**Practical Assignment**

**Objective: - Visual Question Answering**



**Dataset Link: -**

[**https://visualqa.org/vqa\_v1\_download.html**](https://visualqa.org/vqa_v1_download.html)

**Task: -** Create a Web Application using Flask. Use the end user should be able to upload an image and ask questions and get the answers.

**Deployment: -** Any Free Platform(Try to look out for free options.)

**Assignment Submission: -** Only submit the hosted app link. OR GitHub Link

from .layers import ContextVector, PhraseLevelFeatures, AttentionMaps

import tensorflow as tf

from tensorflow.keras.layers import Dense, Embedding, Conv1D, Dropout, Input, LSTM, concatenate

from tensorflow.keras.models import Model

def build\_model(max\_answers, max\_seq\_len, vocab\_size, dim\_d, dim\_k, l\_rate, d\_rate, reg\_value):

"""

Defines the Keras model.

Arguments

----------

max\_answers : Number of output targets of the model.

max\_seq\_len : Length of input sequences

vocab\_size : Size of the vocabulary, i.e. maximum integer index + 1.

dim\_d : Hidden dimension

dim\_k : Hidden attention dimension

l\_rate : Learning rate for the model

d\_rate : Dropout rate

reg\_value : Regularization value

Returns

----------

Returns the Keras model.

"""

# inputs

image\_input = Input(shape=(49, 512, ), name='Image\_Input')

ques\_input = Input(shape=(22, ), name='Question\_Input')

# image feature; (Wb)V # [b, 49, dim\_d]

image\_feat = Dense(dim\_d, activation=None, name='Image\_Feat\_Dense',\

kernel\_regularizer=tf.keras.regularizers.l2(reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=1))(image\_input)

image\_feat = Dropout(d\_rate, seed=1)(image\_feat)

# word level

ques\_feat\_w = Embedding(input\_dim=vocab\_size, output\_dim=dim\_d, input\_length=max\_seq\_len,\

mask\_zero=True)(ques\_input)

Hv\_w, Hq\_w = AttentionMaps(dim\_k, reg\_value, name='AttentionMaps\_Word')(image\_feat, ques\_feat\_w)

v\_w, q\_w = ContextVector(reg\_value, name='ContextVector\_Word')(image\_feat, ques\_feat\_w, Hv\_w, Hq\_w)

feat\_w = tf.add(v\_w,q\_w)

h\_w = Dense(dim\_d, activation='tanh', name='h\_w\_Dense',\

kernel\_regularizer=tf.keras.regularizers.l2(reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=13))(feat\_w)

# phrase level

ques\_feat\_p = PhraseLevelFeatures(dim\_d, name='PhraseLevelFeatures')(ques\_feat\_w)

Hv\_p, Hq\_p = AttentionMaps(dim\_k, reg\_value, name='AttentionMaps\_Phrase')(image\_feat, ques\_feat\_p)

v\_p, q\_p = ContextVector(reg\_value, name='ContextVector\_Phrase')(image\_feat, ques\_feat\_p, Hv\_p, Hq\_p)

feat\_p = concatenate([tf.add(v\_p,q\_p), h\_w], -1)

h\_p = Dense(dim\_d, activation='tanh', name='h\_p\_Dense',\

kernel\_regularizer=tf.keras.regularizers.l2(reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=14))(feat\_p)

# sentence level

ques\_feat\_s = LSTM(dim\_d, return\_sequences=True, input\_shape=(None, max\_seq\_len, dim\_d),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=16))(ques\_feat\_p)

Hv\_s, Hq\_s = AttentionMaps(dim\_k, reg\_value, name='AttentionMaps\_Sent')(image\_feat, ques\_feat\_s)

v\_s, q\_s = ContextVector(reg\_value, name='ContextVector\_Sent')(image\_feat, ques\_feat\_p, Hv\_s, Hq\_s)

feat\_s = concatenate([tf.add(v\_s,q\_s), h\_p], -1)

h\_s = Dense(2\*dim\_d, activation='tanh', name='h\_s\_Dense',\

kernel\_regularizer=tf.keras.regularizers.l2(reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=15))(feat\_s)

z = Dense(2\*dim\_d, activation='tanh', name='z\_Dense',\

kernel\_regularizer=tf.keras.regularizers.l2(reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=16))(h\_s)

z = Dropout(d\_rate, seed=16)(z)

# result

result = Dense(max\_answers, activation='softmax')(z)

model = Model(inputs=[image\_input, ques\_input], outputs=result)

return model

""" Custom Layers:

"""

import tensorflow as tf

from tensorflow.keras.layers import Dense, Conv1D

class AttentionMaps(tf.keras.layers.Layer):

def \_\_init\_\_(self, dim\_k, reg\_value, \*\*kwargs):

super(AttentionMaps, self).\_\_init\_\_(\*\*kwargs)

self.dim\_k = dim\_k

self.reg\_value = reg\_value

self.Wv = Dense(self.dim\_k, activation=None,\

kernel\_regularizer=tf.keras.regularizers.l2(self.reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=2))

self.Wq = Dense(self.dim\_k, activation=None,\

kernel\_regularizer=tf.keras.regularizers.l2(self.reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=3))

def call(self, image\_feat, ques\_feat):

"""

The main logic of this layer.

"""

# Affinity Matrix C

# (QT)(Wb)V

C = tf.matmul(ques\_feat, tf.transpose(image\_feat, perm=[0,2,1])) # [b, 23, 49]

# tanh((QT)(Wb)V)

C = tf.keras.activations.tanh(C)

# (Wv)V

WvV = self.Wv(image\_feat) # [b, 49, dim\_k]

# (Wq)Q

WqQ = self.Wq(ques\_feat) # [b, 23, dim\_k]

# ((Wq)Q)C

WqQ\_C = tf.matmul(tf.transpose(WqQ, perm=[0,2,1]), C) # [b, k, 49]

WqQ\_C = tf.transpose(WqQ\_C, perm =[0,2,1]) # [b, 49, k]

# ((Wv)V)CT # [b, k, 23]

WvV\_C = tf.matmul(tf.transpose(WvV, perm=[0,2,1]), tf.transpose(C, perm=[0,2,1]))

WvV\_C = tf.transpose(WvV\_C, perm =[0,2,1]) # [b, 23, k]

#---------------image attention map------------------

# We find "Hv = tanh((Wv)V + ((Wq)Q)C)" ; H\_v shape [49, k]

H\_v = WvV + WqQ\_C # (Wv)V + ((Wq)Q)C

H\_v = tf.keras.activations.tanh(H\_v) # tanh((Wv)V + ((Wq)Q)C)

#---------------question attention map---------------

# We find "Hq = tanh((Wq)Q + ((Wv)V)CT)" ; H\_q shape [23, k]

H\_q = WqQ + WvV\_C # (Wq)Q + ((Wv)V)CT

H\_q = tf.keras.activations.tanh(H\_q) # tanh((Wq)Q + ((Wv)V)CT)

return [H\_v, H\_q] # [b, 49, k], [b, 23, k]

def get\_config(self):

"""

This method collects the input shape and other information about the layer.

"""

config = {

'dim\_k': self.dim\_k,

'reg\_value': self.reg\_value

}

base\_config = super(AttentionMaps, self).get\_config()

return dict(list(base\_config.items()) + list(config.items()))

class ContextVector(tf.keras.layers.Layer):

def \_\_init\_\_(self, reg\_value, \*\*kwargs):

super(ContextVector, self).\_\_init\_\_(\*\*kwargs)

self.reg\_value = reg\_value

self.w\_hv = Dense(1, activation='softmax',\

kernel\_regularizer=tf.keras.regularizers.l2(self.reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=4))

self.w\_hq = Dense(1, activation='softmax',\

kernel\_regularizer=tf.keras.regularizers.l2(self.reg\_value),\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=5))

def call(self, image\_feat, ques\_feat, H\_v, H\_q):

"""

The main logic of this layer.

"""

# attention probabilities of each image region vn; a\_v = softmax(wT\_hv \* H\_v)

a\_v = self.w\_hv(H\_v) # [b, 49, 1]

# attention probabilities of each word qt ; a\_q = softmax(wT\_hq \* H\_q)

a\_q = self.w\_hq(H\_q) # [b, 23, 1]

# context vector for image

v = a\_v \* image\_feat # [b, 49, dim\_d]

v = tf.reduce\_sum(v, 1) # [b, dim\_d]

# context vector for question

q = a\_q \* ques\_feat # [b, 23, dim\_d]

q = tf.reduce\_sum(q, 1) # [b, dim\_d]

return [v, q]

def get\_config(self):

"""

This method collects the input shape and other information about the layer.

"""

config = {

'reg\_value': self.reg\_value

}

base\_config = super(ContextVector, self).get\_config()

return dict(list(base\_config.items()) + list(config.items()))

class PhraseLevelFeatures(tf.keras.layers.Layer):

def \_\_init\_\_(self, dim\_d, \*\*kwargs):

super().\_\_init\_\_(\*\*kwargs)

self.dim\_d = dim\_d

self.conv\_unigram = Conv1D(dim\_d, kernel\_size=1, strides=1,\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=6))

self.conv\_bigram = Conv1D(dim\_d, kernel\_size=2, strides=1, padding='same',\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=7))

self.conv\_trigram = Conv1D(dim\_d, kernel\_size=3, strides=1, padding='same',\

kernel\_initializer=tf.keras.initializers.glorot\_uniform(seed=8))

def call(self, word\_feat):

"""

The main logic of this layer.

Compute the n-gram phrase embeddings (n=1,2,3)

"""

# phrase level unigram features

x\_uni = self.conv\_unigram(word\_feat) # [b, 23, dim\_d]

# phrase level bigram features

x\_bi = self.conv\_bigram(word\_feat) # [b, 23, dim\_d]

# phrase level trigram features

x\_tri = self.conv\_trigram(word\_feat) # [b, 23, dim\_d]

# Concat

x = tf.concat([tf.expand\_dims(x\_uni, -1),\

tf.expand\_dims(x\_bi, -1),\

tf.expand\_dims(x\_tri, -1)], -1) # [b, 23, dim\_d, 3]

# https://stackoverflow.com/a/36853403

# Max-pool across n-gram features; over-all phrase level feature

x = tf.reduce\_max(x, -1) # [b, 23, dim\_d]

print(x)

return x

def get\_config(self):

"""

This method collects the input shape and other information about the layer.

"""

config = {

'dim\_d': self.dim\_d

}

base\_config = super().get\_config()

return dict(list(base\_config.items()) + list(config.items()))

import tensorflow as tf

import numpy as np

import re

from tensorflow.keras.preprocessing import image

from tensorflow.keras.applications.vgg19 import preprocess\_input

from tensorflow.keras import backend as K

K.set\_image\_data\_format('channels\_first')

def image\_feature\_extractor(image\_file, model):

"""

Extracts (512, 7, 7)-dimensional CNN features

Input:

image\_file: image filenames

Returns:

(512, 7, 7)-dimensional CNN features

"""

# load the input image using the Keras helper utility while ensuring the image is resized to 224x224 pixels

img = image.load\_img(image\_file, target\_size=(224, 224))

img = image.img\_to\_array(img)

# preprocess the image by

# (1) expanding the dimensions to include batch dim and

# (2) subtracting the mean RGB pixel intensity from the ImageNet dataset

img = np.expand\_dims(img, axis=0)

img = preprocess\_input(img)

# pass the images through the network and use the outputs as our actual features

print(img.shape)

features = model.predict(img) # (BATCH\_SIZE, 512, 7, 7)

features = tf.reshape(features, (features.shape[0], features.shape[1], -1)) # (BATCH\_SIZE, 512, 49)

features = tf.transpose(features, perm =[0,2,1]) # (BATCH\_SIZE, 49, 512)

return features

# https://github.com/zcyang/imageqa-san

def process\_sentence(sentence):

"""

Cleans a given raw sentence

Input:

sentence: a raw sentence

Returns:

Returns the cleaned version of the sentence

"""

# remove the character ".", except from floating numbers

periodStrip = re.compile("(?!<=\d)(\.)(?!\d)")

# remove any "," between digits, eg: 5,6

commaStrip = re.compile("(\d)(\,)(\d)")

# list of punctuations to remove

punct = [';', r"/", '[', ']', '"', '{', '}',

'(', ')', '=', '+', '\\', '\_', '-',

'\*', ':', '^', '%', '$', '#', '&',

'>', '<', '@', '`', ',', '?', '!']

# contraction mappings

contractions = {"aint": "ain't", "arent": "aren't", "cant": "can't", "couldve": "could've", "couldnt": "couldn't", \

"couldn'tve": "couldn't've", "couldnt've": "couldn't've", "didnt": "didn't", "doesnt": "doesn't", "dont": "don't", "hadnt": "hadn't", \

"hadnt've": "hadn't've", "hadn'tve": "hadn't've", "hasnt": "hasn't", "havent": "haven't", "hed": "he'd", "hed've": "he'd've", \

"he'dve": "he'd've", "hes": "he's", "howd": "how'd", "howll": "how'll", "hows": "how's", "Id've": "I'd've", "I'dve": "I'd've", \

"Im": "I'm", "Ive": "I've", "isnt": "isn't", "itd": "it'd", "itd've": "it'd've", "it'dve": "it'd've", "itll": "it'll", "let's": "let's", \

"maam": "ma'am", "mightnt": "mightn't", "mightnt've": "mightn't've", "mightn'tve": "mightn't've", "mightve": "might've", \

"mustnt": "mustn't", "mustve": "must've", "neednt": "needn't", "notve": "not've", "oclock": "o'clock", "oughtnt": "oughtn't", \

"ow's'at": "'ow's'at", "'ows'at": "'ow's'at", "'ow'sat": "'ow's'at", "shant": "shan't", "shed've": "she'd've", "she'dve": "she'd've", \

"she's": "she's", "shouldve": "should've", "shouldnt": "shouldn't", "shouldnt've": "shouldn't've", "shouldn'tve": "shouldn't've", \

"somebody'd": "somebodyd", "somebodyd've": "somebody'd've", "somebody'dve": "somebody'd've", "somebodyll": "somebody'll", \

"somebodys": "somebody's", "someoned": "someone'd", "someoned've": "someone'd've", "someone'dve": "someone'd've", \

"someonell": "someone'll", "someones": "someone's", "somethingd": "something'd", "somethingd've": "something'd've", \

"something'dve": "something'd've", "somethingll": "something'll", "thats": "that's", "thered": "there'd", "thered've": "there'd've", \

"there'dve": "there'd've", "therere": "there're", "theres": "there's", "theyd": "they'd", "theyd've": "they'd've", \

"they'dve": "they'd've", "theyll": "they'll", "theyre": "they're", "theyve": "they've", "twas": "'twas", "wasnt": "wasn't", \

"wed've": "we'd've", "we'dve": "we'd've", "weve": "we've", "werent": "weren't", "whatll": "what'll", "whatre": "what're", \

"whats": "what's", "whatve": "what've", "whens": "when's", "whered": "where'd", "where's": "where is", "whereve": "where've", \

"whod": "who'd", "whod've": "who'd've", "who'dve": "who'd've", "wholl": "who'll", "whos": "who's", "whove": "who've", "whyll": "why'll", \

"whyre": "why're", "whys": "why's", "wont": "won't", "wouldve": "would've", "wouldnt": "wouldn't", "wouldnt've": "wouldn't've", \

"wouldn'tve": "wouldn't've", "yall": "y'all", "yall'll": "y'all'll", "y'allll": "y'all'll", "yall'd've": "y'all'd've", \

"y'alld've": "y'all'd've", "y'all'dve": "y'all'd've", "youd": "you'd", "youd've": "you'd've", "you'dve": "you'd've", \

"youll": "you'll", "youre": "you're", "youve": "you've"}

# replace new line with a white space

inText = sentence.replace('\n', ' ')

# replace multiple white space with single white space

inText = inText.replace('\t', ' ')

inText = inText.strip()

outText = inText

for p in punct:

if (p + ' ' in inText or ' ' + p in inText) or \

(re.search(commaStrip, inText) != None):

outText = outText.replace(p, '')

else:

outText = outText.replace(p, ' ')

outText = periodStrip.sub("", outText, re.UNICODE)

outText = outText.lower().split()

for wordId, word in enumerate(outText):

if word in contractions:

outText[wordId] = contractions[word]

outText = ' '.join(outText)

return outText

def predict\_answers(img\_feat, ques\_feat, model, labelencoder):

"""

Prediction function

"""

preds = model([img\_feat, ques\_feat])

y\_classe = tf.argmax(preds, axis=1, output\_type=tf.int32)

y\_predict = (labelencoder.inverse\_transform(y\_classe))

return y\_predict

import joblib

# load the pre-trained tf.Keras tokenizer

with open('pickles/text\_tokenizer.pkl', 'rb') as f:

tok = joblib.load(f)

# load the pre-trained scikit-learn LabelEncoder object

with open('pickles/labelencoder.pkl', 'rb') as f:

labelencoder = joblib.load(f)

def predict\_function(image\_input, question\_input, model, tokenizer, labelencoder):

"""

This function include the entire pipeline, from taking raw data as input,

data preprocessing and then making final predictions.

Inputs:

image\_input : List of image files

question\_input : List of raw question data

model : Keras model object

tokenizer : pre-trained tf.Keras tokenizer

labelencoder : pre-trained scikit-learn LabelEncoder object

Returns:

Predictions on the raw data

"""

MAX\_LEN = 22

#1 --- Extract Image features

print('1/4 Extracting Image Features')

img\_feat = image\_feature\_extractor(image\_input)

#2 --- Clean the questions.

print('2/4 Cleaning the questions')

questions\_processed = pd.Series(question\_input).apply(process\_sentence)

#3 --- Tokenize the question data using a pre-trained tokenizer and pad them

print('3/4 Tokenizing and Padding the questions data')

question\_data = tok.texts\_to\_sequences(questions\_processed)

question\_data = sequence.pad\_sequences(question\_data, \

maxlen=MAX\_LEN,\

padding='post')

#4 --- Predict the answers

print('4/4 Predicting the answers')

y\_predict = predict\_answers(img\_feat, question\_data, model, labelencoder)

return y\_predict