Character-level recurrent sequence-to-sequence model

Author: fchollet
Date created: 2017/09/29
Last modified: 2023/11/22
Description: Character-level recurrent sequence-to-sequence model.

ⓘ This example uses Keras 3

View in Colab • GitHub source

Introduction

This example demonstrates how to implement a basic character-level recurrent sequence-to-sequence model. We apply it to translating short English sentences into short French sentences, character-by-character. Note that it is fairly unusual to do character-level machine translation, as word-level models are more common in this domain.

Summary of the algorithm

- We start with input sequences from a domain (e.g. English sentences) and corresponding target sequences from another domain (e.g. French sentences).
- An encoder LSTM turns input sequences to 2 state vectors (we keep the last LSTM state and discard the outputs).
- A decoder LSTM is trained to turn the target sequences into the same sequence but offset by one timestep in the future, a training process called "teacher forcing" in this context. It uses as initial state the state vectors from the encoder. Effectively, the decoder learns to generate `targets[t+1...]` given `targets[...t]`, conditioned on the input sequence.
- In inference mode, when we want to decode unknown input sequences, we: - Encode the input sequence into state vectors - Start with a target sequence of size 1 (just the start-of-sequence character) - Feed the state vectors and 1-char target sequence to the decoder to produce predictions for the next character - Sample the next character using these predictions (we simply use argmax). - Append the sampled character to the target sequence - Repeat until we generate the end-of-sequence character or we hit the character limit.

Setup

```
import numpy as np
import keras
import os
from pathlib import Path
```

Download the data

```
fpath = keras.utils.get_file(origin="http://www.manythings.org/anki/fra-eng.zip")
dirpath = Path(fpath).parent.absolute()
os.system(f"unzip -q {fpath} -d {dirpath}")
```

```
0
```

```
batch_size = 64  # Batch size for training.
epochs = 100  # Number of epochs to train for.
latent_dim = 256  # Latent dimensionality of the encoding space.
num_samples = 10000  # Number of samples to train on.
# Path to the data txt file on disk.
data_path = os.path.join(dirpath, "fra.txt")
```

```python
# Vectorize the data.
input_texts = []
target_texts = []
input_characters = set()
target_characters = set()
with open(data_path, "r", encoding="utf-8") as f:
    lines = f.read().split("\n")
for line in lines[: min(num_samples, len(lines) - 1)]:
    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)
    for char in input_text:
        if char not in input_characters:
            input_characters.add(char)
    for char in target_text:
        if char not in target_characters:
            target_characters.add(char)

input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)
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("Number of samples:", len(input_texts))
print("Number of unique input tokens:", num_encoder_tokens)
print("Number of unique output tokens:", num_decoder_tokens)
print("Max sequence length for inputs:", max_encoder_seq_length)
print("Max sequence length for outputs:", max_decoder_seq_length)

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)])

encoder_input_data = np.zeros(
    (len(input_texts), max_encoder_seq_length, num_encoder_tokens),
    dtype="float32",
)
decoder_input_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype="float32",
)
decoder_target_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype="float32",
)

for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.0
    encoder_input_data[i, t + 1 :, input_token_index[" "]] = 1.0
    for t, char in enumerate(target_text):
        # 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
    decoder_input_data[i, t + 1 :, target_token_index[" "]] = 1.0
    decoder_target_data[i, t:, target_token_index[" "]] = 1.0
```

```
Number of unique output tokens: 93
Max sequence length for inputs: 14
Max sequence length for outputs: 59
```

Build the model

```
# Define an input sequence and process it.
encoder_inputs = keras.Input(shape=(None, num_encoder_tokens))
encoder = keras.layers.LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)

# We discard `encoder_outputs` and only keep the states.
encoder_states = [state_h, state_c]

# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = keras.Input(shape=(None, num_decoder_tokens))

# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in inference.
decoder_lstm = keras.layers.LSTM(latent_dim, return_sequences=True, return_state=True)
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)

# Define the model that will turn
# `encoder_input_data` & `decoder_input_data` into `decoder_target_data`
model = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

Train the model

```
model.compile(
    optimizer="rmsprop", loss="categorical_crossentropy", metrics=["accuracy"]
)
model.fit(
    [encoder_input_data, decoder_input_data],
    decoder_target_data,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.2,
)
# Save model
model.save("s2s_model.keras")
```

```
1.5465 - val_accuracy: 0.7138 - val_loss: 1.0745
Epoch 2/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.7470 - loss:
0.9546 - val_accuracy: 0.7188 - val_loss: 1.0219
Epoch 3/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.7590 - loss:
0.8659 - val_accuracy: 0.7482 - val_loss: 0.8677
Epoch 4/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.7878 - loss:
0.7588 - val_accuracy: 0.7744 - val_loss: 0.7864
Epoch 5/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.7957 - loss:
0.7092 - val_accuracy: 0.7904 - val_loss: 0.7256
Epoch 6/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.8151 - loss:
0.6375 - val_accuracy: 0.8003 - val_loss: 0.6926
Epoch 7/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.8217 - loss:
0.6095 - val_accuracy: 0.8081 - val_loss: 0.6633
Epoch 8/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8299 - loss:
0.5818 - val_accuracy: 0.8146 - val_loss: 0.6355
Epoch 9/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8346 - loss:
0.5632 - val_accuracy: 0.8179 - val_loss: 0.6285
Epoch 10/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8378 - loss:
0.5496 - val_accuracy: 0.8233 - val_loss: 0.6056
Epoch 11/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8450 - loss:
0.5301 - val_accuracy: 0.8300 - val_loss: 0.5913
Epoch 12/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8487 - loss:
0.5148 - val_accuracy: 0.8324 - val_loss: 0.5805
Epoch 13/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8537 - loss:
0.4996 - val_accuracy: 0.8354 - val_loss: 0.5718
Epoch 14/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8570 - loss:
0.4874 - val_accuracy: 0.8388 - val_loss: 0.5535
Epoch 15/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8603 - loss:
0.4749 - val_accuracy: 0.8428 - val_loss: 0.5451
Epoch 16/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8636 - loss:
0.4642 - val_accuracy: 0.8448 - val_loss: 0.5332
Epoch 17/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8658 - loss:
0.4551 - val_accuracy: 0.8473 - val_loss: 0.5260
Epoch 18/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8689 - loss:
0.4443 - val_accuracy: 0.8465 - val_loss: 0.5236
Epoch 19/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8711 - loss:
0.4363 - val_accuracy: 0.8531 - val_loss: 0.5078
Epoch 20/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8731 - loss:
0.4285 - val_accuracy: 0.8508 - val_loss: 0.5121
Epoch 21/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8759 - loss:
0.4180 - val_accuracy: 0.8546 - val_loss: 0.5005
Epoch 22/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8788 - loss:
0.4075 - val_accuracy: 0.8550 - val_loss: 0.4981
Epoch 23/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8799 - loss:
0.4043 - val_accuracy: 0.8563 - val_loss: 0.4918
Epoch 24/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8820 - loss:
```

```
0.3927 - val_accuracy: 0.8605 - val_loss: 0.4794
Epoch 26/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8852 - loss: 
0.3862 - val_accuracy: 0.8607 - val_loss: 0.4784
Epoch 27/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8877 - loss: 
0.3767 - val_accuracy: 0.8616 - val_loss: 0.4753
Epoch 28/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8890 - loss: 
0.3730 - val_accuracy: 0.8633 - val_loss: 0.4685
Epoch 29/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8897 - loss: 
0.3695 - val_accuracy: 0.8633 - val_loss: 0.4685
Epoch 30/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8924 - loss: 
0.3604 - val_accuracy: 0.8648 - val_loss: 0.4664
Epoch 31/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8946 - loss: 
0.3538 - val_accuracy: 0.8658 - val_loss: 0.4613
Epoch 32/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8948 - loss: 
0.3526 - val_accuracy: 0.8668 - val_loss: 0.4618
Epoch 33/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8972 - loss: 
0.3442 - val_accuracy: 0.8662 - val_loss: 0.4597
Epoch 34/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8969 - loss: 
0.3435 - val_accuracy: 0.8672 - val_loss: 0.4594
Epoch 35/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8996 - loss: 
0.3364 - val_accuracy: 0.8673 - val_loss: 0.4569
Epoch 36/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9003 - loss: 
0.3340 - val_accuracy: 0.8677 - val_loss: 0.4601
Epoch 37/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9024 - loss: 
0.3260 - val_accuracy: 0.8671 - val_loss: 0.4569
Epoch 38/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9048 - loss: 
0.3200 - val_accuracy: 0.8685 - val_loss: 0.4540
Epoch 39/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9051 - loss: 
0.3187 - val_accuracy: 0.8692 - val_loss: 0.4545
Epoch 40/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9071 - loss: 
0.3119 - val_accuracy: 0.8708 - val_loss: 0.4490
Epoch 41/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9085 - loss: 
0.3064 - val_accuracy: 0.8706 - val_loss: 0.4506
Epoch 42/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9092 - loss: 
0.3061 - val_accuracy: 0.8711 - val_loss: 0.4484
Epoch 43/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9100 - loss: 
0.3011 - val_accuracy: 0.8718 - val_loss: 0.4485
Epoch 44/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9101 - loss: 
0.3007 - val_accuracy: 0.8716 - val_loss: 0.4509
Epoch 45/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9126 - loss: 
0.2920 - val_accuracy: 0.8723 - val_loss: 0.4474
Epoch 46/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9144 - loss: 
0.2881 - val_accuracy: 0.8714 - val_loss: 0.4505
Epoch 47/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9155 - loss: 
0.2829 - val_accuracy: 0.8727 - val_loss: 0.4487
Epoch 48/100
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9174 - loss: 0.2763 - val_accuracy: 0.8739 - val_loss: 0.4454
Epoch 50/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9188 - loss: 0.2706 - val_accuracy: 0.8738 - val_loss: 0.4473
Epoch 51/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9199 - loss: 0.2682 - val_accuracy: 0.8716 - val_loss: 0.4542
Epoch 52/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9202 - loss: 0.2665 - val_accuracy: 0.8725 - val_loss: 0.4533
Epoch 53/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9228 - loss: 0.2579 - val_accuracy: 0.8735 - val_loss: 0.4485
Epoch 54/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9230 - loss: 0.2580 - val_accuracy: 0.8735 - val_loss: 0.4507
Epoch 55/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9237 - loss: 0.2546 - val_accuracy: 0.8737 - val_loss: 0.4579
Epoch 56/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9253 - loss: 0.2482 - val_accuracy: 0.8749 - val_loss: 0.4496
Epoch 57/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9264 - loss: 0.2448 - val_accuracy: 0.8755 - val_loss: 0.4503
Epoch 58/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9271 - loss: 0.2426 - val_accuracy: 0.8747 - val_loss: 0.4526
Epoch 59/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9289 - loss: 0.2380 - val_accuracy: 0.8750 - val_loss: 0.4543
Epoch 60/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9292 - loss: 0.2358 - val_accuracy: 0.8745 - val_loss: 0.4563
Epoch 61/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9297 - loss: 0.2339 - val_accuracy: 0.8750 - val_loss: 0.4555
Epoch 62/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9308 - loss: 0.2299 - val_accuracy: 0.8741 - val_loss: 0.4590
Epoch 63/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9324 - loss: 0.2259 - val_accuracy: 0.8761 - val_loss: 0.4611
Epoch 64/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9329 - loss: 0.2247 - val_accuracy: 0.8751 - val_loss: 0.4608
Epoch 65/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9344 - loss: 0.2187 - val_accuracy: 0.8756 - val_loss: 0.4628
Epoch 66/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9354 - loss: 0.2156 - val_accuracy: 0.8750 - val_loss: 0.4664
Epoch 67/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9360 - loss: 0.2136 - val_accuracy: 0.8751 - val_loss: 0.4665
Epoch 68/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9370 - loss: 0.2093 - val_accuracy: 0.8751 - val_loss: 0.4688
Epoch 69/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9385 - loss: 0.2057 - val_accuracy: 0.8747 - val_loss: 0.4757
Epoch 70/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9388 - loss: 0.2039 - val_accuracy: 0.8752 - val_loss: 0.4748
Epoch 71/100
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9393 - loss: 0.2020 - val_accuracy: 0.8749 - val_loss: 0.4749
```

```
Epoch 73/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9417 - loss:
0.1946 - val_accuracy: 0.8752 - val_loss: 0.4774
Epoch 74/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9427 - loss:
0.1911 - val_accuracy: 0.8746 - val_loss: 0.4809
Epoch 75/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9430 - loss:
0.1900 - val_accuracy: 0.8746 - val_loss: 0.4809
Epoch 76/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9443 - loss:
0.1856 - val_accuracy: 0.8749 - val_loss: 0.4836
Epoch 77/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9438 - loss:
0.1867 - val_accuracy: 0.8759 - val_loss: 0.4866
Epoch 78/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9454 - loss:
0.1811 - val_accuracy: 0.8751 - val_loss: 0.4869
Epoch 79/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9462 - loss:
0.1788 - val_accuracy: 0.8767 - val_loss: 0.4899
Epoch 80/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9467 - loss:
0.1777 - val_accuracy: 0.8754 - val_loss: 0.4932
Epoch 81/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9474 - loss:
0.1748 - val_accuracy: 0.8758 - val_loss: 0.4932
Epoch 82/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9481 - loss:
0.1731 - val_accuracy: 0.8751 - val_loss: 0.5027
Epoch 83/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9484 - loss:
0.1708 - val_accuracy: 0.8748 - val_loss: 0.5012
Epoch 84/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9491 - loss:
0.1675 - val_accuracy: 0.8748 - val_loss: 0.5091
Epoch 85/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9514 - loss:
0.1624 - val_accuracy: 0.8744 - val_loss: 0.5082
Epoch 86/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9508 - loss:
0.1627 - val_accuracy: 0.8733 - val_loss: 0.5159
Epoch 87/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9517 - loss:
0.1606 - val_accuracy: 0.8749 - val_loss: 0.5139
Epoch 88/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9519 - loss:
0.1579 - val_accuracy: 0.8746 - val_loss: 0.5189
Epoch 89/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9526 - loss:
0.1565 - val_accuracy: 0.8752 - val_loss: 0.5171
Epoch 90/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9531 - loss:
0.1549 - val_accuracy: 0.8750 - val_loss: 0.5169
Epoch 91/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9543 - loss:
0.1506 - val_accuracy: 0.8740 - val_loss: 0.5182
Epoch 92/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9547 - loss:
0.1497 - val_accuracy: 0.8752 - val_loss: 0.5207
Epoch 93/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9554 - loss:
0.1471 - val_accuracy: 0.8750 - val_loss: 0.5293
Epoch 94/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9560 - loss:
0.1467 - val_accuracy: 0.8749 - val_loss: 0.5298
Epoch 95/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9563 - loss:
```

```
0.1421 - val_accuracy: 0.8728 - val_loss: 0.5391
Epoch 97/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9577 - loss:
0.1390 - val_accuracy: 0.8755 - val_loss: 0.5318
Epoch 98/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9583 - loss:
0.1375 - val_accuracy: 0.8744 - val_loss: 0.5433
Epoch 99/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9591 - loss:
0.1363 - val_accuracy: 0.8746 - val_loss: 0.5359
Epoch 100/100
 125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9592 - loss:
0.1351 - val_accuracy: 0.8738 - val_loss: 0.5482
```

Run inference (sampling)

1. encode input and retrieve initial decoder state
2. run one step of decoder with this initial state and a "start of sequence" token as target. Output will be the next target token.
3. Repeat with the current target token and current states

```
model = keras.models.load_model("s2s_model.keras")

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)

decoder_inputs = model.input[1]  # input_2
decoder_state_input_h = keras.Input(shape=(latent_dim,))
decoder_state_input_c = keras.Input(shape=(latent_dim,))
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)
decoder_model = keras.Model(
    [decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decoder_states
)

# Reverse-lookup token index to decode sequences back to
# something readable.
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())


def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq, verbose=0)

    # 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:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value, verbose=0
        )

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        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
```

You can now generate decoded sentences as such:

```
    # 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)
```

```
-
Input sentence: Go.
Decoded sentence: Va !
```

```
-
Input sentence: Go.
Decoded sentence: Va !
```

```
-
Input sentence: Go.
Decoded sentence: Va !
```

```
-
Input sentence: Go.
Decoded sentence: Va !
```

```
-
Input sentence: Hi.
Decoded sentence: Salut.
```

```
-
Input sentence: Hi.
Decoded sentence: Salut.
```

```
-
Input sentence: Run!
Decoded sentence: Fuyez !
```

```
-
Input sentence: Run!
Decoded sentence: Fuyez !
```

```
-
Input sentence: Run!
Decoded sentence: Fuyez !
```

```
-
Input sentence: Run!
Decoded sentence: Fuyez !
```

```
-
Input sentence: Run!
Decoded sentence: Fuyez !
```

```
Decoded sentence: Fuyez !
```

```
-
Input sentence: Run!
Decoded sentence: Fuyez !
```

```
-
Input sentence: Run!
Decoded sentence: Fuyez !
```

```
-
Input sentence: Run.
Decoded sentence: Courez !
```

```
-
Input sentence: Run.
Decoded sentence: Courez !
```

```
-
Input sentence: Run.
Decoded sentence: Courez !
```

```
-
Input sentence: Run.
Decoded sentence: Courez !
```

```
-
Input sentence: Run.
Decoded sentence: Courez !
```

```
-
Input sentence: Run.
Decoded sentence: Courez !
```