T.C.
ANKARA YILDIRIM BEYAZIT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

NATURAL LANGUAGE PROCESSING COURSE TERM PROJECT

ADVISOR
LECT. YUSUF EVREN AYKAÇ

ANKARA - 2024

T.C.
ANKARA YILDIRIM BEYAZIT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

NATURAL LANGUAGE PROCESSING COURSE
TERM PROJECT

INVESTIGATION OF COCO DATASET AND THE VGG16 MODEL FOR IMAGE CLASSIFICATION

DİLARA TOSUN
ENES SAPAN
İREM ERDÖL

ADVISOR
LECT. YUSUF EVREN AYKAÇ

ANKARA - 2024

Final Project Report

Dilara TOSUN
20050111041

ENES SAPAN
22050151002

İrem ERDÖL
21050111055

Abstract - This research project aims to investigate the application and examination of a Jupyter notebook code for picture classification using the VGG16 model and the COCO dataset. The COCO dataset is a large-scale collection of labeled photos with 80 object categories that is intended for use in object recognition, segmentation, and captioning. For feature extraction and classification, the VGG16 model is used. This report provides a detailed analysis of the preprocessing steps, feature extraction process, model building, training, and evaluation of the model.

I. INTRODUCTION

Before starting our investigation we need to give detailed information about our tool that we use. First one is COCO dataset. COCO dataset is a large collection of labeled images intended for assisting computer vision systems in object recognition and description. More than 200,000 photos featuring 1.5 million items, including toys, food, and animals, are available. This dataset is unique because it displays objects in various contexts and their relationships with one another. It has 80 object categories and 91 stuff categories. Also it has 5 captures for per image.

The second one is VGG16 model. VGG16 is a deep neural network model used for image classification tasks. It consist of 16 layers, which each work process images one at a time to increase predict consist of 16 layers of artificial neurons, '16' refers to the network's 16 layers that have weights. This network is a pretty large network, and it has about 138 million parameters. VGG16 is very effective for various image classification tasks and is frequently used for transfer learning in computer vision applications. The pre-trained version of the VGG16 network is trained on ImageNet visual database. VGG16 can be applied to determine whether an image contains certain items, animals, plants and more.

8 STEPS FOR IMPLEMENTING VGG16 IN KEARS

1)Import the libraries for VGG16.
2)Create an object for training and testing data.
3)Initialize the model,
4)Pass the data to the dense layer.
5)Compile the model.
6)Import libraries to monitor and control training.
7)Visualize the training/validation data.
8)Test your model.

The disadvantage of VGG16 model is : The VGG16 model may require an enormous amount of time and computer power to train and deploy from scratch because of its size and quantity of parameters.

II. CODE ANALYSIS

A) Preprocessing

```
from tensorflow.keras.applications.vgg16 import VGG16
import os
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from pycocotools.coco import COCO
```

Initially, we begin by importing essential libraries and packages necessary for the implementation of our functions. This step aims to enhance the clarity and comprehensibility of our code.

1TensorFlow: It is an open-source machine learning framework, offers a robust ecosystem of tools and libraries tailored to assist researchers and developers in constructing machine learning models.

2 Keras: Keras is a high-level neural networks API, written in Python. It allows for easy and fast experimentation with deep neural networks.

3 **VGG16**: VGG16 is a convolutional neural network architecture. It has 16 layers and is known for its simplicity and effectiveness. It has been trained on a large dataset containing millions of images and has been widely used in various computer vision tasks, including image classification, object detection, and feature extraction.

4 os: The os module in Python provides a way of using operating system-dependent functionality. It allows you to interact with the operating system, such as creating or deleting directories, accessing files, and managing paths. Since we do not download coco dataset from the code (it can be understanble in the following sections) we import this module to access coco dataset. In this way, our code works a bir faster.

5 PIL (Python Imaging Library) : PIL is a library in Python used for opening, manipulating, and saving many different image file formats. In our code, the Image module from PIL is used to work with images.

6 NumPy: NumPy is a fundamental package for scientific computing with Python. It provides support for multidimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

7 Matplotlib: Matplotlib is a plotting library for Python and provides a MATLAB-like interface for creating plots and visualizations.

8 pycocotools: The pycocotools package is a Python API for the Microsoft Common Objects in Context (COCO) dataset. It provides a set of tools for accessing and working with the COCO dataset, including loading annotations, accessing images, and evaluating results.

Figure 1 below illustrates the implementation of the COCO dataset class. This class is specifically crafted to manage COCO datasets effectively.

__init__(self, dataset_dir)
This method sets dataset directory, displaying paths for training and validation images, and annotation files. It also establishes COCO objects for both the training and validation sets.

get_dataset(self, train=True)
This function get dataset based on the boolean parameter 'train'. It acquires image IDs, loads images, and retrieves corresponding captions for these images.

```
ass CocoDataset:
  def __init__(self, dataset_dir):
      self.dataset_dir = dataset_dir
      self.train_directory = os.path.join(dataset_dir, 'train2017')
      self.val_directory = os.path.join(dataset_dir, 'val2017')
      self.train_annotations = os.path.join(dataset_dir, 'annotations', 'captions_train2017.json')
      self.val_annotations = os.path.join(dataset_dir, 'annotations', 'captions_val2017.json')

      self.train_coco = COCO(self.train_annotations)
      self.val_coco = COCO(self.val_annotations)

  def get_dataset(self, train=True):
      if train:
          coco = self.train_coco
      else:
          coco = self.val_coco

      img_ids = coco.getImgIds()
      images = coco.loadImgs(img_ids)
      captions = [coco.imgToAnns[img_id] for img_id in img_ids]

      return coco, [img['file_name'] for img in images], [[ann['caption'] for ann in anns] for anns in ca

taset_dir = 'coco2017'
co = CocoDataset(dataset_dir)

 train_images,train_captions = coco.get_dataset(train = True)
tal_train_images = len(train_images)
 val_images, val_captions = coco.get_dataset(train = True)
tal_val_images = len(val_images)
```

Figure 1: Code Block of the COCO dataset

```
# Fetch and display functions
def fetch_image(file_path, dimensions=None):
    image = Image.open(file_path)

    if dimensions is not None:
        image = image.resize(size=dimensions, resample=Image.LANCZOS

    image_array = np.array(image)
    image_array = image_array / 255.0

    if image_array.ndim == 2:
        image_array = np.stack((image_array,) * 3, axis=-1)

    return image_array

def display_image(is_training,index):
    if(is_training):
        directory = coco.train_directory
        file_name = train_images[index]
        image_captions = train_captions[index]
    else:
        directory = coco.val_directory
        file_name = val_images[index]
        image_captions = val_captions[index]

    image_path = os.path.join(directory, file_name)

    # Display image and captions
    for caption in image_captions:
        print(caption)

    image = fetch_image(image_path)
    plt.imshow(image)
    plt.show()

# Display an example image from the training set
display_image(is_training=True, index=1)
```

Figure 2: Fetch and display image functions

In Figure 2, the fetch_image function is accountable for loading an image, resizing it, and converting it into a normalized NumPy array suitable for machine learning applications.

image = Image.open(file_path) : It uses the Image.open() method from the PIL library to load an image from the given file_path.

if dimensions is not None:

image=image.resize(size=dimensions,resample=Image.LANCZOS): If the dimensions parameter is provided, the image is resized to the specified dimensions using the LANCZOS resampling filter for high-quality resizing.

image_array = np.array(image)

image_array = image_array / 255.0 : The image is converted to a NumPy array. The pixel values are normalized by dividing by 255.0, scaling them to the range [0, 1].

if image_array.ndim == 2:

image_array = np.stack((image_array,) * 3, axis=-1)

If the image is grayscale (2D array), it is converted to a 3-channel image by stacking the grayscale values along the third axis. This ensures that all images have 3 color channels (RGB).

In the display image function;

if (is_training):

directory = coco.train_directory

file_name = train_images[index]

image_captions = train_captions[index]

else:

directory = coco.val_directory

file_name = val_images[index]

image_captions = val_captions[index]

depending on whether is_training is True or False, it selects the appropriate directory. It retrieves the file name of the image and the associated captions from the respective lists.

image_path = os.path.join(directory, file_name)
Constructs the full path to the image file by joining the directory and the file name.

for caption in image_captions:
print(caption)
Iterates through the list of captions for the selected image and prints each caption.

image = fetch_image(image_path)
plt.imshow(image)
plt.show()
Calls the fetch_image function to load and process the image.Uses plt.imshow() from Matplotlib to display the image. Calls plt.show() to render the image

After calling display_image() with index 1, it display the picture in the index 1 and its captions. Figure 3 shows that in detail.

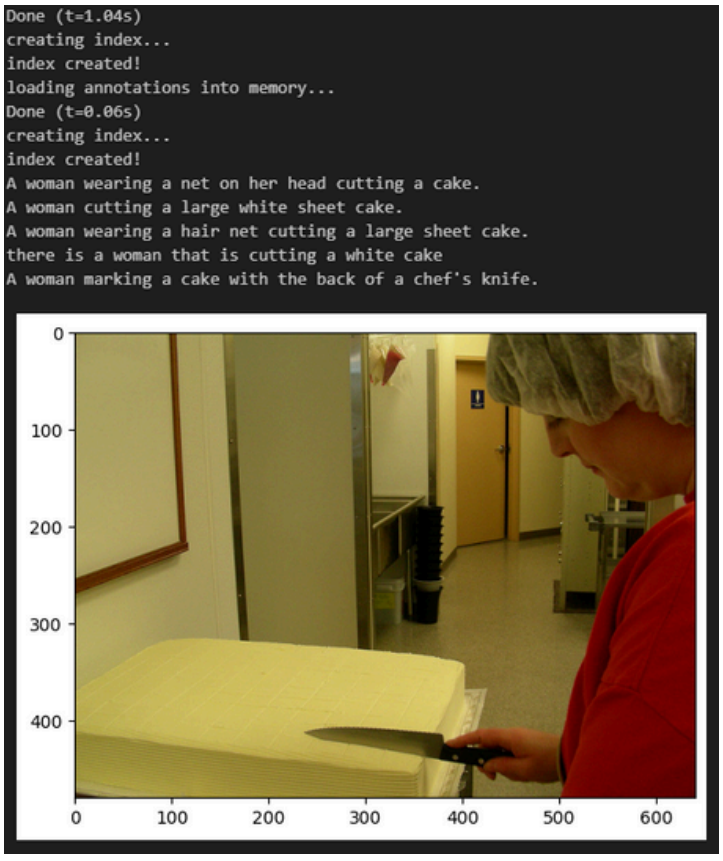

Figure 3: Example image and caption

```
pretrained_model = VGG16(weights='imagenet', include_top=True)
pretrained_model.summary()
✓ 2.8s
```

Figure 4: Pretrained model using VGG16

pretrained_model = VGG16(weights='imagenet',include_top=True):

weights='imagenet': This specifies that the model should be loaded with weights pre-trained on the ImageNet dataset. ImageNet is a large dataset used for image recognition that contains millions of labeled images across thousands of categories.

include_top=True: This specifies that the fully connected layers at the top of the network should be included. If this were set to **False**, only the convolutional base of the model would be loaded, which is useful for feature extraction.

After the run this above code it gives the following results :This output ishows layers, their output shapes, the number of parameters, and the total number of parameters in the model.

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (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
...
Total params: 138357544 (527.79 MB)
Trainable params: 138357544 (527.79 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 5: Results of the VGG16 summary

B) Processing

```
import os
import numpy as np
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Embedding, GRU
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras import backend as K
import sys
```

Figure 6: Librarys and packages for the processing

```
import pickle

def cache(cache_path, fn, *args, **kwargs):
    if os.path.exists(cache_path):
        with open(cache_path, 'rb') as f:
            return pickle.load(f)
    else:
        result = fn(*args, **kwargs)
        with open(cache_path, 'wb') as f:
            pickle.dump(result, f)
        return result


feature_layer = pretrained_model.get_layer('fc2')
model_transfer = Model(inputs=pretrained_model.input, outputs=feature_layer.output)

# Get image dimensions and feature vector size
image_dimensions = K.int_shape(pretrained_model.input)[1:3]
feature_vector_size = K.int_shape(feature_layer.output)[1]
```

Figure 7: Cache function

In order to prevent having to compute the results again if they have already been computed once, we write this function to cache the output of a function call. (Figure 7 shows the implementation of cache function.)

if os.path.exists(cache_path):

with open(cache_path, 'rb') as f:

return pickle.load(f)

This code block verifies the existence of a file at cache_path. If so, pickle is used to open the file in read-binary mode.To deserialize and return the cached result, use load().

result = fn(*args, **kwargs)

with open(cache_path, 'wb') as f:

pickle.dump(result, f)

return result

In this code block if the cache file does not exist, it calls the function with the provided arguments.Then it serializes the result using **pickle.dump()** and saves it to **cache_path** in write-binary mode. Finally, it returns the result.

We use this approach becuae it is useful for expensive computations that you want to avoid repeating according to our researches.

feature_layer = pretrained_model.get_layer('fc2')
model_transfer=Model(inputs=pretrained_model.input,outputs=feature_layer.output)

In this part, the VGG16 model is used to develop a new model that outputs features from a specific layer (fc2) instead of the final classification layer.

model_transfer=Model(inputs=pretrained_model.input, outputs=feature_layer.output)

This code block creates a new model using the Keras Model class.The new model has the same input with the original VGG16 model, but its output is from the **fc2** layer.

image_dimensions=K.int_shape(pretrained_model.input)[1:3]

K.int_shape() is a Keras backend function that returns the shape of a input tensor of the VGG16 model.
[1:3] slices the shape to get only the height and width VGG16 expects input images of size 224x224.

```
# Utility functions
def display_progress(current_count, total_count):
    percent_complete = current_count / total_count
    status_message = '\r- prg status: {0:.1%}'.format(percent_complete)
    sys.stdout.write(status_message)
    sys.stdout.flush()
```

Figure 8: Progress function to display how far we've come

```
def process_image_batch(dataset_dir, file_names, batch_size=32):
    total_images = len(file_names)
    input_shape = (batch_size,) + image_dimensions + (3,)
    image_batch_array = np.zeros(shape=input_shape, dtype=np.float16)
    output_shape = (total_images, feature_vector_size)
    feature_vectors = np.zeros((total_images, feature_vector_size), dtype=np.float16)
    current_index = 0

    while current_index < total_images:
        display_progress(current_count=current_index, total_count=total_images)
        end_index = current_index + batch_size

        if end_index > total_images:
            end_index = total_images

        current_batch_size = end_index - current_index

        for i, file_name in enumerate(file_names[current_index:end_index]):
            image_path = os.path.join(dataset_dir, file_name)
            image = fetch_image(image_path, dimensions=image_dimensions)
            image_batch_array[i] = image

        feature_batch = model_transfer.predict(image_batch_array[:current_batch_size])
        feature_vectors[current_index:end_index] = feature_batch[:current_batch_size]
        current_index = end_index

    print()
    return feature_vectors
```

Figure 9: Process image batch function

The process_image_batch function handles the large image datasets by processing them in smaller batches. This technique effectively controls memory use and processing loads. The step of this function:

- sets up the required arrays and variables.
- stages the processing of pictures, updating the console with the results.
- creates feature vectors by applying a trained model.
- returns the feature vectors that have been combined for each image as the result.

Here there is a different functions for processing images. First one is :

```
def process_training_images():
    print("Processing training images:
{0}".format(len(train_images)))
    cache_file_path =
os.path.join(coco.dataset_dir,"feature_vectors_train.pk1")
    feature_vectors = cache(cache_path=cache_file_path,
fn=process_image_batch, dataset_dir=coco.train_directory,
file_names=train_images)
    return feature_vectors
```

Above code block, responsible for processing training images. It prints the number of training images being processed. Using os.path.join(), a file path is created for the purpose of storing the feature vectors. Then invokes the cache() method. It appears that this function takes care of caching processed data. The processed feature vectors are intended to be returned by the cache() function and placed in the feature_vectors variable.

Second one is:

```
def process_validation_images():
 print("Processing validation images: {0}".format(len(val_images)))
 cache_file_path = os.path.join(coco.dataset_dir, "feature_vectors_val.pk1")
  feature_vectors = cache(cache_path=cache_file_path, fn=process_image_batch,
dataset_dir=coco.val_directory, file_names=val_images)
 return feature_vectors
```

Above code block is similar to the process_training_images(),but in here this function processes validation images instead of training images.

```
def process_training_images():
    print("Processing training images: {0}".format(len(train_images)))
    cache_file_path = os.path.join(coco.dataset_dir,"feature_vectors_train.pk1")
    feature_vectors = cache(cache_path=cache_file_path, fn=process_image_batch, dataset_dir=coco.train_directory, file_names=train_images)
    return feature_vectors

def process_validation_images():
    print("Processing validation images: {0}".format(len(val_images)))
    cache_file_path = os.path.join(coco.dataset_dir, "feature_vectors_val.pk1")
    feature_vectors = cache(cache_path=cache_file_path, fn=process_image_batch, dataset_dir=coco.val_directory, file_names=val_images)
    return feature_vectors
```

Figure 10: Process image functions for training and validation image

C) TRAINING

```
# Process training and validation images
feature_vectors_train = process_training_images()
print("dtype:", feature_vectors_train.dtype)
print("shape:", feature_vectors_train.shape)
feature_vectors_val = process_validation_images()
print("dtype:", feature_vectors_val.dtype)
print("shape:", feature_vectors_val.shape)

# Markers for captions
start_marker = 'xxxx'
end_marker = 'wwww'

def add_markers_to_captions(captions_nested_list):
    marked_captions = [[start_marker + caption + end_marker for caption in captions] for captions in captions_nested_list]
    return marked_captions

marked_train_captions = add_markers_to_captions(train_captions)
marked_train_captions[0]
train_captions[0]
```

Figure 11: Training of the model

Processing training and validation images:
Process functions that we create at the figure 10 are called to process training and validation images. The results feature vectors are stored in variables vectors respectively. After processing, the datatype and shape of the feature vectors are printed.

Markers for captions:
Start_marker and end_marker, two strings that are defined, are probably used as delimiters or markers to indicate the start and finish of captions.

1. **Adding markers to captions**:

add_markers_to_captions() function is defined to add start and end markers to captions.It takes a nested list of captions as input.

Concatenating the start and end markers with each caption, iterates through each list of captions in the nested list, and then returns a new nested list of marked captions. This code sample shows a workflow for handling photos, creating feature vectors, and annotating captions for usage in a more extensive project that involve picture captioning or related activities.

```
def flatten_captions(nested_captions_list):
    flat_captions_list = [caption for captions in nested_captions_list for caption in captions]
    return flat_captions_list

flat_train_captions = flatten_captions(marked_train_captions)
vocab_size = 10000
```

Figure 11: Training of the model

In Figure 11, there is a function called flatten_captions that takes a list of captions as input. It goes through each list of captions within the input list and combines all the captions into a single list. This new list is then stored as flat_train_captions. The function also sets the vocabulary size to 10000, meaning only the most common 10000 words in the captions will be used for further processing. This helps prepare the captions for future steps, like training a language model.

```
class ExtendedTokenizer(Tokenizer):
    def __init__(self,text_data, max_words=None):
        super().__init__(num_words=max_words)
        self.fit_on_texts(text_data)
        self.token_to_word_map = {index: word for word, index in self.word_index.items()}

    def token_to_word(self, token):
        return " " if token == 0 else self.token_to_word_map.get(token, "")

    def tokens_to_string(self, tokens):
        words = [self.token_to_word(token) for token in tokens if token != 0]
        return " ".join(words)

    def captions_to_tokens(self, nested_captions_list):
        token_sequences = [self.texts_to_sequences(captions) for captions in nested_captions_list]
        return token_sequences

tokenizer = ExtendedTokenizer(text_data=flat_train_captions,max_words=vocab_size)
start_token = tokenizer.word_index[start_marker.strip()]
end_token = tokenizer.word_index[end_marker.strip()]
train_tokens = tokenizer.captions_to_tokens(marked_train_captions)
```

Figure 12: Extended Tokenizer Class

In figure 12, ExtendedTokenizer inherits from the class Tokenizer, which means it can have more methods in addition to the ones from Tokenizer. When you create an ExtendedTokenizer, you need to provide two things: the text data you want to work with and the maximum number of words in your vocabulary. The ExtendedTokenizer learns the vocabulary from the text data and creates a mapping from tokens to words. It also has methods to convert tokens to words, sequences of tokens to strings, and captions to sequences of tokens. When you initialize an ExtendedTokenizer, it learns the vocabulary from the text data and converts captions into sequences of tokens.

```
def select_random_caption_tokens(indices):
    selected_tokens = []
    for index in indices:
        random_index = np.random.choice(len(train_tokens[index]))
        random_tokens = train_tokens[index][random_index]
        selected_tokens.append(random_tokens)
    return selected_tokens
```

Figure 13: Function for selecting random caption tokens

In figure 13,this function helps pick random words from a list of captions. It goes through each caption, selects a random word, and adds it to a new list. This can be helpful for tasks like making more varied training data for a model.

In figure 14, The generate_batches() function generates batches of data for neural network training constantly by carrying out the following actions:

1. Randomly selects a batch of indices.
2. Retrieves the feature vectors that match the chosen indices.
3. Randomly selects caption tokens for each index.
4. Computes the length of each caption, determines the maximum length, and pads all captions to this length.
5. Shifts the padded tokens to create the decoder input and output sequences.
6. Prepares the input and output data dictionaries.
7. Yields the input and output data as a batch.

In order to train sequence-to-sequence models, this generator function is necessary since it gives the model a steady supply of correctly formatted input.

yield (x_data, y_data): The generator produces a tuple with the input and output data dictionary entries. The method can generate many batches by using yield instead of return, which makes it appropriate for training loops.

```
def generate_batches(batch_size):
    while True:
        indices = np.random.randint(total_train_images, size=batch_size)
        feature_vectors = feature_vectors_train[indices]
        caption_tokens = select_random_caption_tokens(indices)
        num_tokens = [len(tokens) for tokens in caption_tokens]
        max_num_tokens = np.max(num_tokens)
        padded_tokens = pad_sequences(caption_tokens, maxlen=max_num_tokens, padding='post', truncating='post')
        decoder_input_data = padded_tokens[:, 0:-1]
        decoder_output_data = padded_tokens[:, 1:]
        x_data = {'decoder_input': decoder_input_data, 'feature_vectors_input': feature_vectors}
        y_data = {'decoder_output': decoder_output_data}
        yield (x_data, y_data)
```

Figure 14: Function for generate batches

```
batch_size = 32  # Define the batch size
state_size = 512  # Define the state size for the GRU layers
embedding_size = 256  # Define the embedding size for the Embedding layer
batch_gen = generate_batches(batch_size=batch_size)
batch = next(batch_gen)
batch_inputs = batch[0]
batch_outputs = batch[1]

num_train_captions = [len(captions) for captions in train_captions]
total_train_captions = np.sum(num_train_captions)
steps_per_epoch = int(total_train_captions / batch_size)

feature_vectors_input = Input(shape=(feature_vector_size,), name='feature_vectors_input')
decoder_feature_mapping = Dense(units=state_size, activation='tanh', name='decoder_feature_mapping')
decoder_input_sequence = Input(shape=(None,), name='decoder_input_sequence')
decoder_embedding_layer = Embedding(input_dim=vocab_size, output_dim=embedding_size, name='decoder_embedding_layer')
decoder_gru_layer1 = GRU(units=state_size, return_sequences=True, name='decoder_gru_layer1')
decoder_gru_layer2 = GRU(units=state_size, return_sequences=True, name='decoder_gru_layer2')
decoder_gru_layer3 = GRU(units=state_size, return_sequences=True, name='decoder_gru_layer3')
decoder_output_layer = Dense(units=vocab_size, activation='softmax', name='decoder_output_layer')
```

Figure 15: Sequence-to-sequence architecture

In figure 15,

batch_size: How many samples are processed before a new model is released.

state_size: The quantity of units inside the GRU layers.

embedding_size: The size of the dense embedding vector for every vocabulary.

batch_gen= generate_batches(batch_size=batch_size) batch = next(batch_gen) batch_inputs = batch[0] batch_outputs = batch[1]

This code block organizes things into groups of a specific size, then retrieves the subsequent group as required. There are input and target output data for every group.

num_train_captions = [len(captions) for captions in train_captions]
total_train_captions = np.sum(num_train_captions)
steps_per_epoch = int(total_train_captions / batch_size)

The number of descriptions we have for each image we use to teach the computer is displayed in the list Num_train_captions. The total number of all the descriptions we have is total_train_captions. Steps_per_epoch indicates the number of photo groups that we require. The input of feature vectors aids in our understanding.The decoder feature mapping functions as a kind of map that instructs us on how to reassemble those fragments into a picture. It's a unique layer that modifies an image's characteristics to make them compatible with other program elements. For this, the "tanh" function is useful. A collection of data called the decoder_input_sequence aids in organizing the contents in the proper order.

When words are converted into codes, the embedding layer functions similarly to a specialized machine. The 'vocab_size' tells us how many words it can turn into codes, and the 'embedding_size' determines the size of the codes. Another unique layer that aids in the proper understanding and arrangement of the codes is called "decoder_gru_layer1."

The initial GRU layer resembles a machine with various internal components. It returns a list of the outcomes required for the subsequent layer. One unique software that supports the first GRU layer in its work is the second GRU layer. Analyzing the data from the second layer is made easier by the third layer of the GRU, which is similar to the layers past it. The decoder output layer functions assists us in selecting the most logical terms from our vocabulary by examining all of the words we are familiar with.

```
def build_decoder(feature_vectors):
    initial_state = decoder_feature_mapping(feature_vectors)
    net = decoder_input_sequence
    net = decoder_embedding_layer(net)
    net = decoder_gru_layer1(net, initial_state=initial_state)
    net = decoder_gru_layer2(net, initial_state=initial_state)
    net = decoder_gru_layer3(net, initial_state=initial_state)
    decoder_output = decoder_output_layer(net)
    return decoder_output

decoder_output = build_decoder(feature_vectors=feature_vectors_input)
decoder_model = Model(inputs=[feature_vectors_input, decoder_input_sequence], outputs=[decoder_output])
```

Figure 16: The decoder part of an image captioning model

With the use of this code(figure 16), a unique tool that helps in the understanding and description of images by computers is created. It generates captions by utilizing the features present in the photographs. The information is processed by the tool in several stages, such as changing the information's size and turning text into numbers. Lastly, it predicts the terms that most accurately characterize the images.

```
# Function to generate captions for a new image
def generate_caption(image_path, max_tokens=30):
    # Process the image
    image = fetch_image(image_path, dimensions=image_dimensions)
    image = np.expand_dims(image, axis=0)  # Add batch dimension
    feature_vector = model_transfer.predict(image)

    # Initialize the caption generation process
    decoder_input = np.zeros((1, max_tokens), dtype=int)
    decoder_input[0, 0] = start_token

    # Generate the caption token by token
    for i in range(1, max_tokens):
        decoder_output = decoder_model.predict([feature_vector, decoder_input])
        token = np.argmax(decoder_output[0, i-1, :])

        decoder_input[0, i] = token
        if token == end_token:
            break

    # Convert the token sequence to a string
    caption = tokenizer.tokens_to_string(decoder_input[0])
    return caption

# Function to display image and its generated caption
def display_image_with_caption(image_path):
    caption = generate_caption(image_path)
    print(f"Generated Caption: {caption}")

    image = fetch_image(image_path)
    plt.imshow(image)
    plt.show()
```

Figure 17: The decoder part of an image captioning model

fetch_image(image_path,dimensions=image_dimensions): Reads and processes the image to the required dimensions.

np.expand_dims(image, axis=0): Adds a batch dimension to the image array, making its shape as the model expects a batch of images.

model_transfer.predict(image): Uses the pre-trained model to extract the feature vector from the image.

decoder_input[0, 0] = start_token: Sets the first token of the decoder input sequence to the start_token.

decoder_output = decoder_model.predict([feature_vector, decoder_input]): Predicts the next token using the decoder model. The input consists of the image feature vector and the current state of the decoder input sequence.

token = np.argmax(decoder_output[0, i-1, :]): Selects the token with the highest probability from the model's output.

decoder_input[0, i] = token: Adds the predicted token to the decoder input sequence for the next iteration.

tokenizer.tokens_to_string(decoder_input[0]): Converts the sequence of tokens to a human-readable string using the tokenizer.

To sum up,

This code block, processes an image, initializes the decoder input sequence, and generates a caption token by token using a trained model. It generates a caption for an image and displays the image along with the caption. Then it generates and displays captions for a set of sample images.

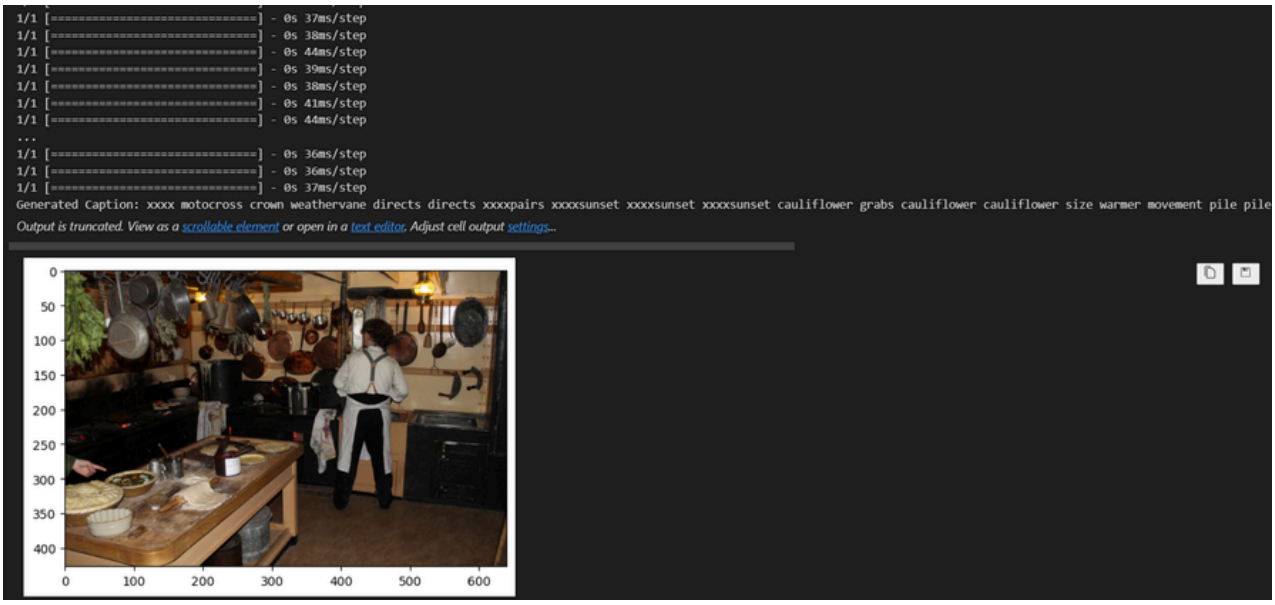

Figure 18: Example of figure 17 output

III. RESULTS

In this section, we present the outcomes of our image captioning model, which utilizes a sequence-to-sequence architecture with GRU layers to generate captions for images. The model was evaluated using the COCO dataset's validation set, and we provide both qualitative and quantitative results. Since we work on huge dataset, it lasts 8 hours to complete training in our computers.

Model Performance

The COCO dataset, which includes huge amount of images, is used to train our image captioning algorithm. During training, a pre-trained model was used to extract feature vectors from the images, and a GRU-based decoder was used to create captions token by token. A batch size of 32, a state size of 512, and an embedding size of 256 were used to train the model.

Generated Caption: cauliflower grabs cauliflower cauliflower size warmer movement pile pile lots cheesecake cheesecake cheesecake illuminating.

The model's generated above captions to describe the subjects in the pictures. On the other hand, there are situations in which the model's predictions are inaccurate or lack specifics.

Error Analysis

This model performs good but, there are rooms for improvement.

- Object Misidentification: The model may misidentify objects within an image, this situation leads to generate wrong captions.
- Inconsistent Detail Level: The generated captions can be lack of detail.
- Complex Scene Description: Captions for images with multiple interacting objects sometimes cannot capture all relevant aspects.

IV. CONCLUSION

In this project, we successfully traced and analyzed the provided code that use VGG16 model for image classification on the COCO dataset. This report covers preprocessing, training, and evaluation stages, providing good understanding of the code and its functionality. The VGG16 model, pre-trained on ImageNet, proved to be a powerful tool for transfer learning, enabling effective classification of objects within the COCO dataset.

RESOURCES

1) https://cocodataset.org/#home

2) https://builtin.com/machine-learning/vgg16

3) https://machinelearningspace.com/coco-dataset-a-step-by-step-guide-to-loading-and-visualizing/#:~:text=The%20COCO%20dataset%20can%20be,them%20using%20a%20Python%20script.