Capstone_Project_On_Colab

April 22, 2021

1 Capstone Project

1.1 Neural translation model

1.1.1 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.

1.1.2 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.

1.1.3 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 [119]: import tensorflow as tf
          import tensorflow_hub as hub
          import unicodedata
          import re
          from tensorflow.keras.preprocessing.text import Tokenizer
          import random
          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, Embedding, Dense
          from tensorflow.keras.models import Model
```

```
import json
import matplotlib.pyplot as plt
from IPython.display import 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.

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.

In [120]:
```
# 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("/con
```

In [121]:
```
# Run this cell to load the dataset

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

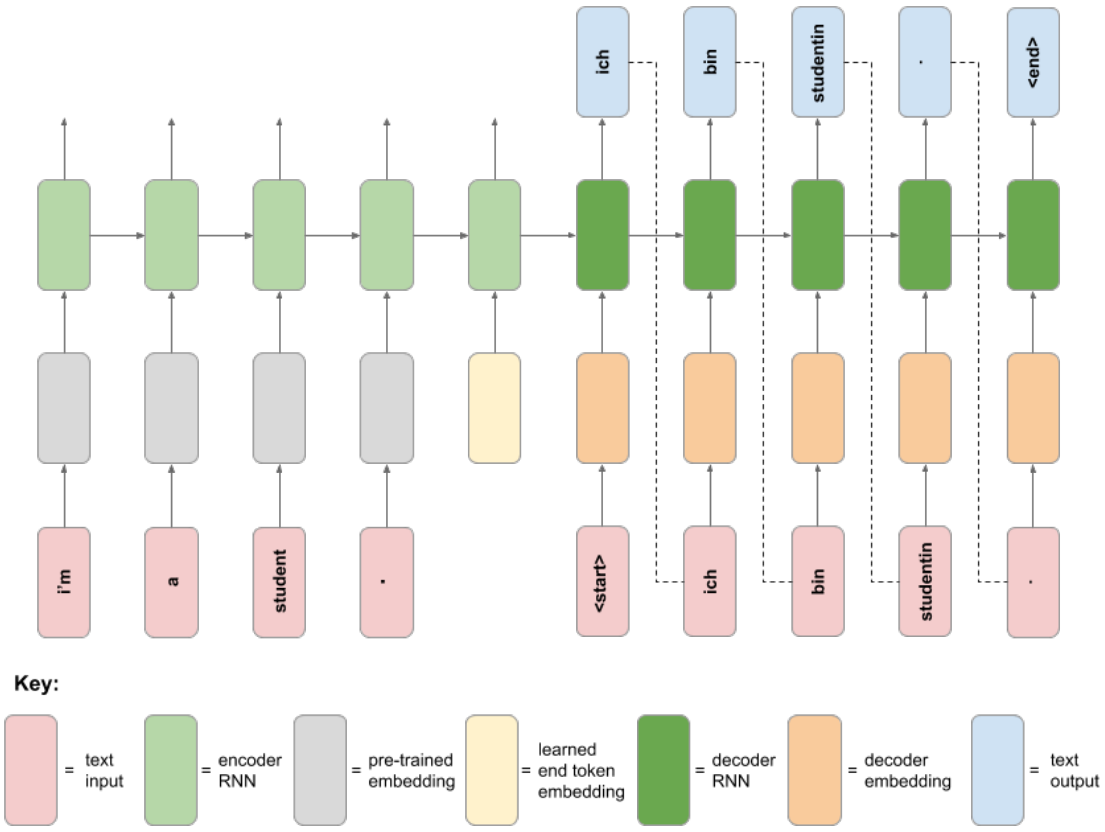
In [122]:
```
# These functions preprocess English and German sentences

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

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.

In [123]:
```
# Run this cell to download and view a schematic diagram for the neural translation mo

!wget -q -O neural_translation_model.png --no-check-certificate "https://docs.google.c
Image("neural_translation_model.png")
```

Out[123]:



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.

1.2 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 [124]: English_sentences = []
          German_sentences = []

In [125]: #Separate English and German Sentences and Preprocess

          for i in range(NUM_EXAMPLES):
              English_sentences.append(preprocess_sentence(data_examples[i].split('\t')[0]))
              German_sentences.append(preprocess_sentence(data_examples[i].split('\t')[1]))

In [126]: #Add <start> and <end> to German Sentences

          for i in range(NUM_EXAMPLES):
              German_sentences[i] = "<start> "+ German_sentences[i] + " <end>"

In [127]: #Tokenize the German Sentences

          tokenizer = Tokenizer(filters='')
          tokenizer.fit_on_texts(German_sentences)
          German_sentence_seq = tokenizer.texts_to_sequences(German_sentences)

In [128]: tokenizer_config = tokenizer.get_config()

          word_counts = json.loads(tokenizer_config['word_counts'])
          index_word = json.loads(tokenizer_config['index_word'])
          word_index = json.loads(tokenizer_config['word_index'])

In [129]: #Print five random English and German Sentences along with the Tokenized Sequences
```

```
        seed_1 = random.sample(range(0,NUM_EXAMPLES), 5)

        for i, idx in enumerate(seed_1):
            print(i+1)
            print("English sentence:", English_sentences[idx])
            print("German sentence:", German_sentences[idx])
            print("Tokenized German sentence:", German_sentence_seq[idx])
            print("\n")
```

```
1
English sentence: tom has children .
German sentence: <start> tom hat kinder . <end>
Tokenized German sentence: [1, 5, 16, 315, 3, 2]


2
English sentence: who's fasting ?
German sentence: <start> wer fastet ? <end>
Tokenized German sentence: [1, 43, 1637, 7, 2]


3
English sentence: tom works here .
German sentence: <start> tom arbeitet hier . <end>
Tokenized German sentence: [1, 5, 459, 33, 3, 2]


4
English sentence: keep the secret .
German sentence: <start> behalten sie das geheimnis fuer sich . <end>
Tokenized German sentence: [1, 306, 8, 11, 955, 77, 34, 3, 2]


5
English sentence: where are you ?
German sentence: <start> wo sind sie ? <end>
Tokenized German sentence: [1, 83, 23, 8, 7, 2]
```

```
In [130]: #Pad the German Sentence Sequences

        padded_German_seqs = pad_sequences(German_sentence_seq, padding = 'post', value = 0.0)
```

1.3 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.

```
In [131]: # Load embedding module from Tensorflow Hub

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

```
In [133]: # Test the layer

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

```
Out[133]: 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.

In [140]:
```
#Split the Data into Train, Validation and Test.
#We will do this in the ratio (0.72, 0.18, 0.10)

train_inputs, test_inputs, train_targets, test_targets = train_test_split(English_sent
                                                                           padded_Germa
                                                                           test_size=0.

train_inputs, val_inputs, train_targets, val_targets = train_test_split(train_inputs,
                                                                        train_targets,
                                                                        test_size=0.2)
```

In [144]:
```
#Check the shapes of Train, Validation and Test Data

print("The size of Train data:", len(train_inputs))
print("The size of Validation data:", len(val_inputs))
print("The size of Test data:", len(test_inputs))
print("The train, validation and test data are in ratio 0.72 : 0.18 : 0.10 ")
```

```
The size of Train data: 14400
The size of Validation data: 3600
The size of Test data: 2000
The train, validation and test data are in ratio 0.72 : 0.18 : 0.10
```

In [145]:
```
#Prepare Train and Validation datasets

train_dataset = tf.data.Dataset.from_tensor_slices((train_inputs, train_targets))
val_dataset = tf.data.Dataset.from_tensor_slices((val_inputs, val_targets))
```

In [146]:
```
#Function to Split English Sentences at Spaces

def Split_English_Sentences(English_sentence, German_sequence):
    return tf.strings.split(English_sentence, sep = ' '), German_sequence
```

In [147]:
```
#Split English Sentences at Spaces

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

In [148]:
```
#Function to Embed English Sentences

def embed_English_words(English_sentence, German_sequence):
    return embedding_layer(English_sentence), German_sequence
```

In [149]:
```
#Embed English Sentences

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

In [150]:
```
#Function to Filter English Sentences of Length Greater than 13

def filter_English_sentences(English_embed_sequence, German_sequence):
    return tf.shape(English_embed_sequence)[0] <= 13
```

In [151]:
```
#Filter English Sentences of Length Greater than 13

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

In [152]:
```
#Function to Pad English Sentences

def pad_English_sequences(English_embed_sequence, German_sequence):
    l = (13 - int(tf.shape(English_embed_sequence)[0]))
    paddings = [[l, 0], [0, 0]]
    padded_English_sequence = tf.pad(English_embed_sequence, paddings)
    return padded_English_sequence, German_sequence
```

In [153]:
```
#Pad English Sentences

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

In [154]:
```
#Batch the Train and Validation Dataset

train_dataset = train_dataset.batch(16)
val_dataset = val_dataset.batch(16)
```

In [155]:
```
#Train and Validation Dataset Element Spec

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

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

In [156]:
```
#Train Dataset Take(1)

print(train_dataset.take(1))
```

```
<TakeDataset shapes: ((None, None, 128), (None, 14)), types: (tf.float32, tf.int32)>
```

In [157]:
```
#Validation Dataset take(1)

print(val_dataset.take(1))
```

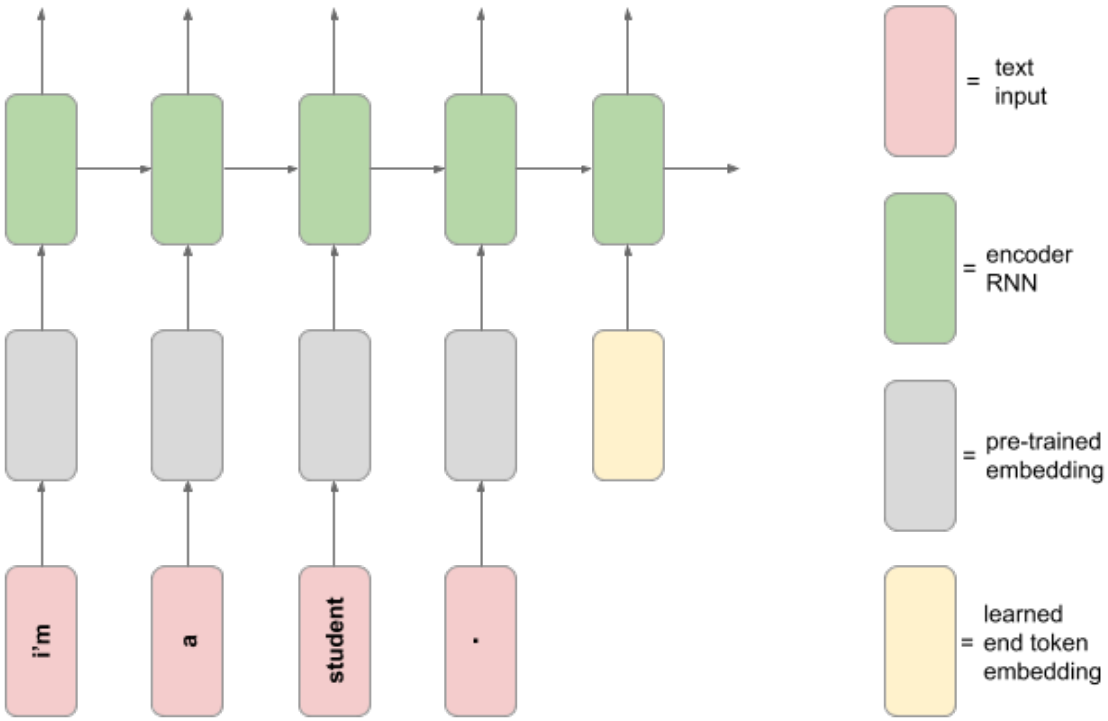
```
<TakeDataset shapes: ((None, None, 128), (None, 14)), types: (tf.float32, tf.int32)>
```

1.4 3. Create the custom layer

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

In [158]:
```
# 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.c
Image("neural_translation_model.png")
```

Out[158]:



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 [159]:
```
#Custom Layer to Include a Trainable End Token in English Sentences

class English_End_Token(Layer):

    def __init__(self, **kwargs):
```

```
            super(English_End_Token, self).__init__(**kwargs)
            self.end = tf.Variable(initial_value = tf.zeros(shape = (1,128)),
                                   trainable = True,
                                   dtype = tf.float32)

        def call(self, inputs):
            v1 = self.end
            v1 = tf.tile(v1, [tf.shape(inputs)[0], 1])
            v1 = tf.expand_dims(v1, axis = 1)
            return tf.concat([inputs, v1], axis = 1)
```

In [160]:
```
#Get an Instance of the Custom Layer

English_End_Token_Layer = English_End_Token()
```

In [161]:
```
#Input and Output Shapes of the Custom Layer

for element in train_dataset.take(1):
    English_data, German_data = element
    print("Shape of the input batch before adding end token:", English_data.shape)
    print("Shape of the input batch after adding end token:", English_End_Token_Layer(
```

```
Shape of the input batch before adding end token: (16, 13, 128)
Shape of the input batch after adding end token: (16, 14, 128)
```

1.5 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 [162]:
```
#Build Encoder Network

def encoder_network(input_shape):

    #Encodes the English data

    inputs = Input(shape = input_shape)
```

```
        h = English_End_Token_Layer(inputs)
        h = Masking(mask_value = 0.0)(h)
        whole_seq_output, final_hidden_state, final_cell_state = LSTM(units = 512,
                                                                      return_sequences =
                                                                      return_state = True

        encoder_model = Model(inputs = inputs,
                              outputs = [final_hidden_state, final_cell_state])

        return encoder_model
```

In [163]:
```
#Get an Instance of the Encoder Network

encoder_model = encoder_network((13, 128))
```

In [164]:
```
#Input and Output Shapes of the Encoder Network

for element in train_dataset.take(1):
    English_data, German_data = element
    print("Shape of the input batch:", English_data.shape)
    print("Shape of the output hidden state:", encoder_model(English_data)[0].shape)
```

```
Shape of the input batch: (16, 13, 128)
Shape of the output hidden state: (16, 512)
```

In [165]:
```
#Encoder Model Summary

encoder_model.summary()
```

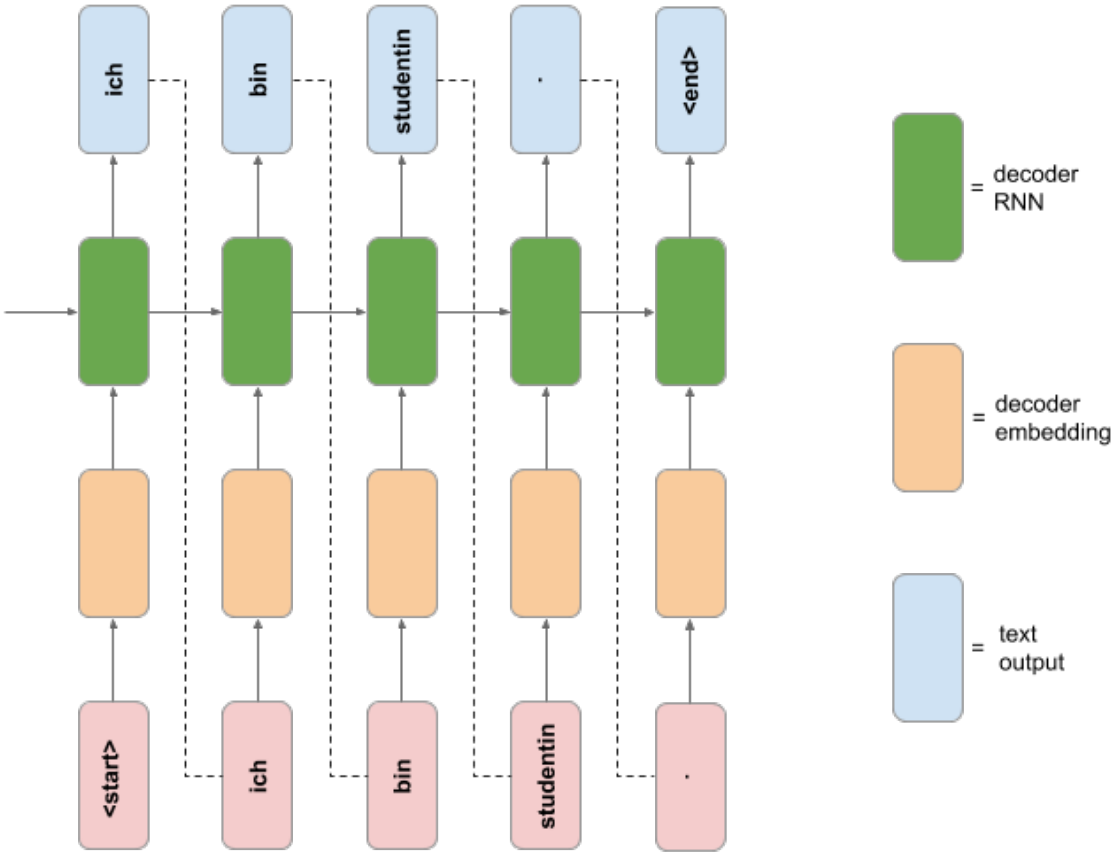
```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 13, 128)]         0
_________________________________________________________________
english__end__token_1 (Engli (None, 14, 128)           128
_________________________________________________________________
masking_1 (Masking)          (None, 14, 128)           0
_________________________________________________________________
lstm_2 (LSTM)                [(None, 14, 512), (None,  1312768
=================================================================
Total params: 1,312,896
Trainable params: 1,312,896
Non-trainable params: 0
_________________________________________________________________
```

1.6 5. Build the decoder network

The decoder network follows the schematic diagram below.

In [166]:
```
# 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.c
Image("neural_translation_model.png")
```

Out[166]:



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 [167]:
```
#Build Decoder Network

class RNN_decoder_network(Model):

    def __init__(self, no_German_tokens, **kwargs):
        super(RNN_decoder_network, self).__init__(**kwargs)
        self.embedding = Embedding(no_German_tokens+1, 128, mask_zero = True)
        self.lstm = LSTM(units = 512,
                         return_sequences = True,
                         return_state = True)
        self.dense = Dense(no_German_tokens+1)

    def call(self, inputs, hidden_st = None, cell_st = None):
        h = self.embedding(inputs)
        h, out_hidden_state, out_cell_state  = self.lstm(h, initial_state = [hidden_st
                                                                             cell_st])
        return self.dense(h), out_hidden_state, out_cell_state
```

In [168]:
```
#Get an Instance of the Decoder Network

decoder_model = RNN_decoder_network(len(word_index))
```

In [169]:
```
#Input and Output Shapes of the Decoder Network

for element in train_dataset.take(1):
    English_data, German_data = element
    print("Shape of the input English batch:", English_data.shape)
    print("Shape of the input German batch:", German_data.shape)
    hidden_state, cell_state = encoder_model(English_data)
    print("Shape of the encoder model hindden state:", hidden_state.shape)
    print("Shape of the encoder model cell state:", cell_state.shape)
    out_German, out_hidden_state, out_cell_state = decoder_model(German_data,
                                                                 hidden_state,
                                                                 cell_state)
    print("Shape of the output German batch:", out_German.shape)
    print("Shape of the output hidden state:", out_hidden_state.shape)
    print("Shape of the output cell state:", out_cell_state.shape)
```

```
Shape of the input English batch: (16, 13, 128)
Shape of the input German batch: (16, 14)
```

```
Shape of the encoder model hindden state: (16, 512)
Shape of the encoder model cell state: (16, 512)
Shape of the output German batch: (16, 14, 5744)
Shape of the output hidden state: (16, 512)
Shape of the output cell state: (16, 512)
```

```
In [170]: #Decoder Model Summary

          decoder_model.summary()
```

```
Model: "rnn_decoder_network_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      multiple                  735232
_________________________________________________________________
lstm_3 (LSTM)                multiple                  1312768
_________________________________________________________________
dense_1 (Dense)              multiple                  2946672
=================================================================
Total params: 4,994,672
Trainable params: 4,994,672
Non-trainable params: 0
_________________________________________________________________
```

1.7 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 [171]:
```
def German_inputs_outputs(German_batch_data):
    initial = 1
    final = German_batch_data.shape[1]-1
    German_inputs = German_batch_data[:, :final]
    German_outputs = German_batch_data[:, initial:]
    return German_inputs, German_outputs
```

In [172]:
```
# Create the optimizer and loss

optimizer_obj = tf.keras.optimizers.Adam(learning_rate=0.00001)
loss_obj = tf.keras.losses.SparseCategoricalCrossentropy()
```

In [173]:
```
@tf.function
def model_forward_backward_pass(encoder_model,
                                decoder_model,
                                English_input,
                                German_input,
                                German_output,
                                loss):

    with tf.GradientTape() as tape:
        encoder_hidden_state, encoder_cell_state = encoder_model(English_input)
        decoder_German_output, _, _ = decoder_model(German_input,
                                                    encoder_hidden_state,
                                                    encoder_cell_state)
        loss_value = loss(German_output, decoder_German_output)
        gradients = tape.gradient(loss_value,
                                  encoder_model.trainable_variables+
                                   decoder_model.trainable_variables)

    return loss_value, gradients
```

In [174]:
```
# Run the custom training loop

num_epochs = 25


train_loss_results = []
val_loss_results = []


for epoch in range(num_epochs):

    train_epoch_loss_avg = tf.keras.metrics.Mean()
    val_epoch_loss_avg = tf.keras.metrics.Mean()
```

```
        for element in train_dataset:
            English_data, German_data = element
            German_inputs, German_outputs = German_inputs_outputs(German_data)
            loss_value, gradients = model_forward_backward_pass(encoder_model,
                                    decoder_model,
                                    English_data,
                                    German_inputs,
                                    German_outputs,
                                    loss_obj)
            optimizer_obj.apply_gradients(zip(gradients,
                                          encoder_model.trainable_variables+
                                           decoder_model.trainable_variables))

            train_epoch_loss_avg(loss_value)

        for element in val_dataset:
            English_data, German_data = element
            German_inputs, German_outputs = German_inputs_outputs(German_data)
            encoder_hidden_state, encoder_cell_state = encoder_model(English_data)
            decoder_German_output, _, _ = decoder_model(German_inputs,
                                                    encoder_hidden_state,
                                                    encoder_cell_state)
            val_loss = loss_obj(German_outputs, decoder_German_output)

            val_epoch_loss_avg(val_loss)

        train_loss_results.append(train_epoch_loss_avg.result().numpy())

        val_loss_results.append(val_epoch_loss_avg.result().numpy())

        print("Epoch {:03d}: Training loss: {:.3f}: Validation loss: {:.3f}".format(epoch,
                                                                                    train_
                                                                                    val_ep
```

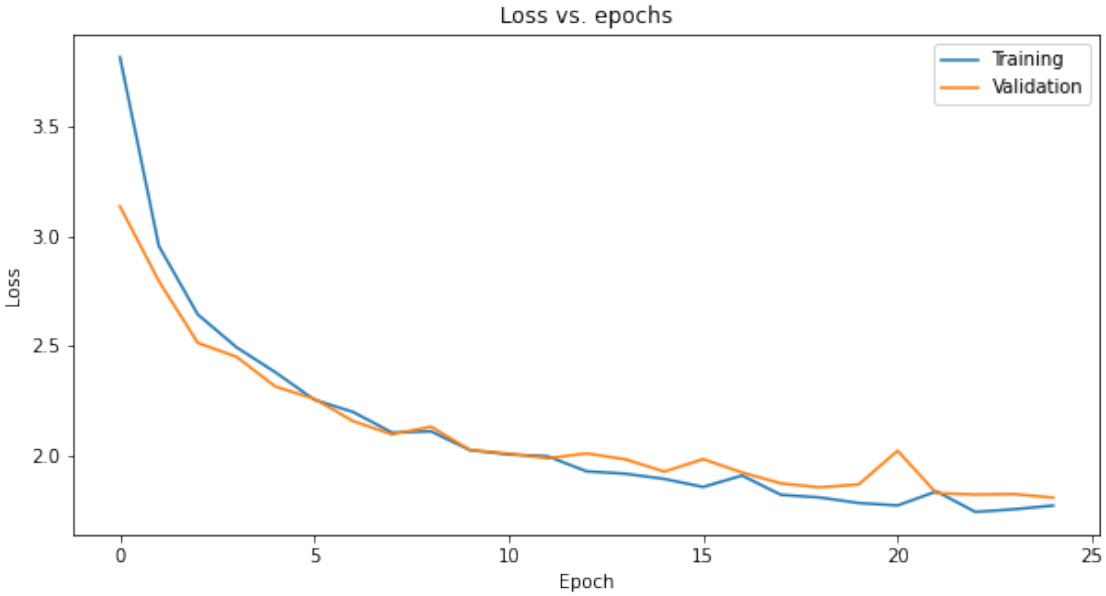
```
Epoch 000: Training loss: 3.819: Validation loss: 3.137
Epoch 001: Training loss: 2.955: Validation loss: 2.795
Epoch 002: Training loss: 2.643: Validation loss: 2.513
Epoch 003: Training loss: 2.492: Validation loss: 2.449
Epoch 004: Training loss: 2.378: Validation loss: 2.313
Epoch 005: Training loss: 2.253: Validation loss: 2.257
Epoch 006: Training loss: 2.196: Validation loss: 2.155
Epoch 007: Training loss: 2.102: Validation loss: 2.094
Epoch 008: Training loss: 2.108: Validation loss: 2.130
Epoch 009: Training loss: 2.023: Validation loss: 2.024
Epoch 010: Training loss: 2.002: Validation loss: 2.007
```

```
Epoch 011: Training loss: 1.995: Validation loss: 1.985
Epoch 012: Training loss: 1.925: Validation loss: 2.007
Epoch 013: Training loss: 1.915: Validation loss: 1.981
Epoch 014: Training loss: 1.891: Validation loss: 1.924
Epoch 015: Training loss: 1.854: Validation loss: 1.982
Epoch 016: Training loss: 1.906: Validation loss: 1.920
Epoch 017: Training loss: 1.818: Validation loss: 1.871
Epoch 018: Training loss: 1.806: Validation loss: 1.852
Epoch 019: Training loss: 1.781: Validation loss: 1.866
Epoch 020: Training loss: 1.770: Validation loss: 2.020
Epoch 021: Training loss: 1.834: Validation loss: 1.826
Epoch 022: Training loss: 1.740: Validation loss: 1.820
Epoch 023: Training loss: 1.752: Validation loss: 1.821
Epoch 024: Training loss: 1.769: Validation loss: 1.805
```

In [175]:
```
train_loss = []
val_loss = []
```

In [176]:
```
for item in train_loss_results:
    train_loss.append(item)


for item in val_loss_results:
    val_loss.append(item)
```

In [177]:
```
#Plot Loss vs Epochs

fig = plt.figure(figsize=(10,5))

plt.plot(train_loss)
plt.plot(val_loss)
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```



1.8 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 [294]:
```
#Select 5 Random English Sentences from the Test Set

seed_2 = random.sample(range(0,len(test_inputs)), 5)
```

In [266]:
```
#Preprocess English Sentence

def preprocess_English_sentence(sample_English_sentence):

  #Split by space
  sample_English_sentence,_ = Split_English_Sentences(sample_English_sentence, None)

  #Embed
  sample_English_sentence,_ = embed_English_words(sample_English_sentence, None)
```

```
            #Eliminate sentences of length > 13
            if tf.shape(sample_English_sentence)[0] > 13:
              return None

            #Pad to the length 13
            sample_English_sentence,_ = pad_English_sequences(sample_English_sentence, None)

            #Add batch dimension of 1
            sample_English_sentence = tf.expand_dims(sample_English_sentence, axis = 0)

            return sample_English_sentence

In [295]: for index, element in enumerate(seed_2):

            print(index+1)
            print("The input English sentence is:", test_inputs[element])
            print(" ")

            English_data = preprocess_English_sentence(test_inputs[element])

            if English_data != None:

              #Pass the processed English data into the encoder model
              encoder_hidden_state, encoder_cell_state = encoder_model(English_data)

              German_data = tf.expand_dims(word_index["<start>"], axis =0)
              German_data = tf.reshape(German_data, (1,1))
              hidden_state = encoder_hidden_state
              cell_state = encoder_cell_state

              German_seq = []
              end_seq = False

              #Loop the decoder model to collect the German output word by word
              while len(German_seq) <= 14 or end_seq == False:
                German_dense_output, hidden_state, cell_state = decoder_model(German_data, hidde
                German_data = tf.math.argmax(German_dense_output, axis=2)
                German_seq.append(tf.squeeze(German_data).numpy())
                if tf.squeeze(German_data).numpy() == 2:
                  end_seq = True

              print("German translation by our model:", end = " ")
              for item in German_seq:
                if item != 0 and item != 2:
                  print(index_word[str(item)], end = " ")
              print("\n")

              print("The expected German translation:", end = " ")
```

```
            for item in test_targets[element]:
              if item !=0 and item != 2 and item != 1:
                print(index_word[str(item)], end = " ")
            print("\n")
            print("\n")

          else:
            print("This sentence has a length greater than 13. So moving on to the next senten
```

```
1
The input English sentence is: you let me down .

German translation by our model: wir haben sie .

The expected German translation: ihr habt mich enttaeuscht .



2
The input English sentence is: where's boston ?

German translation by our model: wo ist ?

The expected German translation: wo liegt boston ?



3
The input English sentence is: are you impressed ?

German translation by our model: wir haben wir ?

The expected German translation: seid ihr beeindruckt ?



4
The input English sentence is: try again .

German translation by our model: war sie .

The expected German translation: versuch es noch einmal .



5
The input English sentence is: i saw your father .
```

```
German translation by our model: ich habe helfen .

The expected German translation: ich habe ihren vater gesehen .
```

In [ ]: