

CS6301 MACHINE LEARNING – MINI PROJECT

SRIHARI. S

T.K.S. ARUNACHALAM

2018103601

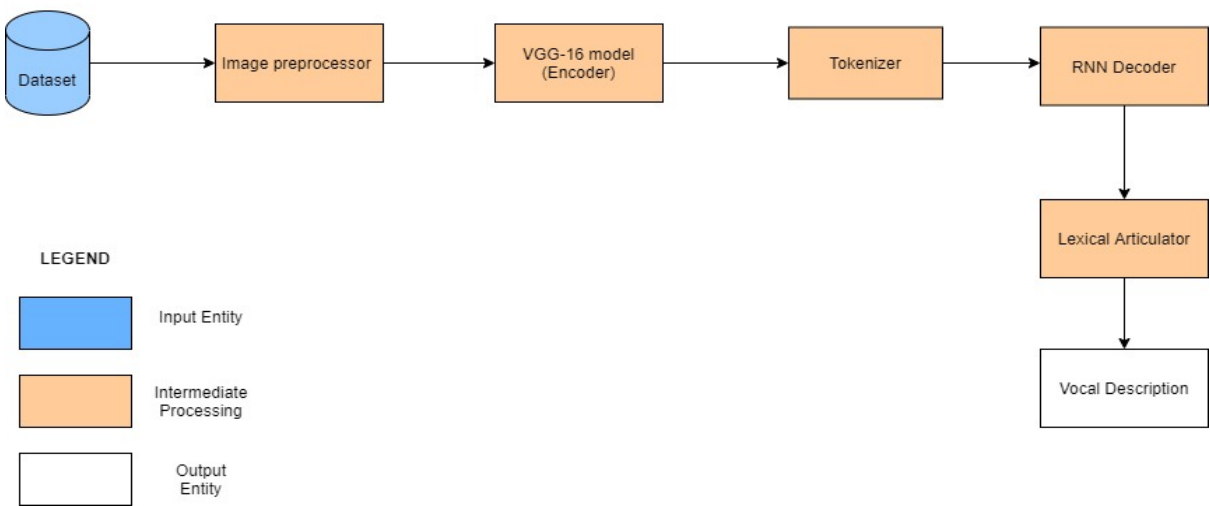
2018103616

IMAGE INSCRIPTION AND INTONATION - A NEURAL NETWORK
APPROACH

100% IMPLEMENTATION UPDATE DOCUMENTATION

DATASET USED: MS-COCO Dataset (<https://cocodataset.org/#download>)


IMAGE INSCRIPTION AND INTONATION - BLOCK DIAGRAM



IMPLEMENTATION PROGRESS	
Completed Modules	Image Pre-processor VGG-16 Model (Encoder) Tokenizer RNN Decoder Lexical Articulator




INDIVIDUAL CONTRIBUTION	
SRIHARIS	ARUNACHALAM.T.K.S.
Caption Pre-processing	Image Pre-processing
Transfer Learning on the dataset using VGG16	Incorporating PyCoCo Tools for dealing with the data
Vocabulary construction	Embeddings productions
Fabrication of the data generator for training the model	Formulate the Recurrent Network
Train the model following Algorithm-1 for 325 epochs (10 hours/day)	Train the model following Algorithm-2 for 325 epochs (10 hours/day)
Formulation of the Lexical Articulator	Measurement of performance using 5 metrics

MODULE WISE TEST CASES

MODULE 1: PRE-PROCESSOR		
INPUT	SEPARATED IMAGE	SEPARATED CAPTIONS
<pre>{ "license": 5, "file_name": "COCO_train2014_000000057870.jpg", "coco_url": "http://images.cocodataset.org/train2014/COCO_train2014_000000000025.jpg", "height": 480, "width": 640, "date_captured": "2013-11-14 16:28:13", "id": 57870 }</pre>		<pre>['A giraffe eating food from the top of the tree.', 'A giraffe standing up nearby a tree ', 'A giraffe mother with its baby in the forest.', 'Two giraffes standing in a tree filled area.', 'A giraffe standing next to a forest filled with trees.']</pre>

<pre>{ "license": 5, "file_name": "COCO_train2014_000000384029.jpg", "coco_url": "http://images.cocodataset.org/val2014/COCO_val2014_000000003014.jpg", "height": 429, "width": 640, "date_captured": "2013-11-14", "16:29:45", "id": 384029 }</pre>		<p>['Three-quarters of a meat-lovers pizza with mushrooms with drink', 'A partially eaten pizza is sitting beside a soda.', 'A table with a partially eaten pizza and blue canned beverage on it.', 'A close up of a pizza and a drink on a table.', 'A pizza with cherry tomatoes has a piece taken out.']</p>
<pre>{ "license": 1, "file_name": "COCO_train2014_000000222016.jpg", "coco_url": "http://images.cocodataset.org/train2014/COCO_train2014_000000000404.jpg", "height": 640, "width": 480, "date_captured": "2013-11-14", "16:37:59", "id": 222016 }</pre>		<p>['a couple of boats that are in some water', 'A pair of boats docked at a pier is shown.', 'Three boats docked in still water with clouds in the sky. ', 'Three boats are docked together on the cloudy day.', 'some colorful boats sitting next to a dock ']</p>
<pre>{ "license": 3, "file_name": "COCO_val2014_000000391895.jpg", "coco_url": "http://images.cocodataset.org/train2014/COCO_train2014_000000000089.jpg", "height": 360, "width": 640, "date_captured": "2013-11-14", "11:18:45", "id": 391895 }</pre>		<p>['An oven with a stove on top of it in a kitchen.', 'A stove with a lighted hood in the kitchen.', 'A small light is on above the polished stove top.', 'Smooth top stove with exhaust fan that has light turned on.', 'A stove top is cleaned with a set of knives on the wall.']</p>
<pre>{ "license": 4, "file_name": "COCO_val2014_000000522418.jpg", "coco_url": "http://images.cocodataset.org/val2014/COCO_val2014_000000000073.jpg", "height": 480, "width": 640, "date_captured": "2013-11-14", "11:38:44", "id": 522418 }</pre>		<p>['A motorcycle parked in a parking space next to another motorcycle.', 'An old motorcycle parked beside other motorcycles with a brown leather seat.', 'Motorcycle parked in the parking lot of asphalt.', 'A close up view of a motorized bicycle, sitting in a rack. ', 'The back tire of an old style motorcycle is resting in a metal stand. ']</p>

MODULE 2: ENCODER


INPUT	OUTPUT – TRANSFER VALUES
	[0. 0. 1.556 ... 0. 0. 0.4502] Dimensions: (4096,)
	[0. 0. 1.378 ... 0. 0. 0.5244] Dimensions: (4096,)
	[0. 0.0756 1.33 ... 0. 0. 0.532] Dimensions: (4096,)

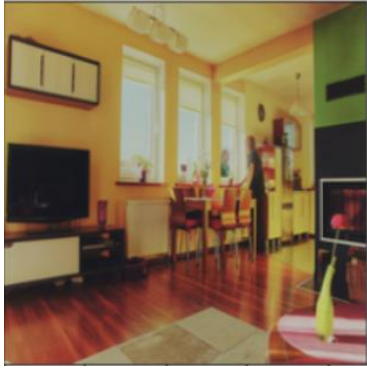
MODULE 3: TOKENIZER

INPUT	INTERMEDIATE OUTPUT	OUTPUT SEQUENCE	EMBEDDINGS
'Closeup of bins of food that include broccoli and bread.'	'ssss Closeup of bins of food that include broccoli and bread. eeee'	[2, 844, 5, 2845, 5, 60, 25, 1933, 248, 9, 438, 3]	[[0.41177 , - 2.223 , -1.0756 , -1.0783 ,]...[0.15155 , 0.78321 , -

			0.91241]]
'A giraffe eating food from the top of the tree.'	'ssss A giraffe eating food from the top of the tree. eeee'	[2, 1, 117, 108, 60, 96, 6, 32, 5, 6, 133, 3]	[[0.50451 , 0.68607 , - 0.59517,...[0.6 0046,-0.13498,.. -0.022801]]
'White vase with different colored flowers sitting inside of it. '	'ssss White vase with different colored flowers sitting inside of it. eeee'	[2, 21, 202, 8, 191, 395, 200, 13, 159, 5, 30, 3]	[[0.32157 , 0.45894 , - 0.32014,...[0.3 2015,-0.23156,.. -0.30215]]
'A cat is lying on its back in a man's lap'	ssss A cat is lying on its back in a man's lap. eeee	[2, 1, 50, 10, 370, 4, 154, 163, 7, 1, 1248, 584, 3]	[[0.09981 , 0.45477 , - 0.87890,...[0.1 2344,-0.98757,.. -0.00854]]
'a giraffe standing in tall grass with trees in the background .'	ssss a giraffe standing in tall grass with trees in the background eeee	[2, 6, 117, 10, 16, 683, 7, 6, 1613, 3]	[[0.54687 , 0.63547 , - 0.77894,...[0.3 0210,-0.65489,.. -0.32154]]

MODULE 4: DECODER

MODULE 4: DECODER		
IMAGE	GENERATED CAPTION	
	ALGORITHM - 1 3 GRU	ALGORITHM - 2 2 GRU + 2 LSTM + 1 GRU
	a plate of food with broccoli and other foods	a close up of a piece of broccoli on a plate



a living room with a
television and a tv

a living room with a couch
table and a tv set



a woman is holding a pink
umbrella in a crowd

a woman in a floral swimsuit
holds a pink umbrella



a close up of a sandwich
with a slice of cheese

a slice of chocolate cake on
a plate with a fork



a man riding a motorcycle
on a city street

a motorcycle parked on the
side of a road

MODULE 5: LEXICAL ARTICULATOR

IMAGE	GENERATED CAPTION FROM DECODER	ARTICULATED CAPTION
	a close up of a piece of broccoli on a plate	Articulated-Caption
	a living room with a couch table and a tv set	Articulated-Caption
	a woman in a floral swimsuit holds a pink umbrella	Articulated-Caption
	a slice of chocolate cake on a plate with a fork	Articulated-Caption



a motorcycle parked on the
side of a road

[Articulated-Caption](#)

MODULE - 1 PRE-PROCESSOR

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import tensorflow as tf
4 import numpy as np
5 import sys
6 import requests
7 import urllib.request
8 import tarfile
9 import zipfile
10 import json
11 import os
12 import pickle
13 import numpy as np
14 from PIL import Image
15 from tensorflow.keras import backend as K
16 from tensorflow.keras.models import Model
17 from tensorflow.keras.layers import Input, Dense
18 from tensorflow.keras.applications import VGG16
```

```
1 data_dir = "/content/data/coco"
2 train_dir = "/content/data/coco/train2014"
3 val_dir = "/content/data/coco/val2014"
4 data_url = "http://images.cocodataset.org/"
```


An iterative approach has been chosen for the implementation of the problem statement. The first goal was to understand the nature of the dataset and pre-process it. The input MS-COCO 2014 dataset is of size 25 GB. In-order to deal with this huge dataset and the constrained computing resources we make use of the dynamic programming paradigm by caching the values, the first time the dataset is downloaded, in-order to make access faster the subsequent times. The dataset consisting of both images and captions together is loaded. The images and the corresponding captions are then segregated and stored separately. The images then undergo normalization followed by scaling to finish the pre-processing. On the other hand, the captions are encoded in a dictionary and are thus pre-processed so that it could be used by the tokenizer.

The cached file is stored as a pickle object and the function to accomplish this is defined as below. This is used to persist the data so it can be reloaded very quickly and easily. If the cache-file exists then the data is reloaded and returned, otherwise the function is called and the result is saved to cache.

```
1 def cache(cache_path, fn, *args, **kwargs):
2     if os.path.exists(cache_path):
3         with open(cache_path, mode='rb') as file:
4             obj = pickle.load(file)
5             print("Data loaded from cache-file: " + cache_path)
6     else:
7         obj = fn(*args, **kwargs)
8
9         with open(cache_path, mode='wb') as file:
10            pickle.dump(obj, file)
11
12            print("Data saved to cache-file: " + cache_path)
13
14    return obj
```

We facilitate the download of the dataset in the desirable format with the aid of the below functions.

```

1 def maybe_download_and_extract_2():
2     filenames = ["zips/train2014.zip", "zips/val2014.zip",
3                 "annotations/annotations_trainval2014.zip"]
4     for filename in filenames:
5         url = data_url + filename
6         print("Downloading " + url)
7         maybe_download_and_extract(url=url, download_dir=data_dir)

```

```

1 def _print_download_progress(count, block_size, total_size):
2     pct_complete = float(count * block_size) / total_size
3     pct_complete = min(1.0, pct_complete)
4     msg = "\r- Download progress: {0:.1%}".format(pct_complete)
5     sys.stdout.write(msg)
6     sys.stdout.flush()

```

```

1 def maybe_download_and_extract(url, download_dir):
2     filename = url.split('/')[-1]
3     file_path = os.path.join(download_dir, filename)
4     if not os.path.exists(file_path):
5         if not os.path.exists(download_dir):
6             os.makedirs(download_dir)
7         file_path, _ = urllib.request.urlretrieve(url=url, filename=file_path,
8                                                  reporthook=_print_download_progress)
9
10        print()
11        print("Download finished. Extracting files.")
12
13        if file_path.endswith(".zip"):
14            zipfile.ZipFile(file=file_path, mode="r").extractall(download_dir)
15        elif file_path.endswith((".tar.gz", ".tgz")):
16            tarfile.open(name=file_path, mode="r:gz").extractall(download_dir)
17        print("Done.")
18    else:
19        print("Data has apparently already been downloaded and unpacked.")

```



```

1 def _load_records(train=True):
2     if train:
3         filename = "captions_train2014.json"
4     else:
5         filename = "captions_val2014.json"
6
7     path = os.path.join(data_dir, "annotations", filename)
8     with open(path, "r", encoding="utf-8") as file:
9         data_raw = json.load(file)
10
11     images = data_raw['images']
12     annotations = data_raw['annotations']
13     records = dict()
14
15     for image in images:
16         image_id = image['id']
17         filename = image['file_name']
18         record = dict()
19         record['filename'] = filename
20         record['captions'] = list()
21         records[image_id] = record
22
23     for ann in annotations:
24         image_id = ann['image_id']
25         caption = ann['caption']
26         record = records[image_id]
27         record['captions'].append(caption)
28
29     records_list = [(key, record['filename'], record['captions'])
30                     for key, record in sorted(records.items())]
31
32     ids, filenames, captions = zip(*records_list)
33     return ids, filenames, captions

```

```

1 _, filenames_train, captions_train = load_records(train=True)
2 _, filenames_val, captions_val = load_records(train=False)
3 num_images_train = len(filenames_train)
4 num_images_val = len(filenames_val)

```

The below given load_image function accomplishes the job of image pre-processing. It loads the image from the given file-path and resizes it to the given size. The images are

scaled so that their pixels fall between 0.0 and 1.0. It is then plotted with the `show_image` function.

```
1 def load_image(path, size=None):
2     img = Image.open(path)
3     if not size is None:
4         img = img.resize(size=size, resample=Image.LANCZOS)
5
6     img = np.array(img)
7     img = img / 255.0
8     if (len(img.shape) == 2):
9         img = np.repeat(img[:, :, np.newaxis], 3, axis=2)
10
11     return img
```

```
1 def show_image(idx, train):
2     if train:
3         dir = train_dir
4         filename = filenames_train[idx]
5         captions = captions_train[idx]
6     else:
7         dir = val_dir
8         filename = filenames_val[idx]
9         captions = captions_val[idx]
10
11     path = os.path.join(dir, filename)
12     for caption in captions:
13         print(caption)
14
15     img = load_image(path)
16     plt.imshow(img)
17     plt.show()
```

The `pycocotools` has been put into use. It is a Python API that assists in loading, parsing and visualizing the annotations in COCO. We instantiate the COCO class by passing the json file as an argument.

```
1 %matplotlib inline
2 from pycocotools.coco import COCO
3 import numpy as np
4 import skimage.io as io
5 import matplotlib.pyplot as plt
6 import pylab
7 pylab.rcParams['figure.figsize'] = (8.0, 10.0)
```

```
1 capFile='/content/drive/MyDrive/data/coco/annotations/captions_val2014.json'
2 coco1=COCO(capFile)
3
```

```
loading annotations into memory...
Done (t=3.15s)
creating index...
index created!
```


Sample of the Annotations:

Annotations is a list of dictionaries. The dictionary contains the image_id, id(caption id), caption as the keys. Here id is the primary key and is used to retrieve a unique caption. img is a dictionary with the following keys. We use the coco_url to load and display the image

```
1 annIds = coco1.getAnnIds()
2 annotations = coco1.loadAnns(annIds)
3
4 for i in range(5):
5     print(annotations[i])
```

```
{'image_id': 203564, 'id': 37, 'caption': 'A bicycle replica with a clock as the front wheel.'}
{'image_id': 179765, 'id': 38, 'caption': 'A black Honda motorcycle parked in front of a garage.'}
{'image_id': 322141, 'id': 49, 'caption': 'A room with blue walls and a white sink and door.'}
{'image_id': 16977, 'id': 89, 'caption': 'A car that seems to be parked illegally behind a legally parked car'}
{'image_id': 106140, 'id': 98, 'caption': 'A large passenger airplane flying through the air.'}
```

```
1 # load and display image
2 img = coco1.loadImgs(imgIds[np.random.randint(0,len(imgIds))])[0]
3 print(img)
4
```

```
{'coco_url': 'http://images.cocodataset.org/val2014/COCO_val2014_000000260818.jpg',
'date_captured': '2013-11-25 21:07:37',
'file_name': 'COCO_val2014_000000260818.jpg',
'flickr_url': 'http://farm3.staticflickr.com/2023/1685630518_00b15897ab_z.jpg',
'height': 500,
'id': 260818,
'license': 5,
'width': 375}
```

Using the load_img method to display an image from the url:

```
1 # load and display image
2 img = coco1.loadImgs(imgIds[np.random.randint(0,len(imgIds))])[0]
3 # Or use url to load image
4 I = io.imread(img['coco_url'])
5 plt.axis('off')
6 plt.imshow(I)
7 plt.show()
8
9
```



MODULE – 2 VGG 16 MODEL ENCODER

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

Model: "vgg16"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000
=====		
Total params: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

We instantiate the VGG16 architecture by importing it from tensorflow.keras.applications. It loads weights pre-trained on ImageNet. The default

input size for this model is 224x224. We remove the last predictions layer and transfer the values of the second fully connected layer.

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

The `get_layer()` method takes as parameters the name of the specific layer which we want and retrieves the layer correspondingly. The `transfer_layer` variable has the `fc2` layer stored. We make use of the `K.int_shape()` function which returns the shape of tensor or variable as a tuple of int or None entries.

- i. `K.int_shape(image_model.input)` – Denotes shape of input vector to the model having value (None, 224, 224, 3).
- ii. `K.int_shape(transfer_layer.output)` – Denotes shape of output vector of `fc2` layer having value (None, 4096).

Thus, we assign `img_size` and `transfer_values_size` the values (224,224) and 4096 respectively.

```
1 transfer_layer = image_model.get_layer('fc2')
2 image_model_transfer = Model(inputs=image_model.input, outputs=transfer_layer.output)
3 img_size = K.int_shape(image_model.input)[1:3]
4 print(img_size)
5 transfer_values_size = K.int_shape(transfer_layer.output)[1]
6 print(transfer_values_size)
```

Next step is to process all the images with the `vgg16` model and cache the values.

In-order to cache the transfer-values, the `cache` function is called upon with the path as `data/coco/transfer_values_train.pkl`.

If the cache path i.e. the `transfer_values_train.pkl` file exists, we return the obj. If it doesn't exist the caching of the transfer-values occurs in batches of images. During this process, we load the image using `load_image()` function defined previously and the images are resized to meet the expected format for the `vgg16` architecture. Thus, we have cached the transfer values for the 82783 images in the training dataset by having the features extracted from the image from the output of the `fc2` layer of `vgg16` model.

```

1 def process_images(data_dir, filenames, batch_size=32):
2     num_images = len(filenames)
3     shape = (batch_size,) + img_size + (3,)
4     image_batch = np.zeros(shape=shape, dtype=np.float16)
5
6     shape = (num_images, transfer_values_size)
7     transfer_values = np.zeros(shape=shape, dtype=np.float16)
8     start_index = 0
9
10    while start_index < num_images:
11        print_progress(count=start_index, max_count=num_images)
12        end_index = start_index + batch_size
13        if end_index > num_images:
14            end_index = num_images
15        current_batch_size = end_index - start_index
16
17        for i, filename in enumerate(filenames[start_index:end_index]):
18            path = os.path.join(data_dir, filename)
19            img = load_image(path, size=img_size)
20            image_batch[i] = img
21
22        transfer_values_batch = \
23            image_model_transfer.predict(image_batch[0:current_batch_size])
24        transfer_values[start_index:end_index] = \
25            transfer_values_batch[0:current_batch_size]
26        start_index = end_index
27
28    print()
29    return transfer_values

```

```

1 def process_images_train():
2     print("Processing {0} images in training-set ...".format(len(filenames_train)))
3     cache_path = os.path.join(data_dir, "transfer_values_train.pkl")
4     transfer_values = cache(cache_path=cache_path, fn=process_images,
5                             data_dir=train_dir, filenames=filenames_train)
6     return transfer_values

```

```

1 def process_images_val():
2     print("Processing {0} images in validation-set ...".format(len(filenames_val)))
3     cache_path = os.path.join(data_dir, "transfer_values_val.pkl")
4     transfer_values = cache(cache_path=cache_path, fn=process_images,
5                             data_dir=val_dir, filenames=filenames_val)
6     return transfer_values

```

```
1 %%time
2 transfer_values_train = process_images_train()
3 print("dtype:", transfer_values_train.dtype)
4 print("shape:", transfer_values_train.shape)
```

```
Processing 82783 images in training-set ...
- Progress: 100.0%
- Data saved to cache-file: data/coco/transfer_values_train.pkl
dtype: float16
shape: (82783, 4096)
CPU times: user 26min 30s, sys: 5min 17s, total: 31min 48s
Wall time: 34min 50s
```

```
1 %%time
2 transfer_values_val = process_images_val()
3 print("dtype:", transfer_values_val.dtype)
4 print("shape:", transfer_values_val.shape)
```

```
Processing 40504 images in validation-set ...
- Progress: 99.9%
- Data saved to cache-file: data/coco/transfer_values_val.pkl
dtype: float16
shape: (40504, 4096)
CPU times: user 12min 56s, sys: 2min 32s, total: 15min 29s
Wall time: 16min 29s
```

```
1 def print_progress(count, max_count):
2     pct_complete = count / max_count
3     msg = "\r- Progress: {0:.1%}".format(pct_complete)
4     sys.stdout.write(msg)
5     sys.stdout.flush()
```

In-order to keep track of the progress, the percentage of completed downloads is constantly updated.

MODULE - 3 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.

```
1 mark_start = 'ssss '  
2 mark_end = ' eeee'
```

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.

```
1 def mark_captions(captions_listlist):  
2     captions_marked = [[mark_start + caption + mark_end  
3                         for caption in captions_list]  
4                         for captions_list in captions_listlist]  
5  
6     return captions_marked
```

The `mark_captions` function wraps all text-strings in the above markers. Since the captions are a list of lists, we use a nested for-loop to process using list-comprehension in python. In the inner loop we iterate over all the captions as we have four to six captions for each image. For each caption we append the start and end marker. Thus, `captions_marked` is a list of lists which is returned by this function to the variable `captions_train_marked`.

```
1 captions_train[0]
```

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

`captions_train` is obtained from the pre-processor module. It's a tuple of lists. Thus, each element of the tuple `captions_train` is as shown above.

```
1 captions_train_marked = mark_captions(captions_train)
```

```
1 captions_train_marked[0]
```

```
['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']
```

captions_train_marked is a list of lists where each caption has the start and end markers added. Thus, each element of the tuple captions_train_marked is like shown above.

```
1 def flatten(captions_listlist):  
2     captions_list = [caption  
3         for captions_list in captions_listlist  
4         for caption in captions_list]  
5  
6     return captions_list
```

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

Now process all the captions in the training-set. Next we make a call to flatten() function to which we pass as parameters the marked captions. The functionality achieved here is pretty straightforward and simple.

Basically, we're converting a list of lists into a single list. But now we unwrap the inner lists to provide a single flattened list. Thus, captions_list is a python list having all the captions in the training dataset. This is returned to captions_train_flat.

```
1 num_words = 10000
```

```
1 %%time  
2 tokenizer = TokenizerWrap(texts=captions_train_flat,  
3 | | | | | | | | | | | | num_words=num_words)  
4 token_start = tokenizer.word_index[mark_start.strip()]  
5 token_end = tokenizer.word_index[mark_end.strip()]  
6 print("Start Token: ", token_start)  
7 print("End Token: ", token_end)
```

```
Start Token: 2
End Token: 3
CPU times: user 6.69 s, sys: 1.76 ms, total: 6.69 s
Wall time: 6.75 s
```

```
End Token: 3
CPU times: user 6.69 s, sys: 1.76 ms, total: 6.69 s
Wall time: 6.75 s
```

```
CPU times: user 6.69 s, sys: 1.76 ms, total: 6.69 s
Wall time: 6.75 s
```

Wall time: 6.75 s

The `TokenizerWrap` is inherited from the `Tokenizer` class in `tensorflow.keras.preprocessing.text` as we need more functionality than provided by this `Tokenizer` class so we wrap it.

We instantiate a `TokenizerWrap` object as below by passing the flat list of captions and `num_words` as parameters. `TokenizerWrap` is used to convert a text into sequence of integers. The maximum number of words in the vocabulary is set using the `num_words` variable to 10000. This means that we will only use the 10000 most frequent words in the captions from the training-data.

```
1 %%time
2 tokens_train = tokenizer.captions_to_tokens(captions_train_marked)
```

```
CPU times: user 6.14 s, sys: 101 ms, total: 6.24 s
Wall time: 6.27 s
```

```
Wall time: 6.27 s
```

Next we pass the flattened caption to the `fit_on_texts()` function of `Tokenizer` class. This function updates internal vocabulary based on a list of texts. Then we iterate over the items of the list.

Now create a tokenizer using all the captions in the training-data. The flattened list of captions is used to create the tokenizer because it cannot take a list-of-lists.

```

1 class TokenizerWrap(Tokenizer):
2     def __init__(self, texts, num_words=None):
3         Tokenizer.__init__(self, num_words=num_words)
4         self.fit_on_texts(texts)
5         self.index_to_word = dict(zip(self.word_index.values(),
6                                     self.word_index.keys()))
7
8     def token_to_word(self, token):
9         word = " " if token == 0 else self.index_to_word[token]
10        return word
11
12    def tokens_to_string(self, tokens):
13        words = [self.index_to_word[token]
14                for token in tokens
15                if token != 0]
16        text = " ".join(words)
17
18        return text
19
20    def captions_to_tokens(self, captions_listlist):
21        tokens = [self.texts_to_sequences(captions_list)
22                 for captions_list in captions_listlist]
23
24        return tokens

```

```

1 def text_to_word_sequence(text,
2                           filters='!"$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',
3                           lower=True, split=" "):
4     if lower:
5         text = text.lower()
6
7     translate_dict = {c: split for c in filters}
8     print("translate_dict = ", translate_dict)
9     translate_map = str.maketrans(translate_dict)
10    print("translate_map = ", translate_map)
11    text = text.translate(translate_map)
12    print(text)
13    seq = text.split(split)
14    print(seq)
15    print([i for i in seq if i])
16    return [i for i in seq if i]

```

```

1 rval = text_to_word_sequence('That #book is Amazing!! && incredible%')
2 print(rval)

```

```

translate_dict = {'!': ' ', '"': ' ', '#': ' ', '$': ' ', '%': ' ', '&': ' ', '(': ' ',
translate_map = {33: ' ', 34: ' ', 35: ' ', 36: ' ', 37: ' ', 38: ' ', 40: ' ', 41:
that book is amazing      incredible
['that', '', 'book', 'is', 'amazing', '', '', '', '', '', 'incredible', '', '']
['that', 'book', 'is', 'amazing', 'incredible']
['that', 'book', 'is', 'amazing', 'incredible']

```

This is the sample execution of `text_to_word_sequence()`. We only pass the `text` parameter which is each string in the flattened list. Initially we convert it to lowercase. `translate_dict` is a dictionary where keys are the punctuations and values are space. The `maketrans()` method returns a mapping table that can be used with the `translate()` method to replace specific characters.

In the mapping table we replace the keys in `translate_dict` dictionary with their corresponding ASCII values. Using the `translate` function we replace the instances of the key in the text with space. i.e We turn all the punctuators into spaces. Now we split the text having space as the delimiter. In the end we return a list from the split text consisting of words which are of size at least 1.

```
1 tokenizer.word_index
```

```
{'a': 1,  
'ssss': 2,  
'eeee': 3,  
'on': 4,  
'of': 5,  
'the': 6,  
'in': 7,  
'with': 8,  
'and': 9,  
'is': 10,  
'man': 11,  
'to': 12,  
'sitting': 13,  
'an': 14,  
'two': 15,  
'standing': 16,
```

Next passing `captions_train_marked` which is a list of lists of marked captions to the function `captions_to_tokens`, we call the function `texts_to_sequences()` of `Tokenizer` class for each list in the variable `caption_listlist`.

```
1 %%time
```

```
2 tokens_train = tokenizer.captions_to_tokens(captions_train_marked)
```

`tokens_train` displays the output of the tokenizer (sequence of integers) .


```
1 captions_train_marked[0][0]
```

```
'ssss Closeup of bins of food that include broccoli and bread. eeee'
```

```
1 tokens_train[0][0]
```

```
[2, 844, 5, 2845, 5, 60, 25, 1933, 248, 9, 438, 3]
```

```
1 tokenizer.word_index
```

```
{'a': 1,  
'ssss': 2,  
'eeee': 3,  
'on': 4,  
'of': 5,  
'the': 6,  
'in': 7,
```

DATA GENERATOR FOR MODEL TRAINING

```
1 def get_random_caption_tokens(idx):  
2     result = []  
3     for i in idx:  
4         j = np.random.choice(len(tokens_train[i]))  
5         tokens = tokens_train[i][j]  
6         result.append(tokens)  
7     return result
```

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. The above function 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.

```

1 def batch_generator(batch_size):
2     while True:
3         idx = np.random.randint(num_images_train, size=batch_size)
4         transfer_values = transfer_values_train[idx]
5         tokens = get_random_caption_tokens(idx)
6         num_tokens = [len(t) for t in tokens]
7         max_tokens = np.max(num_tokens)
8         tokens_padded = pad_sequences(tokens, maxlen=max_tokens, padding='post', truncating='post')
9         decoder_input_data = tokens_padded[:, 0:-1]
10        decoder_output_data = tokens_padded[:, 1:]
11        x_data = { 'decoder_input': decoder_input_data, 'transfer_values_input': transfer_values }
12        y_data = { 'decoder_output': decoder_output_data }
13        yield (x_data, y_data)

```

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

```

1 batch_size = 1024
2 generator = batch_generator(batch_size=batch_size)
3 batch = next(generator)
4 batch_x = batch[0]
5 batch_y = batch[1]

```

Get the pre-computed transfer-values for those images. These are the outputs of the pre-trained image-model. For each of the randomly chosen images there are at least 5 captions describing the contents of the image. Select one of those captions at random and get the associated sequence of integer-tokens.

```
1 batch_x['decoder_input'].shape
```

```
(1024, 42)
```

```
1 batch_y['decoder_output'].shape
```

```
(1024, 42)
```

```
1 batch_x['transfer_values_input'].shape
```

```
(1024, 4096)
```

Pad all the other token-sequences with zeros so they all have the same length and can be input to the neural network as a numpy array. Set the batch-size used during training. This is set very high so the GPU can be used maximally - but this also requires a lot of RAM on the GPU. You may have to lower this number if the training runs out of memory. The steps per epoch is calculated depending upon the total number of captions in the training set and the size of each batch.

MODULE - 4 RNN DECODER

The Recurrent Neural Network (RNN) is trained to map the vectors with transfer-values from the image-recognition model into sequences of integer-tokens that can be converted into text.

```
1 state_size = 512  
2 embedding_size = 128
```

The functional model from Keras is used to build this neural network, because it allows more flexibility in how the neural network can be connected, enabling us to experiment and connect the image-model directly to the decoder. The network construction has been split into two parts: (1) Creation of all the layers that are not yet connected, and (2) a function that connects all these layers.

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

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

The embedding-layer converts integer-tokens into vectors of this length 128.

The transfer-values are used 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.

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

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

A tanh activation function is used to limit the output of the mapping between -1 and 1. This is the embedding-layer which converts sequences of integer-tokens to sequences of vectors. All the GRU units return sequences because we ultimately want to output a sequence of integer-tokens that can be converted into a text-sequence.

ALGORITHM - 1 : DECODER CONSISTING OF 3 GRU (GATED RECURRENT UNITS)

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

```
1 def connect_decoder(transfer_values):
2     initial_state = decoder_transfer_map(transfer_values)
3     net = decoder_input
4     net = decoder_embedding(net)
5     net = decoder_gru1(net, initial_state=initial_state)
6     net = decoder_gru2(net, initial_state=initial_state)
7     net = decoder_gru3(net, initial_state=initial_state)
8     decoder_output = decoder_dense(net)
9     return decoder_output
```

ALGORITHM - 2 : DECODER CONSISTING OF 2 GRU + 2 LSTM + 1 GRU

```
1 decoder_gru1 = GRU(state_size, name='decoder_gru1', return_sequences=True)
2 decoder_gru2 = GRU(state_size, name='decoder_gru2', return_sequences=True)
3
4 decoder_lstm1 = LSTM(state_size, name='decoder_lstm1', dropout=0.2, return_sequences=True)
5 decoder_lstm2 = LSTM(state_size, name='decoder_lstm2', dropout=0.2, return_sequences=True)
6
7 decoder_gru3 = GRU(state_size, name='decoder_gru3', return_sequences=True)
```

```
1 def connect_decoder(transfer_values):
2     initial_state = decoder_transfer_map(transfer_values)
3     net = decoder_input
4     net = decoder_embedding(net)
5     net = decoder_gru1(net, initial_state=initial_state)
6     net = decoder_gru2(net, initial_state=initial_state)
7     net = decoder_lstm1(net, initial_state=[initial_state, initial_state])
8     net = decoder_lstm2(net, initial_state=[initial_state, initial_state])
9     net = decoder_gru3(net, initial_state=initial_state)
10    print(net)
11    decoder_output = decoder_dense(net)
12
13    return decoder_output
```

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. To accomplish this we convert the GRU output to a one-hot encoded array.

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

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

The transfer values are mapped such that the dimensionality matches the internal states of the GRU layers. Hence we use the mapped transfer values as the initial state of the GRU layers. Thus the model 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.

```
1 decoder_output = connect_decoder(transfer_values=transfer_values_input)
2 decoder_model = Model(inputs=[transfer_values_input, decoder_input], outputs=[decoder_output])
```


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.

```
1 decoder_model.compile(optimizer=RMSprop(lr=1e-3), loss='sparse_categorical_crossentropy')

1 path_checkpoint = '/content/drive/MyDrive/data/lstm_gru/lstm_gru.keras'
2 callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint, verbose=1, save_weights_only=True)

1 callback_tensorboard = TensorBoard(log_dir='/content/drive/MyDrive/data/lstm_gru/lstm_gru',
2                                     histogram_freq=0, write_graph=False)
3 callbacks = [callback_checkpoint, callback_tensorboard]
```

However, our data-set contains integer-tokens instead of one-hot encoded arrays. This conversion from integers to one-hot arrays is done in the batch_generator() above.

This is done using sparse cross-entropy loss-function, which does the conversion internally from integers to one-hot encoded arrays. RMSprop optimizer has been used.

```
1 try:
2     decoder_model.load_weights(path_checkpoint)
3     print("Checkpoint loaded successfully")
4 except Exception as error:
5     print("Error trying to load checkpoint.")
6     print(error)
```

Checkpoint loaded successfully

During training we want to save checkpoints and log the progress to TensorBoard so we create the appropriate callbacks for Keras. This is the callback for writing checkpoints during training. The model has been trained for 300 epochs in which each epoch took 10 minutes to run on a GPU.

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

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

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

The number of steps per epoch is found from the ratio of the total number of captions to the batch size.

```
1 %%time
2 decoder_model.fit(x=generator, steps_per_epoch=steps_per_epoch,
3 | | | | | | | | epochs=10, callbacks=callbacks)
```

MODULE - 5 LEXICAL ARTICULATOR

```
1 def generate_caption(image_path, max_tokens=30):
2     image = load_image(image_path, size=img_size)
3     image_batch = np.expand_dims(image, axis=0)
4     transfer_values = image_model_transfer.predict(image_batch)
5
6     shape = (1, max_tokens)
7     decoder_input_data = np.zeros(shape=shape, dtype=np.int)
8     token_int = token_start
9
10    output_text = ''
11    count_tokens = 0
12
13    while token_int != token_end and count_tokens < max_tokens:
14        decoder_input_data[0, count_tokens] = token_int
15
16        x_data = \
17            {
18                'transfer_values_input': transfer_values,
19                'decoder_input': decoder_input_data
20            }
21
```

```

22     decoder_output = decoder_model.predict(x_data)
23     token_onehot = decoder_output[0, count_tokens, :]
24     token_int = np.argmax(token_onehot)
25     sampled_word = tokenizer.token_to_word(token_int)
26     output_text += " " + sampled_word
27     count_tokens += 1
28
29     output_tokens = decoder_input_data[0]
30     output_text = output_text.rsplit(' ',1)[0]
31     plt.imshow(image)
32     plt.show()
33
34     language = 'en'
35     articulator = gTTS(text=output_text, lang=language, slow=False)
36     articulator.save("articulated.wav")
37
38     return output_text
39

```

The `generate_caption()` function loads an image and generates a caption using the trained model. The `google-text-to-speech` library is used to convert the generated caption to speech. We set the language to english and set the pace to slow. The converted audio is saved and played using `IPython-Audio`.

```

1 def generate_caption_coco(idx, train=False):
2     if train:
3         data_dir = train_dir
4         filename = filenames_train[idx]
5         captions = captions_train[idx]
6     else:
7         data_dir = val_dir
8         filename = filenames_val[idx]
9         captions = captions_val[idx]
10
11     return generate_caption(image_path=filename)

```

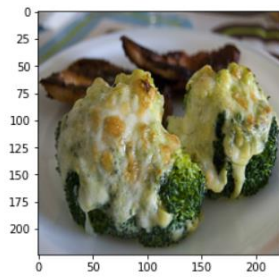
```

1 a = generate_caption_coco(1,False)
2 print("Predicted Caption:",a)
3 IPython.display.Audio('articulated.wav')

```

COMPARISON OF CAPTIONS OF ALGORITHMS

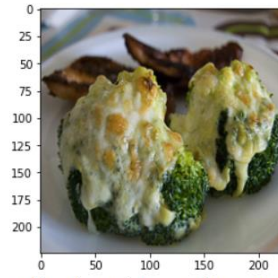
3 GRU



Predicted Caption: a plate of food with broccoli and other foods

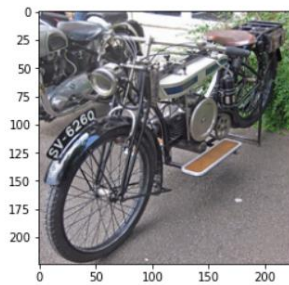
0:01 / 0:03

2 GRU + 2 LSTM + 1 GRU



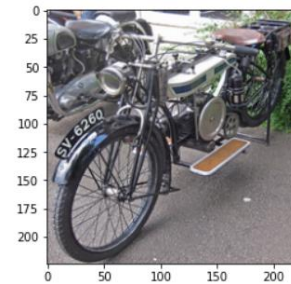
Predicted Caption: a close up of a piece of broccoli on a plate

0:03 / 0:03



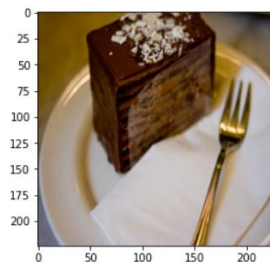
Predicted Caption: a man riding a motorcycle on a city street

0:02 / 0:03



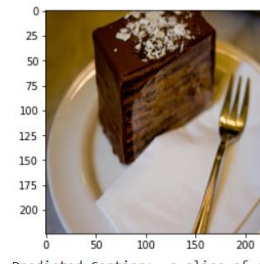
Predicted Caption: a motorcycle parked on the side of a road

0:02 / 0:03



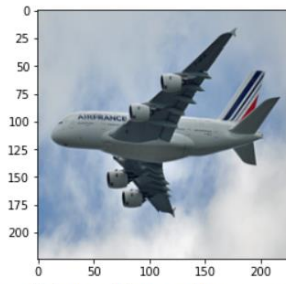
Predicted Caption: a close up of a sandwich with a slice of cheese

0:02 / 0:03



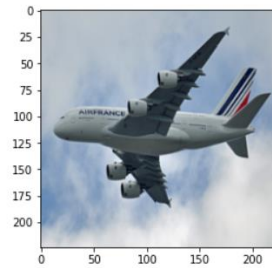
Predicted Caption: a slice of chocolate cake on a plate with a fork

0:01 / 0:03



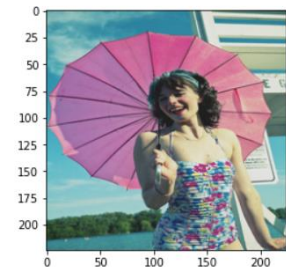
Predicted Caption: a large passenger jet flying through the sky

0:01 / 0:03



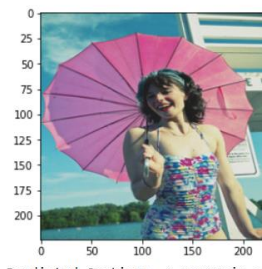
Predicted Caption: a large passenger jet flying through a blue sky

0:01 / 0:03



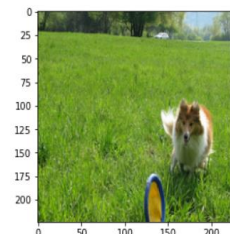
Predicted Caption: a woman is holding a pink umbrella in a crowd

0:01 / 0:03



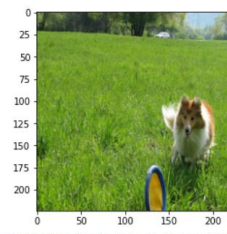
Predicted Caption: a woman in a floral swimsuit holds a pink umbrella

0:01 / 0:03



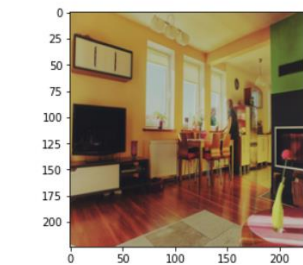
Predicted Caption: a dog is running through a field with a frisbee in its mouth

0:02 / 0:04



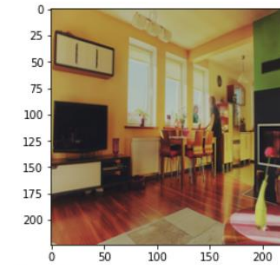
Predicted Caption: a dog running through the grass with a frisbee in its mouth

0:03 / 0:04



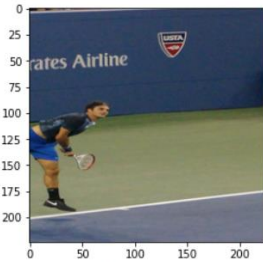
Predicted Caption: a living room with a television and a tv

0:02 / 0:03



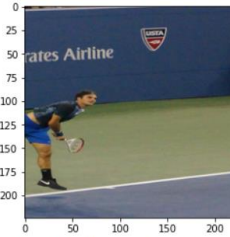
Predicted Caption: a living room with a couch table and a tv set

0:02 / 0:03



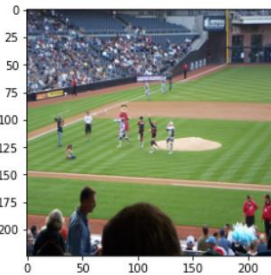
Predicted Caption: a man is swinging a tennis racket on a court

0:02 / 0:03



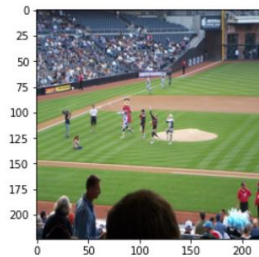
Predicted Caption: a tennis player is playing tennis on the tennis court

0:01 / 0:03



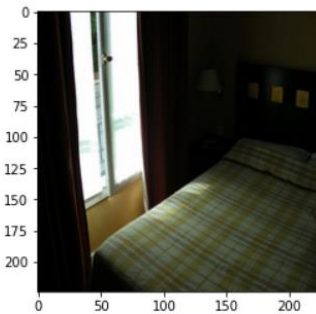
Predicted Caption: a group of people playing soccer on a field

0:02 / 0:03



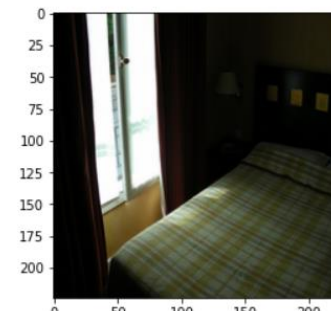
Predicted Caption: a baseball game is being played on a large field

0:02 / 0:03



Predicted Caption: a bedroom with a bed and a window

0:01 / 0:02



Predicted Caption: a bedroom with a bed and a window

0:02 / 0:02

PERFORMANCE METRIC EVALUATION

```
1 %shell
2 CORENLP=stanford-corenlp-full-2015-12-09
3 SPICELIB=pycocoevalcap/spice/lib
4 JAR=stanford-corenlp-3.6.0
5
6 DIR="$( cd "$(dirname "$0")" ; pwd -P )"
7 cd $DIR
8
9 if [ -f $SPICELIB/$JAR.jar ]; then
10 | echo "Found Stanford CoreNLP."
11 else
12 | echo "Downloading..."
13 | wget http://nlp.stanford.edu/software/$CORENLP.zip
14 | echo "Unzipping..."
15 | unzip $CORENLP.zip -d $SPICELIB/
16 | mv $SPICELIB/$CORENLP/$JAR.jar $SPICELIB/
17 | mv $SPICELIB/$CORENLP/$JAR-models.jar $SPICELIB/
18 | rm -f $CORENLP.zip
19 | rm -rf $SPICELIB/$CORENLP/
20 | echo "Done."
21 fi
```

Downloading...

--2021-05-06 18:50:45-- <http://nlp.stanford.edu/software/stanford-corenlp-full-2015-12-09.zip>

Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140

Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80... connected.

HTTP request sent, awaiting response... 302 Found

Location: <https://nlp.stanford.edu/software/stanford-corenlp-full-2015-12-09.zip> [following]

--2021-05-06 18:50:46-- <https://nlp.stanford.edu/software/stanford-corenlp-full-2015-12-09.zip>

Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.

HTTP request sent, awaiting response... 302 FOUND

Location: <https://downloads.cs.stanford.edu/nlp/software/stanford-corenlp-full-2015-12-09.zip> [following]

--2021-05-06 18:50:46-- <https://downloads.cs.stanford.edu/nlp/software/stanford-corenlp-full-2015-12-09.zip>

Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22

Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443... connected.

HTTP request sent, awaiting response... 200 OK

Length: 403157240 (384M) [application/zip]

Saving to: 'stanford-corenlp-full-2015-12-09.zip'

stanford-corenlp-fu 100%[=====>] 384.48M 5.12MB/s in 73s

2021-05-06 18:51:59 (5.26 MB/s) - 'stanford-corenlp-full-2015-12-09.zip' saved [403157240/403157240]

Unzipping...


```
1 annFile = '/content/drive/MyDrive/data/coco/annotations/captions_val2014.json'  
2 resultfile = '/content/drive/MyDrive/data/results/LSTM_GRU_results.json'  
3 coco = COCO(annFile)  
4 cocoRes = coco.loadRes(resultfile)
```

```
loading annotations into memory...  
Done (t=0.41s)  
creating index...  
index created!  
Loading and preparing results...  
DONE (t=0.32s)  
creating index...  
index created!
```

```
1 cocoEval = COCOEvalCap(coco, cocoRes)  
2 cocoEval.params['image_id'] = cocoRes.getImgIds()  
3 cocoEval.evaluate()
```

COMPARISON OF PERFORMANCE METRICS		
METRIC	ALGORITHM - 1	ALGORITHM - 2
BLEU-1	0.497	0.580
BLEU-2	0.297	0.406
BLEU-3	0.179	0.280
BLEU-4	0.103	0.193
METEOR	0.165	0.195
ROUGE-L	0.318	0.396
CIDER	0.471	0.600
SPICE	0.128	0.133

BLEU (Bilingual Evaluation Understudy) is a score for comparing a candidate translation of text to one or more reference translations. Blue score is a number between 0 and 1. This value indicates how similar the candidate text is to the reference texts, with values closer to 1 representing more similar texts. A perfect score of 1 indicates that the candidate is identical to one of the reference translations.

Score Calculation in BLEU

$$\text{Unigram precision } P = \frac{m}{w_t}$$

$$\text{Brevity penalty } p = \begin{cases} 1 & \text{if } c > r \\ e^{(1-\frac{r}{c})} & \text{if } c \leq r \end{cases}$$

$$\text{BLEU} = p \cdot e^{\sum_{n=1}^N \left(\frac{1}{N} * \log P_n \right)}$$

Meteor (Metric for Evaluation of Translation with Explicit Ordering) automatic evaluation metric scores machine translation hypotheses by aligning them to one or more reference translations. Alignments are based on exact, stem, synonym, and paraphrase matches between words and phrases. Segment and system level metric scores are calculated based on the alignments between hypothesis-reference pairs.

Score Calculation in METEOR

$$\text{Unigram precision } P = \frac{m}{w_t}$$

$$\text{Unigram recall } R = \frac{m}{w_r}$$

$$\text{And Harmonic mean } F_{\text{mean}} = \frac{10PR}{R+9P}$$

$$\text{The penalty } p = 0.5 * \left(\frac{c}{U_m} \right)^3$$

$$\text{Final score for a segment, } M = F_{\text{mean}}(1 - p)$$

CIDEr (Consensus-based Image Description Evaluation).

CIDEr metric measures the similarity of a generated sentence against a set of ground truth sentences written by humans. The metric shows high agreement with consensus as assessed by humans.

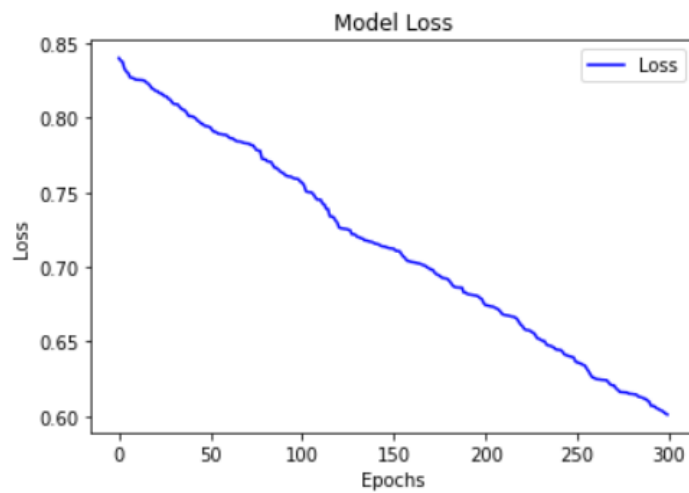
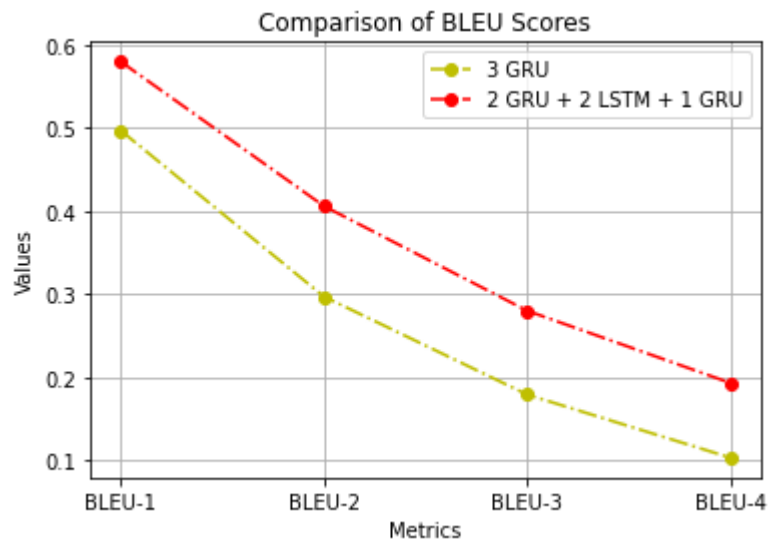
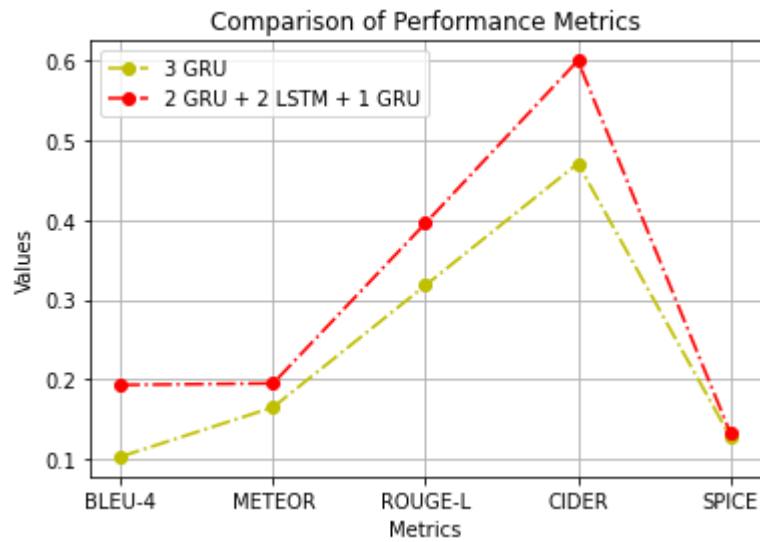
SPICE (Semantic Propositional Image Caption Evaluation)

In the first stage, syntactic dependencies between words in the caption are established using a dependency parser pretrained on a large dataset. In the second stage, dependency trees to scene graphs are mapped using a rule-based system. Given candidate and reference scene graphs, SPICE metric computes an F-score

$$P(c, S) = \frac{|T(G(c)) \otimes T(G(S))|}{|T(G(c))|}$$
$$R(c, S) = \frac{|T(G(c)) \otimes T(G(S))|}{|T(G(S))|}$$
$$SPICE(c, S) = F_1(c, S) = \frac{2 \cdot P(c, S) \cdot R(c, S)}{P(c, S) + R(c, S)}$$

ROUGE-L (Recall-Oriented Understudy for Gisting Evaluation) – measures longest matching sequence of words using LCS. An advantage of using LCS is that it does not require consecutive matches but in-sequence matches that reflect sentence level word order. Since it automatically includes longest in-sequence common n-grams, we don't need a predefined n-gram length.

Graphical analysis of Performance Metrics:



Inferences:

From the above graph it can be inferred that the algorithm 2 where the decoder is composed of a combination of 2 Gated Recurrent Units, 2 Long Short Term Memory and 1 Gated Recurrent Unit performs better than the algorithm 1 where the decoder is composed of 3 Gated Recurrent Units. The two algorithms are compared among 5 metrics - BLEU, METEOR, ROUGE-L, CIDER and SPICE and the Algorithm 2 outperforms the other in all the metrics.