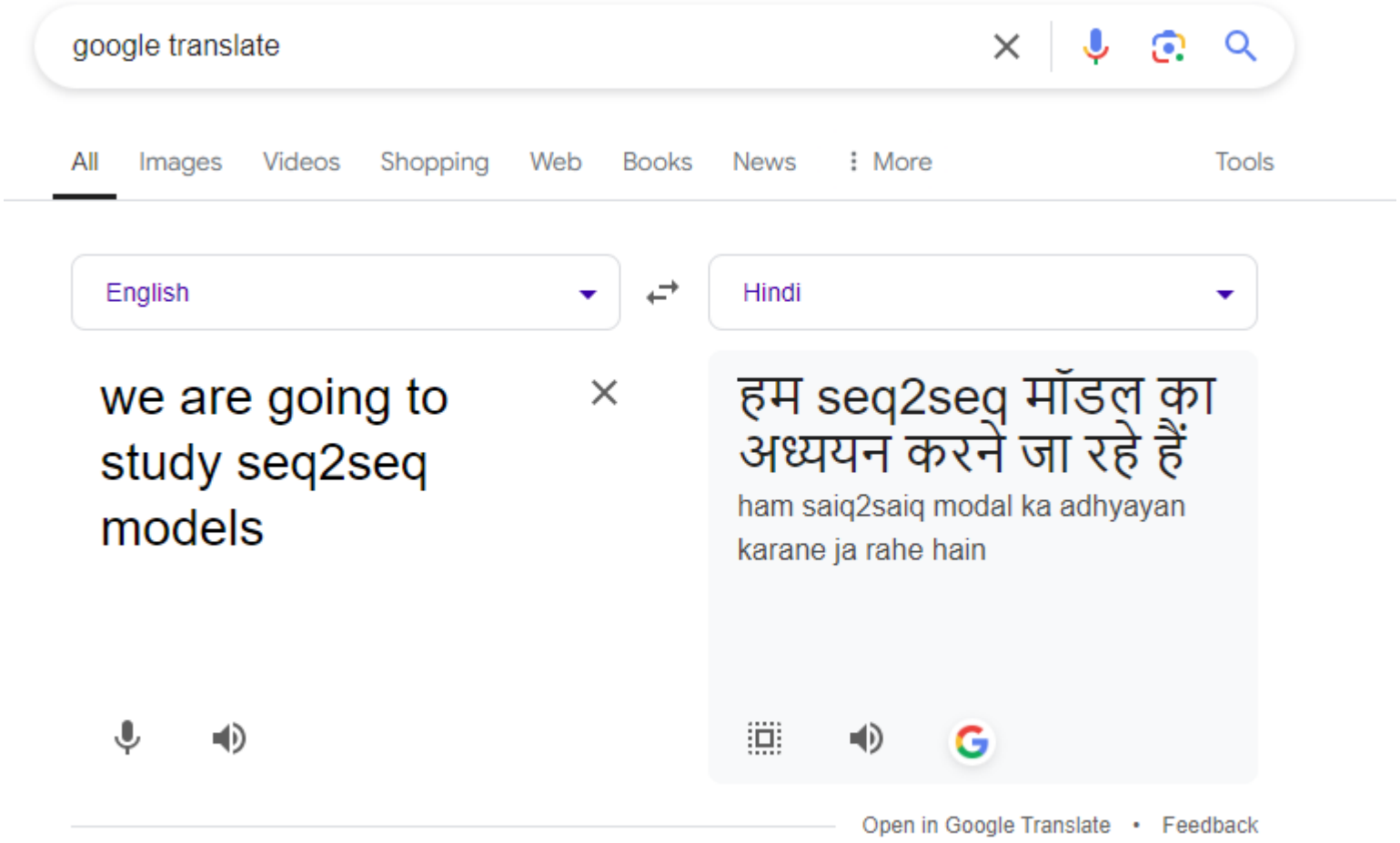
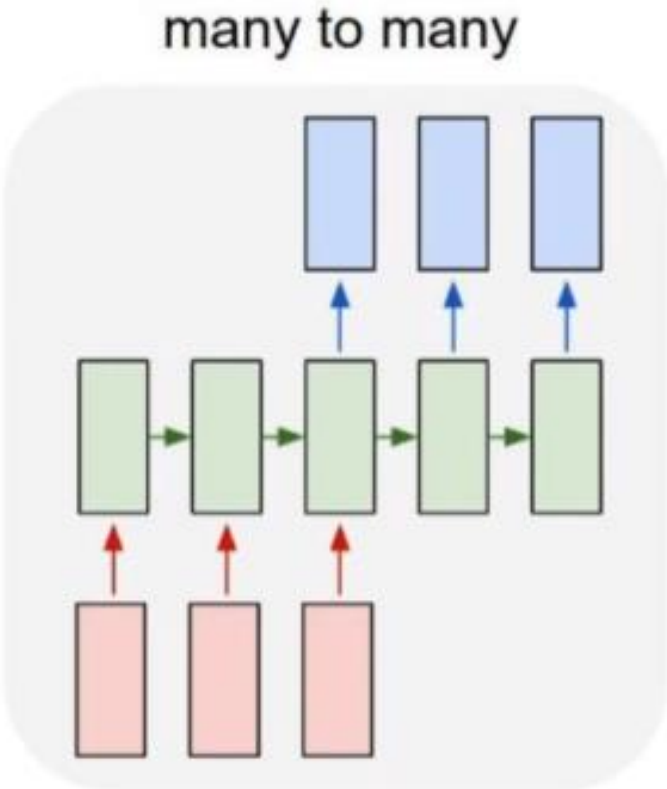
Encoder-Decoder: Sequence to Sequence Models

Where do we use seq2seq?



Introduction (Seq2Seq)

- Encoder-Decoder Sequence-to-Sequence (Seq2Seq) architectures are a popular deep learning framework for tasks where input and output sequences may differ in length and content.
- Introduced by Google in 2014.
- Widely used for applications like machine translation, text summarization, and conversational agents.
- Example, translating “What are you doing today?” from English to Chinese has input of 5 words and output of 7 symbols (今天你在做什麼？)



Machine Language Translation

Les modèles de séquence sont super puissants → Sequence Model → *Sequence models are super powerful*

Text Summarization

A strong analyst have 6 main characteristics. One should master all 6 to be successful in the industry :
1. .....
2. .....

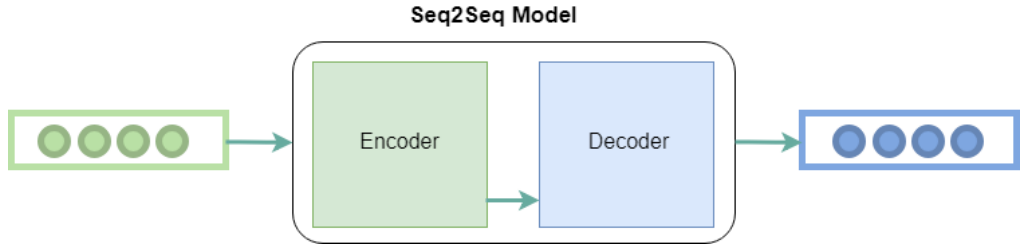
→ Sequence Model → *6 characteristics of successful analyst*

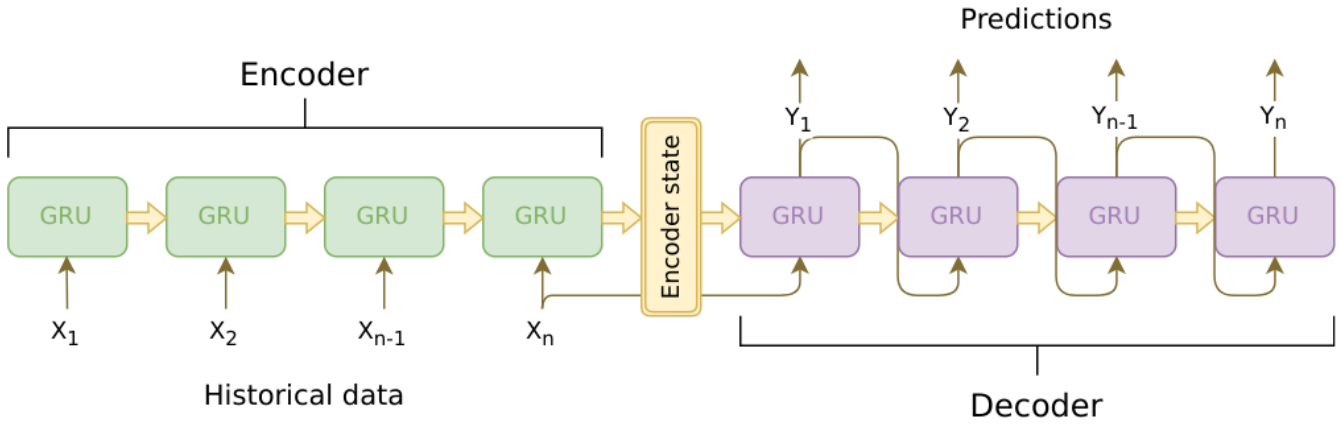
Chatbot

How are you doing today? → Sequence Model → *I am doing well. Thank you. How are you doing today?*

Key components

- A Sequence-to-Sequence (seq2seq) Encoder-Decoder is a Neural Network model consisting of two main components:
  - Encoder
  - Decoder.



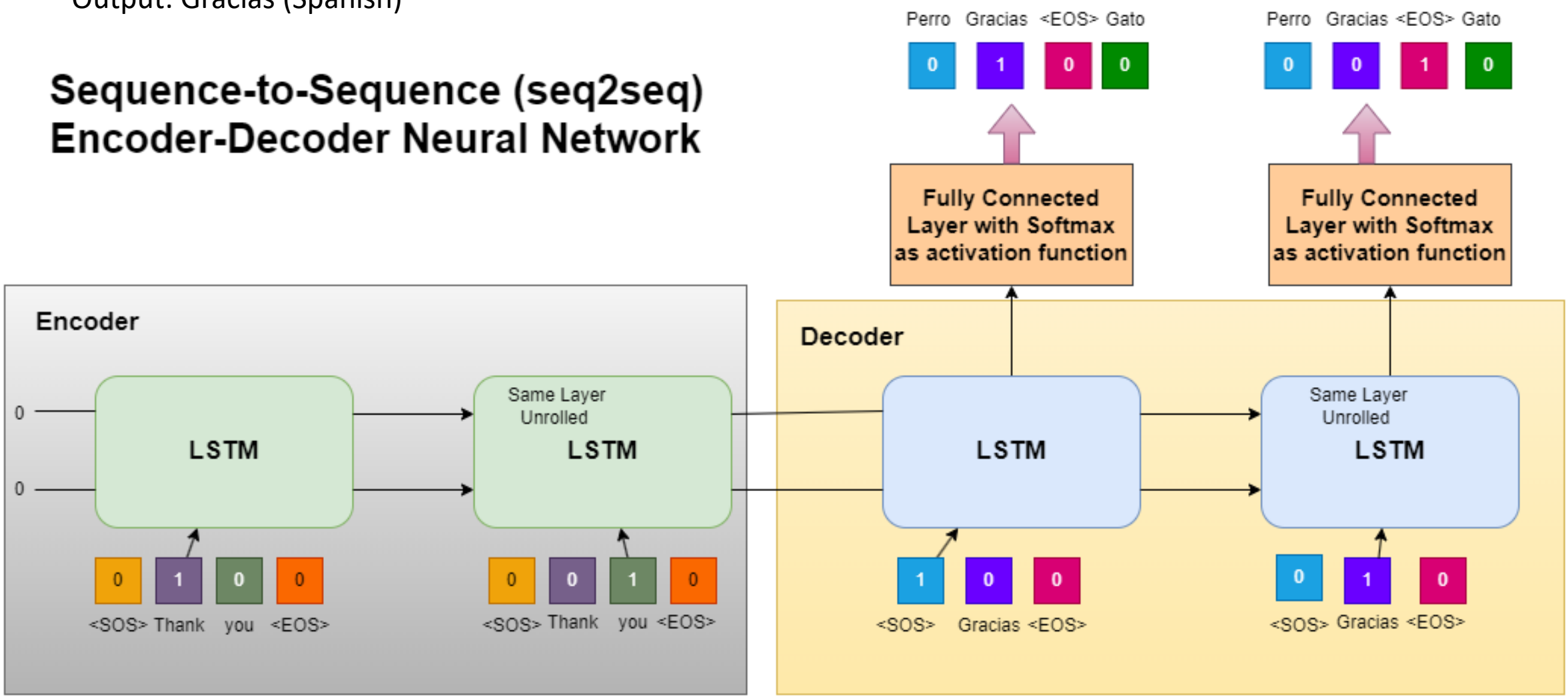


The model consists of 3 parts: encoder, intermediate encoder vector (context vector) and decoder.

Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence (y_1, y_2,.y_n) given an input sequence(x_1, x_2. . . , x_n).

Input: Thank you (English)
Output: Gracias (Spanish)

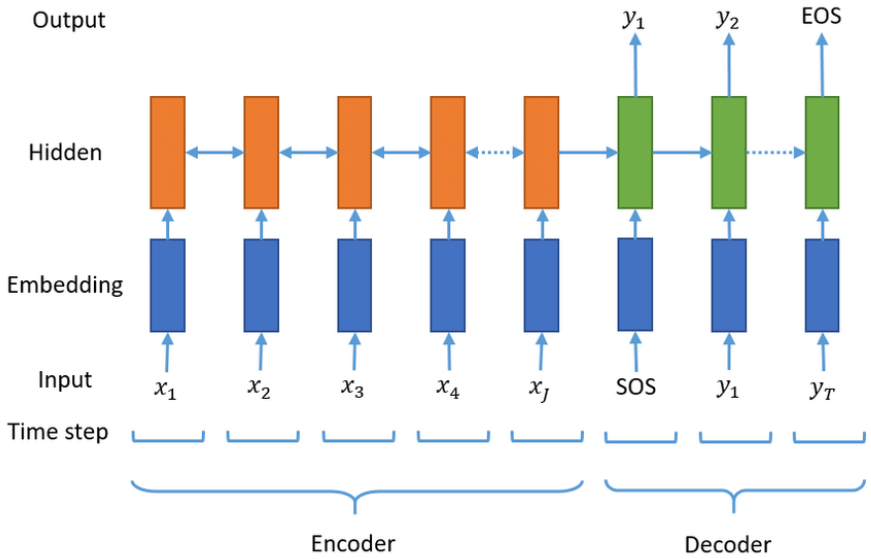
Sequence-to-Sequence (seq2seq) Encoder-Decoder Neural Network



- Example of translating the English sentence:

  "Thank you" into Spanish "Gracias".

- **1. Input Representation**
  - The first step in the seq2seq model is to convert the input sentence into a suitable representation that can be fed into the model.
  - This is done using an embedding layer. The embedding layer transforms each word in the input sentence into a dense vector of fixed size. These vectors capture the semantic properties of the words.
  - For instance, the word "Thank" and "you" are each converted into a vector.
    - e.g. Suppose we have an English vocabulary of 5000 words and each word is represented as a 300-dimensional vector.
    - The words "Thank" and "You" are each converted into a 300-dimensional vector using the embedding layer. These vectors are the input to the encoder.



Build the LSTM Model

Embedding Layer: Converts integer sequences to dense vectors of fixed size (100 in this case).

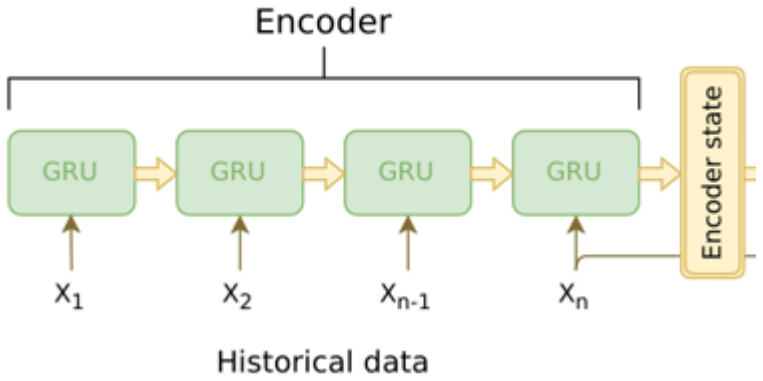
LSTM Layer: A recurrent layer that processes the sequences. Here, we use 110 units.

Dense Layer: The output layer with a softmax activation function for multi-class classification, which predicts the next word from the vocabulary.

```
model = Sequential()
model.add(Embedding(total_words, 100, input_length=max_sequence_length - 1))
model.add(LSTM(110))
model.add(Dense(total_words, activation='softmax'))
```

- **2. Encoder**

  - The input to the encoder is the sequence in the source language (embedding)
  - The main purpose of the encoder block is to process the input sequence and capture information in a fixed-size context vector.
  - The encoder block consists of a stack of several recurrent units (RNN, LSTM or GRU cells) where each accepts a single element of the input sequence (ex: word), collects information for that element and propagates it forward.
  - Throughout this process, the encoder keeps an internal state, and the ultimate hidden state functions as the context vector that encapsulates a compressed representation of the entire input sequence.
  - This context vector captures the semantic meaning and important information of the input sequence.



- The final hidden state of the encoder is then passed as the context vector to the decoder.
- For a given sequence of inputs ($x_1, x_2, \ldots, x_T$), a RNN updates h_t through iterative computation based on the following equation:

$$h_t = \sigma(W^{hx} x_t + W^{hh} h_{t-1} + b)$$

- The final hidden state of the encoder is then passed as the context vector to the decoder.

- Example: English (“Thank you”) to Spanish(“Gracias”) conversion
- Suppose our encoder is made up of RNN cells
  - The RNN processes the input sequence step-by-step, maintaining an internal state that encapsulates the information it has seen so far.
  - For the sentence "Thank you", the RNN processes the word "Thank" first, then updates its internal state(ht).
  - Next, it processes the word "you", and again updates its state.
  - The final state of the RNN, often called the context vector, serves as the output of the encoder.
  - This context vector is a dense vector that represents the entire input sentence.

- **3. Decoder Block**
  - The decoder block is similar to encoder block. The decoder processes the context vector from encoder to generate output sequence incrementally.
  - It consists of a stack of several recurrent units where each predicts an output y_t at a time step t.
  - Each recurrent unit accepts a hidden state and the embedding of the output token from the previous unit to produces its own hidden state (St).
  - St is then used to generate a probability distribution over the possible next tokens. The token with the highest probability is then chosen as the output, and the process continues until the end of the output sequence is reached.
  - Decoder input :

    - The **previous target token** y_{t-1}: the token generated in the previous timestep.
    - The **context vector** c: typically, this is either the final hidden state of the encoder (in a basic Seq2Seq model) or a dynamically computed context vector (when using attention).

The embedding of the decoder tokens may be different from that of the encoder in the case of machine translation (here, spanish)

Hidden state, St, Computation

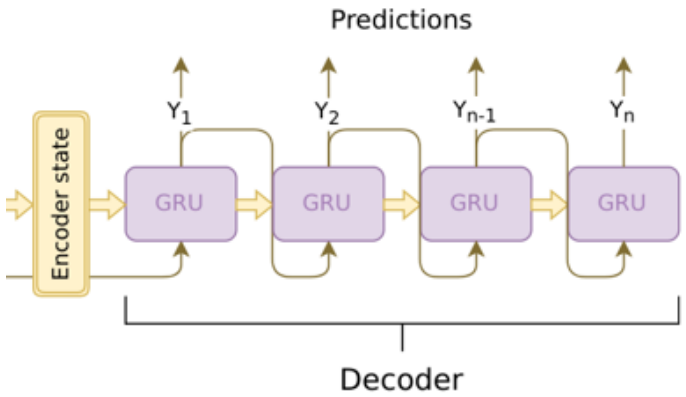
For a simple RNN, the hidden state s_t is computed by applying a linear transformation and an activation function (typically tanh) to the concatenated previous hidden state s_{t-1} and the input x_t:

$$s_t = \tanh\ (W_x . X_t + W_s . S_{t-1} + b)$$

where:

- W is the weight matrix that applies to both s_{t-1} and x_t.
- b is the bias term.
- $\tanh$ is the activation function, which helps keep the values in a bounded range.

Here, Xt, is the o/p from the previous unit



- In the Example: English (“Thank you”) to Spanish(“Gracias”) conversion, now we have the context vector corresponding to “Thank you ” as input to our decoder.
  - The decoder starts by feeding a special start-of-sequence (SOS) token into its embedding layer (target embeddings: here, spanish).
    - e.g. Suppose we have a Spanish vocabulary of 6000 words and each word is represented as a 300-dimensional vector. The SOS token is converted into a 300-dimensional vector using the Spanish embedding layer.
  - The resulting vector, along with the context vector, is fed into the RNN.
  - The RNN processes this input and updates its state.
  - The output of the RNN at this step represents the first word of the output sequence.

4. Fully Connected Layer and Softmax

- The output of the RNN at each step is then passed through a fully connected (dense) layer that transforms it into a vector of the same size as the output (Spanish in this case) vocabulary.
- This vector is then passed through a softmax function, which turns it into a probability distribution. The softmax function ensures that all the values in the output vector are between 0 and 1 and sum up to 1, which allows us to interpret them as probabilities.
    - The output probability calculation

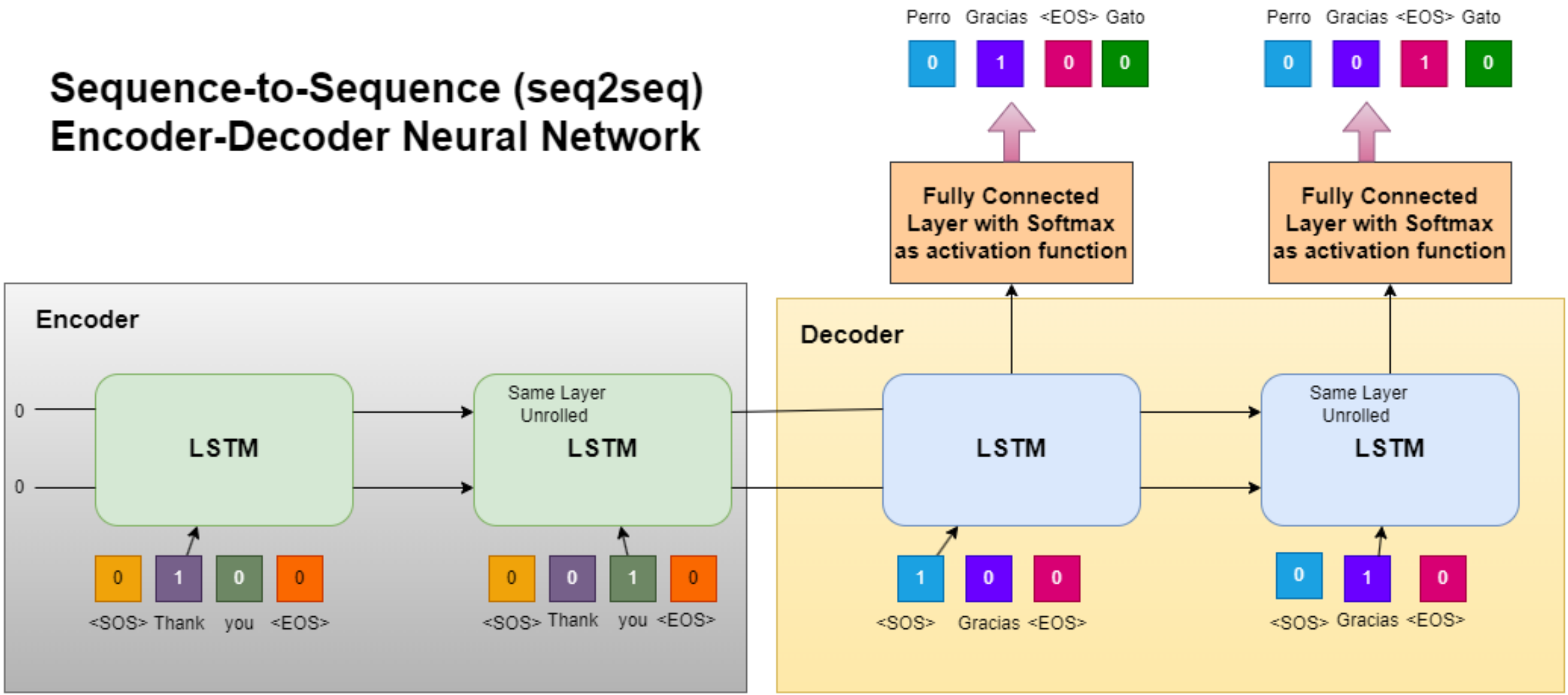
      At each timestep t, the decoder generates a probability distribution over the target vocabulary. This is done by applying a linear transformation to the hidden state s_t and passing the result through a softmax function:

      $$p(y_t|y_{<t}, X) = \text{softmax}(W_o \cdot s_t + b_o)$$

      where W_o and b_o are learned parameters that map the hidden state s_t to the vocabulary size.

- The index of the highest value in the output vector after the softmax function is taken as the index of the predicted word in the Spanish vocabulary. This word is then added to the output sequence.
- i.e. The output of the RNN at this step is passed through a fully connected layer that transforms it into a 6000-dimensional vector (the size of the Spanish vocabulary). This vector is then passed through a softmax function, which turns it into a probability distribution. Suppose the highest value in the output vector corresponds to the word "Gracias" in the Spanish vocabulary. This word is then added to the output sequence.

Sequence-to-Sequence (seq2seq) Encoder-Decoder Neural Network



5. Sequence Generation

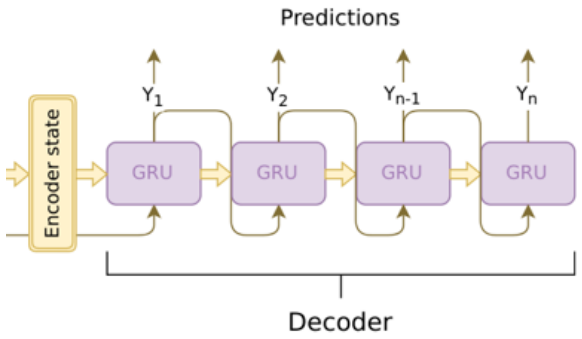
- The decoder continues to generate the output sequence word-by-word.
- After generating the first word, it feeds this word (in its embedded form) back into the RNN, along with the updated state, to generate the next word.
- This process continues until the decoder generates an end-of-sequence (EOS) token, signaling that the translation is complete.

6. Training

- During training, the decoder of the seq2seq model uses a technique called **teacher forcing**.
- Instead of feeding its own predictions back into the RNN, it feeds the actual target words from the training data.
- This helps the model learn more effectively, as it prevents errors from accumulating over time.

7. Inference (testing)

- During inference (i.e., when the model is being used to translate new sentences), the decoder feeds its own predictions back into the RNN, as the actual target words are not known.

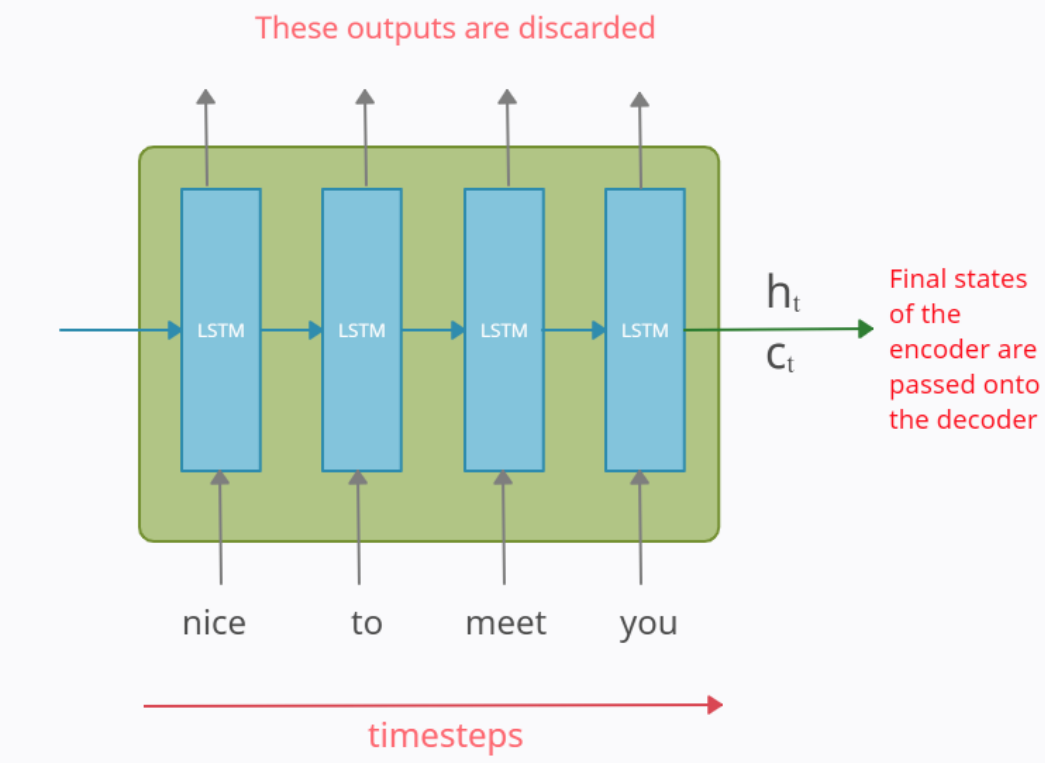


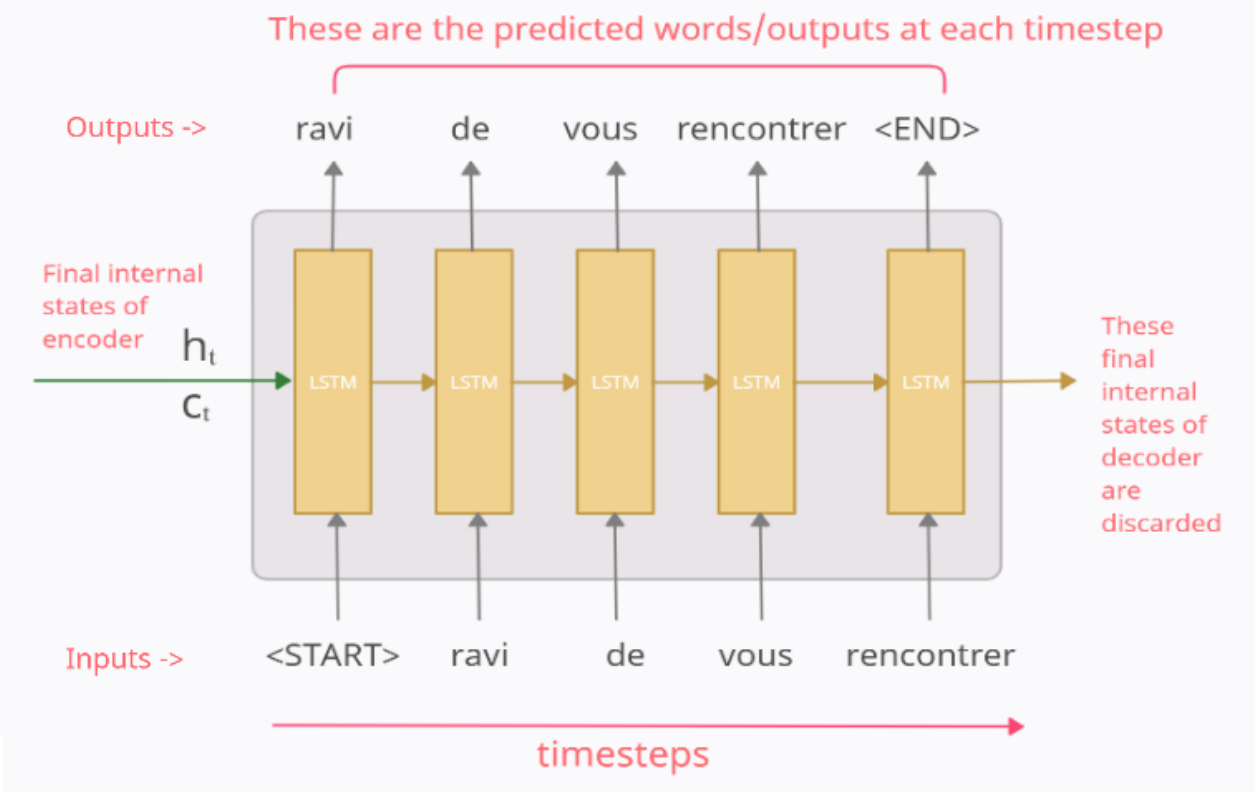
Illustrated example

- **Task**: Predict the French translation for every English sentence used as input.
  - **Input (X): English sentence: “nice to meet you”**
  - **Output(Y_true): French translation: “ravi de vous rencontrer”**

Encoder block and decoder block

Recurrent Neural Network used: LSTM





Decoder

At time-step 1

- The input fed to the decoder at the first time-step is a special symbol **“<START>”**. This is used to signify the start of the output-sequence.
- Now the decoder uses this input and the internal states **(h_t, c_t)** to produce the output in the 1st time-step which is supposed to be the 1st word/token in the target-sequence i.e. **‘ravi’**.

At time-step 2

- At time-step 2, the output from the 1st time-step **“ravi”** is fed as input to the 2nd time-step. The output in the 2nd time-step is supposed to be the 2nd word in the target-sequence i.e. **‘de’**
- And similarly, the output at each time-step is fed as input to the next time-step.
- This continues till we get the “<END>” symbol which is again a special symbol used to mark the end of the output-sequence.
- The final internal states of the decoder are discarded.

Note:

- *These special symbols need not necessarily be “*<START>” and “<END>” *only. These can be any strings given that these are not present in our data corpus so the model doesn’t confuse them with any other word. In the paper, they have used the symbol* “<EOS>” *and in a slightly different manner*.
- The process mentioned above is how an **ideal** decoder will work in the testing phase. But in the training phase, a slightly different implementation is required, to make it train faster.

How it works?
Training and Testing Phase: Getting into the tensors

- **Vectorizing the data**
- The raw data that we have is
  - **X = “nice to meet you” → Y_true = “ravi de vous rencontrer”**

- Now we put the special symbols “<START>” and “<END>” at the start and the end of our target-sequence
  - **X = “nice to meet you” → Y_true = “<START> ravi de vous rencontrer <END>”**

- Next the input and output data is vectorized using one-hot-encoding. Let the input and output be represented as
  - **X = (x1, x2, x3, x4) → Y_true = (y0_true, y1_true, y2_true, y3_true, y4_true, y5_true)**

For input X

‘nice’ → x1 : [1 0 0 0]

‘to’ → x2 : [0 1 0 0]

‘meet’ →x3 : [0 0 1 0]

‘you’ → x4 : [0 0 0 1]

For Output Y_true

‘<START>’ → y0_true : [1 0 0 0 0 0]

‘ravi’ → y1_true : [0 1 0 0 0 0]

‘de’ → y2_true : [0 0 1 0 0 0]

‘vous’ → y3_true : [0 0 0 1 0 0]

‘rencontrer’ → y4_true : [0 0 0 0 1 0]

‘<END>’ → y5_true : [0 0 0 0 0 1]

Note: we use embeddings

Build the LSTM Model

Embedding Layer: Converts integer sequences to dense vectors of fixed size (100 in this case).

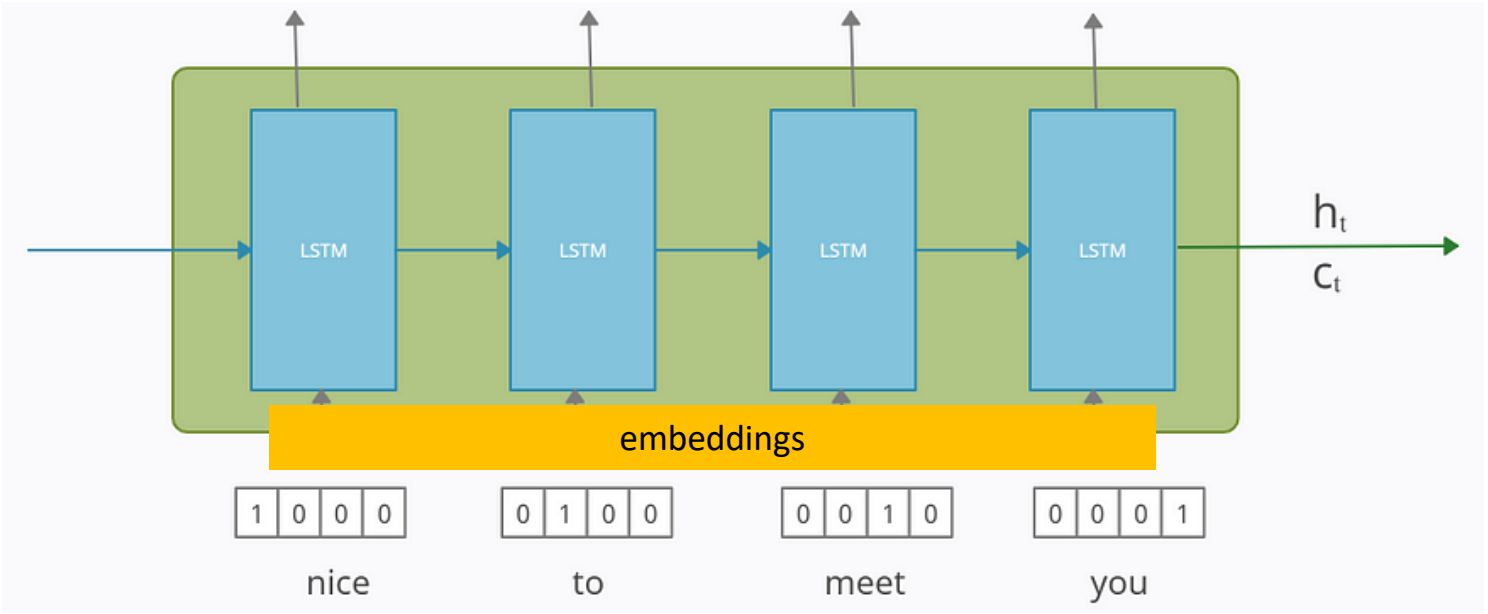
LSTM Layer: A recurrent layer that processes the sequences. Here, we use 110 units.

Dense Layer: The output layer with a softmax activation function for multi-class classification, which predicts the next word from the vocabulary.

```
model = Sequential()
model.add(Embedding(total_words, 100, input_length=max_sequence_length - 1))
model.add(LSTM(110))
model.add(Dense(total_words, activation='softmax'))
```

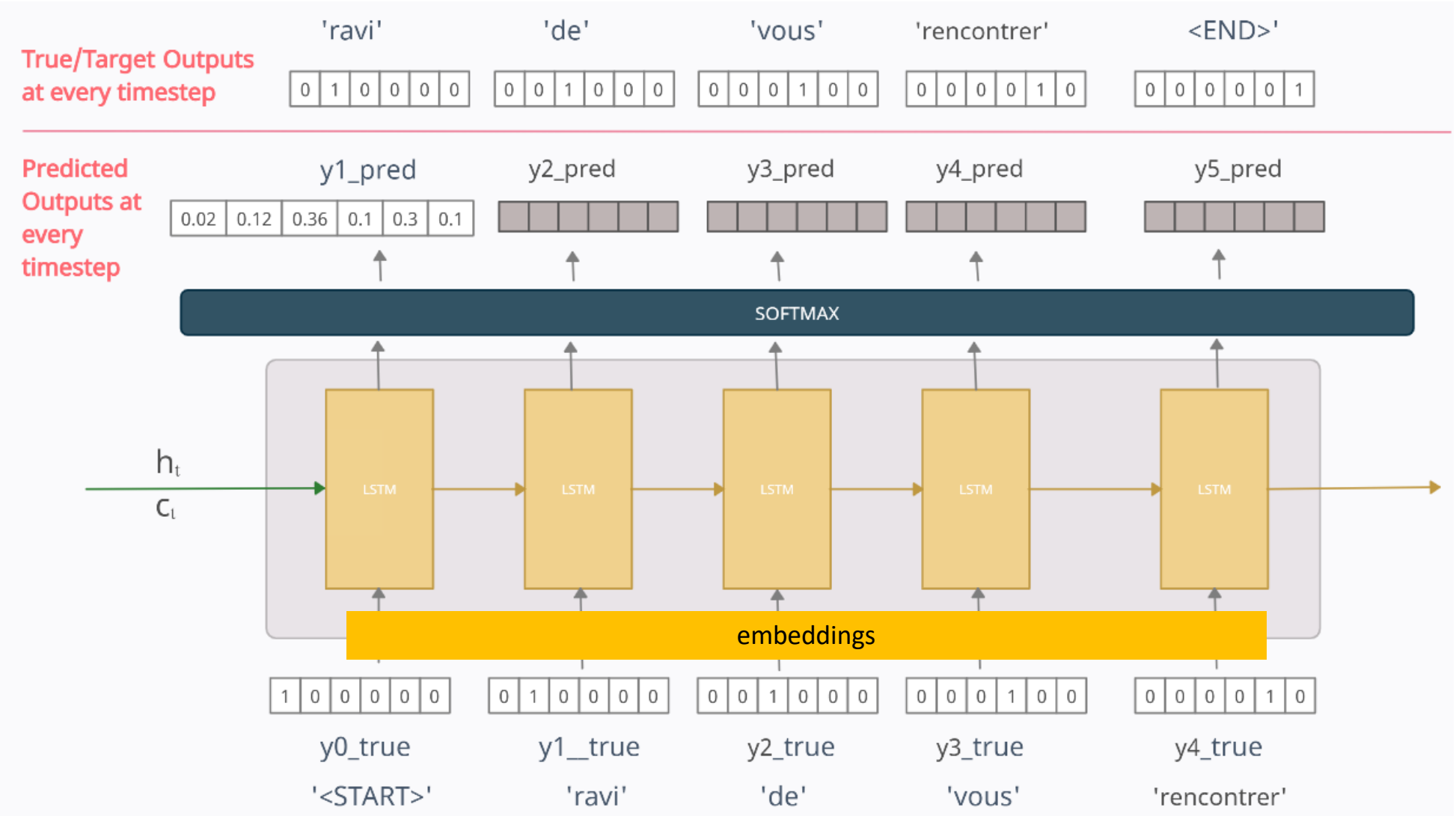
Training and Testing of Encoder

- The working of the encoder is the same in both the training and testing phase.
- It accepts each token/word of the input-sequence one by one and sends the final states to the decoder.
- Its parameters are updated using backpropagation over time.



The Decoder in Training Phase: Teacher Forcing

- The working of the decoder is different during the training and testing phase, unlike the encoder part.
- To train the decoder model, we use a technique called "**Teacher Forcing"** in which we feed the **true** output/token (and **not the predicted** output/token) from the previous time-step as input to the current time-step.



Note that in the decoder, at any time-step **t**, the output **yt_pred** is the probability distribution over the entire vocabulary in the output dataset which is generated by using the Softmax activation function. The token with the maximum probability is chosen to be the predicted word.

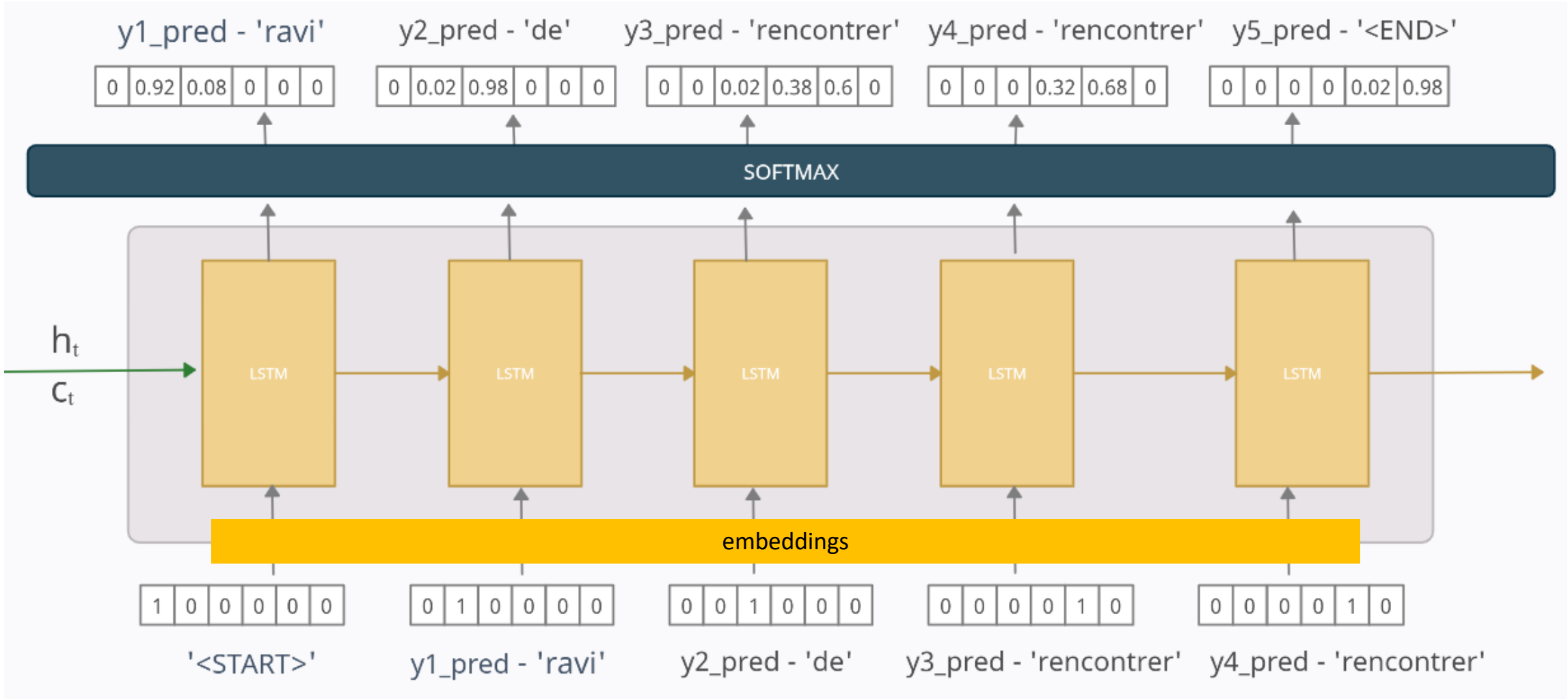
Teacher forcing:

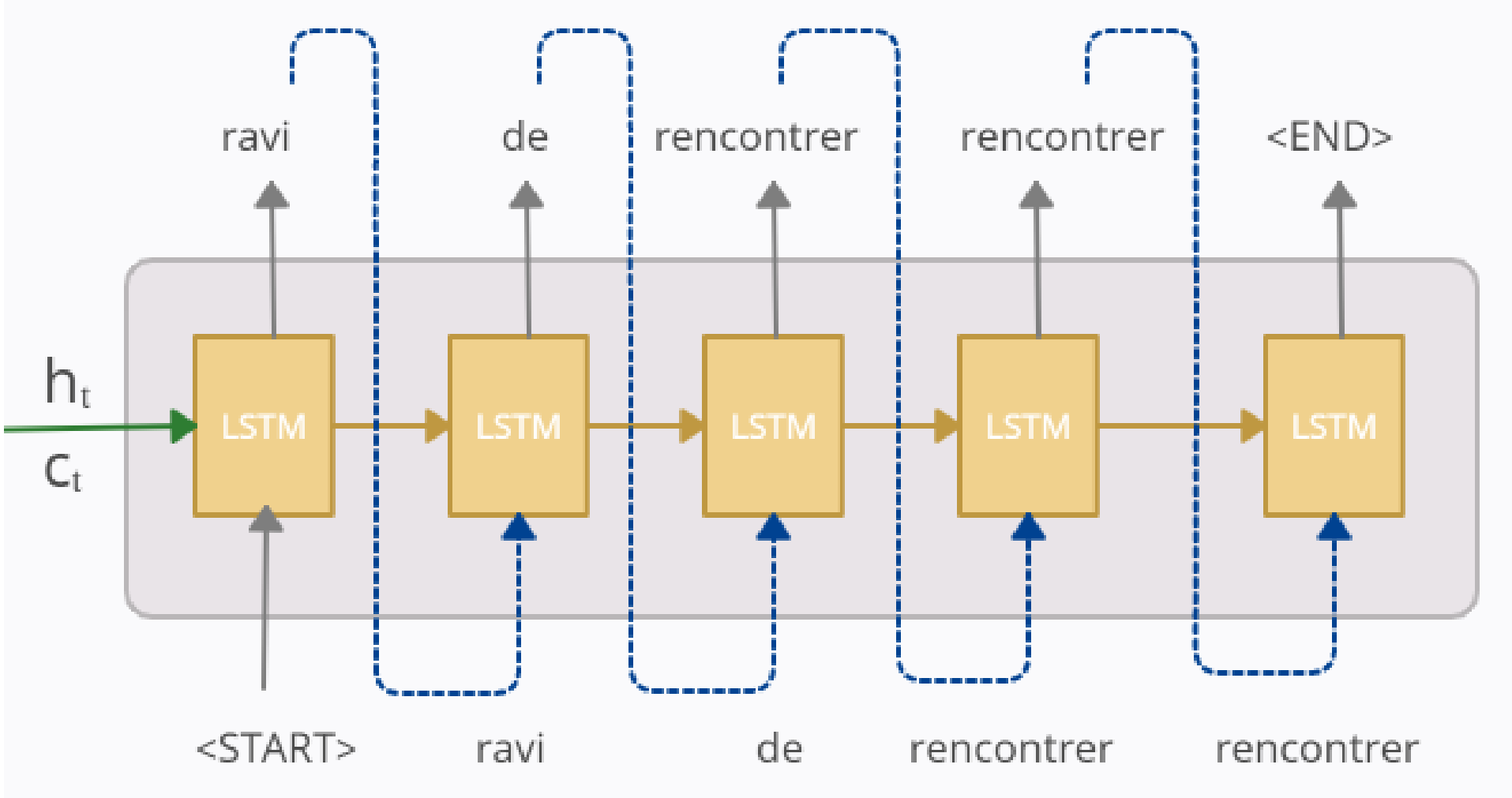
- we feed the true output/token (and not the predicted output) from the previous time-step as input to the current time-step.
- That means the input to the time-step 2 will be y1_true=[0 1 0 0 0 0] and not y1_pred.
- Now the output at time-step 2 will be some random vector y2_pred. But at time-step 3 we will be using input as y2_true=[0 0 1 0 0 0] and not y2_pred. Similarly at each time-step, we will use the true output from the previous time-step.
- Finally, the loss is calculated on the predicted outputs from each time-step and the errors are backpropagated through time to update the parameters of the model.
- The loss function used is the categorical cross-entropy loss function between the target sequence/**Y_true** and the predicted-sequence/**Y_pred** such that:

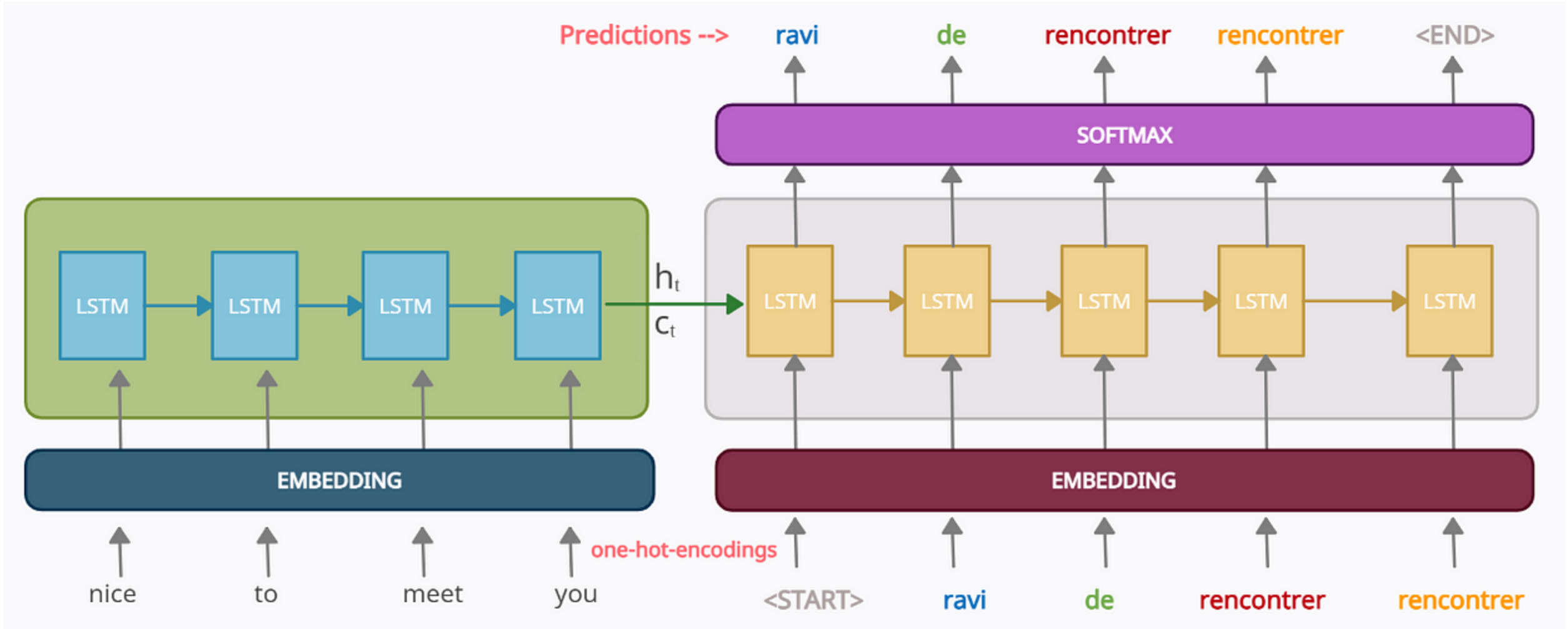
- Y_true = [y0_true, y1_true, y2_true, y3_true, y4_true, y5_true]
- Y_pred = [‘<START>’, y1_pred, y2_pred, y3_pred, y4_pred, y5_pred]

The final states of the decoder are discarded

The Decoder in Test Phase




ravi
de
rencontrer
rencontrer
<END>
h_t
c_t
LSTM
LSTM
LSTM
LSTM
LSTM
<START>
ravi
de
rencontrer
rencontrer



[Encoder-Decoder Seq2Seq Models, Clearly Explained!! | by Kriz Moses | Analytics Vidhya | Medium]

Output Probability Calculation

At each timestep t, the decoder generates a probability distribution over the target vocabulary. This is done by applying a linear transformation to the hidden state s_t and passing the result through a softmax function:

$$p(y_t|y_{<t}, X) = \text{softmax}(W_o \cdot s_t + b_o)$$

where W_o and b_o are learned parameters that map the hidden state s_t to the vocabulary size.