Capstone Project

March 18, 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
```

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.

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

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

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

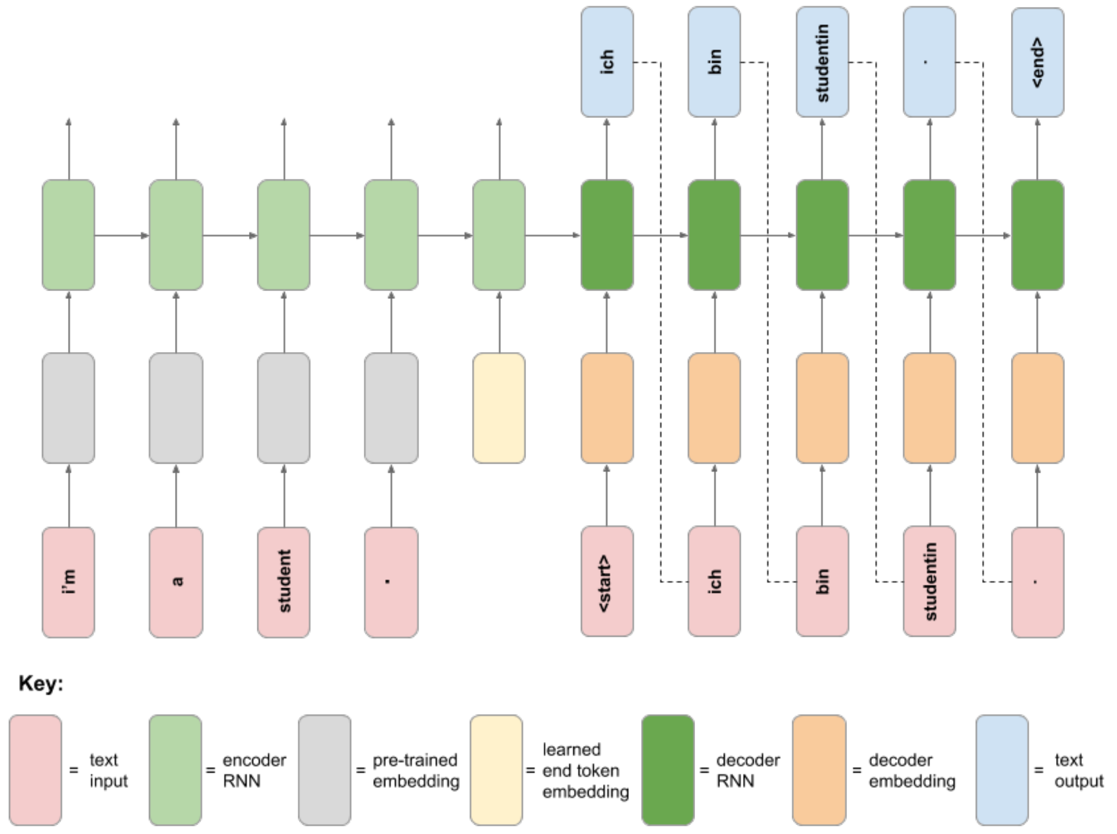
[3]:
```python
# Run this cell to load the dataset

NUM_EXAMPLES = 20_000
data_examples = []
folder = "/content/gdrive/My Drive/Courses/Coursera/Customising your Models␣
 ↪with TensorFlow 2/Week 5/"
with open(f"{folder}deu.txt", "r", encoding="utf8") as f:
    for line in f.readlines():
        if len(data_examples) < NUM_EXAMPLES:
            data_examples.append(line)
        else:
            break
```

[4]:
```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', 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.

[5]:
```
# 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")
```

[5]:



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

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

1.2 1. Text preprocessing

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

[6]:
```python
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
```

[7]:
```python
# Helper functions
def split_eng_ger_sentences(examples):

    english_sentences = []
    german_sentences = []
    for eng, ger in [examples[i].split("\t")[:2] for i in range(len(examples))]:
        english_sentences.append(eng)
        german_sentences.append(ger)
    return english_sentences, german_sentences

def apply_preprocessing(sentences, preproc_func):
    return [preproc_func(sentence) for sentence in sentences]

def add_star_end_tokens(sentences, start_token="<start>", end_token="<end>"):
    return [f"{start_token} {sentence} {end_token}" for sentence in sentences]

def tokenize(sentences):
    tokenizer = Tokenizer(
        filters="",
    )
    tokenizer.fit_on_texts(sentences)
    return tokenizer, tokenizer.texts_to_sequences(sentences)
```

[8]:
```python
# Apply required preprocessing steps
english_sentences, german_sentences = split_eng_ger_sentences(data_examples)
english_sentences = apply_preprocessing(english_sentences, preprocess_sentence)
german_sentences = apply_preprocessing(german_sentences, preprocess_sentence)
german_sentences = add_star_end_tokens(german_sentences)
tokenizer, tokenized_german_sentences = tokenize(german_sentences)
```

[9]:
```
# Print 5 random examples
rnd_idx = np.random.choice(range(len(english_sentences)), size=5)

print(f"{'English Sentence':<20}", f"{'German Sentence':<50}", f"{'Token German␣
↪Sentence':<20}")
for i in rnd_idx:
    print(
        f"{english_sentences[i]:<20}",
        f"{german_sentences[i]:<50}",
        f"{tokenized_german_sentences[i]}"
    )
```

```
English Sentence     German Sentence                                    Token
German Sentence
they're old .        <start> sie sind alt . <end>                       [1, 8,
23, 179, 3, 2]
did you live here ?  <start> hast du frueher hier gewohnt ? <end>       [1, 58,
13, 1442, 33, 2955, 7, 2]
he's annoying .      <start> er ist laestig . <end>                     [1, 14,
6, 1948, 3, 2]
it's too fast .      <start> das ist zu schnell . <end>                 [1, 11,
6, 20, 127, 3, 2]
i ate the apple .    <start> den apfel habe ich gegessen . <end>        [1, 53,
863, 18, 4, 217, 3, 2]
```

[10]:
```
# Pad german sentences
tokenized_german_sentences = pad_sequences(tokenized_german_sentences,␣
↪padding="post")

# Convert other sequences to array
english_sentences = np.asarray(english_sentences, dtype=str)
german_sentences = np.asarray(german_sentences, dtype=str)
```

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.

[11]:
```python
# 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
)
```

[12]:
```python
# Test the layer

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

[12]:
```
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.

[13]:
```python
from sklearn.model_selection import train_test_split

def check_dataset(dataset, iter=3):
    for x, y in dataset.take(iter):
        print(x.shape, y.shape)
```

[14]:
```
# Random split data into train and validation
x_train, x_val, y_train, y_val = train_test_split(english_sentences,␣
 ↪tokenized_german_sentences, test_size=0.2)
print(f"{'x_train shape:':<15}", x_train.shape)
print(f"{'y_train shape:':<15}", y_train.shape)
print(f"{'x_val shape:':<15}", x_val.shape)
print(f"{'y_val shape:':<15}", y_val.shape)
```

```
x_train shape:  (16000,)
y_train shape:  (16000, 14)
x_val shape:    (4000,)
y_val shape:    (4000, 14)
```

[15]:
```
# Create TensorFlow datasets
train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))

check_dataset(train_dataset)
```

```
() (14,)
() (14,)
() (14,)
```

[16]:
```
def split_english_sentences(eng, ger):
    return tf.strings.split(eng), ger

train_dataset = train_dataset.map(split_english_sentences)
val_dataset = val_dataset.map(split_english_sentences)

check_dataset(train_dataset)
```

```
(3,) (14,)
(4,) (14,)
(5,) (14,)
```

[17]:
```
def create_embeddings(eng, ger):
    return embedding_layer(eng), ger

train_dataset = train_dataset.map(create_embeddings)
val_dataset = val_dataset.map(create_embeddings)

check_dataset(train_dataset)
```

```
(3, 128) (14,)
(4, 128) (14,)
(5, 128) (14,)
```

[18]:
```
def filter_get_13(eng, ger):
    return tf.math.less(tf.shape(eng)[0], tf.constant(13))

train_dataset = train_dataset.filter(filter_get_13)
val_dataset = val_dataset.filter(filter_get_13)

check_dataset(train_dataset)
```

```
(3, 128) (14,)
(4, 128) (14,)
(5, 128) (14,)
```

[19]:
```
def pad_embeddings(eng, ger):
    paddings = tf.concat([
        tf.zeros((2, 1), dtype=tf.int32),
        tf.expand_dims(tf.math.subtract(tf.constant([13, 128]), tf.shape(eng)),␣
 ↪axis=1)
        ],
        axis=1)
    return tf.pad(eng, paddings=paddings), ger

train_dataset = train_dataset.map(pad_embeddings)
val_dataset = val_dataset.map(pad_embeddings)

check_dataset(train_dataset)
```

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

[20]:
```
# Shuffle train dataset each epoch
train_dataset = train_dataset.shuffle(x_train.shape[0],␣
 ↪reshuffle_each_iteration=True)

# Batch datasets
batch_size = 16
train_dataset = train_dataset.batch(batch_size, drop_remainder=True)
val_dataset = val_dataset.batch(batch_size, drop_remainder=True)
```

[21]:
```
print(f"{'Training dataset:':<20}", train_dataset.element_spec)
print(f"{'Validation dataset:':<20}", val_dataset.element_spec)
```

```
Training dataset:    (TensorSpec(shape=(16, None, None), dtype=tf.float32,
name=None), TensorSpec(shape=(16, 14), dtype=tf.int32, name=None))
Validation dataset:  (TensorSpec(shape=(16, None, None), dtype=tf.float32,
name=None), TensorSpec(shape=(16, 14), dtype=tf.int32, name=None))
```

[22]:
```
for eng, ger in train_dataset.take(1):
    print(f"{'English shape training dataset:':<35}", eng.shape)

for eng, ger in val_dataset.take(1):
    print(f"{'German shape validation dataset:':<35}", ger.shape)
```

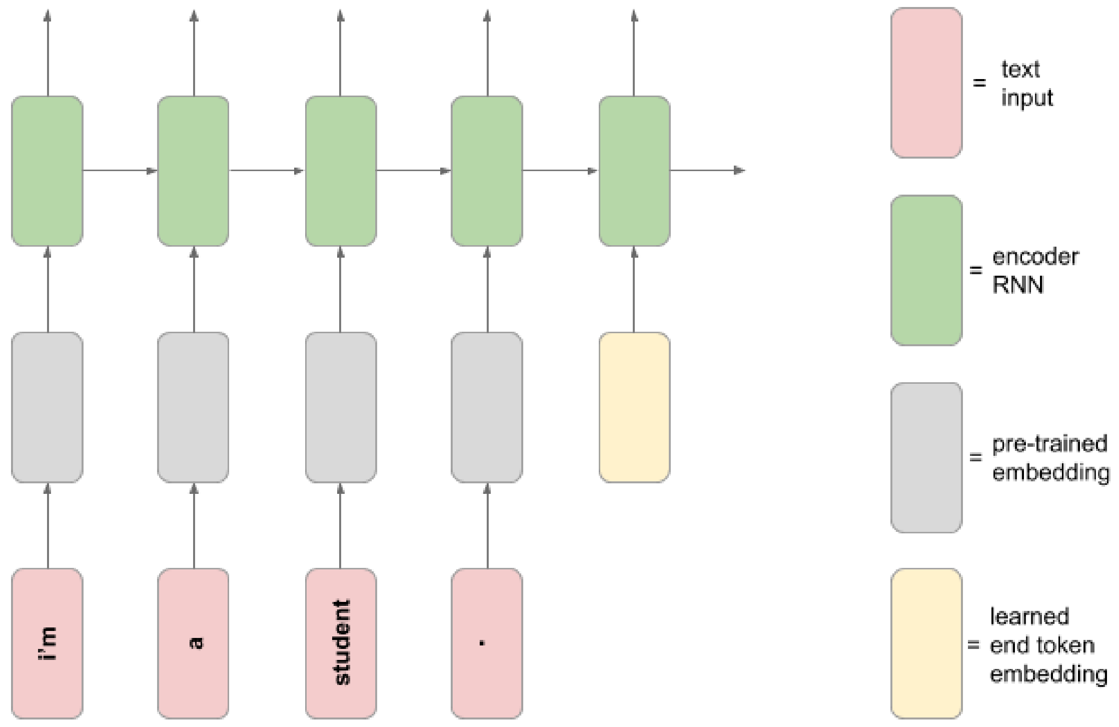
```
English shape training dataset:     (16, 13, 128)
German shape validation dataset:    (16, 14)
```

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:

[23]:
```
# 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=1JrtN0zUJDa0WrK4C-xv-4wUuZaI12sQI"
Image("neural_translation_model.png")
```

[23]:



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).

[24]:
```
from tensorflow.keras.layers import concatenate, Layer
```

[25]:
```
class AddEndTokenEmbedding(Layer):

    def __init__(self, embedding_dim=128, **kwargs):
        super(AddEndTokenEmbedding, self).__init__(**kwargs)
        self.end_token_embedding = tf.Variable(
            initial_value=tf.random.uniform((embedding_dim,)),
            trainable=True
        )

    def call(self, inputs):
        end_token = tf.tile(
            tf.reshape(self.end_token_embedding, shape=(1, 1, self.
 ↪end_token_embedding.shape[0])),
            multiples=[tf.shape(inputs)[0], 1, 1]
        )
        return concatenate([inputs, end_token], axis=1)
```

[26]:
```
# Print extracted test batch
test_batch_x, test_batch_y = list(train_dataset.take(1).as_numpy_iterator())[0]
print(f"{'English sentences batch shape:':<55}", test_batch_x.shape)

# Print test batch after passing through custom layer
end_token_layer = AddEndTokenEmbedding(embedding_dim=128)
print(f"{'English sentences batch shape after adding end token:':<55}",␣
 ↪end_token_layer(test_batch_x).shape)
```

```
English sentences batch shape:                          (16, 13, 128)
English sentences batch shape after adding end token:   (16, 14, 128)
```

1.5 4. Build the encoder network

The encoder network follows the schematic diagram above. You should now build the RNN encoder model. * Using the functional API, build the encoder network according to the following spec: * The model will take a batch of sequences of embedded English words as input, as given by the Dataset objects. * The next layer in the encoder will be the custom layer you created previously, to add a learned end token embedding to the end of the English sequence. * This is followed by a Masking layer, with the `mask_value` set to the distinct padding value you used when you padded the English sequences with the Dataset preprocessing above. * The final layer is an LSTM layer with 512 units, which also returns the hidden and cell states. * The encoder is a multi-output model. There should be two output Tensors of this model: the hidden state and cell states of the LSTM layer. The output of the LSTM layer is unused. * Using the Dataset `.take(1)` method, extract a batch of English data examples from the training Dataset and test the encoder model by calling it on the English data Tensor, and print the shape of the resulting Tensor outputs. * Print the model summary for the encoder network.

[27]:
```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import LSTM, Masking
from tensorflow.keras.utils import plot_model
```

[28]:
```
def get_encoder(input_shape=(None, 128)):
    inputs = Input(shape=input_shape, name="enc_input")
    h = AddEndTokenEmbedding(embedding_dim=128,␣
 ↪name="add_end_token_emb")(inputs)
    h = Masking(mask_value=0., name="mask")(h)
    _, hidden_state, cell_state = LSTM(units=512, return_sequences=True,␣
 ↪return_state=True, name="enc_lstm")(h)

    return Model(inputs=inputs, outputs=[hidden_state, cell_state])
```

[29]:
```
encoder = get_encoder(input_shape=(13, 128))
enc_h_state, enc_c_state = encoder(test_batch_x)
print(f"{'Encoder hidden state shape:':<30}", enc_h_state.shape)
print(f"{'Encoder cell state shape:':<30}", enc_c_state.shape)
```

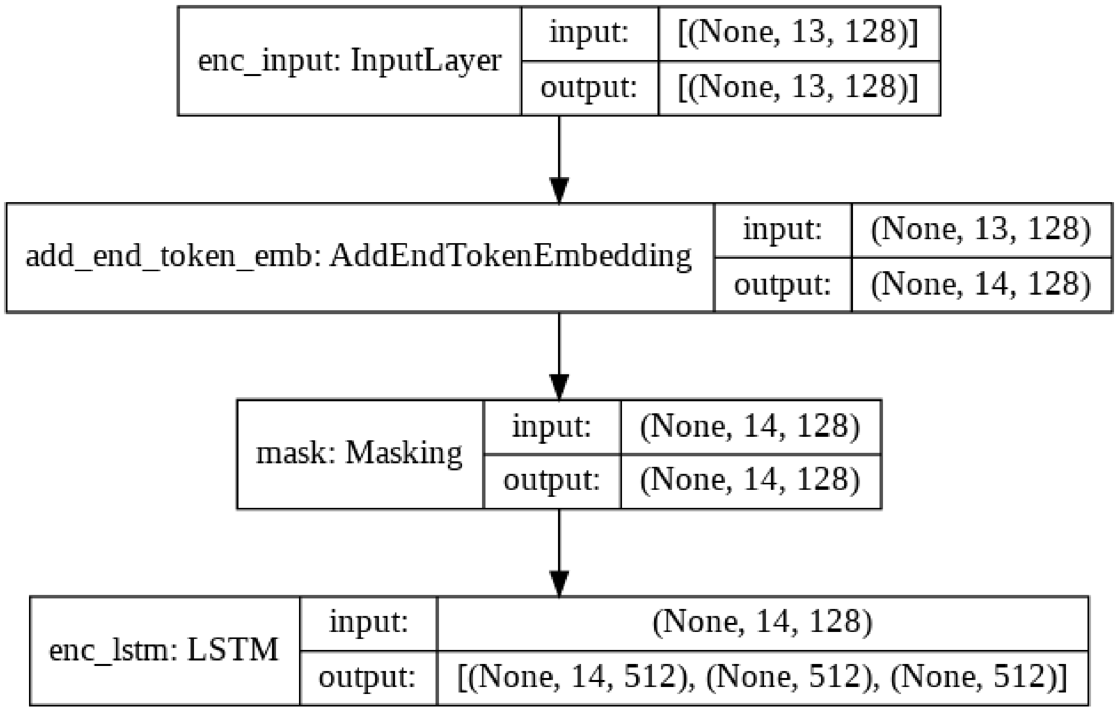
```
Encoder hidden state shape:    (16, 512)
Encoder cell state shape:      (16, 512)
```

[30]:
```
encoder.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
enc_input (InputLayer)       [(None, 13, 128)]         0
_________________________________________________________________
add_end_token_emb (AddEndTok (None, 14, 128)           128
_________________________________________________________________
mask (Masking)               (None, 14, 128)           0
_________________________________________________________________
enc_lstm (LSTM)              [(None, 14, 512), (None,  1312768
=================================================================
Total params: 1,312,896
Trainable params: 1,312,896
Non-trainable params: 0
_________________________________________________________________
```

[31]:
```
plot_model(encoder, show_shapes=True)
```

[31]:

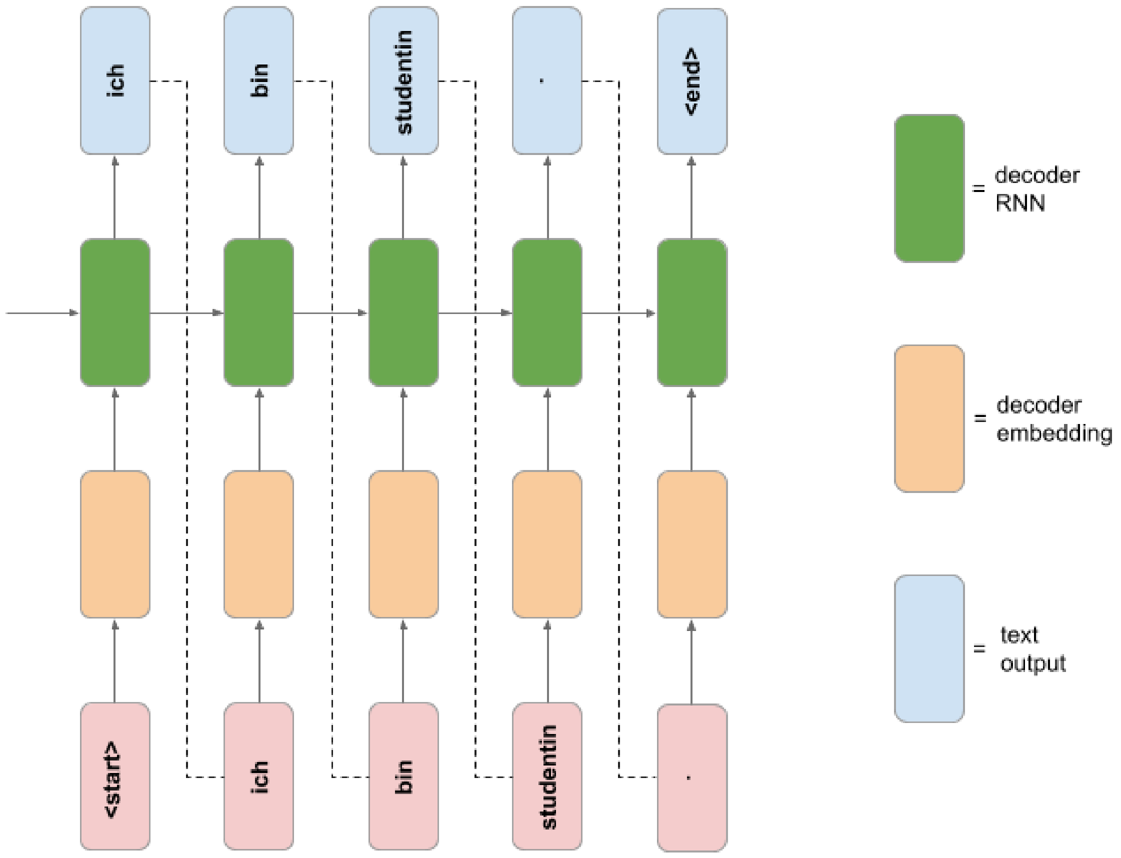


1.6 5. Build the decoder network

The decoder network follows the schematic diagram below.

[32]:
```
# 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")
```

[32]:



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.

[33]:
```
from tensorflow.keras.layers import Dense, Embedding
```

[34]:
```python
class DecoderNetwork(Model):

    def __init__(self, word_index, **kwargs):
        super(DecoderNetwork, self).__init__(**kwargs)
        vocabulary_size = max(word_index.values())
        self.embedding = Embedding(input_dim=vocabulary_size+1, output_dim=128,␣
 ↪mask_zero=True, input_length=14, name="embedding")
        self.lstm = LSTM(units=512, return_sequences=True, return_state=True,␣
 ↪name="decoder_lstm")
        self.dense = Dense(units=vocabulary_size+1, name="decoder_dense")

    def call(self, inputs, hidden_state=None, cell_state=None):
        embeddings = self.embedding(inputs)
        if hidden_state is not None and cell_state is not None:
            lstm_output, hidden_state, cell_state = self.lstm(embeddings,␣
 ↪initial_state=[hidden_state, cell_state])
        else:
            lstm_output, hidden_state, cell_state = self.lstm(embeddings)
        return self.dense(lstm_output), hidden_state, cell_state
```

[35]:
```python
decoder = DecoderNetwork(word_index=tokenizer.word_index)
dec_output, dec_h_state, dec_c_state = decoder(test_batch_y,␣
 ↪*encoder(test_batch_x))
print(f"{'Decoder outputs shape:':<30}", dec_output.shape)
print(f"{'Decoder hidden state shape:':<30}", dec_h_state.shape)
print(f"{'Decoder cell state shape:':<30}", dec_c_state.shape)
```

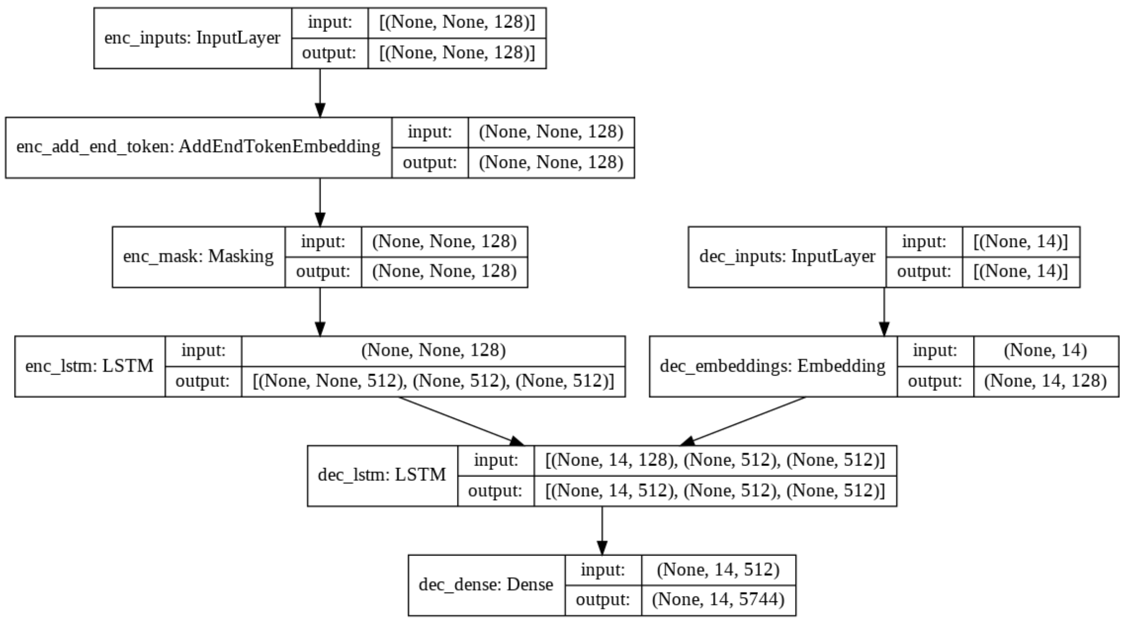
```
Decoder outputs shape:        (16, 14, 5744)
Decoder hidden state shape:   (16, 512)
Decoder cell state shape:     (16, 512)
```

[36]:
```python
decoder.summary()
```

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

[37]:
```
# For illustration here is the full encoder-decoder architecture using the␣
↪functional API
def get_translation_model(word_index, enc_input_shape=(None, 128),␣
↪dec_input_shape=(14,)):
    # Encoder
    enc_inputs = Input(shape=enc_input_shape, name="enc_inputs")
    add_token = AddEndTokenEmbedding(embedding_dim=128,␣
↪name="enc_add_end_token")(enc_inputs)
    mask = Masking(mask_value=0., name="enc_mask")(add_token)
    _, enc_hidden_state, enc_cell_state = LSTM(units=512,␣
↪return_sequences=True, return_state=True, name="enc_lstm")(mask)
    # Decoder
    vocabulary_size = max(word_index.values())
    dec_inputs = Input(shape=dec_input_shape, name="dec_inputs")
    embeddings = Embedding(input_dim=vocabulary_size+1, output_dim=128,␣
↪mask_zero=True, input_length=dec_input_shape[0],␣
↪name="dec_embeddings")(dec_inputs)
    lstm_output, dec_hidden_state, dec_cell_state = LSTM(units=512,␣
↪return_sequences=True, return_state=True, name="dec_lstm")(embeddings,␣
↪initial_state=[enc_hidden_state, enc_cell_state])
    dense = Dense(vocabulary_size+1, name="dec_dense")(lstm_output)
    return Model(inputs=[enc_inputs, dec_inputs], outputs=[dense,␣
↪dec_hidden_state, dec_cell_state])

plot_model(get_translation_model(tokenizer.word_index), show_shapes=True)
```

[37]:


enc_inputs: InputLayer | input: [(None, None, 128)] | output: [(None, None, 128)]
enc_add_end_token: AddEndTokenEmbedding | input: (None, None, 128) | output: (None, None, 128)
enc_mask: Masking | input: (None, None, 128) | output: (None, None, 128)
dec_inputs: InputLayer | input: [(None, 14)] | output: [(None, 14)]
enc_lstm: LSTM | input: (None, None, 128) | output: [(None, None, 512), (None, 512), (None, 512)]
dec_embeddings: Embedding | input: (None, 14) | output: (None, 14, 128)
dec_lstm: LSTM | input: [(None, 14, 128), (None, 512), (None, 512)] | output: [(None, 14, 512), (None, 512), (None, 512)]
dec_dense: Dense | input: (None, 14, 512) | output: (None, 14, 5744)

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.

[38]:
```
import matplotlib.pyplot as plt
```

[40]:
```
def german_data_provider(german_dataset):
    """Creates input and output german datasets.
    The input is obtained by removing the `<start>` tokens while the output
    is obtained by removing the `<end>` token. In both cases the size of the
    tensors is preserved by replacing the tokens by zeros.
    """
    # For the input, set a condition where the german tokenized dataset is␣
↪different from the '<end>' token (!=2)
    condition = tf.not_equal(
        tf.cast(german_dataset, tf.float32),
        tf.constant(2., dtype=tf.float32)
    )
    # Then use tf.where to replace these locations with a 0.
    german_in = tf.where(
        condition=condition,
        x=tf.cast(german_dataset, tf.float32),
        y=tf.constant(0., tf.float32))

    # For the input, remove the first column of the dataset which is filled␣
↪with the '<start>' tokens (==1)
    german_out = tf.concat([
         tf.cast(german_dataset[:, 1:], dtype=tf.float32),
         tf.zeros((16, 1), dtype=tf.float32)
        ], axis=1
    )
    return german_in, german_out

german_in, german_out = german_data_provider(test_batch_y)
print(german_in.shape, german_out.shape)
```

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

[41]:
```
def get_loss(encoder, decoder, english_input, german_input, german_output,␣
 ↪loss_fn):
    """Calculates the loss using the `loss_fn` provided, for a given set of␣
 ↪encoder-decoder models and english / german datasets.
    """
    hidden_state, cell_state = encoder(english_input)
    german_preds, *_ = decoder(german_input, hidden_state=hidden_state,␣
 ↪cell_state=cell_state)
    return loss_fn(y_true=german_output, y_pred=german_preds)

def get_grads(encoder, decoder, english_input, german_input, german_output,␣
 ↪loss_fn):
    """Performs the both the forward and backwards pass for a training step.
    Returns the loss for the given step plus the gradients.
    """
    with tf.GradientTape() as tape:
        current_loss = get_loss(encoder, decoder, english_input, german_input,␣
 ↪german_output, loss_fn)
        grads = tape.gradient(target=current_loss, sources=encoder.
 ↪trainable_variables+decoder.trainable_variables)
    return current_loss, grads


l, g = get_grads(
    encoder=encoder,
    decoder=decoder,
    english_input=test_batch_x,
    german_input=german_in,
    german_output=german_out,
    loss_fn=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
)
print("Loss:", l)
print("Number of gradient sets:", len(g))
```

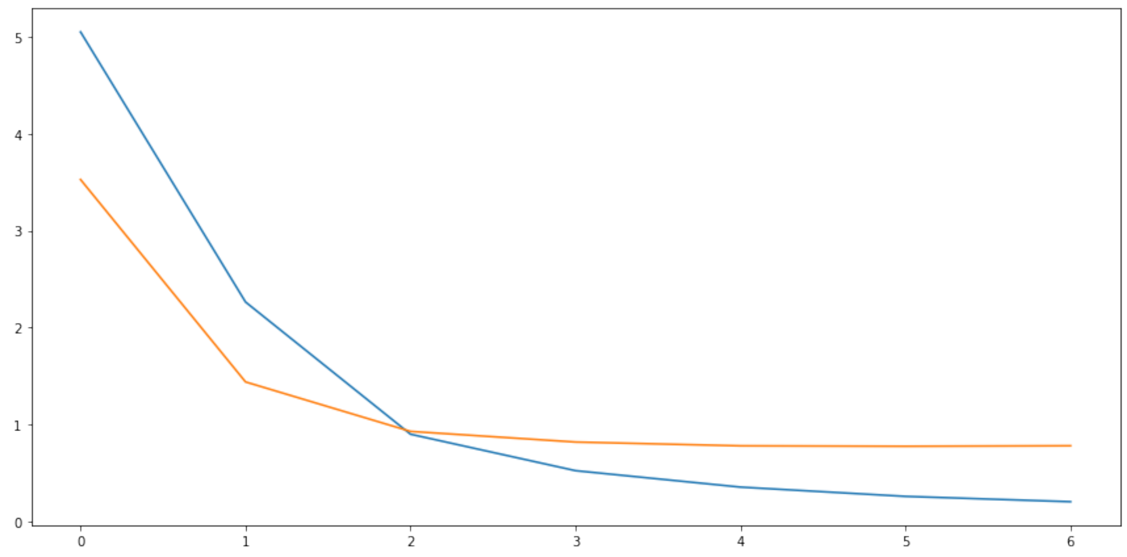
```
Loss: tf.Tensor(8.655368, shape=(), dtype=float32)
Number of gradient sets: 10
```

[42]:
```
def early_stopping(loss_list, min_delta=0.05, patience=3):
    # No early stopping for 2 * patience epochs
    if len(loss_list) // patience < 2:
        return False
    # Mean loss for last patience epochs and second-last patience epochs
    mean_previous = np.mean(loss_list[::-1][patience:2*patience]) #second-last
    mean_recent = np.mean(loss_list[::-1][:patience]) #last
    #you can use relative or absolute change
    delta_abs = np.abs(mean_recent - mean_previous) #abs change
    delta_abs = np.abs(delta_abs / mean_previous)  # relative change
    if delta_abs < min_delta :
        print(f"Early stopping: Validation loss didn't change much from last␣
 ↪{patience} epochs")
        print(f"Early stopping: Percent change in loss value: {delta_abs*1e2}%")
        return True
    else:
        return False

def training_routine(encoder, decoder, train_dataset, val_dataset, num_epochs,␣
 ↪loss_fn, optimizer, grad_fn, min_delta, patience):
    """Trains a encoder-decoder model to perform english to german translation.
    Returns two lists, containing average training and validation losses per␣
 ↪epoch.
    """
    train_loss_results = []
    val_loss_results = []

    for epoch in range(num_epochs):

        # Instantiate objects to average loss per epoch
        epoch_train_loss_avg = tf.keras.metrics.Mean()
        epoch_val_loss_avg = tf.keras.metrics.Mean()

        for english_input, german in train_dataset:
            # Create german_input and german_output
            german_input, german_output = german_data_provider(german)
            # Compute current loss and grads
            current_train_loss, grads = grad_fn(encoder, decoder,␣
 ↪english_input, german_input, german_output, loss_fn)
            # Apply grads to the trainable variables
            optimizer.apply_gradients(zip(grads, encoder.trainable_variables +␣
 ↪decoder.trainable_variables))


            # Compute training loss
            epoch_train_loss_avg(current_train_loss)

        for english_input, german in val_dataset:
            german_input, german_output = german_data_provider(german)
            current_val_loss = get_loss(encoder, decoder, english_input,␣
↪german_input, german_output, loss_fn)

            # Compute validation loss
            epoch_val_loss_avg(current_val_loss)

        # End epoch
        train_loss_results.append(epoch_train_loss_avg.result())
        val_loss_results.append(epoch_val_loss_avg.result())

        print(
            f"Epoch {epoch}:",
            f"Training Loss: {epoch_train_loss_avg.result():.3f}",
            f"Validation Loss: {epoch_val_loss_avg.result():.3f}"
        )

        # Early stopping
        stop_early = early_stopping(val_loss_results, min_delta=min_delta,␣
↪patience=patience)
        if stop_early:
            print(f"Early stopping signal received at epoch {epoch}/
↪{num_epochs}")
            print("Terminating training")
            break

    return train_loss_results, val_loss_results
```

[43]:
```
%%time
# Instantiate arguments for training routine
encoder = get_encoder(input_shape=(None, 128))
decoder = DecoderNetwork(word_index=tokenizer.word_index)
num_epochs = 50
optimizer = tf.keras.optimizers.Nadam(learning_rate=3e-3)
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Run training routine
train_loss_results, val_loss_results = training_routine(
    encoder=encoder,
    decoder=decoder,
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    num_epochs=num_epochs,
    loss_fn=loss_fn,
    optimizer=optimizer,
    grad_fn=get_grads,
    min_delta=0.05,
    patience=2,
)
```

```
Epoch 0: Training Loss: 5.050 Validation Loss: 3.527
Epoch 1: Training Loss: 2.264 Validation Loss: 1.439
Epoch 2: Training Loss: 0.900 Validation Loss: 0.929
Epoch 3: Training Loss: 0.524 Validation Loss: 0.819
Epoch 4: Training Loss: 0.354 Validation Loss: 0.780
Epoch 5: Training Loss: 0.259 Validation Loss: 0.776
Early stopping: Validation loss didn't change much from last 2 epochs
Early stopping: Percent change in loss value: 2.693014033138752%
Early stopping signal received at epoch 6/50
Terminating training
CPU times: user 21min 41s, sys: 1min 48s, total: 23min 29s
Wall time: 29min 58s
```

[44]:
```
def plot_learning_curves(train_loss, val_loss):
    plt.figure(figsize=(12, 6))
    plt.plot(train_loss)
    plt.plot(val_loss)
    plt.tight_layout()
    plt.show()

plot_learning_curves(train_loss_results, val_loss_results)
```



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.

[45]:
```
# Create inverted word index
inv_word_index = {value: key for key, value in tokenizer.word_index.items()}
```

[46]:
```
# Select 5 random english sentences
rnd_idx = np.random.choice(range(len(english_sentences)), size=5)
english_sample, german_sample, german_token_sample =␣
 ↪english_sentences[rnd_idx], german_sentences[rnd_idx],␣
 ↪tokenized_german_sentences[rnd_idx]

for e, g, gt in zip(english_sample, german_sample, german_token_sample):
    print(e, g, gt)
```

```
tom sounds mad . <start> tom klingt verrueckt . <end> [   1    5 609 245    3    2
0    0    0    0    0    0    0    0]
tom would leave . <start> tom wuerde gehen . <end> [   1    5 209   45    3    2    0
0    0    0    0    0    0    0]
i was in a hurry . <start> ich war in eile . <end> [    1     4    24    46  1070
3     2     0     0     0     0     0     0     0]
come over . <start> komm her ! <end> [   1   74 212    9    2    0    0    0    0    0
0    0    0    0]
tom betrayed you . <start> tom hat dich betrogen . <end> [   1    5   16   28 926
3    2    0    0    0    0    0    0    0]
```

[47]:
```
def full_preprocessing(dataset):
    dataset = dataset.map(split_english_sentences)
    dataset = dataset.map(create_embeddings)
    dataset = dataset.filter(filter_get_13)
    dataset = dataset.map(pad_embeddings)
    return dataset

# Preprocess and embed english sentences
sample_dataset = tf.data.Dataset.from_tensor_slices((english_sample,␣
 ↪german_token_sample))
sample_dataset = full_preprocessing(sample_dataset)
sample_dataset = sample_dataset.batch(1)
check_dataset(sample_dataset, iter=5)
```

```
(1, 13, 128) (1, 14)
(1, 13, 128) (1, 14)
```

```
(1, 13, 128) (1, 14)
(1, 13, 128) (1, 14)
(1, 13, 128) (1, 14)
```

[48]:
```
# Create outer list to accumulate examples
sample_preds = []
for english_input, german_input in sample_dataset:
    # Run the english sentence through the encoder
    encoder_h_state, encoder_c_state = encoder(english_input)
    # Create inner list to accumulate the sequence in each example
    example_preds = []
    # Loop over sequence length (max 14 items)
    for i in range(german_input.shape[-1]):
        # Stop loop if german input is 0
        if german_input[None, :, i] == 0:
                break
        # For first iteration in loop, use encoder hidden states
        if i == 0:
            decoder_pred, decoder_h_state, decoder_c_state =␣
 ↪decoder(german_input[None, :, i], encoder_h_state, encoder_c_state)
        # For the subsequent ones use the decoder hidden states
        else:
            decoder_pred, decoder_h_state, decoder_c_state =␣
 ↪decoder(german_input[None, :, i], decoder_h_state, decoder_c_state)
        # Find the token with highest probability
        pred = tf.argmax(tf.squeeze(decoder_pred), axis=-1).numpy()
        # Stop loop if end token is the one predicted
        if pred == 2:
            break
        # Otherwise append pred to sequence
        else:
            example_preds.append(pred)

    # Create the output the sequence of tokens and the translated german␣
 ↪sentence
    # for each example using the inverted word index from the tokenizer
    tuple_preds = (example_preds, " ".join([inv_word_index.get(token, "") for␣
 ↪token in example_preds]))

    # Append each example to the outer list
    sample_preds.append(tuple_preds)
```

[49]:
```
print(f"{'English sentence':<30}{'German Translation'}\n")
for e, gt in zip(english_sample, sample_preds):
    print(f"{e:<30}{gt[1]}")
```

```
English sentence              German Translation
```

```
tom sounds mad .              tom klingt verrueckt .
tom would leave .             tom wuerde entkommen .
i was in a hurry .            ich war in einer .
come over .                   komm um !
tom betrayed you .            tom hat dich verraten .
```

[49]:

```

```