```
%pip install tensorflow pandas numpy matplotlib pillow tqdm requests scikit-lea
```

```
46.9/46.9 MB 15.0 MB/s eta 0:00
322.2/322.2 kB 13.9 MB/s eta 0:
95.2/95.2 kB 6.8 MB/s eta 0:00:
11.5/11.5 MB 55.8 MB/s eta 0:00
72.0/72.0 kB 5.0 MB/s eta 0:00:
62.5/62.5 kB 3.8 MB/s eta 0:00:
```

```
import tensorflow as tf  # Import the TensorFlow library for deep learning task
import os  # Import the os module for interacting with the operating system
import json  # Import the json module for working with JSON data
import pandas as pd  # Import pandas for data manipulation and analysis
import re  # Import the re module for regular expressions
import numpy as np  # Import numpy for numerical operations
import time  # Import the time module for time-related functions
import matplotlib.pyplot as plt  # Import matplotlib for plotting and visualiza
import collections  # Import collections module for specialized container datat
import random  # Import random module for generating random numbers
import requests  # Import requests for making HTTP requests
from math import sqrt  # Import sqrt function from math module for square root
from PIL import Image  # Import Image class from PIL (Python Imaging Library) f
from tqdm.auto import tqdm  # Import tqdm for progress bar visualization
```

```
!pip install pycocotools
!mkdir -p data && cd data && \
 wget http://images.cocodataset.org/zips/train2017.zip && unzip train2017.zip &
 wget http://images.cocodataset.org/zips/val2017.zip && unzip val2017.zip && \
 wget http://images.cocodataset.org/annotations/annotations_trainval2017.zip &&
```

```
Streaming output truncated to the last 5000 lines.
  extracting: val2017/000000577584.jpg
  extracting: val2017/000000346905.jpg
  extracting: val2017/000000433980.jpg
  extracting: val2017/000000228144.jpg
  extracting: val2017/000000041872.jpg
  extracting: val2017/000000117492.jpg
  extracting: val2017/000000368900.jpg
  extracting: val2017/000000376900.jpg
  extracting: val2017/000000352491.jpg
  extracting: val2017/000000330790.jpg
  extracting: val2017/000000384850.jpg
  extracting: val2017/000000032735.jpg
  extracting: val2017/000000197004.jpg
  extracting: val2017/000000526751.jpg
  extracting: val2017/000000041488.jpg
  extracting: val2017/000000153632.jpg
  extracting: val2017/000000501523.jpg
  extracting: val2017/000000405691.jpg
  extracting: val2017/000000040757.jpg
  extracting: val2017/000000219485.jpg
  extracting: val2017/000000428280.jpg
  extracting: val2017/000000209222.jpg
```

```
    extracting: val2017/000000353051.jpg
    extracting: val2017/000000191471.jpg
    extracting: val2017/000000539962.jpg
    extracting: val2017/000000462371.jpg
    extracting: val2017/000000574315.jpg
    extracting: val2017/000000005037.jpg
    extracting: val2017/000000083540.jpg
    extracting: val2017/000000145665.jpg
    extracting: val2017/000000174231.jpg
    extracting: val2017/000000389812.jpg
    extracting: val2017/000000245513.jpg
    extracting: val2017/000000122046.jpg
    extracting: val2017/000000143931.jpg
    extracting: val2017/000000555005.jpg
    extracting: val2017/000000142472.jpg
    extracting: val2017/000000246883.jpg
    extracting: val2017/000000459272.jpg
    extracting: val2017/000000356261.jpg
    extracting: val2017/000000169996.jpg
    extracting: val2017/000000311909.jpg
    extracting: val2017/000000253433.jpg
    extracting: val2017/000000396568.jpg
    extracting: val2017/000000089045.jpg
    extracting: val2017/000000387383.jpg
    extracting: val2017/000000095155.jpg
    extracting: val2017/000000036494.jpg
    extracting: val2017/000000495054.jpg
    extracting: val2017/000000297595.jpg
    extracting: val2017/000000030213.jpg
    extracting: val2017/000000357903.jpg
    extracting: val2017/000000231237.jpg
    extracting: val2017/000000182805.jpg
    extracting: val2017/000000147740.jpg
    extracting: val2017/000000424721.jpg
    extracting: val2017/000000165257.jpg
```

```
import json  # Import the json module for working with JSON data
with open(f'/content/data/annotations/captions_train2017.json', 'r') as f:  # (
    data = json.load(f)  # Load the JSON data
    data = data['annotations']  # Extract the 'annotations' part of the data

img_cap_pairs = []  # Initialize an empty list to hold image-caption pairs

for sample in data:  # Iterate over each sample in the annotations
    img_name = '%012d.jpg' % sample['image_id']  # Format the image_id into a 1
    img_cap_pairs.append([img_name, sample['caption']])  # Append the image fil

captions = pd.DataFrame(img_cap_pairs, columns=['image', 'caption'])  # Create
captions['image'] = captions['image'].apply(  # Update the 'image' column in th
    lambda x: f'/content/data/train2017/{x}'  # Prepend the base path to each i
)
captions = captions.sample(1000)  # Randomly sample 1,000 entries from the Data
captions = captions.reset_index(drop=True)  # Reset the index of the DataFrame,
captions.to_csv('captions_sample.csv', index=False)
captions.head()  # Display the first few rows of the DataFrame
```

	image	caption
0	/content/data/train2017/000000567812.jpg	A man has a teddy bear around his neck.
1	/content/data/train2017/000000508100.jpg	A large grass covered field under a mountain.
2	/content/data/train2017/000000401854.jpg	a crowd of people holding umbrellas in the rain
3	/content/data/	A case of ice treats and soda beside a

Next steps: Generate code with captions | View recommended plots | New interactive sheet

```
from IPython.display import FileLink

# Save the DataFrame to a CSV file
captions.to_csv('captions_sample.csv', index=False)

# Create a link to download the file
FileLink('captions_sample.csv')
```

captions_sample.csv

```
def preprocess(text):
    # Convert the text to lowercase
    text = text.lower()

    # Remove all characters that are not word characters or whitespace
    text = re.sub(r'[^\w\s]', '', text)

    # Replace one or more whitespace characters with a single space
    text = re.sub('\s+', ' ', text)

    # Remove leading and trailing whitespace
    text = text.strip()

    # Add '[start]' at the beginning and '[end]' at the end of the text
    text = '[start] ' + text + ' [end]'

    # Return the preprocessed text
    return text
```

```
captions['caption'] = captions['caption'].apply(preprocess)  # Apply the preprc
captions.head()  # Display the first few rows of the DataFrame to check the pre
```

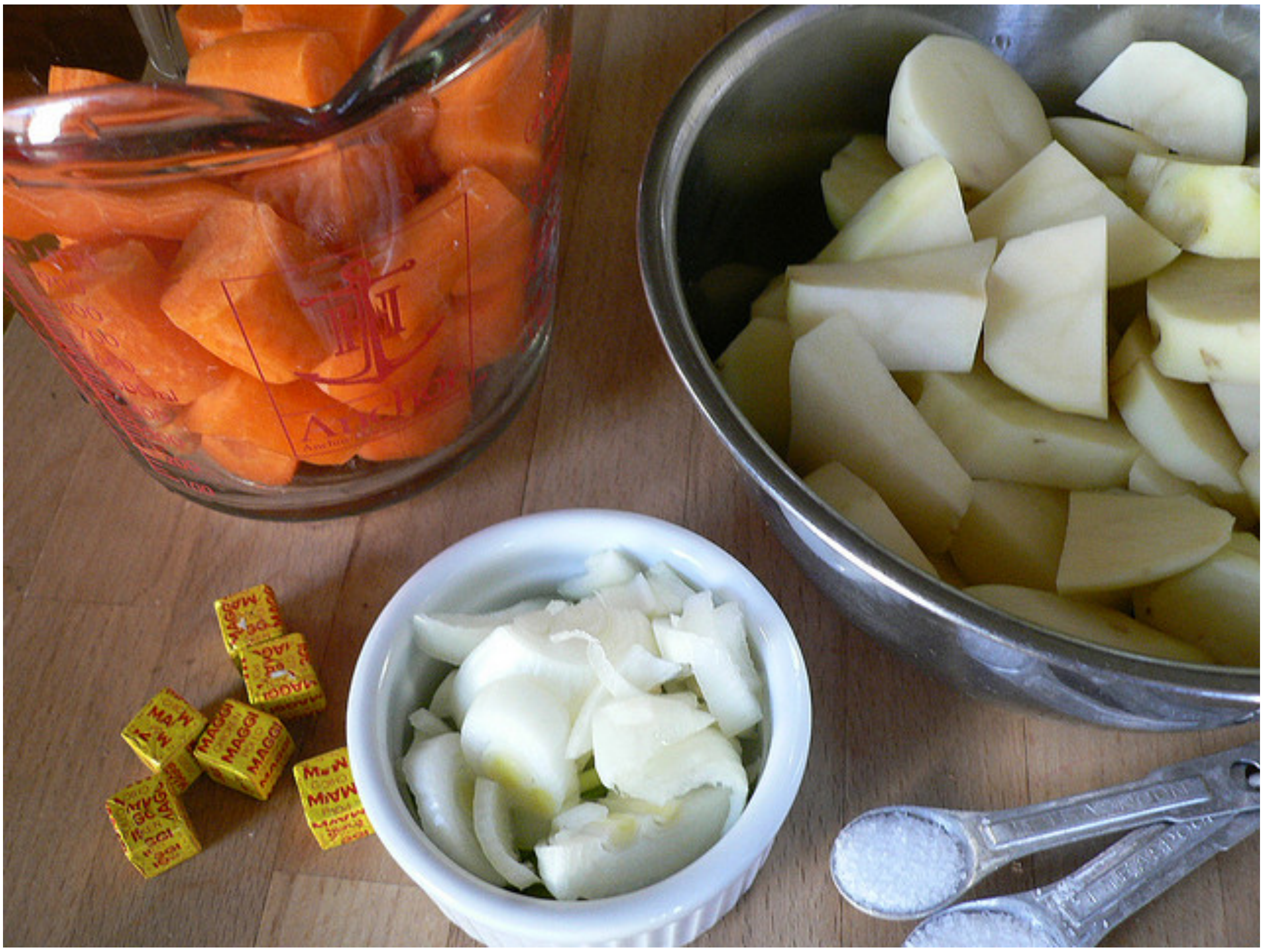
	image	caption
0	/content/data/train2017/000000567812.jpg	[start] a man has a teddy bear around his neck...

1	/content/data/ train2017/000000508100.jpg	[start] a large grass covered field under a mo...
2	/content/data/ train2017/000000401854.jpg	[start] a crowd of people holding umbrellas in...
3	/content/data/	[start] a case of ice treats and soda

Next steps: Generate code with captions | View recommended plots | New interactive sheet

```
random_row = captions.sample(1).iloc[0]  # Randomly sample one row from the Dat
print(random_row.caption)  # Print the caption of the randomly selected row
print()  # Print an empty line for separation
im = Image.open(random_row.image)  # Open the image file corresponding to the r
im  # Display the image
```

```
[start] several containers of different foods including carrots potatoes an
```




```
MAX_LENGTH = 40  # Define the maximum length of the sequences (captions)
VOCABULARY_SIZE = 15000  # Define the size of the vocabulary
BATCH_SIZE = 64  # Define the batch size for training
BUFFER_SIZE = 1000  # Define the buffer size for shuffling the dataset
EMBEDDING_DIM = 512  # Define the dimension of the embedding layer
UNITS = 512  # Define the number of units in the recurrent neural network (RNN)
EPOCHS = 5  # Define the number of epochs for training
```

```
tokenizer = tf.keras.layers.TextVectorization(  # Initialize a TextVectorizatic
    max_tokens=VOCABULARY_SIZE,  # Set the maximum number of tokens (size of th
    standardize=None,  # Do not apply additional standardization since the text
    output_sequence_length=MAX_LENGTH  # Set the output sequence length (maximu
)

tokenizer.adapt(captions['caption'])  # Adapt the tokenizer to the captions in
```

```
tokenizer.vocabulary_size()  # Get the size of the vocabulary built by the toke
```

```
1600
```

```
import pickle  # Import the pickle module for serializing and deserializing Pyt

pickle.dump(tokenizer.get_vocabulary(), open('vocab_coco.file', 'wb'))  # Seria
```

```
word2idx = tf.keras.layers.StringLookup(  # Create a StringLookup layer to map
    mask_token="",  # Specify that there is no mask token
    vocabulary=tokenizer.get_vocabulary()  # Use the vocabulary from the tokeni
)

idx2word = tf.keras.layers.StringLookup(  # Create a StringLookup layer to map
    mask_token="",  # Specify that there is no mask token
    vocabulary=tokenizer.get_vocabulary(),  # Use the same vocabulary from the
    invert=True  # Set invert=True to invert the mapping (indices to words)
)
```

```
img_to_cap_vector = collections.defaultdict(list)  # Create a default dictionar
for img, cap in zip(captions['image'], captions['caption']):  # Iterate over im
    img_to_cap_vector[img].append(cap)  # Append each caption to the list corre

img_keys = list(img_to_cap_vector.keys())  # Get a list of all image keys
random.shuffle(img_keys)  # Shuffle the list of image keys

slice_index = int(len(img_keys) * 0.8)  # Determine the index to split the data
img_name_train_keys, img_name_val_keys = (img_keys[:slice_index],  # Split the
                                          img_keys[slice_index:])  # and valida

train_imgs = []  # Initialize a list to hold training images
train_captions = []  # Initialize a list to hold training captions
for imgt in img_name_train_keys:  # Iterate over training image keys
    capt_len = len(img_to_cap_vector[imgt])  # Get the number of captions for e
    train_imgs.extend([imgt] * capt_len)  # Extend the training images list wit
    train_captions.extend(img_to_cap_vector[imgt])  # Extend the training capti

val_imgs = []  # Initialize a list to hold validation images
val_captions = []  # Initialize a list to hold validation captions
for imgv in img_name_val_keys:  # Iterate over validation image keys
```

```
for imgv in img_name_val_keys:  # Iterate over validation image keys
    capv_len = len(img_to_cap_vector[imgv])  # Get the number of captions for e
    val_imgs.extend([imgv] * capv_len)  # Extend the validation images list wit
    val_captions.extend(img_to_cap_vector[imgv])  # Extend the validation capti
```

```
len(train_imgs), len(train_captions), len(val_imgs), len(val_captions)  # Get t
```

```
(800, 800, 200, 200)
```

```
def load_data(img_path, caption):
    img = tf.io.read_file(img_path)  # Read the image file from the given path
    img = tf.io.decode_jpeg(img, channels=3)  # Decode the JPEG image to a tens
    img = tf.keras.layers.Resizing(299, 299)(img)  # Resize the image to 299x29
    img = tf.keras.applications.inception_v3.preprocess_input(img)  # Preproces
    caption = tokenizer(caption)  # Tokenize the caption using the tokenizer
    return img, caption  # Return the preprocessed image and the tokenized capt
```

```
# Create a TensorFlow dataset from the training images and captions
train_dataset = tf.data.Dataset.from_tensor_slices(
    (train_imgs, train_captions)
)

# Apply the load_data function to each element in the dataset, shuffle, and bat
train_dataset = train_dataset.map(
    load_data, num_parallel_calls=tf.data.AUTOTUNE  # Use AUTOTUNE to optimize
).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)  # Batch the dataset with the specifie

# Create a TensorFlow dataset from the validation images and captions
val_dataset = tf.data.Dataset.from_tensor_slices(
    (val_imgs, val_captions)
)

# Apply the load_data function to each element in the dataset, shuffle, and bat
val_dataset = val_dataset.map(
    load_data, num_parallel_calls=tf.data.AUTOTUNE  # Use AUTOTUNE to optimize
).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)  # Shuffle and Batch the dataset with
```

```
# Create a sequential model for image augmentation
image_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),  # Randomly flip images horizonta
    tf.keras.layers.RandomRotation(0.2),  # Randomly rotate images by up to 20%
    tf.keras.layers.RandomContrast(0.3),  # Randomly adjust contrast by up to 3
])
```

```
def CNN_Encoder():
```

```
def CNN_Encoder():
    # Load the InceptionV3 model pre-trained on ImageNet, excluding the top cla
    inception_v3 = tf.keras.applications.InceptionV3(
        include_top=False,  # Exclude the top classification layer
        weights='imagenet'  # Load weights pre-trained on ImageNet
    )

    # Get the output of the InceptionV3 model
    output = inception_v3.output
    # Reshape the output to have a shape of (-1, channels), where -1 represents
    output = tf.keras.layers.Reshape((-1, output.shape[-1]))(output)

    # Create a new model that takes the same input as InceptionV3 and outputs t
    cnn_model = tf.keras.models.Model(inception_v3.input, output)
    return cnn_model  # Return the CNN model


class TransformerEncoderLayer(tf.keras.layers.Layer):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        # Define the first layer normalization layer
        self.layer_norm_1 = tf.keras.layers.LayerNormalization()
        # Define the second layer normalization layer
        self.layer_norm_2 = tf.keras.layers.LayerNormalization()
        # Define the multi-head attention layer
        self.attention = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads,  # Number of attention heads
            key_dim=embed_dim  # Dimension of the attention key
        )
        # Define a dense layer with ReLU activation
        self.dense = tf.keras.layers.Dense(embed_dim, activation="relu")

    def call(self, x, training):
        # Apply the first layer normalization
        x = self.layer_norm_1(x)
        # Apply the dense layer
        x = self.dense(x)

        # Compute the attention output
        attn_output = self.attention(
            query=x,  # Query for attention
            value=x,  # Value for attention
            key=x,  # Key for attention
            attention_mask=None,  # No attention mask
            training=training  # Training flag
        )

        # Apply the second layer normalization to the sum of input and attentic
        x = self.layer_norm_2(x + attn_output)
        return x  # Return the output


class Embeddings(tf.keras.layers.Layer):
    def __init__(self, vocab_size, embed_dim, max_len):
        super().__init__()
```

```
        # Define the token embeddings layer
        self.token_embeddings = tf.keras.layers.Embedding(
            vocab_size, embed_dim  # Vocabulary size and embedding dimension
        )
        # Define the position embeddings layer
        self.position_embeddings = tf.keras.layers.Embedding(
            max_len, embed_dim, input_shape=(None, max_len)  # Maximum length a
        )

    def call(self, input_ids):
        # Get the length of the input sequence
        length = tf.shape(input_ids)[-1]
        # Create a tensor with position indices
        position_ids = tf.range(start=0, limit=length, delta=1)
        # Expand dimensions to match the input shape
        position_ids = tf.expand_dims(position_ids, axis=0)

        # Get the token embeddings for the input IDs
        token_embeddings = self.token_embeddings(input_ids)
        # Get the position embeddings for the position IDs
        position_embeddings = self.position_embeddings(position_ids)

        # Return the sum of token embeddings and position embeddings
        return token_embeddings + position_embeddings



class TransformerDecoderLayer(tf.keras.layers.Layer):
    def __init__(self, embed_dim, units, num_heads):
        super().__init__()
        # Define the embedding layer using the custom Embeddings class
        self.embedding = Embeddings(
            tokenizer.vocabulary_size(), embed_dim, MAX_LENGTH  # Pass the voca
        )

        # Define the first multi-head attention layer
        self.attention_1 = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads,  # Number of attention heads
            key_dim=embed_dim,  # Dimension of the attention key
            dropout=0.1  # Dropout rate
        )
        # Define the second multi-head attention layer
        self.attention_2 = tf.keras.layers.MultiHeadAttention(
            num_heads=num_heads,  # Number of attention heads
            key_dim=embed_dim,  # Dimension of the attention key
            dropout=0.1  # Dropout rate
        )

        # Define layer normalization layers
        self.layernorm_1 = tf.keras.layers.LayerNormalization()  # First layer
        self.layernorm_2 = tf.keras.layers.LayerNormalization()  # Second layer
        self.layernorm_3 = tf.keras.layers.LayerNormalization()  # Third layer

        # Define feedforward network layers
```

```
        self.ffn_layer_1 = tf.keras.layers.Dense(units, activation='relu')  # F
        self.ffn_layer_2 = tf.keras.layers.Dense(embed_dim)  # Second dense lay

        # Define the output layer
        self.out = tf.keras.layers.Dense(tokenizer.vocabulary_size(), activatic

        # Define dropout layers
        self.dropout_1 = tf.keras.layers.Dropout(0.3)  # First dropout layer wi
        self.dropout_2 = tf.keras.layers.Dropout(0.5)  # Second dropout layer w

    def call(self, input_ids, encoder_output, training, mask=None):
        embeddings = self.embedding(input_ids)  # Get the embeddings for the ir

        combined_mask = None  # Initialize combined mask
        padding_mask = None  # Initialize padding mask

        if mask is not None:  # Check if mask is provided
            # Generate causal mask
            causal_mask = self.get_causal_attention_mask(embeddings)  # Create
            padding_mask = tf.cast(mask[:, :, tf.newaxis], dtype=tf.int32)  # C
            combined_mask = tf.cast(mask[:, tf.newaxis, :], dtype=tf.int32)  #
            combined_mask = tf.minimum(combined_mask, causal_mask)  # Take mini

        # Apply the first attention layer
        attn_output_1 = self.attention_1(
            query=embeddings,  # Query is the embeddings
            value=embeddings,  # Value is also the embeddings
            key=embeddings,  # Key is also the embeddings
            attention_mask=combined_mask,  # Use the combined mask
            training=training  # Specify if training
        )

        out_1 = self.layernorm_1(embeddings + attn_output_1)  # Add the attenti

        # Apply the second attention layer
        attn_output_2 = self.attention_2(
            query=out_1,  # Query is the output from the first attention layer
            value=encoder_output,  # Value is the encoder output
            key=encoder_output,  # Key is the encoder output
            attention_mask=padding_mask,  # Use the padding mask
            training=training  # Specify if training
        )

        out_2 = self.layernorm_2(out_1 + attn_output_2)  # Add the second atter

        ffn_out = self.ffn_layer_1(out_2)  # Apply the first feedforward layer
        ffn_out = self.dropout_1(ffn_out, training=training)  # Apply the first
        ffn_out = self.ffn_layer_2(ffn_out)  # Apply the second feedforward lay

        ffn_out = self.layernorm_3(ffn_out + out_2)  # Add the feedforward outp
        ffn_out = self.dropout_2(ffn_out, training=training)  # Apply the secor
        preds = self.out(ffn_out)  # Get the final predictions from the output
        return preds  # Return the predictions

    def get_causal_attention_mask(self, inputs):
```

```
    def get_causal_attention_mask(self, inputs):
        input_shape = tf.shape(inputs)  # Get the shape of the inputs
        batch_size, sequence_length = input_shape[0], input_shape[1]  # Get bat
        i = tf.range(sequence_length)[:, tf.newaxis]  # Create a range tensor f
        j = tf.range(sequence_length)  # Create another range tensor for sequer
        mask = tf.cast(i >= j, dtype="int32")  # Create a causal mask by compar
        mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))  # Reshape
        mult = tf.concat(
            [tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.int32
            axis=0  # Concatenate along axis 0
        )
        return tf.tile(mask, mult)  # Tile the mask to match the batch size and


class ImageCaptioningModel(tf.keras.Model):

    def __init__(self, cnn_model, encoder, decoder, image_aug=None):
        super().__init__()
        self.cnn_model = cnn_model
        self.encoder = encoder
        self.decoder = decoder
        self.image_aug = image_aug
        self.loss_tracker = tf.keras.metrics.Mean(name="loss")
        self.acc_tracker = tf.keras.metrics.Mean(name="accuracy")  # Ensure acc

    def calculate_loss(self, y_true, y_pred, mask):
        loss = self.loss(y_true, y_pred)
        mask = tf.cast(mask, dtype=loss.dtype)
        loss *= mask
        return tf.reduce_sum(loss) / tf.reduce_sum(mask)

    def calculate_accuracy(self, y_true, y_pred, mask):
        accuracy = tf.equal(y_true, tf.argmax(y_pred, axis=2))
        accuracy = tf.math.logical_and(mask, accuracy)
        accuracy = tf.cast(accuracy, dtype=tf.float32)
        mask = tf.cast(mask, dtype=tf.float32)
        return tf.reduce_sum(accuracy) / tf.reduce_sum(mask)

    def compute_loss_and_acc(self, img_embed, captions, training=True):
        encoder_output = self.encoder(img_embed, training=True)
        y_input = captions[:, :-1]
        y_true = captions[:, 1:]
        mask = (y_true != 0)
        y_pred = self.decoder(y_input, encoder_output, training=True, mask=mask
        loss = self.calculate_loss(y_true, y_pred, mask)
        acc = self.calculate_accuracy(y_true, y_pred, mask)
        return loss, acc

    def train_step(self, batch):
        imgs, captions = batch

        if self.image_aug:
            imgs = self.image_aug(imgs)

        img_embed = self.cnn_model(imgs)
```

```
        with tf.GradientTape() as tape:
            loss, acc = self.compute_loss_and_acc(img_embed, captions)

        train_vars = self.encoder.trainable_variables + self.decoder.trainable_
        grads = tape.gradient(loss, train_vars)
        self.optimizer.apply_gradients(zip(grads, train_vars))
        self.loss_tracker.update_state(loss)
        self.acc_tracker.update_state(acc)

        return {"loss": self.loss_tracker.result(), "accuracy": self.acc_tracke

    def test_step(self, batch):
        imgs, captions = batch

        img_embed = self.cnn_model(imgs)

        loss, acc = self.compute_loss_and_acc(img_embed, captions, training=Fal

        self.loss_tracker.update_state(loss)
        self.acc_tracker.update_state(acc)

        return {"loss": self.loss_tracker.result(), "accuracy": self.acc_tracke

    @property
    def metrics(self):
        return [self.loss_tracker, self.acc_tracker]
```

```
# Instantiate the TransformerEncoderLayer with the specified embedding dimensic
encoder = TransformerEncoderLayer(EMBEDDING_DIM, 1)

# Instantiate the TransformerDecoderLayer with the specified embedding dimensic
decoder = TransformerDecoderLayer(EMBEDDING_DIM, UNITS, 8)

# Instantiate the CNN model for encoding images
cnn_model = CNN_Encoder()

# Create the image captioning model with the specified CNN model, encoder, decc
caption_model = ImageCaptioningModel(
    cnn_model=cnn_model,  # CNN model for image feature extraction
    encoder=encoder,  # Encoder model
    decoder=decoder,  # Decoder model
    image_aug=image_augmentation,  # Image augmentation layer
)
```

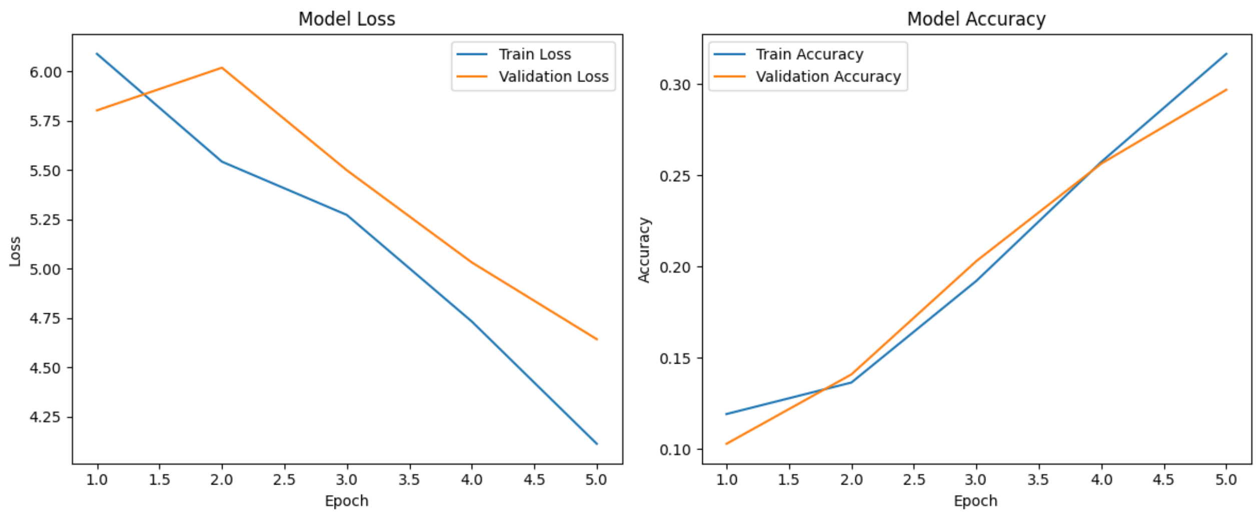
```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:
  super().__init__(**kwargs)
Downloading data from https://storage.googleapis.com/tensorflow/keras-appli
87910968/87910968 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step
```

```
# Define the loss function using SparseCategoricalCrossentropy
cross_entropy = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=False,  # Indicates that the predictions are probabilities
    reduction="none"  # Do not reduce the loss, keep it element-wise
)

# Define an early stopping callback to stop training if validation performance
early_stopping = tf.keras.callbacks.EarlyStopping(
    patience=3,  # Number of epochs to wait after the last improvement before s
    restore_best_weights=True  # Restore model weights from the epoch with the
)

# Compile the captioning model
caption_model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=cross_entropy,
    metrics=["accuracy"]  # Include accuracy in the metrics
)



# Train the captioning model
history = caption_model.fit(
    train_dataset,  # Training dataset
    epochs=EPOCHS,  # Number of epochs to train for
    validation_data=val_dataset,  # Validation dataset
    callbacks=[early_stopping]  # List of callbacks to apply during training
)
```

```
Epoch 1/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 429s 30s/step - accuracy: 0.0971 - loss: 6.5444
Epoch 2/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 371s 29s/step - accuracy: 0.1311 - loss: 5.5368
Epoch 3/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 383s 29s/step - accuracy: 0.1835 - loss: 5.3574
Epoch 4/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 439s 29s/step - accuracy: 0.2449 - loss: 4.8116
Epoch 5/5
13/13 ━━━━━━━━━━━━━━━━━━━━ 402s 31s/step - accuracy: 0.3030 - loss: 4.2136
```

```
# Adjusting the x-axis values to start from 1
epochs = range(1, len(history.history['loss']) + 1)

# Plotting Loss and Accuracy
plt.figure(figsize=(12, 5))  # Create a figure with a specific size

# Subplot for loss
plt.subplot(1, 2, 1)  # 1 row, 2 columns, 1st subplot
plt.plot(epochs, history.history['loss'], label='Train Loss')  # Plot training
plt.plot(epochs, history.history['val_loss'], label='Validation Loss')  # Plot
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()  # Add a legend to differentiate between the training and validati
```

```
plt.legend()  # Add a legend to differentiate between the training and validati

# Subplot for accuracy
plt.subplot(1, 2, 2)  # 1 row, 2 columns, 2nd subplot
plt.plot(epochs, history.history['accuracy'], label='Train Accuracy')  # Plot t
plt.plot(epochs, history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend()  # Add a legend to differentiate between the training and validati

plt.tight_layout()  # Adjust subplots to fit into the figure area.
plt.show()  # Display the plot
```




```
%pip install gradio -q
import gradio as gr  # Import Gradio for creating the web interface
import pandas as pd  # Import pandas for handling the DataFrame
from sklearn.feature_extraction.text import TfidfVectorizer  # Import TF-IDF ve
from sklearn.metrics.pairwise import cosine_similarity  # Import cosine similar
import numpy as np  # Import NumPy for numerical operations
import tensorflow as tf  # Import TensorFlow for deep learning operations
from PIL import Image  # Import PIL for image processing
import os  # Import os for file handling

# Load Captions DataFrame
```

```
captions = pd.read_csv('captions_sample.csv')  # Replace with your actual file

# Compute TF-IDF embeddings for all captions in the dataset
vectorizer = TfidfVectorizer()  # Initialize the TF-IDF vectorizer
caption_embeddings = vectorizer.fit_transform(captions['caption'].tolist())  #

def find_similar_caption(prompt, caption_embeddings, captions):
    # Compute TF-IDF embedding for the input prompt
    prompt_embedding = vectorizer.transform([prompt])  # Transform the input pr
    # Compute cosine similarity between the prompt and all captions
    similarities = cosine_similarity(prompt_embedding, caption_embeddings)  # C
    # Find the index of the most similar caption
    most_similar_idx = np.argmax(similarities)  # Get the index of the highest
    return captions.iloc[most_similar_idx]  # Return the most similar caption e

def load_image_from_path(img_path):
    # Read and preprocess the image from the given path
    img = tf.io.read_file(img_path)  # Read the image file
    img = tf.io.decode_jpeg(img, channels=3)  # Decode the JPEG image
    img = tf.keras.layers.Resizing(299, 299)(img)  # Resize the image to 299x29
    img = tf.image.convert_image_dtype(img, tf.float32)  # Convert the image to
    img = tf.keras.applications.inception_v3.preprocess_input(img)  # Preproces
    return img  # Return the preprocessed image

def generate_caption(img_path):
    # Generate a caption for the image at the given path
    img = load_image_from_path(img_path)  # Load and preprocess the image

    img = tf.expand_dims(img, axis=0)  # Expand dimensions to create a batch of
    img_embed = caption_model.cnn_model(img)  # Get image embeddings from the C
    img_encoded = caption_model.encoder(img_embed, training=False)  # Encode th

    y_inp = '[start]'  # Initialize the input with the start token
    for i in range(MAX_LENGTH - 1):  # Iterate until the maximum length is reac
        tokenized = tokenizer([y_inp])[:, :-1]  # Tokenize the current input
        mask = tf.cast(tokenized != 0, tf.int32)  # Create a mask for non-zero
        pred = caption_model.decoder(tokenized, img_encoded, training=False, ma

        pred_idx = np.argmax(pred[0, i, :])  # Get the index of the highest pro
        pred_idx = tf.convert_to_tensor(pred_idx)  # Convert the prediction inc
        pred_word = idx2word(pred_idx).numpy().decode('utf-8')  # Convert the i
        if pred_word == '[end]':  # Stop if the end token is predicted
            break

        y_inp += ' ' + pred_word  # Append the predicted word to the input sequ

    y_inp = y_inp.replace('[start] ', '')  # Remove the start token from the fi
    return y_inp  # Return the generated caption

def get_image_from_prompt(prompt, caption_embeddings, captions):
    # Find the most similar caption and return the image path and caption
    similar_caption_entry = find_similar_caption(prompt, caption_embeddings, ca
    img_path = similar_caption_entry['image']  # Get the image path
    return img_path  # Return the image path
```

```
def generate_caption_and_display(prompt):
    # Generate a caption and display the image for a given prompt
    img_path = get_image_from_prompt(prompt, caption_embeddings, captions)  # G
    generate_caption(img_path)  # Generate the caption for the image (side effe

    img = Image.open(img_path)  # Open the image

    # Ensure the directory for saving the image exists
    output_dir = "/mnt/data"
    os.makedirs(output_dir, exist_ok=True)

    # Save the image temporarily to provide a download link
    img_temp_path = os.path.join(output_dir, "temp_image.png")
    img.save(img_temp_path)

    return img  # Return the image

# Define the Gradio interface
iface = gr.Interface(
    fn=generate_caption_and_display,  # Function to be called
    inputs=[
        gr.Textbox(label="Prompt")  # Textbox input for the prompt
    ],
    outputs=[
        gr.Image(type="pil", label="Image")  # Image output
    ],
    title="Text to Image Generator",  # Title of the interface
    description="Enter a text prompt to find a similar image."  # Description c
)

# Launch the interface
iface.launch()  # Launch the Gradio interface
```

```
# Save the weights of the captioning model to a file named 'model.weights.h5'
caption_model.save_weights('model.weights.h5')
```

```
Start coding or generate with AI.
```