# Deep Learning

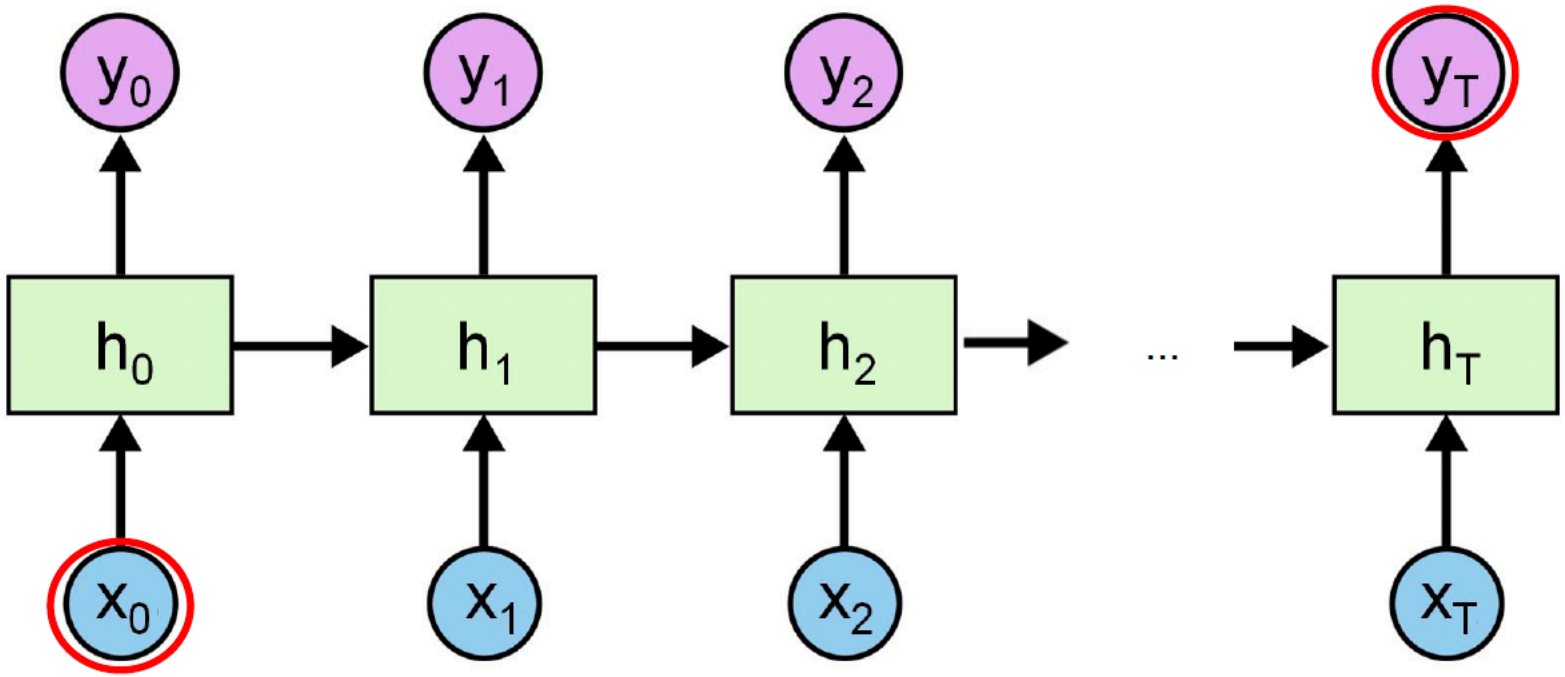
## COMP 6721 Introduction of AI

# Topics

- Training Problems
- RNN Architectures
- Deep RNNs
- Sequence to Sequence Models
- Attention Mechanism

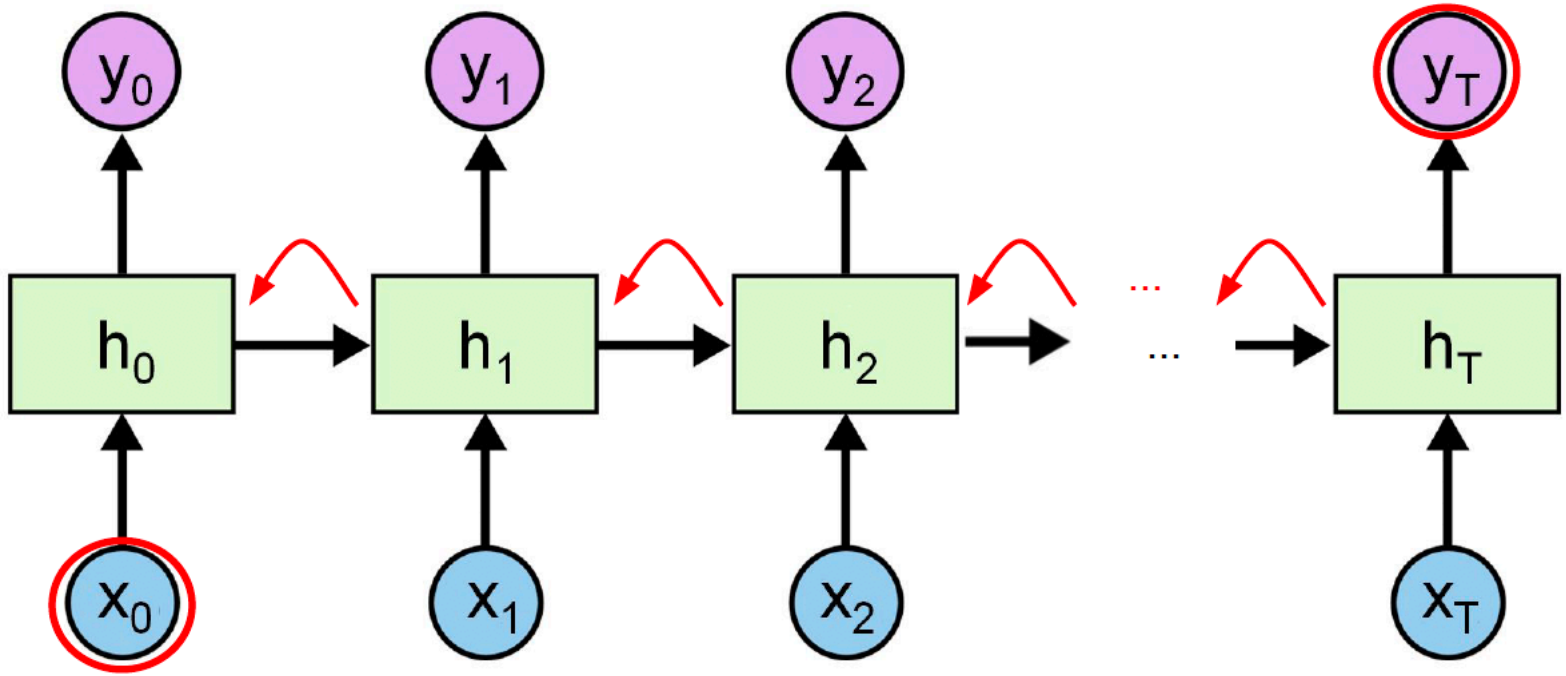
# Long-Term Dependencies

- For long sequences, it may be important to capture long-term dependencies.



# Long-Term Dependencies

- The problem is the long chain of gradients: $\frac{\partial h_T}{\partial h_{T-1}} \cdots \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial h_0}$



# Long-Term Dependencies

- Going back to the main equation for the internal state:

$$h_t = tanh(Ux_t + Wh_{t-1} + b_h)$$

- For a generic $h_t$, the gradient with respect to the internal state at the previous time step is:

$$\frac{\partial h_t}{\partial h_{t-1}} = W \frac{\partial\ tanh(Ux_t + Wh_{t-1} + b_h)}{\partial h_{t-1}}$$

- In particular, note the term W.

# Long-Term Dependencies

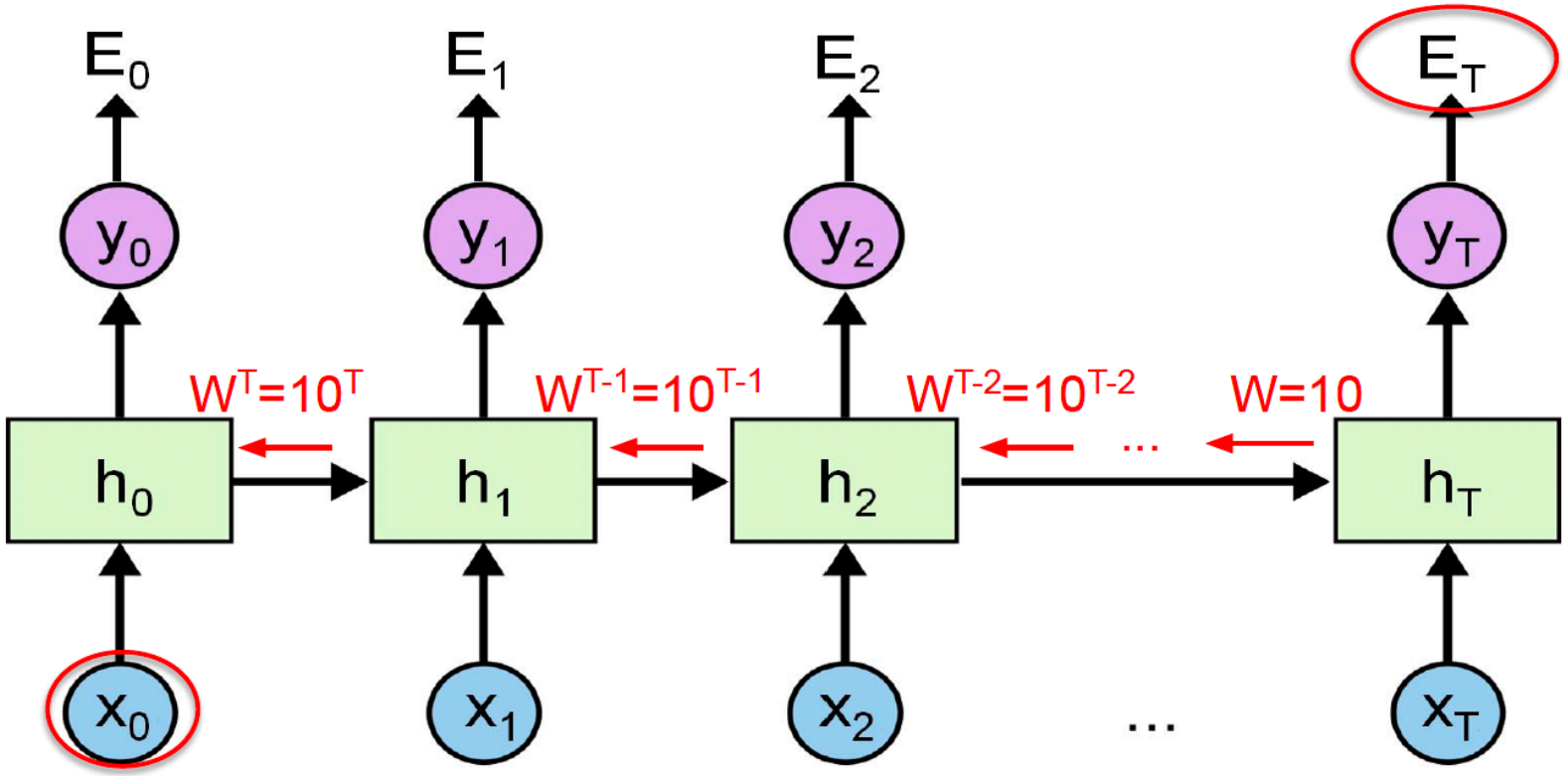
- Given we have a long chain of multiplication...

$$\frac{\partial h_T}{\partial h_{T-1}} \cdots \frac{\partial h_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial h_0}$$

- ...we multiply by W several times.
- This can can make the system unstable.
- In particular, the result can “explode” or “vanish”.

# Exploding Gradient

Simple 1-dimensional example with W = [10]




The gradient increases at every step = exploding gradient!

# Exploding Gradient

- The gradient increases at every step ⇒ exploding gradient!
- Problem: the parameters will diverge.
  - Can lead to overflow problems.
- Simple solution: Gradient Clipping.

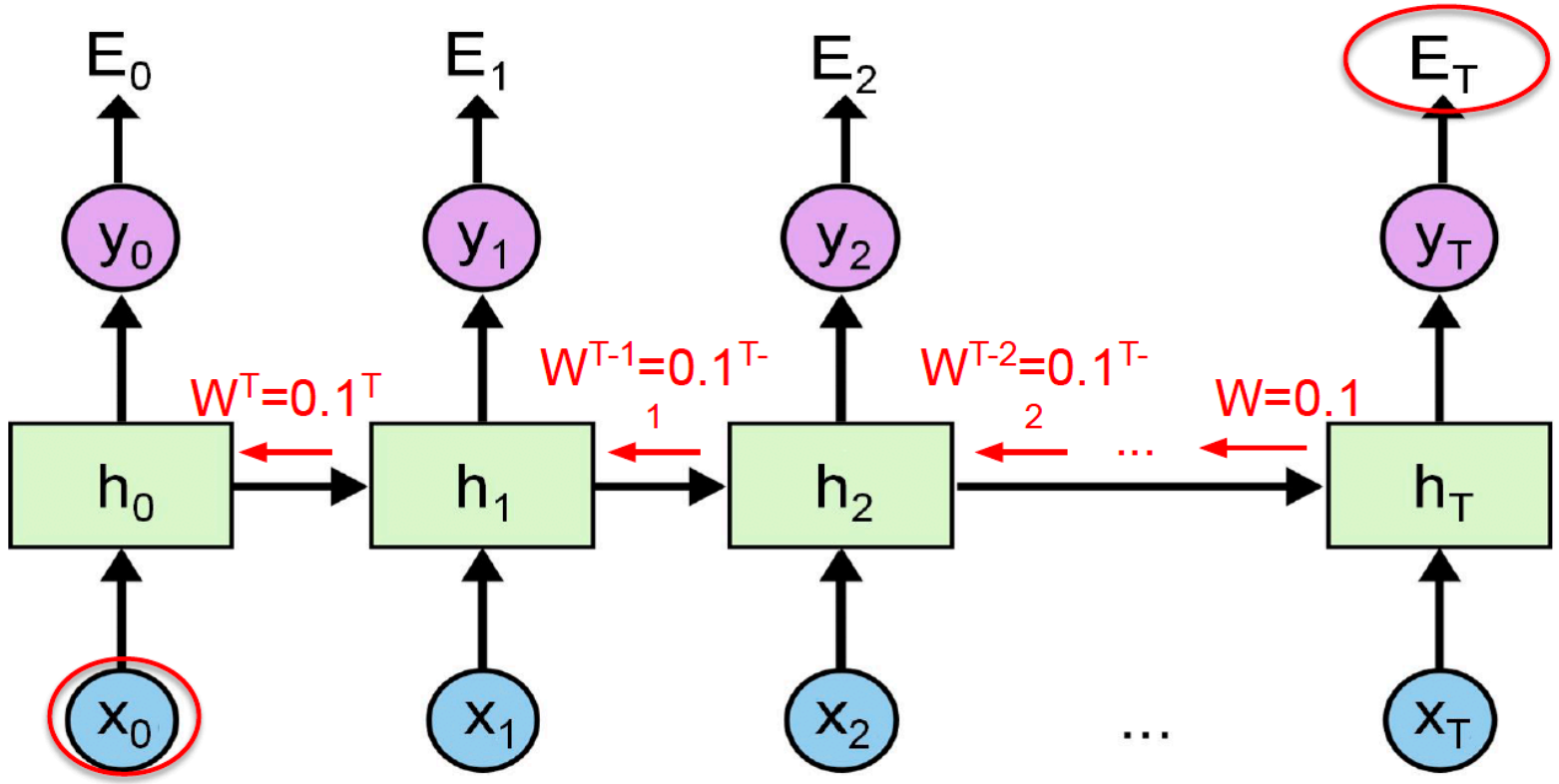
$$g = \frac{\partial E}{\partial W}$$

$$\textbf{if } \|g\| \geq threshold \textbf{ then}$$

$$g \leftarrow \frac{threshold}{\|g\|} g$$

# Exploding Gradient

Simple 1-dimensional example with W = [0.1]




The gradient increases at every step = exploding gradient!

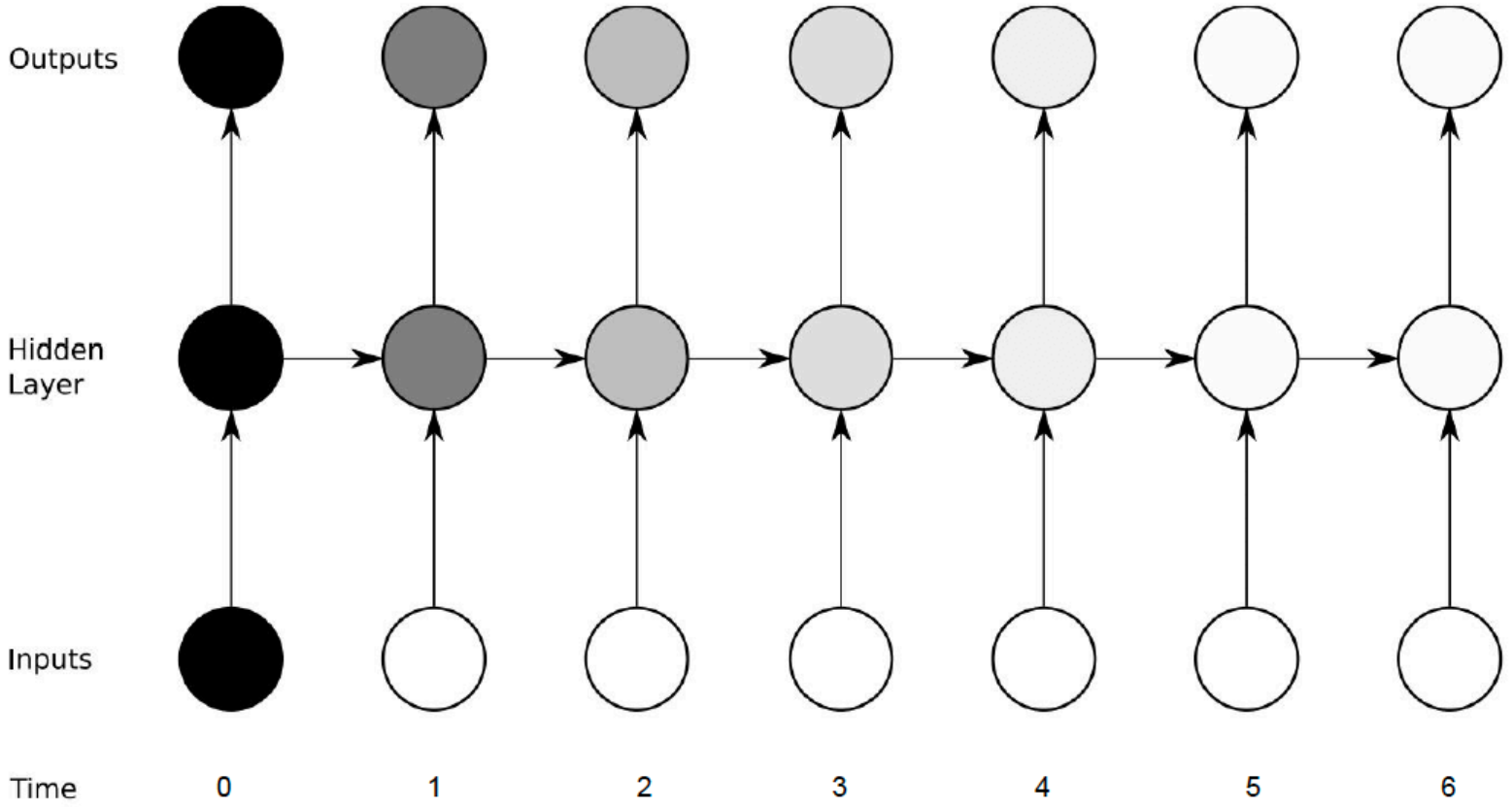
# Exploding Gradient

- The gradient diminishes at every step ⇒ vanishing gradient!
- Problem: very slow learning (or no learning at all).
  - It affects long-term dependency learning.
- There is no easy solution.
- We need to use more complex RNN architectures.

# Topics

- ***Training Problems***
- RNN Architectures
- Deep RNNs
- Sequence to Sequence Models
- Attention Mechanism

# Memory Problems

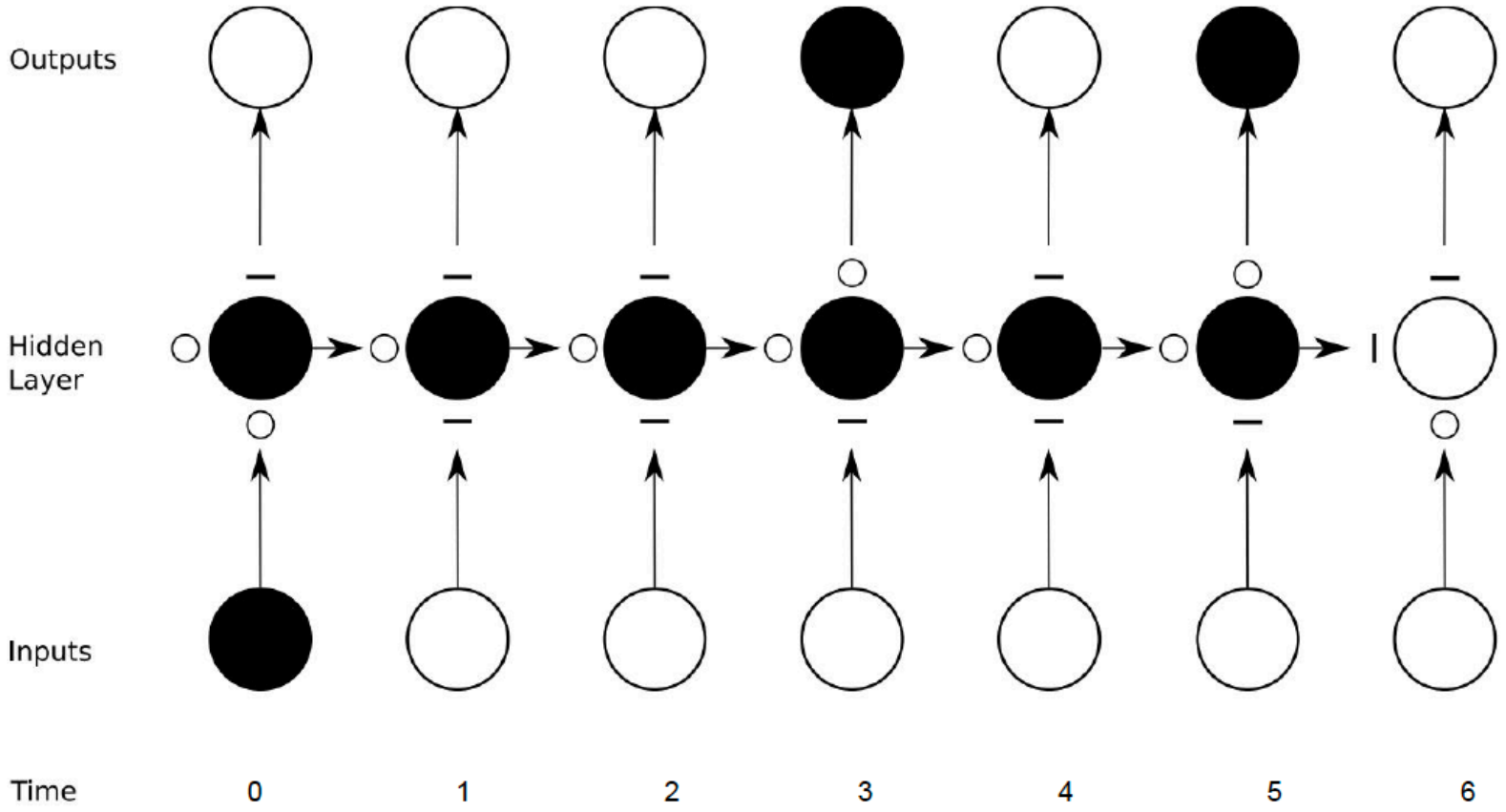



颜色显示了在时间0时输入的影响，随着RNN逐渐忘记该特定输入，该影响随着时间而减小。

The colors show the influence of the input at time 0 which decreases over time as the RNN gradually forgets that particular input.

Graves, Supervised Sequence Labelling with Recurrent Neural Networks

# Memory Problems




By adding gates (o open; - closed), the RNN can selectively control the flow of information (and greatly minimize the vanishing gradient problem).

Graves, Supervised Sequence Labelling with Recurrent Neural Networks

# Remember this slide?

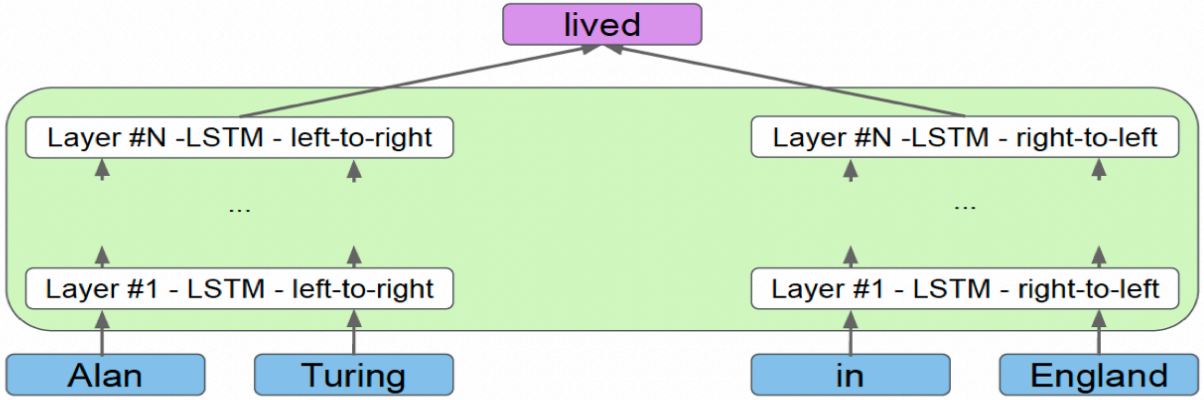
58

## ELMo

- ELMo model is a N-layer bidirectional LSTM.

(Note: **LSTM** ( Long Short Term Memory ) Networks are called fancy recurrent neural networks with some additional features.)

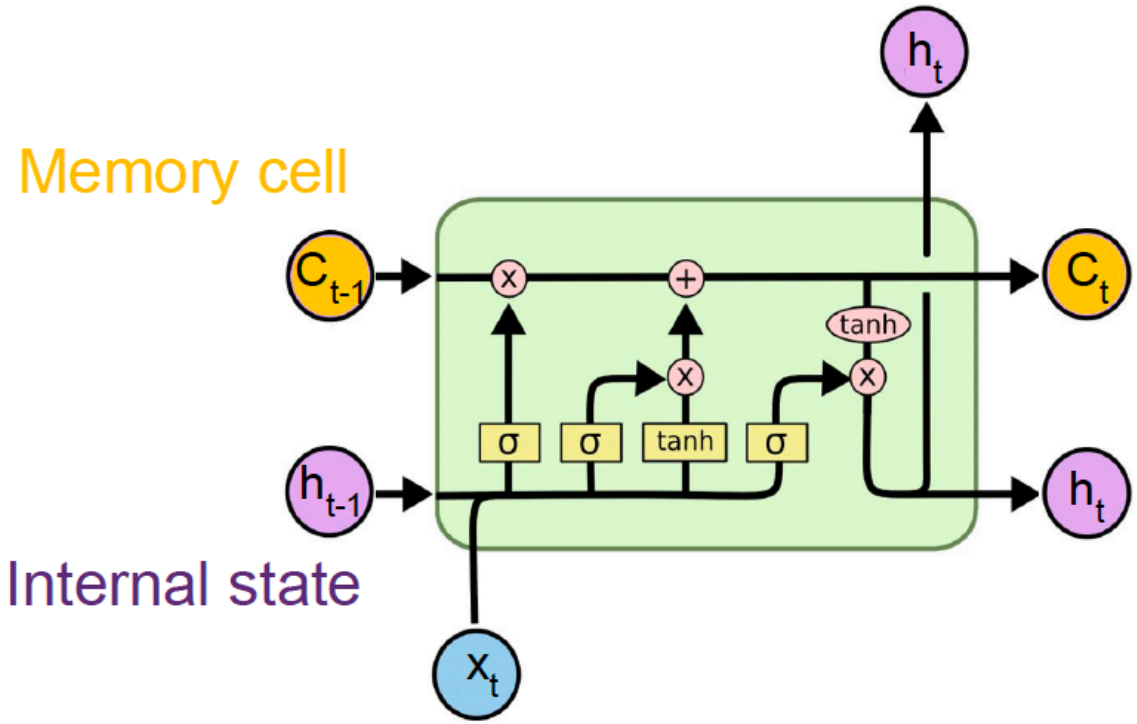
- The contextualized embedding at a time step corresponds to a combination of the hidden states of all the various layers.




Peters et al. "Deep contextualized word representations."

# Long Short-Term Memory (LSTM)

Reduce the vanishing gradient problem using a **gate mechanism** and adding a **memory cell**.

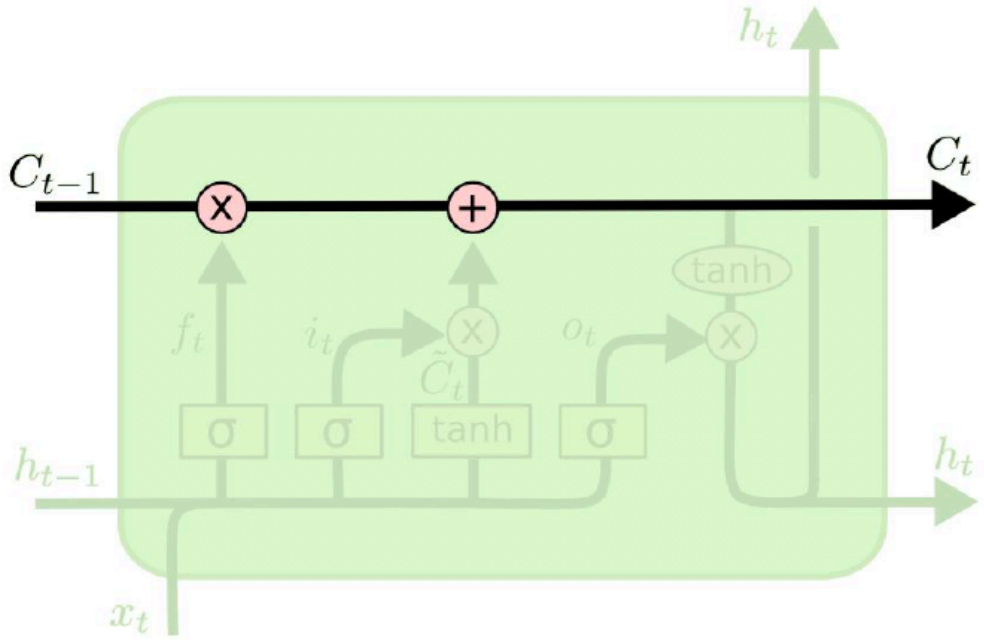



$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$
$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$
$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$$
$$g_t = tanh(U_g x_t + W_g h_{t-1} + b_g)$$
$$C_t = i_t \times g_t + f_t \times C_{t-1}$$
$$h_t = o_t \times tanh(C_t)$$

Image from Christopher Olah's blog
Hochreiter et al., Long short-term memory, Neural Computation 1997

# LSTM – Step-by-Step

- The key idea introduced in the LSTM is the **Memory cell**.
  - Few operations happen there.
  - Information can flow more easily.

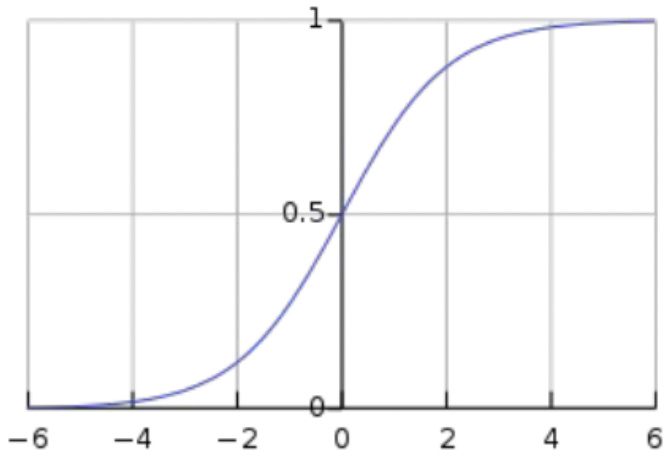


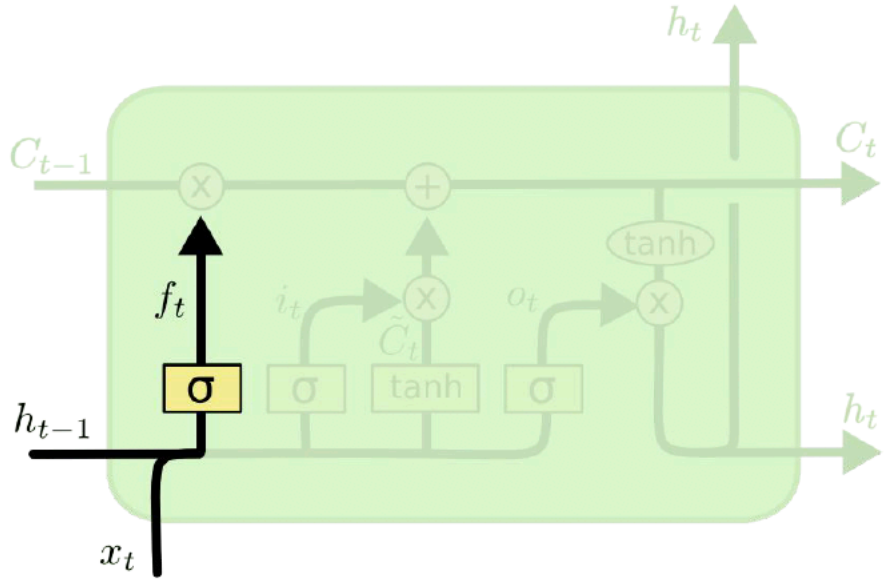
# LSTM – Step-by-Step

- The **Forget gate** is computed from $x_t$ and $h_{t-1}$:

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$

- $\sigma$ is the sigmoid function (bounded between 0 and 1).



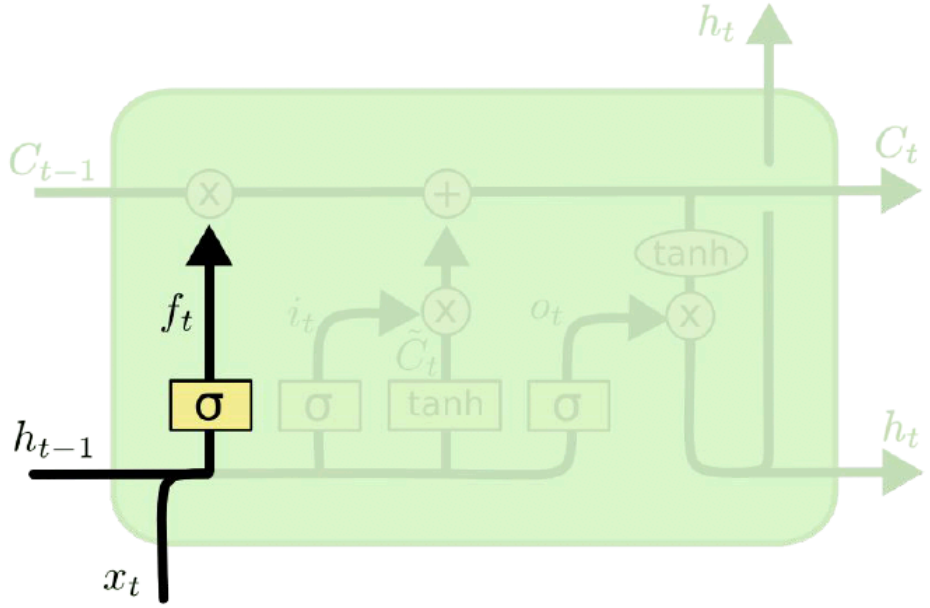



# LSTM – Step-by-Step

- The **Forget gate** is computed from $x_t$ and $h_{t-1}$:

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$

- $\sigma$ is the sigmoid function (bounded between 0 and 1).
- The Forget gate allows the LSTM to delete information from its memory.



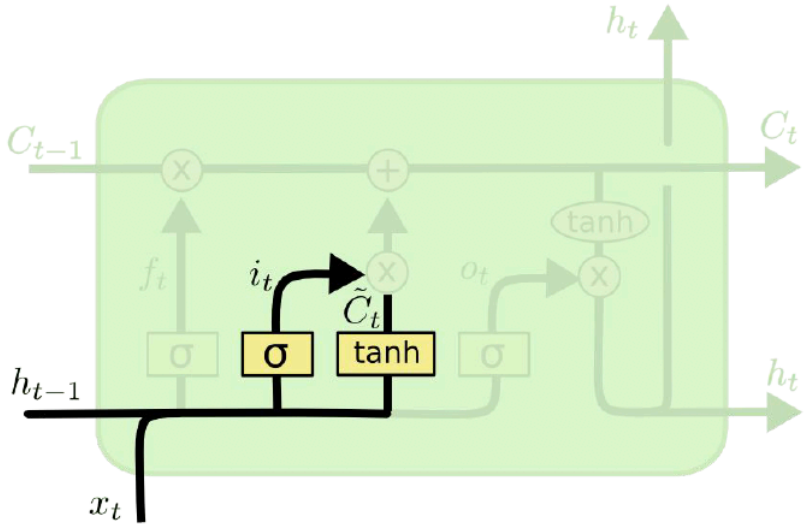
# LSTM – Step-by-Step

- The **Input gate** is computed from $x_t$ and $h_{t-1}$:

$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$

- The Input gate controls how much is added to the memory cell.
- The **candidate value** controls what is added to the memory.

$$g_t = tanh(U_g x_t + W_g h_{t-1} + b_g)$$

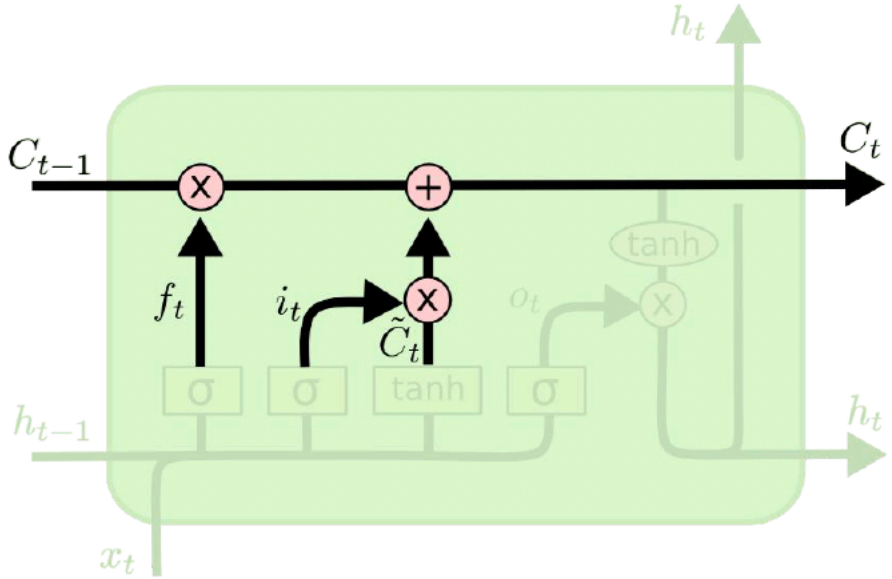


# LSTM – Step-by-Step

- The **Memory cell** is updated using the Input gate and the Forget gate:

$$C_t = i_t \times g_t + f_t \times C_{t-1}$$

- X = element-wise multiplication.
- The Input gate controls the amount of information added to the cell, and the Forget gate controls the amount of information deleted from the cell.



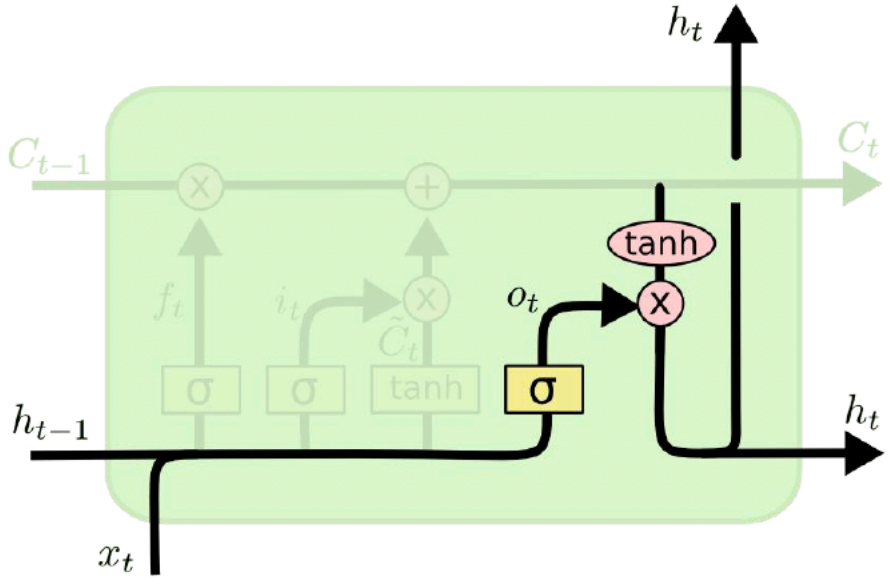
# LSTM – Step-by-Step

- The **Output gate** is computed from $x_t$ and $h_{t-1}$:

$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$$

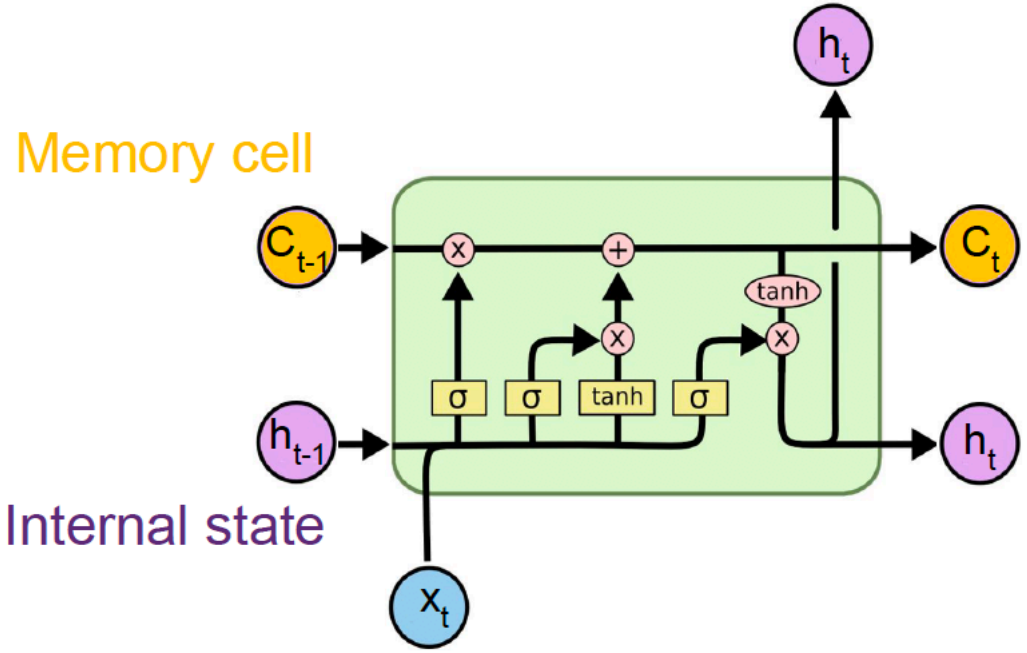
- The Output gate controls the output of the memory cell.
- The **new internal state** is computed as:

$$h_t = o_t \times tanh(C_t)$$



# Long Short-Term Memory (LSTM)

- Reducing the vanishing problem using a **gate mechanism** and adding a **memory cell**.




$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$
$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$
$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$$
$$g_t = tanh(U_g x_t + W_g h_{t-1} + b_g)$$
$$C_t = i_t \times g_t + f_t \times C_{t-1}$$
$$h_t = o_t \times tanh(c_t)$$

Image from Christopher Olah's blog
Hochreiter et al., Long short-term memory, Neural Computation 1997

# Gated Recurrent Unit (GRU)

- A popular variant of the LSTM.
- No dedicated memory cell.
- Input and Forget gates are combined.
- In practice, it provides results similar to an LSTM.
- Faster to compute.

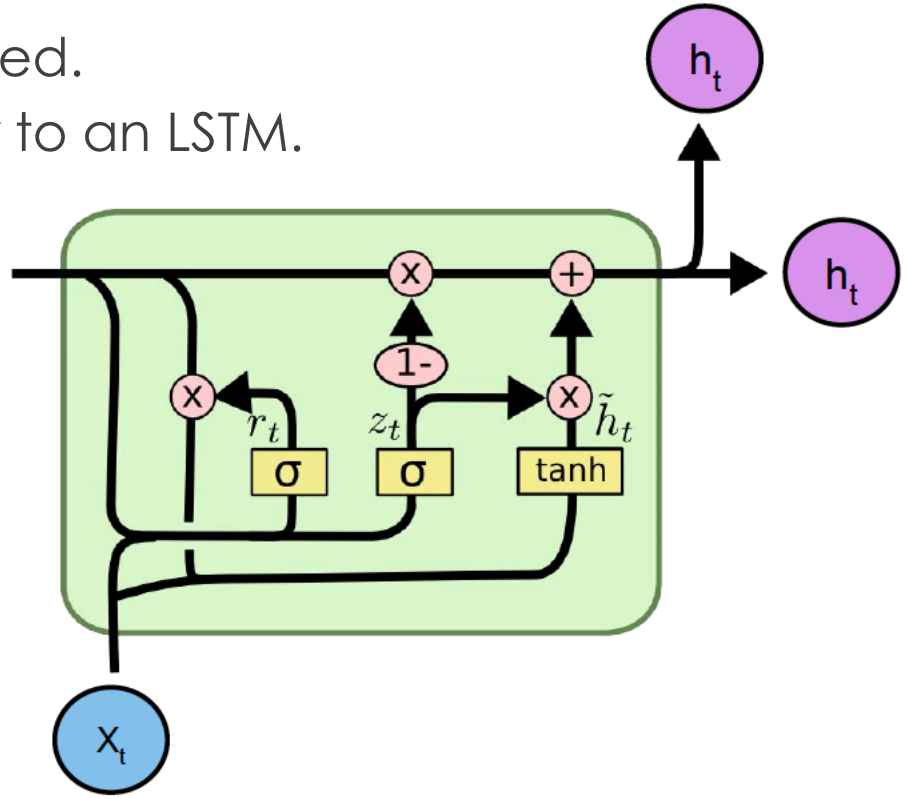



Image from Christopher Olah's blog
Chung et al.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling

# Gated Recurrent Unit (GRU)

$$z_t = \sigma(U_z x_t + W_z h_{t-1} + b_z)$$ ← Update gate

$$r_t = \sigma(U_r x_t + W_r h_{t-1} + b_r)$$ ← Reset gate

$$g_t = tanh(U_g x_t + W_g(r_t \times h_{t-1}) + b_g)$$

$$h_t = z_t \times g_t + (1 - z_t) \times h_{t-1}$$ ← Internal state

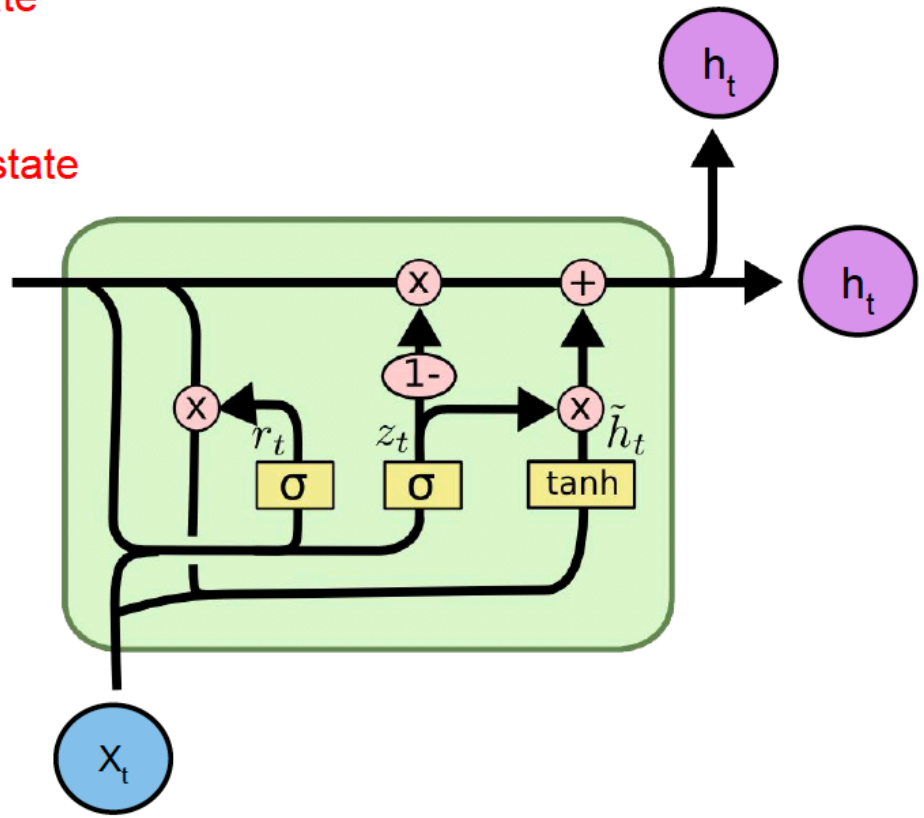



Image from Christopher Olah's blog

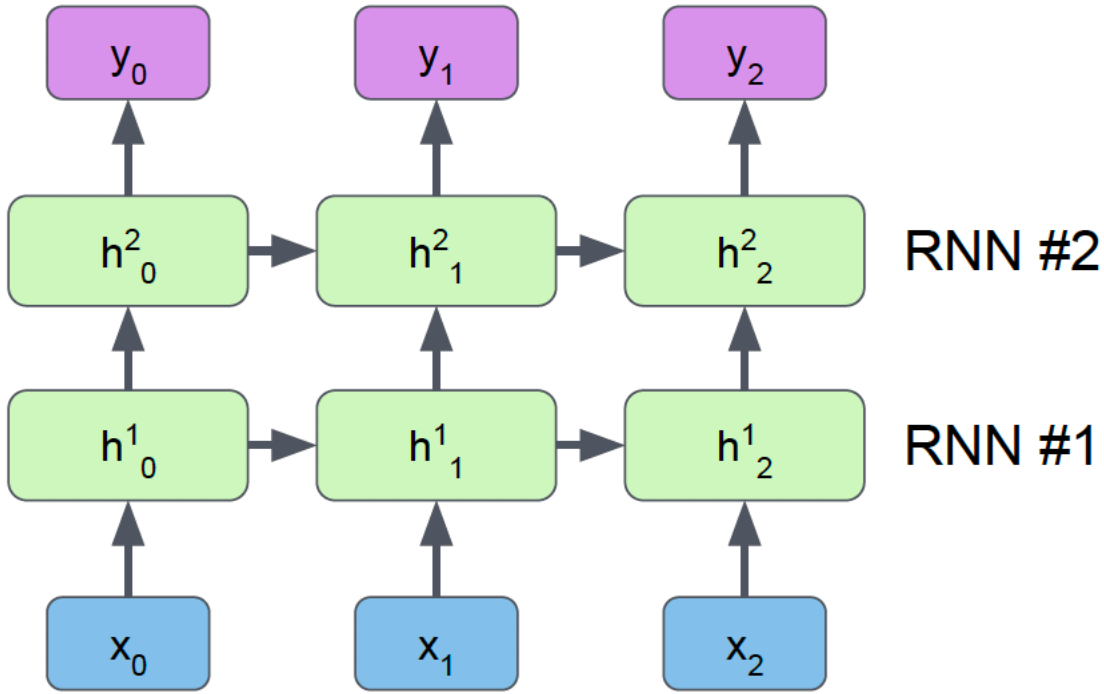
Chung et al.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling

# Topics

- ***Training Problems***
- ***RNN Architectures***
- Deep RNNs
- Sequence to Sequence Models
- Attention Mechanism

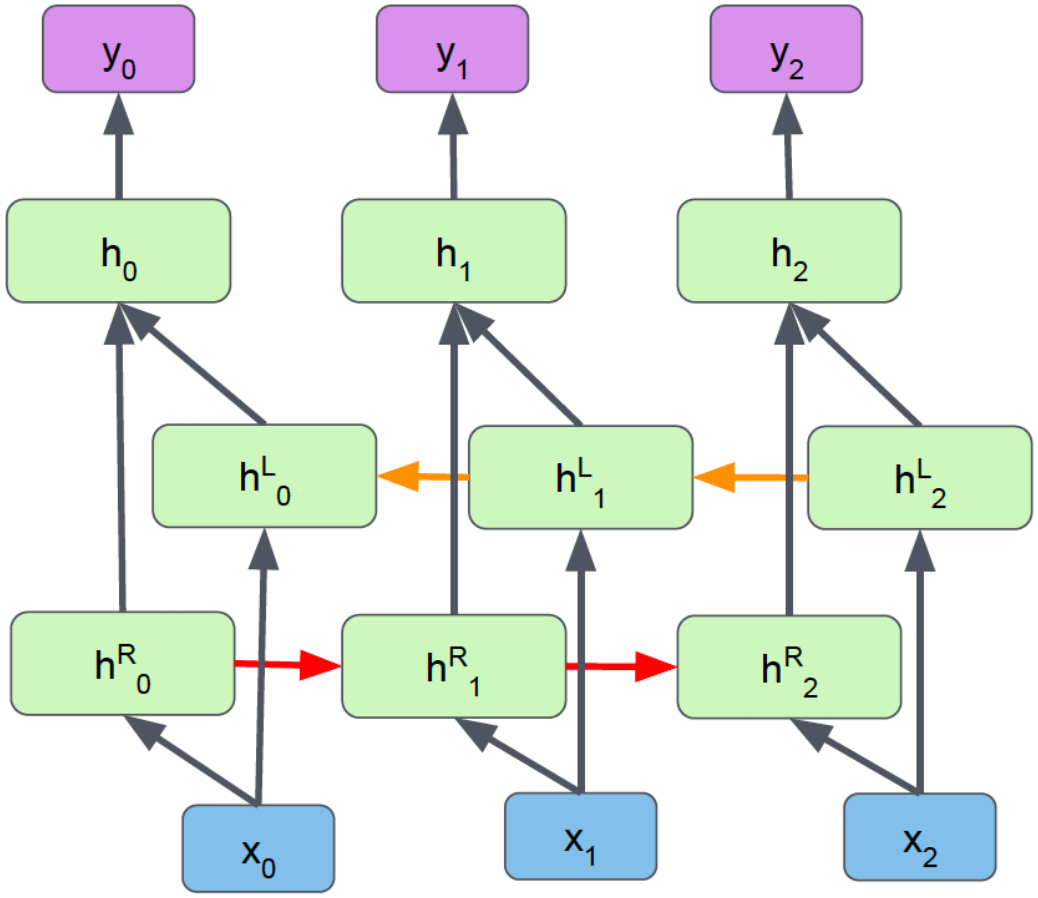
# Deep RNNs

- To create deep RNNs, one can stack several RNN layers.
- The output of the first layer is the input to the second layer, etc.
- Every layer has a different set of parameters.



# Bidirectional RNNs

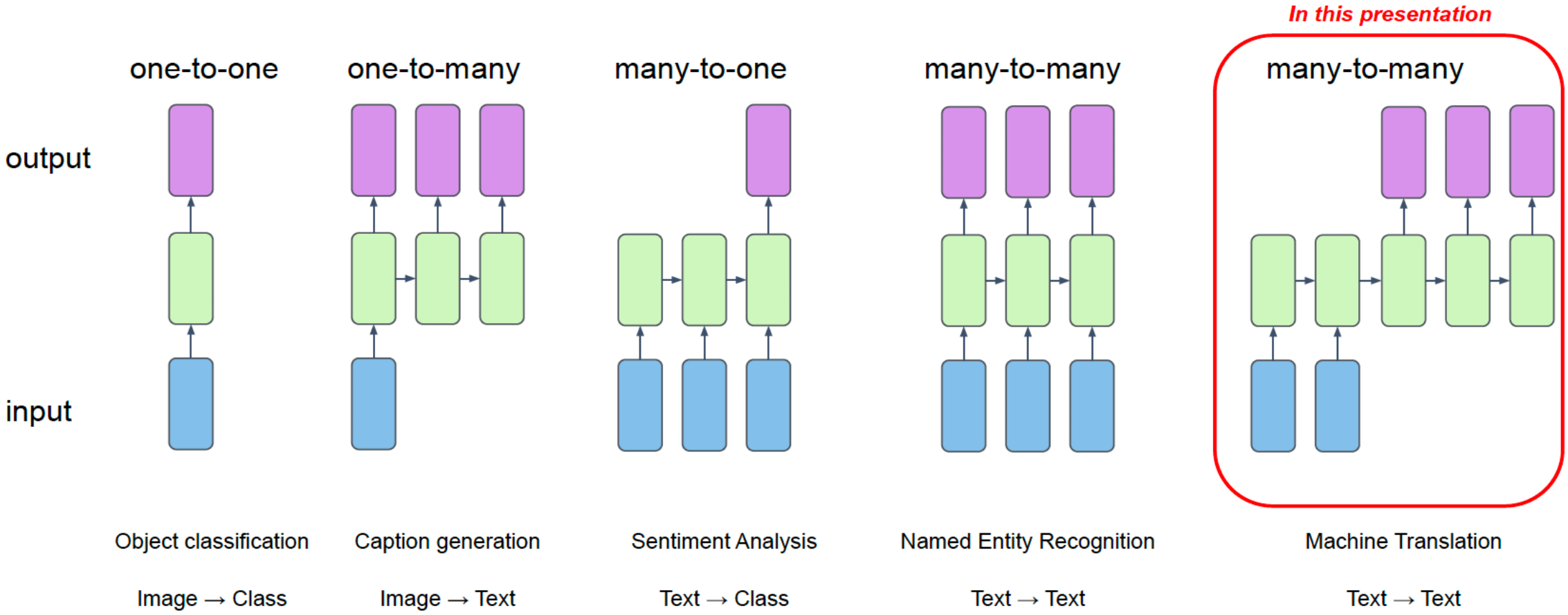
- Use two RNNs: one operating left-to-right, one operating right-to-left.
- This allows to look at information coming from the "past" and "future".
- The two RNNs are different (different parameters).
- The two RNN outputs ($h^R_t$,$h^L_t$) can be "merged" ($h_t$) in various ways: concatenation, sum, ...



# Topics

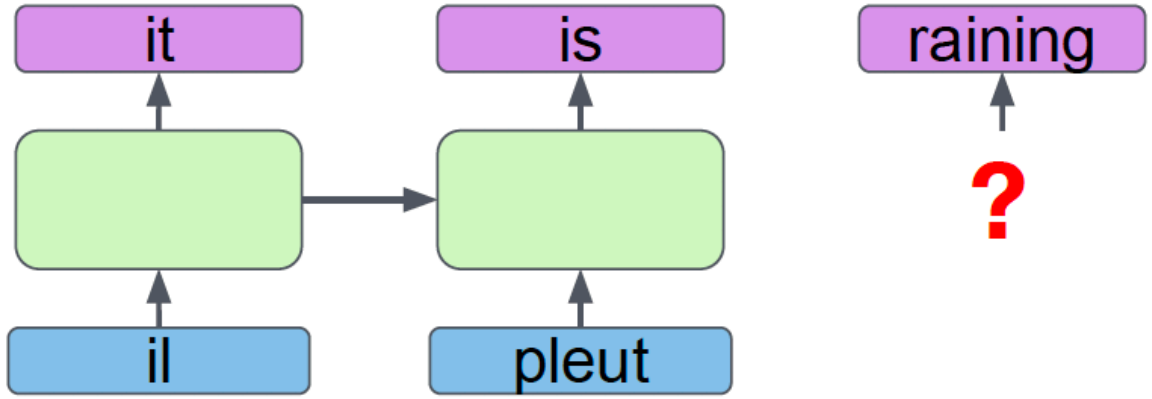
- ***Training Problems***
- ***RNN Architectures***
- ***Deep RNNs***
- Sequence to Sequence Models
- Attention Mechanism

# Modeling Sequences



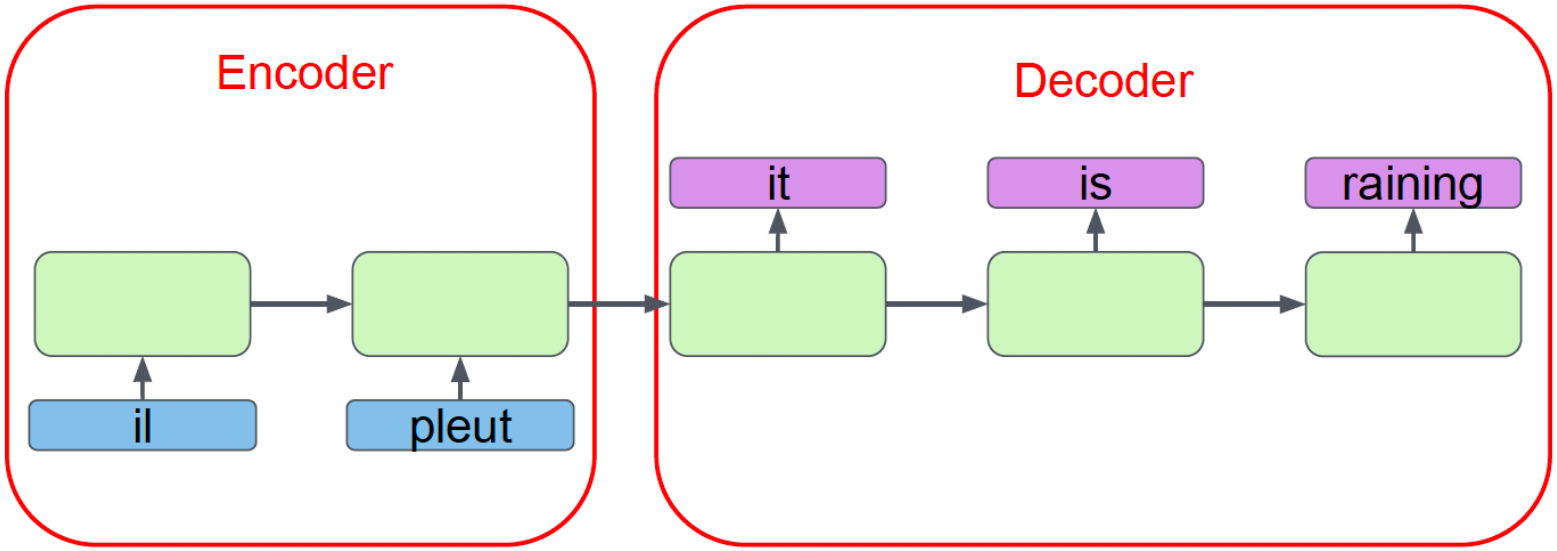
# Modeling Sequences

- How to handle input and output sequences of different lengths?
  - Machine translation.
  - Text summarization.
  - …



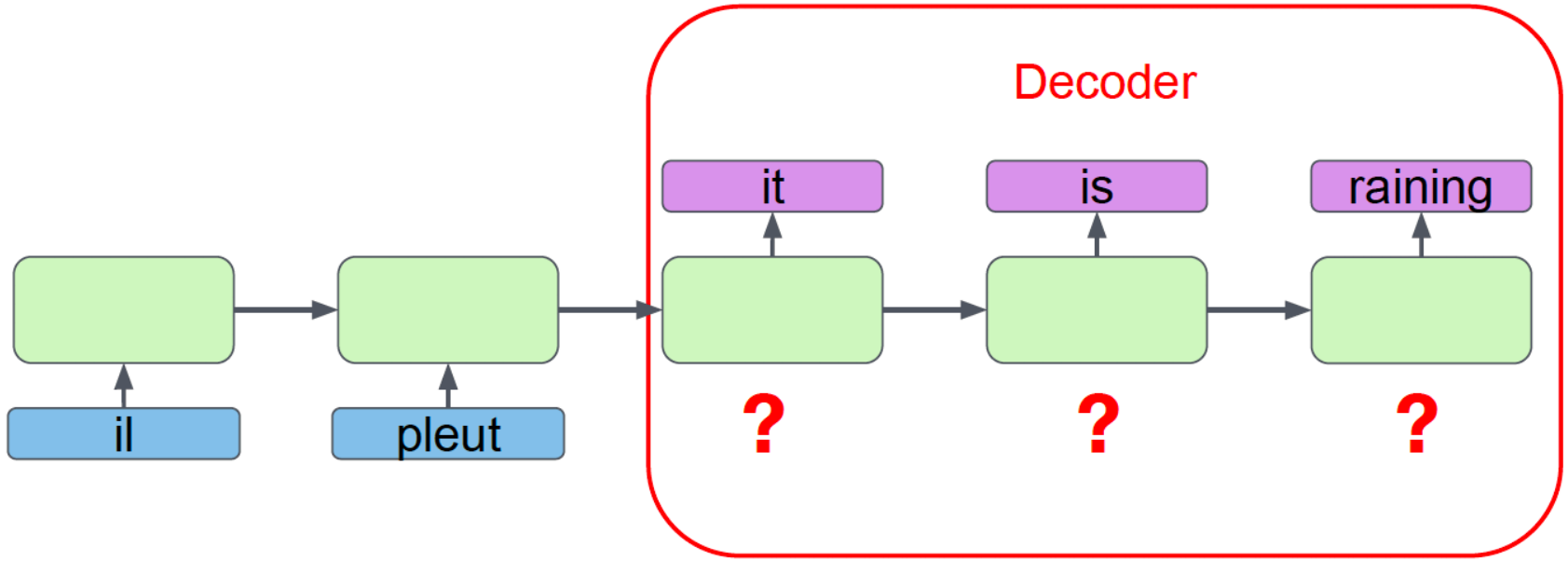
# Different input-output sequence size

- Create an architecture composed of two components (e.g. two different RNNs):
  - Encoder that processes the input sequence.
  - Decoder that generates the output sequence based on the encoded input.



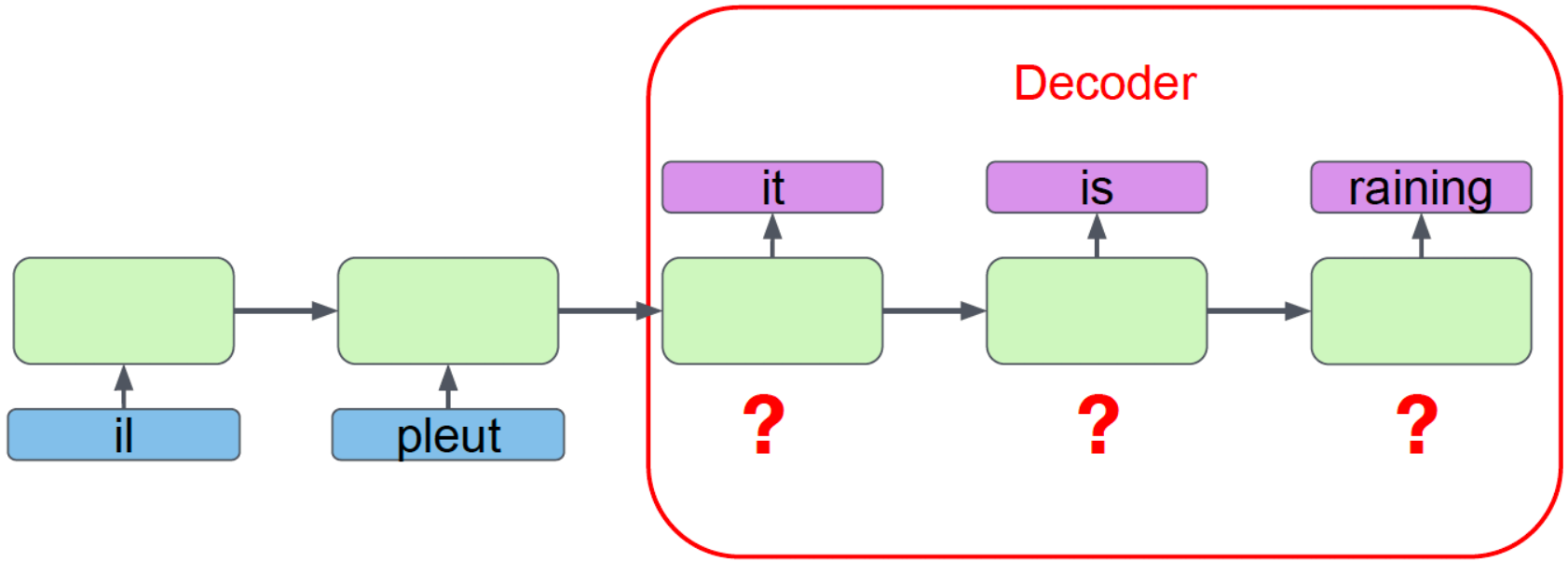
# Different input-output sequence size

- How to implement the decoder?
- Note the missing input. We need a mechanism that will allow the decoder to generate consistent outputs across time.



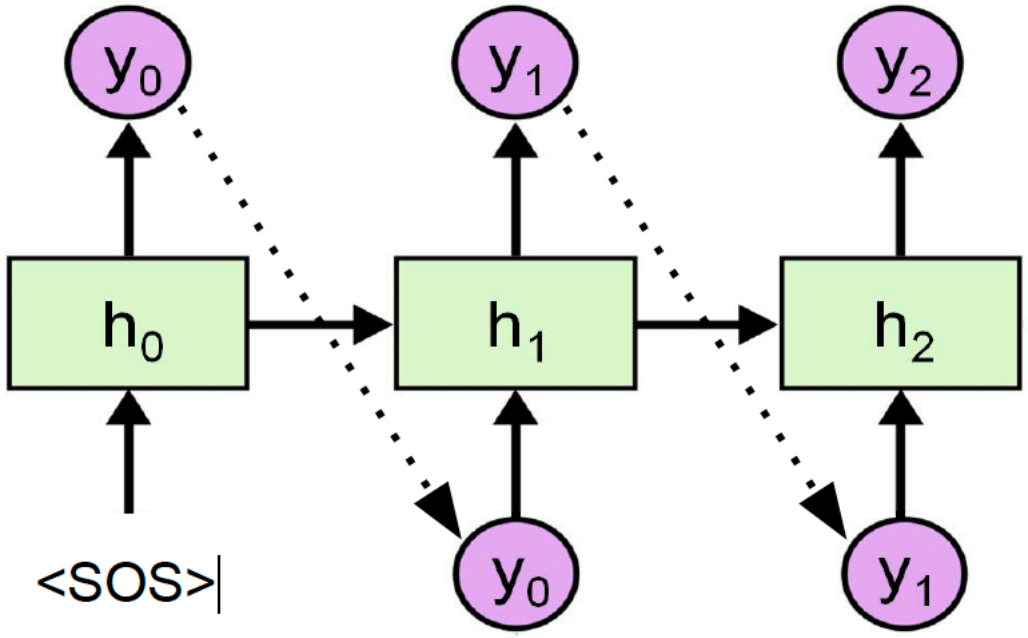
# Different input-output sequence size

- Example: knowing that the decoder generated “it” at time step 0 increases the likelihood of generating “is” at time step 1



# Autoregressive RNNs

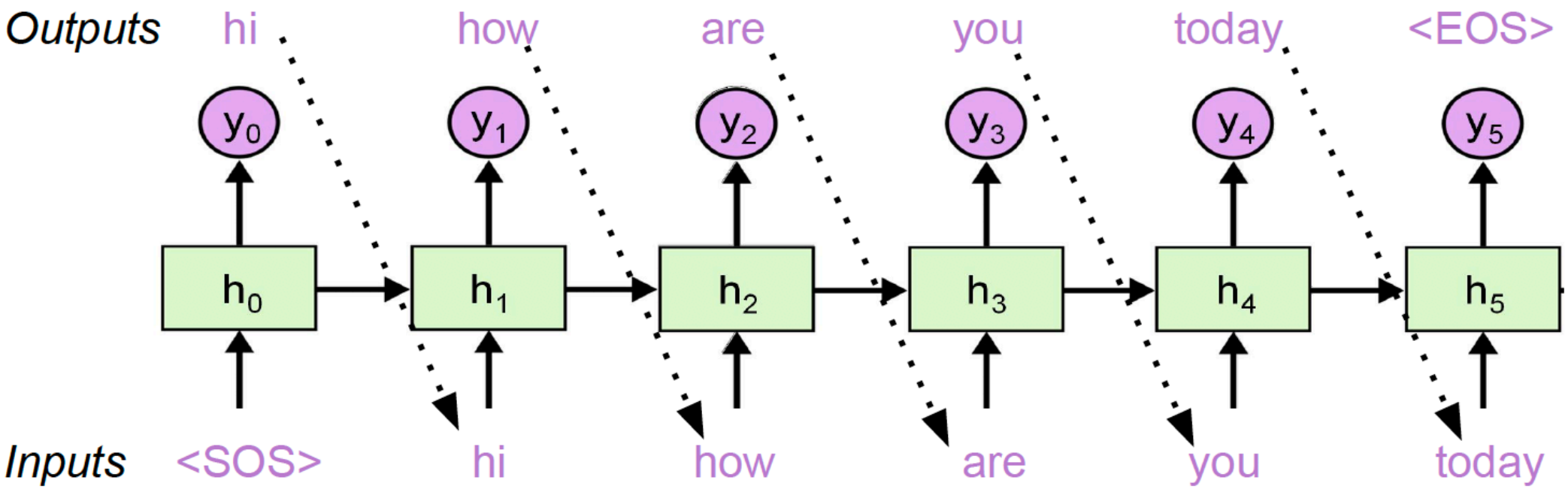
- We can use a RNN to generate a sequence.
- In order to generate consistent outputs across time, we can condition each output on previously generated outputs.
- Such a model is called autoregressive.



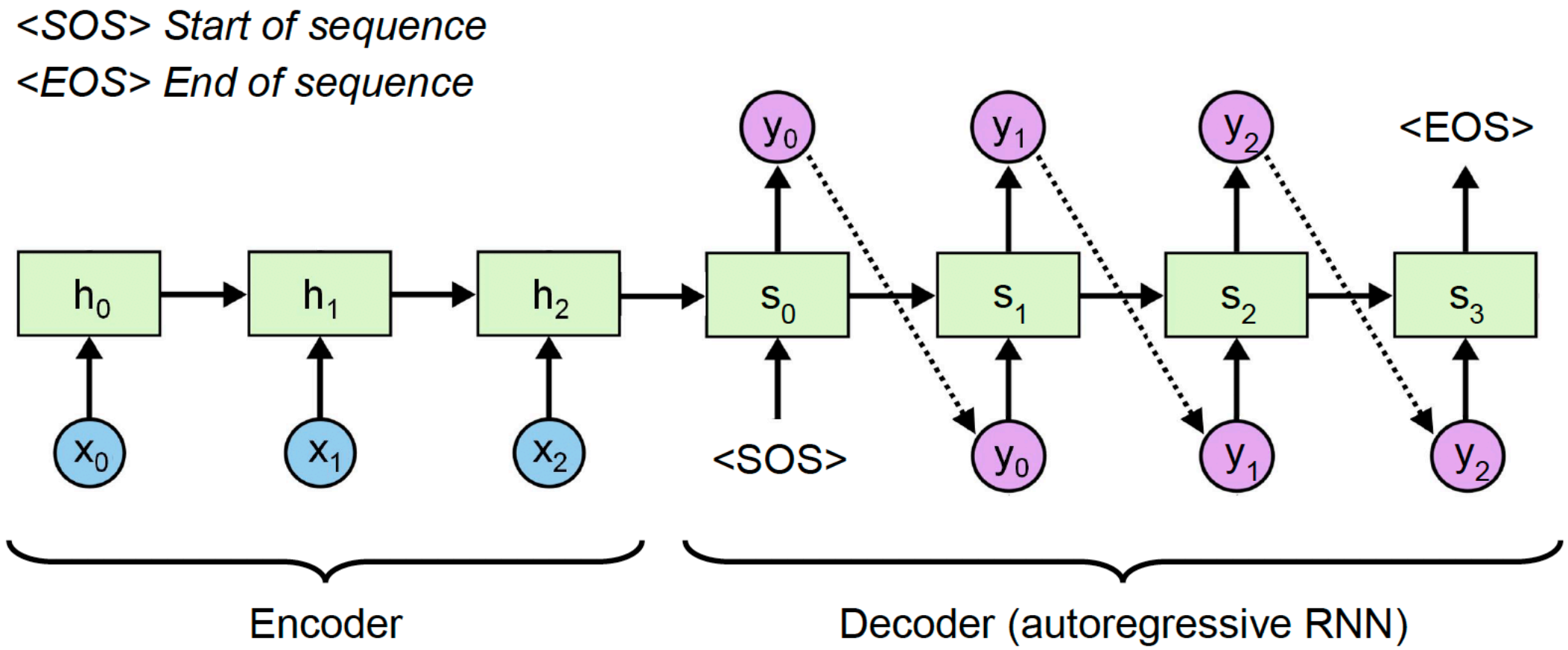

*<SOS> Start of sequence*

# Autoregressive RNNs

<SOS> Start of sequence
<EOS> End of sequence



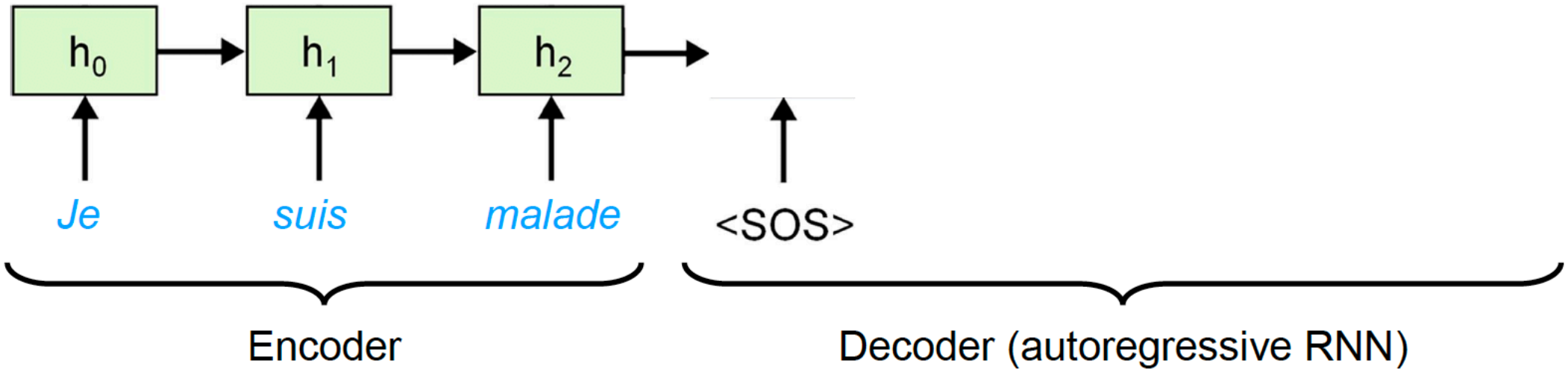
# Sequence-to-Sequence Models




Sutskever et al., Sequence to Sequence Learning with Neural Networks
Cho et al., Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation

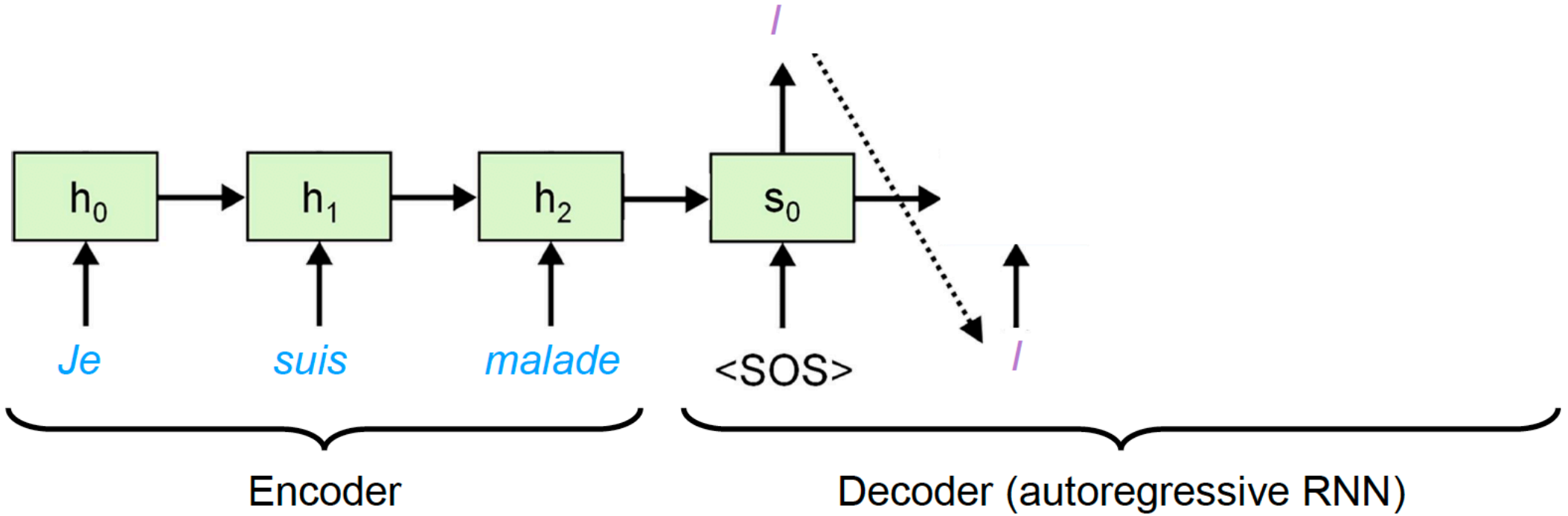
# Sequence-to-Sequence Models - Example

<SOS> Start of sequence
<EOS> End of sequence

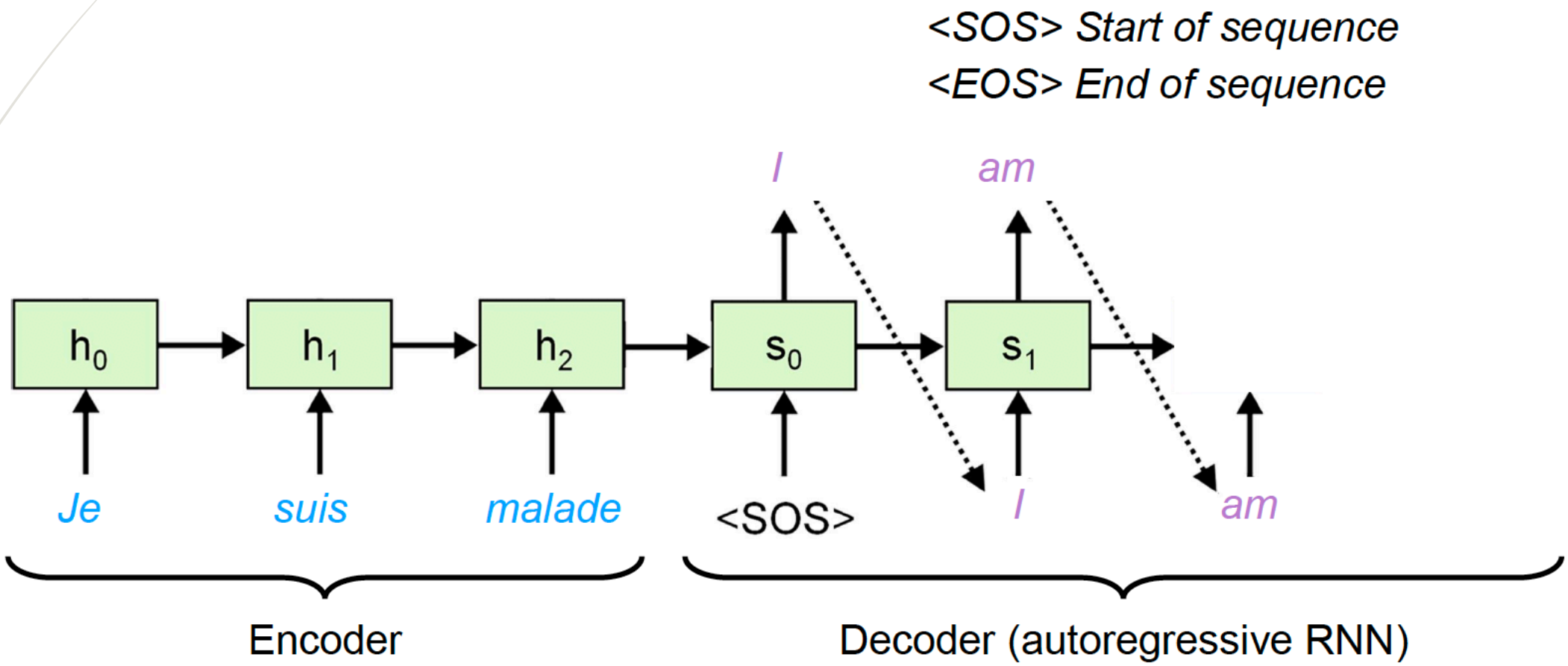


# Sequence-to-Sequence Models - Example

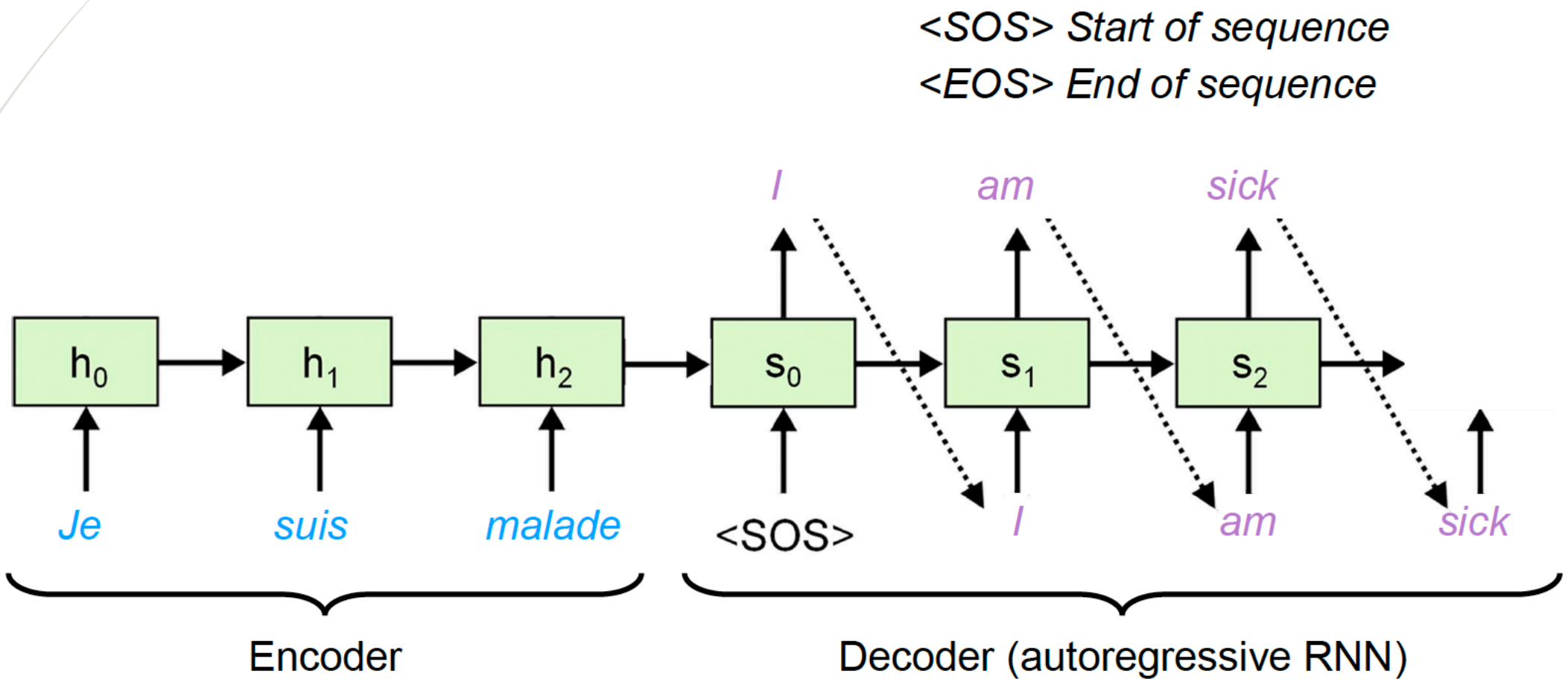
<SOS> Start of sequence
<EOS> End of sequence



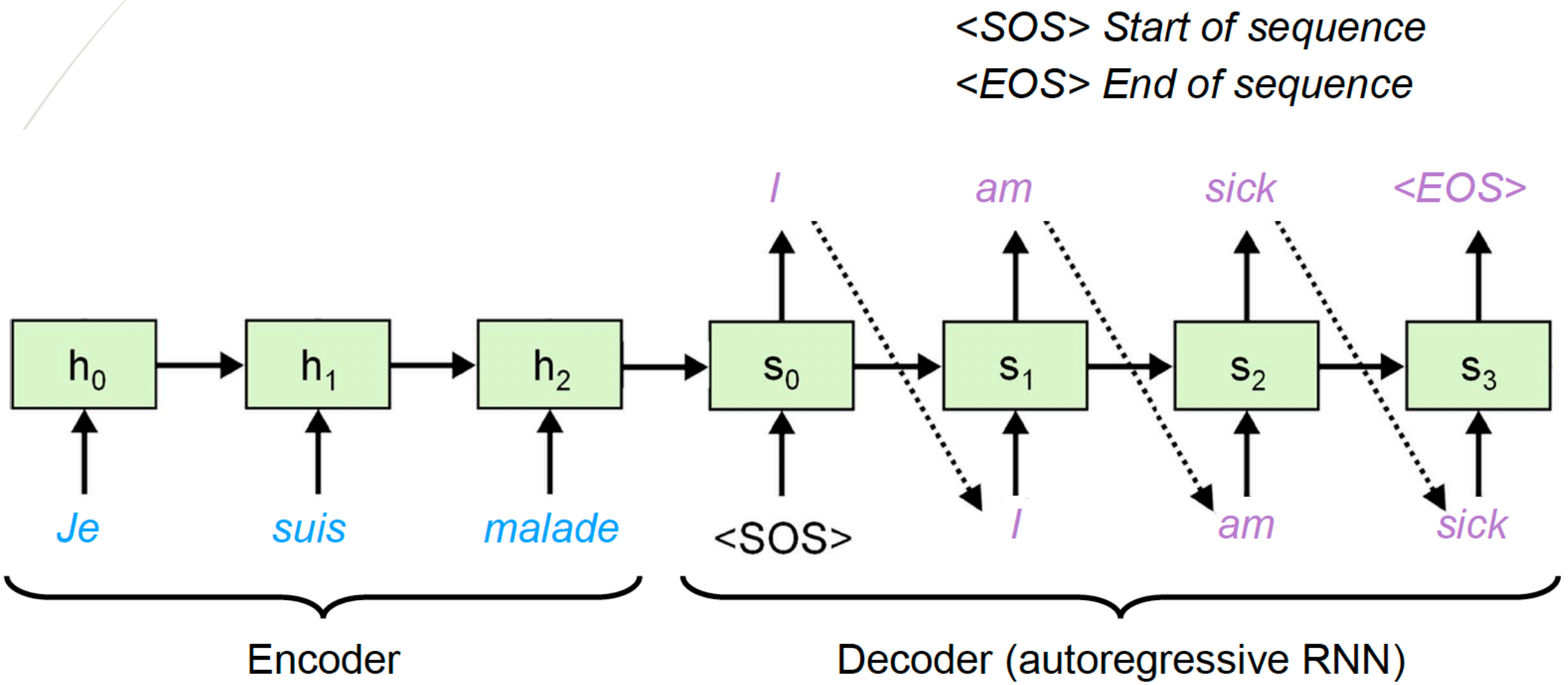
# Sequence-to-Sequence Models - Example



# Sequence-to-Sequence Models - Example



# Sequence-to-Sequence Models - Example



# Topics

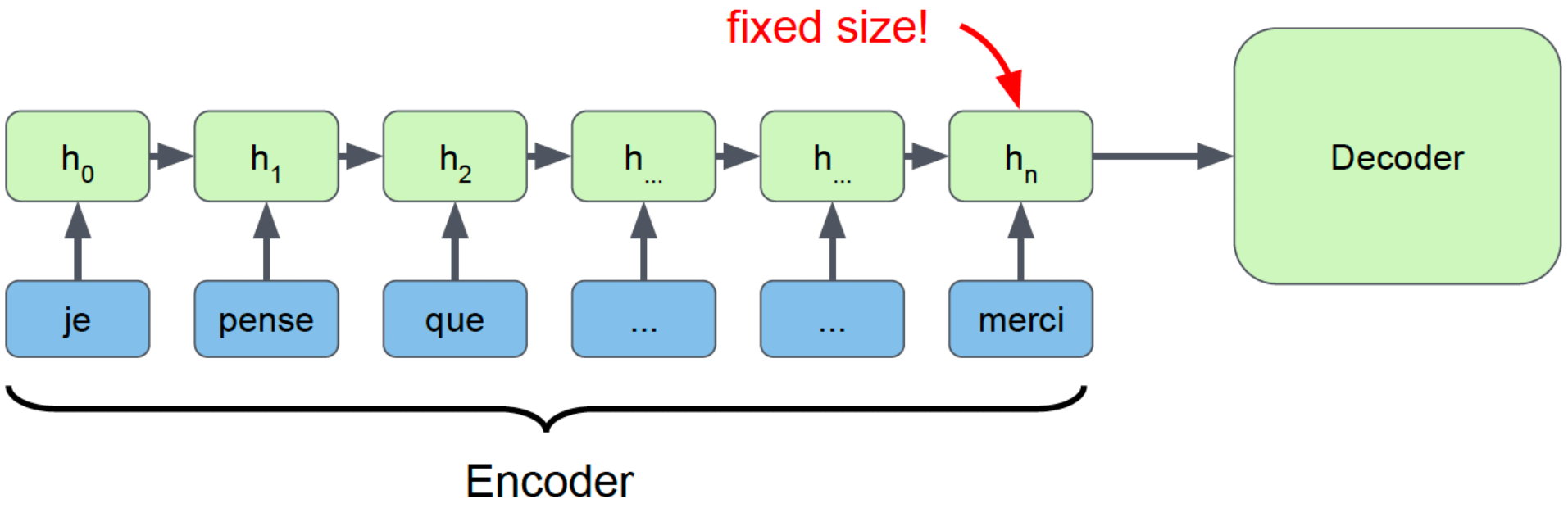
- ***Training Problems***
- ***RNN Architectures***
- ***Deep RNNs***
- ***Sequence to Sequence Models***
- Attention Mechanism

# Sequence-to-Sequence Models Bottleneck

- The encoder has to store/compress all the information from the input into a fixed size vector ($h_2$ in this example).

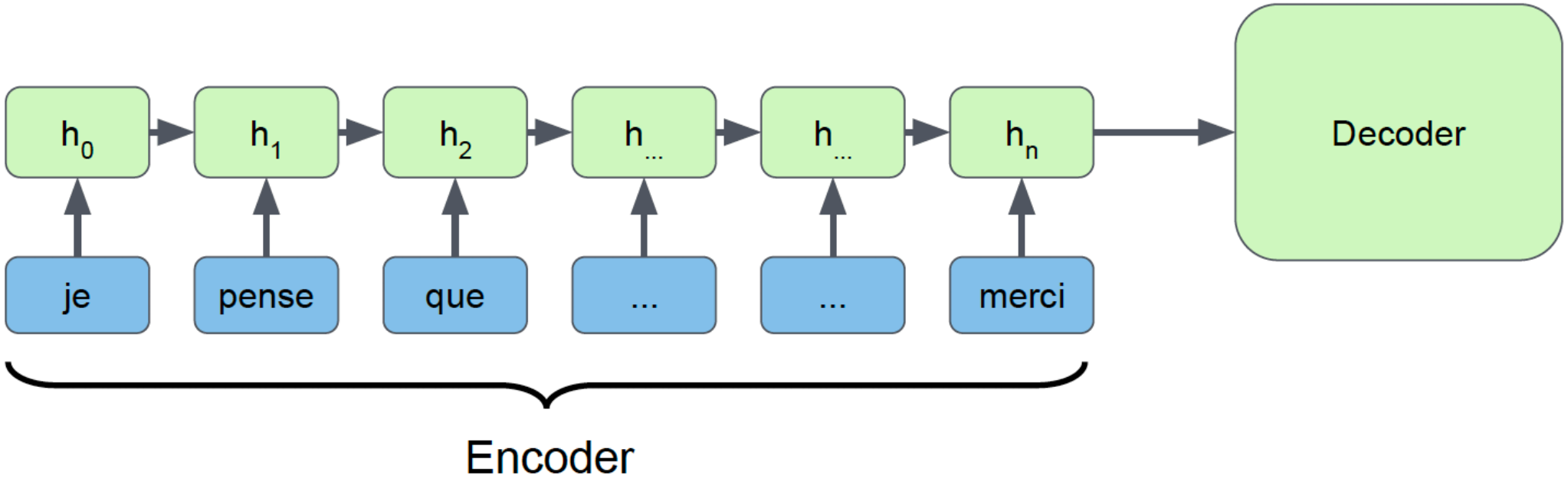
# Sequence-to-Sequence Models Bottleneck

- This is not easy to do with very long input sequences.
- $h_n$ is a bottleneck.



# Attention Mechanism

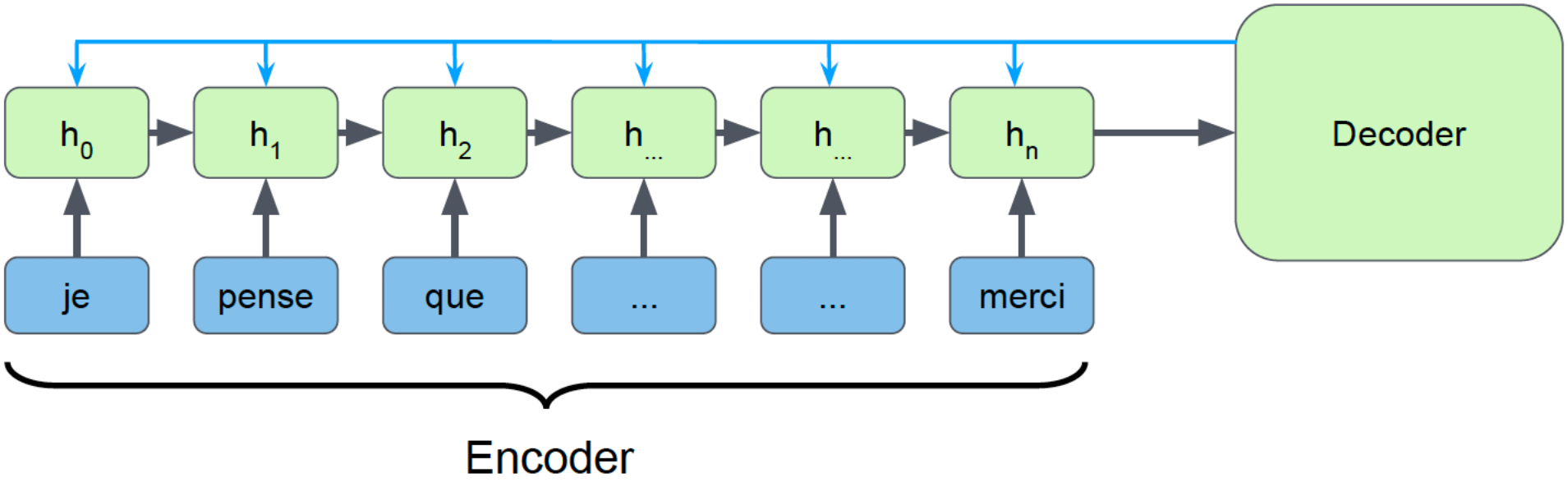
- Problem: it is not easy to store all the necessary information from an **arbitrary long** sequence into a **fixed-size** vector.




Bahdanau et al., Neural Machine Translation by Jointly Learning to Align and Translate

# Attention Mechanism

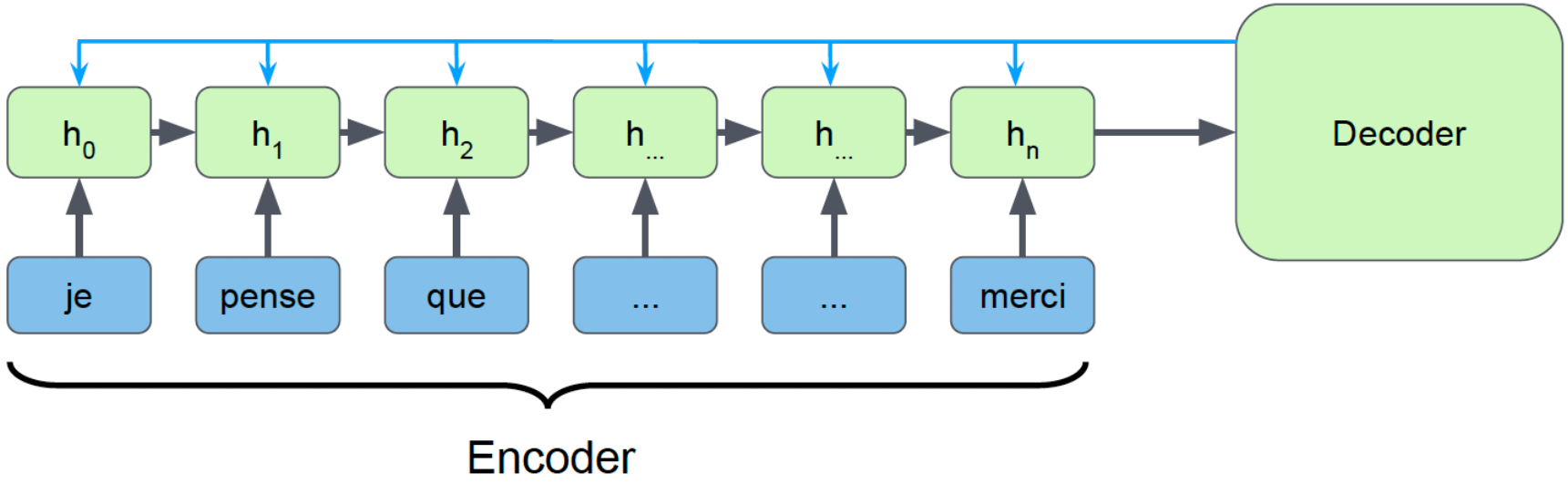
- Problem: it is not easy to store all the necessary information from an **arbitrary long** sequence into a **fixed-size** vector.
- A possible solution can be to allow the decoder to "selectively look back" at the encoded input sequence.




Bahdanau et al., Neural Machine Translation by Jointly Learning to Align and Translate

# Attention Mechanism

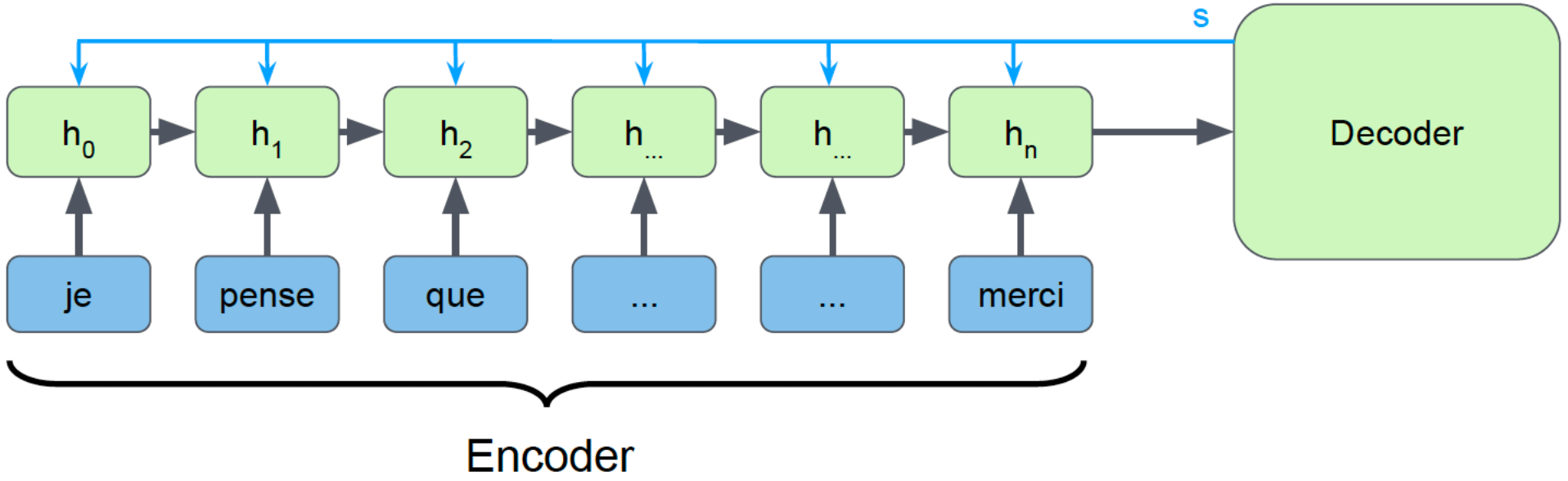
- This can be done with attention:
  - At any decoding time step, the decoder can use attention to fetch the relevant information for that step from the encoded input sequence.
- E.g., when producing the output word “think” (in a machine translation task), the decoder can focus on the encoding of the input word “pense”.




Bahdanau et al., Neural Machine Translation by Jointly Learning to Align and Translate

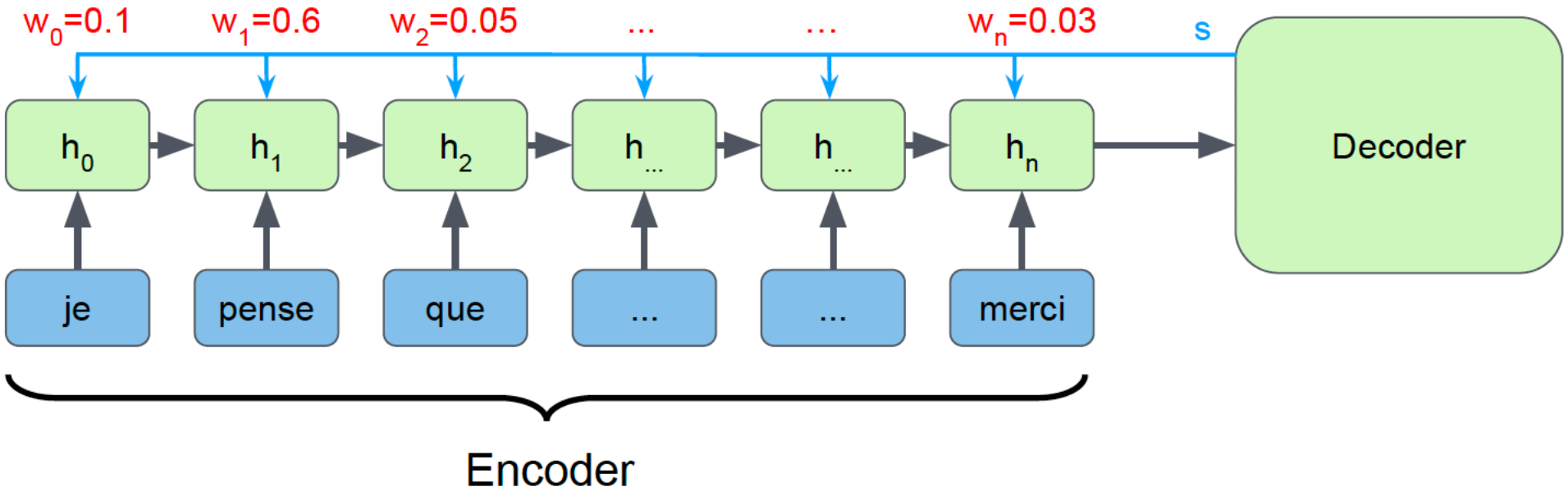
# Attention Mechanism - Formalization

- Attention is a function **A** that, given a decoder state **s** and an encoded input sequence **h**, identifies the elements in **h** that are important at the current decoding time step.



# Attention Mechanism - Formalization

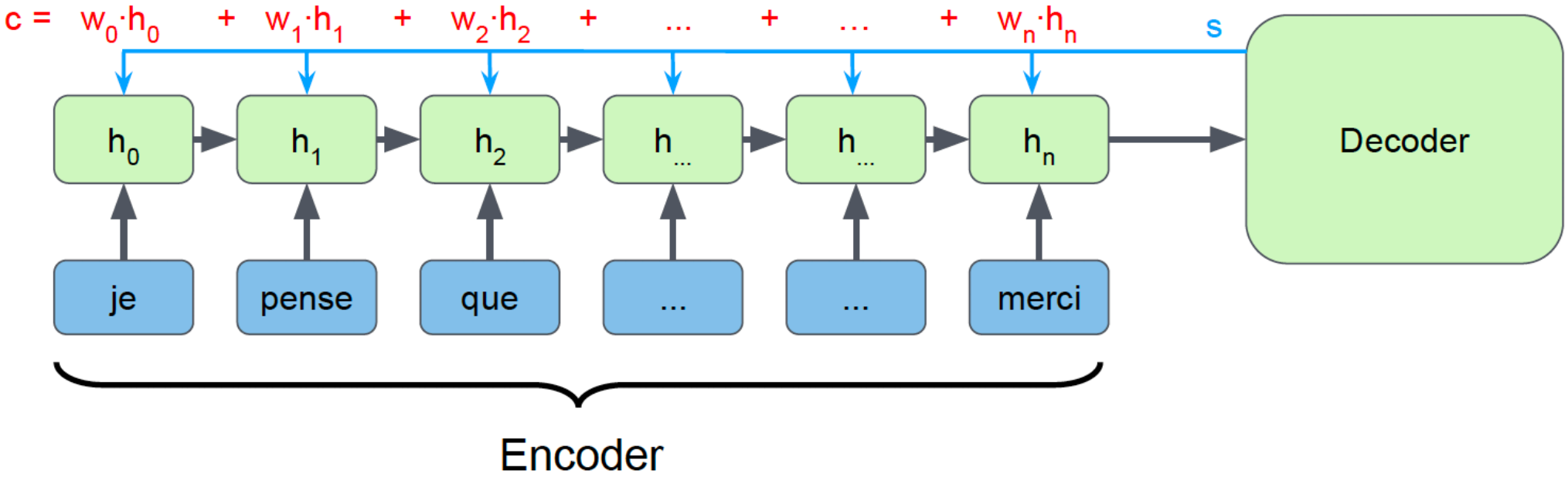
- Attention is a function **A** that, given a decoder state **s** and an encoded input sequence **h**, identifies the elements in **h** that are important at the current decoding time step.
  - A assigns weights w to the elements in h.
  - Those weights are normalized (to sum to 1).



# Attention Mechanism - Formalization

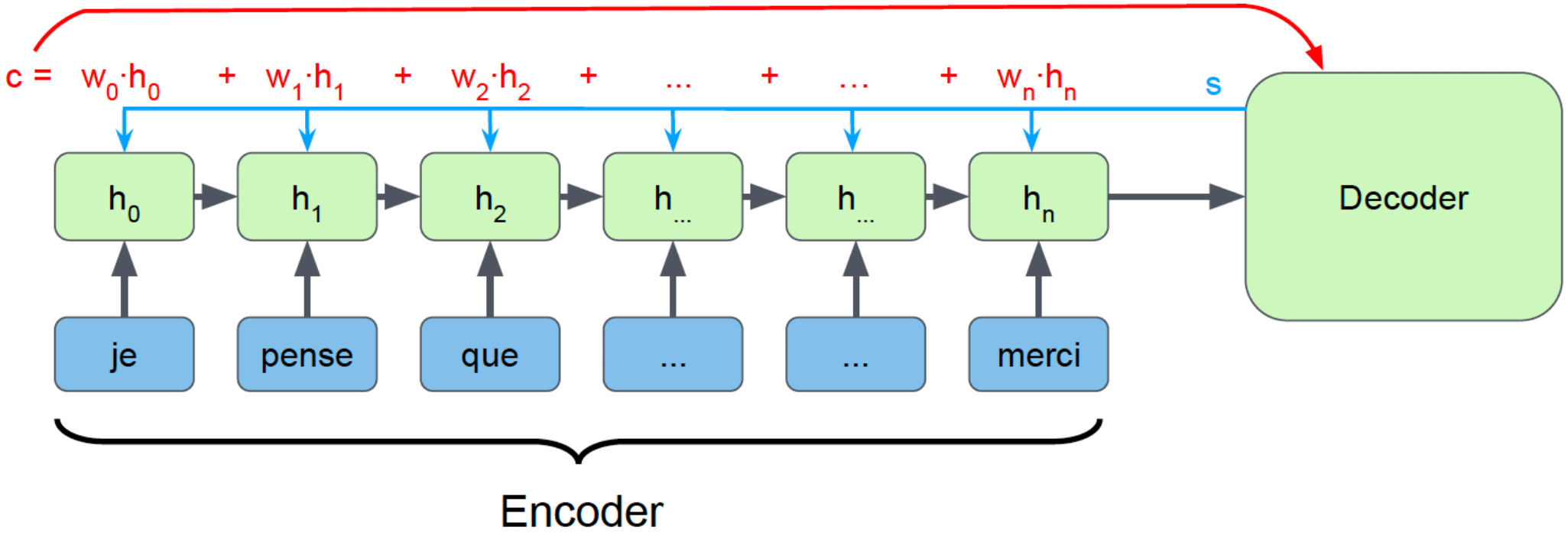
- The weights **w** are used to compute a weighted sum c of the elements in the sequence **h. c** is called **context vector**.

$$c = \sum_{i=0}^{n} w_i \cdot h_i$$



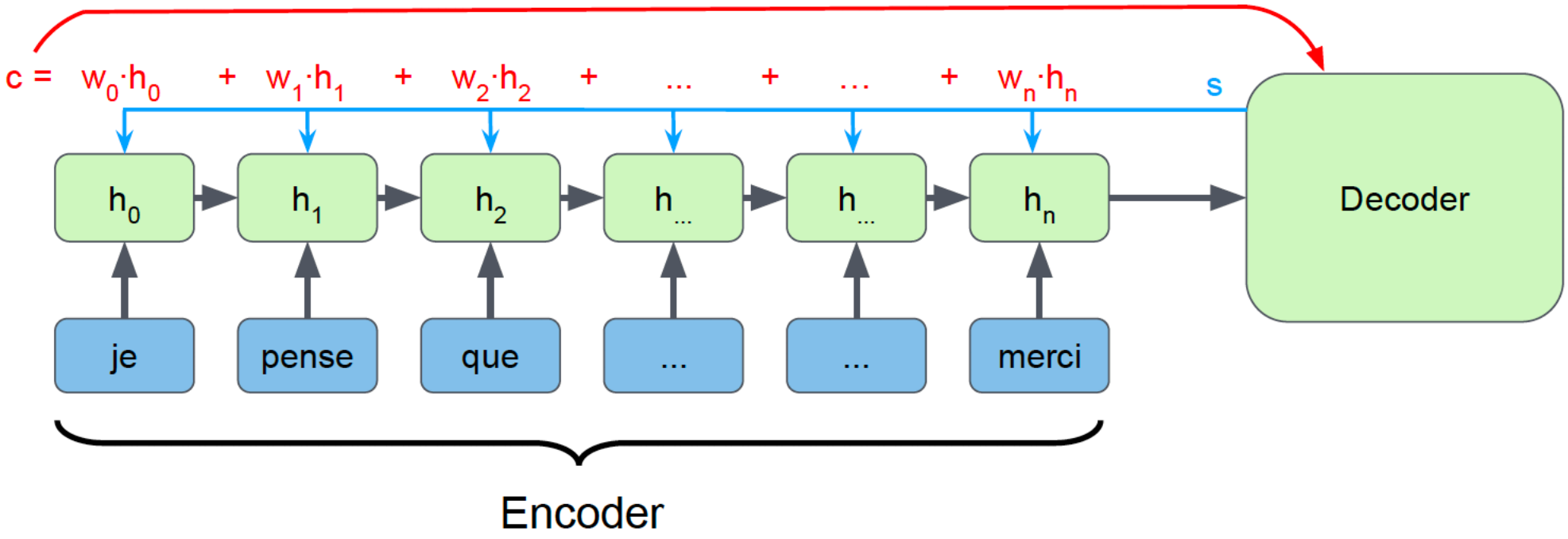
# Attention Mechanism - Formalization

- The context vector **c** is then fed to the decoder.



# Attention Mechanism - Formalization

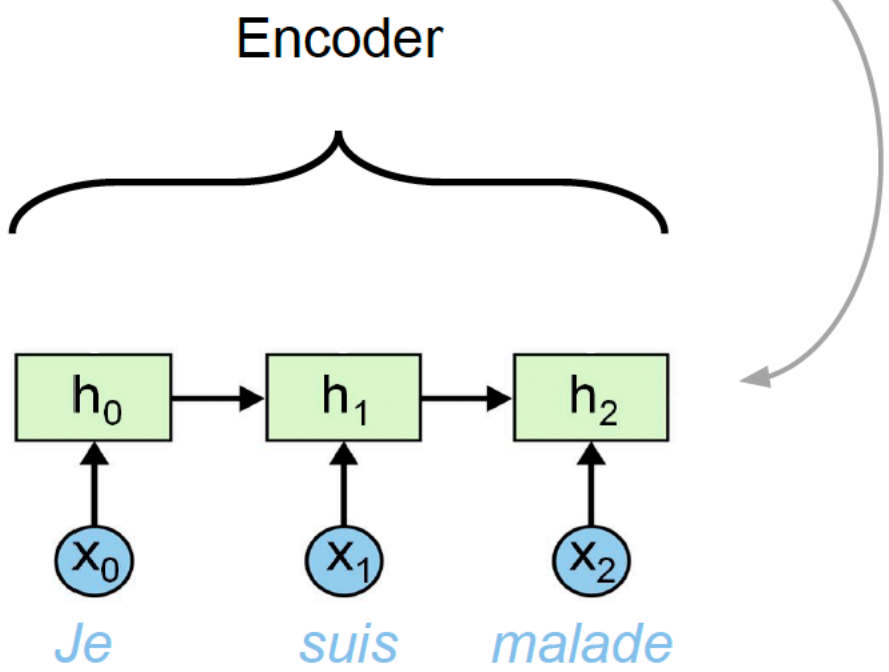
- Let's see a full step-by-step example of the attention mechanism.
- We will then look at how to implement the function **A**.



# Example: Sequence-to-Sequence + Attention

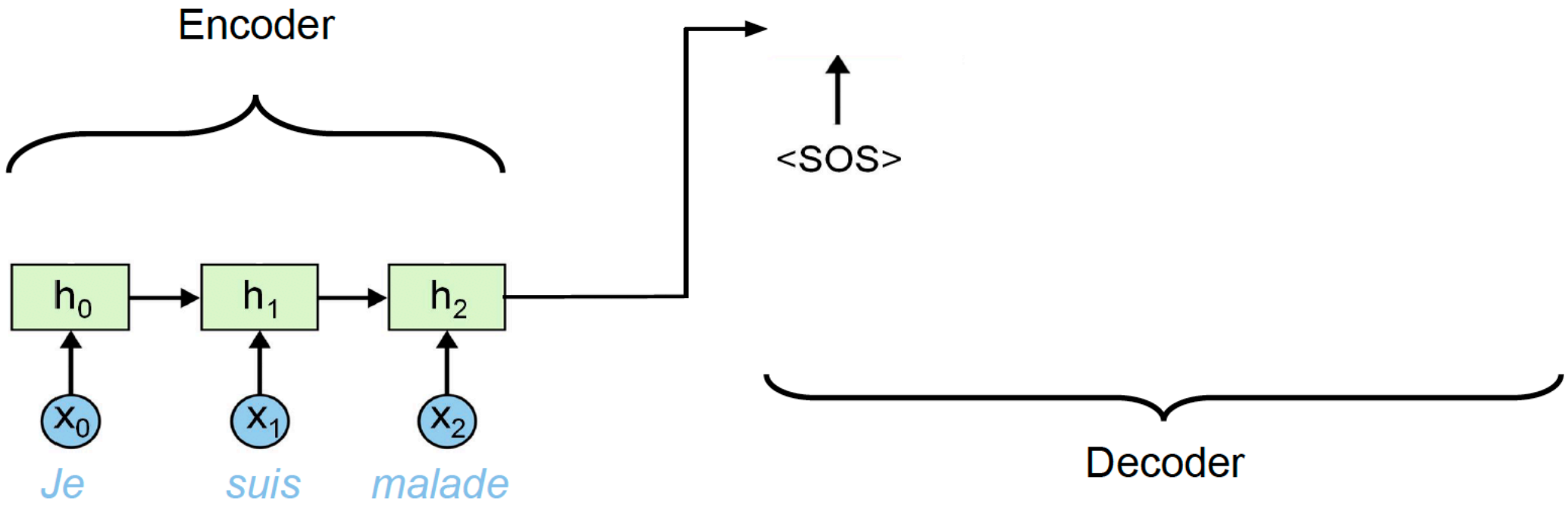
# Example: Sequence-to-Sequence + Attention

All the x sequence is encoded into a vector of **fixed size** (h2).



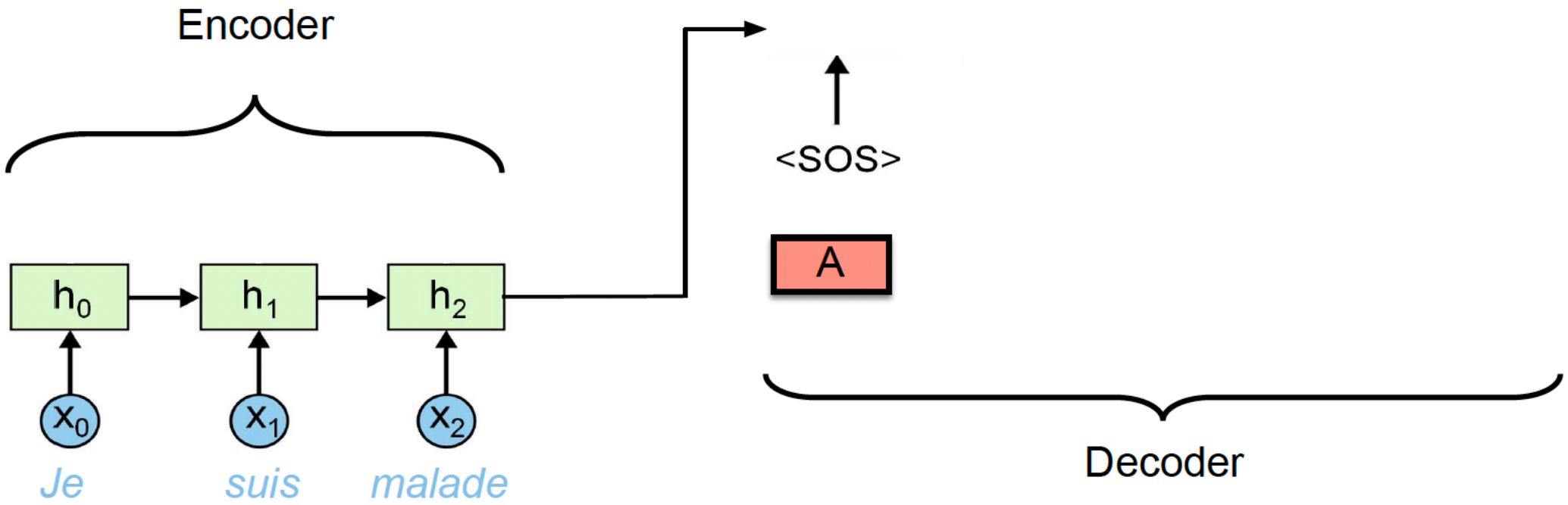
# Example: Sequence-to-Sequence + Attention

The decoder starts with the **<SOS>** symbol.



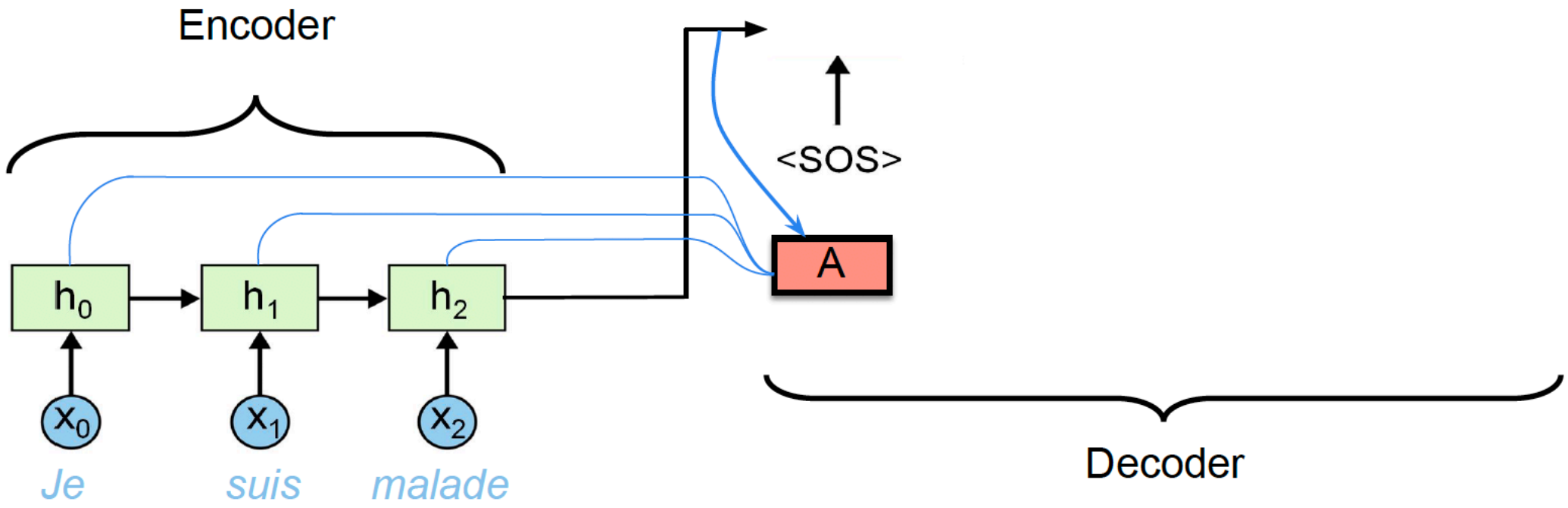
# Example: Sequence-to-Sequence + Attention

The attention model **A** is added to the decoder.



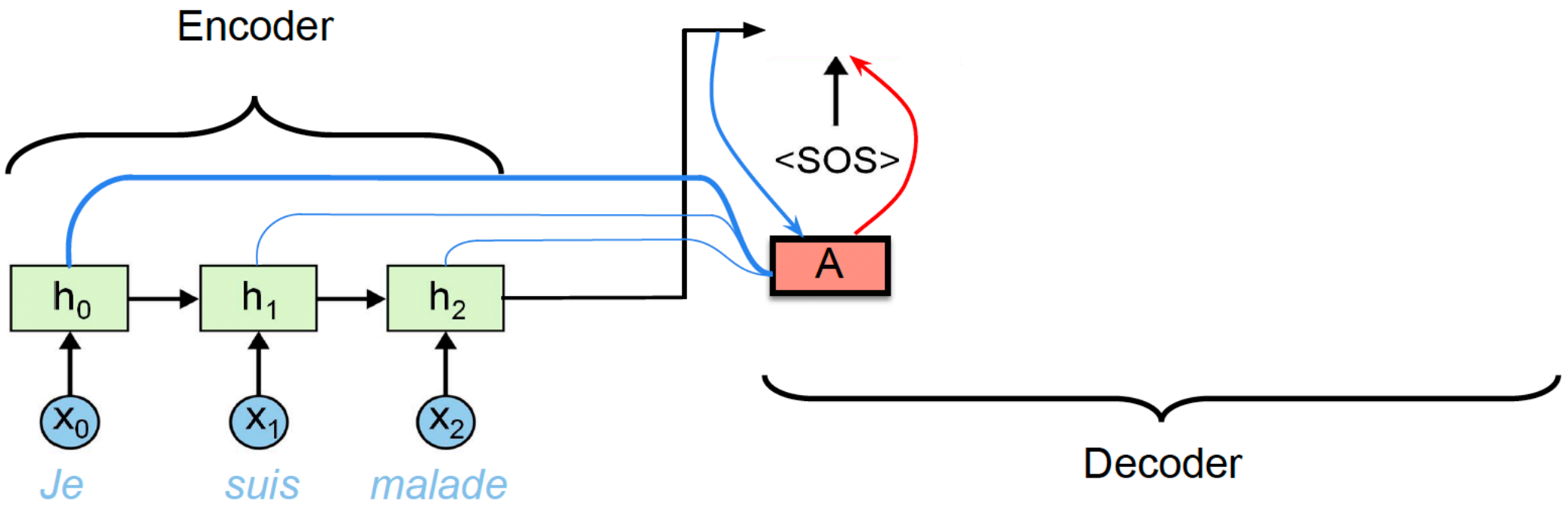
# Example: Sequence-to-Sequence + Attention

The decoder's previous state and the encoded input sequence **h** are fed as inputs to the attention model (**A**).



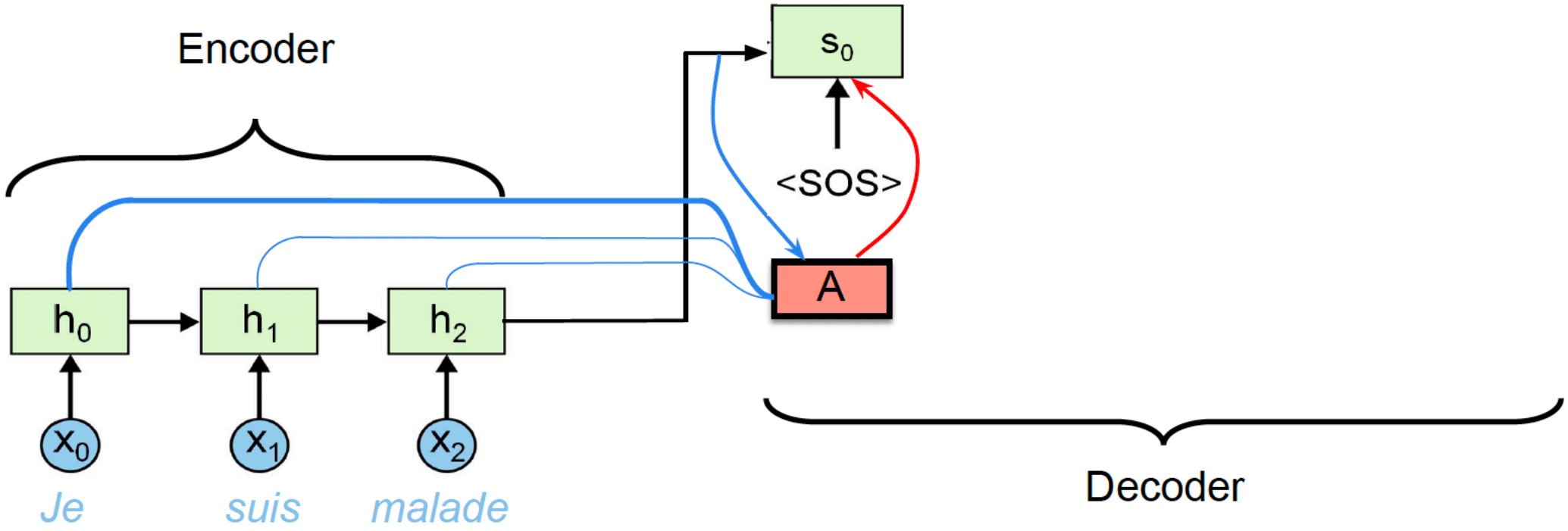
# Example: Sequence-to-Sequence + Attention

The context vector (output of the attention) is fed as an input to the decoder.



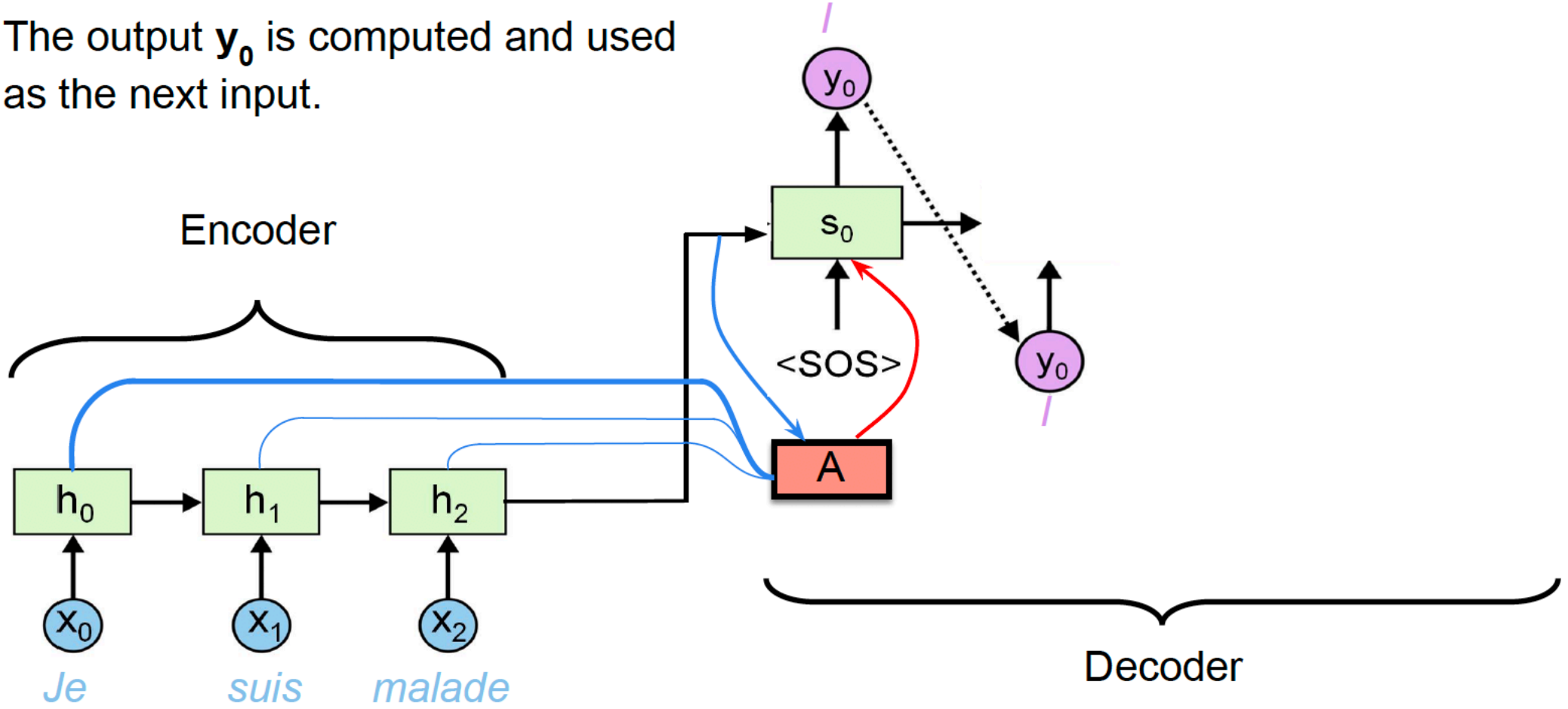
# Example: Sequence-to-Sequence + Attention

The internal state $\mathbf{s_0}$ is computed.

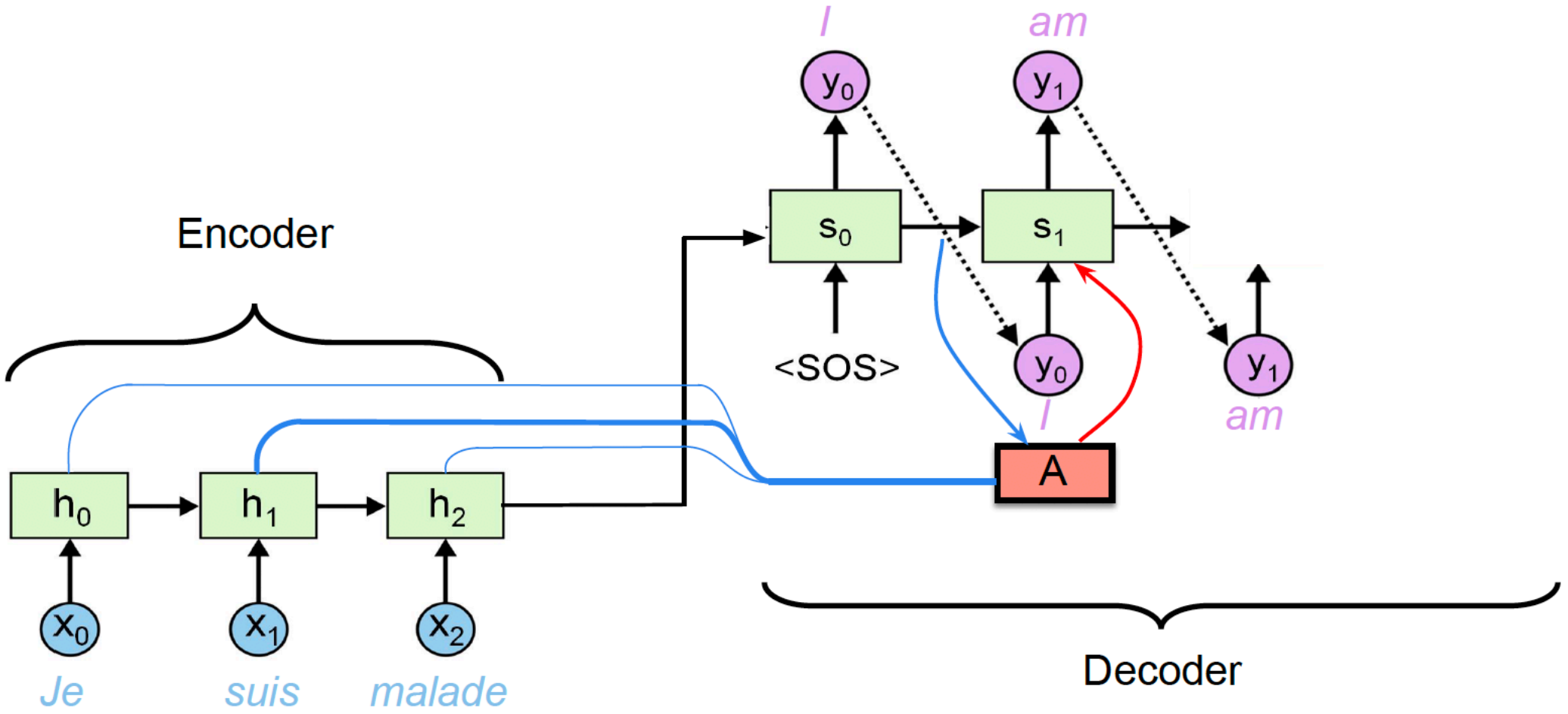


# Example: Sequence-to-Sequence + Attention

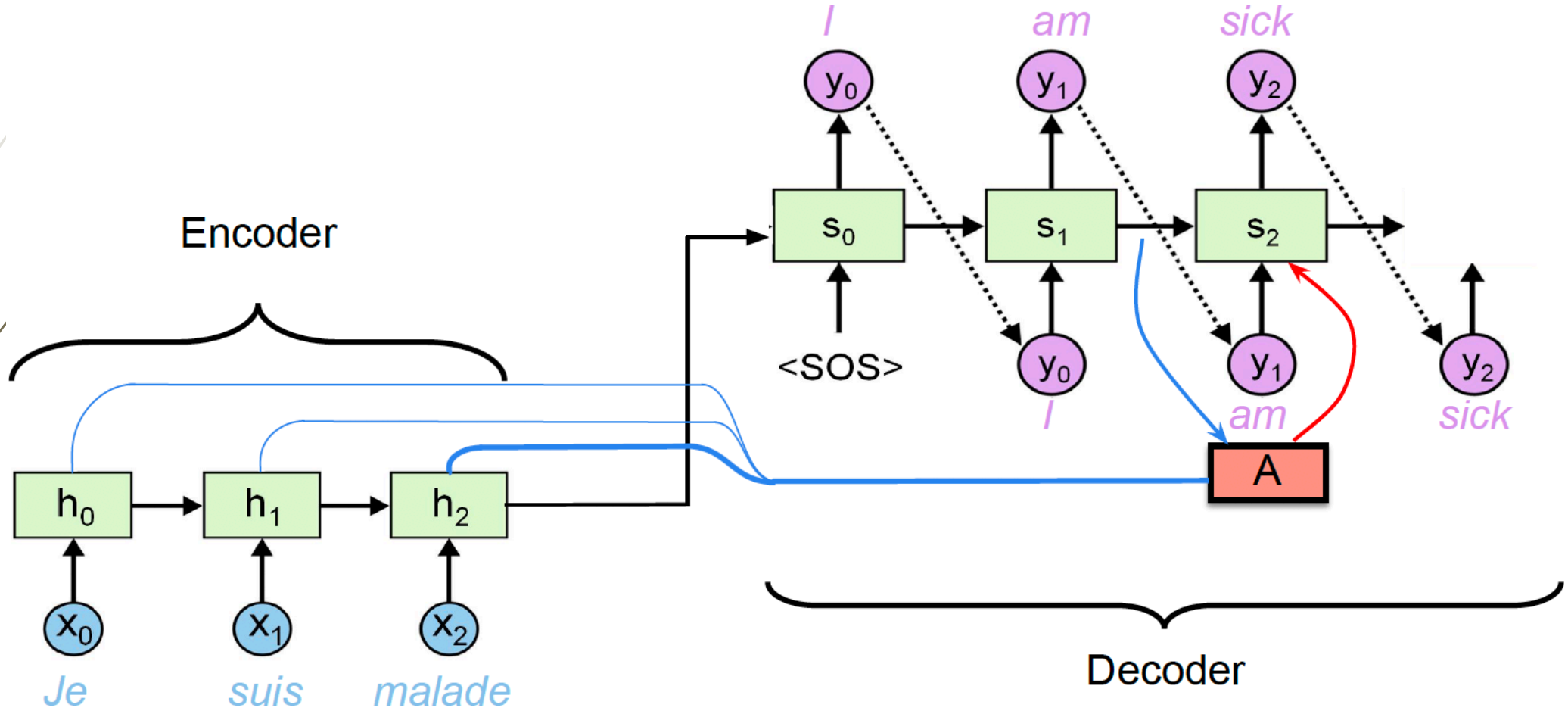
The output $\mathbf{y_0}$ is computed and used as the next input.



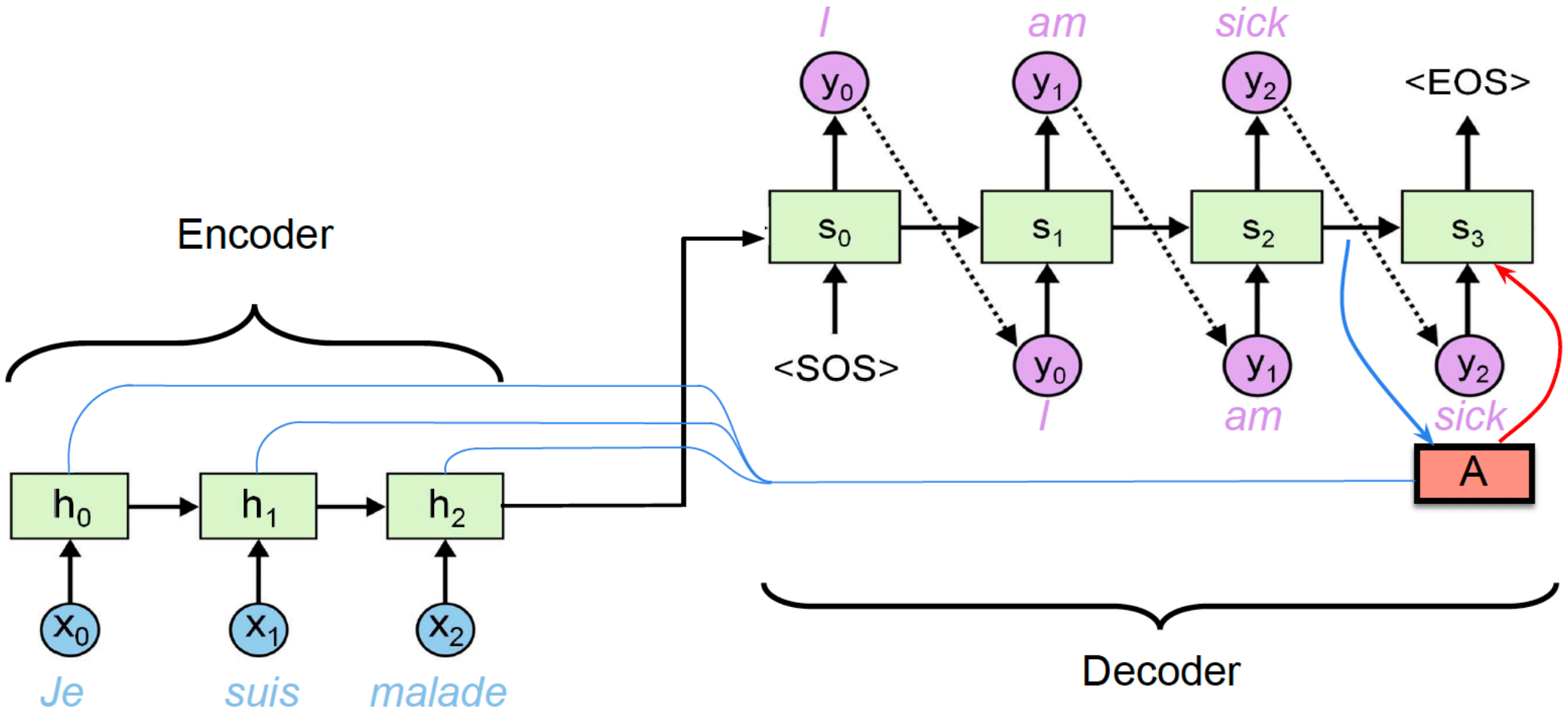
# Example: Sequence-to-Sequence + Attention



# Example: Sequence-to-Sequence + Attention

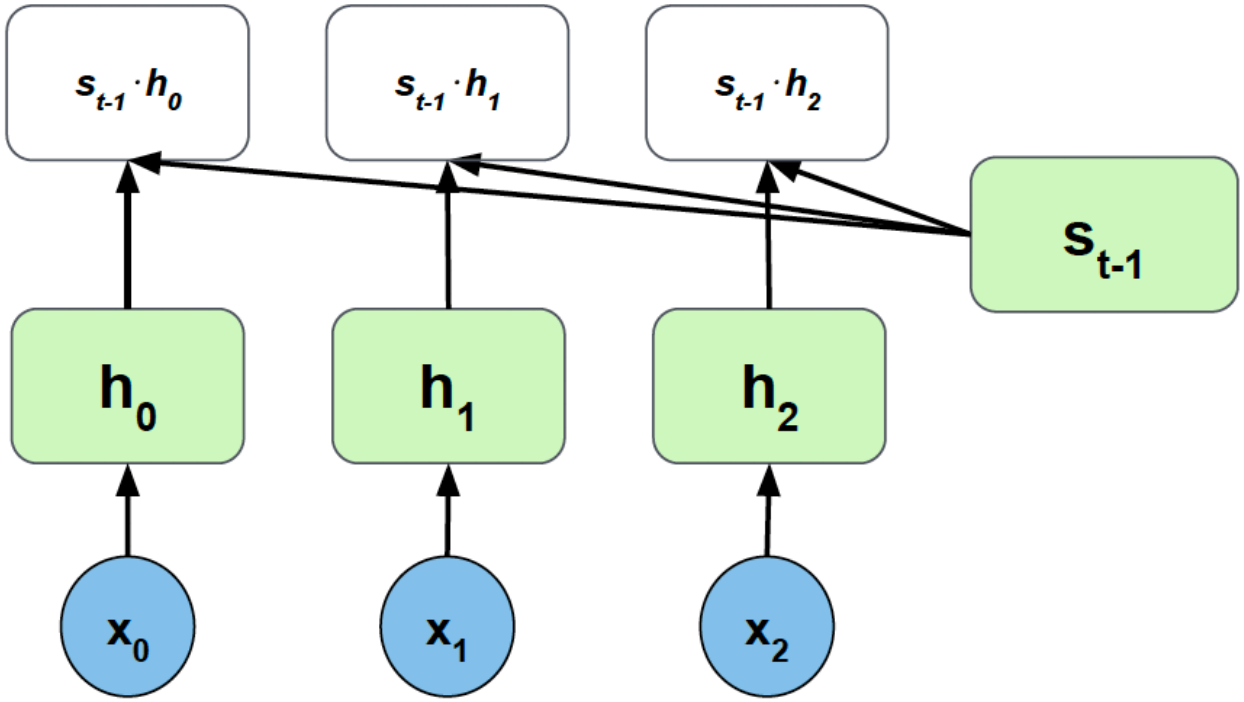


# Example: Sequence-to-Sequence + Attention



# Attention Function

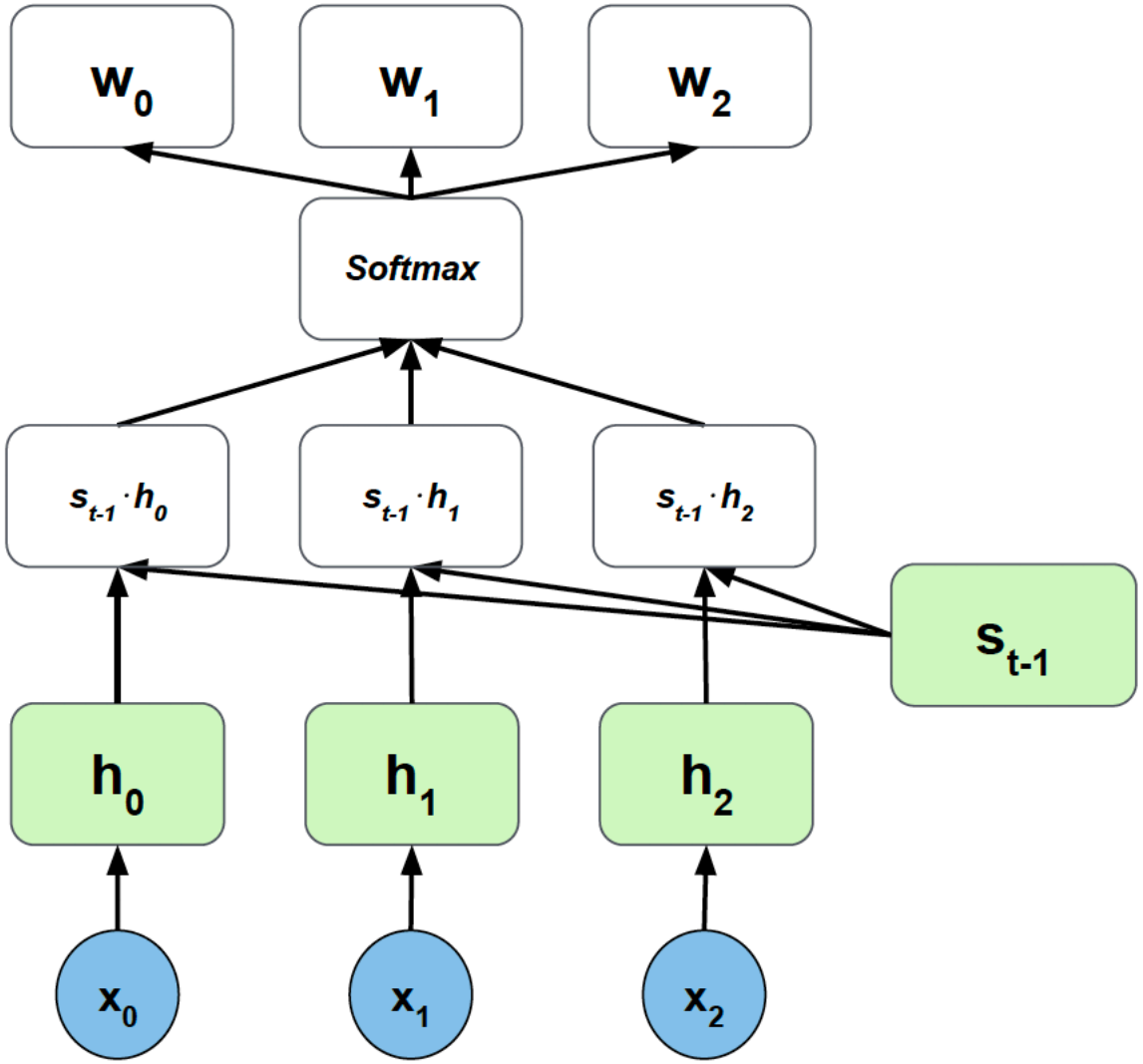
- There are several possible implementations for **A**.
- The most simple version is based on a dot product, i.e., $e_i = s_{t-1} \cdot h_i$ .



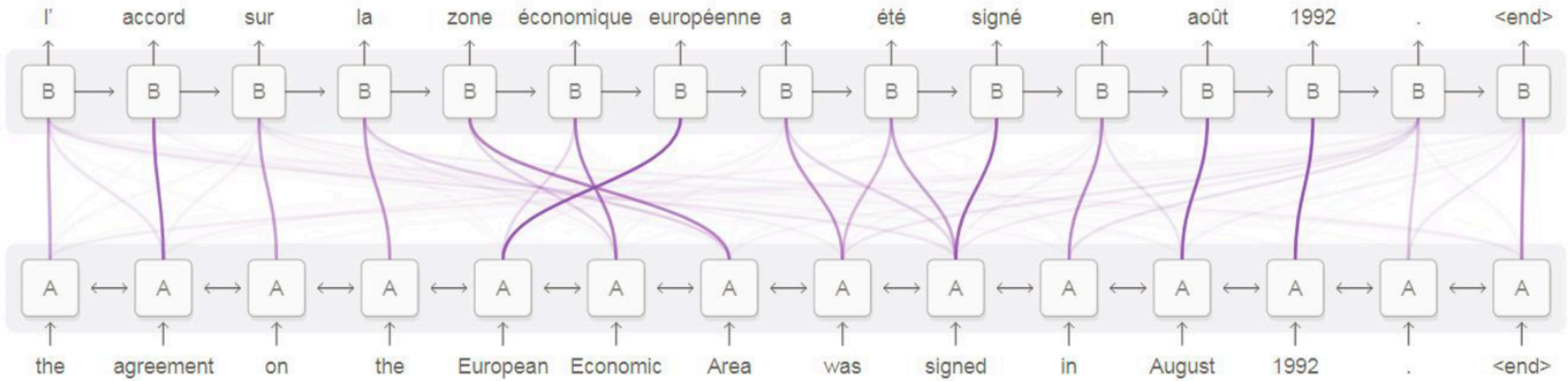
# Attention Function

- The dot product results are passed through a Softmax to get normalized weights **w=[$w_0$, …, $w_n$]**, which indicate how "important" the various elements are.
- The final result is the weighted sum of **h.**

$$c = \sum_{i=0}^{n} w_i \cdot h_i$$



# Visualizing Attention




The thick lines show where the decoder is focusing its attention when analyzing the encoded input sequence.

# Example

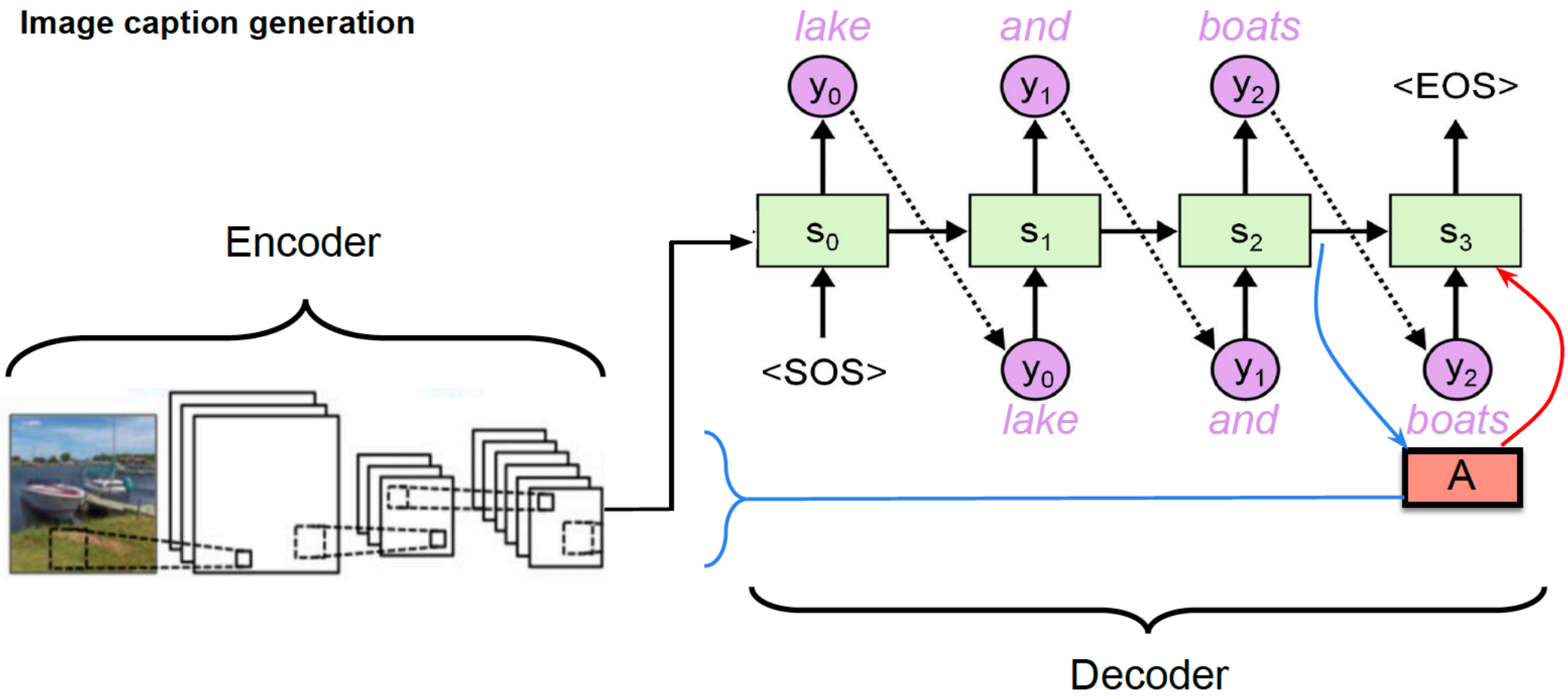


Image caption generation
Encoder
lake
and
boats
<EOS>
$y_0$
$y_1$
$y_2$
$s_0$
$s_1$
$s_2$
$s_3$
<SOS>
$y_0$
$y_1$
$y_2$
lake
and
boats
A
Decoder

# Example

A woman is throwing a frisbee in a park .

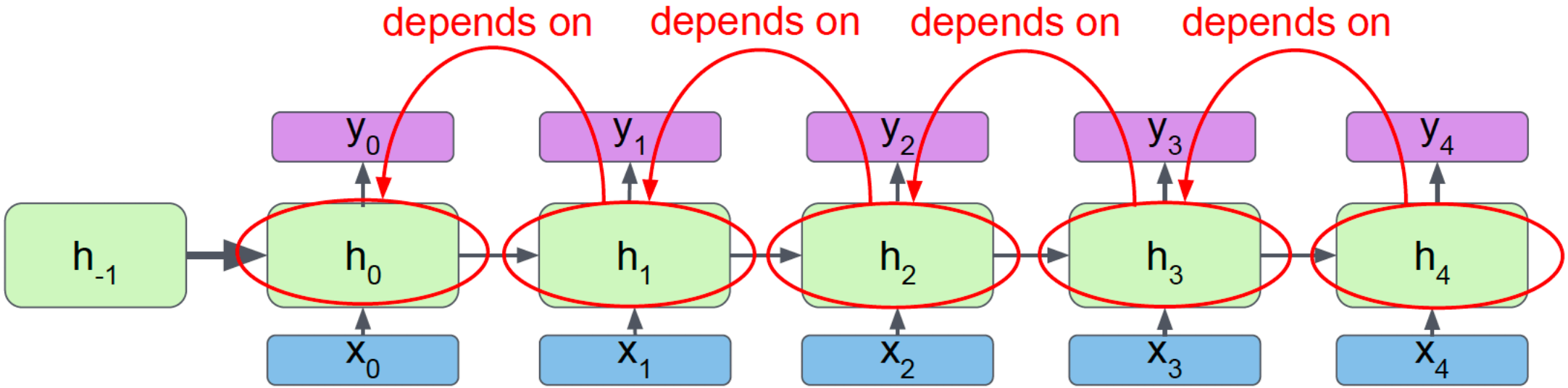

Xu et al., Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, 2015

# Transformer

- Sequence-to-Sequence + Attention systems perform well, but they are based on RNNs (simple RNNs / LSTMs / GRUs).
- RNNs suffer from two problems:
  - Not easy to parallelize.
  - Even in the more "complex" implementations (e.g. LSTMs), they struggle to capture (very) long-term dependencies.

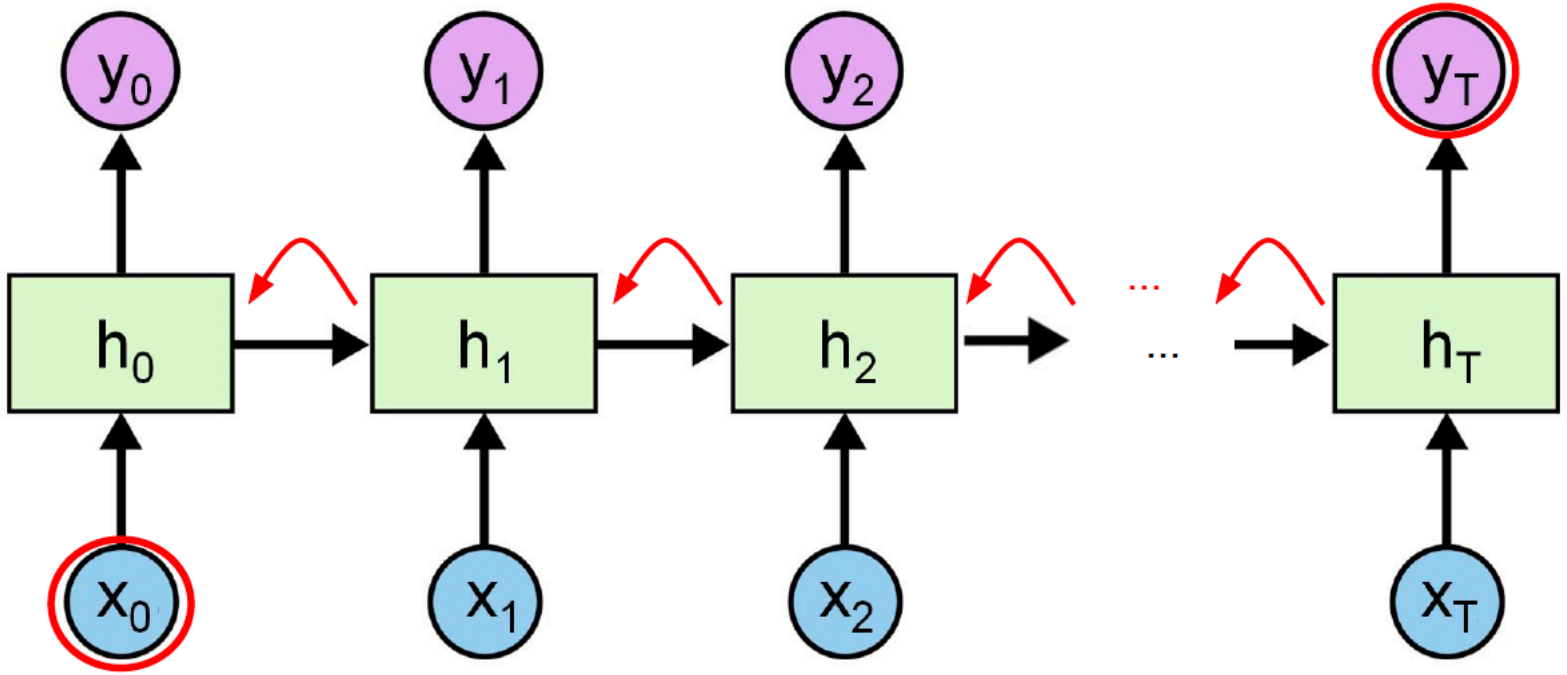
# Beyond RNNs

- Every state in an RNN depends on the previous internal state.
- This creates a chain of computation which prevents parallelization.



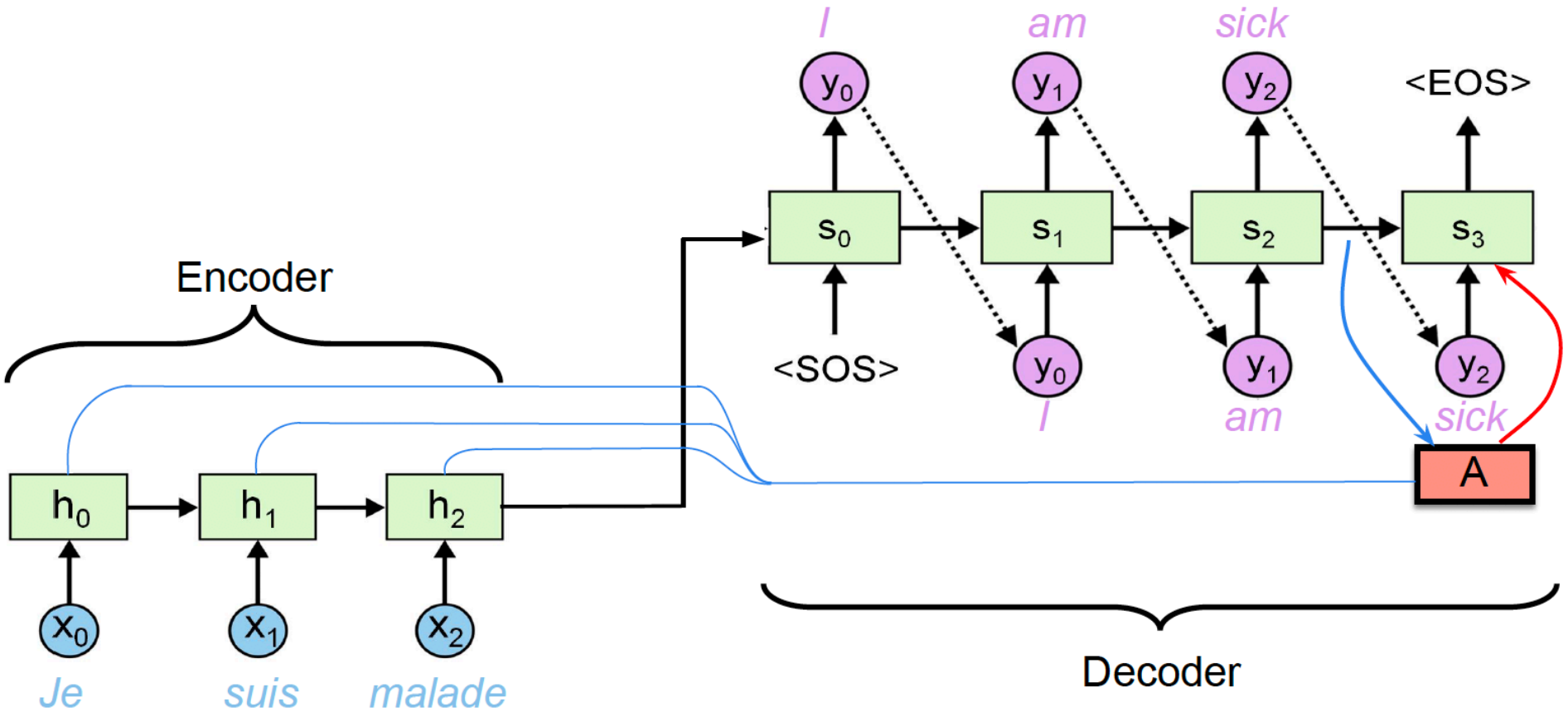
# Beyond RNNs

- Long-term dependencies are hard to capture with RNNs.
- This problem is strongly mitigated using LSTMs / GRUs, but it's still there for very long sequences.



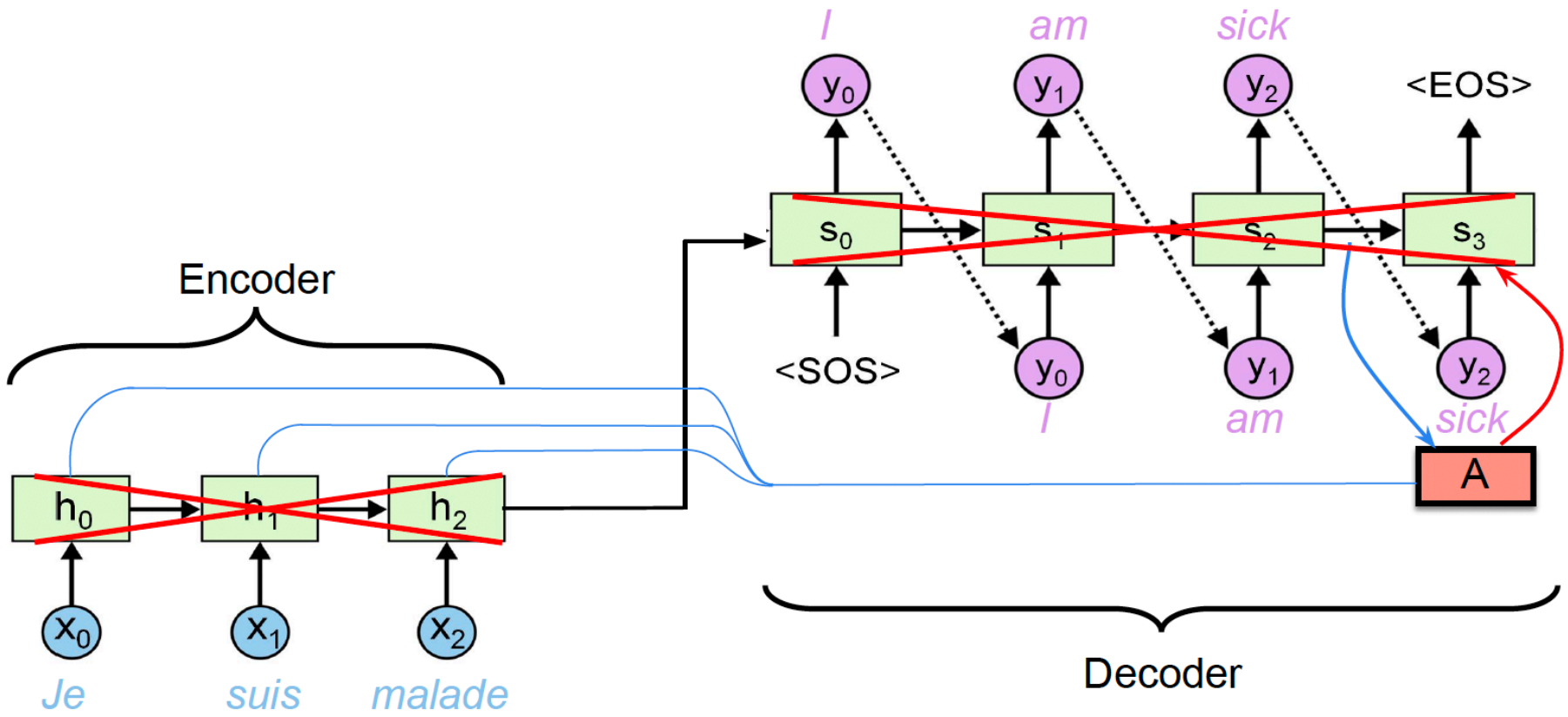
# Beyond RNNs

- There is no easy solution to deal with those RNN-related problems.



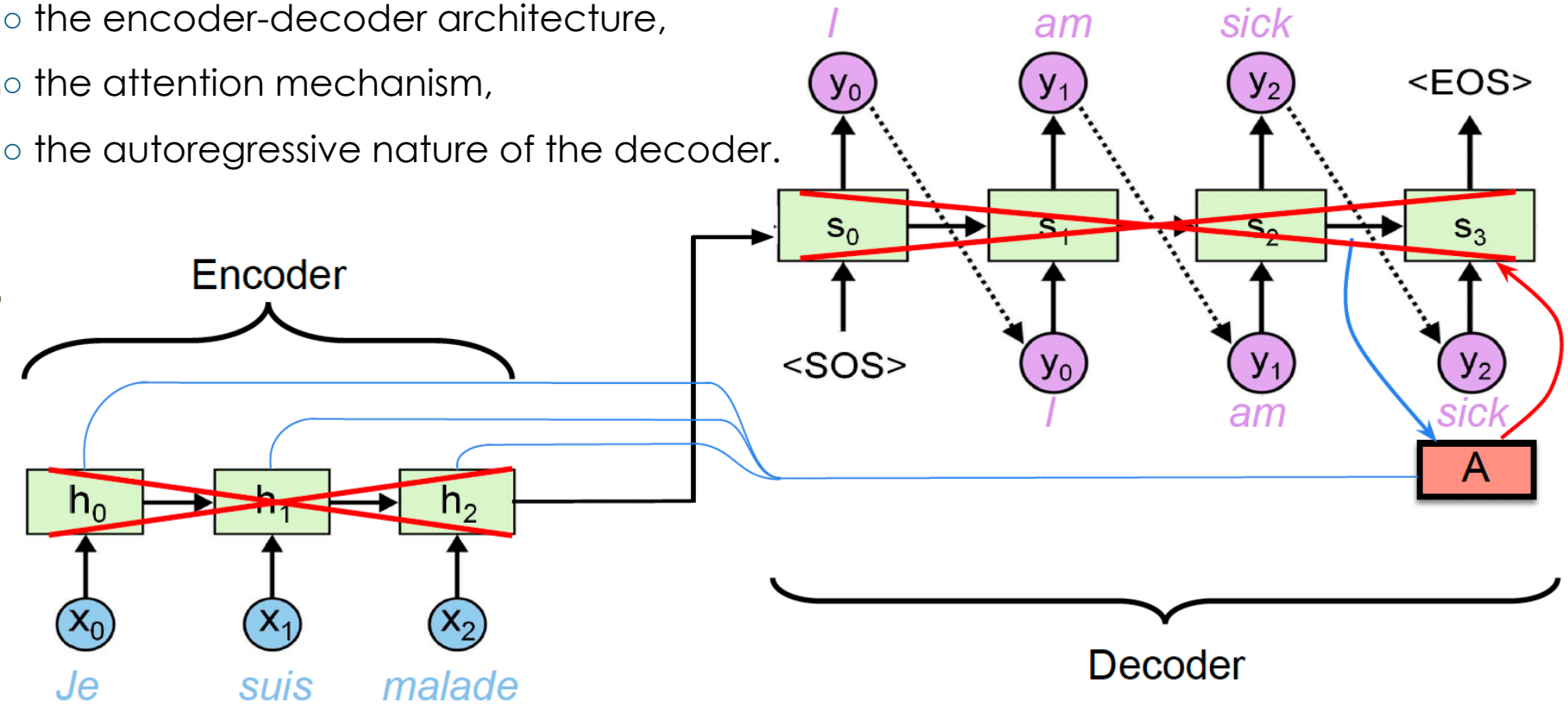
# Beyond RNNs

- To improve seq2seq systems, we need to find a replacement for RNNs.



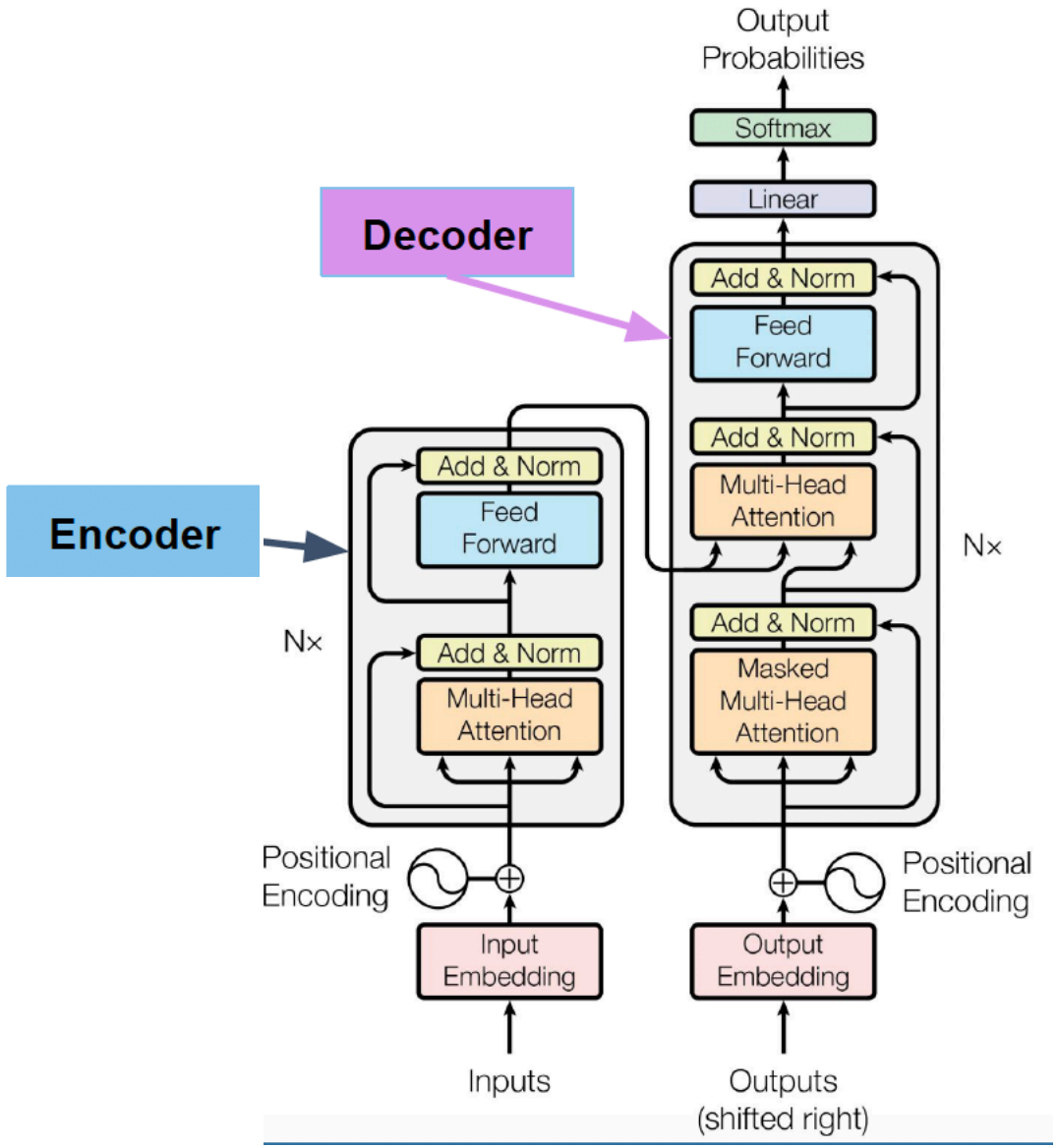
# Beyond RNNs

- To improve seq2seq systems, we need to find a replacement for RNNs.
  - the encoder-decoder architecture,
  - the attention mechanism,
  - the autoregressive nature of the decoder.



# Transformer

- The Transformer architecture was introduced in the paper "Attention is all you need".



# The End