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 (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 numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
```

```
tf.__version__
```

```
'2.4.1'
```

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_remount=True).
```

```
# Run this cell to load the dataset

NUM_EXAMPLES = 100000
data_examples = []
with open('gdrive/MyDrive/Colab Notebooks/Tensorflow Online Course/Customising your models with Tensorflow 2/Week 5/deu.txt', 'r', encoding='ut
    for line in f.readlines():
        if len(data_examples) < NUM_EXAMPLES:
            data_examples.append(line)
        else:
            break
```

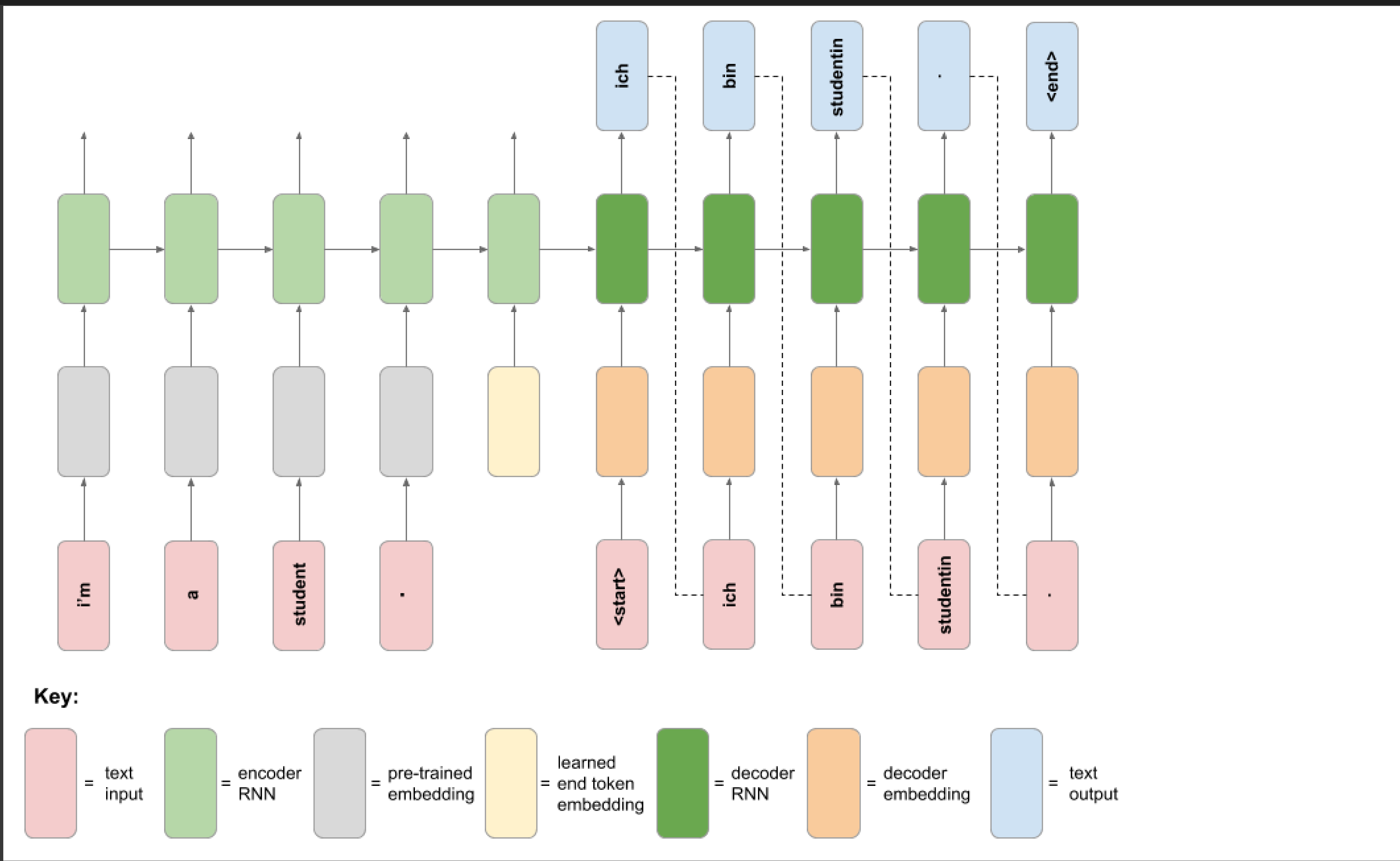
```
# These functions preprocess English and German sentences

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-RbYNilJ9jcscNZvsSPm
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']
```

```
english_sentences = [sentence.split("\t")[0] for sentence in data_examples]
```

```
german_sentences = [sentence.split("\t")[1] for sentence in data_examples]
```

```
english_sentences = list(map(preprocess_sentence, english_sentences))
german_sentences = list(map(preprocess_sentence, german_sentences))
```

```
final_german_sentences = ["<start>" + " " + sentence + " " + "<end>" for sentence in german_sentences]
```

```
from tensorflow.keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(filters='')
```

```
tokenizer.fit_on_texts(final_german_sentences)
```

```
tokenized_german_sentences = tokenizer.texts_to_sequences(final_german_sentences)
```

```
#Show some random (inputs, outputs) pairs

random_idx = np.random.choice(len(data_examples), 5, replace=False)

for idx in random_idx:
  print(f"English sentence: {english_sentences[idx]}")
  print(f"German sentence: {final_german_sentences[idx]}")
  print(f"Tokenized german sentence: {tokenized_german_sentences[idx]}")
```

```
English sentence: i'll take it home with me .
German sentence: <start> ich nehme das mit nach hause . <end>
Tokenized german sentence: [1, 4, 493, 11, 39, 62, 96, 3, 2]
English sentence: it was real hard work .
German sentence: <start> es war echt harte arbeit . <end>
Tokenized german sentence: [1, 13, 29, 582, 3484, 134, 3, 2]
English sentence: behave yourself .
German sentence: <start> benehmt euch . <end>
Tokenized german sentence: [1, 7467, 66, 3, 2]
English sentence: i study art history .
German sentence: <start> ich studiere kunstgeschichte . <end>
Tokenized german sentence: [1, 4, 1666, 11278, 3, 2]
English sentence: i was happy to do it , tom .
German sentence: <start> das habe ich gerne gemacht , tom . <end>
Tokenized german sentence: [1, 11, 21, 4, 102, 118, 8, 5, 3, 2]
```

```
padded_german_sentences = np.array(pad_sequences(tokenized_german_sentences, padding='post'))
```

```
padded_german_sentences
```

```
array([[   1,  929,   16, ...,    0,    0,    0],
       [   1, 3593, 1068, ...,    0,    0,    0],
       [   1, 2849,   16, ...,    0,    0,    0],
       ...,
       [   1,    4,  221, ...,    0,    0,    0],
       [   1,    4,  221, ...,    0,    0,    0],
       [   1,    4, 4936, ...,    0,    0,    0]], dtype=int32)
```

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)
```

```
# Test the layer

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

```
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 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.

```
x_train, x_val, y_train, y_val = train_test_split(np.array(english_sentences), padded_german_sentences, test_size=0.2)
```

```
len(x_train)
```

```
80000
```

```
training_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
```

```
validation_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
```

```
def split_english_sentence(english_sentence, tokenized_german_sentence):
  return tf.strings.split(english_sentence, sep =' '), tokenized_german_sentence
```

```
training_dataset = training_dataset.map(split_english_sentence)
validation_dataset = validation_dataset.map(split_english_sentence)
```

```
def embed_english_sentence(english_sentence, tokenized_german_sentence):
  return embedding_layer(english_sentence), tokenized_german_sentence
```

```
training_dataset = training_dataset.map(embed_english_sentence)
validation_dataset = validation_dataset.map(embed_english_sentence)
```

```
def filter_long_english_sentence(english_sentence, tokenized_german_sentence):
  return tf.shape(english_sentence)[0] <= 13
```

```
for idx,(input,output) in enumerate(validation_dataset.as_numpy_iterator()):
  if idx > 300:
    break
  if input.shape[0] > 13:
    print(input.shape)
    print(output.shape)
```

```
training_dataset = training_dataset.filter(filter_long_english_sentence)
```

```
training_dataset = training_dataset.filter(filter_long_english_sentence)
validation_dataset = validation_dataset.filter(filter_long_english_sentence)
```

```
#Check that after filtering there's no more input sequences longer than 13 words

for idx,(input,output) in enumerate(validation_dataset.as_numpy_iterator()):
  if idx > 300:
    break
  if input.shape[0] > 13:
    print(input.shape)
    print(output.shape)
```

```
def pad_english_sentence(english_sentence, tokenized_german_sentence):
  n_zeros_to_add = tf.math.subtract(13, tf.shape(english_sentence)[0])
  paddings = [[n_zeros_to_add,0], [0, 0]]
  english_sentence = tf.pad(english_sentence, paddings, "CONSTANT")
  return english_sentence, tokenized_german_sentence
```

```
training_dataset = training_dataset.map(pad_english_sentence)
validation_dataset = validation_dataset.map(pad_english_sentence)
```

```
for idx,(input,output) in enumerate(validation_dataset.as_numpy_iterator()):
  if idx > 1:
    break
  print(input.shape)
  print(output.shape)
  print(input)
```

```
(13, 128)
(23,)
[[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 ...
 [ 0.2000517   0.0272701  -0.03822284 ...  0.1073221  -0.01488957
  -0.01846376]
 [-0.07972887 -0.00440909 -0.11799385 ... -0.2464662   0.16568878
  -0.08060557]
 [-0.02885591  0.11113551  0.13474926 ...  0.09192991  0.1173007
  -0.01730555]]
(13, 128)
(23,)
[[ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 [ 0.          0.          0.         ...  0.          0.
   0.        ]
 ...
 [ 0.0834387  -0.01444498  0.0120562  ... -0.05738941  0.18851876
   0.09680488]
 [ 0.0765123   0.1190913  -0.1376916  ...  0.10586941  0.22319877
   0.01715502]
 [ 0.012986    0.08981702  0.16017003 ...  0.06796802  0.13528903
  -0.022035  ]]
```

```
training_dataset = training_dataset.batch(16)
validation_dataset = validation_dataset.batch(16)
```

```
print(training_dataset.element_spec)
print(validation_dataset.element_spec)
```

```
(TensorSpec(shape=(None, None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(None, 23), dtype=tf.int32, name=None))
(TensorSpec(shape=(None, None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(None, 23), dtype=tf.int32, name=None))
```

```
english_example, _ = next(iter(training_dataset.take(1)))
print(english_example.shape)
```

```
(16, 13, 128)
```

```
_, german_example = next(iter(validation_dataset.take(1)))
print(german_example)
```

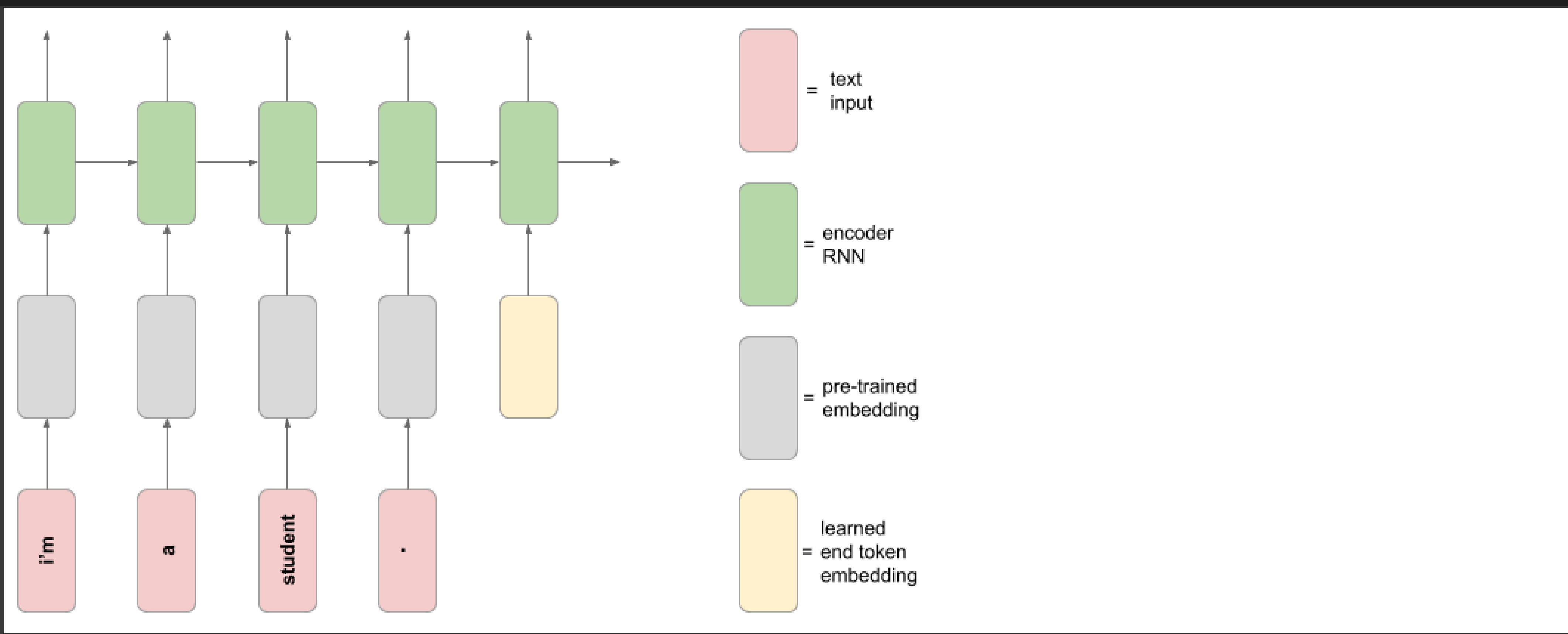
```
tf.Tensor(
[[   1  246   87   16  199   90 1329   16    2    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1   19  495   41 6859  287    3    2    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
```

```
 [   1  116  981    7 5182    3    2    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    4   21   41 7783    3    2    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    4  378    8    4  180  419  174    3    2    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    5    7 5265    3    2    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    5   83  131 2032 4350    3    2    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    4 4016  754   48  644 2104    3    2    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1   13  193   79   17  765    3    2    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1   58 4708  472  347    9   89   33    3    2    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1  143   15   46  906   17  111    3    2    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    4  192    5   42    9    3    2    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    4   65  201   39    5  174    3    2    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1    4   26   22    9  207    8  446    4   44 6761   65    3    2
     0    0    0    0    0    0    0    0    0]
 [   1   11  226    9 1968   80    3    2    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]
 [   1   13   52  267    3    2    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0]], shape=(16, 23), dtype=int32)
```

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=1JrtNOzUJDaOWrK4C-xv-4wUuZaI12sQ
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).

```
from tensorflow.keras.layers import Layer
```

```
class EmbeddingLayer(Layer):
  def __init__(self, **kwargs):
    super(EmbeddingLayer, self).__init__(**kwargs)

  def build(self, input_shape):
    self.end_token_embedding = tf.Variable(initial_value=tf.random.uniform(shape=(128,)), trainable=True)
```

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

```
english_example, _ = next(iter(training_dataset.take(1)))
print(english_example.shape)
```

```
(16, 13, 128)
```

```
embed_layer = EmbeddingLayer()
english_example = embed_layer(english_example)
print(english_example.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.

```
from tensorflow.keras.layers import Input, LSTM, Masking
from tensorflow.keras.models import Model
```

```
#DON'T USE 'input' AS VARIABLE NAME but 'inputs'

def encoder(input_shape):
  inputs = Input(shape=input_shape)
  my_layer = EmbeddingLayer()
  x = my_layer(inputs)
  x = Masking(mask_value=0.0)(x)
  _ , state_h, state_c = LSTM(512, return_state=True)(x)
  encoder_model = tf.keras.models.Model(inputs=inputs, outputs= [state_h, state_c])
  return encoder_model
```

```
my_encoder = encoder((13,128))
```

```
english_example, _ = next(iter(training_dataset.take(1)))
encoded_english_example = my_encoder(english_example)
print(encoded_english_example[0].shape)
print(encoded_english_example[1].shape)
```

```
(16, 512)
(16, 512)
```

```
print(my_encoder.summary())
```

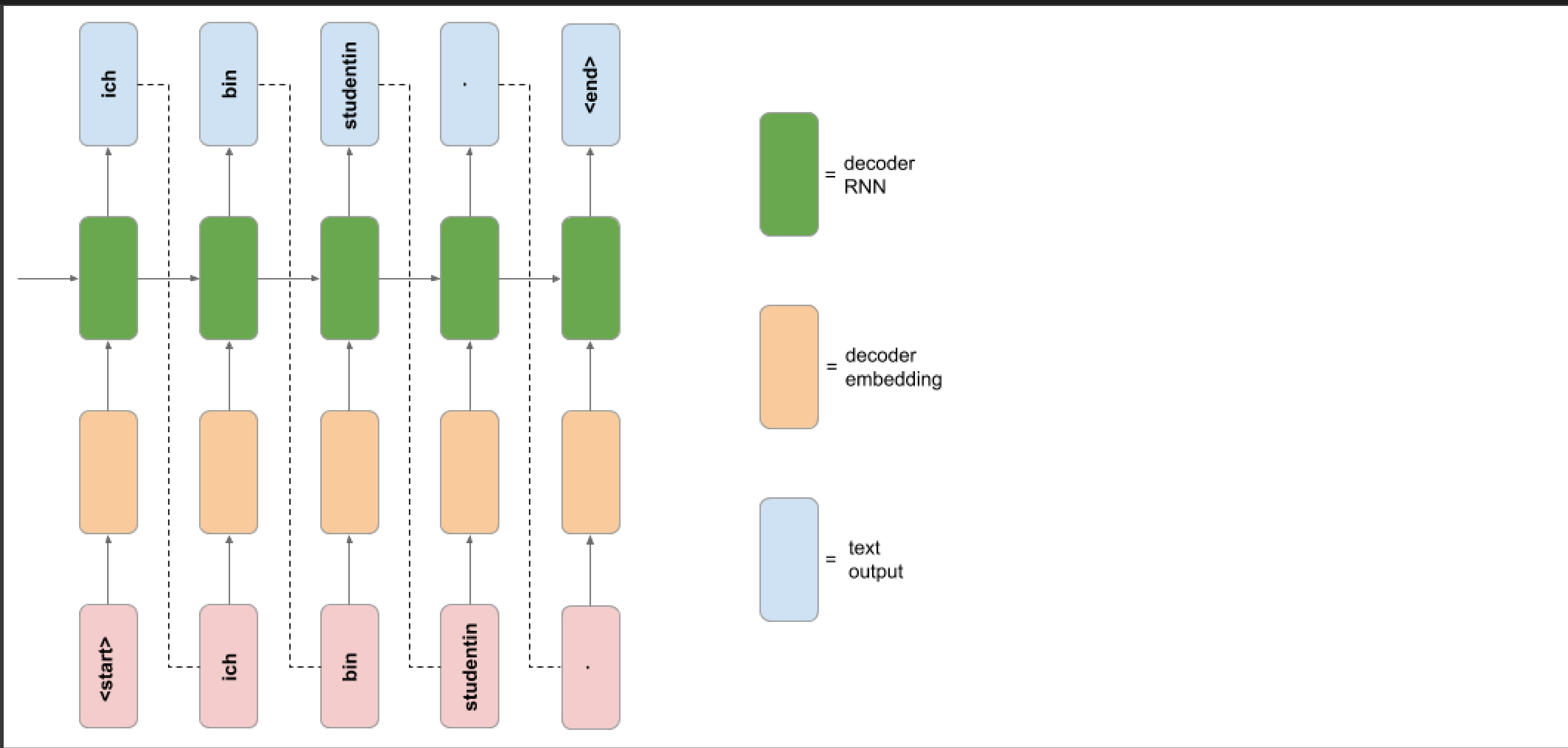
```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 13, 128)]         0
_________________________________________________________________
embedding_layer_1 (Embedding (None, 14, 128)           128
_________________________________________________________________
masking (Masking)            (None, 14, 128)           0
_________________________________________________________________
lstm (LSTM)                  [(None, 512), (None, 512) 1312768
```

```
=================================================================
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.

```
# 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=1DTeaXD8tA8RjkpVrB2mr9csSBOY4LQi
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.

```
from tensorflow.keras.layers import Embedding, Dense
```

```
class Decoder(Model):
  def __init__(self, vocabulary_size, **kwargs):
    super(Decoder, self).__init__(**kwargs)
    self.embedding_layer = Embedding(input_dim=vocabulary_size, output_dim=128,mask_zero=True)
    self.lstm = LSTM(512, return_sequences=True, return_state=True)
    self.dense = Dense(vocabulary_size)

  def call(self, inputs, hidden_state=None, cell_state=None):
    emb_inputs = self.embedding_layer(inputs)
```

```
    output_sequence, state_h, state_c = self.lstm(emb_inputs, initial_state=[hidden_state, cell_state])
    output_dense = self.dense(output_sequence)
    return output_dense, state_h, state_c
```

```
english_example, german_example = next(iter(training_dataset.take(1)))
```

```
my_encoder = encoder((13,128))
state_h, state_c = my_encoder(english_example)
```

```
max_index_value_german = max([max(sublist) for sublist in tokenized_german_sentences])
max_index_value_german
```

```
17005
```

```
my_decoder = Decoder(vocabulary_size=max_index_value_german+1)
output_decoder, state_h, state_c = my_decoder(german_example, state_h, state_c)
```

```
print(output_decoder.shape)
```

```
(16, 23, 17006)
```

```
print(my_decoder.summary())
```

```
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        multiple                  2176768
_________________________________________________________________
lstm_2 (LSTM)                multiple                  1312768
_________________________________________________________________
dense (Dense)                multiple                  8724078
=================================================================
Total params: 12,213,614
Trainable params: 12,213,614
Non-trainable params: 0
_________________________________________________________________
None
```

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.

```
my_encoder = encoder((13,128))
english_example, german_example = next(iter(training_dataset.take(1)))
state_h, state_c = my_encoder(english_example)

max_index_value_german = max([max(sublist) for sublist in tokenized_german_sentences])
print(max_index_value_german)
my_decoder = Decoder(vocabulary_size=max_index_value_german+1)
output_decoder, state_h, state_c = my_decoder(german_example, state_h, state_c)
```

```
17005
```

```
print(my_decoder.summary())
```

```
Model: "decoder_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      multiple                  2176768
_________________________________________________________________
lstm_4 (LSTM)                multiple                  1312768
_________________________________________________________________
dense_1 (Dense)              multiple                  8724078
=================================================================
Total params: 12,213,614
Trainable params: 12,213,614
Non-trainable params: 0
_________________________________________________________________
None
```

```
def in_out_decoder(batch_german_data):
  in_decoder = batch_german_data[:,:-1]
  target_decoder = batch_german_data[:,1:]
  return (in_decoder, target_decoder)
```

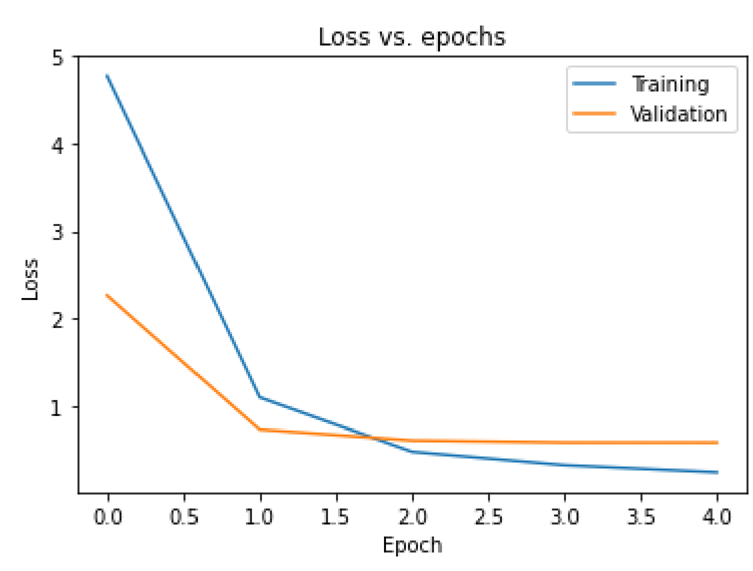
```
@tf.function
def forward_backward_pass(encoder_input, decoder_input, decoder_output_target):
   with tf.GradientTape() as tape:
    state_hidden_encoder, state_cell_encoder = my_encoder(encoder_input)
    output_decoder, _ , _ = my_decoder(decoder_input, state_hidden_encoder, state_cell_encoder)
    loss = tf.keras.losses.sparse_categorical_crossentropy(decoder_output_target, output_decoder, from_logits=True)

   trainable_variables = my_encoder.trainable_variables + my_decoder.trainable_variables
   gradients = tape.gradient(loss, trainable_variables)


   return loss, gradients
```

```
train_loss_results = []
val_loss_results = []

n_epochs = 5

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)




for epoch in range(n_epochs):
  iter = 0
  epoch_loss_avg = tf.keras.metrics.Mean()
  epoch_loss_val_avg = tf.keras.metrics.Mean()
  #Training loop
  for english,german in training_dataset:
    in_decoder, target_decoder = in_out_decoder(german)
    loss_value, grads = forward_backward_pass(english, in_decoder, target_decoder)
    trainable_variables = my_encoder.trainable_variables + my_decoder.trainable_variables
    optimizer.apply_gradients(zip(grads, trainable_variables))

    #Compute current loss average across the batch
    epoch_loss_avg(loss_value)
    if iter % 1000 == 0:
      print(f'Iter: {iter}, loss: {epoch_loss_avg.result()}')
    iter += 1

  train_loss_results.append(epoch_loss_avg.result())

  #Validation loop
  for english,german in validation_dataset:
    in_decoder, target_decoder = in_out_decoder(german)
    loss_value, _ = forward_backward_pass(english, in_decoder, target_decoder)

    #Compute current loss average across the batch
    epoch_loss_val_avg(loss_value)

  val_loss_results.append(epoch_loss_val_avg.result())
```

```
print(f"Epoch: {epoch}, Loss: {epoch_loss_avg.result()}, Val loss: {epoch_loss_val_avg.result()}")
```

```
Iter: 0, loss: 9.741032600402832
Iter: 1000, loss: 7.0906548500061035
Iter: 2000, loss: 6.424267292022705
Iter: 3000, loss: 5.837645530700684
Iter: 4000, loss: 5.291168689727783
Epoch: 0, Loss: 4.77327823638916, Val loss: 2.2612156867980957
Iter: 0, loss: 2.2447988986968994
Iter: 1000, loss: 1.8540246486663818
Iter: 2000, loss: 1.5613899230957031
Iter: 3000, loss: 1.3522875308990479
Iter: 4000, loss: 1.203959345817566
Epoch: 1, Loss: 1.0940756797790527, Val loss: 0.7248156070709229
Iter: 0, loss: 0.6114760637283325
Iter: 1000, loss: 0.569258451461792
Iter: 2000, loss: 0.5380813479423523
Iter: 3000, loss: 0.5122277140617371
Iter: 4000, loss: 0.49101582169532776
Epoch: 2, Loss: 0.4723661243915558, Val loss: 0.6006723642349243
Iter: 0, loss: 0.3827853500843048
Iter: 1000, loss: 0.36261627078056335
Iter: 2000, loss: 0.3496600389480591
Iter: 3000, loss: 0.338462769985199
Iter: 4000, loss: 0.3288933038711548
Epoch: 3, Loss: 0.3198167383670807, Val loss: 0.5787403583526611
Iter: 0, loss: 0.2784232795238495
Iter: 1000, loss: 0.26151612401008606
Iter: 2000, loss: 0.254747211933136
Iter: 3000, loss: 0.24884848296642303
Iter: 4000, loss: 0.2434561401605606
Epoch: 4, Loss: 0.23812176287174225, Val loss: 0.5779154300689697
```

```
import matplotlib.pyplot as plt

plt.figure()
plt.plot(train_loss_results)
plt.plot(val_loss_results)
plt.plot()
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
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.

```python
import numpy as np
```

```python
random_idx = np.random.choice(len(data_examples), 5, replace=False)

for idx in random_idx:
  english_sentence = english_sentences[idx]
  english_sentence = tf.strings.split(english_sentence, sep =' ')
  embedded_english_sentence = embedding_layer(english_sentence)
  n_zeros_to_add = tf.math.subtract(13, tf.shape(embedded_english_sentence)[0])
  paddings = [[n_zeros_to_add,0], [0, 0]]
  embedded_english_sentence = tf.pad(embedded_english_sentence, paddings, "CONSTANT")
  embedded_english_sentence = tf.expand_dims(embedded_english_sentence, axis=0)

  state_hidden_encoder, state_cell_encoder = my_encoder(embedded_english_sentence)

  start_token_decoder = tf.constant(tokenizer.texts_to_sequences(["<start>"]))
  input_decoder = start_token_decoder

  predictions = ""
  predicted_word = ""

  hidden_h_decoder, hidden_c_decoder = state_hidden_encoder, state_cell_encoder
  size_output_sentence = 0

  while True:
    output_decoder, hidden_h_decoder, hidden_c_decoder = my_decoder(input_decoder, hidden_h_decoder, hidden_c_decoder)
    predicted_output_token = np.argmax(output_decoder)
    predicted_word = tokenizer.sequences_to_texts([[predicted_output_token]])
    if (predicted_word == ['<end>']) or (size_output_sentence > 20):
      break

    size_output_sentence += 1
    predictions += " " + predicted_word[0]
    input_decoder = tf.constant(tokenizer.texts_to_sequences(predicted_word))


  print(f"English sentence: {english_sentences[idx]}")
  print(f"Target German sentence: {german_sentences[idx]}")
  print(f"Predicted German sentence: {predictions}")
```

```
English sentence: let's not overreact .
Target German sentence: wir sollten nicht ueberreagieren .
Predicted German sentence:  lasst uns nicht zaehlen .
English sentence: you're all racists .
Target German sentence: ihr seid alle rassisten .
Predicted German sentence:  ihr seid alle rassisten .
English sentence: i'm angry .
Target German sentence: ich bin sauer .
Predicted German sentence:  ich bin wuetend .
English sentence: i was in a bad mood .
Target German sentence: ich war schlechter laune .
Predicted German sentence:  ich war in einem ausweglosen zustand .
English sentence: tom has no manners at all .
Target German sentence: tom hat ueberhaupt keine manieren .
Predicted German sentence:  tom hat keine kraft ueber sich .
```