

# 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
```

For the capstone project, you will use a language dataset from <http://www.manythings.org/anki/> (<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=1KczOciG7sYY7SB9UIBeRP1T9659b121Q> (<https://drive.google.com/open?id=1KczOciG7sYY7SB9UIBeRP1T9659b121Q>)

You should store the unzipped folder in Drive for use in this Colab notebook.

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

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

Drive already mounted at /content/gdrive; to attempt to forcibly remount, call drive.mount("/content/gdrive", force\_remount=True).

```
In [3]: !ls /content/gdrive/MyDrive/data/manythings_deu
        !cp /content/gdrive/MyDrive/data/manythings_deu/deu.txt .
```

deu.txt

```
In [4]: # Run this cell to load the dataset
```

```
NUM_EXAMPLES = 20000
data_examples = []
with open('deu.txt', 'r', encoding='utf8') as f:
    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)
!= 'Mn')

def preprocess_sentence(sentence):
    sentence = sentence.lower().strip()
    sentence = re.sub(r"ü", 'ue', sentence)
    sentence = re.sub(r"ä", 'ae', sentence)
    sentence = re.sub(r"ö", 'oe', sentence)
    sentence = re.sub(r"ß", 'ss', sentence)

    sentence = unicode_to_ascii(sentence)
    sentence = re.sub(r"([?!.])", r" \1 ", sentence)
    sentence = re.sub(r"^[^a-z?!.]+", " ", sentence)
    sentence = re.sub(r'[" "]+' , " ", sentence)

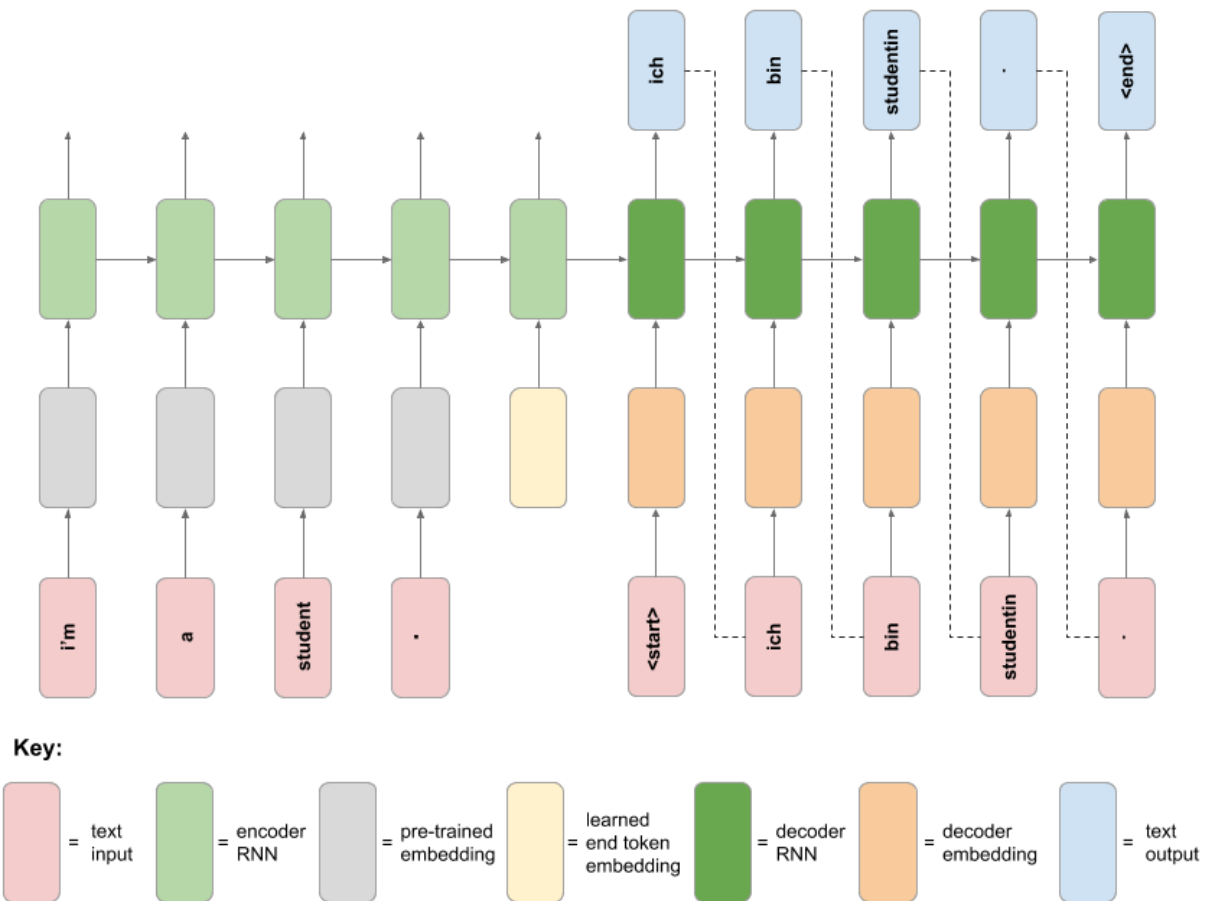
    return sentence.strip()
```

## The custom translation model

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

```
In [6]: # 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")
```



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.

## Examine the Data Format

Observation:

- Words are delimited by a tab (\t)
- First item in the list is English and the second item is German
- Iterate the sentence, split by tab and append to a list

```
In [7]: DEV = False

In [8]: index = 16
        sentence = data_examples[index]
        print(len(sentence), sentence)
        english, german = sentence.split("\t")[0], sentence.split("\t")[1]
        print(english, german)
        preprocess_sentence(english), preprocess_sentence(german)

107 I won!      Ich hab gewonnen!      CC-BY 2.0 (France) Attribution: tatoeba.org #2005192 (CK) & #4941621 (Hans_Adler)

I won! Ich hab gewonnen!

('i won !', 'ich hab gewonnen !')
```

## Preprocess

- Split the data and make a sequence of texts
- Preprocess the text array
- Append prefix and suffix for German data
- Tokenize, Convert string to sequences and padd the sequence.

```
In [9]: import pandas as pd
        def get_preprocessed_splits(data):

            splits = pd.Series(data).apply(lambda x: x.split("\t"))
            lang_1 = []
            lang_2 = []
            for a_split in splits:
                lang_1.append(preprocess_sentence(a_split[0]))
                lang_2.append("<start> " + preprocess_sentence(a_split[1]) + " <end>")

            return lang_1, lang_2

In [10]: if(DEV == True):
        sliced = data_examples[0:10]
        english, german = get_preprocessed_splits(sliced)
        for en, de in zip(english, german):
            print(f"{en} == {de}")
        else:
            english, german = get_preprocessed_splits(data_examples)
```

```
In [11]: def create_tokenizer(list_of_strings):
        """
        This function takes a list of strings as its argument. It should create
        and return a Tokenizer according to the above specifications.
        """
        tokenizer = tf.keras.preprocessing.text.Tokenizer(
            num_words=None,
            char_level=False,
            lower=True,
            filters='',
            split=' ',
            # oov_token='<UNK>',
            # document_count=len(list_of_strings)
        )
        tokenizer.fit_on_texts(list_of_strings)
        return tokenizer

def strings_to_sequences(tokenizer, list_of_strings):
    """
    This function takes a tokenizer object and a list of strings as its arguments.
    It should use the tokenizer to map the text chunks to sequences of tokens and
    then return this list of encoded sequences.
    """
    return tokenizer.texts_to_sequences(list_of_strings)
#english_tokens = create_tokenizer(english)
tokenizer = create_tokenizer(german)
seq_chunks = strings_to_sequences(tokenizer, german)
```

- Find the length of the largest string to identify max length of the padded sequence.

```
In [12]: len(max(seq_chunks, key=len))
```

14

```
In [13]: padded_german = tf.keras.preprocessing.sequence.pad_sequences(
        seq_chunks,
        maxlen=14,
        padding="post",
        value=0.0
    )
padded_german.shape
```

(20000, 14)

## Print Random Texts

```
In [14]: import numpy as np
rand_indices = np.random.choice(len(seq_chunks), size=5, replace=False)

for idx in rand_indices:
    print(f"EN: {english[idx]} \nDE: {german[idx]}\n {padded_german[idx]}")

type(padded_german)

EN: there's no sugar .
DE: <start> es ist kein zucker da . <end>
[ 1 10 6 71 1020 63 3 2 0 0 0 0 0 0]
EN: write to me .
DE: <start> schreiben sie mir ! <end>
[ 1 622 8 21 9 2 0 0 0 0 0 0 0 0]
EN: they're nervous .
DE: <start> sie sind nervoes . <end>
[ 1 8 23 344 3 2 0 0 0 0 0 0 0 0]
EN: tom had a beer .
DE: <start> tom hatte ein bier . <end>
[ 1 5 87 19 303 3 2 0 0 0 0 0 0 0]
EN: what a good shot !
DE: <start> was fuer ein guter schuetze ! <end>
[ 1 38 77 19 515 2864 9 2 0 0 0 0 0 0]

numpy.ndarray
```

## 2. Prepare the data

### 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> (<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 [15]: # 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 [16]: # Test the layer

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

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.

## Create train and validation splits

There are multiple ways to do this, Here the first 80% of the items are taken as train and the rest 20% as validation data.

Ideal way to do is using `sklearn.model_selection.train_test_split`

```
In [17]: # Create train and validation splits
num_train = int(len(padded_german) * 0.8)
german_train = padded_german[:num_train]
english_train = english[:num_train]

german_valid = padded_german[num_train:]
english_valid = english[num_train:]
```

## Load Training and Validation Data

Using `tf.data.Dataset.from_tensor_slices` pack the English and German data as tuples.

```
In [18]: # Create tf.data.Dataset object
def make_tf_dataset(input, target):
    dataset = tf.data.Dataset.from_tensor_slices(
        (input, target)
    )

    return dataset

train_data = make_tf_dataset(english_train, german_train)
valid_data = make_tf_dataset(english_valid, german_valid)
```

```
In [19]: print(train_data.element_spec, english_train[0])
         for input, target in train_data.take(5):
             print(input.numpy(), target.numpy())

(TensorSpec(shape=(), dtype=tf.string, name=None), TensorSpec(shape=(14,), dtype=tf.int32, name=None)) hi .
b'hi .' [ 1 405  9  2  0  0  0  0  0  0  0  0  0  0]
b'hi .' [ 1 3155 663  9  2  0  0  0  0  0  0  0  0  0]
b'run !' [ 1 975  9  2  0  0  0  0  0  0  0  0  0  0]
b'wow !' [ 1 3156  9  2  0  0  0  0  0  0  0  0  0  0]
b'wow !' [ 1 3157  9  2  0  0  0  0  0  0  0  0  0  0]
```

## Split the Sentence

Split the input english sequence using `tf.strings.split`. Please note, default delimiter is space

```
In [20]: # Split the input sentences by a delimiter
         def split(input, target):
             return tf.strings.split(input), target

In [21]: train_data = train_data.map(split)
         valid_data = valid_data.map(split)

         print(train_data.element_spec)
         for input, _ in train_data.take(5):
             print(input.numpy())

(TensorSpec(shape=(None,), dtype=tf.string, name=None), TensorSpec(shape=(14,), dtype=tf.int32, name=None))
[b'hi' b'.']
[b'hi' b'.']
[b'run' b'!']
[b'wow' b'!']
[b'wow' b'!']
```

## Embed the Input

Embed the English dataset using the loaded layer from the transfer learning model

```
In [22]: def embed(input, target):
         embeddings = embedding_layer(input)
         return embeddings, target

         train_data = train_data.map(embed)
         valid_data = valid_data.map(embed)

         print(train_data.element_spec)
         for input, _ in train_data.take(5):
             print(f"Length of Embeddings: {len(input.numpy())}")

(TensorSpec(shape=(None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(14,), dtype=tf.int32, name=None))
Length of Embeddings: 2
Length of Embeddings: 2
Length of Embeddings: 2
Length of Embeddings: 2
Length of Embeddings: 2
```

## Filter Input Data

Filter input data by its maximum length



```
In [23]: def filter_by_length(input, target):
         return tf.shape(input).shape[0] < 13

train_data = train_data.filter(filter_by_length)
valid_data = valid_data.filter(filter_by_length)

print(train_data.element_spec)
for input, _ in valid_data.take(5):
    print(f"Length of Embeddings: {len(input.numpy())}")

(TensorSpec(shape=(None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(14,), dtype=tf.int32, name=None))
Length of Embeddings: 4
Length of Embeddings: 5
Length of Embeddings: 5
Length of Embeddings: 4
Length of Embeddings: 5
```

## Pad Input Data

```
In [24]: def pad_data(input, target):
         pad_max = tf.math.maximum(0, 13 - tf.shape(input)[0])
         paddings = tf.concat((
             [[pad_max, 0]], [[0, 0]]
         ), axis=0)
         return tf.reshape(tf.pad(
             input,
             paddings=paddings,
             mode='CONSTANT',
             constant_values=0
         ), (13, 128)), target

train_data = train_data.map(pad_data)
valid_data = valid_data.map(pad_data)

print(train_data.element_spec)
for input, _ in train_data.take(5):
    print(f"Length of Padded Input: {len(input.numpy())}")

(TensorSpec(shape=(13, 128), dtype=tf.float32, name=None), TensorSpec(shape=(14,), dtype=tf.int32, name=None))
Length of Padded Input: 13
Length of Padded Input: 13
Length of Padded Input: 13
Length of Padded Input: 13
Length of Padded Input: 13
```

## Print Shape of Batched Data

```
In [25]: train_data = train_data.batch(16, drop_remainder=True)
         valid_data = valid_data.batch(16, drop_remainder=True)

In [26]: print(train_data.element_spec)
         print(valid_data.element_spec)

(TensorSpec(shape=(16, 13, 128), dtype=tf.float32, name=None), TensorSpec(shape=(16, 14), dtype=tf.int32, name=None))
(TensorSpec(shape=(16, 13, 128), dtype=tf.float32, name=None), TensorSpec(shape=(16, 14), dtype=tf.int32, name=None))
```

```
In [27]: print(next(iter(train_data.take(1)))[0].shape)
print(valid_data.take(1))

(16, 13, 128)
<TakeDataset shapes: ((16, 13, 128), (16, 14)), types: (tf.float32, tf.int32)>
```

## Print German and English Example Tensors

```
In [28]: english_tensor, german_tensor = next(iter(valid_data.take(1)))

print(f"English Tensor Batch Shape: \n{english_tensor.shape}")
print(f"German Tensor Batch Shape: \n{german_tensor.shape}")
print(f"German Tensor Batch: \n{german_tensor}")

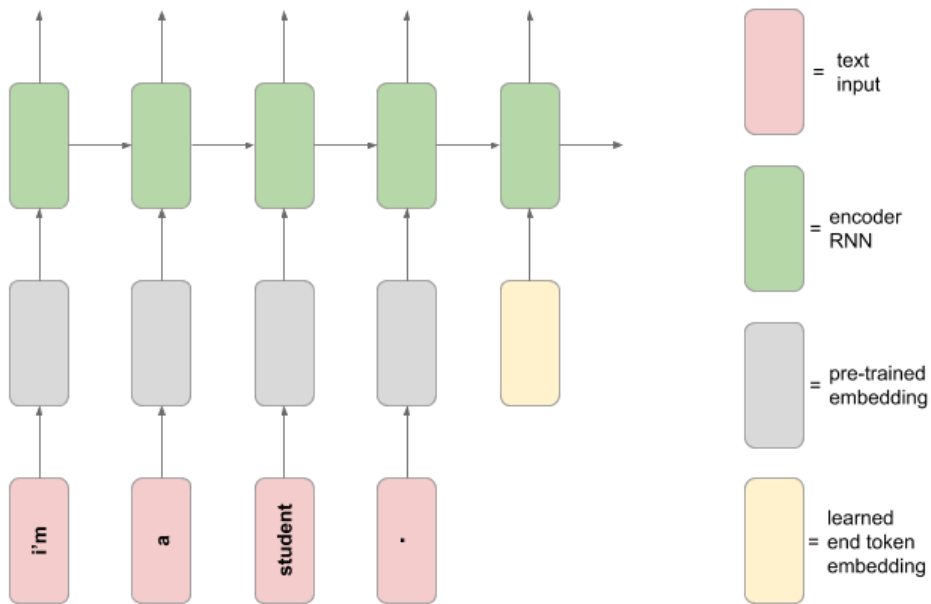
English Tensor Batch Shape:
(16, 13, 128)
German Tensor Batch Shape:
(16, 14)
German Tensor Batch:
[[ 1  11  6 106 538  3  2  0  0  0  0  0  0  0]
 [ 1  11  6 19 5102  3  2  0  0  0  0  0  0  0]
 [ 1  47 427  6 428  3  2  0  0  0  0  0  0  0]
 [ 1  26 1425 2739 5103  3  2  0  0  0  0  0  0  0]
 [ 1  26 1425  6 325  9  2  0  0  0  0  0  0  0]
 [ 1  11 1373  6 360  3  2  0  0  0  0  0  0  0]
 [ 1  26 5104  6 938  3  2  0  0  0  0  0  0  0]
 [ 1  26 591  67  3  2  0  0  0  0  0  0  0  0]
 [ 1  26 591  25  67  3  2  0  0  0  0  0  0  0]
 [ 1  47 1584  6 328  3  2  0  0  0  0  0  0  0]
 [ 1  11 113  6  33  3  2  0  0  0  0  0  0  0]
 [ 1  11 113  6 584  3  2  0  0  0  0  0  0  0]
 [ 1  11 113  6  84  3  2  0  0  0  0  0  0  0]
 [ 1  26 2227  6 938  3  2  0  0  0  0  0  0  0]
 [ 1  26 2229  6 938  3  2  0  0  0  0  0  0  0]
 [ 1  26 2229  6 478  3  2  0  0  0  0  0  0  0]]
```

## 3. Create the custom layer

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

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

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



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).

## End Token Layer

- Create a custom layer
- Initialize the trainable variable from embedding layer
- Option 2 could be using a `tf.random` function to initialize
- Create the layer and apply for an input batch to examine the shape change

```
In [30]: class EndTokenLayer(tf.keras.layers.Layer):

    def __init__(self, **kwargs):
        super(EndTokenLayer, self).__init__(**kwargs)
        self.end_token = tf.Variable(
            embedding_layer(tf.constant(["end"])),
            name="EndToken",
            trainable=True
        )

    def call(self, inputs):
        x = tf.expand_dims(self.end_token, axis=0)
        x = tf.tile(x, (tf.shape(inputs)[0], 1, 1))

        return tf.keras.layers.concatenate([inputs, x], axis=1)

end_token_layer = EndTokenLayer()

In [31]: for an_item in train_data.take(1):
    print(f"Shape of Input: {an_item[0].shape}")
    end_token_output = end_token_layer(an_item[0])
    print(f"Shape after Custom Layer: {end_token_output.shape}")

Shape of Input: (16, 13, 128)
Shape after Custom Layer: (16, 14, 128)
```

```
In [31]:
```

## 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.

### Create Encoder

- Use input layer of shape `sequence length(13)` and `embedding dimension (128)`
- Apply `EndTokenLayer`
- A masking layer of value 0
- Create an LSTM layer and return the states `[output, hidden, cell]`
- Further examine the shape of hidden and cell states

```
In [32]: def create_encoder_model(seq_len=13, embed_dim=128):

    inputs = tf.keras.layers.Input(shape=(seq_len, embed_dim))
    x = EndTokenLayer()(inputs)
    x = tf.keras.layers.Masking(mask_value=0.0)(x)
    output, hidden, cell = tf.keras.layers.LSTM(
        512, return_state=True
    )(x)

    encoder = tf.keras.models.Model(inputs=inputs, outputs=[
        hidden, cell
    ])

    return encoder

encoder = create_encoder_model()
```

```
In [33]: input, target = next(iter(train_data.take(1)))
hidden, cell = encoder(input)
```

```
In [34]: print(f"Shape of Hidden State: {hidden.shape}, \n\t Cell State: {cell.shape}, \n {tf.shap
e(encoder(input))}")

Shape of Hidden State: (16, 512),
      Cell State: (16, 512),
[  2 16 512]
```

```
In [35]: encoder.summary()
```

```
Model: "model"
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 13, 128)]	0
end_token_layer_1 (EndTokenL	(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

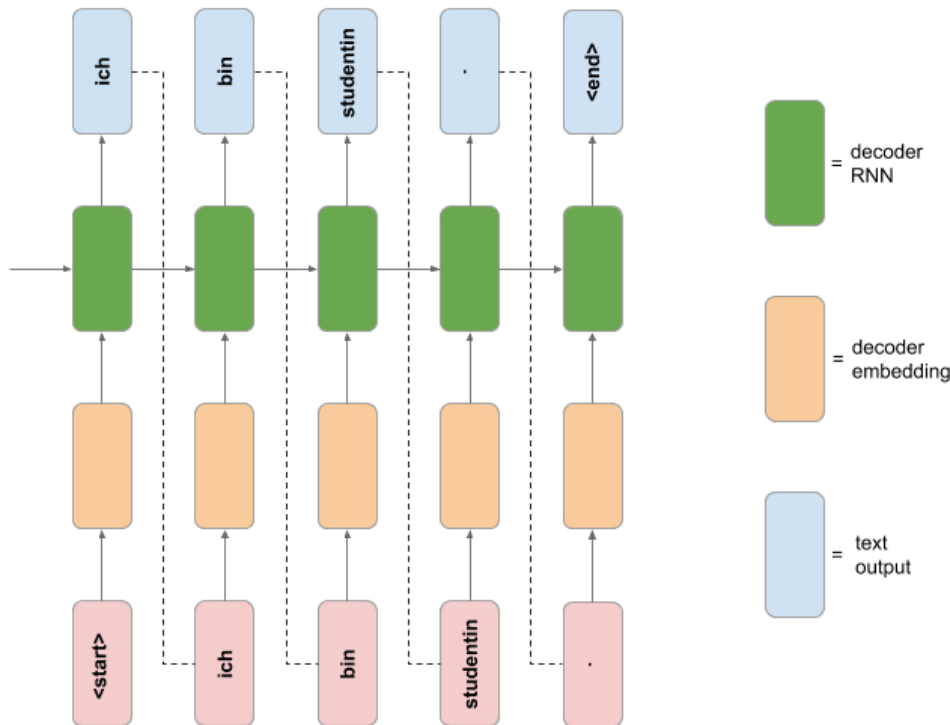
```

## 5. Build the decoder network

The decoder network follows the schematic diagram below.

```
In [36]: # 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")
```



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.

## Decoder Recurrent Network

- Acquired the embedding layer input dimension from word count
- Set embedding dimension as 128
- Create LSTM and Dense layers and return output of the layer along with hidden and cell state

```
In [37]: class DecoderRNN(tf.keras.models.Model):
def __init__(self, input_dim, **kwargs):
    super(DecoderRNN, self).__init__(**kwargs)
    self.embed_layer = tf.keras.layers.Embedding(input_dim, 128, mask_zero=True)
    self.lstm = tf.keras.layers.LSTM(512, return_sequences=True, return_state=True)
    self.dense = tf.keras.layers.Dense(input_dim)

def call(self, inputs, hidden=None, cell=None):
    x = self.embed_layer(inputs)
    x, hidden, cell = self.lstm(x, initial_state=[hidden, cell])
    x = self.dense(x)

    return x, hidden, cell
```

Test the model by calling decoder with data and hidden, cell state acquired from the encoder.

```
In [38]: decoder = DecoderRNN(len(tokenizer.word_index) + 1)
outputs = decoder(
    next(iter(train_data.take(1)))[1],
    hidden=hidden,
    cell=cell
)

In [39]: decoder.summary()
```

Model: "decoder\_rnn"

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		
=====		

## 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.*

### Input and Output for Decoder

```
In [40]: def get_input_and_target(batch):
          return (batch[:, :-1], batch[:, 1:])

          input, target = next(iter(train_data.take(1)))
          input_labels, target_labels = get_input_and_target(target)

          for inp, trg in zip(input_labels[:5], target_labels[:5]):
              print(f"Input: {inp} \n\tTarget: {trg}")

Input: [ 1 405  9  2  0  0  0  0  0  0  0  0  0  0]
      Target: [405  9  2  0  0  0  0  0  0  0  0  0  0]
Input: [ 1 3155 663  9  2  0  0  0  0  0  0  0  0  0]
      Target: [3155 663  9  2  0  0  0  0  0  0  0  0  0]
Input: [ 1 975  9  2  0  0  0  0  0  0  0  0  0  0]
      Target: [975  9  2  0  0  0  0  0  0  0  0  0  0]
Input: [ 1 3156  9  2  0  0  0  0  0  0  0  0  0  0]
      Target: [3156  9  2  0  0  0  0  0  0  0  0  0  0]
Input: [ 1 3157  9  2  0  0  0  0  0  0  0  0  0  0]
      Target: [3157  9  2  0  0  0  0  0  0  0  0  0  0]
```



```
In [41]: @tf.function
def gradient_loss(src_input, trg_input, trg_output):
    with tf.GradientTape() as tape:
        # Get encoder hidden and cell state
        enc_hidden, enc_cell = encoder(src_input)

        # Make predictions and hidden, cell state from decoder
        predictions, dec_hidden, dec_cell = decoder(trg_input, enc_hidden, enc_cell)

        # Calculate the loss
        loss = tf.keras.losses.sparse_categorical_crossentropy(
            trg_output, predictions, from_logits=True
        )
        loss = tf.reduce_mean(loss)

        # Calculate the gradients
        gradients = tape.gradient(
            loss,
            encoder.trainable_variables + decoder.trainable_variables
        )
    return loss, gradients, predictions
```

```

In [42]: epochs = 10
train_loss_results = []
valid_loss_results = []

train_accuracy_results = []
valid_accuracy_results = []
optimizer = tf.keras.optimizers.Adam()

for epoch in range(epochs):
    train_batch_loss_avg = tf.keras.metrics.Mean()
    train_batch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

    for src, trg in train_data:
        decoder_inputs, decoder_labels = get_input_and_target(trg)
        loss, gradients, predictions = gradient_loss(src, decoder_inputs, decoder_labels)

        optimizer.apply_gradients(
            zip(gradients, encoder.trainable_variables + decoder.trainable_variables)
        )

        train_batch_loss_avg(loss)
        train_batch_accuracy(decoder_labels, predictions)

    train_loss_results.append(train_batch_loss_avg.result())
    train_accuracy_results.append(train_batch_accuracy.result())

    valid_batch_loss_avg = tf.keras.metrics.Mean()
    valid_batch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

    for src, trg in valid_data:
        decoder_inputs, decoder_labels = get_input_and_target(trg)
        loss, gradients, predictions = gradient_loss(src, decoder_inputs, decoder_labels)

        # optimizer.apply_gradients(
        #     zip(gradients, encoder.trainable_variables + decoder.trainable_variables)
        # )

        valid_batch_loss_avg(loss)
        valid_batch_accuracy(decoder_labels, predictions)

    valid_loss_results.append(valid_batch_loss_avg.result())
    valid_accuracy_results.append(valid_batch_accuracy.result())

    print(f"Epoch: {epoch:03d}, Train Loss: {train_batch_loss_avg.result()}, Train Acc: {
train_batch_accuracy.result()}")
    print(f"\t\t, Val Loss: {valid_batch_loss_avg.result()}, Val Acc: {valid_batch_accura
cy.result()}")

```

```
Epoch: 000, Train Loss: 5.669101238250732, Train Acc: 0.7208653688430786
, Val Loss: 5.176267623901367, Val Acc: 0.6915384531021118
Epoch: 001, Train Loss: 4.622555732727051, Train Acc: 0.7538365125656128
, Val Loss: 4.398624897003174, Val Acc: 0.7160384654998779
Epoch: 002, Train Loss: 3.747530937194824, Train Acc: 0.7826778888702393
, Val Loss: 3.6389284133911133, Val Acc: 0.7543076872825623
Epoch: 003, Train Loss: 2.9475953578948975, Train Acc: 0.7994086742401123
, Val Loss: 3.027113914489746, Val Acc: 0.7619807720184326
Epoch: 004, Train Loss: 2.2083823680877686, Train Acc: 0.8177307844161987
, Val Loss: 2.481549024581909, Val Acc: 0.7707115411758423
Epoch: 005, Train Loss: 1.576230764389038, Train Acc: 0.8340480923652649
, Val Loss: 2.0743191242218018, Val Acc: 0.7742884755134583
Epoch: 006, Train Loss: 1.1143969297409058, Train Acc: 0.8500240445137024
, Val Loss: 1.794177532196045, Val Acc: 0.783307671546936
Epoch: 007, Train Loss: 0.8123661875724792, Train Acc: 0.8656057715415955
, Val Loss: 1.6472606658935547, Val Acc: 0.7881923317909241
Epoch: 008, Train Loss: 0.6287171840667725, Train Acc: 0.8794903755187988
, Val Loss: 1.5711270570755005, Val Acc: 0.7947307825088501
Epoch: 009, Train Loss: 0.5078410506248474, Train Acc: 0.8922019004821777
, Val Loss: 1.5419209003448486, Val Acc: 0.7969807982444763
```

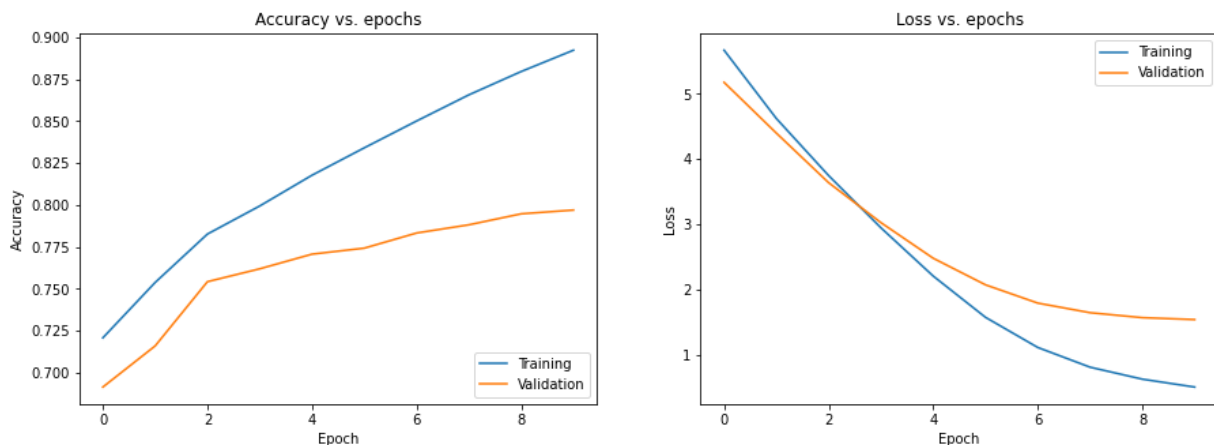
```
In [44]: import matplotlib.pyplot as plt

plt.figure(figsize=(15,5))
plt.subplot(121)

plt.plot(train_accuracy_results)
plt.plot(valid_accuracy_results)

plt.title('Accuracy vs. epochs')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='lower right')

plt.subplot(122)
plt.plot(train_loss_results)
plt.plot(valid_loss_results)
plt.title('Loss vs. epochs')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```



```
In [42]:
```

```
In [42]:
```

## 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 [122]: import numpy as np

def pick_random_sentences(count=5):

    rand_indices = np.random.choice(len(seq_chunks), size=count, replace=False)
    en = []
    de = []
    for idx in rand_indices:
        print(english[idx])
        en.append(english[idx])
        de.append(german[idx])

    return en, de

en, de = pick_random_sentences()

tom was focused .
tom will shoot .
i want to learn .
we're staying .
it's not from me .
```

```
In [123]: def preprocess(tokenizer, de):

    de_chunks = strings_to_sequences(tokenizer, de)
    padded_german = tf.keras.preprocessing.sequence.pad_sequences(
        de_chunks,
        maxlen=14,
        padding="post",
        value=0.0
    )
    test_data = make_tf_dataset(en, padded_german)
    test_data = test_data.map(split)
    test_data = test_data.map(embed)
    test_data = test_data.filter(filter_by_length)
    test_data = test_data.map(pad_data)

    return test_data

test_data = preprocess(tokenizer, de)
test_data.element_spec
```

```
In [112]: data_iter = iter(test_data.take(1))
```

```

In [113]: english_tensor, german_tensor = next(data_iter)

print(f"English Tensor Batch Shape: \n{english_tensor.shape}")
print(f"German Tensor Batch Shape: \n{german_tensor.shape}")
print(f"German Tensor Batch: \n{german_tensor}")

English Tensor Batch Shape:
(13, 128)
German Tensor Batch Shape:
(14,)
German Tensor Batch:
[ 1  4 15 71 708  3  2  0  0  0  0  0  0]

In [138]: hidden, cell = encoder(tf.expand_dims(english_tensor, 0))

In [139]: def translate(hidden, cell):
    translated_tokens = []
    translated_words = []
    tf_start = tf.Variable([[tokenizer.word_index['<start>']]])
    tf_end = tf.Variable([[tokenizer.word_index['<end>']]])
    while True:
        predictions, hidden, cell = decoder(tf_start, hidden, cell)
        output = tf.argmax(predictions, 2).numpy()[0, 0]
        tf_start = tf.Variable([[output]])

        if(output == tf_end):
            break
        else:
            translated_tokens.append(output)
            translated_words.append(tokenizer.index_word[output])

    return translated_tokens, translated_words

In [140]: print(f"{en[0]}, {translate(hidden, cell)}")

tom was focused ., ([10, 6, 12, 151, 21, 3], ['es', 'ist', 'nicht', 'von', 'mir', '.'])

In [142]: data_iter = iter(test_data.take(5))

for i in np.arange(5):
    english_tensor, _ = next(data_iter)
    hidden, cell = encoder(tf.expand_dims(english_tensor, 0))
    _, de_sentence = translate(hidden, cell, _)
    print(f'{en[i]} ==> {de[i][8:-6]}')
    print(f'Translated: {" ".join(de_sentence)}\n')

tom was focused . ==> tom war konzentriert .
Translated: tom war gewalttaetig .

tom will shoot . ==> tom wird schiessen .
Translated: tom wird fliehen .

i want to learn . ==> ich will lernen .
Translated: ich will tatsachen .

we're staying . ==> wir bleiben .
Translated: wir sind optimisten .

it's not from me . ==> das ist nicht von mir .
Translated: es ist nicht von mir .

```

```
In [ ]:
```

