CS6301 MACHINE LEARNING – MINI PROJECT

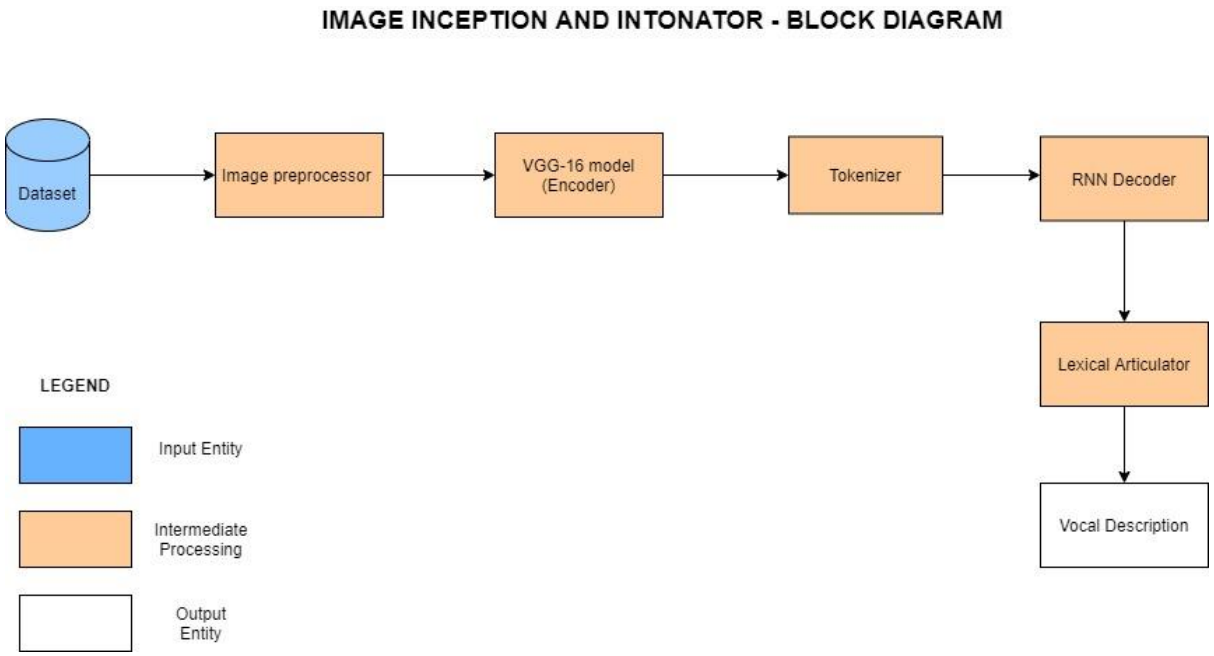
SRIHARI. S T.K.S. ARUNACHALAM

2018103601 2018103616

IMAGE INSCRIPTION AND INTONATOR - A NEURAL NETWORK APPROACH

40% IMPLEMENTATION UPDATE DOCUMENTATION

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



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

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import sys
import requests
import urllib.request
import tarfile
import zipfile
import json
import os
import pickle
import numpy as np
from PIL import Image
from tensorflow.keras import backend as K
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.applications import VGG16
```

```
data_dir = "/content/data/coco"
train_dir = "/content/data/coco/train2014"
val_dir = "/content/data/coco/val2014"
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.

```
def cache(cache_path, fn, *args, **kwargs):
    if os.path.exists(cache_path):
        with open(cache_path, mode='rb') as file:
            obj = pickle.load(file)
        print("Data loaded from cache-file: " + cache_path)
    else:
        obj = fn(*args, **kwargs)

        with open(cache_path, mode='wb') as file:
            pickle.dump(obj, file)

        print("Data saved to cache-file: " + cache_path)

    return obj
```

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

```
def maybe_download_and_extract_2():
    filenames = ["zips/train2014.zip", "zips/val2014.zip",
                 "annotations/annotations_trainval2014.zip"]
    for filename in filenames:
        url = data_url + filename
        print("Downloading " + url)
        maybe_download_and_extract(url=url, download_dir=data_dir)
```

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

```python
def maybe_download_and_extract(url, download_dir):
    filename = url.split('/')[-1]
    file_path = os.path.join(download_dir, filename)
    if not os.path.exists(file_path):
        if not os.path.exists(download_dir):
            os.makedirs(download_dir)
        file_path, _ = urllib.request.urlretrieve(url=url, filename=file_path,
                                                  reporthook=_print_download_progress)

        print()
        print("Download finished. Extracting files.")

        if file_path.endswith(".zip"):
            zipfile.ZipFile(file=file_path, mode="r").extractall(download_dir)
        elif file_path.endswith((".tar.gz", ".tgz")):
            tarfile.open(name=file_path, mode="r:gz").extractall(download_dir)
        print("Done.")
    else:
        print("Data has apparently already been downloaded and unpacked.")
```

Downloading the dataset:

```python
maybe_download_and_extract_2()
```

```
Downloading http://images.cocodataset.org/zips/train2014.zip
- Download progress: 100.0%
Download finished. Extracting files.
Done.
Downloading http://images.cocodataset.org/zips/val2014.zip
- Download progress: 67.7%
```

The COCO data-set contains a large number of images and various data for each image stored in a JSON-file. The load_records function provides the functionality to get a list of image-filenames (but not actually loading the images) along with their associated data such as text-captions describing the contents of the images.

```python
def load_records(train=True):
    if train:
        cache_filename = "records_train.pkl"
    else:
        cache_filename = "records_val.pkl"

    cache_path = os.path.join(data_dir, cache_filename)
    records = cache(cache_path=cache_path,
                    fn=_load_records,
                    train=train)
    return records
```

```python
def _load_records(train=True):
    if train:
        filename = "captions_train2014.json"
    else:
        filename = "captions_val2014.json"

    path = os.path.join(data_dir, "annotations", filename)
    with open(path, "r", encoding="utf-8") as file:
        data_raw = json.load(file)

    images = data_raw['images']
    annotations = data_raw['annotations']
    records = dict()

    for image in images:
        image_id = image['id']
        filename = image['file_name']
        record = dict()
        record['filename'] = filename
        record['captions'] = list()
        records[image_id] = record

    for ann in annotations:
        image_id = ann['image_id']
        caption = ann['caption']
        record = records[image_id]
        record['captions'].append(caption)

    records_list = [(key, record['filename'], record['captions'])
                    for key, record in sorted(records.items())]

    ids, filenames, captions = zip(*records_list)
    return ids, filenames, captions
```

```python
_, filenames_train, captions_train = load_records(train=True)
_, filenames_val, captions_val = load_records(train=False)
num_images_train = len(filenames_train)
num_images_train
```

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.

```
def load_image(path, size=None):
    img = Image.open(path)
    if not size is None:
        img = img.resize(size=size, resample=Image.LANCZOS)

    img = np.array(img)
    img = img / 255.0
    if (len(img.shape) == 2):
        img = np.repeat(img[:, :, np.newaxis], 3, axis=2)

    return img
```

```
def show_image(idx, train):
    if train:
        dir = train_dir
        filename = filenames_train[idx]
        captions = captions_train[idx]
    else:
        dir = val_dir
        filename = filenames_val[idx]
        captions = captions_val[idx]

    path = os.path.join(dir, filename)
    for caption in captions:
        print(caption)

    img = load_image(path)
    plt.imshow(img)
    plt.show()
```

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

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

```
def process_images(data_dir, filenames, batch_size=32):
    num_images = len(filenames)
    shape = (batch_size,) + img_size + (3,)
    image_batch = np.zeros(shape=shape, dtype=np.float16)

    shape = (num_images, transfer_values_size)
    transfer_values = np.zeros(shape=shape, dtype=np.float16)
    start_index = 0

    while start_index < num_images:
        print_progress(count=start_index, max_count=num_images)
        end_index = start_index + batch_size
        if end_index > num_images:
            end_index = num_images
        current_batch_size = end_index - start_index

        for i, filename in enumerate(filenames[start_index:end_index]):
            path = os.path.join(data_dir, filename)
            img = load_image(path, size=img_size)
            image_batch[i] = img

        transfer_values_batch = \
            image_model_transfer.predict(image_batch[0:current_batch_size])
        transfer_values[start_index:end_index] = \
            transfer_values_batch[0:current_batch_size]
        start_index = end_index

    print()
    return transfer_values
```

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

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

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

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

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

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

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.

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

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

```

```
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

```
annIds = coco1.getAnnIds()
annotations = coco1.loadAnns(annIds)

for i in range(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.'}
```

img is a dictionary with the following keys. We use the coco_url to load and display the image

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

```

```
{'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:

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


```