Image Captioning with Transformers

Flicker Dataset

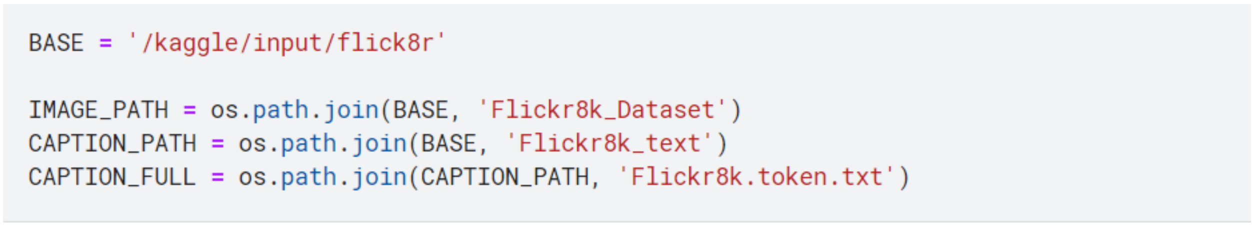
❖ Libraries Imports:

```
import os
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import random
from collections import defaultdict
from tqdm import tqdm
import time

%matplotlib inline
```

- **import os:** Imports the os module, which provides a way to interact with the operating system, such as managing files and directories.
- **import tensorflow as tf:** Imports the TensorFlow library, which is used for building and training machine learning models, particularly deep learning models.
- **import matplotlib.pyplot as plt:** Imports the matplotlib.pyplot module, which is used for creating visualizations like plots and charts.
- **import numpy as np: Imports** the NumPy library, which is used for numerical operations in Python, particularly for handling arrays and matrices.
- **import random:** Imports the random module, which provides functions for generating random numbers and making random selections.
- **from collections import defaultdict**: Imports the defaultdict class from the collections module, which is a subclass of the built-in dict class that returns a default value when a key is not found.

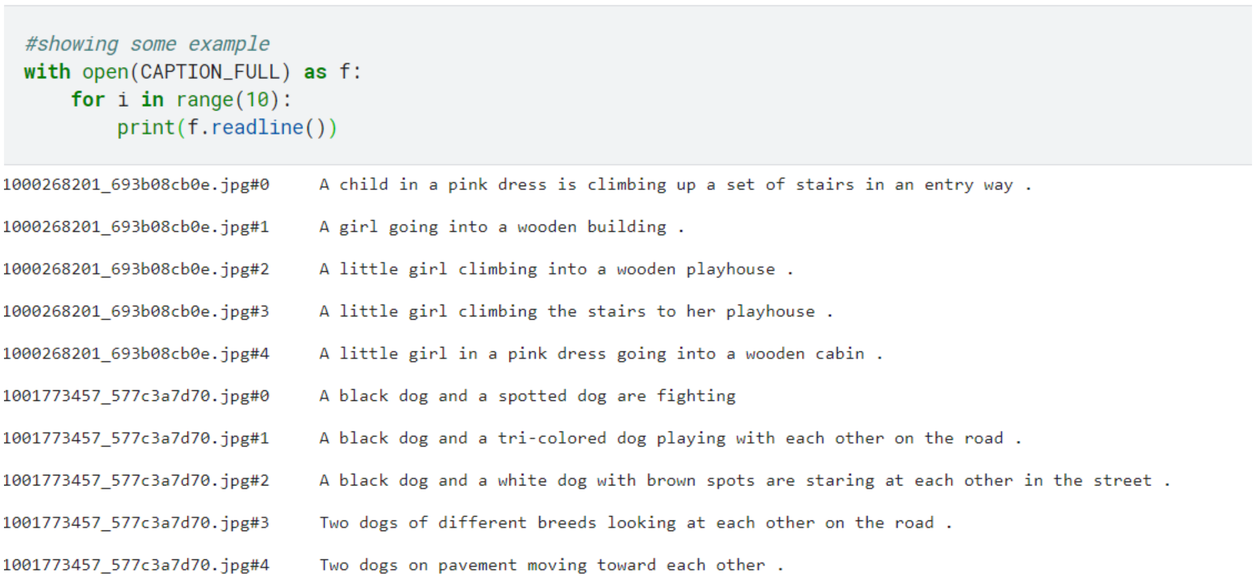
- **from tqdm import tqdm:** Imports the tqdm module, which provides a progress bar for loops and iterations, making it easier to track the progress of tasks.
- **import time**: Imports the time module, which provides functions for working with time, such as measuring the duration of code execution.
- %**matplotlib inline:** This is a magic command specific to Jupyter notebooks and some other IPython environments. It configures matplotlib to display plots inline in the notebook.

❖ Upload Dataset



```
BASE = '/kaggle/input/flick8r'

IMAGE_PATH = os.path.join(BASE, 'Flickr8k_Dataset')
CAPTION_PATH = os.path.join(BASE, 'Flickr8k_text')
CAPTION_FULL = os.path.join(CAPTION_PATH, 'Flickr8k.token.txt')
```

- **BASE = '/kaggle/input/flick8r':** This sets the base directory path where the Flickr8k dataset is located. It seems to be assuming a directory structure commonly used in Kaggle competitions, where datasets are typically stored under /kaggle/input/.
- **IMAGE_PATH = os.path.join(BASE, 'Flickr8k_Dataset'):** This creates a path to the directory containing the images in the Flickr8k dataset. It uses os.path.join() to ensure the path is constructed correctly across different operating systems.
- **CAPTION_PATH = os.path.join(BASE, 'Flickr8k_text'):** This creates a path to the directory containing the caption files for the images in the Flickr8k dataset.
- **CAPTION_FULL = os.path.join(CAPTION_PATH, 'Flickr8k.token.txt'):** This creates a path to the specific caption file named Flickr8k.token.txt within the caption directory. This file likely contains the image file names along with their corresponding captions.

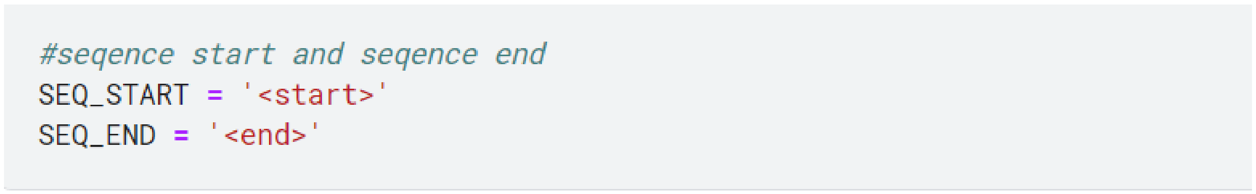


```
#showing some example
with open(CAPTION_FULL) as f:
    for i in range(10):
        print(f.readline())
```

```
1000268201_693b08cb0e.jpg#0	A child in a pink dress is climbing up a set of stairs in an entry way .

1000268201_693b08cb0e.jpg#1	A girl going into a wooden building .

1000268201_693b08cb0e.jpg#2	A little girl climbing into a wooden playhouse .

1000268201_693b08cb0e.jpg#3	A little girl climbing the stairs to her playhouse .

1000268201_693b08cb0e.jpg#4	A little girl in a pink dress going into a wooden cabin .

1001773457_577c3a7d70.jpg#0	A black dog and a spotted dog are fighting

1001773457_577c3a7d70.jpg#1	A black dog and a tri-colored dog playing with each other on the road .

1001773457_577c3a7d70.jpg#2	A black dog and a white dog with brown spots are staring at each other in the street .

1001773457_577c3a7d70.jpg#3	Two dogs of different breeds looking at each other on the road .

1001773457_577c3a7d70.jpg#4	Two dogs on pavement moving toward each other .
```

Reading from a file that contains image captions. Each line consists of an image ID followed by a caption ID and the corresponding caption text.

❖ Caption Mapping For Evaluation:



```
#seqence start and seqence end
SEQ_START = '<start>'
SEQ_END = '<end>'
```

These variables, SEQ_START and SEQ_END, likely represent special tokens used to mark the start and end of a sequence, such as a caption or a sentence. These tokens are commonly used in natural language processing tasks, especially in sequence-to-sequence models like those used for image captioning.

```
os.listdir(BASE)
```

```
['Flickr8k_text', 'Flickr8k_Dataset']
```

The os.listdir(BASE) function call will return a list of the files and directories in the directory specified by the BASE path. It essentially gives you a way to see what files and subdirectories are contained within the BASE directory. This can be useful for exploring the contents of a directory programmatically.

```
#mapping image id to list of captions
captions_map = defaultdict(list)

with open(CAPTION_FULL) as file:
    lines = file.readlines()
    for line in lines:
        data = line.split('\t')
        image_id = data[0].split('#')[0]
        caption = SEQ_START + ' ' + data[1].strip() + ' ' + SEQ_END
        if not os.path.exists(os.path.join(IMAGE_PATH, image_id + '.npy')):
            continue
        captions_map[image_id].append(caption)
```

- **captions_map = defaultdict(list):** This line initializes a defaultdict with a default value of an empty list. This data structure will be used to store the mapping of image IDs to their captions.
- **with open(CAPTION_FULL) as file::** This opens the caption file (Flickr8k.token.txt) specified by the CAPTION_FULL path for reading.
- **lines = file.readlines():** This reads all lines from the file and stores them in the lines list.
- **for line in lines:** This iterates over each line in the file.
- **data = line.split('\t'):** This splits each line into two parts based on the tab character ('\t'). The first part is the image ID and the caption ID, separated by a #.
- **image_id = data[0].split('#')[0]:** This extracts the image ID from the first part of the split line by splitting it again at the # character and taking the first part.
- **caption = SEQ_START + ' ' + data[1].strip() + ' ' + SEQ_END:** This constructs the caption by adding start and end sequence tokens (SEQ_START and SEQ_END, presumably defined elsewhere) to the caption text, which is stripped of leading and trailing whitespace.

- **if not os.path.exists(os.path.join(IMAGE_PATH, image_id + '.npy')):**: This checks if the corresponding image file exists in the IMAGE_PATH directory. It assumes that image files are stored as .npy files with filenames matching the image IDs.
- **captions_map[image_id].append(caption):** If the image file exists, it appends the caption to the list of captions associated with the image ID in the captions_map dictionary.

```python
all_captions = []
all_image_paths = []
for image_id in captions_map:
    all_captions.extend(captions_map[image_id])
    all_image_paths.extend([os.path.join(IMAGE_PATH, image_id)] * len(captions_map[image_id]))
```

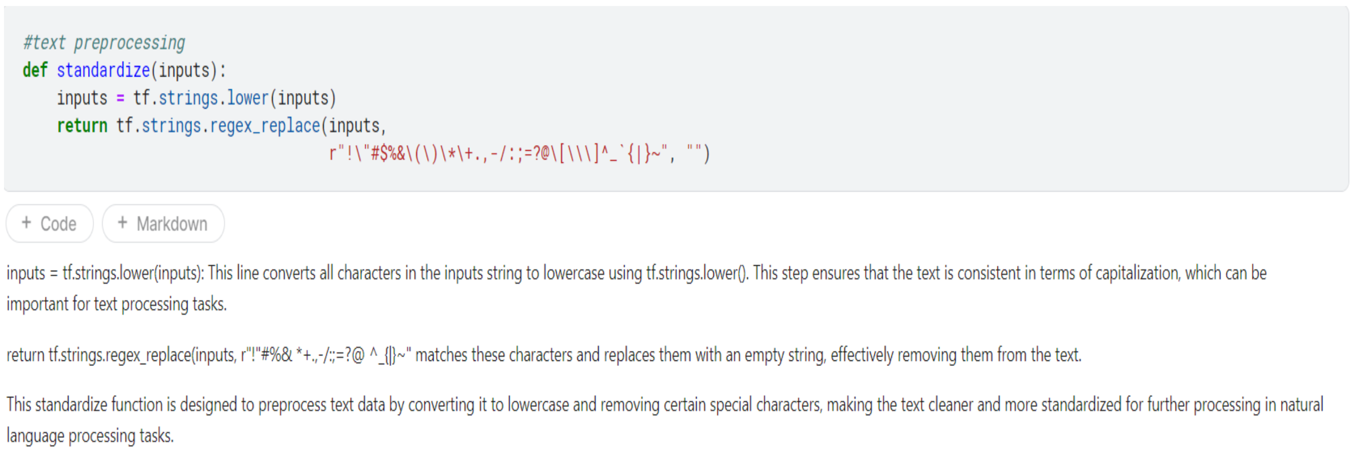
- **all_captions = []:** Initializes an empty list to store all the captions.
- **all_image_paths = []:** Initializes an empty list to store all the image paths.
- **for image_id in captions_map::** Iterates over each key (image ID) in the captions_map dictionary.
- **all_captions.extend(captions_map[image_id]):** Extends the all_captions list with all the captions associated with the current image_id.
- **all_image_paths.extend([os.path.join(IMAGE_PATH, image_id)] * len(captions_map[image_id])):** Extends the all_image_paths list with the full path to the image corresponding to image_id, repeated for each caption associated with that image_id. The * len(captions_map[image_id]) part replicates the image path to match the number of captions for that image.
- After this code snippet executes, all_captions will contain all the captions from the captions_map dictionary, and all_image_paths will contain the corresponding paths to the images. These lists can be used together to associate each caption with its corresponding image path for further processing.

❖ Tokenize Captioning

Tokenizing captions is a crucial step in natural language processing tasks, especially in machine learning models that deal with text. By tokenizing captions, we convert the textual data into a format that can be easily processed by the model. Each word in the caption is replaced with a unique integer, which allows the model to understand and manipulate the text more effectively. Tokenization also helps in standardizing the input data, such as converting all text to lowercase and removing special characters or punctuation marks. Overall, tokenizing captions is essential for preparing textual data for use in machine learning models, enabling them to learn patterns and generate meaningful output based on the input captions.

```
#Convert captions to tensor
caption_dataset = tf.data.Dataset.from_tensor_slices(all_captions)
```

The tf.data.Dataset.from_tensor_slices() method is used to convert the list of all captions (all_captions) into a TensorFlow Dataset object. This conversion allows for efficient handling and processing of the captions using TensorFlow's data pipeline capabilities. The resulting Dataset object can be further manipulated and used in TensorFlow operations, such as batching, shuffling, and preprocessing, to prepare the captions for use in training a machine learning model, such as a sequence-to-sequence model for image captioning.



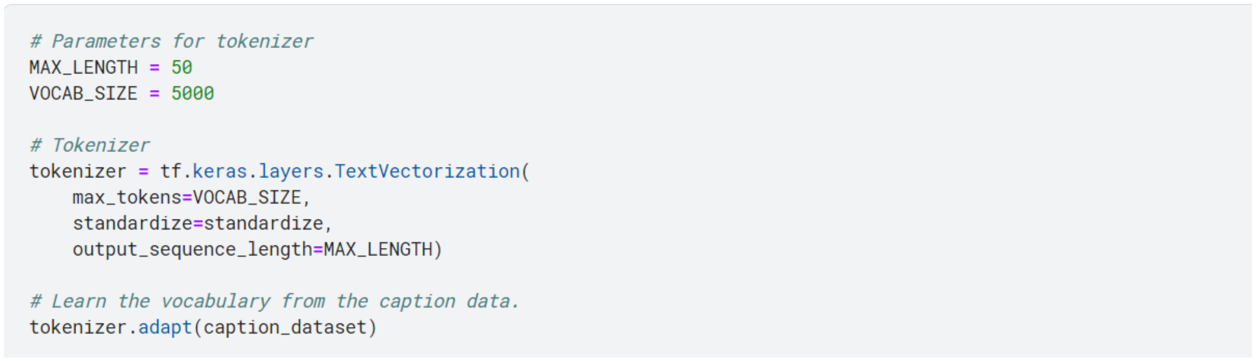
```
#text preprocessing
def standardize(inputs):
    inputs = tf.strings.lower(inputs)
    return tf.strings.regex_replace(inputs,
                                    r"!\"#$%&\(\)\*\+.,-/:;=?@\[\\\]^_`{|}~", "")
```

inputs = tf.strings.lower(inputs): This line converts all characters in the inputs string to lowercase using tf.strings.lower(). This step ensures that the text is consistent in terms of capitalization, which can be important for text processing tasks.

return tf.strings.regex_replace(inputs, r"!"#%& *+.,-/:;=?@ ^_{|}~" matches these characters and replaces them with an empty string, effectively removing them from the text.

This standardize function is designed to preprocess text data by converting it to lowercase and removing certain special characters, making the text cleaner and more standardized for further processing in natural language processing tasks.

❖ <u>Text Vectorization</u>



```
# Parameters for tokenizer
MAX_LENGTH = 50
VOCAB_SIZE = 5000

# Tokenizer
tokenizer = tf.keras.layers.TextVectorization(
    max_tokens=VOCAB_SIZE,
    standardize=standardize,
    output_sequence_length=MAX_LENGTH)

# Learn the vocabulary from the caption data.
tokenizer.adapt(caption_dataset)
```

- **<u>MAX_LENGTH = 50 and VOCAB_SIZE = 5000:</u>** These parameters define the maximum length of sequences (MAX_LENGTH) and the maximum vocabulary size (VOCAB_SIZE) for the tokenizer. Sequences longer than MAX_LENGTH will be truncated, and the tokenizer will only consider the top VOCAB_SIZE most common words in the dataset.
- **<u>tokenizer = tf.keras.layers.TextVectorization(...):</u>**
  - **max_tokens=VOCAB_SIZE:** Specifies the maximum vocabulary size for the tokenizer.
  - **standardize=standardize:** Specifies the standardization function to use for preprocessing text. In this case, it's the standardize function defined earlier, which converts text to lowercase and removes certain special characters.
  - Specifies the maximum length of the output sequences. Sequenceslonger than this will be truncated, and shorter sequences will be padded.
- **t<u>okenizer.adapt(caption_dataset):</u>** This line adapts the tokenizer to the caption dataset, allowing it to learn the vocabulary from the caption data. During adaptation, the tokenizer processes the text and builds its vocabulary based on the specified parameters (VOCAB_SIZE and MAX_LENGTH). This step is important before using the tokenizer to encode text data into sequences of integers.

```python
for cap in caption_vectors.take(1):
    print(cap)
```

```
tf.Tensor(
[  3   2  45   6   2  93 173   9 122  56   2 400  14 397   6  31   1 698
   5   4   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0], shape=(50,), dtype=int64)
```

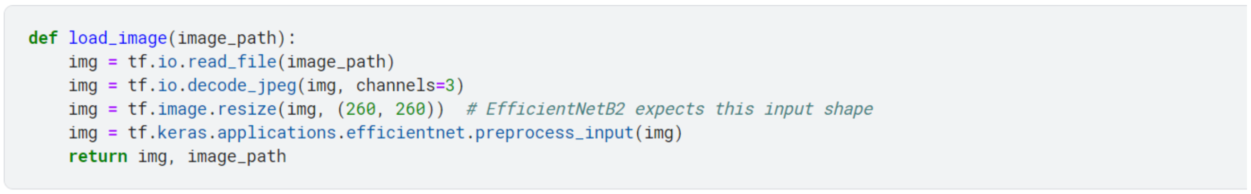
prints the first tokenized caption from the caption_vectors dataset. The dataset contains tokenized versions of captions, likely processed using the configured tokenizer. The take(1) method retrieves only one element from the dataset, and the print function displays it. The output is a tensor representing a single caption, with each integer corresponding to a token in the caption based on the tokenizer's vocabulary. The length of the tensor is determined by MAX_LENGTH (set to 50), ensuring a consistent length for all captions. This provides a preview of how captions are tokenized and prepared for use in machine learning models.

```python
# word2index and index2word (in another version)
def word2index(word):
    return tf.squeeze(word_to_index(tf.constant([word])))

def index2word(index):
    return tf.squeeze(index_to_word(tf.constant([index])))
```

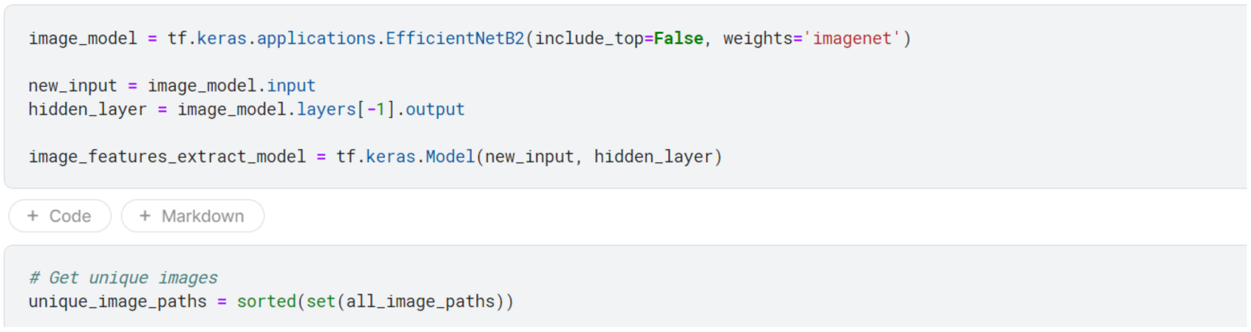
- **word2index(word):** This function takes a word as input, converts it to a TensorFlow constant array (tf.constant([word])), and then presumably uses word_to_index to map the word to its index in the vocabulary. The tf.squeeze function is used to remove any extra dimensions from the output, resulting in a scalar index value.

- **index2word(index):** Similarly, this function takes an index as input, converts it to a TensorFlow constant array (tf.constant([index])), and then presumably uses index_to_word to map the index back to the corresponding word in the vocabulary. The tf.squeeze function is again used to remove any extra dimensions from the output, resulting in a scalar word value.

❖ Feature Extraction



```python
def load_image(image_path):
    img = tf.io.read_file(image_path)
    img = tf.io.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, (260, 260))  # EfficientNetB2 expects this input shape
    img = tf.keras.applications.efficientnet.preprocess_input(img)
    return img, image_path
```



```python
image_model = tf.keras.applications.EfficientNetB2(include_top=False, weights='imagenet')

new_input = image_model.input
hidden_layer = image_model.layers[-1].output

image_features_extract_model = tf.keras.Model(new_input, hidden_layer)
```

```python
# Get unique images
unique_image_paths = sorted(set(all_image_paths))
```

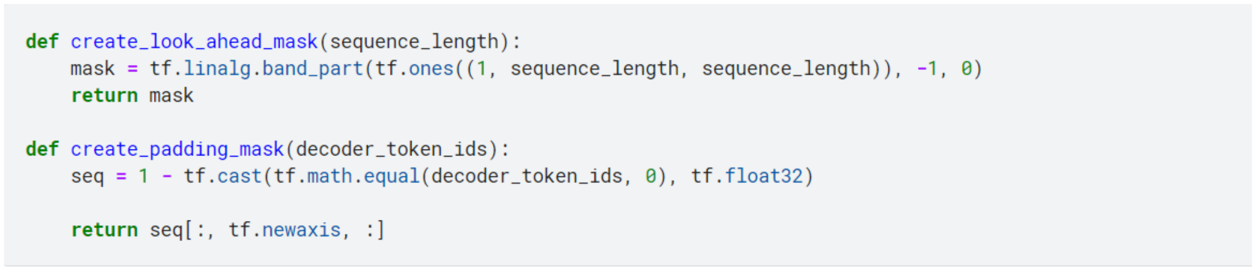
The provided code defines a function load_image(image_path) that reads an image file from a given path, decodes it using JPEG format with three color channels, resizes it to a specific shape required by the EfficientNetB2 model (260x260), and preprocesses it using the preprocess_input function from tf.keras.applications.efficientnet. This function returns the preprocessed image along with its original path.

Next, an instance of the EfficientNetB2 model is created using tf.keras.applications.EfficientNetB2 with include_top=False to exclude the fully connected layers at the top of the network, and weights='imagenet' to initialize the model with weights pre-trained on the ImageNet dataset.

The code then extracts the output of the last convolutional layer of the EfficientNetB2 model, which will be used as features for the images. Finally, a new model image_features_extract_model is created using tf.keras.Model with the input as the original input of the EfficientNetB2 model and the output as the extracted features.

(EfficientNetB2)

EfficientNetB2 is a compact yet powerful convolutional neural network (CNN) architecture known for its efficiency and performance in image classification tasks. It is part of the EfficientNet family, which achieves state-of-the-art results by scaling the network depth, width, and resolution in a balanced manner. EfficientNetB2 strikes a balance between model size, computational resources, and performance, making it suitable for various computer vision tasks where efficiency is crucial.



```python
def create_look_ahead_mask(sequence_length):
    mask = tf.linalg.band_part(tf.ones((1, sequence_length, sequence_length)), -1, 0)
    return mask

def create_padding_mask(decoder_token_ids):
    seq = 1 - tf.cast(tf.math.equal(decoder_token_ids, 0), tf.float32)

    return seq[:, tf.newaxis, :]
```

These functions are used to create masks for the attention mechanism in a transformer model:

- **create_look_ahead_mask(sequence_length):** This function creates a look-ahead mask to prevent the decoder from attending to future tokens during training. It creates a lower triangular matrix of shape (1, sequence_length, sequence_length) where the lower triangle (excluding the diagonal) is filled with ones, and the upper triangle is filled with zeros. This mask is then applied to the decoder's attention mechanism to mask out future tokens.
- **create_padding_mask(decoder_token_ids):** This function creates a padding mask to ignore padding tokens in the input sequences. It first checks which elements in decoder_token_ids are equal to zero (assuming zero is the padding token), resulting in a tensor of the same shape as decoder_token_ids with ones where the tokens are not padding and zeros where they are. It then adds a new axis to the tensor to make it compatible with the attention mechanism in the transformer model.

```python
def positional_encoding_1d(position, D):
    angle_rads = get_angles(np.arange(position)[:, np.newaxis],  # column vector
                            np.arange(D)[np.newaxis, :],  # row vector
                            D)

    # Apply the sine function to even indices
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

    # Apply the cosine function to odd indices
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])

    pos_encoding = angle_rads[np.newaxis, ...]

    return tf.cast(pos_encoding, dtype=tf.float32)
```

Positional encoding 1D:

```
PE(x,2i) = sin(x/10000^(2i/D))
PE(x,2i+1) = cos(x/10000^(2i/D))

Where:
x is a point in 2d space
i is an integer in [0, D/2), where D is the size of the ch dimension
```

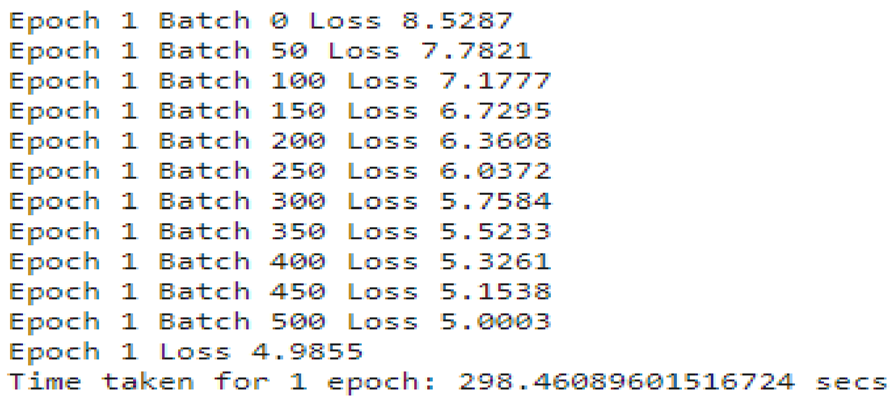
```python
def positional_encoding_2d(row, col, D):
    assert D % 2 == 0
    # first D/2 encode row embedding and second D/2 encode column embedding
    row_pos = np.repeat(np.arange(row), col)[:, np.newaxis]
    col_pos = np.repeat(np.expand_dims(np.arange(col), 0), row, axis=0).reshape(-1, 1)

    angle_rads_row = get_angles(row_pos,
                                np.arange(D // 2)[np.newaxis, :],
                                D // 2)
    angle_rads_col = get_angles(col_pos,
                                np.arange(D // 2)[np.newaxis, :],
                                D // 2)

    # apply sin and cos to odd and even indices resp.
    angle_rads_row[:, 0::2] = np.sin(angle_rads_row[:, 0::2])
    angle_rads_row[:, 1::2] = np.cos(angle_rads_row[:, 1::2])
    angle_rads_col[:, 0::2] = np.sin(angle_rads_col[:, 0::2])
    angle_rads_col[:, 1::2] = np.cos(angle_rads_col[:, 1::2])

    pos_encoding = np.concatenate([angle_rads_row, angle_rads_col], axis=1)[np.newaxis, ...]

    return tf.cast(pos_encoding, dtype=tf.float32)
```

❖ Our Model (Transforms):

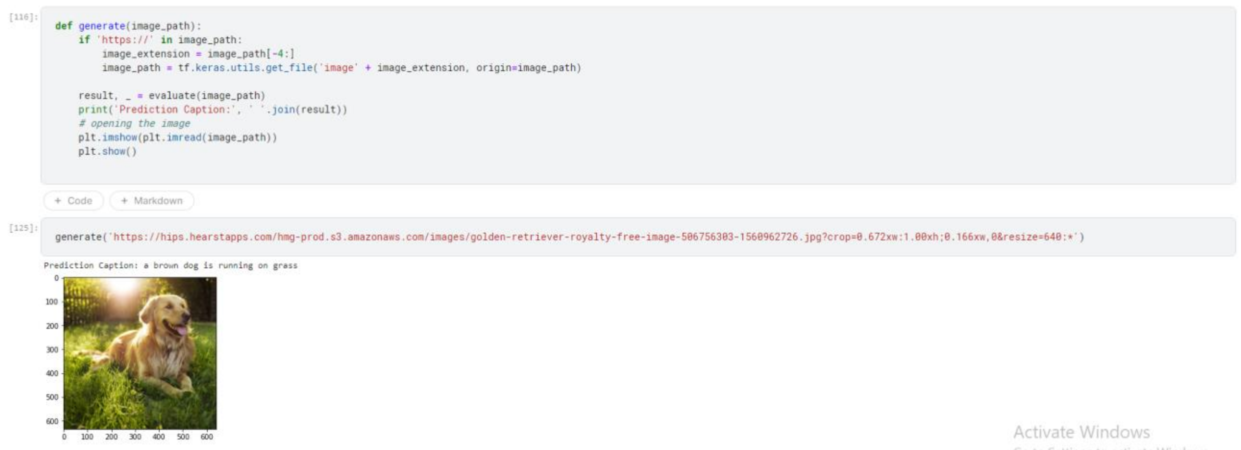
My code defines a transformer model for sequence-to-sequence tasks, such as machine translation or text generation. The model consists of an encoder and a decoder, each comprising multiple layers of encoder and decoder blocks, respectively. The encoder processes the input sequences and produces a context representation, while the decoder generates the output sequences based on this context. Both the encoder and decoder blocks consist of multi-head self-attention mechanisms and feed-forward neural networks, with layer normalization and dropout applied for regularization.

The encoder processes the input sequences by first embedding them into a higher-dimensional space and adding positional encoding to capture the sequential order of the input tokens. The encoder blocks then apply multi-head self-attention to the embedded sequences, followed by feed-forward networks, layer normalization, and dropout. This process is repeated for a specified number of layers to capture complex patterns in the input sequences.

The decoder, on the other hand, takes the context representation from the encoder and generates output sequences token by token. Similar to the encoder, the decoder uses embeddings and positional encoding for the input tokens. It also includes masked multi-head self-attention to prevent attending to future tokens and cross-attention to attend to the encoder's output. The decoder blocks perform these operations iteratively to generate the output sequences, and the final linear layer produces the output logits for each token in the vocabulary.



```
Epoch 1 Batch 0 Loss 8.5287
Epoch 1 Batch 50 Loss 7.7821
Epoch 1 Batch 100 Loss 7.1777
Epoch 1 Batch 150 Loss 6.7295
Epoch 1 Batch 200 Loss 6.3608
Epoch 1 Batch 250 Loss 6.0372
Epoch 1 Batch 300 Loss 5.7584
Epoch 1 Batch 350 Loss 5.5233
Epoch 1 Batch 400 Loss 5.3261
Epoch 1 Batch 450 Loss 5.1538
Epoch 1 Batch 500 Loss 5.0003
Epoch 1 Loss 4.9855
Time taken for 1 epoch: 298.46089601516724 secs
```



```
[116]:
def generate(image_path):
    if 'https://' in image_path:
        image_extension = image_path[-4:]
        image_path = tf.keras.utils.get_file('image' + image_extension, origin=image_path)

    result, _ = evaluate(image_path)
    print('Prediction Caption:', ' '.join(result))
    # opening the image
    plt.imshow(plt.imread(image_path))
    plt.show()
```

```
[125]:
generate('https://hips.hearstapps.com/hmg-prod.s3.amazonaws.com/images/golden-retriever-royalty-free-image-506756303-1560962726.jpg?crop=0.672xw:1.00xh;0.166xw,0&resize=640:*')
```

```
Prediction Caption: a brown dog is running on grass
```