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.

In [1]:
```
from google.colab import drive
drive.mount('gdrive')
directory = "gdrive/My Drive/Colab Notebooks/Curso tensorflow2/caps
tone Customise your model"
```

```
Mounted at gdrive
```

In [2]:
```
import tensorflow as tf
import tensorflow_hub as hub
import unicodedata
import re

from tensorflow.keras.preprocessing.text import Tokenizer
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Layer, Input, Masking, LSTM, Em
bedding, Dense

from tensorflow.keras.models import Model
import matplotlib.pyplot as plt
```

In [3]:
```
tf.__version__
```

Out[3]:
```
'2.7.0'
```

In [5]:
```
print("GPU name: {}".format(tf.test.gpu_device_name()))
```

```
GPU name: /device:GPU:0
```

For the capstone project, you will use a language dataset from http://www.manythings.org/anki/ (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.

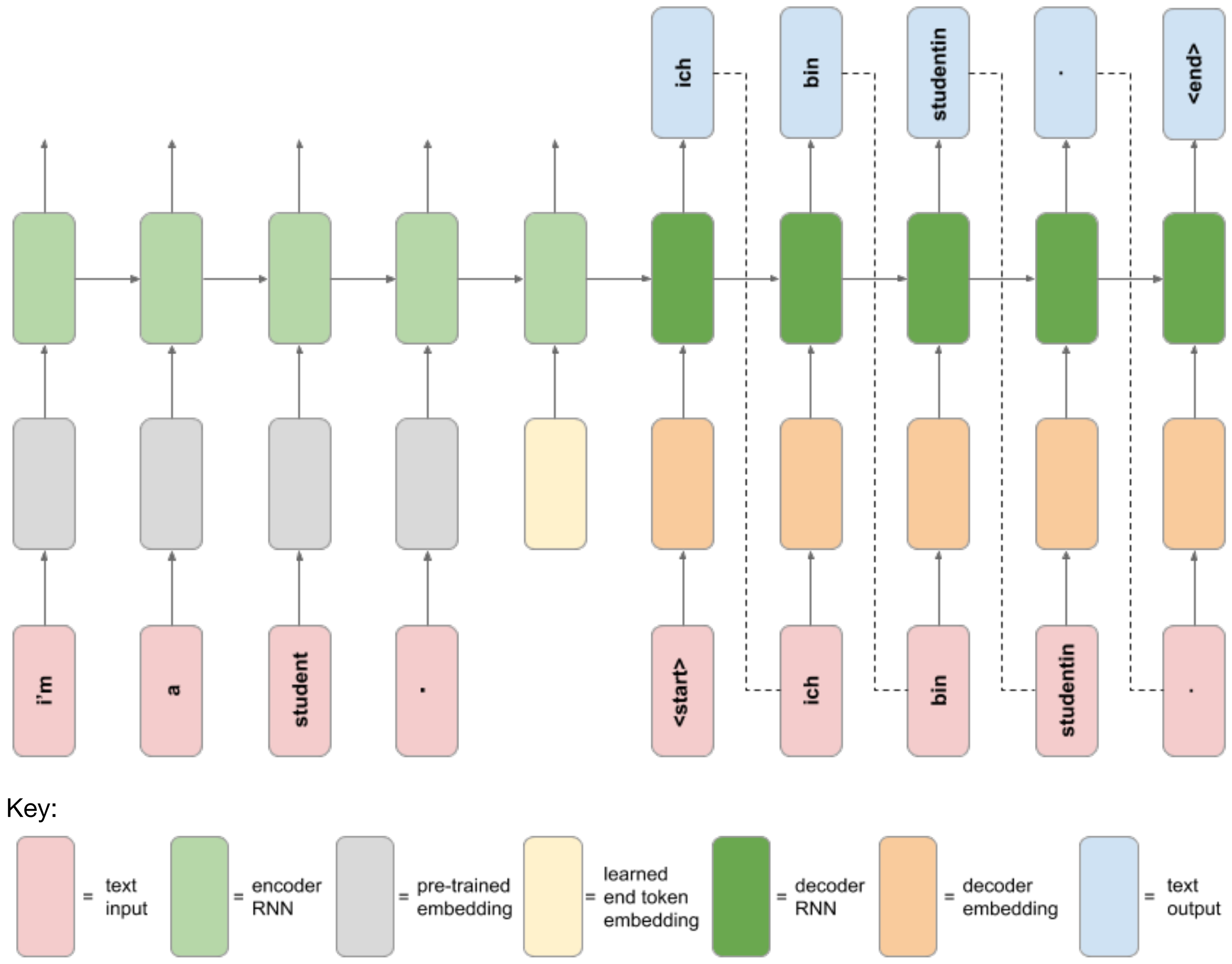
Your goal is to develop a neural translation model from English to German, making use of a pre-trained English word embedding module.

In [6]:
```
# Run this cell to load the dataset

NUM_EXAMPLES = 20000
data_examples = []
with open(directory + '/' + 'data/deu.txt', 'r', encoding='utf8') a
s f:
    for line in f.readlines():
        if len(data_examples) < NUM_EXAMPLES:
            data_examples.append(line)
        else:
            break
```

In [7]:
```
# 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.



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.

In [8]:
```
data_examples[0]
```

Out[8]:
```
'Hi.\tHallo!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123
(CM) & #380701 (cburgmer)\n'
```

In [9]:
```
examples_num = len(data_examples)
eng_sent = [preprocess_sentence(re.split(r'(.*)\t(.*)\t',data_examp
les[x])[1]) for x in range(examples_num)]
ger_sent = ["<start> "+ preprocess_sentence(re.split(r'(.*)\t(.*)\t
',data_examples[x])[2])+" <end>" for x in range(examples_num)]
#print(eng_sent[0:3])
#print(ger_sent[0:3])
```

In [10]:
```
tokenizer = Tokenizer(filters='')
tokenizer.fit_on_texts(ger_sent)
strings_to_sequences = tokenizer.texts_to_sequences(ger_sent)
preprocessed_data = pad_sequences(strings_to_sequences, padding='po
st')

random_idx = np.random.randint(examples_num, size=5)
print("Random samples: ",random_idx)
for i in random_idx:
    print(eng_sent[i], ger_sent[i],strings_to_sequences[i])
    #print(eng_sent[i], ger_sent[i],preprocessed_data[i])
```

```
Random samples:  [ 7971 19153  3819 10290  8216]
may i join you ? <start> kann ich mich euch anschliessen ? <end> [
1, 30, 4, 22, 57, 3956, 7, 2]
i didn't get much . <start> ich habe nicht viel bekommen . <end> [
1, 4, 18, 12, 115, 347, 3, 2]
stop filming . <start> hoer auf zu filmen ! <end> [1, 153, 29, 20,
2491, 9, 2]
i am speechless . <start> ich bin sprachlos . <end> [1, 4, 15, 211
8, 3, 2]
tell everybody . <start> sagen sie es allen . <end> [1, 220, 8, 10
, 654, 3, 2]
```

In [11]:
```
print(preprocessed_data.shape)
```

```
(20000, 14)
```

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 (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.

In [12]:
```
# Load embedding module from Tensorflow Hub

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

#embedding_layer = hub.KerasLayer(directory + '/' + "./models/tf2-p
review_nnlm-en-dim128_1",
#                                  output_shape=[128], input_shape=[
], dtype=tf.string)
```

In [13]:
```
# Test the layer

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

Out[13]:
```
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.

In [14]:
```
X_train, X_val, y_train, y_val = train_test_split(eng_sent, preproc
essed_data, test_size=0.2)
```

In [15]:
```
train_dataset = tf.data.Dataset.from_tensor_slices((X_train, y_trai
n))
val_dataset = tf.data.Dataset.from_tensor_slices((X_val, y_val))
```

In [16]:
```
def split_fun(x, y):

    return tf.strings.split(x, sep=' '), y

train_dataset = train_dataset.map(split_fun)
val_dataset = val_dataset.map(split_fun)
```

In [17]:
```
def emb_fun(x, y):
    return embedding_layer(x), y

train_dataset = train_dataset.map(emb_fun)
val_dataset = val_dataset.map(emb_fun)
```

In [18]:
```
def data_filter(x, y):

    return tf.shape(x)[0]<=13

train_dataset = train_dataset.filter(data_filter)
val_dataset = val_dataset.filter(data_filter)
```

In [19]:
```
def padding_fun(x, y):

    padding = [[13 - tf.shape(x)[0],0],[0,0]]

    return tf.pad(x, padding), y

train_dataset = train_dataset.map(padding_fun)
val_dataset = val_dataset.map(padding_fun)
```

In [20]:
```
train_dataset = train_dataset.batch(16, drop_remainder=True)
val_dataset = val_dataset.batch(16, drop_remainder=True)
```

In [21]:
```
print(train_dataset.element_spec)
print(val_dataset.element_spec)
```

```
(TensorSpec(shape=(16, None, 128), dtype=tf.float32, name=None), T
ensorSpec(shape=(16, 14), dtype=tf.int32, name=None))
(TensorSpec(shape=(16, None, 128), dtype=tf.float32, name=None), T
ensorSpec(shape=(16, 14), dtype=tf.int32, name=None))
```

In [22]:
```
for element in train_dataset.take(1):
    print(element[0].shape)
```

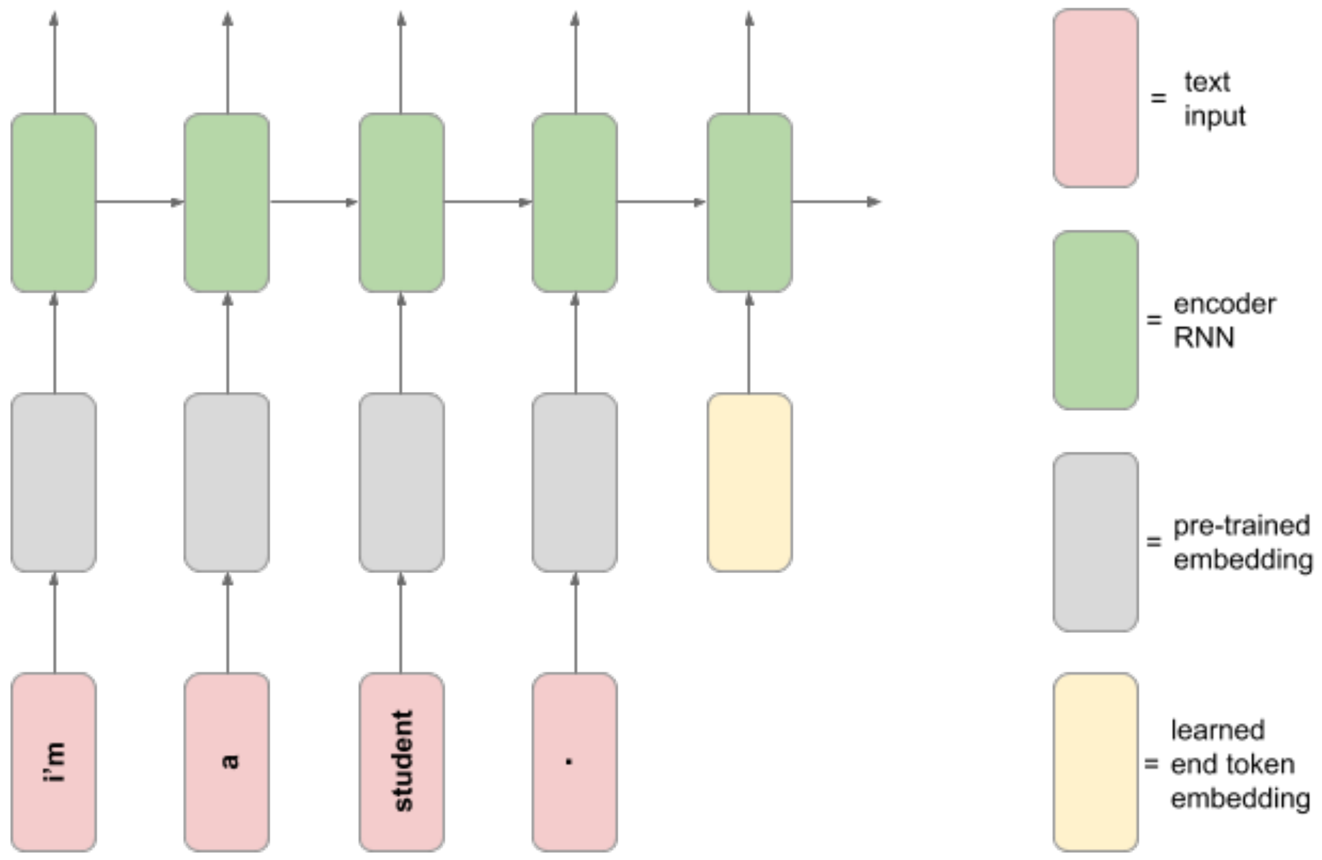
```
(16, 13, 128)
```

In [23]:
```
for element in val_dataset.take(1):
    print(element[1].shape)
```

```
(16, 14)
```

3. Create the custom layer

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



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).

```
In [24]: class CustomLayer(Layer):

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

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

```
In [25]: layer = CustomLayer()
         for element in train_dataset.take(1):
             print(element[0].shape)
             print(layer(element[0]).shape)
```

```
(16, 13, 128)
(16, 14, 128)
```

4. Build the encoder network

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

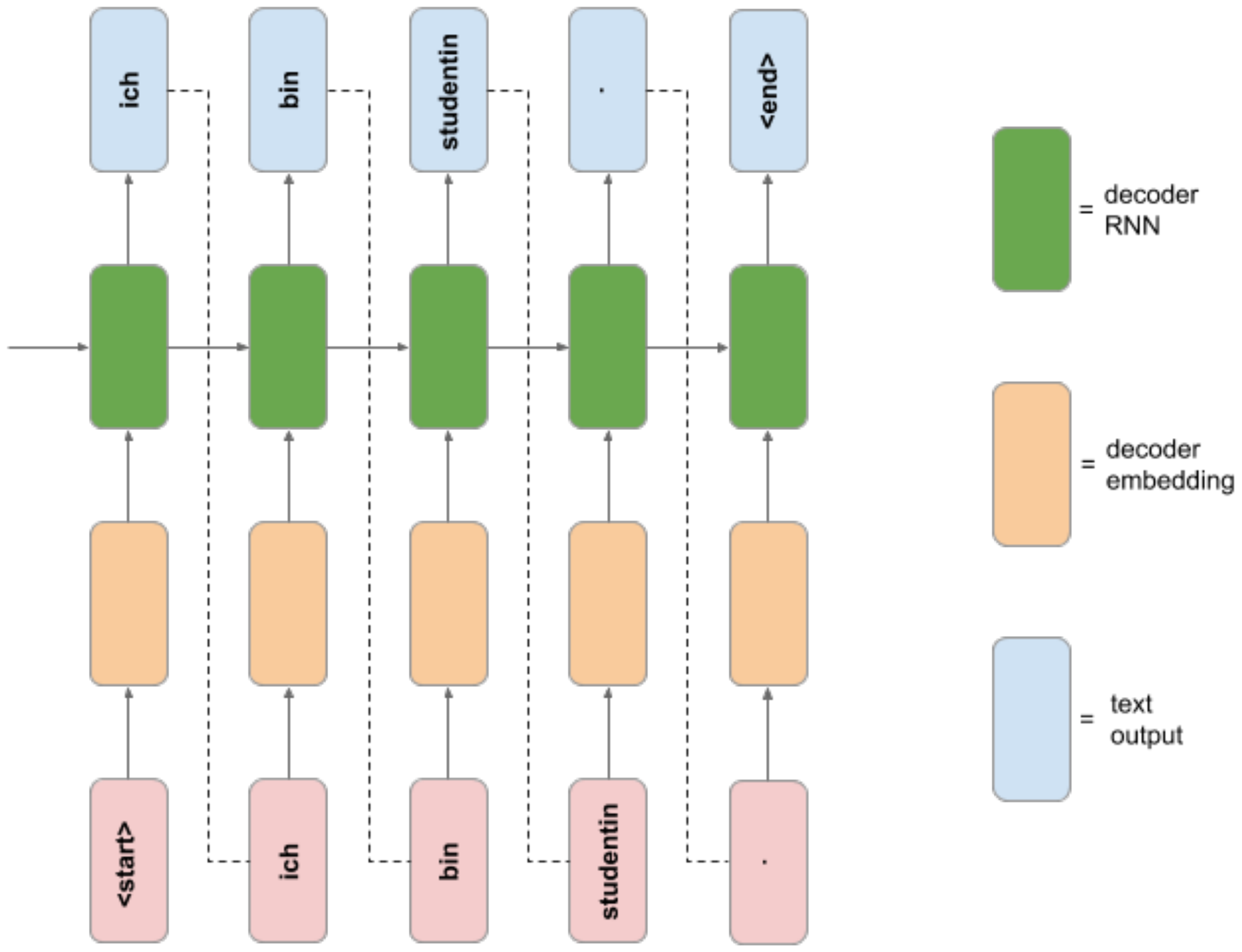
- Using the functional API, build the encoder network according to the following spec:
  - The model will take a batch of sequences of embedded English words as input, as given by the Dataset objects.
  - The next layer in the encoder will be the custom layer you created previously, to add a learned 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.

```
In [26]: inputs = Input(batch_shape=(None, 13, 128))
         h = layer(inputs)
         h = Masking(mask_value = 0)(h)
         h, hidden_state, cell_state = LSTM(512, return_state=True)(h)
         encoder = Model(inputs=inputs, outputs=[hidden_state, cell_state])
```

In [27]:
```
for element in train_dataset.take(1):

    print(encoder(element[0])[0].shape)
```
```
(16, 512)
```
In [28]:
```
encoder.summary()
```
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 13, 128)]         0

 custom_layer (CustomLayer)  (None, 14, 128)           128

 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
_________________________________________________________________
```
In [29]:
```
#encoder.load_weights(directory + '/my_encoder')
```

5. Build the decoder network

The decoder network follows the schematic diagram below.



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.

In [30]:
```
vocab_size= len(tokenizer.word_index) + 1

class DecoderM(Model):

    def __init__(self, **kwargs):
        super(DecoderM, self).__init__(**kwargs)
        self.emb_layer = Embedding(vocab_size, output_dim = 128, ma
sk_zero=True)
        self.myLSTM = LSTM(512, return_sequences=True, return_state
=True)
        self.myDense = Dense(vocab_size)

    def call(self, inputs, hidden_state=None, cell_state=None):
        h = self.emb_layer(inputs)
        if hidden_state is not None and cell_state is not None:
            h, hidden_s, cell_s = self.myLSTM(h, initial_state=[hid
den_state, cell_state])
        else:
            h, hidden_s, cell_s = self.myLSTM(h)
        h = self.myDense(h)
        return h, hidden_s, cell_s
```

In [31]:
```
decoder = DecoderM()
```

```
In [32]: for element in train_dataset.take(1):
             enc_hidden_s, enc_cell_s = encoder(element[0])
             h, hidden_s, cell_s = decoder(element[1], enc_hidden_s, enc_cel
         l_s)
             #print(element[1].shape)

         print(h.shape, hidden_s.shape, cell_s.shape)
```

```
(16, 14, 5744) (16, 512) (16, 512)
```

```
In [33]: decoder.summary()
```

```
Model: "decoder_m"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding (Embedding)       multiple                  735232    
                                                                 
 lstm_1 (LSTM)               multiple                  1312768   
                                                                 
 dense (Dense)               multiple                  2946672   
                                                                 
=================================================================
Total params: 4,994,672
Trainable params: 4,994,672
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.

In [35]:
```python
def create_inputs_and_targets(german):

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

    return (inputs, outputs)
```

In [36]:
```python
#input_seq, target_seq = create_inputs_and_targets(train_dataset)
```

In [37]:
```python
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_lo
gits=True)
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
trainable_variables = encoder.trainable_variables + decoder.trainab
le_variables
```

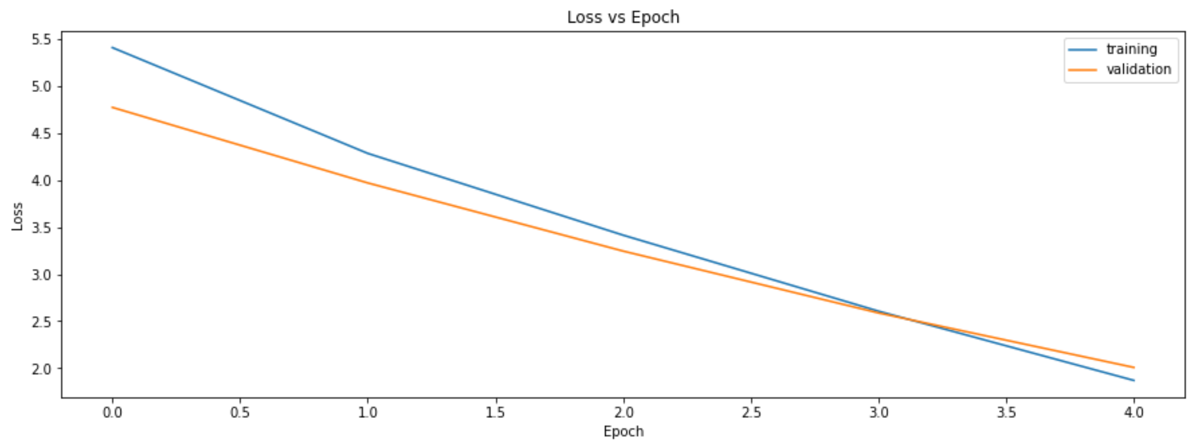
In [38]:
```
@tf.function
def computes_fwd_back(english_in, german_in, german_out, encoder, d
ecoder,
                      optimizer, loss_object):
    with tf.GradientTape() as tape:
        enc_hidden_s, enc_cell_s = encoder(english_in)
        h, hidden_s, cell_s = decoder(german_in, enc_hidden_s, enc_
cell_s)
        loss = loss_object(german_out, h)
        #print("LOSS",loss)
        #print("H",h.take(1))
        grads = tape.gradient(loss, trainable_variables)
    return loss, grads
```

In [39]:
```
def training_loop(num_epochs, encoder, decoder, optimizer, loss_obj
ect):
    tr_epoch_losses = []
    val_epoch_losses = []

    for epoch in range(num_epochs):
        tr_batch_losses = []
        for english, german in train_dataset:
            german_in, german_out = create_inputs_and_targets(germa
n)
            tr_loss, tr_grads = computes_fwd_back(english, german_i
n,
                                                  german_out, encod
er,
                                                  decoder, optimize
r,
                                                  loss_object)


            tr_batch_losses.append(tr_loss)
            optimizer.apply_gradients(zip(tr_grads, trainable_varia
bles))
        tr_epoch_losses.append(np.mean(tr_batch_losses))

        val_batch_losses = []
        for eng, ger in val_dataset:
            ger_in, ger_out = create_inputs_and_targets(ger)
            venc_hidden_s, venc_cell_s = encoder(eng)
            vh, vhidden_s, vcell_s = decoder(ger_in, venc_hidden_s,
venc_cell_s)
            val_batch_losses.append(loss_object(ger_out, vh))
        val_epoch_losses.append(np.mean(val_batch_losses))

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

    return tr_epoch_losses, val_epoch_losses
```

In [40]:
```
tr_epoch_losses, val_epoch_losses = training_loop(5, encoder, decod
er, optimizer, loss_object)
```

```
Epoch : 000
Epoch : 001
Epoch : 002
Epoch : 003
Epoch : 004
```

In [41]:
```
plt.figure(figsize=(15,5))
plt.plot(tr_epoch_losses)
plt.plot(val_epoch_losses)
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.

In [73]:
```
idx = np.random.choice(len(eng_sent)-1, 5, replace=False)

samples = []
german_text = []
for i in idx:
  samples.append(eng_sent[i])
  german_text.append(ger_sent[i])

samples = [eng_sent[i] for i in idx]
german_text = [ger_sent[i] for i in idx]
```

In [74]:
```
for i, sentence in enumerate(samples):

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

    english = embedding_layer(english)
    padding = [[13 - tf.shape(english)[0],0],[0,0]]
    english = tf.pad(english, padding)
    english = np.expand_dims(english, 0)
    e_hidden, e_cell = encoder(english)
    decoder_input = tf.Variable([[tokenizer.word_index['<start>']]]
)

    trans = []
    for j in range(train_dataset.element_spec[1].shape[1]):

      h, e_hidden, e_cell = decoder(decoder_input, e_hidden, e_cell
)
      h = tf.squeeze(tf.argmax(h, axis=2)).numpy()
      if h == tokenizer.word_index['<end>']:
        break

      trans.append(tokenizer.index_word[h])

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

    trans = ' '.join(trans)
    print("English: ", sentence)
    print("German text", german_text[i])
    print("German translated: ", trans)
```

```
English:  can i come over ?
German text <start> kann ich zu dir kommen ? <end>
German translated:  darf ich vorangehen ?
English:  i know tom well .
German text <start> ich kenne tom gut . <end>
German translated:  ich weiss , dass tom es weiss .
English:  you're a genius .
German text <start> du bist ein genie ! <end>
German translated:  du bist ein spion .
English:  was it scary ?
German text <start> war es unheimlich ? <end>
German translated:  war es unheimlich ?
English:  release tom .
German text <start> lassen sie tom frei ! <end>
German translated:  fange tom !
```