Capstone_Project

December 5, 2020

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.

[43]:
```
import tensorflow as tf
import tensorflow_hub as hub
import unicodedata
import re
from IPython.display import Image
import pandas as pd
```

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.

[44]:
```
# Run this cell to connect to your Drive folder

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

[45]:
```
# Run this cell to load the dataset
NUM_EXAMPLES = 20000
data_examples = []
with open('./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
data_examples[:10]
```

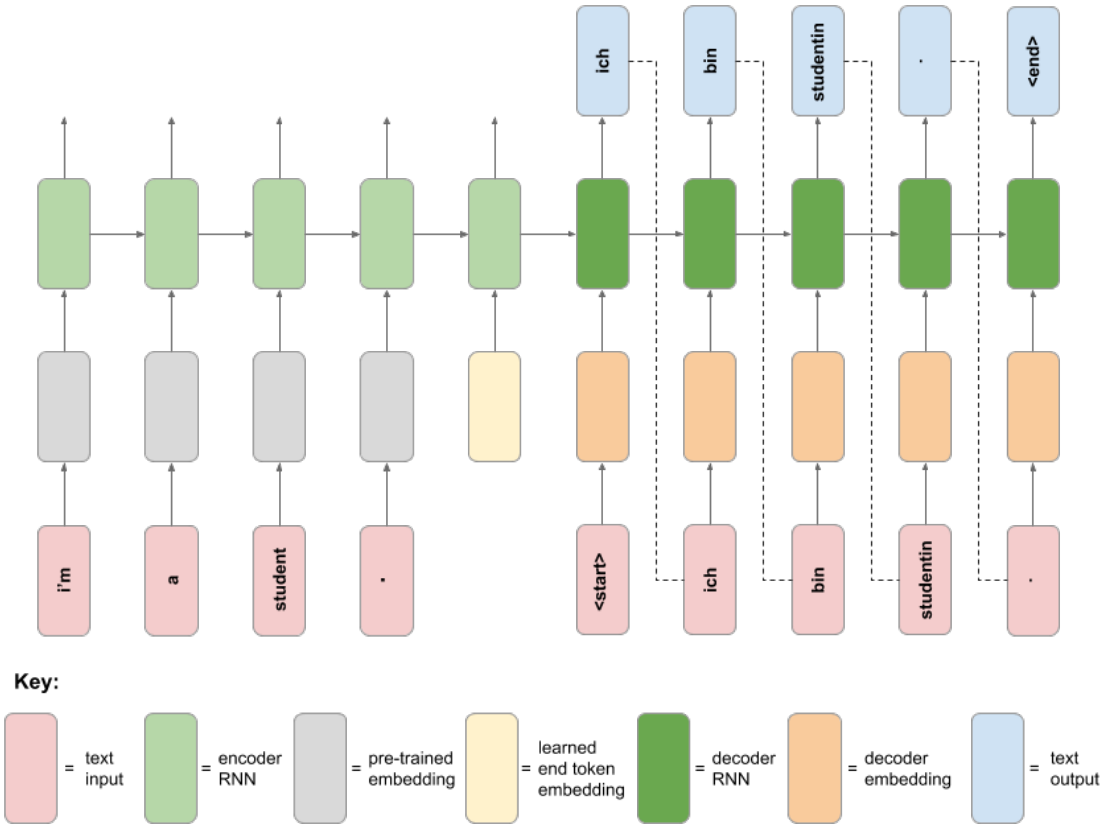
[45]:
```
['Hi.\tHallo!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) &
#380701 (cburgmer)\n',
 'Hi.\tGrüß Gott!\tCC-BY 2.0 (France) Attribution: tatoeba.org #538123 (CM) &
#659813 (Esperantostern)\n',
 'Run!\tLauf!\tCC-BY 2.0 (France) Attribution: tatoeba.org #906328 (papabear) &
#941078 (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']
```

[46]:
```
# 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.

[47]:
```
# 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")
```

[47]:



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.

[48]:
```
#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.

english_sentences = []
german_sentences = []

for i in data_examples:

    # extract sentences
    english_sentence, german_sentence = i.split("CC-BY 2.0")[0].split("\t")[:2]

    # preprocess english
    english_sentence = preprocess_sentence(english_sentence)
    english_sentences.append(english_sentence)

    # preprocess Garman sentences
    german_sentence = preprocess_sentence(german_sentence)
    german_sentence = "<start> " + german_sentence + " <end>"
    german_sentences.append(german_sentence)
```

[49]:
```
#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._
from tensorflow.keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer(filters = "")
tokenizer.fit_on_texts(german_sentences)
```

[50]:
```
#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.
import random
for i in range(5):
    idx = random.randint(0,20000)
    print(f"======== sentence No. {idx}==========")
    print(f"ENGLISH:\t\t{english_sentences[idx]}")
    print(f"GERMAN:\t\t\t{german_sentences[idx]}")
    print(f"TOKENIZED GERMAN:\t{tokenizer.
 ↪texts_to_sequences([german_sentences[idx]])[0]}")
    print()
```

```
======== sentence No. 10680==========
ENGLISH:                i still do that .
GERMAN:                 <start> ich tue das immer noch . <end>
TOKENIZED GERMAN:       [1, 4, 454, 11, 192, 72, 3, 2]

======== sentence No. 8978==========
ENGLISH:                tom was framed .
GERMAN:                 <start> tom wurde reingelegt . <end>
TOKENIZED GERMAN:       [1, 5, 68, 1629, 3, 2]

======== sentence No. 5171==========
ENGLISH:                i'm a student .
GERMAN:                 <start> ich bin schueler . <end>
TOKENIZED GERMAN:       [1, 4, 15, 1172, 3, 2]

======== sentence No. 9557==========
ENGLISH:                you're winning .
GERMAN:                 <start> sie sind am gewinnen . <end>
TOKENIZED GERMAN:       [1, 8, 23, 146, 227, 3, 2]

======== sentence No. 9034==========
ENGLISH:                tom will fight .
GERMAN:                 <start> tom wird kaempfen . <end>
TOKENIZED GERMAN:       [1, 5, 48, 929, 3, 2]
```

[51]:
```
#Pad the end of the tokenized German sequences with zeros, and batch the␣
 ↪complete set of sequences into one numpy array.
from tensorflow.keras.preprocessing.sequence import pad_sequences
german_sequences = tokenizer.texts_to_sequences(german_sentences)
np_german_sequences = pad_sequences(german_sequences, padding="post")
MAX_OUTPUT_LENGTH = np_german_sequences.shape[1]
```

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.

[52]:
```
# 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)
```

[53]:
```
# Test the layer
embedding_layer(tf.constant(["these", "aren't", "the", "droids", "you're",␣
↪"looking", "for"])).shape
```

[53]:
```
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.

[54]:
```
#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).

def train_test_split(english_sentences, np_german_sequences, test_ratio = 0.2):

    assert(len(english_sentences) == np_german_sequences.shape[0])
```

```
    # shuffle both english and german sentences following the same order.
    idx_list = list(range(len(english_sentences)))
    random.seed(0)
    random.shuffle(idx_list)
    english_sentences = [english_sentences[i] for i in idx_list]
    np_german_sequences = np_german_sequences[idx_list]

    # number of test samples
    train_num = int((1-test_ratio) * len(english_sentences))

    # training dataset
    english_sentences_train = english_sentences[:train_num]
    np_german_sequences_train = np_german_sequences[:train_num]

    # testing dataset
    english_sentences_test = english_sentences[train_num:]
    np_german_sequences_test = np_german_sequences[train_num:]

    return (english_sentences_train,
            np_german_sequences_train,
            english_sentences_test,
            np_german_sequences_test)

(english_sentences_train, \
 np_german_sequences_train, \
 english_sentences_test,  \
 np_german_sequences_test) = train_test_split(english_sentences,␣
 ↪np_german_sequences)
```

[55]:
```
# 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.

from tensorflow.data import Dataset
ds_train = Dataset.
 ↪from_tensor_slices((english_sentences_train,np_german_sequences_train))
ds_test = Dataset.
 ↪from_tensor_slices((english_sentences_test,np_german_sequences_test))
```

[56]:
```
#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._
def helper_split(x):
    return tf.strings.split(input=x, sep =" ")

ds_train = ds_train.map(lambda x,y: (helper_split(x),y))
ds_test = ds_test.map(lambda x,y: (helper_split(x),y))
```

[57]:
```
# 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.
def helper_embedding(x):
    return embedding_layer(x)
ds_train = ds_train.map(lambda x,y:(helper_embedding(x),y))
ds_test = ds_test.map(lambda x,y:(helper_embedding(x),y))
```

[58]:
```
# 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.
def filter_fun_less_than_13(x,y):
    return tf.shape(x)[0] < 13
ds_train = ds_train.filter(filter_fun_less_than_13)
ds_test = ds_test.filter(filter_fun_less_than_13)
```

[59]:
```
# 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._

# pad with -10
DISTINCT_VALUE_PADDING_ENGLISH = -10.0
def pad_english_to_len_13(x,y):
    paddings = tf.constant([[1, 0,], [0, 0]]) * (13-tf.shape(x)[0])
    x = tf.pad(x, paddings, "CONSTANT", constant_values =␣
 ↪DISTINCT_VALUE_PADDING_ENGLISH)
    return (x,y)

ds_train = ds_train.map(pad_english_to_len_13)
ds_test = ds_test.map(pad_english_to_len_13)
```

[60]:
```
# Batch both training and validation Datasets with a batch size of 16.
ds_train = ds_train.batch(16)
ds_test = ds_test.batch(16)
```

[61]:
```
# * Print the `element_spec` property for the training and validation Datasets.
print(ds_train.element_spec)
print(ds_test.element_spec)
```

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

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

[62]:
```
# * Using the Dataset `.take(1)` method, print the shape of the English data␣
 ↪example from the training Dataset.
for one_english_sentence,_ in ds_train.take(1):
    print(one_english_sentence.shape)

for one_english_sentence,_ in ds_test.take(1):
    print(one_english_sentence.shape)
```

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

[63]:
```
# * Using the Dataset `.take(1)` method, print the German data example Tensor␣
 ↪from the validation Dataset.
for _, one_german_sentence in ds_train.take(1):
    print(one_german_sentence)

for _, one_german_sentence in ds_test.take(1):
    print(one_german_sentence)
```

```
tf.Tensor(
[[   1    5 2005    3    2    0    0    0    0    0    0    0    0    0]
 [   1    4  133   65   75    3    2    0    0    0    0    0    0    0]
 [   1    5    6  194 1145    3    2    0    0    0    0    0    0    0]
 [   1   23    8   12  409    7    2    0    0    0    0    0    0    0]
 [   1    5  136  135  269    3    2    0    0    0    0    0    0    0]
 [   1   17   35  118   19 1041 2726    3    2    0    0    0    0    0]
 [   1   50  637 5022    3    2    0    0    0    0    0    0    0    0]
 [   1    5    6 1333    3    2    0    0    0    0    0    0    0    0]
 [   1    5   24 2935    3    2    0    0    0    0    0    0    0    0]
 [   1  158    6   49    3    2    0    0    0    0    0    0    0    0]
 [   1    5 1412    3    2    0    0    0    0    0    0    0    0    0]
 [   1   21  150   47  310  181    3    2    0    0    0    0    0    0]
 [   1    4   18    5  346    3    2    0    0    0    0    0    0    0]
 [   1   10  150   31  112  289    3    2    0    0    0    0    0    0]
 [   1   14   24  279    3    2    0    0    0    0    0    0    0    0]
 [   1   11 1369   24   21  513    3    2    0    0    0    0    0    0]],
shape=(16, 14), dtype=int32)
tf.Tensor(
[[   1    5    6   12  467    3    2    0    0    0    0    0    0    0]
 [   1   43   32   13    7    2    0    0    0    0    0    0    0    0]
 [   1 1454   26  269    3    2    0    0    0    0    0    0    0    0]
 [   1   13   32   70  273    9    2    0    0    0    0    0    0    0]
 [   1   30    4   33 1256    7    2    0    0    0    0    0    0    0]
 [   1   10    6 3755    3    2    0    0    0    0    0    0    0    0]
 [   1    4   15  894    3    2    0    0    0    0    0    0    0    0]
 [   1   14  136  238   29   26 1293    3    2    0    0    0    0    0]
```

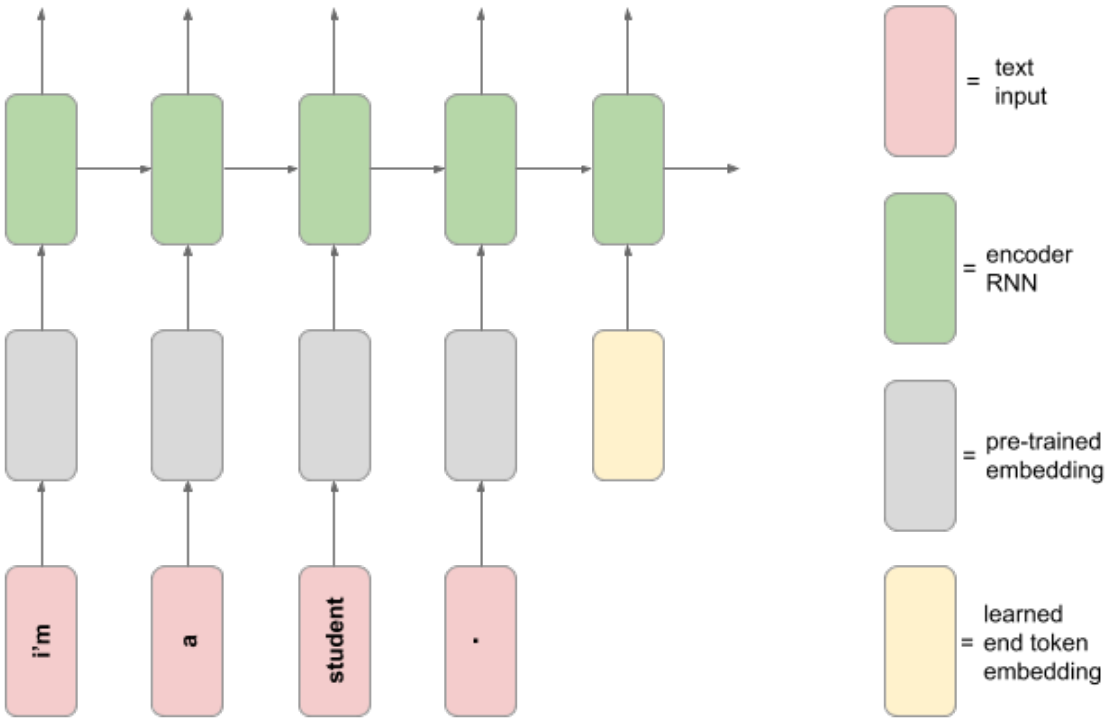
```
 [   1  622    8   21    9    2    0    0    0    0    0    0    0    0]
 [   1  264    8 1827    9    2    0    0    0    0    0    0    0    0]
 [   1    5   68 1629    3    2    0    0    0    0    0    0    0    0]
 [   1    4   18  121   88    3    2    0    0    0    0    0    0    0]
 [   1   10   24  740    3    2    0    0    0    0    0    0    0    0]
 [   1    4   24   55    5  560  588    3    2    0    0    0    0    0]
 [   1   13   58   22 2437    3    2    0    0    0    0    0    0    0]
 [   1    5   24   12  188    3    2    0    0    0    0    0    0    0]],
shape=(16, 14), dtype=int32)
```

2 3. Create the custom layer

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

[64]:
```
# 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")
```

[64]:



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).

[65]:
```
#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._

from tensorflow.keras.layers import Layer

class AddEndEmbeddingLayer(Layer):

    def __init__(self):
        super(AddEndEmbeddingLayer,self).__init__()
        self.end_token_embedding = self.add_weight(    shape=(128,),
                                                   ␣
↪initializer='random_normal',
                                                    trainable=True
                                                )

    def call(self, inputs):

        # expend dimension from (128,) to (1,1,128)
        end_token_embedding = tf.expand_dims(
            tf.expand_dims(
                self.end_token_embedding, axis=0
            ), axis=0
        )

        # tile the dimension from (1,1,128) to (batch_size,1,128)
        inputs_shape = inputs.shape
        end_token_embedding = tf.tile(end_token_embedding, tf.
↪constant([inputs_shape[0],1,1], tf.int32))

        # concat the embedding (batch_size, 1,128) to (batch_size, 13, 128),␣
↪get (batch_size, 14,128)
        return tf.concat([inputs, end_token_embedding], axis = 1 )
```

[66]:
```
# * 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).

layer_add_end_embedding = AddEndEmbeddingLayer()
for english_single_batch,_ in ds_train.take(1):
    print(f"english_single_batch.shape = {english_single_batch.shape}")
    post_layer_result = layer_add_end_embedding(english_single_batch)
    print(f"after the layer layer_add_end_embedding, the shape  =␣
 ↪{post_layer_result.shape}")
```

```
english_single_batch.shape = (16, 13, 128)
after the layer layer_add_end_embedding, the shape  = (16, 14, 128)
```

2.1 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.

[67]:
```
"""
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.
"""

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

input_tensor = Input(shape = (13,128), batch_size = 16, dtype=tf.float32, name␣
↪= "input_layer")
x = input_tensor
x = AddEndEmbeddingLayer()(x)
x = Masking(mask_value = DISTINCT_VALUE_PADDING_ENGLISH, name =␣
↪"masking_layer")(x)
_, state_h, state_c = LSTM(units=512,return_state = True)(x)
encoder = Model(inputs = input_tensor, outputs = [state_h, state_c], name =␣
↪"encoder")
```

[68]:
```
"""
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.
"""
for english_single_batch in ds_train.take(1):
    result_hidden_state, result_output_state = encoder(english_single_batch)
    print("result_hidden_state.shape", result_hidden_state.shape)
    print("result_output_state.shape", result_output_state.shape)
```

```
result_hidden_state.shape (16, 512)
result_output_state.shape (16, 512)
```

[69]:
```
#Print the model summary for the encoder network.
encoder.summary(line_length=114)
```

```
Model: "encoder"
__________________________________________________________________________________
_________________________________
Layer (type)                                    Output Shape
Param #
==================================================================================
=================================
input_layer (InputLayer)                        [(16, 13, 128)]
0
__________________________________________________________________________________
_________________________________
add_end_embedding_layer_3 (AddEndEmbeddingLayer)   (16, 14, 128)
```

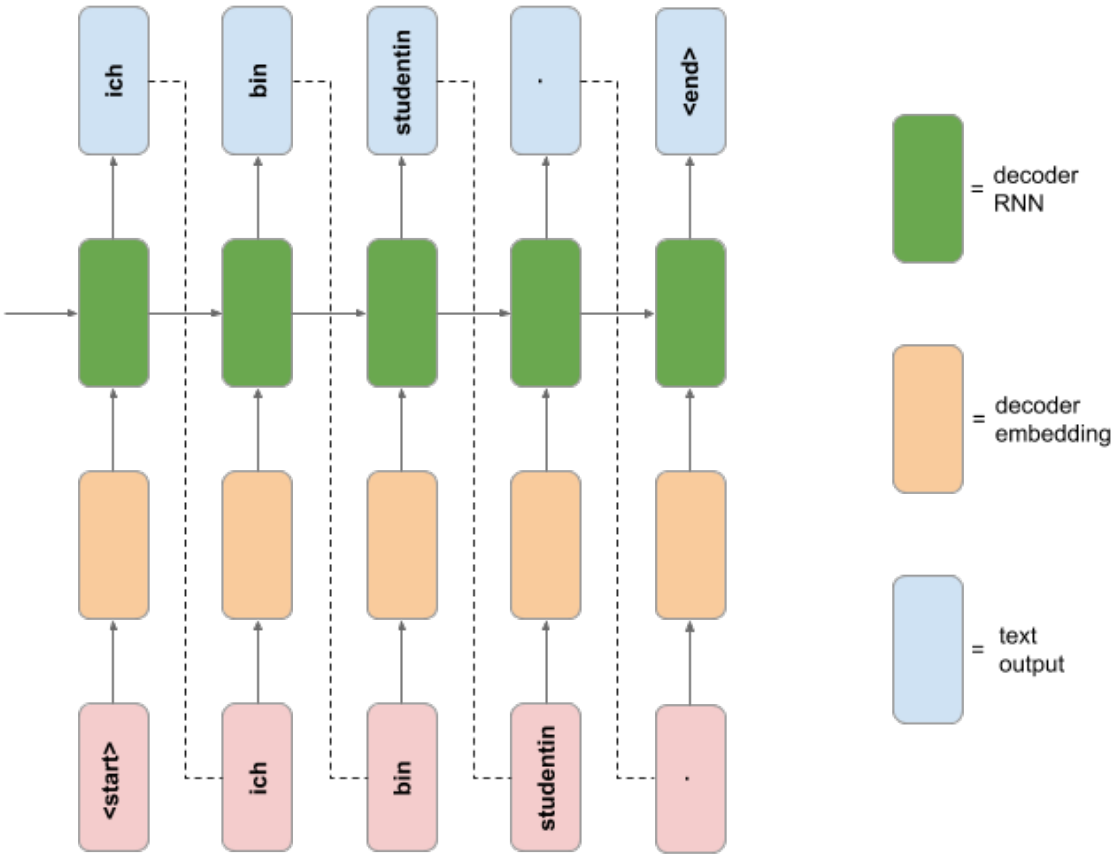
```
128
_________________________________________________________________________________
_________________________________
masking_layer (Masking)                         (16, 14, 128)
0
_________________________________________________________________________________
_________________________________
lstm_2 (LSTM)                                   [(16, 512), (16, 512), (16,
512)]            1312768
=================================================================================
=================================
Total params: 1,312,896
Trainable params: 1,312,896
Non-trainable params: 0
_________________________________________________________________________________
_________________________________
```

2.2 5. Build the decoder network

The decoder network follows the schematic diagram below.

[70]:
```
# 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")
```

[70]:



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.

[71]:
```
"""
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.

"""
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Layer, Embedding, LSTM, Dense


class Decoder(Model):

    def __init__(self):
        super(Decoder, self).__init__()
        num_german_tokens = len(tokenizer.word_index) + 1 # this additional one␣
↪is because tokenizer.work_index start from 1 instad of 0
        self.embedding_layer = Embedding(input_dim = num_german_tokens,␣
↪output_dim=128, mask_zero=True)
        self.lstm_layer = LSTM(units=512, return_sequences=True, return_state=␣
↪True)
        self.dense_layer = Dense(units=num_german_tokens)

    def call(self, inputs, hidden_state = None, cell_state = None):
        x = inputs
        x = self.embedding_layer(x)
        x , state_h, state_c = self.lstm_layer(x) if (hidden_state is None or␣
↪cell_state is None) else self.lstm_layer(x, initial_state=(hidden_state,␣
↪cell_state))
        x = self.dense_layer(x)
```

```
        return (x, state_h, state_c)

decoder = Decoder()
```

[72]:
```
# 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.

for english_sentence_single_batch, german_sequence_single_batch in ds_train.
↪take(1):
    hidden_state, cell_state = encoder(english_sentence_single_batch)
    output = decoder(german_sequence_single_batch, hidden_state, cell_state)
    print(output[0].shape)
```

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

2.3 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.

[73]:
```
#* 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).
def german_input_output_gen(tensor_batch_np_german_sequence):
    return tensor_batch_np_german_sequence[:,:-1],␣
 ↪tensor_batch_np_german_sequence[:,1:]
```

[74]:
```
"""
* 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`
"""
@tf.function
def train_fun(eng_input, german_input, german_output):

    with tf.GradientTape() as g:
        g.watch([eng_input,german_input])
        hidden_state, cell_state = encoder(eng_input)
        decoder_output_sequence = decoder(german_input, hidden_state,␣
 ↪cell_state)[0]
        loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        loss = loss(german_output, decoder_output_sequence)
        loss = tf.reduce_mean(loss)
        grad = g.gradient(loss, encoder.trainable_variables + decoder.
 ↪trainable_variables)
        return (loss, grad)
```

[34]:
```
"""
* 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.
```

```
"""

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
train_loss_results = []
validation_loss_results = []

import tqdm
import numpy as np
import math

for epoch in range(10):
    # initialize the training loss
    epoch_training_loss_avg = tf.keras.metrics.Mean()
    epoch_validation_loss_avg = tf.keras.metrics.Mean()

    # for each batch, train and update the loss
    for eng_single_batch, german_single_batch in(ds_train):
        german_input, german_output =␣
↪german_input_output_gen(german_single_batch)
        (train_loss, gradients) = train_fun(eng_single_batch, german_input,␣
↪german_output)
        optimizer.apply_gradients(zip(gradients, encoder.trainable_variables +␣
↪decoder.trainable_variables))
        epoch_training_loss_avg(train_loss)

    # update the training loss
    train_loss_results.append(epoch_training_loss_avg.result())

    # update the validation loss
    for eng_single_batch, german_single_batch in(ds_test):
        german_input, german_output =␣
↪german_input_output_gen(german_single_batch)
        (validation_loss, gradients) = train_fun(eng_single_batch,␣
↪german_input, german_output)
        epoch_validation_loss_avg(validation_loss)
    validation_loss_results.append(epoch_validation_loss_avg.result())

    print("Epoch  {:03d}: Training Loss: {:.3f}: Valication Loss: {:.3f}".
↪format(epoch,train_loss_results[-1], validation_loss_results[-1]))
```

```
WARNING:tensorflow:The dtype of the watched tensor must be floating (e.g.
tf.float32), got tf.int32
```

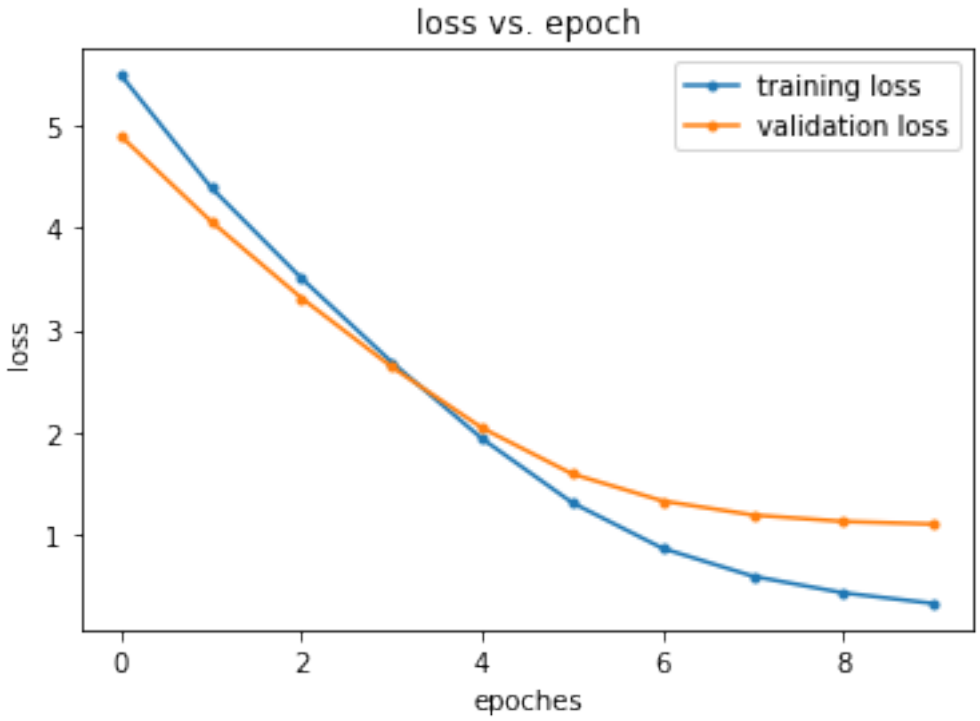
```
WARNING:tensorflow:The dtype of the watched tensor must be floating (e.g.
tf.float32), got tf.int32
```

```
Epoch  000: Training Loss: 5.490: Valication Loss: 4.894
Epoch  001: Training Loss: 4.390: Valication Loss: 4.059
```

```
Epoch  002: Training Loss: 3.503: Valication Loss: 3.310
Epoch  003: Training Loss: 2.682: Valication Loss: 2.636
Epoch  004: Training Loss: 1.935: Valication Loss: 2.040
Epoch  005: Training Loss: 1.308: Valication Loss: 1.590
Epoch  006: Training Loss: 0.863: Valication Loss: 1.326
Epoch  007: Training Loss: 0.590: Valication Loss: 1.189
Epoch  008: Training Loss: 0.428: Valication Loss: 1.126
Epoch  009: Training Loss: 0.324: Valication Loss: 1.100
```

[36]:
```
# print the loss vs. epoch
from matplotlib import pyplot as plt
plt.plot(train_loss_results,".-")
plt.plot(validation_loss_results,".-")
plt.xlabel("epoches")
plt.ylabel("loss")
plt.legend(["training loss", "validation loss"])
plt.title("loss vs. epoch")
```

[36]:
```
Text(0.5, 1.0, 'loss vs. epoch')
```



2.4 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: * Prepro-

cess 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.

[39]:
```
import random
counter = 0
while counter < 5:

    idx = random.randint(0,20000)
    print(f"======== sentence No. {idx}==========")
    print(f"ENGLISH INPUT:\t\t{english_sentences[idx]}")
    print(f"GERMAN TARGET:\t\t\t{german_sentences[idx]}")

    # Preprocess and embed the English sentence according to the model␣
↪requirements
    x = english_sentences[idx]
    x = helper_split(x)
    x = embedding_layer(x)
    if not filter_fun_less_than_13(x, None):
        continue
    x = np.expand_dims(x, axis=0)

    # Pass the embedded sentence through the encoder to get the encoder hidden␣
↪and cell states
    state_h, state_c = encoder(x)

    # 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
    sentences = ["<start>"]
    sequences = [[tokenizer.word_index[sentences[0]]]]

    decoder_output_sequence = decoder(tf.constant(sequences), state_h, state_c)

    # 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.
    for i in range(MAX_OUTPUT_LENGTH):

        # get next word index
        wordidx = np.argmax(decoder_output_sequence[0][0][-1])
```

```
        # get decoder status
        state_h, state_c = decoder_output_sequence[1],decoder_output_sequence[2]

        # convert index back to word and append
        word = tokenizer.index_word[ wordidx ]
        sentences.append(word)

        # update the sequences
        sequences[0].append(wordidx)

        # early stop if <end> is generated
        if word == "<end>":
            break;

        # run the decoder again
        decoder_output_sequence = decoder(tf.constant(sequences), state_h,␣
↪state_c)

    print("PREDICTION OUTPUT:\t", " ".join(sentences))
    counter += 1
```

```
======== sentence No. 12723==========
ENGLISH INPUT:          tom was spoiled .
GERMAN TARGET:                  <start> tom war verwoehnt . <end>
PREDICTION OUTPUT:       <start> tom war . <end>
======== sentence No. 14370==========
ENGLISH INPUT:          i am still alone .
GERMAN TARGET:                  <start> ich bin immer noch allein . <end>
PREDICTION OUTPUT:       <start> ich bin wahrhaben . <end>
======== sentence No. 14312==========
ENGLISH INPUT:          how is tom today ?
GERMAN TARGET:                  <start> wie geht es tom heute ? <end>
PREDICTION OUTPUT:       <start> wie ist tom ? <end>
======== sentence No. 12864==========
ENGLISH INPUT:          watch your step .
GERMAN TARGET:                  <start> pass auf , wo du hintrittst ! <end>
PREDICTION OUTPUT:       <start> schau auf . <end>
======== sentence No. 19021==========
ENGLISH INPUT:          i am years old .
GERMAN TARGET:                  <start> ich bin . <end>
PREDICTION OUTPUT:       <start> ich bin alt . <end>
```

[ ]:

[ ]: