Project Report

Image Caption Generator

Submitted by:

Lareb Amir

Contents

Introduction to image caption Generator

Image caption generator is a process of recognizing the context of an image and annotating it with relevant captions using deep learning, and computer vision. It includes the labelling of an image with English keywords with the help of datasets provided during model training. ImageNet dataset pretrained model InceptionV3 is used to train the CNN model. InceptionV3 is responsible for image feature extraction. These extracted features will be fed to the LSTM model which in turn generates the image caption.

CNN-LSTM Architecture

The CNN-LSTM architecture involves using CNN layers for feature extraction on input data combined with LSTMs to support sequence prediction. This model is specifically designed for sequence prediction problems with spatial inputs, like images or videos. They are widely used in Activity Recognition, Image Description, Video Description and many more.

The general architecture of the CNN-LSTM Model is as follows:

CNN-LSTMs are generally used when their inputs have spatial structure, such as the 2D structure or pixels in an image or the 1D structure of words in a sentence, paragraph, or document and also have a temporal structure in their input such as the order of images in a video or words in text, or require the generation of output with temporal structure such as words in a textual description.

CNN LSTMs were developed for visual time series prediction problems and the application of generating textual descriptions from sequences of images (e.g., videos). Specifically, the problems of:

- **Activity Recognition**: Generating a textual description of an activity demonstrated in a sequence of images.
- **Image Description**: Generating a textual description of a single image.
- **Video Description**: Generating a textual description of a sequence of images.

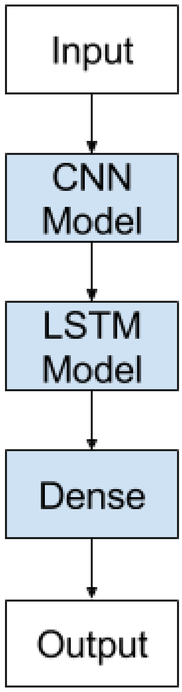


Figure 1 CNN-LSTM Architecture

Approach to the problem statement

We tackle this problem using an Encoder-Decoder model. Here our encoder model will combine both the encoded form of the image and the encoded form of the text caption and feed to the decoder.

Our model will treat CNN as the 'image model' and the RNN/LSTM as the 'language model' to encode the text sequences of varying length. The vectors resulting from both the encodings are then merged and processed by a Dense layer to make a final prediction.

We create a merge architecture in order to keep the image out of the RNN/LSTM and thus 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.

In our merge model, a different representation of the image can be combined with the final RNN state before each prediction.

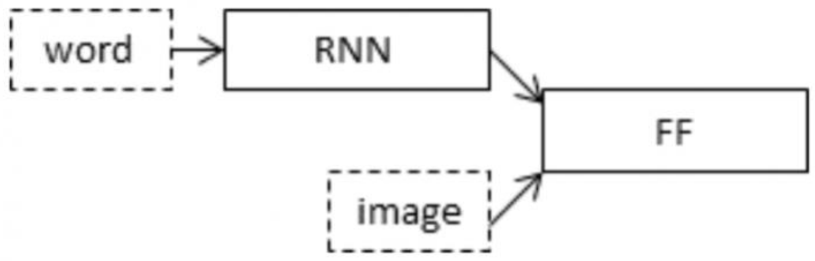


Figure 2: The above diagram is a visual representation of our approach.

The merging of image features with text encodings to a later stage in the architecture is advantageous and can generate better quality captions with smaller layers than the traditional inject architecture (CNN as encoder and RNN as a decoder).

To encode our image features we will make use of transfer learning. There are a lot of models that we can use like VGG-16, InceptionV3, ResNet, etc.

In other words, transfer learning is a machine learning method where we reuse a pre-trained model as the starting point for a model on a new task.
We make use of the inceptionV3 model which has the least number of training parameters in comparison to the others and also outperforms them.

To encode our text sequence, we will map every word to a 200-dimensional vector. For this we use a pre-trained Glove model. This mapping will be done in a separate layer after the input layer called the embedding layer.

To generate the caption we will be using two popular methods which are Greedy Search and Beam Search. These methods will help us in picking the best words to accurately define the image.

Dataset:

the Flickr8k datasetis used in this implementation, each image is associated with five different captions that describe the entities and events depicted in the image that were collected. 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.

Our dataset structure is as follows: -

1. Flick8k/
    a. Flick8k_Dataset/: - *contains the 8000 images*
    b. Flick8k_Text/
        i. Flickr8k.token.txt: - *contains the image id along with the 5 captions*
        ii. Flickr8k.trainImages.txt: - *contains the training image id's*
        iii. Flickr8k.testImages.txt: - *contains the test image id's*

Pre-requisite

Install below libraries, to begin with, the project:

- pip install TensorFlow
- pip install Keras
- pip install NumPy
- Pip install tqdm
- Pip install jupyterlab

Code Implementation:

Step 1: - Import the required libraries

```
import numpy as np
from numpy import array
import matplotlib.pyplot as plt
%matplotlib inline

import string
import os
import glob
from PIL import Image
from time import time

from keras import Input, layers
from keras import optimizers
from tensorflow.keras.optimizers import Adam
from keras.preprocessing import sequence
from keras.preprocessing import image
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.layers import LSTM, Embedding, Dense, Activation, Flatten, Reshape, Dropout
from keras.layers.wrappers import Bidirectional
from keras.layers.merge import add
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from keras.models import Model
from tensorflow.keras.utils import to_categorical
```

Step 2: - Data loading and Pre-processing

We will define all the paths to the files that we require and save the images id and their captions.

```
[2]:
## Step 2 : Data Loading and Pre-Processing

token_path = "../input/flickr-8k/Flickr8k.token.txt"
train_images_path = '../input/flickr-8k/Flickr_8k.trainImages.txt'
test_images_path = '../input/flickr-8k/Flickr_8k.testImages.txt'
images_path = '../input/flickr8k/Images/'
glove_path = '../input/glove6b/'

doc = open(token_path,'r').read()
print(doc[:410])
```

```
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 .
```

So, we can see the format in which our image ids and their captions are stored. Next, we create a dictionary named "descriptions" which contains the name of the image as keys and a list of the 5 captions for the corresponding image as values.

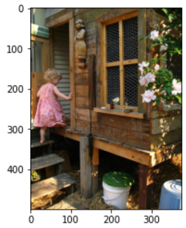
```
[3]:
descriptions = dict()
for line in doc.split('\n'):
        tokens = line.split()
        if len(line) > 2:
          image_id = tokens[0].split('.')[0]
          image_desc = ' '.join(tokens[1:])
          if image_id not in descriptions:
              descriptions[image_id] = list()
          descriptions[image_id].append(image_desc)
```

Now let’s perform some basic text clean to get rid of punctuation and convert our descriptions to lowercase.

[4]:
```
table = str.maketrans('', '', string.punctuation)
for key, desc_list in descriptions.items():
    for i in range(len(desc_list)):
        desc = desc_list[i]
        desc = desc.split()
        desc = [word.lower() for word in desc]
        desc = [w.translate(table) for w in desc]
        desc_list[i] =  ' '.join(desc)
```

Let’s visualize an example image and its captions: -

[5]:
```
## Lets visualize and example image
pic = '1000268201_693b08cb0e.jpg'
x=plt.imread(images_path+pic)
plt.imshow(x)
plt.show()
descriptions['1000268201_693b08cb0e']
```



[5]:
```
['a child in a pink dress is climbing up a set of stairs in an entry way ',
 'a girl going into a wooden building ',
 'a little girl climbing into a wooden playhouse ',
 'a little girl climbing the stairs to her playhouse ',
 'a little girl in a pink dress going into a wooden cabin ']
```

Next, we create a vocabulary of all the unique words present across all the 8000*5 (i.e., 40000) image captions in the data set. We have 8828 unique words across all the 40000 image captions.

[6]:
```
vocabulary = set()
for key in descriptions.keys():
        [vocabulary.update(d.split()) for d in descriptions[key]]
print('Original Vocabulary Size: %d' % len(vocabulary))
```

```
Original Vocabulary Size: 8828
```

Now let’s save the image ids and their new cleaned captions in the same format as the token.txt file.

[7]:
```
lines = list()
for key, desc_list in descriptions.items():
    for desc in desc_list:
        lines.append(key + ' ' + desc)
new_descriptions = '\n'.join(lines)
```

Next, we load all the 6000-training image id’s in a variable train from the ‘Flickr_8k.trainImages.txt’ file: -

[8]:
```
doc = open(train_images_path,'r').read()
dataset = list()
for line in doc.split('\n'):
    if len(line) > 1:
      identifier = line.split('.')[0]
      dataset.append(identifier)

train = set(dataset)
```

Now we save all the training and testing images in train_img and test_img lists respectively: -

[9]:

```
img = glob.glob(images_path + '*.jpg')
train_images = set(open(train_images_path, 'r').read().strip().split('\n'))
train_img = []
for i in img:
    if i[len(images_path):] in train_images:
        train_img.append(i)

test_images = set(open(test_images_path, 'r').read().strip().split('\n'))
test_img = []
for i in img:
    if i[len(images_path):] in test_images:
        test_img.append(i)
```

Now, we load the descriptions of the training images into a dictionary. However, we will add two tokens in every caption, which are **startseq** and **endseq**: -

[10]:

```
train_descriptions = dict()
for line in new_descriptions.split('\n'):
    tokens = line.split()
    image_id, image_desc = tokens[0], tokens[1:]
    if image_id in train:
        if image_id not in train_descriptions:
            train_descriptions[image_id] = list()
        desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
        train_descriptions[image_id].append(desc)
```

```
[11… {'1000268201_693b08cb0e': ['startseq a child in a pink dress is climbing up a set of stairs in an entry way endseq',
  'startseq a girl going into a wooden building endseq',
  'startseq a little girl climbing into a wooden playhouse endseq',
  'startseq a little girl climbing the stairs to her playhouse endseq',
  'startseq a little girl in a pink dress going into a wooden cabin endseq'],
 '1001773457_577c3a7d70': ['startseq a black dog and a spotted dog are fighting endseq',
  'startseq a black dog and a tricolored dog playing with each other on the road endseq',
  'startseq a black dog and a white dog with brown spots are staring at each other in the street endseq',
  'startseq two dogs of different breeds looking at each other on the road endseq',
  'startseq two dogs on pavement moving toward each other endseq'],
 '1002674143_1b742ab4b8': ['startseq a little girl covered in paint sits in front of a painted rainbow with her hands in a bowl endseq',
  'startseq a little girl is sitting in front of a large painted rainbow endseq',
```

Create a list of all the training captions: -

[11]:

```
all_train_captions = []
for key, val in train_descriptions.items():
    for cap in val:
        all_train_captions.append(cap)
```

```
[12… ['startseq a child in a pink dress is climbing up a set of stairs in an entry way endseq',
  'startseq a girl going into a wooden building endseq',
  'startseq a little girl climbing into a wooden playhouse endseq',
  'startseq a little girl climbing the stairs to her playhouse endseq',
  'startseq a little girl in a pink dress going into a wooden cabin endseq',
  'startseq a black dog and a spotted dog are fighting endseq',
  'startseq a black dog and a tricolored dog playing with each other on the road endseq',
  'startseq a black dog and a white dog with brown spots are staring at each other in the street endseq',
  'startseq two dogs of different breeds looking at each other on the road endseq',
  'startseq two dogs on pavement moving toward each other endseq',
  'startseq a little girl covered in paint sits in front of a painted rainbow with her hands in a bowl endseq',
  'startseq a little girl is sitting in front of a large painted rainbow endseq',
```

To make our model more robust we will reduce our vocabulary to only those words which occur at least 10 times in the entire corpus.

[12]:

```
word_count_threshold = 10
word_counts = {}
nsents = 0
for sent in all_train_captions:
    nsents += 1
    for w in sent.split(' '):
        word_counts[w] = word_counts.get(w, 0) + 1
vocab = [w for w in word_counts if word_counts[w] >= word_count_threshold]

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

```
Vocabulary = 1659
```

```
     Vocabulary = 1659
[13… ['startseq',
  'a',
  'child',
  'in',
  'pink',
  'dress',
  'is',
  'climbing',
  'up',
  'set',
  'of',
```

Now we create two dictionaries to map words to an index and vice versa. Also, we append 1 to our vocabulary since we append 0's to make all captions of equal length.

```
[13]:
ixtoword = {}
wordtoix = {}
ix = 1
for w in vocab:
    wordtoix[w] = ix
    ixtoword[ix] = w
    ix += 1

vocab_size = len(ixtoword) + 1
```

```
[15]:
wordtoix
```

```
[15…
{'startseq': 1,
 'a': 2,
 'child': 3,
 'in': 4,
 'pink': 5,
 'dress': 6,
 'is': 7,
 'climbing': 8,
 'up': 9,
 'set': 10,
 'of': 11,
 'stairs': 12,
```

We also need to find out what the max length of a caption can be since we cannot have captions of arbitrary length.

So, for this we take the maximum length of a caption from our training dataset.

```
[14]:
all_desc = list()
for key in train_descriptions.keys():
    [all_desc.append(d) for d in train_descriptions[key]]
lines = all_desc
max_length = max(len(d.split()) for d in lines)

print('Description Length: %d' % max_length)
```

```
Description Length: 38
```

Step 3: Glove Embeddings

Glove is global word vector.

Word vectors map words to a vector space, where similar words are clustered together and different words are separated.

The basic premise behind Glove is that we can derive sematic relationships between words from the co-occurence matrix.

For our model, we will map all the 38-word long caption to a 200-dimension vector using Glove.

```
[17]:
embeddings_index = {}
f = open(os.path.join(glove_path, 'glove.6B.200d.txt'), encoding="utf-8")
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
embeddings_index
```

```
[17…
{'the': array([-7.1549e-02,  9.3459e-02,  2.3738e-02, -9.0339e-02,  5.6123e-02,
         3.2547e-01, -3.9796e-01, -9.2139e-02,  6.1181e-02, -1.8950e-01,
         1.3061e-01,  1.4349e-01,  1.1479e-02,  3.8158e-01,  5.4030e-01,
        -1.4088e-01,  2.4315e-01,  2.3036e-01, -5.5339e-01,  4.8154e-02,
         4.5662e-01,  3.2338e+00,  2.0199e-02,  4.9019e-02, -1.4132e-02,
         7.6017e-02, -1.1527e-01,  2.0060e-01, -7.7657e-02,  2.4328e-01,
         1.6368e-01, -3.4118e-01, -6.6070e-02,  1.0152e-01,  3.8232e-02,
        -1.7668e-01, -8.8153e-01, -3.3895e-01, -3.5481e-02, -5.5095e-01,
        -1.6899e-02, -4.3982e-01,  3.9004e-02,  4.0447e-01, -2.5880e-01,
         6.4594e-01,  2.6641e-01,  2.8009e-01, -2.4625e-02,  6.3302e-01,
        -3.1700e-01,  1.0271e-01,  3.0886e-01,  9.7792e-02, -3.8227e-01,
         8.6552e-02,  4.7075e-02,  2.3511e-01, -3.2127e-01, -2.8538e-01,
```

Now, we make matrix of shape (1660,220) consisting of our vocabulary and the 200-d vector.

```
[18]:
embedding_dim = 200
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, i in wordtoix.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

Step 4: Model building and Training

An approach we have adopted is transfer learning using Inception V-3 network which is pre-trained on the Image-Net dataset.

```
[ ]: # model = In`ceptionV3(weights='imagenet')
```

```
[19]: model = InceptionV3(weights='../input/imagecaptiongeneratorweights/inception_v3_weights_tf_dim_ordering_tf_kernels.h5')
```

```
2022-06-03 03:56:48.970663: I tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool with default inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.
```

Since we are using Inception V-3, we need to pre-process our inputs before feeding it into the model.

Hence, we need to define a pre-process function to reshape the images (299 x 299) and feed the preprocess_input() function.

```
[21]:
def preprocess(image_path):
    img = image.load_img(image_path, target_size=(299, 299))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    return x
```

Now, we can go ahead with training and testing images i.e. we can extract the images vectors of shape (2048 ,)

```
[22]:
def encode(image):
    image = preprocess(image)
    fea_vec = model_new.predict(image)
    fea_vec = np.reshape(fea_vec, fea_vec.shape[1])
    return fea_vec

encoding_train = {}
for img in train_img:
    encoding_train[img[len(images_path):]] = encode(img)
train_features = encoding_train

encoding_test = {}
for img in test_img:
    encoding_test[img[len(images_path):]] = encode(img)
```

```
2022-06-03 03:56:53.312181: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
```

Now let's define our model.

We are creating a Merge model where we combine the image vector and the partial caption. Therefore, our model will have 3 major steps:

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

1. *The Photo Feature Extractor model expects input photo features to be a vector of 2,048 elements. These are processed by a Dense layer to produce a 256-element representation of the photo.*
2. *The Sequence Processor model expects input sequences with a pre-defined length (38 words) which are fed into an Embedding layer that uses a mask to ignore padded values. This is followed by an LSTM layer with 256 memory units.*
3. *Both the input models produce a 256-element vector. Further, both input models use regularization in the form of 50% dropout. This is to reduce overfitting the training dataset, as this model configuration learns very fast.*
4. *The Decoder model merges the vectors from both input models using an addition operation. This is then fed to a Dense 256 neuron layer and then to a final output Dense layer that makes a softmax prediction over the entire output vocabulary for the next word in the sequence.*

[23]:

```python
inputs1 = Input(shape=(2048,))
fe1 = Dropout(0.5)(inputs1)
fe2 = Dense(256, activation='relu')(fe1)

inputs2 = Input(shape=(max_length,))
se1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
se2 = Dropout(0.5)(se1)
se3 = LSTM(256)(se2)

decoder1 = add([fe2, se3])
decoder2 = Dense(256, activation='relu')(decoder1)
outputs = Dense(vocab_size, activation='softmax')(decoder2)

model = Model(inputs=[inputs1, inputs2], outputs=outputs)
model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_3 (InputLayer)            [(None, 38)]         0
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, 2048)]       0
__________________________________________________________________________________________________
embedding (Embedding)           (None, 38, 200)      332000      input_3[0][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 2048)         0           input_2[0][0]
__________________________________________________________________________________________________
dropout_1 (Dropout)             (None, 38, 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, 1660)         426620      dense_1[0][0]
==================================================================================================
Total params: 1,816,924
Trainable params: 1,816,924
Non-trainable params: 0
__________________________________________________________________________________________________
```

Input_3 is the partial caption of max length 38 which is fed into the embedding layer. This is where the words are mapped to the 200-d 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 image vector extracted by our 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 Language model are then concatenated by adding and fed into another Fully Connected layer. The layer is a softmax layer that provides probabilities to our 1660-word vocabulary.

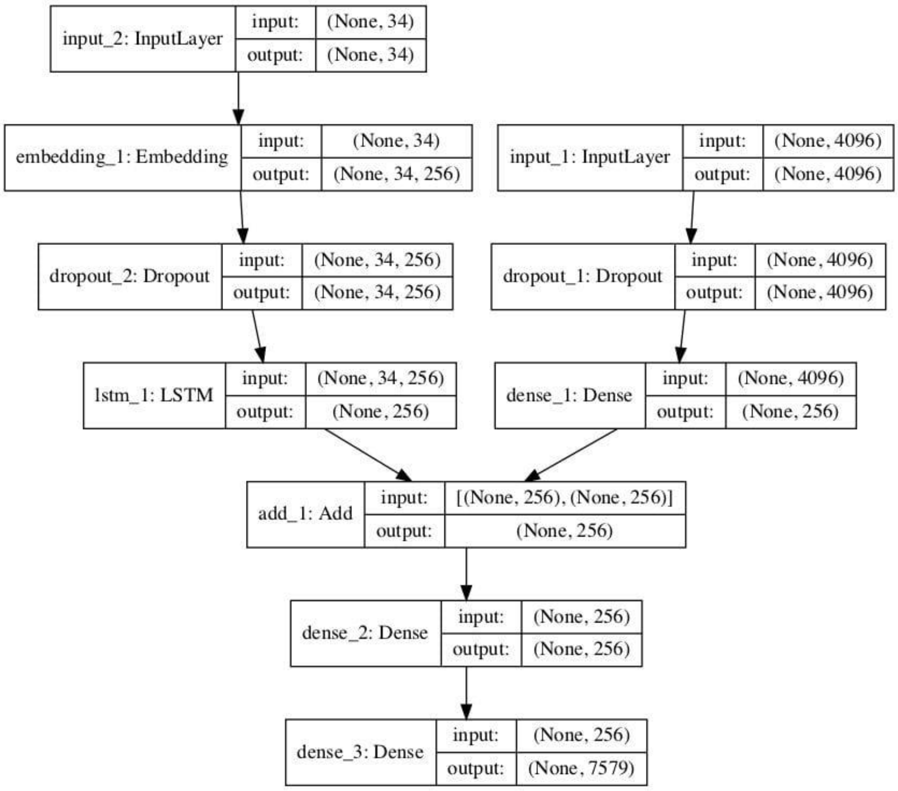


Figure 3 Displays the model creation for CNN-LSTM Architecture.

Dense Layer

Dense Layer is simple layer of neurons in which each neuron receives input from all the neurons of previous layer, thus called as dense. Dense Layer is used to classify image based on output from convolutional layers.

Each Layer in the Neural Network contains neurons, which compute the weighted average of its input and this weighted average is passed through a linear function, called as an “Softmax activation function” for LSTM and "Relu activation function” a non-linear function for Image processing. Result of this activation function is treated as output of that neuron. In similar way, the process is carried out for all neurons of all layers.

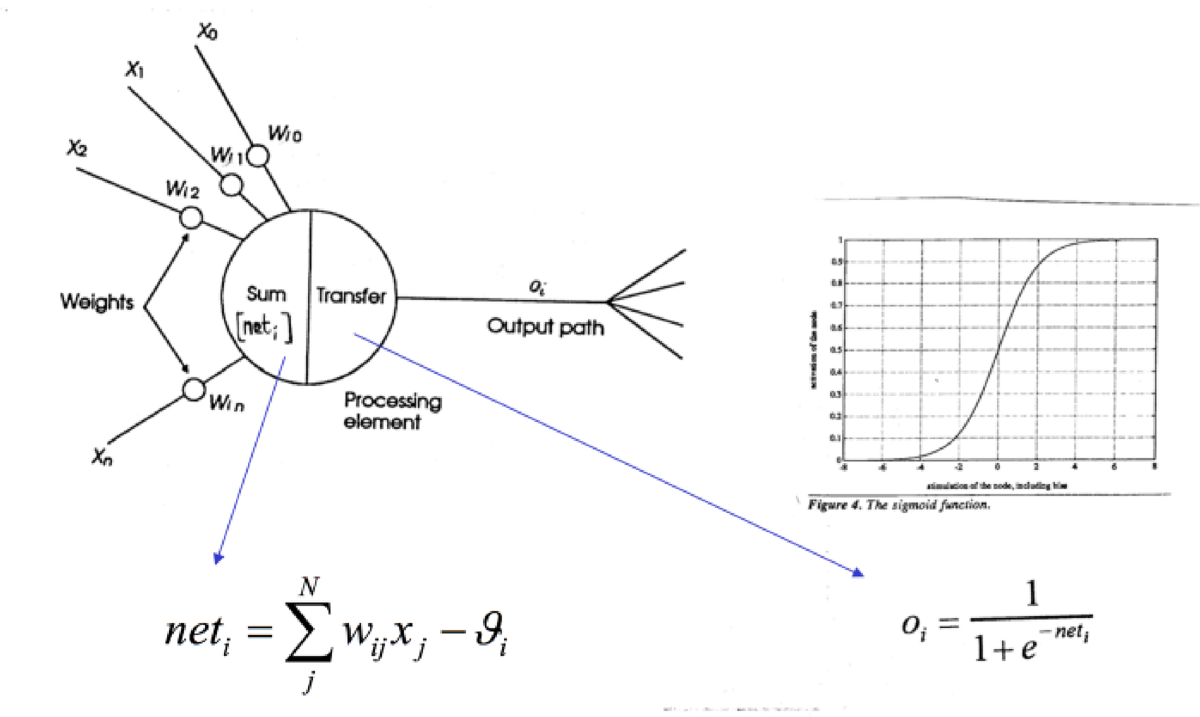


Figure 4: Working of single neuron. A layer contains multiple number of such neurons.

Step 5: Model Training

Before training the model we need to keep in mind that we do not want to retrain the weights in our embedding layer (pre-trained Glove vectors).

```
[24]:
model.layers[2].set_weights([embedding_matrix])
model.layers[2].trainable = False
```

Next, compile the model using Categorical_Crossentropy as the Loss function and Adam as the optimizer.

Categorical crossentropy

Categorical crossentropy is a loss function that is used in multi-class classification tasks. These are tasks where an example can only belong to one out of many possible categories, and the model must decide which one. Formally, it is designed to quantify the difference between two probability distributions.

Adam optimizer involves a combination of two gradient descent methodologies:

Momentum: This algorithm is used to accelerate the gradient descent algorithm by taking into consideration the 'exponentially weighted average' of the gradients.

Using averages makes the algorithm converge towards the minima in a faster pace.

```
[25]: model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Since our dataset has 6000 images and 40000 captions, we will create a function that can train the data in batches.

```
[26]:
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 +'.jpg']
            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
```

Now, let's train our model with for 30 epochs with a batch_size of 10 and 2000 steps per epoch.

```
[27]:
epochs = 30
batch_size = 10
steps = len(train_descriptions)//batch_size
callback = EarlyStopping(monitor='loss', patience=5)
generator = data_generator(train_descriptions, train_features, wordtoix, max_length, batch_size)
history = model.fit(generator, epochs=epochs, steps_per_epoch=steps, verbose=1,callbacks=[callback])
model.save("model.h5")
```

```
Epoch 1/30
600/600 [==============================] - 475s 784ms/step - loss: 3.9176
Epoch 2/30
600/600 [==============================] - 468s 780ms/step - loss: 3.1566
Epoch 3/30
600/600 [==============================] - 469s 781ms/step - loss: 2.9263
Epoch 4/30
600/600 [==============================] - 470s 783ms/step - loss: 2.7932
Epoch 5/30
600/600 [==============================] - 468s 780ms/step - loss: 2.6955
Epoch 6/30
600/600 [==============================] - 468s 779ms/step - loss: 2.6210
Epoch 7/30
600/600 [==============================] - 467s 778ms/step - loss: 2.5601
Epoch 8/30
600/600 [==============================] - 472s 787ms/step - loss: 2.5091
Epoch 9/30
600/600 [==============================] - 470s 783ms/step - loss: 2.4635
Epoch 10/30
600/600 [==============================] - 470s 783ms/step - loss: 2.4241
Epoch 11/30
600/600 [==============================] - 475s 792ms/step - loss: 2.3897
Epoch 12/30
600/600 [==============================] - 471s 784ms/step - loss: 2.3581
Epoch 13/30
600/600 [==============================] - 474s 790ms/step - loss: 2.3305
Epoch 14/30
600/600 [==============================] - 479s 799ms/step - loss: 2.3040
Epoch 15/30
600/600 [==============================] - 473s 789ms/step - loss: 2.2812
Epoch 16/30
600/600 [==============================] - 472s 787ms/step - loss: 2.2599
Epoch 17/30
600/600 [==============================] - 471s 785ms/step - loss: 2.2404
Epoch 18/30
600/600 [==============================] - 472s 787ms/step - loss: 2.2236
Epoch 19/30
600/600 [==============================] - 474s 789ms/step - loss: 2.2063
Epoch 20/30
600/600 [==============================] - 479s 798ms/step - loss: 2.1913
Epoch 21/30
600/600 [==============================] - 476s 793ms/step - loss: 2.1754
Epoch 22/30
600/600 [==============================] - 470s 784ms/step - loss: 2.1628
Epoch 23/30
600/600 [==============================] - 472s 787ms/step - loss: 2.1500
Epoch 24/30
600/600 [==============================] - 480s 799ms/step - loss: 2.1366
Epoch 25/30
600/600 [==============================] - 478s 796ms/step - loss: 2.1273
Epoch 26/30
600/600 [==============================] - 475s 792ms/step - loss: 2.1167
Epoch 27/30
600/600 [==============================] - 470s 783ms/step - loss: 2.1053
Epoch 28/30
600/600 [==============================] - 470s 782ms/step - loss: 2.0959
Epoch 29/30
600/600 [==============================] - 470s 783ms/step - loss: 2.0869
Epoch 30/30
600/600 [==============================] - 470s 783ms/step - loss: 2.0790
```

Step 6: Greedy and Beam Search:

Greedy Search

As the model generates a 1660 long vector with a probability distribution across all the words in the vocabulary, we greedily pick the word with the highest probability to get the next word prediction. This method is called Greedy Search.

[27]:

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

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

Beam Search:

Beam Search is where we take top k predictions, feed them again in the model and then sort them using the probabilities returned by the model. So, the list will always contain the top k predictions and we take the one with the highest probability and go through it till we encounter **endseq** or reach the maximum caption length.

[28]:

```
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
```

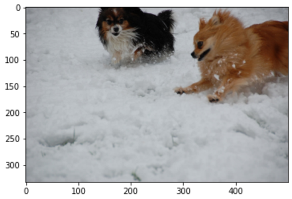
Step 7: Evaluations

Let's now test our model on different images and see what captions it generates. We will also look at the different captions generated by Greedy search and Beam search with different k values.

First, we will take a look at the example image we saw at the start of the article. We saw that the caption for the image was 'A black dog and a brown dog in the snow'. Let's see how our model compares.

[30]:

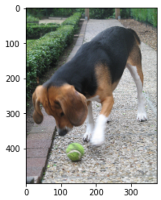
```
pic = '2398605966_1d0c9e6a20.jpg'
image = encoding_test[pic].reshape((1,2048))
x=plt.imread(images_path + pic)
plt.imshow(x)
plt.show()

print("Greedy Search:",greedySearch(image))
print("Beam Search, K = 3:",beam_search_predictions(image, beam_index = 3))
print("Beam Search, K = 5:",beam_search_predictions(image, beam_index = 5))
print("Beam Search, K = 7:",beam_search_predictions(image, beam_index = 7))
print("Beam Search, K = 10:",beam_search_predictions(image, beam_index = 10))
```



```
Greedy Search: a small dog is running through the snow
Beam Search, K = 3: a small white dog running through the snow
Beam Search, K = 5: a brown and white dog is running in the snow
Beam Search, K = 7: a brown and white dog in the snow
Beam Search, K = 10: a brown and white dog in the snow
```

[31]:

```
pic = list(encoding_test.keys())[1]
image = encoding_test[pic].reshape((1,2048))
x=plt.imread(images_path+pic)
plt.imshow(x)
plt.show()

print("Greedy:",greedySearch(image))
print("Beam Search, K = 3:",beam_search_predictions(image, beam_index = 3))
print("Beam Search, K = 5:",beam_search_predictions(image, beam_index = 5))
print("Beam Search, K = 7:",beam_search_predictions(image, beam_index = 7))
```



```
Greedy: a dog is running on the grass
Beam Search, K = 3: a brown and white dog is biting a red tennis ball in a yard
Beam Search, K = 5: a brown and white dog is playing with a red ball
Beam Search, K = 7: a brown and white dog is playing with a red ball
```

Step 8: Comparisons

Model 1

[21]:

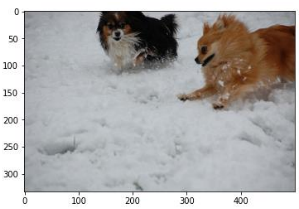
```
inputs1 = Input(shape=(2048,))
fe1 = Dropout(0.5)(inputs1)
fe2 = Dense(256, activation='relu')(fe1)

inputs2 = Input(shape=(max_length,))
se1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
se2 = Dropout(0.5)(se1)
se3 = LSTM(256)(se2)

decoder1 = add([fe2, se3])
decoder2 = Dense(256, activation='relu')(decoder1)
outputs = Dense(vocab_size, activation='softmax')(decoder2)

model = Model(inputs=[inputs1, inputs2], outputs=outputs)
model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________
________
Layer (type)                    Output Shape         Param #     Connected to
==========================================================================================
======
input_3 (InputLayer)            [(None, 38)]         0
__________________________________________________________________________________________
______
input_2 (InputLayer)            [(None, 2048)]       0
__________________________________________________________________________________________
______
embedding (Embedding)           (None, 38, 200)      332000      input_3[0][0]
__________________________________________________________________________________________
______
dropout (Dropout)               (None, 2048)         0           input_2[0][0]
__________________________________________________________________________________________
______
dropout_1 (Dropout)             (None, 38, 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, 1660)         426620      dense_1[0][0]
==========================================================================================
======
Total params: 1,816,924
Trainable params: 1,816,924
Non-trainable params: 0
```

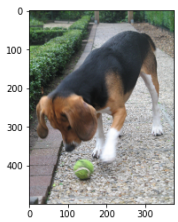
```
In [25]:
epochs = 30
batch_size = 3
steps = len(train_descriptions)//batch_size

generator = data_generator(train_descriptions, train_features, wordtoix, max_length, batch_siz
e)
model.fit(generator, epochs=epochs, steps_per_epoch=steps, verbose=1)
```

```
Epoch 1/30
2000/2000 [==============================] - 215s 108ms/step - loss: 3.6892
Epoch 2/30
2000/2000 [==============================] - 215s 107ms/step - loss: 3.0343
Epoch 3/30
2000/2000 [==============================] - 220s 110ms/step - loss: 2.8368
Epoch 4/30
2000/2000 [==============================] - 223s 111ms/step - loss: 2.7181
Epoch 5/30
2000/2000 [==============================] - 225s 113ms/step - loss: 2.6358
Epoch 6/30
2000/2000 [==============================] - 217s 108ms/step - loss: 2.5726
Epoch 7/30
2000/2000 [==============================] - 218s 109ms/step - loss: 2.5210
Epoch 8/30
2000/2000 [==============================] - 222s 111ms/step - loss: 2.4801
Epoch 9/30
2000/2000 [==============================] - 228s 114ms/step - loss: 2.4428
Epoch 10/30
2000/2000 [==============================] - 223s 111ms/step - loss: 2.4134
Epoch 11/30
2000/2000 [==============================] - 222s 111ms/step - loss: 2.3871
Epoch 12/30
2000/2000 [==============================] - 216s 108ms/step - loss: 2.3641
Epoch 13/30
2000/2000 [==============================] - 226s 113ms/step - loss: 2.3396
Epoch 14/30
2000/2000 [==============================] - 215s 107ms/step - loss: 2.3228
Epoch 15/30
2000/2000 [==============================] - 227s 114ms/step - loss: 2.3070
Epoch 16/30
2000/2000 [==============================] - 231s 115ms/step - loss: 2.2922
Epoch 17/30
2000/2000 [==============================] - 218s 109ms/step - loss: 2.2766
Epoch 18/30
2000/2000 [==============================] - 233s 117ms/step - loss: 2.2654
Epoch 19/30
2000/2000 [==============================] - 219s 109ms/step - loss: 2.2519
Epoch 20/30
2000/2000 [==============================] - 238s 119ms/step - loss: 2.2412
Epoch 21/30
2000/2000 [==============================] - 220s 110ms/step - loss: 2.2328
Epoch 22/30
2000/2000 [==============================] - 223s 111ms/step - loss: 2.2262
Epoch 23/30
2000/2000 [==============================] - 213s 107ms/step - loss: 2.2161
Epoch 24/30
2000/2000 [==============================] - 215s 108ms/step - loss: 2.2057
Epoch 25/30
2000/2000 [==============================] - 224s 112ms/step - loss: 2.1982
Epoch 26/30
2000/2000 [==============================] - 226s 113ms/step - loss: 2.1931
Epoch 27/30
2000/2000 [==============================] - 225s 113ms/step - loss: 2.1883
Epoch 28/30
2000/2000 [==============================] - 205s 102ms/step - loss: 2.1813
Epoch 29/30
2000/2000 [==============================] - 228s 114ms/step - loss: 2.1753
Epoch 30/30
2000/2000 [==============================] - 229s 115ms/step - loss: 2.1687
```

Outputs:




```
Greedy Search: a dog is running through the snow
Beam Search, K = 3: a brown dog is walking through the snow
Beam Search, K = 5: a brown dog is running through the snow
Beam Search, K = 7: a white dog is walking through the snow
Beam Search, K = 10: a white dog is carrying a stick in the snow
```




```
Greedy: a dog is running through the grass
Beam Search, K = 3: a black and white dog is jumping over a fallen tree
Beam Search, K = 5: a black and white dog is licking its nose while sitting on a bed
Beam Search, K = 7: a brown and white dog is licking its nose while sitting on the grass
```

Model 2:

[22]:

```python
inputs1 = Input(shape=(2048,))
fe1 = Dropout(0.5)(inputs1)
fe2 = Dense(512, activation='relu')(fe1)

inputs2 = Input(shape=(max_length,))
se1 = Embedding(vocab_size, embedding_dim, mask_zero=True)(inputs2)
se2 = Dropout(0.5)(se1)
se3 = LSTM(512)(se2)

decoder1 = add([fe2, se3])
decoder2 = Dense(512, activation='relu')(decoder1)
outputs = Dense(vocab_size, activation='softmax')(decoder2)

model = Model(inputs=[inputs1, inputs2], outputs=outputs)
model.summary()
```

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	[(None, 38)]	0	
input_2 (InputLayer)	[(None, 2048)]	0	
embedding (Embedding)	(None, 38, 200)	332000	input_3[0][0]
dropout (Dropout)	(None, 2048)	0	input_2[0][0]
dropout_1 (Dropout)	(None, 38, 200)	0	embedding[0][0]
dense (Dense)	(None, 512)	1049088	dropout[0][0]
lstm (LSTM)	(None, 512)	1460224	dropout_1[0][0]
add (Add)	(None, 512)	0	dense[0][0] lstm[0][0]
dense_1 (Dense)	(None, 512)	262656	add[0][0]
dense_2 (Dense)	(None, 1660)	851580	dense_1[0][0]

Total params: 3,955,548
Trainable params: 3,955,548
Non-trainable params: 0

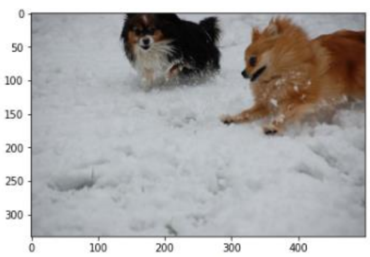
[26]:

```python
from tensorflow.keras.callbacks import EarlyStopping
epochs = 20
batch_size = 10
steps = len(train_descriptions)//batch_size
callback = EarlyStopping(monitor='loss', patience=5)
generator = data_generator(train_descriptions, train_features, wordtoix, max_length, batch_size)
model.fit(generator, epochs=epochs, steps_per_epoch=steps, verbose=1,callbacks=[callback])
```

Draft Session
No Accelerator
Session 0m 12 hours
Disk 3.7GB Max 73.1GB
CPU
CPU 0.00%
RAM 501.9MB Max 16GB

```
Epoch 1/20
600/600 [==============================] - 1221s 2s/step - loss: 3.7789
Epoch 2/20
600/600 [==============================] - 1212s 2s/step - loss: 3.0277
Epoch 3/20
600/600 [==============================] - 1210s 2s/step - loss: 2.7877
Epoch 4/20
600/600 [==============================] - 1207s 2s/step - loss: 2.6346
Epoch 5/20
600/600 [==============================] - 1207s 2s/step - loss: 2.5160
Epoch 6/20
600/600 [==============================] - 1206s 2s/step - loss: 2.4229
Epoch 7/20
600/600 [==============================] - 1207s 2s/step - loss: 2.3382
Epoch 8/20
600/600 [==============================] - 1209s 2s/step - loss: 2.2664
Epoch 9/20
600/600 [==============================] - 1208s 2s/step - loss: 2.1954
Epoch 10/20
600/600 [==============================] - 1207s 2s/step - loss: 2.1389
Epoch 11/20
600/600 [==============================] - 1207s 2s/step - loss: 2.0867
Epoch 12/20
600/600 [==============================] - 1204s 2s/step - loss: 2.0378
Epoch 13/20
600/600 [==============================] - 1202s 2s/step - loss: 2.0001
Epoch 14/20
600/600 [==============================] - 1202s 2s/step - loss: 1.9571
Epoch 15/20
600/600 [==============================] - 1204s 2s/step - loss: 1.9183
Epoch 16/20
600/600 [==============================] - 1210s 2s/step - loss: 1.8861
Epoch 17/20
600/600 [==============================] - 1215s 2s/step - loss: 1.8540
Epoch 18/20
600/600 [==============================] - 1218s 2s/step - loss: 1.8200
Epoch 19/20
600/600 [==============================] - 1215s 2s/step - loss: 1.7955
Epoch 20/20
600/600 [==============================] - 1204s 2s/step - loss: 1.7686
```

[26]: <keras.callbacks.History at 0x7f1510d1d410>



Greedy Search: a brown and white dog is running through the grass
Beam Search, K = 3: a brown and white dog runs through the grass
Beam Search, K = 5: a brown and white dog running through the grass
Beam Search, K = 7: a brown and white dog is running through the grass
Beam Search, K = 10: black and tan dog running in the grass

Greedy: a beagle is resting on a rock
Beam Search, K = 3: a small dog with a ball in its mouth jumps over a large dog
Beam Search, K = 5: a black and brown dog has his mouth open
Beam Search, K = 7: a black and brown dog has his mouth open wide open