Capstone_Project

January 8, 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.

```
[1]: import tensorflow as tf
     import tensorflow_hub as hub
     import unicodedata
     import re
     from IPython.display import Image
     import numpy as np
     import json
     from tensorflow.keras.preprocessing.text import Tokenizer
     from tensorflow.keras.preprocessing.sequence import pad_sequences
     from tensorflow.keras.layers import Layer, Input, Dense, LSTM, Masking, Embedding
```

```python
from tensorflow.keras.models import Model
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

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 pretrained 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.

```python
[2]: # Run this cell to connect to your Drive folder

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

Mounted at /content/gdrive

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

     NUM_EXAMPLES = 20000
     data_examples = []
     with open('gdrive/MyDrive/Colab Notebooks/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
```

```python
[4]: data_examples[:10]
```

```
[4]: ['Hi.\tHallo!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) &
     #380701 (cburgmer)\n',
      'Hi.\tGrüß Gott!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) &
     #659813 (Esperantostern)\n',
      'Run!\tLauf!\tCC-BY 2.0 (France) Attribution: tatoeba.org #906328 (papabear) &
     #941078 (Fingerhut)\n',
      'Wow!\tPotzdonner!\tCC-BY 2.0 (France) Attribution: tatoeba.org #52027 (Zifre)
     & #2122382 (Pfirsichbaeumchen)\n',
      'Wow!\tDonnerwetter!\tCC-BY 2.0 (France) Attribution: tatoeba.org #52027
```

```
(Zifre) & #2122391 (Pfirsichbaeumchen)\n',
 'Fire!\tFeuer!\tCC-BY 2.0 (France) Attribution: tatoeba.org #1829639 (Spamster)
& #1958697 (Tamy)\n',
 'Help!\tHilfe!\tCC-BY 2.0 (France) Attribution: tatoeba.org #435084 (lukaszpp)
& #575889 (MUIRIEL)\n',
 'Help!\tZu Hülf!\tCC-BY 2.0 (France) Attribution: tatoeba.org #435084
(lukaszpp) & #2122375 (Pfirsichbaeumchen)\n',
 'Stop!\tStopp!\tCC-BY 2.0 (France) Attribution: tatoeba.org #448320
(FeuDRenais) & #626467 (jakov)\n',
 'Wait!\tWarte!\tCC-BY 2.0 (France) Attribution: tatoeba.org #1744314 (belgavox)
& #2122378 (Pfirsichbaeumchen)\n']
```

[5]:
```
data_examples[-1]
```

[5]:
```
"I've eaten enough.\tIch habe genug gegessen.\tCC-BY 2.0 (France) Attribution:
tatoeba.org #6803318 (CK) & #5287295 (raggione)\n"
```

[6]:
```python
# 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.

[7]:
```python
# Run this cell to download and view a schematic diagram for the neural␣
 ↪translation model

!wget -q -O neural_translation_model.png --no-check-certificate "https://docs.
 ↪google.com/uc?export=download&id=1XsS1VlXoaEo-RbYNilJ9jcscNZvsSPmd"
Image("neural_translation_model.png")
```

[7]:
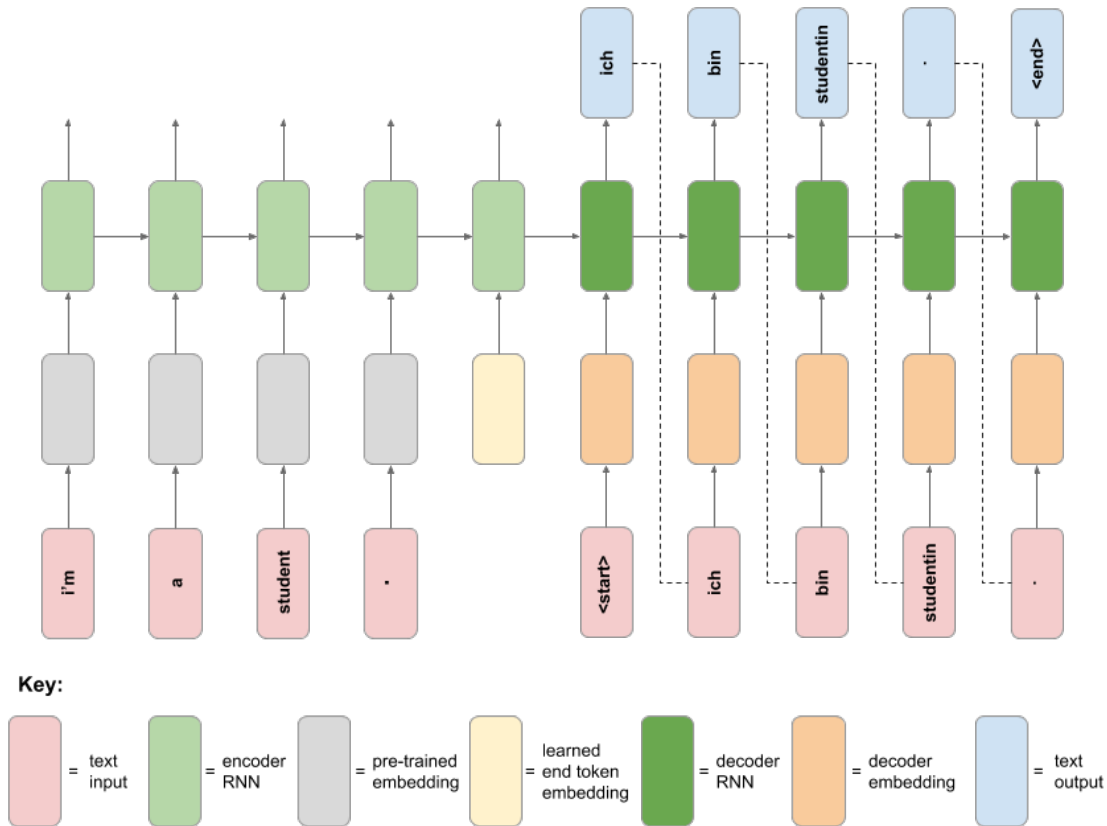
The custom model consists of an encoder RNN and a decoder RNN. The encoder takes words of an English sentence as input, and uses a pre-trained word embedding to embed the words into a 128-dimensional space. To indicate the end of the input sentence, a special end token (in the same 128-dimensional space) 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.

```
[8]: eng_sentences = list()
     ger_sentences = list()

     for i in range(len(data_examples)):
         eng_sentences.append(preprocess_sentence(data_examples[i].split('\t')[0]))
         ger_sentences.append('<start> ' + preprocess_sentence(data_examples[i].
      →split('\t')[1]) + ' <end>')
```

```
[9]: print(eng_sentences[-3:])
     print(ger_sentences[-3:])
```

```
["i've come for tom .", "i've done nothing .", "i've eaten enough ."]
['<start> ich bin fuer tom gekommen . <end>', '<start> ich habe nichts getan .
<end>', '<start> ich habe genug gegessen . <end>']
```

```
[10]: tokenizer = Tokenizer(filters='')
      tokenizer.fit_on_texts(ger_sentences)
      german_tokenized = tokenizer.texts_to_sequences(ger_sentences)
```

```
[11]: german_tokenized_pad_seq = pad_sequences(german_tokenized, padding='post')
```

```
[12]: samples = np.random.choice(len(eng_sentences), 5)
      print('Random samples: {}'.format(samples))

      for sample in samples:
          print(eng_sentences[sample], ger_sentences[sample],␣
       →german_tokenized_pad_seq[sample])
```

```
Random samples: [14732   425 18218 13549  4514]
i love your name . <start> ich liebe ihren namen . <end> [  1   4  61 510 661
  3   2   0   0   0   0   0   0   0]
tell tom . <start> erzaehl es tom . <end> [   1 1090   10    5    3    2    0
  0   0   0   0   0   0   0]
choose one person . <start> waehlt eine person aus . <end> [   1 2434   37 1754
 41    3    2    0    0    0    0    0    0    0]
can we afford it ? <start> koennen wir uns das erlauben ? <end> [   1   81   17
 31   11 4716    7    2    0    0    0    0    0    0]
call security . <start> ruf den sicherheitsdienst ! <end> [   1  406   53 1165
  9    2    0    0    0    0    0    0    0    0]
```

```
[13]: german_tokenized_pad_seq.shape
```

[13]: (20000, 14)

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.

```
[14]: # Load embedding module from Tensorflow Hub

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

```
[15]: # Test the layer

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

[15]: 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.

```python
[16]: x_train, x_val, y_train, y_val = train_test_split(eng_sentences,
       ↪german_tokenized_pad_seq, test_size=0.2)
```

```python
[17]: train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
      val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
```

```python
[18]: def map_split_dataset(x, y):
          return tf.strings.split(x, ' '), y
```

```python
[19]: train_dataset = train_dataset.map(map_split_dataset)
      val_dataset = val_dataset.map(map_split_dataset)
```

```python
[20]: def map_embbed_dataset(x, y):
          return embedding_layer(x), y
```

```python
[21]: train_dataset = train_dataset.map(map_embbed_dataset)
      val_dataset = val_dataset.map(map_embbed_dataset)
```

```python
[22]: train_dataset = train_dataset.filter(lambda x, y: False if ((tf.shape(x)[0]) >
       ↪13) else True)
      val_dataset = val_dataset.filter(lambda x, y: False if ((tf.shape(x)[0]) > 13)
       ↪else True)
```

```python
[23]: def map_pad_dataset(x, y):
          return tf.pad(x, paddings=[[13 - len(x), 0], [0,0]]), y
```

```python
[24]: train_dataset = train_dataset.map(map_pad_dataset)
      val_dataset = val_dataset.map(map_pad_dataset)
```

```python
[25]: train_dataset = train_dataset.batch(16, drop_remainder=True)
      val_dataset = val_dataset.batch(16, drop_remainder=True)
```

```python
[26]: print(train_dataset.element_spec)
      print(val_dataset.element_spec)
```

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

[27]:
```python
tmp_train_data = next(iter(train_dataset.take(1)))
tmp_val_data = next(iter(val_dataset.take(1)))
```

[28]:
```python
print(tmp_train_data[0].shape)
print(tmp_val_data[1].shape)
```

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

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:

[29]:
```python
# Run this cell to download and view a schematic diagram for the encoder model

!wget -q -O neural_translation_model.png --no-check-certificate "https://docs.
 ↪google.com/uc?export=download&id=1JrtNOzUJDaOWrK4C-xv-4wUuZaI12sQI"
Image("neural_translation_model.png")
```

[29]:
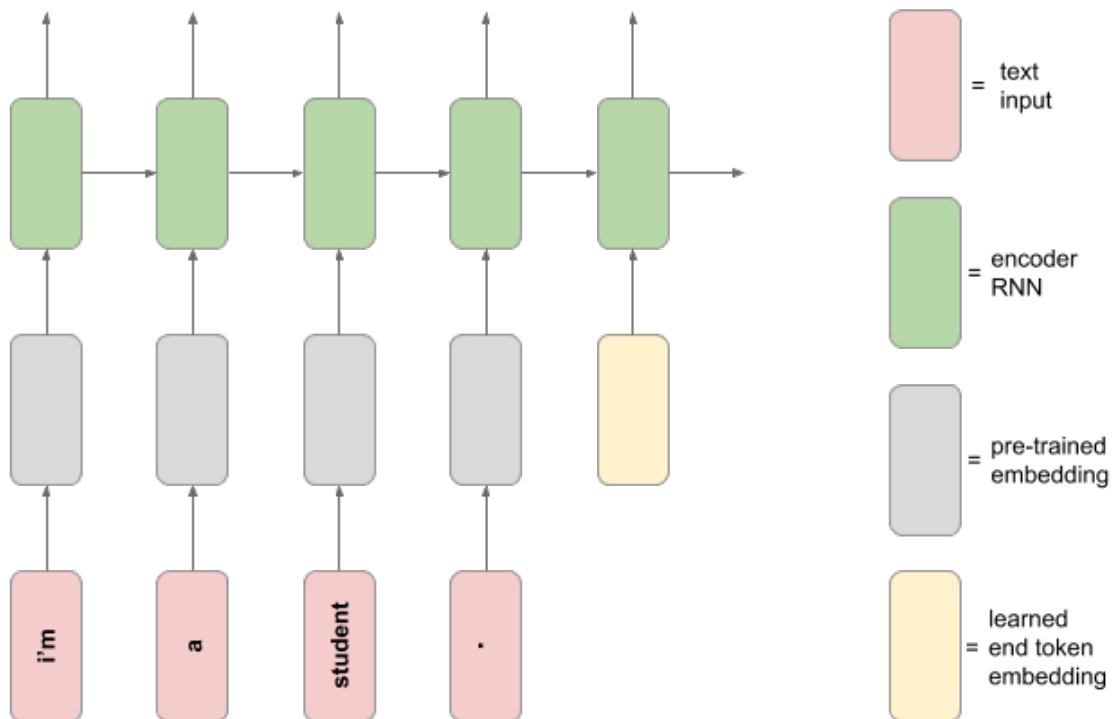
You should now build the custom layer. * Using layer subclassing, create a custom
layer that takes a batch of English data examples from one of the Datasets, and
adds a learned embedded 'end' token to the end of each sequence. * This layer
should create a TensorFlow Variable (that will be learned during 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).

```python
[30]: class MyLayer(Layer):
          def __init__(self):
              super(MyLayer, self).__init__()
              self.emb_space = tf.Variable(initial_value=tf.random.
        →uniform(shape=(128,)), trainable=True)

          def call(self, inputs):
              output = tf.tile(tf.reshape(self.emb_space, shape=(1, 1, tf.shape(self.
        →emb_space)[0])), [tf.shape(inputs)[0], 1, 1])
              return tf.keras.layers.concatenate([inputs, output], axis=1)

      mylayer = MyLayer()
```

```python
[31]: print(tmp_train_data[0].shape)
      print(mylayer(tmp_train_data[0]).shape)
```

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

```
[32]: inputs = Input(batch_shape=(None, 13, 128))
      x = mylayer(inputs)
```

```
[33]: x = Masking(mask_value = 0)(x)
      x, hidden_s, cell_s = LSTM(512, return_state=True)(x)
      EncoderModel = Model(inputs=inputs, outputs=[hidden_s, cell_s])
```

```
[34]: EncoderModel.summary()
```

```
Model: "model"
_____
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 13, 128)]         0
_____
my_layer (MyLayer)           (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
_____
```
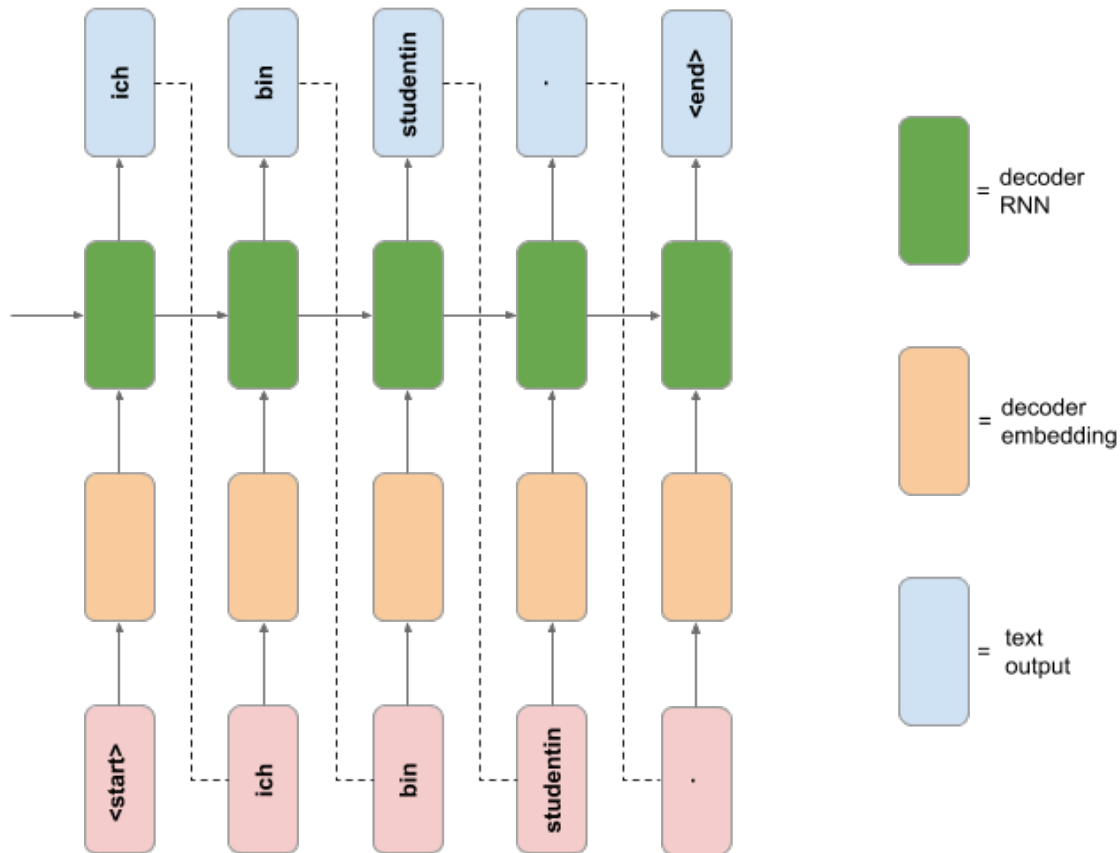
1.6 5. Build the decoder network

The decoder network follows the schematic diagram below.

```
[35]: # Run this cell to download and view a schematic diagram for the decoder model

      !wget -q -O neural_translation_model.png --no-check-certificate "https://docs.
       ↪google.com/uc?export=download&id=1DTeaXD8tA8RjkpVrB2mr9csSBOY4LQiW"
      Image("neural_translation_model.png")
```

```
[35]:
```

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.

```python
[36]: class DecoderModel(Model):
          def __init__(self):
              super(DecoderModel, self).__init__()
              self.layer_1 = Embedding(input_dim=len(tokenizer.word_index) + 1,
      ↪output_dim=128, mask_zero=True)
              self.layer_2 = LSTM(512, return_sequences=True, return_state=True)
              self.layer_3 = Dense(len(tokenizer.word_index) + 1)

          def call(self, inputs, hidden_state=None, cell_state=None):
              x = self.layer_1(inputs)
              if hidden_state is not None and cell_state is not None:
                  x, hidden_s, cell_s = self.layer_2(x, initial_state=[hidden_state,
      ↪cell_state])
              else:
                  x, hidden_s, cell_s = self.layer_2(x)
              x = self.layer_3(x)
              return x, hidden_s, cell_s

      decoder = DecoderModel()
```

```python
[37]: enc_hidden_s, enc_cell_s = EncoderModel(tmp_train_data[0])
      dec_output, dec_hidden_s, dec_cell_s = decoder(tmp_train_data[1], enc_hidden_s,
       ↪enc_cell_s)
      print(dec_output.shape)
      print(dec_hidden_s.shape)
      print(dec_cell_s.shape)
```

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

```python
[38]: decoder.summary()
```

```
Model: "decoder_model"
_____
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
_____
```

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.

```
[40]: tf.config.list_physical_devices('GPU')
```

```
[40]: [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

```
[41]: optimizer_obj = tf.keras.optimizers.Adam()
      loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
      variables = EncoderModel.trainable_variables + decoder.trainable_variables
```

```
[42]: @tf.function
      def grad_fn(en_input, ger_input, ger_output, loss):
          with tf.GradientTape() as tape:
              enc_hidden_s, enc_cell_s = EncoderModel(en_input)
              dec_output, dec_hidden_s, dec_cell_s = decoder(ger_input, enc_hidden_s,␣
       ↪enc_cell_s)

              loss_value = loss(ger_output, dec_output)
          return loss_value, tape.gradient(loss_value, variables)
```

```
[50]: def train_model(num_epochs, train_dataset, val_dataset, grad_fn, optimizer,␣
       ↪loss):
```

```python
    train_loss_results = []
    val_loss_results = []

    for epoch in range(num_epochs):
        epoch_loss_avg = tf.keras.metrics.Mean()
        val_epoch_loss_avg = tf.keras.metrics.Mean()

        for en, ger in train_dataset:
            dec_inputs, dec_outputs = ger[:, :-1], ger[:, 1:]
            loss_value, grads= grad_fn(en, dec_inputs, dec_outputs, loss)
            optimizer.apply_gradients(zip(grads, variables))
            epoch_loss_avg(loss_value)

        for en, ger in val_dataset:
            dec_inputs, dec_outputs = ger[:, :-1], ger[:, 1:]

            hidden_s, cell_s = EncoderModel(en)
            model_preds, _, _ = decoder(dec_inputs, hidden_s, cell_s)
            val_epoch_loss_avg(loss(dec_outputs, model_preds))

        train_loss_results.append(epoch_loss_avg.result())
        val_loss_results.append(val_epoch_loss_avg.result())
        # print(" Epoch :{:03d}: Loss: {:.3f} : Val_Loss: {:.3f}".format(epoch,
    →epoch_loss_avg, val_epoch_loss_avg))

    return train_loss_results,val_loss_results
```
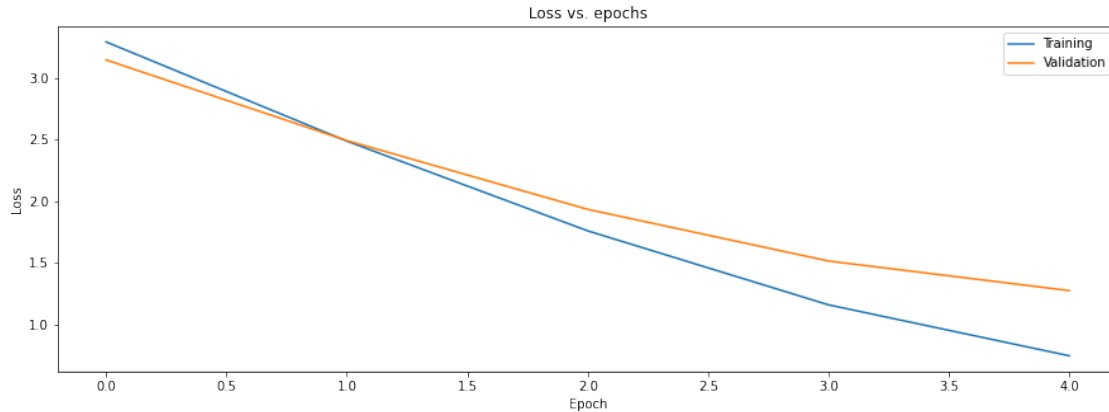
```python
[51]: train_loss_results, val_loss_results = train_model(5, train_dataset,
      →val_dataset, grad_fn, optimizer_obj, loss_obj)
```

```python
[52]: plt.figure(figsize=(15, 5))
      plt.plot(train_loss_results)
      plt.plot(val_loss_results)
      plt.title('Loss vs. epochs')
      plt.ylabel('Loss')
      plt.xlabel('Epoch')
      plt.legend(['Training', 'Validation'])
      plt.show()
```

Loss vs. epochs

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.

```python
[53]: samples = np.random.choice(len(eng_sentences), 5, replace=False)
```

```python
[63]: for i, sample in enumerate(samples):
          en = tf.strings.split(eng_sentences[sample])
          en_emb = embedding_layer(en)
          padding = [[tf.math.maximum(13-tf.shape(en_emb)[0],0), 0], [0,0]]
          en_emb_pad = tf.expand_dims(tf.pad(en_emb, padding), axis=0)
          result = list()
          hidden_s, cell_s = EncoderModel(en_emb_pad)
          dec_input = tf.Variable([[tokenizer.word_index['<start>']]])

          for t in range(train_dataset.element_spec[1].shape[1]):
              dec_output, hidden_s, cell_s = decoder(dec_input, hidden_s, cell_s)
              dec_output = tf.squeeze(tf.argmax(dec_output, axis=2)).numpy()

              if dec_output == tokenizer.word_index['<end>']:
                  break
```

```
        result.append(tokenizer.index_word[dec_output])
        dec_input = tf.Variable([[dec_output]])

    result = ' '.join(word for word in result)
    print("Sample: {}".format(i + 1))
    print("English Sentence: {}".format(eng_sentences[sample]))
    print("German Sentence: {}".format(ger_sentences[sample]))
    print("German Translation: {}".format(result))
    print('-' * 5)
```

```
Sample: 1
English Sentence: the sky was gray .
German Sentence: <start> der himmel war grau . <end>
German Translation: der himmel war grau .
-----
Sample: 2
English Sentence: did tom object ?
German Sentence: <start> hatte tom etwas einzuwenden ? <end>
German Translation: hat tom es versucht ?
-----
Sample: 3
English Sentence: drop your weapons !
German Sentence: <start> lasst eure waffen fallen ! <end>
German Translation: werfen sie ihre waffen weg !
-----
Sample: 4
English Sentence: tom is a drummer .
German Sentence: <start> tom ist schlagzeuger . <end>
German Translation: tom ist pianist .
-----
Sample: 5
English Sentence: tom was hesitant .
German Sentence: <start> tom zoegerte . <end>
German Translation: tom war ungluecklich .
-----
```

[ ]: