Capstone Project

Neural translation model

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 (File -> Download as -> PDF via LaTeX). You should 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


import numpy as np
import matplotlib.pyplot as plt
import random

from sklearn.model_selection import train_test_split

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Layer, BatchNormalization, Conv2D, Dense, Flatt
from tensorflow.keras.layers import Dropout, Softmax, concatenate
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

Flags overview image

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.

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

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

```
    Mounted at /content/gdrive
```

```
# Run this cell to load the dataset

NUM_EXAMPLES = 20000
data_examples = []
with open('/content/gdrive/MyDrive/project/data/deu.txt', 'r', encoding='utf8') as
    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.catego

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.

Model Schematic

Key: Model key

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.

```
tf.__version__
```

```
'2.8.2'
```

```
# try to understand data
print(data_examples[0:5])
```

```
['Hi.\tHallo!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) & #380
```

```
eng_sentences = []
germ_sentences = []

# for each sentence split and preprocess eng and germand sentences
for sentence in data_examples:
    seperated_sentence = re.compile('[.!?]').split(sentence)
    eng_sentences.append(preprocess_sentence(seperated_sentence[0]))
    germ_sentences.append("<start> " + preprocess_sentence(seperated_sentence[1]) +

len_eng_sentences = len(eng_sentences)
len_germ_sentences = len(germ_sentences)


tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='')
tokenizer.fit_on_texts(germ_sentences)
germ_seq = tokenizer.texts_to_sequences(germ_sentences)
vocab_size = max(tokenizer.word_index.values()) + 1
maxlen = max([len(line) for line in germ_seq])
padded_germ = pad_sequences(germ_seq, padding='post',truncating='post',maxlen=maxle
```

```
for _ in range(5):
    random_val = random.random()
    random_index = int(random_val*min(len_eng_sentences,len_germ_sentences))
    print("English: ", eng_sentences[random_index])
    print("German: ", germ_sentences[random_index])
    print("germ seq: ", padded_germ[random_index])
```

```
English:  describe tom
German:  <start> beschreibe tom <end>
germ seq:  [   1 3392    4    2    0    0    0    0    0    0    0    0    0]
English:  do you drink wine
German:  <start> trinkst du wein <end>
germ seq:  [   1 1739   10  391    2    0    0    0    0    0    0    0    0]
English:  i want some food
German:  <start> ich moechte etwas zu essen <end>
germ seq:  [  1   3 169 118  17 151   2   0   0   0   0   0   0]
English:  don't try so hard
German:  <start> streng dich nicht so an <end>
germ seq:  [   1 1496   25    9   67   33    2    0    0    0    0    0    0]
English:  i made that up
German:  <start> ich habe das erfunden <end>
germ seq:  [   1    3   15    8 2463    2    0    0    0    0    0    0    0]
```

2. Prepare the data with tf.data.Dataset objects

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 module has also been made available as a complete saved model in the folder `'./models/tf2-preview_nnlm-en-dim128_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-dim1
                                 output_shape=[128], input_shape=[], dtype=tf.strin
```

```
# Test the layer

embedding_layer(tf.constant(["these", "aren't", "the", "droids", "you're", "looking
```

```
TensorShape([7, 128])
```

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 more 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

```
X, X_val, y, y_val = train_test_split(eng_sentences, padded_germ, test_size=0.2)


dataset = tf.data.Dataset.from_tensor_slices((X, y))
val_dataset = tf.data.Dataset.from_tensor_slices((X_val, y_val))


def split_sentences(dataset, y):
    return (tf.strings.split(dataset, sep=' '), y)


dataset = dataset.map(split_sentences)
val_dataset = val_dataset.map(split_sentences)


def embed_sentences(dataset, y):
    return (embedding_layer(dataset), y)


dataset = dataset.map(embed_sentences)
val_dataset = val_dataset.map(embed_sentences)


def len_filter(dataset,y):
  return tf.shape(dataset)[0]<=13


dataset = dataset.filter(len_filter)
val_dataset = val_dataset.filter(len_filter)


def pad_sentences(dataset,y):

  pad_value = tf.constant([[13,0], [0,0]])
  dataset_padded = tf.pad(dataset, pad_value)
  return (dataset_padded[-13:,:], y)


dataset = dataset.map(pad_sentences)
val_dataset = val_dataset.map(pad_sentences)


dataset = dataset.batch(16)
val_dataset = val_dataset.batch(16)


print(dataset.element_spec)
```

```
print(val_dataset.element_spec)
```

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

```
for english_train,german_train in dataset.take(1):
    print("Shape of the English data example: ", english_train.shape)

for english_val,german_val in val_dataset.take(1):
    print("German data example Tensor: ",german_val)
```

```
Shape of the English data example:  (16, 13, 128)
German data example Tensor:  tf.Tensor(
[[   1    4  936  248    2    0    0    0    0    0    0    0    0]
 [   1    4  994  143    2    0    0    0    0    0    0    0    0]
 [   1    8 3836    3    9    2    0    0    0    0    0    0    0]
 [   1    5    4  186    2    0    0    0    0    0    0    0    0]
 [   1  263    7    9   22    4    2    0    0    0    0    0    0]
 [   1    3   27    7   18    9 1211    2    0    0    0    0    0]
 [   1    3  176  819    2    0    0    0    0    0    0    0    0]
 [   1   11    5 1236  158  334   17   42    2    0    0    0    0]
 [   1    7    5   96   30    2    0    0    0    0    0    0    0]
 [   1    7    5  344    2    0    0    0    0    0    0    0    0]
 [   1    3   15   19  698    2    0    0    0    0    0    0    0]
 [   1   75 3143  987    2    0    0    0    0    0    0    0    0]
 [   1    4  327   41    2    0    0    0    0    0    0    0    0]
 [   1    5    4   62   72 2110    2    0    0    0    0    0    0]
 [   1   14   78 1259    2    0    0    0    0    0    0    0    0]
 [   1   13   11  197    2    0    0    0    0    0    0    0    0]], shape=(1
```

3. Create the custom layer

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

Encoder schematic

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).

```
class MyCustomLayer(Layer):
  def __init__(self, embedding_dim=128, **kwargs):
    super(MyCustomLayer, self).__init__(**kwargs)
    self.token_embed = tf.Variable(initial_value=tf.random.uniform(shape=(embedding

  def call(self, inputs):
    token_alter = tf.reshape(self.token_embed, shape=(1, 1, self.token_embed.shape[
    end_token = tf.tile(token_alter, [tf.shape(inputs)[0],1,1])
    concat = tf.keras.layers.concatenate([inputs, end_token], axis=1)
    return concat


layer = MyCustomLayer()
for english_train,german_train in dataset.take(1):
    print("Shape of the English data example: ", english_train.shape)
    layered_result = layer(english_train)
    print('Shape of the English data example after layer: ', layered_result.shape)
```

```
Shape of the English data example:  (16, 13, 128)
Shape of the English data example after layer:  (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.

```
batch_shape=(None,13,128)
inputs = Input(batch_shape=batch_shape)
h = layer(inputs)
h = Masking(mask_value = -1)(h)
h, hidden, cell = LSTM(512, return_state=True)(h)
model = Model(inputs=inputs, outputs=[hidden, cell])


for english_train,german_train in dataset.take(1):
  hidden, cell=model(english_train)
  print('hidden state: ',hidden.shape)
  print('cell state: ',cell.shape)
```

```
hidden state:  (16, 512)
cell state:  (16, 512)
```

```
print(model.summary())
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 13, 128)]         0         
                                                                 
 my_custom_layer (MyCustomLa  (None, 14, 128)          128       
 yer)                                                            
                                                                 
 masking (Masking)           (None, 14, 128)           0         
                                                                 
 lstm (LSTM)                 [(None, 512),             1312768   
                              (None, 512),                       
                              (None, 512)]                       
                                                                 
=================================================================
Total params: 1,312,896
Trainable params: 1,312,896
Non-trainable params: 0
_________________________________________________________________
None
```

5. Build the decoder network

The decoder network follows the schematic diagram below.

Decoder schematic

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, **kwargs):
        super(DecoderModel, self).__init__(**kwargs)
        self.emb_layer_1= Embedding(input_dim=vocab_size,output_dim= 128,mask_zero=
        self.lstm_1 = LSTM(512, return_sequences=True, return_state=True)
        self.dense_1 = Dense(vocab_size)

    def call(self, inputs, hidden_state=None, cell_state=None):
        h = self.emb_layer_1(inputs)

        h, hidden, cell = self.lstm_1(h, initial_state=[hidden_state, cell_state])

        output = self.dense_1(h)
        return output, hidden, cell


decoder_model = DecoderModel()
for english_train,german_train in dataset.take(1):
  hidden, cell=model(english_train)
  output, hidden_s, cell_s = decoder_model(german_train, hidden, cell)
  print('output: ',output.shape)
  print('hidden state: ',hidden_s.shape)
  print('cell state: ',cell_s.shape)
```

```
output:  (16, 13, 5746)
hidden state:  (16, 512)
cell state:  (16, 512)
```

```
decoder_model.summary()
```

```
Model: "decoder_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       multiple                  735488

 lstm_1 (LSTM)               multiple                  1312768

 dense (Dense)               multiple                  2947698

=================================================================
Total params: 4,995,954
Trainable params: 4,995,954
Non-trainable params: 0
_________________________________________________________________
```

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.

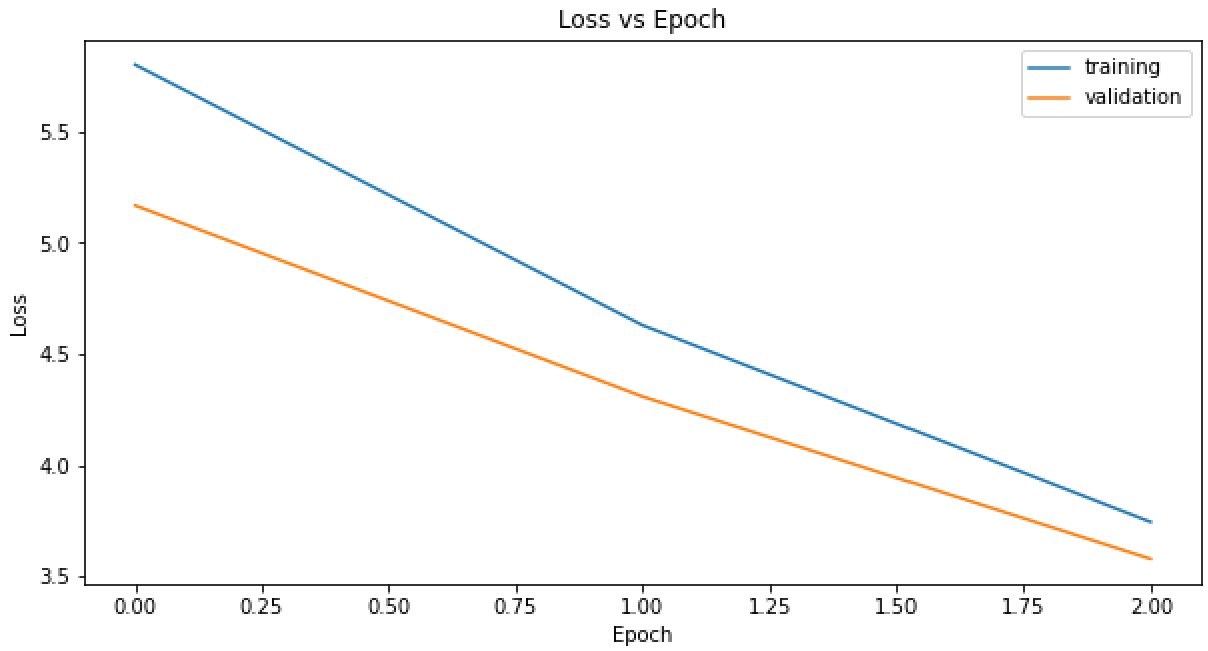
    - Updates the parameters of the translation model using the gradients of the function above and an optimizer object.
    - Every epoch, compute the validation loss on a number of batches from the validation and save the epoch training and validation losses.
- Plot the learning curves for loss vs epoch 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.

```
def create_inputs_and_outputs(german):

    inputs = german[:, :-1]
    outputs = german[:, 1:]

    return (inputs, outputs)


@tf.function
def grad(english, german_in, german_out):
  with tf.GradientTape() as tape:
    hidden,cell = model(english)
    decoder_output, hidden_s, cell_s = decoder_model(german_in, hidden, cell)
    loss = tf.keras.losses.sparse_categorical_crossentropy(german_out,decoder_outpu
    grad_encoder,grad_decoder = tape.gradient(loss, [model.trainable_variables,deco
    return (loss,grad_encoder,grad_decoder)



optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
trainable_variables = model.trainable_variables + decoder_model.trainable_variables
num_epochs = 3
epoch_loss_train = []
epoch_loss_val = []

for epoch in range(num_epochs):
  train_loss_obj = tf.keras.metrics.Mean()
  val_loss_obj = tf.keras.metrics.Mean()

  for english, german in dataset:
    german_in, german_out = create_inputs_and_outputs(german)
    loss_value,grad_encoder,grad_decoder = grad(english,german_in, german_out)
    optimizer.apply_gradients(zip(grad_encoder, model.trainable_variables))
    optimizer.apply_gradients(zip(grad_decoder, decoder_model.trainable_variables))
    train_loss_obj(loss_value)
  epoch_loss_train.append(train_loss_obj.result())

  for val_eng, val_germ in val_dataset:
    german_in, german_out = create_inputs_and_outputs(val_germ)
    hidden, cell = model(val_eng)
    decoder_output,hidden_s,cell_s = decoder_model(german_in,hidden,cell)
    loss_value =tf.keras.losses.sparse_categorical_crossentropy(german_out,decoder_
```

```
    val_loss_obj(loss_value)
  epoch_loss_val.append(val_loss_obj.result())
  #print("Epoch: "+str(epoch)+" loss: "+str(train_loss_obj.result())+" val loss: "+
```

```
plt.figure(figsize=(10,5))
plt.plot(epoch_loss_train)
plt.plot(epoch_loss_val)
plt.title('Loss vs Epoch')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['training', 'validation'])
plt.show()
```



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.

```
randomIndices = random.choices(range(len(eng_sentences)-1),k=5)

eng_samples=[]
germ_samples=[]

for index in randomIndices:
  eng_samples.append(eng_sentences[index])
  germ_samples.append(germ_sentences[index])



sample_num=1
for index, sentence in enumerate(eng_samples):

  print('Sample: ',sample_num)
  print('English Sentence: ', sentence)

  english = tf.strings.split(sentence, sep=' ')

  embedding_out = embedding_layer(english)

  pad_value = tf.constant([[13,0], [0,0]])
  english_padded = tf.pad(embedding_out, pad_value)
  english_padded_out = english_padded[-13:,:]
  embedding_exp = tf.expand_dims(english_padded_out, 0)

  hidden, cell = model(embedding_exp)
  decoder_input = tf.Variable([[tokenizer.word_index['<start>']]])
  end = tokenizer.word_index['<end>']

  translate = []
  for j in range(dataset.element_spec[1].shape[1]):

    decoder_output, hidden, cell = decoder_model(decoder_input, hidden, cell)
    decoder_output = tf.squeeze(tf.argmax(decoder_output, axis=2)).numpy()
    if decoder_output == end:
      break

    translate.append(tokenizer.index_word[decoder_output])

    decoder_input = tf.Variable([[decoder_output]])

  translate = ' '.join(translate)
  sample_num+=1
  print("English: ", sentence)
  print("German text", germ_samples[index])
  print("German translated: ", translate)

# could not handle this one - get stucked
```

```
Sample:  1
English Sentence:  read it to me
English:  read it to me
German text <start> lesen sie es mir vor <end>
German translated:  ich bin nicht beschaeftigt
Sample:  2
English Sentence:  keep the dog out
English:  keep the dog out
German text <start> lassen sie den hund nicht rein <end>
German translated:  ich bin nicht beschaeftigt
Sample:  3
English Sentence:  i'm inside
English:  i'm inside
German text <start> ich bin drinnen <end>
German translated:  ich bin nicht beschaeftigt
Sample:  4
English Sentence:  you'll make it
English:  you'll make it
German text <start> du wirds das schaffen <end>
German translated:  ich bin nicht beschaeftigt
Sample:  5
English Sentence:  don't deceive me
English:  don't deceive me
German text <start> taeuscht mich nicht <end>
German translated:  ich bin nicht beschaeftigt
```