PS/2019/246 W.A.S.H PERERA

Importing Relevant Libraries

Preparing Data

```
import random
import tensorflow as tf
import string
import re
from tensorflow import keras
from tensorflow.keras import layers
```

Mounting the Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mour
```

Reading the Data File

```
text_file = "/content/drive/My Drive/Colab Notebooks/Content.txt"
with open(text_file) as f:  # Opening the file in read mode
    lines = f.read().split("\n")[:-1]   # Reading the lines from the file and splitting by r
```

```
# Print the first 20 lines of the file
i = 0
for line in lines:
  print(line)
  i = i + 1
  if(i==20):
    break
```

```
the water was falling drop by drop      වතුර බිංදුවෙන් බිංදුව වැටෙනවා
the water had started a fire    වතුරෙන් ගින්නක් ඇවිළුණා
then my heart was remembering you       මගේ හදවත ඔයාව සිහිපත් කළා
```

```
beloved now you only tell me    සොදුරියේ දැන් මට කියන්න
what should i do        මම මොනවද කරන්න ඕනේ
without you i can't live anymore        ඔයා නැතුව මට තවත් ජීවත් වෙන්න බෑ
without you what is my existence        ඔයා නැතුව මගේ පැවැත්ම මොකක්ද
then i'll be separated from myself      ඊට පස්සේ මම මාවම වෙන් කර ගන්නවා
i live every day for you        මම හැමදාම ඔයා වෙනුවෙන් ජීවත් වෙනවා
if i become yours       මම ඔයාගේ වුනොත්
because you are the one මොකද ඔයා තමයි
now you are the one     දැන් ඔයා තමයි
you are my life now     ඔයා තමයි දැන් මගේ ජීවිතය
both peace and pain my love are now only you    සාමය සහ වේදනාව මගේ ආදරණීය දැන් ඔ
what kind of relationship is ours       මොන වගේ සම්බන්ධතාවයක්ද අපේ
i cannot bear a moment without you      මට ඔයා නැතුව මොහොතක්වත් ඉන්න බෑ
i have given you all my time    මම ඔයාට මගේ මුළු කාලයම දුන්නා
your name is on every breath i take     මම ගන්න හැම හුස්මකම ඔයාගේ නම තියෙනවා
thank you for your understanding        ඔයාගේ තේරුම් ගැනීමට ස්තූතියි
if i get separated from you     මම ඔයාගෙන් වෙන් වුනොත්
```

```
# Print the last 10 lines of the file
for x in range(len(lines)-10,len(lines)):
  print(lines[x])
```

```
Once upon a time in a small village there lived a young girl named Lily who had a specia
In a distant land a brave knight named Sir Arthur embarked on a perilous journey to resc
Emily a curious explorer set out on an adventure to uncover the hidden treasures of an a
In the peaceful town of Willowbrook a mischievous cat named Oliver had a talent for gett
Sarah a talented pianist dreamt of performing on the grand stage of Carnegie Hall and sp
Deep in the enchanted forest a group of woodland creatures led by a wise old owl embarke
On a sunny summer day a group of friends gathered at the beach for a fun-filled day of s
In a quaint seaside village a mysterious stranger arrived bringing with them an air of e
Thomas an aspiring writer found inspiration in the bustling streets of a vibrant city wh
In a land of mythical creatures a young dragon named Ember struggled to control her fier
```

Spliting the English and Sinhala Translation Pairs

```
text_pairs = []
for line in lines:
    english, sinhala = line.split("\t") # Split the line by tab to separate English and Sinh
    sinhala = "[start] " + sinhala + " [end]" # Add start and end tokens to the Sinhala tran
    text_pairs.append((english, sinhala))  # Append the English-Sinhalah pair to text_pairs
```

```
# Print 3 randomly chosen pairs
for i in range(3):
    print(random.choice(text_pairs))
```

```
("so if you don't mind asking", '[start] ඉතිං ඇහුවාට කමක් නැත්නම් [end]')
("do you have the dog's certificates  ", '[start] ඔයා ළඟ බල්ලාගේ සහතික තියෙනවාද
('give it a minute', '[start] මිනිත්තුවක් දෙන්න [end]')
```

Randomizing the Data

```
import random
random.shuffle(text_pairs)
```

Spliting the Data into Training, Validation and Testing

```
num_val_samples = int(0.15 * len(text_pairs))  # Calculate the number of validation samples
num_train_samples = len(text_pairs) - 2 * num_val_samples  # Calculate the number of trainir
train_pairs = text_pairs[:num_train_samples]  # Assign the first part of shuffled pairs to t
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]  # Assign the
test_pairs = text_pairs[num_train_samples + num_val_samples:]  # Assign the rest to testing


# Print sizes of each set
print("Total sentences:",len(text_pairs))
print("Training set size:",len(train_pairs))
print("Validation set size:",len(val_pairs))
print("Testing set size:",len(test_pairs))
```

```
Total sentences: 80684
Training set size: 56480
Validation set size: 12102
Testing set size: 12102
```

```
len(train_pairs)+len(val_pairs)+len(test_pairs)
```

```
80684
```

Removing Punctuations

```
strip_chars = string.punctuation + "¿"  # Define a string containing punctuation marks and '
strip_chars = strip_chars.replace("[", "")  # Remove "[" character from strip_chars
strip_chars = strip_chars.replace("]", "")  # Remove "]" character from strip_chars

# Print regex pattern for stripping punctuation marks
f"[{re.escape(strip_chars)}]"
```

```
'[!"\\#\\$%\\&\'\\(\\)\\*\\+,\\-\\./:;<=>\\?@\\\\\\^_`\\{\\|\\}\\~¿]'
```

Vectorizing the English and Sinhala Test Pairs

```
def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)  # Convert input string to lowercase
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", "")  # Remove punctuation marks from the s

vocab_size = 15000  # Define the vocabulary size
sequence_length = 20  # Define the sequence length

# Initialize TextVectorization layers for source (English) and target (Sinhala) texts
source_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
target_vectorization = layers.TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length + 1,
    standardize=custom_standardization,
)

# Extract English and Sinhala texts from train_pairs
train_english_texts = [pair[0] for pair in train_pairs]
train_sinhala_texts = [pair[1] for pair in train_pairs]

# Adapt the source vectorization layer to the English texts
source_vectorization.adapt(train_english_texts)
# Adapt the target vectorization layer to the Sinhala texts.
target_vectorization.adapt(train_sinhala_texts)
```

Preparing Datasets for the Translation Task

```
batch_size = 64  # Define batch size

def format_dataset(eng, sin):
    eng = source_vectorization(eng)  # Vectorize English texts
    sin = target_vectorization(sin)  # Vectorize Sinhala texts
    max_length = tf.maximum(tf.shape(eng)[1], tf.shape(sin)[1])  # Get the maximum sequence
    eng = tf.pad(eng, [[0, 0], [0, max_length - tf.shape(eng)[1]]])[:, :max_length]  # Pad c
    sin = tf.pad(sin, [[0, 0], [0, max_length - tf.shape(sin)[1]]])[:, :max_length]  # Pad c
    return ({
        "english": eng[:, :-1],
        "sinhala": sin[:, :-1],
    }, sin[:, 1:])  # Return formatted dataset with English and Sinhala inputs, and shifted

def make_dataset(pairs):
    eng_texts, sin_texts = zip(*pairs)  # Unzip English-Sinhala pairs
    eng_texts = list(eng_texts)  # Convert to list
    sin_texts = list(sin_texts)  # Convert to list
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, sin_texts))  # Create dataset f
    dataset = dataset.batch(batch_size)  # Batch the dataset
    dataset = dataset.map(format_dataset, num_parallel_calls=4)  # Map format_dataset functi
    return dataset.shuffle(2048).prefetch(16).cache()  # Shuffle, prefetch, and cache the da

# Create training and validation datasets
train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)

# Print shapes of inputs and targets from the first batch of training dataset
for inputs, targets in train_ds.take(1):
    print(f"inputs['english'].shape: {inputs['english'].shape}")
    print(f"inputs['sinhala'].shape: {inputs['sinhala'].shape}")
    print(f"targets.shape: {targets.shape}")

# Print a sample from the training dataset
print(list(train_ds.as_numpy_iterator())[50])
```

```
inputs['english'].shape: (64, 20)
inputs['sinhala'].shape: (64, 20)
targets.shape: (64, 20)
({'english': array([[  73,    2,    9, ...,    0,    0,    0],
       [  10,  146,  276, ...,    0,    0,    0],
       [ 108,   22,  108, ...,    0,    0,    0],
       ...,
       [2879,    0,    0, ...,    0,    0,    0],
       [3563, 7704,    0, ...,    0,    0,    0],
       [   7, 6249,   50, ...,    0,    0,    0]]), 'sinhala': array([[    2,    49,
       [    2,   144,   267, ...,     0,     0,     0],
       [    2,    73,    73, ...,     0,     0,     0],
       ...,
       [    2, 13411,     3, ...,     0,     0,     0],
       [    2, 11735, 14875, ...,     0,     0,     0],
       [    2,   106,   733, ...,     0,     0,     0]])}, array([[   49,     5,   134,
       [  144,   267,     6, ...,     0,     0,     0],
```

```
       [   73,    73,    73, ...,     0,     0,     0],
       ...,
       [13411,     3,     0, ...,     0,     0,     0],
       [11735, 14875,     3, ...,     0,     0,     0],
       [  106,   733,    59, ...,     0,     0,     0]]))
```

Transformer Model

Transformer Encoder Implemented as a Subclassed Layer

```
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        # Multi-head self-attention mechanism
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        # Feed-forward neural network layers
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        # Layer normalization for the two sub-layers
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None):
        if mask is not None:
            mask = mask[:, tf.newaxis, :]
        # Self-attention mechanism
        attention_output = self.attention(
            inputs, inputs, attention_mask=mask)
        # Add and normalize the self-attention output with the input
        proj_input = self.layernorm_1(inputs + attention_output)
        # Feed-forward network processing
        proj_output = self.dense_proj(proj_input)
        # Add and normalize the output with the input and self-attention output
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

The Transformer Decorder

```
class TransformerDecoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        # Multi-head self-attention mechanism for decoder input
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        # Multi-head attention mechanism for encoder-decoder attention
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
        # Feed-forward neural network layers
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        # Layer normalization for the three sub-layers
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
        self.supports_masking = True

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config

    def get_causal_attention_mask(self, inputs):
        # Create a causal attention mask to prevent attending to future tokens
        input_shape = tf.shape(inputs)
        batch_size, sequence_length = input_shape[0], input_shape[1]
        i = tf.range(sequence_length)[:, tf.newaxis]
        j = tf.range(sequence_length)
        mask = tf.cast(i >= j, dtype="int32")
        mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
        mult = tf.concat(
            [tf.expand_dims(batch_size, -1),
             tf.constant([1, 1], dtype=tf.int32)], axis=0)
        return tf.tile(mask, mult)

    def call(self, inputs, encoder_outputs, mask=None):
        # Create a causal mask for the decoder input
        causal_mask = self.get_causal_attention_mask(inputs)
        if mask is not None:
            # Combine the input mask with the causal mask
            padding_mask = tf.cast(
```

```
            mask[:, tf.newaxis, :], dtype="int32")
        padding_mask = tf.minimum(padding_mask, causal_mask)
    else:
        padding_mask = mask
    # Self-attention mechanism for decoder input
    attention_output_1 = self.attention_1(
        query=inputs,
        value=inputs,
        key=inputs,
        attention_mask=causal_mask)
    # Add and normalize the self-attention output with the input
    attention_output_1 = self.layernorm_1(inputs + attention_output_1)
    # Attention mechanism for encoder-decoder attention
    attention_output_2 = self.attention_2(
        query=attention_output_1,
        value=encoder_outputs,
        key=encoder_outputs,
        attention_mask=padding_mask,
    )
    # Add and normalize the output with the input and attention output
    attention_output_2 = self.layernorm_2(
        attention_output_1 + attention_output_2)
    # Feed-forward network processing
    proj_output = self.dense_proj(attention_output_2)
    # Add and normalize the output with the attention output and projection output
    return self.layernorm_3(attention_output_2 + proj_output)
```

Positional Encoding

```
# Positional embedding layer for incorporating positional information into token embeddings
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)
        # Embedding layer for token embeddings
        self.token_embeddings = layers.Embedding(
            input_dim=input_dim, output_dim=output_dim)
        # Embedding layer for positional embeddings
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)
        self.sequence_length = sequence_length
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        # Generate token embeddings
        embedded_tokens = self.token_embeddings(inputs)
        # Generate positional embeddings
        embedded_positions = self.position_embeddings(positions)
        # Add positional embeddings to token embeddings
        return embedded_tokens + embedded_positions

    def compute_mask(self, inputs, mask=None):
        # Create a mask to indicate valid tokens
        return tf.math.not_equal(inputs, 0)

    def get_config(self):
        config = super(PositionalEmbedding, self).get_config()
        config.update({
            "output_dim": self.output_dim,
            "sequence_length": self.sequence_length,
            "input_dim": self.input_dim,
        })
        return config
```

End-to-End Transformer

```
# Define parameters for the Transformer model
embed_dim = 256  # Dimensionality of the token embeddings
dense_dim = 2048  # Dimensionality of the feed-forward layer in the transformer blocks
num_heads = 8  # Number of attention heads

# Define inputs for the encoder and decoder
encoder_inputs = keras.Input(shape=(None,), dtype="int64", name="english")  # Input sequence
decoder_inputs = keras.Input(shape=(None,), dtype="int64", name="sinhala")  # Input sequence

# Embedding and encoding for the encoder inputs
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(encoder_inputs)  # Add posit
encoder_outputs = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)  # Apply Transforme

# Embedding, decoding, and output generation for the decoder inputs
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(decoder_inputs)  # Add posit
x = TransformerDecoder(embed_dim, dense_dim, num_heads)(x, encoder_outputs)  # Apply Transfo
x = layers.Dropout(0.5)(x)  # Apply dropout regularization to prevent overfitting
decoder_outputs = layers.Dense(vocab_size, activation="softmax")(x)  # Generate output proba

# Define the end-to-end Transformer model
transformer = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)  # Combine enco

# Print model summary
transformer.summary()
```

```
Model: "model"
__________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==========================================================================================
 english (InputLayer)        [(None, None)]               0         []

 sinhala (InputLayer)        [(None, None)]               0         []

 positional_embedding (Posi  (None, None, 256)            3845120   ['english[0][0]']
 tionalEmbedding)

 positional_embedding_1 (Po  (None, None, 256)            3845120   ['sinhala[0][0]']
 sitionalEmbedding)

 transformer_encoder (Trans  (None, None, 256)            3155456   ['positional_embeddi
 formerEncoder)

 transformer_decoder (Trans  (None, None, 256)            5259520   ['positional_embeddi
 formerDecoder)                                                     ',
                                                                     'transformer_encode

 dropout (Dropout)           (None, None, 256)            0         ['transformer_decode

 dense_4 (Dense)             (None, None, 15000)          3855000   ['dropout[0][0]']

==========================================================================================
Total params: 19960216 (76.14 MB)
Trainable params: 19960216 (76.14 MB)
```

```
Non-trainable params: 0 (0.00 Byte)
```

Training the Sequence-to-Sequence Transformer

```
# Compile the Transformer model with RMSprop optimizer and sparse categorical crossentropy l
transformer.compile(
    optimizer="rmsprop",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"])

# Train the Transformer model on the training dataset for 30 epochs with validation on the v
transformer.fit(train_ds, epochs=30, validation_data=val_ds)
```

```
Epoch 1/30
883/883 [==============================] - 74s 74ms/step - loss: 4.7054 - accuracy: 0
Epoch 2/30
883/883 [==============================] - 60s 67ms/step - loss: 3.9322 - accuracy: 0
Epoch 3/30
883/883 [==============================] - 60s 68ms/step - loss: 3.6283 - accuracy: 0
Epoch 4/30
883/883 [==============================] - 60s 68ms/step - loss: 3.4406 - accuracy: 0
Epoch 5/30
883/883 [==============================] - 60s 68ms/step - loss: 3.3055 - accuracy: 0
Epoch 6/30
883/883 [==============================] - 60s 68ms/step - loss: 3.2046 - accuracy: 0
Epoch 7/30
883/883 [==============================] - 61s 69ms/step - loss: 3.1169 - accuracy: 0
Epoch 8/30
883/883 [==============================] - 60s 68ms/step - loss: 3.0486 - accuracy: 0
Epoch 9/30
883/883 [==============================] - 60s 68ms/step - loss: 2.9891 - accuracy: 0
Epoch 10/30
883/883 [==============================] - 60s 68ms/step - loss: 2.9356 - accuracy: 0
Epoch 11/30
883/883 [==============================] - 60s 68ms/step - loss: 2.8925 - accuracy: 0
Epoch 12/30
883/883 [==============================] - 65s 73ms/step - loss: 2.8482 - accuracy: 0
Epoch 13/30
883/883 [==============================] - 61s 69ms/step - loss: 2.8079 - accuracy: 0
Epoch 14/30
883/883 [==============================] - 61s 69ms/step - loss: 2.7715 - accuracy: 0
Epoch 15/30
883/883 [==============================] - 64s 73ms/step - loss: 2.7432 - accuracy: 0
Epoch 16/30
883/883 [==============================] - 60s 68ms/step - loss: 2.7129 - accuracy: 0
Epoch 17/30
883/883 [==============================] - 60s 68ms/step - loss: 2.6853 - accuracy: 0
Epoch 18/30
883/883 [==============================] - 60s 68ms/step - loss: 2.6640 - accuracy: 0
Epoch 19/30
```

```
883/883 [==============================] - 64s 73ms/step - loss: 2.6402 - accuracy: 0
Epoch 20/30
883/883 [==============================] - 64s 73ms/step - loss: 2.6193 - accuracy: 0
Epoch 21/30
883/883 [==============================] - 61s 69ms/step - loss: 2.5939 - accuracy: 0
Epoch 22/30
883/883 [==============================] - 60s 68ms/step - loss: 2.5732 - accuracy: 0
Epoch 23/30
883/883 [==============================] - 61s 69ms/step - loss: 2.5511 - accuracy: 0
Epoch 24/30
883/883 [==============================] - 60s 68ms/step - loss: 2.5334 - accuracy: 0
Epoch 25/30
883/883 [==============================] - 60s 68ms/step - loss: 2.5107 - accuracy: 0
Epoch 26/30
883/883 [==============================] - 61s 69ms/step - loss: 2.4905 - accuracy: 0
Epoch 27/30
883/883 [==============================] - 60s 68ms/step - loss: 2.4724 - accuracy: 0
Epoch 28/30
883/883 [==============================] - 60s 68ms/step - loss: 2.4546 - accuracy: 0
Epoch 29/30
```

Testing the Trained Model

Testing the Transformer on Test Dataset

```
import numpy as np

# Get the vocabulary and create a lookup dictionary for the target language
sin_vocab = target_vectorization.get_vocabulary()
sin_index_lookup = dict(zip(range(len(sin_vocab)), sin_vocab))

# Define the maximum length for decoding a sentence
max_decoded_sentence_length = 20

# Function to decode a sequence given an input sentence
def decode_sequence(input_sentence):
    # Vectorize the input sentence
    tokenized_input_sentence = source_vectorization([input_sentence])

    # Initialize the decoded sentence with a start token
    decoded_sentence = "[start]"

    # Loop over the maximum decoded sentence length
    for i in range(max_decoded_sentence_length):
        # Vectorize the current decoded sentence (excluding the last token)
        tokenized_target_sentence = target_vectorization([decoded_sentence])[:, :-1]

        # Get predictions from the transformer model
        predictions = transformer([tokenized_input_sentence, tokenized_target_sentence])

        # Sample the token index with the highest probability
        sampled_token_index = np.argmax(predictions[0, i, :])

        # Get the corresponding token from the lookup dictionary
        sampled_token = sin_index_lookup[sampled_token_index]

        # Append the sampled token to the decoded sentence
        decoded_sentence += " " + sampled_token

        # Check if the end token is reached, and break the loop if so
        if sampled_token == "[end]":
            break

    return decoded_sentence
```