Capstone_Project

April 26, 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 [1]: import tensorflow as tf
        import tensorflow_hub as hub
        import unicodedata
        import re
        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 [2]:
```
# Run this cell to connect to your Drive folder

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

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

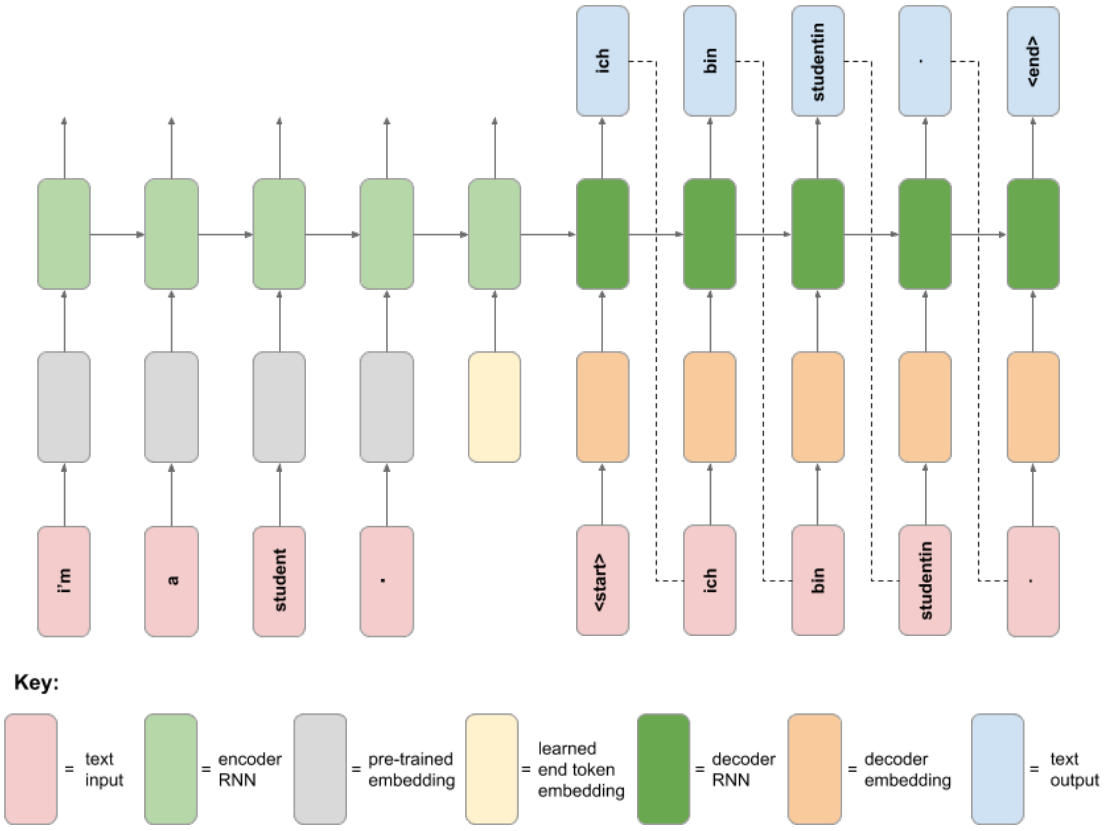
In [3]:
```
# Run this cell to load the dataset

NUM_EXAMPLES = 20000
data_examples = []
with open('/content/gdrive/MyDrive/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 [4]:
```
# 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

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', 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 [5]:
```
# 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[5]:



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 [5]: # This function creates the two lists of german and english
        def english_german_sentences(data):
          english = []
          german = []
          for line in data:
            sentences = line.split('\t')
            english.append(preprocess_sentence(sentences[0]))
            german.append(preprocess_sentence(sentences[1]))
          return english, german

In [6]: # here I separate the sentences
        english_list, german_list = english_german_sentences(data_examples)

In [7]: # just validating tha the number of sentences did not change
        print('Number of sentences in English: {}'.format(len(english_list)))
        print('Number of sentences in German: {}'.format(len(german_list)))

Number of sentences in English: 20000
Number of sentences in German: 20000


In [8]: # here I add the special tokens in the german sentences
        german_list = ['<start> {} <end>'.format(s) for s in german_list]

In [9]: # select some samples
        import numpy as np
        indices = np.random.choice(len(german_list), 10, replace=False)

In [10]: # tokenizer
         tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=None,
                                                           filters='',
                                                           split=' ',
                                                           oov_token='<UNK>'
                                                           )
```

```
In [11]: # tokenizing german sentences
         tokenizer.fit_on_texts(german_list)

In [12]: # show selected sentences in german
         for k in indices:
           print(german_list[k])
```

```
<start> tom geht es besser . <end>
<start> sie stinken . <end>
<start> tom ist nicht schwach . <end>
<start> ich bin beleidigt . <end>
<start> ich bin etwas verrueckt . <end>
<start> geben sie mir ihre hand . <end>
<start> er verlor seine arbeit . <end>
<start> ich kann mehr machen . <end>
<start> mein kopf ist voll . <end>
<start> ich fragte nach dem grund . <end>
```

```
In [13]: # show selected tokenized sentences from german
         tokenizer.texts_to_sequences(np.array(german_list)[indices])

Out[13]: [[2, 6, 60, 11, 294, 4, 3],
          [2, 9, 2310, 4, 3],
          [2, 6, 7, 13, 490, 4, 3],
          [2, 5, 16, 937, 4, 3],
          [2, 5, 16, 122, 246, 4, 3],
          [2, 318, 9, 22, 203, 652, 4, 3],
          [2, 15, 653, 372, 164, 4, 3],
          [2, 5, 31, 145, 87, 4, 3],
          [2, 51, 730, 7, 784, 4, 3],
          [2, 5, 1282, 66, 119, 752, 4, 3]]

In [14]: # show selected sentences in english
         for k in indices:
           print(english_list[k])
```

```
tom is better .
you stink .
tom isn't weak .
i'm offended .
i am a bit crazy .
give me your hand .
he lost his job .
i can do more .
my brain is full .
i asked why .
```

```
In [15]: # Pad the end of the tokenized German sequences with zeros, and batch the complete se
         max_len_en = max([len(s) for s in english_list])
         max_len_ge = max([len(s) for s in german_list])
         max_len = max(max_len_en, max_len_ge)
         tokenized_german = tokenizer.texts_to_sequences(german_list)
         padded_german_list = tf.keras.preprocessing.sequence.pad_sequences(tokenized_german,
                                                                            padding='post',
                                                                            maxlen=max_len)

In [16]: padded_german_list.shape

Out[16]: (20000, 87)

In [17]: german_data = padded_german_list.tolist()
```

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 [18]: # 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 [19]: # Test the layer

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

Out[19]: 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 [20]:
```
# creat random training and validation data from the english and german lists
i_data = np.arange(len(german_data))
i_data = np.random.choice(len(i_data), len(i_data), replace=False)
n_train = np.round(0.80 * len(i_data)).astype(np.int32)
english_train = [english_list[i] for i in i_data[:n_train]]
english_val = [english_list[i] for i in i_data[n_train:]]
german_train = [german_data[i] for i in i_data[:n_train]]
german_val = [german_data[i] for i in i_data[n_train:]]
```

In [21]:
```
# check dimensions
print(len(english_train), len(english_val))
print(len(german_train), len(german_val))
```

```
16000 4000
16000 4000
```

In [22]:
```
# create loading
train_data = tf.data.Dataset.from_tensor_slices((english_train, german_train))
val_data = tf.data.Dataset.from_tensor_slices((english_val, german_val))
```

In [23]:
```
# map used to split english sentences into their building tokens
def split_english_tokens(dataset):
  def split_sentences(s1, s2):
    return (tf.strings.split(s1, sep=' '), s2)

  dataset = dataset.map(split_sentences)
  return dataset
```

In [24]:
```
# apply split-tokens map
train_data = split_english_tokens(train_data)
val_data = split_english_tokens(val_data)
```

```
In [25]: # map used to embedd english tokens
         def embedd_english_tokens(dataset):
           def embedd_tokens(s1, s2):
             return (embedding_layer(s1), s2)

           dataset  = dataset.map(embedd_tokens)
           return dataset

In [26]: # apply map function to embedd english tokens
         train_data = embedd_english_tokens(train_data)
         val_data = embedd_english_tokens(val_data)

In [27]: # filter used to remove english sentences with more than 13 embedded tokens
         def chunk_embedded_english_sentences(dataset, max_len):
           def chunk_data(s1, s2):
             return tf.less_equal(tf.shape(s1)[0], tf.constant(max_len, dtype=tf.int32))

           dataset = dataset.filter(chunk_data)
           return dataset

In [28]: # apply filter
         max_len_en = 13
         train_data = chunk_embedded_english_sentences(train_data, max_len_en)
         val_data = chunk_embedded_english_sentences(val_data, max_len_en)

In [29]: tf.pad(np.random.rand(4,128), paddings=[[9,0],[0,0]], mode='CONSTANT').shape

Out[29]: TensorShape([13, 128])

In [30]: # pad embedded english tokens so that each sentence would be (max_len_en, 128)
         def padding_english_token(dataset, max_len):
           def do_map(s1, s2):
             s_len = tf.shape(s1)[0]
             if tf.less(s_len, tf.cast(max_len, tf.int32)):
               s1 = tf.pad(s1, paddings=[[max_len - s_len, 0], [0, 0]], mode='CONSTANT')
             return s1, s2

           dataset = dataset.map(do_map)
           return dataset

In [31]: # apply padding map function to embedded english tokens
         train_data = padding_english_token(train_data, max_len_en)
         val_data = padding_english_token(val_data, max_len_en)

In [33]: # check embedded lengths
         def check_embedded_len(dataset, max_len):
           dataset_iter = iter(dataset)
           for x,y in dataset_iter:
             if x.shape[0] != max_len:
```

```
            print('Found invalid length!')
            return
        print('All lengths are valid')
```

In [34]:
```
# check train_data
check_embedded_len(train_data, max_len_en)
```

```
All lengths are valid
```

In [35]:
```
# check val_data
check_embedded_len(val_data, max_len_en)
```

```
All lengths are valid
```

In [32]:
```
# set batch size to 16 for both datasets
train_data = train_data.batch(batch_size=16)
val_data = val_data.batch(batch_size=16)
```

In [33]:
```
# show element specification from training dataset
train_data.element_spec
```

Out[33]:
```
(TensorSpec(shape=(None, None, 128), dtype=tf.float32, name=None),
 TensorSpec(shape=(None, 87), dtype=tf.int32, name=None))
```

In [34]:
```
# show element specification from validation dataset
val_data.element_spec
```

Out[34]:
```
(TensorSpec(shape=(None, None, 128), dtype=tf.float32, name=None),
 TensorSpec(shape=(None, 87), dtype=tf.int32, name=None))
```

In [35]:
```
# show english element from training dataset
a = train_data.take(1)
s1, _ = next(iter(a))
print(s1.shape)
```

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

In [36]:
```
# show german element from validation dataset
a = val_data.take(1)
_, s2 = next(iter(a))
print(s2.shape)
```

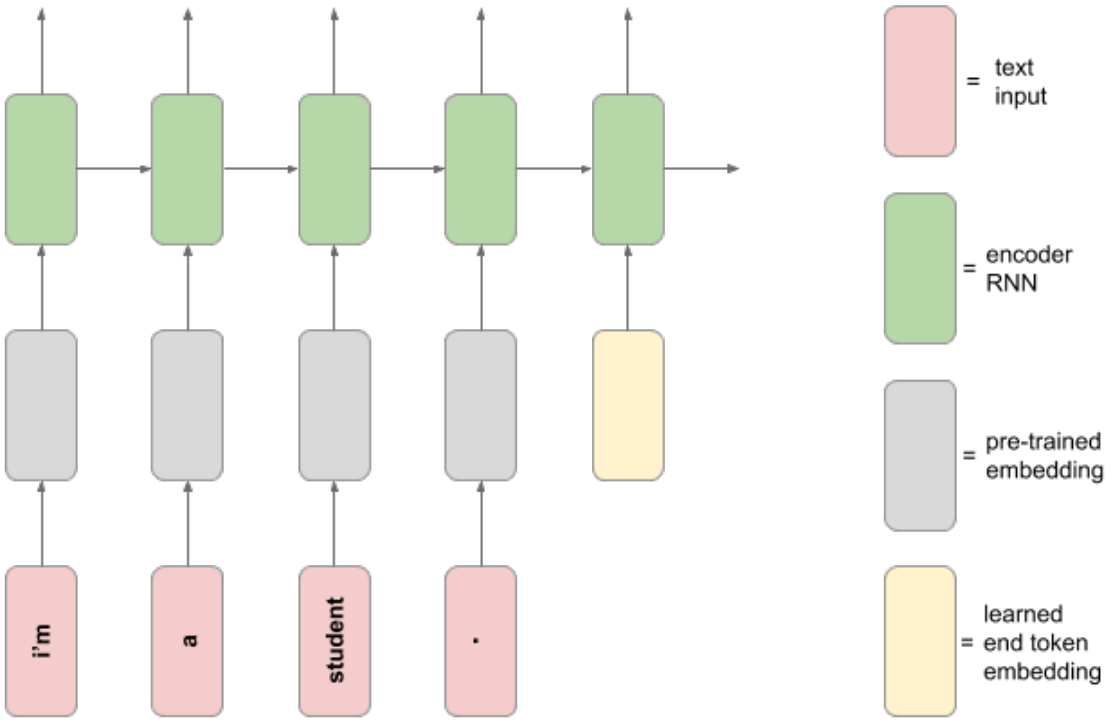
```
(16, 87)
```

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 [38]:
```
# 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.
Image("neural_translation_model.png")
```

Out[38]:



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 [37]:
```
class AppendLayer(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super(AppendLayer, self).__init__(**kwargs)
    self.token = tf.Variable(initial_value=tf.random.uniform(shape=(1, 128)),
                             trainable=True, name='AppendLayer')
```

```
        def call(self, inputs):
          out = tf.map_fn(lambda x: tf.concat([x, self.token], axis=0), inputs)
          return out
```

```
In [38]: a = tf.random.uniform(shape=(3,2,5))
         b = tf.ones(shape=(1,5))
         c = tf.map_fn(lambda x: tf.concat([x, b], axis=0), a)
         c
```

```
Out[38]: <tf.Tensor: shape=(3, 3, 5), dtype=float32, numpy=
         array([[[0.6690376 , 0.684734  , 0.5029422 , 0.887985  , 0.2627312 ],
                 [0.8662598 , 0.2457856 , 0.76179564, 0.6990887 , 0.89905787],
                 [1.        , 1.        , 1.        , 1.        , 1.        ]],

                [[0.8199105 , 0.3815521 , 0.9305825 , 0.3656783 , 0.361565  ],
                 [0.37621784, 0.63981485, 0.6620909 , 0.82071626, 0.7254156 ],
                 [1.        , 1.        , 1.        , 1.        , 1.        ]],

                [[0.0452863 , 0.9250672 , 0.7216401 , 0.45978677, 0.24611759],
                 [0.15534377, 0.9287919 , 0.7484138 , 0.9056667 , 0.7214905 ],
                 [1.        , 1.        , 1.        , 1.        , 1.        ]]],
               dtype=float32)>
```

```
In [39]: a = train_data.take(1)
         english_b, _ = next(iter(a))
         print('Original batch size: {}'.format(english_b.shape))
         b = tf.keras.models.Sequential([AppendLayer()])
         print('Appended batch size: {}'.format(b(english_b).shape))
```

```
Original batch size: (16, 13, 128)
Appended batch size: (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 [40]: def get_encoder():
             input_layer = tf.keras.layers.Input(shape=(13, 128))
             appended_token = AppendLayer()(input_layer)
             mask_layer = tf.keras.layers.Masking(mask_value=0.0)(appended_token)
             sequences, hidden_state, carry_state = tf.keras.layers.LSTM(
                 units=512, return_state=True)(mask_layer)
             model = tf.keras.models.Model(
                 inputs = input_layer, outputs=[hidden_state, carry_state])
             return model

In [41]: # instantiate model
         model_encoder = get_encoder()

In [42]: a = train_data.take(1)
         english_b, _ = next(iter(a))
         h, c = model_encoder(english_b)
         print(h.shape)
         print(c.shape)

(16, 512)
(16, 512)


In [43]: # model summary
         model_encoder.summary()

Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 13, 128)]         0
_________________________________________________________________
append_layer_1 (AppendLayer) (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
_________________________________________________________________
```

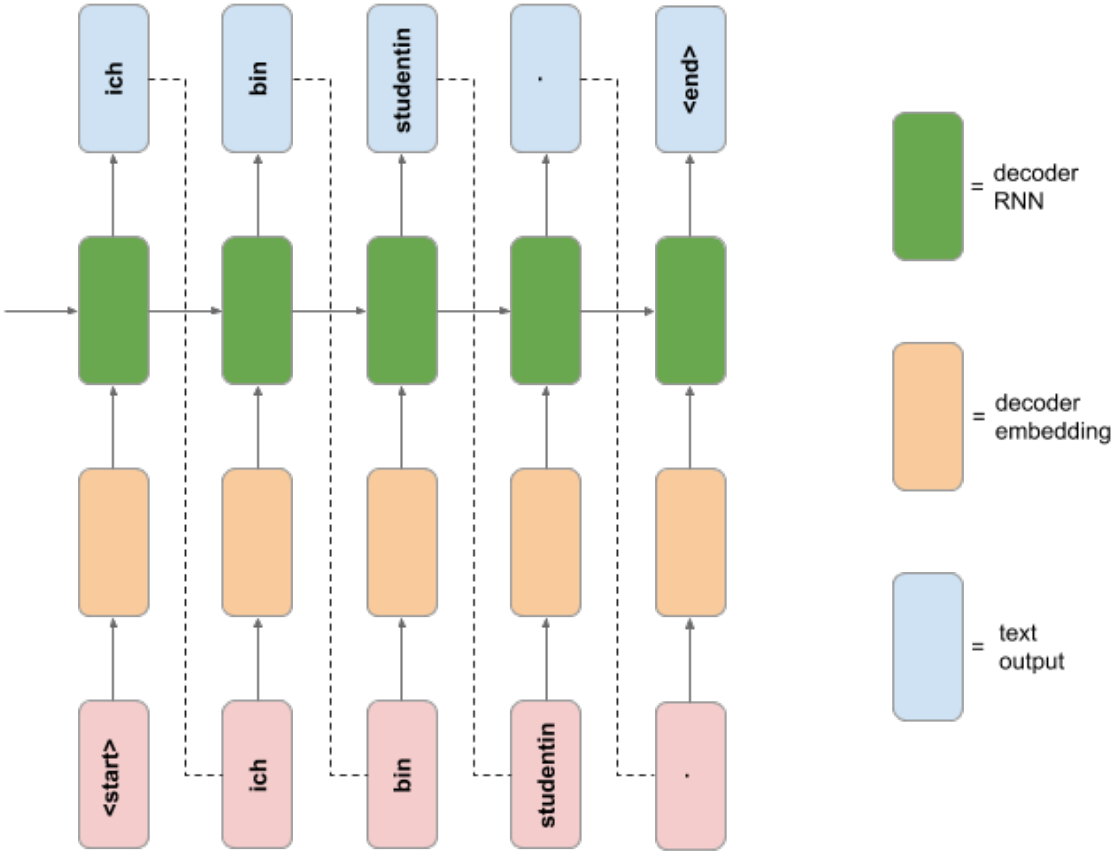
1.6 5. Build the decoder network

The decoder network follows the schematic diagram below.

```
In [55]: # 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.
Image("neural_translation_model.png")
```

Out[55]:



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 [44]: # create decoder model
         class MyDecoder(tf.keras.models.Model):
           def __init__(self, vocab_size, **kwargs):
             super(MyDecoder, self).__init__(**kwargs)
             self.embedded = tf.keras.layers.Embedding(
                 input_dim=vocab_size, output_dim=128, mask_zero=True)
             self.rnn = tf.keras.layers.LSTM(units=512, return_sequences=True,
                                             return_state=True)
             self.fc = tf.keras.layers.Dense(vocab_size, activation=None)

           def call(self, inputs, encoder_states=None):
             h = self.embedded(inputs)
             outputs, hidden, carry = self.rnn(h, initial_state=encoder_states)
             logits = self.fc(outputs)
             return logits, hidden, carry

In [45]: # get vocabulary size from german tokens
         num_tokens_ge = max([max(german_data[i]) for i in range(len(german_data))])
         print(num_tokens_ge)

5744


In [46]: # instantiate decoder model
         model_decoder = MyDecoder(num_tokens_ge+1)

In [47]: # test encoder-decoder
         a = train_data.take(1)
         batch_en, batch_ge = next(iter(a))

In [48]: # check dimensions of data
         print(batch_en.shape)
         print(batch_ge.shape)

(16, 13, 128)
(16, 87)


In [49]: # encode english tokens
         h_encoder, c_encoder = model_encoder(batch_en)
         print(h_encoder.shape)
         print(c_encoder.shape)

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

```
In [50]: # decode german tokens using hidden and carry states from encoder
         logits, h_decoder, c_decoder = model_decoder(batch_ge, [h_encoder, c_encoder])

In [51]: # print out shapes
         print(logits.shape)
         print(h_decoder.shape)
         print(c_decoder.shape)
```

```
(16, 87, 5745)
(16, 512)
(16, 512)
```

```
In [52]: # decoder summary
         model_decoder.summary()
```

```
Model: "my_decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        multiple                  735360
_________________________________________________________________
lstm_1 (LSTM)                multiple                  1312768
_________________________________________________________________
dense (Dense)                multiple                  2947185
=================================================================
Total params: 4,995,313
Trainable params: 4,995,313
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 [53]: # function that generates the german trainin data
         def decoder_inputs_and_targets(batch_sentences):
           input_data = batch_sentences[:, :-1]
           target_data = batch_sentences[:, 1:]
           return (input_data, target_data)
```

```
In [54]: class EncoderDecoder(tf.keras.models.Model):
           def __init__(self, vocab_size, **kwargs):
             super(EncoderDecoder, self).__init__(**kwargs)
             self.decoder = MyDecoder(vocab_size, **kwargs)
             self.encoder = get_encoder()

           def call(self, input_encoder, input_decoder):
             # encode input sentences
             hidden_encoder, carry_encoder = self.encoder(input_encoder)
             # decode input sentences
             logits, _, _ = self.decoder(input_decoder, [hidden_encoder, carry_encoder])
             return logits
```

```
In [95]: # define function that returns logits
         def encoder_decoder(model_encoder, model_decoder, input_encoder, input_decoder):
             # encode input sentences
             hidden_encoder, carry_encoder = model_encoder(input_encoder)
             # decode input sentences
             logits, _, _ = model_decoder(input_decoder,
                                          [hidden_encoder, carry_encoder])
             return logits
```

```
In [55]: # encoder-decoder function
         @tf.function
         def grads_encoder_decoder(model, input_encoder, input_decoder, output_decoder):
           loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
           with tf.GradientTape() as tape:
             # compute logits from sequences
             logits = model(input_encoder, input_decoder)
             # compute loss using logits
             loss_val = loss_fn(output_decoder, logits)
           return loss_val, tape.gradient(loss_val, model.trainable_variables)
```

```
In [72]: # training function
         def main_train(model, train_data,
                        val_data, epochs=20, optimizer=tf.keras.optimizers.Adam()):
```

```
train_acc = []
train_loss = []
val_acc = []
val_loss = []
for epoch in range(epochs):
  # define metrics to be tracked
  train_acc_epoch = tf.keras.metrics.SparseCategoricalAccuracy()
  train_loss_epoch = tf.keras.metrics.Mean()
  val_acc_epoch = tf.keras.metrics.SparseCategoricalAccuracy()
  val_loss_epoch = tf.keras.metrics.Mean()

  # epoch loop
  for input_encoder, target_sec in train_data:
    # extract input and output sequences
    input_decoder, output_decoder = decoder_inputs_and_targets(target_sec)
    # obtain loss and gradients
    loss_val, grads = grads_encoder_decoder(model,
                                            input_encoder,
                                            input_decoder,
                                            output_decoder)
    # apply gradients
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    # track performance values
    logits = model(input_encoder, input_decoder)
    train_acc_epoch(output_decoder, tf.nn.softmax(logits, axis=-1))
    train_loss_epoch(loss_val)

  # validation loop
  for input_encoder, target_sec in val_data:
    # extract input and output sequences
    input_decoder, output_decoder = decoder_inputs_and_targets(target_sec)
    logits = model(input_encoder, input_decoder)
    val_acc_epoch(output_decoder, tf.nn.softmax(logits, axis=-1))
    val_loss_epoch(loss_val)

  #track values
  train_acc.append(train_acc_epoch.result())
  train_loss.append(train_loss_epoch.result())
  val_acc.append(val_acc_epoch.result())
  val_loss.append(val_loss_epoch.result())

  print('Epoch: {0:3d} - Train Acc.: {1:.4f} - Train Loss: {2:.4f} - \
  Val. Acc.: {3:.4f} - Val. Loss: {4:.4f}'.format(epoch+1,
                                                  train_acc_epoch.result(),
                                                  train_loss_epoch.result(),
                                                  val_acc_epoch.result(),
                                                  val_loss_epoch.result()))
```

```
    return (train_acc, train_loss), (val_acc, val_loss)
```

In [63]:
```
# instantiate custom encoder-decoder model
model = EncoderDecoder(num_tokens_ge + 1)
```

In [64]:
```
# need to create and run the model once so that @tf.function can be used
a = train_data.take(1)
s1, s2 = next(iter(a))
s2in, s2out = decoder_inputs_and_targets(s2)
model(s1, s2in).shape
```

Out[64]:
```
TensorShape([16, 86, 5745])
```

In [65]:
```
# test model with small datasets
reduced_train_data = train_data.take(1000)
reduced_val_data = val_data.take(100)
num_epochs = 5
```

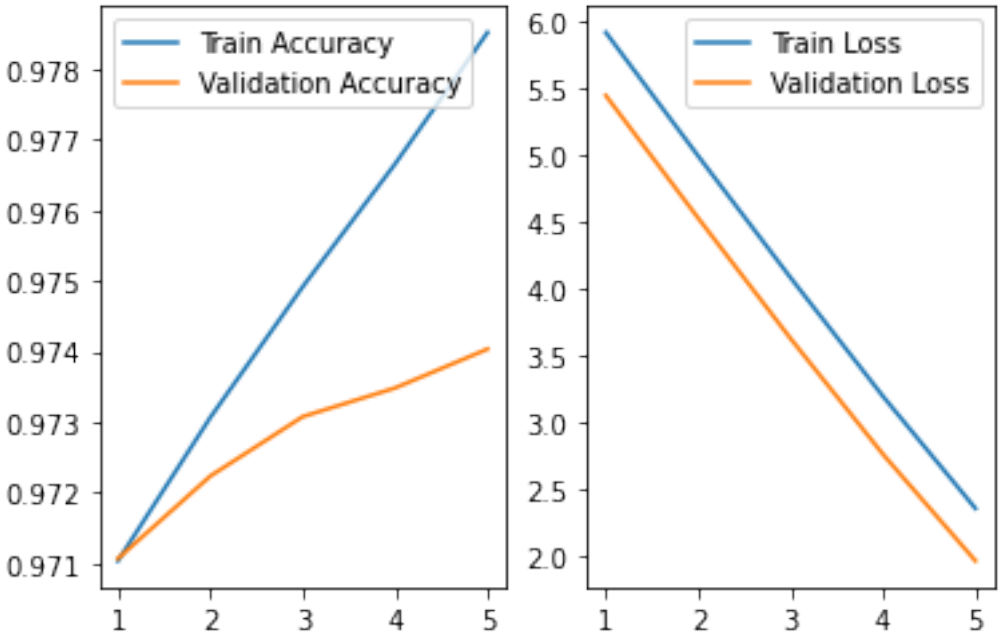
In [66]:
```
weitghs_ok = False
try:
  model.load_weights('/content/gdrive/MyDrive/model_capstone_project_c2')
  weights_ok = True
  print('Models weights have been loaded...')
except:
  pass
```

In [69]:
```
if not weights_ok:
  optimizer = tf.keras.optimizers.Adam(learning_rate=0.0005)
  (train_acc, train_loss), (val_acc, val_loss) = main_train(model,
                                                            reduced_train_data,
                                                            reduced_val_data,
                                                            epochs=num_epochs,
                                                            optimizer=optimizer)
```

```
Epoch:   0 - Train Acc.: 0.9710 - Train Loss: 5.9096 -     Val. Acc.: 0.9711 - Val. Loss: 5.438
Epoch:   1 - Train Acc.: 0.9731 - Train Loss: 4.9857 -     Val. Acc.: 0.9722 - Val. Loss: 4.521
Epoch:   2 - Train Acc.: 0.9749 - Train Loss: 4.0762 -     Val. Acc.: 0.9731 - Val. Loss: 3.621
Epoch:   3 - Train Acc.: 0.9767 - Train Loss: 3.1898 -     Val. Acc.: 0.9735 - Val. Loss: 2.757
Epoch:   4 - Train Acc.: 0.9785 - Train Loss: 2.3525 -     Val. Acc.: 0.9740 - Val. Loss: 1.960
```

In [70]:
```
model.save_weights('/content/gdrive/MyDrive/model_capstone_project_c2')
```

In [71]:
```
import matplotlib.pyplot as plt
fig, axis = plt.subplots(1,2)
axis[0].plot(np.arange(1, num_epochs+1), train_acc, label='Train Accuracy')
axis[0].plot(np.arange(1, num_epochs+1), val_acc, label='Validation Accuracy')
axis[0].legend()
axis[1].plot(np.arange(1, num_epochs+1), train_loss, label='Train Loss')
axis[1].plot(np.arange(1, num_epochs+1), val_loss, label='Validation Loss')
axis[1].legend()
plt.show()
```



In [ ]:

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 [73]: # select five randomly segments
         num_sentences = 5
         indices = np.random.choice(len(english_list), num_sentences, replace=False)

In [74]: sel_english_sentences = [english_list[i] for i in indices]
         test_data = tf.data.Dataset.from_tensor_slices((sel_english_sentences,
                                                         padded_german_list[indices]))
         # preprocess steps for english sentences
         max_len_en = 13
         test_data = split_english_tokens(test_data)
         test_data = embedd_english_tokens(test_data)
```

```
test_data = chunk_embedded_english_sentences(test_data, max_len_en)
test_data = padding_english_token(test_data, max_len_en)
test_data = test_data.batch(1)
```

In [109]:
```
# process sentence by sentence
max_ge_length = 50
ge_generated_tokens = []
for en_sentence, ge_sentence in test_data:
  # get embedded input sequence
  hidden, carry = model.encoder(en_sentence)
  # go over the german sentence until <end>
  ge_token = ge_sentence[0,0].numpy() # it should be the token for <start>
  ge_generated_loop = [ge_token] # append to the translated sentence
  ge_length = 1 # counting the number of translated words
  while (ge_token != 3) and (ge_length < max_ge_length):
    logits, hidden, carry = model.decoder(np.expand_dims([ge_token], 0),
                                          [hidden, carry])
    # emit one token
    ge_token = tf.random.categorical(logits[0], 1)[0].numpy()[0]
    # record into generated tokens
    ge_generated_loop.append(ge_token)
    # increment number of generated words
    max_ge_length += 1
  # append translated sentence
  ge_generated_tokens.append(ge_generated_loop)
```

In [114]:
```
# go over the english sentences and its translations
for i in range(num_sentences):
  print('English sentence: {}'.format(sel_english_sentences[i]))
  print('German translation: {}'.format(
      tokenizer.sequences_to_texts([ge_generated_tokens[i]])[0]))
# print(german_list[indices[0]])
# print(tokenized_german[indices[0]])
# print(padded_german_list[indices[0]])
```

```
English sentence: do you like juice ?
German translation: <start> kennt sie mich ? <end>
English sentence: does he come here ?
German translation: <start> kennt sie sie ? <end>
English sentence: i'm very happy .
German translation: <start> ich habe pech nur sicher . <end>
English sentence: tom is unstable .
German translation: <start> tom ist anpassungsfaehig . <end>
English sentence: we saw it .
German translation: <start> wir haben ihn gesehen . <end>
```

In [ ]: