Calvin Tran

Given a tab-delimited bilingual sentence pair (https://www.manythings.org/anki/) contained in dataset with the following format:

English + TAB + German + TAB + Attribution

Develop a sequence to sequence model using RNN to translate English to German, character-wise which is uncommon in practice but a good example of the model implementation.

In [1]:
```
import os
import random
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.callbacks import EarlyStopping

import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
```

In [2]:
```
SEED = 1234
os.environ['PYTHONHASHSEED']=str(SEED)
os.environ['TF_CUDNN_DETERMINISTIC'] = '1'
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)
```

Load the data, Preprocessing

- Read the data into Python, and split the lines separated by the '\n' character.
- Separate input texts from target texts, which are separated by the '\t' character.
- Discard the translator attribution, which is only present as required for the license to distribute the data.
- Store the unique characters in the input and target texts, which will be used later by the model.
- Similarly store the maximum sequence length in the input and target texts to also be used later by the model.
- Convert the data from text format to numeric for the model to handle.

In [3]:
```
#read in data and split lines
data_path = "C:\\Users\\Cal\\Desktop\\DSCI 619 Deep Learning\\Week 6 Sequence to Se
with open(data_path, "r", encoding="utf-8") as f:
```

```
    lines = f.read().split("\n")
print(lines[0])
```

```
Go.	Geh.	CC-BY 2.0 (France) Attribution: tatoeba.org #2877272 (CM) & #8597805
(Roujin)
```

In [4]:

```
#set some hyperparameters to feed the data in batches

# Set the batch size
batch_size = 64
# Set the epochs number
epochs = 100

# Latent dimensionality of the encoding space.
latent_dim = 256
# Number of samples to use.
num_samples = 10000
```

In [5]:

```
# Obtain the features (input) and labels (target)
input_texts = []
target_texts = []
# Unique characters in the inputs and targets
input_characters = set()
target_characters = set()


# Process line by line
for line in lines[: min(num_samples, len(lines) - 1)]:
    # Data format English + TAB + The Other Language + TAB + Attribution
    # It returns: English, The other language, and Attribution, which is discarded
    input_text, target_text, _ = line.split("\t")

    # We use "tab" as the "start sequence" character
    # for the targets, and "\n" as "end sequence" character.
    target_text = "\t" + target_text + "\n"
    input_texts.append(input_text)
    target_texts.append(target_text)

    # Get the unique char from the input texts
    for char in input_text:
        if char not in input_characters:
            input_characters.add(char)
    # Get the unique char from the target texts
    for char in target_text:
        if char not in target_characters:
            target_characters.add(char)
```

In [6]:

```
print(input_characters)
print('')
print(target_characters)
```

```
{'v', 'B', 'x', 'o', 'u', 'e', "'", 'Q', '8', 'f', '-', 'c', 'F', 'D', 'i', '$',
'3', '.', 'a', '%', 'I', 'm', '5', 'G', 't', 'l', 'd', '4', 'E', 'N', 'Y', 'H', 'n',
'2', 'V', '"', 'J', '!', 'C', 'p', '1', 'T', 'S', 'M', 's', 'A', 'g', 'h', ' ', 'j',
'O', 'R', 'k', 'w', 'z', 'P', 'U', '7', '6', ',', '?', 'K', ':', 'b', 'L', 'q', 'r',
'y', 'W', '0', '9'}

{'v', "'", 'Q', '\u202f', 'D', '’', 'ö', '3', '.', 'l', 'd', 'J', 'S', '\t', 'ü',
'M', 'Ö', 'h', 'j', 'P', 'z', 'Ä', ',', 'K', 'x', '“', 'o', 'ä', 'f', 'Z', 'i', 'I',
'ß', '4', 'E', 'N', '„', 'H', 'A', 'w', '0', 'c', '$', 'a', '5', 'G', 't', 'n', '!',
'1', 's', 'k', 'U', '7', '6', ':', 'L', 'y', '\n', 'B', 'e', 'u', '8', '-', 'F',
'%', 'm', 'Y', '2', 'V', 'T', 'C', 'p', '\xa0', 'g', ' ', 'O', 'R', 'Ü', '?', 'b',
'q', 'r', 'W', '9'}
```

Preprocessing the unique characters in the input and target texts

- Also obtain max sequence lengths

In [7]:
```python
#sort the unique characters
input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))

#count the length of unique characters
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)

#determine the maximum length of any sequence within the input and target texts
max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

print(f'Number of samples:= {len(input_texts)}')
print(f"Number of unique input tokens: = {num_encoder_tokens}")
print(f"Number of unique output tokens: = {num_decoder_tokens}")
print(f"Max sequence length for inputs: = {max_encoder_seq_length}")
print(f"Max sequence length for outputs: = {max_decoder_seq_length}")
print(f'Sorted input characters: \n {input_characters}')
print(f'Sorted target characters: \n {target_characters}')
```

```
Number of samples:= 10000
Number of unique input tokens: = 71
Number of unique output tokens: = 85
Max sequence length for inputs: = 15
Max sequence length for outputs: = 45
Sorted input characters:
 [' ', '!', '"', '$', '%', "'", ',', '-', '.', '0', '1', '2', '3', '4', '5', '6',
'7', '8', '9', ':', '?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'Y', 'a', 'b', 'c', 'd', 'e',
'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
'w', 'x', 'y', 'z']
Sorted target characters:
 ['\t', '\n', ' ', '!', '$', '%', "'", ',', '-', '.', '0', '1', '2', '3', '4', '5',
'6', '7', '8', '9', ':', '?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'Y', 'Z', 'a', 'b', 'c',
'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
'u', 'v', 'w', 'x', 'y', 'z', '\xa0', 'Ä', 'Ö', 'Ü', 'ß', 'ä', 'ö', 'ü', ''', '“',
'„', '\u202f']
```

Convert text data to numeric for use with deep learning model

In [8]:

```python
#create simple mappings of characters to indices and store in dictionaries
input_token_index = dict([(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict([(char, i) for i, char in enumerate(target_characters)])
print(input_token_index)
print(target_token_index)
```

```
{' ': 0, '!': 1, '"': 2, '$': 3, '%': 4, "'": 5, ',': 6, '-': 7, '.': 8, '0': 9,
'1': 10, '2': 11, '3': 12, '4': 13, '5': 14, '6': 15, '7': 16, '8': 17, '9': 18,
':': 19, '?': 20, 'A': 21, 'B': 22, 'C': 23, 'D': 24, 'E': 25, 'F': 26, 'G': 27,
'H': 28, 'I': 29, 'J': 30, 'K': 31, 'L': 32, 'M': 33, 'N': 34, 'O': 35, 'P': 36,
'Q': 37, 'R': 38, 'S': 39, 'T': 40, 'U': 41, 'V': 42, 'W': 43, 'Y': 44, 'a': 45,
'b': 46, 'c': 47, 'd': 48, 'e': 49, 'f': 50, 'g': 51, 'h': 52, 'i': 53, 'j': 54,
'k': 55, 'l': 56, 'm': 57, 'n': 58, 'o': 59, 'p': 60, 'q': 61, 'r': 62, 's': 63,
't': 64, 'u': 65, 'v': 66, 'w': 67, 'x': 68, 'y': 69, 'z': 70}
{'\t': 0, '\n': 1, ' ': 2, '!': 3, '$': 4, '%': 5, "'": 6, ',': 7, '-': 8, '.': 9,
'0': 10, '1': 11, '2': 12, '3': 13, '4': 14, '5': 15, '6': 16, '7': 17, '8': 18,
'9': 19, ':': 20, '?': 21, 'A': 22, 'B': 23, 'C': 24, 'D': 25, 'E': 26, 'F': 27,
'G': 28, 'H': 29, 'I': 30, 'J': 31, 'K': 32, 'L': 33, 'M': 34, 'N': 35, 'O': 36,
'P': 37, 'Q': 38, 'R': 39, 'S': 40, 'T': 41, 'U': 42, 'V': 43, 'W': 44, 'Y': 45,
'Z': 46, 'a': 47, 'b': 48, 'c': 49, 'd': 50, 'e': 51, 'f': 52, 'g': 53, 'h': 54,
'i': 55, 'j': 56, 'k': 57, 'l': 58, 'm': 59, 'n': 60, 'o': 61, 'p': 62, 'q': 63,
'r': 64, 's': 65, 't': 66, 'u': 67, 'v': 68, 'w': 69, 'x': 70, 'y': 71, 'z': 72, '\x
a0': 73, 'Ä': 74, 'Ö': 75, 'Ü': 76, 'ß': 77, 'ä': 78, 'ö': 79, 'ü': 80, ''': 81,
'“': 82, '„': 83, '\u202f': 84}
```

Convert the actual data

Convert texts to 3D matrix:

- first dimension = number of samples
- second dimension: maximum length of inputs/outputs respectively
- third dimension: number of unique characters

Each element of the matrix only has two values, 0 or 1

- 0 means the corresponding character does not show up in the texts
- 1 means that the corresponding character shows up in the texts

In [9]:
```
# Initialize the 3D matrix
encoder_input_data = np.zeros((len(input_texts), max_encoder_seq_length, num_encode
decoder_input_data = np.zeros((len(target_texts), max_decoder_seq_length, num_decod
decoder_target_data = np.zeros((len(target_texts), max_decoder_seq_length, num_deco
```

In [10]:
```
# Convert the input texts and target texts to numerical values
for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    # Convert input texts to numerical values
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.0
    # Pad the remaining using space that is mapped to 1
    encoder_input_data[i, t + 1 :, input_token_index[" "]] = 1.0
    # Convert target texts to numerical values
    for t, char in enumerate(target_text):
        # decoder input is used to forecast decoder target
        # decoder_target_data is ahead of decoder_input_data by one timestep
        decoder_input_data[i, t, target_token_index[char]] = 1.0
        if t > 0:
            # decoder_target_data will be ahead by one timestep
            # and will not include the start character.
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.0
    # Pad the remaining using space that is mapped to 1
    decoder_input_data[i, t + 1 :, target_token_index[" "]] = 1.0
    decoder_target_data[i, t:, target_token_index[" "]] = 1.0
```

In [11]:
```
print(encoder_input_data[0])

print(encoder_input_data.shape)
print(decoder_input_data.shape)
print(decoder_target_data.shape)
```

```
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [1. 0. 0. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]]
(10000, 15, 71)
(10000, 45, 85)
(10000, 45, 85)
```

Build the Inference Model

Two key layers

- Encoder layer (LSTM layer): Processes inputs and returns its own state. The outputs are disarded. The hidden states will be used as the context of the decoder layer.
- Decoder layer (LSTM layer): Predicts next characters of the target sequence given the previous characters of that target sequence. Receives state vectors from the encoder layer as initial state to obtain information on what to generate.

The dimensions of the encoder and decoder layers were defined above with other hyperparameters.

Develop an encoder for the model

- for the encoder, the outputs are discarded. The states are kept and passed to the decoder.

In [13]:
```
#encoder, inputs and layer
encoder_inputs = keras.Input(shape=(None, num_encoder_tokens)) #num_encoder_tokens
encoder = keras.layers.LSTM(latent_dim, return_state=True)

# Apply encoder LSTM layer, return the ouputs and hidden states of the encoder, but
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]
```

Develop a decoder for the model

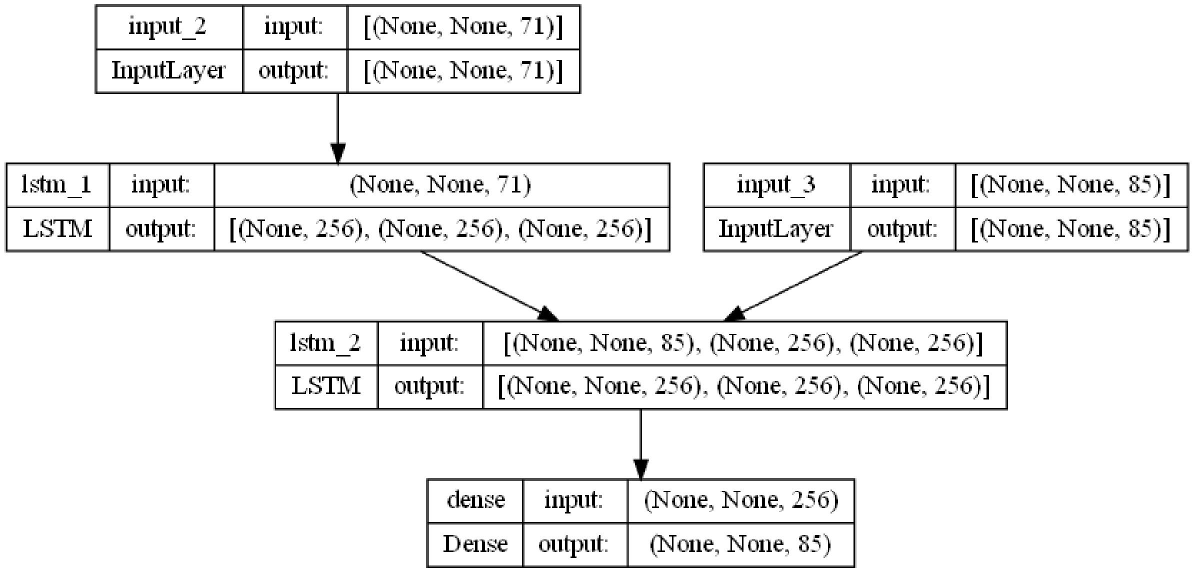
In [14]:
```
#decoder, inputs and layer
decoder_inputs = keras.Input(shape=(None, num_decoder_tokens))
decoder_lstm = keras.layers.LSTM(latent_dim, return_sequences=True, return_state=Tr

#Apply decoder LSTM layer save the outputs of the decoder, pass these to the final
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = keras.layers.Dense(num_decoder_tokens, activation="softmax")
decoder_outputs = decoder_dense(decoder_outputs)
```

Build the sequence to sequence model

In [15]:
```
#define the model and specify optimizer, loss function, metrics
model = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer="rmsprop", loss="categorical_crossentropy", metrics=["accur
```

In [16]:
```
#vizualize model input and output sizes
tf.keras.utils.plot_model(model, show_shapes=True)
```

Out[16]:



Train the model and save the results in history

In [17]:

```
#from tensorflow.keras.callbacks import EarlyStopping
# Set up early stopping criteria
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    patience=5,
    min_delta=0.001,
    mode='max'
)

#fit the model
history = model.fit(
    [encoder_input_data, decoder_input_data], #inputs are a list of both
    decoder_target_data,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.2,
    callbacks=[early_stopping],
)
# Save model
model.save("s2s")
```

```
Epoch 1/100
125/125 [==============================] - 17s 123ms/step - loss: 1.4461 - accuracy:
0.6471 - val_loss: 1.3301 - val_accuracy: 0.6244
Epoch 2/100
125/125 [==============================] - 15s 120ms/step - loss: 1.0408 - accuracy:
0.7235 - val_loss: 1.0104 - val_accuracy: 0.7327
Epoch 3/100
125/125 [==============================] - 15s 122ms/step - loss: 0.8415 - accuracy:
0.7664 - val_loss: 0.8904 - val_accuracy: 0.7510
Epoch 4/100
125/125 [==============================] - 15s 121ms/step - loss: 0.7415 - accuracy:
0.7890 - val_loss: 0.8528 - val_accuracy: 0.7564
Epoch 5/100
125/125 [==============================] - 15s 121ms/step - loss: 0.6793 - accuracy:
0.8045 - val_loss: 0.7797 - val_accuracy: 0.7774
Epoch 6/100
125/125 [==============================] - 16s 125ms/step - loss: 0.6343 - accuracy:
0.8163 - val_loss: 0.7331 - val_accuracy: 0.7925
Epoch 7/100
125/125 [==============================] - 16s 126ms/step - loss: 0.5967 - accuracy:
0.8274 - val_loss: 0.7027 - val_accuracy: 0.7986
Epoch 8/100
125/125 [==============================] - 15s 120ms/step - loss: 0.5655 - accuracy:
0.8363 - val_loss: 0.6768 - val_accuracy: 0.8046
Epoch 9/100
125/125 [==============================] - 15s 121ms/step - loss: 0.5394 - accuracy:
0.8432 - val_loss: 0.6550 - val_accuracy: 0.8110
Epoch 10/100
125/125 [==============================] - 15s 121ms/step - loss: 0.5163 - accuracy:
0.8496 - val_loss: 0.6492 - val_accuracy: 0.8124
Epoch 11/100
125/125 [==============================] - 15s 120ms/step - loss: 0.4960 - accuracy:
0.8559 - val_loss: 0.6260 - val_accuracy: 0.8215
Epoch 12/100
125/125 [==============================] - 15s 120ms/step - loss: 0.4781 - accuracy:
0.8607 - val_loss: 0.6131 - val_accuracy: 0.8249
Epoch 13/100
125/125 [==============================] - 15s 121ms/step - loss: 0.4607 - accuracy:
0.8656 - val_loss: 0.6026 - val_accuracy: 0.8278
Epoch 14/100
125/125 [==============================] - 15s 118ms/step - loss: 0.4445 - accuracy:
0.8701 - val_loss: 0.5996 - val_accuracy: 0.8290
Epoch 15/100
125/125 [==============================] - 15s 118ms/step - loss: 0.4290 - accuracy:
0.8745 - val_loss: 0.5894 - val_accuracy: 0.8331
Epoch 16/100
125/125 [==============================] - 15s 121ms/step - loss: 0.4152 - accuracy:
0.8787 - val_loss: 0.5964 - val_accuracy: 0.8304
Epoch 17/100
125/125 [==============================] - 15s 119ms/step - loss: 0.4014 - accuracy:
0.8828 - val_loss: 0.5807 - val_accuracy: 0.8361
Epoch 18/100
125/125 [==============================] - 15s 120ms/step - loss: 0.3891 - accuracy:
0.8863 - val_loss: 0.5703 - val_accuracy: 0.8384
Epoch 19/100
125/125 [==============================] - 15s 120ms/step - loss: 0.3763 - accuracy:
```

```
0.8901 - val_loss: 0.5681 - val_accuracy: 0.8393
Epoch 20/100
125/125 [==============================] - 15s 120ms/step - loss: 0.3647 - accuracy: 0.8936 - val_loss: 0.5677 - val_accuracy: 0.8423
Epoch 21/100
125/125 [==============================] - 15s 118ms/step - loss: 0.3532 - accuracy: 0.8975 - val_loss: 0.5673 - val_accuracy: 0.8424
Epoch 22/100
125/125 [==============================] - 15s 118ms/step - loss: 0.3431 - accuracy: 0.9002 - val_loss: 0.5613 - val_accuracy: 0.8436
Epoch 23/100
125/125 [==============================] - 15s 120ms/step - loss: 0.3324 - accuracy: 0.9030 - val_loss: 0.5626 - val_accuracy: 0.8461
Epoch 24/100
125/125 [==============================] - 15s 119ms/step - loss: 0.3227 - accuracy: 0.9063 - val_loss: 0.5636 - val_accuracy: 0.8448
Epoch 25/100
125/125 [==============================] - 15s 119ms/step - loss: 0.3134 - accuracy: 0.9090 - val_loss: 0.5599 - val_accuracy: 0.8460
Epoch 26/100
125/125 [==============================] - 15s 121ms/step - loss: 0.3038 - accuracy: 0.9116 - val_loss: 0.5643 - val_accuracy: 0.8466
Epoch 27/100
125/125 [==============================] - 15s 120ms/step - loss: 0.2950 - accuracy: 0.9142 - val_loss: 0.5673 - val_accuracy: 0.8456
Epoch 28/100
125/125 [==============================] - 15s 119ms/step - loss: 0.2864 - accuracy: 0.9168 - val_loss: 0.5668 - val_accuracy: 0.8473
Epoch 29/100
125/125 [==============================] - 15s 121ms/step - loss: 0.2785 - accuracy: 0.9190 - val_loss: 0.5706 - val_accuracy: 0.8477
Epoch 30/100
125/125 [==============================] - 15s 119ms/step - loss: 0.2711 - accuracy: 0.9209 - val_loss: 0.5706 - val_accuracy: 0.8484
Epoch 31/100
125/125 [==============================] - 15s 119ms/step - loss: 0.2632 - accuracy: 0.9234 - val_loss: 0.5759 - val_accuracy: 0.8473
Epoch 32/100
125/125 [==============================] - 15s 120ms/step - loss: 0.2558 - accuracy: 0.9258 - val_loss: 0.5783 - val_accuracy: 0.8492
Epoch 33/100
125/125 [==============================] - 15s 121ms/step - loss: 0.2486 - accuracy: 0.9280 - val_loss: 0.5820 - val_accuracy: 0.8479
Epoch 34/100
125/125 [==============================] - 19s 149ms/step - loss: 0.2419 - accuracy: 0.9299 - val_loss: 0.5830 - val_accuracy: 0.8483
Epoch 35/100
125/125 [==============================] - 16s 131ms/step - loss: 0.2350 - accuracy: 0.9316 - val_loss: 0.5889 - val_accuracy: 0.8479
```

```
WARNING:absl:Found untraced functions such as lstm_cell_1_layer_call_fn, lstm_cell_1_layer_call_and_return_conditional_losses, lstm_cell_2_layer_call_fn, lstm_cell_2_layer_call_and_return_conditional_losses while saving (showing 4 of 4). These functions will not be directly callable after loading.
INFO:tensorflow:Assets written to: s2s\assets
```

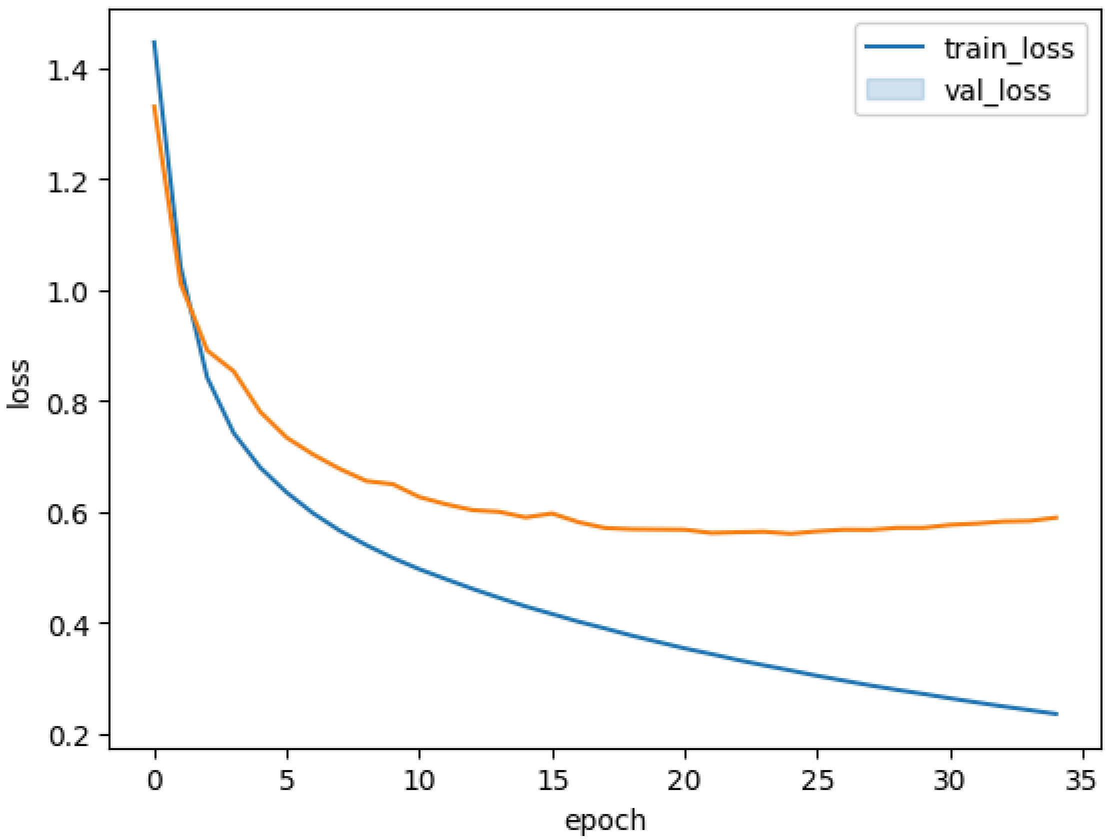

```
INFO:tensorflow:Assets written to: s2s\assets
```

Evaluate model fit

In [21]:
```
#convert history to dataframe for easier plotting
train_history = pd.DataFrame(history.history)
train_history['epoch'] = history.epoch
```

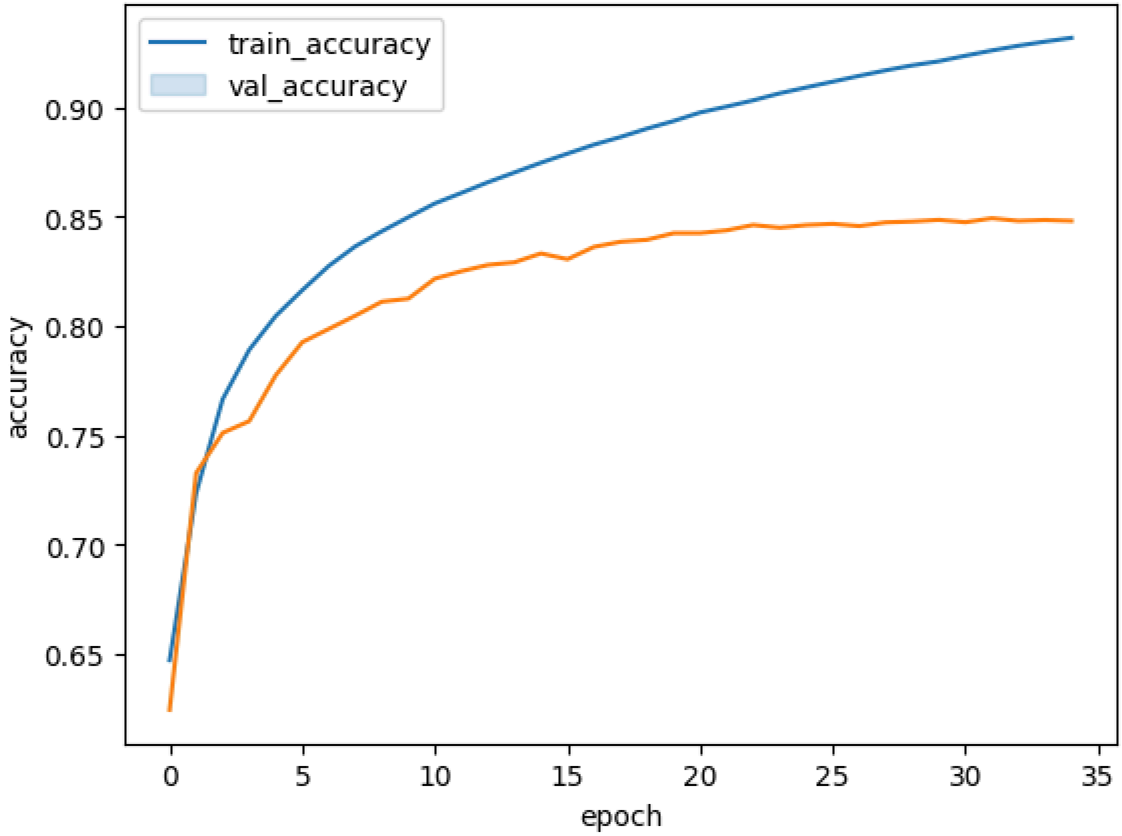
In [22]:
```
#train and validation loss
sns.lineplot(x='epoch', y ='loss', data =train_history)
sns.lineplot(x='epoch', y ='val_loss', data =train_history)
plt.legend(labels=['train_loss', 'val_loss'])
```

Out[22]: <matplotlib.legend.Legend at 0x20415e4d250>



In [42]:
```
#train and validation accuracy
sns.lineplot(x='epoch', y ='accuracy', data =train_history)
sns.lineplot(x='epoch', y ='val_accuracy', data =train_history)
plt.legend(labels=['train_accuracy', 'val_accuracy'])
```

Out[42]: <matplotlib.legend.Legend at 0x20410b59610>



The validation results level off more quickly than the training data, while the training loss and accuracy are quite good, suggesting that the model overfits the data as the model does not perform nearly as well on new data.

Run the Inference

- load the trained model to utilize the weights
- create a new encoder and decoder for the inference model in which the decoder input is offset by one index value
- Use an inverse transformation to map the indices to the corresponding characters

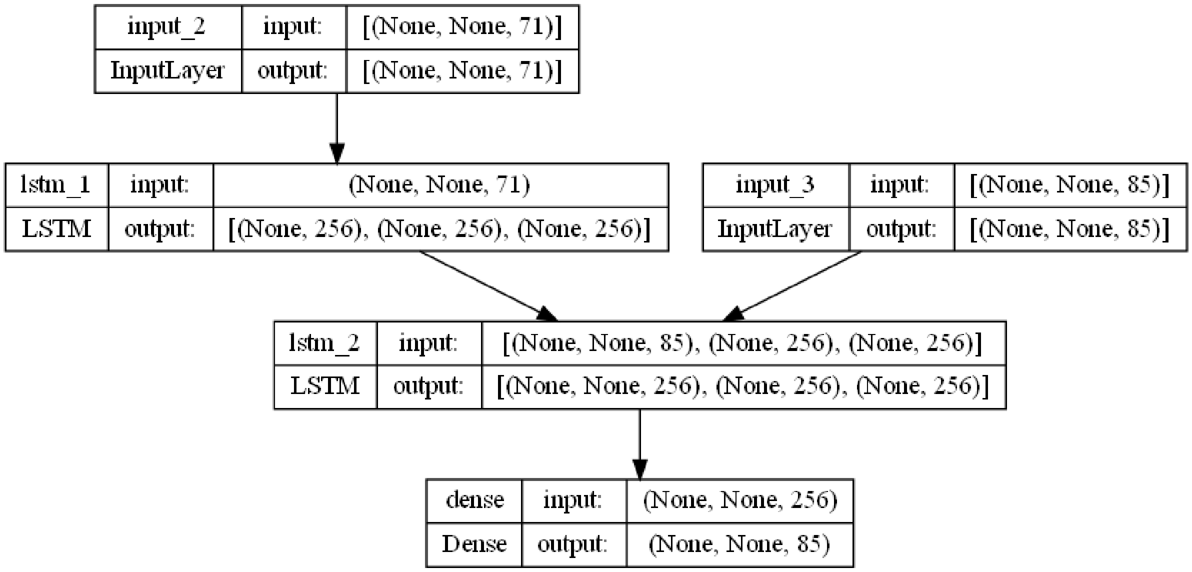
In [ ]:

```
#load the model
model = keras.models.load_model("s2s")
```

In [46]:

```
tf.keras.utils.plot_model(model, show_shapes=True)
```

Out[46]:



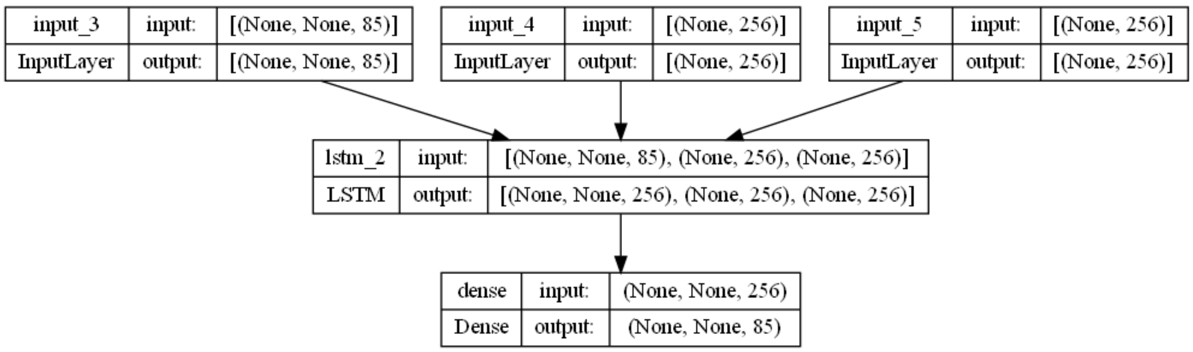
In [36]:

```
# Define the inference models
# Restore the model and construct the encoder and decoder.
model = keras.models.load_model("s2s")

# Set up the encoder model
encoder_inputs = model.input[0]  # input_1
encoder_outputs, state_h_enc, state_c_enc = model.layers[2].output  # lstm_1
encoder_states = [state_h_enc, state_c_enc]
encoder_model = keras.Model(encoder_inputs, encoder_states)

# Set up the decoder layer
decoder_inputs = model.input[1]  # input_2
# Set up the inputs for the hidden states
decoder_state_input_h = keras.Input(shape=(latent_dim,), name="input_4")
decoder_state_input_c = keras.Input(shape=(latent_dim,), name="input_5")
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_lstm = model.layers[3]
decoder_outputs, state_h_dec, state_c_dec = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs
)
decoder_states = [state_h_dec, state_c_dec]
decoder_dense = model.layers[4]
decoder_outputs = decoder_dense(decoder_outputs)

# The decoder model with inputs and states inputs with the context info
# The decoder model with outputs = decoder outputs and states with the context info
decoder_model = keras.Model(
    [decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decoder_states
)
```

In [47]:

```
tf.keras.utils.plot_model(decoder_model, show_shapes=True)
```

Out[47]:



In [37]:

```
#inverse transformation: map indices to text characters
reverse_input_char_index = dict((i, char) for char, i in input_token_index.items())
reverse_target_char_index = dict((i, char) for char, i in target_token_index.items(
```

In [38]:

```
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the start character.
    target_seq[0, 0, target_token_index["\t"]] = 1.0

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
    stop_condition = False
    decoded_sentence = ""
    while not stop_condition:
        # forcast the outputs and hidden states
        output_tokens, h, c = decoder_model.predict([target_seq] + states_value)
        # Get the indices of the non-empty character
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        # Map the indices to the corresponding character
        sampled_char = reverse_target_char_index[sampled_token_index]
        # Concatenate the predicted char
        decoded_sentence += sampled_char

        # Exit condition: either hit max length
        # or find stop character.
        if sampled_char == "\n" or len(decoded_sentence) > max_decoder_seq_length:
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.0

        # Update states
        states_value = [h, c]
    return decoded_sentence
```

View example decoding

In [39]:

```python
for seq_index in range(10):
    # Take one sequence (part of the training set)
    # for trying out decoding.
    input_seq = encoder_input_data[seq_index : seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print("-")
    print("Input sentence:", input_texts[seq_index])
    print("Decoded sentence:", decoded_sentence)
```

```
1/1 [==============================] - 0s 195ms/step
1/1 [==============================] - 0s 167ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
-
Input sentence: Go.
Decoded sentence: Verzieh dich!

1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
-
Input sentence: Hi.
Decoded sentence: Wack!

1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 18ms/step
-
Input sentence: Hi.
Decoded sentence: Wack!

1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
```

```
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
-
Input sentence: Run!
Decoded sentence: Wachen Sie bit zu schinden.

1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
-
Input sentence: Run.
Decoded sentence: Lauf mich!

1/1 [==============================] - 0s 10ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
-
Input sentence: Wow!
Decoded sentence: Warte!

1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
-
Input sentence: Wow!
```

```
Decoded sentence: Warte!

1/1 [==============================] - 0s 10ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
-
Input sentence: Fire!
Decoded sentence: Verguffen!

1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
-
Input sentence: Help!
Decoded sentence: Hallt!

1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
1/1 [==============================] - 0s 11ms/step
1/1 [==============================] - 0s 12ms/step
-
Input sentence: Help!
Decoded sentence: Hallt!
```