Recurrent Neural Networks

UNN Deep Learning Lecture Course – 2020

Alexey Sidnev & Alexander Shain

October 28, 2020

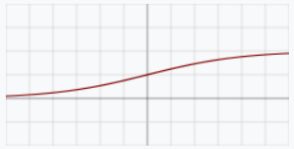
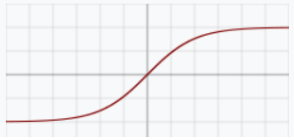
Agenda

- Preliminaries
- Recurrent Neural Networks
- LSTM
- Attention mechanism
- Transformer
- Conclusion

Preliminaries

Activation functions

- Tanh and Sigmoid activations

Name	Plot	Equation	Derivative (with respect to x)	Range
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ [1]	$f'(x) = f(x)(1-f(x))$	$(0,1)$
TanH		$f(x) = \tanh(x) = \frac{(e^{x}-e^{-x})}{(e^{x}+e^{-x})}$	$f'(x) = 1-f(x)^2$	$(-1,1)$

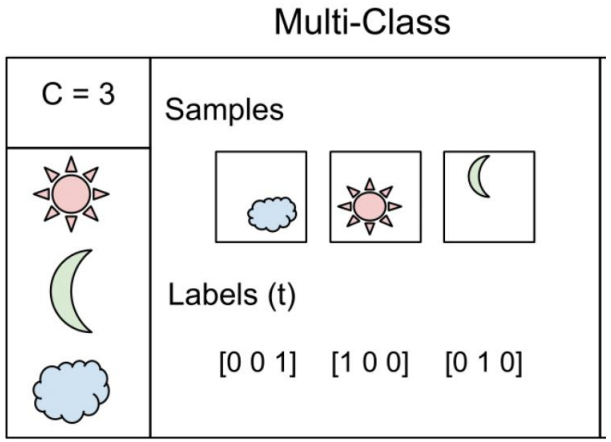
- Softmax function

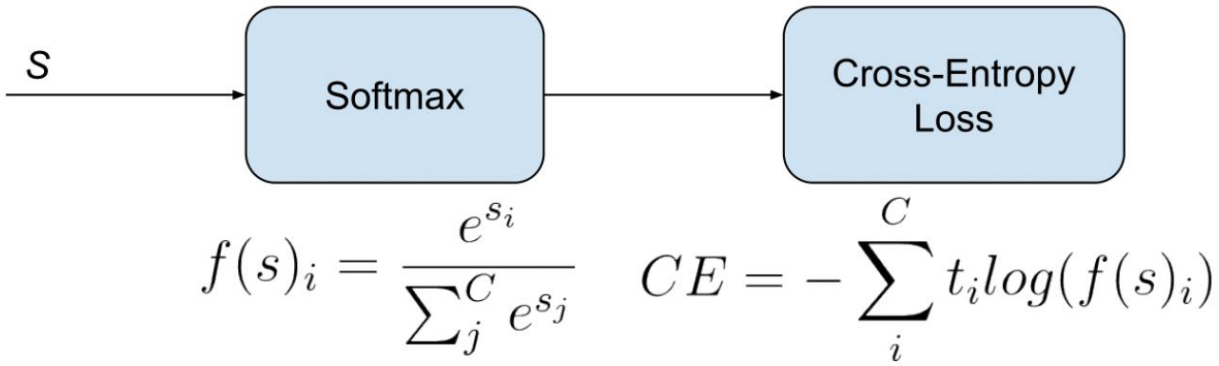
Name	Equation	Derivatives	Range
Softmax	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$ for $i = 1, \ldots, J$	$\frac{\partial f_i(\vec{x})}{\partial x_j} = f_i(\vec{x})(\delta_{ij} - f_j(\vec{x}))$ [3] [4]	$(0,1)$

[en.wikipedia.org]

Cross-Entropy Loss and One-Hot Encoding

- one_hot[id] = [1 *if* i == id *else* 0 *for* i in *range*(n_classes)]
- id[**cloud**] = 2 => one_hot[**cloud**] = [0, 0, 1]





[gombru.github.io, 2018]

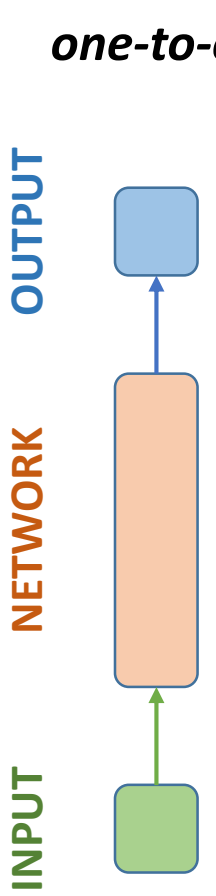
Chain Rule

$$\frac{\partial(error)}{\partial(weight\ 1)} = \frac{\partial(error)}{\partial(pred)} * \frac{\partial(pred)}{\partial(h1)} * \frac{\partial(h1)}{\partial(w1)}$$

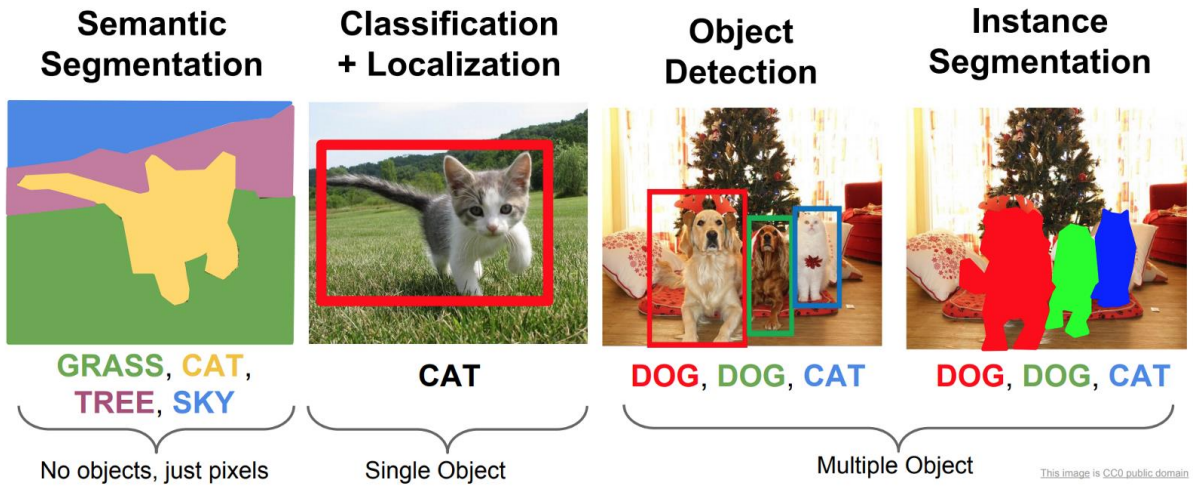
- Sum operation “shares” gradient
- Multiplication operation “distributes” gradient

Recurrent Neural Networks

Feed-Forward Neural Networks: Applications



- Classification Tasks
- Regression Tasks
- CV domain: (Image classification, objects detection, segmentation, feature extraction, etc.)



INPUT/OUTPUT DATA CAN BE PASSED TO/ OBTAINED FROM THE MODEL AT THE ”SINGLE” MOMENT OF TIME

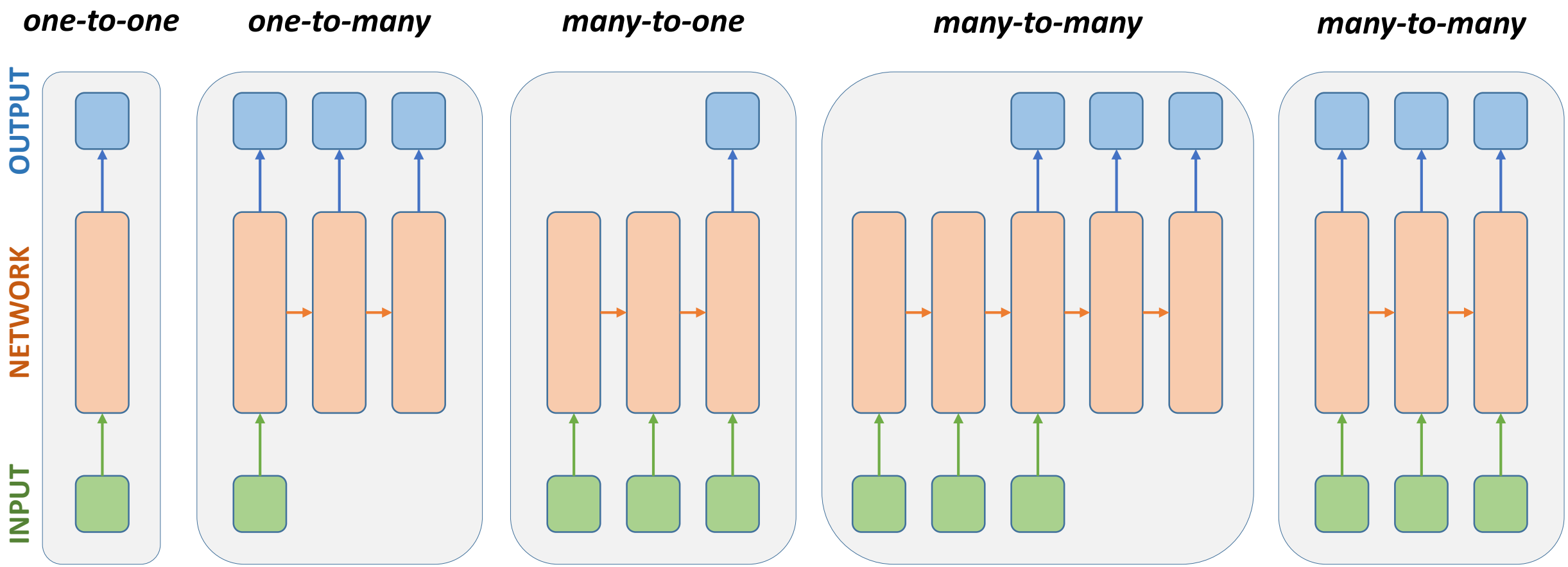
OR COMPUTATIONAL GRAPH HAS NO CYCLES

Recurrent Neural Networks: Reasoning

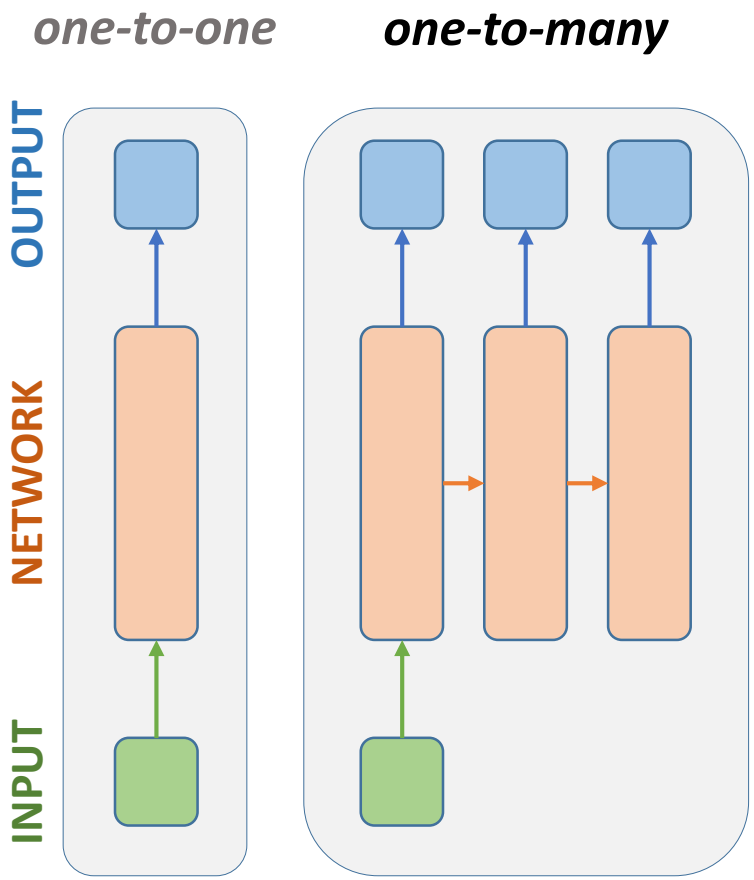
- In tasks such as natural language processing (NLP) or online text recognition (Online OCR) words / characters order matters.
- *BEATRIX KILLED BILL / BILL KILLED BEATRIX*
- We need models architectures that can have a kind “memory” of its previous states -> **RNN**s here!

Photo by: Mary Evans

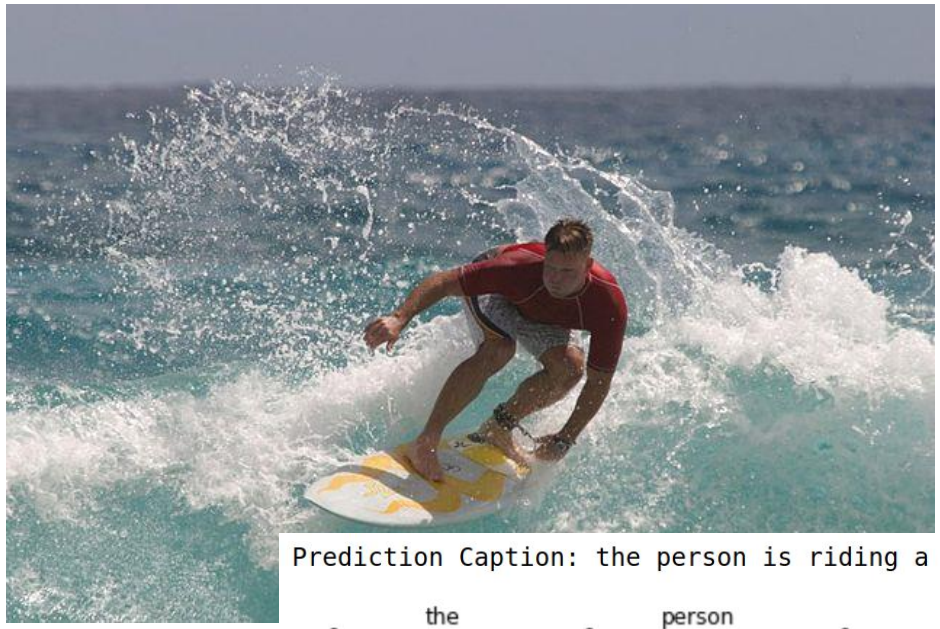
Recurrent Neural Networks Applications



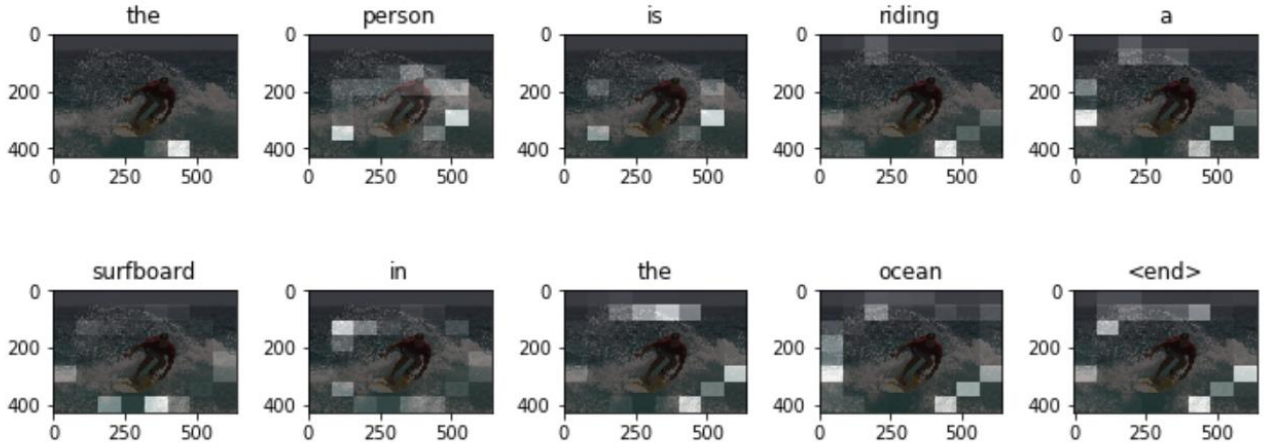
Recurrent Neural Networks Applications



e.g. Image captioning: Image -> Sequence of words;
Music generation



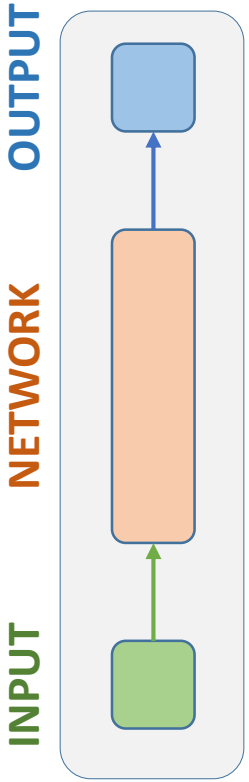
Prediction Caption: the person is riding a surfboard in the ocean <end>



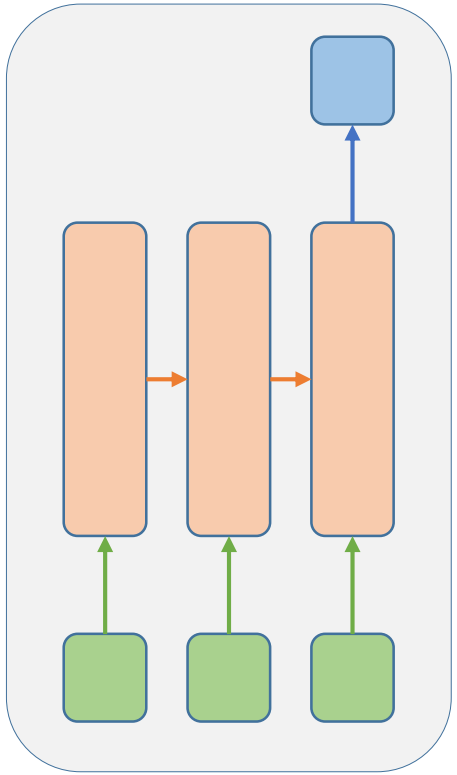
[TF tutorials]

Recurrent Neural Networks Applications

one-to-one

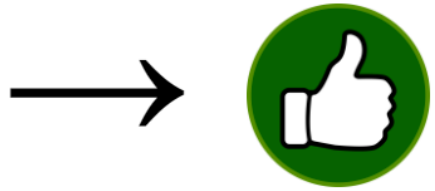


many-to-one



e.g. Sentiment classification,
video action prediction

"I love this movie.
I've seen it many times
and it's still awesome."

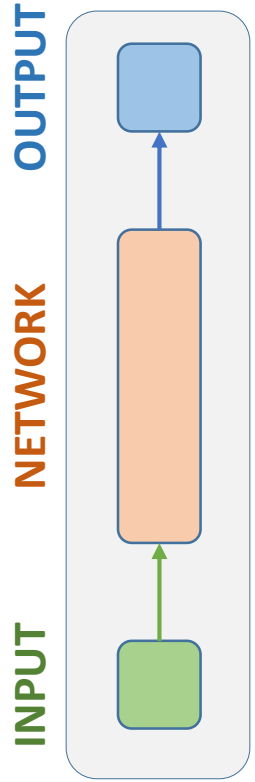


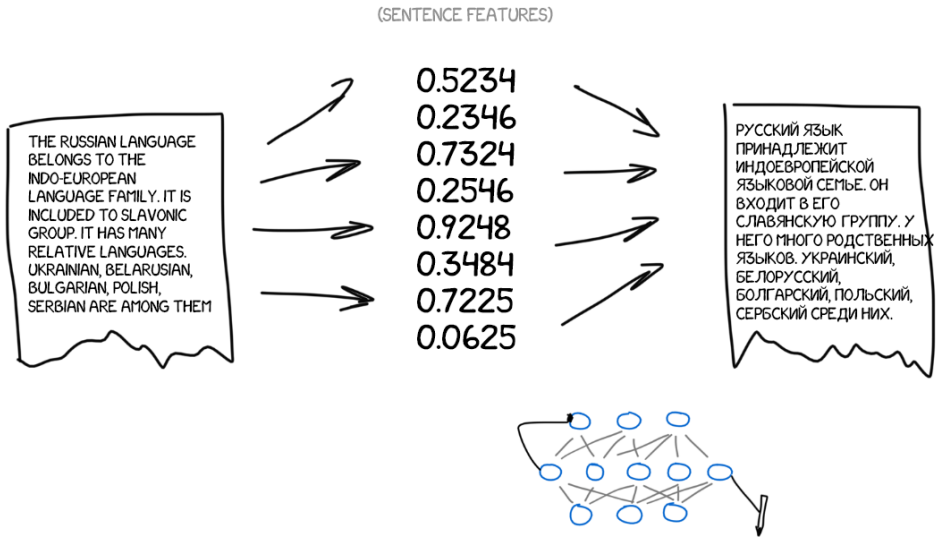
"This movie is bad.
I don't like it it all.
It's terrible."

[CFML blog]

Recurrent Neural Networks Applications

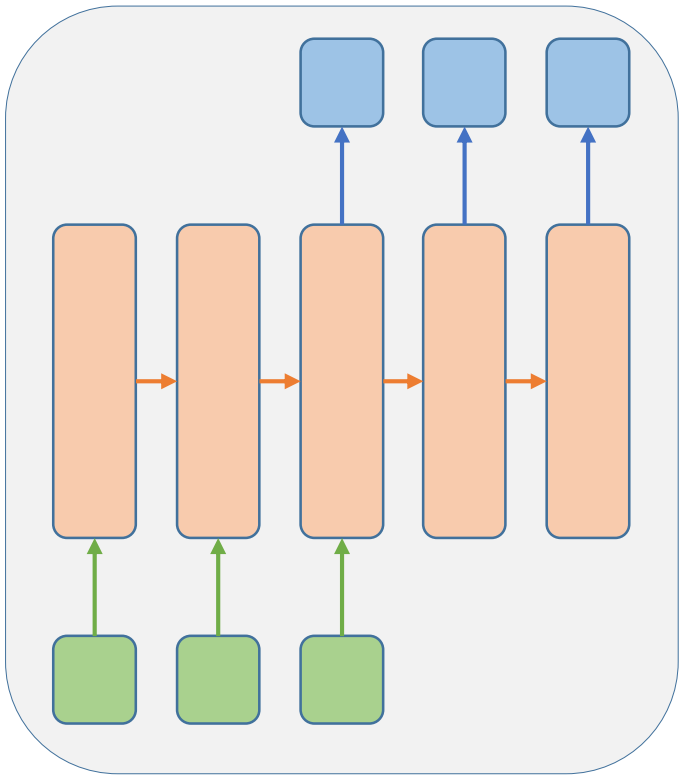
one-to-one





[vas3k blog]

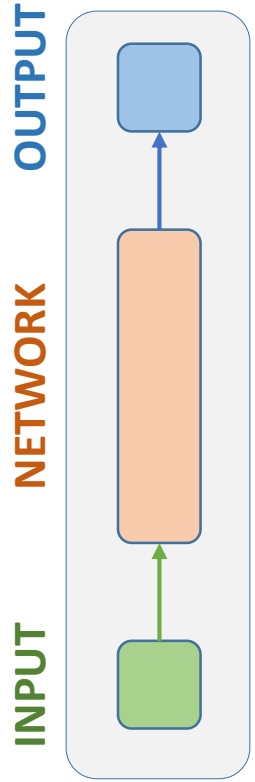
many-to-many



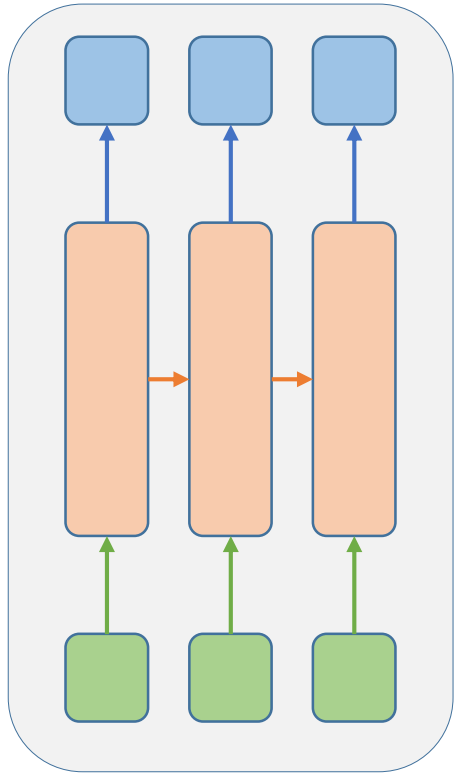
e.g. machine translation,
video captioning

Recurrent Neural Networks Applications

one-to-one

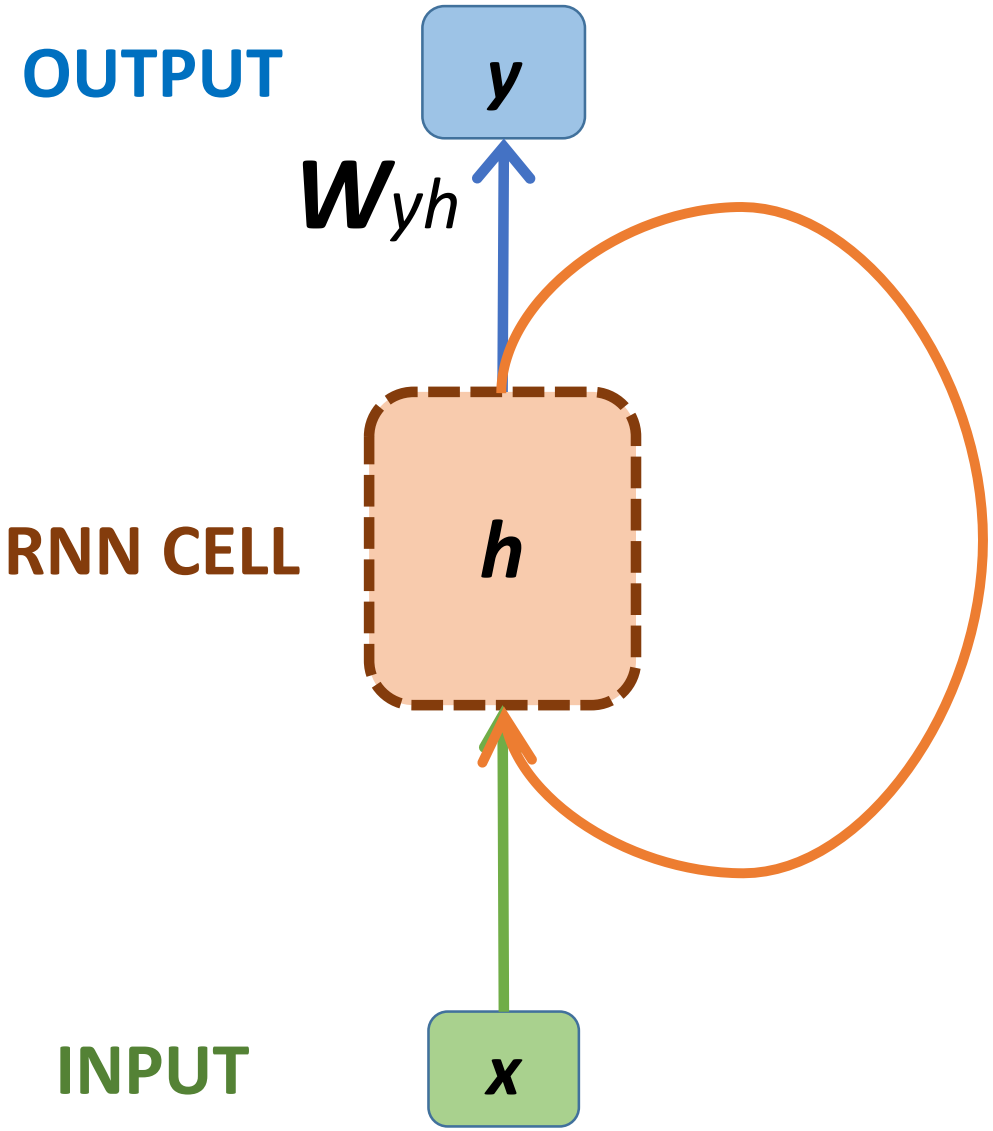


many-to-many



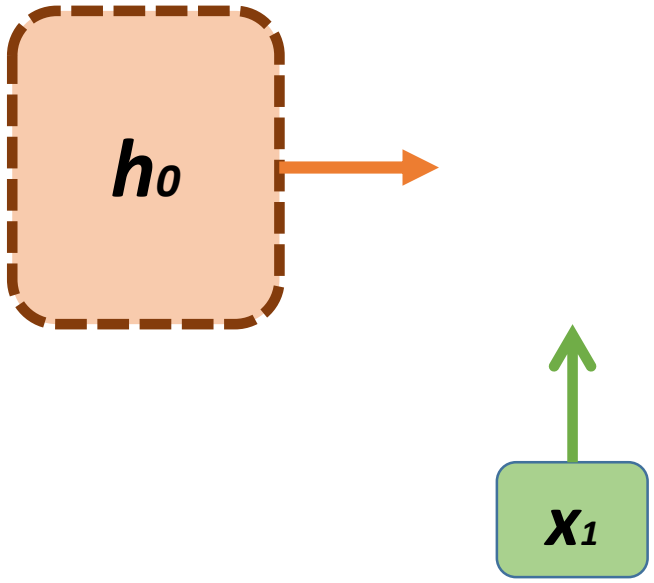
e.g. name entity recognition, video classification

Recurrent Neural Network



- Recurrently updating ***hidden state h*** using given ***input x*** and ***itself*** until the end of input or reaching programmed stop-signal
- Optionally returns ***output y*** as *weighted* ***hidden state h***

Unrolled RNN

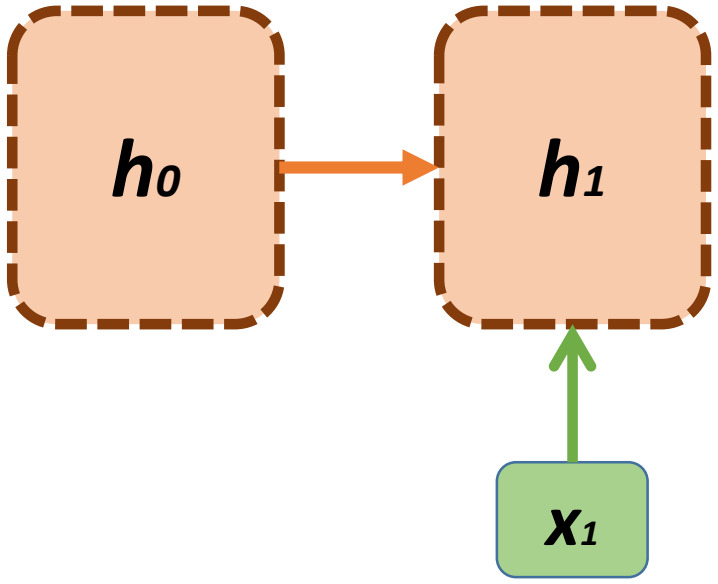


i = [1, 2, 3, …, T-1, T]

INPUT SEQUENCE (Xi)

Unrolled RNN

$$h_i = F_W(h_{i-1}, x_i)$$

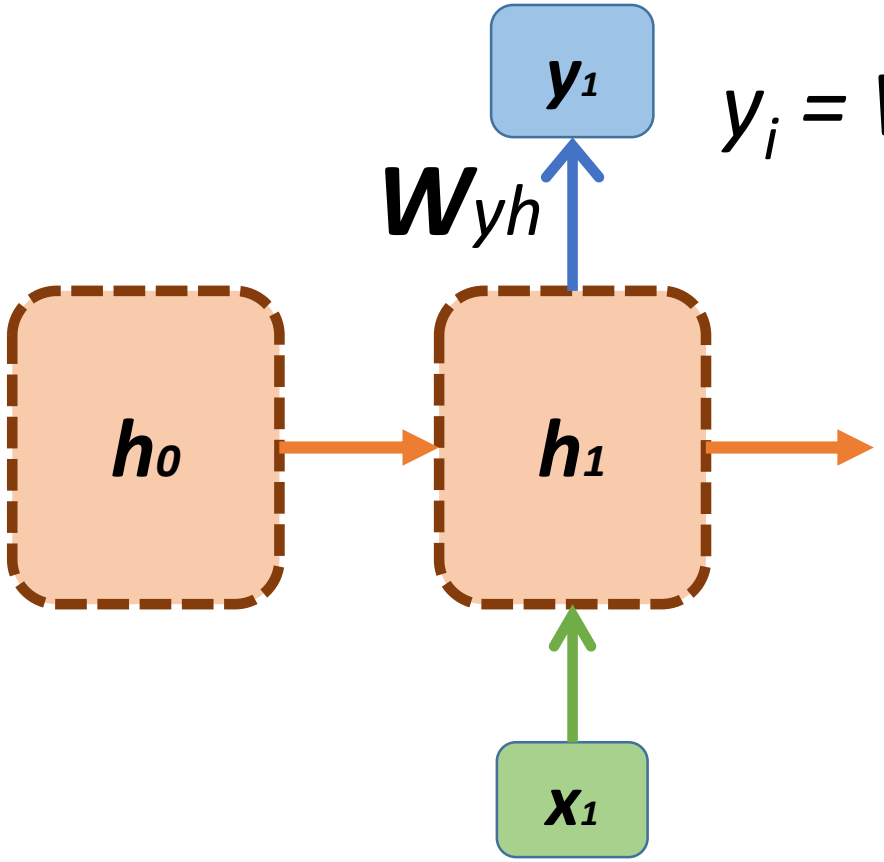


i = [1, 2, 3, …, T-1, T]

INPUT SEQUENCE (Xi)

Unrolled RNN

OUTPUT SEQUENCE (Yi) (optional, depends on use case)



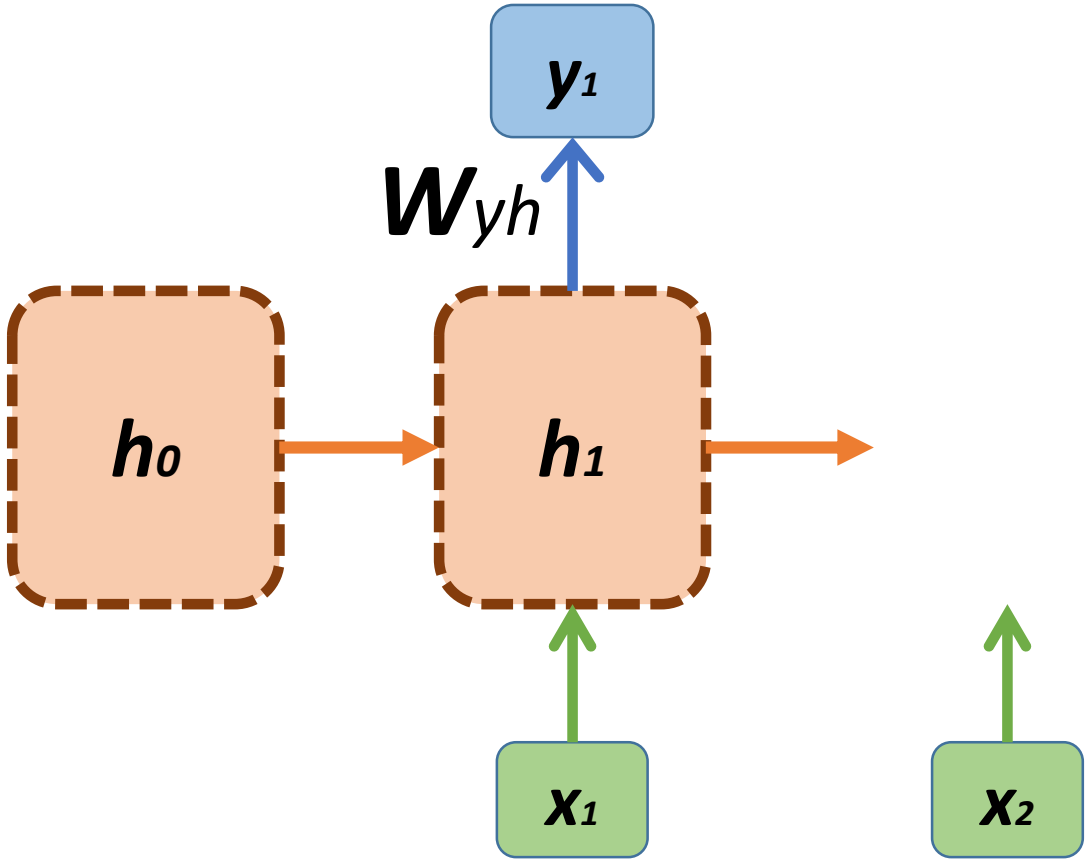
$$y_i = W_{yh} h_i + b_y$$

i = [1, 2, 3, …, T-1, T]

INPUT SEQUENCE (Xi)

Unrolled RNN

OUTPUT SEQUENCE (Yi) (optional, depends on use case)

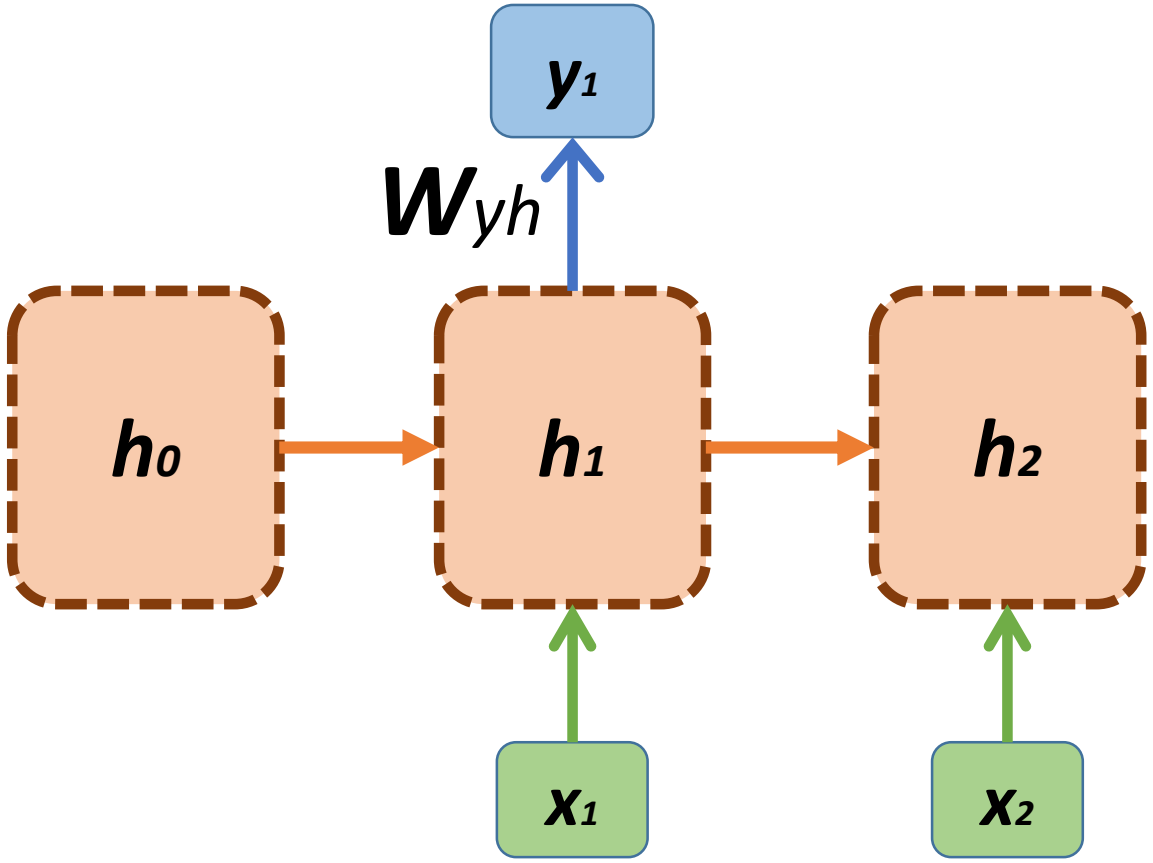


i = [1, 2, 3, …, T-1, T]

INPUT SEQUENCE (Xi)

Unrolled RNN

OUTPUT SEQUENCE (Yi) (optional, depends on use case)



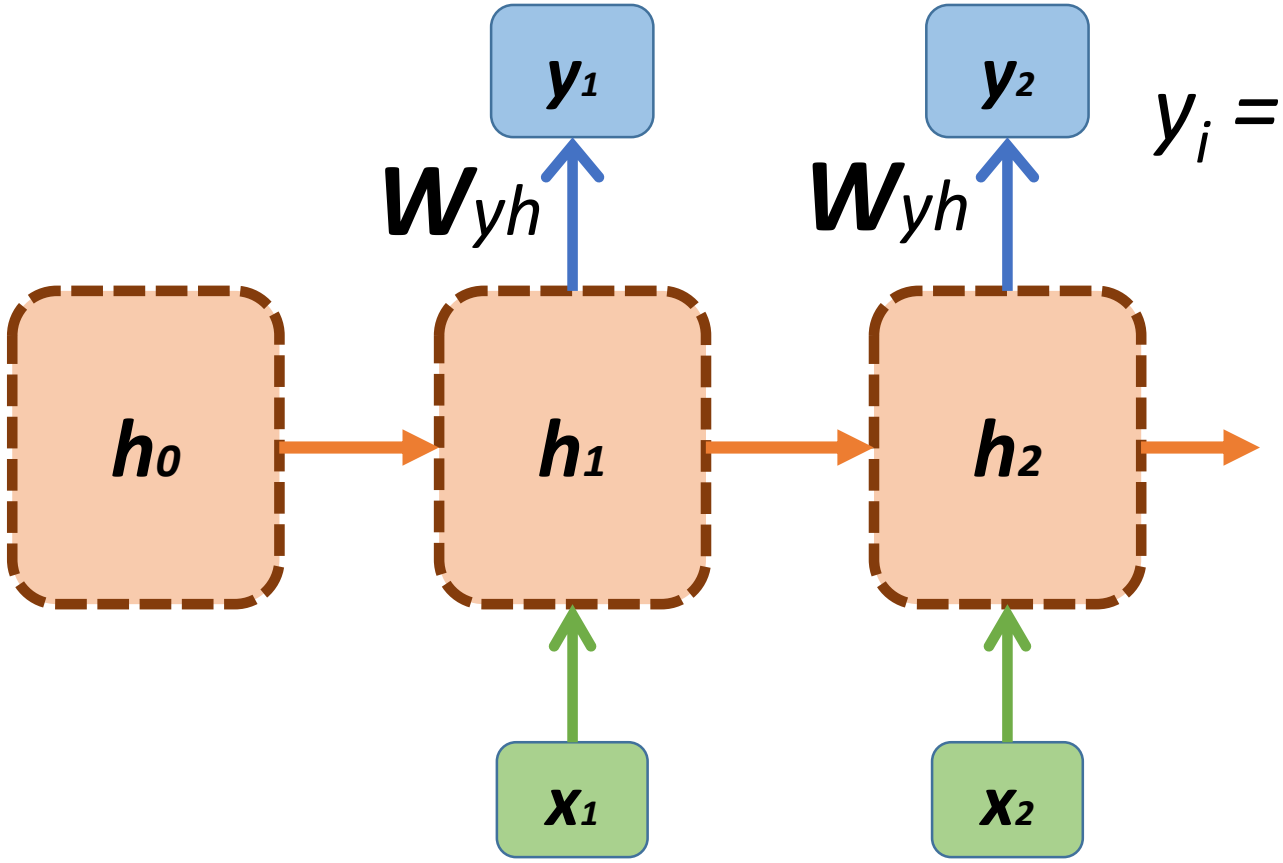
$$h_i = F_W(h_{i-1}, x_i)$$

i = [1, 2, 3, …, T-1, T]

INPUT SEQUENCE (Xi)

Unrolled RNN

OUTPUT SEQUENCE (Yi) (optional, depends on use case)



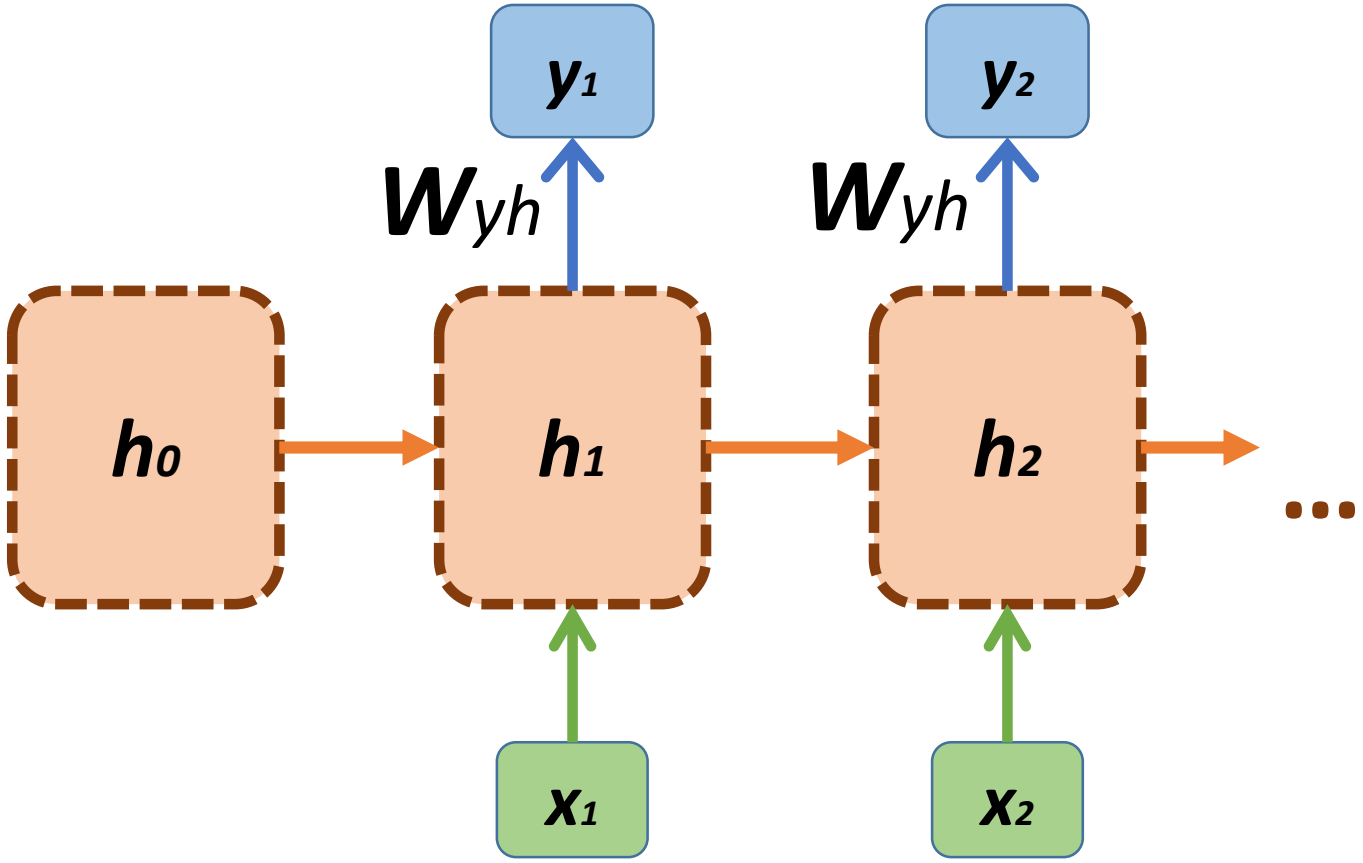
$$y_i = W_{yh} h_i + b_y$$

i = [1, 2, 3, …, T-1, T]

INPUT SEQUENCE (Xi)

Unrolled RNN

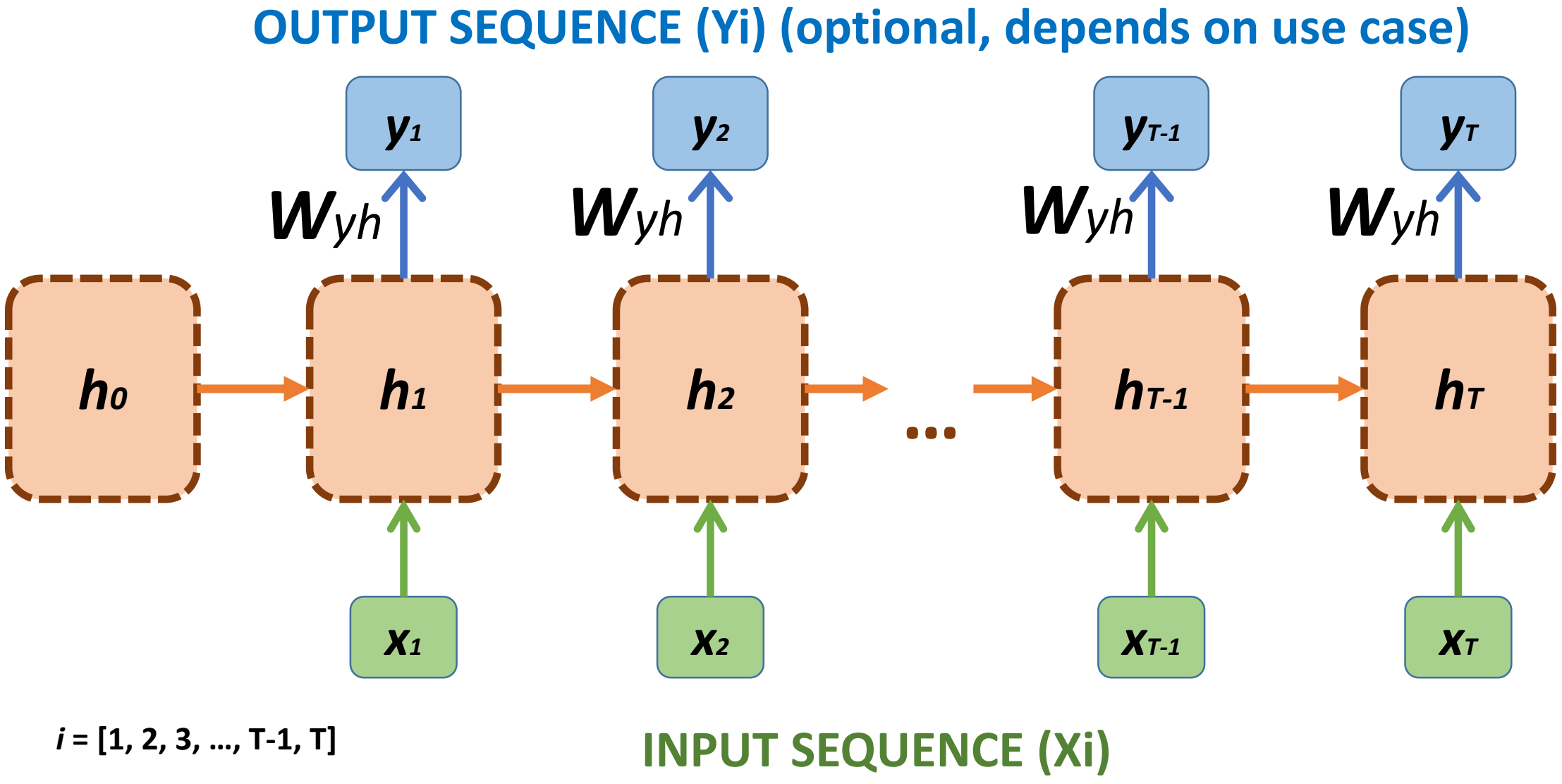
OUTPUT SEQUENCE (Yi) (optional, depends on use case)



i = [1, 2, 3, …, T-1, T]

INPUT SEQUENCE (Xi)

Unrolled RNN



An Elman RNN cell

$$h_i = F_{\boldsymbol{W}}(h_{i-1}, x_i)$$

$$h_i = tanh(\boldsymbol{W}_{hh}h_{i-1} + \boldsymbol{W}_{hx}x_i + b_h)$$

$$y_i = \boldsymbol{W}_{yh}h_i + b_y$$

Recurrent weights – $\boldsymbol{W}_{hh}$

Hidden bias – b_h

Input weights – $\boldsymbol{W}_{hx}$

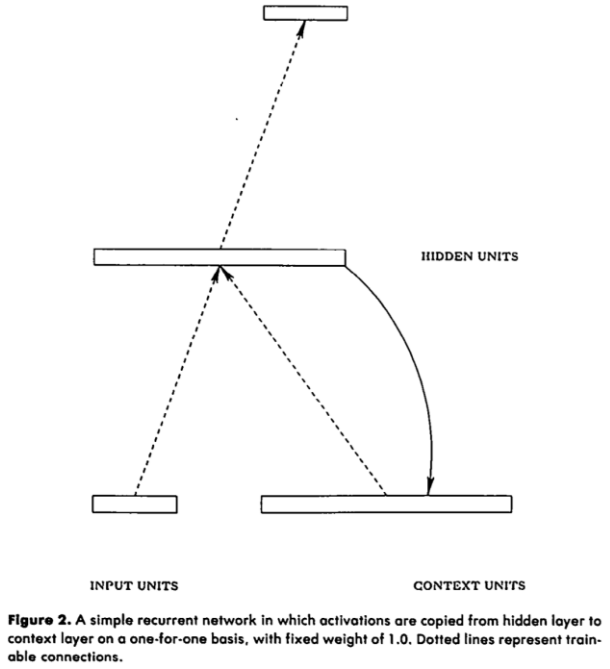
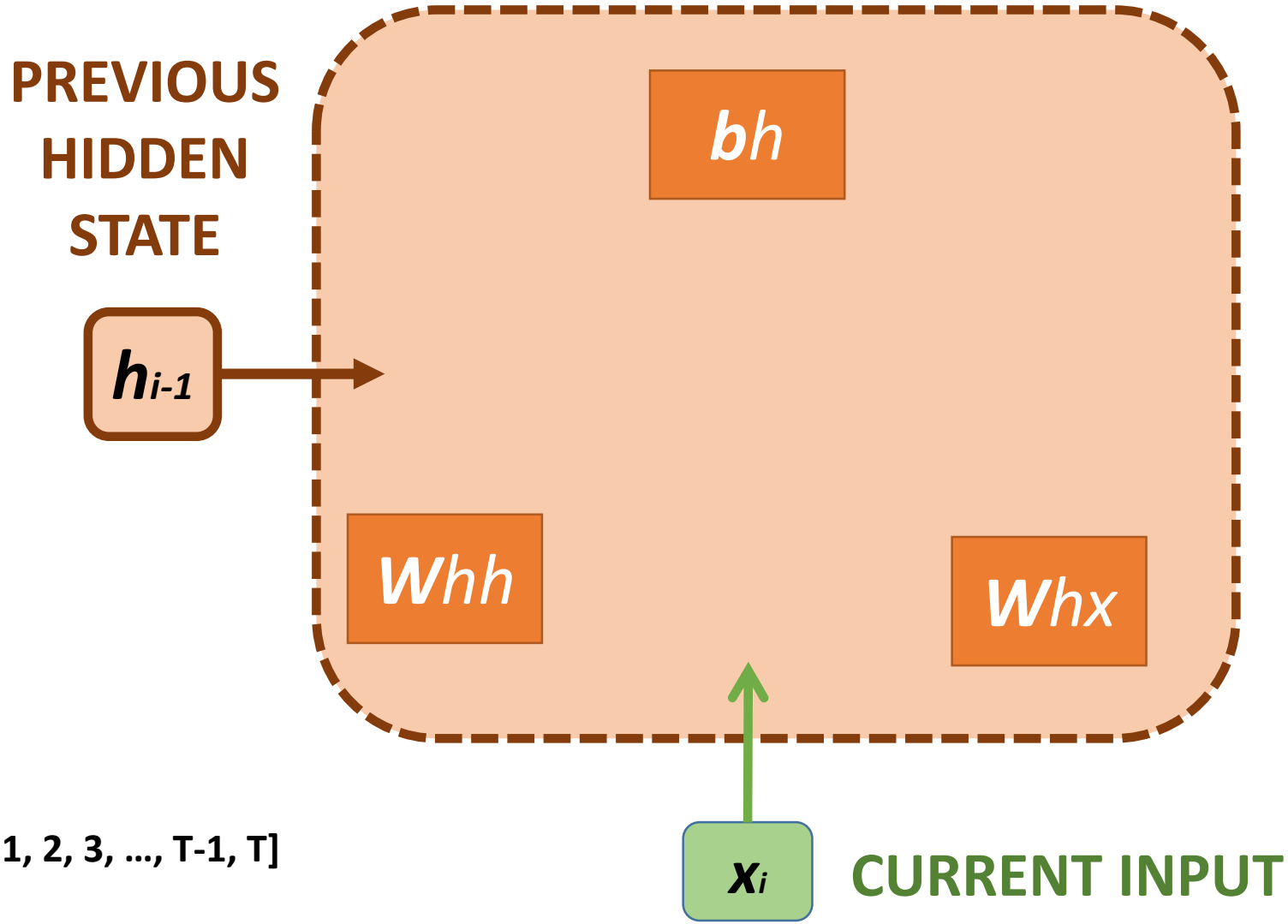


Figure 2. A simple recurrent network in which activations are copied from hidden layer to context layer on a one-for-one basis, with fixed weight of 1.0. Dotted lines represent trainable connections.

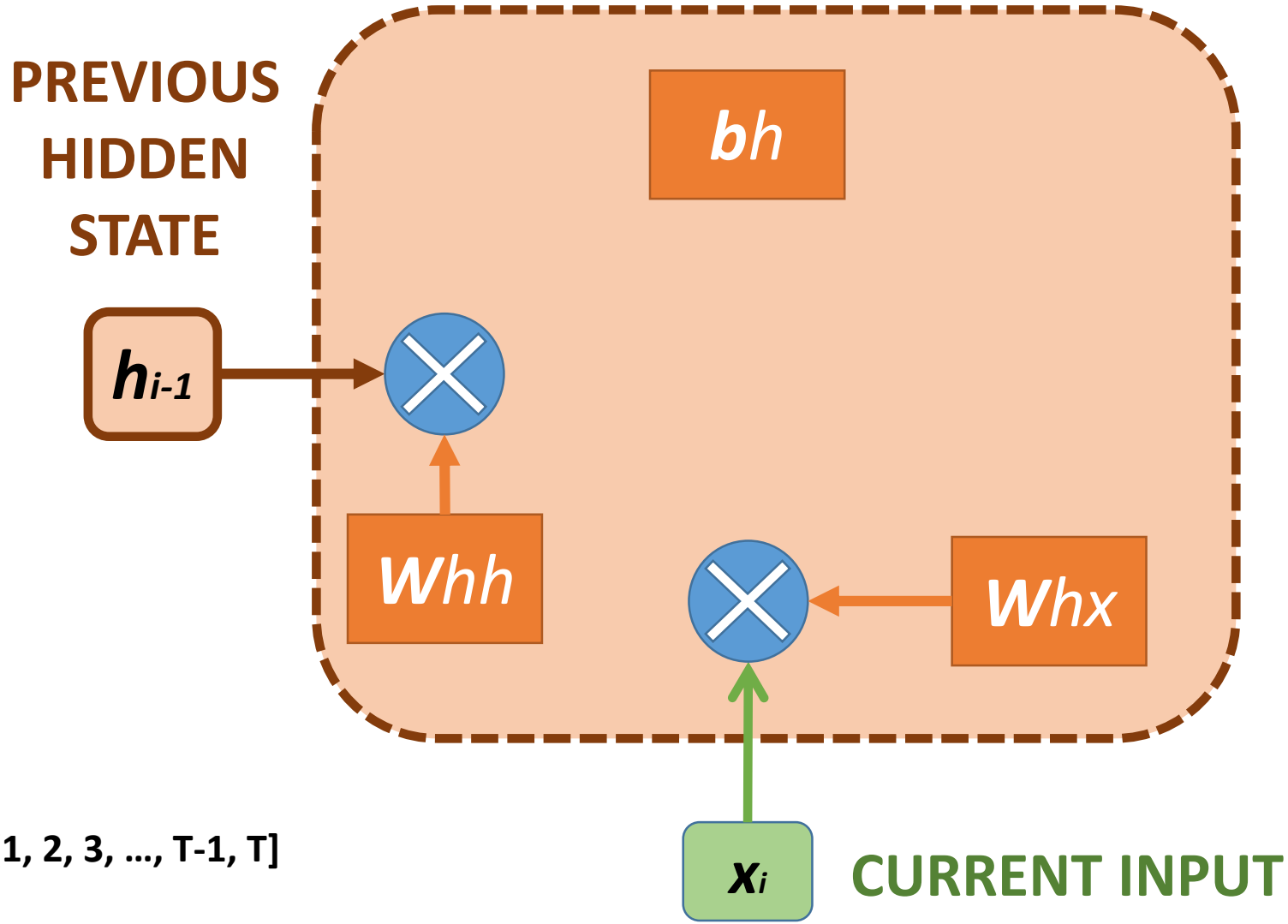
Elman, 1990

An Elman RNN cell



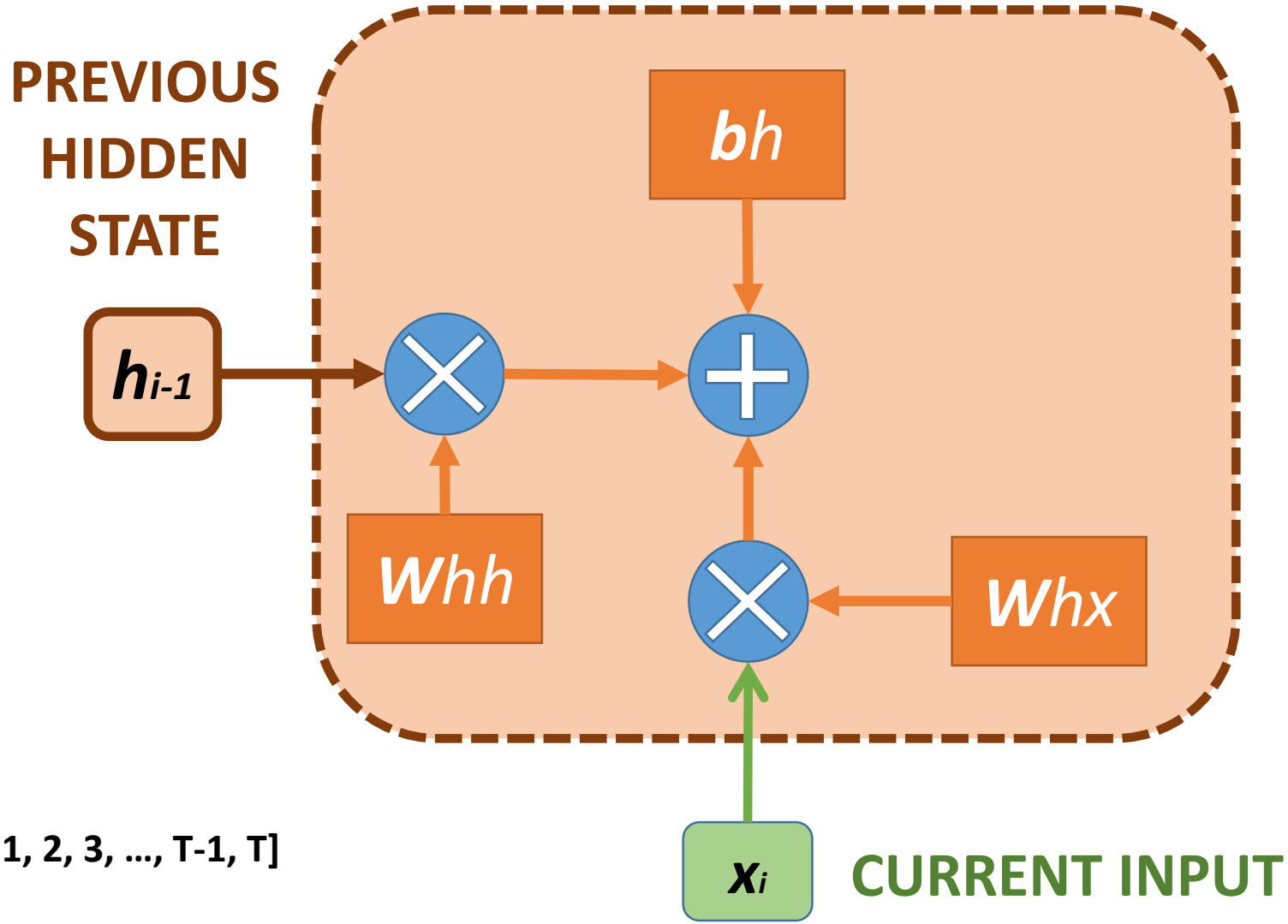
i = [1, 2, 3, …, T-1, T]

An Elman RNN cell



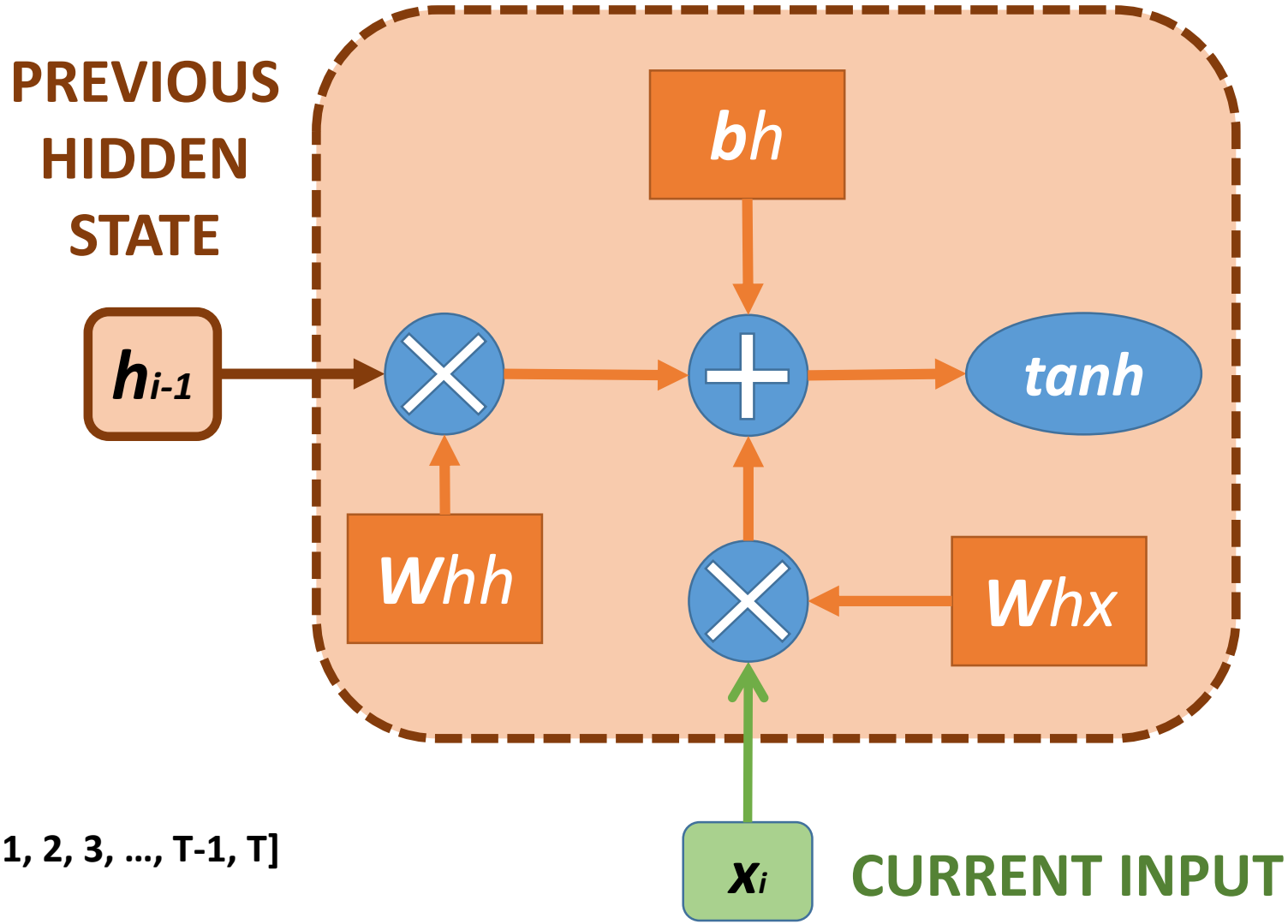
i = [1, 2, 3, …, T-1, T]

An Elman RNN cell



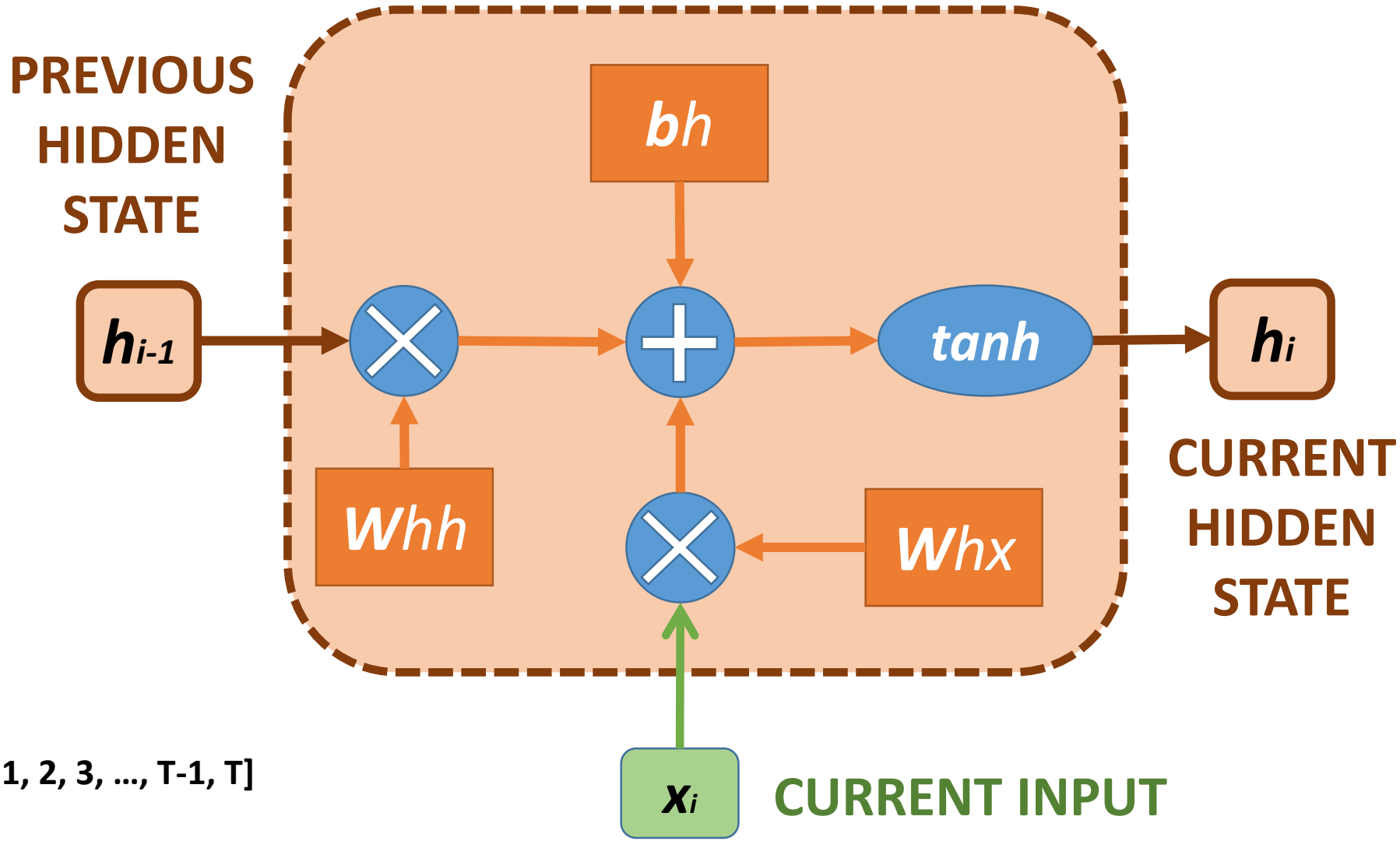
i = [1, 2, 3, …, T-1, T]

An Elman RNN cell



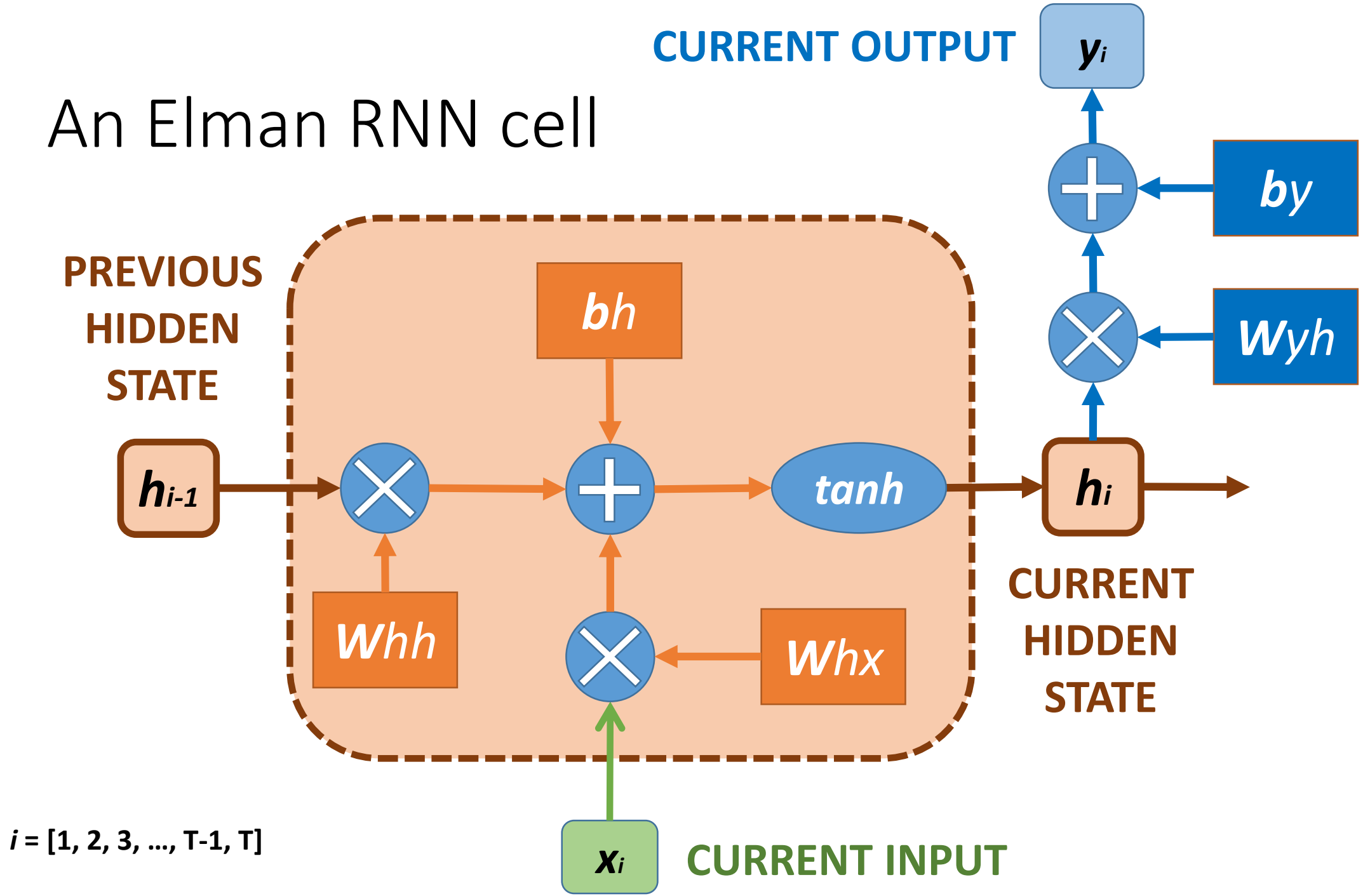
i = [1, 2, 3, …, T-1, T]

An Elman RNN cell



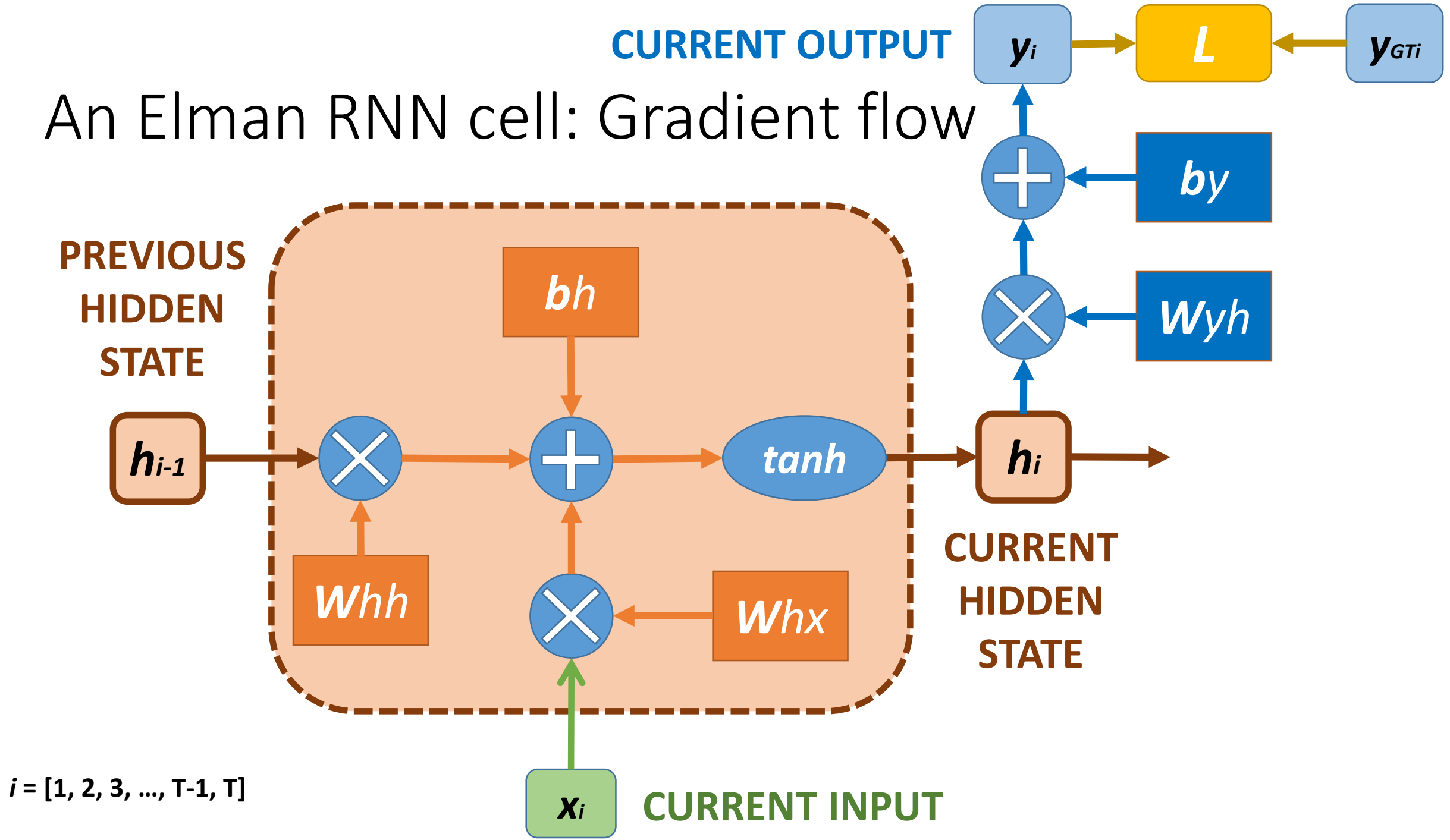
i = [1, 2, 3, …, T-1, T]

An Elman RNN cell

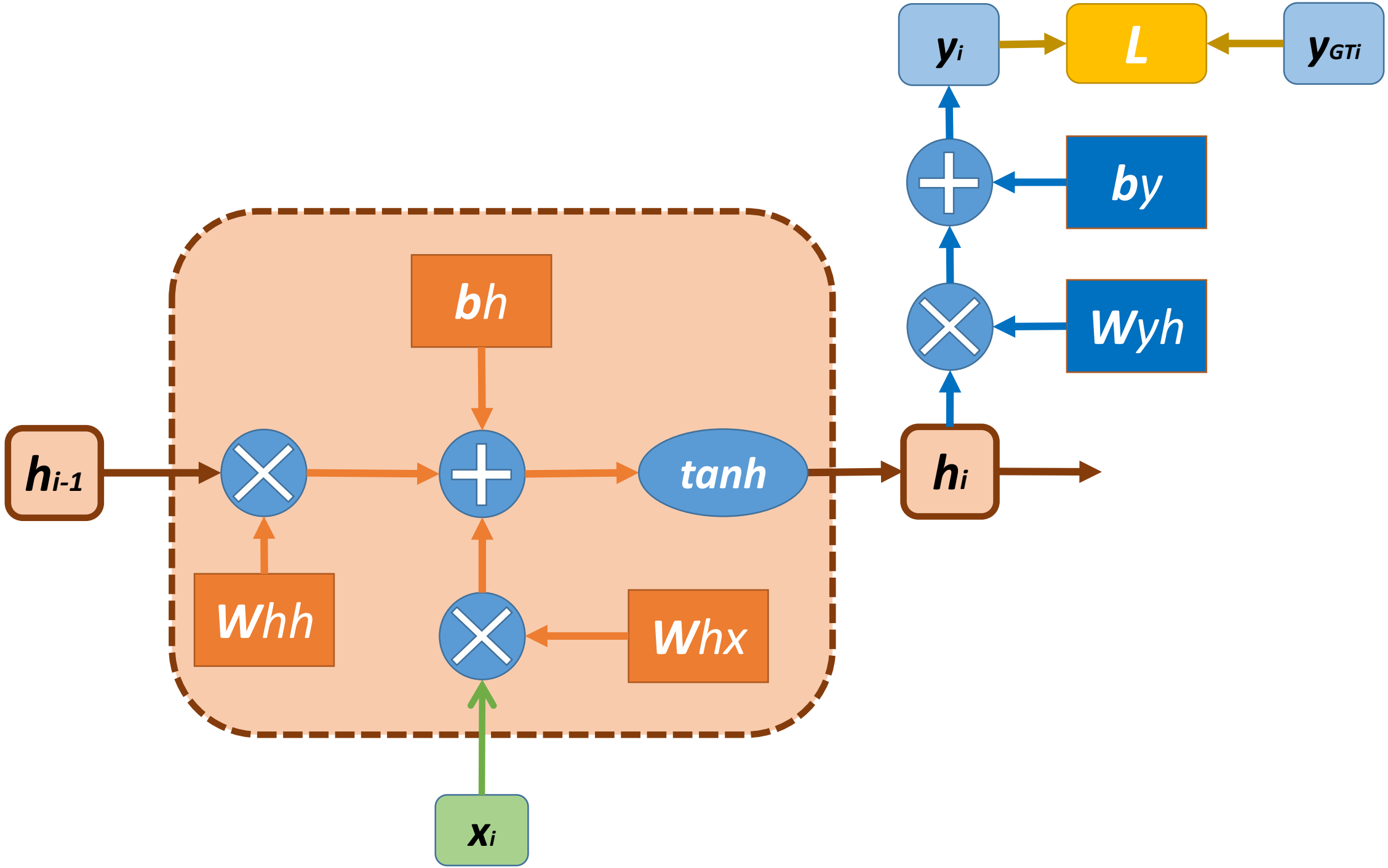


i = [1, 2, 3, …, T-1, T]

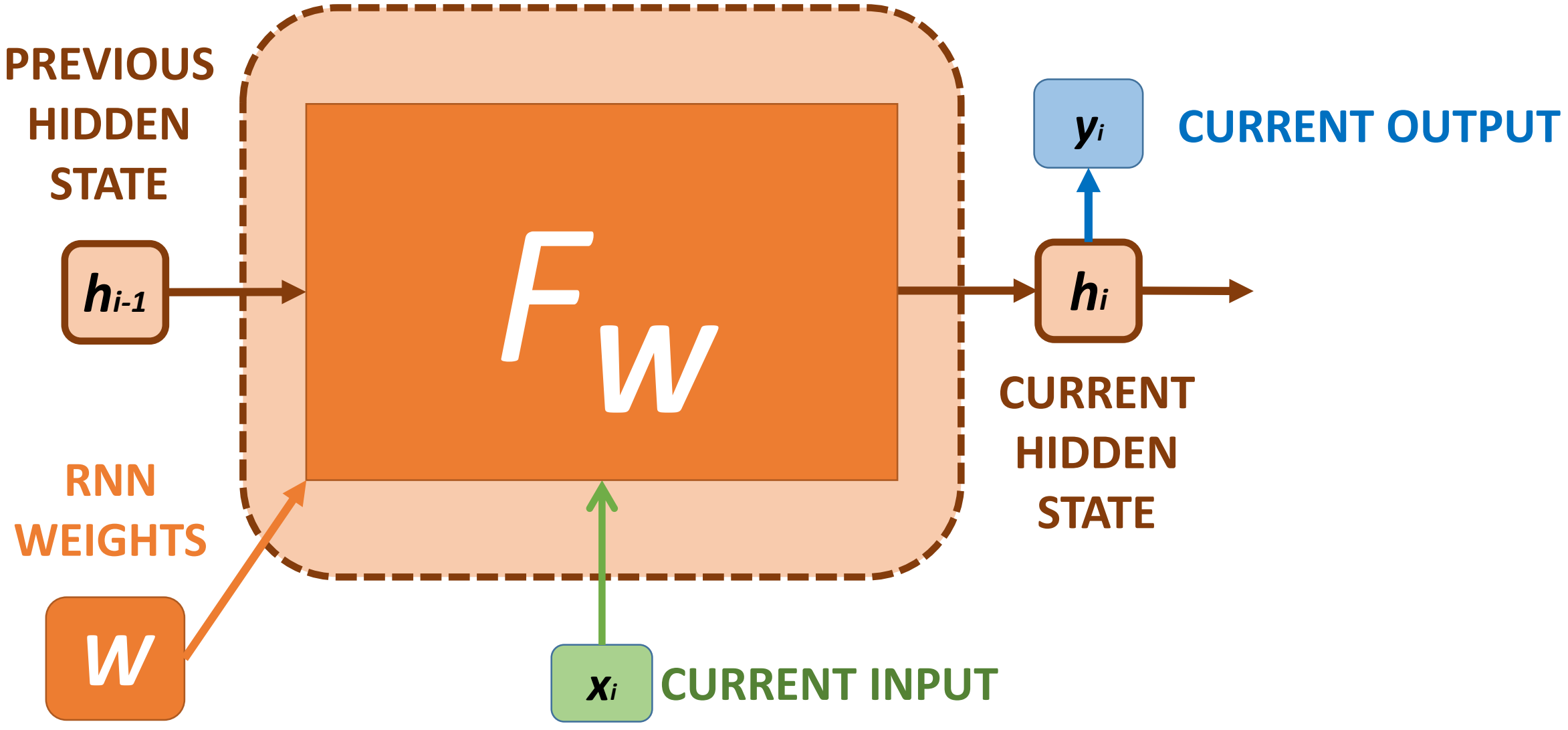
An Elman RNN cell: Gradient flow



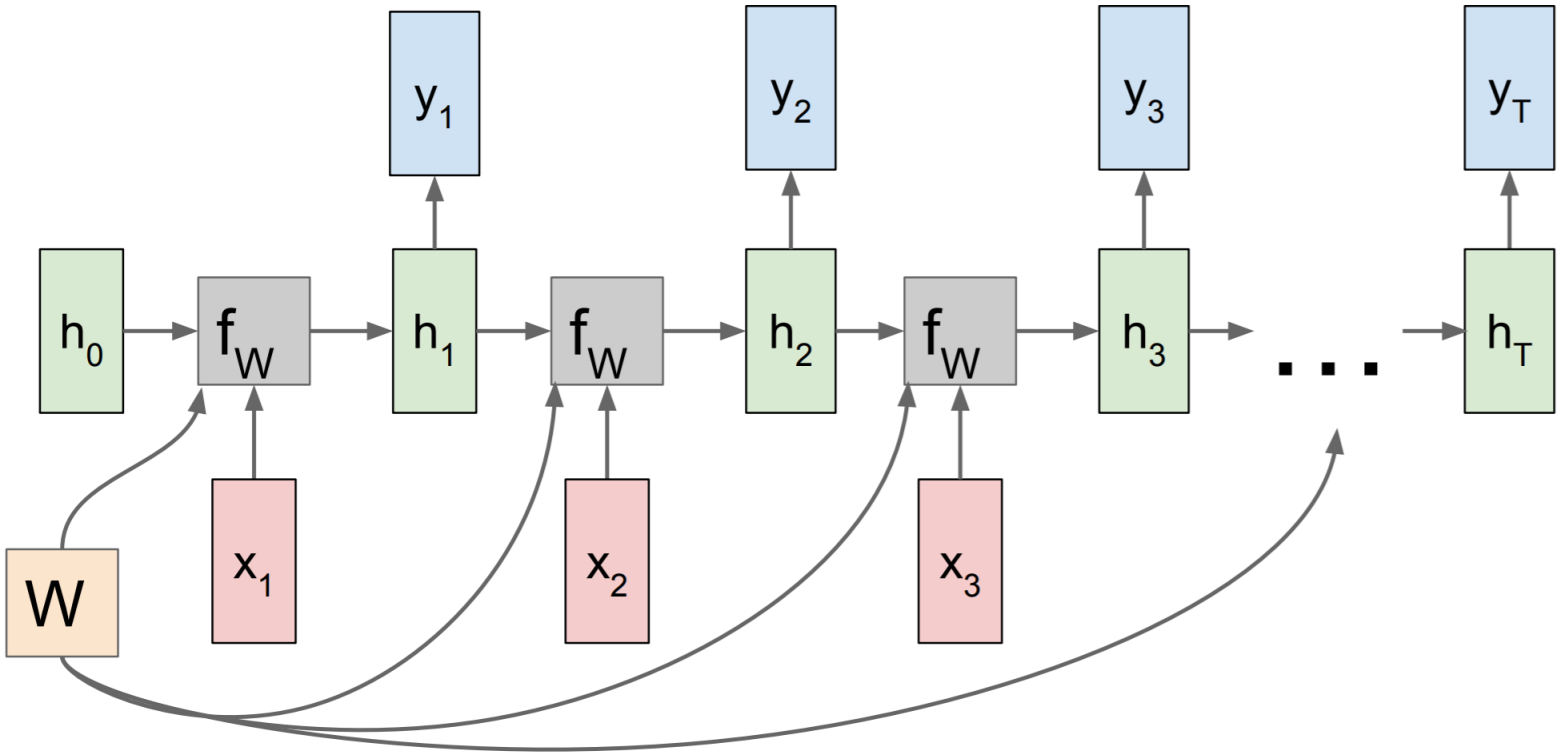
i = [1, 2, 3, …, T-1, T]


yi
L
yGTi
by
Wyh
bh
hi-1
tanh
hi
Whh
Whx
xi

An Elman RNN cell: another representation



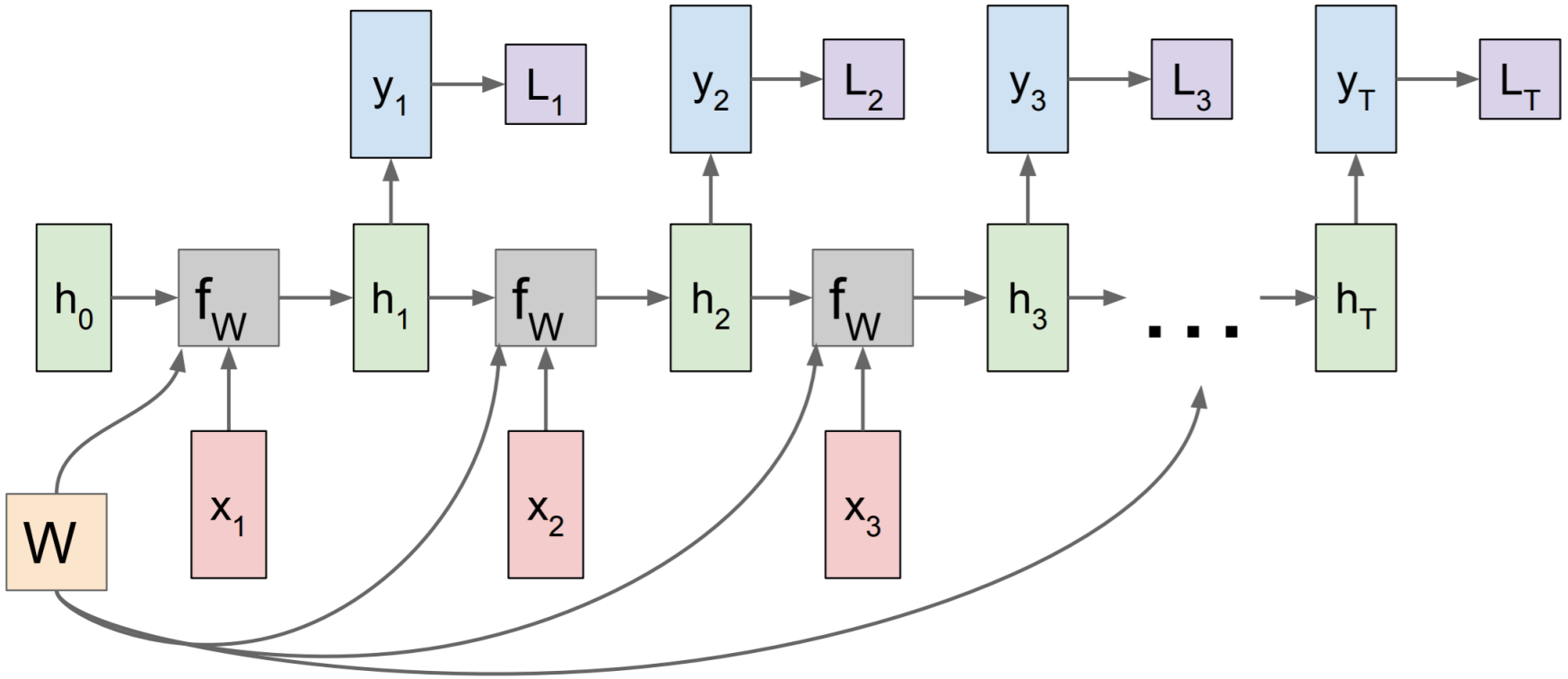
RNN: Computational Graph: many to many



The same weight matrix over all timesteps!

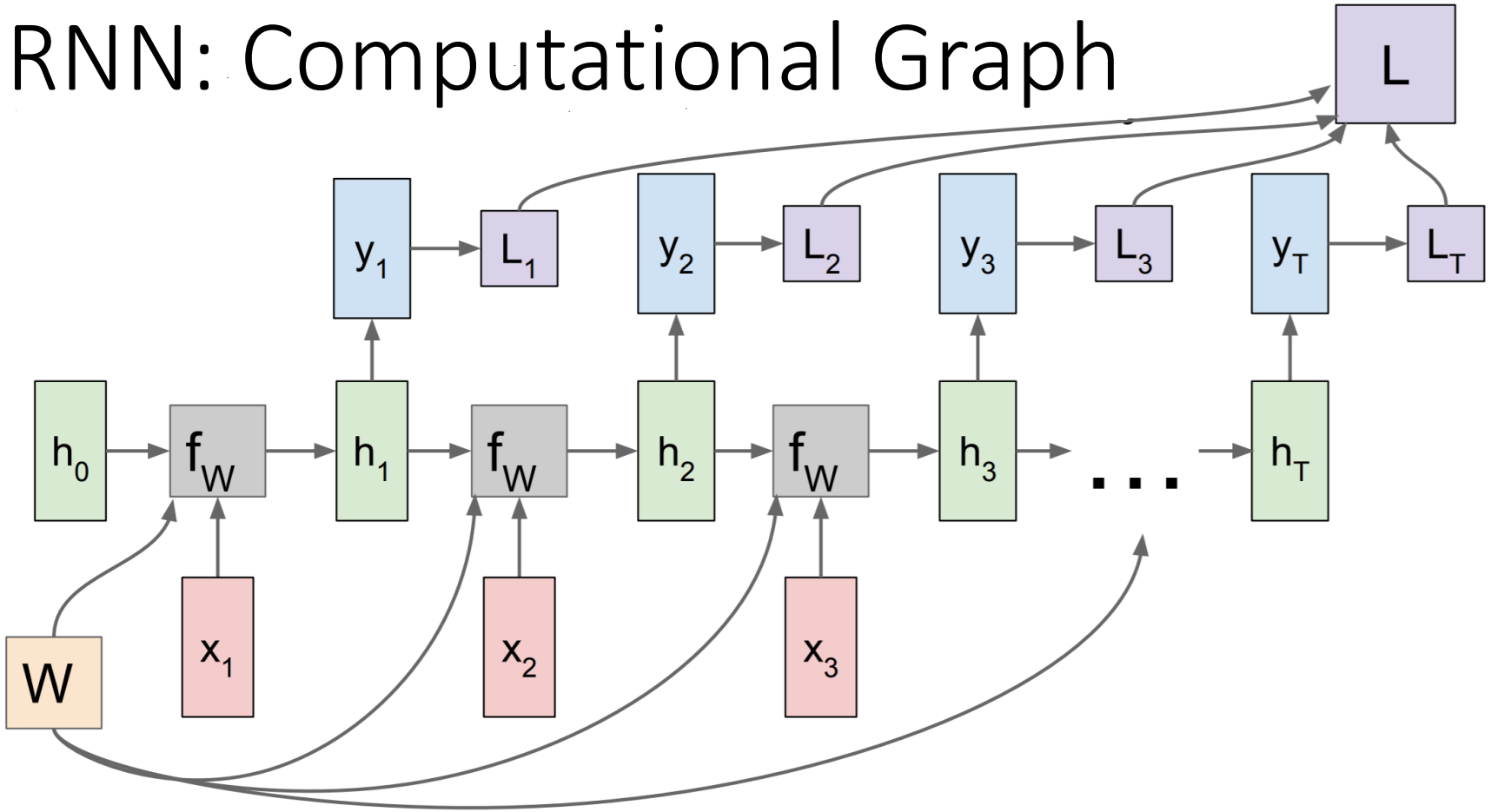
[Fei-Fei Li et al., 2020]

RNN: Computational Graph: many to many



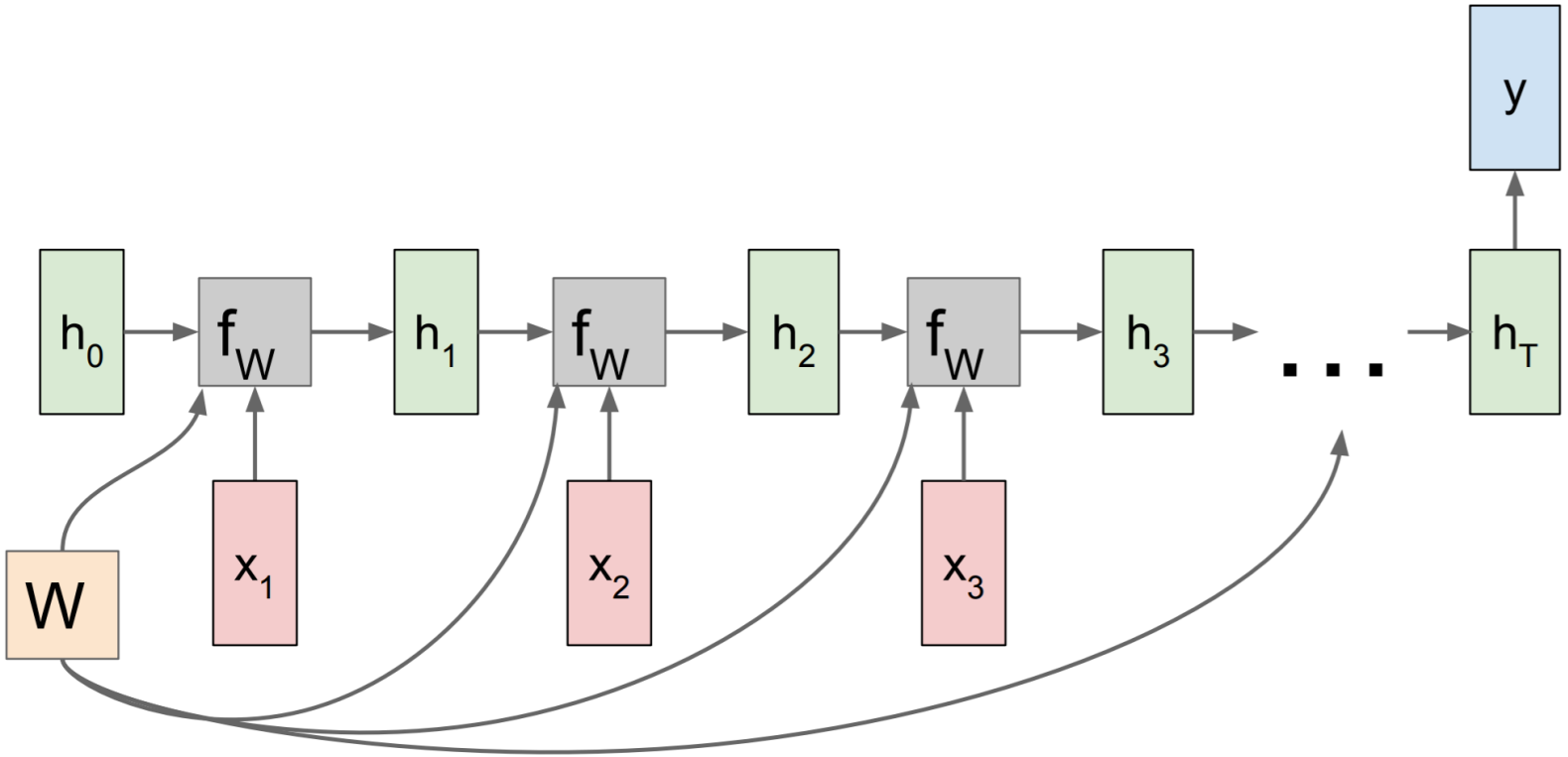
[Fei-Fei Li et al., 2020]

RNN: Computational Graph



[Fei-Fei Li et al., 2020]

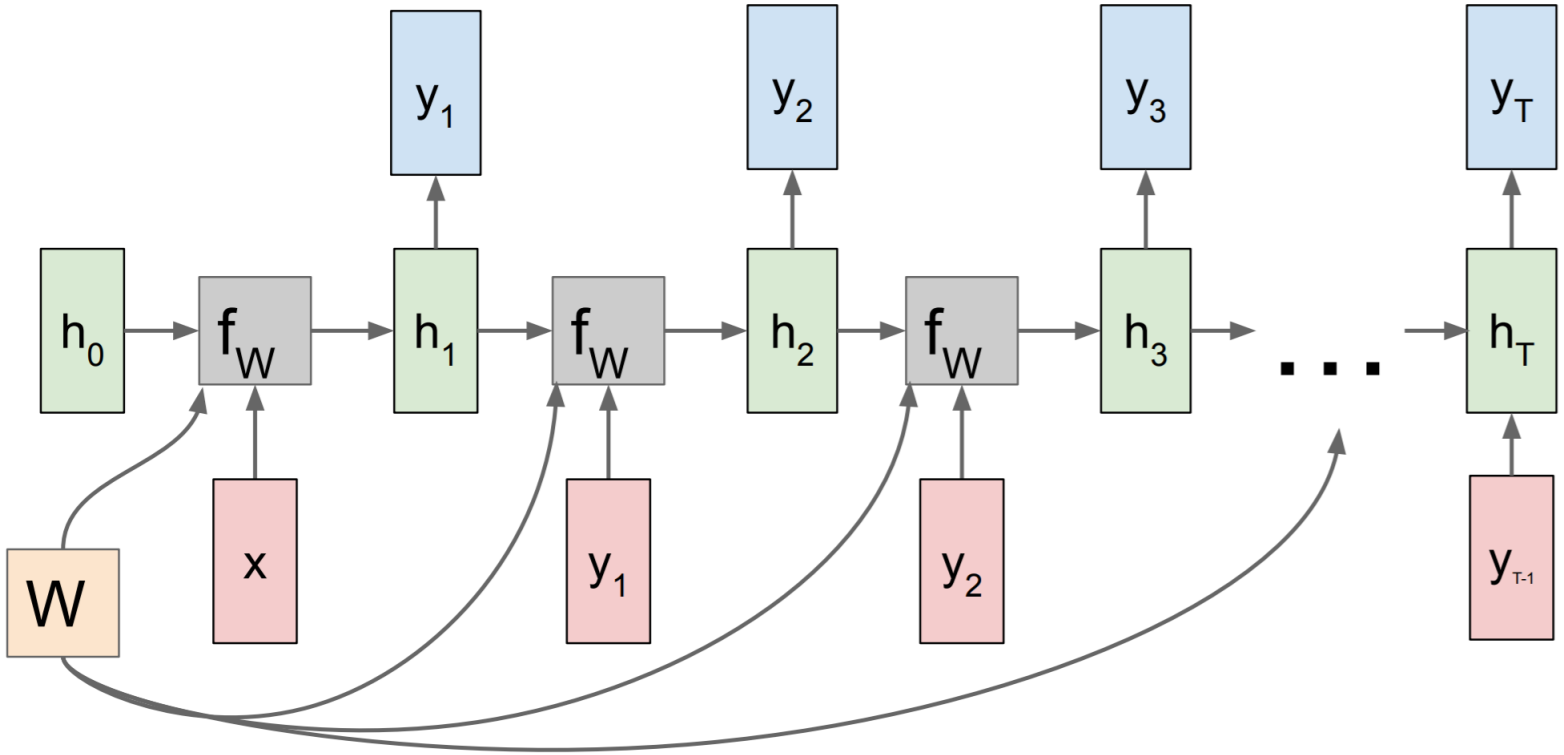
RNN: Computational Graph: many to one



The same weight matrix over all timesteps!

[Fei-Fei Li et al., 2020]

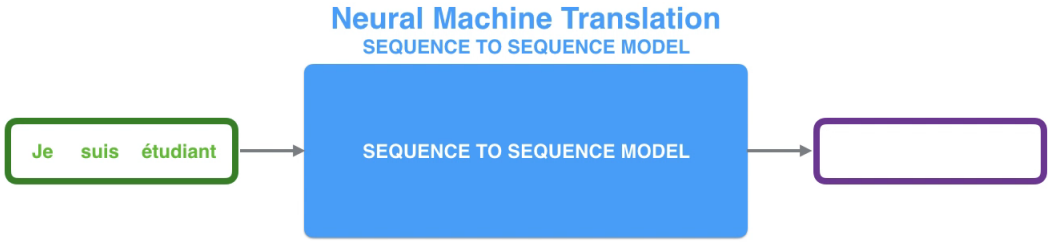
RNN: Computational Graph: one to many



The same weight matrix over all timesteps!

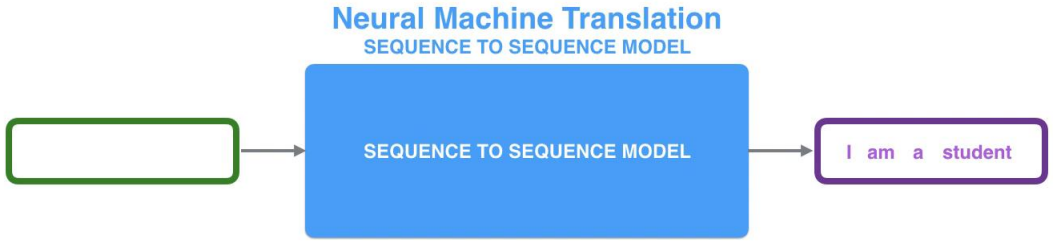
[Fei-Fei Li et al., 2020]

RNN: Sequence to Sequence



[jalammar.github.io, 2018] **videos on this and next slides available by corresponding links*

RNN: Sequence to Sequence



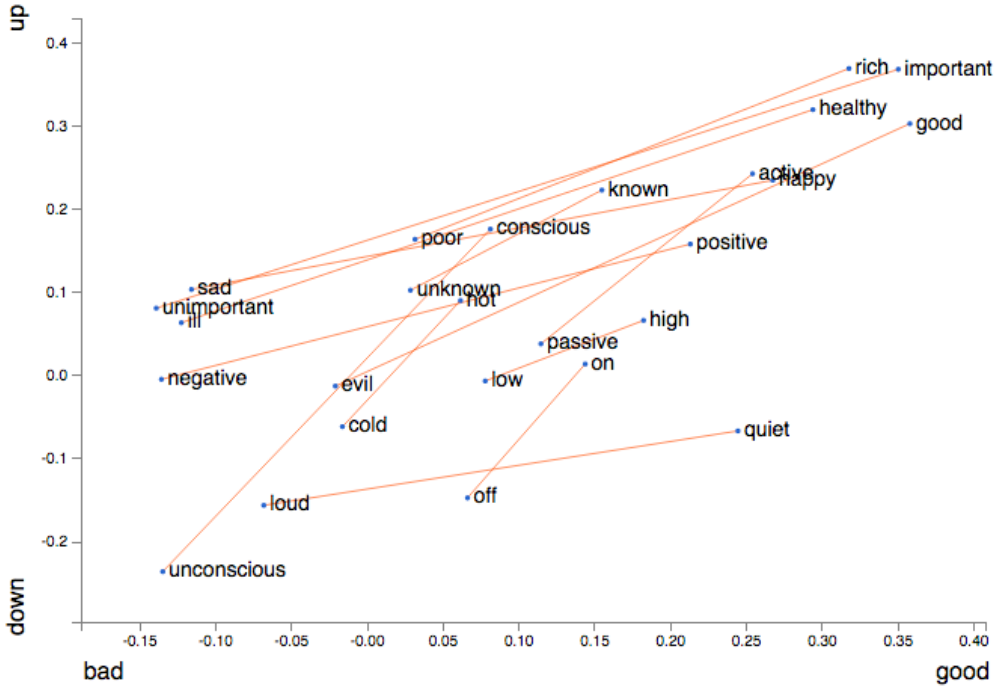
[jalammar.github.io, 2018] **videos on this and next slides available by corresponding links*

RNN: Seq-to-Seq: Encoder – Decoder Arch

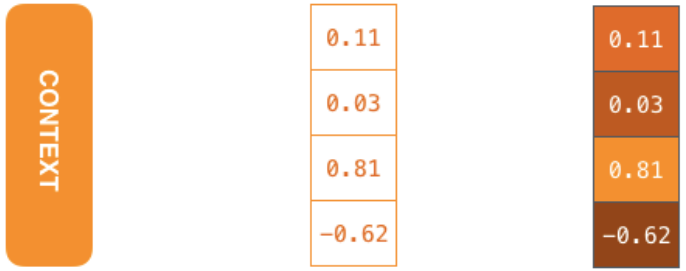
- Problem: How to deal with variable length sequences?

RNN: Seq-to-Seq: Encoder – Decoder Arch

- Problem: How to deal with variable length sequences?
- Possible solution: Encode sequence as a vector (named ***context***)!

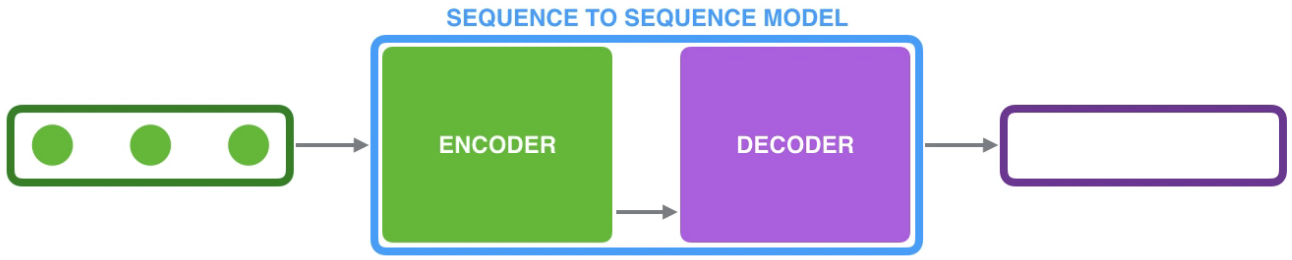


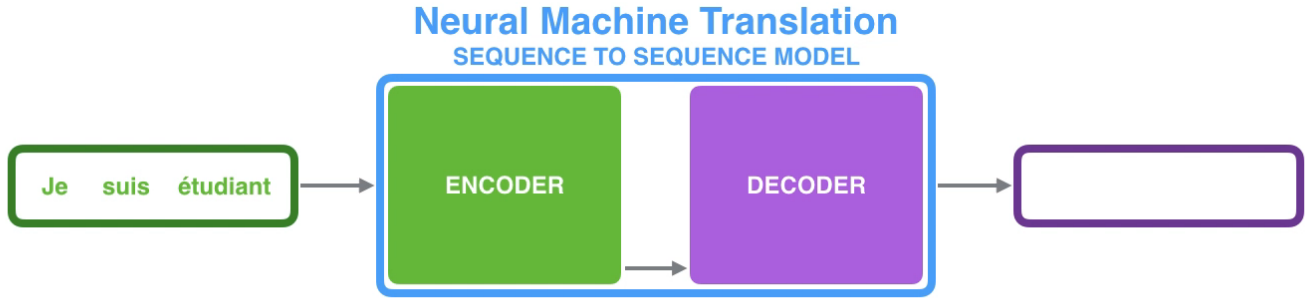
[Piotr Migdal blog, 2017]



[jalammar.github.io, 2018]

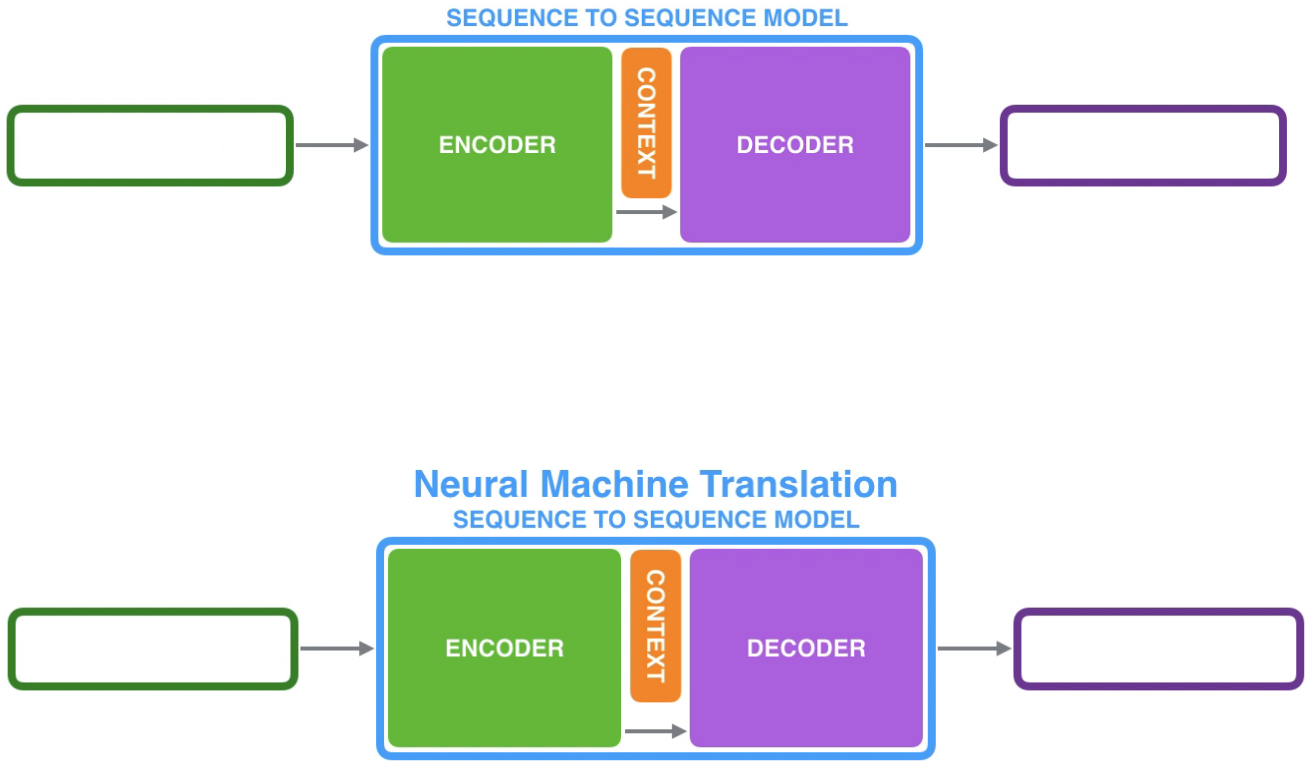
RNN: Seq-to-Seq: Encoder – Decoder Arch





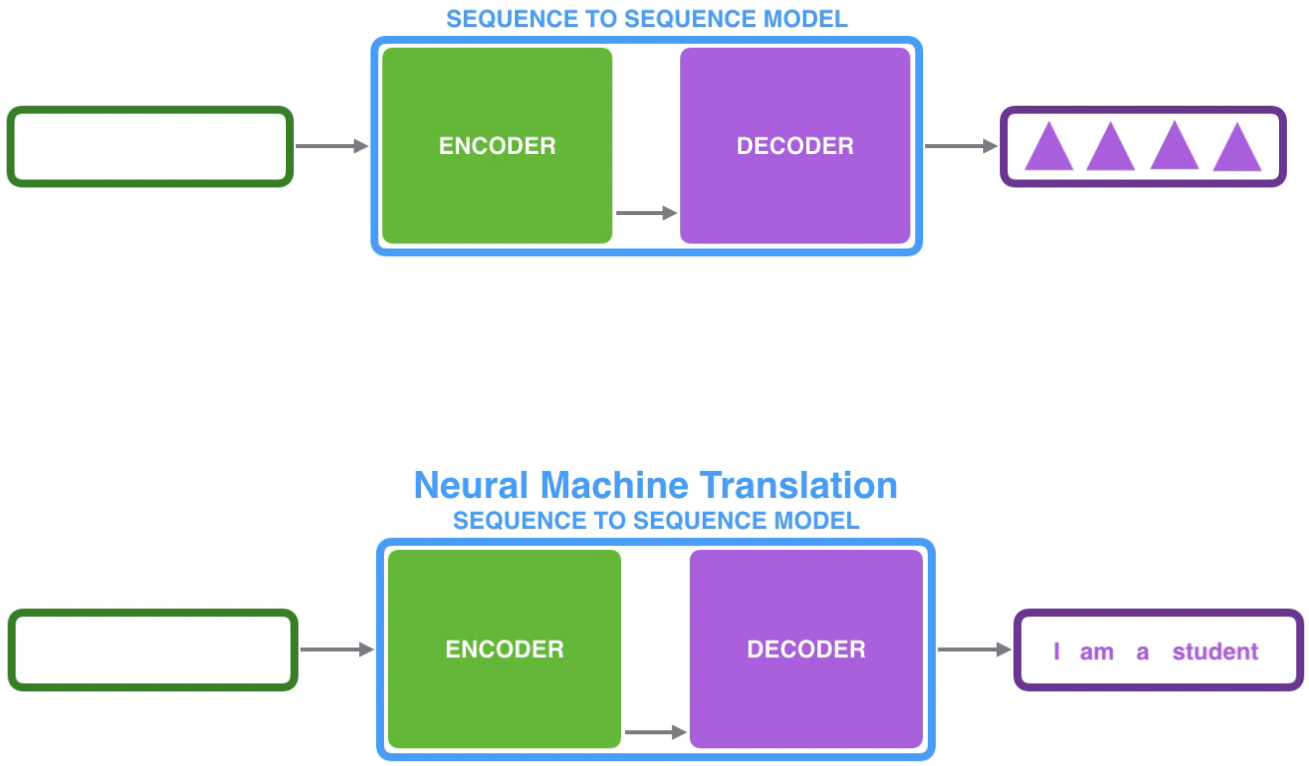
[jalammar.github.io, 2018]

RNN: Seq-to-Seq: Encoder – Decoder Arch



[jalammar.github.io, 2018]

RNN: Seq-to-Seq: Encoder – Decoder Arch



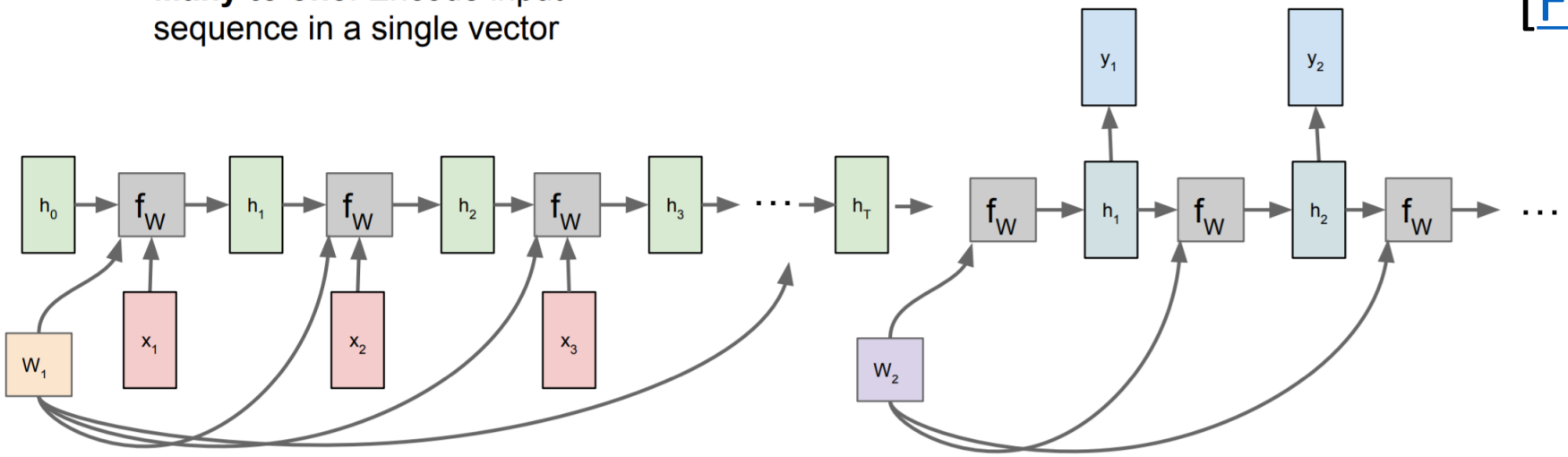
[jalammar.github.io, 2018]

RNN: Seq-to-Seq: Encoder – Decoder Arch

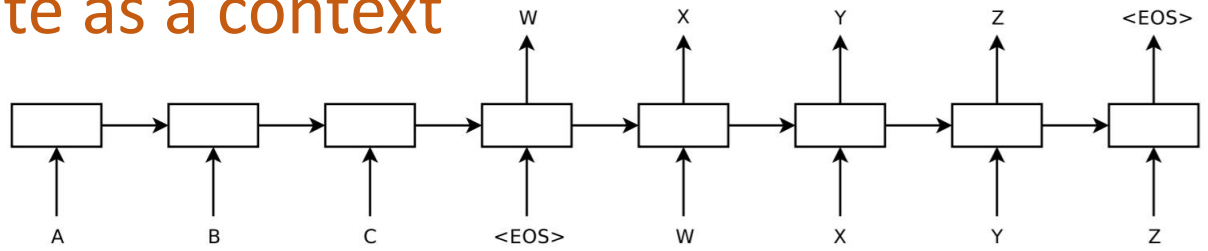
Many to one: Encode input sequence in a single vector

One to many: Produce output sequence from single input vector

[Fei-Fei Li et al., 2020]



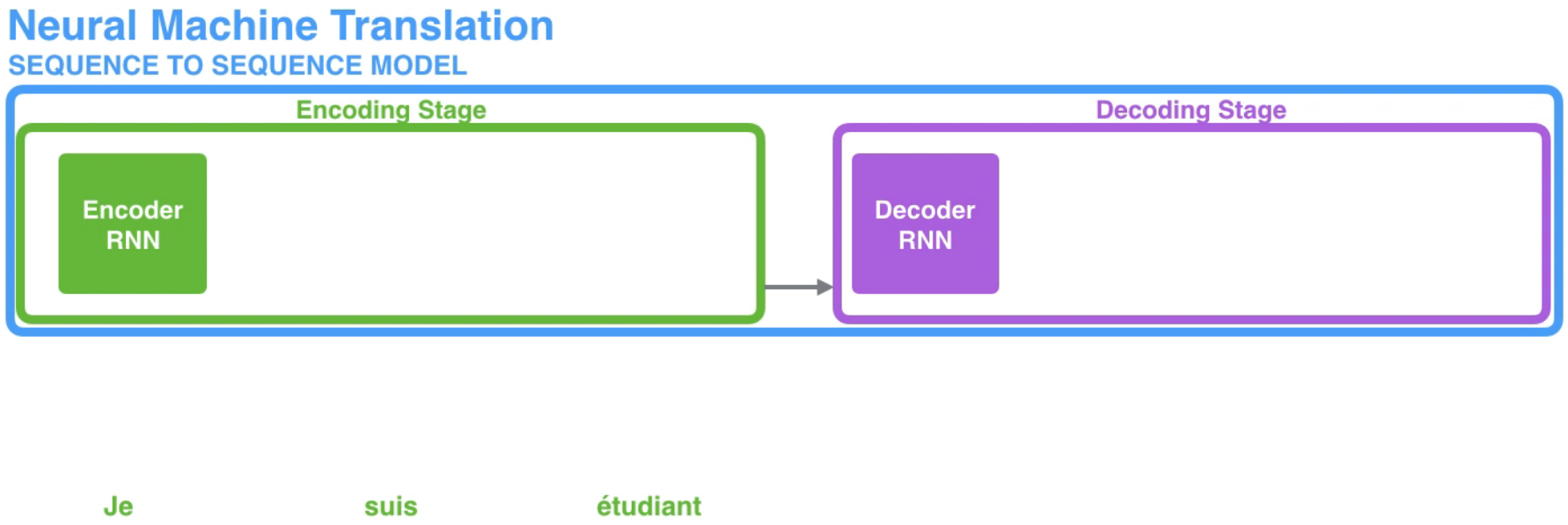
- Using last encoder hidden state as a context



[Sutskever et al., 2014]

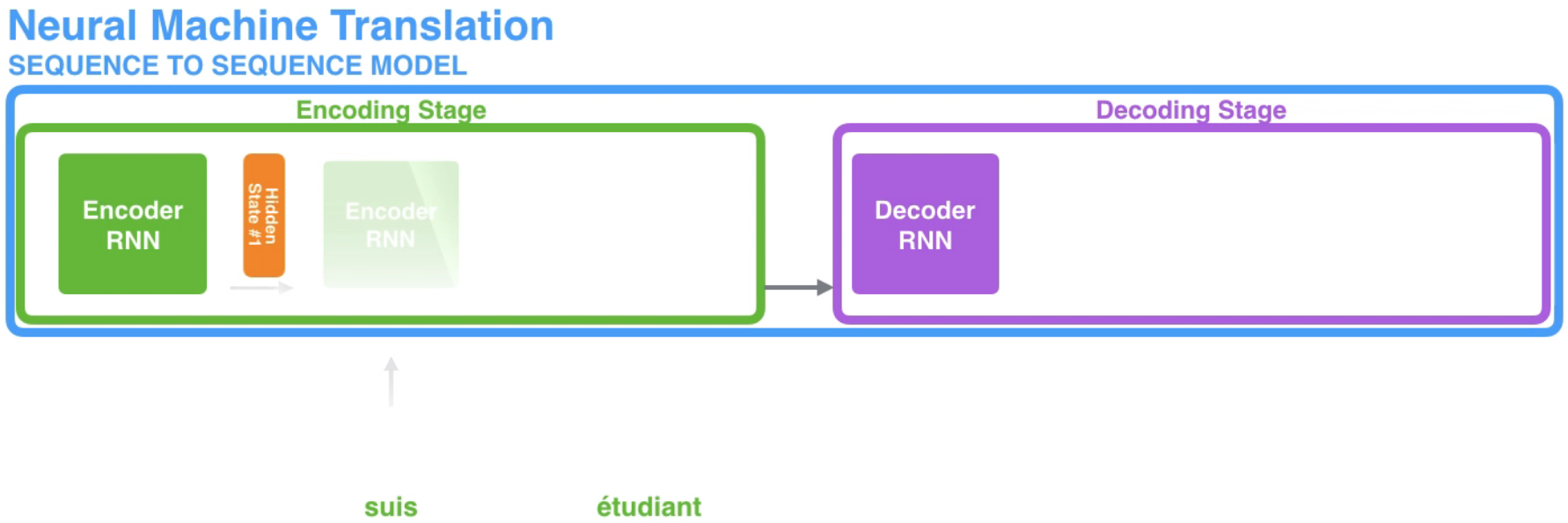
Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

RNN: Seq-to-Seq: Encoder – Decoder Arch



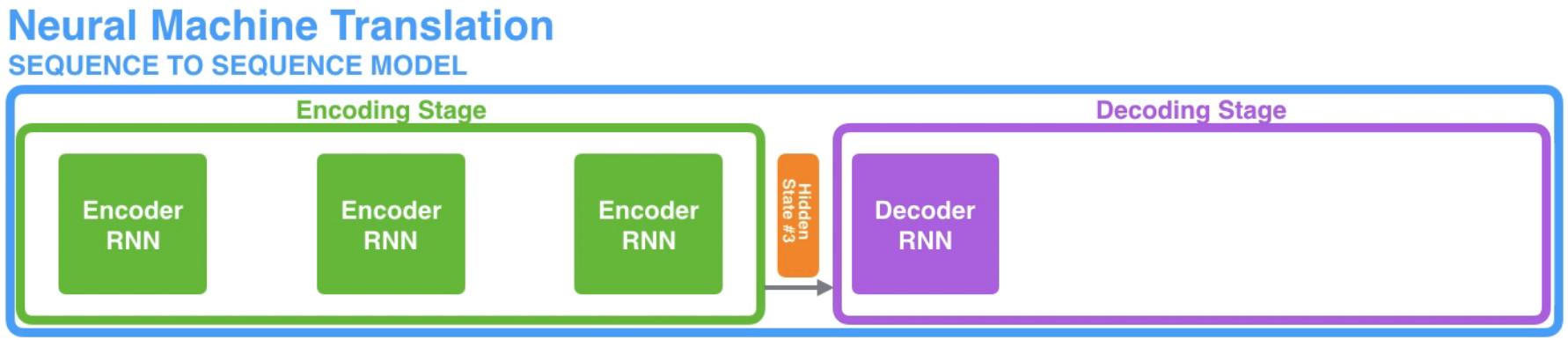
[jalammar.github.io, 2018]

RNN: Seq-to-Seq: Encoder – Decoder Arch



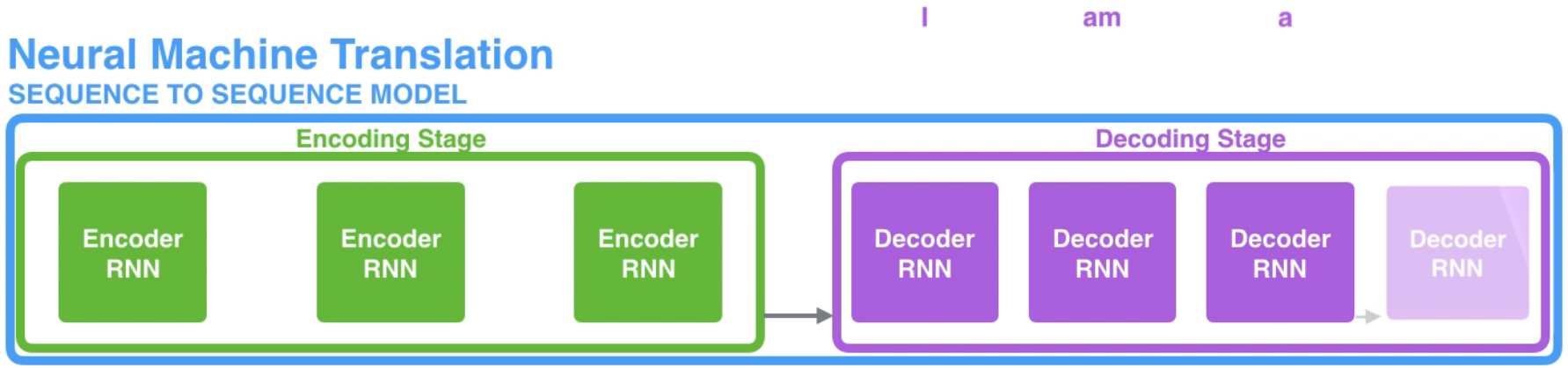
[jalammar.github.io, 2018]

RNN: Seq-to-Seq: Encoder – Decoder Arch



[jalammar.github.io, 2018]

RNN: Seq-to-Seq: Encoder – Decoder Arch



[jalammar.github.io, 2018]

RNN: Training example

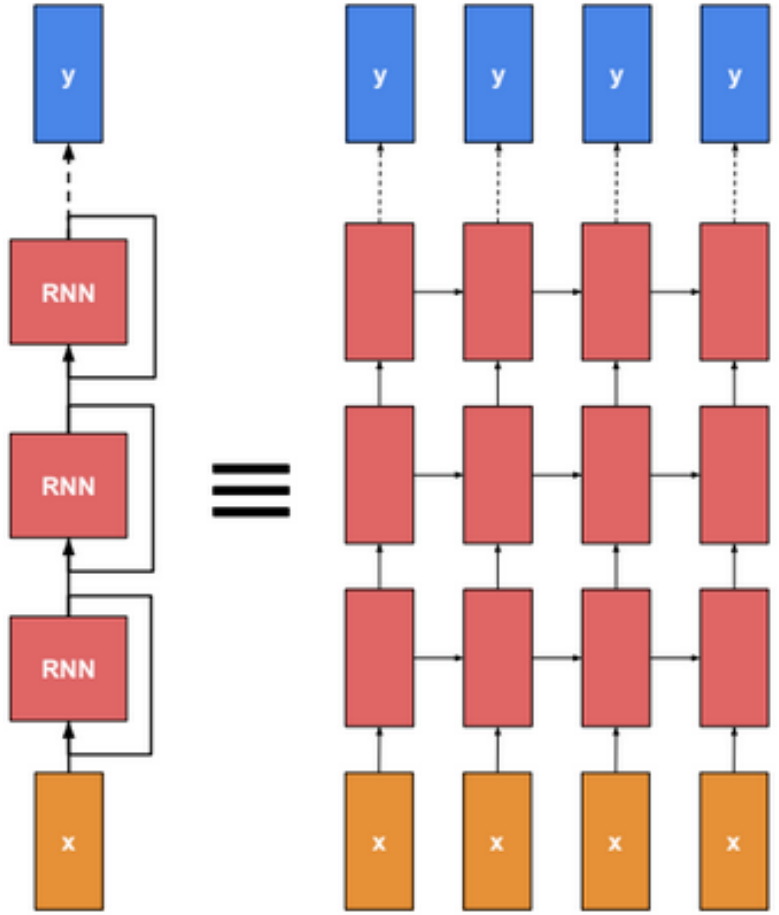
- Try to predict ‘HELLO’ from ‘H’
- Vocabulary = ‘EHLO’
- How to explain the observed number of model parameters?

[colab notebook]

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn (SimpleRNN)       (None, None, 3)           24
_________________________________________________________________
dense (Dense)                (None, None, 4)           16
=================================================================
Total params: 40
Trainable params: 40
Non-trainable params: 0
_________________________________________________________________
loss: [1.8906832]
HLEEE
loss: [1.4843457]
HLELE
loss: [1.2807729]
HLLLL
loss: [1.0895083]
HLLLL
loss: [0.91750413]
HLOLO
loss: [0.7990667]
HLOLO
loss: [0.69465196]
HLOLO
loss: [0.60004973]
HLLLL
loss: [0.5188638]
HLLLL
loss: [0.4523027]
HELLL
```

Multilayer RNN

- Allows to compute more complex representations like in CNNs: Deeper = Better
- Bottom layers can learn syntax, Top layers learn semantics



[sqlml, 2017]

Bidirectional RNN

- Trying to obtain both past and future contexts
- Applications: filling in missing words, annotating tokens (e.g., for named entity recognition), and encoding sequences wholesale as a step in a sequence processing pipeline (e.g., for machine translation).

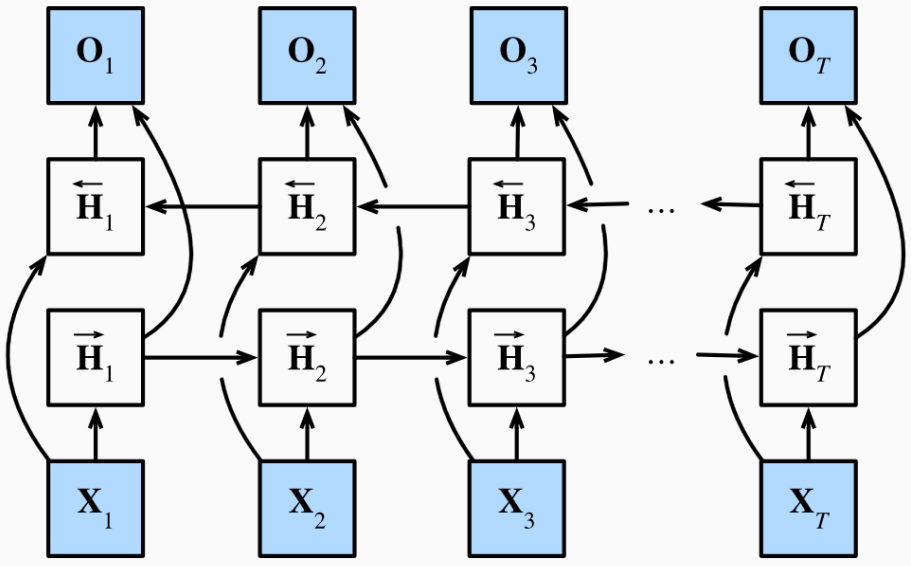
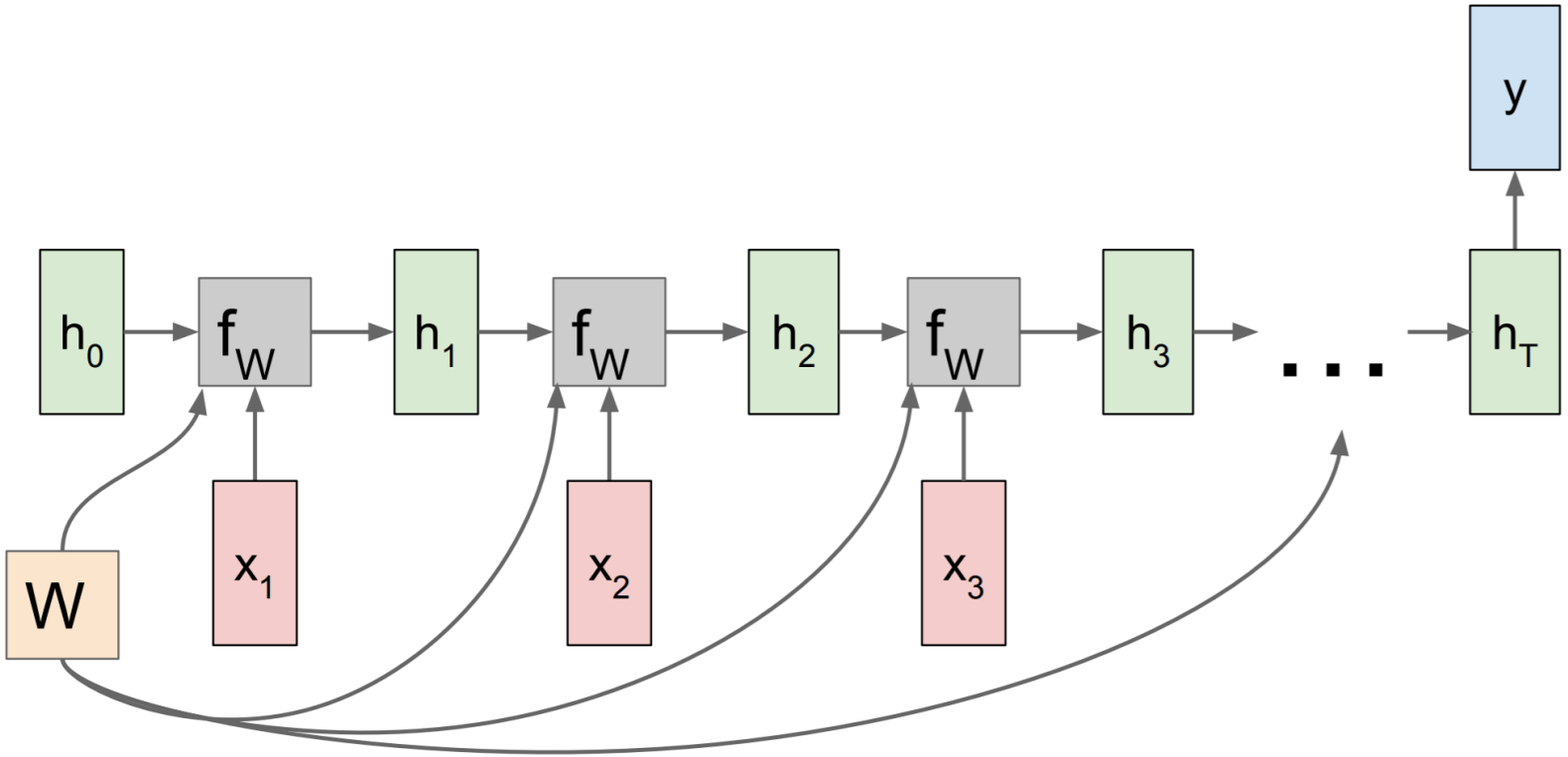


Fig. 9.4.2 Architecture of a bidirectional RNN.

[d2l.ai]

LSTM Cell

Gradient flow through Elman RNN

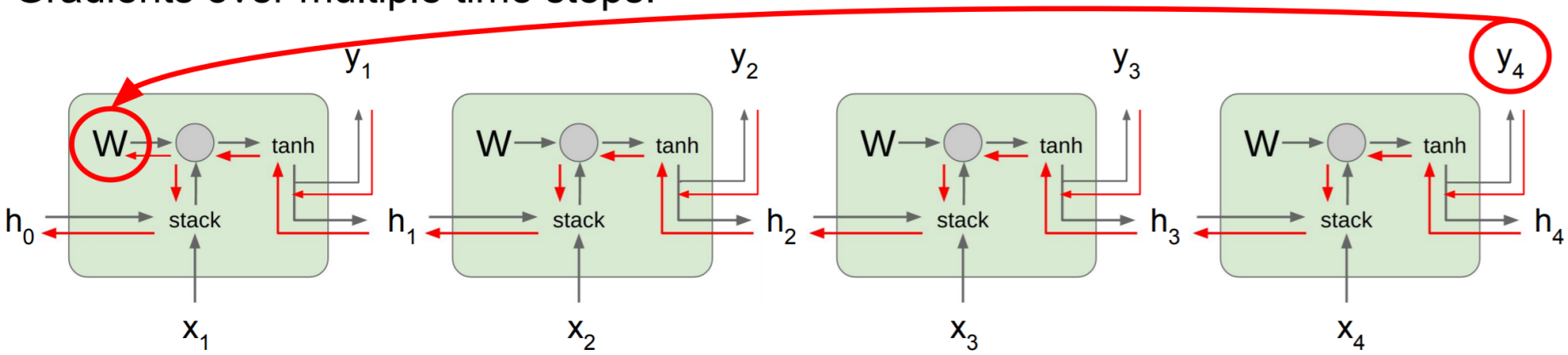


The same weight matrix over all timesteps!

[Fei-Fei Li et al., 2020]

Gradient flow through Elman RNN

Gradients over multiple time steps:



$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial h_t}{\partial h_{t-1}} = tanh'(W_{hh} h_{t-1} + W_{xh} x_t) W_{hh}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

- **Vanishing or exploding gradients!**

[Fei-Fei Li et al., 2020]

Long Short-Term Memory

- **Elman RNN**

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

- **LSTM**

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

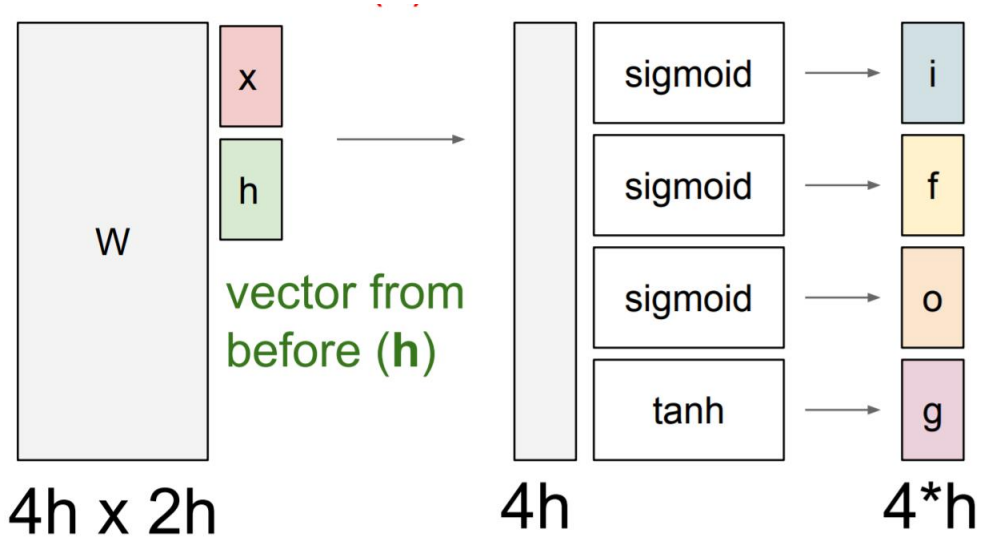
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

[Hochreiter and Schmidhuber, 1997]

[Fei-Fei Li et al., 2020]

Long Short-Term Memory

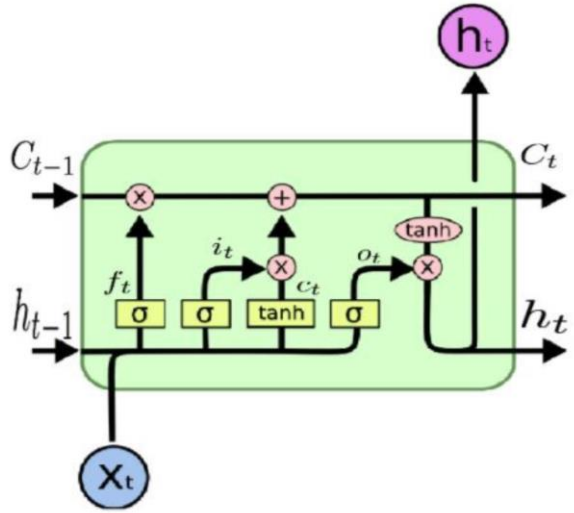


i: Input gate, whether to write to cell
f: Forget gate, Whether to erase cell
o: Output gate, How much to reveal cell
g: Gate gate (?), How much to write to cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

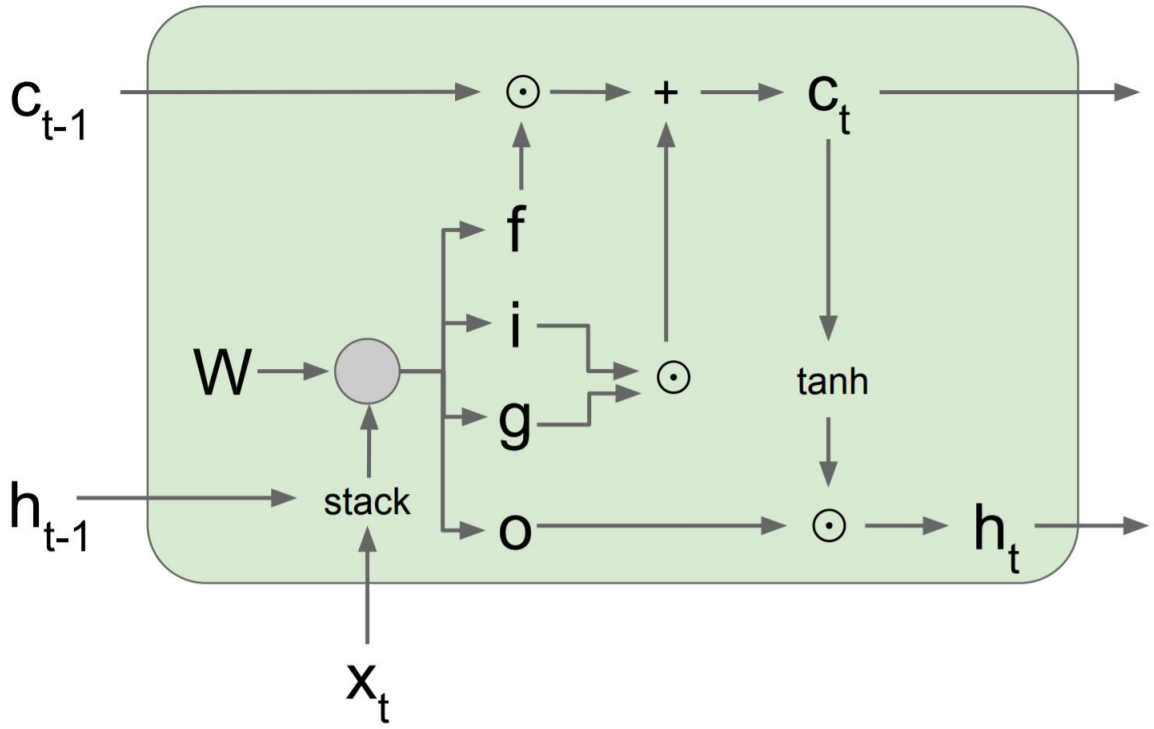
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$



[Fei-Fei Li et al., 2020]

Long Short-Term Memory: Graph



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

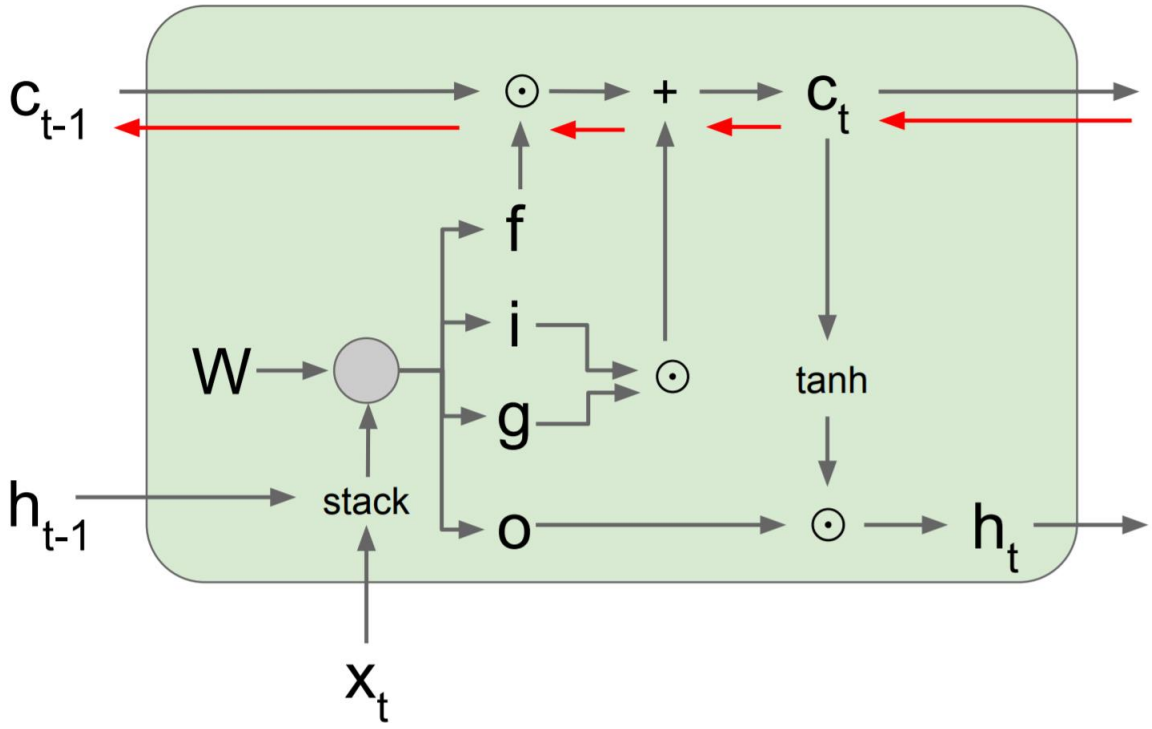
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

[Fei-Fei Li et al., 2020]

Long Short-Term Memory: Gradient flow

[Hochreiter et al., 1997]



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f, no matrix multiply by W

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

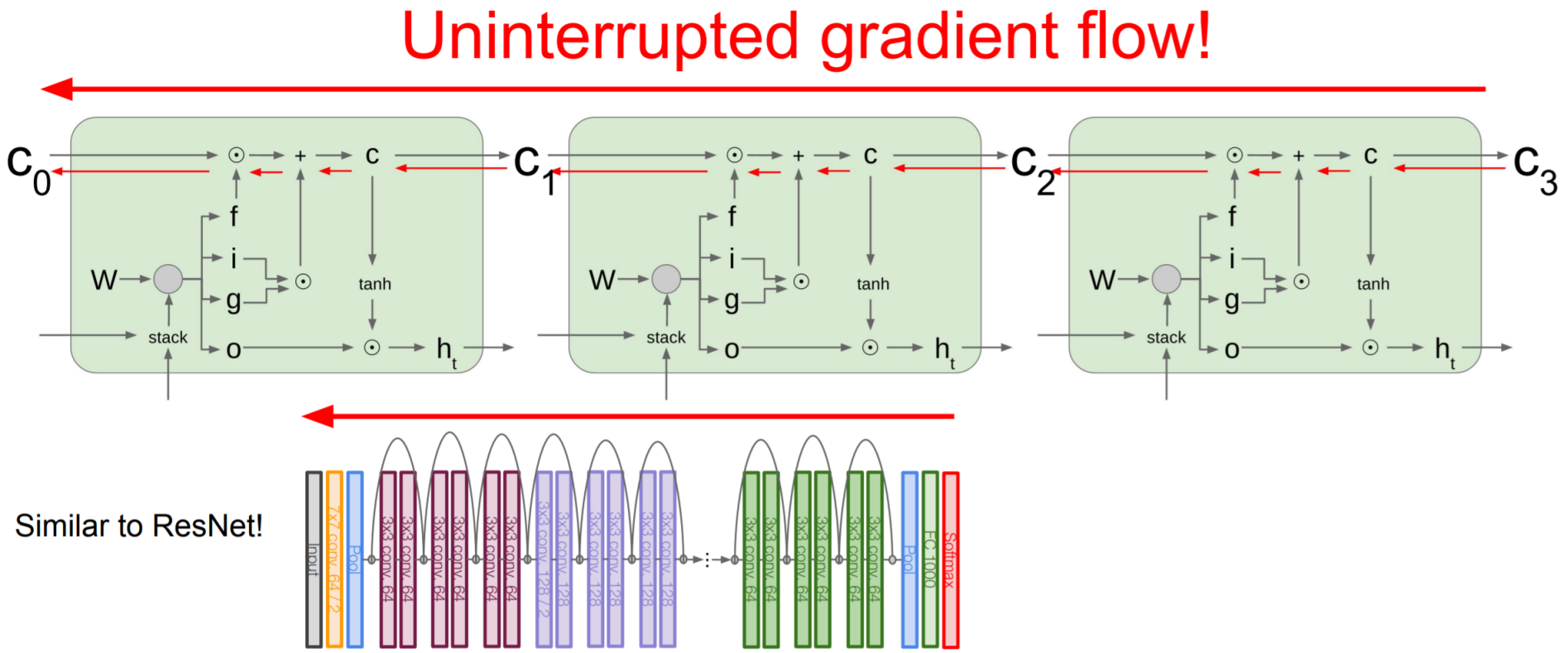
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

- Looks like…?

[Fei-Fei Li et al., 2020]

Long Short-Term Memory: Gradient flow



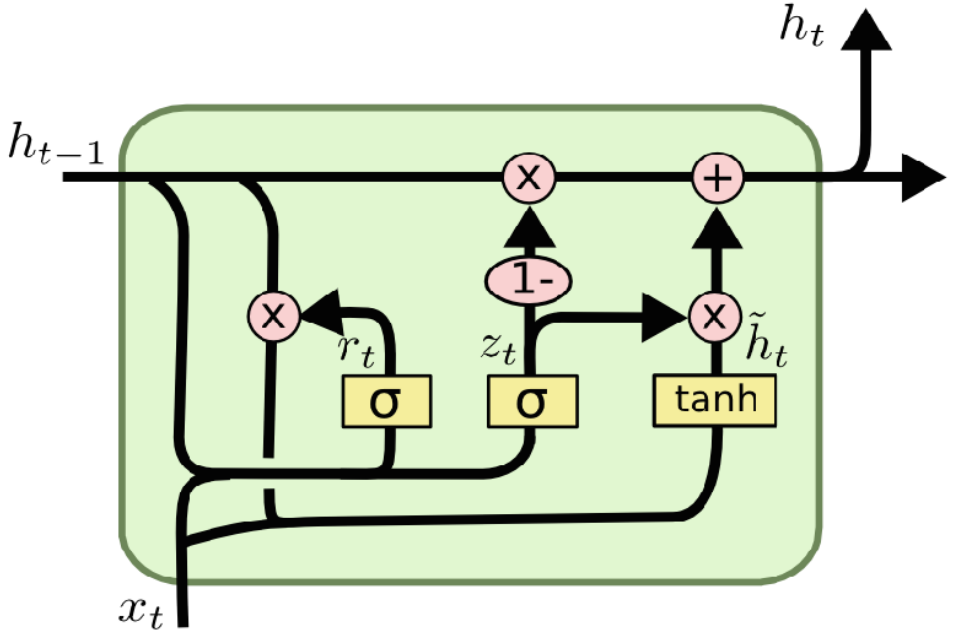
- Vanishing gradients problem solution!
  - Long-term dependencies better learned!

[Fei-Fei Li et al., 2020]

Gated Recurrent Unit

GRU [*Learning phrase representations using rnn encoder-decoder for statistical machine translation*, Cho et al. 2014]

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$
$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$
$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$



[Fei-Fei Li et al., 2020]

[technopremium]

Attention mechanism

Attention in sequence to sequence

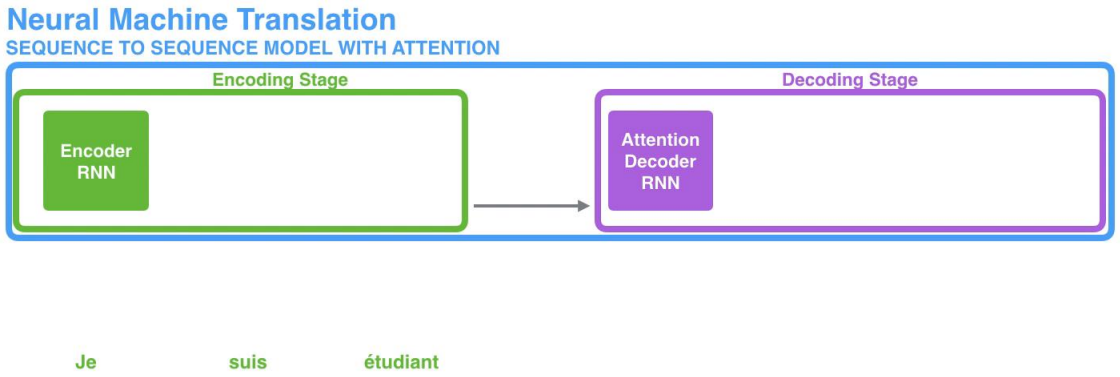
- The ***context*** vector turned out to be a bottleneck for Encoder-Decoder types of models
- ***Attention*** allows the model to focus on the relevant parts of the input sequence as needed

At time step 7, the attention mechanism enables the decoder to focus on the word "étudiant" ("student" in french) before it generates the English translation. This ability to amplify the signal from the relevant part of the input sequence makes attention models produce better results than models without attention.

[jalammar.github.io, 2018]

Attention in sequence to sequence

- Introduced in papers: [Bahdanau et al., 2014] and [Luong et al., 2015].
- We use as context ***all hidden states from encoder*** and pass them to the ***attention module*** inside decoder



[jalammar.github.io, 2018]

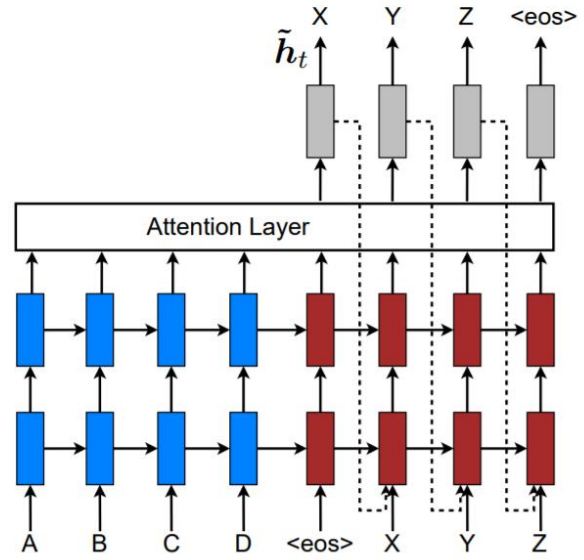
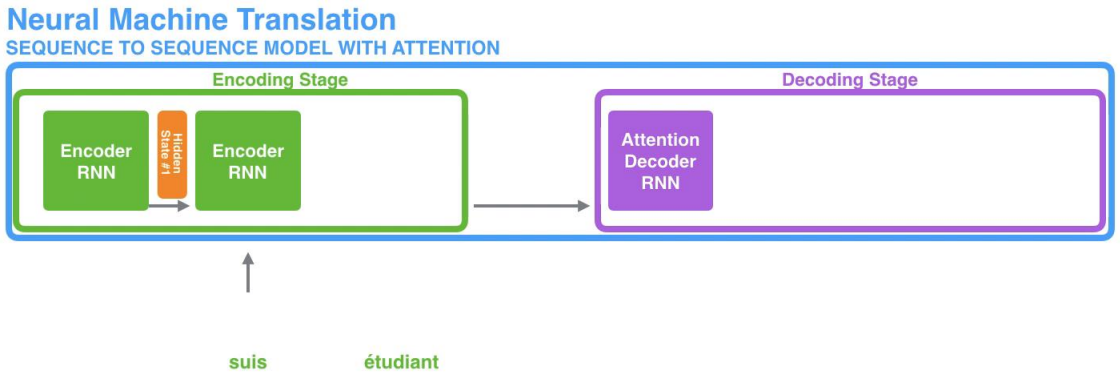


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{h}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.

Attention in sequence to sequence

- Introduced in papers: [Bahdanau et al., 2014] and [Luong et al., 2015].
- We use as context ***all hidden states from encoder*** and pass them to the ***attention module*** inside decoder



[jalammar.github.io, 2018]

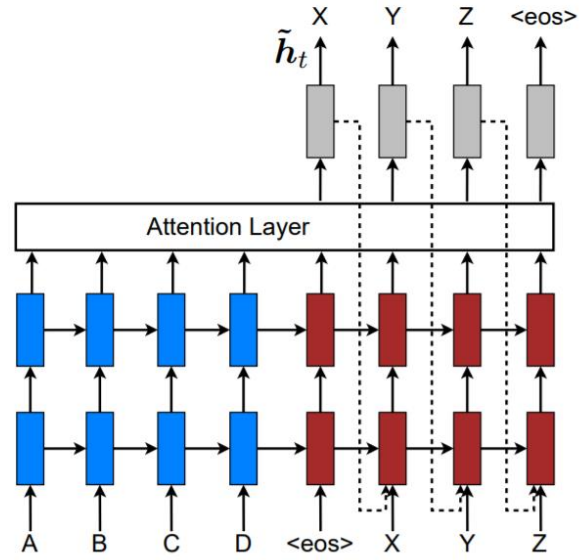
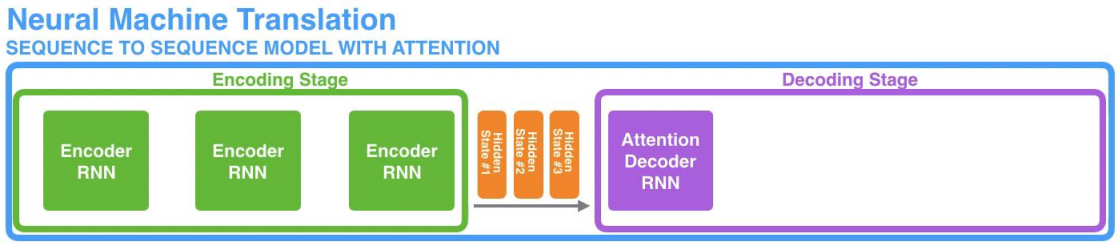


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{h}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.

Attention in sequence to sequence

- Introduced in papers: [Bahdanau et al., 2014] and [Luong et al., 2015].
- We use as context ***all hidden states from encoder*** and pass them to the ***attention module*** inside decoder



[jalammar.github.io, 2018]

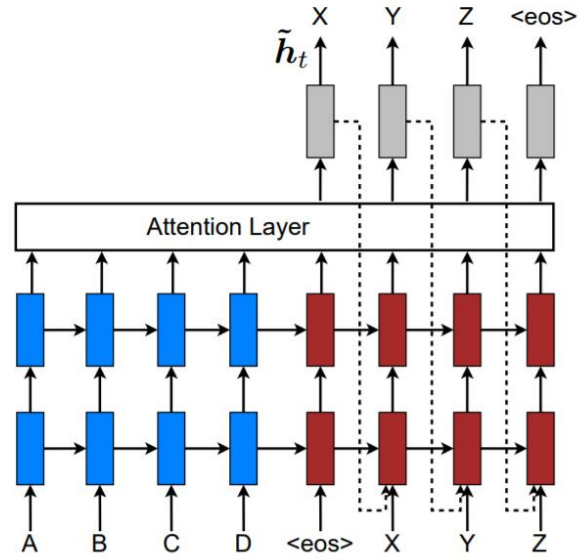
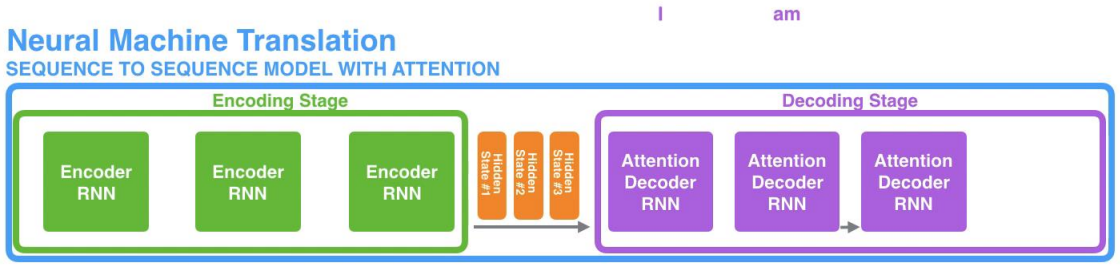


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{h}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.

Attention in sequence to sequence

- Introduced in papers: [Bahdanau et al., 2014] and [Luong et al., 2015].
- We use as context ***all hidden states from encoder*** and pass them to the ***attention module*** inside decoder



[jalammar.github.io, 2018]

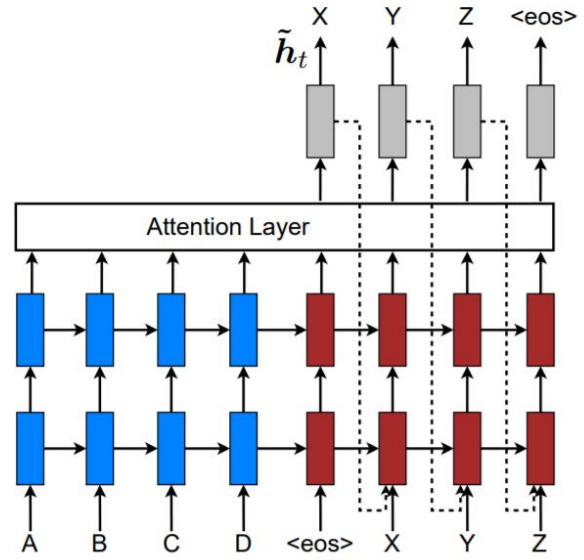
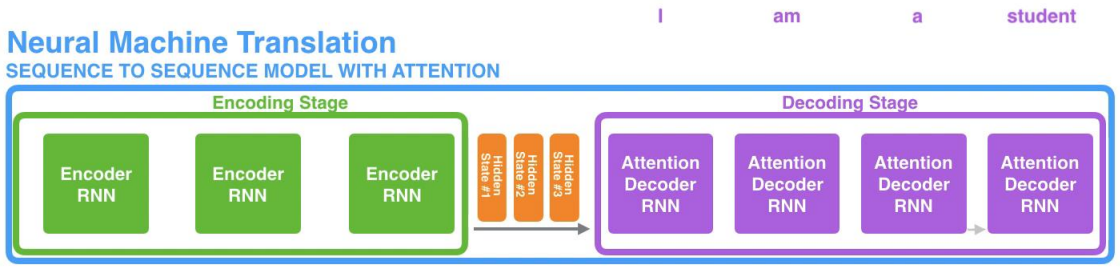


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{h}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.

Attention in sequence to sequence

- Introduced in papers: [Bahdanau et al., 2014] and [Luong et al., 2015].
- We use as context ***all hidden states from encoder*** and pass them to the ***attention module*** inside decoder



[jalammar.github.io, 2018]

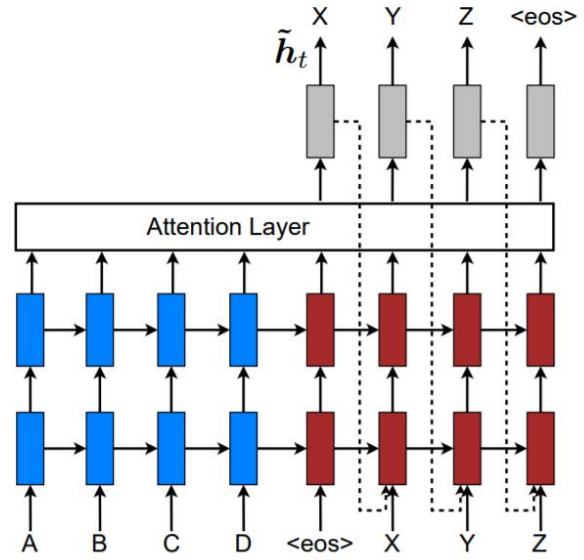


Figure 4: **Input-feeding approach** – Attentional vectors $\tilde{h}_t$ are fed as inputs to the next time steps to inform the model about past alignment decisions.

Attention in sequence to sequence: Decoder

$$h_{i\,D} = F_{w\,D}(h_{i-1\,D}, y_{i-1\,D}, c_i)$$

The context vector c_i is, then, computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \tag{5}$$

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}, \tag{6}$$

where

$$e_{ij} = a(s_{i-1}, h_j)$$

is an *alignment model* which scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden state s_{i-1} (just before emitting y_i, Eq. (4)) and the j-th annotation h_j of the input sentence.

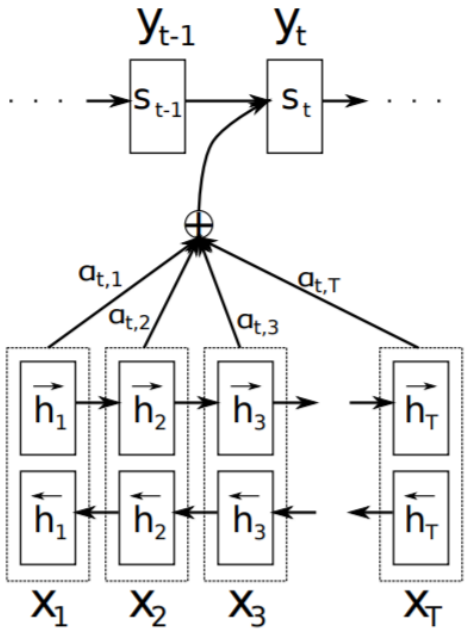


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

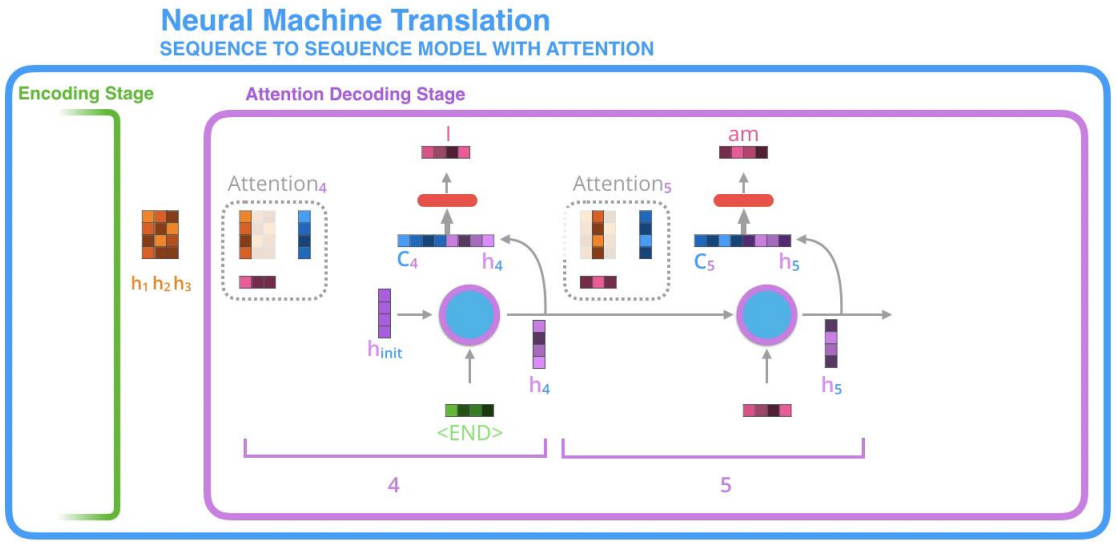
[Bahdanau et al., 2014]

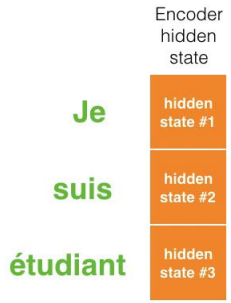
Attention in sequence to sequence

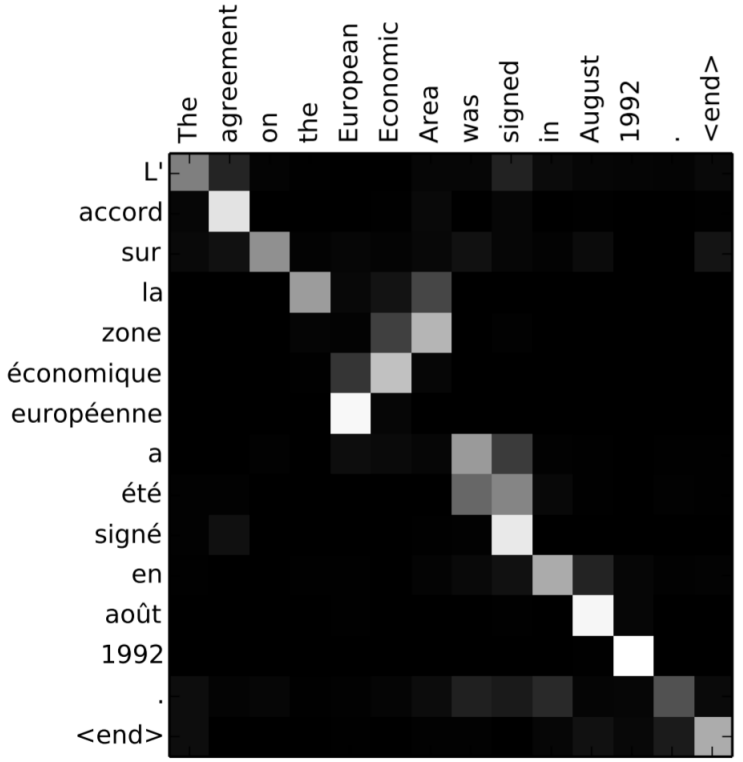
- Give each hidden state a score
- Multiply each hidden states by its softmaxed score, thus amplifying hidden states with high scores
- Sum score-weighted hidden states to create context vector for this timestep

[jalammar.github.io, 2018]

Attention in sequence to sequence







[Bahdanau et al., 2014]

[jalammar.github.io, 2018]

Transformers

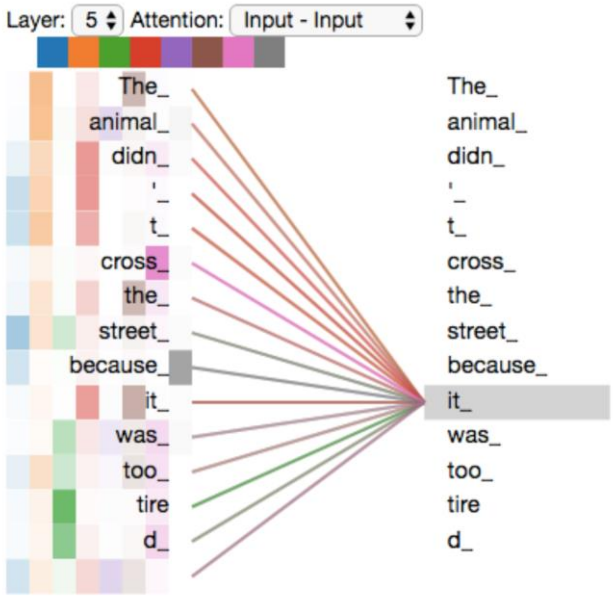
Transformer

- State of the art solutions in the Natural Language Processing obtained with the Transformer architecture which relates only on attention mechanism (BERT, GPT - models)
- Introduced in paper [Vaswani et al., 2017]

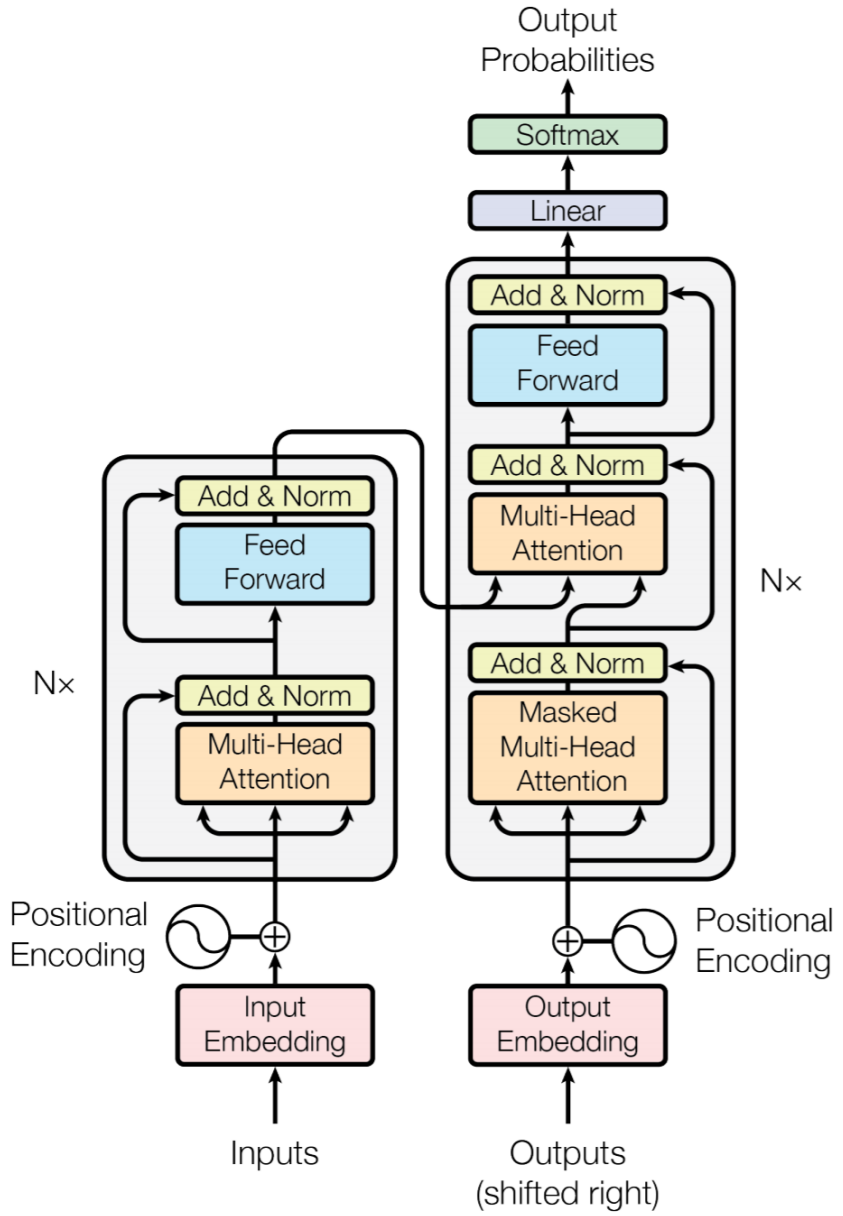
[jalammar.github.io]

Transformer

- Key Ideas: Multi-Head self-attention and positional encoding



[jalammar.github.io]



[Vaswani et al., 2017]

Conclusions

Summary

- Sequence related tasks can be solved by Recurrent Neural Networks
- There are several implementations of RNN's: Elman (Simple, Vanilla) RNN, LSTM, GRU (the last two are better to use in practice) and more
- RNNs can be used in multilayer or bi-directional (BiLSTM) configurations to learn deeper representations and long-term range dependencies
- Encoder-decoder architecture is a common architecture in practice and used not only in sequence to sequence tasks
- Attention is all you need. State of the art solutions in sequence to sequence task obtained with Transformer architecture based on this technique