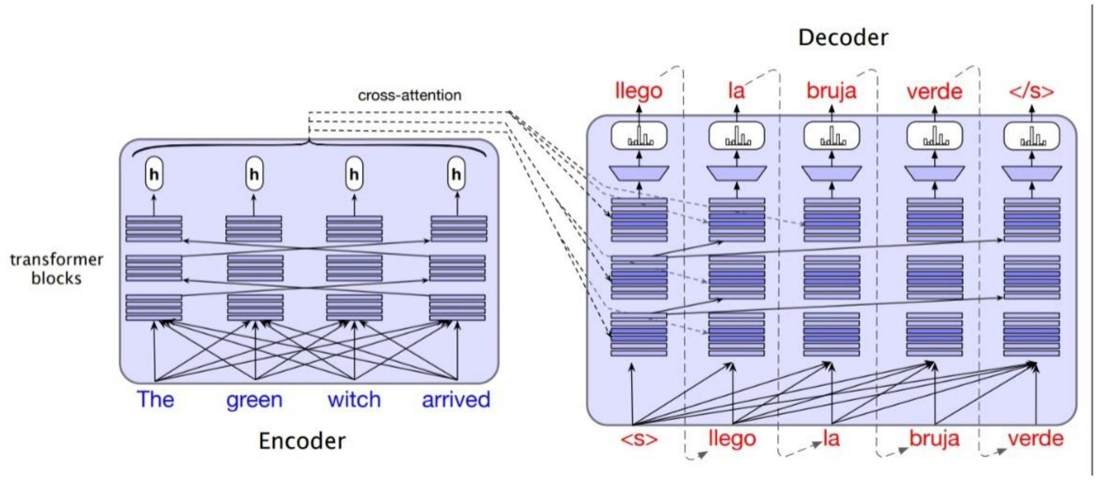
Aim: - Implement language translator using encoder decoder model.

Requirement: - Python versions 3.7, 3.8, 3.9, 3.10 or 3.11, jupyter notebook

Theory: -

The encoder-decoder architecture can also be implemented using transformers (rather than RNN/LSTMs) as the component modules. It consists of an encoder that takes the source language input words X = x1,...,x_T and maps them to an output representation H^{enc} = h1,...,h_T ; usually via N = 6 stacked encoder blocks. The decoder, just like the encoder-decoder RNN, is essentially a conditional language model that attends to the encoder representation and generates the target words one by one, at each timestep conditioning on the source sentence and the previously generated target language words.



But the components of the architecture differ somewhat from the RNN and also from the transformer block we've seen. First, in order to attend to the source language, the transformer blocks in the decoder has an extra cross-attention layer.

The decoder transformer block includes an cross-attention extra layer with a special kind of attention, cross-attention (also sometimes called encoder-decoder attention orsource attention). Cross-attention has the same form as the multi-headed self-attention in a normal transformer block, except that while the queries as usual come from the previous layer of the decoder, the keys and values come from the output of the encoder.

That is, the final output of the encoder $H^{enc} = h1,...,ht$ is multiplied by the cross-attention layer's key weights W^K and value weights W^V, but the output from the prior decoder layer $H^{dec[i-1]}$ is multiplied by the cross-attention layer's query weights W^Q:

$$\mathbf{Q} = \mathbf{W}^{\mathbf{Q}}\mathbf{H}^{dec[i-1]}; \;\; \mathbf{K} = \mathbf{W}^{\mathbf{K}}\mathbf{H}^{enc}; \;\; \mathbf{V} = \mathbf{W}^{\mathbf{V}}\mathbf{H}^{enc}$$

$$CrossAttention(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\intercal}}{\sqrt{d_k}}\right)\mathbf{V}$$

The cross attention thus allows the decoder to attend to each of the source language words as projected into the the entire encoder final output representations. The other attention layer in each decoder block, the self-attention layer, is the same causal (left to-right) self-attention. The self-attention in the encoder, however, is allowed to look ahead at the entire source language text. In training, just as for RNN encoder-decoders, we use teacher forcing, and train autoregressively, at each time step predicting the next token in the target language, using cross-entropy loss.

Implement language translator using encoder decoder model

```python
# This Python 3 environment comes with many helpful analytics libraries i
nstalled
# It is defined by the kaggle/python Docker image: https://github.com/kag
gle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) wil
l list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 5GB to the current directory (/kaggle/working/) tha
t gets preserved as output when you create a version using "Save & Run Al
l"
# You can also write temporary files to /kaggle/temp/, but they won't be
saved outside of the current session
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,LSTM,Dense

batch_size=64
epochs=100
latent_dim=256 # here latent dim represent hidden state or cell state
num_samples=10000

data_path='../input/french-english-translated-words-and-phrases/fra.txt
'
# 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)
```

In [4]:

```
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])
```

In [5]:

```
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)
```

```
Number of samples: 10000
Number of unique input tokens: 71
Number of unique output tokens: 93
Max sequence length for inputs: 15
Max sequence length for outputs: 59
```

In [6]:

```
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)])
```

In [7]:

```
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')
```

In [8]:

```
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.
    encoder_input_data[i, t + 1:, input_token_index[' ']] = 1.
    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.
        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.
    decoder_input_data[i, t + 1:, target_token_index[' ']] = 1.
    decoder_target_data[i, t:, target_token_index[' ']] = 1.
```

Defining the encoder and decoder

In [9]:

```
# Define an input sequence and process it.
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = 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 = 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 = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

In [10]:

linkcode

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

# Run training
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)
```

```
Epoch 1/100
125/125 [==============================] - 2s 19ms/step - loss: 1.1462
- accuracy: 0.7361 - val_loss: 1.0484 - val_accuracy: 0.7116
Epoch 2/100
125/125 [==============================] - 2s 13ms/step - loss: 0.8173
- accuracy: 0.7788 - val_loss: 0.8203 - val_accuracy: 0.7729
Epoch 3/100
125/125 [==============================] - 2s 13ms/step - loss: 0.6520
- accuracy: 0.8167 - val_loss: 0.6947 - val_accuracy: 0.7982
Epoch 4/100
125/125 [==============================] - 2s 14ms/step - loss: 0.5682
- accuracy: 0.8354 - val_loss: 0.6350 - val_accuracy: 0.8160
```

```
Epoch 5/100
125/125 [==============================] - 2s 13ms/step - loss: 0.5201
- accuracy: 0.8479 - val_loss: 0.5817 - val_accuracy: 0.8315
Epoch 6/100
125/125 [==============================] - 2s 13ms/step - loss: 0.4851
- accuracy: 0.8575 - val_loss: 0.5506 - val_accuracy: 0.8408
Epoch 7/100
125/125 [==============================] - 2s 14ms/step - loss: 0.4581
- accuracy: 0.8646 - val_loss: 0.5300 - val_accuracy: 0.8448
Epoch 8/100
125/125 [==============================] - 2s 13ms/step - loss: 0.4355
- accuracy: 0.8706 - val_loss: 0.5217 - val_accuracy: 0.8469
Epoch 9/100
125/125 [==============================] - 2s 13ms/step - loss: 0.4153
- accuracy: 0.8758 - val_loss: 0.4983 - val_accuracy: 0.8533
Epoch 10/100
125/125 [==============================] - 2s 14ms/step - loss: 0.3968
- accuracy: 0.8811 - val_loss: 0.4859 - val_accuracy: 0.8563
Epoch 11/100
125/125 [==============================] - 2s 13ms/step - loss: 0.3797
- accuracy: 0.8861 - val_loss: 0.4781 - val_accuracy: 0.8591
Epoch 12/100
125/125 [==============================] - 2s 13ms/step - loss: 0.3646
- accuracy: 0.8906 - val_loss: 0.4711 - val_accuracy: 0.8605
Epoch 13/100
125/125 [==============================] - 2s 13ms/step - loss: 0.3498
- accuracy: 0.8950 - val_loss: 0.4620 - val_accuracy: 0.8644
Epoch 14/100
125/125 [==============================] - 2s 13ms/step - loss: 0.3364
- accuracy: 0.8985 - val_loss: 0.4546 - val_accuracy: 0.8662
Epoch 15/100
125/125 [==============================] - 2s 13ms/step - loss: 0.3234
- accuracy: 0.9023 - val_loss: 0.4505 - val_accuracy: 0.8682
Epoch 16/100
125/125 [==============================] - 2s 13ms/step - loss: 0.3117
- accuracy: 0.9057 - val_loss: 0.4502 - val_accuracy: 0.8686
Epoch 17/100
125/125 [==============================] - 2s 14ms/step - loss: 0.3003
- accuracy: 0.9096 - val_loss: 0.4473 - val_accuracy: 0.8703
Epoch 18/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2893
- accuracy: 0.9125 - val_loss: 0.4471 - val_accuracy: 0.8712
Epoch 19/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2790
- accuracy: 0.9156 - val_loss: 0.4438 - val_accuracy: 0.8733
Epoch 20/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2690
- accuracy: 0.9191 - val_loss: 0.4468 - val_accuracy: 0.8730
Epoch 21/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2597
- accuracy: 0.9214 - val_loss: 0.4431 - val_accuracy: 0.8742
```

```
Epoch 22/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2508
- accuracy: 0.9242 - val_loss: 0.4457 - val_accuracy: 0.8738
Epoch 23/100
125/125 [==============================] - 2s 14ms/step - loss: 0.2423
- accuracy: 0.9266 - val_loss: 0.4506 - val_accuracy: 0.8742
Epoch 24/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2343
- accuracy: 0.9289 - val_loss: 0.4500 - val_accuracy: 0.8750
Epoch 25/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2263
- accuracy: 0.9314 - val_loss: 0.4526 - val_accuracy: 0.8748
Epoch 26/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2188
- accuracy: 0.9338 - val_loss: 0.4538 - val_accuracy: 0.8749
Epoch 27/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2119
- accuracy: 0.9355 - val_loss: 0.4540 - val_accuracy: 0.8757
Epoch 28/100
125/125 [==============================] - 2s 13ms/step - loss: 0.2050
- accuracy: 0.9375 - val_loss: 0.4651 - val_accuracy: 0.8736
Epoch 29/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1986
- accuracy: 0.9397 - val_loss: 0.4678 - val_accuracy: 0.8751
Epoch 30/100
125/125 [==============================] - 2s 14ms/step - loss: 0.1928
- accuracy: 0.9413 - val_loss: 0.4667 - val_accuracy: 0.8753
Epoch 31/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1866
- accuracy: 0.9433 - val_loss: 0.4732 - val_accuracy: 0.8756
Epoch 32/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1809
- accuracy: 0.9447 - val_loss: 0.4750 - val_accuracy: 0.8758
Epoch 33/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1753
- accuracy: 0.9465 - val_loss: 0.4822 - val_accuracy: 0.8743
Epoch 34/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1703
- accuracy: 0.9477 - val_loss: 0.4819 - val_accuracy: 0.8752
Epoch 35/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1651
- accuracy: 0.9495 - val_loss: 0.4874 - val_accuracy: 0.8757
Epoch 36/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1606
- accuracy: 0.9508 - val_loss: 0.4936 - val_accuracy: 0.8741
Epoch 37/100
125/125 [==============================] - 2s 14ms/step - loss: 0.1560
- accuracy: 0.9522 - val_loss: 0.5039 - val_accuracy: 0.8731
Epoch 38/100
125/125 [==============================] - 2s 14ms/step - loss: 0.1515
- accuracy: 0.9538 - val_loss: 0.5008 - val_accuracy: 0.8752
```

```
Epoch 39/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1474
- accuracy: 0.9548 - val_loss: 0.5084 - val_accuracy: 0.8741
Epoch 40/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1429
- accuracy: 0.9561 - val_loss: 0.5140 - val_accuracy: 0.8751
Epoch 41/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1392
- accuracy: 0.9570 - val_loss: 0.5158 - val_accuracy: 0.8751
Epoch 42/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1354
- accuracy: 0.9583 - val_loss: 0.5198 - val_accuracy: 0.8752
Epoch 43/100
125/125 [==============================] - 2s 14ms/step - loss: 0.1319
- accuracy: 0.9592 - val_loss: 0.5249 - val_accuracy: 0.8746
Epoch 44/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1286
- accuracy: 0.9604 - val_loss: 0.5287 - val_accuracy: 0.8739
Epoch 45/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1248
- accuracy: 0.9614 - val_loss: 0.5410 - val_accuracy: 0.8731
Epoch 46/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1218
- accuracy: 0.9625 - val_loss: 0.5446 - val_accuracy: 0.8731
Epoch 47/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1186
- accuracy: 0.9632 - val_loss: 0.5511 - val_accuracy: 0.8734
Epoch 48/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1159
- accuracy: 0.9638 - val_loss: 0.5552 - val_accuracy: 0.8732
Epoch 49/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1131
- accuracy: 0.9647 - val_loss: 0.5620 - val_accuracy: 0.8735
Epoch 50/100
125/125 [==============================] - 2s 14ms/step - loss: 0.1101
- accuracy: 0.9658 - val_loss: 0.5646 - val_accuracy: 0.8732
Epoch 51/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1077
- accuracy: 0.9662 - val_loss: 0.5736 - val_accuracy: 0.8731
Epoch 52/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1049
- accuracy: 0.9670 - val_loss: 0.5736 - val_accuracy: 0.8728
Epoch 53/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1028
- accuracy: 0.9675 - val_loss: 0.5801 - val_accuracy: 0.8728
Epoch 54/100
125/125 [==============================] - 2s 13ms/step - loss: 0.1000
- accuracy: 0.9685 - val_loss: 0.5913 - val_accuracy: 0.8715
Epoch 55/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0979
- accuracy: 0.9689 - val_loss: 0.5913 - val_accuracy: 0.8729
```

```
Epoch 56/100
125/125 [==============================] - 2s 14ms/step - loss: 0.0959
- accuracy: 0.9695 - val_loss: 0.5999 - val_accuracy: 0.8721
Epoch 57/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0935
- accuracy: 0.9704 - val_loss: 0.6031 - val_accuracy: 0.8711
Epoch 58/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0918
- accuracy: 0.9707 - val_loss: 0.6075 - val_accuracy: 0.8713
Epoch 59/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0899
- accuracy: 0.9712 - val_loss: 0.6128 - val_accuracy: 0.8721
Epoch 60/100
125/125 [==============================] - 2s 14ms/step - loss: 0.0878
- accuracy: 0.9719 - val_loss: 0.6186 - val_accuracy: 0.8706
Epoch 61/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0858
- accuracy: 0.9725 - val_loss: 0.6273 - val_accuracy: 0.8714
Epoch 62/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0839
- accuracy: 0.9729 - val_loss: 0.6266 - val_accuracy: 0.8719
Epoch 63/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0824
- accuracy: 0.9734 - val_loss: 0.6288 - val_accuracy: 0.8723
Epoch 64/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0809
- accuracy: 0.9738 - val_loss: 0.6300 - val_accuracy: 0.8715
Epoch 65/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0785
- accuracy: 0.9744 - val_loss: 0.6408 - val_accuracy: 0.8716
Epoch 66/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0777
- accuracy: 0.9748 - val_loss: 0.6417 - val_accuracy: 0.8721
Epoch 67/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0760
- accuracy: 0.9751 - val_loss: 0.6504 - val_accuracy: 0.8703
Epoch 68/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0748
- accuracy: 0.9755 - val_loss: 0.6581 - val_accuracy: 0.8712
Epoch 69/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0731
- accuracy: 0.9761 - val_loss: 0.6598 - val_accuracy: 0.8707
Epoch 70/100
125/125 [==============================] - 2s 14ms/step - loss: 0.0716
- accuracy: 0.9764 - val_loss: 0.6602 - val_accuracy: 0.8710
Epoch 71/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0704
- accuracy: 0.9767 - val_loss: 0.6638 - val_accuracy: 0.8723
Epoch 72/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0697
- accuracy: 0.9770 - val_loss: 0.6683 - val_accuracy: 0.8704
```

```
Epoch 73/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0679
- accuracy: 0.9774 - val_loss: 0.6748 - val_accuracy: 0.8711
Epoch 74/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0665
- accuracy: 0.9778 - val_loss: 0.6747 - val_accuracy: 0.8718
Epoch 75/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0657
- accuracy: 0.9780 - val_loss: 0.6745 - val_accuracy: 0.8718
Epoch 76/100
125/125 [==============================] - 2s 14ms/step - loss: 0.0642
- accuracy: 0.9786 - val_loss: 0.6827 - val_accuracy: 0.8707
Epoch 77/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0634
- accuracy: 0.9788 - val_loss: 0.6823 - val_accuracy: 0.8713
Epoch 78/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0626
- accuracy: 0.9789 - val_loss: 0.6914 - val_accuracy: 0.8704
Epoch 79/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0611
- accuracy: 0.9792 - val_loss: 0.6931 - val_accuracy: 0.8708
Epoch 80/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0600
- accuracy: 0.9796 - val_loss: 0.7026 - val_accuracy: 0.8701
Epoch 81/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0595
- accuracy: 0.9797 - val_loss: 0.6990 - val_accuracy: 0.8695
Epoch 82/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0585
- accuracy: 0.9803 - val_loss: 0.7052 - val_accuracy: 0.8712
Epoch 83/100
125/125 [==============================] - 2s 14ms/step - loss: 0.0574
- accuracy: 0.9805 - val_loss: 0.7037 - val_accuracy: 0.8722
Epoch 84/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0563
- accuracy: 0.9808 - val_loss: 0.7161 - val_accuracy: 0.8702
Epoch 85/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0556
- accuracy: 0.9809 - val_loss: 0.7139 - val_accuracy: 0.8712
Epoch 86/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0548
- accuracy: 0.9811 - val_loss: 0.7203 - val_accuracy: 0.8704
Epoch 87/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0540
- accuracy: 0.9815 - val_loss: 0.7220 - val_accuracy: 0.8701
Epoch 88/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0533
- accuracy: 0.9816 - val_loss: 0.7272 - val_accuracy: 0.8714
Epoch 89/100
125/125 [==============================] - 2s 14ms/step - loss: 0.0525
- accuracy: 0.9818 - val_loss: 0.7263 - val_accuracy: 0.8701
```

```
Epoch 90/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0516
- accuracy: 0.9820 - val_loss: 0.7380 - val_accuracy: 0.8697
Epoch 91/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0512
- accuracy: 0.9821 - val_loss: 0.7337 - val_accuracy: 0.8713
Epoch 92/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0505
- accuracy: 0.9823 - val_loss: 0.7326 - val_accuracy: 0.8714
Epoch 93/100
125/125 [==============================] - 2s 15ms/step - loss: 0.0494
- accuracy: 0.9826 - val_loss: 0.7400 - val_accuracy: 0.8710
Epoch 94/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0491
- accuracy: 0.9828 - val_loss: 0.7434 - val_accuracy: 0.8709
Epoch 95/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0484
- accuracy: 0.9830 - val_loss: 0.7378 - val_accuracy: 0.8718
Epoch 96/100
125/125 [==============================] - 2s 14ms/step - loss: 0.0480
- accuracy: 0.9830 - val_loss: 0.7483 - val_accuracy: 0.8704
Epoch 97/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0474
- accuracy: 0.9833 - val_loss: 0.7452 - val_accuracy: 0.8704
Epoch 98/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0467
- accuracy: 0.9833 - val_loss: 0.7533 - val_accuracy: 0.8691
Epoch 99/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0460
- accuracy: 0.9836 - val_loss: 0.7550 - val_accuracy: 0.8702
Epoch 100/100
125/125 [==============================] - 2s 13ms/step - loss: 0.0458
- accuracy: 0.9835 - val_loss: 0.7579 - val_accuracy: 0.8698
```

Out[10]:

```
<tensorflow.python.keras.callbacks.History at 0x7fc7100f9a50>
```

In [11]:

```
model.save('eng2french.h5')
```

In [12]:

```
# Define sampling models
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = 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())
```

In [13]:

```
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 charact
er.
    target_seq[0, 0, target_token_index['\t']] = 1.

    # 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)

        # 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.

        # Update states
        states_value = [h, c]

    return decoded_sentence
```

In [14]:

```
for seq_index in range(100):
    # Take one sequence (part of the training set)
    # for trying out decoding.
    input_seq = encoder_input_data[seq_index: seq_index + 1]
```

```python
    decoded_sentence = decode_sequence(input_seq)
    print('-')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)
```

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