Image Captioning with Keras

Submitted By: Sargam Jain

Sjain15@kent.edu

Student ID: 811187404

Contents

Summary:

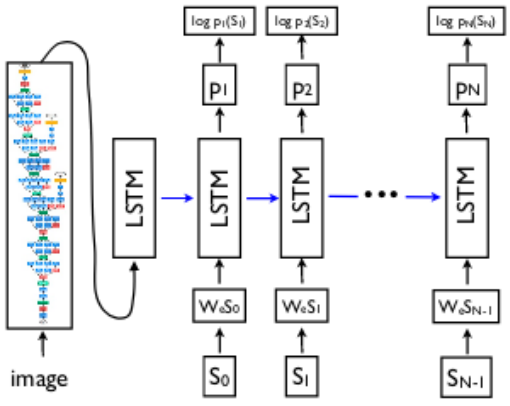
Given an image, the goal is to generate captions that convey key information about it in English. I have approached the problem in supervised setting where I have worked with dataset that has images and corresponding descriptions with them.

Given an image, it's easy for us, as humans, to give a glance to the picture and describe it in our language. But, what about computers? Is it anyway? The caption generation - an auto-generated description of the given image - is a challenging artificial intelligence problem where a textual description must be generated for a given picture. It requires a computer vision approach to understand the content of the image and a language model to let the computer describe the picture with natural language with a strong semantic understanding of the visual scene and all its elements.

Applications on Image Captioning:

- Probably, will be useful in cases/fields where text is most used and with the use of this, we can infer/generate text from images. As in, use the information directly from any image in a textual format automatically.
- Would serve as a huge help for visually impaired people. Lots of applications can be developed in that space.
- Social Media. Platforms like Facebook can infer directly from the image, where you are at (beach, cafe etc.), what you wear (color) and more importantly what you're doing also (in a way).

Recurrent Neural Networks (RNN) are used for varied number of applications including machine translation. The Encoder-Decoder architecture is utilized for such settings where a varied-length input sequence is mapped to the varied-length output sequence. The same network can also be used for image captioning. In image captioning, the core idea is to use CNN as encoder and a normal RNN as decoder.



Problem Statement:

The project is aimed at detecting features of an image and producing a caption that effectively describes it using a combination of different deep neural nets.

The model consists of CNN-Encoder and RNN-Decoder. The CNN-Encoder is used to extract the information of the input image to generate the intermediate representation H, and then use RNN-Decode to gradually decode the H to generate a text description corresponding to the image.

An example is as follows:

black and white dog jumps over bar

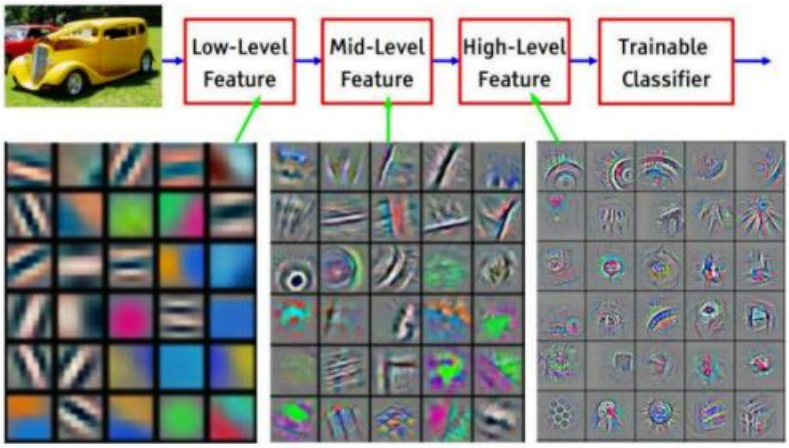
After searching numerous ML applications on internet, I found this problem very convincing. A picture is worth 1000 words. This decade has been the decade of images. Image collection have become extremely large with current camera technology and smart phones advancing to new heights. Image tagging and captioning is one of the famous and challenging problems with many applications. Many products like Google, Facebook, Flicker etc., take use of image tagging, to a great degree of success. The use of captions with images can provide numerous useful functions. Some examples are mentioned below:

- Point out a specific piece of content.
- Explain some icon or graphic for sight-impaired people
- Summarize the meaning of some region. Most searches are done via textual queries, thus there must be a mechanism to link applicable keywords or phrases to images. For blind persons, being able to convey information about the image in another medium would be good for accessibility.

Technique:

Input:

As mentioned in the workflow earlier, the input to our model is [image, part_caption]. We route the image through the CNN to extract features that are characteristic of that image. We shall be using pretrained CNNs.

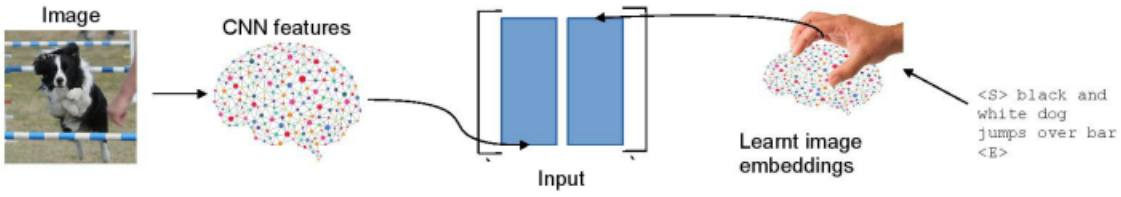


Example representation learnt by a CNN

During training time, I construct a dictionary that maps each word to an index. Similarly, I construct an inverse dictionary that maps each index to a word. This will be helpful during validation/testing. The size of the dictionary will be referred to as VOCAB_SIZE thereafter. All captions in the dataset have length ~20. I have constructed a matrix representation of the caption. This is a MAX_CAP_LENGTH x VOCAB_SIZE, where each row of the matrix is a one hot vector whose i^{th} entry is marked one, where *i* is the index of the word in the dictionary. For example, say the words 'black', 'and', 'white', 'dog', 'jumps', 'over', 'bar' have indices 4,2,5,8,7,3,6 respectively, our matrix will look like this:

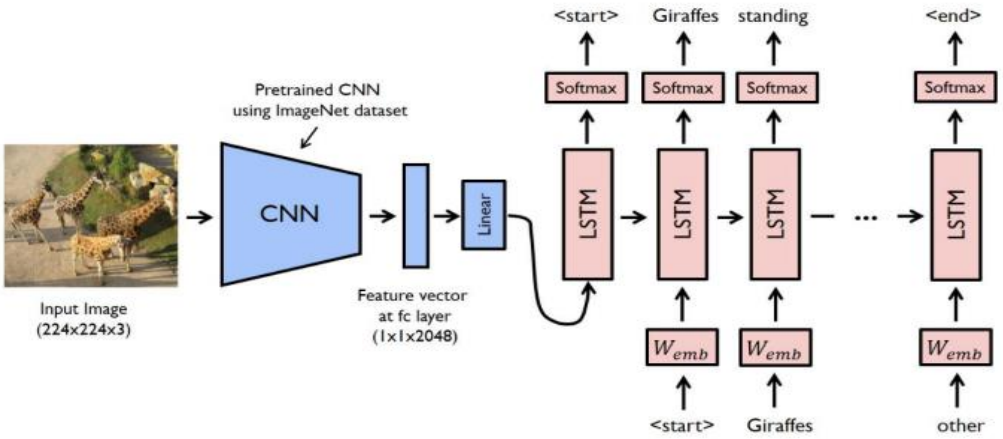
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ - & - & - & - & - & - & - & - & - & \dots & - & - \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ - & - & - & - & - & - & - & - & - & \dots & - & - \\ - & - & - & - & - & - & - & - & - & \dots & - & - \end{bmatrix}$$

The matrix is zero-indexed. Observe that the 0^{th} and the $(n + 1)^{th}$ entry also have 1s at some index. These indices correspond to the start and end tokens respectively. The reason for providing these is that we need to know when the sentence has ended. The start token is needed during testing time, whose details have been given below. The start token and end token will be referenced to as <S> and <E> from now onwards. We will be learning embeddings for the words using a language model. Also note that '-' represents 'nothing'.



To achieve goal, we basically need: data, vector of words, a Convolutional Neural Network to encode our images: a convolutional neural network is a neural network that that uses some mathematical operations called 'Convolutions' instead of general matrix multiplication in at least one layer. It consists of an input and output layers with multiple hidden layers, the hidden layers are a series of convolutional layers that 'convolve' with a multiplication. And finally, a Recurrent neural network as decoder: a recurrent neural network is a neural network in which connections between nodes have also a temporal sequence, this allows it to provide a temporal dynamic behavior.

Image Captioning is the task of describing the content of an image in words. This task lies at the intersection of computer vision and natural language processing. Most image captioning systems use an encoder-decoder framework, where an input image is encoded into an intermediate representation of the information in the image, and then decoded into a descriptive text sequence



Training:

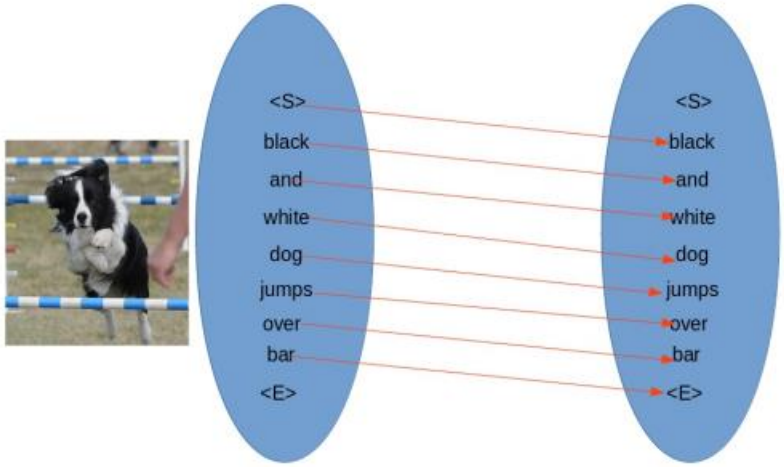
- Words in the captions are represented with an embedding model. Each word in the vocabulary is associated with a fixed-length vector representation, that is, the embedding which is learned during training. We need to encode each word into a fixed sized vector, this process is called Word Embeddings. The word embeddings are the text converted into numbers and there may be different numerical representations of the same text.
- For language modelling we are using Long Short-Term Memory networks and their variants. RNNs are commonly used for sequence modeling tasks such as language modeling and machine translation.
- In the Show and Tell model, the RNN network is trained as a language model conditioned on the image encoding.
- For each word in a caption, we add the image encoding obtained as an output of CNN, with the embedding of word. We treat occurrence of each word in a caption as a time step and send the add vector as input to the RNN.
- RNN uses its hidden state along with the input vector to compute the output at this step. The sequence of outputs obtained are fed through a dense network, which get back the one hot representations. This output is compared with the true output.

Output:

The output is a matrix as well. Essentially it is the same matrix as above, except for the fact that it is shifted up by one row so that each token now corresponds to its next token. The corresponding matrix for the above matrix is:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \\ - & - & - & - & - & - & - & - & - & \dots & - & - \\ - & - & - & - & - & - & - & - & - & \dots & - & - \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ - & - & - & - & - & - & - & - & - & \dots & - & - \\ - & - & - & - & - & - & - & - & - & \dots & - & - \end{bmatrix}$$

For this example, we get the following mapping:



This mapping represents the ground truth. While testing, I have applied a soft-max layer over the outputs of the last layer and take the index which has the greatest value as the next predicted word, given a partial caption.

RNN Model

In this case, the recurrent neural network is used as a text generation model. Our model takes as input, both the image vectors and the partial captions (input layer 1 and 2), then in the embedding layer, every index of the captions is mapped to a 200-dimensional vector. Then we have two dropout layers, to try to avoid over fitting. Then we add some Dense (for image model) and LSTM (For caption model) LSTM, Long Short-Term Memory, is a specialized Recurrent Neural Network that processes not only single data point, like images, but also entire sequences of data (like speech or text).

Experimental Details:

Data:

Experiments were conducted on the following datasets:

1. FLICKR8K: Includes images obtained from the Flickr website. 8000 images with 5 captions for each image. The images do not contain any famous person or place so that the entire image can be learnt based on all the different objects in the image.
2. MICROSOFT COCO: 20000 images with 5 caption per image.
3. IAPR: 20000 images with 1 caption per image

Preprocessing:

Owing to shortage of space on the CSE GPU server, I had to subset the COCO dataset because it is very large. A sample of 30000 images and their corresponding captions were taken.

Now, needed to encode all images in a fixed size (in our case, 299x299), and to give them in input to a convolutional neural network. I've used the Inception Model, a pre-trained network that due to its low error score and efficiency has become one of the most used networks for image classification. The inception model is trained on the ImageNet dataset, with over 15 million labeled-high resolution images. There are four versions of Inception, for this project I've tried the InceptionV3 Model.

Tuning:

Since the model learnt almost everything except the CNN weights, many of the parameters were tuned. A few salient tuning parameters are as follows:

- Dataset: Upon testing with Flickr8k, COCO and IAPR datasets, I have found that the best dataset was indeed COCO, and IAPR was significantly worse than the other two.
- Choice of CNN: From among Inception and VGG16, Inception gave significantly better results. This confirms the fact that Inception has an accuracy of 79%, whereas VGG16 has 71%.
- Use LSTM / GRU: Although it is a known fact that GRUs are better than LSTMs, we decided to test this because it is theoretically believed that LSTMs have more memory than GRU. Since our caption lengths were restricted to 20, LSTM did not show any significant difference. Training with GRU was fast and gave good results.

Conclusion:

This model, using the InceptionV3, and a 30k images dataset perform the image captioning task quite well, but since no model is perfect, basing on the Convolutional neural network used, the captions style changes: for instance, InceptionV3 model, recognize animals very well, while has many problems with people. The InceptionResNetV2, recognize people much better but rarely has some mistakes on animals, also both networks have problems recognizing colors. The captions depend also on the dataset, since the coco dataset has a lot of images about dogs, the dog's captions are almost perfect. Finally, some examples of both correct and wrong captions are below.

Image: dog is running through the grass

Image: man is doing trick on his bicycle

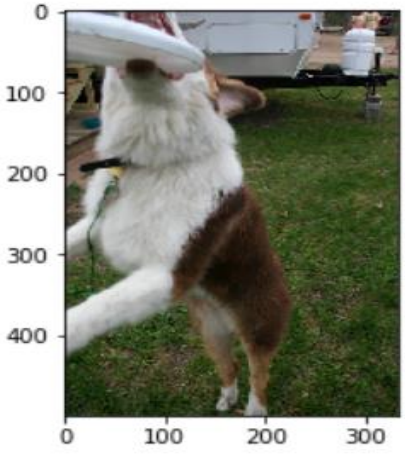


Image: dog is jumping up to catch frisbee

Image: man in blue shirt is driving tractor

Image: man shovels snow off of the road

Image: group of people are sitting at table eating food

Contributions:

I'm able to apply CNN (Encoder) to extract features of image and RNN (Decoder) to generate relevant text for images in my model and successfully generate good captions for the images. The ultimate purpose of Image caption generator is to make users experience better by generating automated captions.

I managed to learn quite a lot about how neural networks work. I managed to learn how to train a language model and a recurrent neural network added with a convolutional neural network. In future, it is possible to include an attention model that will focus on areas of the picture and help classify the images better. Since image captioning is indeed a very difficult task, we have a long way to go till we give the machine sufficient intelligence so that it can understand any image correctly.

Image captioning can contribute well towards below areas:

Self-driving automobiles - Automatic driving is one of the most difficult difficulties and captioning the area around the car can help the self-driving system.

Aid for the blind — We can develop a product for the blind that will lead them on the roads without the need for anyone else's assistance. This may be accomplished by first turning the scene to text, then the text to speech. Both are now well-known Deep Learning applications.

CCTV cameras are already everywhere, but if we can provide appropriate captions in addition to watching the world, we can trigger warnings as soon as criminal behavior is detected at someplace. This is likely to help minimize crime and/or accidents.

Automatic captioning might help Google Image Search become as good as Google Search, because every image could be transformed into a caption first, and then searches could be conducted based on the caption.

References:

[1] Image Captioning: Transforming Objects into Words Lakshminarasimhan Srinivasan1, Dinesh Sreekanthan2, Amutha A.L3

[2] Image Captioning Based on Deep Neural Networks Shuang Liu1, Liang Bai1, a, Yanli Hu1 and Haoran Wang1

[3] Image Captioning with Keras , harsh lamba https://towardsdatascience.com/image-captioning-withkeras-teaching-computers-to-describe-picturesc88a46a311b8

[4] IMAGE CAPTION GENERATOR CNN-LSTM Architecture and Image Captioning Arsh Chowdhry https://blog.clairvoyantsoft.com/image-caption-generator535b8e9a66ac

[5] Image Captioning Based on Deep Neural Networks Shuang Liu1, Liang Bai1,a, Yanli Hu1 and Haoran Wang1

Appendix

```
[ ] from __future__ import absolute_import, division, print_function, unicode_literals
```

```
#!pip install -q tensorflow-gpu==2.0.0-beta1
import tensorflow as tf

# You'll generate plots of attention in order to see which parts of an image
# our model focuses on during captioning
import matplotlib.pyplot as plt

# Scikit-learn includes many helpful utilities
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

import re
import numpy as np
import os
import time
import json
from glob import glob
from PIL import Image
import pickle
```

```
[ ] annotation_zip = tf.keras.utils.get_file('captions.zip',
                                             cache_subdir=os.path.abspath('.'),
                                             origin = 'http://images.cocodataset.org/annotations/annotations_trainval2014.zip',
                                             extract = True)
    annotation_file = os.path.dirname(annotation_zip)+'/annotations/captions_train2014.json'

    name_of_zip = 'train2014.zip'
    if not os.path.exists(os.path.abspath('.') + '/' + name_of_zip):
      image_zip = tf.keras.utils.get_file(name_of_zip,
                                          cache_subdir=os.path.abspath('.'),
                                          origin = 'http://images.cocodataset.org/zips/train2014.zip',
                                          extract = True)
      PATH = os.path.dirname(image_zip)+'/train2014/'
    else:
      PATH = os.path.abspath('.')+'/train2014/'
```

```
[ ] # Read the json file
    with open(annotation_file, 'r') as f:
        annotations = json.load(f)

    # Store captions and image names in vectors
    all_captions = []
    all_img_name_vector = []

    for annot in annotations['annotations']:
        caption = ' ' + annot['caption'] + ' '
        image_id = annot['image_id']
        full_coco_image_path = PATH + 'COCO_train2014_' + '%012d.jpg' % (image_id)

        all_img_name_vector.append(full_coco_image_path)
        all_captions.append(caption)

    # Shuffle captions and image_names together
    # Set a random state
    train_captions, img_name_vector = shuffle(all_captions,
                                              all_img_name_vector,
                                              random_state=1)

    # Selecting the first 30000 captions from the shuffled set
    num_examples = 30000
    train_captions = train_captions[:num_examples]
    img_name_vector = img_name_vector[:num_examples]
```

```
[ ] len(train_captions), len(all_captions)
```

Creating a tf.keras model where the output layer is the last convolutional layer in the InceptionV3 architecture. The shape of the output of this layer is 8x8x2048. Using the last convolutional layer because we are using attention in this example. We don't perform this initialization during training because it could become a bottleneck.

- We forward each image through the network and store the resulting vector in a dictionary (image_name --> feature_vector).
- After all the images are passed through the network, we pickle the dictionary and save it to disk.

We will pre-process each image with InceptionV3 and cache the output to disk. Caching the output in RAM would be faster but also memory intensive, requiring 8 * 8 * 2048 floats per image. At the time of writing, this exceeds the memory limitations of Colab (currently 12GB of memory).

```
[ ]  def load_image(image_path):
         img = tf.io.read_file(image_path)
         img = tf.image.decode_jpeg(img, channels=3)
         img = tf.image.resize(img, (299, 299))
         img = tf.keras.applications.inception_v3.preprocess_input(img)
         return img, image_path
```

```
[ ]  image_model = tf.keras.applications.InceptionV3(include_top=False,
                                                     weights='imagenet')
     new_input = image_model.input
     hidden_layer = image_model.layers[-1].output

     image_features_extract_model = tf.keras.Model(new_input, hidden_layer)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5
87910968/87910968 [==============================] - 0s 0us/step
```

```
[ ]  !pip install -q tqdm
```

```
[ ]  from tqdm import tqdm
```

```
[ ]  for img, path in image_dataset:
```

Preprocess and tokenize the captions

- First, we'll tokenize the captions (for example, by splitting on spaces). This gives us a vocabulary of all the unique words in the data (for example, "surfing", "football", and so on).
- Next, we'll limit the vocabulary size to the top 5,000 words (to save memory). We'll replace all other words with the token "UNK" (unknown).
- We then create word-to-index and index-to-word mappings.
- Finally, we pad all sequences to the same length as the longest one.

```
[ ]  # Get unique images
     encode_train = sorted(set(img_name_vector))

     image_dataset = tf.data.Dataset.from_tensor_slices(encode_train)
     image_dataset = image_dataset.map(
       load_image, num_parallel_calls=tf.data.experimental.AUTOTUNE).batch(16)

     for img, path in image_dataset:
       batch_features = image_features_extract_model(img)
       batch_features = tf.reshape(batch_features,
                                   (batch_features.shape[0], -1, batch_features.shape[3]))

       for bf, p in zip(batch_features, path):
         path_of_feature = p.numpy().decode("utf-8")
         np.save(path_of_feature, bf.numpy())
```

```
[ ]  # Finding the maximum length of any caption in our dataset
     def calc_max_length(tensor):
         return max(len(t) for t in tensor)
```

```
[ ]  # Choosing the top 5000 words from the vocabulary
     top_k = 5000
     tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=top_k,
                                                       oov_token="",
                                                       filters='!"#$%&()*+.,-/:;=?@[\]^_`{|}~ ')
     tokenizer.fit_on_texts(train_captions)
     train_seqs = tokenizer.texts_to_sequences(train_captions)
```

Split the data into training and testing

```
[ ]  tokenizer.word_index[''] = 0
     tokenizer.index_word[0] = ''
```

```
[ ]  # Creating the tokenized vectors
     train_seqs = tokenizer.texts_to_sequences(train_captions)
```

```
[ ]  # Pad each vector to the max_length of the captions
     cap_vector = tf.keras.preprocessing.sequence.pad_sequences(train_seqs, padding='post')
```

```
[ ]  # Calculates the max_length, which is used to store the attention weights
     max_length = calc_max_length(train_seqs)
```

```
[ ]  # Create training and validation sets using an 80-20 split
     img_name_train, img_name_val, cap_train, cap_val = train_test_split(img_name_vector,
                                                                         cap_vector,
                                                                         test_size=0.2,
                                                                         random_state=0)
```

```
[ ]  len(img_name_train), len(cap_train), len(img_name_val), len(cap_val)

     (24000, 24000, 6000, 6000)
```

Create a tf.data dataset for training

```
[ ] BATCH_SIZE = 64
    BUFFER_SIZE = 1000
    embedding_dim = 256
    units = 512
    vocab_size = len(tokenizer.word_index) + 1
    num_steps = len(img_name_train) // BATCH_SIZE
    # Shape of the vector extracted from InceptionV3 is (64, 2048)
    # These two variables represent that vector shape
    features_shape = 2048
    attention_features_shape = 64
```

```
[ ] # Load the numpy files
    def map_func(img_name, cap):
      img_tensor = np.load(img_name.decode('utf-8')+'.npy')
      return img_tensor, cap
```

```
[ ] dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))

    # Use map to load the numpy files in parallel
    dataset = dataset.map(lambda item1, item2: tf.numpy_function(
              map_func, [item1, item2], [tf.float32, tf.int32]),
              num_parallel_calls=tf.data.experimental.AUTOTUNE)

    # Shuffle and batch
    dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
    dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
```

- In this example, we extract the features from the lower convolutional layer of InceptionV3 giving us a vector of shape (8, 8, 2048).
- We squash that to a shape of (64, 2048).
- This vector is then passed through the CNN Encoder (which consists of a single Fully connected layer).
- The RNN (here GRU) attends over the image to predict the next word

```
[ ] class BahdanauAttention(tf.keras.Model):
      def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

      def call(self, features, hidden):
        # features(CNN_encoder output) shape == (batch_size, 64, embedding_dim)

        # hidden shape == (batch_size, hidden_size)
        # hidden_with_time_axis shape == (batch_size, 1, hidden_size)
        hidden_with_time_axis = tf.expand_dims(hidden, 1)

        # score shape == (batch_size, 64, hidden_size)
        score = tf.nn.tanh(self.W1(features) + self.W2(hidden_with_time_axis))

        # attention_weights shape == (batch_size, 64, 1)
        # you get 1 at the last axis because you are applying score to self.V
        attention_weights = tf.nn.softmax(self.V(score), axis=1)

        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * features
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights
```

```
class CNN_Encoder(tf.keras.Model):
    # Since we have already extracted the features and dumped it using pickle
    # This encoder passes those features through a Fully connected layer
    def __init__(self, embedding_dim):
        super(CNN_Encoder, self).__init__()
        # shape after fc == (batch_size, 64, embedding_dim)
        self.fc = tf.keras.layers.Dense(embedding_dim)

    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x
```

```
[ ]
class RNN_Decoder(tf.keras.Model):
  def __init__(self, embedding_dim, units, vocab_size):
    super(RNN_Decoder, self).__init__()
    self.units = units

    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')
    self.fc1 = tf.keras.layers.Dense(self.units)
    self.fc2 = tf.keras.layers.Dense(vocab_size)

    self.attention = BahdanauAttention(self.units)

  def call(self, x, features, hidden):
    # defining attention as a separate model
    context_vector, attention_weights = self.attention(features, hidden)

    # x shape after passing through embedding == (batch_size, 1, embedding_dim)
    x = self.embedding(x)

    # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

    # passing the concatenated vector to the GRU
    output, state = self.gru(x)
```

```
    # shape == (batch_size, max_length, hidden_size)
    x = self.fc1(output)

    # x shape == (batch_size * max_length, hidden_size)
    x = tf.reshape(x, (-1, x.shape[2]))

    # output shape == (batch_size * max_length, vocab)
    x = self.fc2(x)

    return x, state, attention_weights

  def reset_state(self, batch_size):
    return tf.zeros((batch_size, self.units))
```

```
[ ] encoder = CNN_Encoder(embedding_dim)
    decoder = RNN_Decoder(embedding_dim, units, vocab_size)
```

```
[ ] optimizer = tf.keras.optimizers.Adam()
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True, reduction='none')

    def loss_function(real, pred):
      mask = tf.math.logical_not(tf.math.equal(real, 0))
      loss_ = loss_object(real, pred)

      mask = tf.cast(mask, dtype=loss_.dtype)
      loss_ *= mask

      return tf.reduce_mean(loss_)
```

```
[ ] checkpoint_path = "./checkpoints/train"
    ckpt = tf.train.Checkpoint(encoder=encoder,
                               decoder=decoder,
                               optimizer = optimizer)
    ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)
```

```
[ ] start_epoch = 0
    if ckpt_manager.latest_checkpoint:
      start_epoch = int(ckpt_manager.latest_checkpoint.split('-')[-1])
```

```
[ ] # adding this in a separate cell because if you run the training cell
    # many times, the loss_plot array will be reset
    loss_plot = []
```

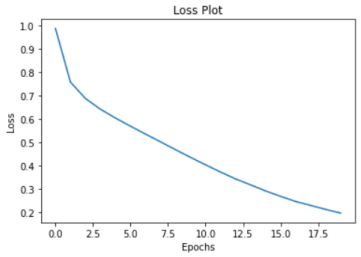
Training

- Extracting the features stored in the respective .npy files and then pass those features through the encoder.
- The encoder output, hidden state (initialized to 0) and the decoder input (which is the start token) is passed to the decoder.
- The decoder returns the predictions and the decoder hidden state.
- The decoder hidden state is then passed back into the model and the predictions are used to calculate the loss.
- Use teacher forcing to decide the next input to the decoder.
- Teacher forcing is the technique where the target word is passed as the next input to the decoder.
- The final step is to calculate the gradients and apply it to the optimizer and backpropagate.

```
[ ]  @tf.function
     def train_step(img_tensor, target):
       loss = 0

       # initializing the hidden state for each batch
       # because the captions are not related from image to image
       hidden = decoder.reset_state(batch_size=target.shape[0])

       dec_input = tf.expand_dims([tokenizer.word_index['']] * BATCH_SIZE, 1)

       with tf.GradientTape() as tape:
           features = encoder(img_tensor)

           for i in range(1, target.shape[1]):
               # passing the features through the decoder
               predictions, hidden, _ = decoder(dec_input, features, hidden)

               loss += loss_function(target[:, i], predictions)

               # using teacher forcing
               dec_input = tf.expand_dims(target[:, i], 1)

       total_loss = (loss / int(target.shape[1]))

       trainable_variables = encoder.trainable_variables + decoder.trainable_variables

       gradients = tape.gradient(loss, trainable_variables)

       optimizer.apply_gradients(zip(gradients, trainable_variables))

       return loss, total_loss
```

```
[ ]  EPOCHS = 20

     for epoch in range(start_epoch, EPOCHS):
         start = time.time()
         total_loss = 0

         for (batch, (img_tensor, target)) in enumerate(dataset):
             batch_loss, t_loss = train_step(img_tensor, target)
             total_loss += t_loss

             if batch % 100 == 0:
                 print ('Epoch {} Batch {} Loss {:.4f}'.format(
                   epoch + 1, batch, batch_loss.numpy() / int(target.shape[1])))
         # storing the epoch end loss value to plot later
         loss_plot.append(total_loss / num_steps)

         if epoch % 5 == 0:
           ckpt_manager.save()

         print ('Epoch {} Loss {:.6f}'.format(epoch + 1,
                                              total_loss/num_steps))
         print ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))
```

```
[ ]  plt.plot(loss_plot)
     plt.xlabel('Epochs')
     plt.ylabel('Loss')
     plt.title('Loss Plot')
     plt.show()
```




```
[ ]  def evaluate(image):
         attention_plot = np.zeros((max_length, attention_features_shape))

         hidden = decoder.reset_state(batch_size=1)

         temp_input = tf.expand_dims(load_image(image)[0], 0)
         img_tensor_val = image_features_extract_model(temp_input)
         img_tensor_val = tf.reshape(img_tensor_val, (img_tensor_val.shape[0], -1, img_tensor_val.shape[3]))

         features = encoder(img_tensor_val)

         dec_input = tf.expand_dims([tokenizer.word_index['']], 0)
         result = []

         for i in range(max_length):
             predictions, hidden, attention_weights = decoder(dec_input, features, hidden)

             attention_plot[i] = tf.reshape(attention_weights, (-1, )).numpy()

             predicted_id = tf.argmax(predictions[0]).numpy()
             result.append(tokenizer.index_word[predicted_id])

             if tokenizer.index_word[predicted_id] == '':
                 return result, attention_plot

             dec_input = tf.expand_dims([predicted_id], 0)

         attention_plot = attention_plot[:len(result), :]
         return result, attention_plot
```

```
[ ]  def plot_attention(image, result, attention_plot):
         temp_image = np.array(Image.open(image))

         fig = plt.figure(figsize=(10, 10))

         len_result = len(result)
         for l in range(len_result):
             temp_att = np.resize(attention_plot[l], (8, 8))
             ax = fig.add_subplot(len_result//2, len_result//2, l+1)
             ax.set_title(result[l])
             img = ax.imshow(temp_image)
             ax.imshow(temp_att, cmap='gray', alpha=0.6, extent=img.get_extent())

         plt.tight_layout()
         plt.show()
```

```
[ ]  # captions on the validation set
     rid = np.random.randint(0, len(img_name_val))
     image = img_name_val[rid]
     real_caption = ' '.join([tokenizer.index_word[i] for i in cap_val[rid] if i not in [0]])
     result, attention_plot = evaluate(image)

     print ('Real Caption:', real_caption)
     print ('Prediction Caption:', ' '.join(result))
     plot_attention(image, result, attention_plot)
     # opening the image
     Image.open(img_name_val[rid])
```

```
[ ]  image_url = 'https://tensorflow.org/images/surf.jpg'
     image_extension = image_url[-4:]
     image_path = tf.keras.utils.get_file('image'+image_extension,
                                          origin=image_url)

     result, attention_plot = evaluate(image_path)
     print ('Prediction Caption:', ' '.join(result))
     plot_attention(image_path, result, attention_plot)
     # opening the image
     Image.open(image_path)
```