eYSIP-2018

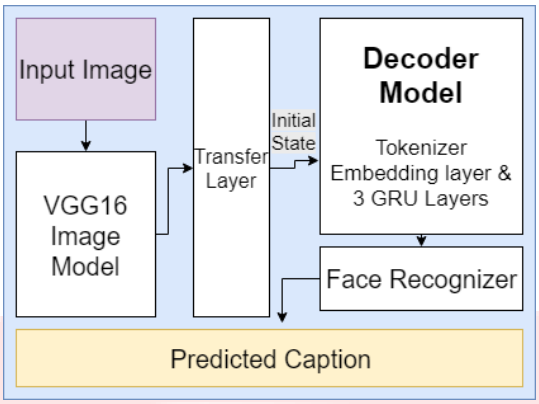
Image Captioning with Face Recognition Tutorial

Authors:
Swapnil Masurekar
Abhishek Sharma

Duration of Internship: 21/05/2018 – 06/07/2018

2018, e-Yantra Publication

0.1 Overview



The network consist of a image classifier, Face recognition model, decoder transfer layer and a decoder network. VGG16 a pre-trained model is used as an image summarizer, the output vector which is vector of 4096 elements are mapped to 512 elements using decoder transfer layer which is taken as the initials state for the GRU layers is the decoder model. The decoder model is then trained on the caption(which is sequence of integer tokens) of the summarized image such that it predicts the next integer token based on the decoder input and the initial-state of the GRU Layers.

0.2 Installation Instructions:

Anaconda a scientific Python distribution for Data Science
Hardware used: *Graphical Processing unit used to train model is NVIDIA GeForce GTX1050Ti*

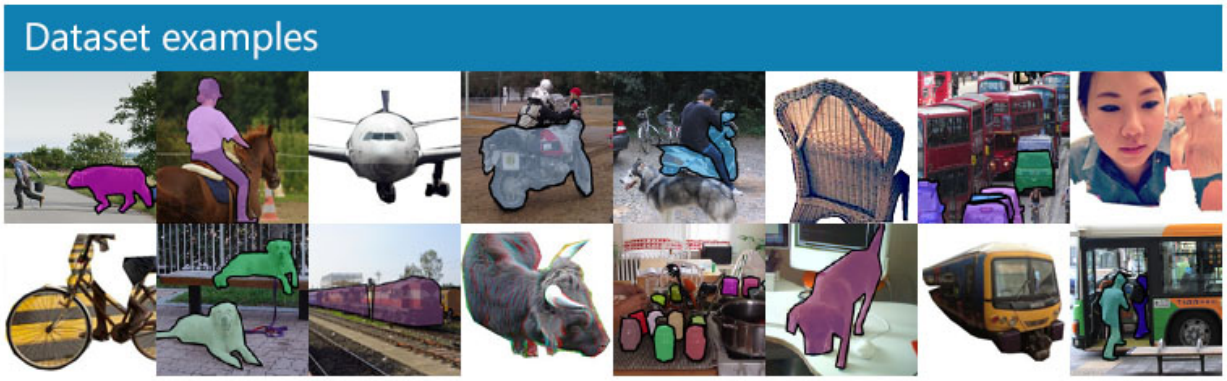
Extra Libraries Required are:

- Tensorflow-GPU v1.1.0
- PIL v5.1.0
- Keras v2.1.5
- Tensorflow-CPU v1.8.0
- Pandas v0.22.0
- OpenCV v3.4.1

For Installation Instructions refer this page: Installation Instructions

0.3 Dataset Description

COCO is a large-scale object detection, segmentation, and captioning dataset.



0.4 Introduction

Machine Translation translate text from one human language to another. It worked by having two Recurrent Neural Networks (RNN), the first called an encoder and the second called a decoder. The first RNN encodes the source-text as a single vector of numbers and the second RNN decodes this vector into the destination-text. The intermediate vector between the encoder and decoder is a kind of summary of the source-text, which is sometimes called a "thought-vector". The reason for using this intermediate summary-vector is to understand the whole source-text before it is being translated. This also allows for the source- and destination-texts to have different lengths.

The image-model recognizes what the image contains and outputs that as a vector of numbers - the "thought-vector" or summary-vector, which is then input to a Recurrent Neural Network that decodes this vector into text

0.5 Importing Libraries

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import sys
import os
from PIL import Image
```

```
from cache import cache
```

0.5.1 Import Model from keras

```
from keras import backend as K
from keras.models import Model
from keras.layers import Input, Dense, GRU, Embedding
from keras.applications import VGG16
from keras.optimizers import RMSprop
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import keras
```

0.6 Load Dataset

Here We will use the COCO data-set which contains many images with text-captions.You can download it from here Training set, Validation set, Training Validation. first unpacked the data set and store all folders into **data/coco/** folder. **Note: These data-files are VERY large! The file for the training-data is 19 GB and the file for the validation-data is 816 MB**

```
import coco
coco.set_data_dir("data/coco/")
```

Get the filenames and captions for the images in the training-set. **Python code snippet:**

```
_, filenames_train, captions_train = coco.load_records(train=
    True)
```

Get the filenames and captions for the images in the validation-set. **Python code snippet:**

```
_, filenames_val, captions_val = coco.load_records(train=False)
```

0.6.1 Resizing Images

This is a helper-function for loading and resizing an image. **Python code snippet:**

```
def load_image(path, size=None):
    """
    Load the image from the given file-path and resize it
    to the given size if not None.
    """

```

```
    # Load the image using PIL.
    img = Image.open(path)

    # Resize image if desired.
    if not size is None:
        img = img.resize(size=size, resample=Image.LANCZOS)

    # Convert image to numpy array.
    img = np.array(img)

    # Scale image-pixels so they fall between 0.0 and 1.0
    img = img / 255.0

    # Convert 2-dim gray-scale array to 3-dim RGB array.
    if (len(img.shape) == 2):
        img = np.repeat(img[:, :, np.newaxis], 3, axis=2)

    return img
```

This is a helper-function for showing an image from the data-set along with its captions.

Python code snippet:

```
def show_image(idx, train):
    """
    Load and plot an image from the training- or validation-set
    with the given index.
    """

    if train:
        # Use an image from the training-set.
        dir = coco.train_dir
        filename = filenames_train[idx]
        captions = captions_train[idx]
    else:
        # Use an image from the validation-set.
        dir = coco.val_dir
        filename = filenames_val[idx]
        captions = captions_val[idx]

    # Path for the image-file.
    path = os.path.join(dir, filename)

    # Print the captions for this image.
    for caption in captions:
        print(caption)

    # Load the image and plot it.
    img = load_image(path)
    plt.imshow(img)
```

```
28     plt.show()
```

It will show an example image and captions from the training-set.
Python code snippet:

```
1 show_image(idx=7, train=True)
```

for output refer to this link

0.6.2 Pre-Trained Image Model (VGG16)

The following creates an instance of the VGG16 model using the Keras API. This automatically downloads the required files if you don't have them already.

The VGG16 model was pre-trained on the ImageNet data-set for classifying images. The VGG16 model contains a convolutional part and a fully-connected (or dense) part which is used for the image classification.

If include_top=True then the whole VGG16 model is downloaded which is about 528 MB. If include_top=False then only the convolutional part of the VGG16 model is downloaded which is just 57 MB. **Python code snippet:**

```
1 image_model = VGG16(include_top=True, weights='imagenet')
2 image_model.summary()
```

for output refer to this link

We will use the output of the layer prior to the final classification-layer which is named fc2. This is a fully-connected (or dense) layer. **Python code snippet:**

```
1 transfer_layer = image_model.get_layer('fc2')
```

We call it the "transfer-layer" because we will transfer its output to another model that creates the image captions.

To do this, first we need to create a new model which has the same input as the original VGG16 model but outputs the transfer-values from the fc2 layer.
Python code snippet:

```
1 image_model_transfer = Model(inputs=image_model.input,
2                               outputs=transfer_layer.output)
```

The model expects input images to be of this size:
Python code snippet:

```
img_size = K.int_shape(image_model.input)[1:3]
img_size
```

Output will be

```
(224, 224)
```

For each input image, the new model will output a vector of transfer-values with this length: **Python code snippet:**

```
transfer_values_size = K.int_shape(transfer_layer.output)[1]
transfer_values_size
```

Output will be

```
4096
```

0.6.3 Processing All Images

We now make functions for processing all images in the data-set using the pre-trained image-model and saving the transfer-values in a cache-file so they can be reloaded quickly.

We effectively create a new data-set of the transfer-values. This is because it takes a long time to process an image in the VGG16 model. We will not be changing all the parameters of the VGG16 model, so every time it processes an image, it gives the exact same result.
Python code snippet:

```
\textbf{Python code snippet:}
\begin{lstlisting}[language=Python]
transfer_values_size = K.int_shape(transfer_layer.output)[1]
transfer_values_size
```

Below is the function for processing the given files using the VGG16-model and returning their transfer-values.
Python code snippet:

```
def process_images(data_dir, filenames, batch_size=32):
    """
    Process all the given files in the given data_dir using the
    pre-trained image-model and return their transfer-values.

    Note that we process the images in batches to save
    memory and improve efficiency on the GPU.
    """

    # Number of images to process.
    num_images = len(filenames)
```

```
12
13      # Pre-allocate input-batch-array for images.
14      shape = (batch_size,) + img_size + (3,)
15      image_batch = np.zeros(shape=shape, dtype=np.float16)
16
17      # Pre-allocate output-array for transfer-values.
18      # Note that we use 16-bit floating-points to save memory.
19      shape = (num_images, transfer_values_size)
20      transfer_values = np.zeros(shape=shape, dtype=np.float16)
21
22      # Initialize index into the filenames.
23      start_index = 0
24
25      # Process batches of image-files.
26      while start_index < num_images:
27          # Print the percentage-progress.
28          print_progress(count=start_index, max_count=num_images)
29
30          # End-index for this batch.
31          end_index = start_index + batch_size
32
33          # Ensure end-index is within bounds.
34          if end_index > num_images:
35              end_index = num_images
36
37          # The last batch may have a different batch-size.
38          current_batch_size = end_index - start_index
39
40          # Load all the images in the batch.
41          for i, filename in enumerate(filenames[start_index:
    end_index]):
42              # Path for the image-file.
43              path = os.path.join(data_dir, filename)
44
45              # Load and resize the image.
46              # This returns the image as a numpy-array.
47              img = load_image(path, size=img_size)
48
49              # Save the image for later use.
50              image_batch[i] = img
51
52          # Use the pre-trained image-model to process the image.
53          # Note that the last batch may have a different size,
54          # so we only use the relevant images.
55          transfer_values_batch = \
56              image_model_transfer.predict(image_batch[0:
    current_batch_size])
57
58          # Save the transfer-values in the pre-allocated array.
```

```
        transfer_values[start_index:end_index] = \
            transfer_values_batch[0:current_batch_size]

        # Increase the index for the next loop-iteration.
        start_index = end_index

    # Print newline.
    print()

    return transfer_values
```

below is the function for processing all images in the training-set. This saves the transfer-values in a cache-file for fast reloading.

Python code snippet:

```
def process_images_train():
    print("Processing {0} images in training-set ...".format(len
    (filenames_train)))

    # Path for the cache-file.
    cache_path = os.path.join(coco.data_dir,
                              "transfer_values_train.pkl")

    # If the cache-file already exists then reload it,
    # otherwise process all images and save their transfer-
    values
    # to the cache-file so it can be reloaded quickly.
    transfer_values = cache(cache_path=cache_path,
                            fn=process_images,
                            data_dir=coco.train_dir,
                            filenames=filenames_train)

    return transfer_values
```

below is the function for processing all images in the validation-set.

Python code snippet:

```
def process_images_val():
    print("Processing {0} images in validation-set ...".format(
    len(filenames_val)))

    # Path for the cache-file.
    cache_path = os.path.join(coco.data_dir, "
    transfer_values_val.pkl")

    # If the cache-file already exists then reload it,
    # otherwise process all images and save their transfer-
    values
    # to the cache-file so it can be reloaded quickly.
    transfer_values = cache(cache_path=cache_path,
```

```
                                fn=process_images,
                                data_dir=coco.val_dir,
                                filenames=filenames_val)

    return transfer_values
```

now we will process all images in the training-set and save the transfer-values to a cache-file

Python code snippet:

```
%%time
transfer_values_train = process_images_train()
print("dtype:", transfer_values_train.dtype)
print("shape:", transfer_values_train.shape)
```

Output will be

```
Processing 118287 images in training-set ...
- Data loaded from cache-file: data/coco/transfer_values_train.
    pkl
dtype: float16
shape: (118287, 4096)
Wall time: 3.64 s
```

Now we will process all images in the validation-set and save the transfer-values to a cache-file.

Python code snippet:

```
%%time
transfer_values_val = process_images_val()
print("dtype:", transfer_values_val.dtype)
print("shape:", transfer_values_val.shape)
```

Output will be

```
Processing 5000 images in validation-set ...
- Data loaded from cache-file: data/coco/transfer_values_val.pkl
dtype: float16
shape: (5000, 4096)
Wall time: 146 ms
```

0.6.4 Tokenizer

Neural Networks cannot work directly on text-data. We use a two-step process to convert text into numbers that can be used in a neural network. The first step is to convert text-words into so-called integer-tokens. The second step is to convert integer-tokens into vectors of floating-point numbers using a so-called embedding-layer

Before we can start processing the text, we first need to mark the beginning and end of each text-sequence with unique words that most likely aren't present in the data.

Python code snippet:

```
%%time
mark_start = 'ssss '
mark_end = ' eeee'
```

Below function wraps all text-strings in the above markers. Note that the captions are a list of list, so we need a nested for-loop to process it. This can be done using so-called list-comprehension in Python.

Python code snippet:

```
def mark_captions(captions_listlist):
    captions_marked = [[mark_start + caption + mark_end
                        for caption in captions_list]
                        for captions_list in captions_listlist]

    return captions_marked
```

Now process all the captions in the training-set and show an example.

Python code snippet:

```
captions_train_marked = mark_captions(captions_train)
captions_train_marked[0]
```

Output will be

```
['ssss Closeup of bins of food that include broccoli and bread.
   eeee',
 'ssss A meal is presented in brightly colored plastic trays.
   eeee',
 'ssss there are containers filled with different kinds of foods
    eeee',
 'ssss Colorful dishes holding meat, vegetables, fruit, and
   bread. eeee',
 'ssss A bunch of trays that have different food. eeee']
```

This Below function converts a list-of-list to a flattened list of captions.

Python code snippet:

```
def flatten(captions_listlist):
    captions_list = [caption
                     for captions_list in captions_listlist
                     for caption in captions_list]

    return captions_list
```

Now use the function to convert all the marked captions from the training set.

Python code snippet:

```
captions_train_flat = flatten(captions_train_marked)
```

Set the maximum number of words in our vocabulary. This means that we will only use e.g. the 10000 most frequent words in the captions from the training-data.

Python code snippet:

```
num_words = 10000
```

We need a few more functions than provided by Keras' Tokenizer-class so we wrap it.

Python code snippet:

```
class TokenizerWrap(Tokenizer):
    """Wrap the Tokenizer-class from Keras with more
    functionality."""

    def __init__(self, texts, num_words=None):
        """
        :param texts: List of strings with the data-set.
        :param num_words: Max number of words to use.
        """

        Tokenizer.__init__(self, num_words=num_words)

        # Create the vocabulary from the texts.
        self.fit_on_texts(texts)

        # Create inverse lookup from integer-tokens to words.
        self.index_to_word = dict(zip(self.word_index.values(),
                                      self.word_index.keys()))

    def token_to_word(self, token):
        """Lookup a single word from an integer-token."""

        word = " " if token == 0 else self.index_to_word[token]
        return word

    def tokens_to_string(self, tokens):
        """Convert a list of integer-tokens to a string."""

        # Create a list of the individual words.
        words = [self.index_to_word[token]
                 for token in tokens
                 if token != 0]

        # Concatenate the words to a single string
        # with space between all the words.
        text = " ".join(words)

```

```
        return text

    def captions_to_tokens(self, captions_listlist):
        """
        Convert a list-of-list with text-captions to
        a list-of-list of integer-tokens.
        """

        # Note that text_to_sequences() takes a list of texts.
        tokens = [self.texts_to_sequences(captions_list)
                  for captions_list in captions_listlist]

        return tokens
```

Now create a tokenizer using all the captions in the training-data. Note that we use the flattened list of captions to create the tokenizer because it cannot take a list-of-lists.

Python code snippet:

```
%%time
tokenizer = TokenizerWrap(texts=captions_train_flat,
                          num_words=num_words)
```

Here we will get the integer-token for the start-marker (the word "ssss"). We will need this further below.

Python code snippet:

```
%%time
token_start = tokenizer.word_index[mark_start.strip()]
token_start
```

Output will be

```
2
```

Here we will Get the integer-token for the end-marker (the word "eeee").

Python code snippet:

```
%%time
token_end = tokenizer.word_index[mark_end.strip()]
token_end
```

Output will be

```
3
```

Convert all the captions from the training-set to sequences of integer-tokens. We get a list-of-list as a result

Python code snippet:

```
%%time
tokens_train = tokenizer.captions_to_tokens(
    captions_train_marked)
```

Here is an example of the integer-tokens for the captions of the first image in the training-set:

Python code snippet:

```
tokens_train[0]
```

Output will be

```
[[2, 841, 5, 2864, 5, 61, 26, 1984, 238, 9, 433, 3],
 [2, 1, 429, 10, 3310, 7, 1025, 390, 501, 1110, 3],
 [2, 63, 19, 993, 143, 8, 190, 958, 5, 743, 3],
 [2, 299, 725, 25, 343, 208, 264, 9, 433, 3],
 [2, 1, 170, 5, 1110, 26, 446, 190, 61, 3]]
```

And below these are the corresponding text-captions:

Python code snippet:

```
captions_train_marked[0]
```

Output will be

```
['ssss Closeup of bins of food that include broccoli and bread.
   eeee',
 'ssss A meal is presented in brightly colored plastic trays.
   eeee',
 'ssss there are containers filled with different kinds of foods
    eeee',
 'ssss Colorful dishes holding meat, vegetables, fruit, and
   bread. eeee',
 'ssss A bunch of trays that have different food. eeee']
```

0.6.5 Data Generator

Each image in the training-set has at least 5 captions describing the contents of the image. The neural network will be trained with batches of transfer-values for the images and sequences of integer-tokens for the captions. If we were to have matching numpy arrays for the training-set, we would either have to only use a single caption for each image and ignore the rest of this valuable data, or we would have to repeat the image transfer-values for each of the captions, which would waste a lot of memory.

A better solution is to create a custom data-generator for Keras that will create a batch of data with randomly selected transfer-values and token-sequences.

This below function returns a list of random token-sequences for the images with the given indices in the training-set.

Python code snippet:

```
def get_random_caption_tokens(idx):
    """
    Given a list of indices for images in the training-set,
    select a token-sequence for a random caption,
    and return a list of all these token-sequences.
    """

    # Initialize an empty list for the results.
    result = []

    # For each of the indices.
    for i in idx:
        # The index i points to an image in the training-set.
        # Each image in the training-set has at least 5 captions
        # which have been converted to tokens in tokens_train.
        # We want to select one of these token-sequences at
    random.

        # Get a random index for a token-sequence.
        j = np.random.choice(len(tokens_train[i]))

        # Get the j'th token-sequence for image i.
        tokens = tokens_train[i][j]

        # Add this token-sequence to the list of results.
        result.append(tokens)

    return result
```

This generator function creates random batches of training-data for use in training the neural network.

Python code snippet:

```
def batch_generator(batch_size):
    """
    Generator function for creating random batches of training-
    data.

    Note that it selects the data completely randomly for each
    batch, corresponding to sampling of the training-set with
    replacement. This means it is possible to sample the same
    data multiple times within a single epoch - and it is also
    possible that some data is not sampled at all within an
    epoch.
    However, all the data should be unique within a single batch
    .
    """

    # Infinite loop.
    while True:
```

```
15        # Get a list of random indices for images in the
      training-set.
16        idx = np.random.randint(num_images_train,
17                                size=batch_size)
18
19        # Get the pre-computed transfer-values for those images.
20        # These are the outputs of the pre-trained image-model.
21        transfer_values = transfer_values_train[idx]
22
23        # For each of the randomly chosen images there are
24        # at least 5 captions describing the contents of the
      image.
25        # Select one of those captions at random and get the
26        # associated sequence of integer-tokens.
27        tokens = get_random_caption_tokens(idx)
28
29        # Count the number of tokens in all these token-
      sequences.
30        num_tokens = [len(t) for t in tokens]
31
32        # Max number of tokens.
33        max_tokens = np.max(num_tokens)
34
35        # Pad all the other token-sequences with zeros
36        # so they all have the same length and can be
37        # input to the neural network as a numpy array.
38        tokens_padded = pad_sequences(tokens,
39                                      maxlen=max_tokens,
40                                      padding='post',
41                                      truncating='post')
42
43        # Further prepare the token-sequences.
44        # The decoder-part of the neural network
45        # will try to map the token-sequences to
46        # themselves shifted one time-step.
47        decoder_input_data = tokens_padded[:, 0:-1]
48        decoder_output_data = tokens_padded[:, 1:]
49
50        # Dict for the input-data. Because we have
51        # several inputs, we use a named dict to
52        # ensure that the data is assigned correctly.
53        x_data = \
54        {
55            'decoder_input': decoder_input_data,
56            'transfer_values_input': transfer_values
57        }
58
59        # Dict for the output-data.
60        y_data = \
```

```
        {
            'decoder_output': decoder_output_data
        }

        yield (x_data, y_data)
```

Set the batch-size used during training. but this also requires a lot of momory on the GPU. You may have to lower this number if the training runs out of memory.

Python code snippet:

```
batch_size = 256
```

Create an instance of the data-generator.

Python code snippet:

```
generator = batch_generator(batch_size=batch_size)
```

Now test the data-generator by creating a batch of data.

Python code snippet:

```
batch = next(generator)
batch_x = batch[0]
batch_y = batch[1]
```

Example of the transfer-values for the first image in the batch.

Python code snippet:

```
batch_x['transfer_values_input'][0]
```

Outout will be:

```
array([ 0.        ,  0.        ,  1.140625  , ...,  0.        ,
        0.        ,  0.68798828], dtype=float16)
```

Example of the token-sequence for the first image in the batch. This is the input to the decoder-part of the neural network.

Python code snippet:

```
batch_x['decoder_input'][0]
```

Outout will be:

```

array([  2,   1, 185, 102,   7,   1,  21, 3597,   4,   1,  71,
         3,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0])
```

Now below is the token-sequence for the output of the decoder. Note how it is the same as the sequence above, except it is shifted one time-step.

Python code snippet:

```
batch_y['decoder_output'][0]
```

Outout will be:

```
array([   1,  185,  102,    7,    1,   21, 3597,    4,    1,
       71,    3,
          0,    0,    0,    0,    0,    0,    0,    0,    0,
       0,    0,
          0,    0,    0,    0,    0,    0,    0])
```

0.6.6 Steps Per Epoch

One epoch is a complete processing of the training-set. We would like to process each image and caption pair only once per epoch. However, because each batch is chosen completely at random in the above batch-generator, it is possible that an image occurs in multiple batches within a single epoch, and it is possible that some images may not occur in any batch at all within a single epoch.

Nevertheless, we still use the concept of an 'epoch' to measure approximately how many iterations of the training-data we have processed. But the data-generator will generate batches for eternity, so we need to manually calculate the approximate number of batches required per epoch.

Below is the number of captions for each image in the training-set.
Python code snippet:

```
num_captions_train = [len(captions) for captions in
    captions_train]
```

Below is the total number of captions in the training-set.
Python code snippet:

```
total_num_captions_train = np.sum(num_captions_train)
```

Below is the approximate number of batches required per epoch, if we want to process each caption and image pair once per epoch.
Python code snippet:

```
steps_per_epoch = int(total_num_captions_train / batch_size)
steps_per_epoch
```

Outout will be:

```
2311
```

0.6.7 Create the Recurrent Neural Network

We will now create the Recurrent Neural Network (RNN) that will be trained to map the vectors with transfer-values from the image-recognition model into sequences of integer-tokens that can be converted into text. We call this neural network for the 'decoder' as it is almost identical to the decoder when doing Machine Translation

we are using the functional model from Keras to build this neural network, because it allows more flexibility in how the neural network can be connected. This means we have split the network construction into two parts: (1) Creation of all the layers that are not yet connected, and (2) a function that connects all these layers.

Below is the decoder consists of 3 GRU layers whose internal state-sizes are: **Python code snippet:**

```
state_size = 512
```

Below embedding-layer converts integer-tokens into vectors of this length:
Python code snippet:

```
embedding_size = 128
```

Below is the inputs transfer-values to the decoder:
Python code snippet:

```
transfer_values_input = Input(shape=(transfer_values_size,),
                              name='transfer_values_input')
```

Here we want to use the transfer-values to initialize the internal states of the GRU units. This informs the GRU units of the contents of the images. The transfer-values are vectors of length 4096 but the size of the internal states of the GRU units are only 512, so we use a fully-connected layer to map the vectors from 4096 to 512 elements.

Note that we use a tanh activation function to limit the output of the mapping between -1 and 1, otherwise this does not seem to work.
Python code snippet:

```
decoder_transfer_map = Dense(state_size,
                             activation='tanh',
                             name='decoder_transfer_map')
```

Below is the input for token-sequences to the decoder. Using None in the shape means that the token-sequences can have arbitrary lengths.
Python code snippet:

```
decoder_input = Input(shape=(None, ), name='decoder_input')
```

Below is the embedding-layer which converts sequences of integer-tokens to sequences of vectors.

Python code snippet:

```
decoder_embedding = Embedding(input_dim=num_words,
                              output_dim=embedding_size,
                              name='decoder_embedding')
```

This creates the 3 GRU layers of the decoder. Note that they all return sequences because we ultimately want to output a sequence of integer-tokens that can be converted into a text-sequence.

Python code snippet:

```
decoder_gru1 = GRU(state_size, name='decoder_gru1',
                   return_sequences=True)
decoder_gru2 = GRU(state_size, name='decoder_gru2',
                   return_sequences=True)
decoder_gru3 = GRU(state_size, name='decoder_gru3',
                   return_sequences=True)
```

The GRU layers output a tensor with shape [batch_size, sequence_length, state_size], where each "word" is encoded as a vector of length state_size. We need to convert this into sequences of integer-tokens that can be interpreted as words from our vocabulary.

One way of doing this is to convert the GRU output to a one-hot encoded array. It works but it is extremely wasteful, because for a vocabulary of e.g. 10000 words we need a vector with 10000 elements, so we can select the index of the highest element to be the integer-token.

Note that the activation-function is set to linear instead of softmax as we would normally use for one-hot encoded outputs, because there is apparently a bug in Keras so we need to make our own loss-function, as described in detail further below.

Python code snippet:

```
decoder_dense = Dense(num_words,
                      activation='linear',
                      name='decoder_output')
```

0.6.8 Connect and Create the Training Model

The decoder is built using the functional API of Keras, which allows more flexibility in connecting the layers e.g. to have multiple inputs. This is useful e.g. if you want to connect the image-model directly with the decoder

instead of using pre-calculated transfer-values.

This below function connects all the layers of the decoder to some input of transfer-values.

Python code snippet:

```
def connect_decoder(transfer_values):
    # Map the transfer-values so the dimensionality matches
    # the internal state of the GRU layers. This means
    # we can use the mapped transfer-values as the initial state
    # of the GRU layers.
    initial_state = decoder_transfer_map(transfer_values)

    # Start the decoder-network with its input-layer.
    net = decoder_input

    # Connect the embedding-layer.
    net = decoder_embedding(net)

    # Connect all the GRU layers.
    net = decoder_gru1(net, initial_state=initial_state)
    net = decoder_gru2(net, initial_state=initial_state)
    net = decoder_gru3(net, initial_state=initial_state)

    # Connect the final dense layer that converts to
    # one-hot encoded arrays.
    decoder_output = decoder_dense(net)

    return decoder_output
```

Now we will connect and create the model used for training. This takes as input transfer-values and sequences of integer-tokens and outputs sequences of one-hot encoded arrays that can be converted into integer-tokens.

Python code snippet:

```
decoder_output = connect_decoder(transfer_values=
    transfer_values_input)

decoder_model = Model(inputs=[transfer_values_input,
    decoder_input],
                      outputs=[decoder_output])
```

0.6.9 Loss Function

The output of the decoder is a sequence of one-hot encoded arrays. In order to train the decoder we need to supply the one-hot encoded arrays that we desire to see on the decoder's output, and then use a loss-function like cross-

entropy to train the decoder to produce this desired output.

Keras supports some weighting of loss-values across the batch but the semantics are unclear so to be sure that we calculate the loss-function across the entire batch and across the entire sequences, we manually calculate the loss average.

Python code snippet:

```
def sparse_cross_entropy(y_true, y_pred):
    """
    Calculate the cross-entropy loss between y_true and y_pred.

    y_true is a 2-rank tensor with the desired output.
    The shape is [batch_size, sequence_length] and it
    contains sequences of integer-tokens.

    y_pred is the decoder's output which is a 3-rank tensor
    with shape [batch_size, sequence_length, num_words]
    so that for each sequence in the batch there is a one-hot
    encoded array of length num_words.
    """

    # Calculate the loss. This outputs a
    # 2-rank tensor of shape [batch_size, sequence_length]
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels
   =y_true, logits=y_pred)

    # Keras may reduce this across the first axis (the batch)
    # but the semantics are unclear, so to be sure we use
    # the loss across the entire 2-rank tensor, we reduce it
    # to a single scalar with the mean function.
    loss_mean = tf.reduce_mean(loss)

    return loss_mean
```

0.6.10 Compile the Training Model

for compiling the taining model we have used RMSprop as given below.

Python code snippet:

```
optimizer = RMSprop(lr=1e-3)
```

Below we have created a placeholder variable for the decoder's output. The shape is set to (None, None) which means the batch can have an arbitrary number of sequences, which can have an arbitrary number of integer-tokens.

Python code snippet:

```
decoder_target = tf.placeholder(dtype='int32', shape=(None, None
    ))
```

We can now compile the model using our custom loss-function as given below.
Python code snippet:

```
decoder_model.compile(optimizer=optimizer,
                      loss=sparse_cross_entropy,
                      target_tensors=[decoder_target])
```

0.6.11 Callback Functions

During training we want to save checkpoints and log the progress to TensorBoard so we create the appropriate callbacks for Keras. Below is the callback for writing checkpoints during training.
Python code snippet:

```
path_checkpoint = '22_checkpoint.keras'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      verbose=1,
                                      save_weights_only=True)
```

Below is the callback for writing the TensorBoard log during training.
Python code snippet:

```
callback_tensorboard = TensorBoard(log_dir='./22_logs/',
                                   histogram_freq=0,
                                   write_graph=False)
callbacks = [callback_checkpoint, callback_tensorboard]
```

0.6.12 Load Checkpoint

we can reload the last saved checkpoint so we don't have to train the model every time we want to use it.
Python code snippet:

```
try:
    decoder_model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

0.6.13 Train the Model

Now we will train the decoder so it can map transfer-values from the image-model to sequences of integer-tokens for the captions of the images.It took

around 5 hours for 20 epoch over Nvidia gtx 1050ti gpu
Python code snippet:

```
try:
    decoder_model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

Face Recognizer:

The face recognizer module includes the following steps for identifying faces and substituting the human-like words detected in captions by name of the predominant person recognized in the image.

- **Detecting Faces:** As face needs to be detected from a image and not video, we use cascade classifier in this case for detecting faces. Cascade classifier seemed to work efficiently. The detected faces are then cropped for further processing.
- **Normalize intensity:** This is done by equalizing histogram of a gray-scaled image.
- **Resizing image:** Image is resized to standard size of 50x50 numpy array.
- **Building local Dataset:** Preprocessing is done on all the images from live video feed and 200 normalized images of the subject are saved in the directory.
- **Train Eigen face recognizer:**Eigen face recognizer is based on the principle component analysis of the data. The model is trained on local dataset of people.
- **Prediction:** After the prediction of correct face the human-like words detected in captions are substituted by name of the predominant person recognized in the image.

For understanding the face recognition system in detail kindly go through this link: Face Recognition
For importing this module the code snippet is, **Python code snippet:**

```
import face_recognizer
```

0.6.14 Generate Captions

This function loads an image and generates a caption using the model we have trained.
Python code snippet:

```
def generate_caption(image_path, max_tokens=30):
    """
    Generate a caption for the image in the given path.
    The caption is limited to the given number of tokens (words).
    """

    # Load and resize the image.
    image = load_image(image_path, size=img_size)

    # Expand the 3-dim numpy array to 4-dim
    # because the image-model expects a whole batch as input,
    # so we give it a batch with just one image.
    image_batch = np.expand_dims(image, axis=0)

    # Process the image with the pre-trained image-model
    # to get the transfer-values.
    transfer_values = image_model_transfer.predict(image_batch)

    # Pre-allocate the 2-dim array used as input to the decoder.
    # This holds just a single sequence of integer-tokens,
    # but the decoder-model expects a batch of sequences.
    shape = (1, max_tokens)
    decoder_input_data = np.zeros(shape=shape, dtype=np.int)

    # The first input-token is the special start-token for 'ssss'.
    token_int = token_start

    # Initialize an empty output-text.
    output_text = ''

    # Initialize the number of tokens we have processed.
    count_tokens = 0

    # While we haven't sampled the special end-token for ' eeee'
    # and we haven't processed the max number of tokens.
    while token_int != token_end and count_tokens < max_tokens:
        # Update the input-sequence to the decoder
        # with the last token that was sampled.
        # In the first iteration this will set the
        # first element to the start-token.
        decoder_input_data[0, count_tokens] = token_int

        # Wrap the input-data in a dict for clarity and safety,
        # so we are sure we input the data in the right order.
        x_data = \
        {
            'transfer_values_input': transfer_values,
```

```
            'decoder_input': decoder_input_data
        }

        # Note that we input the entire sequence of tokens
        # to the decoder. This wastes a lot of computation
        # because we are only interested in the last input
        # and output. We could modify the code to return
        # the GRU-states when calling predict() and then
        # feeding these GRU-states as well the next time
        # we call predict(), but it would make the code
        # much more complicated.

        # Input this data to the decoder and get the predicted
    output.
        decoder_output = decoder_model.predict(x_data)


        # Get the last predicted token as a one-hot encoded
    array.
        # Note that this is not limited by softmax, but we just
        # need the index of the largest element so it doesn't
    matter.

        token_onehot = decoder_output[0, count_tokens, :]

        # Convert to an integer-token.
        token_int = np.argmax(token_onehot)

        # Lookup the word corresponding to this integer-token.
        sampled_word = tokenizer.token_to_word(token_int)

        # Append the word to the output-text.
        output_text += " " + sampled_word

        # Increment the token-counter.
        count_tokens += 1

    # This is the sequence of tokens output by the decoder.
    output_tokens = decoder_input_data[0]

    # Plot the image.
    plt.imshow(image)
    plt.show()

    # Print the predicted caption.
    output_text = face_recognizer.generate_caption_on_face(
    output_text, image_path)
    print("Predicted caption:")
    output_text_temp = output_text.split()
```

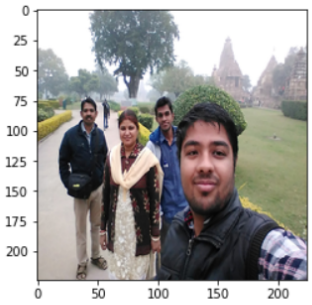
```
    if(output_text_temp[len(output_text_temp)-1]=="eeee"):
        output_text=""
        for i in range(len(output_text_temp)-1):
            output_text = output_text+" "+output_text_temp[i]
    print(output_text)
    print()
```

Below is the function for loading an image from the COCO data-set and printing the true captions as well as the predicted caption.

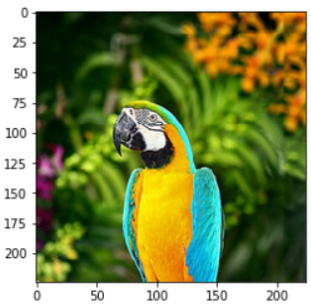
Python code snippet:

```
def generate_caption_coco(idx, train=False):
    """
    Generate a caption for an image in the COCO data-set.
    Use the image with the given index in either the
    training-set (train=True) or validation-set (train=False).
    """

    if train:
        # Use image and captions from the training-set.
        data_dir = coco.train_dir
        filename = filenames_train[idx]
        captions = captions_train[idx]
    else:
        # Use image and captions from the validation-set.
        data_dir = coco.val_dir
        filename = filenames_val[idx]
        captions = captions_val[idx]

    # Path for the image-file.
    path = os.path.join(data_dir, filename)

    # Use the model to generate a caption of the image.
    generate_caption(image_path=path)

    # Print the true captions from the data-set.
    print("True captions:")
    for caption in captions:
        print(caption)
```

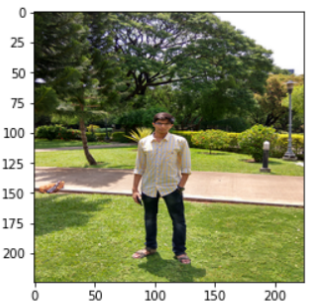
Now we will try this on a picture from the training-set that the model has been trained on. In some cases the generated caption is actually better than the human-generated captions. The example of the output is as shown, as you can see that the faces are correctly identified and the correct words are substituted by the person's name in the caption.



```
Predicted caption:
 Abhishek and a woman are standing
```



```
Predicted caption:
 a bird sitting on a tree branch
 with a blurry background
```



```
Predicted caption:
 Swapnil is standing in a field
```

Useful ML Resources:

- Text Documents
    1. Machine Learning and Algorithms Here you will get the brief idea about ML and Its basic learning algorithm
    2. Basic Terminology related to ML Here you will learn about the basic terminologies and jargons used in ML and Neural Network
    3. Artificial Neural network A Quick introduction to ANN
    4. Convolution Neural network Introduction to Architecture of CNN
- For video tutorial over machine learning refer to following links
    1. Machine learning course by Andrew ng - Click here (enroll for free)
    2. Machine learning A-Z Hands on python in data science - click here(refer tutorials for Python only)
    3. Recurrent Neural Network by Andrew ng - Click here(refer video till LSTM only)
    4. Deep learning by Andrew ng - Click here
    5. Convolution Neural Network by Andrew ng - Click here