MultiHeadClassificationModel-date-classfication-by-transformer

September 9, 2021

[244]:
```
from faker import Faker
from tensorflow.keras.layers import RepeatVector, Concatenate, Dense, Dot,␣
 ↪Activation
from tensorflow.keras.layers import Input, Bidirectional, LSTM
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.models import load_model
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.callbacks import Callback
from tensorflow.keras.preprocessing.text import Tokenizer

import tensorflow as tf
import numpy as np
import random
import os
import pickle
import time
import re

from babel.dates import format_date
```

[245]:
```
files_to_save = {
    "tokenizer_human": "tokenizer_human",
    "MAX_SEQ_LENGTH_HUMAN": "MAX_SEQ_LENGTH_HUMAN"
}
```

[246]:
```
def retrieve_files(files_to_save, target_dir_path):
    stored = {}

    for name, _ in files_to_save.items():
        with open(f"{target_dir_path}/{name}.pickle", "rb") as f:
            stored[name] = pickle.load(f)

    return stored
```

[247]:

```
def save_files(files_to_save, target_dir_path):
    for name, value in files_to_save.items():
        file_path = f"{target_dir_path}/{name}.pickle"
        if os.path.exists(file_path) is not True:
            with open(f"{target_dir_path}/{name}.pickle", "wb") as f:
                pickle.dump(value, f, protocol=pickle.HIGHEST_PROTOCOL)

    return stored
```

[248]:

```
save_files(files_to_save, "./output_files")
```

[248]:

```
{'tokenizer_human': <keras_preprocessing.text.Tokenizer at 0x7feadbf435e0>,
 'MAX_SEQ_LENGTH_HUMAN': 29}
```

[249]:

```
stored = retrieve_files(files_to_save, "./output_files")
tokenizer_human = stored["tokenizer_human"]
MAX_SEQ_LENGTH_HUMAN = stored["MAX_SEQ_LENGTH_HUMAN"]
```

[250]:

```
MAX_SEQ_LENGTH_HUMAN
```

[250]:

```
29
```

[258]:

```
faker = Faker()
BATCH_SIZE=128
training_size = int(BATCH_SIZE*100*10*2)
BUFFER_SIZE = 20000
BATCH_SIZE = 64
```

[259]:

```
FORMATS = [
    'short',
    'medium',
    'medium',
    'medium',
    'long','long',
    'long','long',
    'long','full',
    'full','full',
    'd MMM YYY',
    'dd MMM YYY',
    'd MMMM YYY',
    'd MMMM YYY',
    'd MMMM YYY',
    'dd MMMM YYY',
    'dd MMMM YYY',
    'dd MMMM YYY',
    'dd MMMM YYY',
    'd MMMM YYY',
    'd MMMM YYY',
    'dd MMMM YYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'YYYY/MM/dd',
    'YYYY/MM/dd',
    'YYYY/MM/dd',
    'EE d, MMM YYY',
    'EE dd, MMM YYY',
    'EEEE d, MMMM YYY',
    'EEEE dd, MMMM YYY',
    'MMM d, YYY',
    'MMM dd, YYY',
    'MMMM d, YYY',
    'MMMM dd, YYY',
    'YYY, d MMM',
    'YYY, d MMMM',
    'YYY, dd MMMM',
    'YYY, dd MMMM',
    'EE YYY, d MMMM',
    'EE YYY, dd MMMM',
    'YYYY-MM-d',
     'YYYY-MM-dd',
     'YYYY-MM-dd',
     'YYYY-MM-dd'
]
```

[263]:

```
for format in FORMATS:
    dt = faker.date_time_between(start_date = '-50y',end_date='+50y')
    date = format_date(dt, format=format, locale='en')
    print(date)
```

```
11/28/07
Aug 26, 1986
Jul 21, 2059
May 12, 2015
June 28, 1987
May 19, 1985
February 26, 2043
April 7, 1998
```

```
July 9, 2018
Thursday, July 4, 2002
Monday, May 18, 2054
Monday, January 19, 2071
8 Sep 1982
23 Aug 2004
5 November 2044
22 June 2068
16 November 1975
10 August 1999
27 June 1975
16 April 1979
10 April 2026
6 April 2007
7 September 1992
08 March 1993
7/09/1979
5/12/1982
26/06/2035
2/02/2037
22/02/2021
14/02/2061
22/03/2004
02/08/1996
06/02/1995
16/02/2067
1986/06/01
1979/02/12
1992/12/04
Thu 10, May 1973
Mon 13, Mar 2051
Sunday 18, June 2006
Friday 14, September 2001
Oct 28, 1994
May 28, 2058
December 16, 2052
August 06, 1998
2029, 13 Sep
1988, 17 January
2068, 05 June
2007, 27 May
Wed 2059, 17 September
Thu 2012, 19 July
2062-05-1
2064-02-16
2045-04-26
2013-05-01
```

[264]:

```
def preprocess_date(date):
    return date.lower().replace(',', '#')
```

[265]:

```
def random_date():
    dt = faker.date_time_between(start_date = '-50y',end_date='+50y')
    try:
        date = format_date(dt, format=random.choice(FORMATS), locale='en')
        human_readable = preprocess_date(date)
        machine_readable = format_date(dt, format="YYYY-MM-dd", locale='en')
    except AttributeError as e:
        return None, None, None
    return human_readable, machine_readable
```

[266]:

```
# MAX_SEQ_LENGTH_HUMAN
```

[267]:

```
random_date()
```

[267]:

```
('jul 01# 1976', '1976-07-01')
```

[268]:

```
dataset_human = []
dataset_machine = []

for i in range(training_size):
    human_data, machine_data = random_date()
    dataset_human.append("<" + human_data+ ">")
    # use # to replace , as it cannot be tokenized
    # Use < as start token and > as end token
    dataset_machine.append("<" + machine_data + ">")
```

[269]:

```
# MAX_SEQ_LENGTH_HUMAN = max([len(data) for data in dataset_human])
MAX_SEQ_LENGTH_MACHINE = max([len(data) for data in dataset_machine])
```

[270]:

```
dataset_human = [re.sub("\<|\>", "", data)  for data in dataset_human]
```

[271]:

```
encoder_input = pad_sequences(
    tokenizer_human.texts_to_sequences(dataset_human),
    maxlen=MAX_SEQ_LENGTH_HUMAN,
    padding="post"
)
print(encoder_input.shape)
```

```
(256000, 29)
```

[272]:

```
decoder_input_ = pad_sequences(
    tokenizer_human.texts_to_sequences(dataset_machine),
    maxlen=MAX_SEQ_LENGTH_MACHINE+1,
    padding="post"
)
```

```
print(decoder_input_.shape)
```

```
(256000, 13)
```

[273]:

```
sos_index = tokenizer_human.word_index["<"]
eos_index = tokenizer_human.word_index[">"]
```

[274]:

```
decoder_input = decoder_input_[:, :-1]
decoder_input_real = decoder_input_[:, 1:-1]
```

[275]:

```
decoder_input_filter_eos_mask = decoder_input!=eos_index
decoder_input_filter_eos_mask = decoder_input_filter_eos_mask.astype("int")
decoder_input = decoder_input * decoder_input_filter_eos_mask
decoder_input = decoder_input[:,:-1]
```

[276]:

```
decoder_input.shape
```

[276]:

```
(256000, 11)
```

[277]:

```
def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
    return pos * angle_rates
```

[278]:

```
def positional_encoding(position, d_model):
    angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                            np.arange(d_model)[np.newaxis, :],
                            d_model)

    # apply sin to even indices in the array; 2i
    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

    # apply cos to odd indices in the array; 2i+1
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])

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

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

[279]:

```
def create_padding_mask(seq):
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

    # add extra dimensions to add the padding
    # to the attention logits.
    return seq[:, tf.newaxis, tf.newaxis, :]  # (batch_size, 1, 1, seq_len)
```

[280]:

```
def create_look_ahead_mask(size):
    mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
    return mask  # (seq_len, seq_len)
```

[281]:

```
def scaled_dot_product_attention(q, k, v, mask):
    """Calculate the attention weights.
    q, k, v must have matching leading dimensions.
    k, v must have matching penultimate dimension, i.e.: seq_len_k = seq_len_v.
    The mask has different shapes depending on its type(padding or look ahead)
    but it must be broadcastable for addition.

    Args:
    q: query shape == (..., seq_len_q, depth)
    k: key shape == (..., seq_len_k, depth)
    v: value shape == (..., seq_len_v, depth_v)
    mask: Float tensor with shape broadcastable
          to (..., seq_len_q, seq_len_k). Defaults to None.

    Returns:
    output, attention_weights
    """

    matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

    # scale matmul_qk
    dk = tf.cast(tf.shape(k)[-1], tf.float32)
    scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

    # add the mask to the scaled tensor.
    if mask is not None:
        scaled_attention_logits += (mask * -1e9)

    # softmax is normalized on the last axis (seq_len_k) so that the scores
    # add up to 1.
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (...
 ↪, seq_len_q, seq_len_k)

    output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)

    return output, attention_weights
```

[282]:

```
class MultiHeadAttention(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model

        assert d_model % self.num_heads == 0

        self.depth = d_model // self.num_heads
        self.wq = tf.keras.layers.Dense(d_model)
        self.wk = tf.keras.layers.Dense(d_model)
        self.wv = tf.keras.layers.Dense(d_model)

        self.dense = tf.keras.layers.Dense(d_model)

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


    def split_heads(self, x, batch_size):
        """Split the last dimension into (num_heads, depth).
        Transpose the result such that the shape is (batch_size, num_heads,␣
↪seq_len, depth)
        """
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, q, k, v, mask):
        batch_size = tf.shape(q)[0]

        q = self.wq(q)  # (batch_size, seq_len, d_model)
        k = self.wk(k)  # (batch_size, seq_len, d_model)
        v = self.wv(v)  # (batch_size, seq_len, d_model)

        q = self.split_heads(q, batch_size)  # (batch_size, num_heads,␣
↪seq_len_q, depth)
        k = self.split_heads(k, batch_size)  # (batch_size, num_heads,␣
↪seq_len_k, depth)
        v = self.split_heads(v, batch_size)  # (batch_size, num_heads,␣
↪seq_len_v, depth)

        # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
        # attention_weights.shape == (batch_size, num_heads, seq_len_q,␣
↪seq_len_k)
        scaled_attention, attention_weights = scaled_dot_product_attention(
            q, k, v, mask)

        scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  #␣
↪(batch_size, seq_len_q, num_heads, depth)

        concat_attention = tf.reshape(scaled_attention,
                                      (batch_size, -1, self.d_model))  #␣
 ↪(batch_size, seq_len_q, d_model)

        output = self.dense(concat_attention)  # (batch_size, seq_len_q,␣
 ↪d_model)

        return output, attention_weights
```

[283]:

```
def point_wise_feed_forward_network(d_model, dff):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len,␣
 ↪dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])
```

[284]:

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

        self.mha = MultiHeadAttention(d_model, num_heads)
        self.ffn = point_wise_feed_forward_network(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)

    def get_config(self):
        config = super(EncoderLayer, self).get_config()
        config.update({"d_model": self.d_model,
                       "num_heads": self.num_heads,
                       "dff": self.dff,
                       "rate": self.rate
                       })
        return config


    def call(self, x, training, mask):

        attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len,␣
 ↪d_model)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len,␣
↪d_model)

        ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
        ffn_output = self.dropout2(ffn_output, training=training)
        out2 = self.layernorm2(out1 + ffn_output)  # (batch_size,␣
↪input_seq_len, d_model)

        return out2
```

[285]:

```
class DecoderLayer(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, dff, rate=0.1):
        super(DecoderLayer, self).__init__()
        self.d_model = d_model
        self.num_heads = num_heads
        self.dff = dff
        self.rate = rate

        self.mha1 = MultiHeadAttention(d_model, num_heads)
        self.mha2 = MultiHeadAttention(d_model, num_heads)

        self.ffn = point_wise_feed_forward_network(d_model, dff)

        self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = tf.keras.layers.Dropout(rate)
        self.dropout2 = tf.keras.layers.Dropout(rate)
        self.dropout3 = tf.keras.layers.Dropout(rate)

    def get_config(self):
        config = super(DecoderLayer, self).get_config()
        config.update({"d_model": self.d_model,
                       "num_heads": self.num_heads,
                       "dff": self.dff,
                       "rate": self.rate
                       })
        return config

    def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):
    # enc_output.shape == (batch_size, input_seq_len, d_model)

        attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  #␣
↪(batch_size, target_seq_len, d_model)
        attn1 = self.dropout1(attn1, training=training)
        out1 = self.layernorm1(attn1 + x)

        attn2, attn_weights_block2 = self.mha2(
            out1, enc_output, enc_output, padding_mask)  # (batch_size,␣
↪target_seq_len, d_model)
        attn2 = self.dropout2(attn2, training=training)
        out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len,␣
↪d_model)

        ffn_output = self.ffn(out2)  # (batch_size, target_seq_len, d_model)
        ffn_output = self.dropout3(ffn_output, training=training)
        out3 = self.layernorm3(ffn_output + out2)  # (batch_size,␣
↪target_seq_len, d_model)

        return out3, attn_weights_block1, attn_weights_block2
```

[286]:

```
def get_embedding_matrix(vocab_size):
    embedding_matrix = np.zeros((vocab_size, vocab_size))

    for index in range(vocab_size):
        one_hot = np.zeros((vocab_size,))
        one_hot[index] = 1
        embedding_matrix[index] = one_hot

    return embedding_matrix
```

[287]:

```
class Encoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               maximum_position_encoding, rate=0.1):
        super(Encoder, self).__init__()
        self.num_layers = num_layers
        self.d_model = d_model
        self.num_heads = num_heads
        self.dff = dff
        self.input_vocab_size = input_vocab_size
        self.maximum_position_encoding = maximum_position_encoding,
        self.rate = rate


        self.embedding_matrix = get_embedding_matrix(input_vocab_size)

        enc_input_identity = tf.identity(input_vocab_size)
        self.embedding = layers.Embedding(input_vocab_size,
                                          input_vocab_size,
                                          weights=[self.embedding_matrix],
                                          trainable=False)
        self.pos_encoding = positional_encoding(maximum_position_encoding,
                                                self.d_model)

        self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
                           for _ in range(num_layers)]

        self.dropout = tf.keras.layers.Dropout(rate)

    def get_config(self):
        config = super(Encoder, self).get_config()
        config.update({"num_layers": self.num_layers,
                       "d_model": self.d_model,
                       "num_heads": self.num_heads,
                       "dff": self.dff,
                       "input_vocab_size": self.input_vocab_size,
                       "maximum_position_encoding": self.
↪maximum_position_encoding,
                       "rate": self.rate
                       })
        return config



    def call(self, x, training, mask):

        seq_len = tf.shape(x)[1]
        # adding embedding and position encoding.
        x = self.embedding(x)  # (batch_size, input_seq_len, d_model)
        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]

        x = self.dropout(x, training=training)

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

        return x  # (batch_size, input_seq_len, d_model)
```

[288]:

```
class Decoder(tf.keras.layers.Layer):
    def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
               maximum_position_encoding, rate=0.1):
        super(Decoder, self).__init__()

        self.num_layers = num_layers
        self.d_model = d_model
        self.num_heads = num_heads
        self.dff = dff
        self.target_vocab_size = target_vocab_size
        self.maximum_position_encoding = maximum_position_encoding,
        self.rate = rate

        enc_input_identity = tf.identity(target_vocab_size)

        self.embedding_matrix = get_embedding_matrix(target_vocab_size)

        self.embedding = layers.Embedding(target_vocab_size,
                                          target_vocab_size,
                                          weights=[self.embedding_matrix],
                                          trainable=False)

        self.pos_encoding = positional_encoding(maximum_position_encoding,␣
↪d_model)

        self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                           for _ in range(num_layers)]
        self.dropout = tf.keras.layers.Dropout(rate)

    def get_config(self):
        config = super(Decoder, self).get_config()
        config.update({"num_layers": self.num_layers,
                       "d_model": self.d_model,
                       "num_heads": self.num_heads,
                       "dff": self.dff,
                       "target_vocab_size": self.target_vocab_size,
                       "maximum_position_encoding": self.
↪maximum_position_encoding,
                       "rate": self.rate,
                       })
        return config


    def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):

        seq_len = tf.shape(x)[1]
        attention_weights = {}

        x = self.embedding(x)  # (batch_size, target_seq_len, d_model)
        x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
        x += self.pos_encoding[:, :seq_len, :]

        x = self.dropout(x, training=training)

        for i in range(self.num_layers):
          x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                                 look_ahead_mask, padding_mask)

          attention_weights[f'decoder_layer{i+1}_block1'] = block1
          attention_weights[f'decoder_layer{i+1}_block2'] = block2

        # x.shape == (batch_size, target_seq_len, d_model)
        return x, attention_weights
```

[289]:

```
class Transformer(tf.keras.Model):
    def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               target_vocab_size, pe_input, pe_target, rate=0.1):
        super(Transformer, self).__init__()
        self.num_layers = num_layers
        self.d_model = d_model
        self.num_heads = num_heads
        self.dff = dff
        self.input_vocab_size = input_vocab_size
        self.target_vocab_size = target_vocab_size
        self.pe_input = pe_input
        self.pe_target = pe_target
        self.rate = rate

        self.tokenizer = Encoder(num_layers, d_model, num_heads, dff,
                                 input_vocab_size, pe_input, rate)

        self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                               target_vocab_size, pe_target, rate)

        self.final_layer = tf.keras.layers.Dense(target_vocab_size)

    def get_config(self):
        config = {"num_layers": self.num_layers,
                  "d_model": self.d_model,
                  "num_heads": self.num_heads,
                  "dff": self.dff,
                  "input_vocab_size": self.input_vocab_size,
                  "target_vocab_size": self.target_vocab_size,
                  "pe_input": self.pe_input,
                  "pe_target": self.pe_target,
                  "rate": self.rate
                 }
        return config

    @classmethod
    def from_config(cls, config):
        return cls(**config)
    def call(self, inp, training, **kwargs):
        tar = kwargs.get('tar', None)
        enc_padding_mask = kwargs.get('enc_padding_mask', None)
        look_ahead_mask = kwargs.get('look_ahead_mask', None)
        dec_padding_mask = kwargs.get('dec_padding_mask', None)

        if tar is None or enc_padding_mask is None or look_ahead_mask is None␣
↪or dec_padding_mask is None:
            return (None, None)

        enc_output = self.tokenizer(inp, training, enc_padding_mask)  #␣
↪(batch_size, inp_seq_len, d_model)

        # dec_output.shape == (batch_size, tar_seq_len, d_model)
        dec_output, attention_weights = self.decoder(
            tar, enc_output, training, look_ahead_mask, dec_padding_mask)

        final_output = self.final_layer(dec_output)  # (batch_size,␣
↪tar_seq_len, target_vocab_size)

        return final_output, attention_weights
```

[290]:
```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000):
        super(CustomSchedule, self).__init__()

        self.d_model = d_model
        self.d_model = tf.cast(self.d_model, tf.float32)

        self.warmup_steps = warmup_steps

    def __call__(self, step):
        arg1 = tf.math.rsqrt(step)
        arg2 = step * (self.warmup_steps ** -1.5)

        return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)
```

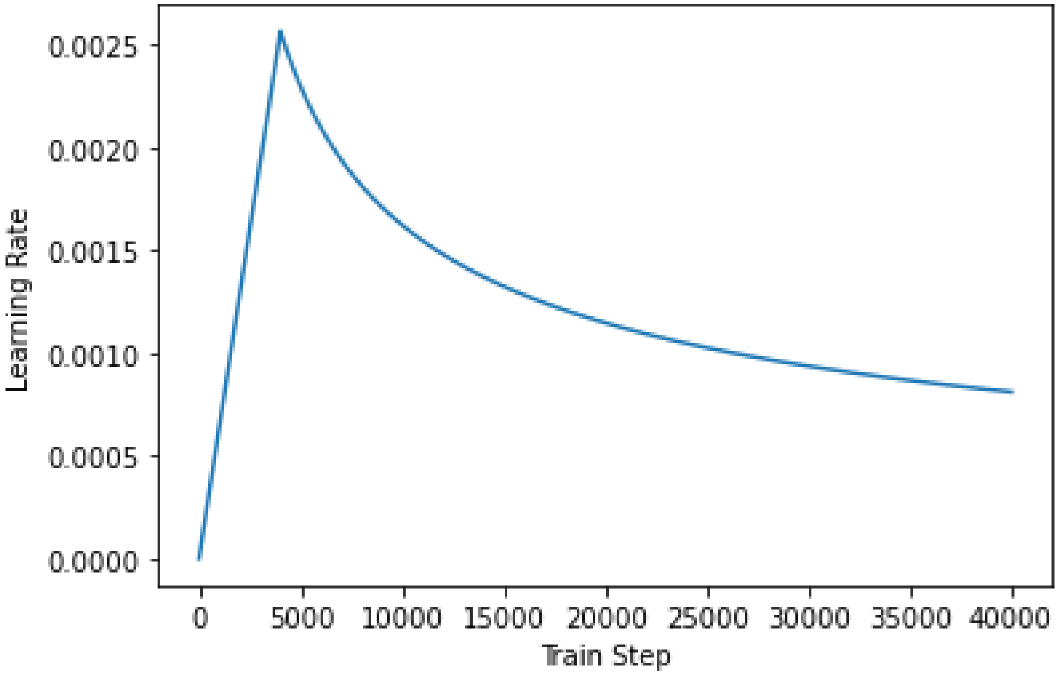
[291]:
```
d_model = 38
```

[292]:
```
learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98,
                                     epsilon=1e-9)
```

[293]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

```
temp_learning_rate_schedule = CustomSchedule(d_model)

plt.plot(temp_learning_rate_schedule(tf.range(40000, dtype=tf.float32)))
plt.ylabel("Learning Rate")
plt.xlabel("Train Step")
```

[293]:
```
Text(0.5, 0, 'Train Step')
```



[294]:
```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
```

[295]:
```
def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_sum(loss_)/tf.reduce_sum(mask)


def accuracy_function(real, pred):
    accuracies = tf.equal(real, tf.cast(tf.argmax(pred, axis=2), dtype=tf.
 ↪int32))
```

```
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    accuracies = tf.math.logical_and(mask, accuracies)

    accuracies = tf.cast(accuracies, dtype=tf.float32)
    mask = tf.cast(mask, dtype=tf.float32)
    return tf.reduce_sum(accuracies)/tf.reduce_sum(mask)
```

[296]:

```
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.Mean(name='train_accuracy')
```

[297]:

```
transformer = Transformer(num_layers=2,
                          d_model=d_model,
                          num_heads=2,
                          dff=1024,
                          input_vocab_size=38,
                          target_vocab_size=38,
                          pe_input=1000,
                          pe_target=1000,
                          rate=0.1)
```

[ ]:

```
Transformer.get_config()
```

[300]:

```
def create_masks(inp, tar):
    # Encoder padding mask

    enc_padding_mask = create_padding_mask(inp)

    # Used in the 2nd attention block in the decoder.
    # This padding mask is used to mask the encoder outputs.
    dec_padding_mask = create_padding_mask(inp)

    # Used in the 1st attention block in the decoder.
    # It is used to pad and mask future tokens in the input received by
    # the decoder.
    look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
    dec_target_padding_mask = create_padding_mask(tar)
    combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)

    return enc_padding_mask, combined_mask, dec_padding_mask
```

[301]:

```
EPOCHS = 10
```

[302]:

```
ds = tf.data.Dataset.from_tensor_slices(
    (encoder_input, decoder_input, decoder_input_real)
) \
.cache() \
.shuffle(BUFFER_SIZE) \
.batch(BATCH_SIZE) \
.take(40)
.prefetch(tf.data.AUTOTUNE)
```

[303]:

```
def train_step(inp, tar, tar_real):

    enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar)

    with tf.GradientTape() as tape:
        predictions, _ = transformer(inp, True,
                                     tar=tar,
                                     enc_padding_mask=enc_padding_mask,
                                     look_ahead_mask=combined_mask,
                                     dec_padding_mask=dec_padding_mask)
        loss = loss_function(tar_real, predictions)

    gradients = tape.gradient(loss, transformer.trainable_variables)
    optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))

    train_loss(loss)
    train_accuracy(accuracy_function(tar_real, predictions))
```

[ ]:

```
for epoch in range(0,8):
    start = time.time()
    train_loss.reset_states()
    train_accuracy.reset_states()

    for (batch, (inp, tar, tar_real)) in enumerate(ds):
        train_step(inp, tar, tar_real)

        if batch % 50 == 0:
            print(f'Epoch {epoch} Batch {batch} Loss {train_loss.result():.4f}␣
 ↪Accuracy {train_accuracy.result():.4f}')

    checkpoint_path = f"./output_files/transformer-weights/checkpoint-{epoch}.
 ↪hdf5"
    transformer.save_weights(checkpoint_path)
    print(f'Saving checkpoint for epoch {epoch} at {checkpoint_path}')

    print(f'Epoch {epoch} Loss {train_loss.result():.4f} Accuracy␣
 ↪{train_accuracy.result():.4f}')

    print(f'Time taken for 1 epoch: {time.time() - start:.2f} secs\n')
```

[305]:

```
for epoch in range(4,8):
    start = time.time()
    train_loss.reset_states()
    train_accuracy.reset_states()

    for (batch, (inp, tar, tar_real)) in enumerate(ds):
        train_step(inp, tar, tar_real)

        if batch % 50 == 0:
            print(f'Epoch {epoch} Batch {batch} Loss {train_loss.result():.4f}␣
↪Accuracy {train_accuracy.result():.4f}')

    checkpoint_path = f"./output_files/transformer-weights/checkpoint-{epoch}.
↪hdf5"
    transformer.save_weights(checkpoint_path)
    print(f'Saving checkpoint for epoch {epoch} at {checkpoint_path}')

    print(f'Epoch {epoch} Loss {train_loss.result():.4f} Accuracy␣
↪{train_accuracy.result():.4f}')

    print(f'Time taken for 1 epoch: {time.time() - start:.2f} secs\n')
```

```
Epoch 4 Batch 0 Loss 0.0409 Accuracy 0.9858
Epoch 4 Batch 50 Loss 0.0385 Accuracy 0.9857
Epoch 4 Batch 100 Loss 0.0379 Accuracy 0.9856
Epoch 4 Batch 150 Loss 0.0373 Accuracy 0.9859
Epoch 4 Batch 200 Loss 0.0367 Accuracy 0.9859
Epoch 4 Batch 250 Loss 0.0365 Accuracy 0.9860
Epoch 4 Batch 300 Loss 0.0368 Accuracy 0.9860
Epoch 4 Batch 350 Loss 0.0367 Accuracy 0.9860
Epoch 4 Batch 400 Loss 0.0364 Accuracy 0.9861
Epoch 4 Batch 450 Loss 0.0366 Accuracy 0.9860
Epoch 4 Batch 500 Loss 0.0364 Accuracy 0.9861
Epoch 4 Batch 550 Loss 0.0369 Accuracy 0.9859
Epoch 4 Batch 600 Loss 0.0368 Accuracy 0.9860
Epoch 4 Batch 650 Loss 0.0370 Accuracy 0.9860
Epoch 4 Batch 700 Loss 0.0370 Accuracy 0.9860
Epoch 4 Batch 750 Loss 0.0370 Accuracy 0.9859
Epoch 4 Batch 800 Loss 0.0370 Accuracy 0.9860
Epoch 4 Batch 850 Loss 0.0369 Accuracy 0.9860
Epoch 4 Batch 900 Loss 0.0371 Accuracy 0.9860
Epoch 4 Batch 950 Loss 0.0371 Accuracy 0.9859
Epoch 4 Batch 1000 Loss 0.0371 Accuracy 0.9859
Epoch 4 Batch 1050 Loss 0.0370 Accuracy 0.9860
Epoch 4 Batch 1100 Loss 0.0370 Accuracy 0.9860
Epoch 4 Batch 1150 Loss 0.0369 Accuracy 0.9861
Epoch 4 Batch 1200 Loss 0.0367 Accuracy 0.9861
Epoch 4 Batch 1250 Loss 0.0366 Accuracy 0.9861
Epoch 4 Batch 1300 Loss 0.0365 Accuracy 0.9861
Epoch 4 Batch 1350 Loss 0.0365 Accuracy 0.9862
Epoch 4 Batch 1400 Loss 0.0365 Accuracy 0.9862
Epoch 4 Batch 1450 Loss 0.0365 Accuracy 0.9862
Epoch 4 Batch 1500 Loss 0.0365 Accuracy 0.9862
Epoch 4 Batch 1550 Loss 0.0364 Accuracy 0.9862
Epoch 4 Batch 1600 Loss 0.0363 Accuracy 0.9862
Epoch 4 Batch 1650 Loss 0.0363 Accuracy 0.9862
Epoch 4 Batch 1700 Loss 0.0363 Accuracy 0.9862
Epoch 4 Batch 1750 Loss 0.0364 Accuracy 0.9862
Epoch 4 Batch 1800 Loss 0.0363 Accuracy 0.9862
Epoch 4 Batch 1850 Loss 0.0364 Accuracy 0.9862
Epoch 4 Batch 1900 Loss 0.0364 Accuracy 0.9862
Epoch 4 Batch 1950 Loss 0.0363 Accuracy 0.9862
Epoch 4 Batch 2000 Loss 0.0363 Accuracy 0.9862
Epoch 4 Batch 2050 Loss 0.0362 Accuracy 0.9862
Epoch 4 Batch 2100 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2150 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2200 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2250 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2300 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2350 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2400 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2450 Loss 0.0363 Accuracy 0.9863
Epoch 4 Batch 2500 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2550 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2600 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2650 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2700 Loss 0.0362 Accuracy 0.9863
Epoch 4 Batch 2750 Loss 0.0361 Accuracy 0.9863
Epoch 4 Batch 2800 Loss 0.0361 Accuracy 0.9863
Epoch 4 Batch 2850 Loss 0.0361 Accuracy 0.9863
Epoch 4 Batch 2900 Loss 0.0361 Accuracy 0.9863
Epoch 4 Batch 2950 Loss 0.0361 Accuracy 0.9863
Epoch 4 Batch 3000 Loss 0.0361 Accuracy 0.9863
Epoch 4 Batch 3050 Loss 0.0361 Accuracy 0.9863
Epoch 4 Batch 3100 Loss 0.0360 Accuracy 0.9863
Epoch 4 Batch 3150 Loss 0.0360 Accuracy 0.9863
Epoch 4 Batch 3200 Loss 0.0360 Accuracy 0.9863
Epoch 4 Batch 3250 Loss 0.0360 Accuracy 0.9863
Epoch 4 Batch 3300 Loss 0.0360 Accuracy 0.9863
Epoch 4 Batch 3350 Loss 0.0360 Accuracy 0.9863
Epoch 4 Batch 3400 Loss 0.0359 Accuracy 0.9863
Epoch 4 Batch 3450 Loss 0.0359 Accuracy 0.9863
Epoch 4 Batch 3500 Loss 0.0358 Accuracy 0.9863
Epoch 4 Batch 3550 Loss 0.0358 Accuracy 0.9864
Epoch 4 Batch 3600 Loss 0.0359 Accuracy 0.9863
Epoch 4 Batch 3650 Loss 0.0359 Accuracy 0.9863
Epoch 4 Batch 3700 Loss 0.0358 Accuracy 0.9864
Epoch 4 Batch 3750 Loss 0.0358 Accuracy 0.9863
Epoch 4 Batch 3800 Loss 0.0358 Accuracy 0.9863
Epoch 4 Batch 3850 Loss 0.0358 Accuracy 0.9863
Epoch 4 Batch 3900 Loss 0.0357 Accuracy 0.9864
Epoch 4 Batch 3950 Loss 0.0357 Accuracy 0.9864
Saving checkpoint for epoch 4 at ./output_files/transformer-
weights/checkpoint-4.hdf5
Epoch 4 Loss 0.0357 Accuracy 0.9864
Time taken for 1 epoch: 1095.31 secs

Epoch 5 Batch 0 Loss 0.0494 Accuracy 0.9830
Epoch 5 Batch 50 Loss 0.0357 Accuracy 0.9864
Epoch 5 Batch 100 Loss 0.0344 Accuracy 0.9867
Epoch 5 Batch 150 Loss 0.0351 Accuracy 0.9865
Epoch 5 Batch 200 Loss 0.0353 Accuracy 0.9867
Epoch 5 Batch 250 Loss 0.0357 Accuracy 0.9866
Epoch 5 Batch 300 Loss 0.0355 Accuracy 0.9864
Epoch 5 Batch 350 Loss 0.0354 Accuracy 0.9864
Epoch 5 Batch 400 Loss 0.0356 Accuracy 0.9864
Epoch 5 Batch 450 Loss 0.0355 Accuracy 0.9864
Epoch 5 Batch 500 Loss 0.0355 Accuracy 0.9865
Epoch 5 Batch 550 Loss 0.0353 Accuracy 0.9865
Epoch 5 Batch 600 Loss 0.0350 Accuracy 0.9866
Epoch 5 Batch 650 Loss 0.0351 Accuracy 0.9865
Epoch 5 Batch 700 Loss 0.0353 Accuracy 0.9865
Epoch 5 Batch 750 Loss 0.0352 Accuracy 0.9865
Epoch 5 Batch 800 Loss 0.0350 Accuracy 0.9866
Epoch 5 Batch 850 Loss 0.0350 Accuracy 0.9866
Epoch 5 Batch 900 Loss 0.0348 Accuracy 0.9866
Epoch 5 Batch 950 Loss 0.0348 Accuracy 0.9866
Epoch 5 Batch 1000 Loss 0.0347 Accuracy 0.9866
Epoch 5 Batch 1050 Loss 0.0345 Accuracy 0.9867
Epoch 5 Batch 1100 Loss 0.0347 Accuracy 0.9867
Epoch 5 Batch 1150 Loss 0.0346 Accuracy 0.9867
Epoch 5 Batch 1200 Loss 0.0346 Accuracy 0.9868
Epoch 5 Batch 1250 Loss 0.0345 Accuracy 0.9868
Epoch 5 Batch 1300 Loss 0.0344 Accuracy 0.9867
Epoch 5 Batch 1350 Loss 0.0344 Accuracy 0.9867
Epoch 5 Batch 1400 Loss 0.0345 Accuracy 0.9867
Epoch 5 Batch 1450 Loss 0.0344 Accuracy 0.9867
Epoch 5 Batch 1500 Loss 0.0343 Accuracy 0.9868
Epoch 5 Batch 1550 Loss 0.0343 Accuracy 0.9868
Epoch 5 Batch 1600 Loss 0.0343 Accuracy 0.9868
Epoch 5 Batch 1650 Loss 0.0343 Accuracy 0.9868
Epoch 5 Batch 1700 Loss 0.0342 Accuracy 0.9868
Epoch 5 Batch 1750 Loss 0.0342 Accuracy 0.9868
Epoch 5 Batch 1800 Loss 0.0342 Accuracy 0.9868
Epoch 5 Batch 1850 Loss 0.0341 Accuracy 0.9868
Epoch 5 Batch 1900 Loss 0.0341 Accuracy 0.9868
Epoch 5 Batch 1950 Loss 0.0341 Accuracy 0.9868
Epoch 5 Batch 2000 Loss 0.0340 Accuracy 0.9869
Epoch 5 Batch 2050 Loss 0.0339 Accuracy 0.9869
Epoch 5 Batch 2100 Loss 0.0339 Accuracy 0.9869
Epoch 5 Batch 2150 Loss 0.0339 Accuracy 0.9869
Epoch 5 Batch 2200 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 2250 Loss 0.0339 Accuracy 0.9869
Epoch 5 Batch 2300 Loss 0.0339 Accuracy 0.9869
Epoch 5 Batch 2350 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2400 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2450 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2500 Loss 0.0339 Accuracy 0.9869
Epoch 5 Batch 2550 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 2600 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 2650 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2700 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2750 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2800 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2850 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 2900 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 2950 Loss 0.0340 Accuracy 0.9868
Epoch 5 Batch 3000 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3050 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3100 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3150 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3200 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3250 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3300 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3350 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3400 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3450 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 3500 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 3550 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 3600 Loss 0.0339 Accuracy 0.9868
Epoch 5 Batch 3650 Loss 0.0338 Accuracy 0.9868
Epoch 5 Batch 3700 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 3750 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 3800 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 3850 Loss 0.0338 Accuracy 0.9869
Epoch 5 Batch 3900 Loss 0.0337 Accuracy 0.9869
Epoch 5 Batch 3950 Loss 0.0337 Accuracy 0.9869
Saving checkpoint for epoch 5 at ./output_files/transformer-
weights/checkpoint-5.hdf5
Epoch 5 Loss 0.0337 Accuracy 0.9869
Time taken for 1 epoch: 1163.60 secs

Epoch 6 Batch 0 Loss 0.0345 Accuracy 0.9858
Epoch 6 Batch 50 Loss 0.0342 Accuracy 0.9857
Epoch 6 Batch 100 Loss 0.0328 Accuracy 0.9864
Epoch 6 Batch 150 Loss 0.0331 Accuracy 0.9864
Epoch 6 Batch 200 Loss 0.0331 Accuracy 0.9864
Epoch 6 Batch 250 Loss 0.0329 Accuracy 0.9865
Epoch 6 Batch 300 Loss 0.0328 Accuracy 0.9866
Epoch 6 Batch 350 Loss 0.0329 Accuracy 0.9866
Epoch 6 Batch 400 Loss 0.0333 Accuracy 0.9865
Epoch 6 Batch 450 Loss 0.0331 Accuracy 0.9866
Epoch 6 Batch 500 Loss 0.0332 Accuracy 0.9866
Epoch 6 Batch 550 Loss 0.0331 Accuracy 0.9866
Epoch 6 Batch 600 Loss 0.0331 Accuracy 0.9866
Epoch 6 Batch 650 Loss 0.0330 Accuracy 0.9866
Epoch 6 Batch 700 Loss 0.0328 Accuracy 0.9867
Epoch 6 Batch 750 Loss 0.0330 Accuracy 0.9867
Epoch 6 Batch 800 Loss 0.0329 Accuracy 0.9868
Epoch 6 Batch 850 Loss 0.0329 Accuracy 0.9867
Epoch 6 Batch 900 Loss 0.0330 Accuracy 0.9867
Epoch 6 Batch 950 Loss 0.0331 Accuracy 0.9867
Epoch 6 Batch 1000 Loss 0.0330 Accuracy 0.9868
Epoch 6 Batch 1050 Loss 0.0329 Accuracy 0.9867
Epoch 6 Batch 1100 Loss 0.0329 Accuracy 0.9867
Epoch 6 Batch 1150 Loss 0.0329 Accuracy 0.9868
Epoch 6 Batch 1200 Loss 0.0328 Accuracy 0.9868
Epoch 6 Batch 1250 Loss 0.0328 Accuracy 0.9868
Epoch 6 Batch 1300 Loss 0.0328 Accuracy 0.9868
Epoch 6 Batch 1350 Loss 0.0327 Accuracy 0.9869
Epoch 6 Batch 1400 Loss 0.0327 Accuracy 0.9869
Epoch 6 Batch 1450 Loss 0.0327 Accuracy 0.9869
Epoch 6 Batch 1500 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 1550 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 1600 Loss 0.0325 Accuracy 0.9869
Epoch 6 Batch 1650 Loss 0.0325 Accuracy 0.9869
Epoch 6 Batch 1700 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 1750 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 1800 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 1850 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 1900 Loss 0.0326 Accuracy 0.9870
Epoch 6 Batch 1950 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 2000 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 2050 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 2100 Loss 0.0326 Accuracy 0.9869
Epoch 6 Batch 2150 Loss 0.0325 Accuracy 0.9869
Epoch 6 Batch 2200 Loss 0.0325 Accuracy 0.9869
Epoch 6 Batch 2250 Loss 0.0325 Accuracy 0.9869
Epoch 6 Batch 2300 Loss 0.0325 Accuracy 0.9870
Epoch 6 Batch 2350 Loss 0.0325 Accuracy 0.9870
Epoch 6 Batch 2400 Loss 0.0325 Accuracy 0.9870
Epoch 6 Batch 2450 Loss 0.0324 Accuracy 0.9870
Epoch 6 Batch 2500 Loss 0.0324 Accuracy 0.9870
Epoch 6 Batch 2550 Loss 0.0324 Accuracy 0.9870
Epoch 6 Batch 2600 Loss 0.0323 Accuracy 0.9870
Epoch 6 Batch 2650 Loss 0.0323 Accuracy 0.9871
Epoch 6 Batch 2700 Loss 0.0324 Accuracy 0.9870
Epoch 6 Batch 2750 Loss 0.0323 Accuracy 0.9871
Epoch 6 Batch 2800 Loss 0.0323 Accuracy 0.9871
Epoch 6 Batch 2850 Loss 0.0323 Accuracy 0.9871
Epoch 6 Batch 2900 Loss 0.0323 Accuracy 0.9871
Epoch 6 Batch 2950 Loss 0.0323 Accuracy 0.9871
Epoch 6 Batch 3000 Loss 0.0322 Accuracy 0.9871
Epoch 6 Batch 3050 Loss 0.0322 Accuracy 0.9871
Epoch 6 Batch 3100 Loss 0.0322 Accuracy 0.9871
Epoch 6 Batch 3150 Loss 0.0322 Accuracy 0.9871
Epoch 6 Batch 3200 Loss 0.0322 Accuracy 0.9871
Epoch 6 Batch 3250 Loss 0.0321 Accuracy 0.9871
Epoch 6 Batch 3300 Loss 0.0321 Accuracy 0.9871
Epoch 6 Batch 3350 Loss 0.0321 Accuracy 0.9872
Epoch 6 Batch 3400 Loss 0.0321 Accuracy 0.9872
Epoch 6 Batch 3450 Loss 0.0321 Accuracy 0.9872
Epoch 6 Batch 3500 Loss 0.0321 Accuracy 0.9872
Epoch 6 Batch 3550 Loss 0.0321 Accuracy 0.9872
Epoch 6 Batch 3600 Loss 0.0320 Accuracy 0.9872
Epoch 6 Batch 3650 Loss 0.0320 Accuracy 0.9872
Epoch 6 Batch 3700 Loss 0.0320 Accuracy 0.9872
Epoch 6 Batch 3750 Loss 0.0320 Accuracy 0.9872
Epoch 6 Batch 3800 Loss 0.0320 Accuracy 0.9872
Epoch 6 Batch 3850 Loss 0.0321 Accuracy 0.9872
Epoch 6 Batch 3900 Loss 0.0320 Accuracy 0.9872
Epoch 6 Batch 3950 Loss 0.0320 Accuracy 0.9872
Saving checkpoint for epoch 6 at ./output_files/transformer-
weights/checkpoint-6.hdf5
Epoch 6 Loss 0.0320 Accuracy 0.9872
Time taken for 1 epoch: 1235.89 secs

Epoch 7 Batch 0 Loss 0.0206 Accuracy 0.9901
Epoch 7 Batch 50 Loss 0.0309 Accuracy 0.9873
Epoch 7 Batch 100 Loss 0.0302 Accuracy 0.9874
Epoch 7 Batch 150 Loss 0.0297 Accuracy 0.9879
Epoch 7 Batch 200 Loss 0.0302 Accuracy 0.9876
Epoch 7 Batch 250 Loss 0.0308 Accuracy 0.9874
Epoch 7 Batch 300 Loss 0.0307 Accuracy 0.9874
Epoch 7 Batch 350 Loss 0.0310 Accuracy 0.9874
Epoch 7 Batch 400 Loss 0.0307 Accuracy 0.9875
Epoch 7 Batch 450 Loss 0.0308 Accuracy 0.9873
Epoch 7 Batch 500 Loss 0.0311 Accuracy 0.9873
Epoch 7 Batch 550 Loss 0.0313 Accuracy 0.9872
Epoch 7 Batch 600 Loss 0.0313 Accuracy 0.9873
Epoch 7 Batch 650 Loss 0.0314 Accuracy 0.9872
Epoch 7 Batch 700 Loss 0.0314 Accuracy 0.9873
Epoch 7 Batch 750 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 800 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 850 Loss 0.0314 Accuracy 0.9873
Epoch 7 Batch 900 Loss 0.0314 Accuracy 0.9873
Epoch 7 Batch 950 Loss 0.0313 Accuracy 0.9873
Epoch 7 Batch 1000 Loss 0.0313 Accuracy 0.9873
Epoch 7 Batch 1050 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 1100 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 1150 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 1200 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 1250 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 1300 Loss 0.0314 Accuracy 0.9873
Epoch 7 Batch 1350 Loss 0.0314 Accuracy 0.9873
Epoch 7 Batch 1400 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 1450 Loss 0.0315 Accuracy 0.9873
Epoch 7 Batch 1500 Loss 0.0314 Accuracy 0.9873
Epoch 7 Batch 1550 Loss 0.0314 Accuracy 0.9874
Epoch 7 Batch 1600 Loss 0.0314 Accuracy 0.9874
Epoch 7 Batch 1650 Loss 0.0313 Accuracy 0.9874
Epoch 7 Batch 1700 Loss 0.0313 Accuracy 0.9874
Epoch 7 Batch 1750 Loss 0.0312 Accuracy 0.9874
Epoch 7 Batch 1800 Loss 0.0313 Accuracy 0.9874
Epoch 7 Batch 1850 Loss 0.0313 Accuracy 0.9874
Epoch 7 Batch 1900 Loss 0.0313 Accuracy 0.9874
Epoch 7 Batch 1950 Loss 0.0312 Accuracy 0.9874
Epoch 7 Batch 2000 Loss 0.0311 Accuracy 0.9874
Epoch 7 Batch 2050 Loss 0.0312 Accuracy 0.9874
Epoch 7 Batch 2100 Loss 0.0312 Accuracy 0.9874
Epoch 7 Batch 2150 Loss 0.0311 Accuracy 0.9874
Epoch 7 Batch 2200 Loss 0.0311 Accuracy 0.9874
Epoch 7 Batch 2250 Loss 0.0311 Accuracy 0.9874
Epoch 7 Batch 2300 Loss 0.0311 Accuracy 0.9874
Epoch 7 Batch 2350 Loss 0.0311 Accuracy 0.9874
Epoch 7 Batch 2400 Loss 0.0311 Accuracy 0.9874
Epoch 7 Batch 2450 Loss 0.0310 Accuracy 0.9874
Epoch 7 Batch 2500 Loss 0.0310 Accuracy 0.9875
Epoch 7 Batch 2550 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 2600 Loss 0.0310 Accuracy 0.9875
Epoch 7 Batch 2650 Loss 0.0310 Accuracy 0.9875
Epoch 7 Batch 2700 Loss 0.0310 Accuracy 0.9875
Epoch 7 Batch 2750 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 2800 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 2850 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 2900 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 2950 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3000 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3050 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3100 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3150 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3200 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3250 Loss 0.0310 Accuracy 0.9875
Epoch 7 Batch 3300 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3350 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3400 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3450 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3500 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3550 Loss 0.0308 Accuracy 0.9875
Epoch 7 Batch 3600 Loss 0.0308 Accuracy 0.9875
Epoch 7 Batch 3650 Loss 0.0308 Accuracy 0.9875
Epoch 7 Batch 3700 Loss 0.0308 Accuracy 0.9875
Epoch 7 Batch 3750 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3800 Loss 0.0308 Accuracy 0.9875
Epoch 7 Batch 3850 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3900 Loss 0.0309 Accuracy 0.9875
Epoch 7 Batch 3950 Loss 0.0309 Accuracy 0.9875
Saving checkpoint for epoch 7 at ./output_files/transformer-
weights/checkpoint-7.hdf5
Epoch 7 Loss 0.0309 Accuracy 0.9875
Time taken for 1 epoch: 1142.36 secs
```

[321]:

```python
transformer.load_weights("./output_files/transformer-weights/checkpoint-6.hdf5")
```

[ ]:

```python

```

[322]:

```python
def pad_seq(seq, max_length=MAX_SEQ_LENGTH_HUMAN):
    sentence_length = len(seq)
    if sentence_length >= MAX_SEQ_LENGTH_HUMAN:
        return sentence_length[0:max_length]
    else:
        container = np.zeros((max_length,))

        container[0: sentence_length] = seq
        container[sentence_length: max_length] = 0

        return container

def pad_seqs(seqs, max_length=MAX_SEQ_LENGTH_HUMAN):
    return np.array([pad_seq(seq) for seq in seqs])
```

[323]:

```python
def decode_index_seq(index_array):
    return "".join([tokenizer_human.index_word[index] if index!=0 else "" for index in index_array])
```

[324]:

```
sos_token = tokenizer_human.word_index["<"]
eos_token = tokenizer_human.word_index[">"]
```

[325]:

```
def translate(sentence, max_length=MAX_SEQ_LENGTH_HUMAN):
    tokenized_sentences = tokenizer_human.texts_to_sequences(sentence) # shape:␣
 ↪(None, None, 1)
    padded_tokenized_sentences = pad_seqs(tokenized_sentences)

    _encoder_input = tf.convert_to_tensor(padded_tokenized_sentences)

    # decoder final output:
    output = tf.repeat(tf.convert_to_tensor([[sos_token]]), tf.
 ↪shape(_encoder_input)[0], axis=0)


    for i in range(max_length):
        enc_padding_mask, combined_mask, dec_padding_mask = create_masks(
            _encoder_input, output)

        # predictions.shape == (batch_size, seq_len, vocab_size)

        predictions, _ = transformer(_encoder_input,
                                     False,
                                     tar=output,
                                     enc_padding_mask=enc_padding_mask,
                                     look_ahead_mask=combined_mask,
                                     dec_padding_mask=dec_padding_mask)

        # select the last word from the seq_len dimension

        predictions = predictions[..., -1:,:]
        predicted_id = tf.cast(tf.argmax(predictions, axis=-1), dtype=tf.int32)

        # concatentate the predicted_id to the output which is given to the␣
 ↪decoder
        # as its input.
        output = tf.concat([output, predicted_id], axis=-1)

        # return the result if the predicted_id is equal to the end token
        is_eos = predicted_id == eos_token
        if np.any(is_eos):
            break


    output = list(np.array(output))
    output = [decode_index_seq(seq) for seq in output]
    output = [re.sub(r"\<|\>","", sentence) for sentence in output]

    return output
```

[326]:

```
translate([
    "Sunday 13, June 2021",
    "2021-06-07",
    "June 21, 2021",
    "10 June 2021",
    "2021/06/28",
    "June 18, 2021",
    "05/06/2021",
    "Jun 24, 2021",
    "7/06/2021",
    "2021, 27 June",
    "19 June 2021",
    "Jun 7, 2021",
    "2021-06-05",
    "28 June 2021",
    "June 24, 2021",
    "June 11, 2021",
    "Thursday, June 10, 2021",
    "Sun 06, Jun 2021",
    "Jun 15, 2021",
    "Fri 2021, 25 June",
    "Jun 23, 2021",
    "11/06/2021",
    "Jun 23, 2021",
    "2021-06-15",
    "5 Jun 2021",
    "2021-06-04",
    "2021, 26 Jun",
    "Wed 02, Jun 2021",
    "2021-06-10",
    "Jun 14, 2021",
    "26/06/2021",
    "28/06/2021",
    "2021-06-18",
    "2021, 17 June",
    "14/06/2021",
    "11/06/2021",
    "12/06/2021",
    "25 Jun 2021",
    "01 June 2021",
    "09 June 2021",
    "Tuesday 15, June 2021",
    "2021/06/23"
])
```

```
[326]: ['2021-06-13',
        '2021-06-07',
        '2021-06-21',
        '2021-06-10',
        '2021-06-28',
        '2021-06-18',
        '2021-06-05',
        '2021-06-24',
        '2021-06-07',
        '2021-06-02',
        '2021-06-19',
        '2021-06-07',
        '2021-06-05',
        '2021-06-28',
        '2021-06-24',
        '2021-06-11',
        '2021-06-10',
        '2021-06-06',
        '2021-06-15',
        '2021-06-25',
        '2021-06-02',
        '2021-06-11',
        '2021-06-02',
        '2021-06-15',
        '2021-06-05',
        '2021-06-04',
        '2021-06-02',
        '2021-06-02',
        '2021-06-10',
        '2021-06-14',
        '2021-06-26',
        '2021-06-28',
        '2021-06-18',
        '2021-06-07',
        '2021-06-14',
        '2021-06-11',
        '2021-06-12',
        '2021-06-25',
        '2021-06-01',
        '2021-06-09',
        '2021-06-15',
        '2021-06-23']
```

```
[138]:
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-138-95c74b54868e> in <module>
----> 1 tf.saved_model.save("./output_files/saved_model")

TypeError: save() missing 1 required positional argument: 'export_dir'
```

0.0.1 Minimal Working Sample to Load the Model (ignore the above training part)

[7]:

```
from faker import Faker
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer

import tensorflow as tf
import numpy as np
import random
import os
import pickle
import time
import re

from babel.dates import format_date
```

[8]:

```
files_to_save = {
    "tokenizer_human": "tokenizer_human",
    "MAX_SEQ_LENGTH_HUMAN": "MAX_SEQ_LENGTH_HUMAN"
}
```

[278]:

```
def retrieve_files(files_to_save, target_dir_path):
    stored = {}

    for name, _ in files_to_save.items():
        with open(f"{target_dir_path}/{name}.pickle", "rb") as f:
            stored[name] = pickle.load(f)

    return stored
```

[279]:

```
MODEL_DIR_PATH = "./output_files"
```

[280]:

```
!pwd
```

```
/Users/cc.lee/code/python/2021-04-30-data-importer/2021-04-30-dataimporter-
model/TASK_date_classfication/MultiHeadClassificationModel
```

[281]:

```
stored = retrieve_files(files_to_save, MODEL_DIR_PATH)
tokenizer_human = stored["tokenizer_human"]
MAX_SEQ_LENGTH_HUMAN = stored["MAX_SEQ_LENGTH_HUMAN"]
```

[327]:

```
def create_masks(inp, tar):
    # Encoder padding mask

    enc_padding_mask = create_padding_mask(inp)

    # Used in the 2nd attention block in the decoder.
    # This padding mask is used to mask the encoder outputs.
    dec_padding_mask = create_padding_mask(inp)

    # Used in the 1st attention block in the decoder.
    # It is used to pad and mask future tokens in the input received by
    # the decoder.
    look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
    dec_target_padding_mask = create_padding_mask(tar)
    combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)

    return enc_padding_mask, combined_mask, dec_padding_mask
```

[313]:

```
transformer = Transformer(num_layers=2,
                          d_model=d_model,
                          num_heads=2,
                          dff=1024,
                          input_vocab_size=38,
                          target_vocab_size=38,
                          pe_input=1000,
                          pe_target=1000,
                          rate=0.1)
```

[314]:

```
def preprocess_date(date):
    return date.lower().replace(',', '#')
```

[328]:

```
sos_token = tokenizer_human.word_index["<"]
eos_token = tokenizer_human.word_index[">"]
```

[329]:

```
def build_model():
    x_1 = layers.Input(shape=(29,))

    output = tf.repeat(tf.convert_to_tensor([[sos_token]]),
                       tf.shape(x_1)[0],
                       axis=0)

    for i in range(10):
        enc_padding_mask, combined_mask, dec_padding_mask = create_masks(x_1,␣
 ↪output)

        predictions, _ = transformer(x_1,
                                     False,
                                     tar=output,
                                     enc_padding_mask=enc_padding_mask,
                                     look_ahead_mask=combined_mask,
                                     dec_padding_mask=dec_padding_mask)

        predictions = predictions[..., -1:,:]
        predicted_id = tf.cast(tf.argmax(predictions, axis=-1), dtype=tf.int32)

        output = tf.concat([output, predicted_id], axis=-1)

        is_eos = predicted_id == eos_token

    model = Model(inputs=x_1, outputs=output)
    return model
```

[330]:

```
model = build_model()
```

[ ]:

```
model.summary()
```

[352]:

```
model.trainable_weights
```

[352]:

```
[<tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
/dense_553/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[-0.40964976,  0.52039236,  0.73332024, …, -0.00207204,
          0.70778155,  0.40114227],
        [-0.10679632, -0.0086732 , -0.01255206, …, -0.0019949 ,
          0.09598341,  0.02720796],
        [ 0.30787286,  0.21252446, -0.7091959 , …, -0.5588082 ,
          0.35762155,  0.21563461],
        …,
        [ 0.06075841, -0.0233986 ,  0.00787538, …,  0.13683212,
          0.13005884, -0.00201062],
        [ 0.04399237, -0.01950354,  0.24194726, …,  0.05044842,
          0.00906758,  0.25222495],
        [-0.22005951,  0.06688623,  0.12523176, …,  0.18628588,
          0.02747087, -0.01742271]], dtype=float32)>,
 <tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
/dense_553/bias:0' shape=(38,) dtype=float32, numpy=
 array([-0.17396176,  0.17488591,  0.285062  ,  0.07139803,  0.27562732,
        -0.13243161, -0.08228957,  0.05809747, -0.11128088,  0.25609273,
         0.16956443,  0.30741817,  0.10948374,  0.088388  , -0.12397283,
         0.1121521 ,  0.1291345 ,  0.23913114, -0.1407217 , -0.07656072,
         0.0425591 ,  0.14546505,  0.07015373, -0.2274161 , -0.0426412 ,
         0.09781221, -0.16642112,  0.04331673,  0.07856534,  0.16243832,
         0.38539526, -0.08105636, -0.00938103,  0.38535684,  0.03021771,
         0.1334466 , -0.10111689,  0.01209339], dtype=float32)>,
 <tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
```

```
/dense_554/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[-0.05614511,  0.2485269 ,  0.12740715, …,  0.09997939,
          0.6213989 ,  0.34511894],
        [-0.07723328,  0.13786645,  0.16586709, …, -0.02156743,
         -0.29469487,  0.09203945],
        [ 0.25073436, -0.08776088, -0.5273123 , …, -0.07173521,
          0.04100277, -0.10176151],
        …,
        [-0.1103022 , -0.08649568, -0.15463723, …,  0.17499195,
          0.08590125, -0.01099982],
        [ 0.44808567, -0.21855074,  0.00937106, …,  0.13107853,
          0.20239705,  0.08865139],
        [-0.10620292,  0.02254059,  0.01653521, …, -0.04522082,
          0.05983902,  0.01639634]], dtype=float32)>,
 <tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
/dense_554/bias:0' shape=(38,) dtype=float32, numpy=
 array([-0.09435342,  0.00511545,  0.05416962, -0.05452011,  0.01201749,
         0.0936664 ,  0.02016408,  0.00164627,  0.03067008,  0.04712483,
        -0.00504218,  0.10289849, -0.08646251, -0.00309596, -0.04767759,
        -0.017753  ,  0.00053704,  0.1472078 , -0.04978077, -0.00571504,
         0.08776194, -0.07935676,  0.04927724, -0.038705  ,  0.01940401,
         0.02423641,  0.10932047,  0.01592523,  0.00308596, -0.08504897,
        -0.04678663, -0.09390771,  0.04294543, -0.08921918, -0.14624096,
        -0.04823877, -0.04606212, -0.11117227], dtype=float32)>,
 <tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
/dense_555/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[-0.57913387,  0.24791612,  0.06436277, …,  0.0018548 ,
         -0.3455958 ,  0.31903067],
        [ 0.1105804 ,  0.01049349,  0.07685983, …,  0.16393523,
          0.15510103, -0.13687034],
        [-0.06287875, -0.07620062,  0.10172954, …,  0.00973485,
          0.05687642, -0.1120838 ],
        …,
        [ 0.10742815,  0.17359309,  0.09840602, …, -0.15760875,
         -0.10991785,  0.16712272],
        [ 0.493637  , -0.27510735,  0.78977203, …, -0.10840479,
         -0.26639906, -0.03347263],
        [-0.12961325, -0.04322093, -0.14316519, …,  0.00156705,
          0.17266247, -0.07692775]], dtype=float32)>,
 <tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
/dense_555/bias:0' shape=(38,) dtype=float32, numpy=
 array([-0.03422258,  0.14583133, -0.05856824,  0.01912346,  0.05418937,
        -0.10558899, -0.02229548,  0.06750526, -0.04342275,  0.07756125,
        -0.08730002, -0.04927358,  0.08470028, -0.06623462, -0.02899747,
        -0.04493   ,  0.00046435,  0.03188503, -0.003989  , -0.0377093 ,
        -0.14682981,  0.06568363,  0.06670118,  0.00430415, -0.0471765 ,
         0.00248798,  0.01213492,  0.2054107 ,  0.05987086, -0.0823665 ,
        -0.01907016,  0.0005494 ,  0.03048177, -0.03899961, -0.01366006,
         0.08373517,  0.08015466, -0.13397042], dtype=float32)>,
 <tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
/dense_556/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[ 0.101093  , -0.2846211 ,  0.00613981, …, -0.1273577 ,
         -0.01263796, -0.07374382],
        [ 0.04348213,  0.03092283,  0.14033365, …,  0.13606687,
         -0.04051998, -0.03358254],
        [-0.10530789, -0.0613075 ,  0.06546019, …,  0.4049575 ,
          0.22281529, -0.22130397],
        …,
        [-0.05728371, -0.09547812,  0.18171309, …, -0.01003754,
         -0.14786296,  0.07841884],
        [ 0.00927031,  0.20235391,  0.0227643 , …,  0.15608878,
         -0.04843368, -0.19474517],
        [ 0.08831471, -0.11347775, -0.07253319, …,  0.11185706,
         -0.35264862,  0.12284529]], dtype=float32)>,
 <tf.Variable 'transformer_9/encoder_9/encoder_layer_34/multi_head_attention_102
/dense_556/bias:0' shape=(38,) dtype=float32, numpy=
 array([-2.9591875e-02,  1.5690219e-02, -3.7913572e-02,  5.7595450e-02,
         9.2812322e-02, -6.8399757e-02,  7.4976841e-03, -2.5629986e-02,
        -6.6355832e-02,  1.2356480e-01,  6.8673514e-02, -1.3033906e-01,
        -4.0094439e-02, -6.7517541e-02, -2.2365913e-02, -1.6877525e-01,
         1.2031831e-01, -1.0284259e-01, -3.9868366e-02, -8.4628202e-02,
        -4.0790495e-02,  2.0412178e-01, -2.1762188e-01,  3.5357168e-01,
        -1.1089748e-01,  2.8207579e-01, -1.1910884e-01,  4.3171045e-01,
         4.9790923e-02,  3.2970425e-01,  5.9655301e-02,  3.8679105e-01,
        -4.6603213e-04,  5.2377361e-01,  2.4046272e-02,  4.9898729e-01,
        -4.0643569e-02,  4.2117539e-01], dtype=float32)>,
 <tf.Variable 'dense_557/kernel:0' shape=(38, 128) dtype=float32, numpy=
 array([[ 7.2859511e-02,  3.1414893e-02,  9.2166938e-02, …,
          1.2826498e-01, -4.3010708e-02,  3.2820794e-01],
        [ 9.7702816e-02, -1.1222623e-01,  1.7639142e-01, …,
         -2.6399583e-02, -3.6738813e-02,  2.4288872e-02],
        [-3.2522824e-01, -8.2599064e-03,  1.7594469e-01, …,
          2.0985143e-02,  4.1133101e-04, -2.1436587e-01],
        …,
        [ 3.3758003e-01,  8.3187088e-02,  5.6435231e-02, …,
          5.0215840e-01,  1.9626394e-01,  7.6801524e-02],
        [-6.0370017e-02, -3.5692219e-02,  1.0110032e-01, …,
          4.4030103e-01, -5.6006674e-02,  7.6261580e-02],
        [ 3.5004160e-01,  1.6350052e-01,  8.2365982e-02, …,
         -3.4994817e-01,  2.8660113e-01, -4.0078040e-02]], dtype=float32)>,
 <tf.Variable 'dense_557/bias:0' shape=(128,) dtype=float32, numpy=
 array([ 0.07279857, -0.09298829, -0.05012977, -0.04635949, -0.23055279,
        -0.19698651, -0.32846102, -0.03817198, -0.16998336, -0.12299047,
        -0.23685399, -0.10733417, -0.1594766 , -0.085467  , -0.16442393,
        -0.02571048, -0.23844483, -0.06659303, -0.08353699, -0.12803598,
        -0.03314498, -0.21346226, -0.1743571 , -0.05281466, -0.33412358,
        -0.3251028 , -0.28044325, -0.09373222, -0.37490425, -0.13416018,
        -0.16708666, -0.19218637, -0.21007426, -0.05774385, -0.04596848,
        -0.05573226, -0.07991489, -0.23106782, -0.3000064 , -0.08232519,
        -0.25355557, -0.22579157, -0.20936508, -0.3426416 , -0.21112114,
        -0.3184992 ,  0.04843695, -0.08444095, -0.06127245, -0.15780665,
        -0.26596794, -0.06770577, -0.13669677, -0.24408792, -0.12918001,
        -0.16798775, -0.2489354 , -0.0544064 , -0.17567626, -0.07871567,
        -0.12103759,  0.07077026, -0.18035436, -0.11594912, -0.22665411,
        -0.16341951, -0.2778996 , -0.11151218,  0.086652  , -0.02141662,
        -0.07683555, -0.07677241, -0.15558454, -0.1317716 , -0.14854123,
        -0.04051173, -0.37150246, -0.24034329, -0.09236562, -0.24974328,
        -0.02015428, -0.1380309 , -0.18903175,  0.04795325, -0.17703061,
        -0.32024494,  0.1808613 , -0.20358416, -0.08585177, -0.04238582,
        -0.14736164, -0.34588307, -0.18803555, -0.21116364, -0.05861874,
        -0.16573213, -0.14452465, -0.01214066, -0.05360868, -0.08776929,
        -0.1110662 , -0.11233146, -0.00474726, -0.13215154, -0.07157333,
         0.01828861, -0.0871405 , -0.10788428, -0.22117646, -0.09753168,
        -0.09945745, -0.12955782, -0.22426295, -0.10001817, -0.1698543 ,
         0.04447776, -0.08800023, -0.13676575, -0.08085251, -0.24385698,
        -0.02595037, -0.3110761 , -0.11035013, -0.09557807, -0.1054066 ,
        -0.14253557, -0.02887652, -0.14043023], dtype=float32)>,
 <tf.Variable 'dense_558/kernel:0' shape=(128, 38) dtype=float32, numpy=
 array([[-0.12143448,  0.00086272, -0.12989312, …,  0.34947187,
          0.03206484, -0.31035718],
        [ 0.09715738,  0.04074096, -0.11775273, …, -0.0018328 ,
          0.18343222, -0.18801796],
        [-0.08550633, -0.28503412, -0.07979903, …,  0.00057556,
          0.23043714,  0.01656632],
        …,
        [-0.06419316, -0.2662552 ,  0.47011316, …,  0.3096561 ,
          0.11349862, -0.12106936],
        [ 0.09785876, -0.25526813, -0.23646364, …, -0.21116072,
         -0.40160817, -0.0378092 ],
        [-0.09204739,  0.20455204, -0.20066504, …,  0.3140181 ,
         -0.14127573,  0.11065537]], dtype=float32)>,
 <tf.Variable 'dense_558/bias:0' shape=(38,) dtype=float32, numpy=
 array([ 0.07492775, -0.02611829, -0.02109644, -0.07532184,  0.07839853,
         0.02384976, -0.05237533, -0.0075986 , -0.03042823,  0.00442716,
         0.04246526, -0.02596316,  0.08829019, -0.02041399,  0.19893932,
         0.0037833 , -0.12906267, -0.03613678, -0.00312253,  0.10811041,
         0.02395962, -0.12231601, -0.01281108,  0.0336025 ,  0.06283749,
         0.05651107, -0.2196605 ,  0.17426094,  0.15436055, -0.05249318,
         0.02758104, -0.01425404, -0.06324998,  0.02877528, -0.08017723,
         0.10314448,  0.08613847,  0.05814329], dtype=float32)>,
 <tf.Variable
```

```
'transformer_9/encoder_9/encoder_layer_34/layer_normalization_170/gamma:0'
shape=(38,) dtype=float32, numpy=
 array([1.4441075 , 0.9087335 , 1.1534461 , 1.5371553 , 0.97141045,
        0.8617656 , 0.88696855, 1.2100633 , 0.9104784 , 1.5465591 ,
        0.99959236, 0.83958155, 1.0936724 , 0.765909  , 1.1761596 ,
        0.7354444 , 1.3790406 , 0.69977444, 1.278676  , 0.7616678 ,
        1.3475194 , 0.6877871 , 1.1618495 , 0.6708797 , 0.898661  ,
        0.75182104, 1.1292858 , 0.70937014, 1.2457285 , 0.71617264,
        0.8856458 , 0.76127404, 0.99337506, 0.7343389 , 1.042325  ,
        0.70729077, 1.0173405 , 0.6571918 ], dtype=float32)>,
 <tf.Variable
'transformer_9/encoder_9/encoder_layer_34/layer_normalization_170/beta:0'
shape=(38,) dtype=float32, numpy=
 array([-0.08934474,  0.25050724, -0.04216085, -0.09586551,  0.21670455,
         0.50021994,  0.3058453 ,  0.21612415,  0.34004596, -0.15694661,
         0.06843518,  0.12466445,  0.31883526,  0.04657959,  0.17855161,
         0.01774144, -0.11170631,  0.0079764 , -0.1204358 ,  0.0006353 ,
         0.00578145, -0.24547802,  0.0891439 , -0.11764094, -0.00384085,
        -0.4353755 , -0.03084978, -0.44108155,  0.06776571, -0.15716192,
         0.18012734, -0.16526619, -0.00514767, -0.26425686, -0.03593765,
        -0.44331378,  0.15190312, -0.40976134], dtype=float32)>,
 <tf.Variable
'transformer_9/encoder_9/encoder_layer_34/layer_normalization_171/gamma:0'
shape=(38,) dtype=float32, numpy=
 array([0.61614674, 0.7933342 , 0.7537389 , 0.48273864, 0.72205186,
        0.94069713, 0.89254975, 0.74126536, 0.9208768 , 0.5656058 ,
        0.600388  , 0.88794595, 0.9011245 , 0.86559564, 0.74879706,
        0.9310839 , 0.8704166 , 0.9221775 , 0.8394824 , 0.9278534 ,
        0.84650904, 0.63494885, 0.6797921 , 0.6505988 , 0.9522494 ,
        0.53420085, 0.76917505, 0.6577253 , 0.67920715, 0.7019772 ,
        0.69639355, 0.7475698 , 0.7075492 , 0.7297494 , 0.8201964 ,
        0.4870421 , 0.81492853, 0.7477482 ], dtype=float32)>,
 <tf.Variable
'transformer_9/encoder_9/encoder_layer_34/layer_normalization_171/beta:0'
shape=(38,) dtype=float32, numpy=
 array([ 0.193478  , -0.05034456,  0.09901688,  0.20105301, -0.00357458,
        -0.05045284, -0.11369929,  0.07628641, -0.06013993, -0.01621233,
         0.04026378, -0.0477518 ,  0.02139799, -0.07890806,  0.04003808,
        -0.05555304,  0.0975629 , -0.07841974,  0.07696766, -0.09665344,
         0.15774477, -0.09358294,  0.1640362 , -0.06960419,  0.15408053,
        -0.09763283,  0.1955942 , -0.14547953,  0.11938479, -0.08030584,
        -0.00089731, -0.13937546,  0.1186595 , -0.13043748,  0.13834724,
        -0.10591041,  0.13026702, -0.0438425 ], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
/dense_559/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[ 0.5122677 , -0.8593446 ,  0.25851855, …, -0.00296698,
         -1.155949  , -0.14604145],
        [ 0.15652563,  0.12143001,  0.13017376, …, -0.06932735,
         -0.16675884, -0.2963495 ],
        [ 0.28306362,  0.3441071 , -0.3434764 , …, -0.08505845,
         -0.36557958,  0.05731918],
        …,
        [ 0.06833701,  0.16458839, -0.11208953, …, -0.08162221,
          0.14390485, -0.03568856],
        [-0.72885484,  1.083351  ,  0.68470424, …,  0.5273254 ,
         -0.8383624 ,  0.9015188 ],
        [-0.10852708,  0.12091807, -0.17261806, …, -0.2204123 ,
         -0.12379612,  0.18589333]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
/dense_559/bias:0' shape=(38,) dtype=float32, numpy=
 array([-0.2724132 ,  0.12756538,  0.23844959, -0.02302676,  0.2618235 ,
        -0.30619708,  0.04482136,  0.17513728,  0.00134359, -0.13297331,
        -0.06126351, -0.09933072, -0.36841473, -0.11386275,  0.00155631,
        -0.24167816, -0.18080364, -0.4363927 , -0.24533892, -0.06223096,
         0.13904737,  0.0621074 , -0.06372875, -0.11159159, -0.11176862,
        -0.07867468,  0.04707687, -0.12149544,  0.09844906,  0.07998662,
         0.04424372,  0.06095042,  0.02965046, -0.21325557,  0.1591373 ,
         0.07555739,  0.02688783,  0.14284115], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
/dense_560/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[ 0.03615829, -0.49230054, -0.09027486, …, -0.2069461 ,
          0.02189347,  0.11696358],
        [ 0.20079096, -0.28398052, -0.26313162, …,  0.02449342,
          0.1555148 ,  0.1974333 ],
        [-0.18094224, -0.38870302, -0.06502187, …,  0.2626095 ,
         -0.06314043, -0.00425463],
        …,
        [-0.07493248,  0.1774602 ,  0.00481497, …,  0.03942215,
         -0.06714854,  0.01539691],
        [-2.3103151 ,  1.6229631 , -1.7490324 , …, -0.9801129 ,
          1.1298845 ,  1.7388387 ],
        [ 0.0897113 , -0.04774931,  0.24818611, …,  0.14417838,
          0.03631557, -0.12709579]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
/dense_560/bias:0' shape=(38,) dtype=float32, numpy=
 array([-0.26315603,  0.07918405, -0.03201231,  0.02564438,  0.17927708,
        -0.13992032,  0.06333365, -0.10354129,  0.00106964, -0.1398153 ,
        -0.06956096,  0.1796922 , -0.33248058, -0.25049987,  0.08270468,
        -0.02674901,  0.05818898, -0.18507047, -0.03265176, -0.03186063,
         0.04639814, -0.02869091,  0.09891887,  0.03196888,  0.06617368,
         0.02977878, -0.03455308,  0.12222799, -0.0728898 ,  0.00379757,
        -0.06212729,  0.04568791, -0.05821988, -0.06101574, -0.00898274,
        -0.00114885, -0.05077039,  0.08198435], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
```

```
/dense_561/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[ 0.40959352, -0.1155557 , -0.05102868, …,  0.08901176,
         -0.40350407,  0.2287776 ],
        [ 0.12902984, -0.14433266, -0.12248006, …, -0.08871748,
         -0.06338531,  0.13679105],
        [-0.22371642, -0.02248274,  0.1851166 , …, -0.7629164 ,
          0.0272869 ,  0.5404891 ],
        …,
        [-0.05079174, -0.1940269 , -0.1863002 , …, -0.17472543,
         -0.14576487, -0.2144651 ],
        [ 0.04168563,  0.86545295, -0.15478645, …, -0.30097932,
          0.23239292, -0.0181663 ],
        [ 0.09282206, -0.08921923,  0.11757588, …, -0.00687648,
         -0.02335708,  0.01080824]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
/dense_561/bias:0' shape=(38,) dtype=float32, numpy=
 array([ 0.03064043, -0.04693512, -0.1475027 , -0.17006081,  0.14125717,
         0.11489132, -0.03169391, -0.05674642, -0.09179134,  0.11786098,
         0.18077356, -0.09464739,  0.10064812, -0.15209799,  0.12721594,
         0.10141379, -0.00184545,  0.12792176, -0.06928033, -0.114085  ,
        -0.15763219, -0.06009815,  0.1272002 , -0.0150491 ,  0.06231992,
        -0.02629622,  0.02466792, -0.01117752, -0.21382308, -0.03784376,
        -0.14663595, -0.21799947, -0.13553035,  0.10361103,  0.16772854,
         0.0825173 ,  0.05905463, -0.25817156], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
/dense_562/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[ 0.04623173,  0.12279942,  0.02697926, …, -0.08757572,
         -0.09982454,  0.19446594],
        [-0.0604334 ,  0.12068441,  0.07173846, …,  0.25976807,
         -0.04189776,  0.58954334],
        [-0.03626135,  0.10406281,  0.03897978, …,  0.14192766,
         -0.12000033, -0.36216095],
        …,
        [ 0.09382671, -0.00419206,  0.05544872, …,  0.22104727,
          0.05084592, -0.29562876],
        [ 0.02815729,  0.0888342 ,  0.01748947, …, -0.11954075,
          0.09734182,  0.10083756],
        [ 0.00086038,  0.03327445,  0.06526689, …,  0.6823637 ,
         -0.24564326,  0.04988049]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_103
/dense_562/bias:0' shape=(38,) dtype=float32, numpy=
 array([ 0.04901576, -0.00614506, -0.13652262,  0.06622887, -0.11190352,
        -0.06217531, -0.18171209,  0.1304895 , -0.13554646, -0.03979671,
        -0.00875088, -0.1753945 , -0.06411467, -0.27034965,  0.06420738,
        -0.11982404, -0.06194221, -0.21236442,  0.18011476, -0.4603114 ,
        -0.09812141,  0.24748242,  0.03961634,  0.03878481, -0.0753622 ,
         0.16359201, -0.04447437,  0.0498897 ,  0.04882631,  0.11940278,
        -0.0012266 ,  0.13213891, -0.17886864, -0.12689872,  0.00196903,
         0.06042568, -0.13889085, -0.06069304], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_563/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[-0.23909353,  0.10513999, -0.17998959, …,  0.17054094,
          0.14178367,  0.18801732],
        [-0.02896626, -0.07108904, -0.04616426, …, -0.08295235,
         -0.0049803 ,  0.00936669],
        [ 0.43091816,  0.12679954,  0.04057688, …,  0.08955174,
         -0.28370252, -0.11438134],
        …,
        [ 0.15878358, -0.0593024 ,  0.14055473, …, -0.06256706,
         -0.15593676, -0.1962178 ],
        [ 0.07565884,  0.05900953, -0.02711248, …,  0.13364051,
          0.06978628,  0.0159635 ],
        [-0.02677721, -0.22275908, -0.150723  , …, -0.32492   ,
          0.43927172,  0.19480678]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_563/bias:0' shape=(38,) dtype=float32, numpy=
 array([-0.04213307, -0.18420586, -0.18815912,  0.1573103 , -0.1364249 ,
        -0.1375839 , -0.0503662 , -0.01428482, -0.32733712, -0.16189444,
         0.194131  , -0.11593497,  0.15835348,  0.11389456, -0.42050466,
         0.06777368, -0.13214627,  0.02063339, -0.1662821 ,  0.01917141,
        -0.05869396,  0.09279157, -0.00815394,  0.10485836, -0.3336128 ,
        -0.16502123, -0.3333029 ,  0.25045848, -0.1076412 , -0.06982894,
        -0.24644092, -0.09739772, -0.3650454 , -0.2020506 ,  0.02270378,
         0.01093754, -0.09304817, -0.05307072], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_564/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[ 0.02817391,  0.47441742, -0.15243113, …,  0.00840249,
         -0.34969676, -0.19189976],
        [ 0.08172954,  0.13313198,  0.07581536, …,  0.18436092,
          0.1440607 ,  0.02809223],
        [ 0.00691862,  0.36922082, -0.02664289, …, -0.22597341,
         -0.13403103,  0.11229295],
        …,
        [-0.19543658, -0.25517428, -0.12575269, …,  0.10471652,
          0.3161027 ,  0.11952748],
        [-0.13718133,  0.00433165,  0.08398394, …,  0.00354886,
          0.07686553,  0.24687767],
        [ 0.02507645, -0.39561513, -0.01551186, …, -0.1077125 ,
         -0.05929069,  0.10870571]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_564/bias:0' shape=(38,) dtype=float32, numpy=
 array([-0.02115355, -0.02567852,  0.05889203, -0.04844748, -0.01296771,
         0.0804908 , -0.03147435, -0.00866771, -0.04095369, -0.03429606,
         0.02522344,  0.1311743 , -0.08012091, -0.00716591,  0.10993794,
         0.04596991, -0.05586449, -0.02810492,  0.02654796, -0.06001386,
         0.09012826,  0.05075996,  0.00626152,  0.01752804, -0.03248662,
         0.300461  ,  0.08728015, -0.00213676,  0.195369  , -0.09977771,
         0.05511489, -0.0617052 ,  0.00561759, -0.11607698,  0.0116101 ,
         0.14862612, -0.08767483, -0.01976826], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_565/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[ 0.00328444,  0.16639233,  0.09068017, …,  0.04943657,
         -0.03618049, -0.12583558],
        [ 0.04129633, -0.19988902,  0.37053227, …,  0.01403085,
          0.01126862,  0.19876364],
        [-0.09322397, -0.25050187, -0.05088215, …, -0.04556053,
         -0.00413279, -0.07601038],
        …,
        [-0.03871234,  0.09790694, -0.16926831, …,  0.00505675,
         -0.05924555, -0.00128056],
        [ 0.14622901, -0.08100359,  0.03097776, …, -0.09974961,
          0.02570326, -0.08451513],
        [ 0.01862809, -0.1767745 , -0.38918844, …, -0.29624388,
         -0.08623454, -0.00557421]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_565/bias:0' shape=(38,) dtype=float32, numpy=
 array([ 0.01423602,  0.07704682, -0.02392614,  0.04296039,  0.02883121,
         0.07227495, -0.01798539, -0.00975168, -0.04325554,  0.03991675,
         0.03903126, -0.01668845, -0.08770995,  0.086165  ,  0.01037316,
         0.01840986, -0.02489488, -0.01985141, -0.0223605 ,  0.06496608,
         0.00655224,  0.01317514,  0.01007705, -0.01446813, -0.01920914,
        -0.01510383, -0.04481786, -0.04136822, -0.02507942,  0.04150674,
        -0.00962299, -0.02657003, -0.01123558, -0.05298484,  0.04520378,
        -0.0593414 ,  0.02965568, -0.02910415], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_566/kernel:0' shape=(38, 38) dtype=float32, numpy=
 array([[-0.01844557, -0.04396041, -0.00303952, …, -0.45263445,
          0.00926171,  0.01442608],
        [ 0.04105371, -0.0293091 ,  0.04102304, …,  0.2619573 ,
          0.21488126, -0.29581577],
        [ 0.0107398 , -0.0819639 , -0.01965968, …, -0.14425716,
         -0.11077019, -0.3809548 ],
        …,
        [ 0.06666189, -0.01204396,  0.16239089, …,  0.14632633,
         -0.24231407, -0.2296653 ],
        [-0.0364794 , -0.06718186, -0.08264346, …,  0.29824468,
         -0.0426051 , -0.25299293],
        [-0.01786508, -0.00108848,  0.05456856, …,  0.37134343,
         -0.04919647,  0.07543079]], dtype=float32)>,
 <tf.Variable 'transformer_9/decoder_9/decoder_layer_34/multi_head_attention_104
/dense_566/bias:0' shape=(38,) dtype=float32, numpy=
 array([ 0.01239801,  0.02118396,  0.00689456, -0.06943326, -0.04863349,
         0.05029562,  0.04543632, -0.0366031 ,  0.07887061, -0.01506601,
        -0.01418243,  0.34683272,  0.10690716,  0.14178228, -0.0781272 ,
        -0.00258412, -0.04193587,  0.10845644, -0.0489529 , -0.13886252,
        -0.06959227, -0.04532966, -0.03724841,  0.15228315,  0.04020084,
        -0.08548796, -0.13145547, -0.04071851, -0.04164707,  0.08079514,
        -0.10270299,  0.10014996,  0.09753255,  0.12289149, -0.0891965 ,
         0.02431174, -0.03227336,  0.05817857], dtype=float32)>,
 <tf.Variable 'dense_567/kernel:0' shape=(38, 128) dtype=float32, numpy=
 array([[-0.11345743,  0.53206   ,  0.07336575, …,  0.47902018,
         -0.28135318,  0.5334185 ],
        [ 0.07085124, -0.0226413 ,  0.0384376 , …, -0.30535975,
          0.10561415, -0.19601434],
        [ 0.01107866, -0.5130392 , -0.3785011 , …, -0.37143615,
         -0.28648958, -0.42555013],
        …,
        [ 0.06445488,  0.06335384, -0.19152938, …, -0.19619541,
         -0.03616106, -0.11482652],
        [ 0.04466072,  0.03020582,  0.45490143, …,  0.35496297,
         -0.09342265,  0.2010467 ],
        [-0.01885774, -0.07955883,  0.24069275, …,  0.07380032,
          0.10447666,  0.00623529]], dtype=float32)>,
 <tf.Variable 'dense_567/bias:0' shape=(128,) dtype=float32, numpy=
 array([-0.04450607,  0.00643441, -0.34799632, -0.2854496 , -0.31288245,
        -0.2553996 , -0.23176432,  0.09330481,  0.01927978, -0.14014933,
        -0.22359848, -0.156146  , -0.01200701, -0.02362067, -0.13733053,
        -0.11210506, -0.2604786 ,  0.02041765, -0.12784728, -0.06911938,
        -0.24935961, -0.09365843, -0.00407667, -0.0559851 ,  0.02343519,
        -0.17694783, -0.2725479 , -0.07313289, -0.04563813, -0.06483173,
        -0.0500353 , -0.10518472, -0.15182802, -0.14006603,  0.01279324,
        -0.10069408,  0.05377126, -0.2979015 , -0.26450497, -0.12309876,
        -0.13054182, -0.02683314, -0.31674203, -0.0537986 , -0.3770124 ,
        -0.0590496 , -0.0122818 , -0.2927137 ,  0.02255167, -0.10973098,
        -0.24036027, -0.01330859, -0.1023332 , -0.30984682, -0.23056524,
        -0.14532581, -0.13621537,  0.06472182, -0.05730509, -0.303498  ,
        -0.11705742, -0.2008475 , -0.03751678, -0.33040679, -0.22369495,
        -0.25842854, -0.06034556, -0.41684726, -0.15185602, -0.20964861,
        -0.1028665 , -0.13236164, -0.1688572 , -0.11370926, -0.29362902,
        -0.06474832, -0.20416184, -0.25766364, -0.11634442, -0.03995949,
        -0.08388825, -0.03605665, -0.24201915, -0.5028232 , -0.15468885,
        -0.0975058 , -0.18141021, -0.0722219 , -0.22559695, -0.16122521,
         0.02068361, -0.04937406, -0.07253445, -0.1033738 , -0.18033561,
        -0.08608153, -0.11232701, -0.17222945, -0.20807336, -0.11550017,
        -0.20125172, -0.12049185, -0.27498537, -0.11131138,  0.00941635,
        -0.02420812, -0.06532788, -0.14124383, -0.42632994, -0.00407722,
        -0.16310488, -0.00499444, -0.22961308, -0.19175336, -0.00344441,
        -0.23411204,  0.02502402, -0.162139  , -0.00617188, -0.13287501,
        -0.08217458, -0.02065757, -0.24516797,  0.01497035, -0.40977368,
        -0.28295413,  0.00107345, -0.2144726 ], dtype=float32)>,
 <tf.Variable 'dense_568/kernel:0' shape=(128, 38) dtype=float32, numpy=
 array([[ 0.04121266, -0.50431895,  0.06173474, …, -0.1203643 ,
          0.10226789, -0.0729204 ],
        [-0.04651457,  0.18026944, -0.03008608, …, -0.03720729,
          0.03458369,  0.03185432],
        [-0.23258151,  0.48106062,  0.45643368, …, -0.2393764 ,
          0.02849752,  0.10205806],
        …,
        [ 0.55200344,  0.24641575, -0.10293944, …,  0.21341255,
         -0.05931345,  0.12927438],
        [ 0.35909674, -0.19406265, -0.17039405, …, -0.04543066,
         -0.08216937,  0.03954235],
        [ 0.927332  , -0.20366476, -0.32332426, …, -0.0983675 ,
         -0.35188174,  0.00918702]], dtype=float32)>,
 <tf.Variable 'dense_568/bias:0' shape=(38,) dtype=float32, numpy=
 array([ 0.06638553, -0.20525779,  0.03973005, -0.05664673,  0.07777157,
         0.02314693,  0.24583189, -0.04399427, -0.09853701, -0.34205914,
        -0.00281038,  0.05659873, -0.03889117, -0.03553912, -0.04911485,
         0.00352409, -0.06583348,  0.06671998, -0.02991689, -0.00703496,
         0.05881587, -0.15423271,  0.08533871, -0.03629664, -0.04536777,
        -0.04499539,  0.07627622,  0.12151721, -0.11201126,  0.01448958,
         0.0493449 , -0.0658195 , -0.057404  ,  0.06174005, -0.09977899,
         0.09143578,  0.06726239,  0.00837653], dtype=float32)>,
 <tf.Variable
'transformer_9/decoder_9/decoder_layer_34/layer_normalization_172/gamma:0'
shape=(38,) dtype=float32, numpy=
 array([2.1704924 , 1.225318  , 1.8312612 , 1.7594877 , 1.354147  ,
        1.1730243 , 1.0098956 , 1.4106195 , 0.55447006, 0.58302796,
        0.8152077 , 0.52825713, 0.5668207 , 0.4527319 , 0.83088636,
        0.5885602 , 0.91691184, 0.6301479 , 0.61067265, 0.21271679,
        0.8037579 , 0.654818  , 0.70515215, 0.44588193, 0.78645766,
        0.5818058 , 0.76069134, 0.30317038, 1.0429999 , 0.26305634,
        1.340068  , 0.667818  , 0.62588286, 0.35356596, 1.0193939 ,
        0.5254164 , 0.9056856 , 0.26208937], dtype=float32)>,
 <tf.Variable
'transformer_9/decoder_9/decoder_layer_34/layer_normalization_172/beta:0'
shape=(38,) dtype=float32, numpy=
 array([ 0.09001599,  0.19988117,  0.16872774, -0.14969957, -0.03339565,
         0.13667277,  0.00968201,  0.04000726, -0.0963216 , -0.12799208,
         0.1270502 ,  0.38250995,  0.13497642,  0.03629829, -0.2745367 ,
        -0.04204547,  0.09248626, -0.0297507 ,  0.05336047,  0.043662  ,
        -0.3148286 , -0.13802877,  0.1231536 ,  0.17788804,  0.11459637,
        -0.0744679 , -0.1818755 , -0.23143971, -0.14917514, -0.06870557,
        -0.41925305,  0.08683546,  0.0108069 ,  0.00658735, -0.02936873,
         0.02569089, -0.06599325, -0.00329601], dtype=float32)>,
 <tf.Variable
'transformer_9/decoder_9/decoder_layer_34/layer_normalization_173/gamma:0'
shape=(38,) dtype=float32, numpy=
 array([1.7253504 , 1.1977031 , 1.5437125 , 1.3287625 , 1.091606  ,
        1.0044185 , 1.0651112 , 1.1400944 , 0.5182299 , 0.85382134,
        0.85832554, 0.80455655, 0.55156463, 0.5755133 , 0.9583106 ,
        0.69024014, 0.9269032 , 0.77170885, 0.8313819 , 0.5824907 ,
        1.1707153 , 0.9259094 , 0.89365596, 0.94507945, 0.8562681 ,
        0.75768477, 0.99629533, 0.76712674, 0.9963508 , 0.91172606,
        1.1978464 , 0.94591004, 0.7595127 , 0.8792927 , 0.89082956,
        0.82158095, 1.0177672 , 0.8680439 ], dtype=float32)>,
 <tf.Variable
'transformer_9/decoder_9/decoder_layer_34/layer_normalization_173/beta:0'
shape=(38,) dtype=float32, numpy=
 array([ 0.09721234,  0.06307127,  0.2674986 , -0.04291401, -0.04249191,
         0.13242711,  0.02531108,  0.16793984,  0.02858364, -0.30451217,
        -0.09365593,  0.41610718,  0.17515914,  0.06544274, -0.13649814,
        -0.01192411,  0.01087341,  0.02612657,  0.06588717, -0.34517133,
        -0.39919746,  0.16714874,  0.12337035,  0.34371647, -0.01895471,
        -0.06988403, -0.06411455, -0.10860896, -0.0171715 , -0.03402647,
        -0.16739963,  0.24044973,  0.08231743,  0.14082499, -0.110163  ,
         0.19589186, -0.07703603, -0.05606548], dtype=float32)>,
 <tf.Variable
'transformer_9/decoder_9/decoder_layer_34/layer_normalization_174/gamma:0'
shape=(38,) dtype=float32, numpy=
 array([1.282566  , 0.7039827 , 1.117681  , 1.1117822 , 1.4512475 ,
        0.8483    , 0.5914458 , 1.5661248 , 1.5026959 , 0.41828337,
        1.1636777 , 1.2268984 , 1.3035855 , 1.5387734 , 1.564264  ,
        1.2212363 , 1.4728042 , 1.3713944 , 1.5356042 , 1.4319199 ,
        1.3741368 , 1.4559921 , 1.6108603 , 1.2478675 , 1.4700769 ,
        1.6149068 , 1.4420239 , 1.6399839 , 1.5337337 , 1.4881635 ,
        1.2257255 , 1.1856728 , 1.5213978 , 1.4670684 , 1.4275941 ,
        1.3757225 , 1.3970706 , 1.423031  ], dtype=float32)>,
 <tf.Variable
'transformer_9/decoder_9/decoder_layer_34/layer_normalization_174/beta:0'
shape=(38,) dtype=float32, numpy=
 array([-0.04230683,  0.30814168, -0.00518634, -0.07762142, -0.06435762,
        -0.02938261,  0.0529917 ,  0.24522799, -0.3641168 ,  0.13658899,
         0.02236634, -0.04648977,  0.13047308,  0.12753733, -0.1406914 ,
         0.01709023, -0.27026856,  0.16150656,  0.2575025 , -0.0615279 ,
        -0.04017812,  0.0604474 ,  0.117746  , -0.02153707, -0.04259911,
        -0.1803709 , -0.02515393, -0.11654523, -0.20185106, -0.15495804,
         0.13870934,  0.01690633, -0.17693733,  0.23524852, -0.00975856,
         0.2870341 ,  0.11089627, -0.12598243], dtype=float32)>,
 <tf.Variable 'transformer_9/dense_569/kernel:0' shape=(38, 38) dtype=float32,
numpy=
 array([[ 0.13887621,  0.2220973 ,  0.1915359 , …,  0.19834204,
          0.11811643,  0.15140921],
        [-1.1949714 ,  0.00243711, -1.0948259 , …, -1.0747619 ,
         -1.2435822 , -1.1505017 ],
        [-0.07221192, -0.21547844,  0.02028738, …,  0.01513114,
         -0.10240151, -0.05020587],
        …,
        [-0.10907816,  0.34475675, -0.12040789, …, -0.07399792,
         -0.15999767, -0.07231469],
        [ 0.8442512 ,  0.03864761,  0.9379114 , …,  0.882565  ,
          0.87856746,  0.8509698 ],
        [ 0.8700645 ,  0.03151403,  0.9655878 , …,  1.0177331 ,
          0.8729916 ,  0.9534256 ]], dtype=float32)>,
 <tf.Variable 'transformer_9/dense_569/bias:0' shape=(38,) dtype=float32, numpy=
 array([-4.3363819e+00,  2.3043098e-01, -4.4240046e+00, -4.4137888e+00,
        -1.6209798e-01,  4.2511251e-02,  1.7421097e-01, -4.3750267e+00,
         7.3238358e-02, -4.4304643e+00, -4.5995874e+00,  4.3004258e-03,
         5.5965573e-02, -7.1851742e-03, -4.5344810e+00,  6.2081009e-02,
        -4.4871869e+00, -1.7544076e-02, -4.4146318e+00, -1.1480830e-01,
        -4.4776087e+00, -4.5030494e+00, -4.4123726e+00, -4.4620233e+00,
        -4.3905745e+00, -4.4489698e+00, -4.4163470e+00, -4.4944372e+00,
        -4.4361134e+00, -4.5222316e+00,  4.5920338e-02, -4.3954329e+00,
        -4.4196019e+00, -4.3765998e+00, -4.4132986e+00, -4.4068823e+00,
        -4.3814650e+00, -4.3686018e+00], dtype=float32)>]
```

[331]:

```
model.save("./output_files/saved_model_20210610")
```

```
WARNING:absl:Found untraced functions such as encoder_5_layer_call_fn,
encoder_5_layer_call_and_return_conditional_losses, decoder_5_layer_call_fn,
decoder_5_layer_call_and_return_conditional_losses, dense_149_layer_call_fn
while saving (showing 5 of 305). These functions will not be directly callable
after loading.
WARNING:absl:Found untraced functions such as encoder_5_layer_call_fn,
encoder_5_layer_call_and_return_conditional_losses, decoder_5_layer_call_fn,
decoder_5_layer_call_and_return_conditional_losses, dense_149_layer_call_fn
while saving (showing 5 of 305). These functions will not be directly callable
after loading.
```

```
INFO:tensorflow:Assets written to: ./output_files/saved_model_20210610/assets
```

```
INFO:tensorflow:Assets written to: ./output_files/saved_model_20210610/assets
```

[213]:

```
json_config = model.to_json()
```

[ ]:

```
restore_model = tf.keras.models.load_model(
    "./output_files/saved_model_20210610")
```

[182]:

```
files_to_save = {
    "tokenizer_human": "tokenizer_human",
    "MAX_SEQ_LENGTH_HUMAN": "MAX_SEQ_LENGTH_HUMAN"
}

def retrieve_files(files_to_save, target_dir_path):
    stored = {}

    for name, _ in files_to_save.items():
        with open(f"{target_dir_path}/{name}.pickle", "rb") as f:
            stored[name] = pickle.load(f)

    return stored

stored = retrieve_files(files_to_save, "./output_files")
tokenizer_human = stored["tokenizer_human"]
MAX_SEQ_LENGTH_HUMAN = stored["MAX_SEQ_LENGTH_HUMAN"]
```

[164]:

```
def pad_seq(seq, max_length=MAX_SEQ_LENGTH_HUMAN):
    sentence_length = len(seq)
    if sentence_length >= MAX_SEQ_LENGTH_HUMAN:
        return sentence_length[0:max_length]
    else:
        container = np.zeros((max_length,))

        container[0: sentence_length] = seq
        container[sentence_length: max_length] = 0

        return container

def pad_seqs(seqs, max_length=MAX_SEQ_LENGTH_HUMAN):
    return np.array([pad_seq(seq) for seq in seqs])

def decode_index_seq(index_array):
    return "".join([tokenizer_human.index_word[index] if index!=0 else "" for␣
 ↪index in index_array])
```

[165]:

```
def translate_by_keras_predict(dates):
    tokenized_sentences = tokenizer_human.texts_to_sequences(dates)
    padded_tokenized_sentences = pad_seqs(tokenized_sentences)
    output = restore_model.predict(padded_tokenized_sentences)
    output = list(np.array(output))
    output = [decode_index_seq(seq) for seq in output]
    output = [re.sub(r"\<|\>","", sentence) for sentence in output]
    return output
```

[166]:

```
def preprocess_date(date):
    return date.lower().replace(',', '#')
```

[167]:

```
faker = Faker()

def random_date():
    dt = faker.date_time_between(start_date = '-500y',end_date='+50y')
    try:
        date = format_date(dt, format=random.choice(FORMATS), locale='en')
        human_readable = preprocess_date(date)
        machine_readable = format_date(dt, format="YYYY-MM-dd", locale='en')
    except AttributeError as e:
        return None, None, None
    return human_readable, machine_readable
```

[168]:

```
FORMATS = [
    'short',
    'medium',
    'medium',
    'medium',
    'long','long',
    'long','long',
    'long','full',
    'full','full',
    'd MMM YYY',
    'dd MMM YYY',
    'd MMMM YYY',
    'd MMMM YYY',
    'd MMMM YYY',
    'dd MMMM YYY',
    'dd MMMM YYY',
    'dd MMMM YYY',
    'dd MMMM YYY',
    'd MMMM YYY',
    'd MMMM YYY',
    'dd MMMM YYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'dd/MM/YYYY',
    'YYYY/MM/dd',
    'YYYY/MM/dd',
    'YYYY/MM/dd',
    'EE d, MMM YYY',
    'EE dd, MMM YYY',
    'EEEE d, MMMM YYY',
    'EEEE dd, MMMM YYY',
    'MMM d, YYY',
    'MMM dd, YYY',
    'MMMM d, YYY',
    'MMMM dd, YYY',
    'YYY, d MMM',
    'YYY, d MMMM',
    'YYY, dd MMMM',
    'YYY, dd MMMM',
    'EE YYY, d MMMM',
    'EE YYY, dd MMMM',
    'YYYY-MM-d',
     'YYYY-MM-dd',
     'YYYY-MM-dd',
     'YYYY-MM-dd'
]
```

[169]:

```
dates_human = []
dates_machine = []


for _ in range(20):
    human, machine = random_date()
    # in our random_date generation, "," is replaced by "#", convert it back to␣
 ↪mimic real situation
    human = re.sub(r"#", ",", human)


    dates_human.append(preprocess_date(human))
    dates_machine.append(machine)

translated_dates = translate_by_keras_predict(dates_human)
final_list = zip(dates_human, translated_dates, dates_machine)

for human_date, translated_date, ground_truth_machine_date in final_list:
    human_date_ = re.sub(r"#",",", human_date)
    print(f"human:\t {human_date_}")
    print(f"trans:\t {translated_date}")
    print(f"mach:\t {ground_truth_machine_date}")
    print(f"\n")
```

```
human:   thu 24, mar 2033
trans:   2033-03-24
mach:    2033-03-24
```

```
human:   thursday, november 25, 1677
trans:   1677-11-25
mach:    1677-11-25


human:   sunday 17, august 1969
trans:   1969-08-17
mach:    1969-08-17


human:   1854-10-10
trans:   1854-10-10
mach:    1854-10-10


human:   2021-04-13
trans:   2021-04-13
mach:    2021-04-13


human:   7 august 1885
trans:   1885-08-07
mach:    1885-08-07


human:   13/12/2056
trans:   2056-12-13
mach:    2056-12-13


human:   08/08/2052
trans:   2052-08-08
mach:    2052-08-08


human:   tuesday, november 18, 1625
trans:   1625-11-18
mach:    1625-11-18


human:   26 september 1912
trans:   1912-09-26
mach:    1912-09-26


human:   sun 1812, 15 march
trans:   1812-03-15
mach:    1812-03-15
```

```
human:   22 october 1834
trans:   1834-10-22
mach:    1834-10-22


human:   1882-10-01
trans:   1882-10-01
mach:    1882-10-01


human:   may 20, 1821
trans:   1821-05-20
mach:    1821-05-20


human:   1667/06/24
trans:   1667-06-24
mach:    1667-06-24


human:   1777/06/08
trans:   1777-06-08
mach:    1777-06-08


human:   dec 23, 1526
trans:   1526-12-23
mach:    1526-12-23


human:   1932, 25 april
trans:   1932-04-25
mach:    1932-04-25


human:   april 27, 1563
trans:   1563-04-27
mach:    1563-04-27


human:   mon 1651, 3 april
trans:   1651-04-03
mach:    1651-04-03
```

[ ]:

```
model.save_weights(saved_model_weights_path)
```