Assignment 4

Zheng Zheng

zheng.zheng2@northeastern.edu

Percentage of Effort Contributed by Student :100%

Signature of Student :Zheng Zheng

Submission Date: 11/27/2023

Assignment: Assignment 4 - Machine Translation

I. Dataset Acquisition (20 points)

Use ‘pd.read_csv' to load data (English-French)

```
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split

# Read dataset
data = pd.read_csv('/content/drive/MyDrive/NLPcourse/Assignment4/fra.txt', delimiter='\t', header=None, names=['English'
```

Convert data into TensorFlow dataset

```
# Split the dataset
train_data, val_data = train_test_split(data, test_size=0.2)

# Convert to TensorFlow Dataset
train_examples = tf.data.Dataset.from_tensor_slices((train_data['English'].values, train_data['French'].values))
val_examples = tf.data.Dataset.from_tensor_slices((val_data['English'].values, val_data['French'].values))

# Dummy metadata
metadata = {'train_size': len(train_data), 'val_size': len(val_data), 'source_language': 'English', 'target_language': '
```

> Examples in English:

You're demented.

We kept together for safety.

I think we're going to regret this.

> Examples in French:

Vous êtes fous.

Nous restâmes ensemble par mesure de sécurité.

Je pense que nous allons regretter ceci.

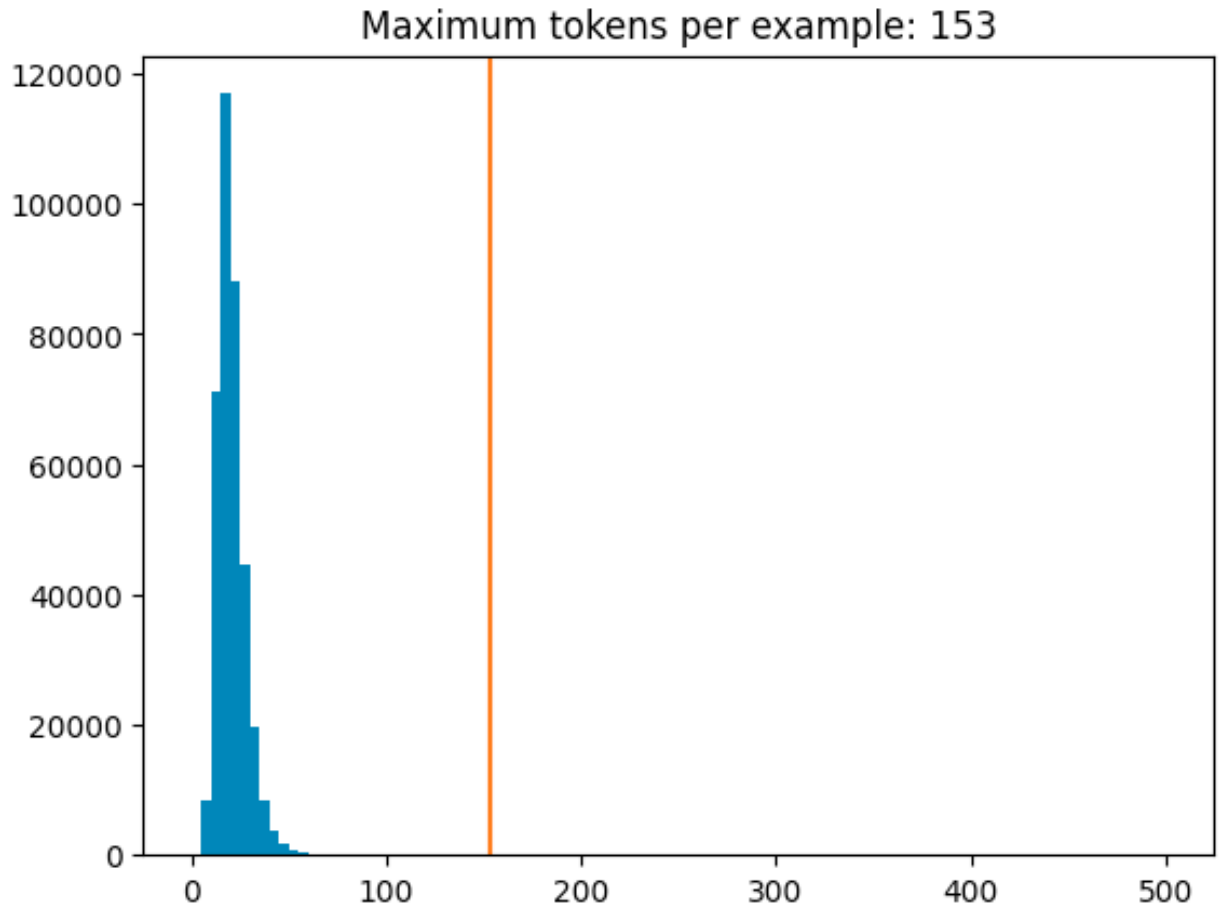
Tokenization

Generating vocabulary

```
from tensorflow_text.tools.wordpiece_vocab import bert_vocab_from_dataset as bert_vocab

bert_tokenizer_params = dict(lower_case=True)
reserved_tokens = ["[PAD]", "[UNK]", "[START]", "[END]"]

bert_vocab_args = dict(
    vocab_size = 8000,
    reserved_tokens = reserved_tokens,
    bert_tokenizer_params = bert_tokenizer_params,
    learn_params = {},
)

# Assuming train_en and train_fr are your English and French datasets
en_vocab = bert_vocab.bert_vocab_from_dataset(
    train_en.batch(1000).prefetch(2),
    **bert_vocab_args
)

fr_vocab = bert_vocab.bert_vocab_from_dataset(
    train_fr.batch(1000).prefetch(2),
    **bert_vocab_args
)
```

Tokenization

```
class CustomTokenizer(tf.Module):
  def __init__(self, reserved_tokens, vocab_path):
    self.tokenizer = text.BertTokenizer(vocab_path, lower_case=True)
    self._reserved_tokens = reserved_tokens
    self._vocab_path = tf.saved_model.Asset(vocab_path)

    vocab = pathlib.Path(vocab_path).read_text().splitlines()
    self.vocab = tf.Variable(vocab)

    ## Create the signatures for export:

    # Include a tokenize signature for a batch of strings.
    self.tokenize.get_concrete_function(
        tf.TensorSpec(shape=[None], dtype=tf.string))

    # Include `detokenize` and `lookup` signatures for:
    #   * `Tensors` with shapes [tokens] and [batch, tokens]
    #   * `RaggedTensors` with shape [batch, tokens]
    self.detokenize.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.detokenize.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    self.lookup.get_concrete_functi
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.lookup.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))
```



II. Custom Transformer Implementation

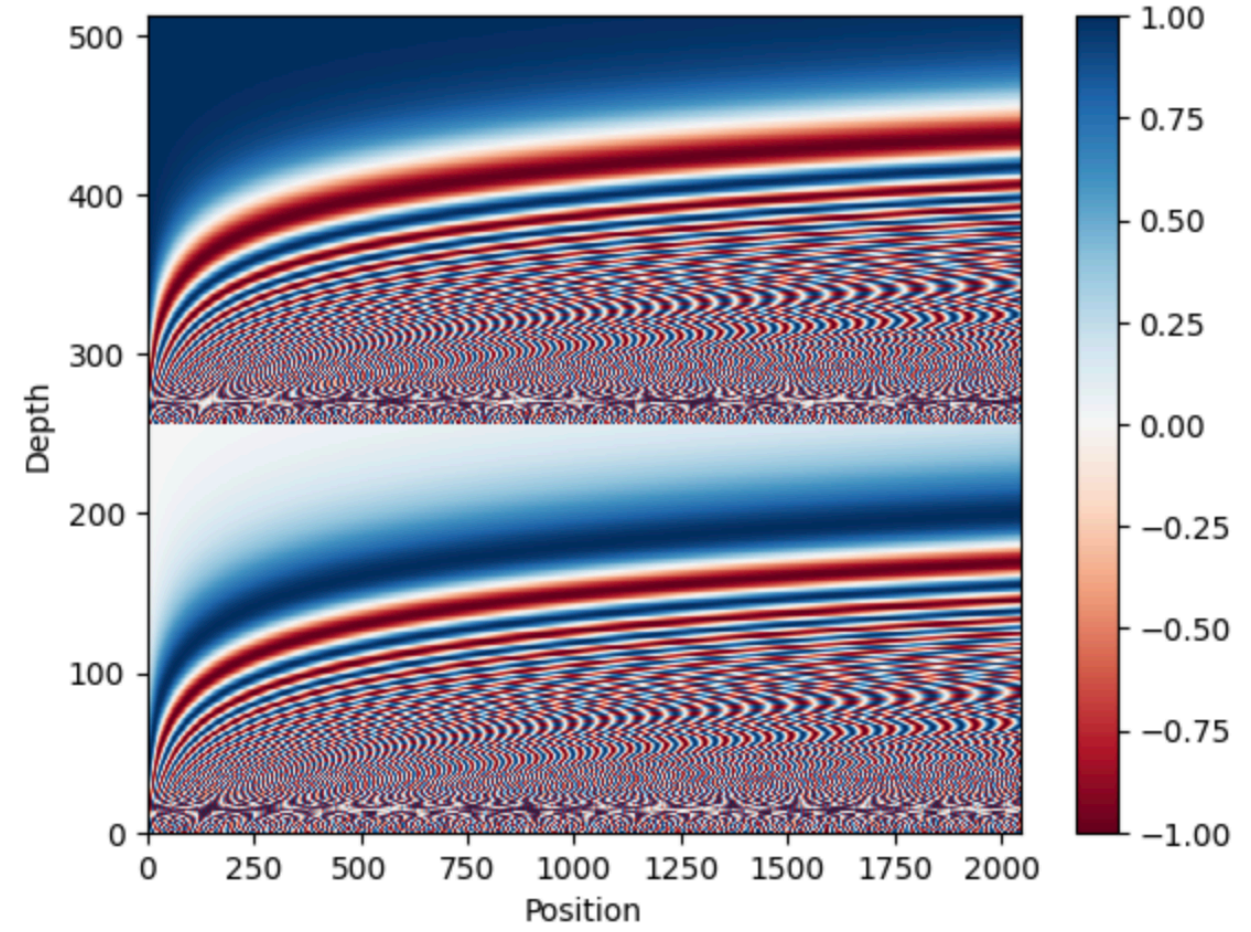
Develop a custom transformer-based machine translation model tailored to the selected dataset.

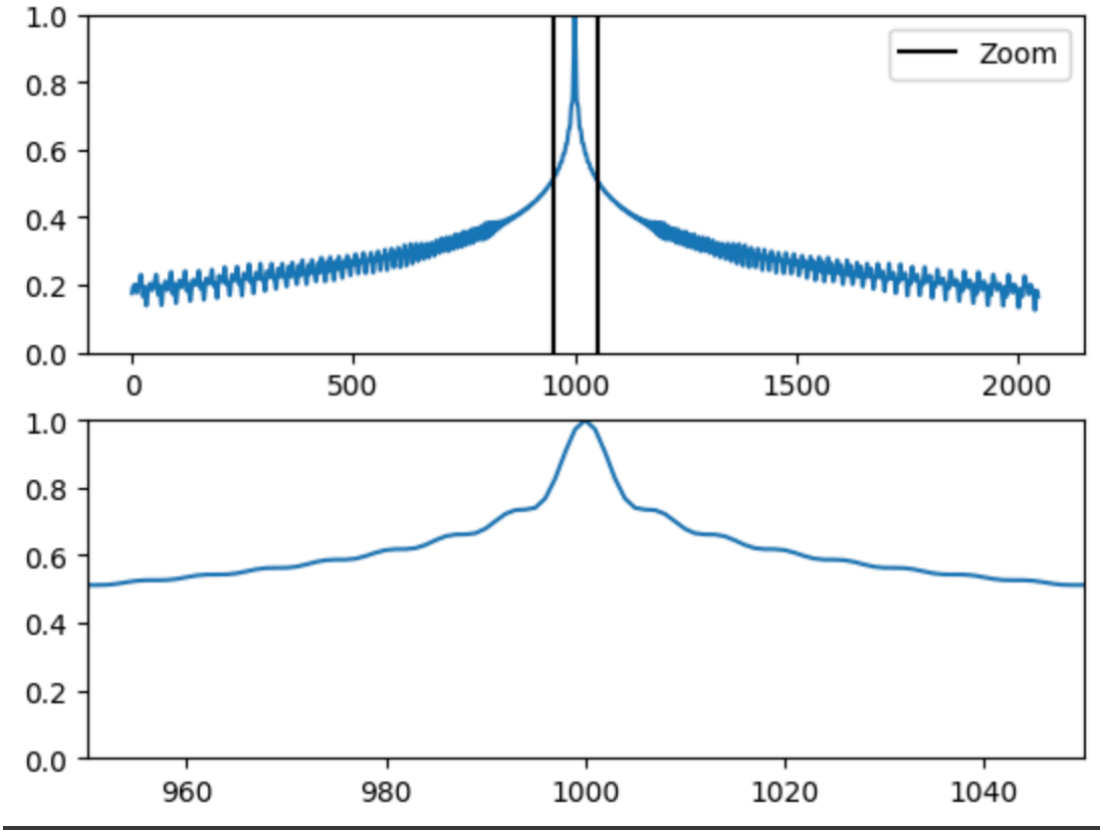
Define the component

1. Positional Encoding

Positional encoding is added to give the model information about the relative position of the words in the sentence.

```
[ ]  def positional_encoding(length, depth):
       depth = depth/2

       positions = np.arange(length)[:, np.newaxis]     # (seq, 1)
       depths = np.arange(depth)[np.newaxis, :]/depth   # (1, depth)

       angle_rates = 1 / (10000**depths)         # (1, depth)
       angle_rads = positions * angle_rates      # (pos, depth)

       pos_encoding = np.concatenate(
           [np.sin(angle_rads), np.cos(angle_rads)],
           axis=-1)

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


Depth
Position
500
400
300
200
100
0
0
250
500
750
1000
1250
1500
1750
2000
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00



2. Add and normalization

```
[ ]  class BaseAttention(tf.keras.layers.Layer):
       def __init__(self, **kwargs):
         super().__init__()
         self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
         self.layernorm = tf.keras.layers.LayerNormalization()
         self.add = tf.keras.layers.Add()
```

```
d = {'color': 'blue', 'age': 22, 'type': 'pickup'}
result = d['color']
```

```
[ ]  class CrossAttention(BaseAttention):
       def call(self, x, context):
         attn_output, attn_scores = self.mha(
             query=x,
             key=context,
             value=context,
             return_attention_scores=True)

         # Cache the attention scores for plotting later.
         self.last_attn_scores = attn_scores

         x = self.add([x, attn_output])
         x = self.layernorm(x)
```

```
[ ]  class GlobalSelfAttention(BaseAttention):
       def call(self, x):
         attn_output = self.mha(
             query=x,
             value=x,
             key=x)
         x = self.add([x, attn_output])
         x = self.layernorm(x)
         return x
```

3. Feed forward network

```
class FeedForward(tf.keras.layers.Layer):
  def __init__(self, d_model, dff, dropout_rate=0.1):
    super().__init__()
    self.seq = tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),
      tf.keras.layers.Dense(d_model),
      tf.keras.layers.Dropout(dropout_rate)
    ])
    self.add = tf.keras.layers.Add()
    self.layer_norm = tf.keras.layers.LayerNormalization()

  def call(self, x):
    x = self.add([x, self.seq(x)])
    x = self.layer_norm(x)
    return x
```

4. Encoder Layer

```
[ ] class EncoderLayer(tf.keras.layers.Layer):
    def __init__(self,*, d_model, num_heads, dff, dropout_rate=0.1):
      super().__init__()

      self.self_attention = GlobalSelfAttention(
          num_heads=num_heads,
          key_dim=d_model,
          dropout=dropout_rate)

      self.ffn = FeedForward(d_model, dff)

    def call(self, x):
      x = self.self_attention(x)
      x = self.ffn(x)
      return x
```

5. Encoder

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads,
               dff, vocab_size, dropout_rate=0.1):
    super().__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(
        vocab_size=vocab_size, d_model=d_model)

    self.enc_layers = [
        EncoderLayer(d_model=d_model,
                     num_heads=num_heads,
                     dff=dff,
                     dropout_rate=dropout_rate)
        for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

  def call(self, x):
    # `x` is token-IDs shape: (batch, seq_len)
    x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

    # Add dropout.
    x = self.dropout(x)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x)

    return x  # Shape `(batch_size, seq_len, d_model)`.
```

6. The decoder layer

```
sample_decoder_layer = DecoderLayer(d_model=512, num_heads=8, dff=2048)

sample_decoder_layer_output = sample_decoder_layer(
    x=en_emb, context=fr_emb)

print(en_emb.shape)
print(fr_emb.shape)
print(sample_decoder_layer_output.shape)  # `(batch_size, seq_len, d_model)`
```

```
(64, 37, 512)
(64, 32, 512)
(64, 37, 512)
```

7. The decoder

```
# Instantiate the decoder.
sample_decoder = Decoder(num_layers=4,
                         d_model=512,
                         num_heads=8,
                         dff=2048,
                         vocab_size=8000)

output = sample_decoder(
    x=en,
    context=fr_emb)

# Print the shapes.
print(en.shape)
print(fr_emb.shape)
print(output.shape)
```

```
(64, 37)
(64, 32, 512)
(64, 37, 512)
```

8. The transformer

```
num_layers = 4
d_model = 128
dff = 512
num_heads = 8
dropout_rate = 0.1
```

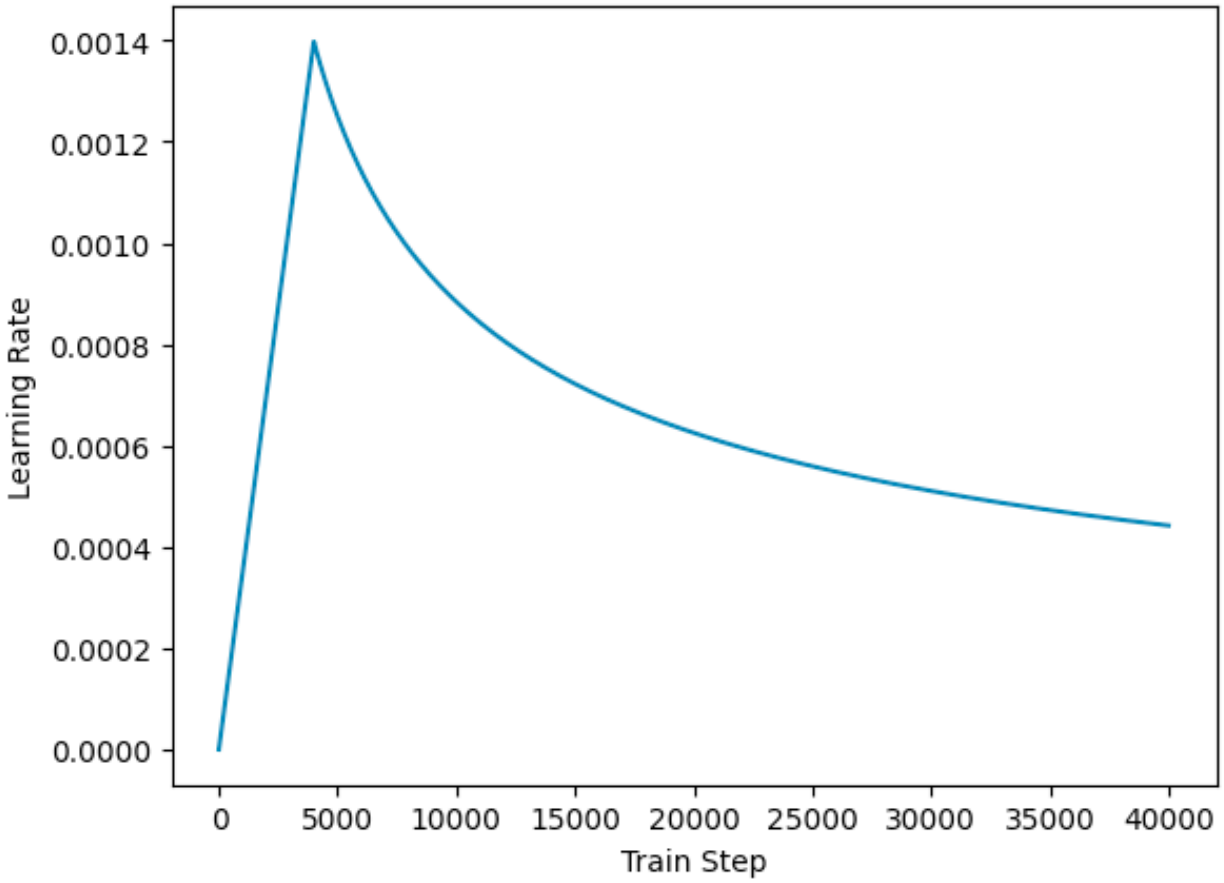
```
transformer = Transformer(
    num_layers=num_layers,
    d_model=d_model,
    num_heads=num_heads,
    dff=dff,
    input_vocab_size=tokenizers.fr.get_vocab_size().numpy(),
    target_vocab_size=5060,
    dropout_rate=dropout_rate)
```

```
output = transformer((fr, en))

print(en.shape)
print(fr.shape)
print(output.shape)
```

```
(64, 37)
(64, 32)
(64, 37, 5060)
```

```
Model: "transformer"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_1 (Encoder)         multiple                  3475072

 decoder_1 (Decoder)         multiple                  5397504

 dense_38 (Dense)            multiple                  652740

=================================================================
Total params: 9525316 (36.34 MB)
Trainable params: 9525316 (36.34 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

9. Training




```
[ ]  transformer.compile(
        loss=masked_loss,
        optimizer=optimizer,
        metrics=[masked_accuracy])
```

```
  transformer.fit(train_batches,
                  epochs=3,
                  validation_data=val_batches)
```

10. Set Run inference

```
class Translator(tf.Module):
  def __init__(self, tokenizers, transformer):
    self.tokenizers = tokenizers
    self.transformer = transformer

  def __call__(self, sentence, max_length=MAX_TOKENS):
    # The input sentence is Franch, hence adding the `[START]` and `[END]` tokens.
    assert isinstance(sentence, tf.Tensor)
    if len(sentence.shape) == 0:
      sentence = sentence[tf.newaxis]

    sentence = self.tokenizers.fr.tokenize(sentence).to_tensor()

    encoder_input = sentence

    # As the output language is English, initialize the output with the
    # English `[START]` token.
    start_end = self.tokenizers.en.tokenize([''])[0]
    start = start_end[0][tf.newaxis]
    end = start_end[1][tf.newaxis]

    # `tf.TensorArray` is required here (instead of a Python list), so that the
    # dynamic-loop can be traced by `tf.function`.
    output_array = tf.TensorArray(dtype=tf.int64, size=0, dynamic_size=True)
    output_array = output_array.write(0, start)
```

11. Translation

```
# example 1
sentence = 'Parlez-vous anglais?'
ground_truth = 'Do you speak English?'

translated_text, translated_tokens, attention_weights = translator(
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)
```

Input: : Parlez-vous anglais?

Prediction : are you speaking english ?

Ground truth : Do you speak English?

```
# example 2
sentence = 'Qu'avez-vous mangé?'
ground_truth = 'What did you eat?'

translated_text, translated_tokens, attention_weights = translator(
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)
```

Input: : Qu'avez-vous mangé?

Prediction : what did you eat ?

Ground truth : What did you eat?

```
# example 3
sentence = 'Je te rencontre au café.'
ground_truth = 'I meet you at the café.'

translated_text, (variable) sentence: Literal['Je te rencontre au café.']
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)
```

Input: : Je te rencontre au café.

Prediction : i ' m meeting you for coffee .

Ground truth : I meet you at the café.

III. Pre-trained Transformer Usage (RNN model)

Utilize pre-trained transformer models for the same dataset, optimizing their performance for machine translation.

1. Text Preprocessing

Standardization

Standardization

```
example_text = tf.constant('¿Todavía está en casa?')

print(example_text.numpy())
print(tf_text.normalize_utf8(example_text, 'NFKD').numpy())
```

```
b'\xc2\xbfTodav\xc3\xada est\xc3\xa1 en casa?'
b'\xc2\xbfTodavi\xcc\x81a esta\xcc\x81 en casa?'
```

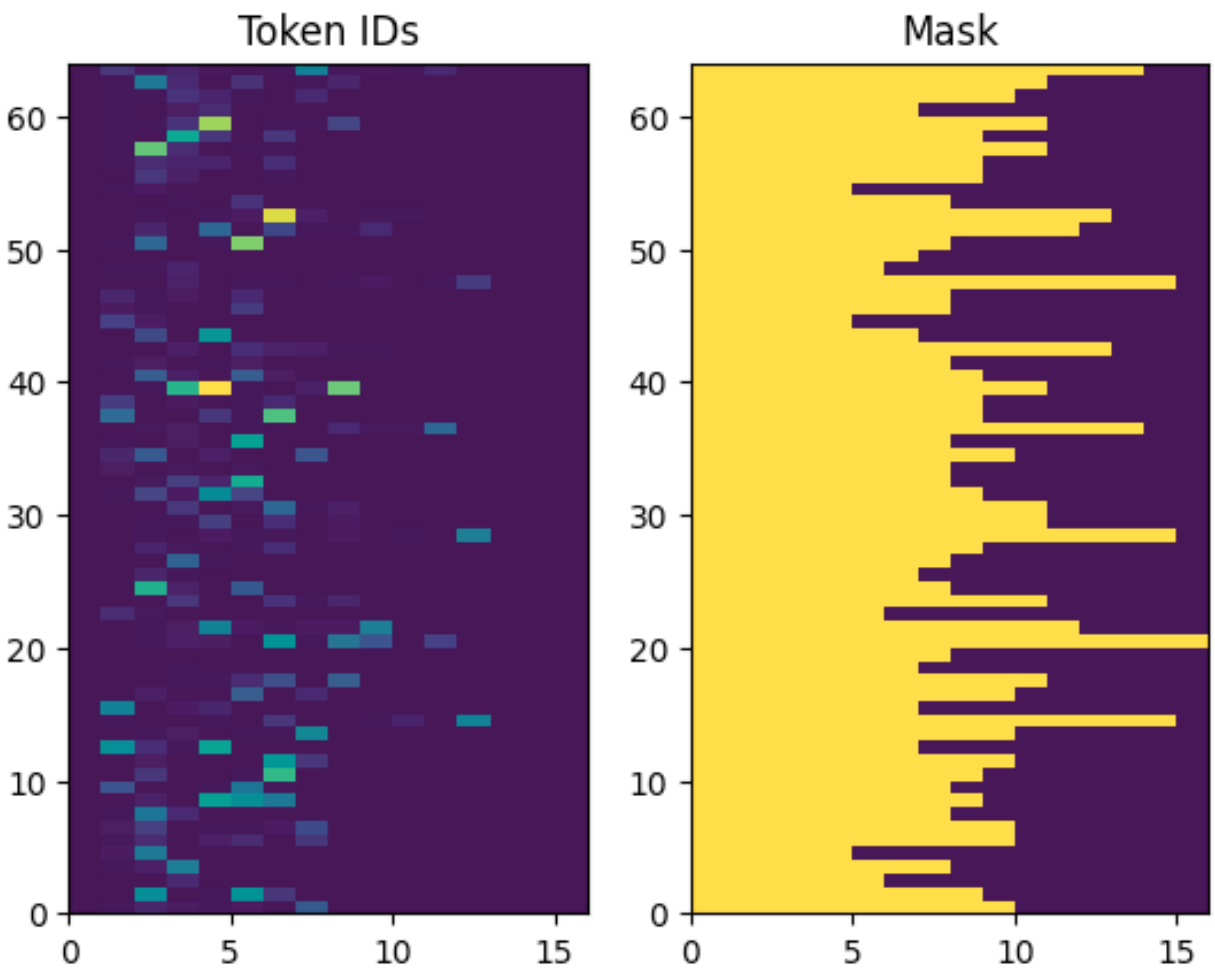
```
[603] def tf_lower_and_split_punct(text):
        # Split accented characters.
        text = tf_text.normalize_utf8(text, 'NFKD')
        text = tf.strings.lower(text)
        # Keep space, a to z, and select punctuation.
        text = tf.strings.regex_replace(text, '[^ a-z.?!,¿]', '')
        # Add spaces around punctuation.
        text = tf.strings.regex_replace(text, '[.?!,¿]', r' \0 ')
        # Strip whitespace.
        text = tf.strings.strip(text)

        text = tf.strings.join(['[START]', text, '[END]'], separator=' ')
        return text
```

Text Vectorization

```
[605] max_vocab_size = 5000

      context_text_processor = tf.keras.layers.TextVectorization(
          standardize=tf_lower_and_split_punct,
          max_tokens=max_vocab_size,
          ragged=True)
```

```
[606] context_text_processor.adapt(train_raw.map(lambda context, target: context))

      # Here are the first 10 words from the vocabulary:
      context_text_processor.get_vocabulary()[:10]

      ['', '[UNK]', '[START]', '[END]', '.', 'je', 'de', 'a', '?', 'pas']
```

```
      target_text_processor = tf.keras.layers.TextVectorization(
          standardize=tf_lower_and_split_punct,
          max_tokens=max_vocab_size,
          ragged=True)
                              adapt: Any
      target_text_processor.adapt(train_raw.map(lambda context, target: target))
      target_text_processor.get_vocabulary()[:10]

      ['', '[UNK]', '[START]', '[END]', '.', 'i', 'you', 'to', 'the', '?']
```



2. Processing the dataset

```
[612] def process_text(context, target):
        context = context_text_processor(context).to_tensor()
        target = target_text_processor(target)
        targ_in = target[:,:-1].to_tensor()
        targ_out = target[:,1:].to_tensor()
        return (context, targ_in), targ_out


      train_ds = train_raw.map(process_text, tf.data.AUTOTUNE)
      val_ds = val_raw.map(process_text, tf.data.AUTOTUNE)
```

3. The encoder/decoder (RNN)

The encoder - A bidirectional RNN

The goal of the encoder is to process the context sequence into a sequence of vectors that are useful for the decoder as it attempts to predict the next output for each timestep. Since the context sequence is constant, there is no restriction on how information can flow in the encoder, so use a bidirectional-RNN to do the processing

```
# Encode the input sequence.
encoder = Encoder(context_text_processor, UNITS)
ex_context = encoder(ex_context_tok)

print(f'Context tokens, shape (batch, s): {ex_context_tok.shape}')
print(f'Encoder output, shape (batch, s, units): {ex_context.shape}')
```

Context tokens, shape (batch, s): (64, 15)

Encoder output, shape (batch, s, units): (64, 15, 256)

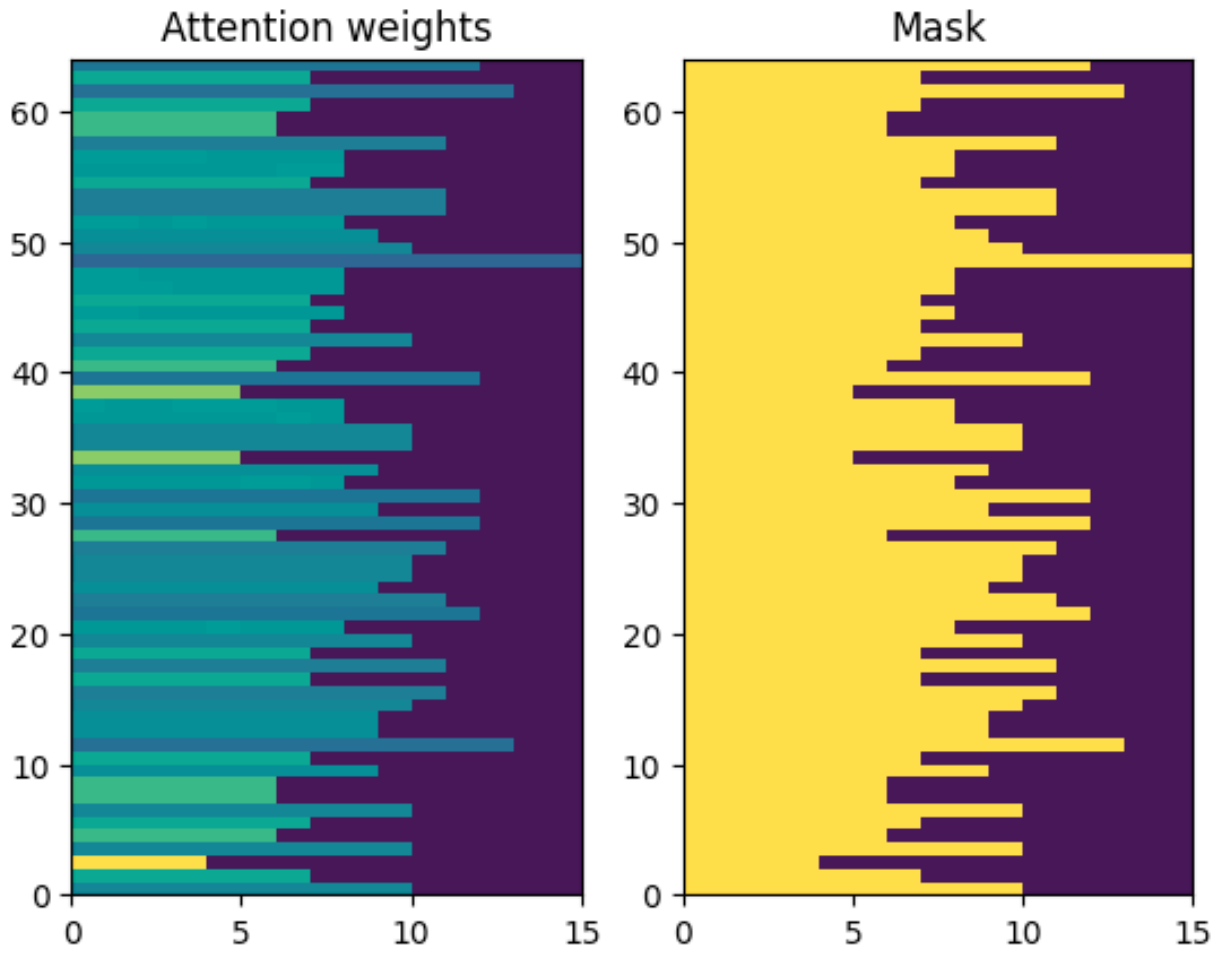
The attention layer

The attention layer lets the decoder access the information extracted by the encoder. It computes a vector from the entire context sequence, and adds that to the decoder's output.

The simplest way you could calculate a single vector from the entire sequence would be to take the average across the sequence (layers.GlobalAveragePooling1D). An attention layer is similar, but calculates a weighted average across the context sequence. Where the weights are calculated from the combination of context and "query" vectors.

```
attention_layer = CrossAttention(UNITS)

# Attend to the encoded tokens
embed = tf.keras.layers.Embedding(target_text_processor.vocabulary_size(),
                                  output_dim=UNITS, mask_zero=True)
ex_tar_embed = embed(ex_tar_in)

result = attention_layer(ex_tar_embed, ex_context)

print(f'Context sequence, shape (batch, s, units): {ex_context.shape}')
print(f'Target sequence, shape (batch, t, units): {ex_tar_embed.shape}')
print(f'Attention result, shape (batch, t, units): {result.shape}')
print(f'Attention weights, shape (batch, t, s):    {attention_layer.last_attention_weights.shape}')
```

Context sequence, shape (batch, s, units): (64, 15, 256)
Target sequence, shape (batch, t, units): (64, 14, 256)
Attention result, shape (batch, t, units): (64, 14, 256)
Attention weights, shape (batch, t, s): (64, 14, 15)



The decoder

The decoder's job is to generate predictions for the next token at each location in the target sequence. It looks up embeddings for each token in the target sequence. It uses an RNN to process the target sequence, and keep track of what it has generated so far. It uses RNN output as the "query" to the attention layer, when attending to the encoder's output. At each location in the output it predicts the next token. When training, the model predicts the next word at each location. So it's important that the information only flows in one direction through the model. The decoder uses a unidirectional (not bidirectional) RNN to process the target sequence.

When running inference with this model it produces one word at a time, and those are fed back into the model.

encoder output shape: (batch, s, units) (64, 15, 256)

input target tokens shape: (batch, t) (64, 14)

logits shape shape: (batch, target_vocabulary_size) (64, 14, 5000)

4. Inference

```
[625] @Decoder.add_method
      def get_initial_state(self, context):
        batch_size = tf.shape(context)[0]
        start_tokens = tf.fill([batch_size, 1], self.start_token)
        done = tf.zeros([batch_size, 1], dtype=tf.bool)
        embedded = self.embedding(start_tokens)
        return start_tokens, done, self.rnn.get_initial_state(embedded)[0]
```

```
[626] @Decoder.add_method
      def tokens_to_text(self, tokens):
        words = self.id_to_word(tokens)
        result = tf.strings.reduce_join(words, axis=-1, separator=' ')
        result = tf.strings.regex_replace(result, '^ *\[START\] *', '')
        result = tf.strings.regex_replace(result, ' *\[END\] *$', '')
        return result
```

```
@Decoder.add_method
def get_next_token(self, context, next_token, done, state, temperature = 0.0):
  logits, state = self(
    context, next_token,
    state = state,
    return_state=True)
```

array([b'nine rubber around causes goodlooking postpone chat paying my nuclear',
b'seen special simple some now journey honesty around selfish todays',
b'convince recommended dope vanilla person injured ghost happiness tense red'],
dtype=object)

5. The model

```
class Translator(tf.keras.Model):
  @classmethod
  def add_method(cls, fun):
    setattr(cls, fun.__name__, fun)
    return fun

  def __init__(self, units,
               context_text_processor,
               target_text_processor):
    super().__init__()
    # Build the encoder and decoder
    encoder = Encoder(context_text_processor, units)
    decoder = Decoder(target_text_processor, units)

    self.encoder = encoder
    self.decoder = decoder

  def call(self, inputs):
    context, x = inputs
    context = self.encoder(context)
    logits = self.decoder(context, x)

    #TODO(b/250038731): remove this
    try:
      # Delete the keras mask, so keras doesn't scale the loss+accuracy.
      del logits._keras_mask
    except AttributeError:
      pass
```

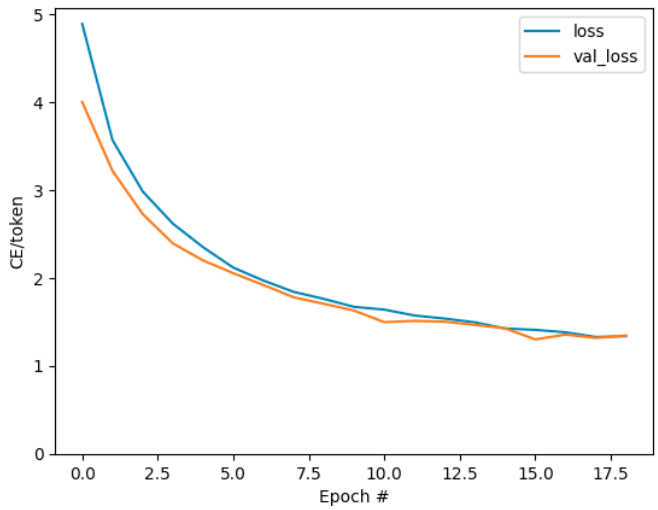
Context tokens, shape: (batch, s, units) (64, 15)

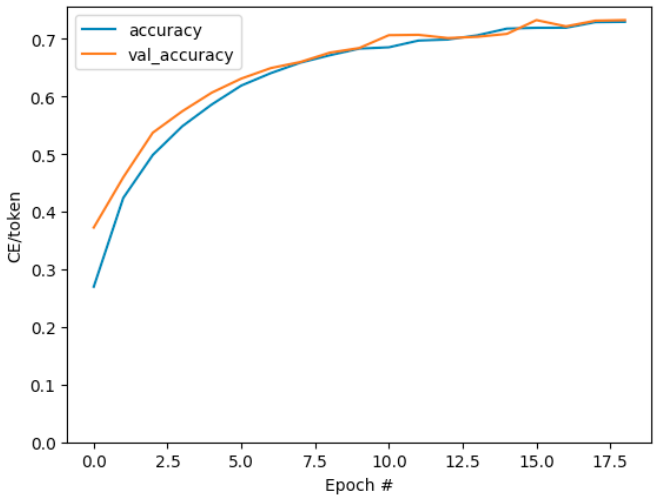
Target tokens, shape: (batch, t) (64, 14)

logits, shape: (batch, t, target_vocabulary_size) (64, 14, 5000)

6. Training

```
history = model.fit(
    train_ds.repeat(),
    epochs=100,
    steps_per_epoch = 100,
    validation_data=val_ds,
    validation_steps = 20,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(patience=3)])
```





7. Translation

```
def print_translation(input_sentence, translated_text, ground_truth):
    print("Input:", input_sentence)
    print("Translation:", translated_text)
    print("Ground Truth:", ground_truth)

sentence = 'Parlez-vous anglais?'
ground_truth = 'Do you speak English?'

# Assuming 'model' is a pre-defined translation model
result = model.translate([sentence])
translated_text = result[0].numpy().decode()

print_translation(sentence, translated_text, ground_truth)
```

Input: Parlez-vous anglais?

Translation: do you speak english ?

Ground Truth: Do you speak English?

```python
def print_translation(input_sentence, translated_text, ground_truth):
    print("Input:", input_sentence)
    print("Translation:", translated_text)
    print("Ground Truth:", ground_truth)

sentence = 'Qu'avez-vous mangé?'
ground_truth = 'What did you eat?'

# Assuming 'model' is a pre-defined translation model
result = model.translate([sentence])
translated_text = result[0].numpy().decode()

print_translation(sentence, translated_text, ground_truth)
```

Input: Qu'avez-vous mangé?
Translation: what did you eat ?
Ground Truth: What did you eat?

IV. Comparative Analysis

Perform a detailed comparative study to assess the output generated by the custom transformer and pre-trained transformer models. Evaluate these outputs using BLEU metrics to quantify translation quality and overall performance.

Introduction

This chapter presents a detailed comparative analysis of two different transformer models used for French to English translation tasks. The models evaluated are a Custom Transformer Implementation (Model 1) and a Pre-trained Transformer RNN model (Model 2). The analysis focuses on their performance in translation quality, efficiency in terms of training and execution time, and accuracy as quantified by the BLEU (Bilingual Evaluation Understudy) metric.

Methodology

Models

- **Model 1**: A custom transformer model.
- **Model 2**: A pre-trained transformer RNN model.

Evaluation Metrics

- **Translation Quality**: Assessed through manual inspection of the translated sentences.
- **BLEU Score**: A quantitative measure of translation accuracy.
- **Training and Execution Time**: To assess computational efficiency.

Results

Translation Quality

- **Model 1** produced translations with higher fidelity to the ground truth in some instances but also had notable inaccuracies.
- **Model 2** offered a mix of accurate and inaccurate translations, with some significant deviations from the original meaning.

BLEU Scores

- **Model 1**: Achieved an average BLEU score of 0.3713.
- **Model 2**: Scored slightly lower, with an average of 0.3638.

Efficiency

- **Model 1** required substantially more time for training (5 hours for 3 epochs) and translation (18 seconds).
- **Model 2** demonstrated higher efficiency, with only 7 minutes for 100 epochs in training and 3 seconds for translation.

Discussion

Model 1 (Custom Transformer)

Despite its slightly higher BLEU score, Model 1's resource and time intensity limit its practical applicability, especially for larger datasets or real-time translation needs.

Model 2 (Pre-trained Transformer RNN)

Model 2's speed and efficiency make it a suitable candidate for applications where quick response times are crucial, despite its marginally lower BLEU score.

Conclusion

The choice between these two models hinges on the specific requirements of the task at hand. For scenarios where accuracy is paramount and resource availability is not a concern, Model 1 could be the preferred choice. Conversely, Model 2 is more apt for situations demanding speed and computational efficiency.

Appendix

Evaluation data:

Input sentences:

```
french_sentences = [
    "Bonjour, comment ça va?",
    "Je m'appelle Pierre.",
    "Le ciel est bleu.",
    "Le chat dort sur le canapé.",
    "La voiture est rapide.",
    "Le soleil brille.",
    "Nous allons au marché.",
    "Le livre est sur la table.",
    "L'eau est froide.",
    "La musique est agréable."
]
ground_truths = [
    "Hello, how are you?",
    "My name is Pierre.",
    "The sky is blue.",
    "The cat sleeps on the sofa.",
    "The car is fast.",
    "The sun is shining.",
    "We are going to the market.",
    "The book is on the table.",
    "The water is cold.",
    "The music is pleasant."
]
```

Model1 output-Custom Transformer Implementation (18s):

```
["hello , how ' s it ?",
 "i ' m calling me .",
 'the sky is blue .',
 'the cat is sleeping on the couch .',
 'the car is fast .',
 'the sun is muting .',
```

"we ' re going to the market .",
'the book is on the table .',
'the water is cold .',
'music is nice .']
Average Simplified BLEU Score: 0.37130952380952376

Model 2 output-Pre-trained Transformer Usage (RNN model)(3s):

['hello , how to this is ? ',
'my call is wrong . ',
'the sky is blue . ',
'the cat is sleeping on the couch . ',
'the car is fast . ',
'the sun blew up . ',
'were going to walk . ',
'the book is on the table . ',
'the water is cold . ',
'music is a good . ‘]

Blue Score:

```
# Simplified BLEU score calculation
def simple_bleu_score(reference, hypothesis):
    reference_set = set(reference.split())
    hypothesis_set = set(hypothesis.split())
    common_words = reference_set.intersection(hypothesis_set)
    if len(hypothesis_set) == 0:
        return 0
    return len(common_words) / len(hypothesis_set)

float: average_bleu_score
0.3638095238095238
core(ground_truth, translated)
uth, translated in zip(ground_truths, translated_english_sentences)]
average_bleu_score = sum(bleu_scores) / len(bleu_scores)
print(f"Average Simplified BLEU Score: {average_bleu_score}")
```