

```
import tensorflow as tf
import tensorflow_hub as hub
import unicodedata
import re
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Layer
from google.colab import drive
drive.mount('/content/drive')
print(tf.__version__)
```

```
Mounted at /content/drive
2.15.0
```

✓ 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 (File -> Download as -> PDF via LaTeX). You should 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.

 Flags overview 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.

```
# Run this cell to load the dataset

NUM_EXAMPLES = 20000
data_examples = []

with open('/content/drive/MyDrive/Colab Notebooks/data/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.

 Model Schematic

Key:  Model key

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.

```
tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='!?!')
```

```
eng_sent = []
ger_sent = []
for line in data_examples:
    tmp_sentence = re.split(r'[\?!]', preprocess_sentence(line))
    eng_sent.append(tmp_sentence[0])
    ger_sent.append('<start>' + tmp_sentence[1] + '<end>')
for t in np.random.random((5,)):
    t = int(t)
    print('English sentence: ', eng_sent[t], '\n')
    print('German Sentence: ', ger_sent[t], '\n')
```

English sentence: hi

German Sentence: <start> hallo <end>

English sentence: hi

German Sentence: <start> hallo <end>

```

English sentence:  hi

German Sentence:  <start> hallo <end>

English sentence:  hi

German Sentence:  <start> hallo <end>

English sentence:  hi

German Sentence:  <start> hallo <end>

```

Double-click (or enter) to edit

```

tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='')
tokenizer.fit_on_texts(ger_sent)
ger_sent = tokenizer.texts_to_sequences(ger_sent)
ger_padd_sent = tf.keras.preprocessing.sequence.pad_sequences(ger_sent, padding='post')

```

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

✓ Load the embedding layer

As part of the dataset preprocessing for this project, you will use a pre-trained English word embedding module from TensorFlow Hub. The URL for the module is <https://tfhub.dev/google/tf2-preview/nnlm-en-dim128-with-normalization/1>. This module has also been made available as a complete saved model in the folder `./models/tf2-preview_nnlm-en-dim128_1`.

This embedding takes a batch of text tokens in a 1-D tensor of strings as input. It then embeds the separate tokens into a 128-dimensional space.

The code to load and test the embedding layer is provided for you below.

NB: this model can also be used as a sentence embedding module. The module will process each token by removing punctuation and splitting on spaces. It then averages the word embeddings over a sentence to give a single embedding vector. However, we will use it only as a word embedding module, and will pass each word in the input sentence as a separate token.

```

# 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 more than 13 (embedded) tokens in length. Apply this function to both Dataset objects using the filter method.
- Create a function to map over the datasets that pads each English sequence of embeddings with some distinct padding value before the sequence, so that each sequence is length 13. Apply this function to both Dataset objects using the map method. *Hint: look at the tf.pad function. You can extract a Tensor shape using tf.shape; you might also find the tf.math.maximum function useful.*

- Batch both training and validation Datasets with a batch size of 16.
- Print the `element_spec` property for the training and validation Datasets.
- Using the Dataset `.take(1)` method, print the shape of the English data example from the training Dataset.
- Using the Dataset `.take(1)` method, print the German data example Tensor from the validation Dataset.

```
eng_train, eng_test, ger_train, ger_test = train_test_split( eng_sent, ger_padd_sent, test_size=0.2)
dataset_val = tf.data.Dataset.from_tensor_slices((eng_test, ger_test))
dataset_train = tf.data.Dataset.from_tensor_slices((eng_train, ger_train))
```

```
def split_sentence_space(dataset):
    def map_data(eng_sent, ger_sent):
        tmp_sent_list = tf.strings.split(eng_sent, sep=' ')
        return tmp_sent_list, ger_sent
    dataset = dataset.map(map_data)
    return dataset
dataset_train = split_sentence_space(dataset_train)
dataset_val = split_sentence_space(dataset_val)
```

```
def embed_sentence(dataset):
    def map_data(eng_word, ger_sent):
        tmp_word_list = embedding_layer(eng_word)
        return tmp_word_list, ger_sent
    dataset = dataset.map(map_data)
    return dataset
embedded_dataset_train = embed_sentence(dataset_train)
embedded_dataset_val = embed_sentence(dataset_val)
```

```
def filter_long_length(dataset):
    def filter_fn(token, ger_sent):
        return tf.shape(token)[0] <= 13
    filtered_data = dataset.filter(filter_fn)
    return filtered_data
filtered_long_train_dataset = filter_long_length(embedded_dataset_train)
filtered_long_val_dataset = filter_long_length(embedded_dataset_val)
```

```
def filter_dataset(dataset):
    def pad_to_length_13(eng_emb, ger_sent):
        padding_value = -1 # You can set any distinct padding value
        padding_needed = tf.math.maximum(0, 13 - tf.shape(eng_emb)[0])
        padded_eng_emb = tf.pad(eng_emb, paddings=[[0, padding_needed], [0, 0]], constant_values=padding_value)
        return padded_eng_emb, ger_sent
    padded_dataset = dataset.map(pad_to_length_13)
    return padded_dataset
filtered_train_dataset = filter_dataset(filtered_long_train_dataset)
filtered_val_dataset = filter_dataset(filtered_long_val_dataset)
```


```
batched_train_dataset = filtered_train_dataset.batch(16)
batched_val_dataset = filtered_val_dataset.batch(16)
```

```
# Print the shape of the English data example from the training dataset
sample_data_train = next(iter(batched_train_dataset.take(1)))
print("Shape of English Data Example (Training):", sample_data_train[0].shape)
sample_data_val = next(iter(batched_val_dataset.take(1)))
print("Shape of German Data Example (Validation):", sample_data_val[1].shape)
```

```
Shape of English Data Example (Training): (16, 13, 128)
Shape of German Data Example (Validation): (16, 13)
```

✓ 3. Create the custom layer

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

 Encoder schematic

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 Embedding_Layer_class(Layer):
    def __init__(self, **kwargs):
        super(Embedding_Layer_class, self).__init__(**kwargs)

    def build(self, input_shape):
        # Load embedding module from Tensorflow Hub
        self.embedding_layer = hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim128/1",
                                              output_shape=[128], input_shape=[], dtype=tf.string)

    def call(self, inputs):
        # Load the 'end' token embedding
        end_token_embedding = self.embedding_layer(tf.constant(['end']))
        # Get the batch size
        inputs = tf.convert_to_tensor(inputs)
        batch_size = tf.shape(inputs)[0]
        # Tile the 'end' token embedding across the sequence length and batch size
        tiled_end_token_embedding = tf.tile(
            end_token_embedding, # Add batch dimension
            [batch_size, 1] # Tile across the sequence length
        )
        output = tf.concat([inputs, tf.expand_dims(tiled_end_token_embedding,axis=1)], axis=1)
        return output

sample_data_batch_eng = next(iter(batched_train_dataset))[0]
embedded_layer = Embedding_Layer_class()
output_embedded_layer = embedded_layer(sample_data_batch_eng)
```

```
output_embedded_layer.shape

TensorShape([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.

```

class EncoderModel(tf.keras.Model):
    def __init__(self, embedding_dim=128):
        super().__init__()
        self.input_layer = tf.keras.layers.Input(shape=(None, 128))
        self.embedding = Embedding_Layer_class()
        self.masking_layer = tf.keras.layers.Masking(mask_value=-1) # Replace with your padding value
        self.lstm_layer = tf.keras.layers.LSTM(512, return_sequences=True, return_state=True)

    def call(self, inputs):
        embedded_inputs = self.embedding(inputs)
        masked_inputs = self.masking_layer(embedded_inputs)
        _, hidden_state, cell_state = self.lstm_layer(masked_inputs)
        return hidden_state, cell_state

embedding_layer_class = Embedding_Layer_class()
sample_data_eng = next(iter(batched_train_dataset))[0]

encoder_model = EncoderModel()

out_1, out_2 = encoder_model(sample_data_eng)
out_1.shape, out_2.shape

(TensorShape([16, 512]), TensorShape([16, 512]))

encoder_model.summary()

```

Model: "encoder_model"

Layer (type)	Output Shape	Param #
embedding_layer_class_2 (Embedding_Layer_class)	multiple	124642688
masking (Masking)	multiple	0
lstm (LSTM)	multiple	1312768
Total params: 125955456 (480.48 MB)		
Trainable params: 1312768 (5.01 MB)		
Non-trainable params: 124642688 (475.47 MB)		

5. Build the decoder network

The decoder network follows the schematic diagram below.



You should now build the RNN decoder model.

- Using Model subclassing, build the decoder network according to the following spec:
 - The initializer should create the following layers:
 - An Embedding layer with vocabulary size set to the number of unique German tokens, embedding dimension 128, and set to mask zero values in the input.
 - An LSTM layer with 512 units, that returns its hidden and cell states, and also returns sequences.
 - A Dense layer with number of units equal to the number of unique German tokens, and no activation function.
 - The call method should include the usual `inputs` argument, as well as the additional keyword arguments `hidden_state` and `cell_state`. The default value for these keyword arguments should be `None`.
 - The call method should pass the inputs through the Embedding layer, and then through the LSTM layer. If the `hidden_state` and `cell_state` arguments are provided, these should be used for the initial state of the LSTM layer. *Hint: use the `initial_state` keyword argument when calling the LSTM layer on its input.*
 - The call method should pass the LSTM output sequence through the Dense layer, and return the resulting Tensor, along with the hidden and cell states of the LSTM layer.
- Using the Dataset `.take(1)` method, extract a batch of English and German data examples from the training Dataset. Test the decoder model by first calling the encoder model on the English data Tensor to get the hidden and cell states, and then call the decoder model on

the German data Tensor and hidden and cell states, and print the shape of the resulting decoder Tensor outputs.

- Print the model summary for the decoder network.

```
vocab_size_ger = len(tokenizer.word_index) + 1
class DecoderNetwork(tf.keras.Model):
    def __init__(self, vocab_size=vocab_size_ger, embedding_dim=128):
        super(DecoderNetwork, self).__init__()
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim, mask_zero=True)
        self.lstm = tf.keras.layers.LSTM(512, return_sequences=True, return_state=True)
        self.dense = tf.keras.layers.Dense(vocab_size)

    def call(self, inputs, hidden_state=None, cell_state=None):
        embedded_inputs = self.embedding(inputs)
        lstm_outputs, hidden_state, cell_state = self.lstm(embedded_inputs, initial_state=(hidden_state, cell_state))
        predictions = self.dense(lstm_outputs)
        return predictions, hidden_state, cell_state

data=next(iter(batched_train_dataset.take(1)))
english_data = data[0]
german_data = data[1]
encoder_hidden_state, encoder_cell_state = EncoderModel()(english_data)

decoder = DecoderNetwork()
# Call the decoder model on the German data Tensor
decoder_outputs, decoder_hidden_state, decoder_cell_state = decoder(german_data, hidden_state=encoder_hidden_state, cell_state=encoder_cell_state)
print(decoder_outputs.shape)

(16, 13, 5746)
```

```
decoder.summary()
```

Model: "decoder_network"

Layer (type)	Output Shape	Param #
=====		
embedding (Embedding)	multiple	735488
lstm_2 (LSTM)	multiple	1312768
dense (Dense)	multiple	2947698
=====		
Total params: 4995954 (19.06 MB)		
Trainable params: 4995954 (19.06 MB)		
Non-trainable params: 0 (0.00 Byte)		

✓ 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 german_outs(german_data):
    german_input = german_data[:, :-1]
    german_output = german_data[:, 1:]
    return german_input, german_output

optimizer = tf.keras.optimizers.Adam()
decoder_model = DecoderNetwork()
encoder_model = EncoderModel()
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

@tf.function
def forward_backward_pass(english_input, german_input, german_output, loss_obj, encoder_obj, decoder_obj):

    hidden_state, cell_state = encoder_obj(english_input)

    decoder_outputs, _, _ = decoder_obj(german_input, hidden_state=hidden_state, cell_state=cell_state)

    # Cast decoder_outputs to the data type of german_output
    german_output = tf.cast(german_output, dtype=decoder_outputs.dtype)
    all_variables = encoder_model.trainable_variables + decoder_model.trainable_variables

    # Compute gradients manually using tf.gradient
    loss_value = loss_obj(german_output, decoder_outputs)
    mean_loss = tf.reduce_mean(loss_value)

    return mean_loss, tf.gradients(mean_loss, all_variables)

def training_loop(dataset, val_dataset, epochs, grad_fn, loss_obj, encoder_obj, decoder_obj, optim_obj):
    train_losses = []

    val_losses = []

    for epoch in range(epochs):
        epoch_loss_avg = tf.keras.metrics.Mean()

        for eng_data, german_data in dataset:
            germ_input, germ_output = german_outs(german_data=german_data)

            mean_loss, gradient = grad_fn(eng_data, germ_input, germ_output, loss_obj, encoder_obj, decoder_obj)
            #gradient = [tf.convert_to_tensor(g) if g is not None else None for g in gradient]
            optim_obj.apply_gradients(zip(gradient, encoder_obj.trainable_variables + decoder_obj.trainable_variables))

            epoch_loss_avg(mean_loss)

        train_losses.append(epoch_loss_avg.result())

        # Validation
        val_loss_avg = tf.keras.metrics.Mean()
        for val_eng_data, val_german_data in val_dataset:
            val_germ_input, val_germ_output = german_outs(val_german_data)

            val_mean_loss, _ = grad_fn(val_eng_data, val_germ_input, val_germ_output, loss_obj, encoder_obj, decoder_obj)

            val_loss_avg(val_mean_loss)

        val_losses.append(val_loss_avg.result())

        print("Epoch {}: Training Loss: {:.4f}, Validation Loss: {:.4f}".format(epoch, epoch_loss_avg.result(), val_loss_avg.result()))

    return train_losses, val_losses
```

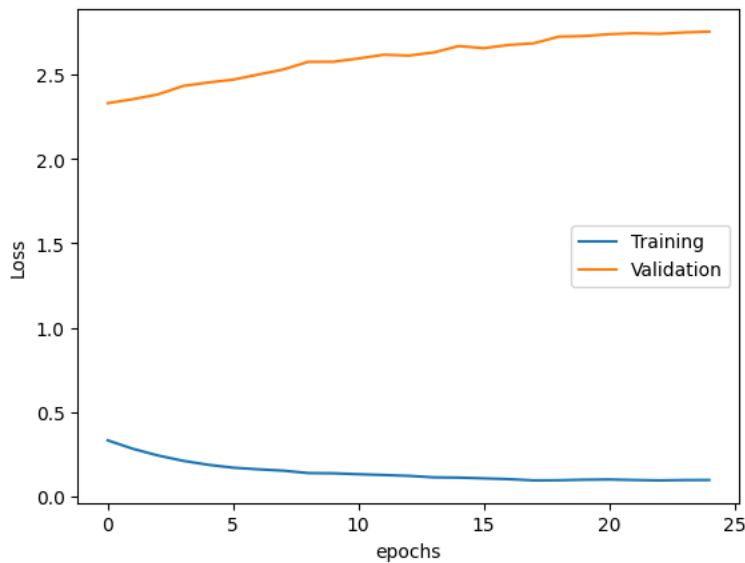


```
loss_train, loss_val = training_loop(batched_train_dataset, batched_val_dataset, epochs=25,
                                     loss_obj=loss_fn, grad_fn=forward_backward_pass, encoder_obj=encoder_model, decoder_obj=decoder_model,
```

Epoch 0: Training Loss: 0.3328, Validation Loss: 2.3304
Epoch 1: Training Loss: 0.2829, Validation Loss: 2.3539
Epoch 2: Training Loss: 0.2433, Validation Loss: 2.3823
Epoch 3: Training Loss: 0.2118, Validation Loss: 2.4321
Epoch 4: Training Loss: 0.1883, Validation Loss: 2.4528
Epoch 5: Training Loss: 0.1709, Validation Loss: 2.4696
Epoch 6: Training Loss: 0.1615, Validation Loss: 2.4998
Epoch 7: Training Loss: 0.1533, Validation Loss: 2.5299
Epoch 8: Training Loss: 0.1391, Validation Loss: 2.5752
Epoch 9: Training Loss: 0.1377, Validation Loss: 2.5757
Epoch 10: Training Loss: 0.1323, Validation Loss: 2.5948
Epoch 11: Training Loss: 0.1275, Validation Loss: 2.6173
Epoch 12: Training Loss: 0.1226, Validation Loss: 2.6124
Epoch 13: Training Loss: 0.1137, Validation Loss: 2.6304
Epoch 14: Training Loss: 0.1121, Validation Loss: 2.6688
Epoch 15: Training Loss: 0.1077, Validation Loss: 2.6563
Epoch 16: Training Loss: 0.1031, Validation Loss: 2.6752
Epoch 17: Training Loss: 0.0958, Validation Loss: 2.6852
Epoch 18: Training Loss: 0.0966, Validation Loss: 2.7245
Epoch 19: Training Loss: 0.0997, Validation Loss: 2.7279
Epoch 20: Training Loss: 0.1011, Validation Loss: 2.7386
Epoch 21: Training Loss: 0.0980, Validation Loss: 2.7448
Epoch 22: Training Loss: 0.0956, Validation Loss: 2.7409
Epoch 23: Training Loss: 0.0976, Validation Loss: 2.7495
Epoch 24: Training Loss: 0.0981, Validation Loss: 2.7542

```
import matplotlib.pyplot as plt
```

```
plt.plot(loss_train, label='Training')
plt.plot(loss_val, label='Validation')
plt.xlabel('epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



▼ 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.

```

# Function to preprocess and embed the English sentence
def preprocess_and_embed(sentence):
    sentence = preprocess_sentence(sentence)
    sentence = tf.strings.split(sentence, sep=' ')
    embedded_sentence = embedding_layer(sentence)
    return tf.expand_dims(embedded_sentence, axis=0)

# Function to translate an English sentence to German using the trained model
def translate_sentence(encoder, decoder, sentence):
    # Preprocess and embed the English sentence
    embedded_sentence = preprocess_and_embed(sentence)

    # Pass the embedded sentence through the encoder to get hidden and cell states
    encoder_hidden, encoder_cell = encoder(embedded_sentence)

    # Initialize the decoder input with the '<start>' token
    decoder_input = tf.constant(tokenizer.texts_to_sequences(['<start>']), dtype=tf.float32)

    # Initialize an empty list to store the decoded words
    decoded_words = []

    # Create a loop to generate the translation
    for _ in range(embedded_sentence.shape[1]+1):
        # Pass the decoder input and encoder states through the decoder
        predictions, decoder_hidden, decoder_cell = decoder(decoder_input, hidden_state=encoder_hidden, cell_state=encoder_cell)

        # Get the predicted word index
        predicted_word_index = tf.argmax(predictions, axis=-1).numpy()[0,-1]

        # If the predicted word is '<end>', terminate the loop
        if predicted_word_index == tokenizer.word_index['<end>']:
            break

        # Convert the predicted word index to its corresponding word
        predicted_word = tokenizer.index_word[predicted_word_index]

        # Append the predicted word to the list of decoded words
        decoded_words.append(predicted_word)

        # Update the decoder input for the next iteration
        decoder_input = tf.constant(tokenizer.texts_to_sequences([predicted_word]), dtype=tf.float32)

    # Join the decoded words to form the German translation
    german_translation = ' '.join(decoded_words)

    return german_translation

# Choose five random indices from the validation dataset
random_indices = np.random.choice(len(eng_test), size=5, replace=False)

# Translate and print the English sentence along with the model's German translation
for index in random_indices:
    english_sentence = eng_test[index]
    german_translation = translate_sentence(encoder_model, decoder_model, english_sentence)

    print("English Sentence:", english_sentence)
    print("German Translation:", german_translation)
    print()

    English Sentence: what did she say
    German Translation: was was was was was

    English Sentence: is it really you
    German Translation: bist es ist es ist

    English Sentence: i like sandwiches
    German Translation: ich mag ich mag

```

English Sentence: isn't that mine
German Translation: ist das ist das

English Sentence: i sat beside her
German Translation: ich ich ich ich ich