generatePhotoCaptions

December 16, 2020

[1]:
```
from numpy import array
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Embedding
from keras.layers import Dropout
from keras.layers.merge import add
from keras.callbacks import ModelCheckpoint
```

[2]:
```
from os import listdir
from pickle import dump
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.models import Model

# extract features from each photo in the directory
def extract_features(directory):
    # load the model
    model = VGG16()
    # re-structure the model
    model = Model(inputs=model.inputs, outputs=model.layers[-2].output)
    # summarize
    print(model.summary())
    # extract features from each photo
    features = dict()
    for name in listdir(directory):
        # load an image from file
        filename = directory + '/' + name
        image = load_img(filename, target_size=(224, 224))
```

```
        # convert the image pixels to a numpy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], image.
 ↪shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # get features
        feature = model.predict(image, verbose=0)
        # get image id
        image_id = name.split('.')[0]
        # store feature
        features[image_id] = feature
    return features

# extract features from all images
directory = 'Flickr8k_Dataset'
features = extract_features(directory)
print('Extracted Features: %d' % len(features))
# save to file
dump(features, open('features.pkl', 'wb'))
```

```
Model: "model"
_________________________________________________________________
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
=================================================================
Total params: 134,260,544
Trainable params: 134,260,544
Non-trainable params: 0
_________________________________________________________________
None
Extracted Features: 8091
```

[3]:
```python
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
```

[4]:
```python
# extract descriptions for images
def load_descriptions(doc):
    mapping = dict()
    # process lines
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        if len(line) < 2:
```

```
            continue
        # take the first token as the image id, the rest as the description
        image_id, image_desc = tokens[0], tokens[1:]
        # remove filename from image id
        image_id = image_id.split('.')[0]
        # convert description tokens back to string
        image_desc = ' '.join(image_desc)
        # create the list if needed
        if image_id not in mapping:
            mapping[image_id] = list()
        # store description
        mapping[image_id].append(image_desc)
    return mapping
```

[5]:
```
import string

def clean_descriptions(descriptions):
    # prepare translation table for removing punctuation
    table = str.maketrans('', '', string.punctuation)
    for key, desc_list in descriptions.items():
        for i in range(len(desc_list)):
            desc = desc_list[i]
            # tokenize
            desc = desc.split()
            # convert to lower case
            desc = [word.lower() for word in desc]
            # remove punctuation from each token
            desc = [w.translate(table) for w in desc]
            # remove hanging 's' and 'a'
            desc = [word for word in desc if len(word)>1]
            # remove tokens with numbers in them
            desc = [word for word in desc if word.isalpha()]
            # store as string
            desc_list[i] =  ' '.join(desc)
```

[6]:
```
# convert the loaded descriptions into a vocabulary of words
def to_vocabulary(descriptions):
    # build a list of all description strings
    all_desc = set()
    for key in descriptions.keys():
        [all_desc.update(d.split()) for d in descriptions[key]]
    return all_desc
```

[36]:
```
# save descriptions to file, one per line
def save_descriptions(descriptions, filename):
    lines = list()
    for key, desc_list in descriptions.items():
```

```
        for desc in desc_list:
            lines.append(key + ' ' + desc)
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()
```

[ ]:
```
# convert the loaded descriptions into a vocabulary of words
def to_vocabulary(descriptions):
    # build a list of all description strings
    all_desc = set()
    for key in descriptions.keys():
        [all_desc.update(d.split()) for d in descriptions[key]]
    return all_desc
```

[ ]:
```
filename = 'Flickr8k_text/Flickr8k.token.txt'
# load descriptions
doc = load_doc(filename)
# parse descriptions
descriptions = load_descriptions(doc)
print('Loaded: %d ' % len(descriptions))
# clean descriptions
clean_descriptions(descriptions)
# summarize vocabulary
vocabulary = to_vocabulary(descriptions)
print('Vocabulary Size: %d' % len(vocabulary))
# save to file
save_descriptions(descriptions, 'descriptions.txt')
```

```
Loaded: 8092
Vocabulary Size: 8763
```

[ ]:
```
# load a pre-defined list of photo identifiers
def load_set(filename):
    doc = load_doc(filename)
    dataset = list()
    # process line by line
    for line in doc.split('\n'):
        # skip empty lines
        if len(line) < 1:
            continue
        # get the image identifier
        identifier = line.split('.')[0]
        dataset.append(identifier)
    return set(dataset)
```

[ ]:
```
# load clean descriptions into memory
def load_clean_descriptions(filename, dataset):
    # load document
    doc = load_doc(filename)
    descriptions = dict()
    for line in doc.split('\n'):
        # split line by white space
        tokens = line.split()
        # split id from description
        image_id, image_desc = tokens[0], tokens[1:]
        # skip images not in the set
        if image_id in dataset:
            # create list
            if image_id not in descriptions:
                descriptions[image_id] = list()
            # wrap description in tokens
            desc = 'startseq ' + ' '.join(image_desc) + ' endseq'
            # store
            descriptions[image_id].append(desc)
    return descriptions
```

[ ]:
```
# load photo features
def load_photo_features(filename, dataset):
    # load all features
    all_features = load(open(filename, 'rb'))
    # filter features
    features = {k: all_features[k] for k in dataset}
    return features
```

[ ]:
```
from pickle import load
# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# photo features
train_features = load_photo_features('features.pkl', train)
print('Photos: train=%d' % len(train_features))
```

```
Dataset: 6000
Descriptions: train=6000
Photos: train=6000
```

[ ]:
```
# convert a dictionary of clean descriptions to a list of descriptions
def to_lines(descriptions):
```

```
    all_desc = list()
    for key in descriptions.keys():
        [all_desc.append(d) for d in descriptions[key]]
    return all_desc
```

[ ]:
```
# fit a tokenizer given caption descriptions
def create_tokenizer(descriptions):
    lines = to_lines(descriptions)
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer
```

[ ]:
```
from keras.preprocessing.text import Tokenizer
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
```

```
Vocabulary Size: 7579
```

[ ]:
```
# create sequences of images, input sequences and output words for an image
def create_sequences(tokenizer, max_length, descriptions, photos, vocab_size):
    X1, X2, y = list(), list(), list()
    # walk through each image identifier
    for key, desc_list in descriptions.items():
        # walk through each description for the image
        for desc in desc_list:
            # encode the sequence
            seq = tokenizer.texts_to_sequences([desc])[0]
            # split one sequence into multiple X,y pairs
            for i in range(1, len(seq)):
                # split into input and output pair
                in_seq, out_seq = seq[:i], seq[i]
                # pad input sequence
                in_seq = pad_sequences([in_seq], maxlen=max_length)[0]
                # encode output sequence
                out_seq = to_categorical([out_seq], num_classes=vocab_size)[0]
                # store
                X1.append(photos[key][0])
                X2.append(in_seq)
                y.append(out_seq)
    return array(X1), array(X2), array(y)
```

[ ]:
```
# calculate the length of the description with the most words
def max_length(descriptions):
    lines = to_lines(descriptions)
    return max(len(d.split()) for d in lines)
```

[ ]:
```
def define_model(vocab_size, max_length):
    # feature extractor model
    inputs1 = Input(shape=(4096,))
    fe1 = Dropout(0.5)(inputs1)
    fe2 = Dense(256, activation='relu')(fe1)
    # sequence model
    inputs2 = Input(shape=(max_length,))
    se1 = Embedding(vocab_size, 256, mask_zero=True)(inputs2)
    se2 = Dropout(0.5)(se1)
    se3 = LSTM(256)(se2)
    # decoder model
    decoder1 = add([fe2, se3])
    decoder2 = Dense(256, activation='relu')(decoder1)
    outputs = Dense(vocab_size, activation='softmax')(decoder2)
    # tie it together [image, seq] [word]
    model = Model(inputs=[inputs1, inputs2], outputs=outputs)
    model.compile(loss='categorical_crossentropy', optimizer='adam')
    # summarize model
    print(model.summary())
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

[ ]:
```
# train dataset

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# photo features
train_features = load_photo_features('features.pkl', train)
print('Photos: train=%d' % len(train_features))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# determine the maximum sequence length
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)
# prepare sequences
X1train, X2train, ytrain = create_sequences(tokenizer, max_length,␣
 ↪train_descriptions, train_features, vocab_size)
```

```
Dataset: 6000
Descriptions: train=6000
```

```
Photos: train=6000
Vocabulary Size: 7579
Description Length: 34
```

[ ]:
```
# dev dataset

# load test set
filename = 'Flickr8k_text/Flickr_8k.devImages.txt'
test = load_set(filename)
print('Dataset: %d' % len(test))
# descriptions
test_descriptions = load_clean_descriptions('descriptions.txt', test)
print('Descriptions: test=%d' % len(test_descriptions))
# photo features
test_features = load_photo_features('features.pkl', test)
print('Photos: test=%d' % len(test_features))
# prepare sequences
X1test, X2test, ytest = create_sequences(tokenizer, max_length,␣
 ↪test_descriptions, test_features, vocab_size)
```

```
Dataset: 1000
Descriptions: test=1000
Photos: test=1000
```

[ ]:
```
from datetime import datetime
logdir = "logs/scalars/" + datetime.now().strftime("%Y%m%d-%H%M%S")
```

[ ]:
```
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import TensorBoard
# fit model

# define the model
model = define_model(vocab_size, max_length)
# define callback
filepath = 'model-ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5'
my_callbacks = [
    EarlyStopping(monitor='val_loss', mode='min', verbose=1,patience=2),
    ModelCheckpoint(filepath, monitor='val_loss', verbose=1,␣
 ↪save_best_only=True, mode='min'),
    TensorBoard(log_dir=logdir),
]

# fit model
model.fit([X1train, X2train], ytrain, epochs=20, verbose=2,␣
 ↪callbacks=my_callbacks,
          validation_data=([X1test, X2test], ytest))
```

```
Model: "model_1"
```

```
__________________________________________________________________________________
__________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================
==================
input_3 (InputLayer)            [(None, 34)]         0
__________________________________________________________________________________
__________________
input_2 (InputLayer)            [(None, 4096)]       0
__________________________________________________________________________________
__________________
embedding (Embedding)           (None, 34, 256)      1940224     input_3[0][0]
__________________________________________________________________________________
__________________
dropout (Dropout)               (None, 4096)         0           input_2[0][0]
__________________________________________________________________________________
__________________
dropout_1 (Dropout)             (None, 34, 256)      0           embedding[0][0]
__________________________________________________________________________________
__________________
dense (Dense)                   (None, 256)          1048832     dropout[0][0]
__________________________________________________________________________________
__________________
lstm (LSTM)                     (None, 256)          525312      dropout_1[0][0]
__________________________________________________________________________________
__________________
add (Add)                       (None, 256)          0           dense[0][0]
                                                                 lstm[0][0]
__________________________________________________________________________________
__________________
dense_1 (Dense)                 (None, 256)          65792       add[0][0]
__________________________________________________________________________________
__________________
dense_2 (Dense)                 (None, 7579)         1947803     dense_1[0][0]
==================================================================================
==================
Total params: 5,527,963
Trainable params: 5,527,963
Non-trainable params: 0
__________________________________________________________________________________
__________________
None
Epoch 1/20
9576/9576 - 1749s - loss: 4.5202 - val_loss: 4.0662

Epoch 00001: val_loss improved from inf to 4.06623, saving model to model-
ep001-loss4.520-val_loss4.066.h5
Epoch 2/20
```

```
9576/9576 - 1742s - loss: 3.8547 - val_loss: 3.9431

Epoch 00002: val_loss improved from 4.06623 to 3.94305, saving model to model-
ep002-loss3.855-val_loss3.943.h5
Epoch 3/20
9576/9576 - 1760s - loss: 3.6498 - val_loss: 3.8795

Epoch 00003: val_loss improved from 3.94305 to 3.87946, saving model to model-
ep003-loss3.650-val_loss3.879.h5
Epoch 4/20
9576/9576 - 1736s - loss: 3.5471 - val_loss: 3.8886

Epoch 00004: val_loss did not improve from 3.87946
Epoch 5/20
9576/9576 - 1744s - loss: 3.4895 - val_loss: 3.9177

Epoch 00005: val_loss did not improve from 3.87946
Epoch 00005: early stopping
```

[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x13771ca20>
```

[ ]:
```
%load_ext tensorboard
%tensorboard --logdir logs/scalars
```

```
Reusing TensorBoard on port 6008 (pid 99367), started 9:53:40 ago. (Use '!kill␣
↪99367' to kill it.)

<IPython.core.display.HTML object>
```

[ ]:
```
# Evaluate (skore) the model using BLEU
from numpy import argmax
from pickle import load
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model
from nltk.translate.bleu_score import corpus_bleu
```

[ ]:
```
# calculate the length of the description with the most words
def max_length(descriptions):
    lines = to_lines(descriptions)
    return max(len(d.split()) for d in lines)
```

[ ]:
```
# map an integer to a word
def word_for_id(integer, tokenizer):
    for word, index in tokenizer.word_index.items():
        if index == integer:
            return word
    return None
```

[ ]:
```
# generate a description for an image
def generate_desc(model, tokenizer, photo, max_length):
    # seed the generation process
    in_text = 'startseq'
    # iterate over the whole length of the sequence
    for i in range(max_length):
        # integer encode input sequence
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        # pad input
        sequence = pad_sequences([sequence], maxlen=max_length)
        # predict next word
        yhat = model.predict([photo,sequence], verbose=0)
        # convert probability to integer
        yhat = argmax(yhat)
        # map integer to word
        word = word_for_id(yhat, tokenizer)
        # stop if we cannot map the word
        if word is None:
            break
        # append as input for generating the next word
        in_text += ' ' + word
        # stop if we predict the end of the sequence
        if word == 'endseq':
            break
    return in_text
```

[ ]:
```
# evaluate the skill of the model
def evaluate_model(model, descriptions, photos, tokenizer, max_length):
    actual, predicted = list(), list()
    # step over the whole set
    for key, desc_list in descriptions.items():
        # generate description
        yhat = generate_desc(model, tokenizer, photos[key], max_length)
        # store actual and predicted
        references = [d.split() for d in desc_list]
        actual.append(references)
        predicted.append(yhat.split())
    # calculate BLEU score
    print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
    print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0,␣
↪0)))
    print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3,␣
↪0)))
    print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.
↪25, 0.25)))
```

[ ]:
```
# prepare tokenizer on train set

# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
vocab_size = len(tokenizer.word_index) + 1
print('Vocabulary Size: %d' % vocab_size)
# determine the maximum sequence length
max_length = max_length(train_descriptions)
print('Description Length: %d' % max_length)
```

```
Dataset: 6000
Descriptions: train=6000
Vocabulary Size: 7579
Description Length: 34
```

[ ]:
```
# prepare test set

# load test set
filename = 'Flickr8k_text/Flickr_8k.testImages.txt'
test = load_set(filename)
print('Dataset: %d' % len(test))
# descriptions
test_descriptions = load_clean_descriptions('descriptions.txt', test)
print('Descriptions: test=%d' % len(test_descriptions))
# photo features
test_features = load_photo_features('features.pkl', test)
print('Photos: test=%d' % len(test_features))
```

```
Dataset: 1000
Descriptions: test=1000
Photos: test=1000
```

[ ]:
```
# load the model
filename = 'model-ep003-loss3.634-val_loss3.867.h5'
model = load_model(filename)
# evaluate model
evaluate_model(model, test_descriptions, test_features, tokenizer, max_length)
```

```
BLEU-1: 0.428795
BLEU-2: 0.227460
BLEU-3: 0.152939
```

```
BLEU-4: 0.064423
```

[54]:
```
# generate a caption for sample image

from keras.preprocessing.text import Tokenizer
from pickle import dump
from pickle import load
from numpy import argmax
from keras.preprocessing.sequence import pad_sequences
from keras.applications.vgg16 import VGG16
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.models import Model
from keras.models import load_model
```

[55]:
```
# load training dataset (6K)
filename = 'Flickr8k_text/Flickr_8k.trainImages.txt'
train = load_set(filename)
print('Dataset: %d' % len(train))
# descriptions
train_descriptions = load_clean_descriptions('descriptions.txt', train)
print('Descriptions: train=%d' % len(train_descriptions))
# prepare tokenizer
tokenizer = create_tokenizer(train_descriptions)
# save the tokenizer
dump(tokenizer, open('tokenizer.pkl', 'wb'))
```

```
Dataset: 6000
Descriptions: train=6000
```

[56]:
```
# extract features from one photo
def extract_features(filename):
        # load the model
        model = VGG16()
        # re-structure the model
        model = Model(inputs=model.inputs, outputs=model.layers[-2].output)
        # load the photo
        image = load_img(filename, target_size=(224, 224))
        # convert the image pixels to a numpy array
        image = img_to_array(image)
        # reshape data for the model
        image = image.reshape((1, image.shape[0], image.shape[1], image.
 ↪shape[2]))
        # prepare the image for the VGG model
        image = preprocess_input(image)
        # get features
```

```
        feature = model.predict(image, verbose=0)
        return feature
```

[57]:
```
# load the tokenizer
tokenizer = load(open('tokenizer.pkl', 'rb'))
# pre-define the max sequence length (from training)
max_length = 34
# load the model
model = load_model('model-ep003-loss3.634-val_loss3.867.h5')
# load and prepare the photograph
photo = extract_features('sample.jpg')
# generate description
description = generate_desc(model, tokenizer, photo, max_length)
print(description)
```

```
startseq two dogs are running through the grass endseq
```

[ ]:
```

```

model2_train

December 18, 2020

[ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.utils import shuffle
from imageio import imread
import scipy.io
import cv2
import os
import json
from tqdm import tqdm
import pickle
```

[2]:
```
batch_size = 128
maxlen = 20
image_size = 224

MEAN_VALUES = np.array([123.68, 116.779, 103.939]).reshape((1, 1, 3))

def load_data(image_dir, annotation_path):
    with open(annotation_path, 'r') as fr:
        annotation = json.load(fr)

    ids = []
    captions = []
    image_dict = {}
    for i in tqdm(range(len(annotation['annotations']))):
        item = annotation['annotations'][i]
        caption = item['caption'].strip().lower()
        caption = caption.replace('.', '').replace(',', '').replace("'", '').
 ↪replace('"', '')
        caption = caption.replace('&', 'and').replace('(', '').replace(')', '').
 ↪replace('-', ' ').split()
        caption = [w for w in caption if len(w) > 0]

        if len(caption) <= maxlen:
            if not item['image_id'] in image_dict:
```

```
                img = imread(image_dir + '%012d.jpg' % item['image_id'])
                h = img.shape[0]
                w = img.shape[1]
                if h > w:
                    img = img[h // 2 - w // 2: h // 2 + w // 2, :]
                else:
                    img = img[:, w // 2 - h // 2: w // 2 + h // 2]
                img = cv2.resize(img, (image_size, image_size))

                if len(img.shape) < 3:
                    img = np.expand_dims(img, -1)
                    img = np.concatenate([img, img, img], axis=-1)

                image_dict[item['image_id']] = img

            ids.append(item['image_id'])
            captions.append(caption)

    return ids, captions, image_dict

train_json = 'data/train/captions_train2014.json'
train_ids, train_captions, train_dict = load_data('data/train/images/
 ↪COCO_train2014_', train_json)
print(len(train_ids))
```

```
100%|      | 414113/414113 [11:10<00:00, 617.48it/s]
```

```
411593
```

[3]:
```
data_index = np.arange(len(train_ids))
np.random.shuffle(data_index)
N = 4
data_index = data_index[:N]
plt.figure(figsize=(12, 20))
for i in range(N):
    caption = train_captions[data_index[i]]
    img = train_dict[train_ids[data_index[i]]]
    plt.subplot(4, 1, i + 1)
    plt.imshow(img)
    plt.title(' '.join(caption))
    plt.axis('off')
```

a juvenile horse is galloping on a grass field

a stop sign on the corner of montview blvd and roslyn st

a woman with an umbrella rides her bike across the street

a girl on a ladder next to a tree in an apple orchard

[4]:
```
vocabulary = {}
for caption in train_captions:
    for word in caption:
        vocabulary[word] = vocabulary.get(word, 0) + 1

vocabulary = sorted(vocabulary.items(), key=lambda x:-x[1])
vocabulary = [w[0] for w in vocabulary]

word2id = {'<pad>': 0, '<start>': 1, '<end>': 2}
for i, w in enumerate(vocabulary):
    word2id[w] = i + 3
id2word = {i: w for w, i in word2id.items()}

print(len(vocabulary), vocabulary[:20])

with open('dictionary.pkl', 'wb') as fw:
    pickle.dump([vocabulary, word2id, id2word], fw)

def translate(ids):
    words = [id2word[i] for i in ids if i >= 3]
    return ' '.join(words) + '.'
```

```
23728 ['a', 'on', 'of', 'the', 'in', 'with', 'and', 'is', 'man', 'to',
'sitting', 'an', 'two', 'standing', 'at', 'people', 'are', 'next', 'white',
'woman']
```

[5]:
```
def convert_captions(data):
    result = []
    for caption in data:
        vector = [word2id['<start>']]
        for word in caption:
            if word in word2id:
                vector.append(word2id[word])
        vector.append(word2id['<end>'])
        result.append(vector)

    array = np.zeros((len(data), maxlen + 2), np.int32)
    for i in tqdm(range(len(result))):
        array[i, :len(result[i])] = result[i]
    return array

train_captions = convert_captions(train_captions)
print(train_captions.shape)
print(train_captions[0])
print(translate(train_captions[0]))
```

```
100%|      | 411593/411593 [00:00<00:00, 418182.64it/s]

(411593, 22)
[  1   3 141 503   9 629 414 274  57   2   0   0   0   0   0   0   0   0
   0   0   0   0]
a very clean and well decorated empty bathroom.
```

[6]:
```
vgg = scipy.io.loadmat('imagenet-vgg-verydeep-19.mat')
vgg_layers = vgg['layers']

def vgg_endpoints(inputs, reuse=None):
    with tf.variable_scope('endpoints', reuse=reuse):
        def _weights(layer, expected_layer_name):
            W = vgg_layers[0][layer][0][0][2][0][0]
            b = vgg_layers[0][layer][0][0][2][0][1]
            layer_name = vgg_layers[0][layer][0][0][0][0]
            assert layer_name == expected_layer_name
            return W, b

        def _conv2d_relu(prev_layer, layer, layer_name):
            W, b = _weights(layer, layer_name)
            W = tf.constant(W)
            b = tf.constant(np.reshape(b, (b.size)))
            return tf.nn.relu(tf.nn.conv2d(prev_layer, filter=W, strides=[1, 1,␣
 ↪1, 1], padding='SAME') + b)

        def _avgpool(prev_layer):
            return tf.nn.avg_pool(prev_layer, ksize=[1, 2, 2, 1], strides=[1,␣
 ↪2, 2, 1], padding='SAME')

        graph = {}
        graph['conv1_1']  = _conv2d_relu(inputs, 0, 'conv1_1')
        graph['conv1_2']  = _conv2d_relu(graph['conv1_1'], 2, 'conv1_2')
        graph['avgpool1'] = _avgpool(graph['conv1_2'])
        graph['conv2_1']  = _conv2d_relu(graph['avgpool1'], 5, 'conv2_1')
        graph['conv2_2']  = _conv2d_relu(graph['conv2_1'], 7, 'conv2_2')
        graph['avgpool2'] = _avgpool(graph['conv2_2'])
        graph['conv3_1']  = _conv2d_relu(graph['avgpool2'], 10, 'conv3_1')
        graph['conv3_2']  = _conv2d_relu(graph['conv3_1'], 12, 'conv3_2')
        graph['conv3_3']  = _conv2d_relu(graph['conv3_2'], 14, 'conv3_3')
        graph['conv3_4']  = _conv2d_relu(graph['conv3_3'], 16, 'conv3_4')
        graph['avgpool3'] = _avgpool(graph['conv3_4'])
        graph['conv4_1']  = _conv2d_relu(graph['avgpool3'], 19, 'conv4_1')
        graph['conv4_2']  = _conv2d_relu(graph['conv4_1'], 21, 'conv4_2')
        graph['conv4_3']  = _conv2d_relu(graph['conv4_2'], 23, 'conv4_3')
        graph['conv4_4']  = _conv2d_relu(graph['conv4_3'], 25, 'conv4_4')
```

```
        graph['avgpool4'] = _avgpool(graph['conv4_4'])
        graph['conv5_1']  = _conv2d_relu(graph['avgpool4'], 28, 'conv5_1')
        graph['conv5_2']  = _conv2d_relu(graph['conv5_1'], 30, 'conv5_2')
        graph['conv5_3']  = _conv2d_relu(graph['conv5_2'], 32, 'conv5_3')
        graph['conv5_4']  = _conv2d_relu(graph['conv5_3'], 34, 'conv5_4')
        graph['avgpool5'] = _avgpool(graph['conv5_4'])

        return graph

X = tf.placeholder(tf.float32, [None, image_size, image_size, 3])
encoded = vgg_endpoints(X - MEAN_VALUES)['conv5_3']
print(encoded)
```

```
Tensor("endpoints/Relu_14:0", shape=(?, 14, 14, 512), dtype=float32)
```

[7]:
```
k_initializer = tf.contrib.layers.xavier_initializer()
b_initializer = tf.constant_initializer(0.0)
e_initializer = tf.random_uniform_initializer(-1.0, 1.0)

def dense(inputs, units, activation=tf.nn.tanh, use_bias=True, name=None):
    return tf.layers.dense(inputs, units, activation, use_bias,
                           kernel_initializer=k_initializer,␣
 ↪bias_initializer=b_initializer, name=name)

def batch_norm(inputs, name):
    return tf.contrib.layers.batch_norm(inputs, decay=0.95, center=True,␣
 ↪scale=True, is_training=True,
                                        updates_collections=None, scope=name)

def dropout(inputs):
    return tf.layers.dropout(inputs, rate=0.5, training=True)

num_block = 14 * 14
num_filter = 512
hidden_size = 1024
embedding_size = 512

encoded = tf.reshape(encoded, [-1, num_block, num_filter]) # batch_size,␣
 ↪num_block, num_filter
contexts = batch_norm(encoded, 'contexts')

Y = tf.placeholder(tf.int32, [None, maxlen + 2])
Y_in = Y[:, :-1]
Y_out = Y[:, 1:]
mask = tf.to_float(tf.not_equal(Y_out, word2id['<pad>']))

with tf.variable_scope('initialize'):
```

```
    context_mean = tf.reduce_mean(contexts, 1)
    state = dense(context_mean, hidden_size, name='initial_state')
    memory = dense(context_mean, hidden_size, name='initial_memory')

with tf.variable_scope('embedding'):
    embeddings = tf.get_variable('weights', [len(word2id), embedding_size],␣
 ↪initializer=e_initializer)
    embedded = tf.nn.embedding_lookup(embeddings, Y_in)

with tf.variable_scope('projected'):
    projected_contexts = tf.reshape(contexts, [-1, num_filter]) # batch_size *␣
 ↪num_block, num_filter
    projected_contexts = dense(projected_contexts, num_filter, activation=None,␣
 ↪use_bias=False, name='projected_contexts')
    projected_contexts = tf.reshape(projected_contexts, [-1, num_block,␣
 ↪num_filter]) # batch_size, num_block, num_filter

lstm = tf.nn.rnn_cell.BasicLSTMCell(hidden_size)
loss = 0
alphas = []
```

[8]:
```
for t in range(maxlen + 1):
    with tf.variable_scope('attend'):
        h0 = dense(state, num_filter, activation=None, name='fc_state') #␣
 ↪batch_size, num_filter
        h0 = tf.nn.relu(projected_contexts + tf.expand_dims(h0, 1)) #␣
 ↪batch_size, num_block, num_filter
        h0 = tf.reshape(h0, [-1, num_filter]) # batch_size * num_block,␣
 ↪num_filter
        h0 = dense(h0, 1, activation=None, use_bias=False, name='fc_attention')␣
 ↪# batch_size * num_block, 1
        h0 = tf.reshape(h0, [-1, num_block]) # batch_size, num_block

        alpha = tf.nn.softmax(h0) # batch_size, num_block
        # contexts:                 batch_size, num_block, num_filter
        # tf.expand_dims(alpha, 2): batch_size, num_block, 1
        context = tf.reduce_sum(contexts * tf.expand_dims(alpha, 2), 1,␣
 ↪name='context') # batch_size, num_filter
        alphas.append(alpha)

    with tf.variable_scope('selector'):
        beta = dense(state, 1, activation=tf.nn.sigmoid, name='fc_beta') #␣
 ↪batch_size, 1
        context = tf.multiply(beta, context, name='selected_context')  #␣
 ↪batch_size, num_filter
```

```
    with tf.variable_scope('lstm'):
        h0 = tf.concat([embedded[:, t, :], context], 1) # batch_size,␣
↪embedding_size + num_filter
        _, (memory, state) = lstm(inputs=h0, state=[memory, state])

    with tf.variable_scope('decode'):
        h0 = dropout(state)
        h0 = dense(h0, embedding_size, activation=None, name='fc_logits_state')
        h0 += dense(context, embedding_size, activation=None, use_bias=False,␣
↪name='fc_logits_context')
        h0 += embedded[:, t, :]
        h0 = tf.nn.tanh(h0)

        h0 = dropout(h0)
        logits = dense(h0, len(word2id), activation=None, name='fc_logits')

    loss += tf.reduce_sum(tf.nn.
↪sparse_softmax_cross_entropy_with_logits(labels=Y_out[:, t], logits=logits)␣
↪* mask[:, t])
    tf.get_variable_scope().reuse_variables()
```

[9]:
```
alphas = tf.transpose(tf.stack(alphas), (1, 0, 2)) # batch_size, maxlen + 1,␣
↪num_block
alphas = tf.reduce_sum(alphas, 1) # batch_size, num_block
attention_loss = tf.reduce_sum(((maxlen + 1) / num_block - alphas) ** 2)
total_loss = (loss + attention_loss) / batch_size

with tf.variable_scope('optimizer', reuse=tf.AUTO_REUSE):
    global_step = tf.Variable(0, trainable=False)
    vars_t = [var for var in tf.trainable_variables() if not var.name.
↪startswith('endpoints')]
    train = tf.contrib.layers.optimize_loss(total_loss, global_step, 0.001,␣
↪'Adam', clip_gradients=5.0, variables=vars_t)
```

[ ]:
```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

saver = tf.train.Saver()
OUTPUT_DIR = 'model'
if not os.path.exists(OUTPUT_DIR):
    os.mkdir(OUTPUT_DIR)

tf.summary.scalar('losses/loss', loss)
tf.summary.scalar('losses/attention_loss', attention_loss)
tf.summary.scalar('losses/total_loss', total_loss)
summary = tf.summary.merge_all()
```

```
writer = tf.summary.FileWriter(OUTPUT_DIR)

epochs = 20
for e in range(epochs):
    train_ids, train_captions = shuffle(train_ids, train_captions)
    for i in tqdm(range(len(train_ids) // batch_size)):
        X_batch = np.array([train_dict[x] for x in train_ids[i * batch_size: i␣
↪* batch_size + batch_size]])
        Y_batch = train_captions[i * batch_size: i * batch_size + batch_size]

        _ = sess.run(train, feed_dict={X: X_batch, Y: Y_batch})

        if i > 0 and i % 100 == 0:
            writer.add_summary(sess.run(summary,
                                        feed_dict={X: X_batch, Y: Y_batch}),
                                        e * len(train_ids) // batch_size + i)
            writer.flush()

    saver.save(sess, os.path.join(OUTPUT_DIR, 'image_caption'))
```

```
100%|      | 3215/3215 [34:11<00:00,  1.65it/s]
 90%|      | 2903/3215 [30:41<03:43,  1.40it/s]
```

[ ]:

model2_validate

December 18, 2020

[ ]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.utils import shuffle
from imageio import imread
import scipy.io
import cv2
import os
import json
from tqdm import tqdm
import pickle
```

[ ]:
```
batch_size = 128
maxlen = 20
image_size = 224

MEAN_VALUES = np.array([123.68, 116.779, 103.939]).reshape((1, 1, 3))

def load_data(image_dir, annotation_path):
    with open(annotation_path, 'r') as fr:
        annotation = json.load(fr)

    ids = []
    captions = []
    image_dict = {}
    for i in tqdm(range(len(annotation['annotations']))):
        item = annotation['annotations'][i]
        caption = item['caption'].strip().lower()
        caption = caption.replace('.', '').replace(',', '').replace("'", '').
 ↪replace('"', '')
        caption = caption.replace('&', 'and').replace('(', '').replace(')', '').
 ↪replace('-', ' ').split()
        caption = [w for w in caption if len(w) > 0]

        if len(caption) <= maxlen:
            if not item['image_id'] in image_dict:
```

```
                img = imread(image_dir + '%012d.jpg' % item['image_id'])
                h = img.shape[0]
                w = img.shape[1]
                if h > w:
                    img = img[h // 2 - w // 2: h // 2 + w // 2, :]
                else:
                    img = img[:, w // 2 - h // 2: w // 2 + h // 2]
                img = cv2.resize(img, (image_size, image_size))

                if len(img.shape) < 3:
                    img = np.expand_dims(img, -1)
                    img = np.concatenate([img, img, img], axis=-1)

                image_dict[item['image_id']] = img

            ids.append(item['image_id'])
            captions.append(caption)

    return ids, captions, image_dict

val_json = 'data/val/captions_val2014.json'
val_ids, val_captions, val_dict = load_data('data/val/images/COCO_val2014_',␣
 ↪val_json)
print(len(val_ids))
```

[ ]:
```
gt = {}
for i in tqdm(range(len(val_ids))):
    val_id = val_ids[i]
    if not val_id in gt:
        gt[val_id] = []
    gt[val_id].append(' '.join(val_captions[i]) + ' .')
print(len(gt))
```

[ ]:
```
with open('dictionary.pkl', 'rb') as fr:
    [vocabulary, word2id, id2word] = pickle.load(fr)

def translate(ids):
    words = [id2word[i] for i in ids if i >= 3]
    return ' '.join(words) + ' .'
```

[ ]:
```
vgg = scipy.io.loadmat('imagenet-vgg-verydeep-19.mat')
vgg_layers = vgg['layers']

def vgg_endpoints(inputs, reuse=None):
    with tf.variable_scope('endpoints', reuse=reuse):
        def _weights(layer, expected_layer_name):
            W = vgg_layers[0][layer][0][0][2][0][0]
```

```
            b = vgg_layers[0][layer][0][0][2][0][1]
            layer_name = vgg_layers[0][layer][0][0][0][0]
            assert layer_name == expected_layer_name
            return W, b

        def _conv2d_relu(prev_layer, layer, layer_name):
            W, b = _weights(layer, layer_name)
            W = tf.constant(W)
            b = tf.constant(np.reshape(b, (b.size)))
            return tf.nn.relu(tf.nn.conv2d(prev_layer, filter=W, strides=[1, 1,␣
↪1, 1], padding='SAME') + b)

        def _avgpool(prev_layer):
            return tf.nn.avg_pool(prev_layer, ksize=[1, 2, 2, 1], strides=[1,␣
↪2, 2, 1], padding='SAME')

        graph = {}
        graph['conv1_1']  = _conv2d_relu(inputs, 0, 'conv1_1')
        graph['conv1_2']  = _conv2d_relu(graph['conv1_1'], 2, 'conv1_2')
        graph['avgpool1'] = _avgpool(graph['conv1_2'])
        graph['conv2_1']  = _conv2d_relu(graph['avgpool1'], 5, 'conv2_1')
        graph['conv2_2']  = _conv2d_relu(graph['conv2_1'], 7, 'conv2_2')
        graph['avgpool2'] = _avgpool(graph['conv2_2'])
        graph['conv3_1']  = _conv2d_relu(graph['avgpool2'], 10, 'conv3_1')
        graph['conv3_2']  = _conv2d_relu(graph['conv3_1'], 12, 'conv3_2')
        graph['conv3_3']  = _conv2d_relu(graph['conv3_2'], 14, 'conv3_3')
        graph['conv3_4']  = _conv2d_relu(graph['conv3_3'], 16, 'conv3_4')
        graph['avgpool3'] = _avgpool(graph['conv3_4'])
        graph['conv4_1']  = _conv2d_relu(graph['avgpool3'], 19, 'conv4_1')
        graph['conv4_2']  = _conv2d_relu(graph['conv4_1'], 21, 'conv4_2')
        graph['conv4_3']  = _conv2d_relu(graph['conv4_2'], 23, 'conv4_3')
        graph['conv4_4']  = _conv2d_relu(graph['conv4_3'], 25, 'conv4_4')
        graph['avgpool4'] = _avgpool(graph['conv4_4'])
        graph['conv5_1']  = _conv2d_relu(graph['avgpool4'], 28, 'conv5_1')
        graph['conv5_2']  = _conv2d_relu(graph['conv5_1'], 30, 'conv5_2')
        graph['conv5_3']  = _conv2d_relu(graph['conv5_2'], 32, 'conv5_3')
        graph['conv5_4']  = _conv2d_relu(graph['conv5_3'], 34, 'conv5_4')
        graph['avgpool5'] = _avgpool(graph['conv5_4'])

        return graph

X = tf.placeholder(tf.float32, [None, image_size, image_size, 3])
encoded = vgg_endpoints(X - MEAN_VALUES)['conv5_3']
print(encoded)
```

[ ]:
```
k_initializer = tf.contrib.layers.xavier_initializer()
b_initializer = tf.constant_initializer(0.0)
```

```
e_initializer = tf.random_uniform_initializer(-1.0, 1.0)

def dense(inputs, units, activation=tf.nn.tanh, use_bias=True, name=None):
    return tf.layers.dense(inputs, units, activation, use_bias,
                           kernel_initializer=k_initializer,␣
 ↪bias_initializer=b_initializer, name=name)

def batch_norm(inputs, name):
    return tf.contrib.layers.batch_norm(inputs, decay=0.95, center=True,␣
 ↪scale=True, is_training=False,
                                        updates_collections=None, scope=name)

def dropout(inputs):
    return tf.layers.dropout(inputs, rate=0.5, training=False)

num_block = 14 * 14
num_filter = 512
hidden_size = 1024
embedding_size = 512

encoded = tf.reshape(encoded, [-1, num_block, num_filter]) # batch_size,␣
 ↪num_block, num_filter
contexts = batch_norm(encoded, 'contexts')

with tf.variable_scope('initialize'):
    context_mean = tf.reduce_mean(contexts, 1)
    initial_state = dense(context_mean, hidden_size, name='initial_state')
    initial_memory = dense(context_mean, hidden_size, name='initial_memory')

contexts_phr = tf.placeholder(tf.float32, [None, num_block, num_filter])
last_memory = tf.placeholder(tf.float32, [None, hidden_size])
last_state = tf.placeholder(tf.float32, [None, hidden_size])
last_word = tf.placeholder(tf.int32, [None])

with tf.variable_scope('embedding'):
    embeddings = tf.get_variable('weights', [len(word2id), embedding_size],␣
 ↪initializer=e_initializer)
    embedded = tf.nn.embedding_lookup(embeddings, last_word)

with tf.variable_scope('projected'):
    projected_contexts = tf.reshape(contexts_phr, [-1, num_filter]) #␣
 ↪batch_size * num_block, num_filter
    projected_contexts = dense(projected_contexts, num_filter, activation=None,␣
 ↪use_bias=False, name='projected_contexts')
    projected_contexts = tf.reshape(projected_contexts, [-1, num_block,␣
 ↪num_filter]) # batch_size, num_block, num_filter
```

```
lstm = tf.nn.rnn_cell.BasicLSTMCell(hidden_size)
```

[ ]:
```
with tf.variable_scope('attend'):
    h0 = dense(last_state, num_filter, activation=None, name='fc_state') # batch_size, num_filter
    h0 = tf.nn.relu(projected_contexts + tf.expand_dims(h0, 1)) # batch_size, num_block, num_filter
    h0 = tf.reshape(h0, [-1, num_filter]) # batch_size * num_block, num_filter
    h0 = dense(h0, 1, activation=None, use_bias=False, name='fc_attention') # batch_size * num_block, 1
    h0 = tf.reshape(h0, [-1, num_block]) # batch_size, num_block

    alpha = tf.nn.softmax(h0) # batch_size, num_block
    # contexts:                 batch_size, num_block, num_filter
    # tf.expand_dims(alpha, 2): batch_size, num_block, 1
    context = tf.reduce_sum(contexts_phr * tf.expand_dims(alpha, 2), 1, name='context') # batch_size, num_filter

with tf.variable_scope('selector'):
    beta = dense(last_state, 1, activation=tf.nn.sigmoid, name='fc_beta') # batch_size, 1
    context = tf.multiply(beta, context, name='selected_context')  # batch_size, num_filter

with tf.variable_scope('lstm'):
    h0 = tf.concat([embedded, context], 1) # batch_size, embedding_size + num_filter
    _, (current_memory, current_state) = lstm(inputs=h0, state=[last_memory, last_state])

with tf.variable_scope('decode'):
    h0 = dropout(current_state)
    h0 = dense(h0, embedding_size, activation=None, name='fc_logits_state')
    h0 += dense(context, embedding_size, activation=None, use_bias=False, name='fc_logits_context')
    h0 += embedded
    h0 = tf.nn.tanh(h0)

    h0 = dropout(h0)
    logits = dense(h0, len(word2id), activation=None, name='fc_logits')
    probs = tf.nn.softmax(logits)
```

[ ]:
```
MODEL_DIR = 'model'
sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

```
saver = tf.train.Saver()
saver.restore(sess, tf.train.latest_checkpoint(MODEL_DIR))

beam_size = 1
id2sentence = {}

val_ids = list(set(val_ids))
if len(val_ids) % batch_size != 0:
    for i in range(batch_size - len(val_ids) % batch_size):
        val_ids.append(val_ids[0])
print(len(val_ids))

for i in tqdm(range(len(val_ids) // batch_size)):
    X_batch = np.array([val_dict[x] for x in val_ids[i * batch_size: i *␣
↪batch_size + batch_size]])
    contexts_, initial_memory_, initial_state_ = sess.run([contexts,␣
↪initial_memory, initial_state], feed_dict={X: X_batch})

    result = []
    complete = []
    for b in range(batch_size):
        result.append([{
            'sentence': [],
            'memory': initial_memory_[b],
            'state': initial_state_[b],
            'score': 1.0,
            'alphas': []
        }])
        complete.append([])

    for t in range(maxlen + 1):
        cache = [[] for b in range(batch_size)]
        step = 1 if t == 0 else beam_size
        for s in range(step):
            if t == 0:
                last_word_ = np.ones([batch_size], np.int32) *␣
↪word2id['<start>']
            else:
                last_word_ = np.array([result[b][s]['sentence'][-1] for b in␣
↪range(batch_size)], np.int32)

            last_memory_ = np.array([result[b][s]['memory'] for b in␣
↪range(batch_size)], np.float32)
            last_state_ = np.array([result[b][s]['state'] for b in␣
↪range(batch_size)], np.float32)
```

```
            current_memory_, current_state_, probs_, alpha_ = sess.run(
                [current_memory, current_state, probs, alpha], feed_dict={
                    contexts_phr: contexts_,
                    last_memory: last_memory_,
                    last_state: last_state_,
                    last_word: last_word_
                    })

            for b in range(batch_size):
                word_and_probs = [[w, p] for w, p in enumerate(probs_[b])]
                word_and_probs.sort(key=lambda x:-x[1])
                word_and_probs = word_and_probs[:beam_size + 1]

                for w, p in word_and_probs:
                    item = {
                        'sentence': result[b][s]['sentence'] + [w],
                        'memory': current_memory_[b],
                        'state': current_state_[b],
                        'score': result[b][s]['score'] * p,
                        'alphas': result[b][s]['alphas'] + [alpha_[b]]
                    }

                    if id2word[w] == '<end>':
                        complete[b].append(item)
                    else:
                        cache[b].append(item)

        for b in range(batch_size):
            cache[b].sort(key=lambda x:-x['score'])
            cache[b] = cache[b][:beam_size]
        result = cache.copy()

    for b in range(batch_size):
        if len(complete[b]) == 0:
            final_sentence = result[b][0]['sentence']
        else:
            final_sentence = complete[b][0]['sentence']

        val_id = val_ids[i * batch_size + b]
        if not val_id in id2sentence:
            id2sentence[val_id] = [translate(final_sentence)]

print(len(id2sentence))
```

[ ]:
```
with open('generated.txt', 'w') as fw:
    for i in id2sentence.keys():
```

```
        fw.write(str(i) + '^' + id2sentence[i][0] + '^' + '_'.join(gt[i]) +␣
↪'\n')
```

[2]:
```
from pycocoevalcap.bleu.bleu import Bleu
from pycocoevalcap.rouge.rouge import Rouge
from pycocoevalcap.cider.cider import Cider

id2sentence = {}
gt = {}
with open('generated.txt', 'r') as fr:
    lines = fr.readlines()
    for line in lines:
        line = line.strip('\n').split('^')
        i = line[0]
        id2sentence[i] = [line[1]]
        gt[i] = line[2].split('_')

scorers = [
    (Bleu(4), ['Bleu_1', 'Bleu_2', 'Bleu_3', 'Bleu_4']),
    (Rouge(), 'ROUGE_L'),
    (Cider(), 'CIDEr')
]

for scorer, name in scorers:
    score, _ = scorer.compute_score(gt, id2sentence)
    if type(score) == list:
        for n, s in zip(name, score):
            print(n, s)
    else:
        print(name, score)
```

```
{'testlen': 435873, 'reflen': 431236, 'guess': [435873, 395369, 354865, 314361],
'correct': [299794, 132419, 57743, 25805]}
ratio: 1.0107528128449363
Bleu_1 0.6878012632119891
Bleu_2 0.4799603146327394
Bleu_3 0.33466909679908385
Bleu_4 0.2355214731874305
ROUGE_L 0.5304592073317095
CIDEr 0.7292663614238141
```

[ ]:

model2_test

December 18, 2020

[2]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from imageio import imread
import scipy.io
import cv2
import pickle
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

[3]:
```
batch_size = 1
maxlen = 20
image_size = 224

MEAN_VALUES = np.array([123.68, 116.779, 103.939]).reshape((1, 1, 3))

with open('dictionary.pkl', 'rb') as fr:
    [vocabulary, word2id, id2word] = pickle.load(fr)
```

[4]:
```
def translate(ids):
    words = [id2word[i] for i in ids if i >= 3]
    return ' '.join(words) + ' .'
```

[5]:
```
vgg = scipy.io.loadmat('imagenet-vgg-verydeep-19.mat')
vgg_layers = vgg['layers']

def vgg_endpoints(inputs, reuse=None):
    with tf.variable_scope('endpoints', reuse=reuse):
        def _weights(layer, expected_layer_name):
            W = vgg_layers[0][layer][0][0][2][0][0]
            b = vgg_layers[0][layer][0][0][2][0][1]
            layer_name = vgg_layers[0][layer][0][0][0][0]
            assert layer_name == expected_layer_name
            return W, b

        def _conv2d_relu(prev_layer, layer, layer_name):
            W, b = _weights(layer, layer_name)
            W = tf.constant(W)
```

```
            b = tf.constant(np.reshape(b, (b.size)))
            return tf.nn.relu(tf.nn.conv2d(prev_layer, filter=W, strides=[1, 1,␣
↪1, 1], padding='SAME') + b)

        def _avgpool(prev_layer):
            return tf.nn.avg_pool(prev_layer, ksize=[1, 2, 2, 1], strides=[1,␣
↪2, 2, 1], padding='SAME')

        graph = {}
        graph['conv1_1']  = _conv2d_relu(inputs, 0, 'conv1_1')
        graph['conv1_2']  = _conv2d_relu(graph['conv1_1'], 2, 'conv1_2')
        graph['avgpool1'] = _avgpool(graph['conv1_2'])
        graph['conv2_1']  = _conv2d_relu(graph['avgpool1'], 5, 'conv2_1')
        graph['conv2_2']  = _conv2d_relu(graph['conv2_1'], 7, 'conv2_2')
        graph['avgpool2'] = _avgpool(graph['conv2_2'])
        graph['conv3_1']  = _conv2d_relu(graph['avgpool2'], 10, 'conv3_1')
        graph['conv3_2']  = _conv2d_relu(graph['conv3_1'], 12, 'conv3_2')
        graph['conv3_3']  = _conv2d_relu(graph['conv3_2'], 14, 'conv3_3')
        graph['conv3_4']  = _conv2d_relu(graph['conv3_3'], 16, 'conv3_4')
        graph['avgpool3'] = _avgpool(graph['conv3_4'])
        graph['conv4_1']  = _conv2d_relu(graph['avgpool3'], 19, 'conv4_1')
        graph['conv4_2']  = _conv2d_relu(graph['conv4_1'], 21, 'conv4_2')
        graph['conv4_3']  = _conv2d_relu(graph['conv4_2'], 23, 'conv4_3')
        graph['conv4_4']  = _conv2d_relu(graph['conv4_3'], 25, 'conv4_4')
        graph['avgpool4'] = _avgpool(graph['conv4_4'])
        graph['conv5_1']  = _conv2d_relu(graph['avgpool4'], 28, 'conv5_1')
        graph['conv5_2']  = _conv2d_relu(graph['conv5_1'], 30, 'conv5_2')
        graph['conv5_3']  = _conv2d_relu(graph['conv5_2'], 32, 'conv5_3')
        graph['conv5_4']  = _conv2d_relu(graph['conv5_3'], 34, 'conv5_4')
        graph['avgpool5'] = _avgpool(graph['conv5_4'])

        return graph
```

[6]:
```
X = tf.placeholder(tf.float32, [None, image_size, image_size, 3])
encoded = vgg_endpoints(X - MEAN_VALUES)['conv5_3']

k_initializer = tf.contrib.layers.xavier_initializer()
b_initializer = tf.constant_initializer(0.0)
e_initializer = tf.random_uniform_initializer(-1.0, 1.0)
```

[7]:
```
def dense(inputs, units, activation=tf.nn.tanh, use_bias=True, name=None):
    return tf.layers.dense(inputs, units, activation, use_bias,
                           kernel_initializer=k_initializer,␣
↪bias_initializer=b_initializer, name=name)
```

[8]:
```
def batch_norm(inputs, name):
    return tf.contrib.layers.batch_norm(inputs, decay=0.95, center=True,␣
 ↪scale=True, is_training=False,
                                        updates_collections=None, scope=name)
```

[9]:
```
def dropout(inputs):
    return tf.layers.dropout(inputs, rate=0.5, training=False)
```

[10]:
```
num_block = 14 * 14
num_filter = 512
hidden_size = 1024
embedding_size = 512

encoded = tf.reshape(encoded, [-1, num_block, num_filter]) # batch_size,␣
 ↪num_block, num_filter
contexts = batch_norm(encoded, 'contexts')

with tf.variable_scope('initialize'):
    context_mean = tf.reduce_mean(contexts, 1)
    initial_state = dense(context_mean, hidden_size, name='initial_state')
    initial_memory = dense(context_mean, hidden_size, name='initial_memory')

contexts_phr = tf.placeholder(tf.float32, [None, num_block, num_filter])
last_memory = tf.placeholder(tf.float32, [None, hidden_size])
last_state = tf.placeholder(tf.float32, [None, hidden_size])
last_word = tf.placeholder(tf.int32, [None])

with tf.variable_scope('embedding'):
    embeddings = tf.get_variable('weights', [len(word2id), embedding_size],␣
 ↪initializer=e_initializer)
    embedded = tf.nn.embedding_lookup(embeddings, last_word)

with tf.variable_scope('projected'):
    projected_contexts = tf.reshape(contexts_phr, [-1, num_filter]) #␣
 ↪batch_size * num_block, num_filter
    projected_contexts = dense(projected_contexts, num_filter, activation=None,␣
 ↪use_bias=False, name='projected_contexts')
    projected_contexts = tf.reshape(projected_contexts, [-1, num_block,␣
 ↪num_filter]) # batch_size, num_block, num_filter

lstm = tf.nn.rnn_cell.BasicLSTMCell(hidden_size)

with tf.variable_scope('attend'):
    h0 = dense(last_state, num_filter, activation=None, name='fc_state') #␣
 ↪batch_size, num_filter
```

```
    h0 = tf.nn.relu(projected_contexts + tf.expand_dims(h0, 1)) # batch_size,␣
↪num_block, num_filter
    h0 = tf.reshape(h0, [-1, num_filter]) # batch_size * num_block, num_filter
    h0 = dense(h0, 1, activation=None, use_bias=False, name='fc_attention') #␣
↪batch_size * num_block, 1
    h0 = tf.reshape(h0, [-1, num_block]) # batch_size, num_block

    alpha = tf.nn.softmax(h0) # batch_size, num_block
    # contexts:                 batch_size, num_block, num_filter
    # tf.expand_dims(alpha, 2): batch_size, num_block, 1
    context = tf.reduce_sum(contexts_phr * tf.expand_dims(alpha, 2), 1,␣
↪name='context') # batch_size, num_filter

with tf.variable_scope('selector'):
    beta = dense(last_state, 1, activation=tf.nn.sigmoid, name='fc_beta') #␣
↪batch_size, 1
    context = tf.multiply(beta, context, name='selected_context')  #␣
↪batch_size, num_filter

with tf.variable_scope('lstm'):
    h0 = tf.concat([embedded, context], 1) # batch_size, embedding_size +␣
↪num_filter
    _, (current_memory, current_state) = lstm(inputs=h0, state=[last_memory,␣
↪last_state])

with tf.variable_scope('decode'):
    h0 = dropout(current_state)
    h0 = dense(h0, embedding_size, activation=None, name='fc_logits_state')
    h0 += dense(context, embedding_size, activation=None, use_bias=False,␣
↪name='fc_logits_context')
    h0 += embedded
    h0 = tf.nn.tanh(h0)

    h0 = dropout(h0)
    logits = dense(h0, len(word2id), activation=None, name='fc_logits')
    probs = tf.nn.softmax(logits)
```

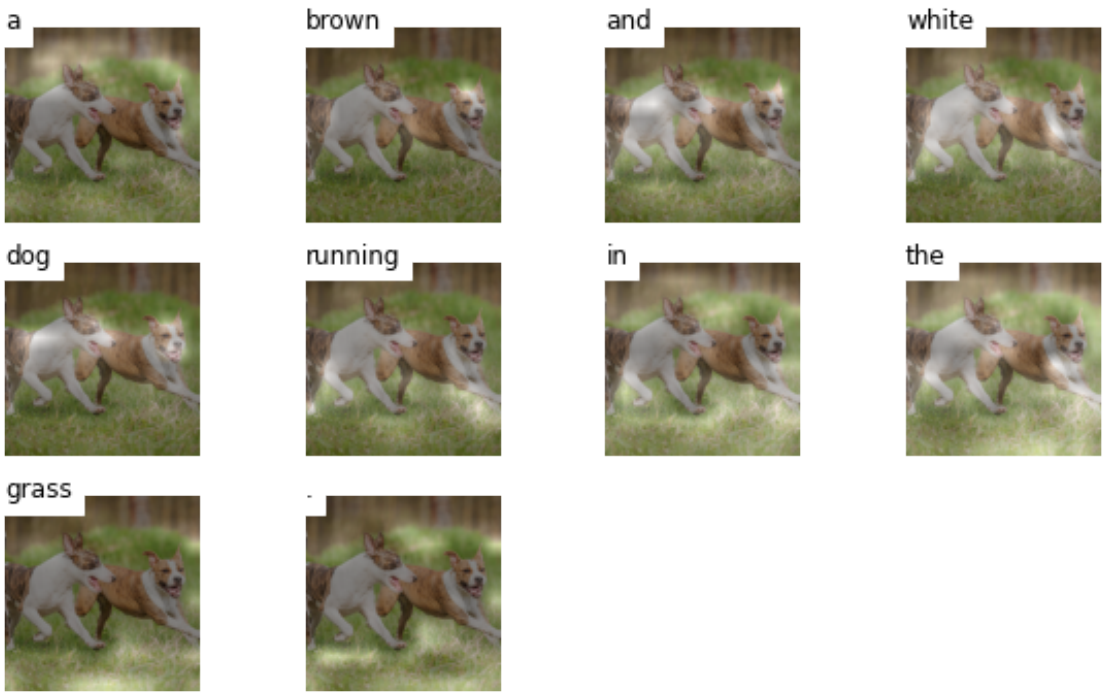
[11]:
```
MODEL_DIR = 'model'
sess = tf.Session()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
saver.restore(sess, tf.train.latest_checkpoint(MODEL_DIR))

beam_size = 3
img = imread('sample.jpg')
if img.shape[-1] == 4:
```

```
    img = img[:, :, :-1]
h = img.shape[0]
w = img.shape[1]
if h > w:
    img = img[h // 2 - w // 2: h // 2 + w // 2, :]
else:
    img = img[:, w // 2 - h // 2: w // 2 + h // 2]
img = cv2.resize(img, (image_size, image_size))
X_data = np.expand_dims(img, 0)

contexts_, initial_memory_, initial_state_ = sess.run([contexts,␣
↪initial_memory, initial_state], feed_dict={X: X_data})

result = [{
    'sentence': [],
    'memory': initial_memory_[0],
    'state': initial_state_[0],
    'score': 1.0,
    'alphas': []
}]
complete = []
for t in range(maxlen + 1):
    cache = []
    step = 1 if t == 0 else beam_size
    for s in range(step):
        if t == 0:
            last_word_ = np.ones([batch_size], np.int32) * word2id['<start>']
        else:
            last_word_ = np.array([result[s]['sentence'][-1]], np.int32)

        last_memory_ = np.array([result[s]['memory']], np.float32)
        last_state_ = np.array([result[s]['state']], np.float32)

        current_memory_, current_state_, probs_, alpha_ = sess.run(
            [current_memory, current_state, probs, alpha], feed_dict={
                contexts_phr: contexts_,
                last_memory: last_memory_,
                last_state: last_state_,
                last_word: last_word_
                })

        word_and_probs = [[w, p] for w, p in enumerate(probs_[0])]
        word_and_probs.sort(key=lambda x:-x[1])
        word_and_probs = word_and_probs[:beam_size + 1]

        for w, p in word_and_probs:
            item = {
```

```
                'sentence': result[s]['sentence'] + [w],
                'memory': current_memory_[0],
                'state': current_state_[0],
                'score': result[s]['score'] * p,
                'alphas': result[s]['alphas'] + [alpha_[0]]
            }
            if id2word[w] == '<end>':
                complete.append(item)
            else:
                cache.append(item)

    cache.sort(key=lambda x:-x['score'])
    cache = cache[:beam_size]
    result = cache.copy()

if len(complete) == 0:
    final_sentence = result[0]['sentence']
    alphas = result[0]['alphas']
else:
    final_sentence = complete[0]['sentence']
    alphas = complete[0]['alphas']

sentence = translate(final_sentence)
print(sentence)
sentence = sentence.split(' ')
```

```
a brown and white dog running in the grass .
```

[12]:
```
img = (img - img.min()) / (img.max() - img.min())
n = int(np.ceil(np.sqrt(len(sentence))))
plt.figure(figsize=(10, 8))
for i in range(len(sentence)):
    word = sentence[i]
    a = np.reshape(alphas[i], (14, 14))
    a = cv2.resize(a, (image_size, image_size))
    a = np.expand_dims(a, -1)
    a = (a - a.min()) / (a.max() - a.min())
    combine = 0.5 * img +  0.5 * a
    plt.subplot(n, n, i + 1)
    plt.text(0, 1, word, color='black', backgroundcolor='white', fontsize=12)
    plt.imshow(combine)
    plt.axis('off')
plt.show()
```


a
brown
and
white
dog
running
in
the
grass
.

[ ]: