Neural translation model

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.

```
import tensorflow as tf
import tensorflow_hub as hub
import unicodedata
import re
import numpy as np
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

https://drive.google.com/open?id=1KczOciG7sYY7SB9UlBeRP1T9659b121Q

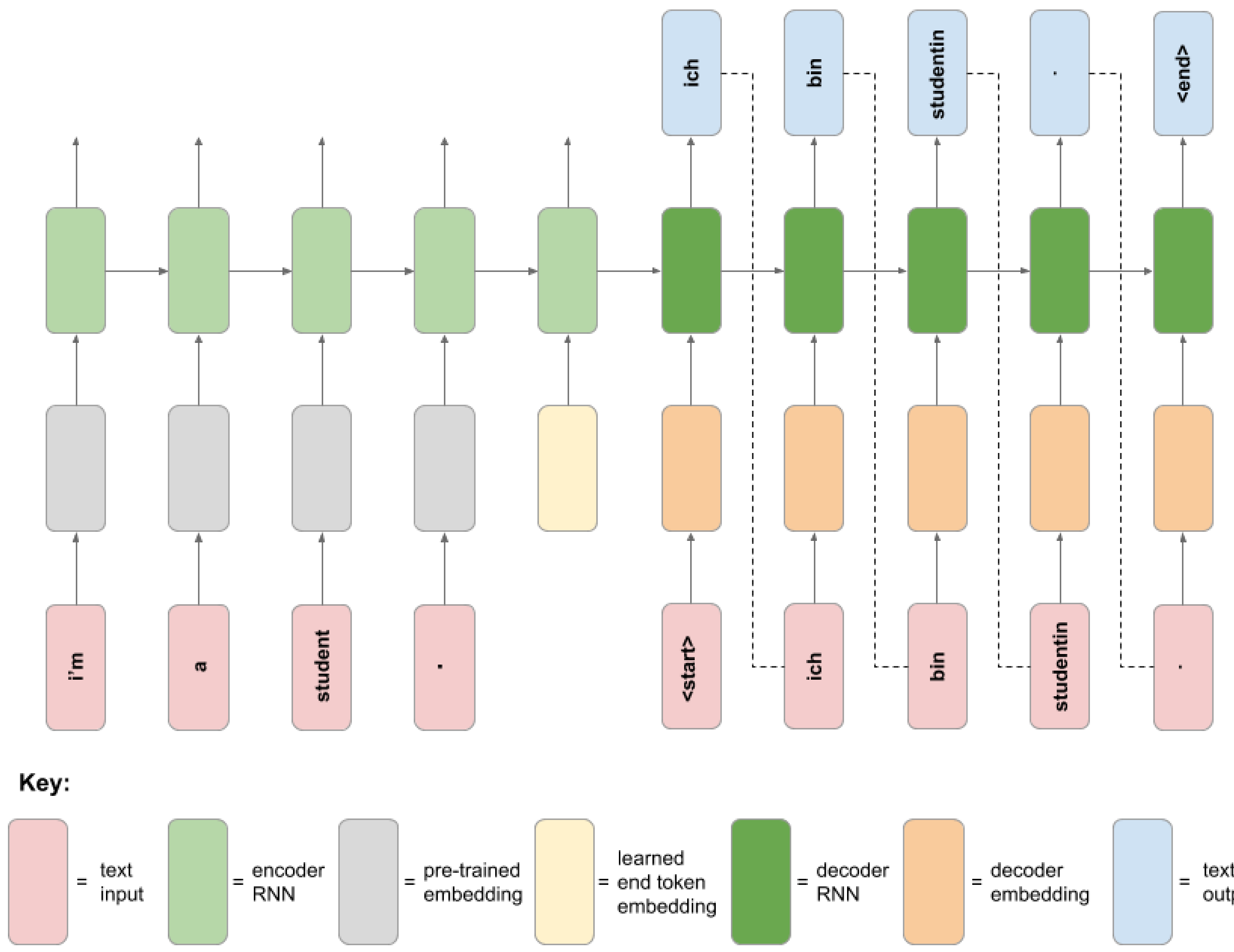
```
# Run this cell to load the dataset
from google.colab import drive
drive.mount('/content/gdrive')

NUM_EXAMPLES = 20000
data_examples = []
with open('/content/gdrive/MyDrive/Colab Notebooks/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()
```

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

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-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.

```
ENGLISH = []
GERMAN = []
for j in range(0,len(data_examples)):
    ans = data_examples[j].split('\t')
    ENGLISH = ENGLISH + [preprocess_sentence(ans[0])]
    GERMAN = GERMAN + ["<start> "+ preprocess_sentence(ans[1]) + " <end>"]
```

```
from tensorflow.keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer(num_words=None,filters = '',lower=False,char_level=False)

tokenizer.fit_on_texts(GERMAN)
sequences = tokenizer.texts_to_sequences(GERMAN)

print(len(tokenizer.word_index) + 1)
```

```
5744
```

```
import random
for i in range(5):
    i = random.randint(0,20000)
    print((ENGLISH[i],GERMAN[i]))
    print(sequences[i])

sequences = tf.keras.preprocessing.sequence.pad_sequences(
                        sequences, maxlen = None, dtype='int32', padding='post',
                        truncating='post', value=0.0
                        )
sequences.shape
```

```
('the book is red .', '<start> das buch ist rot . <end>')
[1, 11, 113, 6, 352, 3, 2]
('it works .', '<start> es funktioniert . <end>')
[1, 10, 298, 3, 2]
('the wind howled .', '<start> der wind heulte . <end>')
[1, 47, 4455, 2771, 3, 2]
('no one was stung .', '<start> niemand wurde getroffen . <end>')
[1, 99, 68, 346, 3, 2]
('tom was dizzy .', '<start> tom war schwindelig . <end>')
[1, 5, 24, 1900, 3, 2]
(20000, 14)
```

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(ENGLISH,  sequences,  test_size=0.2,  random_state=1)


datasetTrain  =  tf.data.Dataset.from_tensor_slices((X_train,  y_train))
datasetTest  =  tf.data.Dataset.from_tensor_slices((X_test,  y_test))

def  split(data):
    return  data.map(lambda  x,y:  (tf.strings.split(x,sep  =  '  '),y))
datasetTrain  =  split(datasetTrain)
datasetTest  =  split(datasetTest)

def  embed(data):
    return  data.map(lambda  x,y:  (embedding_layer(x),y))
datasetTrain  =  embed(datasetTrain)
datasetTest  =  embed(datasetTest)

def  filt(data):
    return  data.filter(lambda  x,y:  tf.shape(x)[0]  <=  13)
datasetTrain  =  filt(datasetTrain)
datasetTest  =  filt(datasetTest)

def  pad(x,y):
    while  (tf.shape(x)[0]  !=  13):
        paddings  =  tf.concat(([[1,0]],  [[0,0]]),  axis=0)
        x  =  tf.pad(x,  paddings)
    tf.ensure_shape(x,  [13,  128])
    #  x  =  tf.pad(x,  [tf.math.maximum([13-tf.shape(x)[0]  ,0],  tf.constant([0,0])),  tf.constant([0,0])],  "CONSTANT",  constant_values=0)
    return  x,y
datasetTrain  =  datasetTrain.map(pad)
datasetTest  =  datasetTest.map(pad)


datasetTrain  =  datasetTrain.batch(16,drop_remainder=  True)
datasetTest  =  datasetTest.batch(16,drop_remainder=  True)


print(datasetTrain.element_spec)
print(datasetTest.element_spec)
```

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

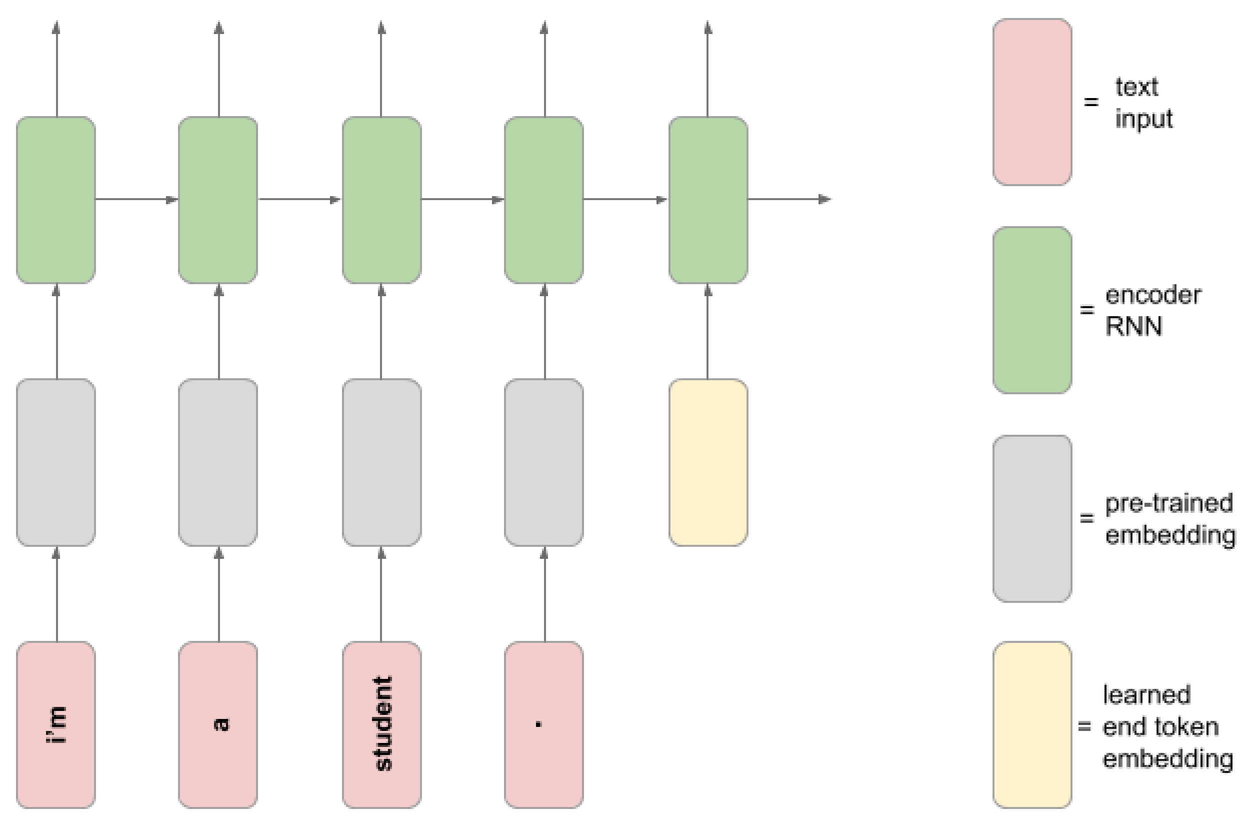
```
for  eng,ger  in  datasetTrain.take(1):
        print(tf.shape(eng))
        print(ger)
```

```
tf.Tensor([ 16  13 128], shape=(3,), dtype=int32)
tf.Tensor(
[[   1   14    6 3816    3    2    0    0    0    0    0    0    0    0]
 [   1   11   24   12  479    3    2    0    0    0    0    0    0    0]
 [   1    5   30   12  227    3    2    0    0    0    0    0    0    0]
 [   1   14    6 3819  363    3    2    0    0    0    0    0    0    0]
 [   1    5   16 1280 5284    3    2    0    0    0    0    0    0    0]
 [   1   63   58   13  330 1109    3    2    0    0    0    0    0    0]
 [   1  233    8  126    9    2    0    0    0    0    0    0    0    0]
 [   1    4   18   11 2842  142    3    2    0    0    0    0    0    0]
 [   1   32   13   12   20  219    7    2    0    0    0    0    0    0]
 [   1    4   15 1146    3    2    0    0    0    0    0    0    0    0]
 [   1    4  133 1702 1050    3    2    0    0    0    0    0    0    0]
 [   1    4   15  552    3    2    0    0    0    0    0    0    0    0]
 [   1   17   35    8  116    3    2    0    0    0    0    0    0    0]
 [   1   43  496   33    7    2    0    0    0    0    0    0    0    0]
 [   1    5    6 2420    3    2    0    0    0    0    0    0    0    0]
 [   1    4 3314   22    3    2    0    0    0    0    0    0    0    0]], shape=(16, 14), dtype=int32)
```

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=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).

```
class  myLayer(tf.keras.layers.Layer):

    def  __init__(self,embedding_dim=128,  **kwargs):
```

```
        super(myLayer, self).__init__(**kwargs)
    def build(self, input_shape):
        self.end_token_embedding = self.add_weight(shape=(input_shape[-1],),initializer='random_uniform',trainable= True)
    def call(self, inputs):
        end_token = tf.tile(tf.reshape(self.end_token_embedding, shape=(1, 1, self.end_token_embedding.shape[0])), [tf.shape(inputs)[0],1,1])
        return tf.keras.layers.concatenate([inputs, end_token], axis=1)
```

```
testLayer = myLayer()
for english,german in datasetTrain.take(1):
    endEmbeddedOut = testLayer(english)
    print(english.shape)
    print(endEmbeddedOut.shape)
```

```
(16, 13, 128)
(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 import Model
def encoder():
    inputs = Input([13,128])
    h = myLayer()(inputs)
    h = Masking(mask_value = tf.zeros((1,128)))(h)
    h, hidden_state, cell_state = LSTM(512, return_sequences = True, return_state=True)(h)

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

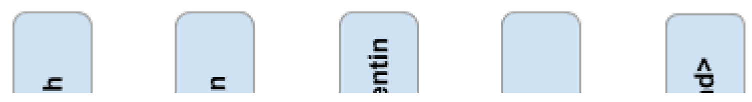
```
encoder1 = encoder()
for eng,ger in datasetTrain.take(1):
        print(eng.shape)
        state,cell = encoder1(eng)

print(tf.shape(state))
print(tf.shape(cell))
encoder1.summary()
```

```
(16, 13, 128)
tf.Tensor([ 16 512], shape=(2,), dtype=int32)
tf.Tensor([ 16 512], shape=(2,), dtype=int32)
Model: "model"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_1 (InputLayer)        [(None, 13, 128)]         0
_________________________________________________________________
my_layer_1 (myLayer)        (None, 14, 128)           128
_________________________________________________________________
masking (Masking)           (None, 14, 128)           0
_________________________________________________________________
lstm (LSTM)                 [(None, 14, 512), (None,  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.

```
# 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.

```
uniqueGermanTokens = len(tokenizer.word_index) + 1

from tensorflow.keras.layers import Dense, Embedding
class decoder(Model):
    def __init__(self, **kwargs):
        super(decoder, self).__init__(**kwargs)
        self.embedding = Embedding(input_dim=uniqueGermanTokens, output_dim = 128, mask_zero = True)
        self.LSTM = LSTM(512, return_sequences= True, return_state=True)
        self.dense = Dense(uniqueGermanTokens)

    def call(self, inputs, hidden_state=None, cell_state=None):
        h = self.embedding(inputs)
        lstm, hidden,cell = self.LSTM(h, initial_state = [hidden_state, cell_state])
        h = self.dense(lstm)
        return h,hidden,cell



decoderModel = decoder()
for eng, ger in datasetTrain.take(1):
    state,cell = encoder()(eng)
    res, hidden, cell = decoderModel(ger, state,cell)

    print(tf.shape(res))
    print(tf.shape(hidden))
    print(tf.shape(cell))

    decoderModel.summary()
```

```
tf.Tensor([  16   14 5744], shape=(3,), dtype=int32)
tf.Tensor([ 16 512], shape=(2,), dtype=int32)
tf.Tensor([ 16 512], shape=(2,), dtype=int32)
Model: "decoder"
_________________________________________________________________
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.

```
def gerIO(y):
    gerIn = np.zeros(y.shape)
    gerOut = np.zeros(y.shape)

    text = np.array(tokenizer.sequences_to_texts(np.array(y)))
    for i in range(y.shape[0]):
        tempIn = ' '.join(text[i].split(' ')[0:-1])
```

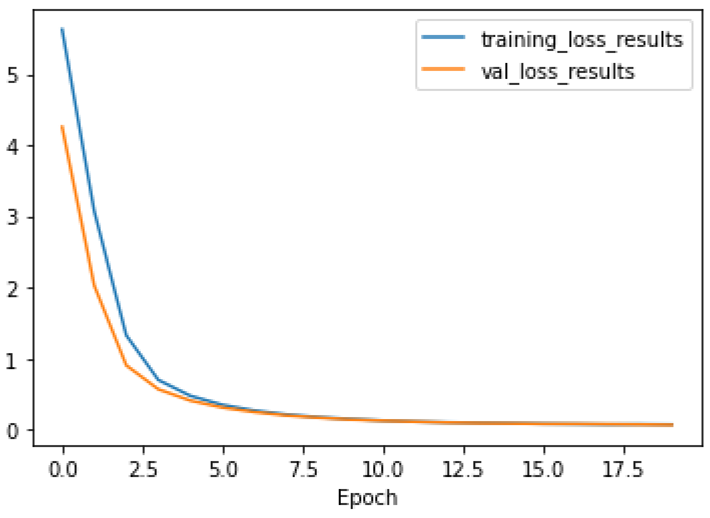
```
        tempOut = ' '.join(text[i].split(' ')[1:])
        tempSeqIn = tokenizer.texts_to_sequences([tempIn])
        tempSeqOut = tokenizer.texts_to_sequences([tempOut])

        gerIn[i,:] = tf.keras.preprocessing.sequence.pad_sequences(
                                tempSeqIn, maxlen = 14, dtype='int32', padding='post',
                                truncating='post', value=0.0
                        )
        gerOut[i,:] = tf.keras.preprocessing.sequence.pad_sequences(
                                tempSeqOut, maxlen = 14, dtype='int32', padding='post',
                                truncating='post', value=0.0
                        )
    return tf.convert_to_tensor(gerIn, dtype=y.dtype),tf.convert_to_tensor(gerOut, dtype=y.dtype)


loss_obj = tf.keras.losses.SparseCategoricalCrossentropy(from_logits = True)
optimizer = tf.keras.optimizers.Adam(learning_rate=0.002)

@tf.function
def grad(x, gerIn,gerOut):
    with tf.GradientTape() as g:
        state,cell = myEncoder(x)
        res,_ ,_ = myDecoder(gerIn, state,cell)
        loss = tf.math.reduce_mean(loss_obj(gerOut, res))
        grads = g.gradient(loss, myEncoder.trainable_variables + myDecoder.trainable_variables)
    return loss, grads


myEncoder = encoder()
myDecoder = decoder()

loss_results = []
val_loss_results = []

for epochs in range(20):

    Tb = 0
    epoch_loss_avg = tf.keras.metrics.Mean()
    print('Epoch {:02d}: '.format(epochs + 1))
    for x, y in datasetTrain:
        gerIn,gerOut = gerIO(y)
        loss, grads = grad(x, gerIn, gerOut)
        optimizer.apply_gradients(zip(grads, myEncoder.trainable_variables + myDecoder.trainable_variables))
        epoch_loss_avg.update_state(loss)
        Tb += 1
        print("\r     trainingBatch {:02d}/1000, epoch_loss_avg {:.6f}".format(Tb,epoch_loss_avg.result().numpy()),end = "")
    loss_results.append(epoch_loss_avg.result())



    Tb = 0
    epoch_avg_val_loss = tf.keras.metrics.Mean()
    print('')
    for x, y in datasetTest:
        gerIn,gerOut = gerIO(y)
        loss, grads = grad(x, gerIn, gerOut)
        optimizer.apply_gradients(zip(grads, myEncoder.trainable_variables + myDecoder.trainable_variables))
        epoch_avg_val_loss.update_state(loss)
        Tb += 1
        print("\r     testBatch {:02d}/250, epoch_avg_val_loss {:.6f}".format(Tb,epoch_avg_val_loss.result().numpy()),end = "")
    val_loss_results.append(epoch_avg_val_loss.result())



    print("\n     Epoch {:02d} DONE! Avg. training loss = {:.6f}, Avg. validation loss = {:.6f} ".format(epochs + 1,epoch_loss_avg.result(), epoch_avg_val_loss.result()))
```

```
      testBatch 250/250, epoch_avg_val_loss 0.310274
      Epoch 06 DONE! Avg. training loss = 0.348172, Avg. validation loss = 0.310274
   Epoch 07:
      trainingBatch 1000/1000, epoch_loss_avg 0.267270
      testBatch 250/250, epoch_avg_val_loss 0.246731
      Epoch 07 DONE! Avg. training loss = 0.267270, Avg. validation loss = 0.246731
   Epoch 08:
      trainingBatch 1000/1000, epoch_loss_avg 0.211748
      testBatch 250/250, epoch_avg_val_loss 0.198619
      Epoch 08 DONE! Avg. training loss = 0.211748, Avg. validation loss = 0.198619
   Epoch 09:
      trainingBatch 1000/1000, epoch_loss_avg 0.174011
      testBatch 250/250, epoch_avg_val_loss 0.167424
      Epoch 09 DONE! Avg. training loss = 0.174011, Avg. validation loss = 0.167424
   Epoch 10:
      trainingBatch 1000/1000, epoch_loss_avg 0.148091
      testBatch 250/250, epoch_avg_val_loss 0.141159
      Epoch 10 DONE! Avg. training loss = 0.148091, Avg. validation loss = 0.141159
   Epoch 11:
      trainingBatch 1000/1000, epoch_loss_avg 0.126562
      testBatch 250/250, epoch_avg_val_loss 0.124563
      Epoch 11 DONE! Avg. training loss = 0.126562, Avg. validation loss = 0.124563
   Epoch 12:
      trainingBatch 1000/1000, epoch_loss_avg 0.111674
      testBatch 250/250, epoch_avg_val_loss 0.111510
      Epoch 12 DONE! Avg. training loss = 0.111674, Avg. validation loss = 0.111510
   Epoch 13:
      trainingBatch 1000/1000, epoch_loss_avg 0.101332
      testBatch 250/250, epoch_avg_val_loss 0.098443
      Epoch 13 DONE! Avg. training loss = 0.101332, Avg. validation loss = 0.098443
   Epoch 14:

      trainingBatch 1000/1000, epoch_loss_avg 0.093936
      testBatch 250/250, epoch_avg_val_loss 0.090527
      Epoch 14 DONE! Avg. training loss = 0.093936, Avg. validation loss = 0.090527
   Epoch 15:
      trainingBatch 1000/1000, epoch_loss_avg 0.088343
      testBatch 250/250, epoch_avg_val_loss 0.086296
      Epoch 15 DONE! Avg. training loss = 0.088343, Avg. validation loss = 0.086296
   Epoch 16:
      trainingBatch 1000/1000, epoch_loss_avg 0.081727
      testBatch 250/250, epoch_avg_val_loss 0.080545
      Epoch 16 DONE! Avg. training loss = 0.081727, Avg. validation loss = 0.080545
   Epoch 17:
      trainingBatch 1000/1000, epoch_loss_avg 0.077855
      testBatch 250/250, epoch_avg_val_loss 0.079805
      Epoch 17 DONE! Avg. training loss = 0.077855, Avg. validation loss = 0.079805
   Epoch 18:
      trainingBatch 1000/1000, epoch_loss_avg 0.073231
      testBatch 250/250, epoch_avg_val_loss 0.076242
      Epoch 18 DONE! Avg. training loss = 0.073231, Avg. validation loss = 0.076242
   Epoch 19:
      trainingBatch 1000/1000, epoch_loss_avg 0.075484
      testBatch 250/250, epoch_avg_val_loss 0.071658
```

```
    Epoch 19 DONE! Avg. training loss = 0.075484, Avg. validation loss = 0.071658
Epoch 20:
    trainingBatch 1000/1000, epoch_loss_avg 0.071796
    testBatch 250/250, epoch_avg_val_loss 0.065618
    Epoch 20 DONE! Avg. training loss = 0.071796, Avg. validation loss = 0.065618
```

```
from matplotlib.pyplot import plot
import matplotlib.pyplot as plt

plot(np.array(loss_results))
plot(np.array(val_loss_results))
plt.legend(['training_loss_results','val_loss_results'])
plt.xlabel('Epoch')
```

```
Text(0.5, 0, 'Epoch')
```



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.

```
rand = np.random.choice(20000,5)

ENGLISH = []
GERMAN = []
for j in rand:
    ans = data_examples[j].split('\t')
    ENGLISH = ENGLISH + [preprocess_sentence(ans[0])]
    GERMAN = GERMAN + [preprocess_sentence(ans[1])]

embbededEnglish = np.zeros((5,13,128))
for i in range(len(ENGLISH)):
    english = embedding_layer(tf.strings.split(ENGLISH[i],sep = ' '))
    embbededEnglish[i,:] = tf.pad(english, [[13-len(english), 0], [0, 0]], constant_values = 0)
print(embbededEnglish.shape)

import pandas as pd
reshaped = np.stack((ENGLISH, GERMAN),axis=-1)
Ans = pd.DataFrame(reshaped, columns=['English','German'])
Ans.style.set_properties(**{'text-align': 'left'})
```

```
(5, 13, 128)
```

	English	German
0	i want some money .	ich will geld .
1	tom doesn't sing .	tom singt nicht .
2	you need therapy .	sie brauchen eine therapie .
3	allow me to go .	gestatten sie mir , zu gehen .
4	i will stay .	ich werde bleiben .

```
start = tokenizer.word_index['<start>']
end = tokenizer.word_index['<end>']

res = []

for i in range(0,5):
    hidden_state, cell_state = myEncoder(tf.expand_dims(embbededEnglish[i],0))
    seq = []
    tf_token = tf.Variable([[start]])

    while True:
        output_1, hidden_state, cell_state = myDecoder(tf_token, hidden_state, cell_state)
        output_2 = tf.argmax(output_1,2).numpy()[0][0]
        if output_2 == 2:
            tf_token = tf.Variable([[start]])
            break
        seq += [output_2]
        tf_token = tf.Variable([[output_2]])
    res += [seq]

TRANSLATION = tokenizer.sequences_to_texts(res)


reshaped = np.stack((ENGLISH, TRANSLATION, GERMAN),axis=-1)
Ans = pd.DataFrame( reshaped, columns=['English','Translated','Origional German'])
Ans.style.set_properties(**{'text-align': 'left'})
```

	English	Translated	Origional German
0	i want some money .	ich will eine gute nachricht .	ich will geld .
1	tom doesn't sing .	tom lacht nie .	tom singt nicht .
2	you need therapy .	sie brauchen musik .	sie brauchen eine therapie .
3	allow me to go .	gebt mich zu gehen .	gestatten sie mir , zu gehen .
4	i will stay .	ich werde bleiben .	ich werde bleiben .

0s completed at 1:54 AM