Capstone Project

Neural translation model

Instructions

In this notebook, you will create a neural network that translates from English to German. You will use concepts from throughout this course, including building more flexible model architectures, freezing layers, data processing pipeline and sequence modelling.

This project is peer-assessed. Within this notebook you will find instructions in each section for how to complete the project. Pay close attention to the instructions as the peer review will be carried out according to a grading rubric that checks key parts of the project instructions. Feel free to add extra cells into the notebook as required.

How to submit

When you have completed the Capstone project notebook, you will submit a pdf of the notebook for peer review. First ensure that the notebook has been fully executed from beginning to end, and all of the cell outputs are visible. This is important, as the grading rubric depends on the reviewer being able to view the outputs of your notebook. Save the notebook as a pdf (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.

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

import os
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import scipy as sc
from tqdm import tqdm

# import others
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.layers import RNN, Embedding, LSTM, Dense, Flatten, Add, Input, L
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Model, Sequential
```

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]:

```
ls
```

```
sample_data/
```

In [3]:

```
# Run this cell to connect to your Drive folder

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

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

In [4]:

```
# Run this cell to load the dataset

NUM_EXAMPLES = 20000
#NUM_EXAMPLES = 100000
data_examples = []
with open('gdrive/MyDrive/tensorflow/ELP Tensorflow 2/deu (1).txt', 'r', encoding='utf8
    for line in f.readlines():
        if len(data_examples) < NUM_EXAMPLES:
            data_examples.append(line)
        else:
            break
```

In [5]:

```
# 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', '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()
```

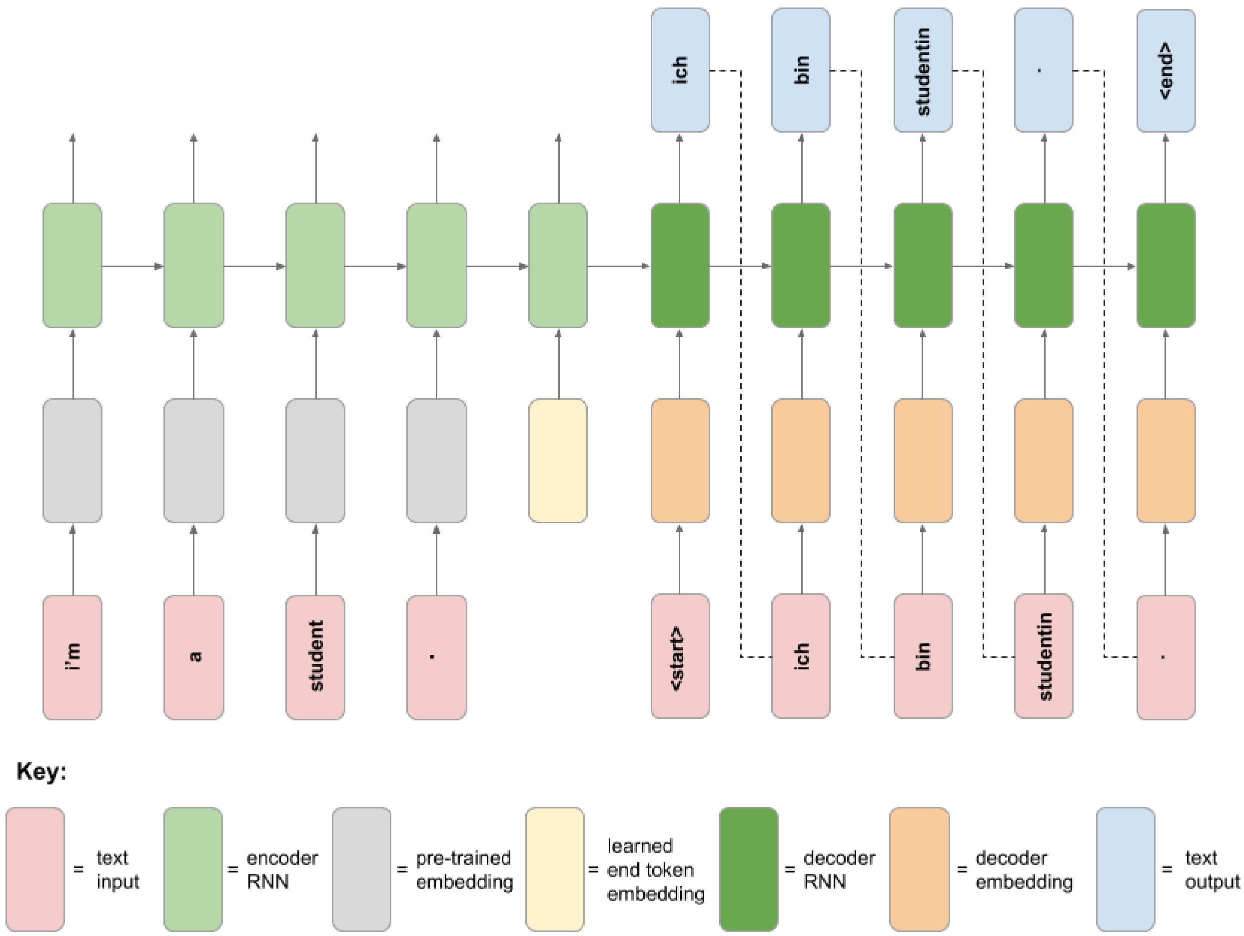
The custom translation model

The following is a schematic of the custom translation model architecture you will develop in this project.

In [6]:

```
# Run this cell to download and view a schematic diagram for the neural translation mod

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

Out[6]:



The custom model consists of an encoder RNN and a decoder RNN. The encoder takes words of an English sentence as input, and uses a pre-trained word embedding to embed the words into a 128-dimensional space. To indicate the end of the input sentence, a special end token (in the same 128-dimensional space) is passed in as an input. This token is a TensorFlow Variable that is learned in the training phase (unlike the pre-trained word embedding, which is frozen).

The decoder RNN takes the internal state of the encoder network as its initial state. A start token is passed in as the first input, which is embedded using a learned German word embedding. The decoder RNN then makes a prediction for the next German word, which during inference is then passed in as the following input, and this process is repeated until the special `<end>` token is emitted from the decoder.

1. Text preprocessing

- Create separate lists of English and German sentences, and preprocess them using the `preprocess_sentence` function provided for you above.
- Add a special `"<start>"` and `"<end>"` token to the beginning and end of every German sentence.
- Use the Tokenizer class from the `tf.keras.preprocessing.text` module to tokenize the German sentences, ensuring that no character filters are applied. *Hint: use the Tokenizer's "filter" keyword argument.*
- Print out at least 5 randomly chosen examples of (preprocessed) English and German sentence pairs. For the German sentence, print out the text (with start and end tokens) as well as the tokenized sequence.
- Pad the end of the tokenized German sequences with zeros, and batch the complete set of sequences into one numpy array.

In [7]:
```
eng_data = [sentence.split('\t')[0] for sentence in data_examples]
ger_data = [sentence.split('\t')[1] for sentence in data_examples]
print('Checking')
print('first 5 eng data is ', eng_data[:5])
print('first 5 ger data is ', ger_data[:5])
```

```
Checking
first 5 eng data is  ['Hi.', 'Hi.', 'Run!', 'Wow!', 'Wow!']
first 5 ger data is  ['Hallo!', 'Grüß Gott!', 'Lauf!', 'Potzdonner!', 'Donnerwetter!']
```

In [8]:
```
# preprocess them using provided preprocess_sentence function
eng_data = [preprocess_sentence(s) for s in eng_data]
ger_data = [preprocess_sentence(s) for s in ger_data]
```

In [9]:
```
# Add a special "<start>" and "<end>" token to the beginning and end of every German se
ger_data = ["<start> " + sentence + " <end>" for sentence in ger_data]
print(ger_data[0])
```

```
<start> hallo ! <end>
```

In [10]:
```
# Use the Tokenizer class from the tf.keras.preprocessing.text module to tokenize the G
# ensuring that no character filters are applied.
# Hint: use the Tokenizer's "filter" keyword argument.
tokenizer = Tokenizer(
    num_words=None,
    filters=''
)

tokenizer.fit_on_texts(ger_data)
```

In [11]:
```
ger_sequence = tokenizer.texts_to_sequences(ger_data)
ger_sequence[:5]
```

Out[11]:
```
[[1, 405, 9, 2],
 [1, 3155, 663, 9, 2],
 [1, 975, 9, 2],
```

```
[1, 3156, 9, 2],
[1, 3157, 9, 2]]
```

In [12]:

```
# Print out at least 5 randomly chosen examples of (preprocessed) English and German se
# For the German sentence, print out the text (with start and end tokens) as well as th
for _ in range(5):
    i = np.random.choice(len(ger_sequence))
    print(eng_data[i],' ',ger_data[i],' ',ger_sequence[i])
```

```
you can't win .   <start> es ist aussichtslos . <end>   [1, 10, 6, 3755, 3, 2]
am i dreaming ?   <start> traeume ich ? <end>   [1, 940, 4, 7, 2]
he's not in .   <start> er ist nicht daheim . <end>   [1, 14, 6, 12, 1505, 3, 2]
who remembers ?   <start> wer erinnert sich ? <end>   [1, 43, 2007, 34, 7, 2]
i took a walk .   <start> ich habe einen spaziergang gemacht . <end>   [1, 4, 18, 40, 19
64, 101, 3, 2]
```

In [13]:

```
# Pad the end of the tokenized German sequences with zeros, and batch the complete set
padded_ger_sequences = pad_sequences(ger_sequence, padding='post')
print('padded data shape is ',padded_ger_sequences.shape)
print('first 5 padded sequences')
for i in range(5):
    print(f'i = {i}, {padded_ger_sequences[i]}')
```

```
padded data shape is  (20000, 14)
first 5 padded sequences
i = 0, [  1 405    9    2    0    0    0    0    0    0    0    0    0    0]
i = 1, [   1 3155  663    9    2    0    0    0    0    0    0    0    0    0]
i = 2, [  1 975    9    2    0    0    0    0    0    0    0    0    0    0]
i = 3, [   1 3156    9    2    0    0    0    0    0    0    0    0    0    0]
i = 4, [   1 3157    9    2    0    0    0    0    0    0    0    0    0    0]
```

In [13]:

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 [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)
```

In [15]:

```
# Test the layer

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

Out[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.

In [16]:

```
# Create a random training and validation set split of the data, reserving e.g. 20% of
# (NB: each English dataset example is a single sentence string, and each German datase
# of padded integer tokens).

from sklearn.model_selection import train_test_split

eng_train, eng_valid, ger_train, ger_valid = train_test_split(eng_data, padded_ger_sequ
                                                              test_size=0.2,
                                                             random_state=2021)
```

In [17]:

```
# 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.

train_dataset = tf.data.Dataset.from_tensor_slices((eng_train, ger_train))
test_dataset = tf.data.Dataset.from_tensor_slices((eng_valid, ger_valid))
```

In [18]:

```
# Create a function to map over the datasets that splits each English sentence at space
# Apply this function to both Dataset objects using the map method.
# Hint: look at the tf.strings.split function.

def split_eng(x,y):
    return tf.strings.split(x,' '),y

train_dataset = train_dataset.map(split_eng)
test_dataset = test_dataset.map(split_eng)
```

In [19]:

```
# Create a function to map over the datasets that embeds each sequence of English words
# Apply this function to both Dataset objects using the map method.

def embbed_eng(x,y):
    return embedding_layer(x),y

train_dataset = train_dataset.map(embbed_eng)
test_dataset = test_dataset.map(embbed_eng)
```

In [20]:

```
# Create a function to filter out dataset examples where the English sentence is more t
# Apply this function to both Dataset objects using the filter method.

def filter_on_len(eng_emb,deu_seq):
    return tf.shape(eng_emb).shape[0]<=13

train_dataset = train_dataset.filter(filter_on_len)
test_dataset = test_dataset.filter(filter_on_len)
```

In [21]:

```
# Create a function to map over the datasets that pads each English sequence of embeddi
# 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.

def pad_map(eng, deu):
    d = tf.shape(eng).shape[0]
    #paddings = tf.constant([[13-d,0],[0,0]])
    #paddings = tf.concat(([[13-tf.shape(eng)[0],0]], [[0,0]]), axis=0)
    paddings = [[13-tf.shape(eng)[0],0],[0,0]]
    return tf.pad(eng, paddings=paddings, mode='CONSTANT', constant_values=0), deu

train_dataset = train_dataset.map(pad_map)
test_dataset = test_dataset.map(pad_map)
```

In [22]:

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

In [23]:
```
# Print the element_spec property for the training and validation Datasets.

print('training dataset element_spec', train_dataset.element_spec)
print('testing dataset element_spec', test_dataset.element_spec)
```

```
training dataset element_spec (TensorSpec(shape=(None, None, 128), dtype=tf.float32, nam
e=None), TensorSpec(shape=(None, 14), dtype=tf.int32, name=None))
testing dataset element_spec (TensorSpec(shape=(None, None, 128), dtype=tf.float32, name
=None), TensorSpec(shape=(None, 14), dtype=tf.int32, name=None))
```

In [24]:
```
# Using the Dataset .take(1) method, print the shape of the English data example from t

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

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

In [25]:
```
# Using the Dataset .take(1) method, print the German data example Tensor from the vali

print(test_dataset.take(1))
for x,y in test_dataset.take(1):
    print(x.shape, y.shape)
```

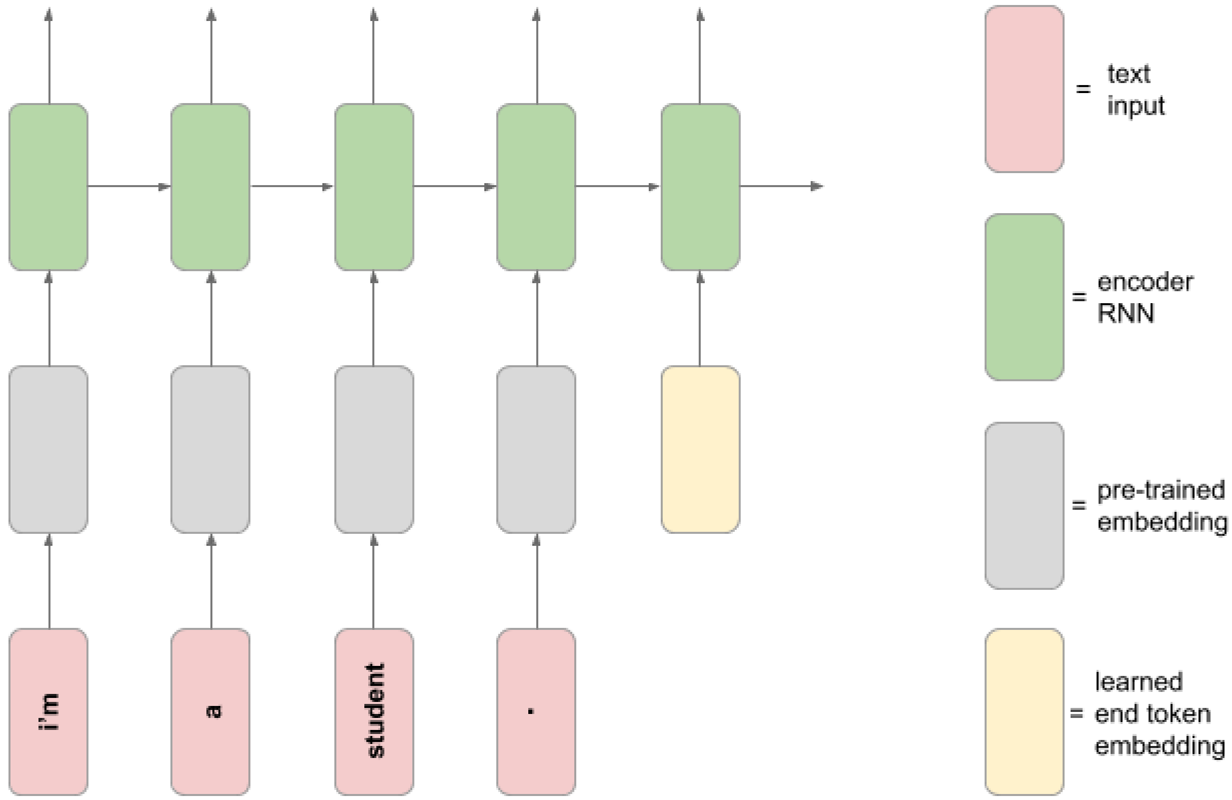
```
<TakeDataset shapes: ((None, None, 128), (None, 14)), types: (tf.float32, tf.int32)>
(16, 13, 128) (16, 14)
```

3. Create the custom layer

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

In [26]:
```
# 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.co
Image("neural_translation_model.png")
```

Out[26]:



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 [27]:

```
# Using layer subclassing, create a custom layer that takes a batch of English data exa
# 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)
# (the size of the embedding space). Hint: you may find it helpful in the call method t
# the tf.tile function to replicate the end token embedding across every element in the

class TileLayer(Layer):
    def __init__(self, **kwargs):
        super(TileLayer, self).__init__(**kwargs)
        self.ly = tf.Variable(initial_value=tf.random.uniform(shape=(1,128)), trainable

    def call(self, inputs):
        x = self.ly
        end_token = tf.tile(x, [tf.shape(inputs)[0],1])
        end_token = tf.expand_dims(end_token,axis=1)
        return tf.keras.layers.concatenate([inputs, end_token],axis=1)
```

In [28]:
```
CustomLayer = TileLayer()
```

In [29]:
```
# Using the Dataset .take(1) method, extract a batch of English data examples from the
# Test the custom layer by calling the layer on the English data batch Tensor and print
# shape (the layer should increase the sequence length by one).

for x, y in train_dataset.take(1):
    print(x.shape)
    print(y.shape)
    #print(CustomLayer((x,y)))
```

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

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 [30]:
```
def Encoder():
    inputs = Input(batch_shape=(None,13,128))
    h = CustomLayer(inputs)
    x = Masking(mask_value=0)(h)
    x, hidden, cell = LSTM(units=512,return_state=True)(x)
    model = Model(inputs = inputs, outputs = [hidden, cell])
    return model
```

In [31]:
```
model = Encoder()
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
```

```
input_1 (InputLayer)         [(None, 13, 128)]         0
_________________________________________________________________
tile_layer (TileLayer)       (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
_________________________________________________________________
```

In [32]:

```
ds = next(iter(train_dataset.take(1)))[0]
#print(ds)
model(ds)
```

Out[32]:

```
[<tf.Tensor: shape=(16, 512), dtype=float32, numpy=
 array([[-0.07765123, -0.01525668, -0.06121036, ..., -0.07484505,
         -0.03508214, -0.04541859],
        [-0.06949915, -0.0097357 , -0.05673667, ..., -0.06193984,
         -0.02824094, -0.04089544],
        [-0.0708662 , -0.00631354, -0.05924416, ..., -0.0705665 ,
         -0.02895202, -0.04381373],
        ...,
        [-0.06700589, -0.016586  , -0.06000941, ..., -0.06413918,
         -0.02512879, -0.04251159],
        [-0.07434743, -0.00551682, -0.06395164, ..., -0.06288337,
         -0.02436605, -0.04464319],
        [-0.07116499, -0.00876783, -0.05935578, ..., -0.06803543,
         -0.01826751, -0.04789128]], dtype=float32)>,
 <tf.Tensor: shape=(16, 512), dtype=float32, numpy=
 array([[-0.1497643 , -0.0309961 , -0.13179529, ..., -0.16259263,
         -0.06072225, -0.08270259],
        [-0.13414004, -0.01978139, -0.12249025, ..., -0.13446352,
         -0.04898724, -0.0745308 ],
        [-0.1366695 , -0.01283358, -0.12770236, ..., -0.15301877,
         -0.05021338, -0.07979214],
        ...,
        [-0.12917928, -0.03366211, -0.12961014, ..., -0.13900866,
         -0.04354443, -0.07736023],
        [-0.14369164, -0.01121718, -0.13825348, ..., -0.13624752,
         -0.0422579 , -0.08137679],
        [-0.13755874, -0.01777643, -0.12801385, ..., -0.14776577,
         -0.03164446, -0.08719438]], dtype=float32)>]
```

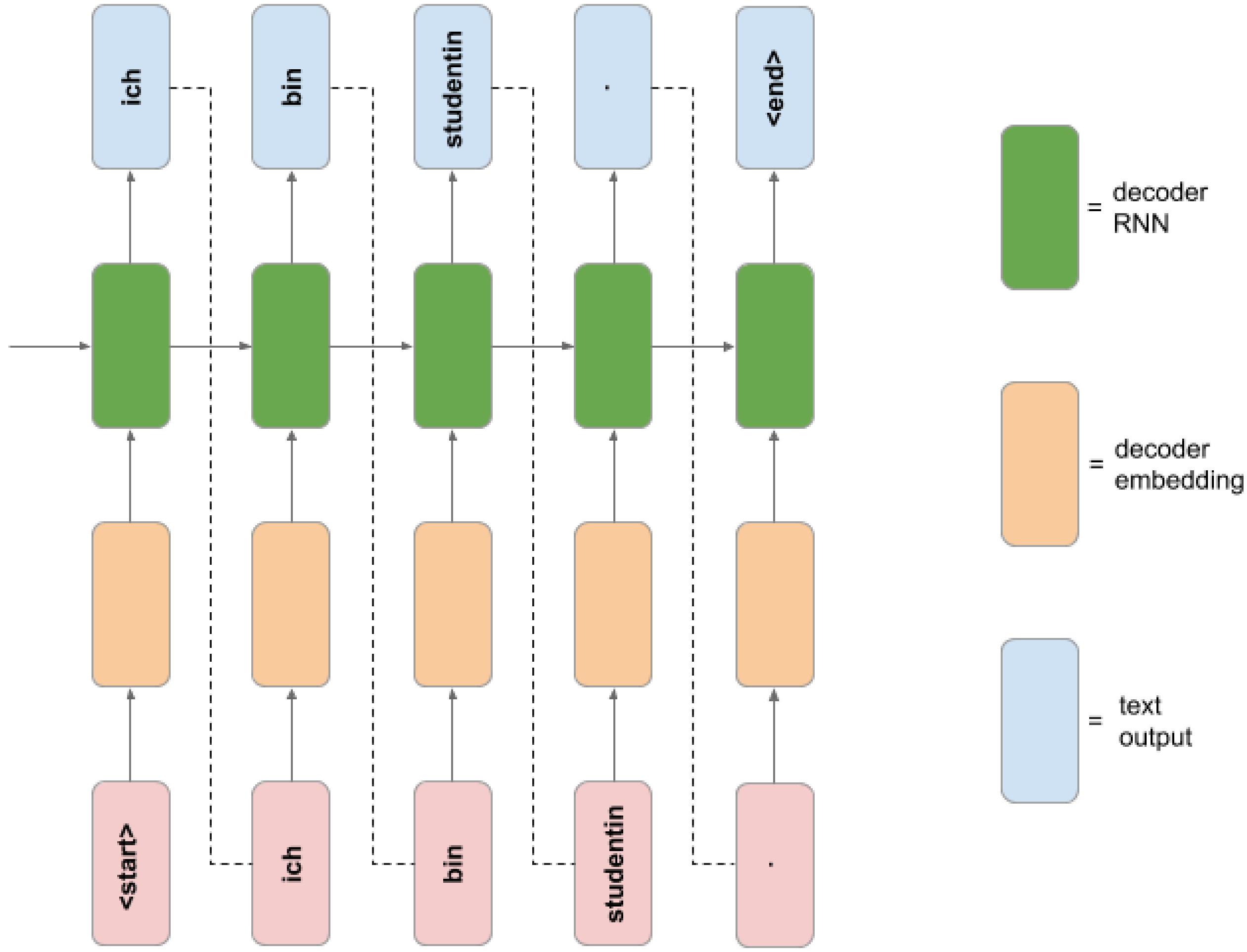
5. Build the decoder network

The decoder network follows the schematic diagram below.

In [33]:

```
# 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.co
Image("neural_translation_model.png")
```

Out[33]:



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 [34]:
```
max([v for k,v in tokenizer.word_index.items()])
```

Out[34]: 5743

In [35]:
```
class Decoder(Model):
    def __init__(self,**kwargs):
        super(Decoder,self).__init__(**kwargs)
        self.emb_ge = Embedding(input_dim=len(tokenizer.word_index)+1,output_dim=128, m
        #self.emb_ge = Embedding(input_dim=5744,output_dim=128, mask_zero=True)
        self.lstm = LSTM(units=512,return_sequences=True, return_state=True)
        self.dense = Dense(len(tokenizer.word_index)+1,activation=None)
        #self.dense = Dense(5744)

    def call(self,inputs,hidden=None,cell=None):
        x = self.emb_ge(inputs)
        if hidden is None and cell is None:
            x, hidden, cell = self.lstm(x)
        else:
            x, hidden, cell = self.lstm(x, initial_state=(hidden,cell))
        x = self.dense(x)
        return x, hidden, cell
```

In [36]:
```
decoder = Decoder()
hidden, cell = model(next(iter(train_dataset.take(1)))[0])
print('Encoder: \nhidden shape is ', tf.shape(hidden), '\n cell shape is ', tf.shape(ce
x, hidden, cell = decoder(next(iter(train_dataset.take(1)))[1],hidden,cell)
print('Decoder: \nx shape ',tf.shape(x), ' \nhidden shape ', tf.shape(hidden), ' \ncell
```

```
Encoder:
hidden shape is  tf.Tensor([ 16 512], shape=(2,), dtype=int32)
 cell shape is  tf.Tensor([ 16 512], shape=(2,), dtype=int32)
Decoder:
x shape  tf.Tensor([  16   14 5744], shape=(3,), dtype=int32)
hidden shape  tf.Tensor([ 16 512], shape=(2,), dtype=int32)
cell shape  tf.Tensor([ 16 512], shape=(2,), dtype=int32)
```

In [36]:
```
```

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 [37]:
```
# Define a function that takes a Tensor batch of German data (as extracted from the tra
# and returns a tuple containing German inputs and outputs for the decoder model (refer

def German_InOut(data):
    return (tf.cast(data[:,0:-1], tf.float32), tf.cast(data[:, 1:], tf.float32))
```

In [38]:
```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
valid_loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()
```

In [39]:
```
@tf.function
def grad(engs_data, ger_in, ger_out):
    trainable_variables = encoder.trainable_variables + decoder.trainable_variables
    with tf.GradientTape() as tape:
        # forward
        hidden_e, cell_e = encoder(engs_data)
        #print('Encoder:\n hidden shape ',tf.shape(hidden_e), '\n cell shape is ', tf.s
        x,_,_ = decoder(ger_in, hidden_e, cell_e)
        #print('decoder shape ', tf.shape(x))
        current_loss = loss_object(ger_out,x)
        #print('current loss = ',current_loss)
    # backward
    grads = tape.gradient(current_loss, trainable_variables)
    return current_loss, grads
```

In [40]:

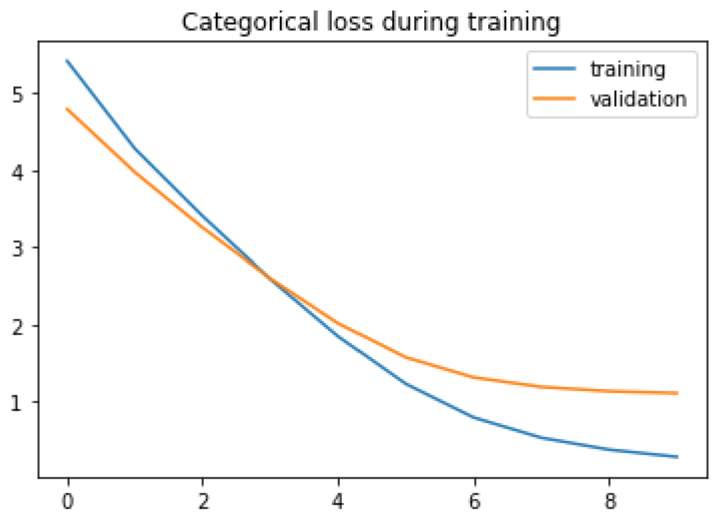
```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
valid_loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = tf.keras.optimizers.Adam()

encoder = Encoder()
decoder = Decoder()

NUM_EPOCHS = 10
train_losses = []
valid_losses = []
for epoch in range(NUM_EPOCHS):
    epoch_loss_avg = tf.keras.metrics.Mean()
    for eng_ds,ge in train_dataset:
        ger_in, ger_out = German_InOut(ge)
        #print('Split data: \nger in shape is ',tf.shape(ger_in), '\n ger out shape is
        trainable_variables = encoder.trainable_variables + decoder.trainable_variables
        with tf.GradientTape() as tape:
            # forward
            hidden_e, cell_e = encoder(eng_ds)
            #print('Encoder:\n hidden shape ',tf.shape(hidden_e), '\n cell shape is ',
            x,_,_ = decoder(ger_in, hidden_e, cell_e)
            #print('decoder shape ', tf.shape(x))
            current_loss = loss_object(ger_out,x)
            #print('current loss = ',current_loss)
        # backward
        grads = tape.gradient(current_loss, trainable_variables)

        # update parameters
        optimizer.apply_gradients(zip(grads, trainable_variables))

        # calculate loss
        epoch_loss_avg(current_loss)
    train_losses.append(epoch_loss_avg.result())

    # validation
    epoch_valid_loss_avg = tf.keras.metrics.Mean()
    for eng_ds, ge in test_dataset:
        ger_in, ger_out = German_InOut(ge)
        hidden_e, cell_e = encoder(eng_ds)
        x,_,_ = decoder(ger_in, hidden_e, cell_e)
        valid_loss = valid_loss_object(ger_out,x)

        # calculate loss
        epoch_valid_loss_avg(valid_loss)
    valid_losses.append(epoch_valid_loss_avg.result())

    print ("Epoch {:03d}: Training Loss: {:.3f} Validation Loss: {:.3f} ".format(epoch+
                                                                                epoch_
                                                                               epoch_v
```

```
Epoch 001: Training Loss: 5.416 Validation Loss: 4.792
Epoch 002: Training Loss: 4.279 Validation Loss: 3.973
Epoch 003: Training Loss: 3.404 Validation Loss: 3.254
Epoch 004: Training Loss: 2.590 Validation Loss: 2.599
Epoch 005: Training Loss: 1.848 Validation Loss: 2.016
Epoch 006: Training Loss: 1.228 Validation Loss: 1.573
Epoch 007: Training Loss: 0.793 Validation Loss: 1.313
Epoch 008: Training Loss: 0.530 Validation Loss: 1.190
```

```
Epoch 009: Training Loss: 0.375 Validation Loss: 1.134
Epoch 010: Training Loss: 0.282 Validation Loss: 1.109
```

In [41]:

```
# Plot the learning curves for loss vs epoch for both training and validation sets.
plt.figure()
plt.plot(np.arange(10),train_losses, label='training')
plt.plot(np.arange(10),valid_losses, label='validation')
plt.legend()
plt.title('Categorical loss during training')
plt.show()
```



7. Use the model to translate

Now it's time to put your model into practice! You should run your translation for five randomly sampled English sentences from the dataset. For each sentence, the process is as follows:

- Preprocess and embed the English sentence according to the model requirements.
- Pass the embedded sentence through the encoder to get the encoder hidden and cell states.
- Starting with the special `"<start>"` token, use this token and the final encoder hidden and cell states to get the one-step prediction from the decoder, as well as the decoder's updated hidden and cell states.
- Create a loop to get the next step prediction and updated hidden and cell states from the decoder, using the most recent hidden and cell states. Terminate the loop when the `"<end>"` token is emitted, or when the sentence has reached a maximum length.
- Decode the output token sequence into German text and print the English text and the model's German translation.

In [62]:

```
def translate_fromdata():
    eng_samples = []
    ger_samples = []
    padded_ger_samples = []

    i = np.random.choice(len(ger_sequence))

    eng_samples.append(eng_data[i])
    ger_samples.append(ger_data[i])
    padded_ger_samples.append(padded_ger_sequences[i])
```

```
    sample_dataset = tf.data.Dataset.from_tensor_slices((eng_samples, padded_ger_sample
    sample_dataset = sample_dataset.map(split_eng).map(embbed_eng).map(pad_map).batch(1

    results = []
    res = []

    ds = next(iter(sample_dataset.take(1)))[0]
    hidden, cell = encoder(ds)
    start = np.zeros((1,14))
    start[:,0] = tokenizer.word_index['<start>']

    for j in range(13):
        x, hidden, cell = decoder(start,hidden,cell)
        [predict_idx] = tf.argmax(x,axis=2).numpy()

        start[0,j+1] = predict_idx[0]
        results.append(predict_idx)

        if 2 in predict_idx[:j+2]:
            start[0,j+2] = 2
            break
        res.append(start[0,:])
    print('Eng: ',eng_data[i],'------ Deus target:',ger_data[i],'------ Deus prediction
    #return tokenizer.sequences_to_texts(res)[-1]

for _ in range(5):
    translate_fromdata()
```

```
Eng:  love lasts . ------ Deus target: <start> die liebe bleibt . <end> ------ Deus pred
iction: <start> liebe tut . . <end>
Eng:  i don't see why . ------ Deus target: <start> ich verstehe nicht , warum . <end> -
----- Deus prediction: <start> ich gefaellt nichts . . <end>
Eng:  she looks happy . ------ Deus target: <start> sie sieht gluecklich aus . <end> ---
--- Deus prediction: <start> sie sieht mich . . <end>
Eng:  it has begun . ------ Deus target: <start> es hat angefangen . <end> ------ Deus p
rediction: <start> es ist ! . <end>
Eng:  they were dirty . ------ Deus target: <start> sie waren dreckig . <end> ------ Deu
s prediction: <start> sie war gut . . <end>
```

In [63]:

```
def translate_random():
    data=input()
    eng_samples = []
    ger_samples = []
    padded_ger_samples = []

    eng_samples.append(data)
    ger_samples.append('<start>')

    padded_ger_samples.append(padded_ger_sequences[0])

    sample_dataset = tf.data.Dataset.from_tensor_slices((eng_samples, padded_ger_sample
    sample_dataset = sample_dataset.map(split_eng).map(embbed_eng).map(pad_map).batch(1

    results = []
    res = []

    ds = next(iter(sample_dataset.take(1)))[0]
    hidden, cell = encoder(ds)
    start = np.zeros((1,14))
    start[:,0] = tokenizer.word_index['<start>']
```

```
    for i in range(10):
        x, hidden, cell = decoder(start,hidden,cell)
        [predict_idx] = tf.argmax(x,axis=2).numpy()

        start[0,i+1] = predict_idx[0]
        results.append(predict_idx)
        if 2 in predict_idx[:i+2]:
            start[0,i+2] = 2
            break
        res.append(start[0,:])

    return tokenizer.sequences_to_texts(res)[-1]

translate_random()
```

```
thanks for your reviewing
```

Out[63]:

```
'<start> es hat mich . . <end>'
```