Capstone Project

Neural translation model

By Hamilton Hitchings

2021-1-12

Instructions

In this notebook, you will create a neural network that translates from English to German. You will use concepts from throughout this course, including building more flexible model architectures, freezing layers, data processing pipeline and sequence modelling.

This project is peer-assessed. Within this notebook you will find instructions in each section for how to complete the project. Pay close attention to the instructions as the peer review will be carried out according to a grading rubric that checks key parts of the project instructions. Feel free to add extra cells into the notebook as required.

How to submit

When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able to view the outputs of your notebook. Save the notebook as a pdf (you could download the notebook with File -> Download .ipynb, open the notebook locally, and then File -> Download as -> PDF via LaTeX), and then submit this pdf for review.

Let's get started!

We'll start by running some imports, and loading the dataset. For this project you are free to make further imports throughout the notebook as you wish.

```
import tensorflow as tf
import tensorflow_hub as hub
import unicodedata
import re
from IPython.display import Image
import matplotlib.pyplot as plt

import json
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Layer, Input, Masking, LSTM, Dense, Embedding
from tensorflow.keras.models import Model

from tensorflow.keras.utils import to_categorical
```

For the capstone project, you will use a language dataset from http://www.manythings.org/anki/ to build a neural translation model. This dataset consists of over 200,000 pairs of sentences in English and German. In order to make the training quicker, we will restrict to our dataset to 20,000 pairs. Feel free to change this if you wish - the size of the dataset used is not part of the grading rubric.

Your goal is to develop a neural translation model from English to German, making use of a pre-trained English word embedding module.

Import the data

The dataset is available for download as a zip file at the following link:

https://drive.google.com/open?id=1KczOciG7sYY7SB9UlBeRP1T9659b121Q

You should store the unzipped folder in Drive for use in this Colab notebook.

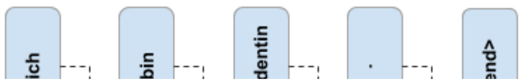
```
# Run this cell to connect to your Drive folder

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

```
    Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force_re
# Run this cell to load the dataset

NUM_EXAMPLES = 20000
data_examples = []
with open('/content/gdrive/MyDrive/data/deu.txt', 'r', encoding='utf8') as f:
    for line in f.readlines():
        if len(data_examples) < NUM_EXAMPLES:
            data_examples.append(line)
        else:
            break


# These functions preprocess English and German sentences

def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s) if unicodedata.category(c) != 'Mn')

def preprocess_sentence(sentence):
    sentence = sentence.lower().strip()
    sentence = re.sub(r"ü", 'ue', sentence)
    sentence = re.sub(r"ä", 'ae', sentence)
    sentence = re.sub(r"ö", 'oe', sentence)
    sentence = re.sub(r'ß', 'ss', sentence)

    #sentence = unicode_to_ascii(sentence)
    sentence = re.sub(r"([?.!,])", r" \1 ", sentence)
    sentence = re.sub(r"[^a-z?.!,']+", " ", sentence)
    sentence = re.sub(r'[" "]+', " ", sentence)

    return sentence.strip()
```

The custom translation model

The following is a schematic of the custom translation model architecture you will develop in this project.

```
# Run this cell to download and view a schematic diagram for the neural translation model

!wget -q -O neural_translation_model.png --no-check-certificate "https://docs.google.com/uc?export=download&id=1XsS1VlXoaEo
Image("neural_translation_model.png")
```



The custom model consists of an encoder RNN and a decoder RNN. The encoder takes words of an English sentence as input, and uses a pre-trained word embedding to embed the words into a 128-dimensional space. To indicate the end of the input sentence, a special end token (in the same 128-dimensional space) is passed in as an input. This token is a TensorFlow Variable that is learned in the training phase (unlike the pre-trained word embedding, which is frozen).

The decoder RNN takes the internal state of the encoder network as its initial state. A start token is passed in as the first input, which is embedded using a learned German word embedding. The decoder RNN then makes a prediction for the next German word, which during inference is then passed in as the following input, and this process is repeated until the special `<end>` token is emitted from the decoder.

1. Text preprocessing

- Create separate lists of English and German sentences, and preprocess them using the `preprocess_sentence` function provided for you above.
- Add a special `"<start>"` and `"<end>"` token to the beginning and end of every German sentence.
- Use the Tokenizer class from the `tf.keras.preprocessing.text` module to tokenize the German sentences, ensuring that no character filters are applied. *Hint: use the Tokenizer's "filter" keyword argument.*
- Print out at least 5 randomly chosen examples of (preprocessed) English and German sentence pairs. For the German sentence, print out the text (with start and end tokens) as well as the tokenized sequence.
- Pad the end of the tokenized German sequences with zeros, and batch the complete set of sequences into one numpy array.

```
data_examples[0:5]
```

```
    ['Hi.\tHallo!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) & #380701 (cburgmer)\n',
     'Hi.\tGrüß Gott!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) & #659813 (Esperantostern)\n',
     'Run!\tLauf!\tCC-BY 2.0 (France) Attribution: tatoeba.org #906328 (papabear) & #941078 (Fingerhut)\n',
     'Wow!\tPotzdonner!\tCC-BY 2.0 (France) Attribution: tatoeba.org #52027 (Zifre) & #2122382 (Pfirsichbaeumchen)\n',
     'Wow!\tDonnerwetter!\tCC-BY 2.0 (France) Attribution: tatoeba.org #52027 (Zifre) & #2122391 (Pfirsichbaeumchen)\n']
```

```
# Create separate lists of English and German sentences, and preprocess them using the preprocess_sentence
# function provided for you above.
# Add a special "<start>" and "<end>" token to the beginning and end of every German sentence.
def preprocess_example(sentence):
  sentences = sentence.split("\t")
  english_sentence = preprocess_sentence(sentences[0])
  english_sentence = unicode_to_ascii(english_sentence)
  german_sentence = "<start> " + preprocess_sentence(sentences[1]) + " <end>"
  german_sentence = unicode_to_ascii(german_sentence)

  return (english_sentence, german_sentence)



english_sentences = []
german_sentences = []

for example in data_examples:
  (english_sentence, german_sentence) = preprocess_example(example)
  english_sentences.append(english_sentence)
  german_sentences.append(german_sentence)

print(f'Number of English sentences: {len(english_sentences)}')
print(f'Number of German sentences: {len(german_sentences)}')
```

```
    Number of English sentences: 20000
    Number of German sentences: 20000
```

```
# Use the Tokenizer class from the tf.keras.preprocessing.text module to tokenize the German sentences,
# ensuring that no character filters are applied. Hint: use the Tokenizer's "filter" keyword argument.
tokenizer = Tokenizer(filters='')

# Build the Tokenizer vocabulary
```

```
tokenizer.fit_on_texts(german_sentences)

# Pad the end of the tokenized German sequences with zeros,
german_seq_chunks = tokenizer.texts_to_sequences(german_sentences)


# Print out at least 5 randomly chosen examples of (preprocessed) English and German sentence pairs.
# For the German sentence, print out the text (with start and end tokens) as well as the tokenized sequence.
for i in np.random.randint(20000, size=5):
  print(f'English sentence: \"{english_sentences[i]}\"')
  print(f'German sentence: \"{german_sentences[i]}\"')
  print(f'German sequence: {german_seq_chunks[i]}')
  print('')
```

```
English sentence: "get on with it ."
German sentence: "<start> nun machen sie schon weiter ! <end>"
German sequence: [1, 1049, 86, 8, 96, 126, 9, 2]

English sentence: "tom was unhappy ."
German sentence: "<start> tom war ungluecklich . <end>"
German sequence: [1, 5, 24, 1008, 3, 2]

English sentence: "the week is over ."
German sentence: "<start> die woche ist vorbei . <end>"
German sequence: [1, 26, 1756, 6, 300, 3, 2]

English sentence: "tom seemed tough ."
German sentence: "<start> tom schien widerstandsfaehig zu sein . <end>"
German sequence: [1, 5, 296, 5300, 20, 54, 3, 2]

English sentence: "come closer ."
German sentence: "<start> kommen sie naeher . <end>"
German sequence: [1, 92, 8, 727, 3, 2]
```

```
tokenizer_config = tokenizer.get_config()

# Used later on
german_index_word = json.loads(tokenizer_config['index_word'])
german_word_index = json.loads(tokenizer_config['word_index'])

german_vocabulary_length = len(german_index_word)
print(f'German Vocabulatry Length={german_vocabulary_length}')
print(len(tokenizer.word_index))
```

```
German Vocabulatry Length=5744
5744
```

```
# Pad the end of the tokenized German sequences with zeros, and batch the complete set of sequences into one numpy array.
german_padded_sequences = np.array(tf.keras.preprocessing.sequence.pad_sequences(german_seq_chunks,
                                                        padding='post',
                                                        value=0))

print(f'german_padded_sequences.shape={german_padded_sequences.shape}')
```

```
german_padded_sequences.shape=(20000, 14)
```

2. Prepare the data

Load the embedding layer

As part of the dataset preproceessing for this project, you will use a pre-trained English word embedding module from TensorFlow Hub. The URL for the module is https://tfhub.dev/google/tf2-preview/nnlm-en-dim128-with-normalization/1.

This embedding takes a batch of text tokens in a 1-D tensor of strings as input. It then embeds the separate tokens into a 128-dimensional space.

The code to load and test the embedding layer is provided for you below.

NB: this model can also be used as a sentence embedding module. The module will process each token by removing punctuation and splitting on spaces. It then averages the word embeddings over a sentence to give a single embedding vector. However, we will use it only as a word embedding module, and will pass each word in the input sentence as a separate token.

```
# Load embedding module from Tensorflow Hub

embedding_layer = hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim128/1",
                                 output_shape=[128], input_shape=[], dtype=tf.string)

    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
```

```
# Test the layer

embedding_layer(tf.constant(["these", "aren't", "the", "droids", "you're", "looking", "for"])).shape

    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
    WARNING:tensorflow:11 out of the last 11 calls to <function recreate_function.<locals>.restored_function_body at 0x7f3
    TensorShape([7, 128])
```

from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.1)You should now prepare the training and validation Datasets.

- Create a random training and validation set split of the data, reserving e.g. 20% of the data for validation (NB: each English dataset example is a single sentence string, and each German dataset example is a sequence of padded integer tokens).
- Load the training and validation sets into a tf.data.Dataset object, passing in a tuple of English and German data for both training and validation sets.
- Create a function to map over the datasets that splits each English sentence at spaces. Apply this function to both Dataset objects using the map method. *Hint: look at the tf.strings.split function.*
- Create a function to map over the datasets that embeds each sequence of English words using the loaded embedding layer/model. Apply this function to both Dataset objects using the map method.
- Create a function to filter out dataset examples where the English sentence is greater than or equal to than 13 (embedded) tokens in length. Apply this function to both Dataset objects using the filter method.
- Create a function to map over the datasets that pads each English sequence of embeddings with some distinct padding value before the sequence, so that each sequence is length 13. Apply this function to both Dataset objects using the map method. *Hint: look at the tf.pad function. You can extract a Tensor shape using tf.shape; you might also find the tf.math.maximum function useful.*
- Batch both training and validation Datasets with a batch size of 16.
- Print the `element_spec` property for the training and validation Datasets.
- Using the Dataset `.take(1)` method, print the shape of the English data example from the training Dataset.
- Using the Dataset `.take(1)` method, print the German data example Tensor from the validation Dataset.

```
# Create a random training and validation set split of the data, reserving e.g. 20% of the data for validation
X_train, X_valid, y_train, y_valid = train_test_split(english_sentences, german_padded_sequences, test_size=0.2)
```

```
# Load the training and validation sets into a tf.data.Dataset object, passing in a tuple of English and German data
# for both training and validation sets.
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
print(f'train_dataset={train_dataset}')

valid_dataset = tf.data.Dataset.from_tensor_slices((X_valid, y_valid))

    train_dataset=<TensorSliceDataset shapes: ((), (14,)), types: (tf.string, tf.int32)>
```

```
# Create a function to map over the datasets that splits each English sentence at spaces.
# Apply this function to both Dataset objects using the map method. Hint: look at the tf.strings.split function

def split_words(english_sentence, german_sentence):
  return tf.strings.split(english_sentence, sep=" "), german_sentence

train_dataset = train_dataset.map(split_words)
valid_dataset = valid_dataset.map(split_words)
```

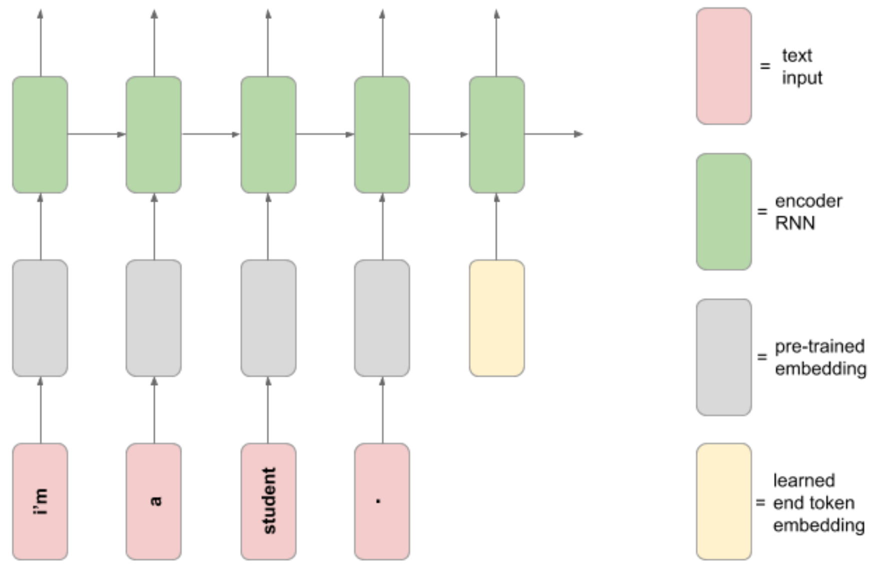
```
# Create a function to map over the datasets that embeds each sequence of English words using the loaded embedding layer/mo
# Apply this function to both Dataset objects using the map method
# Apply this function to both Dataset objects using the map method.

def embed_english_words(english_sentence, german_sentence):
  return embedding_layer(english_sentence), german_sentence

train_dataset = train_dataset.map(embed_english_words)
valid_dataset = valid_dataset.map(embed_english_words)


# Create a function to filter out dataset examples where the English sentence is greater than or equal to than 13
# (embedded) tokens in length. Apply this function to both Dataset objects using the filter method.

def filter_long_sentences(english_sentence, german_sentence):
  return len(english_sentence) < 13

train_dataset = train_dataset.filter(filter_long_sentences)
valid_dataset = valid_dataset.filter(filter_long_sentences)

print(train_dataset.element_spec)

    (TensorSpec(shape=(None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(14,), dtype=tf.int32, name=None))


# Create a function to map over the datasets that pads each English sequence of embeddings with some distinct padding
# value before the sequence, so that each sequence is length 13. Apply this function to both Dataset objects using the map
# Hint: look at the tf.pad function. You can extract a Tensor shape using tf.shape;
# you might also find the tf.math.maximum function useful.

def pad_english(english_sentence, german_sentence):
  paddings = [[0, 13 - tf.shape(english_sentence)[0]], [0, 0]]
  english_sentence = tf.pad(english_sentence, paddings, 'CONSTANT', constant_values=0)
  # english_sentence = tf.reshape(english_sentence, [13, 128])
  return english_sentence, german_sentence

train_dataset = train_dataset.map(pad_english)
valid_dataset = valid_dataset.map(pad_english)


# Batch both training and validation Datasets with a batch size of 16.

batch_size = 16
train_dataset = train_dataset.batch(batch_size, drop_remainder=True)
valid_dataset = valid_dataset.batch(batch_size, drop_remainder=True)


# Print the element_spec property for the training and validation Datasets.
print("train_dataset.element_spec=")
print(train_dataset.element_spec)
print("valid_dataset.element_spec=")
print(valid_dataset.element_spec)

# Using the Dataset .take(1) method, print the shape of the English data example from the training Dataset.
for elem in train_dataset.take(1):
  english_data = elem[0]
  print(f'train_dataset english data.shape={english_data.shape}')

# Using the Dataset .take(1) method, print the German data example Tensor from the validation Dataset.
for elem in valid_dataset.take(1):
  german_data = elem[1]
  print(f'valid_dataset german data.shape={german_data.shape}')
  print(f'valid_dateset german_data={german_data}')

    train_dataset.element_spec=
    (TensorSpec(shape=(16, None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(16, 14), dtype=tf.int32, name=None)
    valid_dataset.element_spec=
    (TensorSpec(shape=(16, None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(16, 14), dtype=tf.int32, name=None)
    train_dataset english data.shape=(16, 13, 128)
    valid_dataset german data.shape=(16, 14)
    valid_dateset german_data=[[   1    4  813   57   12    3    2    0    0    0    0    0    0    0]
     [   1   26  814    6 4449    3    2    0    0    0    0    0    0    0]
     [   1    8  719 3985    3    2    0    0    0    0    0    0    0    0]
     [   1   64   31   65   75   45    3    2    0    0    0    0    0    0]
     [   1   92    8  377    3    2    0    0    0    0    0    0    0    0]
     [   1    4   18    8  196   88    3    2    0    0    0    0    0    0]
     [   1    4   15  104 1001   98    3    2    0    0    0    0    0    0]
     [   1    4   15 1853    3    2    0    0    0    0    0    0    0    0]
     [   1   64   22   46  378    3    2    0    0    0    0    0    0    0]
```

```
 [   1   27   73   56   98    3    2    0    0    0    0    0    0    0]
 [   1    4   15 3913    3    2    0    0    0    0    0    0    0    0]
 [   1   42 1033    4   37    7    2    0    0    0    0    0    0    0]
 [   1    4   39 1677  546    3    2    0    0    0    0    0    0    0]
 [   1    8 1000  458    3    2    0    0    0    0    0    0    0    0]
 [   1    5    6  250    3    2    0    0    0    0    0    0    0    0]
 [   1   17  142   11  568    3    2    0    0    0    0    0    0    0]]
```

3. Create the custom layer

You will now create a custom layer to add the learned end token embedding to the encoder model:

```
# Run this cell to download and view a schematic diagram for the encoder model

!wget -q -O neural_translation_model.png --no-check-certificate "https://docs.google.com/uc?export=download&id=1JrtNOzUJDaOW
Image("neural_translation_model.png")
```



You should now build the custom layer.

- Using layer subclassing, create a custom layer that takes a batch of English data examples from one of the Datasets, and adds a learned embedded 'end' token to the end of each sequence.
- This layer should create a TensorFlow Variable (that will be learned during training) that is 128-dimensional (the size of the embedding space). *Hint: you may find it helpful in the call method to use the tf.tile function to replicate the end token embedding across every element in the batch.*
- Using the Dataset `.take(1)` method, extract a batch of English data examples from the training Dataset and print the shape. Test the custom layer by calling the layer on the English data batch Tensor and print the resulting Tensor shape (the layer should increase the sequence length by one).

```
# Experimental code to figure out how to add 128 element vector
a = tf.convert_to_tensor(np.ones((2, 3, 4)))
a = tf.cast(a, tf.float32)
matrix = tf.tile(tf.constant([-1]), tf.constant([4]))
matrix = tf.reshape(matrix, [1, 4])
matrix = tf.convert_to_tensor([[[-1, -1, -1, -1]], [[-1, -1, -1, -1]]])

matrix2 = tf.random.uniform(shape=(2,1,4))
print(f'matrix.shape={matrix.shape} matrix2.shape={matrix2.shape}')
print(f'a={a}')
print(f'matrix={matrix}')
tf.concat([a, matrix2], 1)
```

```
matrix.shape=(2, 1, 4) matrix2.shape=(2, 1, 4)
a=[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
```

```
   [1. 1. 1. 1.]
   [1. 1. 1. 1.]]]
 matrix=[[[-1 -1 -1 -1]]

  [[-1 -1 -1 -1]]]
 <tf.Tensor: shape=(2, 4, 4), dtype=float32, numpy=
 array([[[1.        , 1.        , 1.        , 1.        ],
         [1.        , 1.        , 1.        , 1.        ],
         [1.        , 1.        , 1.        , 1.        ],
         [0.42859316, 0.41299498, 0.33211005, 0.7381059 ]],

        [[1.        , 1.        , 1.        , 1.        ],
         [1.        , 1.        , 1.        , 1.        ],
         [1.        , 1.        , 1.        , 1.        ],
         [0.70960367, 0.453974  , 0.8199049 , 0.8455565 ]]], dtype=float32)>
```

```
class CustomLayer(Layer):

  def __init__(self, embedding_dim=128, **kwargs):
    super(CustomLayer, self).__init__(**kwargs)
    self.embedding_dim = embedding_dim
    self.end_token_embedding = tf.Variable(initial_value=tf.random.uniform(shape=(embedding_dim,)),
                                           trainable=True)

  def call(self, inputs):
    num_batches = tf.shape(inputs)[0]
    end_token = tf.tile(tf.reshape(self.end_token_embedding, shape=(1, 1, self.embedding_dim)),
                        [num_batches,1,1])
    return tf.keras.layers.concatenate([inputs, end_token], axis=1)
```

```
# Using the Dataset .take(1) method, extract a batch of English data examples from the training Dataset and print the shape
# Test the custom layer by calling the layer on the English data batch Tensor and print the resulting Tensor shape
# (the layer should increase the sequence length by one).
custom_layer = CustomLayer()
for elem in train_dataset.take(1):
  print(f'Train Shape {elem[0].shape}')
  print(f'Custom Layer Output Shape: {custom_layer(elem[0]).shape}')
```

```
    Train Shape (16, 13, 128)
    Custom Layer Output Shape: (16, 14, 128)
```

4. Build the encoder network

The encoder network follows the schematic diagram above. You should now build the RNN encoder model.

- Using the functional API, build the encoder network according to the following spec:
    - The model will take a batch of sequences of embedded English words as input, as given by the Dataset objects.
    - The next layer in the encoder will be the custom layer you created previously, to add a learned end token embedding to the end of the English sequence.
    - This is followed by a Masking layer, with the `mask_value` set to the distinct padding value you used when you padded the English sequences with the Dataset preprocessing above.
    - The final layer is an LSTM layer with 512 units, which also returns the hidden and cell states.
    - The encoder is a multi-output model. There should be two output Tensors of this model: the hidden state and cell states of the LSTM layer. The output of the LSTM layer is unused.
- Using the Dataset `.take(1)` method, extract a batch of English data examples from the training Dataset and test the encoder model by calling it on the English data Tensor, and print the shape of the resulting Tensor outputs.
- Print the model summary for the encoder network.

```
# Input expects a 3 dimension tensor, (batch size, sequence, features)
# Flexible sequence length with 128 features per time step
inputs = Input(shape=(13, 128)) # (None, None, 128) - output dimensions

# The next layer in the encoder will be the custom layer you created previously,
# to add a learned end token embedding to the end of the English sequence.
h = CustomLayer()(inputs)

# This is followed by a Masking layer, with the mask_value set to the distinct padding value you
# used when you padded the English sequences with the Dataset preprocessing above.
```

```
h = Masking(mask_value=0)(h) # (None, None, 128)

# The final layer is an LSTM layer with 512 units, which also returns the hidden and cell states.
lstm_out, hidden_state, cell_state = LSTM(512, return_state=True, name='lstm_layer')(h)

# The encoder is a multi-output model. There should be two output Tensors of this model: the hidden state and
# cell states of the LSTM layer. The output of the LSTM layer is unused.
encoder_model = Model(inputs=[inputs], outputs=[hidden_state, cell_state])
```

```
# Using the Dataset .take(1) method, extract a batch of English data examples from the training Dataset and
# test the encoder model by calling it on the English data Tensor, and print the shape of the resulting Tensor outputs.
for elem in train_dataset.take(1):
  english_data = elem[0]
  print(f'Input English training data batch shape {english_data.shape}')
  model_output = encoder_model(english_data)
  print("Resulting tensors:")
  print(f'model_output[0].shape: {model_output[0].shape} model_output[1].shape: {model_output[1].shape}')
```

```
Input English training data batch shape (16, 13, 128)
Resulting tensors:
model_output[0].shape: (16, 512) model_output[1].shape: (16, 512)
```

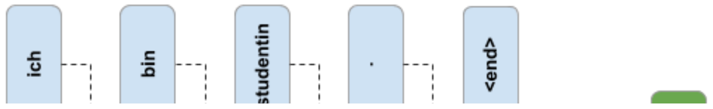
```
encoder_model.summary()
```

```
Model: "model_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_11 (InputLayer)        [(None, 13, 128)]         0
_________________________________________________________________
custom_layer_21 (CustomLayer (None, 14, 128)           128
_________________________________________________________________
masking_10 (Masking)         (None, 14, 128)           0
_________________________________________________________________
lstm_layer (LSTM)            [(None, 512), (None, 512) 1312768
=================================================================
Total params: 1,312,896
Trainable params: 1,312,896
Non-trainable params: 0
_________________________________________________________________
```

5. Build the decoder network

The decoder network follows the schematic diagram below.

```
# Run this cell to download and view a schematic diagram for the decoder model

!wget -q -O neural_translation_model.png --no-check-certificate "https://docs.google.com/uc?export=download&id=1DTeaXD8tA8R
Image("neural_translation_model.png")
```



You should now build the RNN decoder model.

- Using Model subclassing, build the decoder network according to the following spec:
    - The initializer should create the following layers:
        - An Embedding layer with vocabulary size set to the number of unique German tokens, embedding dimension 128, and set to mask zero values in the input.
        - An LSTM layer with 512 units, that returns its hidden and cell states, and also returns sequences.
        - A Dense layer with number of units equal to the number of unique German tokens, and no activation function.
    - The call method should include the usual `inputs` argument, as well as the additional keyword arguments `hidden_state` and `cell_state`. The default value for these keyword arguments should be `None`.
    - The call method should pass the inputs through the Embedding layer, and then through the LSTM layer. If the `hidden_state` and `cell_state` arguments are provided, these should be used for the initial state of the LSTM layer. *Hint: use the* `initial_state` *keyword argument when calling the LSTM layer on its input.*
    - The call method should pass the LSTM output sequence through the Dense layer, and return the resulting Tensor, along with the hidden and cell states of the LSTM layer.
- Using the Dataset `.take(1)` method, extract a batch of English and German data examples from the training Dataset. Test the decoder model by first calling the encoder model on the English data Tensor to get the hidden and cell states, and then call the decoder model on the German data Tensor and hidden and cell states, and print the shape of the resulting decoder Tensor outputs.
- Print the model summary for the decoder network.

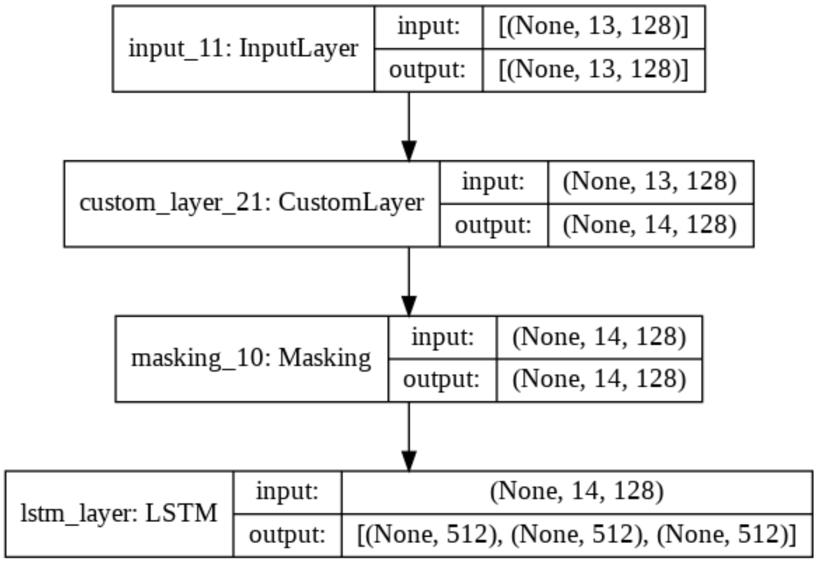
```
class DecoderModel(Model):

  def __init__(self, vocab_size, **kwargs):
    super(DecoderModel, self).__init__(**kwargs)
    self.embedding = Embedding(input_dim=vocab_size+1, output_dim=128,
                               mask_zero=True)
    self.lstm = LSTM(512, return_sequences=True, return_state=True, name='lstm_layer')
    self.dense = Dense(vocab_size+1)

  def call(self, inputs, hidden_state=None, cell_state=None):
    h = self.embedding(inputs)
    if hidden_state is not None:
      h, hidden_state_2, cell_state_2 = self.lstm(h, initial_state=(hidden_state,
                                                                    cell_state))
    else:
      h, hidden_state_2, cell_state_2 = self.lstm(h)
    h = self.dense(h)
    return h, hidden_state_2, cell_state_2

# german_vocabulary_length was computed much earlier in the Text Preprocessing section
decoder_model = DecoderModel(german_vocabulary_length)


# Using the Dataset .take(1) method, extract a batch of English and German data examples from the training Dataset.
# Test the decoder model by first calling the encoder model on the English data Tensor to get the hidden and cell states,
# and then call the decoder model on the German data Tensor and hidden and cell states, and print the shape of the resultin
# decoder Tensor outputs. Hamilton H. 2021-1-12

for elem in train_dataset.take(1):
  english_data = elem[0]
  german_data = elem[1]
  print(f'English training data batch shape {english_data.shape}')
  encoder_hidden_state, encoder_cell_state = encoder_model(english_data)
  result, decoder_hidden_state, decoder_cell_state = decoder_model(german_data,
                                                                   hidden_state=encoder_hidden_state, cell_state=encoder_ce
  print(f'result.shape={result.shape} decoder_hidden_state.shape: {decoder_hidden_state.shape}')
  print(f'decoder_cell_state.shape: {decoder_cell_state.shape}')
```

```
    English training data batch shape (16, 13, 128)
    result.shape=(16, 14, 5745) decoder_hidden_state.shape: (16, 512)
    decoder_cell_state.shape: (16, 512)
```

```
decoder_model.summary()
```

```
Model: "decoder_model_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_11 (Embedding)     multiple                  735360    
_________________________________________________________________
lstm_layer (LSTM)            multiple                  1312768   
_________________________________________________________________
dense_11 (Dense)             multiple                  2947185   
=================================================================
Total params: 4,995,313
Trainable params: 4,995,313
Non-trainable params: 0
_________________________________________________________________
```

```
from keras.utils.vis_utils import plot_model
plot_model(encoder_model, to_file='model.png', show_shapes=True)
```



6. Make a custom training loop

You should now write a custom training loop to train your custom neural translation model.

- Define a function that takes a Tensor batch of German data (as extracted from the training Dataset), and returns a tuple containing German inputs and outputs for the decoder model (refer to schematic diagram above).
- Define a function that computes the forward and backward pass for your translation model. This function should take an English input, German input and German output as arguments, and should do the following:
    - Pass the English input into the encoder, to get the hidden and cell states of the encoder LSTM.
    - These hidden and cell states are then passed into the decoder, along with the German inputs, which returns a sequence of outputs (the hidden and cell state outputs of the decoder LSTM are unused in this function).
    - The loss should then be computed between the decoder outputs and the German output function argument.
    - The function returns the loss and gradients with respect to the encoder and decoder's trainable variables.
    - Decorate the function with `@tf.function`
- Define and run a custom training loop for a number of epochs (for you to choose) that does the following:
    - Iterates through the training dataset, and creates decoder inputs and outputs from the German sequences.
    - Updates the parameters of the translation model using the gradients of the function above and an optimizer object.
    - Every epoch, compute the validation loss on a number of batches from the validation and save the epoch training and validation losses.
- Plot the learning curves for loss vs epoch for both training and validation sets.

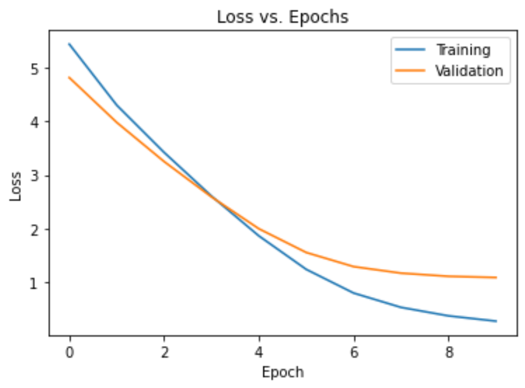
Hint: This model is computationally demanding to train. The quality of the model or length of training is not a factor in the grading rubric. However, to obtain a better model we recommend using the GPU accelerator hardware on Colab.

```
optimizer_obj = tf.keras.optimizers.Adam()
loss_obj2 = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
variables = encoder_model.trainable_variables + decoder_model.trainable_variables


def decode_german(german_input_sequence_batch):
    return german_input_sequence_batch[:, :-1], german_input_sequence_batch[:, 1:]


# Test function
for elem in train_dataset.take(1):
  print(f'german decoder input returned {elem[1]}')
  print(f'german decoder output returned {decode_german(elem[1])[1]}')

     german decoder input returned [[    1  874  434   55    3    2    0    0    0    0    0    0    0    0]
      [    1 1263   52   37   41    3    2    0    0    0    0    0    0    0]
      [    1   83    6   50 2789    7    2    0    0    0    0    0    0    0]
      [    1    5    6   71 1561    3    2    0    0    0    0    0    0    0]
      [    1   60 3963   91  181    3    2    0    0    0    0    0    0    0]
      [    1   83    6  117  256    7    2    0    0    0    0    0    0    0]
      [    1   10  229  103 1018    3    2    0    0    0    0    0    0    0]
      [    1   43    6  218    7    2    0    0    0    0    0    0    0    0]
      [    1    5   16   21  826    3    2    0    0    0    0    0    0    0]
      [    1    4  136  974    3    2    0    0    0    0    0    0    0    0]
      [    1    4  380   11    3    2    0    0    0    0    0    0    0    0]
      [    1   35    8   34  250    7    2    0    0    0    0    0    0    0]
      [    1    8 3664   44    3    2    0    0    0    0    0    0    0    0]
      [    1   32   13  488 1559    7    2    0    0    0    0    0    0    0]
      [    1  120   55    3    2    0    0    0    0    0    0    0    0    0]
      [    1    4   15   19  771  157    3    2    0    0    0    0    0    0]]
     german decoder output returned [[ 874  434   55    3    2    0    0    0    0    0    0    0    0]
      [1263   52   37   41    3    2    0    0    0    0    0    0    0]
      [  83    6   50 2789    7    2    0    0    0    0    0    0    0]
      [   5    6   71 1561    3    2    0    0    0    0    0    0    0]
      [  60 3963   91  181    3    2    0    0    0    0    0    0    0]
      [  83    6  117  256    7    2    0    0    0    0    0    0    0]
      [  10  229  103 1018    3    2    0    0    0    0    0    0    0]
      [  43    6  218    7    2    0    0    0    0    0    0    0    0]
      [   5   16   21  826    3    2    0    0    0    0    0    0    0]
      [   4  136  974    3    2    0    0    0    0    0    0    0    0]
      [   4  380   11    3    2    0    0    0    0    0    0    0    0]
      [  35    8   34  250    7    2    0    0    0    0    0    0    0]
      [   8 3664   44    3    2    0    0    0    0    0    0    0    0]
      [  32   13  488 1559    7    2    0    0    0    0    0    0    0]
      [ 120   55    3    2    0    0    0    0    0    0    0    0    0]
      [   4   15   19  771  157    3    2    0    0    0    0    0    0]]


# Define a function that computes the forward and backward pass for your translation model.
# This function should take an English input, German input and German output as arguments, and should do the following:
# The function returns the loss and gradients with respect to the encoder and decoder’s trainable variables.
# Decorate the function with @tf.function

loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
trainable_variables = encoder_model.trainable_variables + decoder_model.trainable_variables

@tf.function
def compute_forward_and_backward_pass(english_input, german_input, german_output):
  with tf.GradientTape() as tape:
    global encoder_model, decoder_model, loss_obj, trainable_variables

    # Pass the English input into the encoder, to get the hidden and cell states of the encoder LSTM.
    encoder_hidden_state, encoder_cell_state = encoder_model(english_input)

    # These hidden and cell states are then passed into the decoder, along with the German inputs, which returns a sequence
    # (the hidden and cell state outputs of the decoder LSTM are unused in this function).
    pred_german_output, decoder_hidden_state, decoder_cell_state = decoder_model(german_input,
                                                                       hidden_state=encoder_hidden_state,
                                                                       cell_state=encoder_cell_state)
    #targets = german_input[:,1:]
    #preds = pred_german_output[:,:pred_german_output.shape[1]-1,:]

    # The loss should then be computed between the decoder outputs and the German output function argument.
    loss_value = loss_obj(german_output, pred_german_output)

    grad = tape.gradient(loss_value, trainable_variables)

    return loss_value, grad, pred_german_output
```

```
# Define and run a custom training loop for a number of epochs (for you to choose) that does the following:
# Iterates through the training dataset, and creates decoder inputs and outputs from the German sequences.
# Updates the parameters of the translation model using the gradients of the function above and an optimizer object.
# Every epoch, compute the validation loss on a number of batches from the validation and save the epoch training and valid

# Keep results for plotting
train_loss_results = []
valid_loss_results = []
train_accuracy_results = []
valid_accuracy_results = []
num_epochs = 10

# Create the optimizer
optimizer_obj = tf.keras.optimizers.Adam()

for epoch in range(num_epochs):

    print("Train translation: Epoch {:03d}".format(epoch))

    train_epoch_loss_avg = tf.keras.metrics.Mean()
    valid_epoch_loss_avg = tf.keras.metrics.Mean()
    train_epoch_accuracy = tf.keras.metrics.CategoricalAccuracy()
    valid_epoch_accuracy = tf.keras.metrics.CategoricalAccuracy()

    # Training loop
    for english_x, german_y in train_dataset:

            # creates decoder inputs and outputs from the German sequences.
            # german_input, german_decoder_output = decode_german_old(german_y)
            german_input, german_output = decode_german(german_y)

            # Updates the parameters of the translation model using the gradients of the function above and an optimizer ob
            #train_loss_value, grads, targets, preds = compute_forward_and_backward_pass(english_x, german_y, german_decode
            train_loss_value, grads, preds = compute_forward_and_backward_pass(english_x,
                                                                               german_input,
                                                                               german_output)
            # Optimize the model
            optimizer_obj.apply_gradients(zip(grads, trainable_variables))

            # Every epoch, compute the validation loss on a number of batches from the validation and save the
            # epoch training and validation losses.
            train_epoch_loss_avg(train_loss_value)
            train_epoch_accuracy(to_categorical(german_output), preds)

    train_loss_results.append(train_epoch_loss_avg.result())
    train_accuracy_results.append(train_epoch_accuracy.result())
    print("Validation translation: Epoch {:03d}".format(epoch))

    # Validation loop
    for english_x, german_y in valid_dataset:
            # creates decoder inputs and outputs from the German sequences.
            german_input, german_output = decode_german(german_y)

            encoder_hidden_state, encoder_cell_state = encoder_model(english_x)
            preds, decoder_hidden_state, decoder_cell_state = decoder_model(german_input,
                                                                            hidden_state=encoder_hidden_state,
                                                                            cell_state=encoder_cell_state)
            valid_loss_value = loss_obj(german_output, preds)

            valid_epoch_loss_avg(valid_loss_value)
            valid_epoch_accuracy(to_categorical(german_output), preds)

    valid_loss_results.append(valid_epoch_loss_avg.result())
    valid_accuracy_results.append(valid_epoch_accuracy.result())

    print("Epoch {:03d}: Training Loss: {:.3f} Validation Loss {:.3f}".format(
        epoch, train_epoch_loss_avg.result(), valid_epoch_loss_avg.result()))

    #print("Epoch {:03d}: Training Loss: {:.3f} Validation Loss {:.3f} Training Accuracy: {:.3f} Validation Accuracy {:.3f}
    #    epoch, train_epoch_loss_avg.result(), valid_epoch_loss_avg.result(),
    #    train_epoch_accuracy.result(), valid_epoch_accuracy.result()))
    # End epoch
```

```
print("DONE training")

    Train translation: Epoch 000
    Validation translation: Epoch 000
    Epoch 000: Training Loss: 5.429 Validation Loss 4.808
    Train translation: Epoch 001
    Validation translation: Epoch 001
    Epoch 001: Training Loss: 4.298 Validation Loss 3.977
    Train translation: Epoch 002
    Validation translation: Epoch 002
    Epoch 002: Training Loss: 3.420 Validation Loss 3.250
    Train translation: Epoch 003
    Validation translation: Epoch 003
    Epoch 003: Training Loss: 2.609 Validation Loss 2.593
    Train translation: Epoch 004
    Validation translation: Epoch 004
    Epoch 004: Training Loss: 1.869 Validation Loss 2.002
    Train translation: Epoch 005
    Validation translation: Epoch 005
    Epoch 005: Training Loss: 1.247 Validation Loss 1.561
    Train translation: Epoch 006
    Validation translation: Epoch 006
    Epoch 006: Training Loss: 0.808 Validation Loss 1.298
    Train translation: Epoch 007
    Validation translation: Epoch 007
    Epoch 007: Training Loss: 0.542 Validation Loss 1.177
    Train translation: Epoch 008
    Validation translation: Epoch 008
    Epoch 008: Training Loss: 0.384 Validation Loss 1.119
    Train translation: Epoch 009
    Validation translation: Epoch 009
    Epoch 009: Training Loss: 0.287 Validation Loss 1.097
    DONE training
```

```
#fig = plt.figure(figsize=(15, 5))

#fig.add_subplot(121)
plt.plot(train_loss_results)
plt.plot(valid_loss_results)
plt.title('Loss vs. Epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')

#fig.add_subplot(122)
#plt.plot(train_accuracy_results)
#plt.plot(valid_accuracy_results)
#plt.title('Accuracy vs. Epochs')
#plt.ylabel('Accuracy')
#plt.xlabel('Epoch')
#plt.legend(['Training', 'Validation'], loc='upper right')

    <matplotlib.legend.Legend at 0x7f3533fc9be0>
```



7. Use the model to translate

Now it's time to put your model into practice! You should run your translation for five randomly sampled English sentences from the dataset. For each sentence, the process is as follows:

- Preprocess and embed the English sentence according to the model requirements.
- Pass the embedded sentence through the encoder to get the encoder hidden and cell states.
- Starting with the special `"<start>"` token, use this token and the final encoder hidden and cell states to get the one-step prediction from the decoder, as well as the decoder's updated hidden and cell states.
- Create a loop to get the next step prediction and updated hidden and cell states from the decoder, using the most recent hidden and cell states. Terminate the loop when the `"<end>"` token is emitted, or when the sentence has reached a maximum length.
- Decode the output token sequence into German text and print the English text and the model's German translation.

```
random_data_examples = np.random.choice(data_examples, 5)
print(f'random data examples: {random_data_examples}')
```

```
    random data examples: ['I need my key.\tIch brauche meinen Schlüssel.\tCC-BY 2.0 (France) Attribution: tatoeba.org #68
     'I have won.\tIch gewann.\tCC-BY 2.0 (France) Attribution: tatoeba.org #2245902 (CK) & #5028814 (Tamy)\n'
     'Take a bus.\tNehmen Sie einen Bus.\tCC-BY 2.0 (France) Attribution: tatoeba.org #35369 (CK) & #1313735 (al_ex_an_der
     'Do you know that?\tWisst ihr das?\tCC-BY 2.0 (France) Attribution: tatoeba.org #3738668 (CK) & #3758751 (Jens_Odo)\n
     "That'll be fun.\tDas wird ein Spaß!\tCC-BY 2.0 (France) Attribution: tatoeba.org #2251018 (CK) & #1768935 (Pfirsichb
```

```
# Get five randomly sampled English sentences from the dataset
test_english_sentences = []
test_german_sentences = []

for example in random_data_examples:
  (english_sentence, german_sentence) = preprocess_example(example)
  test_english_sentences.append(english_sentence)
  test_german_sentences.append(german_sentence)

print(f'Number of English sentences: {len(test_english_sentences)}')
print(f'Test English Sentences {test_english_sentences}')
print(f'Test German Sentences {test_german_sentences}')
print(f'German token sequence {tokenizer.texts_to_sequences(test_german_sentences)}')
```

```
    Number of English sentences: 5
    Test English Sentences ['i need my key .', 'i have won .', 'take a bus .', 'do you know that ?', "that'll be fun ."]
    Test German Sentences ['<start> ich brauche meinen schluessel . <end>', '<start> ich gewann . <end>', '<start> nehmen
    German token sequence [[1, 4, 94, 176, 204, 3, 2], [1, 4, 1265, 3, 2], [1, 216, 8, 40, 314, 3, 2], [1, 1740, 27, 11, 7
```

```
test_dataset = tf.data.Dataset.from_tensor_slices(test_english_sentences)
```

```
# Preprocess and embed the English sentence according to the model requirements.
def test_split_words(sentence):
    return tf.strings.split(sentence, sep=" ")

test_dataset = test_dataset.map(test_split_words)

for elem in test_dataset.take(5):
  print(elem)
```

```
    tf.Tensor([b'i' b'need' b'my' b'key' b'.'], shape=(5,), dtype=string)
    tf.Tensor([b'i' b'have' b'won' b'.'], shape=(4,), dtype=string)
    tf.Tensor([b'take' b'a' b'bus' b'.'], shape=(4,), dtype=string)
    tf.Tensor([b'do' b'you' b'know' b'that' b'?'], shape=(5,), dtype=string)
    tf.Tensor([b"that'll" b'be' b'fun' b'.'], shape=(4,), dtype=string)
```

```
def test_embed_english_sentence(english_sentence):
 return embedding_layer(english_sentence)

test_dataset = test_dataset.map(test_embed_english_sentence)

print(test_dataset)

for elem in test_dataset.take(2): # Takes the first element of the dataset
  print(elem.shape)
```

```
    <MapDataset shapes: (None, 128), types: tf.float32>
    (5, 128)
    (4, 128)
```

```
def test_pad_english(english_sentence):
  print(f'english_sentence={english_sentence} shape english_sentence={tf.shape(english_sentence)[0]}')
  paddings = [[0, 13 - tf.shape(english_sentence)[0]], [0, 0]]
  english_sentence = tf.pad(english_sentence, paddings, 'CONSTANT', constant_values=0)
  print(tf.convert_to_tensor(english_sentence).shape)
  # english_sentence = tf.reshape(english_sentence, [13, 128])
  return english_sentence

test_dataset = test_dataset.map(test_pad_english)

#for elem in test_dataset.take(2): # Takes the first element of the dataset
#  print(elem)
```

```
    english_sentence=Tensor("args_0:0", shape=(None, 128), dtype=float32) shape english_sentence=Tensor("strided_slice:0",
    (None, 128)
```

```
test_dataset.element_spec
```

```
    TensorSpec(shape=(None, 128), dtype=tf.float32, name=None)
```

```
test_dataset = test_dataset.batch(1)

print(test_dataset.element_spec)
```

```
    TensorSpec(shape=(None, None, 128), dtype=tf.float32, name=None)
```

```
for elem in test_dataset.take(5):
  print(elem.shape)
```

```
    (1, 13, 128)
    (1, 13, 128)
    (1, 13, 128)
    (1, 13, 128)
    (1, 13, 128)
```

```
def decipher_output(output_list):
  token_index = np.argmax(output_list)
  predicted_word = german_index_word[str(token_index)]
  return predicted_word, token_index
```

```
# Pass the embedded sentence through the encoder to get the encoder hidden and cell states.
# Starting with the special "<start>" token, use this token and the final encoder hidden and cell states to get
# the one-step prediction from the decoder, as well as the decoder's updated hidden and cell states.
# Create a loop to get the next step prediction and updated hidden and cell states from the decoder, using the
# most recent hidden and cell states. Terminate the loop when the "<end>" token is emitted, or when the sentence has reache
# maximum length. Hamilton H 2021-1-12
# Decode the output token sequence into German text and print the English text and the model's German translation.

decoder_model.get_layer('lstm_layer')

list_pair = (test_dataset.take(5), test_english_sentences)

for (english_data, english_sentence) in zip(*list_pair):
  test_encoder_hidden_state, test_encoder_cell_state = encoder_model(english_data)

  german_sentence = []
  german_tokens = []
  german_token = german_word_index['<start>']
  decoder_hidden_state = test_encoder_hidden_state
  decoder_cell_state = test_encoder_cell_state
  german_word = ''

  # print(f'About to enter while loop with german_sentence={german_sentence} german_tokens={german_tokens}')
  while german_word != '<end>':
    # result, decoder_hidden_state, decoder_cell_state = decoder_model(tf.convert_to_tensor([[german_token]]),

    result, decoder_hidden_state, decoder_cell_state = decoder_model(tf.Variable([[german_token]]),
                                                                      hidden_state=decoder_hidden_state, cell_state=decoder_
    #print(result)
    result_array = result[0][0]
    german_word, german_token = decipher_output(result_array)
    dec_output = tf.squeeze(tf.argmax(result, axis=2)).numpy()
    # print(f'german_word dec_output={tokenizer.index_word[dec_output]} german_word={german_word}, german_token={german_tok
    if german_word != '<end>':
      german_sentence.append(german_word)
      german_tokens.append(german_token)

  print(f'english sentence: {english_sentence}')
  print(f'german sentence: {" ".join(german_sentence)}')
  print(f'german tokens: {german_tokens}')
  print('')
```

```
english sentence: i need my key .
german sentence: ich brauche meine zeit .
german tokens: [4, 94, 60, 143, 3]

english sentence: i have won .
german sentence: ich habe gewonnen .
german tokens: [4, 18, 224, 3]

english sentence: take a bus .
german sentence: machen sie ein nickerchen .
german tokens: [86, 8, 19, 1489, 3]

english sentence: do you know that ?
german sentence: weisst du das ?
german tokens: [533, 13, 11, 7]

english sentence: that'll be fun .
german sentence: es wird gut sein .
german tokens: [10, 48, 49, 54, 3]
```