Neural Machine Translation: a seq2seq implementation to translate English to German

In [1]:
```
%%capture
!pip install "tensorflow-text"==2.15.0
```

In [2]:
```
# import necessary libs
import numpy as np
import re
import os
import random

%matplotlib inline
import matplotlib.pyplot as plt

import tensorflow_text as tf_text
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import (Layer, Dense, LSTM, Embedding,
                        TextVectorization, Bidirectional, Add,
                        LayerNormalization, Activation)
```

Introduction

This notebook is to implement the seq2seq model with Attention proposed by Bahdanau et al., 2015.

In [4]:
```
# Let's define some constant variables
max_vocab_size=16000
DROPOUT=0.3

BUFFER_SIZE=1024
BATCH_SIZE=64

embedding_size=128
hidden_units=128

data_file="deu.txt"
data_dir='/content/data/'
os.makedirs(data_dir, exist_ok=True)
```

Data Preprocessing

We will use the English to German dataset from Manythings.org.

In [5]:
```
# Download dataset
!wget --no-check-certificate 'https://www.manythings.org/anki/deu-eng.zip' -O deu-eng.zip
!unzip deu-eng.zip -d data/
```

```
--2024-05-09 17:58:21--  https://www.manythings.org/anki/deu-eng.zip
Resolving www.manythings.org (www.manythings.org)... 173.254.30.110
Connecting to www.manythings.org (www.manythings.org)|173.254.30.110|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10364105 (9.9M) [application/zip]
Saving to: ‘deu-eng.zip’

deu-eng.zip         100%[===================>]   9.88M  24.7MB/s    in 0.4s

2024-05-09 17:58:22 (24.7 MB/s) - ‘deu-eng.zip’ saved [10364105/10364105]

Archive:  deu-eng.zip
  inflating: data/deu.txt
  inflating: data/_about.txt
```

In [6]:
```
# Take a look at the first lines
with open(os.path.join(data_dir, data_file)) as f:
    for n, line in enumerate(f):
        print(line.strip())

        if n == 4:
            break
```

```
Go.     Geh.    CC-BY 2.0 (France) Attribution: tatoeba.org #2877272 (CM) & #8597805 (Roujin)
Hi.     Hallo!  CC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) & #380701 (cburgmer)
Hi.     Grüß Gott!      CC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) & #659813 (Esperantostern)
Run!    Lauf!   CC-BY 2.0 (France) Attribution: tatoeba.org #906328 (papabear) & #941078 (Fingerhut)
Run.    Lauf!   CC-BY 2.0 (France) Attribution: tatoeba.org #4008918 (JSakuragi) & #941078 (Fingerhut)
```

To maintain coherence and prevent loss of word's meaning, contracted terms are expanded.

In [7]:
```
en_contraction_map = {
    "let's": "let us",
    "'d better": " had better",
    "'s": " is",
    "'re": " are",
    "'m": " am",
    "'ll": " will",
    "'d": " would",
    "'ve": " have",
    "'em": " them",
    "won't": "will not",
    "n't": " not",
    "cannot": "can not",
}

ger_contraction_map = {
    "'s": " ist",
    "ä": "ae",
    "ö": "oe",
    "ü": "ue",
    "ß": "ss",
    "'ne ": "eine ",
    "'n ": "ein ",
    "am ": "an dem ",
    "ans ": "an das ",
    "aufs ": "auf das ",
    "durchs ": "durch das ",
    "fuers ": "fuer das ",
    "hinterm ": "hinter dem ",
    "im ": "in dem ",
    "uebers ": "ueber das ",
    "ums ": "um das ",
    "unters ": "unter das ",
    "unterm ": "unter dem ",
    "vors ": "vor das ",
    "vorm ": "vor dem ",
    "zum ": "zu dem ",
    "ins ": "in das ",
    "vom ": "von dem" ,
    "beim ": "bei dem ",
    "zur  ": "zu der ",
}

def expand_contractions(text, mapping):
    for key, value in mapping.items():
        text = re.sub(key, value, text)
    return text
```

In [8]:
```
# Let's test the function
print(expand_contractions("He definitely didn't do it. He must've been forced to commit crime. He won't do it again.", en_contraction_map))
print(expand_contractions("Hinterm Haus, das am Fluss liegt, steht ein großer Baum.", ger_contraction_map))
```

```
He definitely did not do it. He must have been forced to commit crime. He will not do it again.
Hinterm Haus, das an dem Fluss liegt, steht ein grosser Baum.
```

The dataset are still in unprocessed form. It is necessary to preprocess and store them in appropriate form.

In [9]:
```
english = []
german = []

with open(os.path.join(data_dir, data_file)) as f:
    for line in f:
        line = line.split("CC-BY")

        if len(line) > 0:
            sample = line[0]
            sample = sample.strip().split('\t')

            english.append(expand_contractions(sample[0].lower(), en_contraction_map))
            german.append(expand_contractions(sample[1].lower(), ger_contraction_map))

english = np.array(english)
german = np.array(german)
```

In [10]:
```
# Take a look at 5 random examples
for i in range(5):
    rdi = random.randint(0, len(english))
    print("{:4} --> {:4}".format(english[rdi], german[rdi]))
```

```
this road is dangerous. --> der weg ist gefaehrlich.
this material is not suitable for a dress. --> dieser stoff ist nicht geeignet fuer ein kleid.
tom hates climbing ladders. --> tom steigt nicht gern auf leitern.
why do you want to do this? --> warum willst du das tun?
he does not get up early. --> er steht nicht frueh auf.
```

In [11]:
```
# english = english[:400]
# german = german[:400]
```

In [12]:
```
# Create mask
mask = np.full((len(english),), False)
train_mask = np.copy(mask)
train_mask[:int(len(english) * 0.8)] = True
np.random.shuffle(train_mask)

false_indices = np.where(train_mask == False)[0]
np.random.shuffle(false_indices)
border_idx = int(len(false_indices) / 2)

val_mask = np.copy(mask)
val_mask[false_indices[:border_idx]] = True

test_mask = np.copy(mask)
test_mask[false_indices[border_idx:]] = True
```

In [13]:
```
train_raw = (
    tf.data.Dataset
    .from_tensor_slices((english[train_mask], german[train_mask]))
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE)
    .prefetch(tf.data.experimental.AUTOTUNE))

val_raw = (
    tf.data.Dataset
    .from_tensor_slices((english[val_mask], german[val_mask]))
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE)
    .prefetch(tf.data.experimental.AUTOTUNE))

test_raw = (
    tf.data.Dataset
    .from_tensor_slices((english[test_mask], german[test_mask]))
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE)
    .prefetch(tf.data.experimental.AUTOTUNE))
```

It can be inferred from the plots that despite some outliers, sentence length tends to remain stable along both dataset. Therefore, it is not necessary to implement bucketing by length.

Tokenization

Computer obviously cannot handle raw text. Instead, they need to be converted into numerical form for further calculations. Besides, while both removing punctuation and lowercasing all words are common practice in NLP tasks, it is not really the case for Neural Machine Translation. Punctuation is important to mark the start or end of a sentence. Therefore, we may well necessarily tokenize them.

In [14]:
```
def text_standardize(text):
  # Split accented characters.
  text = tf_text.normalize_utf8(text, 'NFKD')
  text = tf.strings.lower(text)

  # Keep space, a to z, and select punctuation.
  text = tf.strings.regex_replace(text, '[^ a-z.?!,¿]', '')

  # Add spaces around punctuation.
  text = tf.strings.regex_replace(text, '[.?!,¿]', r' \0 ')

  # Strip whitespace.
  text = tf.strings.strip(text)
  text = tf.strings.join(['[START]', text, '[END]'], separator=' ')

  return text
```

In [15]:
```
# Vectorizer initial
en_vec = TextVectorization(max_tokens=max_vocab_size,
                           standardize=text_standardize,
                           ragged=True)
ger_vec = TextVectorization(max_tokens=max_vocab_size,
                            standardize=text_standardize,
                            ragged=True)
```

In [16]:
```
en_vec.adapt(train_raw.map(lambda x, y: x))
ger_vec.adapt(train_raw.map(lambda x, y: y))
```

In [17]:
```
# Vocabulary
en_voc = en_vec.get_vocabulary()
ger_voc = ger_vec.get_vocabulary()
```

In [18]:
```
# Word to Idx for prediction
word_to_idx = {}

for i in range(len(ger_voc)):
    word_to_idx[ger_voc[i]] = i
```

In [19]:
```
# Assign vocab size of each vectorizer
input_vocab_size = len(en_vec.get_vocabulary())
output_vocab_size = len(ger_vec.get_vocabulary())
print(input_vocab_size)
print(output_vocab_size)
```

```
16000
16000
```

Data Preparation

Structure the dataset to use the tf.keras.models.Model's fit() method.

In [20]:
```
def process_text(context, target):
  context = en_vec(context).to_tensor()
  target = ger_vec(target)
  targ_in = target[:, :-1].to_tensor()
  targ_out = target[:, 1:].to_tensor()
  return (context, targ_in), targ_out

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

Model implementation

In seq2seq, we need a RNN block, which is also known as Encoder, to encode the input sequence to a fixed-length vector, then another RNN block called Decoder to decode it. Block generally consists of LSTM cells.

Encoder

Encoder can be defined multi-layered RNN network. For the sake of simplicity, I will implement it as an one-layer RNN network with 1 cell at each timestep.

Each RNN cell receives a source word and previous hidden state as inputs.

$$s_i = tanh(Ws_{i-1} + Ux_i)$$

According to the formula, the i^{th} hidden state s_i is calculated from the $(i-1)^{th}$ hidden state and the i^{th} input.

In [21]:
```
# Let's define the encoder
class Encoder(Layer):
    def __init__(self,
                tokenizer,
                embedding_size,
                hidden_units,
                dropout=DROPOUT):
        """
            Encoder Block in seq2seq

        :param tokenizer: tokenizer of the source language
        :param embedding_size: dimensionality of the embedding layer
        :param hidden_units: dimensionality of the output
        """

        super().__init__()
        self.tokenizer = tokenizer
        self.embedding_size = embedding_size
        self.hidden_units = hidden_units
        self.vocab_size = tokenizer.vocabulary_size()
        self.embedding = Embedding(input_dim=self.vocab_size,
                                   output_dim=embedding_size)
        self.rnn = Bidirectional(
            merge_mode="sum",
            layer=LSTM(units=hidden_units,
                    dropout=dropout,
                    return_sequences=True,
                    return_state=True))

    def call(self,
            x,
            training=True):
        """
        :param x: [batch, time_steps]
        :param training: is training or not
        :return:
            encoder_hidden_state: [batch, hidden_state_dim]
            state_h: [batch, hidden_state_dim]
            state_c: [batch, hidden_state_dim]
        """
        mask = tf.where(x != 0, True, False)
        x = self.embedding(x)
        x, forward_h, forward_c, backward_h, backward_c = self.rnn(x, mask=mask,
                                                                   training=training)

        return x, forward_h + backward_h, forward_c + backward_c
```

Attention Layer

The Attention Mechanism used in this project is Bahdanau Attention, which is first introduced in the Neural Machine Translation by Jointly Learning to Align and Translate paper by Bahdanau et al., 2015. \ To recap, at each inference step i, Decoder incorporates information from both the previous Decoder timestep s_{i-1} and all encoder states $h = (\{h_1, h_2, \ldots, h_{Tx}\})$ to take only the most relevant words to y_{i-1} through alignment model a.

$$e_{ij} = a(s_{i-1}, h_j) = v_a^T.tanh(W_a s_{i-1} + U_a h_j)$$

Therefore, the context vector c_i can be calculated as

$$c_i = \sum_{j=1}^{Tx} \alpha_{ij} h_j$$

in which

$$\alpha_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{Tx} exp(e_{ik})}$$

During the training process, we will implement Teacher Forcing by combining context vector c_i with Decoder input x_i.

In [22]:
```
class BahdanauAttention(Layer):
    def __init__(self,
                 hidden_units):
        super().__init__()
        self.Va = Dense(1)
        self.Wa = Dense(hidden_units)
        self.Ua = Dense(hidden_units)
        self.norm = LayerNormalization()
        self.tanh = Activation(tf.keras.activations.tanh)
        self.add = Add()

    def call(self,
             context, x):
        """
            Calculate the context vector based on all encoder hidden states and
            previous decoder state.

        :param: context: tensor, all encoder hidden states
        :param: x: tensor, previous state from Decoder
        :return:
            context_vector: tensor, the calculated context vector based on the
            input parameters
        """
        # Expand dims to ensure scores shape = [batch, Ty, Tx]
        context = tf.expand_dims(context, axis=1)
        x = tf.expand_dims(x, axis=2)

        scores = self.Va(self.tanh(self.add([self.Wa(context), self.Ua(x)])))
        scores = tf.squeeze(scores, axis=-1)
        attn_weights = tf.nn.softmax(scores, axis=-1)

        # NOTE: context shape = [batch, 1, Tx, feature] so that expand
        # dim of attention weights
        context_vector = tf.expand_dims(attn_weights, axis=-1) * context
        context_vector = tf.reduce_sum(context_vector, axis=-2)
        context_vector = self.norm(context_vector)
        context_vector = self.add([context_vector, tf.squeeze(x, -2)])

        return context_vector
```

Decoder

Encoder and Decoder share the same structure as well as hidden units but the last dense layer at each state which holds for predicting the next word using a softmax.

In [23]:
```
# Let's define the decoder
class Decoder(Layer):
    def __init__(self,
                tokenizer,
                embedding_size,
                hidden_units,
                dropout=DROPOUT):
        """
            Decoder Block in seq2seq

        :param tokenizer: tokenizer of the source language
        :param embedding_size: dimensionality of the embedding layer
        :param hidden_units: dimensionality of the output
        """

        super().__init__()
        self.tokenizer = tokenizer
        self.embedding_size = embedding_size
        self.hidden_units = hidden_units
        self.vocab = tokenizer.get_vocabulary()
        self.vocab_size = tokenizer.vocabulary_size()
        self.embedding = Embedding(input_dim=self.vocab_size,
                                   output_dim=embedding_size)
        self.rnn = LSTM(units=hidden_units,
                        dropout=dropout,
                        return_sequences=True,
                        return_state=True)
        self.attention = BahdanauAttention(hidden_units)
        self.dense = Dense(self.vocab_size)

    def call(self,
            context, x,
            encoder_state,
            training=True,
            return_state=False):
        """
        :param context: all encoder states
        :param x: all initial decoder states
        :param encoder_state: last state from encoder
        :param training:
        :param return_state:

        :return:
            logits:
            state_h: hidden state
            state_c: cell state
        """
        mask = tf.where(x != 0, True, False)
        x = self.embedding(x)
        decoder_outputs, state_h, state_c = self.rnn(x, initial_state=encoder_state,
                                                     mask=mask,
                                                     training=training)
        dense_inputs = self.attention(context, decoder_outputs)
        logits = self.dense(dense_inputs)

        if return_state:
            return logits, state_h, state_c
        else:
            return logits
```

seq2seq

Now we have got the Encoder and Decoder. Let's combine them into seq2seq model.

In [24]:
```
class NMT(Model):
    @classmethod
    def add_method(cls, fun):
        setattr(cls, fun.__name__, fun)
        return fun

    def __init__(self,
                 input_tokenizer,
                 output_tokenizer,
                 embedding_size,
                 hidden_units):
        """
            Initialize an instance for Neural Machine Translation Task

        :param input_tokenizer: tokenizer of the input language
        :param output_tokenizer: tokenizer of the output language
        :param embedding_size: dimensionality of embedding layer
        :param hidden_units: dimensionality of the output
        """

        super().__init__()
        self.input_tokenizer = input_tokenizer
        self.output_tokenizer = output_tokenizer
        self.embedding_size = embedding_size
        self.hidden_units = hidden_units
        self.encoder = Encoder(input_tokenizer,
                               embedding_size,
                               hidden_units)
        self.decoder = Decoder(output_tokenizer,
                               embedding_size,
                               hidden_units)

    def call(self,
             inputs):
        encoder_inputs, decoder_inputs = inputs
        encoder_outputs, state_h, state_c = self.encoder(encoder_inputs)
        logits = self.decoder(encoder_outputs, decoder_inputs,
                              [state_h, state_c])

        return logits

    def get_config(self):
        config = super().get_config()
        config.update({
            "input_tokenizer": tf.keras.utils.serialize_keras_object(self.input_tokenizer),
            "output_tokenizer": tf.keras.utils.serialize_keras_object(self.output_tokenizer),
            "embedding_size": self.embedding_size,
            "hidden_units": self.hidden_units
        })

        return {**config}
```

In [25]:
```
@NMT.add_method
def translate(self, next_inputs,
              maxlen=40):
    """
    """
    def sampling(logits):
        probs = tf.nn.softmax(logits)
        dist = probs.numpy().squeeze()
        idx = np.random.choice(range(self.decoder.vocab_size), p=dist)

        return idx

    translation = []
    next_inputs = expand_contractions(next_inputs.lower(), en_contraction_map)
    next_idx = np.asarray(self.encoder.tokenizer(next_inputs))

    while next_idx.ndim != 2:
        next_idx = tf.expand_dims(next_idx, axis=0)

    encoder_outputs, state_h, state_c = self.encoder(next_idx, training=False)

    next_inputs = "[START]"
    next_idx = np.asarray(word_to_idx[next_inputs])

    for i in range(maxlen):
        while next_idx.ndim != 2:
            next_idx = tf.expand_dims(next_idx, axis=0)

        logits, state_h, state_c = self.decoder(encoder_outputs, next_idx,
                                                [state_h, state_c],
                                                training=False,
                                                return_state=True)
        next_idx = sampling(logits)
        next_inputs = self.decoder.vocab[next_idx]

        if next_inputs == "[END]":
            break
        elif next_inputs == "[UNK]":
            continue
        else:
            translation.append(next_inputs)

    return " ".join(translation)
```

In [26]:
```
model = NMT(en_vec, ger_vec, embedding_size, hidden_units)
model.compile(optimizer=tf.keras.optimizers.Adam(0.005),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=["accuracy"])
```

In [30]:
```
history = model.fit(train_ds,
                    epochs=5,
                    validation_data=val_ds)
```

```
Epoch 1/5
3474/3474 [==============================] - 187s 54ms/step - loss: 0.6670 - accuracy: 0.8328 - val_loss: 0.9623 - val_accuracy: 0.7962
Epoch 2/5
3474/3474 [==============================] - 186s 54ms/step - loss: 0.6525 - accuracy: 0.8352 - val_loss: 0.9489 - val_accuracy: 0.7992
Epoch 3/5
3474/3474 [==============================] - 188s 54ms/step - loss: 0.6415 - accuracy: 0.8366 - val_loss: 0.9559 - val_accuracy: 0.7979
Epoch 4/5
3474/3474 [==============================] - 182s 52ms/step - loss: 0.6330 - accuracy: 0.8383 - val_loss: 0.9536 - val_accuracy: 0.7993
Epoch 5/5
3474/3474 [==============================] - 185s 53ms/step - loss: 0.6275 - accuracy: 0.8390 - val_loss: 0.9461 - val_accuracy: 0.8004
```

In [28]:
```
model.evaluate(test_ds)
```

```
435/435 [==============================] - 16s 36ms/step - loss: 0.9508 - accuracy: 0.7982
```

Out[28]:
```
[0.9508467316627502, 0.7981686592102051]
```

In [29]:
```
model.save_weights("model_v8.weights.h5")
```

In [39]:
```
model.translate("Come on!")
```

Out[39]:
```
'komm demnaechst !'
```

In [ ]: