Université du Luxembourg

UNIVERSITÉ DU LUXEMBOURG

Natural Language Processing

RNNs and Transformer

ARIA NOURBAKHSH, SALIMA LAMSIYAH, UNIVERSITY OF LUXEMBOURG

aria.nourbakhsh@uni.lu
Salima.lamsiyah@uni.lu

Lecture Plan

1. Recurrent Neural Networks

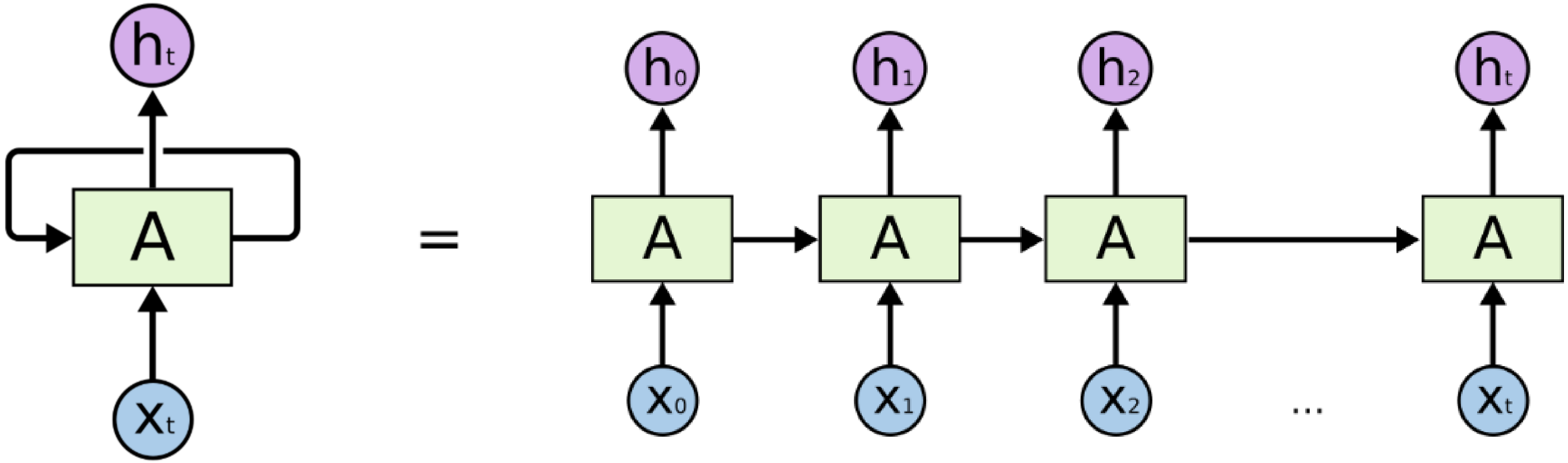
2. Transformers

Recurrent Neural Networks

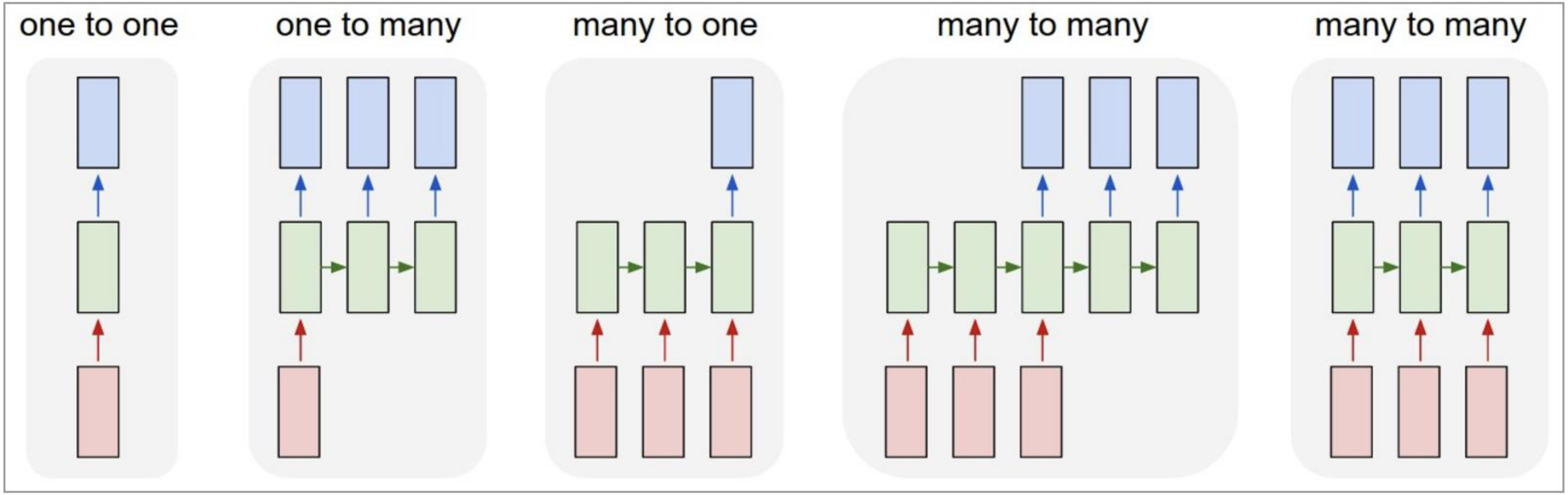
Why do we need RNNs

- How to represent these two sentences with word embeddings? Adding vectors? Averaging?
  - the dog eats the cat.
  - the cat eats the dog.

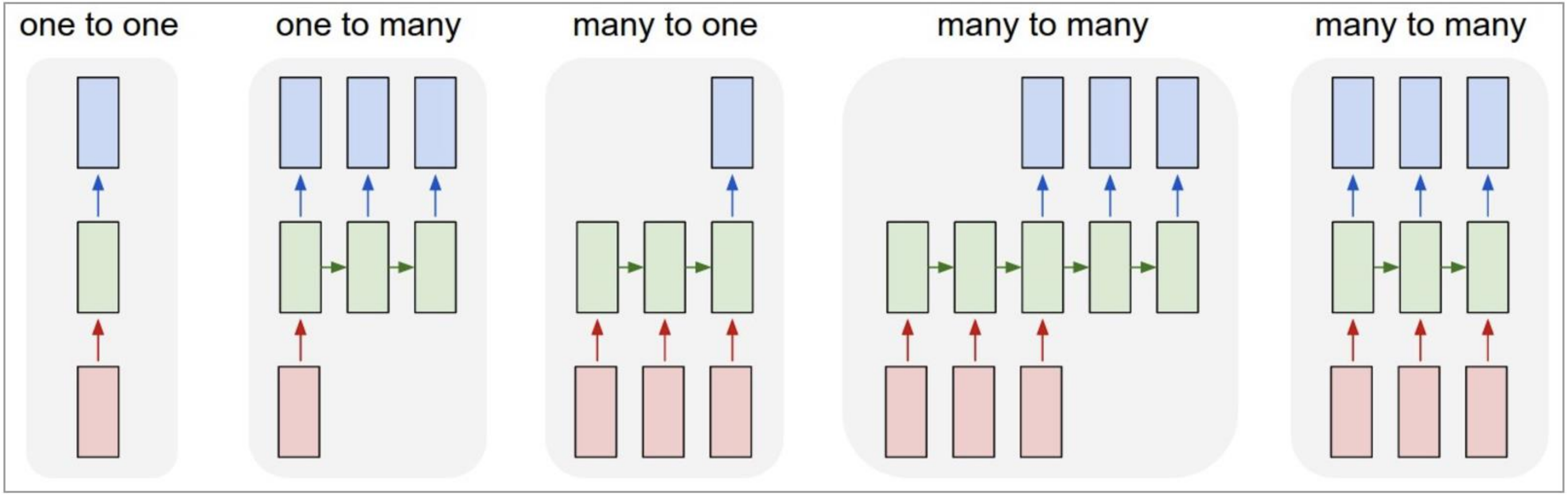
RNN design



Sequence Learning



Sequence Learning



Vanilla NN

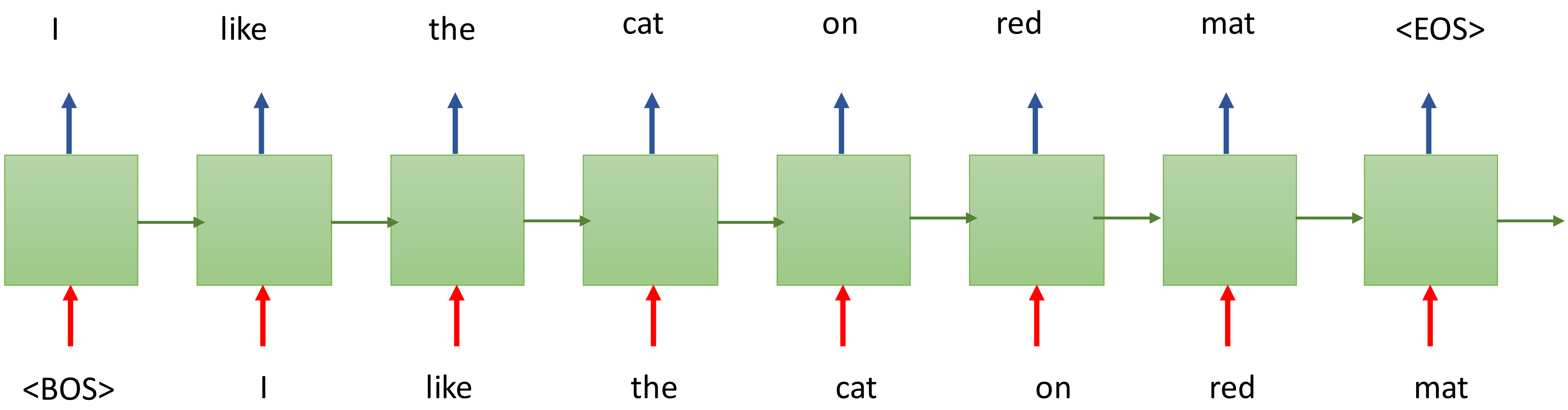
Image Captioning

Classification
(Sentiment analysis)

Sequence to Sequence
Machine Translation
Chatbot

Language Modelling
Video captioning
POS Tagging
NER

Neural Language Modelling



Introduction to RNNs

- **Language & Temporal Phenomenon:** Language unfolds over time; requires sequential modeling.

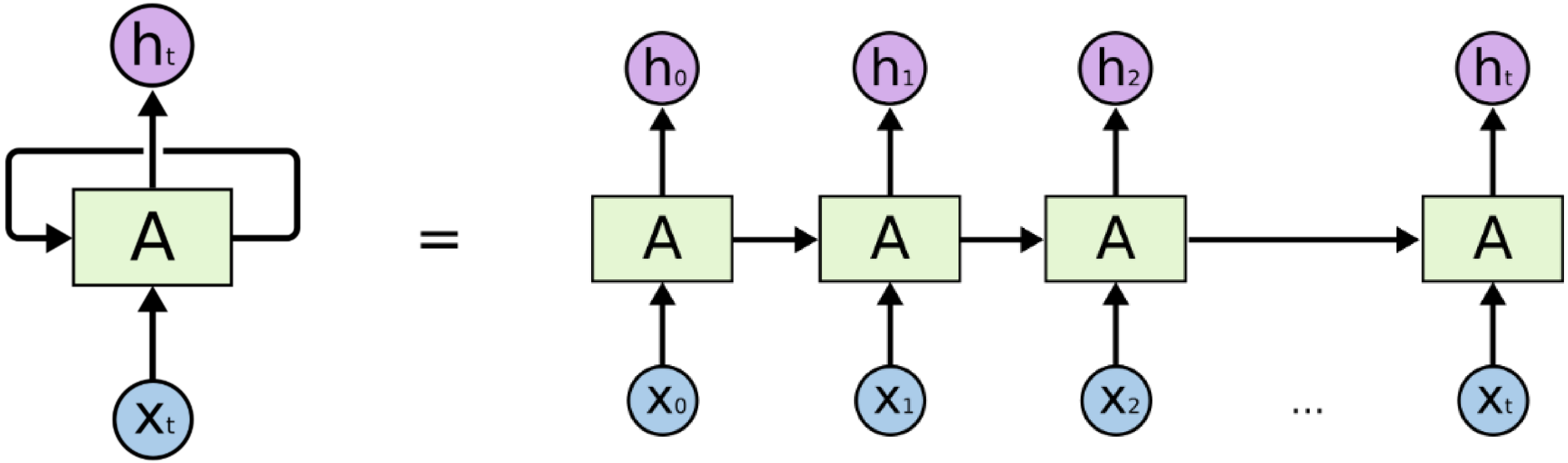
- **RNNs:** Designed to process sequences, capturing dependencies across time without fixed context windows.

- **Applications:** Language modeling, sequence labeling, text classification.

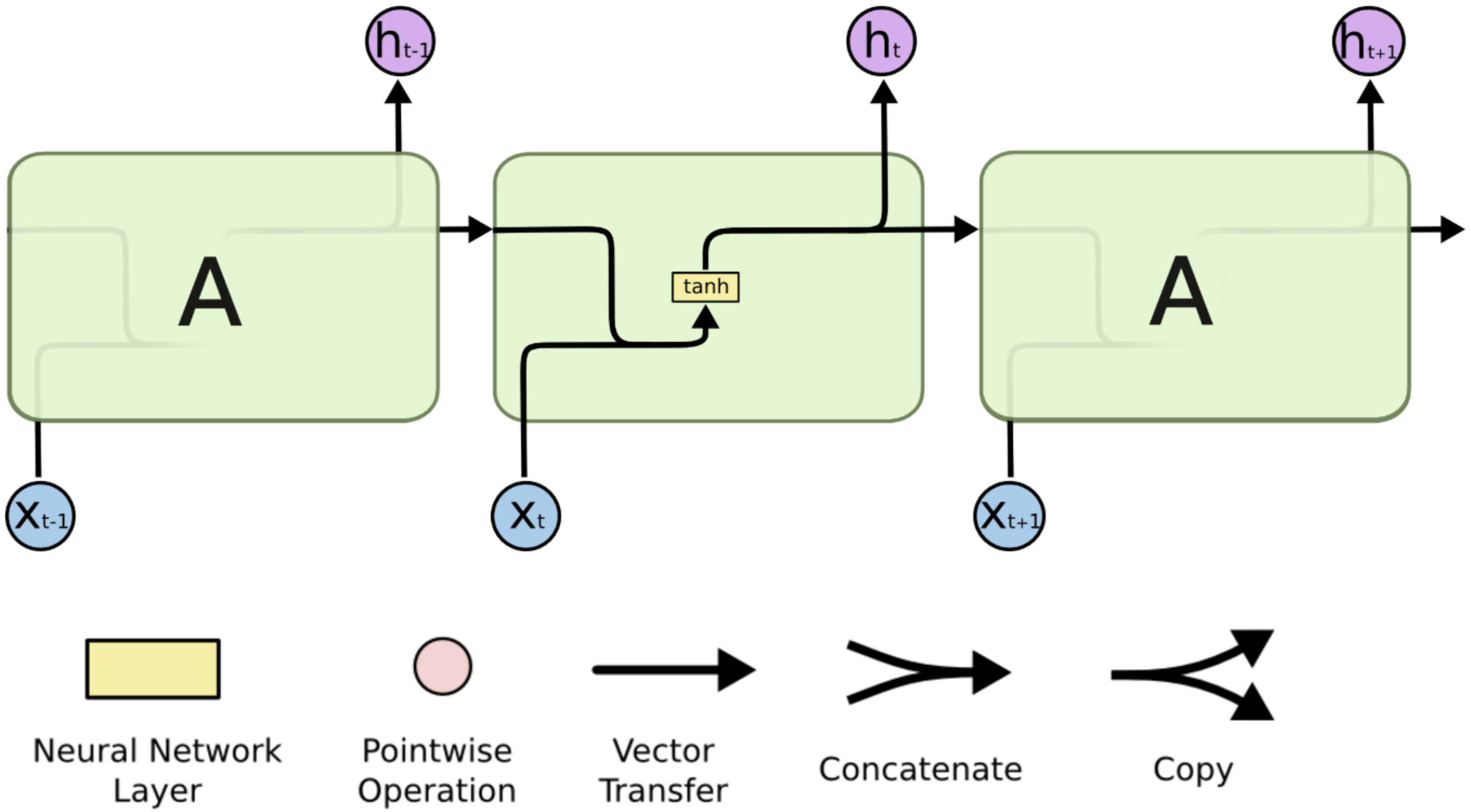
Introduction to RNNs

- Core Concept: Recurrent connections enable "memory" of previous states.
- Unrolling: Process input sequence step-by-step; weights shared across time.

Recurrent Neural Network

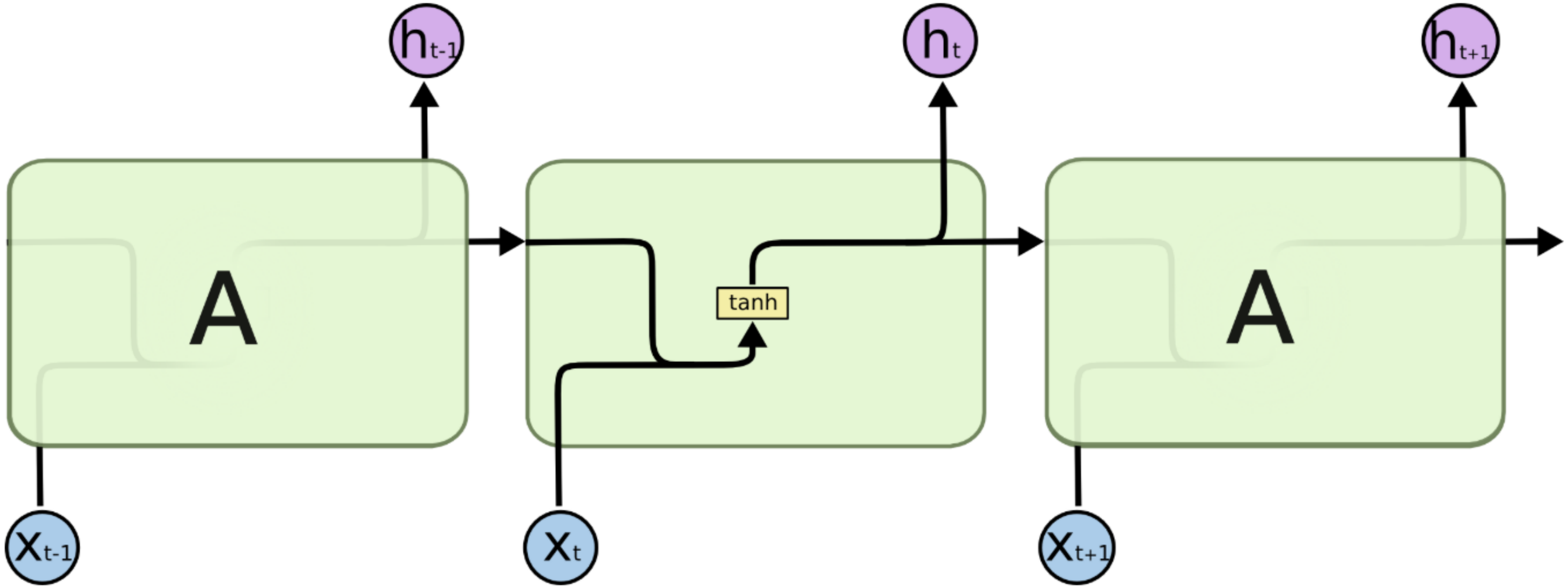


Simple RNN



h t-1
h t
h t+1
A
tanh
A
X t-1
X t
X t+1
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

Simple RNN



$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$$

Training RNNs

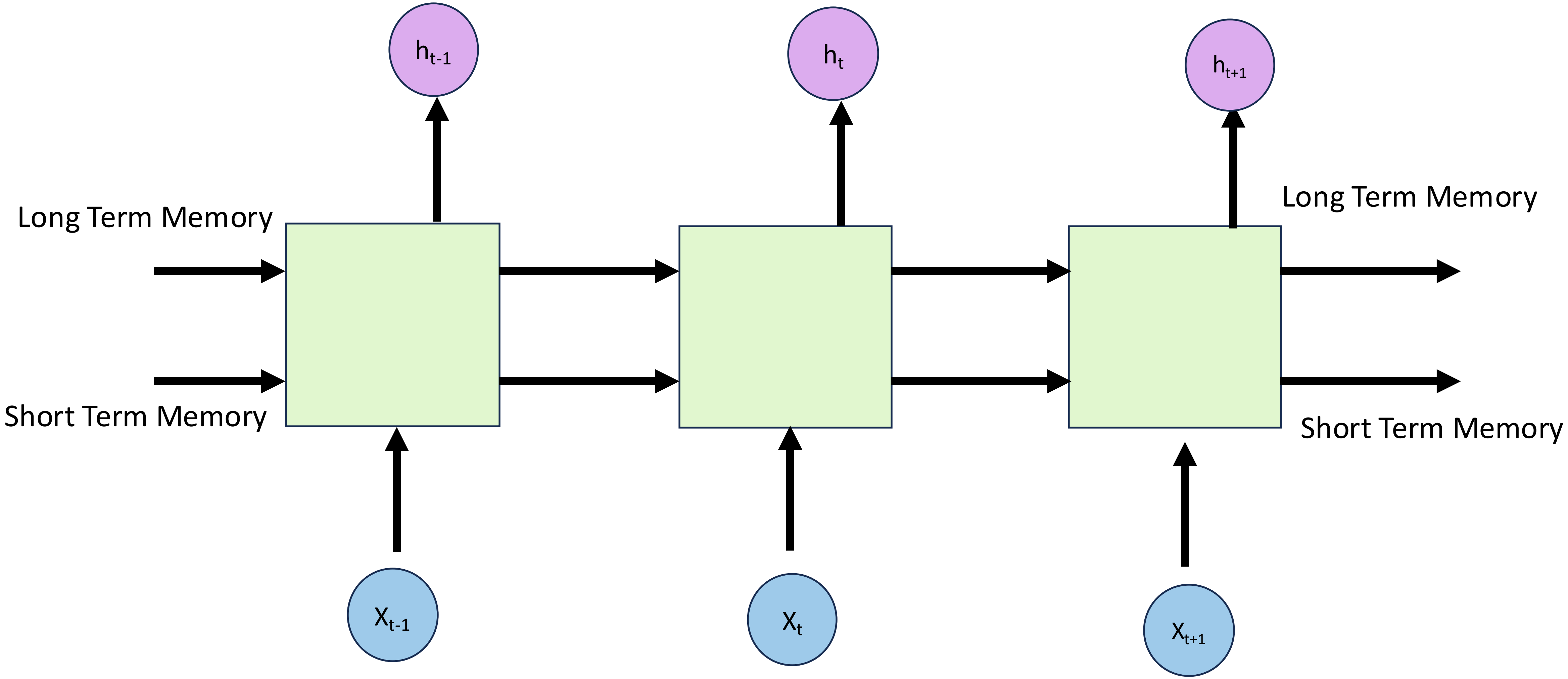
- Backpropagation Through Time (BPTT): Two-phase training:
  - Forward pass: Compute activations and loss.
  - Backward pass: Compute gradients through the unrolled network.

Problems with RNN

- **Vanishing Gradient Problem**
  - gradients can become very small (vanish) as they are propagated backward through time

- **RNNs struggle to maintain information over long sequences:**
  - They are inherently biased toward recent inputs

LSTM (Long Short Term Memory)



LSTM (Long Short Term Memory)

- LSTMs include a "memory cell" that can maintain information over long periods.

- By using memory cells and gates, LSTMs allow gradients to flow through time *more smoothly*, avoiding both vanishing and exploding gradient problems.

LSTM (Long Short Term Memory)

- LSTMs fix this by introducing:
  - A **cell state** c_t that can carry information *almost* unchanged over many steps.
  - **Gates** that control what to keep, what to forget, and what to output.

an Example

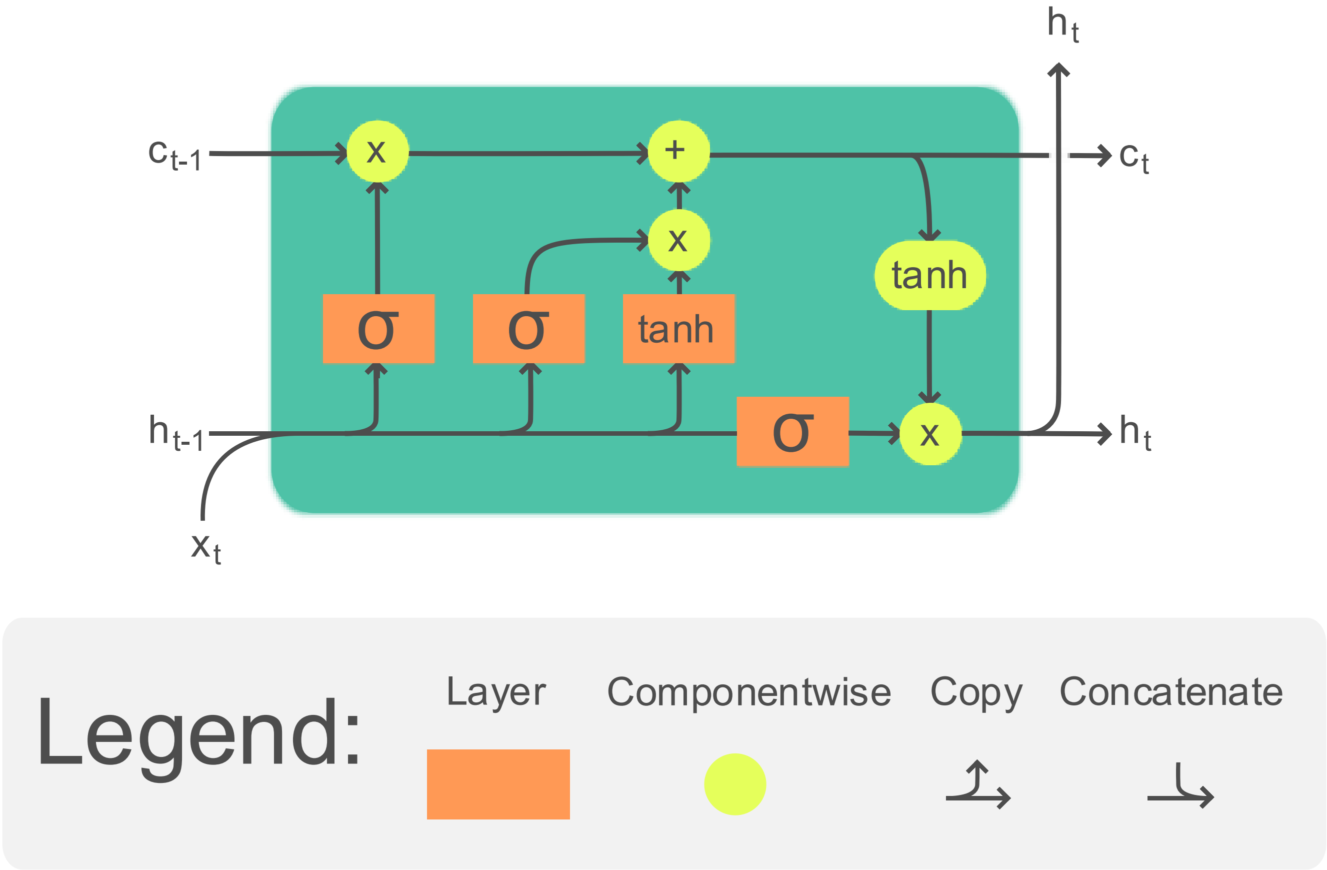
- ”In **2010**, she moved to Paris. She lived there for **five years**.”

Imagine network is at this timestep

- And we want to predict:

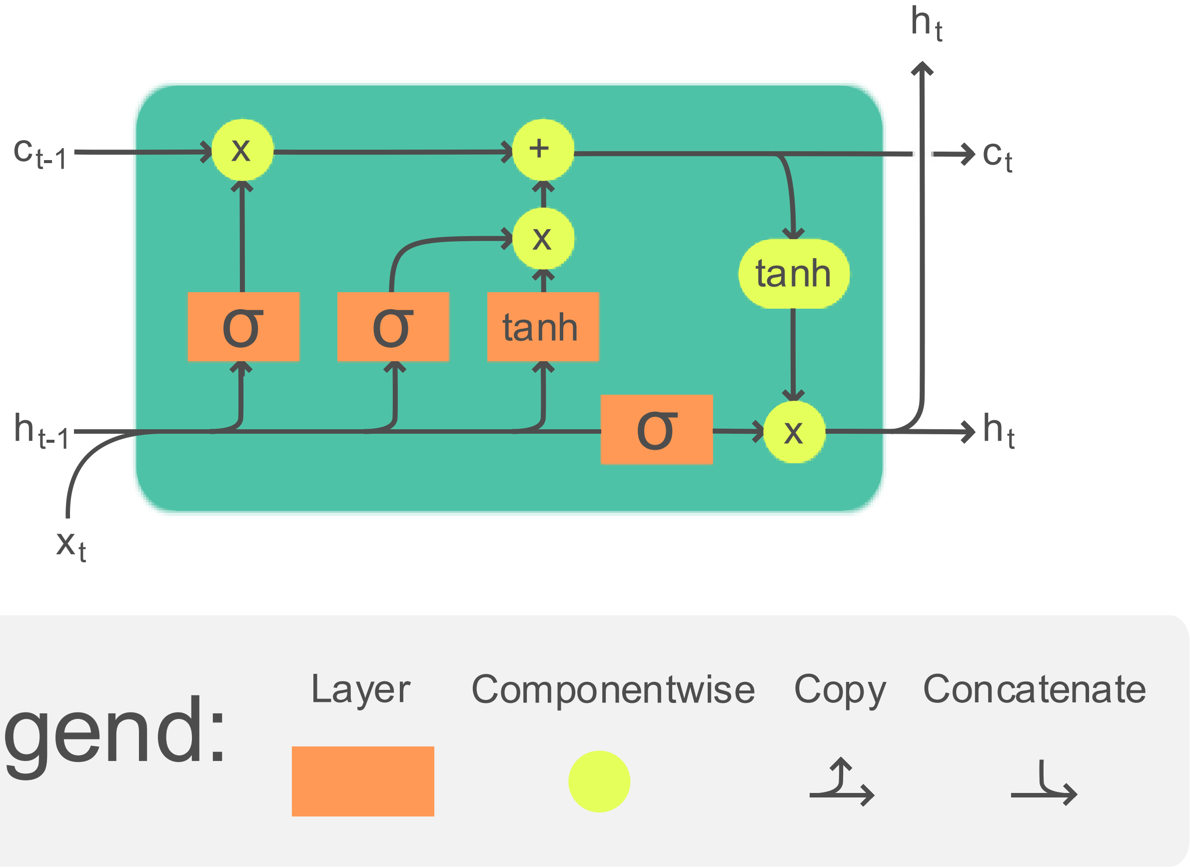
  in **2015** she returned

LSTM architecture



LSTM architecture

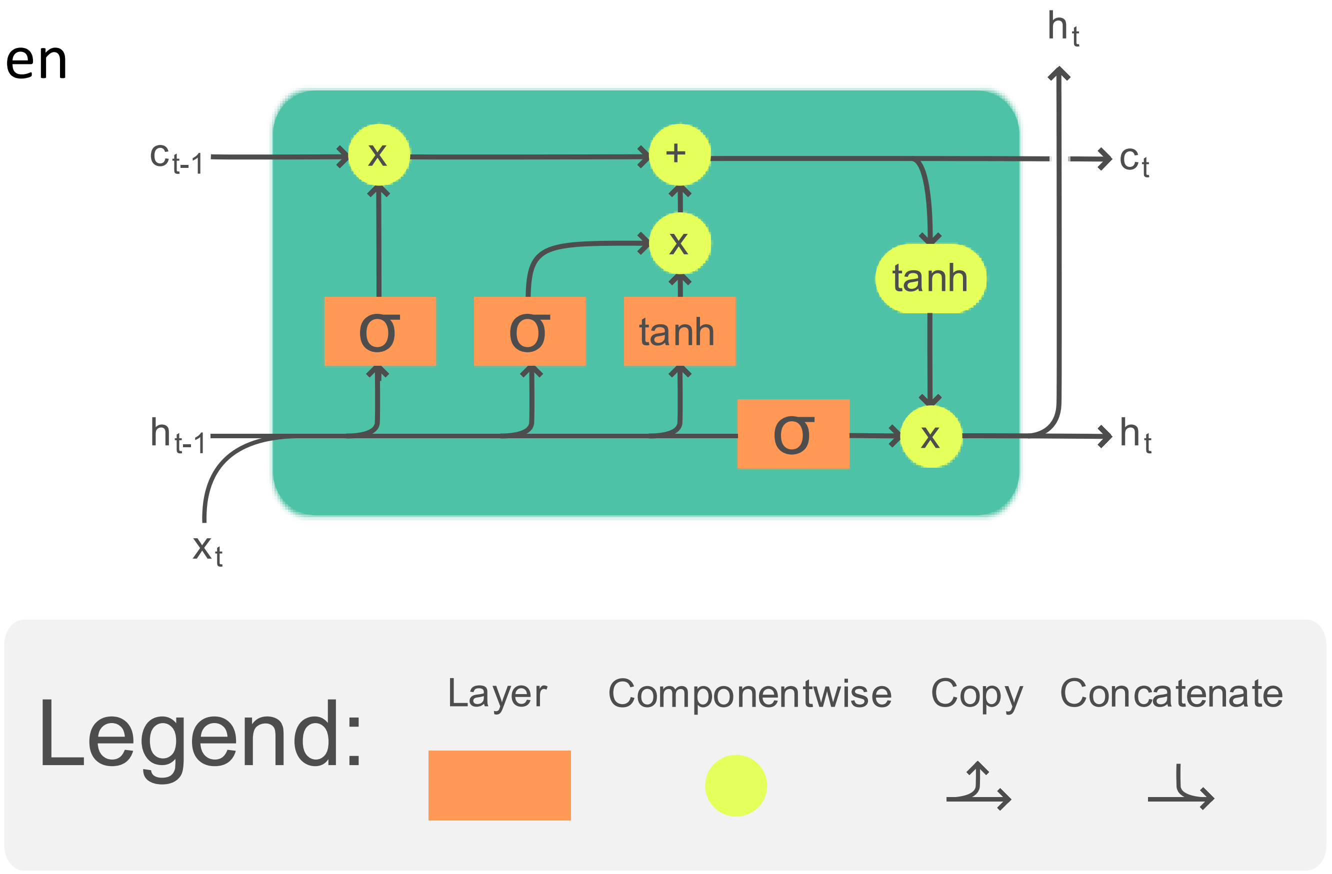
- **Forget gate**: what part of old memory to erase.
- **Input gate**: what new information to write.
- **Output gate**: what part of the memory to expose as the hidden state.



LSTM architecture

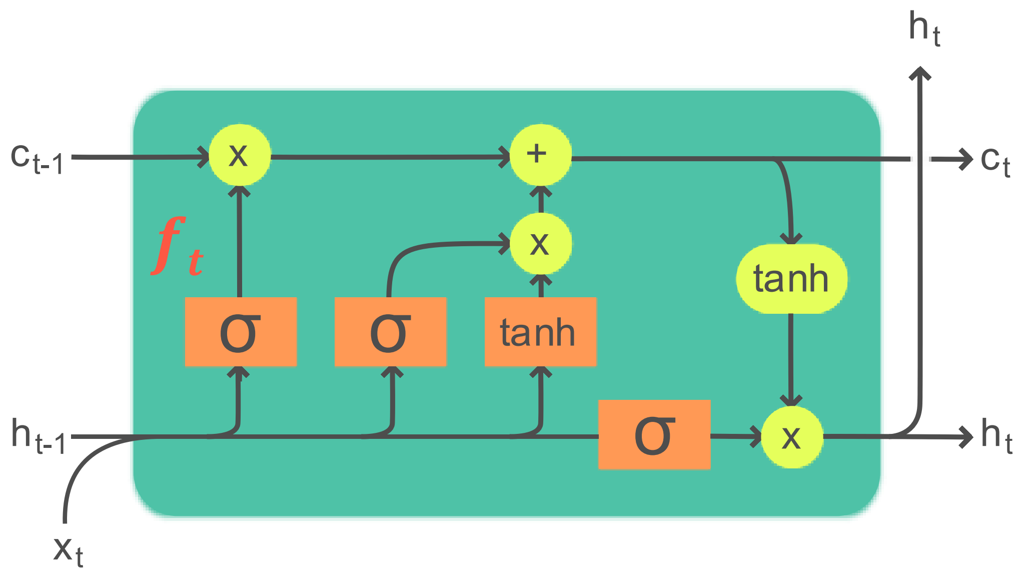
- **Concatenate** input and previous hidden state:

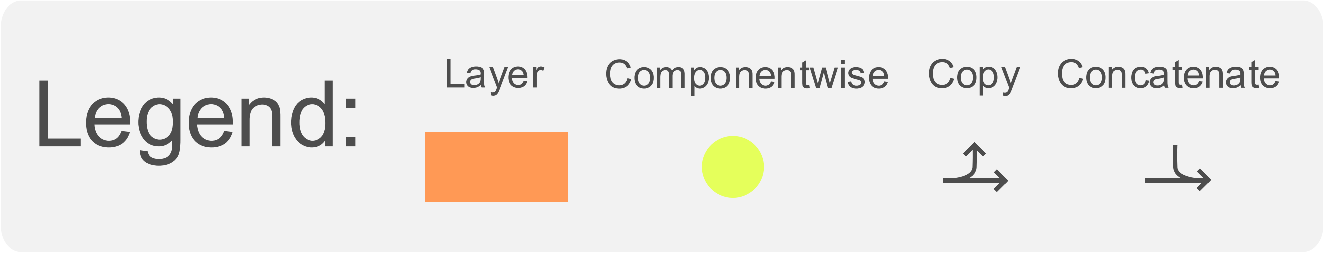
$$z_t = [h_{t\,-\,1}; x_t]$$



LSTM architecture

- **Concatenate** input and previous hidden state:
$$z_t = [h_{t-1}; x_t]$$
- **Gates** (each element is in $(0, 1)$ via sigmoid σ):
  - Forget gate:
$$f_t = \sigma(W_f z_t + bf)$$





LSTM architecture

- **Concatenate** input and previous hidden state:

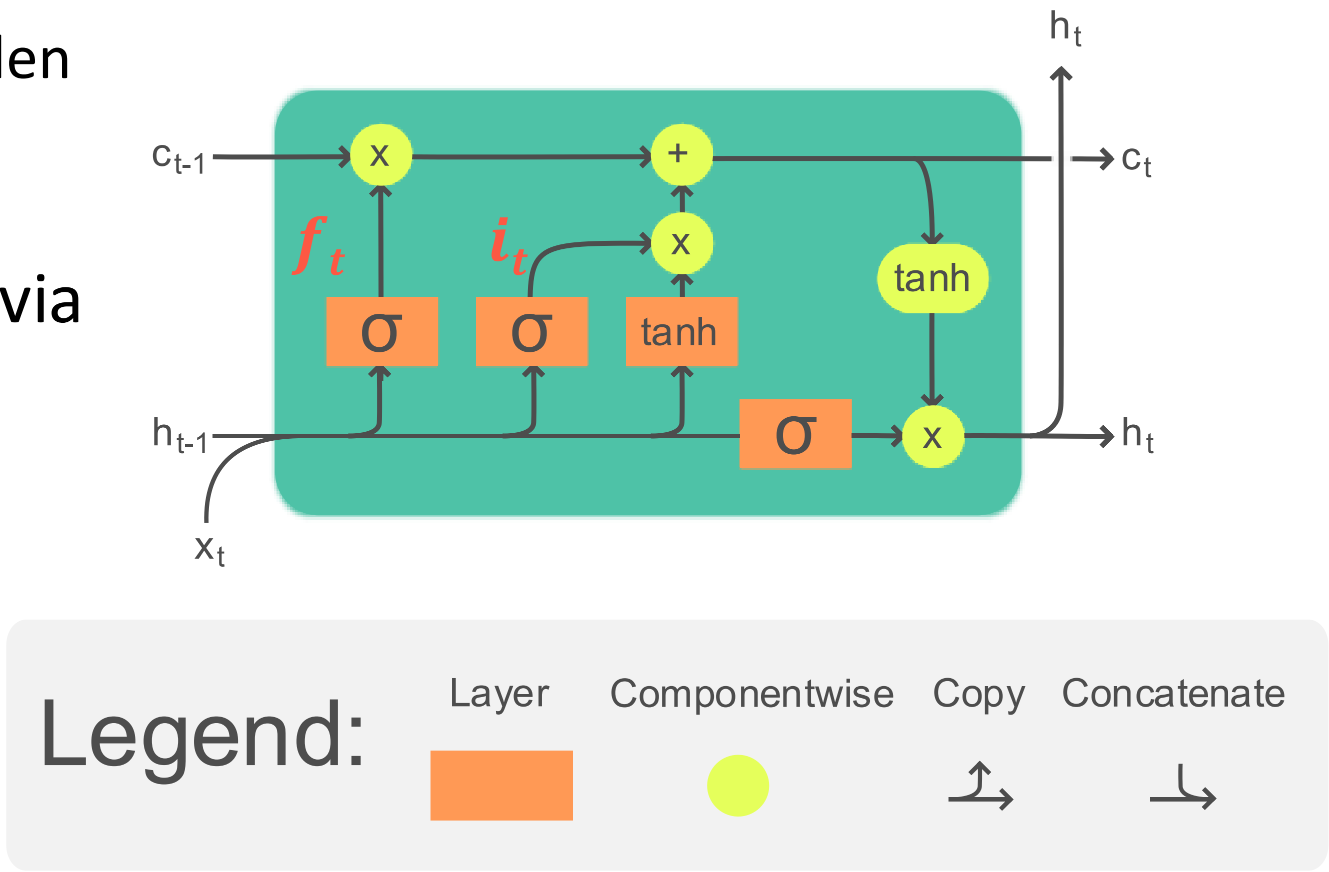
$$z_t = [h_{t-1}; x_t]$$

- **Gates** (each element is in $(0, 1)$ via sigmoid σ):
  - Forget gate:

$$f_t = \sigma(W_f z_t + bf)$$

  - Input gate:

$$i_t = \sigma(W_i z_t + bi)$$



LSTM architecture

- **Concatenate** input and previous hidden state:

$$z_t = [h_{t\,\_\,1}; x_t]$$

- **Gates** (each element is in $(0, 1)$via sigmoid σ):
  - Forget gate:

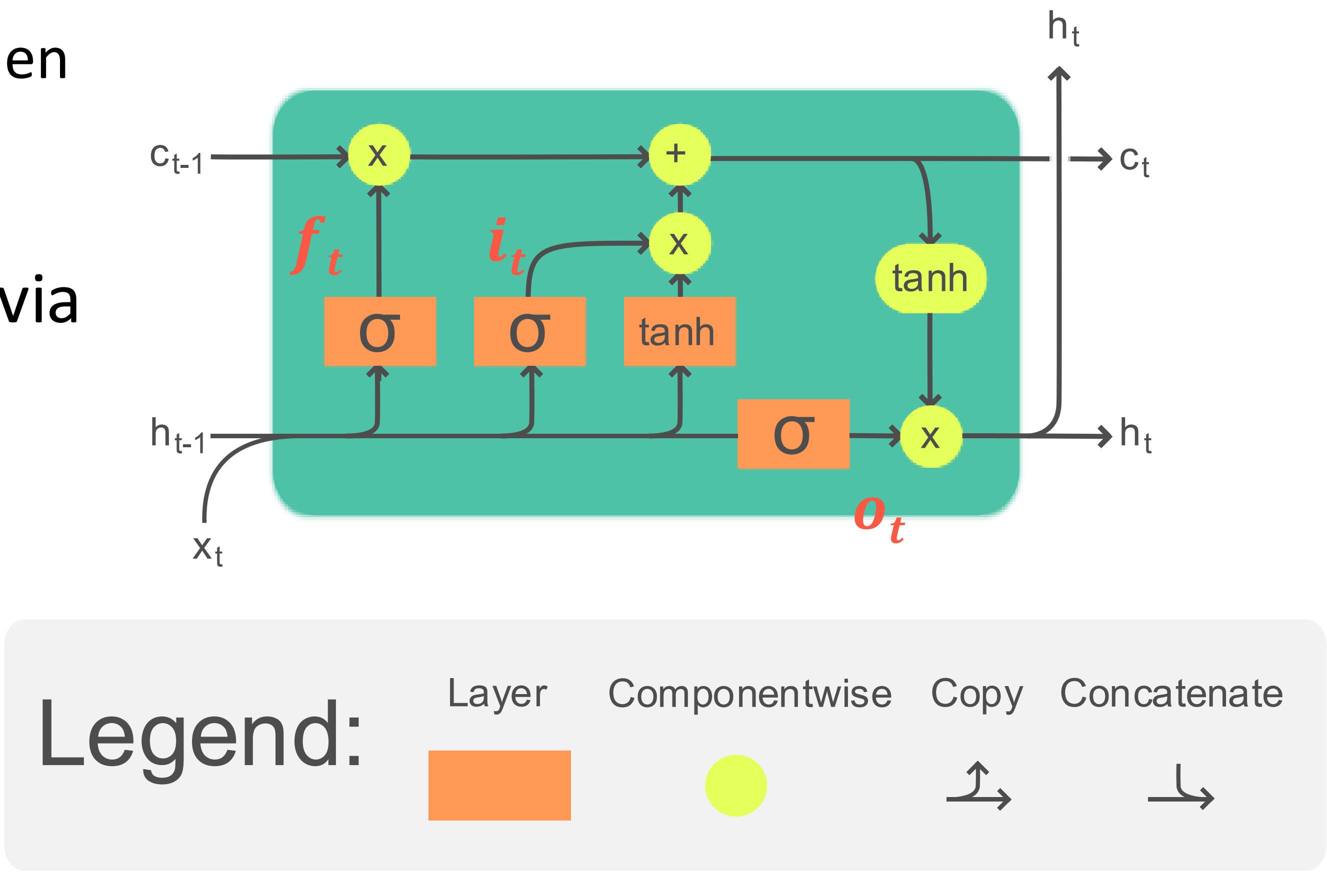
$$f_t = \sigma(W_f z_t + bf)$$

  - Input gate:

$$i_t = \sigma(W_i z_t + bi)$$

  - Output gate:

$$o_t = \sigma(W_o z_t + b_o)$$

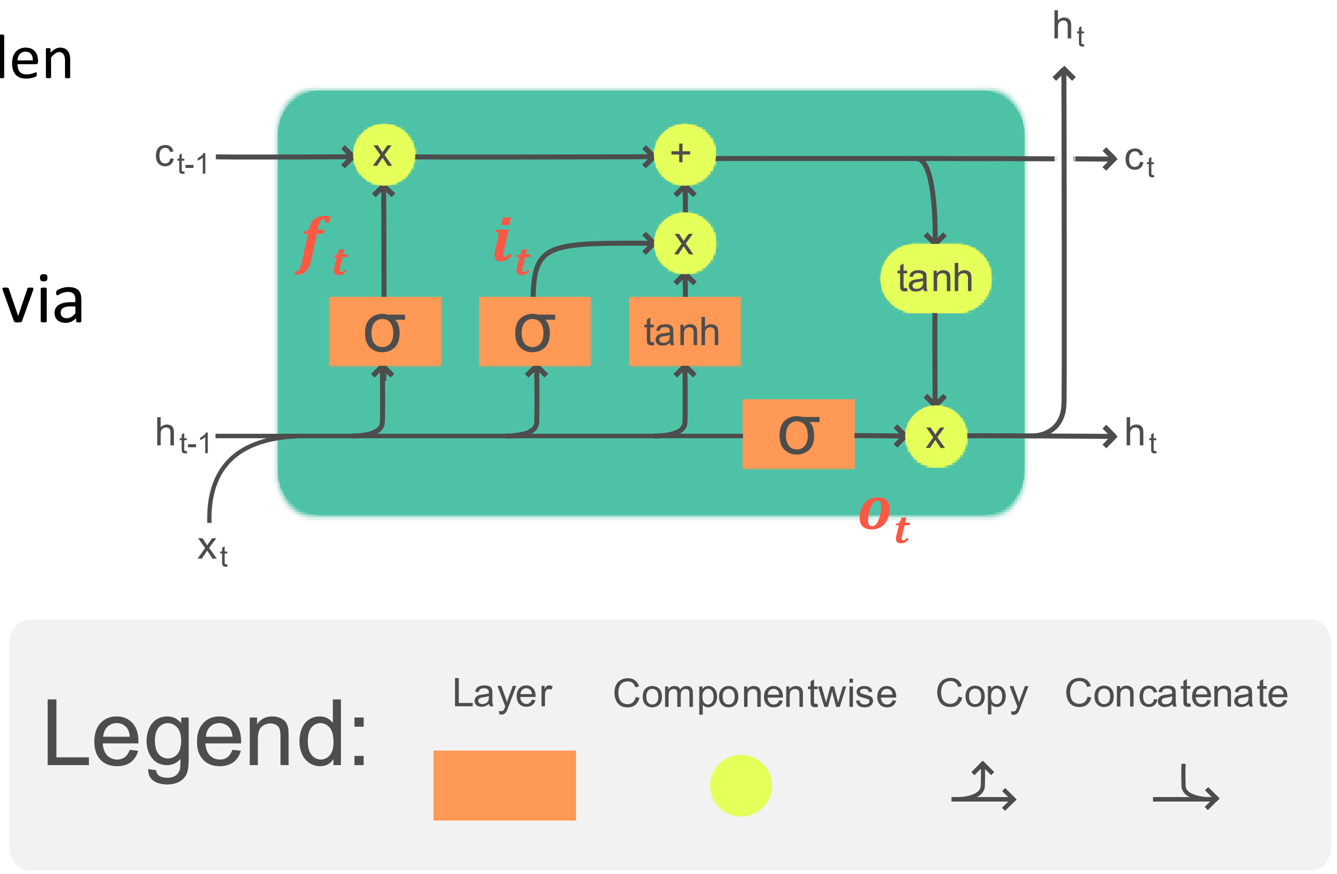


LSTM architecture

- **Concatenate** input and previous hidden state:

$$z_t = [h_{t-1}; x_t]$$

- **Gates** (each element is in $(0, 1)$ via sigmoid σ):
  - Forget gate:
    $$f_t = \sigma(W_f z_t + bf)$$
  - Input gate:
    $$i_t = \sigma(W_i z_t + bi)$$
  - Output gate:
    $$o_t = \sigma(W_o z_t + b_o)$$

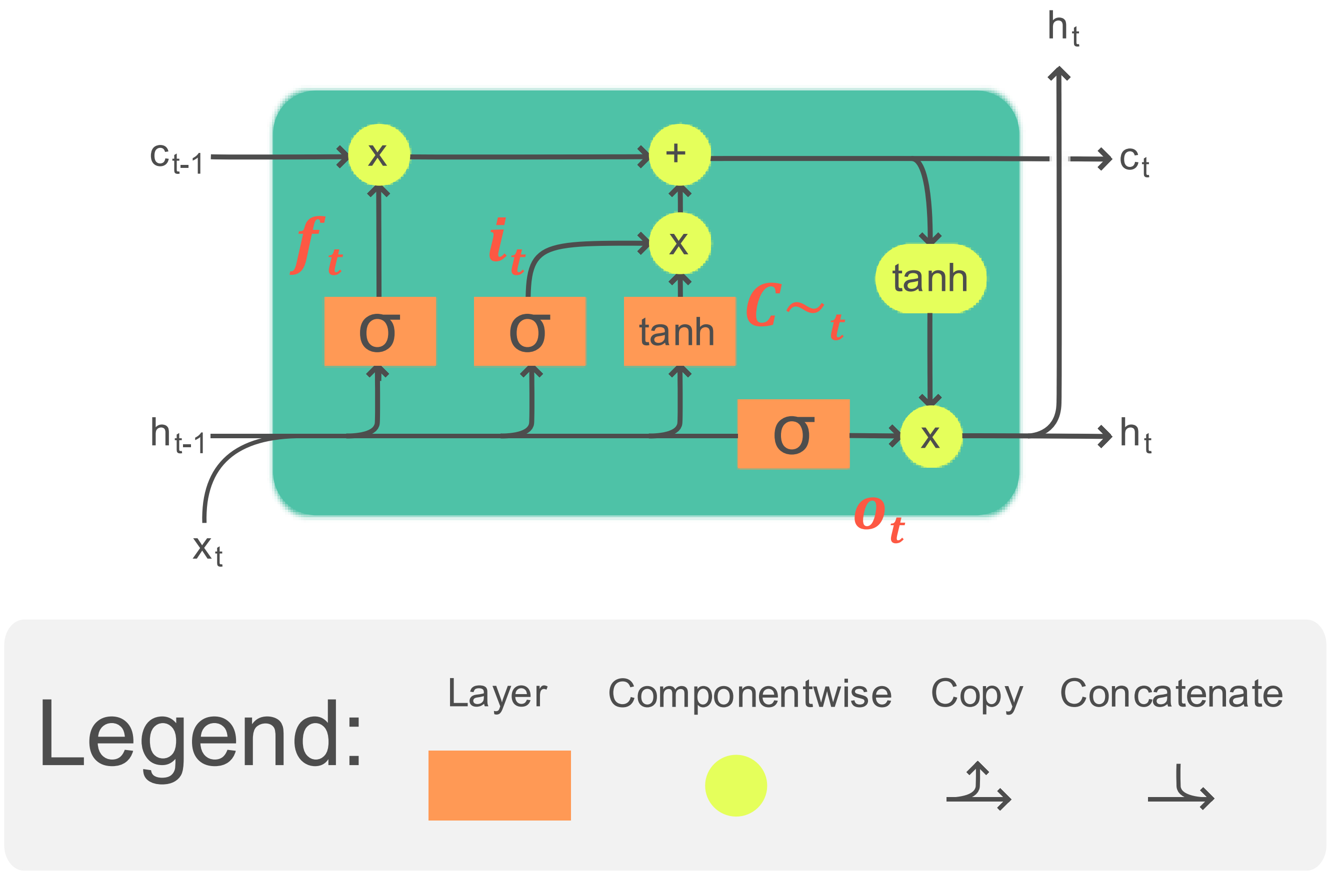


LSTM architecture

- **Candidate content** to add to memory

tanh squashes to $(-1, 1)$:

  - $c\sim_t = \tanh(W_c z_t + b_c)$



LSTM architecture

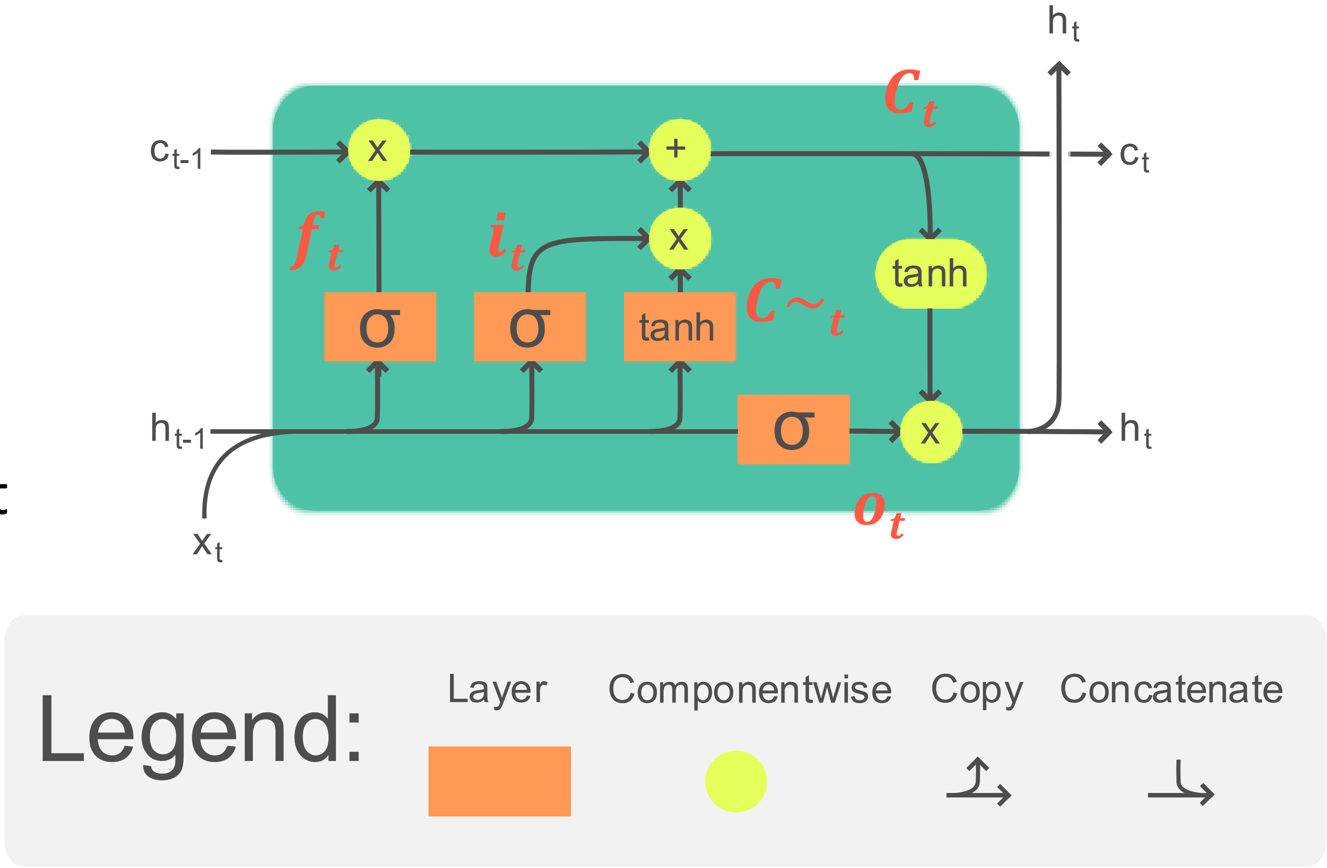
- **Candidate content** to add to memory

tanh squashes to $(-1, 1)$:

- $c\sim_t = \tanh(W_c z_t + b_c)$

- **Update cell state**:
  - $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$

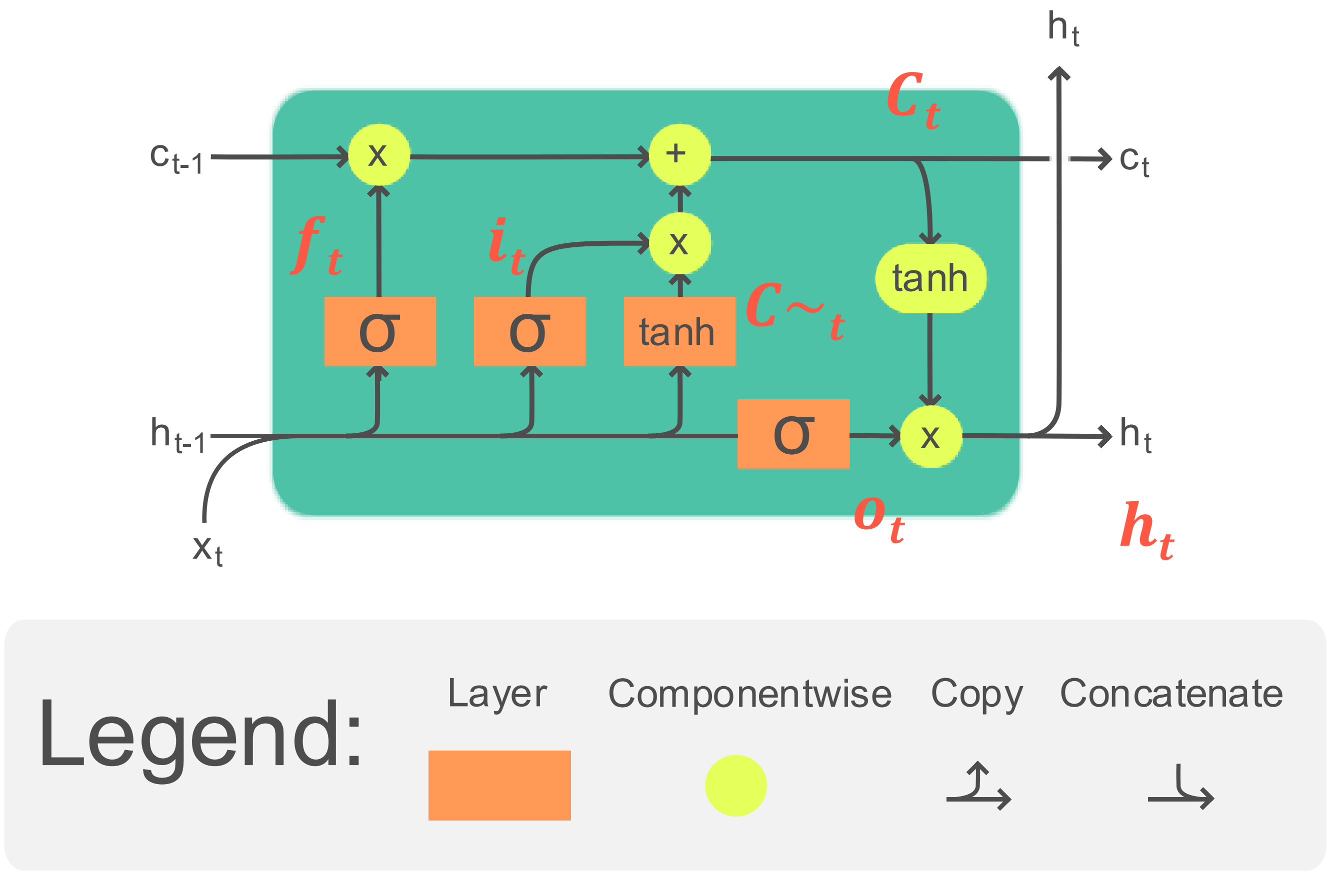
- If a component of f_t is close to 1, that part of the old memory is kept.
- If a component of i_t is close to 1, we write the new candidate into memory.



LSTM architecture

- **Update hidden state**:

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

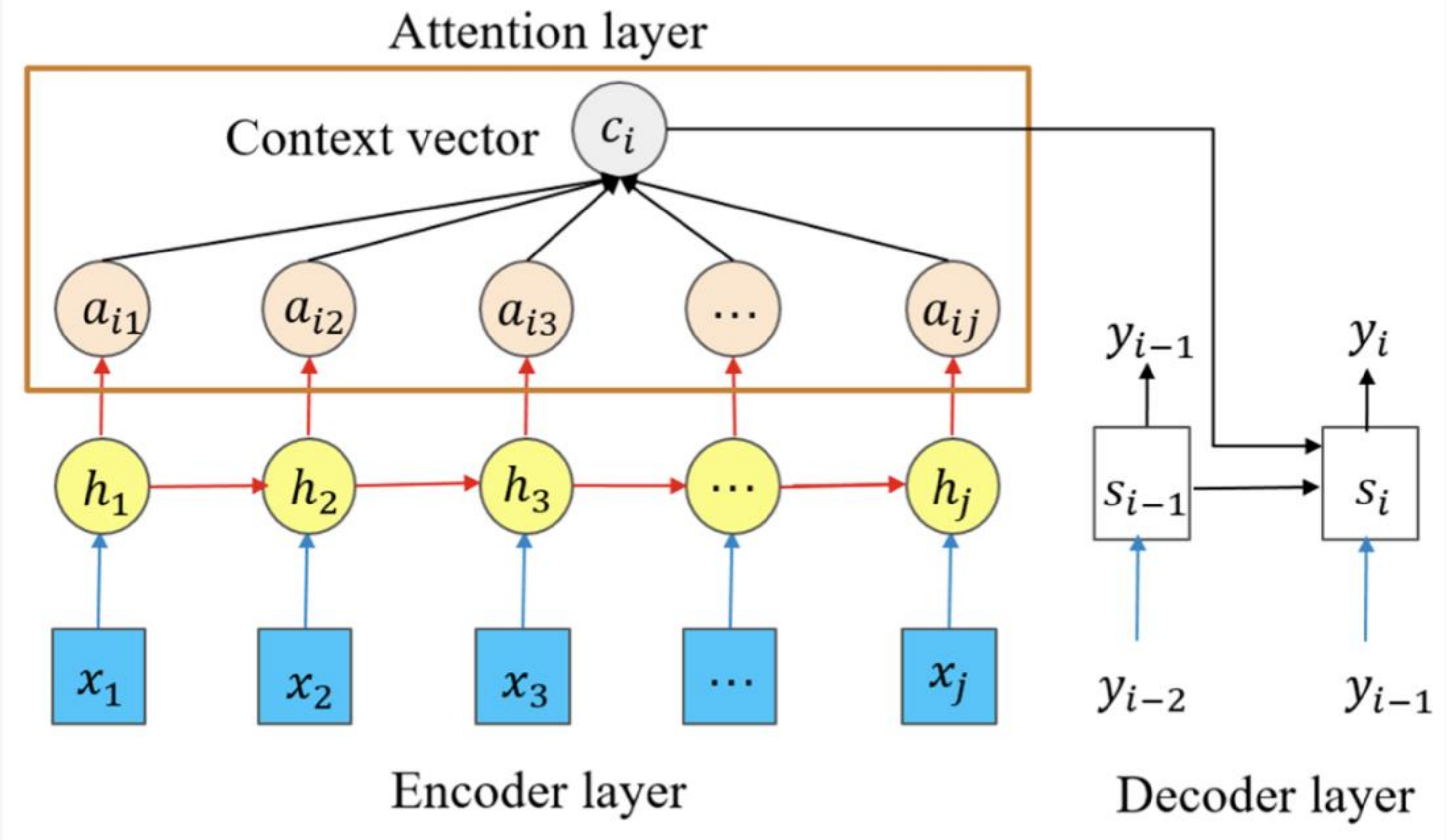


LSTM Variants

- Gated Recurrent Unit
  - SEE FOR YOURSELF!

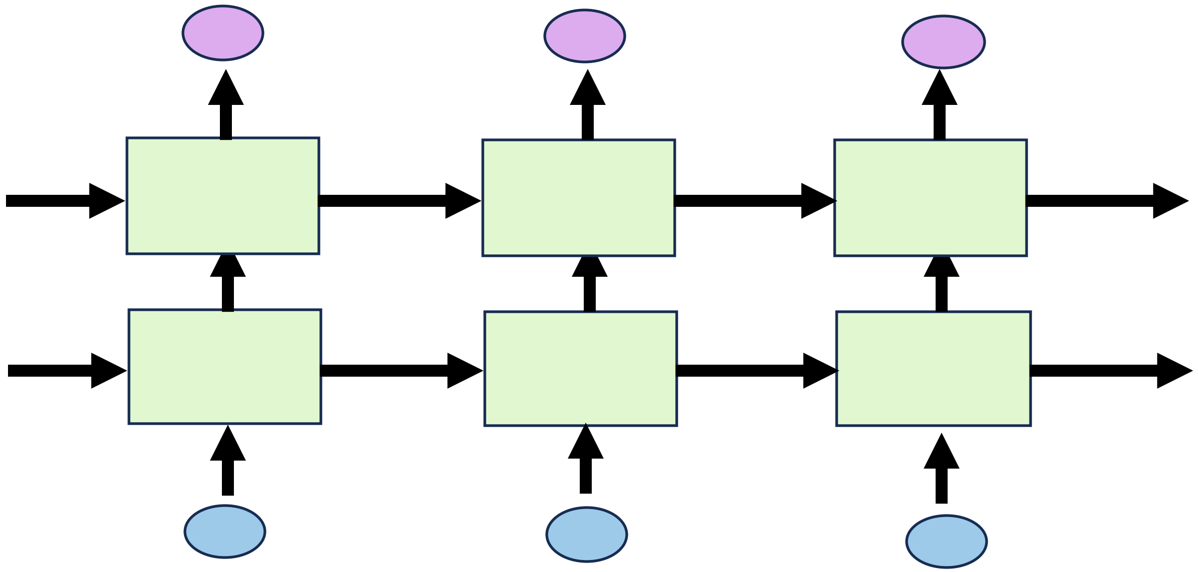
LSTM with attention

- LSTM attention mechanism
- Attention = extra mechanism that lets the decoder look back at all encoder hidden states instead of only a single fixed vector.



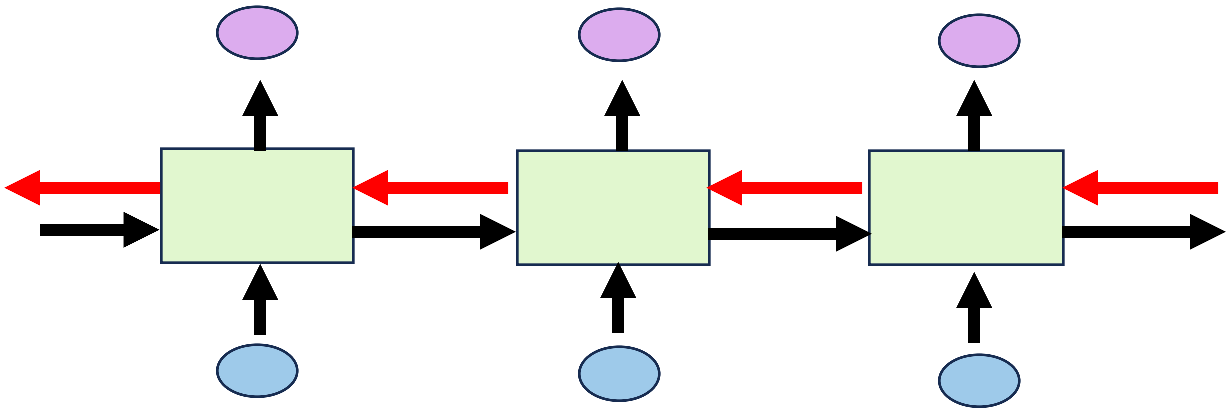
Variants of RNNs

- Stacked RNNs:
  - Multiple RNN layers; output of one feeds into the next.



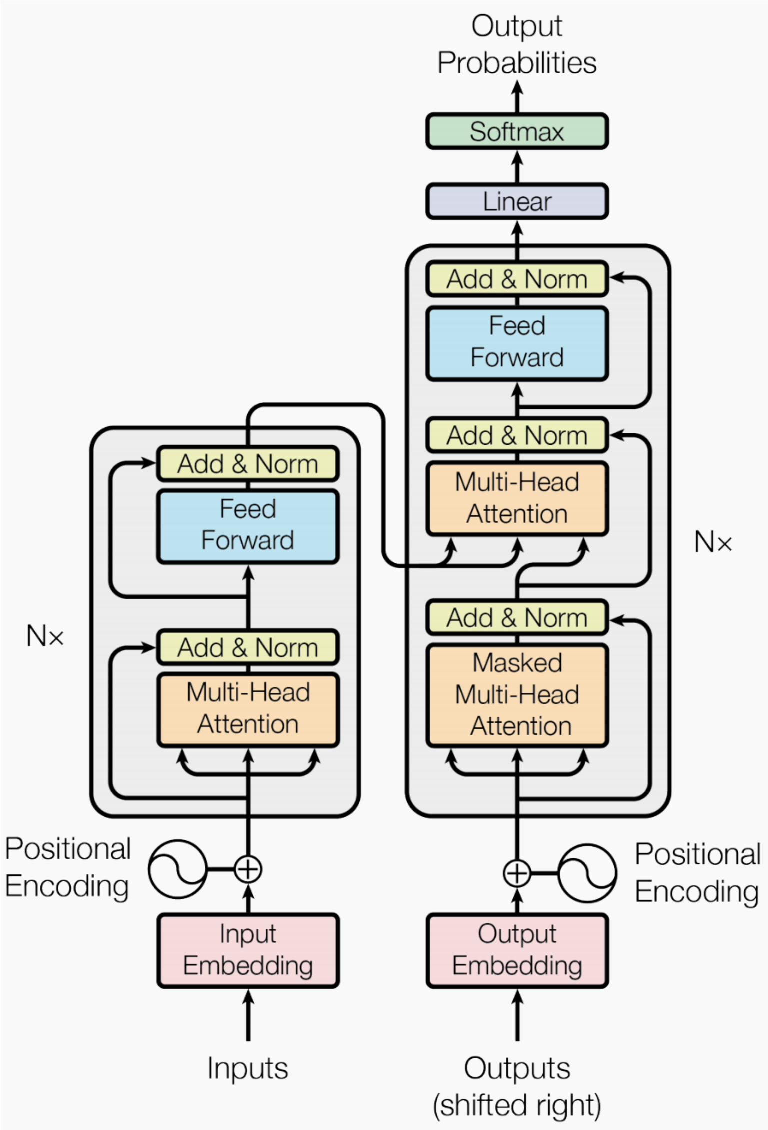
Variants of RNNs

- Bidirectional RNNs:
  - Combines forward and backward RNNs for richer context.
  - Applications: Sequence labelling and classification.



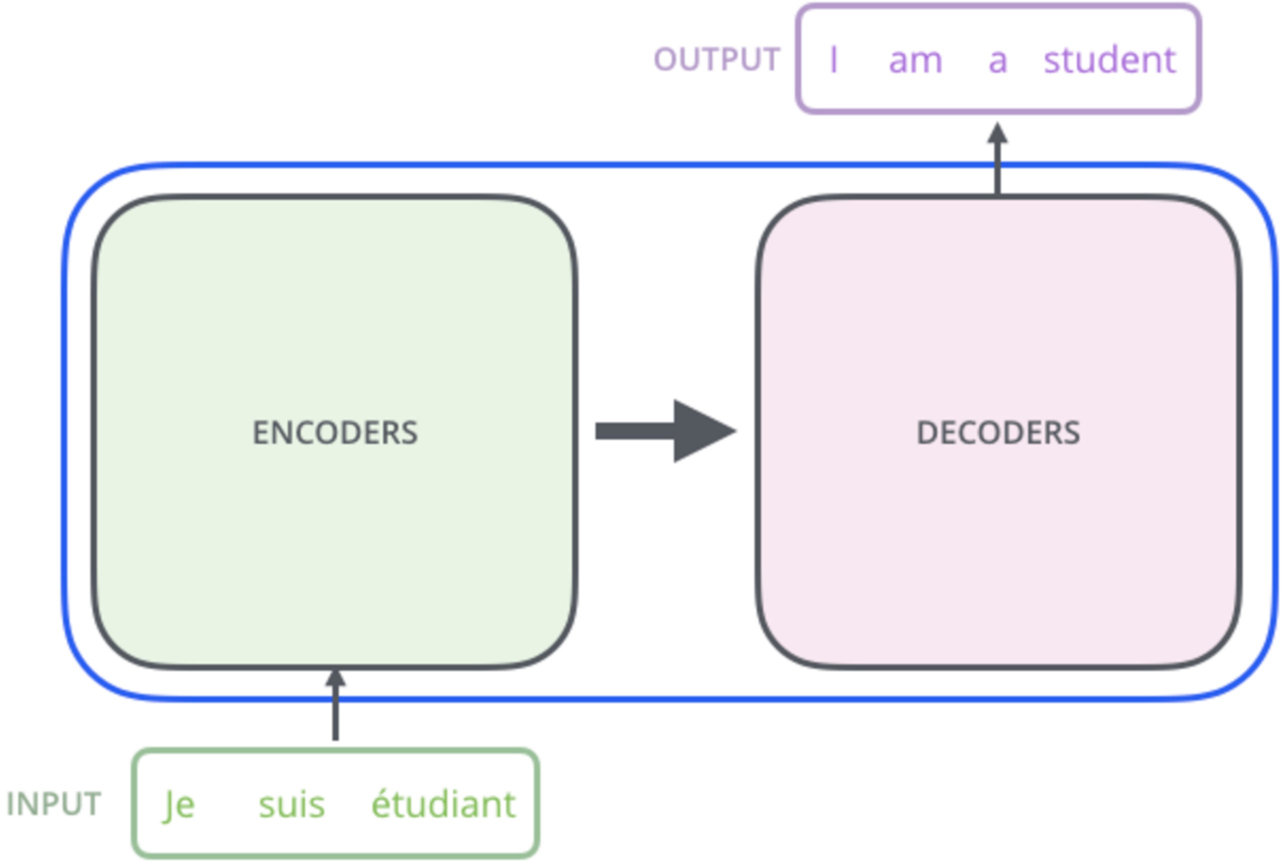
Transformers

Transformers



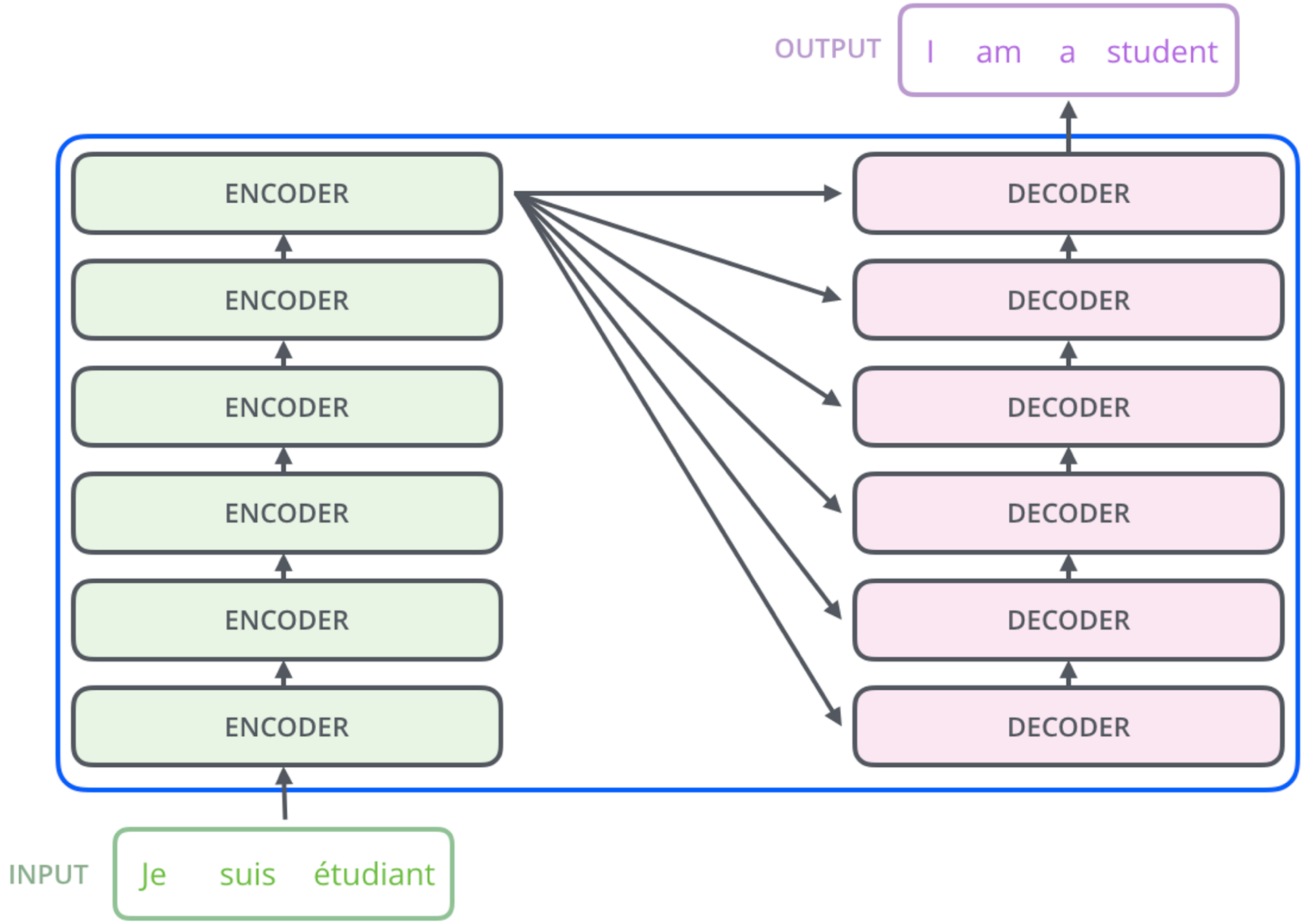
Introduction to Transformers

- Stacks of **encoders** and **decoders**



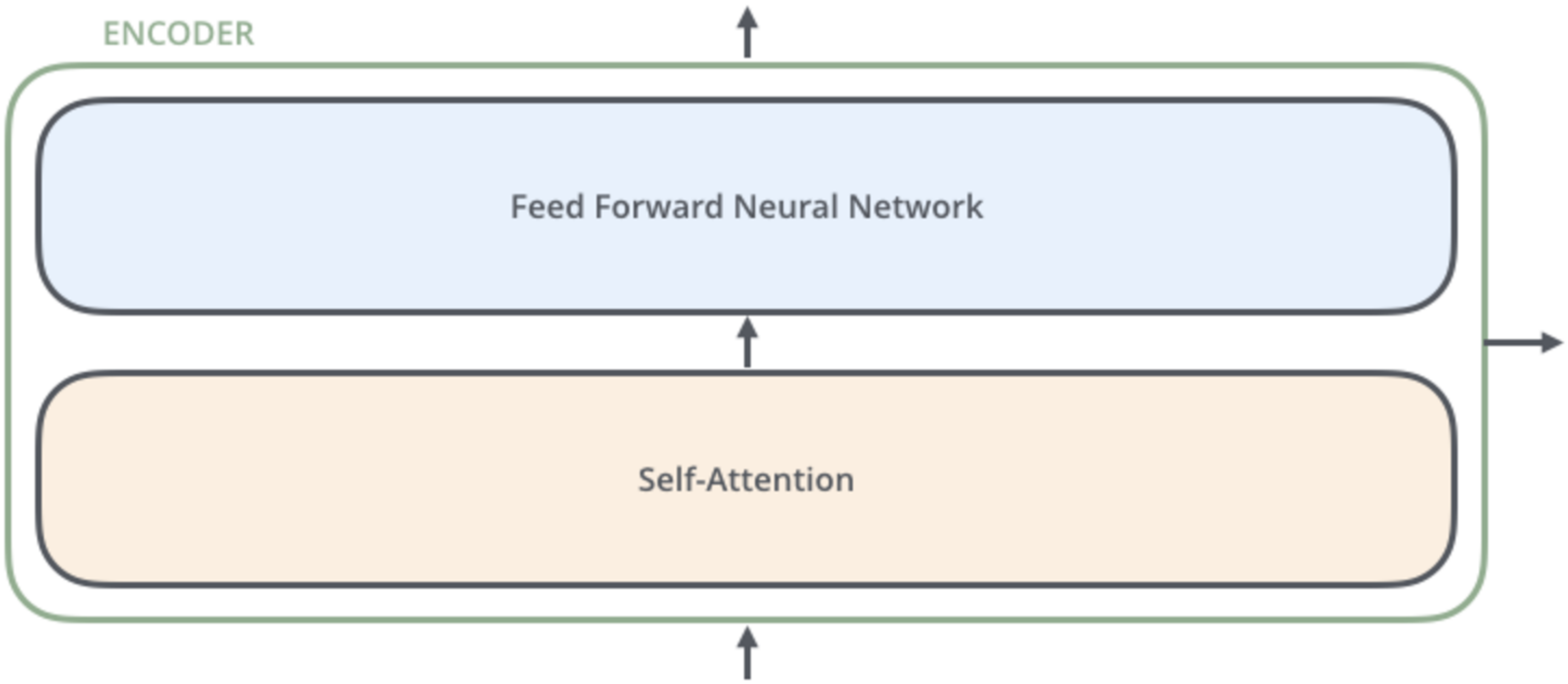
Introduction to Transformers

- Stacks of **encoders** and **decoders**



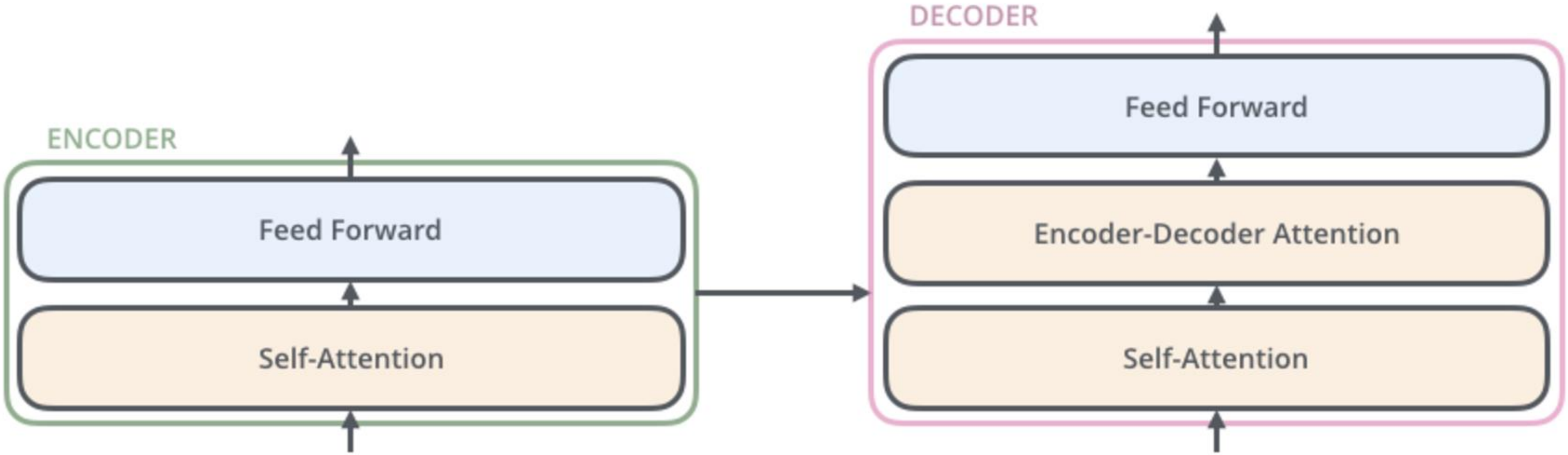
Encoder

- Stacks of **encoders**

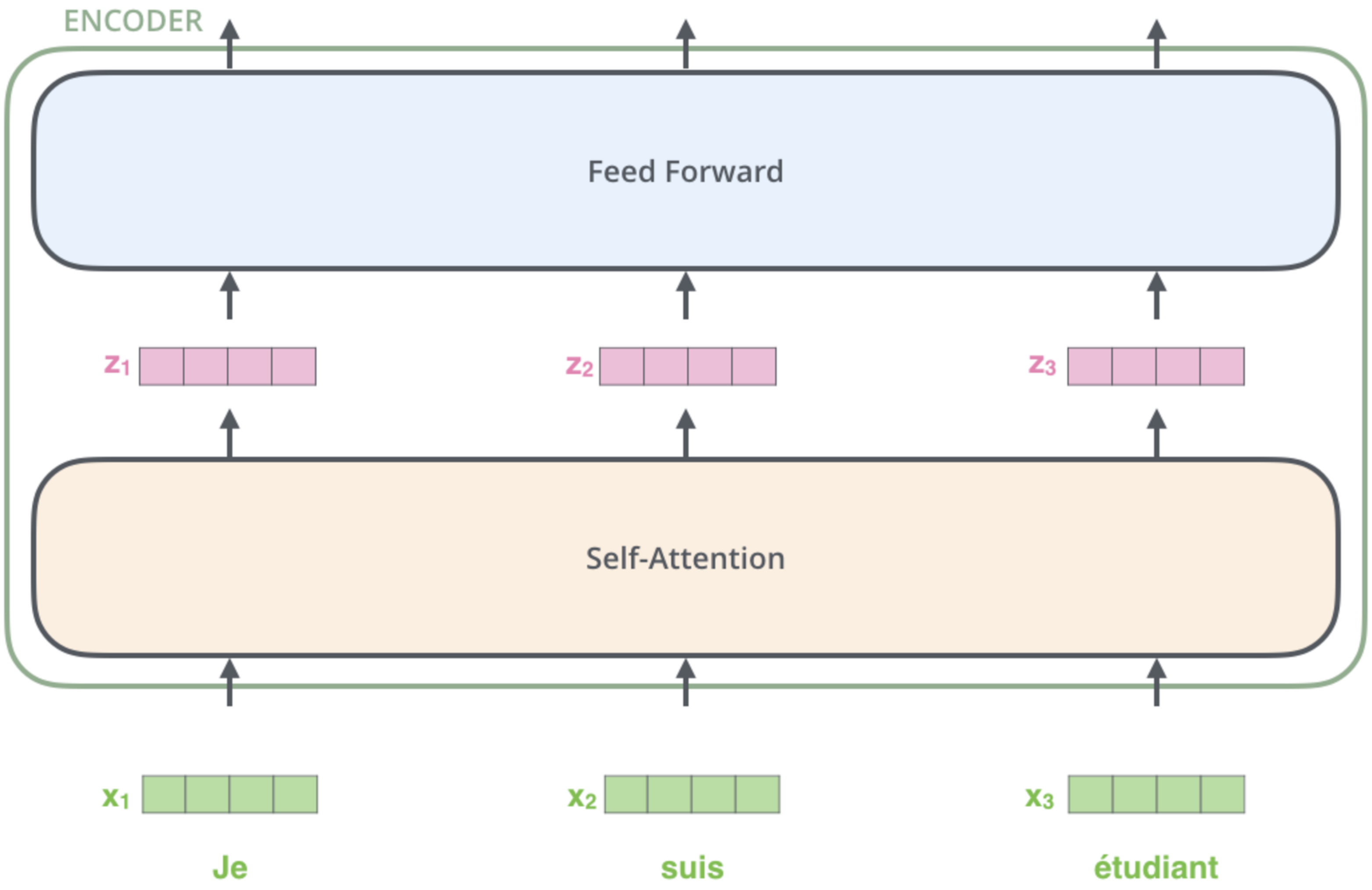


Encoder - Decoder

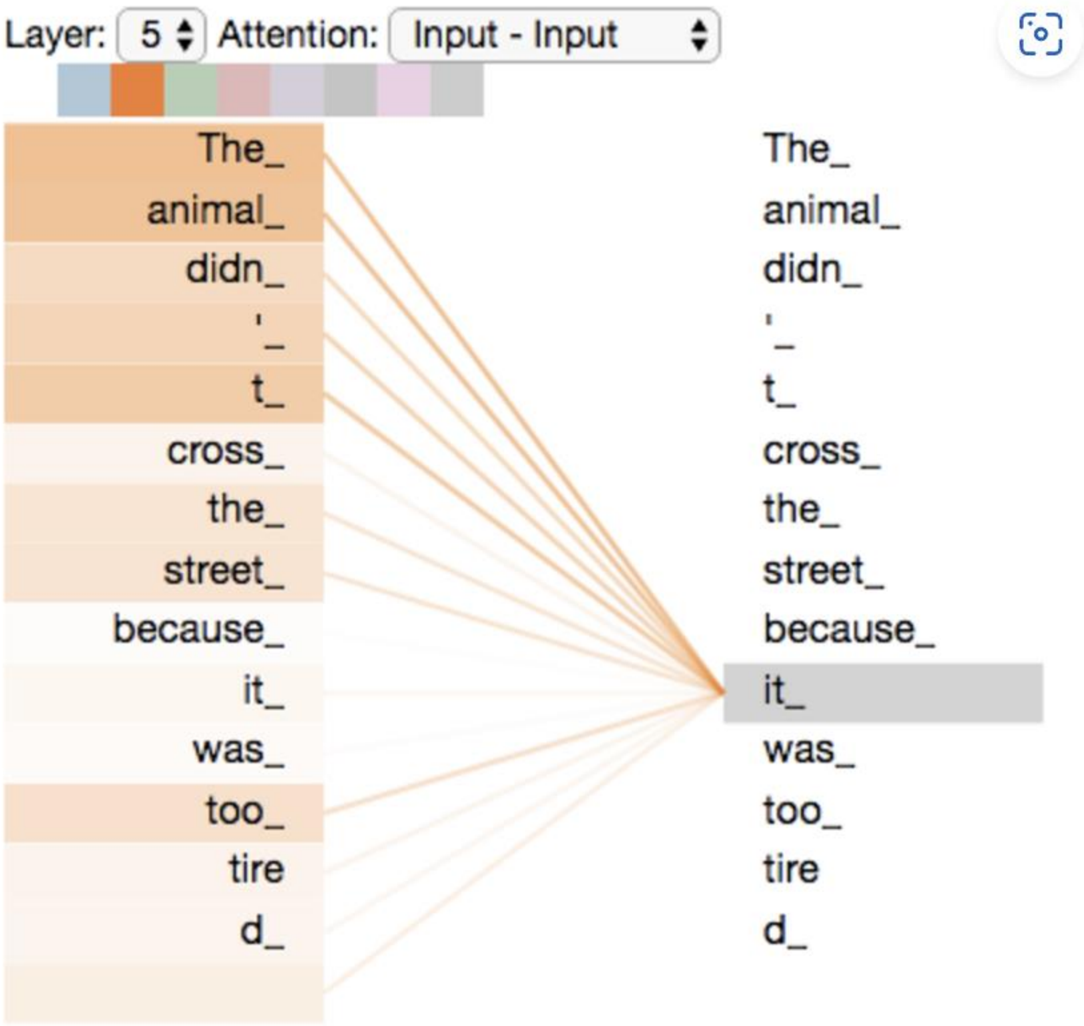
- Stacks of **encoders** and **decoders**



Attention mechanism



Self-Attention

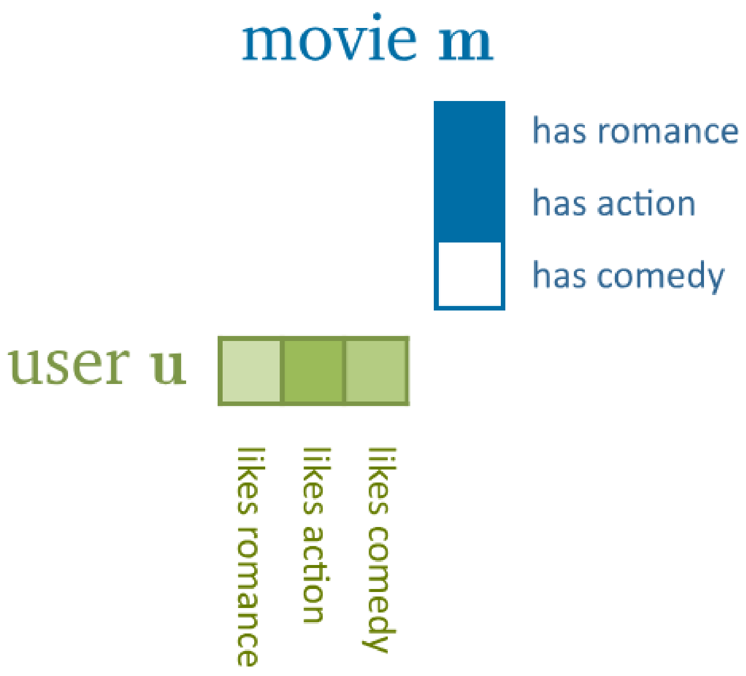


Key Mechanism: Attention

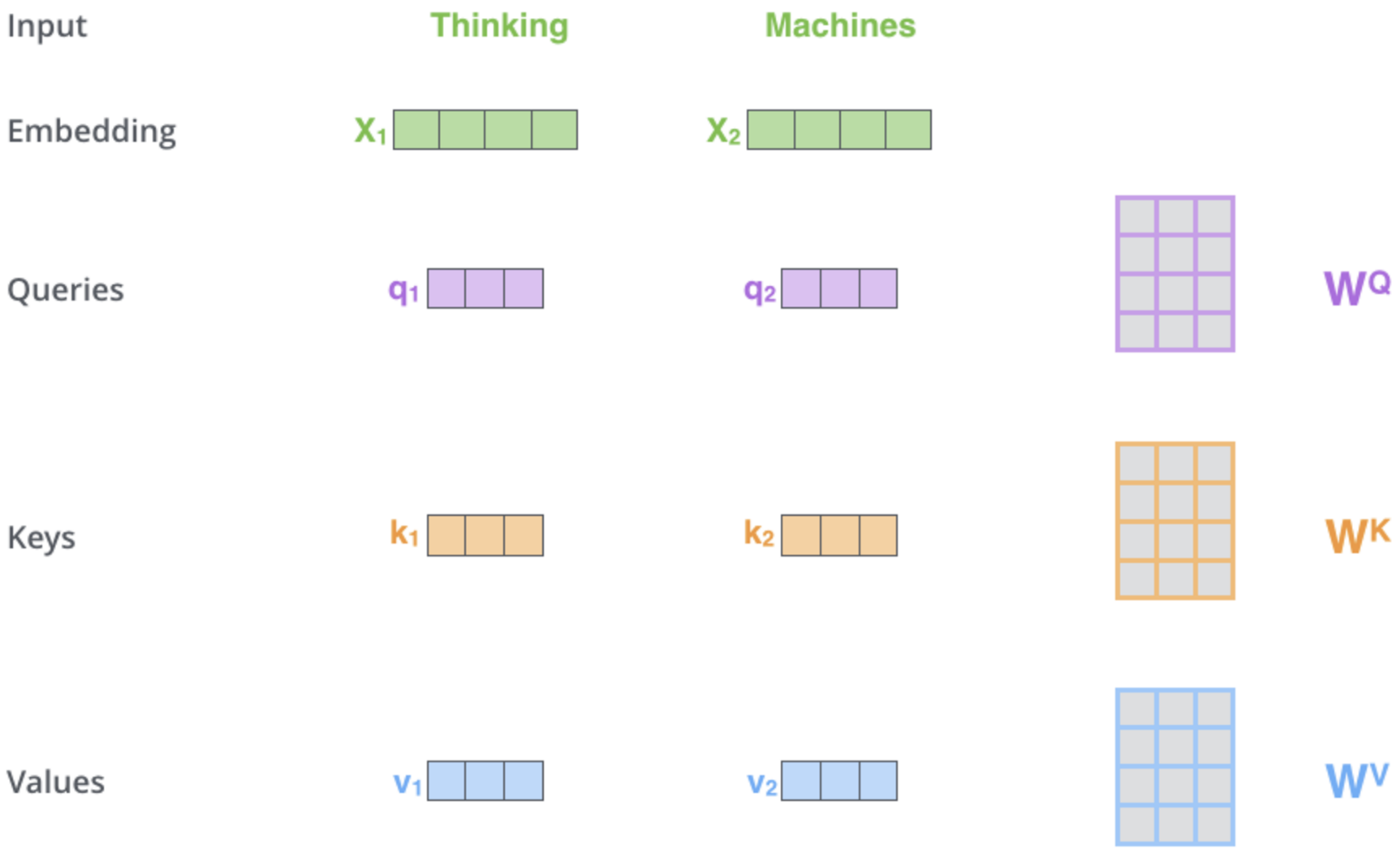
- **Self-Attention:** Computes contextualized token embeddings by attending to other tokens.

- **Multi-Head Attention:**
  - Multiple parallel attention heads for capturing diverse linguistic features

Dot product


movie m
has romance
has action
has comedy
user u
likes romance
likes action
likes comedy

Self-Attention



Self-attention

- Start with input token embeddings as a matrix $X \in \mathbb{R}^{T \times d_{model}}$:
- T :number of tokens in the sequence

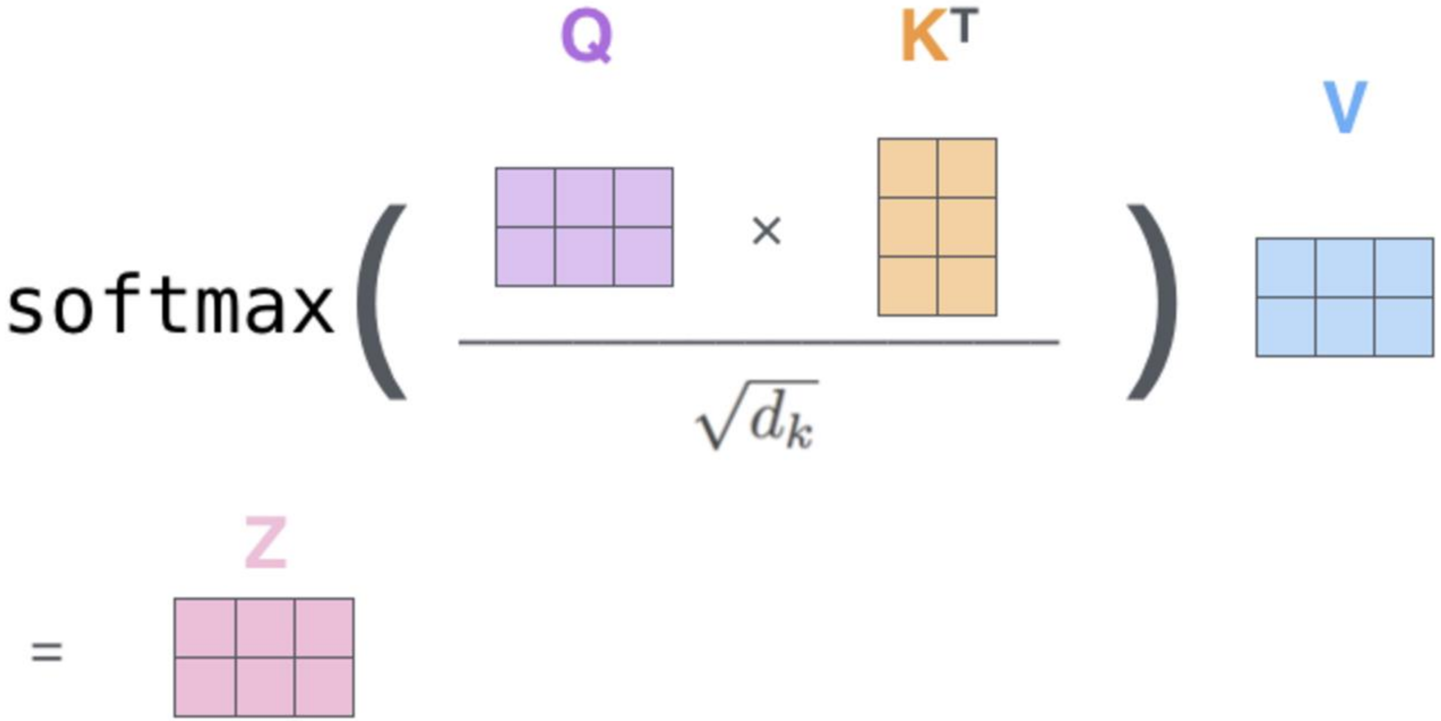
- d_{model} :embedding / hidden size

- The transformer makes three new matrices:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$

- where W_Q, W_K, W_V are learned weight matrices.

Matrix Calculation of Self-Attention



Matrix Calculation of Self-Attention

$$\text{Attention}(Q, K, V) = Softmax\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Self-attention

- **1. Query matrix** Q

Shape: $T \times d_k$

Row Q_i= " What does token i want to look for in other tokens?"

- **Intuition:**
  Each token asks a question: *"Which other tokens are relevant to me, given my role in this position?"*
  That question is encoded as the query vector Q_i.
  - You use these queries when you compute attention scores:
  - $\text{score}_{i,j} = Q_i \cdot K_j$
  - $Q_i$is the "search pattern" of token i.

Self-attention

- **2. Key matrix K**
  - Shape: $T \times d_k$
  - Row K_j“ =What information does token $j$offer?”
- **Intuition:**
  Each token advertises what it is about.
  The key vector $K_j$describes features that others might want to attend to: subject, object, position, etc., depending on what the model has learned.
- In the score $Q_i \cdot K_j$,you are matching:
  - “What token i is looking for” (query)
    with
    “What token $j$offers” (key).
- High dot product = strong match = high attention weight from i to j.

Self-attention

- **3.Value matrix** V
  - Shape: $T \times d_v$
  - Row V_j" =What information does token j*pass along* if you decide to attend to it?"
- **Intuition:**
  Once the model decides that token $j$is relevant to token i(via query–key similarity), it still needs to know *what* to take from j. That payload is V_j.
- The final output for token $i$is a weighted average of values:

$$\text{output}_i = \sum_{j=1}^{T} \alpha_{i,j} V_j,$$

- where $\alpha_{i,j}$are the attention weights (softmax of scores).

Self-attention recap

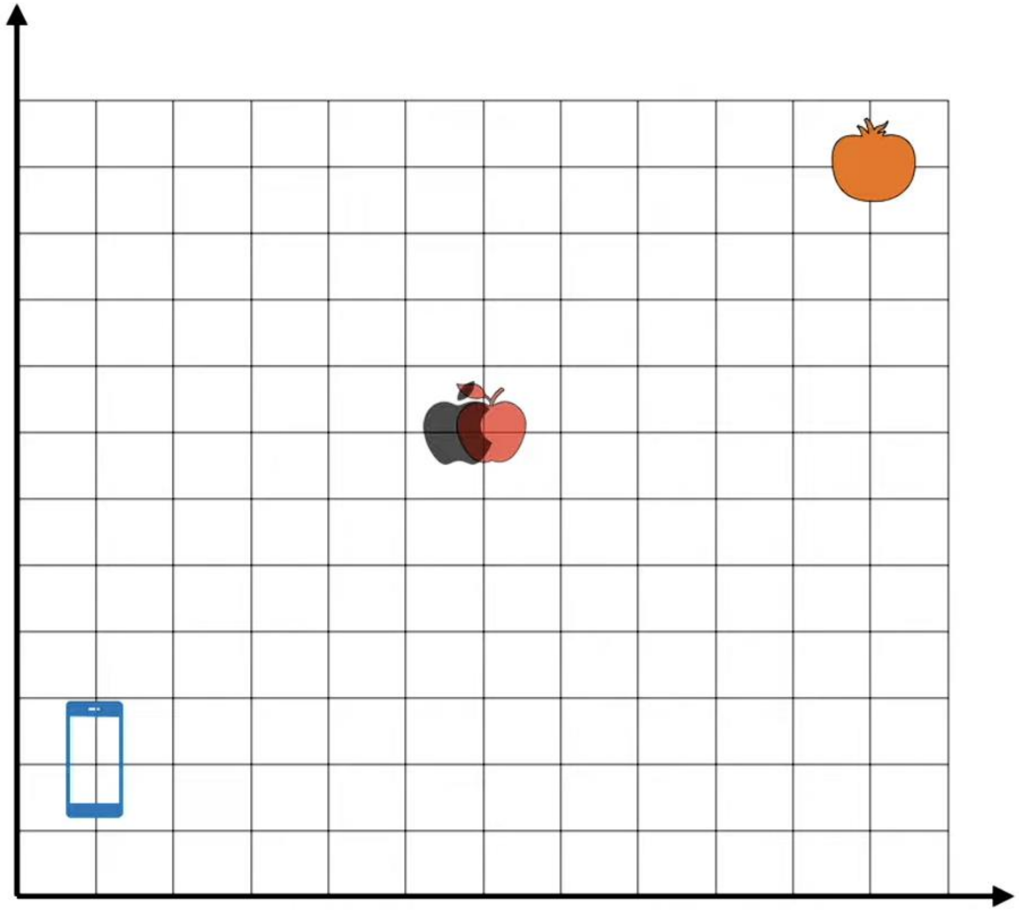
- Queries: what each token is asking for.

- Keys: what each token offers.

- Values: what information each token contributes when others attend to it.

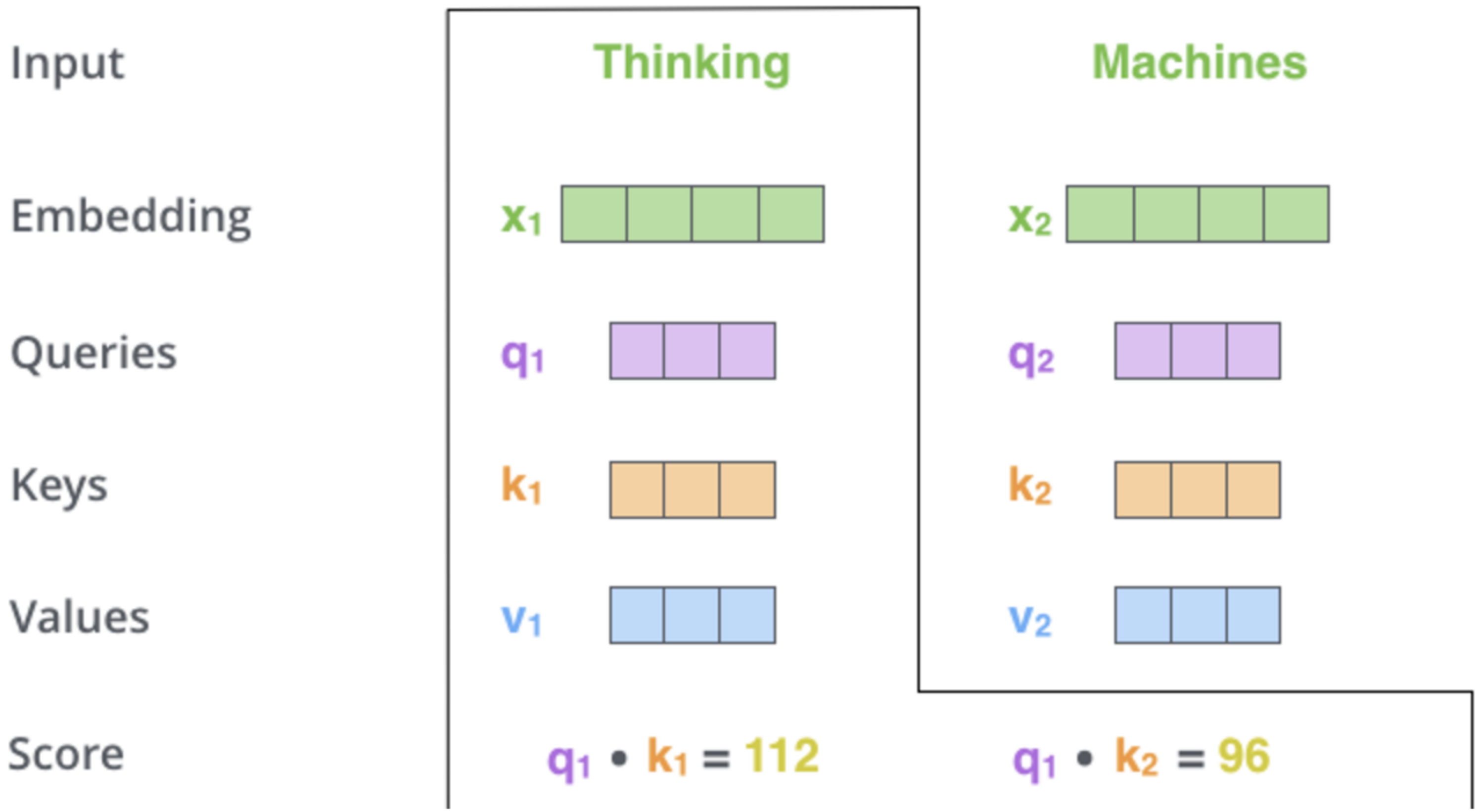
Attention

please buy an apple and an $\boxed{\text{orange}}$

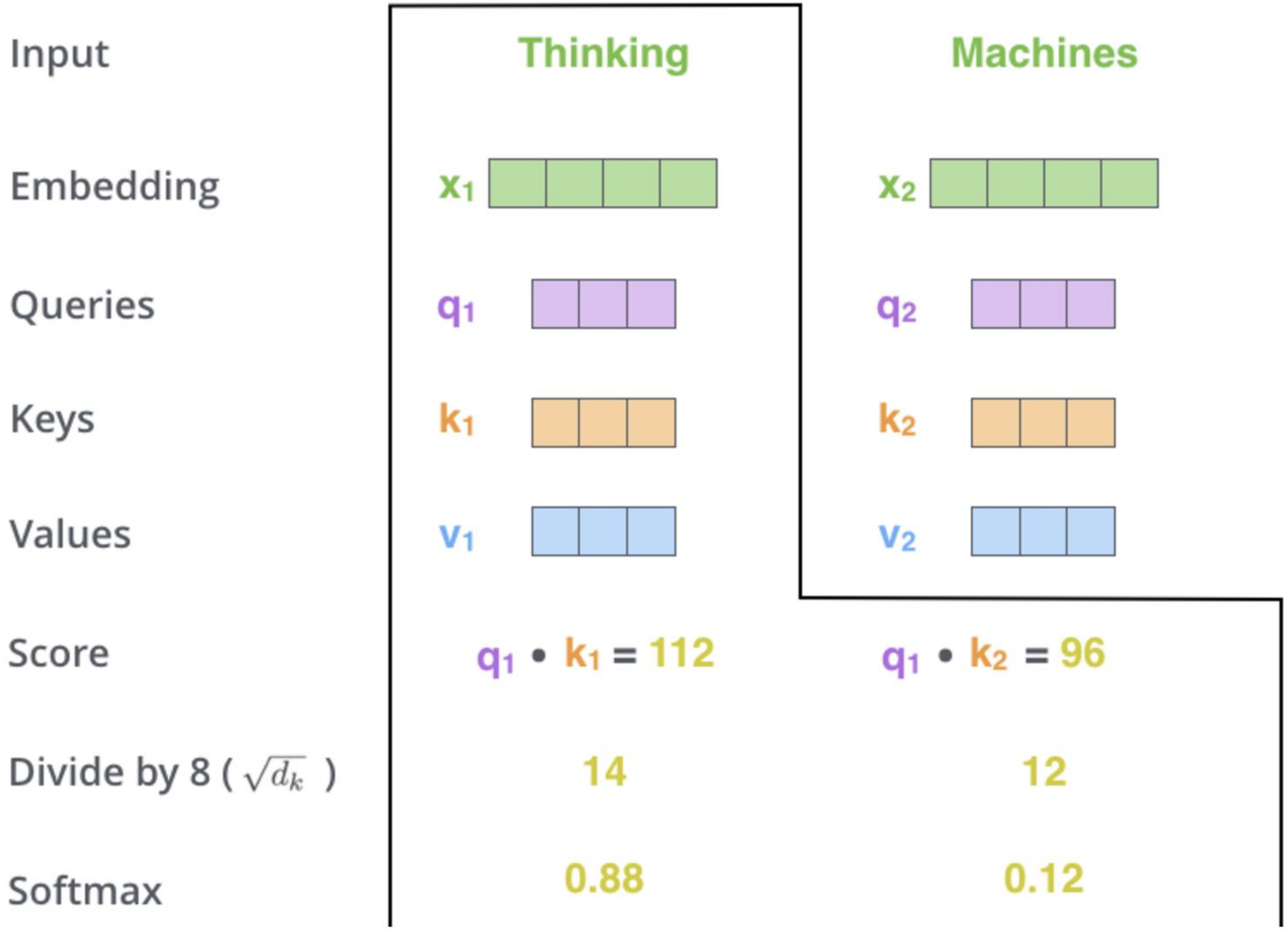


apple unveiled the new phone

Self-Attention

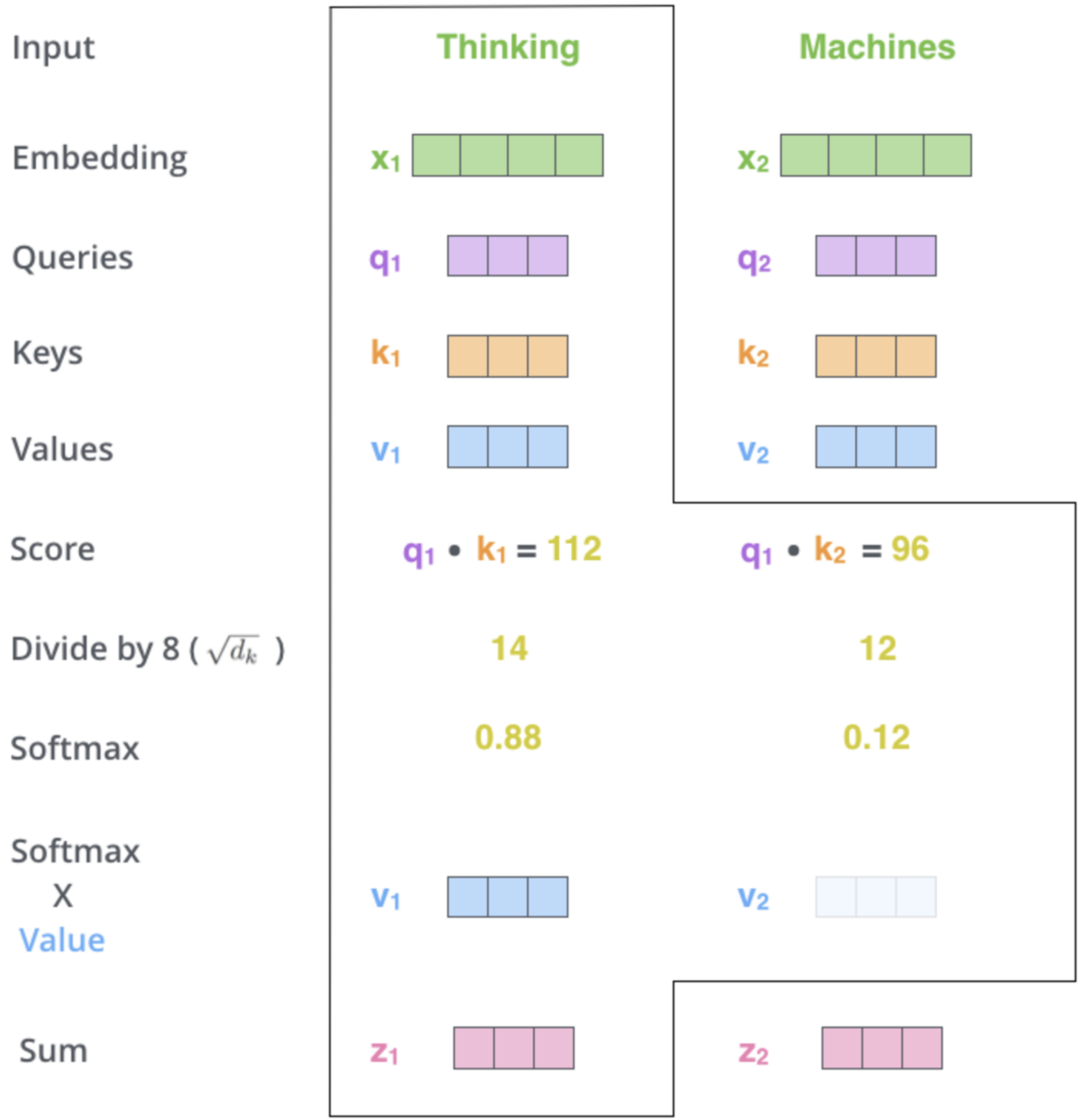


Self-Attention

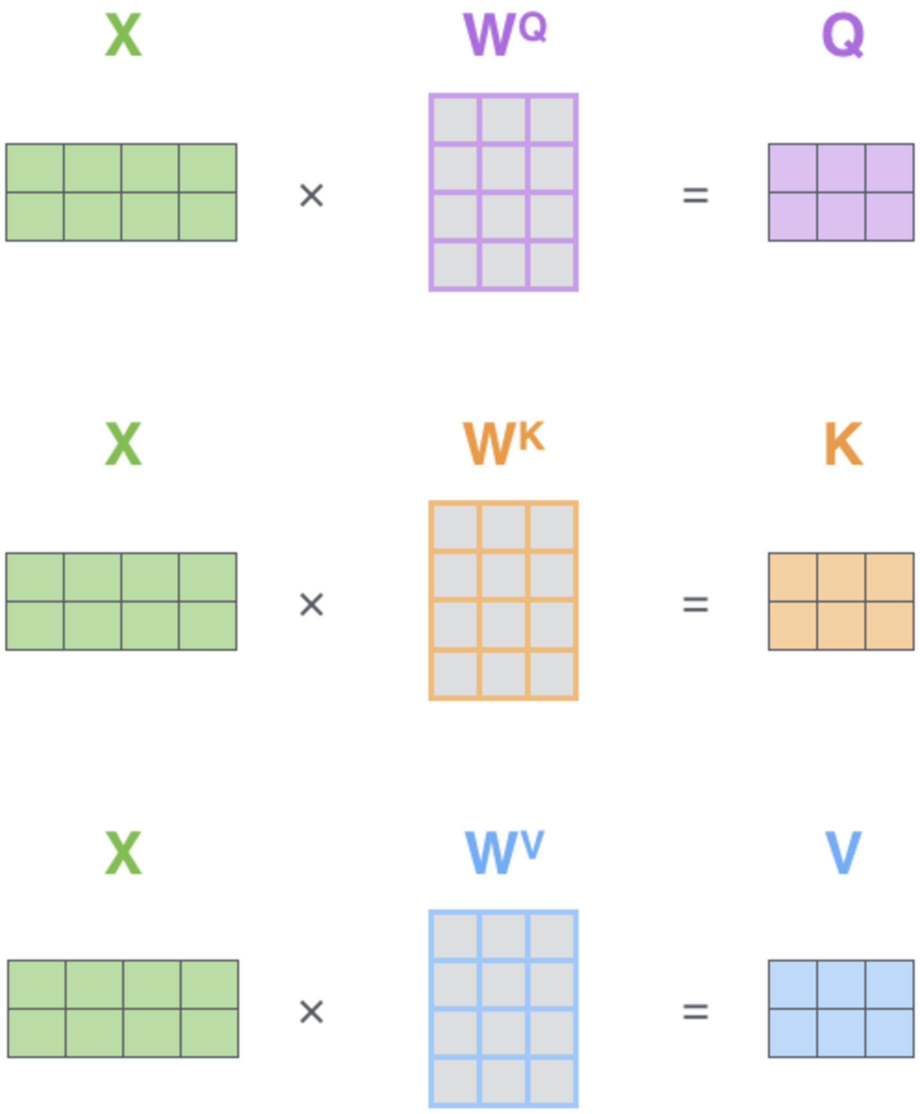


Self-Attention

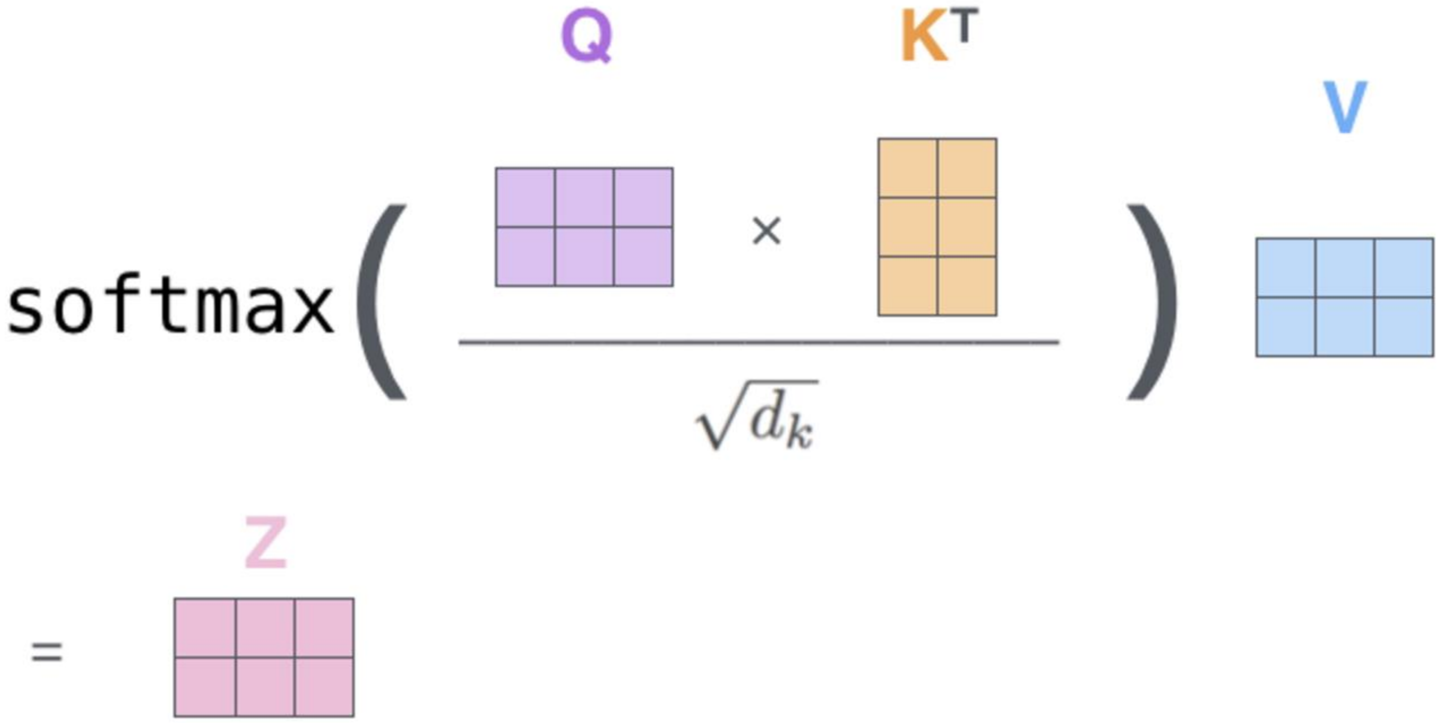
- **Efficiency:**
  - Computation parallelized across sequence tokens.
  - Masking applied for causal language models.



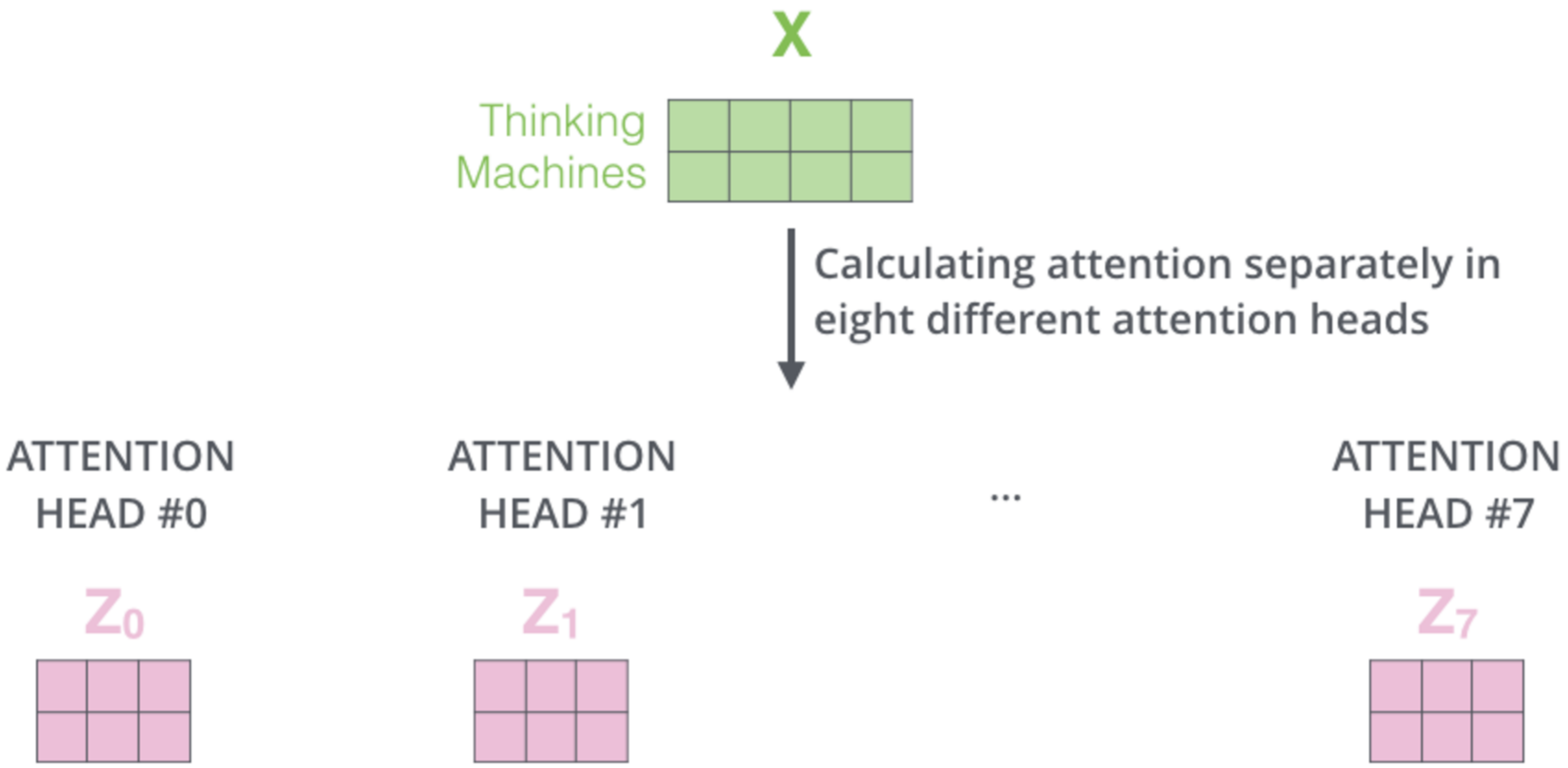
Matrix Calculation of Self-Attention



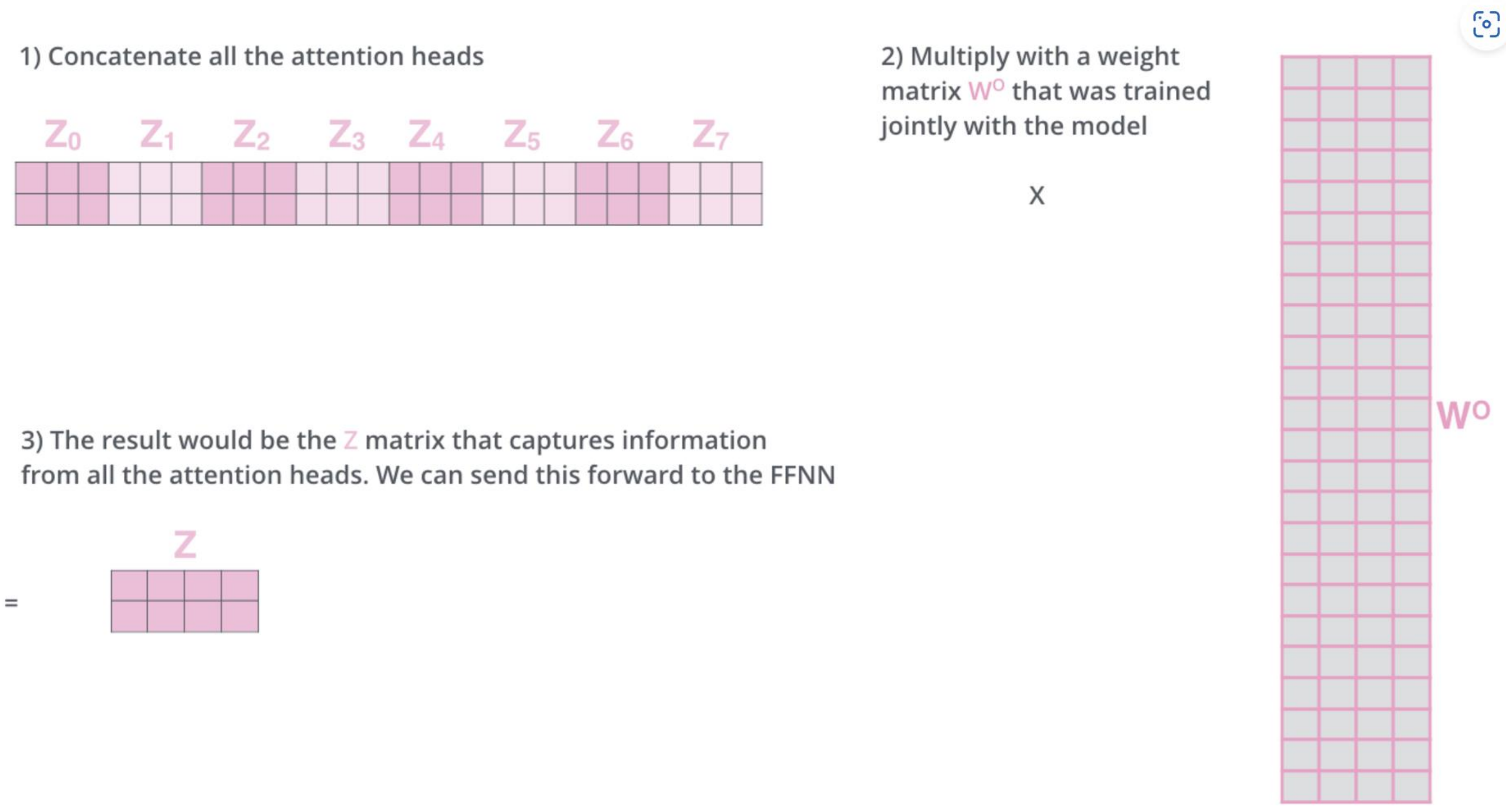
Matrix Calculation of Self-Attention



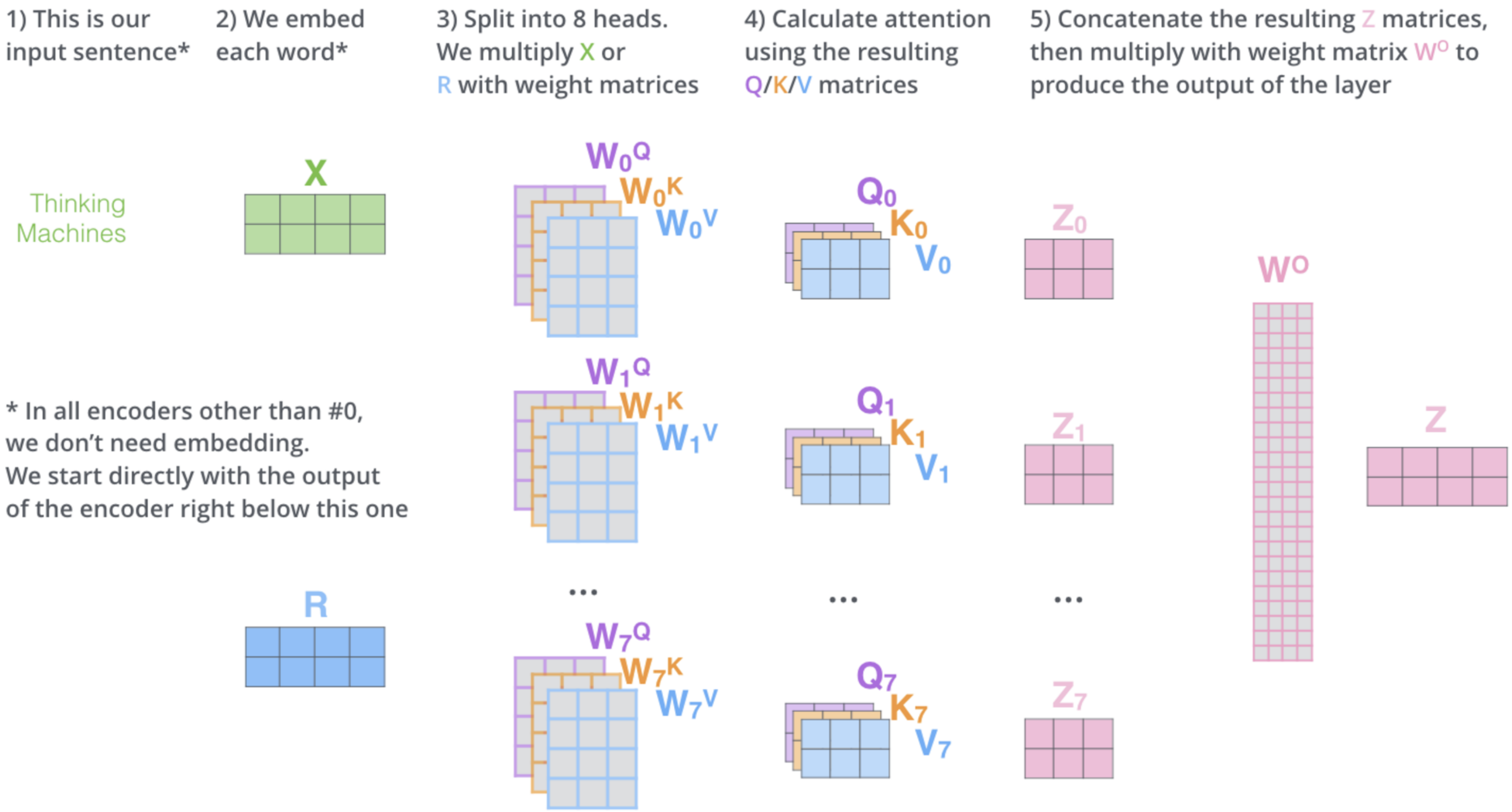
Multiple Attention Head



Multiple Attention Head



Multiple Attention Head- Matrix operation



Example

- Assume we already have 2-dimensional token embeddings:

  Token 1: “I”

  Token 2: “like”

  Token 3: “pizza”

- Write them as a matrix $X \in \mathbb{R}^{3\times 2}$:

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

- So:

  3 tokens (rows)

  embedding size $d_{model} = 2$

Example

- The transformer learns three weight matrices:

$$W_Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, W_K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, W_V = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Example

- For self-attention in a transformer:
- $Q = XW_Q, K = XW_K, V = XW_V$
- Given our choices, W_Q and W_K are identity, so:

$$Q = X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, K = X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

- Now compute values V:

$$V = XW_V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 0 \cdot 1 & 1 \cdot 1 + 0 \cdot 0 \\ 0 \cdot 1 + 1 \cdot 1 & 0 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 1 \cdot 1 & 1 \cdot 1 + 1 \cdot 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Example

- So each token has:

$$Q_1 = [1, 0] \ , K_1 = [1, 0] \ , V_1 = [1, 1]$$
$$Q_2 = [0, 1] \ , K_2 = [0, 1] \ , V_2 = [1, 0]$$
$$Q_3 = [1, 1] \ , K_3 = [1, 1] \ , V_3 = [2, 1]$$

Example

Compute attention scores $QK^\top$

- Attention scores between all query–key pairs:

$$\text{scores} = \frac{QK^\top}{\sqrt{d_k}}, d_k = 2$$

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} = \text{So} \quad {\color{red}QK^\top} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

- Now scale by $1/\sqrt{2} \approx 0.7071$:

$$\text{scores} \approx \begin{bmatrix} 0.7071 & 0 & 0.7071 \\ 0 & 0.7071 & 0.7071 \\ 0.7071 & 0.7071 & 1.4142 \end{bmatrix}$$

Example

- Interpretation:
  - Row 1 are the *raw* attention scores for token 1 “I” toward tokens $\{1, 2, 3\}$.
  - Row 2 for token 2 “like”.
  - Row 3 for token 3 “pizza”.

- scores $= \mathrm{Softmax}\left(\frac{QK^{\top}}{\sqrt{d_k}}, d_k\right) \Rightarrow A \approx \begin{bmatrix} 0.40 & 0.20 & 0.40 \\ 0.20 & 0.40 & 0.40 \\ 0.25 & 0.25 & 0.50 \end{bmatrix}$

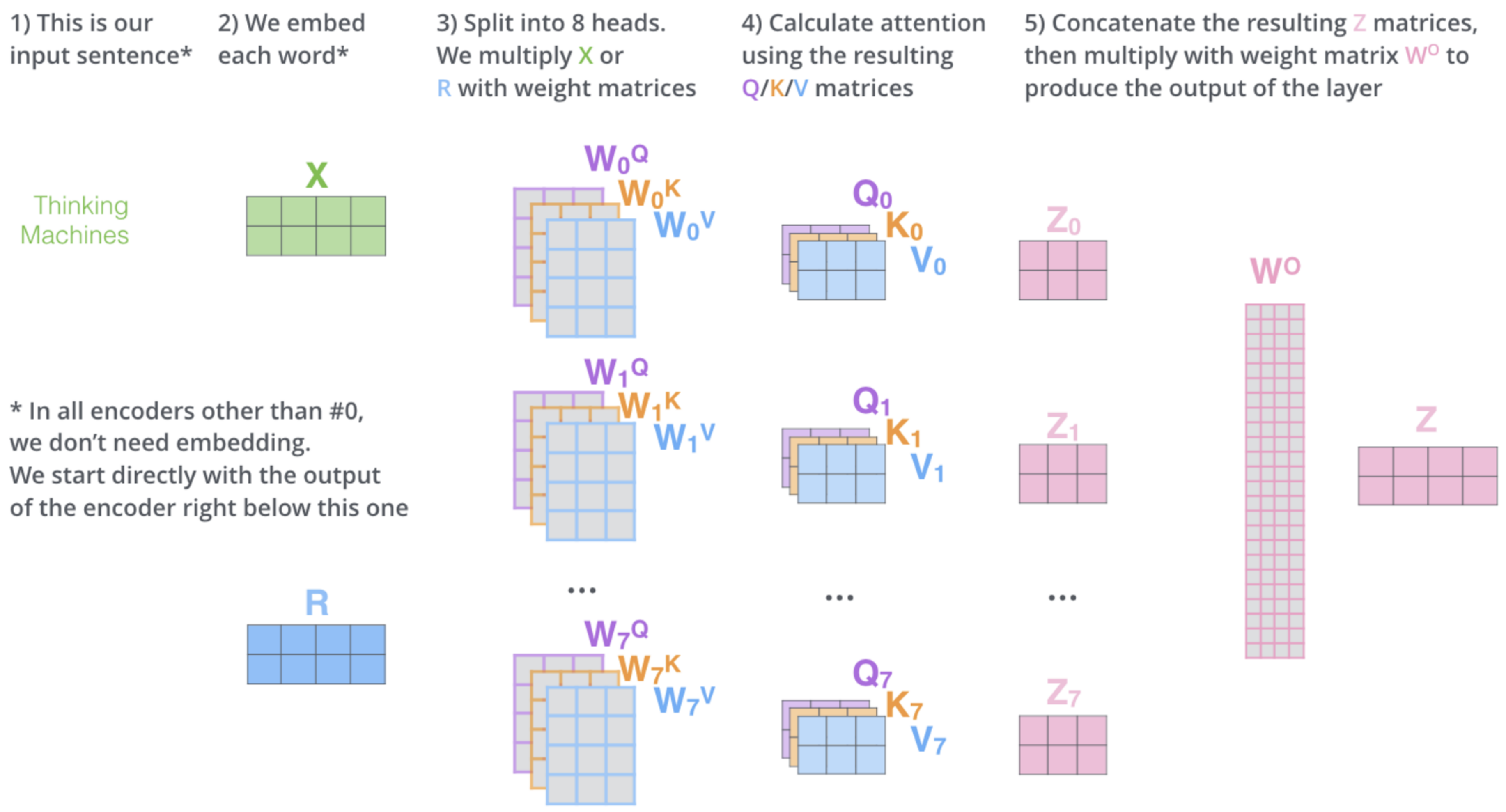
- Each row sums to 1.

Example

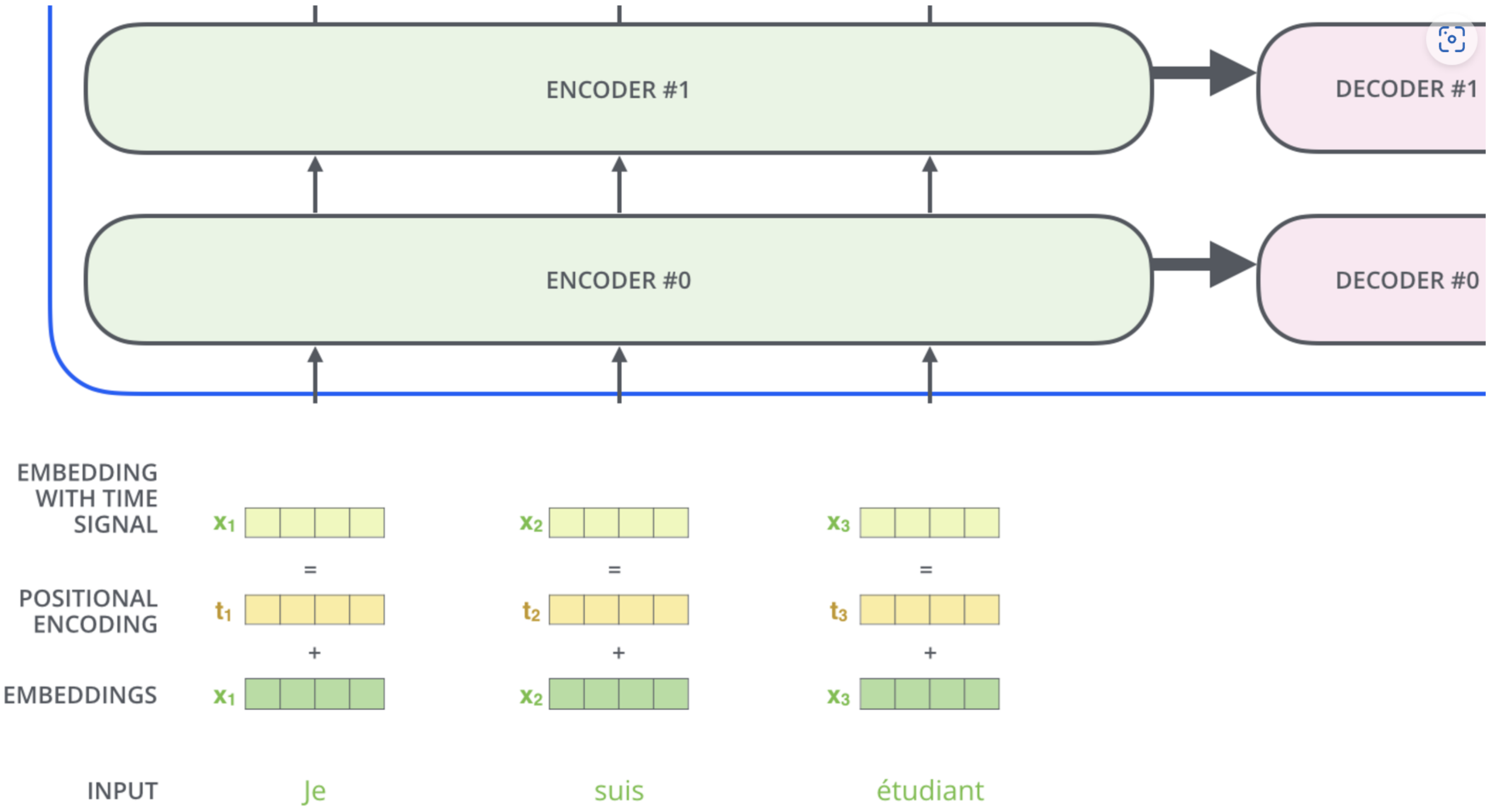
- AV

$$\begin{bmatrix} 0.40 & 0.20 & 0.40 \\ 0.20 & 0.40 & 0.40 \\ 0.25 & 0.25 & 0.50 \end{bmatrix} \text{x} \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \approx \begin{bmatrix} 1.40 & 0.80 \\ 1.40 & 0.60 \\ 1.50 & 0.75 \end{bmatrix}$$

Multiple Attention Head- Matrix operation

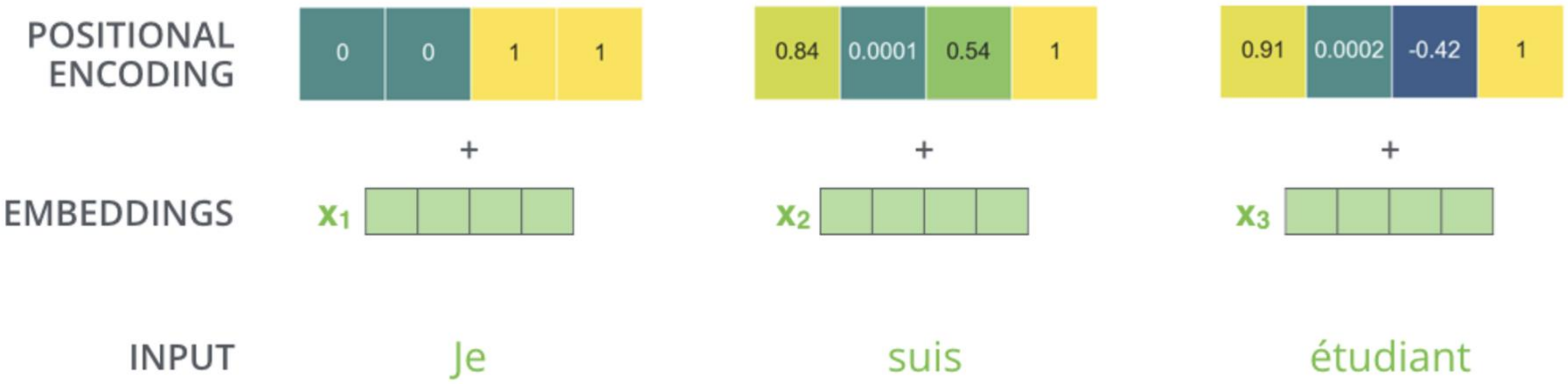


Representing The Order

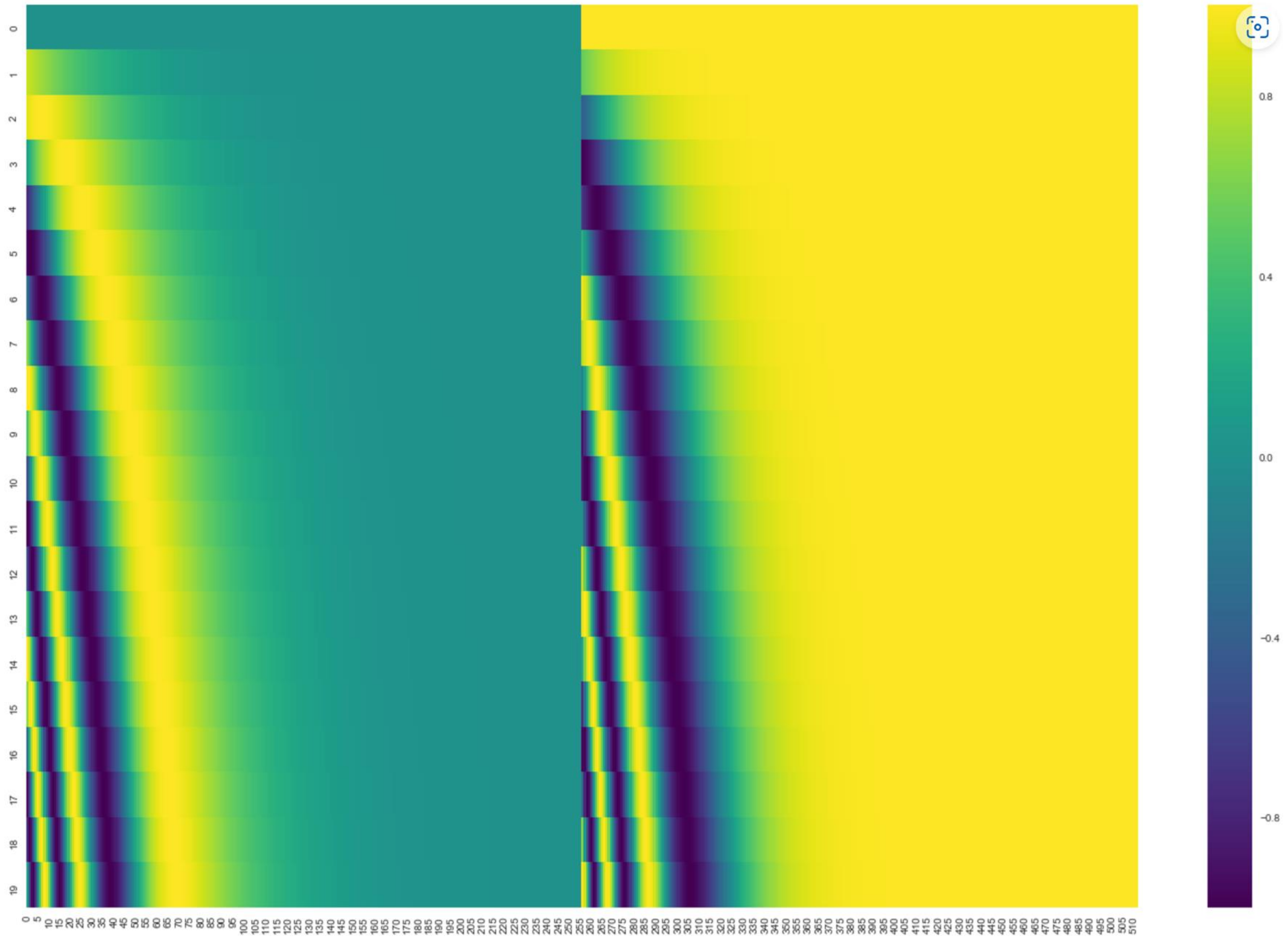


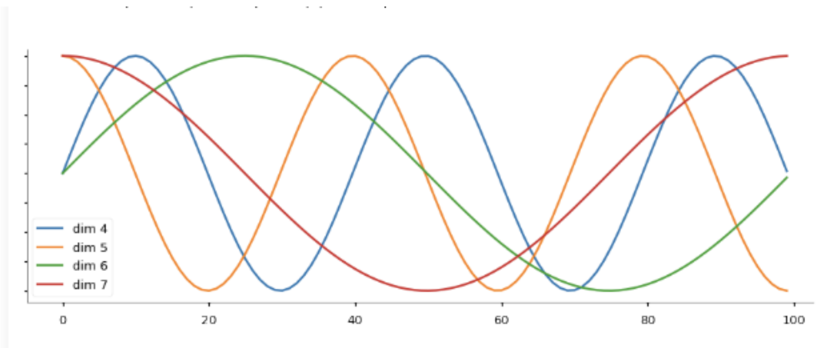
Representing the Order

- We also assign each position in our vocabulary an embedding vector

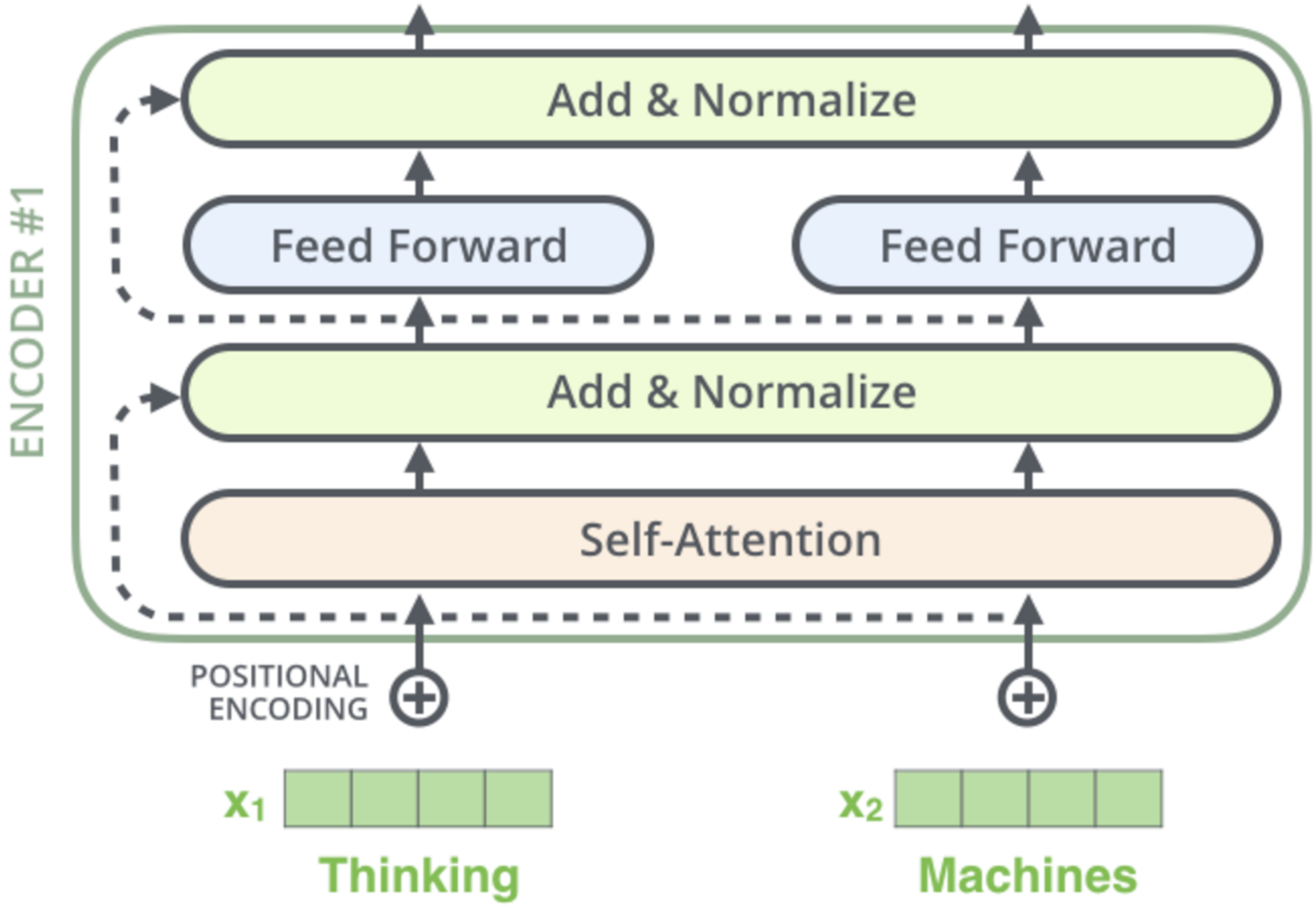


Representing the Order

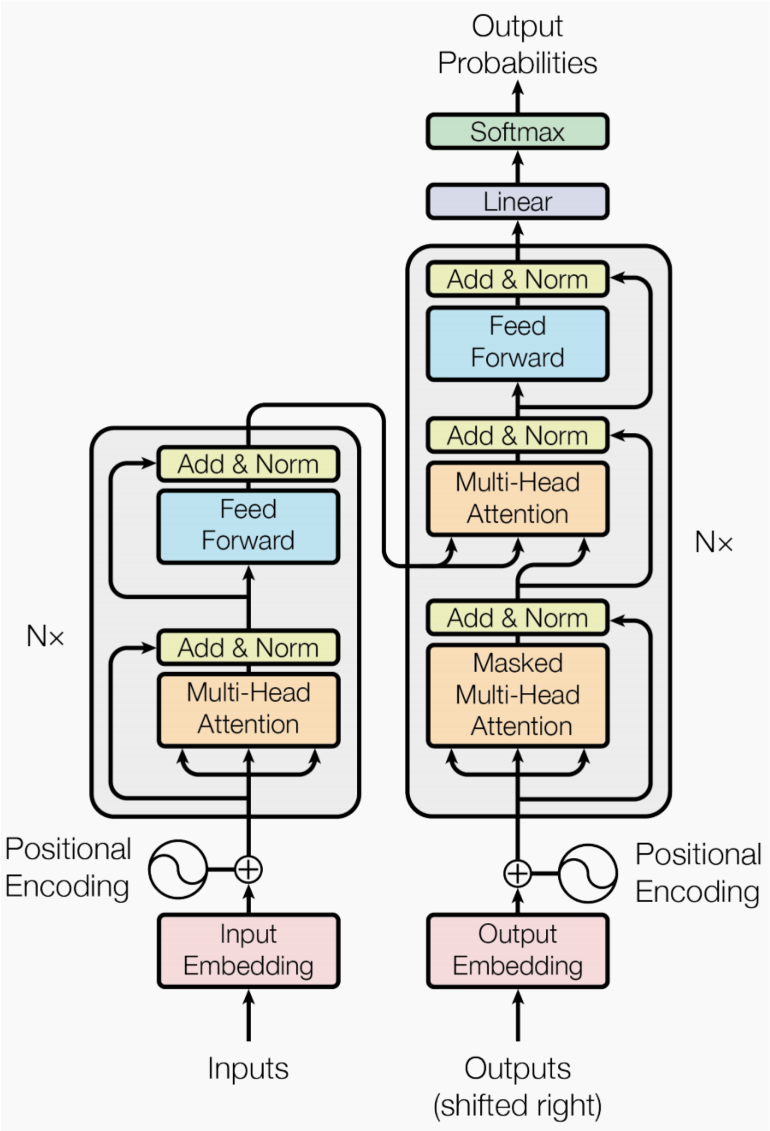




The Residuals



Transformers



RNN - Transformers

- The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.

- Understanding LSTM Networks -- colah's blog