

Capstone Project

November 24, 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 (File -> Download as -> PDF via LaTeX). You should then submit this pdf for review.

1.1.3 Let's get started!

We'll start by running some imports, and loading the dataset. For this project you are free to make further imports throughout the notebook as you wish.

```
In [1]: from google.colab import drive
        drive.mount('/content/drive')
```

Mounted at /content/drive

```
In [2]: import tensorflow as tf
        import numpy as np
        import tensorflow_hub as hub
        import unicodedata
        import re
```



Flags overview image

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Layer, concatenate, Masking, LSTM, Dense, Input, Embedding
```

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.

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

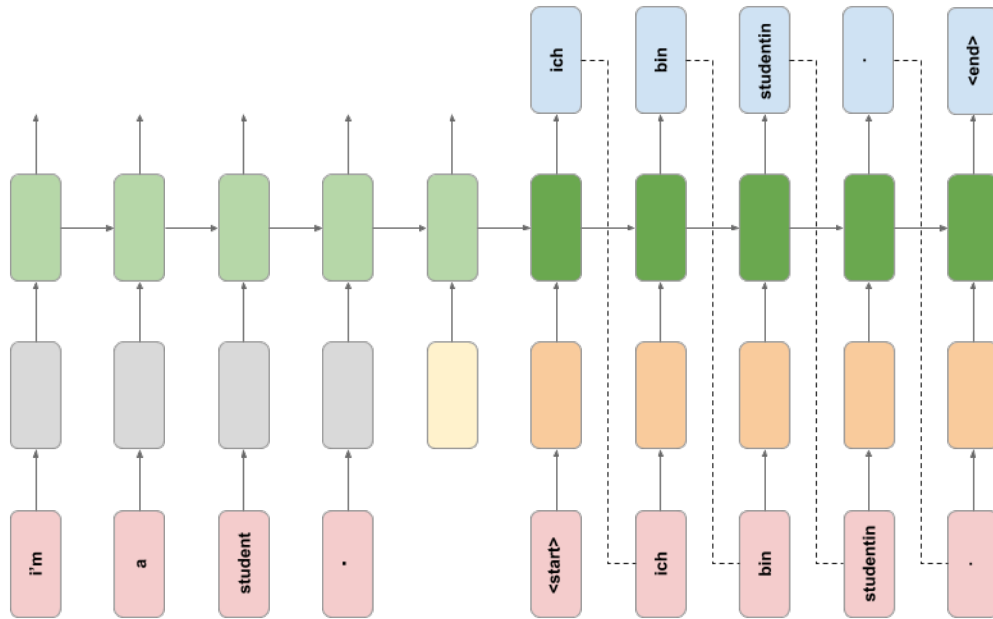
```
NUM_EXAMPLES = 20000
data_examples = []
with open('/content/drive/MyDrive/Customizing your models with tensorflow 2/Week 5 Capstone Project/data/train.txt') as f:
    for line in f.readlines():
        if len(data_examples) < NUM_EXAMPLES:
            data_examples.append(line)
        else:
            break
```

In [4]: *# These functions preprocess English and German sentences*

```
def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s) if unicodedata.category(c) != '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)
```



Model Schematic

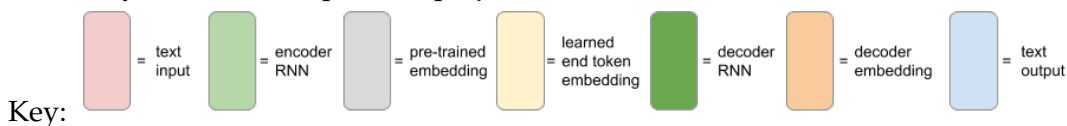
```

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.



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

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

1.2 1. Text preprocessing

- Create separate lists of English and German sentences, and preprocess them using the `preprocess_sentence` function provided for you above.

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

```
In [5]: english_sentences = []
        german_sentences = []

        for sentence in data_examples:
            texts = sentence.split('\t')
            english_sentences.append(preprocess_sentence(texts[0]))
            german_sentences.append("<start> " + preprocess_sentence(texts[1]) + " <end>")
```

```
In [6]: tokenizer = Tokenizer(filters='')

        tokenizer.fit_on_texts(german_sentences)

        german_sentences_tokenized = tokenizer.texts_to_sequences(german_sentences)
```

```
In [7]: idx = np.random.choice(len(english_sentences), 5)
        print(f"Selected Sentences: {idx}")
        print()

        for i in idx:
            print(f"English Sentence: {english_sentences[i]}")
            print(f"German Translation: {german_sentences[i]}")
            print(f"Tokenized German Sentence: {german_sentences_tokenized[i]}")
            print()
```

```
Selected Sentences: [ 936 1053 16444 10355 3402]
```

```
English Sentence: what's on ?
German Translation: <start> was ist los ? <end>
Tokenized German Sentence: [1, 38, 6, 207, 7, 2]
```

```
English Sentence: drink this .
German Translation: <start> trink das ! <end>
Tokenized German Sentence: [1, 680, 11, 9, 2]
```

```
English Sentence: tom has an ulcer .
German Translation: <start> tom hat ein geschwuer . <end>
Tokenized German Sentence: [1, 5, 16, 19, 5166, 3, 2]
```

English Sentence: i cooked dinner .
German Translation: <start> ich habe das abendessen gekocht . <end>
Tokenized German Sentence: [1, 4, 18, 11, 773, 2558, 3, 2]

English Sentence: i'll ask tom .
German Translation: <start> ich frage mal tom . <end>
Tokenized German Sentence: [1, 4, 541, 76, 5, 3, 2]

```
In [8]: padded_german_sentences = pad_sequences(german_sentences_tokenized, padding='post')
        padded_german_sentences_array = np.array(padded_german_sentences)
        print(padded_german_sentences_array.shape)

(20000, 14)
```

1.3 2. Prepare the data with tf.data.Dataset objects

Load the embedding layer As part of the dataset preprocessing 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 module has also been made available as a complete saved model in the folder './models/tf2-preview_nnlm-en-dim128_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 [9]: # 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 [10]: # Test the layer
```

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

```
Out[10]: 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 more 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 [11]: from sklearn.model_selection import train_test_split
```

```
        x_train, x_val, y_train, y_val = train_test_split(english_sentences, padded_german_sentences,
```

```
In [12]: train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
        val_dataset = tf.data.Dataset.from_tensor_slices((x_val, y_val))
```

```
In [13]: def split_english_sentences(x,y):
        x_l = tf.strings.split(x, sep=' ')
        return (x_l,y)

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

```
In [14]: def embed_english_seqs(x,y):
        embeddings = embedding_layer(x)
        return (embeddings, y)

        train_dataset = train_dataset.map(embed_english_seqs)
        val_dataset = val_dataset.map(embed_english_seqs)
```

```
In [15]: train_dataset.element_spec
```

```
Out[15]: (TensorSpec(shape=(None, 128), dtype=tf.float32, name=None),
        TensorSpec(shape=(14,), dtype=tf.int32, name=None))
```

```
In [16]: def remove_longer_english_seqs(x,y):
        return len(x)<=13
```

```
train_dataset = train_dataset.filter(remove_longer_english_seqs)
val_dataset = val_dataset.filter(remove_longer_english_seqs)
```

```
In [17]: def pad_english_seqs(x,y):
```

```
    paddings = tf.constant([[13,0],[0,0]])
    x_padded = tf.pad(x, paddings)
```

```
    return x_padded[-13:,:], y
```

```
train_dataset = train_dataset.map(pad_english_seqs)
val_dataset = val_dataset.map(pad_english_seqs)
```

```
In [18]: train_dataset = train_dataset.batch(16)
val_dataset = val_dataset.batch(16)
```

```
In [19]: for x, y in train_dataset.take(1):
    print(x.shape)
    print(y)
```

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

```
tf.Tensor(
```

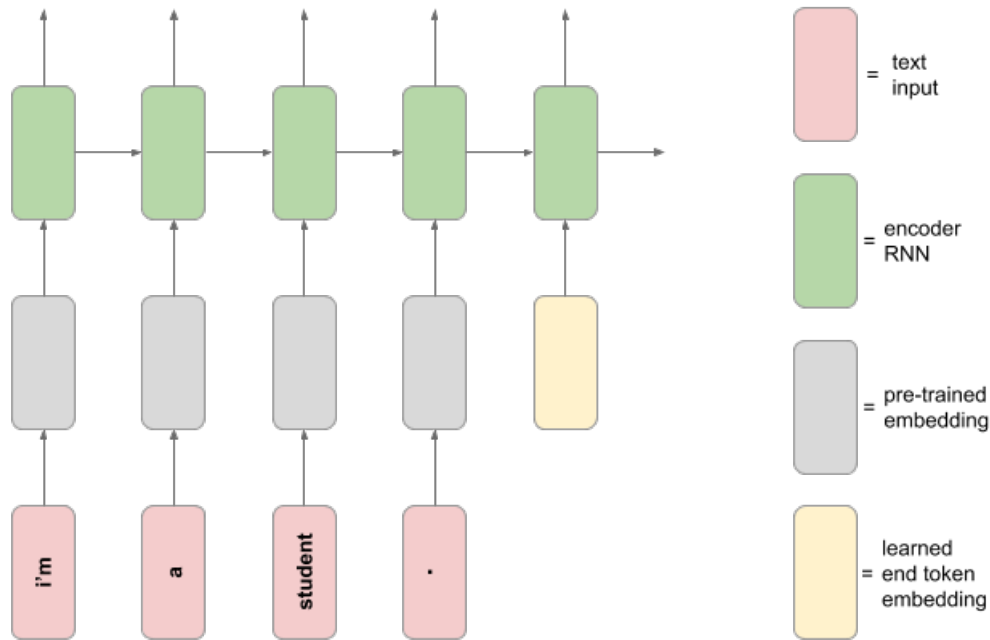
```
[[ 1  8 23 62 138 3 2 0 0 0 0 0 0 0]
 [ 1  4 2113 57 654 3 2 0 0 0 0 0 0 0]
 [ 1  4 15 71 1147 3 2 0 0 0 0 0 0 0]
 [ 1 67 631 8 34 3 2 0 0 0 0 0 0 0]
 [ 1 131 8 11 46 378 9 2 0 0 0 0 0 0]
 [ 1 158 6 102 3 2 0 0 0 0 0 0 0 0]
 [ 1 4 18 60 2814 3 2 0 0 0 0 0 0 0]
 [ 1 38 2015 7 2 0 0 0 0 0 0 0 0 0]
 [ 1 239 11 12 72 76 9 2 0 0 0 0 0 0]
 [ 1 67 882 28 9 2 0 0 0 0 0 0 0 0]
 [ 1 5 16 22 2627 3 2 0 0 0 0 0 0 0]
 [ 1 14 434 12 20 3 2 0 0 0 0 0 0 0]
 [ 1 8 1935 104 199 3 2 0 0 0 0 0 0 0]
 [ 1 74 12 20 219 9 2 0 0 0 0 0 0 0]
 [ 1 14 6 19 2669 3 2 0 0 0 0 0 0 0]
 [ 1 10 6 108 465 3 2 0 0 0 0 0 0 0]], shape=(16, 14), dtype=
```

```
In [20]: for x, y in val_dataset.take(1):
    print(x.shape)
    print(y)
```

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

```
tf.Tensor(
```

```
[[ 1 406 104 2247 36 9 2 0 0 0 0 0 0 0]
 [ 1 10 229 223 751 313 1070 3 2 0 0 0 0 0]
```



Encoder schematic

```
[ 1  14  6 194 322 668  3  2  0  0  0  0  0  0]
[ 1 4736 22  12  3  2  0  0  0  0  0  0  0  0]
[ 1  5  6 2720  3  2  0  0  0  0  0  0  0  0]
[ 1  4 463  25  5 16 10 395  3  2  0  0  0  0]
[ 1 5480 21  19 931  3  2  0  0  0  0  0  0  0]
[ 1  5  6 297  3  2  0  0  0  0  0  0  0  0]
[ 1  4 15  62 1307  3  2  0  0  0  0  0  0  0]
[ 1 42 48 10 101  7  2  0  0  0  0  0  0  0]
[ 1 133 161 1556  5  3  2  0  0  0  0  0  0  0]
[ 1  4 18  44 244  3  2  0  0  0  0  0  0  0]
[ 1  5 110 171  41  3  2  0  0  0  0  0  0  0]
[ 1  4 440  26 761  3  2  0  0  0  0  0  0  0]
[ 1 38 1664  8  7  2  0  0  0  0  0  0  0  0]
[ 1 14 16 22 198 309 438  3  2  0  0  0  0  0]], shape=(16, 14), dtype=
```

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:

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 [21]: class AddLearnedEndTokenLayer(Layer):
```

```
    def __init__(self, **kwargs):

        super(AddLearnedEndTokenLayer, self).__init__(**kwargs)
        self.end_token_embed = tf.Variable(initial_value=tf.random.uniform(shape=(128,1)))

    def call(self, inputs):

        end_token = tf.reshape(self.end_token_embed, (1,1,128))
        end_token = tf.tile(end_token, [tf.shape(inputs)[0],1,1])

        return concatenate([inputs, end_token], axis=1)
```

```
In [22]: for x,y in train_dataset.take(1):
        print(x.shape)
        print(AddLearnedEndTokenLayer()(x).shape)
```

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

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

1.5 4. Build the encoder network

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

```
In [23]: inputs = Input(shape=(13,128))
        end_token = AddLearnedEndTokenLayer()(inputs)
        mask_layer = Masking(mask_value=-1)(end_token)
        lstm, hidden, cell = LSTM(512, return_state=True)(mask_layer)
        encoder_model = Model(inputs=inputs, outputs=[hidden, cell])
```

```
In [24]: for x,y in train_dataset.take(1):
        hidden_state, cell_state = encoder_model(x)
        print(hidden_state.shape)
        print(cell_state.shape)
```

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

```
In [25]: encoder_model.summary()
```

```
Model: "functional_1"
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 13, 128)]	0
add_learned_end_token_layer_	(None, 14, 128)	128
masking (Masking)	(None, 14, 128)	0
lstm (LSTM)	[(None, 512), (None, 512)]	1312768

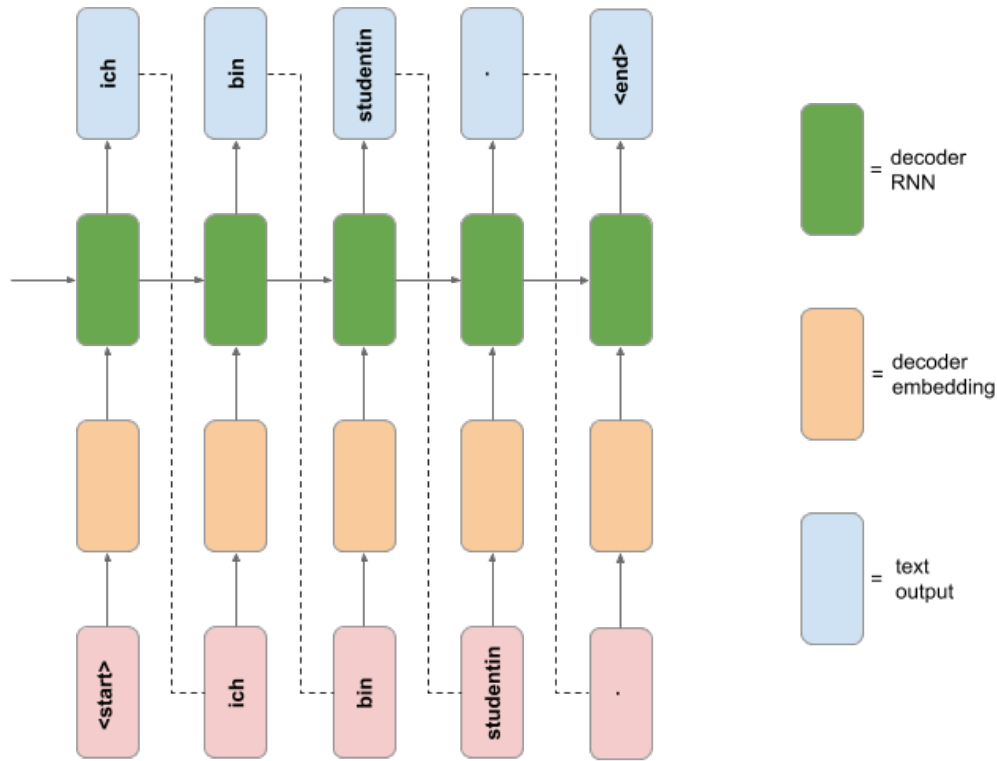
Total params: 1,312,896
Trainable params: 1,312,896
Non-trainable params: 0

1.6 5. Build the decoder network

The decoder network follows the schematic diagram below.

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 [26]: vocab_size = max(tokenizer.word_index.values()) + 1
```



Decoder schematic

```
In [27]: class decoder_model(Model):
```

```
    def __init__(self):
        super(decoder_model, self).__init__()
        self.embedding = Embedding(vocab_size, 128, mask_zero=True)
        self.lstm = LSTM(512, return_sequences=True, return_state=True)
        self.fc = Dense(vocab_size)

    def call(self, inputs, hidden_state=None, cell_state=None):

        x = self.embedding(inputs)
        x, h, c = self.lstm(inputs=x, initial_state=[hidden_state, cell_state])
        x = self.fc(x)

        return (x,h,c)
```

```
In [28]: decoder_model = decoder_model()
```

```
In [29]: for x,y in train_dataset.take(1):
        h, c = encoder_model(x)
        outputs, h, c = decoder_model(y, h, c)
        print(outputs.shape)
        print(h.shape)
        print(c.shape)
```

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

```
In [30]: decoder_model.summary()
```

```
Model: "decoder_model"
```

```
-----
Layer (type)                 Output Shape              Param #
-----
embedding (Embedding)        multiple                  735232
-----
lstm_1 (LSTM)                 multiple                  1312768
-----
dense (Dense)                 multiple                  2946672
=====
Total params: 4,994,672
Trainable params: 4,994,672
Non-trainable params: 0
-----
```

1.7 6. Make a custom training loop

You should now write a custom training loop to train your custom neural translation model.

- * Define a function that takes a Tensor batch of German data (as extracted from the training Dataset), and returns a tuple containing German inputs and outputs for the decoder model (refer to schematic diagram above).
- * Define a function that computes the forward and backward pass for your translation model. This function should take an English input, German input and German output as arguments, and should do the following:
 - * Pass the English input into the encoder, to get the hidden and cell states of the encoder LSTM.
 - * These hidden and cell states are then passed into the decoder, along with the German inputs, which returns a sequence of outputs (the hidden and cell state outputs of the decoder LSTM are unused in this function).
 - * The loss should then be computed between the decoder outputs and the German output function argument.
 - * The function returns the loss and gradients with respect to the encoder and decoder's trainable variables.
- * Decorate the function with `@tf.function`
- * Define and run a custom training loop for a number of epochs (for you to choose) that does the following:
 - * Iterates through the training dataset, and creates decoder inputs and outputs from the German sequences.
 - * Updates the parameters of the translation model using the gradients of the function above and an optimizer object.
 - * Every epoch, compute the validation loss on a number of batches from the validation and save the epoch training and validation losses.
 - * Plot the learning curves for loss vs epoch for both training and validation sets.

Hint: This model is computationally demanding to train. The quality of the model or length of training is not a factor in the grading rubric. However, to obtain a better model we recommend using the GPU accelerator hardware on Colab.

```
In [31]: def create_output_data(y):
         german_i = y[:, :-1]
```

```

    german_o = y[:,1:]
    return (german_i, german_o)

```

In [32]: @tf.function

```

def grad(x, y_i, y_o):

    with tf.GradientTape() as tape:
        h, c = encoder_model(x)
        y_hat, h, c = decoder_model(y_i, h, c)
        loss = tf.keras.losses.sparse_categorical_crossentropy(y_o, y_hat, from_logits=True)

    encoder_grads, decoder_grads = tape.gradient(loss, [encoder_model.trainable_variables, decoder_model.trainable_variables])

    return (loss, encoder_grads, decoder_grads)

```

In [33]: optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

In [36]: num_epochs = 100

```

epoch_training_losses = []
epoch_val_losses = []

```

```

for epoch in range(num_epochs):

    train_losses = tf.keras.metrics.Mean()
    val_losses = tf.keras.metrics.Mean()

    for x,y in train_dataset:
        y_i, y_o = create_output_data(y)
        loss, encoder_grads, decoder_grads = grad(x, y_i, y_o)
        optimizer.apply_gradients(zip(encoder_grads, encoder_model.trainable_variables))
        optimizer.apply_gradients(zip(decoder_grads, decoder_model.trainable_variables))
        train_losses(loss)

    epoch_training_losses.append(train_losses.result())

    for x,y in val_dataset:
        y_i, y_o = create_output_data(y)
        h_val, c_val = encoder_model(x)
        y_hat, h_val, c_val = decoder_model(y_i, h_val, c_val)
        val_loss = tf.keras.losses.sparse_categorical_crossentropy(y_o, y_hat, from_logits=True)
        val_losses(val_loss)

    epoch_val_losses.append(val_losses.result())

    print(f"Epoch: {epoch+1} Training Loss: {train_losses.result():.2f} Validation Loss: {val_losses.result():.2f}")

```

Epoch: 1 Training Loss: 1.29 Validation Loss: 1.47

Epoch: 2 Training Loss: 1.16 Validation Loss: 1.50

```

Epoch: 3 Training Loss: 1.08 Validation Loss: 1.54
Epoch: 4 Training Loss: 1.04 Validation Loss: 1.57
Epoch: 5 Training Loss: 1.01 Validation Loss: 1.61
Epoch: 6 Training Loss: 0.98 Validation Loss: 1.63
Epoch: 7 Training Loss: 0.98 Validation Loss: 1.66
Epoch: 8 Training Loss: 0.96 Validation Loss: 1.71
Epoch: 9 Training Loss: 0.95 Validation Loss: 1.74
Epoch: 10 Training Loss: 0.95 Validation Loss: 1.77
Epoch: 11 Training Loss: 0.94 Validation Loss: 1.79
Epoch: 12 Training Loss: 0.94 Validation Loss: 1.82
Epoch: 13 Training Loss: 0.93 Validation Loss: 1.84
Epoch: 14 Training Loss: 0.93 Validation Loss: 1.89
Epoch: 15 Training Loss: 0.93 Validation Loss: 1.91
Epoch: 16 Training Loss: 0.93 Validation Loss: 1.92

```

KeyboardInterrupt

Traceback (most recent call last)

```

<ipython-input-36-54f79159a269> in <module>()
    10     for x,y in train_dataset:
    11         y_i, y_o = create_output_data(y)
---> 12     loss, encoder_grads, decoder_grads = grad(x, y_i, y_o)
    13     optimizer.apply_gradients(zip(encoder_grads, encoder_model.trainable_variables))
    14     optimizer.apply_gradients(zip(decoder_grads, decoder_model.trainable_variables))

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/def_function.py in __call__(self, *args, **kwargs)
    778     else:
    779         compiler = "nonXla"
--> 780         result = self._call(*args, **kwargs)
    781
    782         new_tracing_count = self._get_tracing_count()

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/def_function.py in _call(self, *args, **kwargs)
    812     # In this case we have not created variables on the first call. So we can
    813     # run the first trace but we should fail if variables are created.
--> 814     results = self._stateful_fn(*args, **kwargs)
    815     if self._created_variables:
    816         raise ValueError("Creating variables on a non-first call to a function")

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in __call__(self, *args, **kwargs)
    2827     with self._lock:
    2828         graph_function, args, kwargs = self._maybe_define_function(args, kwargs)

```

```

-> 2829     return graph_function._filtered_call(args, kwargs) # pylint: disable=protected
2830
2831     @property

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in _filtered
1846         resource_variable_ops.BaseResourceVariable))],
1847         captured_inputs=self.captured_inputs,
-> 1848         cancellation_manager=cancellation_manager)
1849
1850     def _call_flat(self, args, captured_inputs, cancellation_manager=None):

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in _call_fl
1922         # No tape is watching; skip to running the function.
1923         return self._build_call_outputs(self._inference_function.call(
-> 1924             ctx, args, cancellation_manager=cancellation_manager))
1925         forward_backward = self._select_forward_and_backward_functions(
1926             args,

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.py in call(sel
548         inputs=args,
549         attrs=attrs,
--> 550         ctx=ctx)
551     else:
552         outputs = execute.execute_with_cancellation(

/usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/execute.py in quick_exe
58     ctx.ensure_initialized()
59     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name, op_name,
---> 60         inputs, attrs, num_outputs)
61 except core._NotOkStatusException as e:
62     if name is not None:

```

KeyboardInterrupt:

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

```

fig, axes = plt.subplots(2, sharex=True, figsize=(12, 8))
fig.suptitle('Metrics')

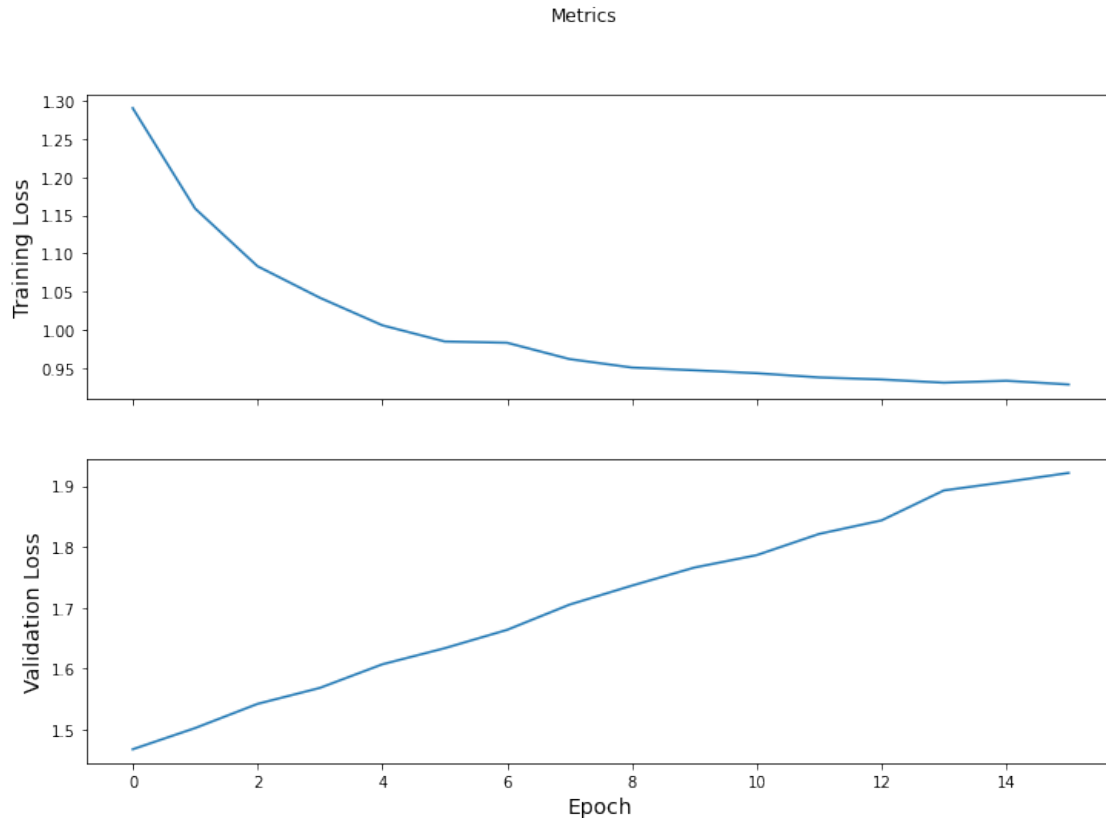
axes[0].set_ylabel("Training Loss", fontsize=14)
axes[0].plot(epoch_training_losses)

```

```

axes[1].set_ylabel("Validation Loss", fontsize=14)
axes[1].set_xlabel("Epoch", fontsize=14)
axes[1].plot(epoch_val_losses)
plt.show()

```



1.8 7. Use the model to translate

Now it's time to put your model into practice! You should run your translation for five randomly sampled English sentences from the dataset. For each sentence, the process is as follows: * Preprocess and embed the English sentence according to the model requirements. * Pass the embedded sentence through the encoder to get the encoder hidden and cell states. * Starting with the special "<start>" token, use this token and the final encoder hidden and cell states to get the one-step prediction from the decoder, as well as the decoder's updated hidden and cell states. * Create a loop to get the next step prediction and updated hidden and cell states from the decoder, using the most recent hidden and cell states. Terminate the loop when the "<end>" token is emitted, or when the sentence has reached a maximum length. * Decode the output token sequence into German text and print the English text and the model's German translation.

```

In [41]: start_token = tf.constant([[1]])
         start_token.numpy()[0][0]

```

Out [41]: 1


```

In [43]: ind_to_word = tokenizer.index_word

In [44]: type(ind_to_word)

Out[44]: dict

In [57]: idx = np.random.choice(len(english_sentences), 5)
print()

for j,i in enumerate(idx):
    eng_sen = english_sentences[i]
    print(f"Example: {j+1}")
    print(f"English Sentence: {eng_sen}")
    x_1 = tf.strings.split(eng_sen, sep=' ')
    embeddings = embedding_layer(x_1)
    x = tf.expand_dims(embeddings, 0)
    h, c = encoder_model(x)
    start_token = tf.constant([[1]])
    decoder_input = start_token
    german_translated = []
    while decoder_input.numpy()[0][0] != 2 :
        decoder_output, h, c = decoder_model(decoder_input, h, c)
        predicted_output_index = np.argmax(decoder_output, axis=-1)
        predicted_output_index = predicted_output_index[0][0]
        german_translated.append(ind_to_word[predicted_output_index])
        decoder_input = tf.constant([[predicted_output_index]])

    trans_german = " ".join(german_translated)

    print(f"Translated German Sentence: {trans_german}")
    print(f"original German Translation: {german_sentences[i]}")
    print()
    print()

```

Example: 1

English Sentence: it was black .

Translated German Sentence: ich bin ein bisschen groggy . <end>

original German Translation: <start> es war schwarz . <end>

Example: 2

English Sentence: it's not a fish .

Translated German Sentence: ich bin ein moench . <end>

original German Translation: <start> das ist kein fisch . <end>

Example: 3

English Sentence: what's her name ?

Translated German Sentence: ich bin ein bisschen groggy . <end>
original German Translation: <start> wie ist ihr name ? <end>

Example: 4

English Sentence: i can't help it .

Translated German Sentence: ich bin ein moench . <end>

original German Translation: <start> ich kann nichts dafuer . <end>

Example: 5

English Sentence: there are more .

Translated German Sentence: ich bin ein bisschen groggy . <end>

original German Translation: <start> es gibt mehr davon . <end>

In []:

In []:

In []: