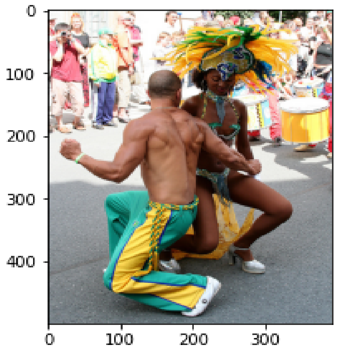
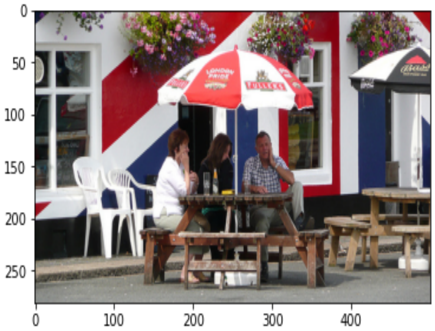
 0 50 100 150 200 250 300 350 400 0 100 200 300 400 500 600 gettyimages	***1. a boy in a blue shirt is running on a basketball court*** *2. a young boy wearing a blue shirt and blue shorts is running* *3. a basketball player dribbles the ball* *4. a basketball player dribbles the ball* *5. a basketball player dribbles the ball*
 0 200 400 600 800 1000 0 200 400 600 800 1000	***1.*** **a basketball player dribbles the ball** *2. a basketball player dribbles the ball* *3. a basketball player dribbles the ball* *4. a basketball player dribbles the ball on the court* *5. a basketball player dribbles the ball on the court*

And...we get some impressive results! Our model can successfully predict the evolution (from boy to man) of Giannis Antetokounmpo's career in basketball!

7. Results

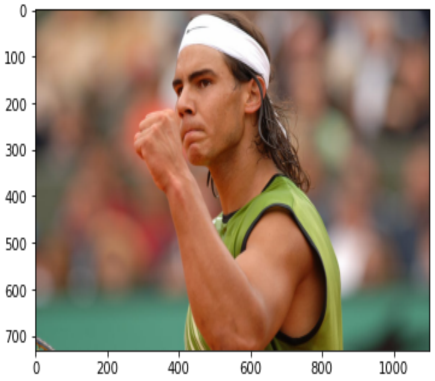
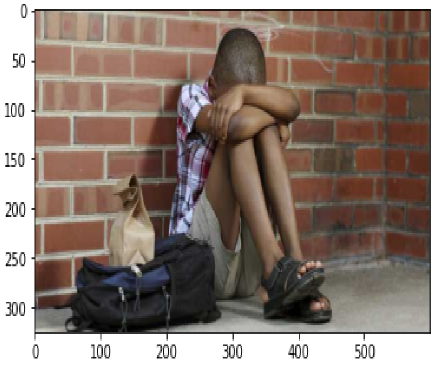
However, our model does not always produce nice results. Let's see first how the model performs when we try to generate captions for the images in the test set:

 0 50 100 150 200 250 300 0 100 200 300 400	1. *a dog jumps to catch a frisbee* 2. *a dog jumps to catch a frisbee in front of a building* 3. *a dog jumps to catch a frisbee in front of a building* 4. *a dog jumps to catch a frisbee in front of a building* 5. *a dog jumps to catch a frisbee in front of a building*
 0 50 100 150 200 250 300 0 100 200 300 400	1. *a brown dog is running on the sand* 2. *a large brown dog jumps over a smaller brown dog* 3. *a large brown dog jumps over a hurdle* 4. *a large brown dog jumps over a hurdle* 5. *a brown dog is running through the snow*
 0 50 100 150 200 250 300 350 0 100 200 300 400	1. *a dog swims in a pool* 2. *a brown and white dog swims in a pool* 3. *a brown and white dog swims in a pool* 4. *a brown and white dog swims in a pool* 5. *a brown and white dog swimming in a pool*

	1. *a man in a blue shirt is walking on a street* 2. *a man in a blue shirt is walking on a busy street* 3. *a man in a blue shirt is walking on the sidewalk* 4. *a man in a blue shirt is standing on a busy street* 5. *there is a man in a blue shirt carrying a baby in a crowded plaza*
 0 50 100 150 200 250 0 100 200 300 400	1. *a man is sitting on a bench in front of a large building* 2. *three people sit on a bench in front of a building* 3. *a group of people sit on a bench in front of a building* 4. *a group of people sit on a bench in front of a building* 5. *a group of people sit on a bench in front of a building*

Our model seems to perform nicely for images from the test dataset. In particular, we observe that Greedy Search produces the least descriptive caption, while Beam Search for all K produces more detailed descriptions. Beam Search for K=5, 7, 10 produces captions with mistaken information, while Beam Search for K=3 produces the best captions across all experiments. Furthermore, we observe that there are cases where our model cannot produce correct captions. For example, *'a brown dog is running on the sand'* instead of *'a brown dog is running on the snow' or 'a man in a blue shirt '* instead of *'a man without shirt'*.

Let's also see some incorrect predictions when feeding the model with images outside of the dataset:

 0 100 200 300 400 500 600 700 0 200 400 600 800 1000	1. *a man in a blue shirt is standing in a crowd* 2. *a man in a blue shirt is standing in a crowd* 3. *a man with a mohawk and a woman in a crowd* 4. *there is a man in a blue shirt with a mohawk on his hand* 5. *there is a man with a mohawk and a woman in a crowd*
 0 50 100 150 200 250 300 0 100 200 300 400 500	1. *a man in a black shirt and black shorts is sitting on a step* 2. *a man in a white shirt and black shorts is sitting on a sidewalk* 3. *a man wearing a black shirt and black shorts is sitting on a sidewalk* 4. *a man wearing a black shirt and black shorts is sitting on a sidewalk* 5. *there is a shirtless man and a woman sitting on a stone wall*

8. Conclusion

Implementing a model for image captioning is a very nice exercise for learning both Computer Vision and NLP techniques and yet, you can see them work together. However, in this project we implemented a basic model for image captioning. There are a series of next steps to be done in order to achieve a better performance. These steps include:

- Using more advanced datasets with more images and richer vocabularies (eg. Flickr30k, MS COCO)

- Improving dataset and training pipeline, incorporating more Keras utilities

- Better fine-tuning of the final model

- Implementing an Attention Based model. Attention-based mechanisms are becoming increasingly popular in deep learning because they can dynamically focus on the various parts of the input image while the output sequences are being produced. (see here)

- Using the BLEU evaluation metric to measure the quality of machine-generated text (see here)

- Developing a web-based application for deploying caption generation models

References

1. https://towardsdatascience.com/a-guide-to-image-captioning-e9fd5517f350

2. https://medium.com/@raman.shinde15/image-captioning-with-flickr8k-dataset-bleu-4bcba0b52926

3. https://www.analyticsvidhya.com/blog/2020/11/create-your-own-image-caption-generator-using-keras/

4. https://towardsdatascience.com/image-captioning-with-keras-teaching-computers-to-describe-pictures-c88a46a311b8

5. https://wiki.Image Captioning with Keras. Table of Contents: | by Harshall Lambapython.org/moin/Generators

6. http://karpathy.github.io/2015/05/21/rnn-effectiveness/

7. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

8. https://www.tensorflow.org/text/guide/word_embeddings

9. https://keras.io/examples/nlp/pretrained_word_embeddings/