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.

```
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.

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

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

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

```
# Run this cell to load the dataset

NUM_EXAMPLES = 30000
data_examples = []
with open('/content/gdrive/MyDrive/Keras TF book/deu.txt', 'r', encoding='utf8') as f:
    for line in f.readlines():
        if len(data_examples) < NUM_EXAMPLES:
            data_examples.append(line)
        else:
            break
```

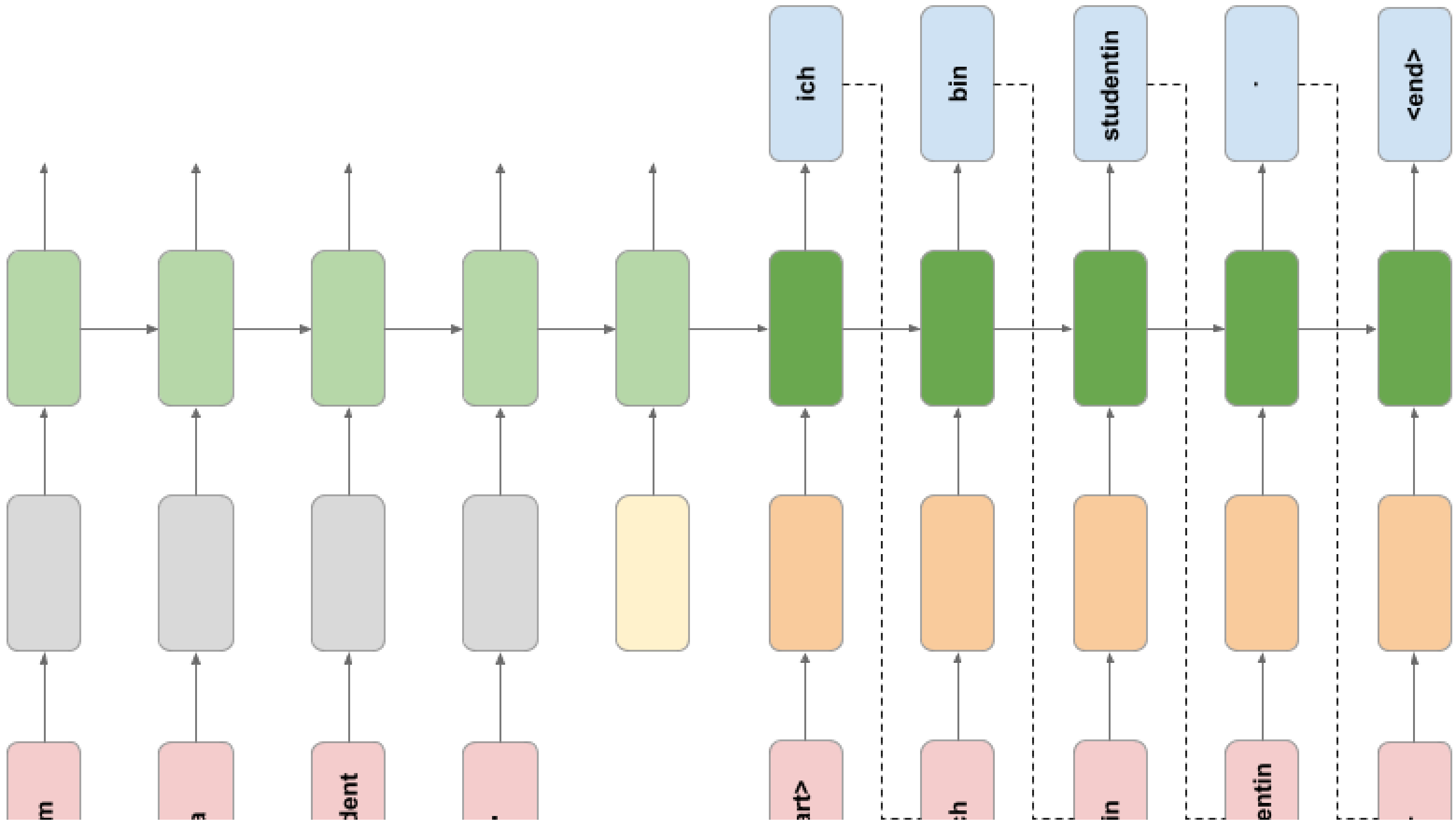
```
# These functions preprocess English and German sentences
```

```
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.

```
# 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-
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.

```
def split_and_preprocess_sentences(input_data, first_n_words=NUM_EXAMPLES):
  eng_data = []
  german_data = []

  start_token = "<start>"
  end_token = "<end>"

  for sentence in input_data[:first_n_words]:
    eng_word, german_word, _ = sentence.split('\t')
    filtered_german_sentence = start_token + ' ' + preprocess_sentence(german_word) + ' ' + end_token

    eng_data.append(preprocess_sentence(eng_word))
    german_data.append(filtered_german_sentence)

  return eng_data, german_data


def tokenize_sentences(sentences):
  tokenizer = tf.keras.preprocessing.text.Tokenizer(
    num_words=None,
    filters='!"#$%&()*+,-./:;=?@[\\]^_`{|}~\t\n',
    lower=True,
    split=' ',
    char_level=False,
    oov_token=None,
```

```
  )
  tokenizer.fit_on_texts(sentences)
  return tokenizer, tokenizer.texts_to_sequences(sentences)
```

```
# Get preprocessed sentences from dataset
preprocessed_eng_sentence, preprocessed_german_sentence = split_and_preprocess_sentences(data_examples)

# Get tokenizer and tokens for German sentences
tokenizer, german_tokens = tokenize_sentences(preprocessed_german_sentence)
```

```
import numpy as np

def get_random_sentence_pairs(eng_sentence, german_sentence, tokens=german_tokens, num_pairs=5):
  random_indexes = np.random.randint(0, NUM_EXAMPLES, num_pairs)
  for ind in random_indexes:
    print(
        f'Eng sentence: {eng_sentence[ind]}, German sentence: {german_sentence[ind]}\n'\
        f'German tokens: {tokens[ind]}\n'
    )
```

```
words_indexes = tokenizer.word_index
inv_word_indexes = {value: key for key, value in words_indexes.items()}

random_ind = np.random.randint(0, NUM_EXAMPLES, 3)

for ind in random_ind:
  words_list = [inv_word_indexes[token] for token in german_tokens[ind]]
  print(' '.join(words_list))
```

```
<start> werden sie uns helfen <end>
<start> ich will helfen <end>
<start> ich liebe videospiele <end>
```

```
get_random_sentence_pairs(preprocessed_eng_sentence,
                          preprocessed_german_sentence)
```

```
Eng sentence: what a hot day !, German sentence: <start> was fuer ein heisser tag ! <end>
German tokens: [1, 24, 63, 15, 1845, 206, 2]

Eng sentence: i read the book ., German sentence: <start> ich lese das buch . <end>
German tokens: [1, 4, 547, 7, 94, 2]

Eng sentence: tom is focused ., German sentence: <start> tom ist konzentriert . <end>
German tokens: [1, 3, 5, 2225, 2]

Eng sentence: it'll be all right ., German sentence: <start> es wird gutgehen . <end>
German tokens: [1, 8, 44, 2626, 2]

Eng sentence: she was brave ., German sentence: <start> sie war tapfer . <end>
German tokens: [1, 6, 20, 917, 2]
```

```
padded_german_tokens = tf.keras.preprocessing.sequence.pad_sequences(
    german_tokens,
    padding='post',
)
```

```
PADDED_lenght = padded_german_tokens.shape[1]
print(padded_german_tokens.shape)
```

```
(30000, 12)
```

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.

```
# 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)
```

```
# 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.

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
    preprocessed_eng_sentence,
    padded_german_tokens,
    train_size=0.8
)
```

```
train_ds = tf.data.Dataset.from_tensor_slices((x_train, y_train))
val_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test))
```

```
# Split both dataset each sentence with spaces

@tf.function
def split_eng_sentence(x, y):
  return tf.strings.split(x), y

train_ds = train_ds.map(split_eng_sentence)
val_ds = val_ds.map(split_eng_sentence)
```

```
# Creating embedding function for English sentences
```

```
@tf.function
def embed_eng_sentence(x, y):
  return embedding_layer(x), y

train_ds = train_ds.map(embed_eng_sentence)
val_ds = val_ds.map(embed_eng_sentence)
```

```
# Filtering out embeddings with more than or equal to 13 tokens

@tf.function
def filter_tokens(x, y):
  return tf.shape(x)[0] <= 13

train_ds = train_ds.filter(filter_tokens)
val_ds = val_ds.filter(filter_tokens)
```

```
# Padding English embeddings

@tf.function
def padding_eng_embd(x, y):
  lenght_embd = tf.shape(x)[0]
  pre_padding_size = tf.math.maximum(0, 13-lenght_embd)

  padded_sequnce = tf.pad(
      x,
      [[pre_padding_size, 0], [0, 0]],
      constant_values=0.0
  )

  return padded_sequnce, y

train_ds = train_ds.map(padding_eng_embd)
val_ds = val_ds.map(padding_eng_embd)
```

```
# Batch both datasets

```

```
train_ds = train_ds.batch(16)
val_ds = val_ds.batch(16)
```

```
# Print the element_spec property

print(train_ds.element_spec)
print(val_ds.element_spec)
```

```
(TensorSpec(shape=(None, None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(None, 12), dtype=tf.int32, name=N
(TensorSpec(shape=(None, None, 128), dtype=tf.float32, name=None), TensorSpec(shape=(None, 12), dtype=tf.int32, name=N
```

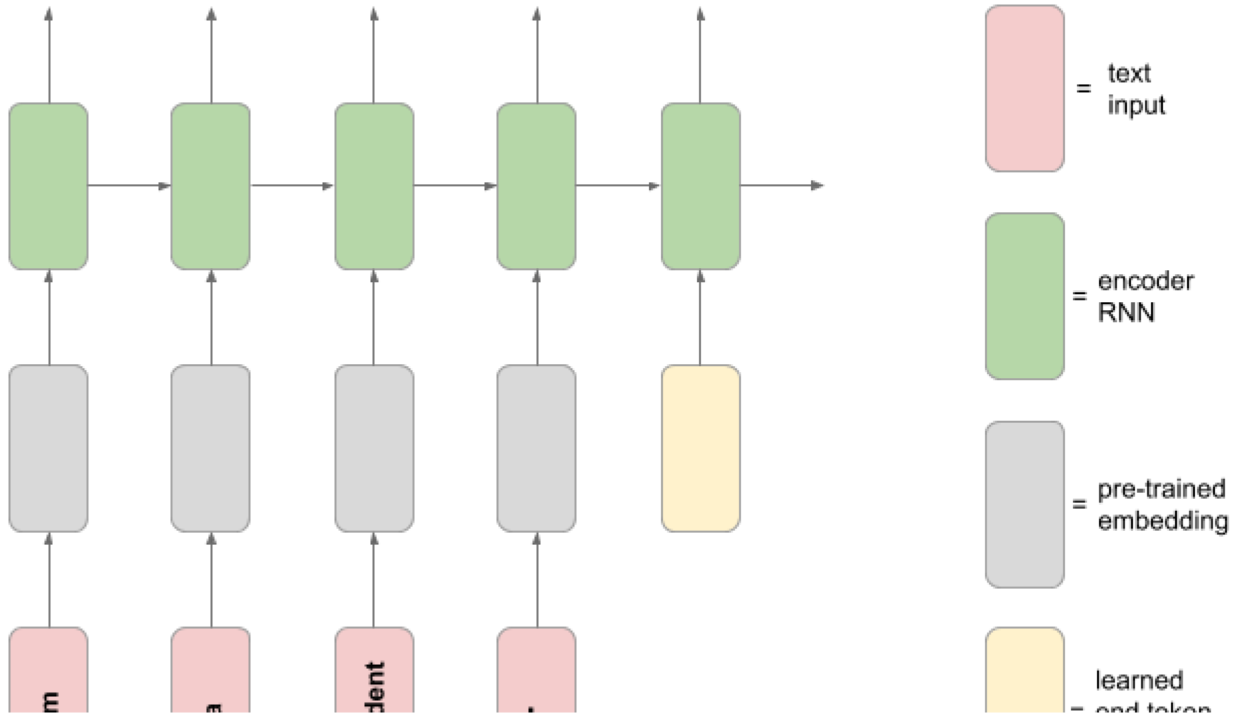
```
# Printing shape of single batch in datasets

for i in train_ds.take(1):
  print(i[0].shape)
  print(i[1].shape)
```

```
(16, 13, 128)
(16, 12)
```

3. Create the custom layer

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

```
# 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=1JrtNOzUJDaOw
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).

```
class CustomLayer(tf.keras.layers.Layer):
  def __init__(self, embedding_dim, **kwargs):
    super(CustomLayer, self).__init__(**kwargs)
    self.embed = tf.Variable(
```

```
        initial_value=tf.zeros(shape=(1,embedding_dim)),
        trainable=True,
        dtype='float32'
        )

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

```
custom_layer = CustomLayer(128)

test_data = next(train_ds.take(1).as_numpy_iterator())
print(f'Input data shape is: {test_data[0].shape}')

x = custom_layer(test_data[0])
print(f'Processed data from custom_layer: {x.shape}')
```

```
Input data shape is: (16, 13, 128)
Processed data from custom_layer: (16, 14, 128)
```

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.

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

```
def EncoderModel(name, batch_size=16):
  EMBEDDING_DIM = 128

  inputs = Input(shape=(13, EMBEDDING_DIM), batch_size=batch_size, dtype=tf.float32)
  x = CustomLayer(EMBEDDING_DIM)(inputs)
  x = Masking(mask_value=0.0)(x)
  x = LSTM(512, return_state=True, return_sequences=True)(x)

  outputs = (x[1], x[2])
  encoder_model = Model(inputs=inputs, outputs=[x[1], x[2]], name=name)

  return encoder_model

encoder_model = EncoderModel(name='encoder_model')
```

```
test_data = next(iter(train_ds.take(1)))[0]
print(f'Input data shape is: {test_data.shape}')
prediction = encoder_model(test_data)
print(f'Resulting tensor_1 shape of the model: {prediction[0].shape}\n'\
      f'Resulting tensor_2 shape of the model: {prediction[1].shape}')
```

```
Input data shape is: (16, 13, 128)
Resulting tensor_1 shape of the model: (16, 512)
Resulting tensor_2 shape of the model: (16, 512)
```

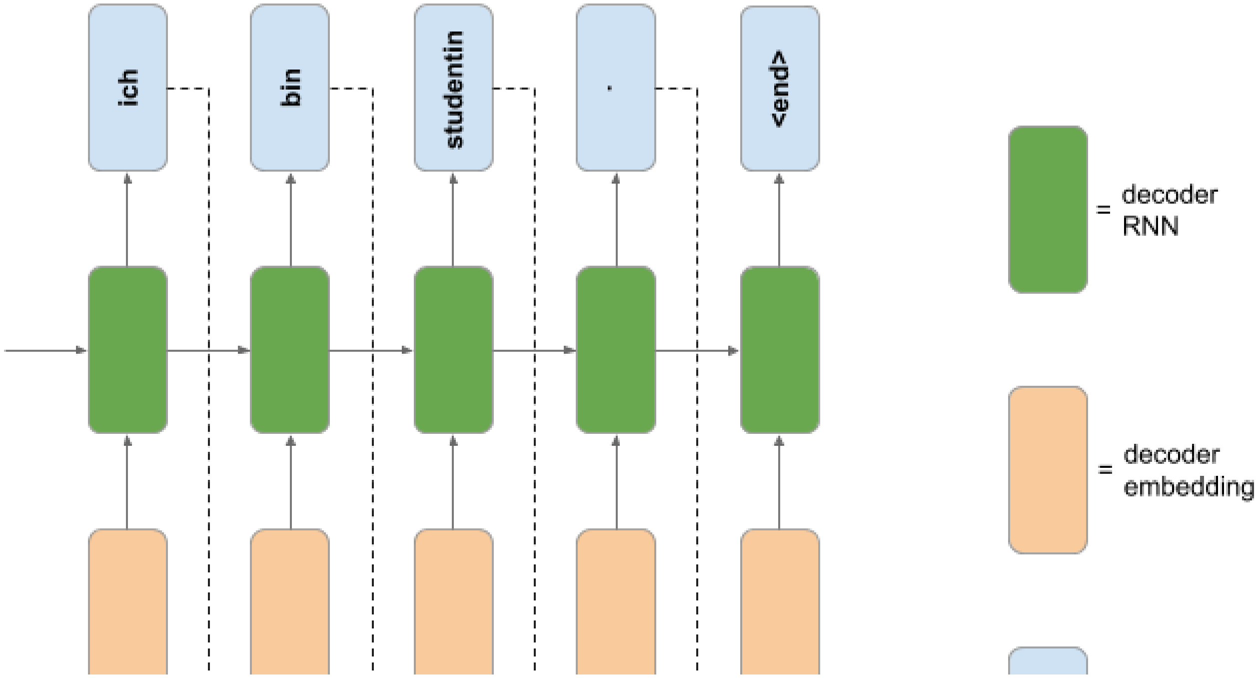
```
encoder_model.summary()
```

```
Model: "encoder_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(16, 13, 128)]           0         
                                                                 
 custom_layer_1 (CustomLayer  (16, 14, 128)            128       
 )                                                               
                                                                 
 masking (Masking)           (16, 14, 128)             0         
                                                                 
 lstm (LSTM)                 [(16, 14, 512),           1312768   
                              (16, 512),                         
                              (16, 512)]                         
                                                                 
=================================================================
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.

```
# 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=1DTeaXD8tA8Rj
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.

```
from tensorflow.keras.layers import Embedding, Dense

UNIQUE_GERMAN_TOKENS = max(words_indexes.values())

class RNNDecoder(Model):
  def __init__(self, embedding_dim=UNIQUE_GERMAN_TOKENS, **kwargs):
    super(RNNDecoder, self).__init__(**kwargs)

    self.embedding_1 = Embedding(input_dim=embedding_dim+1, output_dim=128, mask_zero=True, name='embedding_layer_1')
    self.lstm_1 = LSTM(units=512, return_sequences=True, return_state=True, name='lstm_layer_1')
    self.dense_1 = Dense(embedding_dim+1, name='dense_layer_1')

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

    x = self.embedding_1(inputs)
    x, final_memory_state, final_carry_state = self.lstm_1(x, initial_state=[hidden_state, cell_state])
    x = self.dense_1(x)
    outputs = (x, final_memory_state, final_carry_state)
    return outputs
```

```
decoder_model = RNNDecoder(name='decoder_model')
encoder_model = EncoderModel(name='encoder_model')
```

```
eng_data, german_data = next(iter(train_ds.take(3)))

hidden_state, cell_state = encoder_model(eng_data)
x, final_memory_state, final_carry_state = decoder_model(german_data, hidden_state, cell_state)
```

```
print(f'Resulting decoder outputs of pred: {x.shape}, hid_st:' \
      f'{final_memory_state.shape} and cell_state: {final_carry_state.shape}')
```

```
Resulting decoder outputs of pred: (16, 12, 7468), hid_st:(16, 512) and cell_state: (16, 512)
```

```
decoder_model.summary()
```

```
Model: "decoder_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 embedding_layer_1 (Embeddin  multiple                 955904    
 g)                                                              
                                                                 
 lstm_layer_1 (LSTM)         multiple                  1312768   
                                                                 
 dense_layer_1 (Dense)       multiple                  3831084   
                                                                 
=================================================================
Total params: 6,099,756
Trainable params: 6,099,756
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.

```
# Define a function that takes a Tensor batch of German data and returns a tuple containing German inputs and outputs for th

@tf.function
def preprocess_batch_of_german_data(batch):
  inputs_batch = batch[:, :-1]
  outputs_batch = batch[:, 1:]
  return inputs_batch, outputs_batch
```

```
@tf.function
def get_loss_and_grads_values(eng_input,
                              german_input, german_output,
                              encoder, decoder,
                              loss_obj, compute_grads=True):

  with tf.GradientTape() as tape:
    hidden_state, cell_state = encoder(eng_input)
    y_pred, hidden_state, c_s = decoder(german_input, hidden_state, cell_state)
    loss_val = loss_obj(y_pred=y_pred, y_true=german_output)
```

```
  grads = None

  if compute_grads:
    grads = tape.gradient(loss_val, encoder.trainable_variables + decoder.trainable_variables)

  return loss_val, grads
```

```
def train(encoder,
          decoder,
          loss_obj,
          optimizer,
          train_dataset=train_ds,
          validation_dataset=val_ds,
          epochs=5):


  train_loss = []
  validation_loss = []

  for epoch in range(epochs):

    epoch_loss_avg_train = tf.keras.metrics.Mean()
    epoch_loss_avg_val = tf.keras.metrics.Mean()

    number_of_batch = 0

    tf.print(f'\nEpoch: {epoch+1}\n')


    for eng_data, german_data in train_dataset:

      german_input_train, german_output_train = preprocess_batch_of_german_data(german_data)
      loss_value, gradients = get_loss_and_grads_values(
          eng_input=eng_data,
          german_input=german_input_train,
          german_output=german_output_train,
          encoder=encoder,
```

```
            decoder=decoder,
            loss_obj=loss_obj
        )

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

        number_of_batch += 1

        if number_of_batch % 200 == 0:
          tf.print(f'Number of batch: {number_of_batch}, Loss value: {loss_value.numpy()}')

      train_dataset = train_dataset.shuffle(np.random.randint(100, 1000, 1)[0])

      for eng_data_val, german_data_val in validation_dataset:

        german_input_val, german_output_val = preprocess_batch_of_german_data(german_data_val)

        loss_value_val, grad_val = get_loss_and_grads_values(
            eng_input=eng_data_val,
            german_input=german_input_val,
            german_output=german_output_val,
            encoder=encoder,
            decoder=decoder,
            loss_obj=loss_obj,
            compute_grads=False
        )

        epoch_loss_avg_val.update_state(loss_value_val)

      train_loss.append(epoch_loss_avg_train.result())
      validation_loss.append(epoch_loss_avg_val.result())

      tf.print(f'\nTrain_loss: {epoch_loss_avg_train.result()}, Val_loss: {epoch_loss_avg_val.result()}')

    return train_loss, validation_loss
```

```python
TRAIN_MODEL = True

if TRAIN_MODEL:
  lr = 1e-3

  model_optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
  model_loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

  encoder_model = EncoderModel(name='trained_encoder_model')
  decoder_model = RNNDecoder(name='trained_decoder_model')

  history = train(
      encoder=encoder_model,
      decoder=decoder_model,
      loss_obj=model_loss_obj,
      optimizer=model_optimizer,
      epochs=9
  )
```

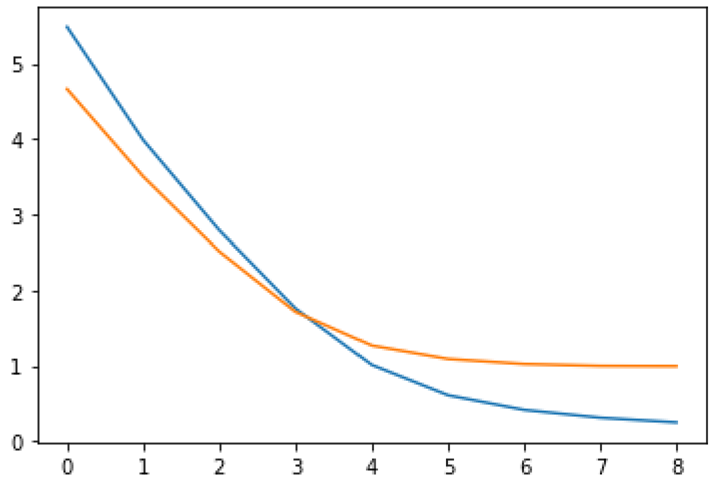
```
Train_loss: 3.986802339553833, Val_loss: 3.504516363143921
Epoch: 2

Number of batch: 200, Loss value: 3.112260341644287
Number of batch: 400, Loss value: 2.843500852584839
Number of batch: 600, Loss value: 2.8117129802703857
Number of batch: 800, Loss value: 2.6977434158325195
Number of batch: 1000, Loss value: 2.8098785877227783
Number of batch: 1200, Loss value: 2.468799114227295
Number of batch: 1400, Loss value: 2.5018675327301025

Train_loss: 2.792592763900757, Val_loss: 2.5117287635803223
Epoch: 3

Number of batch: 200, Loss value: 2.059675455093384
Number of batch: 400, Loss value: 1.9827632904052734
Number of batch: 600, Loss value: 1.8329294919967651
Number of batch: 800, Loss value: 1.6215317249298096
Number of batch: 1000, Loss value: 1.6256183385849
Number of batch: 1200, Loss value: 1.4236302375793457
```

```
Number of batch: 1400, Loss value: 1.3017754554748535

Train_loss: 1.7588766813278198, Val_loss: 1.7135850191116333
Epoch: 4

Number of batch: 200, Loss value: 1.1773056983947754
Number of batch: 400, Loss value: 1.1316502094268799
Number of batch: 600, Loss value: 1.166587233543396
Number of batch: 800, Loss value: 0.8583892583847046
Number of batch: 1000, Loss value: 0.9297859072685242
Number of batch: 1200, Loss value: 0.8244050741195679
Number of batch: 1400, Loss value: 0.8632346987724304

Train_loss: 1.0096814632415771, Val_loss: 1.2667741775512695
Epoch: 5

Number of batch: 200, Loss value: 0.6862034797668457
Number of batch: 400, Loss value: 0.7618101239204407
Number of batch: 600, Loss value: 0.5839195847511292
Number of batch: 800, Loss value: 0.7792690396308899
Number of batch: 1000, Loss value: 0.6771319508552551
Number of batch: 1200, Loss value: 0.5893025994300842
Number of batch: 1400, Loss value: 0.44237184524536133

Train_loss: 0.6083022356033325, Val_loss: 1.089266061782837
Epoch: 6

Number of batch: 200, Loss value: 0.3608807921409607
Number of batch: 400, Loss value: 0.3894067108631134
Number of batch: 600, Loss value: 0.36211663484573364
Number of batch: 800, Loss value: 0.4564225971698761
Number of batch: 1000, Loss value: 0.48281317949295044
Number of batch: 1200, Loss value: 0.410505473613739
Number of batch: 1400, Loss value: 0.45152023434638977

Train_loss: 0.4136466383934021, Val_loss: 1.0218168497085571
Epoch: 7
```

```
import matplotlib.pyplot as plt

train_loss, val_loss = history
```

```
num_epochs = len(train_loss)

plt.plot([i for i in range(num_epochs)], train_loss)
plt.plot([i for i in range(num_epochs)], val_loss)
```

```
[<matplotlib.lines.Line2D at 0x7ff7f1442f90>]
```




```
if TRAIN_MODEL:
  encoder_model.save('/content/gdrive/MyDrive/models_trained/capstone_project_course2/encoder')
  decoder_model.save('/content/gdrive/MyDrive/models_trained/capstone_project_course2/decoder')
else:
  encoder_model = tf.keras.models.load_model('/content/gdrive/MyDrive/models_trained/capstone_project_course2/encoder')
  decoder_model = tf.keras.models.load_model('/content/gdrive/MyDrive/models_trained/capstone_project_course2/decoder')
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` w
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` w
WARNING:absl:Found untraced functions such as lstm_cell_3_layer_call_fn, lstm_cell_3_layer_call_and_return_conditional
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/models_trained/capstone_project_course2/encoder/assets
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/models_trained/capstone_project_course2/encoder/assets
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7ff7e15a5690> has the same name 'LSTMCell' as a built-in Ker
WARNING:absl:Found untraced functions such as lstm_cell_4_layer_call_fn, lstm_cell_4_layer_call_and_return_conditional
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/models_trained/capstone_project_course2/decoder/assets
INFO:tensorflow:Assets written to: /content/gdrive/MyDrive/models_trained/capstone_project_course2/decoder/assets
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7ff7e14ac710> has the same name 'LSTMCell' as a built-in Ker
```

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.

```
num_of_sentences = 5
random_indices = np.random.randint(0, NUM_EXAMPLES, num_of_sentences)

random_data_examples = np.array(data_examples)[random_indices]
eng_data, german_data = split_and_preprocess_sentences(random_data_examples,
                                                       first_n_words=num_of_sentences)
```

```
padded_german_data_test = tf.keras.preprocessing.sequence.pad_sequences(
    tokenizer.texts_to_sequences(german_data),
    maxlen=PADDED_lenght,
    padding='post'
)
```

```
print(eng_data)
print(padded_german_data_test.shape)
```

```
["here's your lunch .", "i'm tom's mom .", "he's coming closer .", 'have you ever lost ?', "they're unreliable ."]
```

```
(5, 12)
```

```
test_ds = tf.data.Dataset.from_tensor_slices((eng_data, padded_german_data_test))
```

```
test_ds = test_ds.map(split_eng_sentence)
test_ds = test_ds.map(embed_eng_sentence)
test_ds = test_ds.filter(lambda x, y: tf.shape(x)[0] <= 13)
test_ds = test_ds.map(padding_eng_embd)
```

```
test_ds = test_ds.batch(1)
```

```
start_token = np.array(tokenizer.texts_to_sequences(['<start>']))
end_token = np.array(tokenizer.texts_to_sequences(['<end>']))

for ind, e in enumerate(test_ds):
  hidden_state, cell_state = encoder_model(e[0])

  german_tokens_pred = []
  german_input = start_token

  german_out, hidden_state, cell_state = decoder_model(german_input,
                                                        hidden_state,
                                                        cell_state)

  german_out = tf.argmax(german_out, axis=2)

  german_tokens_pred.append(
      tokenizer.index_word.get(tf.squeeze(german_out).numpy())
  )
  german_input = german_out

  while german_out != 2:

    german_out, hidden_state, cell_state = decoder_model(german_input,
                                                          hidden_state,
                                                          cell_state)
```

```
    german_out = tf.argmax(german_out, axis=2)

    german_input = german_out

    german_tokens_pred.append(tokenizer.index_word.get(tf.squeeze(german_out).numpy(), 'UNK'))
  print(f'English Text: {eng_data[ind]}')
  print(f'German Translation: {" ".join(german_tokens_pred[:-1])}')
  print(f'German true output: {german_data[ind]}')
  print()
```

```
English Text: here's your lunch .
German Translation: hier ist ihr mittagessen
German true output: <start> hier ist dein mittagessen . <end>

English Text: i'm tom's mom .
German Translation: ich bin toms gast
German true output: <start> ich bin toms mutter . <end>

English Text: he's coming closer .
German Translation: er kommt naeher
German true output: <start> er kommt naeher . <end>

English Text: have you ever lost ?
German Translation: hast du schon einmal verloren
German true output: <start> hast du jemals einen verlust erlitten ? <end>

English Text: they're unreliable .
German Translation: sie sind unzuverlaessig
German true output: <start> sie sind unzuverlaessig . <end>
```

0 с завершено о 17:15