EE-508: Hardware Foundations for Machine Learning Transformers

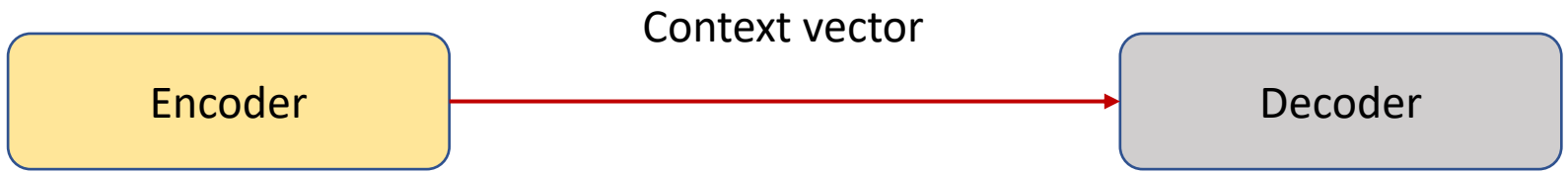
University of Southern California

Ming Hsieh Department of Electrical and Computer Engineering

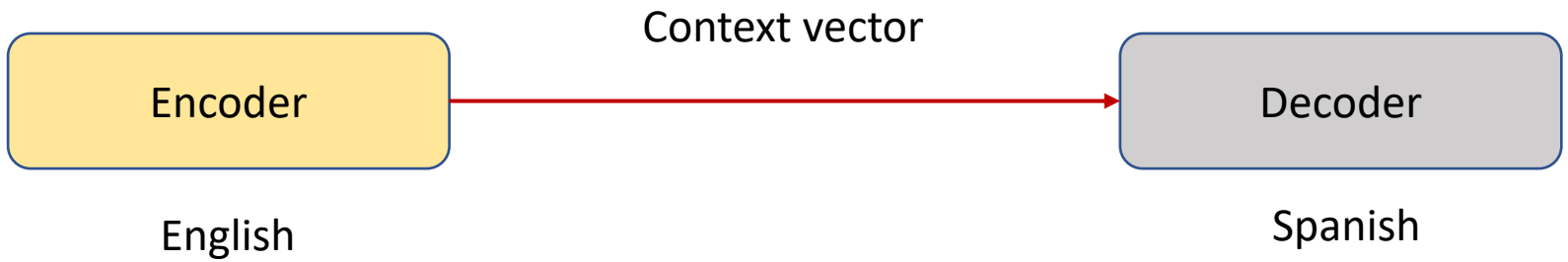
Instructors:
Arash Saifhashemi

Attention In Neural Networks

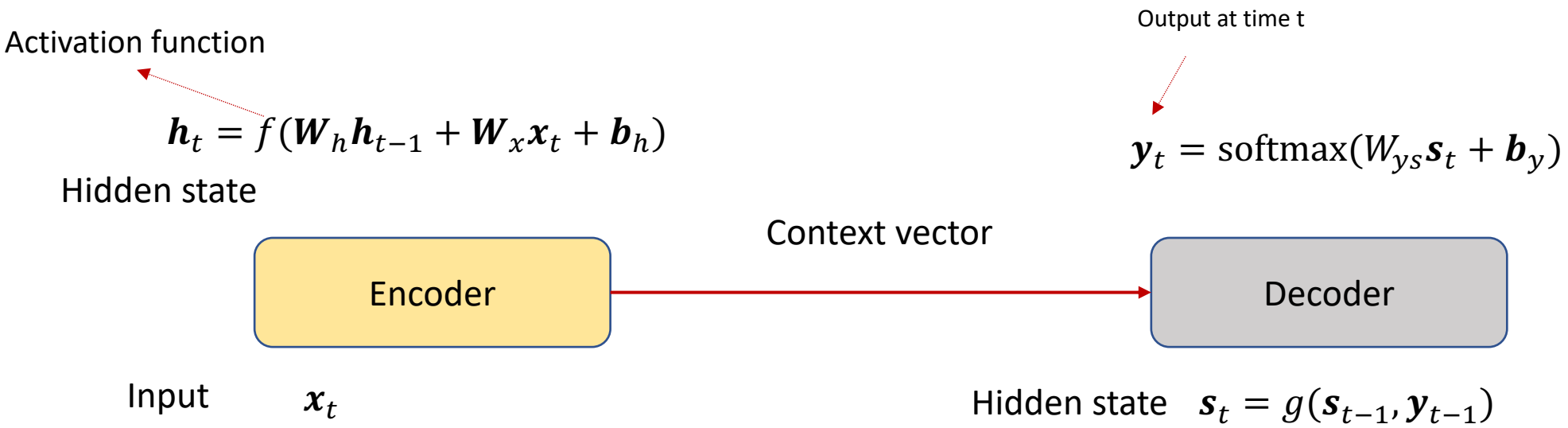
Seq2Seq Encoder-Decoder Summary



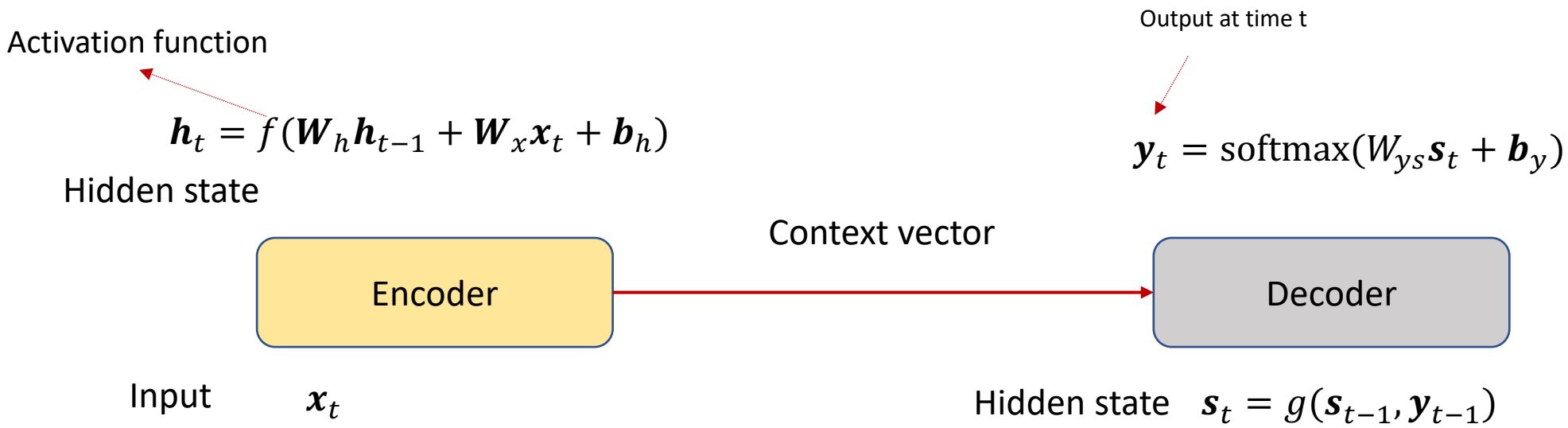
Seq2Seq Encoder-Decoder Summary



Seq2Seq Encoder-Decoder Summary



Seq2Seq Encoder-Decoder Summary



In simple RNN

$$\boldsymbol{s}_t = \text{RNN}(\boldsymbol{s}_{t-1}, \boldsymbol{y}_{t-1})$$

Example Implementation with tanh and embedding (E)

$$\boldsymbol{s}_t = \tanh\left(\boldsymbol{W}_{ss}\boldsymbol{s}_{t-1} + \boldsymbol{W}_{sy}\boldsymbol{E}(y_{t-1}) + \boldsymbol{b}_s\right)$$

In LSTM

$$\boldsymbol{s}_t, \boldsymbol{C}_t = \text{LSTM}(\boldsymbol{s}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{C}_{t-1})$$

$$\boldsymbol{s}_t = \boldsymbol{O}_t.\tanh(\boldsymbol{C}_t)$$

$$\boldsymbol{C}_t = \boldsymbol{f}_t \cdot \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \widetilde{\boldsymbol{C}}_t$$

Seq2Seq Encoder-Decoder Summary

Activation function

$$\boldsymbol{h}_t = f(\boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{b}_h)$$

Hidden state

Output at time t

$$\boldsymbol{y}_t = \text{softmax}(W_{ys} \boldsymbol{s}_t + \boldsymbol{b}_y)$$

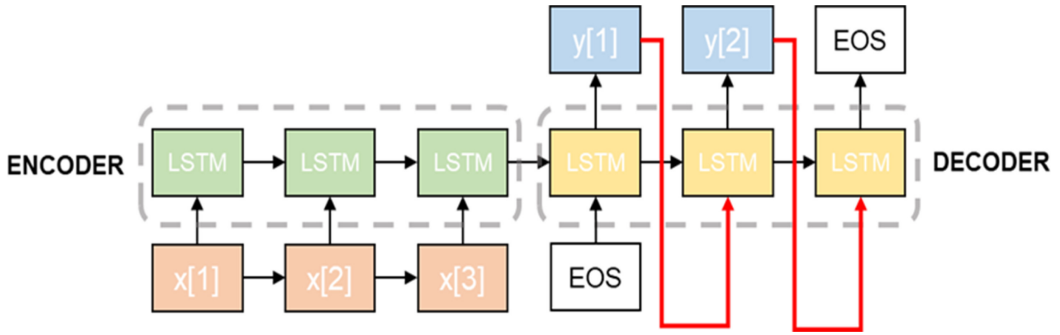
Context vector

Encoder

Decoder

Input $\boldsymbol{x}_t$

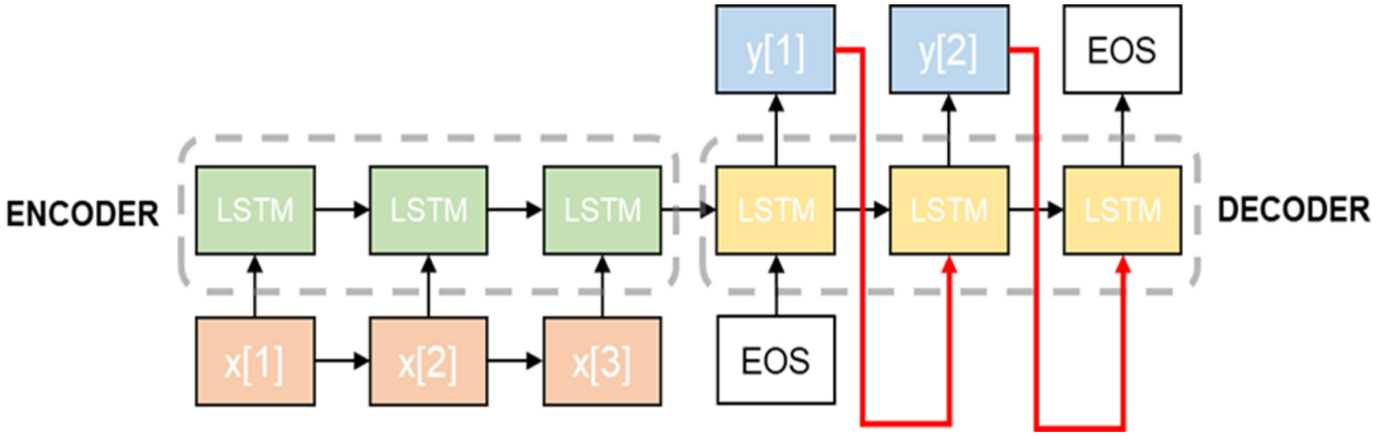
Hidden state $\boldsymbol{s}_t = g(\boldsymbol{s}_{t-1}, \boldsymbol{y}_{t-1})$



In this image each x[i] and y[i] is a vector

Source: "Electricity load forecasting using advanced feature selection and optimal deep learning model for the variable refrigerant flow systems", Woohyun Kim et al.

In Other Words...



Source: "Electricity load forecasting using advanced feature selection and optimal deep learning model for the variable refrigerant flow systems", Woohyun Kim et al.

- Each y[t] is:
  - The **softmax output** of the decoder at time t
  - A **probability distribution** over all words in the vocabulary
- It is computed as:

$$\boldsymbol{y}_t = \text{softmax}(\boldsymbol{W}_{ys}\boldsymbol{s}_t + \boldsymbol{b}_y)$$

Hidden state at time t

Weight vector

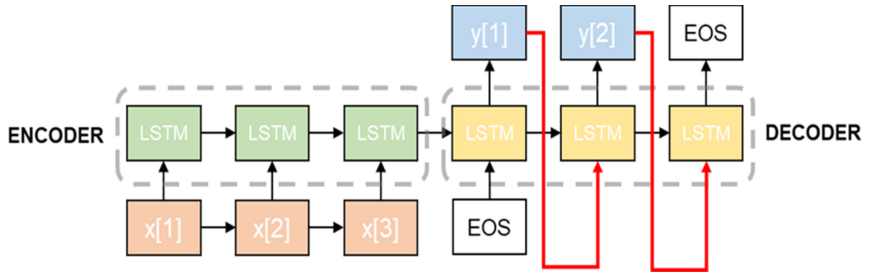
bias

Seq2Seq Decoder Output Example (with Softmax)

- Given:
- Vocabulary: ["I", "am", "you"]

$$s_1 = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix} \qquad W_y = \begin{bmatrix} 1.0 & 1.0 \\ 0.5 & -1.0 \\ -1.0 & 2.0 \end{bmatrix} \qquad b_y = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

$$z_1 = W_y \cdot s_1 + b_y = \begin{bmatrix} 1.0 \\ -0.55 \\ 1.1 \end{bmatrix}$$



$$\text{softmax}(z_1) = \frac{[e^{1.0}, e^{-0.55}, e^{1.1}]}{e^{1.0} + e^{-0.55} + e^{1.1}} = [0.431, 0.092, 0.477]$$

$$y_1 = [0.431, 0.092, 0.477]$$

Seq2Seq Decoder Output Example (with Softmax)

- Given:
- Vocabulary: ["I", "am", "you"]

$$s_1 = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix} \qquad W_y = \begin{bmatrix} 1.0 & 1.0 \\ 0.5 & -1.0 \\ -1.0 & 2.0 \end{bmatrix} \qquad b_y = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

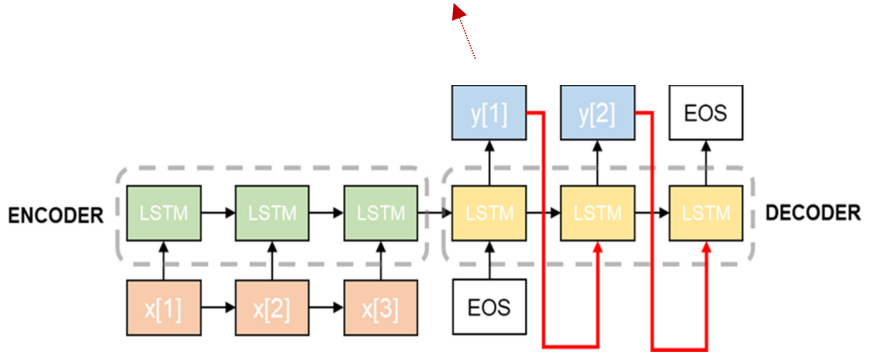
$$z_1 = W_y \cdot s_1 + b_y = \begin{bmatrix} 1.0 \\ -0.55 \\ 1.1 \end{bmatrix}$$

$$\text{softmax}(z_1) = \frac{[e^{1.0}, e^{-0.55}, e^{1.1}]}{e^{1.0} + e^{-0.55} + e^{1.1}} = [0.431, 0.092, 0.477]$$

$$y_1 = [0.431, 0.092, 0.477]$$

["I", "am", "you"]

$y_1 = [0.431, 0.092, 0.477]$



Seq2Seq Decoder Output Example (with Softmax)

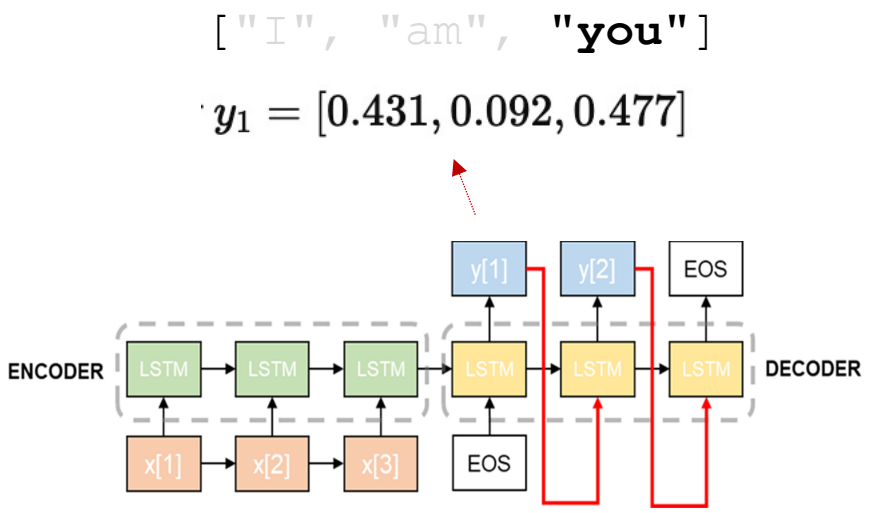
- Given:
- Vocabulary: ["I", "am", "you"]

$$s_1 = \begin{bmatrix} 0.3 \\ 0.7 \end{bmatrix} \qquad W_y = \begin{bmatrix} 1.0 & 1.0 \\ 0.5 & -1.0 \\ -1.0 & 2.0 \end{bmatrix} \qquad b_y = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

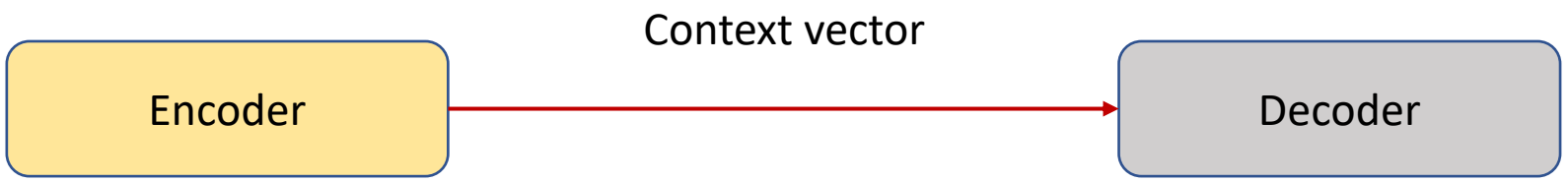
$$z_1 = W_y \cdot s_1 + b_y = \begin{bmatrix} 1.0 \\ -0.55 \\ 1.1 \end{bmatrix}$$

$$\text{softmax}(z_1) = \frac{[e^{1.0}, e^{-0.55}, e^{1.1}]}{e^{1.0} + e^{-0.55} + e^{1.1}} = [0.431, 0.092, 0.477]$$

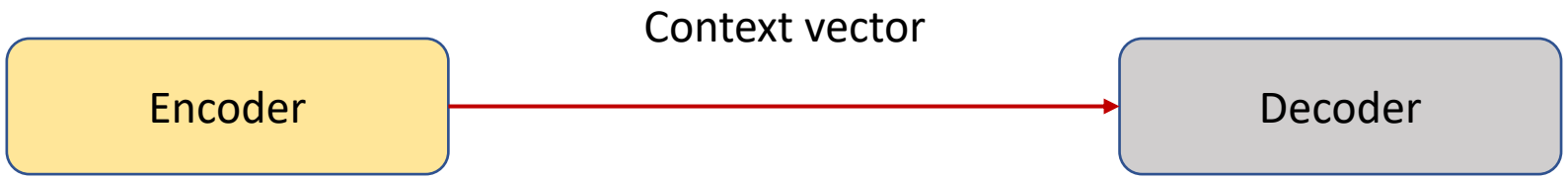
$$y_1 = [0.431, 0.092, 0.477]$$



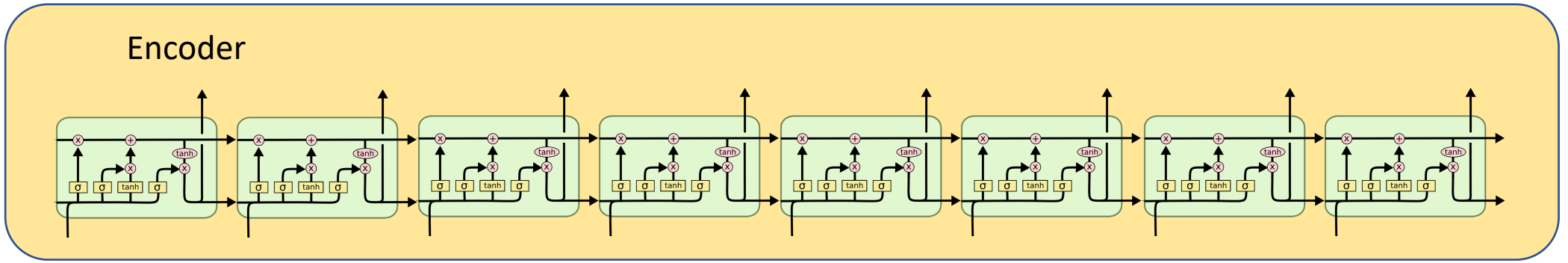
Seq2Seq Encoder-Decoder With Long Sequences



Seq2Seq Encoder-Decoder With Cross Attention



Example: *The bank manager, who had just reviewed the detailed report, which highlighted discrepancies in accounts that previously seemed impeccable, suggested that a specialized audit team, experienced in tracing intricate money trails, should be brought in to investigate the transactions, some of which appeared to violate standard financial regulations, before drawing any definitive conclusions.*



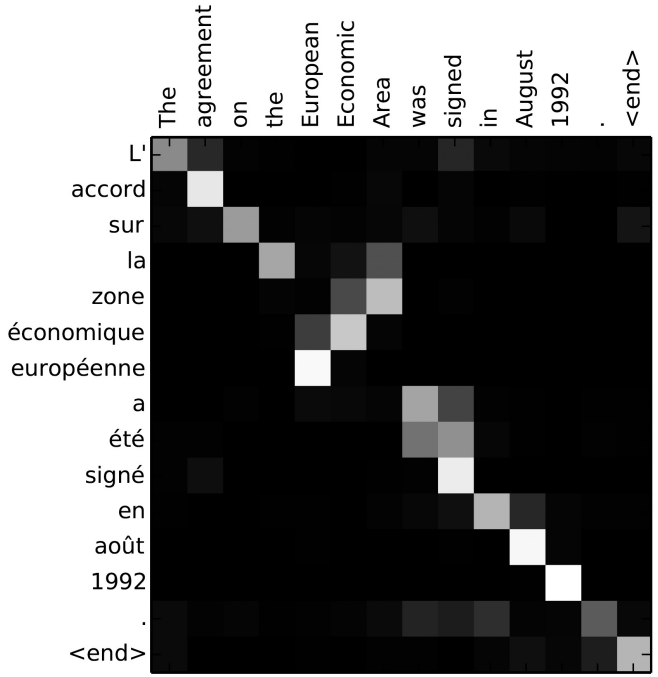
For long sequences, the context can become less precise.
It becomes harder to pack all information in a single vector.

Attention Mechanism

Helps a computer model focus on important words when it's translating or understanding a sentence

Attention Mechanism

Helps a computer model focus on important words when it's translating or understanding a sentence



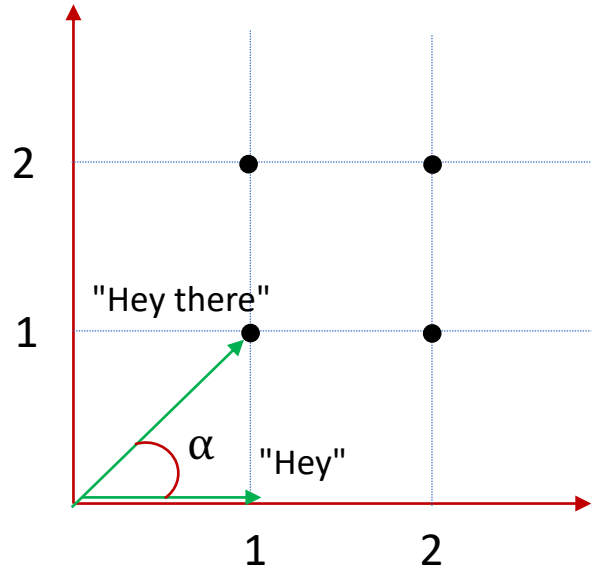
Alignment and Similarity

Cosine Similarity

Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	1	0
"Hey"	0	1

Cosine Similarity

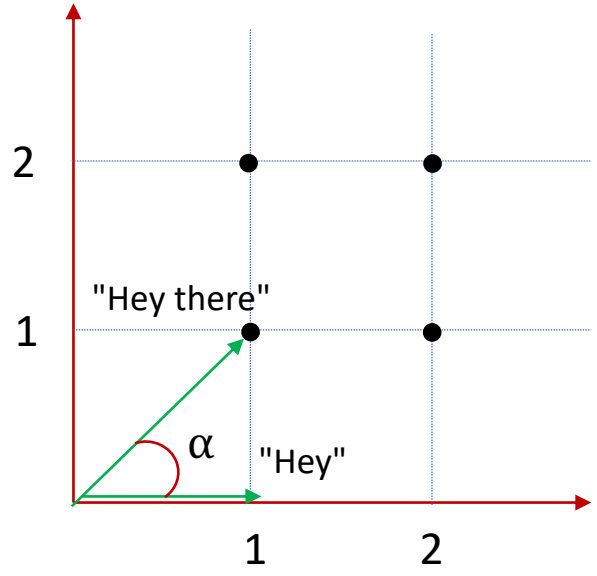
Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	0	1
"Hey"	1	0



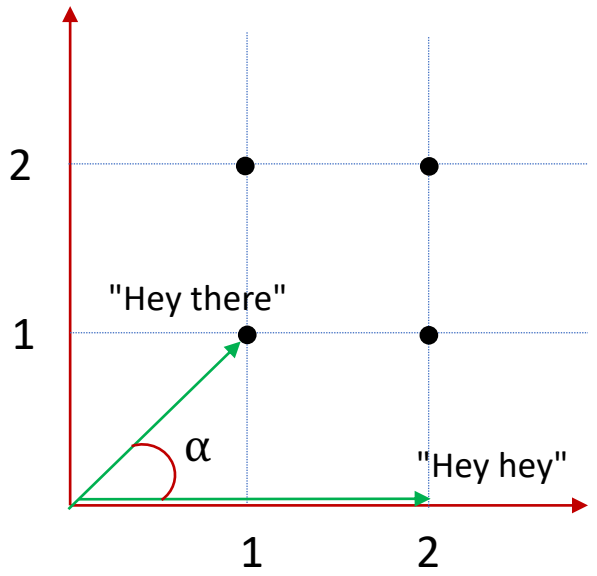
$$\cos \alpha = 0.71$$

Cosine Similarity

Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	0	1
"Hey"	1	0



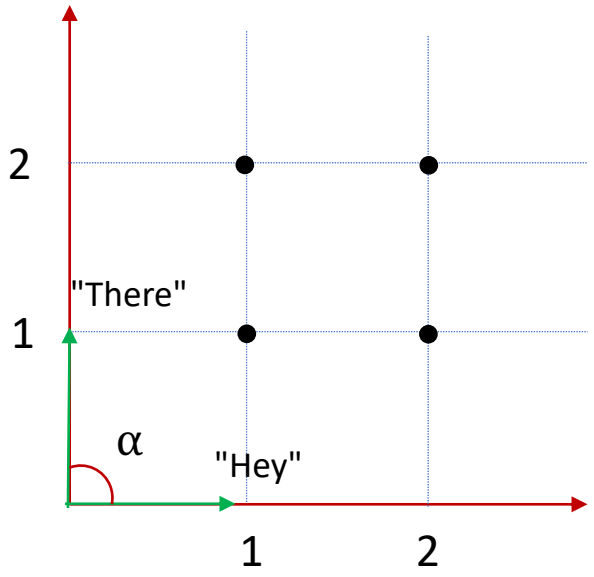
$\cos \alpha = 0.71$



$\cos \alpha = 0.71$

Cosine Similarity

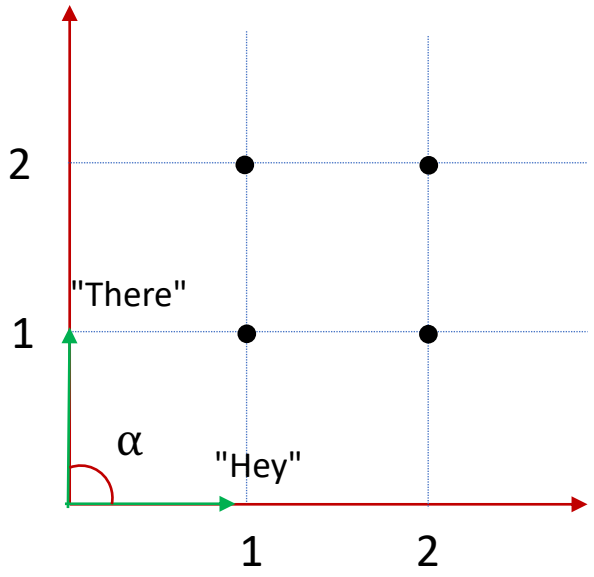
Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	0	1
"Hey"	1	0



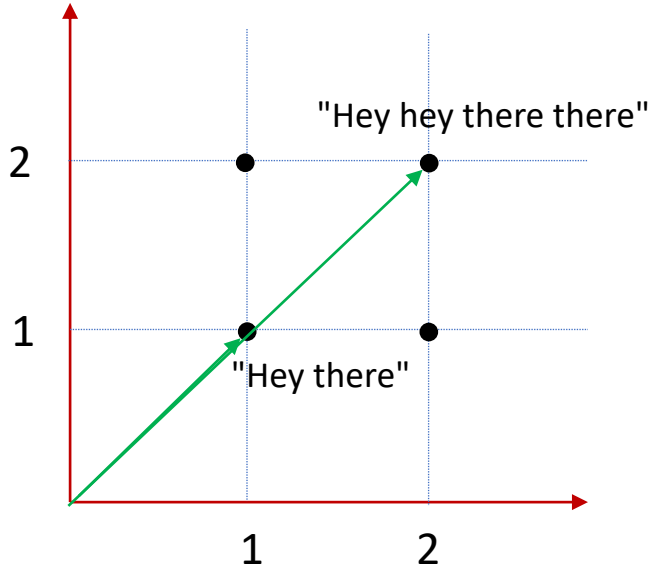
$$\cos \alpha = 0$$

Cosine Similarity

Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	0	1
"Hey"	1	0



$\cos \alpha = 0$



$\cos 0 = 1$

Cosine Similarity

Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	0	1
"Hey"	1	0

$$\text{Cosine Similarity}(A, B) = \frac{A.B}{\|A\|.\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Cosine Similarity

Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	1	0
"Hey"	0	1

$$\text{Cosine Similarity}(A, B) = \frac{A.B}{\|A\|.\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

$$\text{Cosine Similarity}(\text{"}Hey\text{"}, \text{"}Hey\ there\text{"}) = \frac{0\times1 + 1\times1}{\sqrt{0^2 + 1^2}\sqrt{1^2 + 1^2}} = \frac{1}{\sqrt{2}} = 0.71$$

Cosine Similarity

Phrase	Hey	There
"Hey there"	1	1
"Hey hey"	2	0
"Hey hey there there"	2	2
"There"	1	0
"Hey"	0	1

Dot product!

$$\text{Cosine Similarity}(A, B) = \frac{A.B}{\|A\|.\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

$$\text{Cosine Similarity}(\text{"}Hey\text{"}, \text{"}Hey\ there\text{"}) = \frac{0\times1 + 1\times1}{\sqrt{0^2 + 1^2}\sqrt{1^2 + 1^2}} = \frac{1}{\sqrt{2}} = 0.71$$

Seq2Seq Encoder-Decoder Summary

Activation function

$$h_t = f(W_h . h_{t-1} + W_x . x_t + b_h)$$

$$y_t = \text{softmax}(W_{ys} s_t + b_y)$$

Hidden state $\boldsymbol{H} = (h_1, h_2, \dots, h_{T_x})$

Output $\boldsymbol{Y} = (y_1, y_2, \dots, y_{T_y})$

Context vector

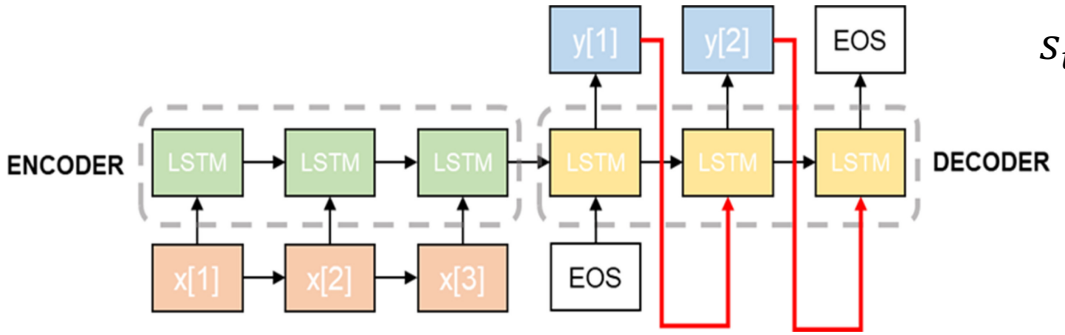
Encoder

Decoder

Input $\boldsymbol{X} = (x_1, x_2, \dots, x_{T_x})$

Hidden state $\boldsymbol{S} = (s_1, s_2, \dots, s_{T_y})$

$$s_t = g(s_{t-1}, y_{t-1})$$

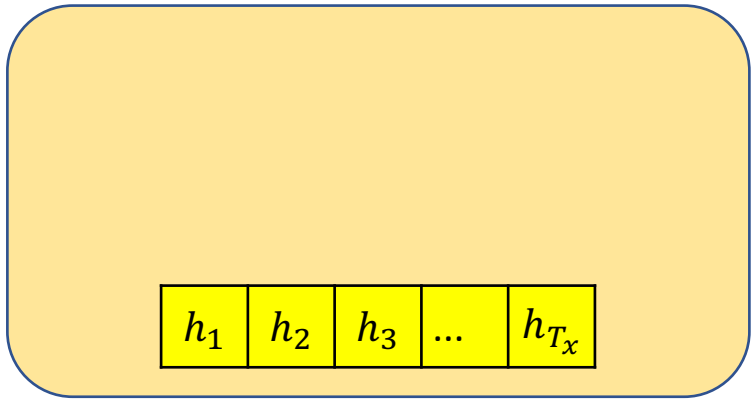


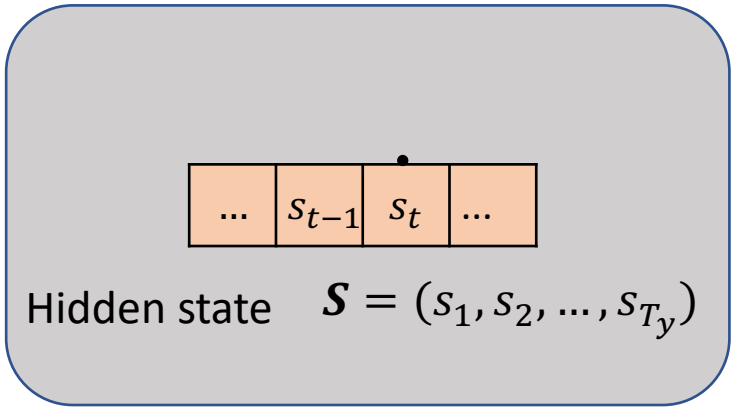
Source: "Electricity load forecasting using advanced feature selection and optimal deep learning model for the variable refrigerant flow systems", Woohyun Kim et al.

Seq2Seq Encoder-Decoder With Cross Attention

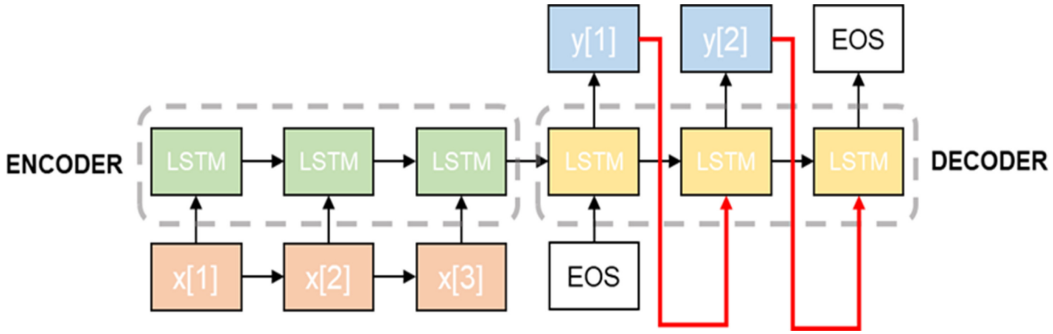
Hidden state

Output $\boldsymbol{Y} = (y_1, y_2, \ldots, y_{T_y})$





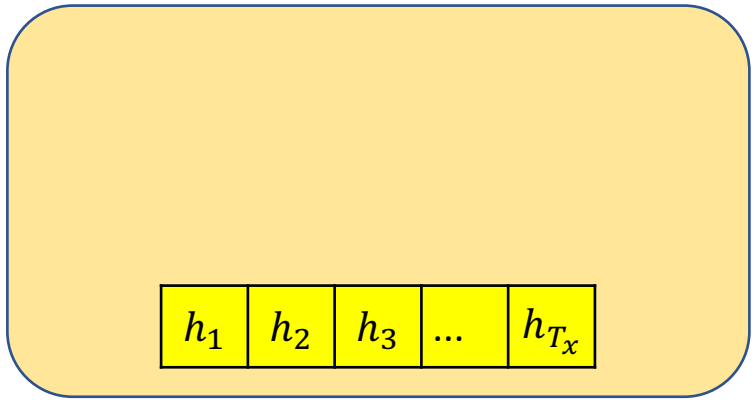
Input x_1 x_2 x_3 ... x_{T_x}



Source: "Electricity load forecasting using advanced feature selection and optimal deep learning model for the variable refrigerant flow systems", Woohyun Kim et al.

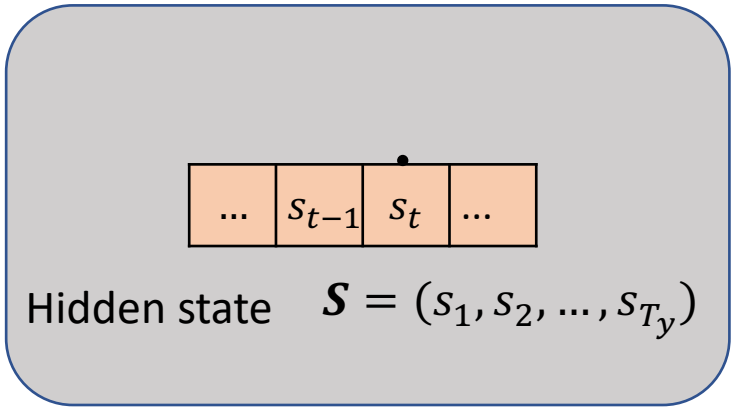
Seq2Seq Encoder-Decoder With Cross Attention

Hidden state



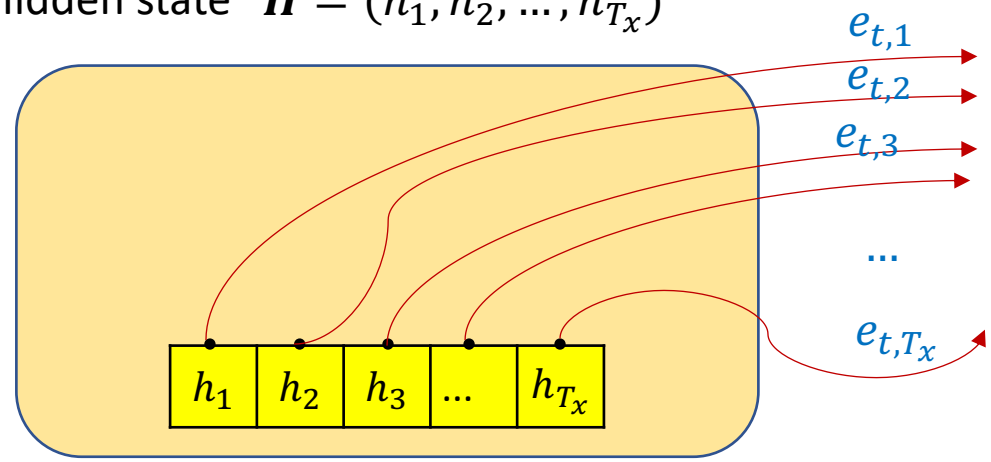
Input x_1 x_2 x_3 ... x_{T_x}

Output $\boldsymbol{Y} = (y_1, y_2, \ldots, y_{T_y})$



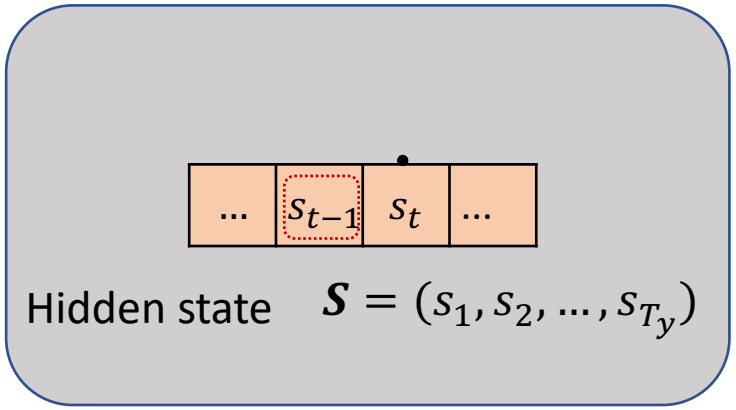
Seq2Seq Encoder-Decoder With Cross Attention

Hidden state $\boldsymbol{H} = (h_1, h_2, \dots, h_{T_x})$



Input x_1 x_2 x_3 ... x_{T_x}

Output $\boldsymbol{Y} = (y_1, y_2, \dots, y_{T_y})$

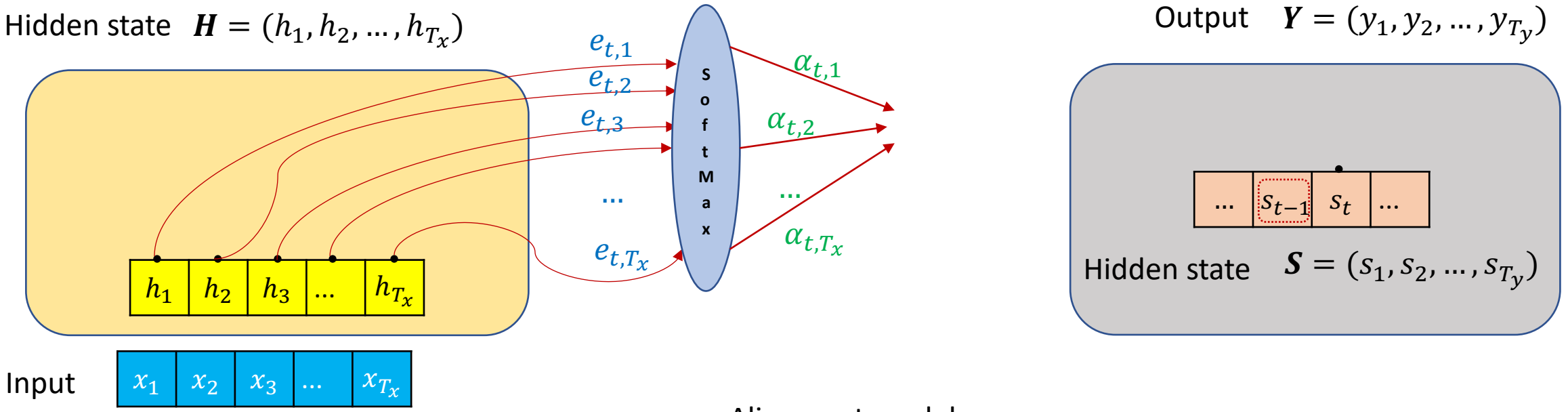


Alignment model

$$e_{t,i} = a(s_{t-1}, h_i)$$

Energy score: compatibility between input i and output t.

Seq2Seq Encoder-Decoder With Cross Attention



Alignment model

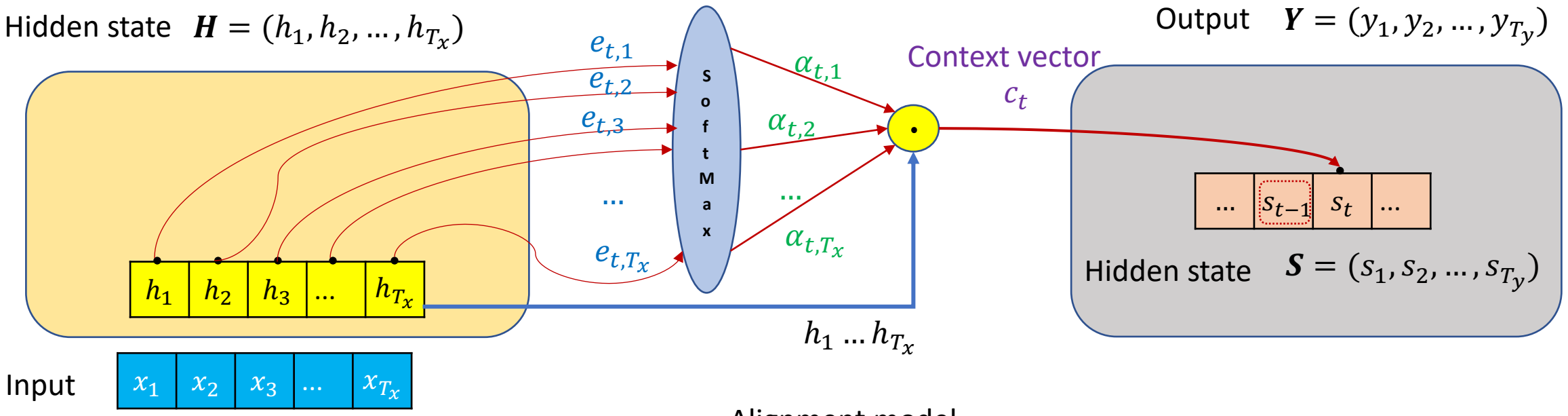
$$e_{t,i} = a(s_{t-1}, h_i)$$

Energy score: compatibility between input i and output t.

Softmax: Produce a probability distribution over the input sequence

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})}$$

Seq2Seq Encoder-Decoder With Cross Attention



Alignment model

$$e_{t,i} = a(s_{t-1}, h_i)$$

Energy score: compatibility between input i and output t.

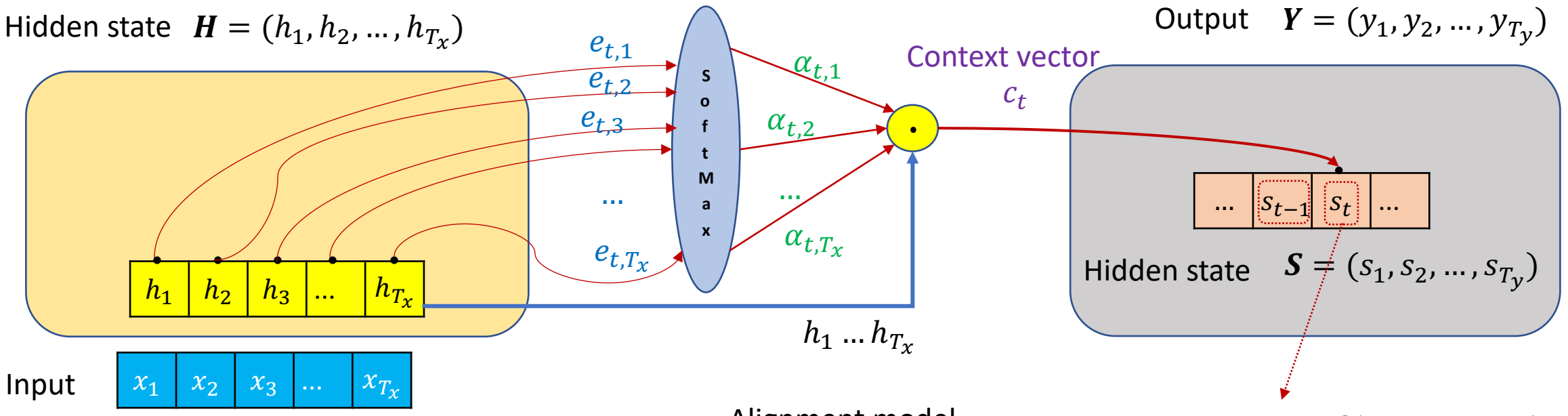
Softmax: Produce a probability distribution over the input sequence

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})}$$

Context Vector: How much of each hidden state should we take?

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i$$

Seq2Seq Encoder-Decoder With Cross Attention



Alignment model

$$e_{t,i} = a(s_{t-1}, h_i)$$

$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$

Energy score: compatibility between input i and output t.

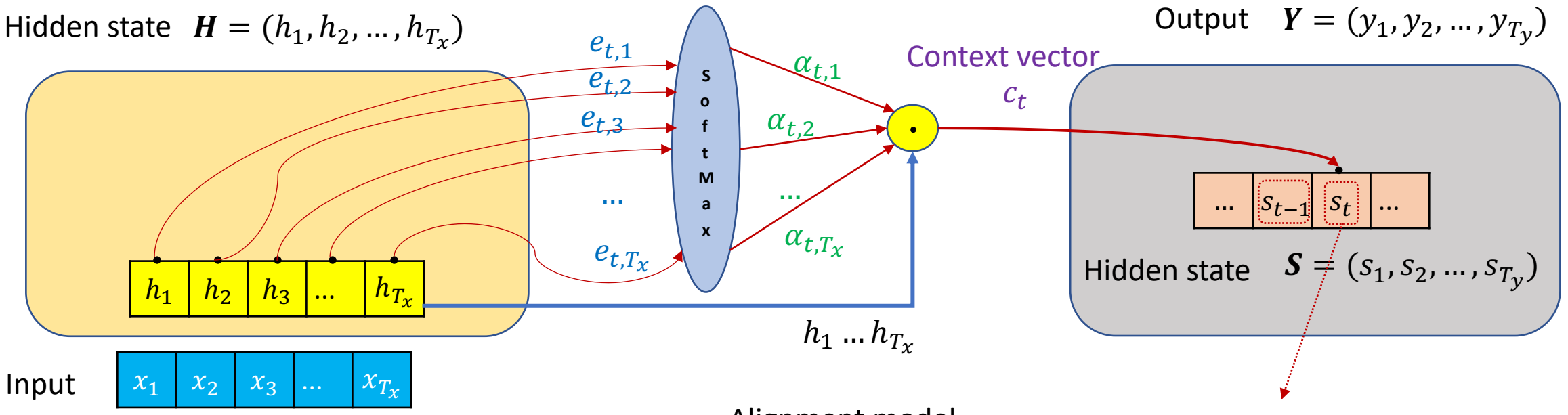
Softmax: Produce a probability distribution over the input sequence

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})}$$

Context Vector: How much of each hidden state should we take?

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i$$

Seq2Seq Encoder-Decoder With Cross Attention



$$s_t = f(s_{t-1}, y_{t-1}, c_t)$$

Alignment model

$$e_{t,i} = a(s_{t-1}, h_i)$$

Energy score: compatibility between input i and output t.

Softmax: Produce a probability distribution over the input sequence

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})}$$

Context Vector: How much of each hidden state should we take?

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i$$

$$s_t = \text{RNN}(s_{t-1}, [y_{t-1}, c_t])$$

$$s_t, C_t = \text{LSTM}(s_{t-1}, [y_{t-1}, c_t], C_{t-1})$$

LSTM cell state

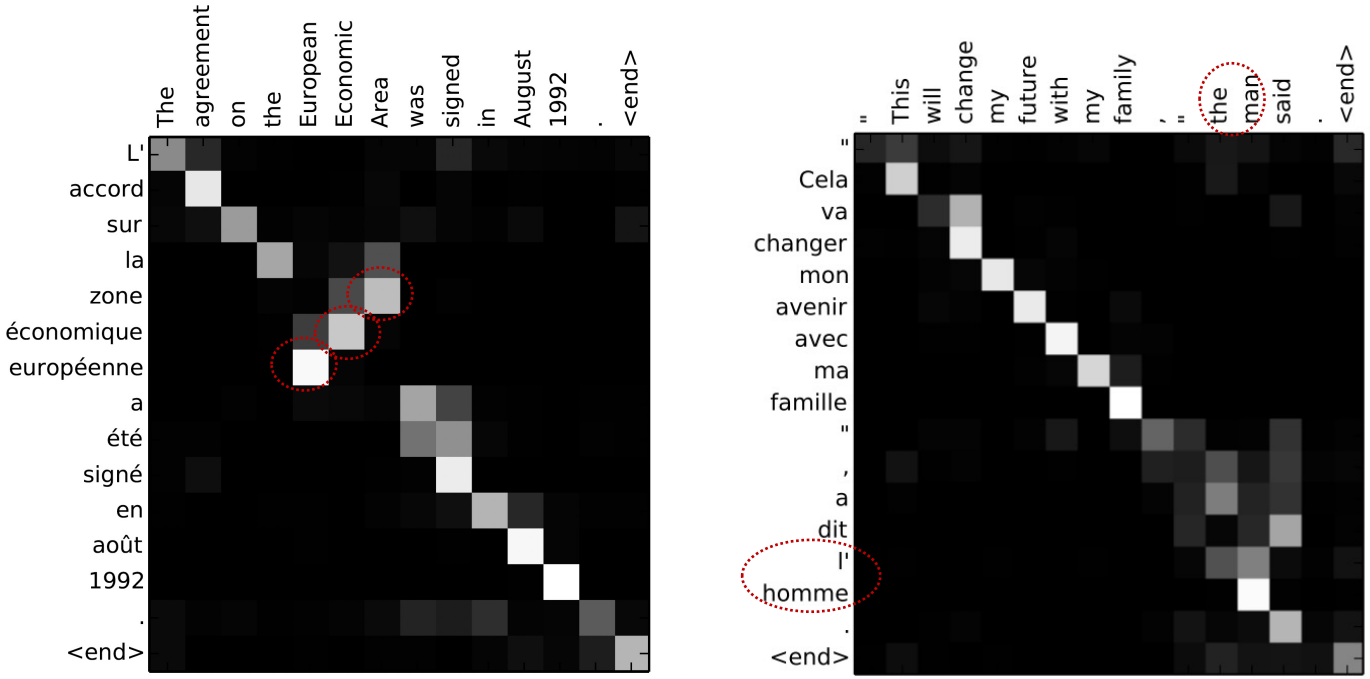
Context vector

Bahdanau (Additive) Attention

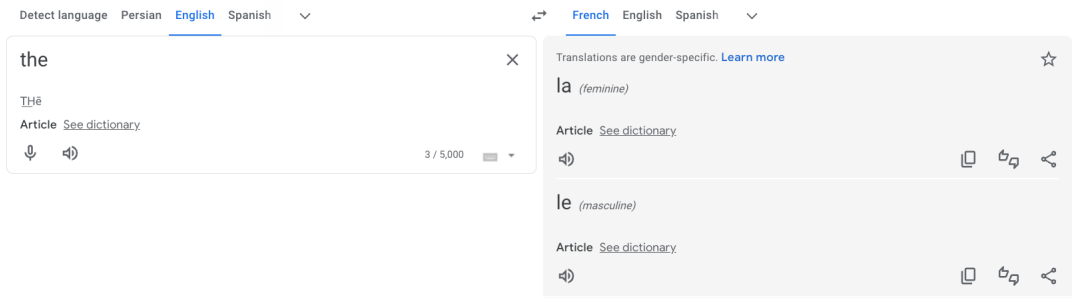
- Bahdanau attention (2015) helps a decoder **focus** on relevant parts of the input sequence **dynamically** at each time step.
- **Alignment score (energy)**: $e_{t,i} = a(s_{t-1}, h_i)$
  - Measures how well input position iii matches current decoder step t.
  - **a** is a small feedforward neural network (the **alignment model**).
- **Attention weights (softmax over energy)**: $\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})}$
- **Context vector** (weighted sum): $\boldsymbol{c}_t = \sum_{i=1}^{T_x} \alpha_{t,i} \boldsymbol{h}_i$
- **Decoder uses $\boldsymbol{c}_t$ and $\boldsymbol{s}_{t-1}$ to produce $\boldsymbol{s}_t$ and $\boldsymbol{y}_t$.**

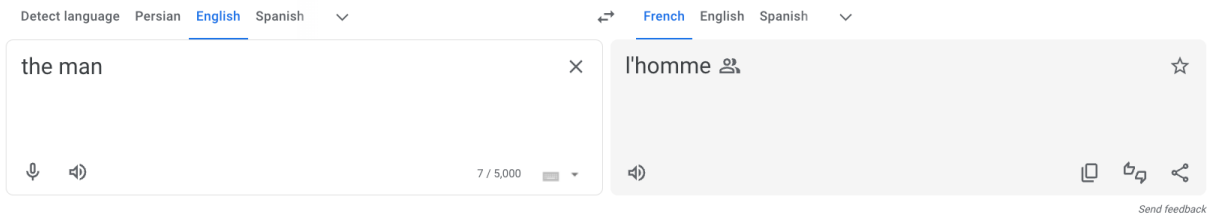
In additive attention, a is a neural network: $a(s_{t-1}, h_i) = v_a^\top \tanh(W_s s_{t-1} + W_h h_i)$

Alignment Examples



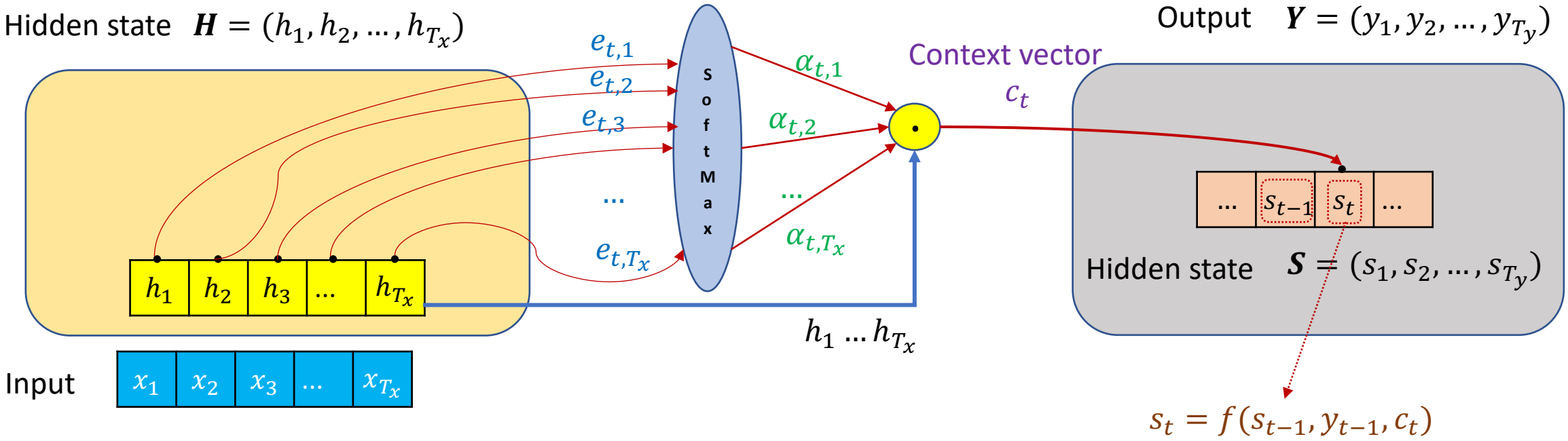
Source: Dzmitry Bahdanau et al.





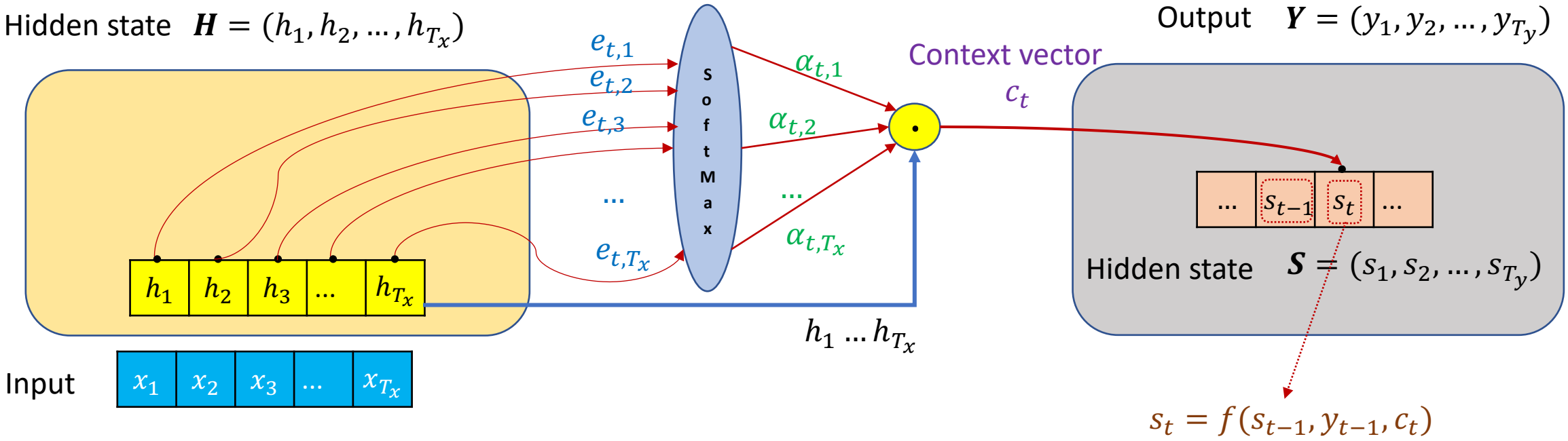
Alignment Score Functions

$$e_{t,i} = a(s_{t-1}, h_i)$$



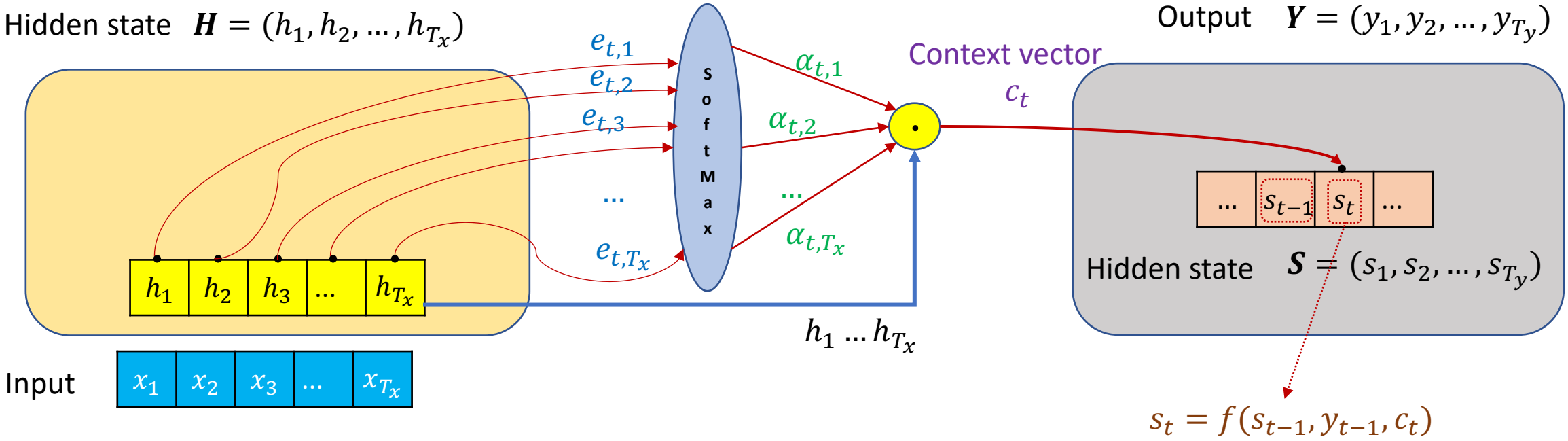
Alignment Score Functions

$$e_{t,i} = a(s_{t-1}, h_i) \qquad \alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})}$$



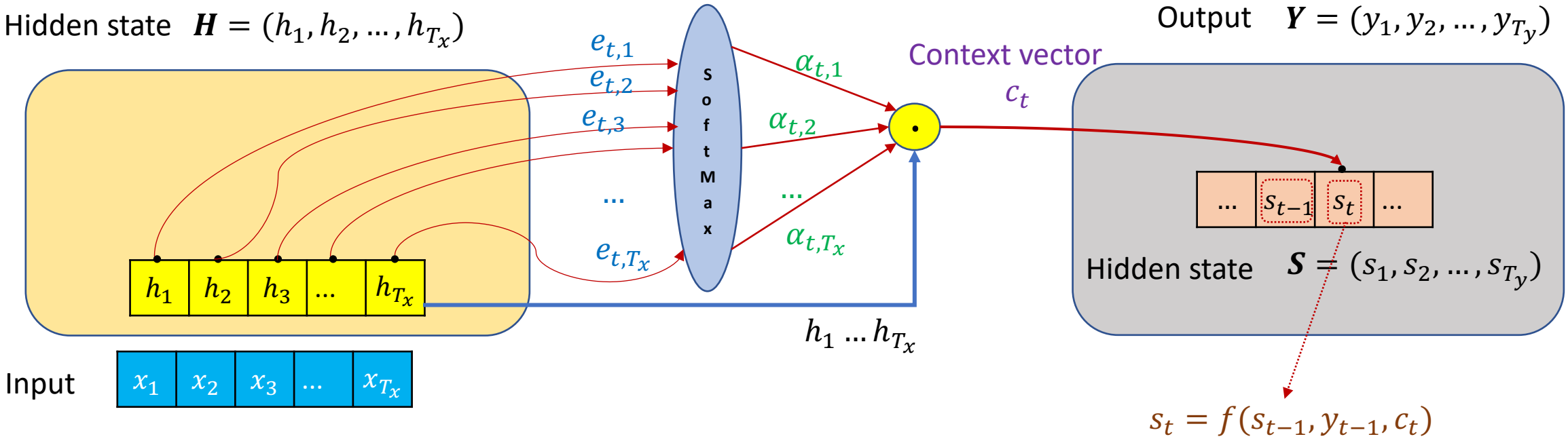
Alignment Score Functions

$$e_{t,i} = a(s_{t-1}, h_i) \qquad \alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})} \qquad c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i$$



Alignment Score Functions

$$e_{t,i} = a(s_{t-1}, h_i) \qquad \alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})} \qquad c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i \qquad s_t = f(s_{t-1}, y_{t-1}, c_t)$$



What is the Alignment Model?

- It's a **function** that computes a **scalar score** $e_{t,i}$ indicating **how well the input token at position i** matches the **decoder's need at time t**.
- **Common Form (Bahdanau / Additive)**

$$e_{t,i} = a(s_{t-1}, h_i) = v_a^\top \tanh(W_s s_{t-1} + W_h h_i)$$

  - s_{t-1}: decoder hidden state at previous step $t-1$ (vector)
  - h_i: encoder hidden state at position i (vector)
  - W_s, W_h: learned weight matrices
  - v_a: learned weight vector (projects to scalar)
  - $\tanh$: nonlinearity

•The alignment model learns **what encoder states are relevant** at each decoder step.
•Think of it as a **"relevance scorer"** between past decoder context and each input position.

Alignment Score Functions

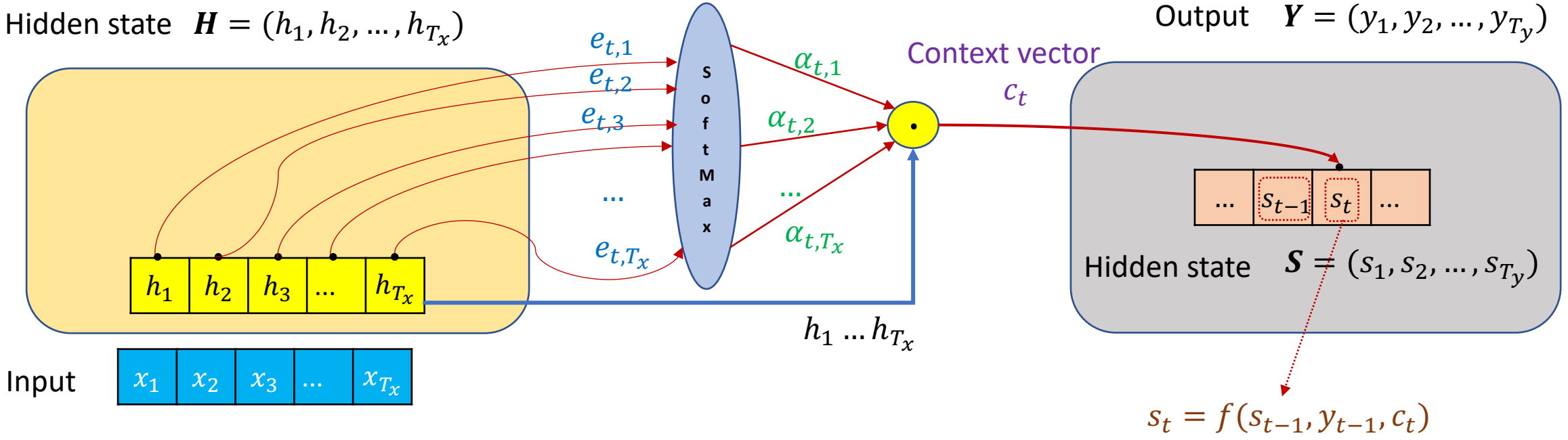
$$e_{t,i} = a(s_{t-1}, h_i) \qquad \alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T_x} \exp(e_{t,k})} \qquad c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i \qquad s_t = f(s_{t-1}, y_{t-1}, c_t)$$

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

Source: lilianweng.github.io

Attention Mechanism Challenges

- **Complexity Analysis**
  - Quadratic complexity: $O(n^2)$ for sequences of length n.
  - For input length n and output length m, complexity is $O(nm)$.
  - Inefficient for long sequences - high memory and time cost.
- **Efficient Alternatives**
  - Innovations like multi-head, sparse, and local attention.



Sparse Attention (Common in Self-Attention)

- **What is Sparse Attention?**
  - Instead of attending to **all tokens**, each token attends to **a small subset**.
  - Reduces computation from $O(n^2)$ to $O(n \cdot k)$, where $k \ll n$.
- **Benefits**
  - Works on **long sequences** (e.g. 10,000+ tokens)
  - Efficient for tasks like document summarization or genomic data

Token	Attends to...
t_1	t_1, t_2, t_5
t_2	t_1, t_2, t_3
t_5	t_2, t_4, t_5

Each token attends to k=3 random tokens across the sequence

Local Attention (Common in Self-Attention)

- **What is Local Attention?**
  - Each token attends only to **neighboring tokens** (within a fixed window).
  - Example: Attend to ±2 tokens around each position.
  - Useful when **local patterns** matter (e.g., speech, time series)
- **Benefits**
  - Faster than global attention
  - Low memory cost
  - Captures **local structure well**

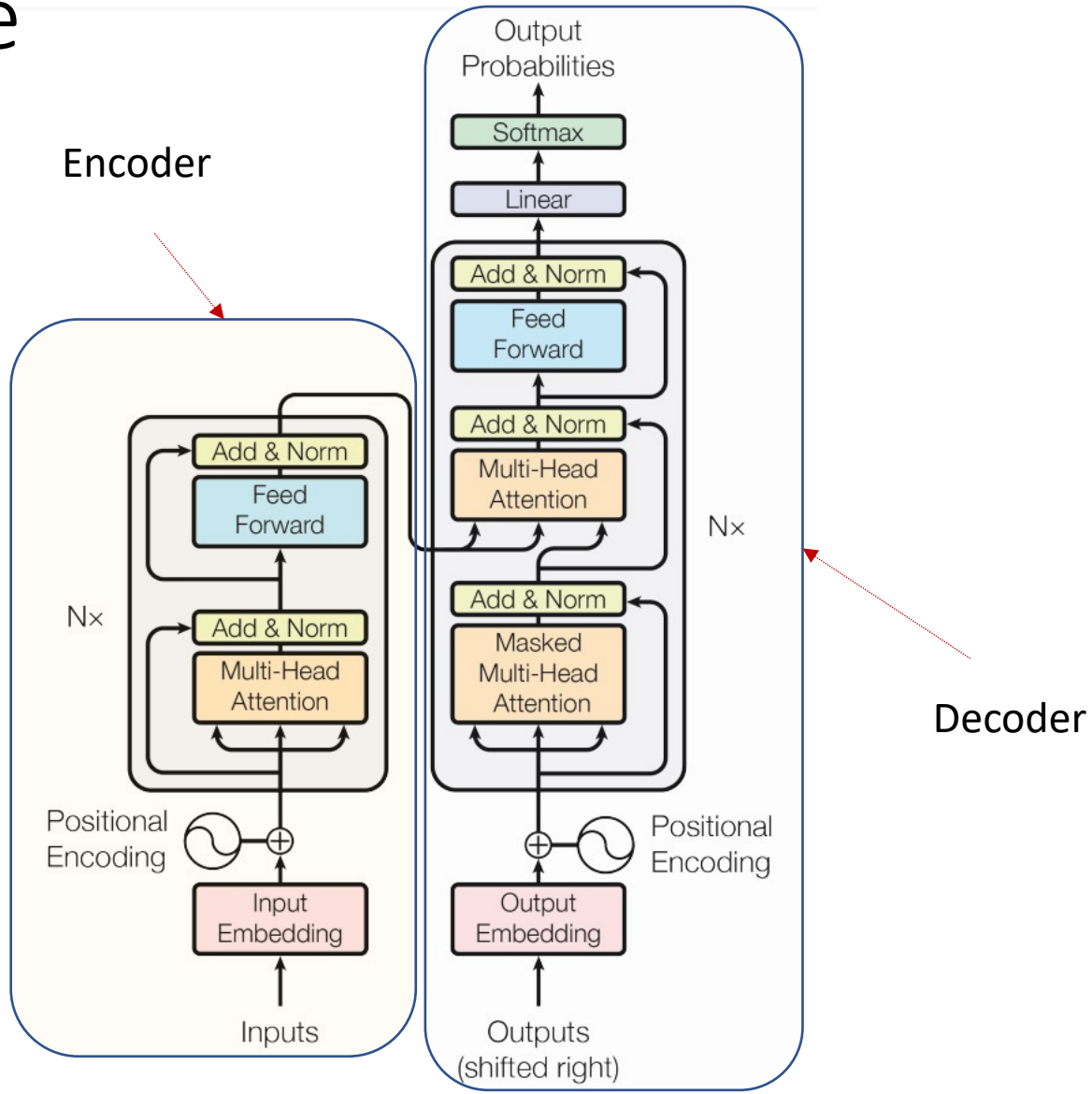
Token	Attends to...
t_4	t_2, t_3, t_4, t_5, t_6
t_5	t_3, t_4, t_5, t_6, t_7

Each token attends to the 2 tokens on the right and left

Transformers

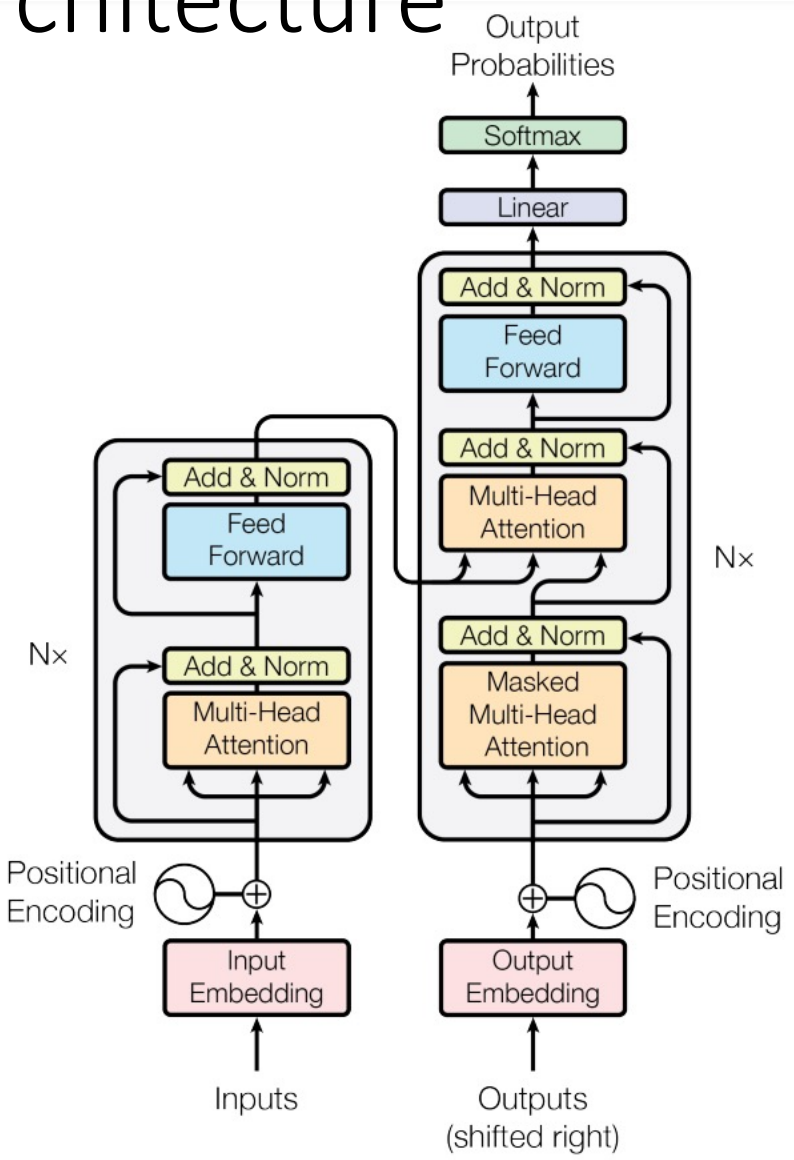
The Transformer Architecture

- **Parallelization**
  - Self-attention allows simultaneous processing of all tokens
  - Faster than RNNs — no sequential dependencies
- **Scalability**
  - Handles sequences with thousands of tokens
  - Efficient for long text, code, or biological data
  - Improved further with Flash, Sparse, and Local Attention
- **Transfer Learning**
  - Pretrained models like BERT, GPT, and T5 generalize well
  - Fine-tuning enables strong performance on many NLP tasks:
    - Sentiment analysis
    - Question answering
    - Text classification
    - Translation



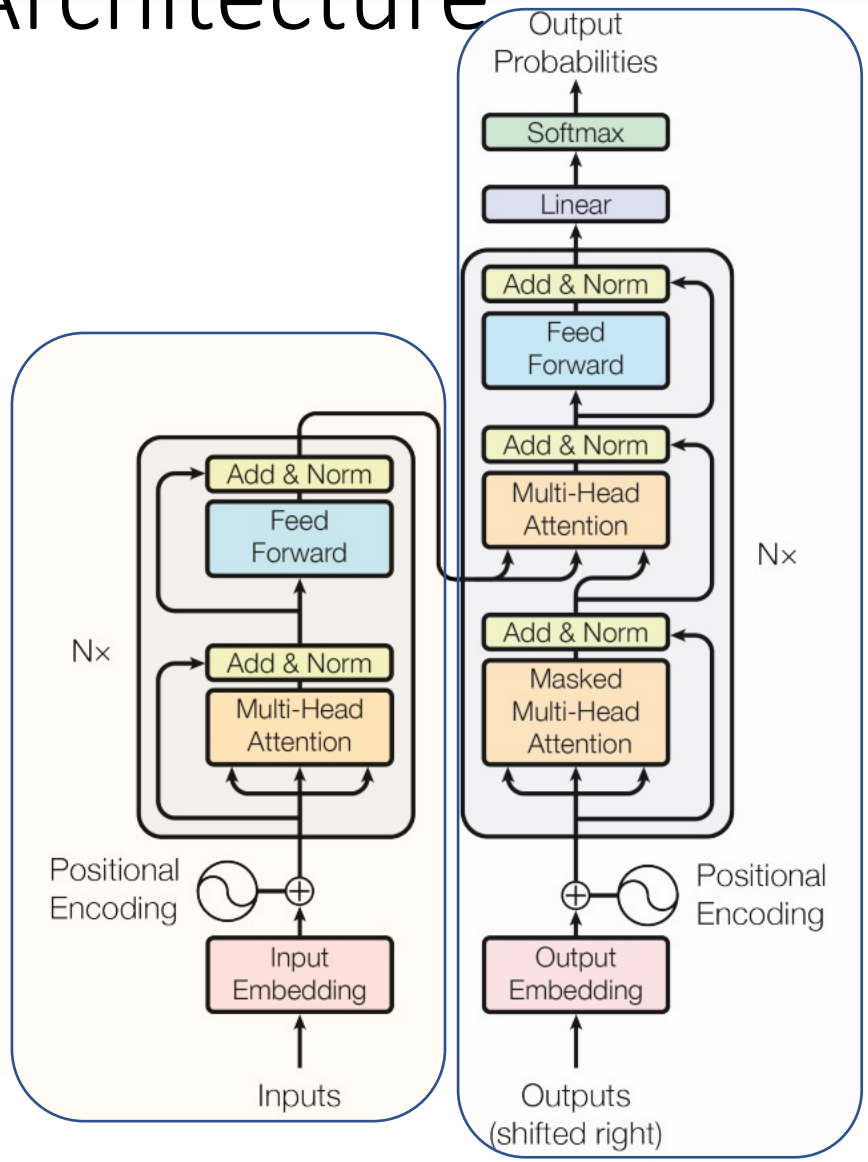
Source: "Attention is all you need", Vaswani et al.

The Transformer Architecture



Source: "Attention is all you need", Vaswani et al.

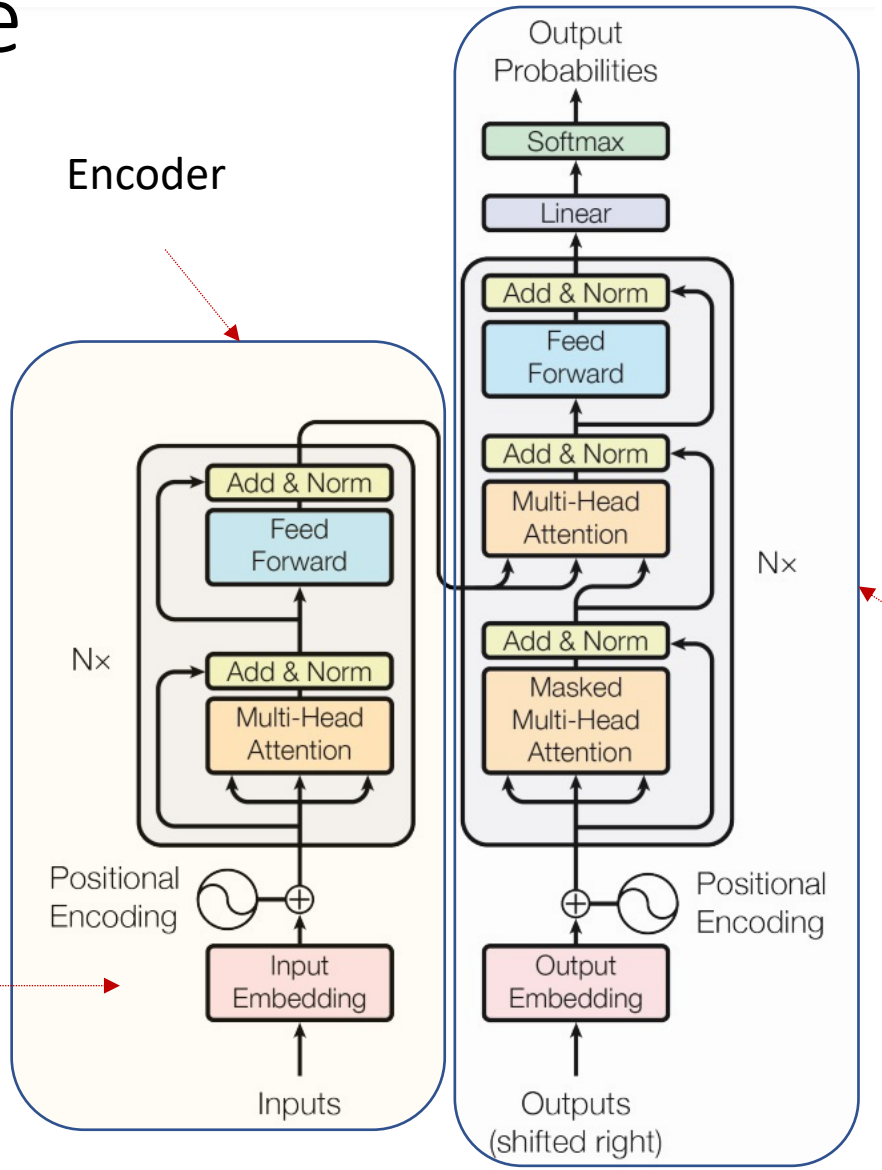
The Transformer Architecture



Source: "Attention is all you need", Vaswani et al.

The Transformer Architecture

- Encoder-Decoder Structure



Encoder

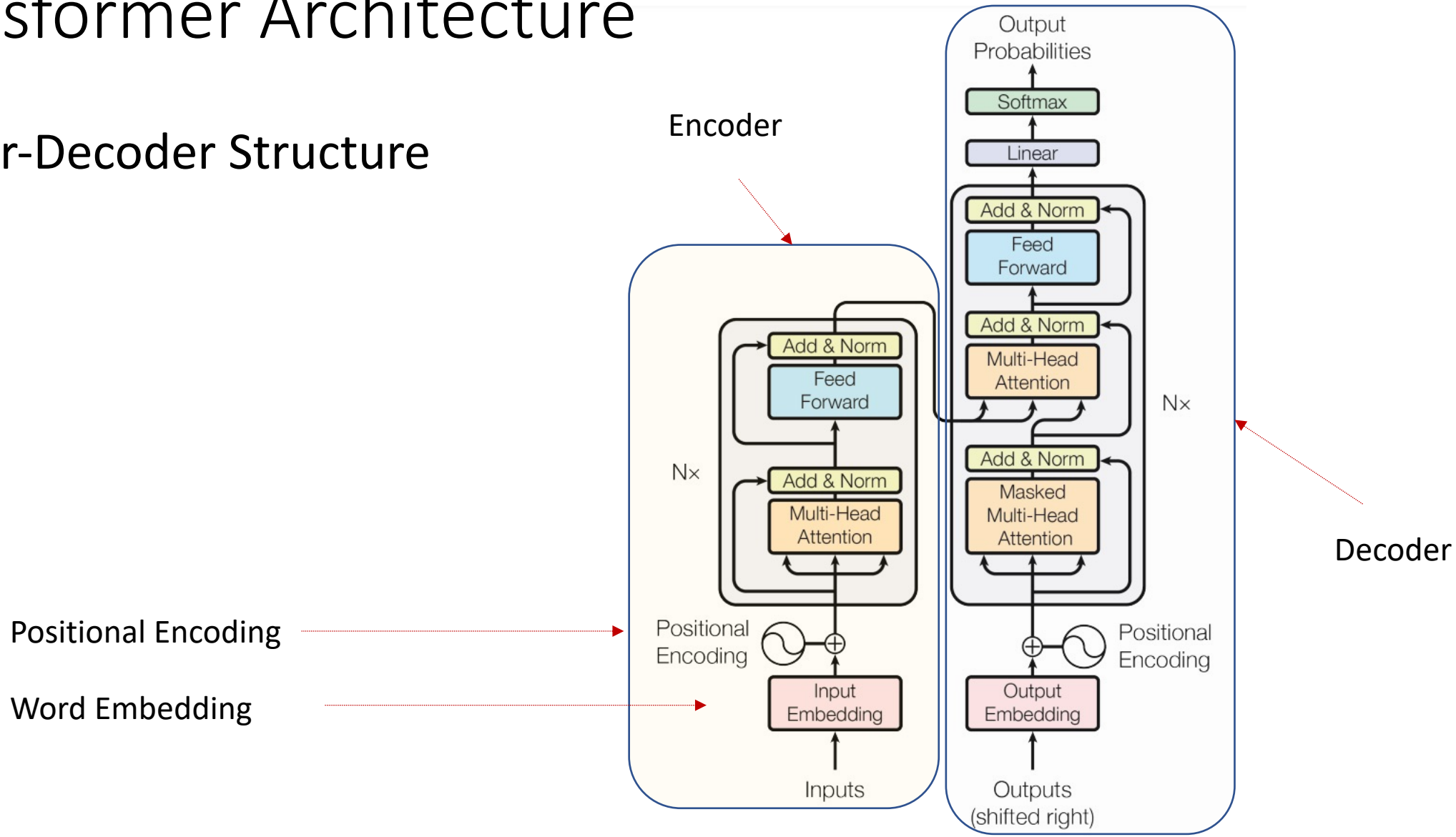
Decoder

Word Embedding

Source: "Attention is all you need", Vaswani et al.

The Transformer Architecture

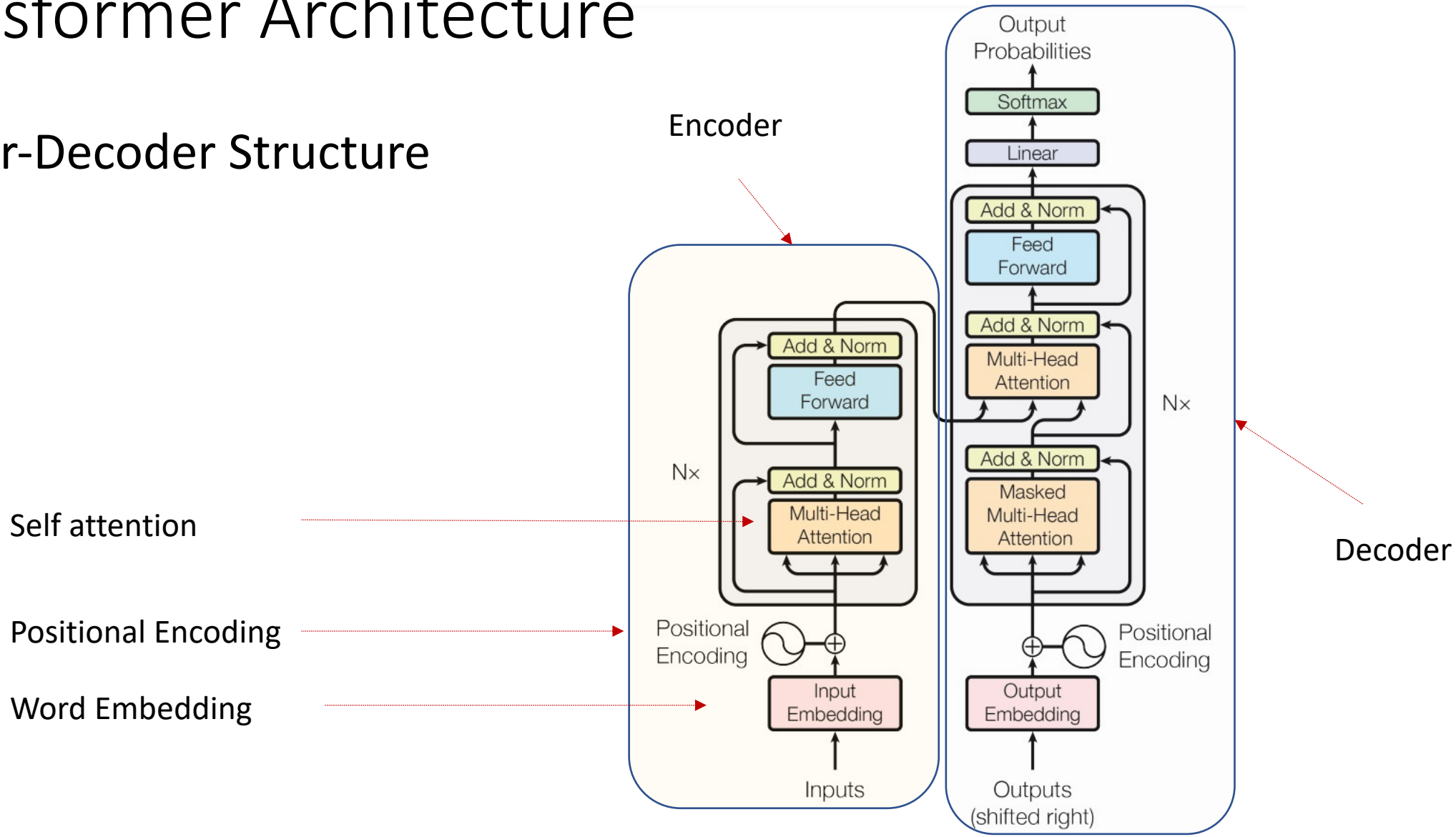
- Encoder-Decoder Structure



Source: "Attention is all you need", Vaswani et al.

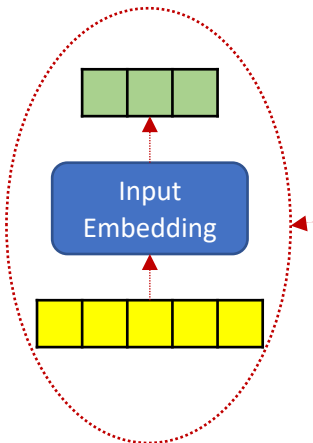
The Transformer Architecture

- Encoder-Decoder Structure



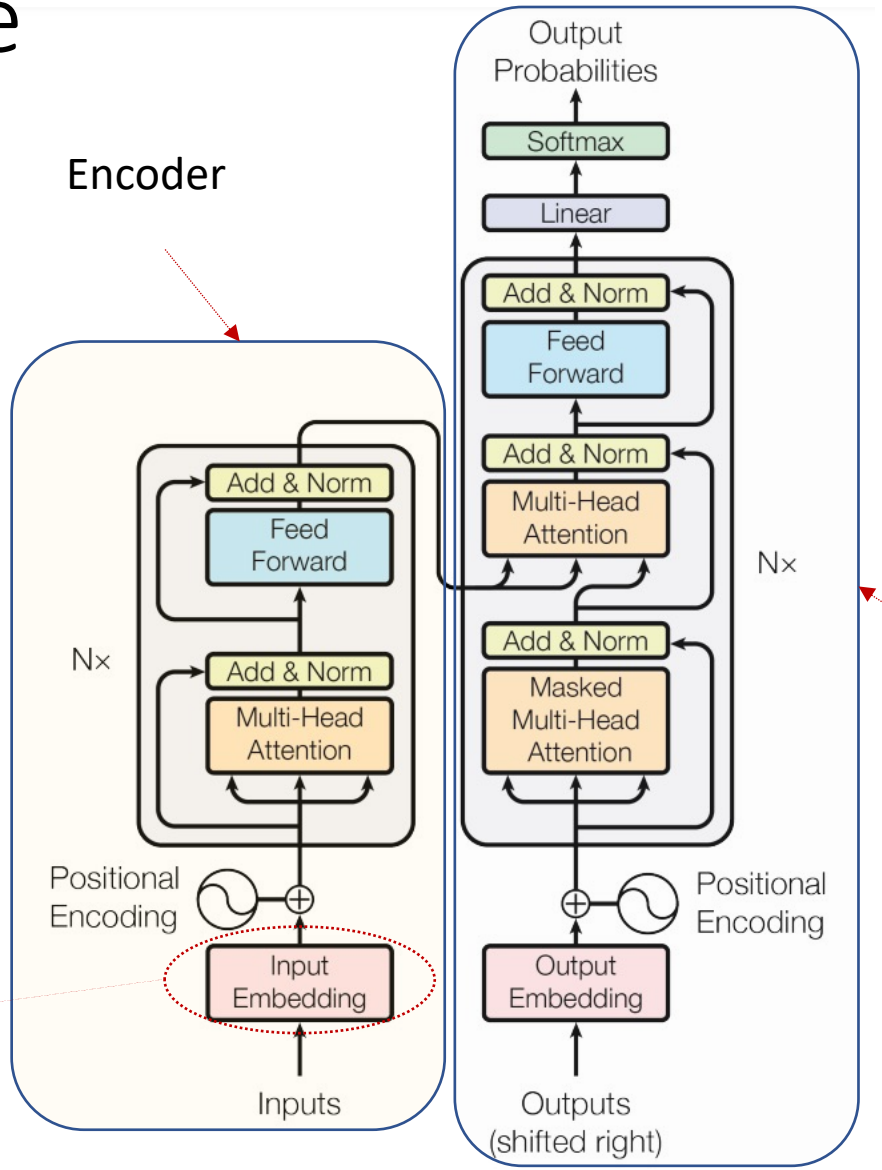
Source: "Attention is all you need", Vaswani et al.

The Transformer Architecture



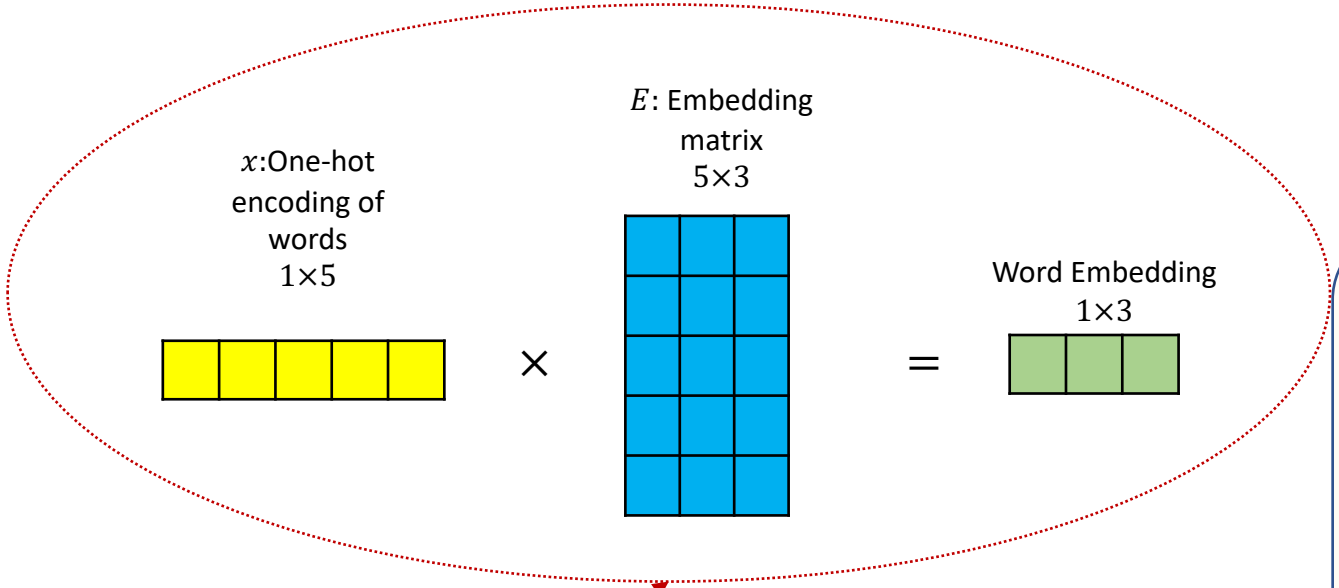
Encoder

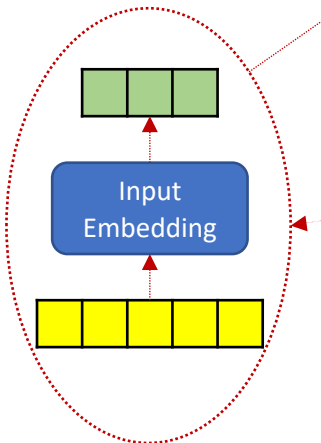
Decoder



Source: "Attention is all you need", Vaswani et al.

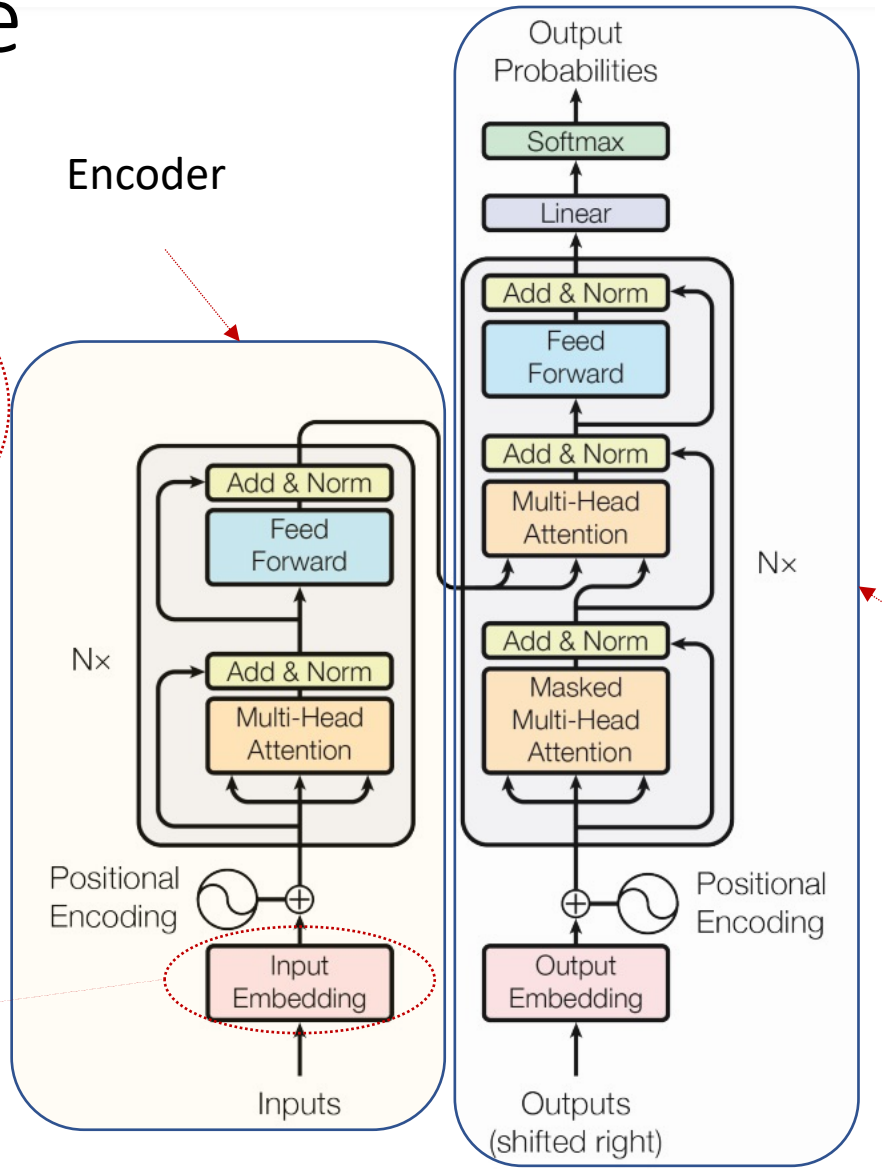
The Transformer Architecture





Encoder

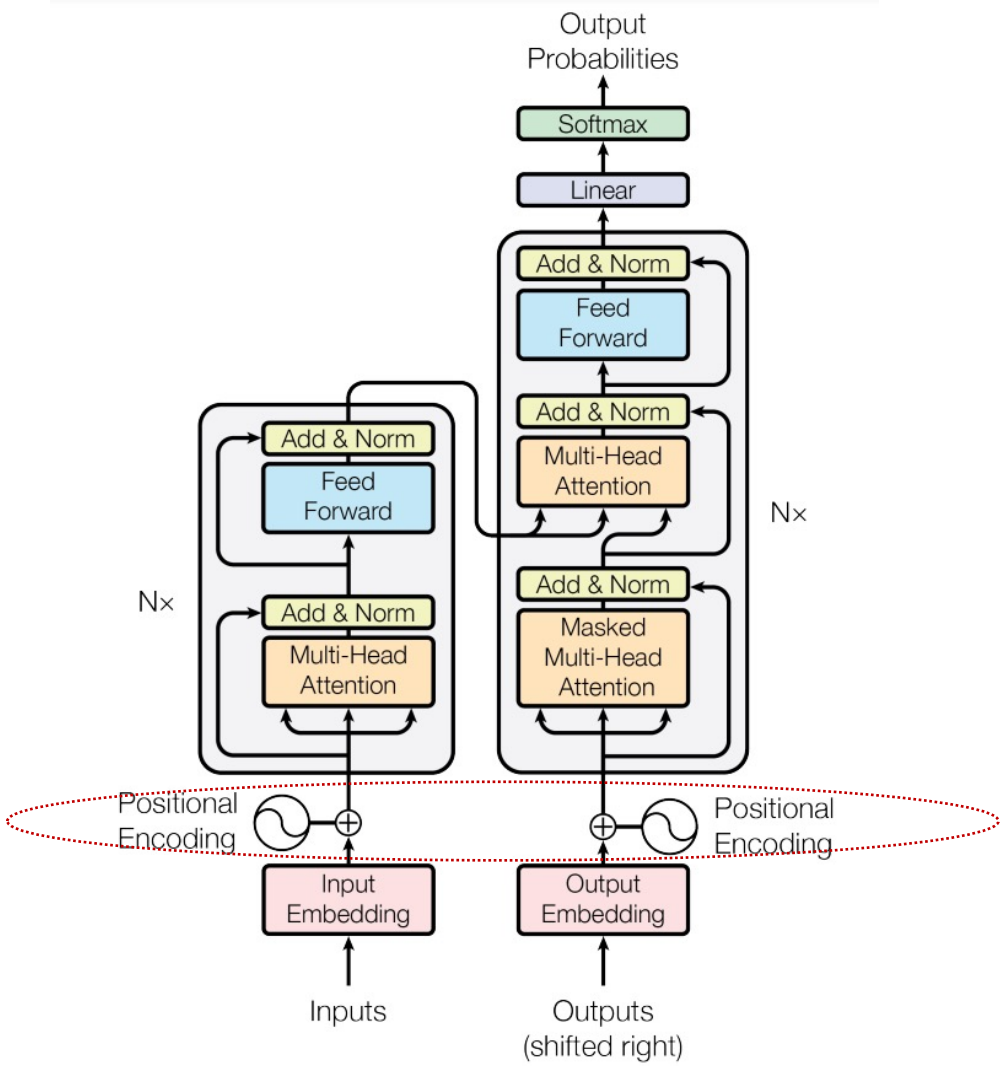
Decoder



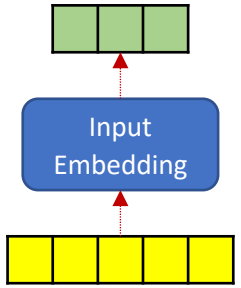
Source: "Attention is all you need", Vaswani et al.

Positional Encoding

- Self-attention (described later) does not consider the position of the tokens.
  - Let's add the positional encodings to the input embeddings .
- They can be a function or learned values

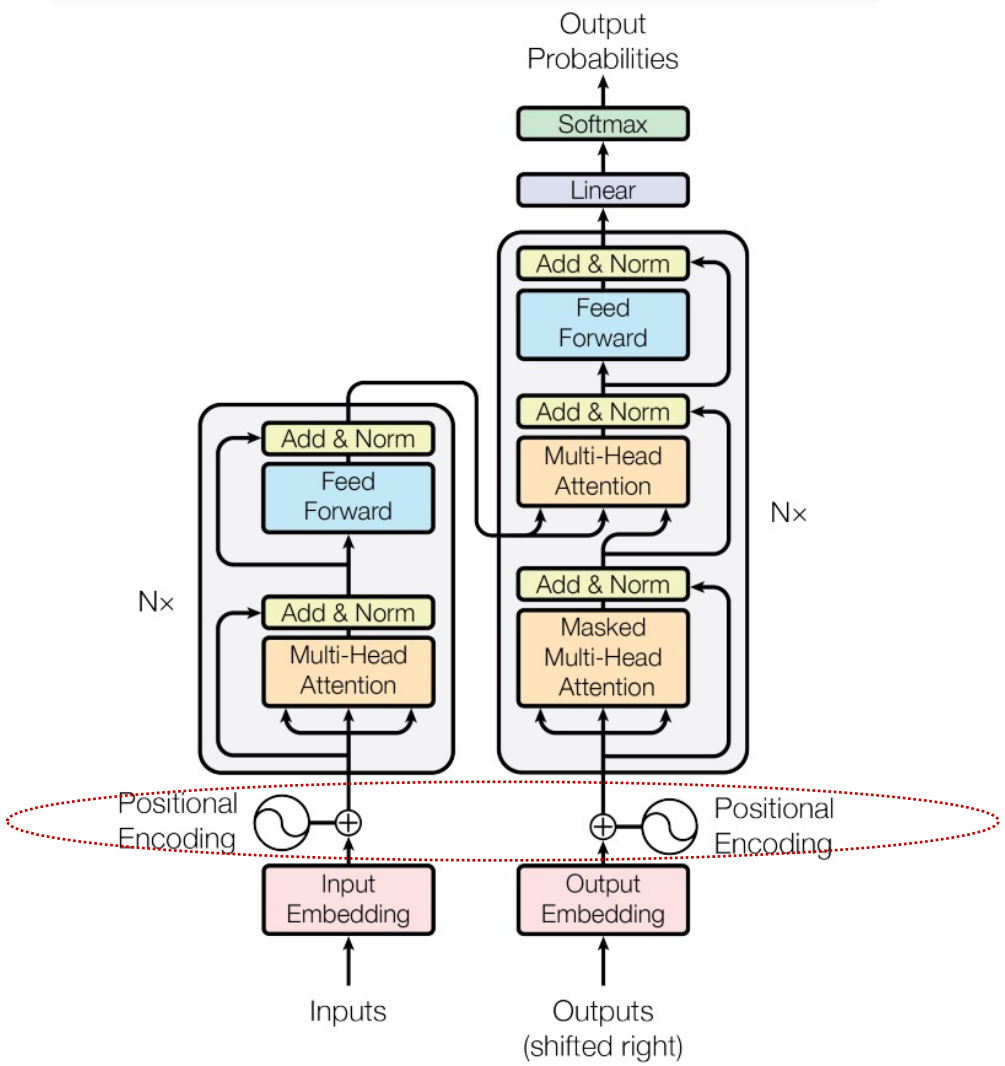


Positional Encoding



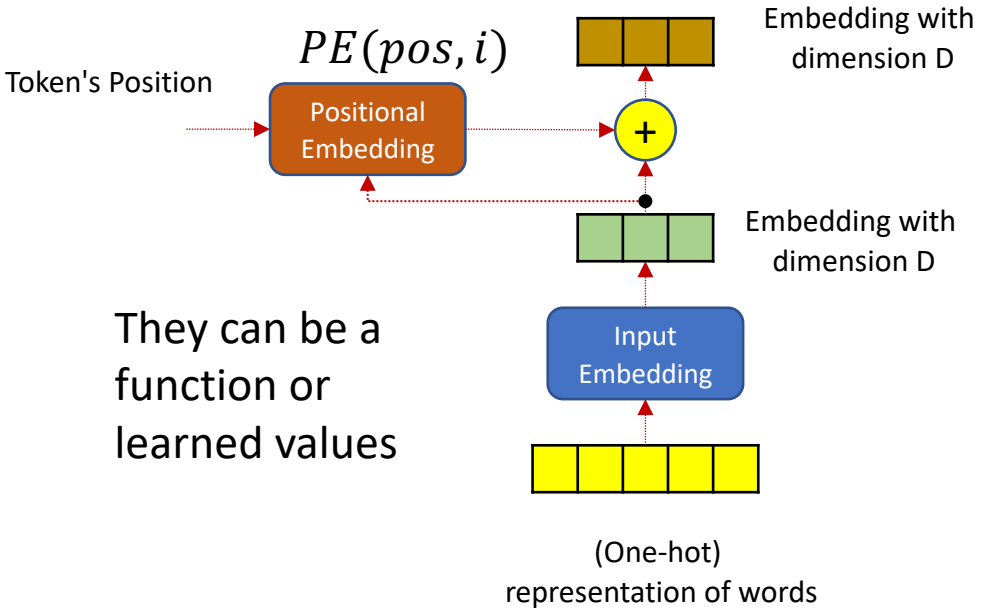
(One-hot)
representation of words

The man and the cat sat on the mat



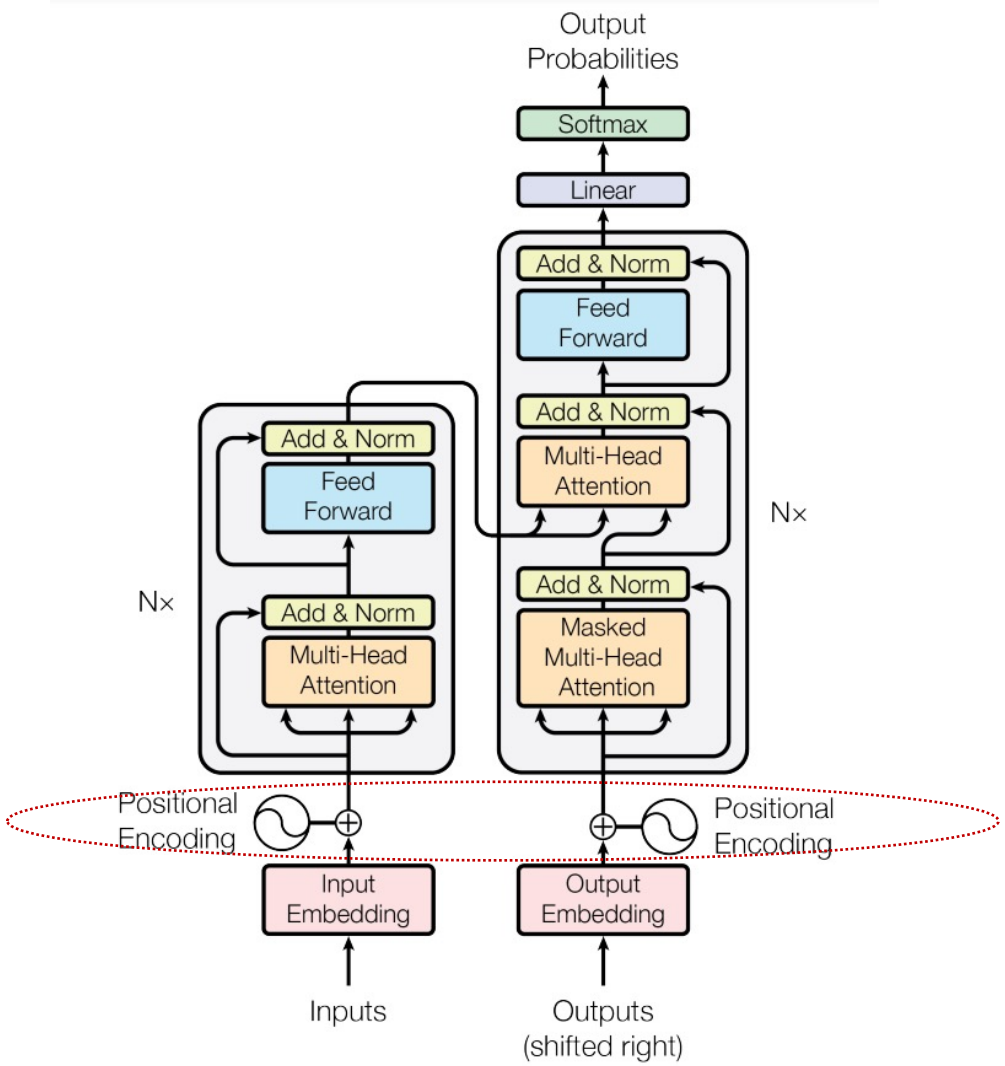
Positional Encoding

pos is the position in the sequence, and *i* is the dimension in the encoding space



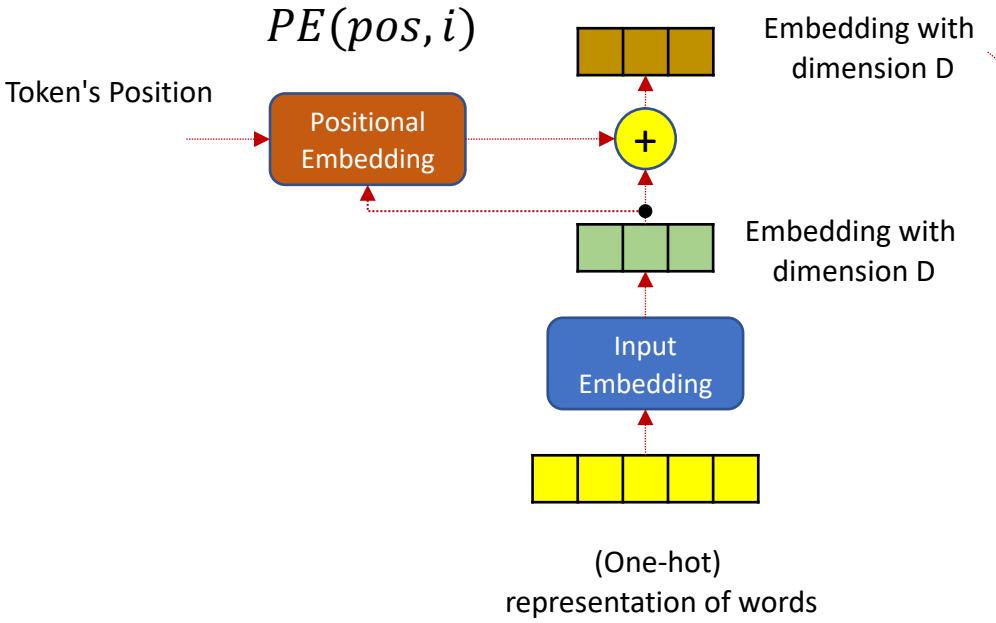
They can be a function or learned values

The man and the cat sat on the mat



Positional Encoding

pos is the position in the sequence, and *i* is the **dimension index** in the encoding space

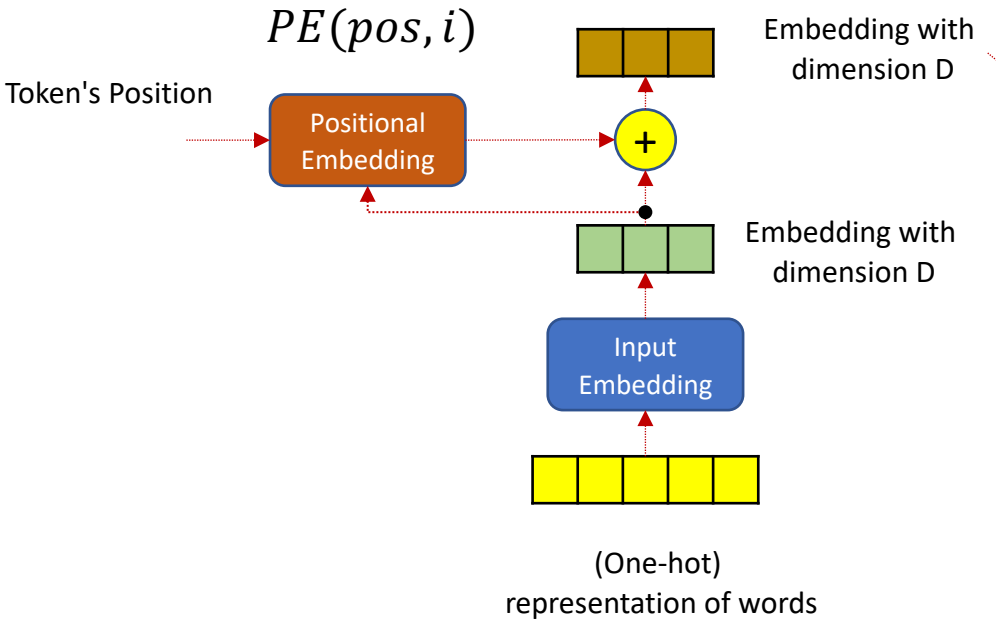


The same word at different positions will have different encodings

*"The white **cat** and the brown **cat** sat on the mat"*

Positional Encoding

pos is the position in the sequence, and *i* is the **dimension index** in the encoding space



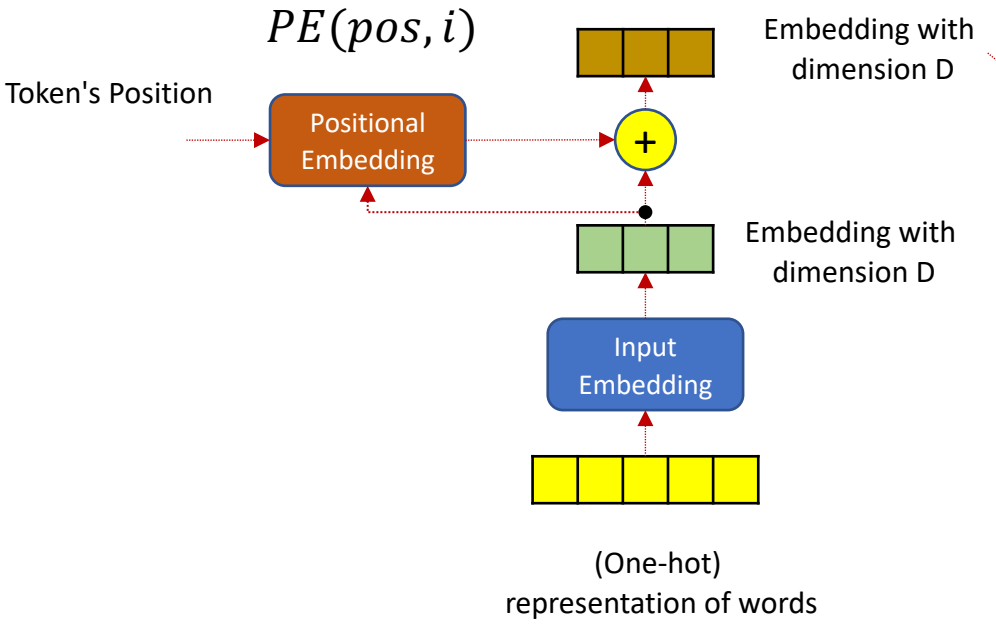
The same word at different positions will have different encodings

*"The white **cat** and the brown **cat** sat on the mat"*

Why not $PE(pos)$, instead of $PE(pos, i)$?

Positional Encoding

pos is the position in the sequence, and *i* is the **dimension index** in the encoding space



The same word at different positions will have different encodings

*"The white **cat** and the brown **cat** sat on the mat"*

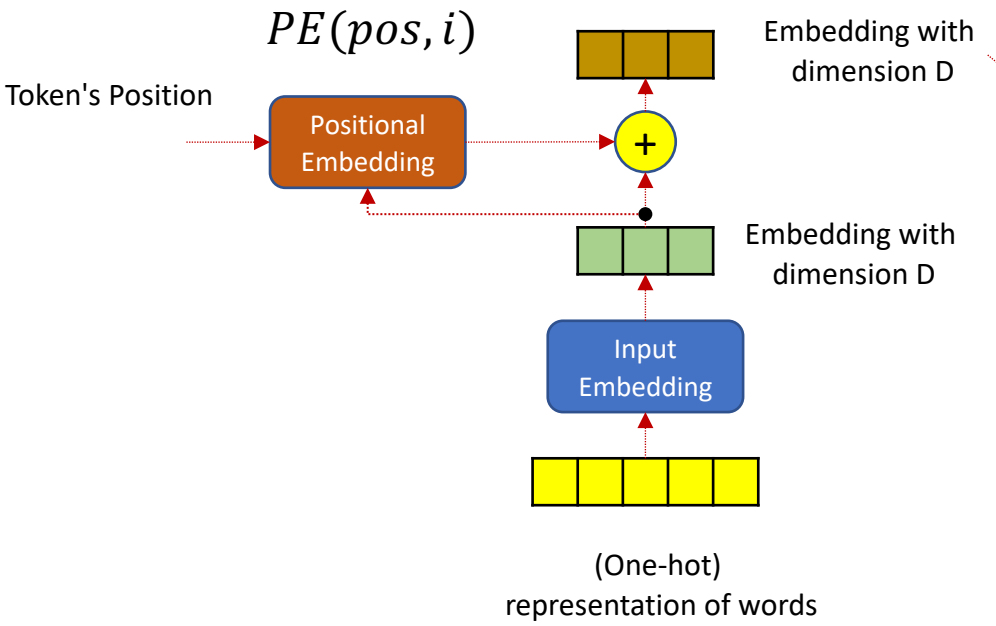
Why not $PE(pos)$, instead of $PE(pos, i)$?

Because **positional encoding is a vector**, not a scalar.

PE(pos,i)=the i-th component of the vector for position pos

Positional Encoding

pos is the position in the sequence, and *i* is the dimension in the encoding space



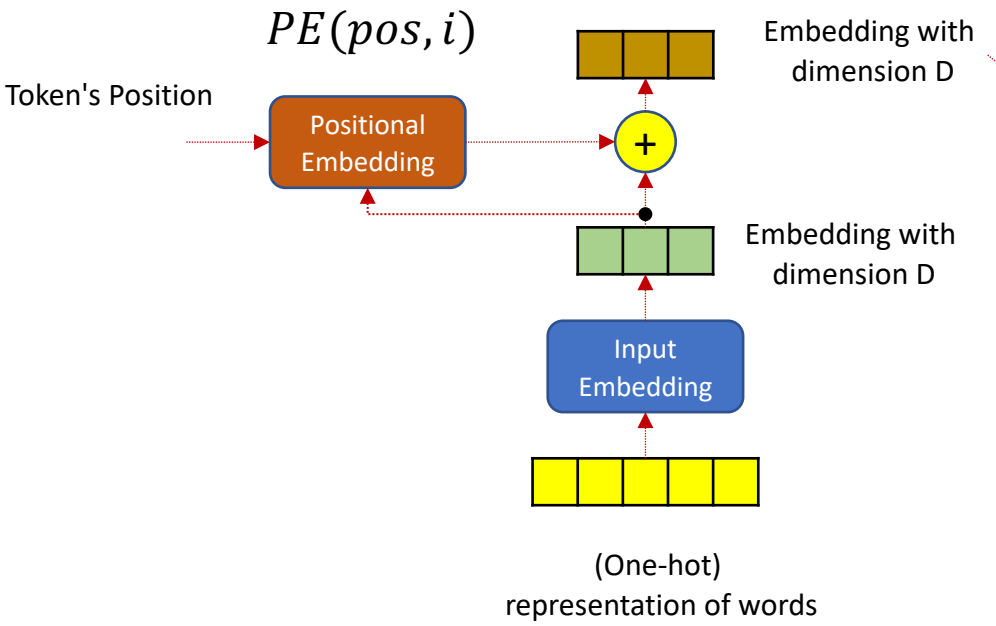
The same word at different positions will have different encodings

*"The white **cat** and the brown **cat** sat on the mat"*

Question: What function should we pick for $PE(pos, i)$?

Positional Encoding

pos is the position in the sequence, and *i* is the dimension in the encoding space



The same word at different positions will have different encodings

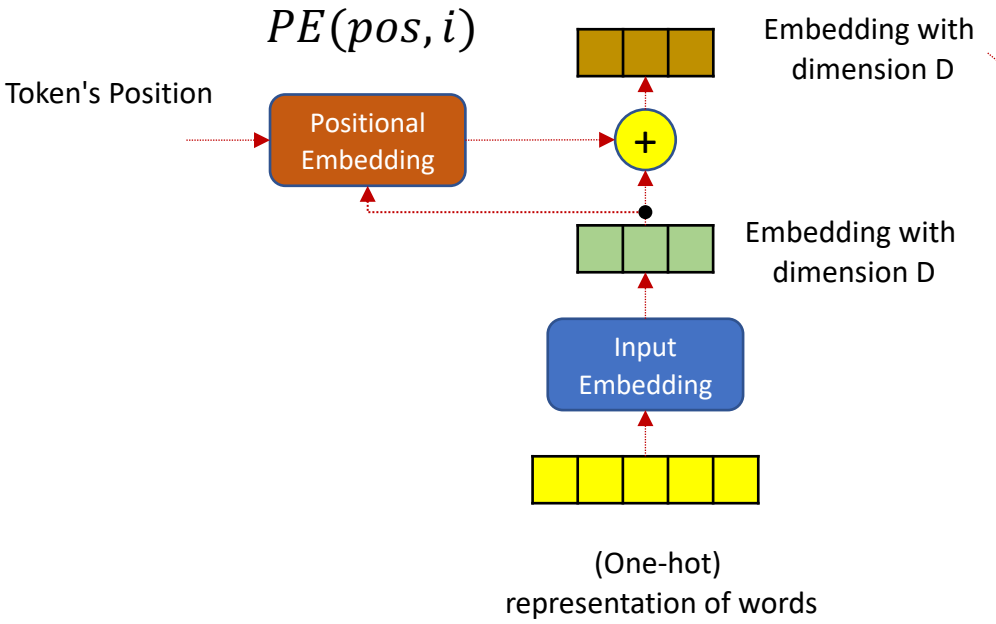
*"The white **cat** and the brown **cat** sat on the mat"*

Question: What function should we pick for $PE(pos, i)$?

Monotonic function

Positional Encoding

pos is the position in the sequence, and *i* is the dimension in the encoding space



The same word at different positions will have different encodings

*"The white **cat** and the brown **cat** sat on the mat"*

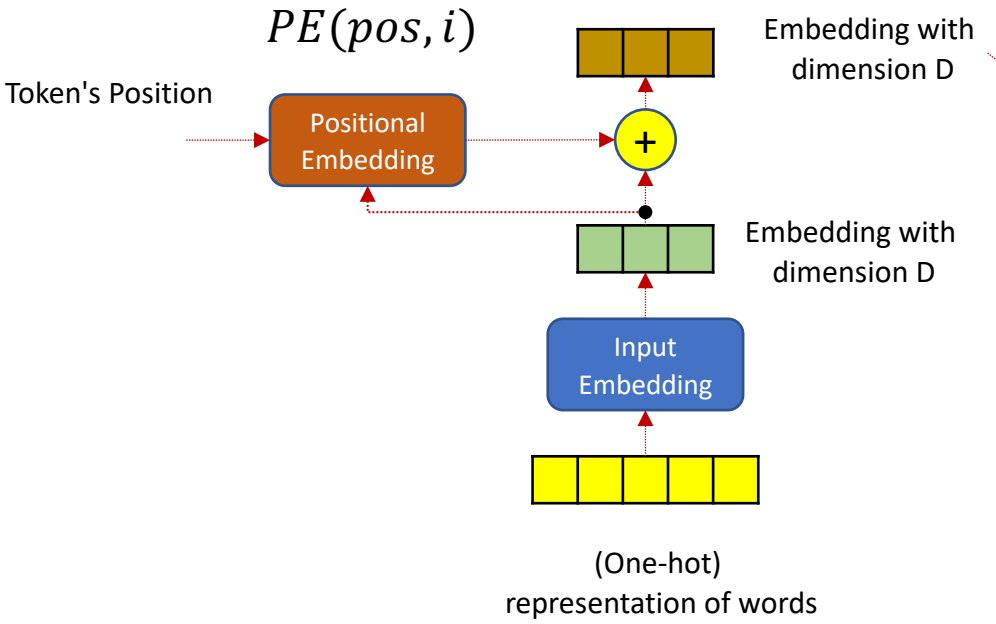
Question: What function should we pick for $PE(pos, i)$?

Monotonic function

Periodic function

Positional Encoding

pos is the position in the sequence, and *i* is the dimension in the encoding space



The same word at different positions will have different encodings

*"The white **cat** and the brown **cat** sat on the mat"*

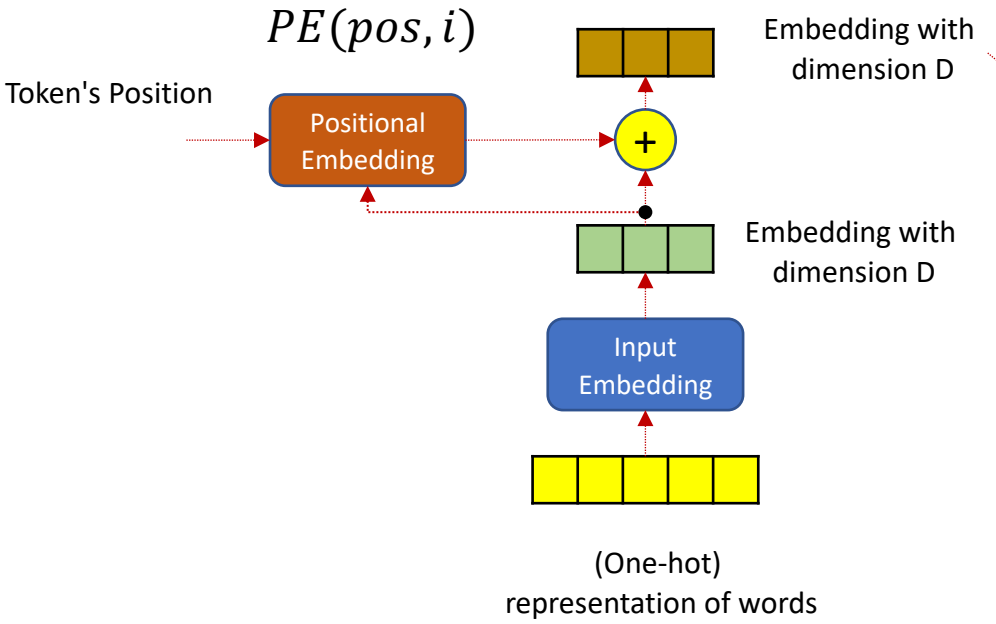
Question: What function should we pick for $PE(pos, i)$?

Monotonic function: exploding gradients. Huge values (poor numerical stability).

Periodic function

Positional Encoding

pos is the position in the sequence, and *i* is the dimension in the encoding space



The same word at different positions will have different encodings

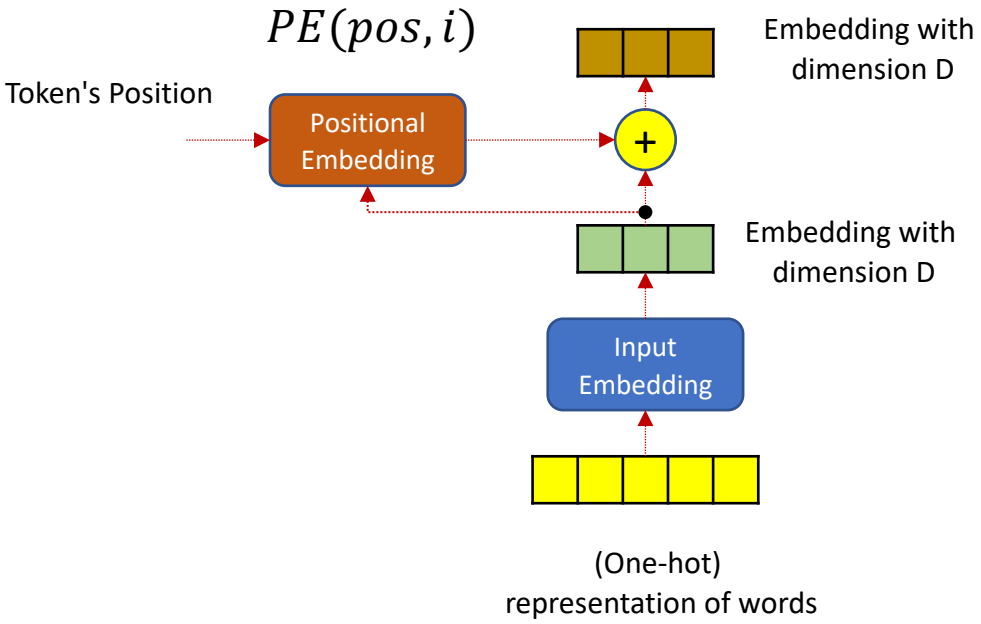
*"The white **cat** and the brown **cat** sat on the mat"*

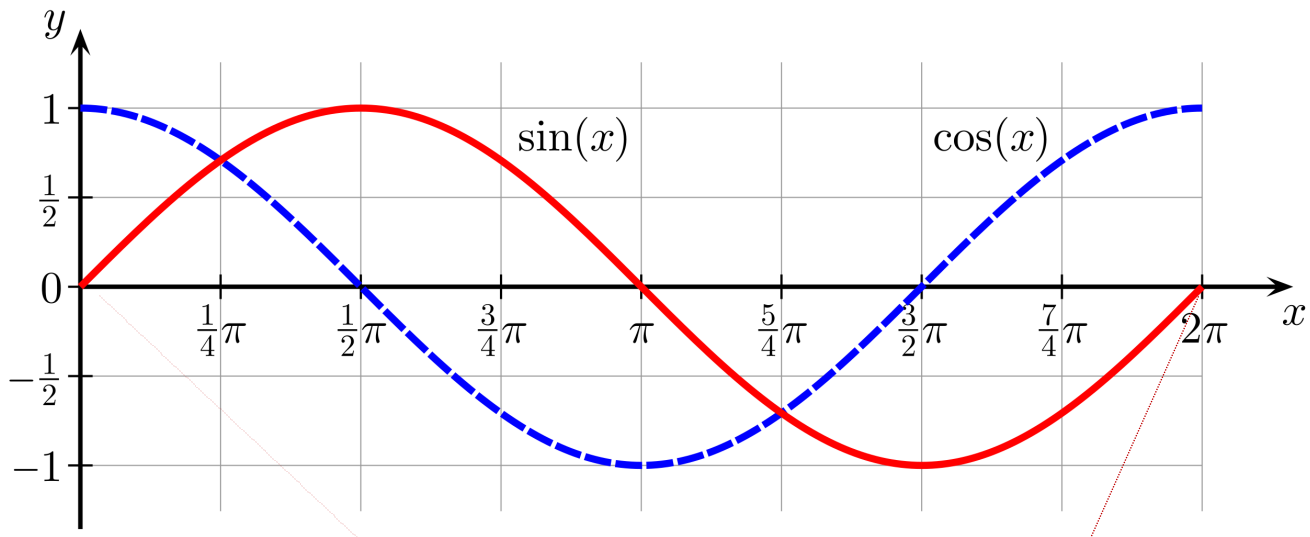
Question: What function should we pick for $PE(pos, i)$?

Monotonic function: exploding gradients. Huge values (poor numerical stability).

Periodic function: the embedding of different positions would be the same!

Positional Encoding



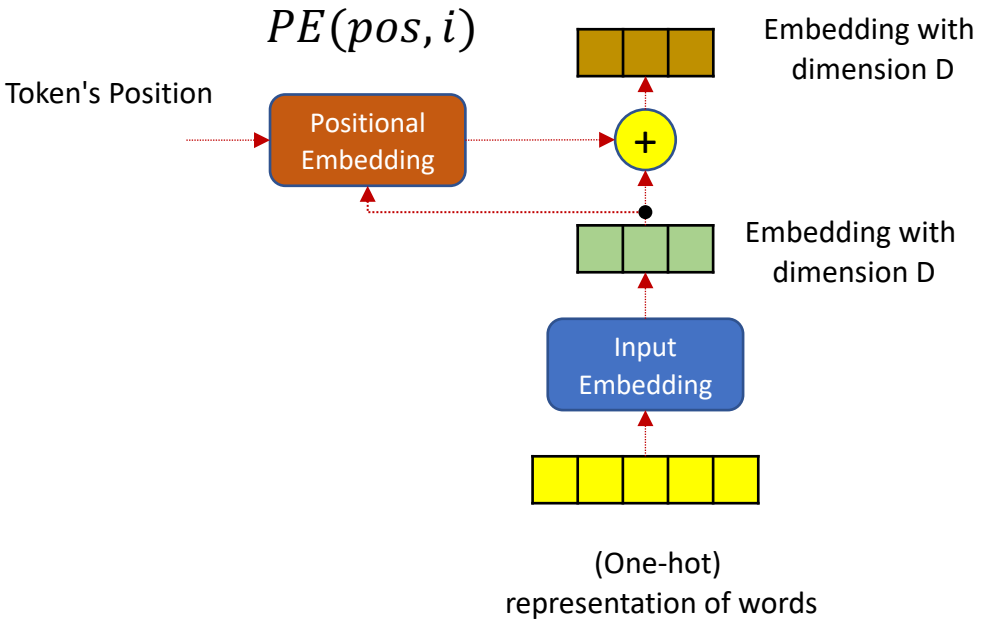


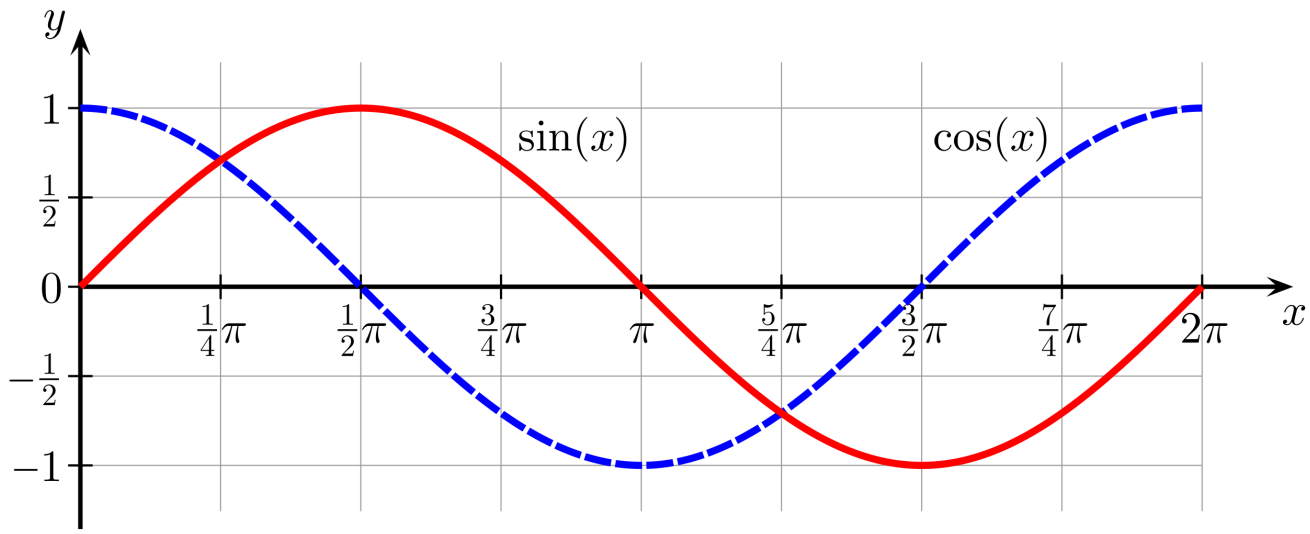
Question: What function should we pick for $PE(pos)$?

Monotonic function: exploding gradients. Huge values (poor numerical stability).

Periodic function: the embedding of different positions would be the same!

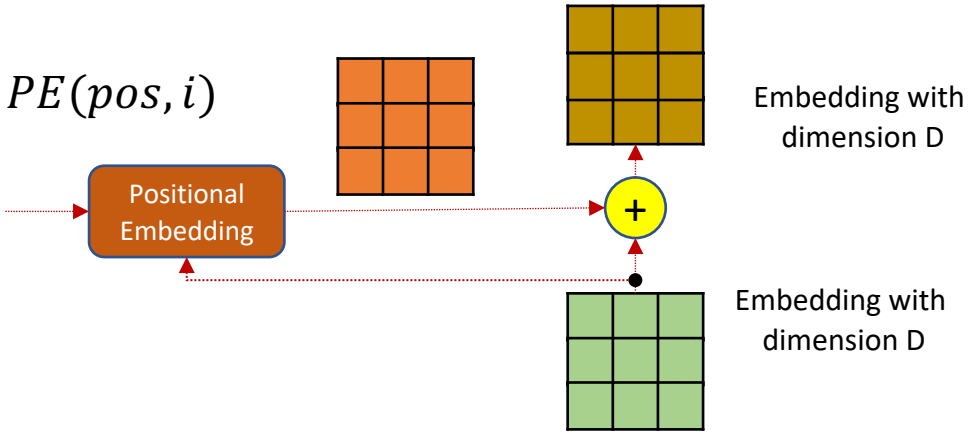
Positional Encoding





- The Transformer uses **many frequencies** across dimensions.
- **Result: No two positions have the same encoding, unless the sequence is extremely long (beyond model capacity).**
  - Each **positional vector** is **uniquely determined** by its combination of sinusoids.
  - Like a **Fourier fingerprint** for each position.

Positional Encoding As a Matrix



Position of the token

i[th] dimension

$0 \leq i \leq \frac{D}{2} - 1$

Length of embedding i.e. each token is a vector of D numbers

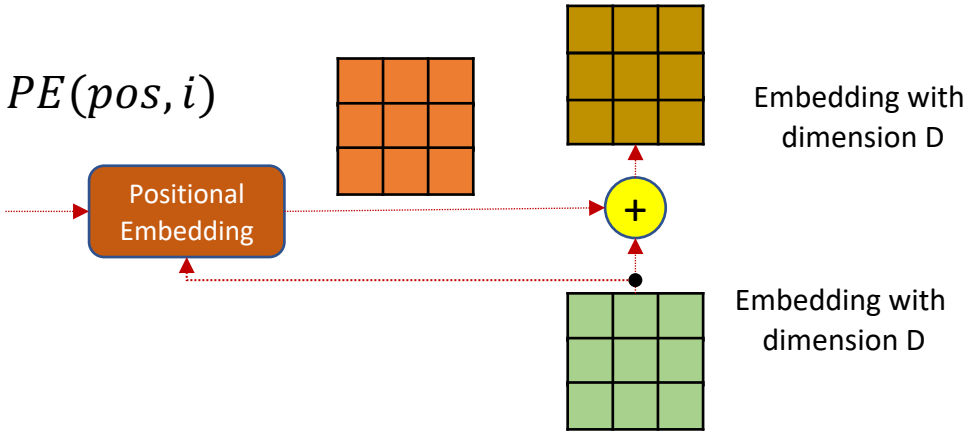
$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{n^{2i/D}}\right) \end{cases}$$

$$n = 10000$$

Pos	Word	i=0 $\sin\left(\frac{Pos}{1}\right)$	i=0 $\cos\left(\frac{pos}{100^{2/4}}\right)$	i=1 $\sin\left(\frac{pos}{100^{4/4}}\right)$	i=1 $\cos\left(\frac{pos}{100^{6/4}}\right)$
0	The	0.00	1.00	0.00	1.00
1	cat	0.84	0.54	0.10	1.00
2	sat	0.91	-0.42	0.20	0.98
3	on	0.14	-0.99	0.30	0.96
4	the	-0.76	-0.65	0.39	0.92
5	mat	-0.96	0.28	0.48	0.88

Positional embeddings with D=4 and n=100

Positional Encoding As a Matrix

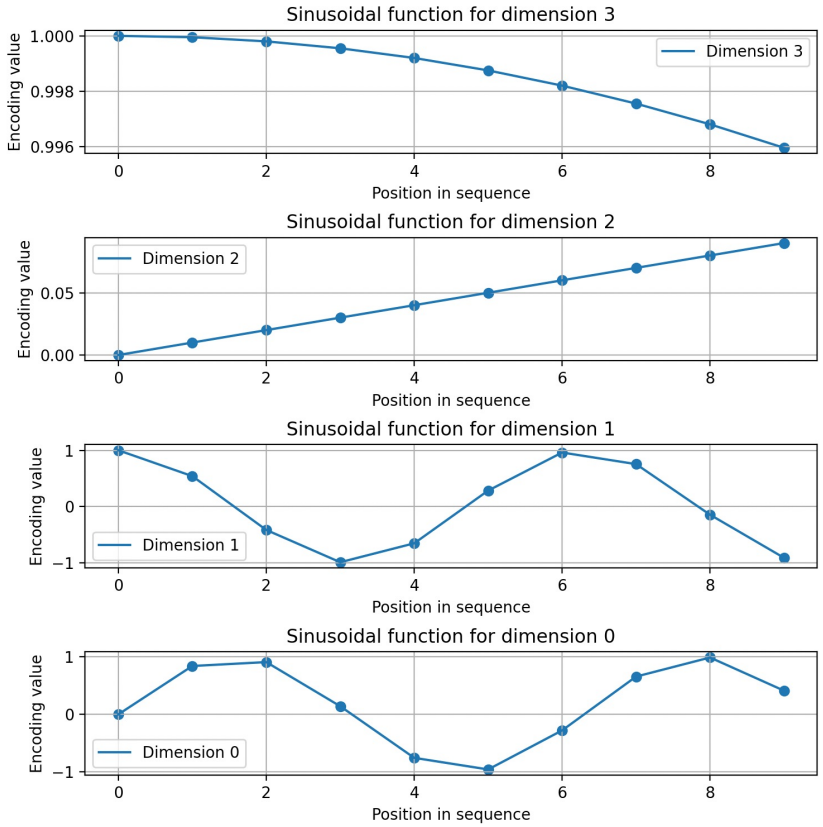


Position of the token

i[th] dimension

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{n^{2i/D}}\right) \end{cases}$$

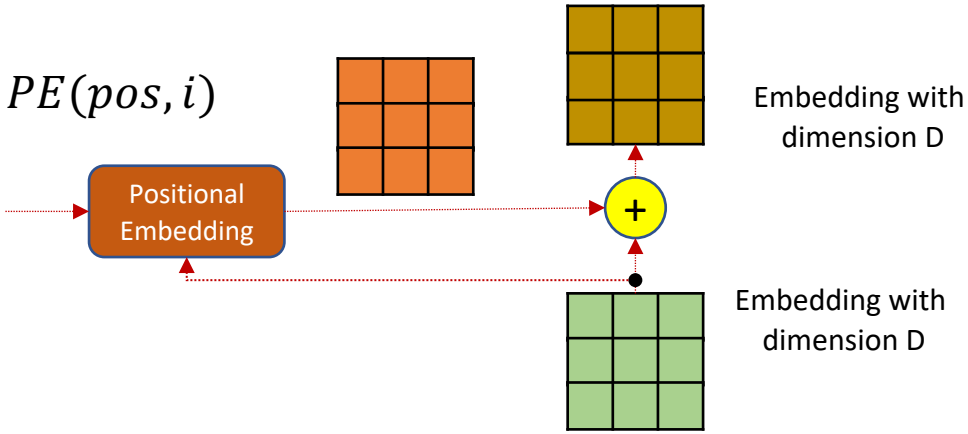
$$n = 10000$$

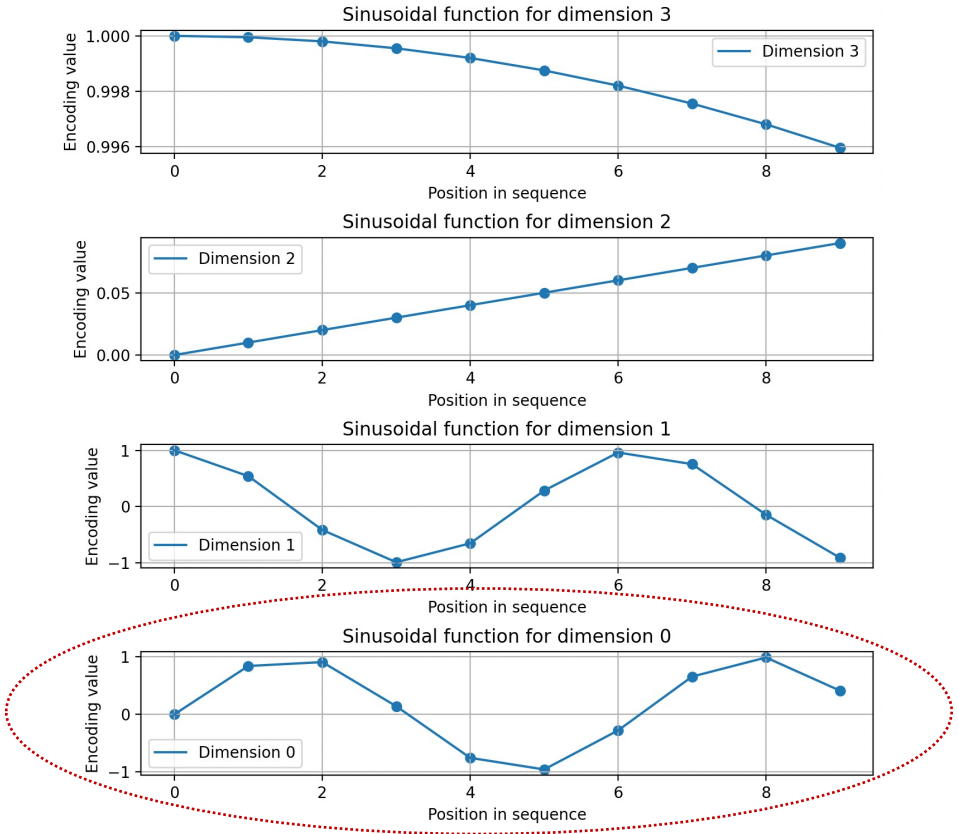


Pos	Word	i=0 $sin\left(\frac{Pos}{1}\right)$	i=0 $\cos\left(\frac{pos}{100^{2/4}}\right)$	i=1 $\sin\left(\frac{pos}{100^{4/4}}\right)$	i=1 $\cos\left(\frac{pos}{100^{6/4}}\right)$
0	The	0.00	1.00	0.00	1.00
1	cat	0.84	0.54	0.10	1.00
2	sat	0.91	-0.42	0.20	0.98
3	on	0.14	-0.99	0.30	0.96
4	the	-0.76	-0.65	0.39	0.92
5	mat	-0.96	0.28	0.48	0.88

Positional embeddings with D=4 and n=100

Positional Encoding As a Matrix





Position of the token

i[th] dimension

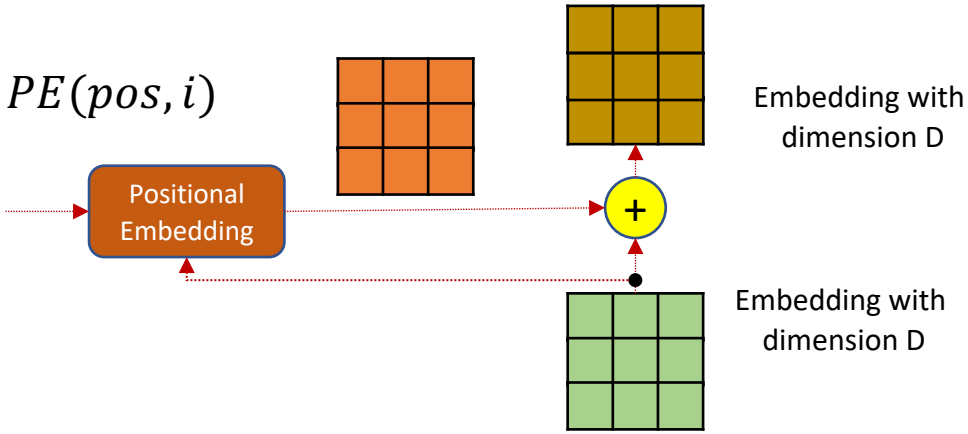
$$\begin{cases} PE(pos, 2i) = \sin\left(\dfrac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\dfrac{pos}{n^{2i/D}}\right) \end{cases}$$

$$n = 10000$$

Pos	Word	i=0 $\sin\left(\frac{Pos}{1}\right)$	i=0 $\cos\left(\frac{pos}{100^{2/4}}\right)$	i=1 $\sin\left(\frac{pos}{100^{4/4}}\right)$	i=1 $\cos\left(\frac{pos}{100^{6/4}}\right)$
0	The	0.00	1.00	0.00	1.00
1	cat	0.84	0.54	0.10	1.00
2	sat	0.91	-0.42	0.20	0.98
3	on	0.14	-0.99	0.30	0.96
4	the	-0.76	-0.65	0.39	0.92
5	mat	-0.96	0.28	0.48	0.88

Positional embeddings with D=4 and n=100

Positional Encoding As a Matrix

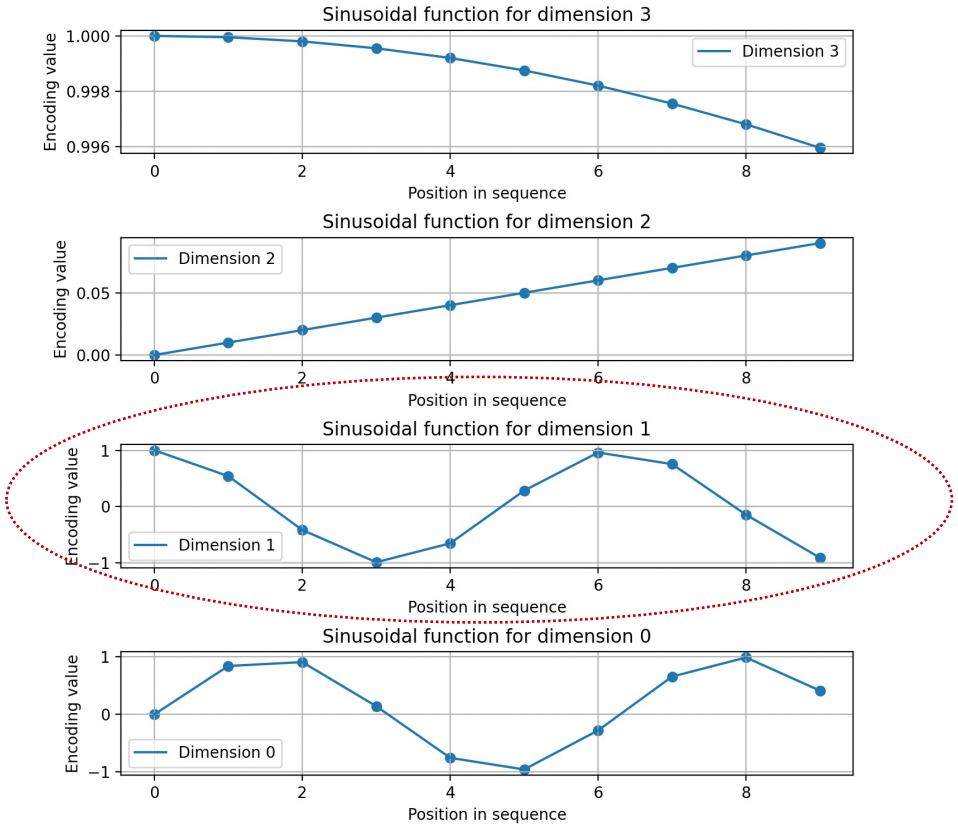


Position of the token

i[th] dimension

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{n^{2i/D}}\right) \end{cases}$$

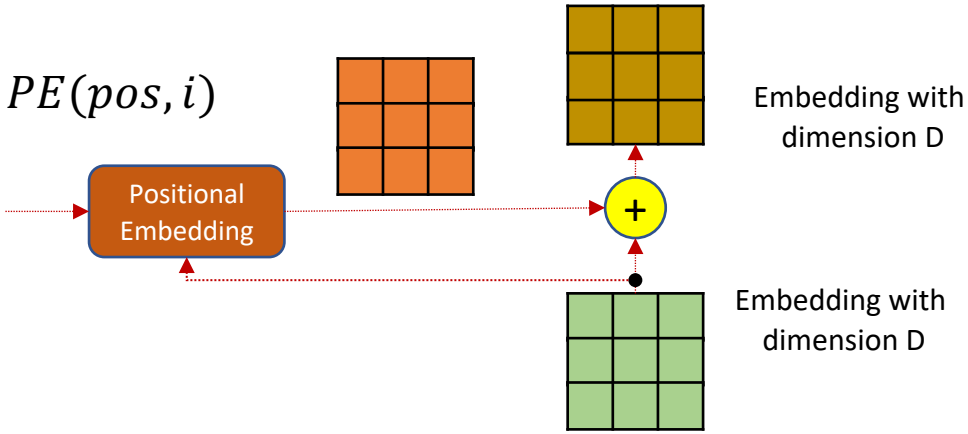
$$n = 10000$$



Pos	Word	i=0 $sin\left(\frac{Pos}{1}\right)$	i=0 $\cos\left(\frac{pos}{100^{2/4}}\right)$	i=1 $\sin\left(\frac{pos}{100^{4/4}}\right)$	i=1 $\cos\left(\frac{pos}{100^{6/4}}\right)$
0	The	0.00	1.00	0.00	1.00
1	cat	0.84	0.54	0.10	1.00
2	sat	0.91	-0.42	0.20	0.98
3	on	0.14	-0.99	0.30	0.96
4	the	-0.76	-0.65	0.39	0.92
5	mat	-0.96	0.28	0.48	0.88

Positional embeddings with D=4 and n=100

Positional Encoding As a Matrix

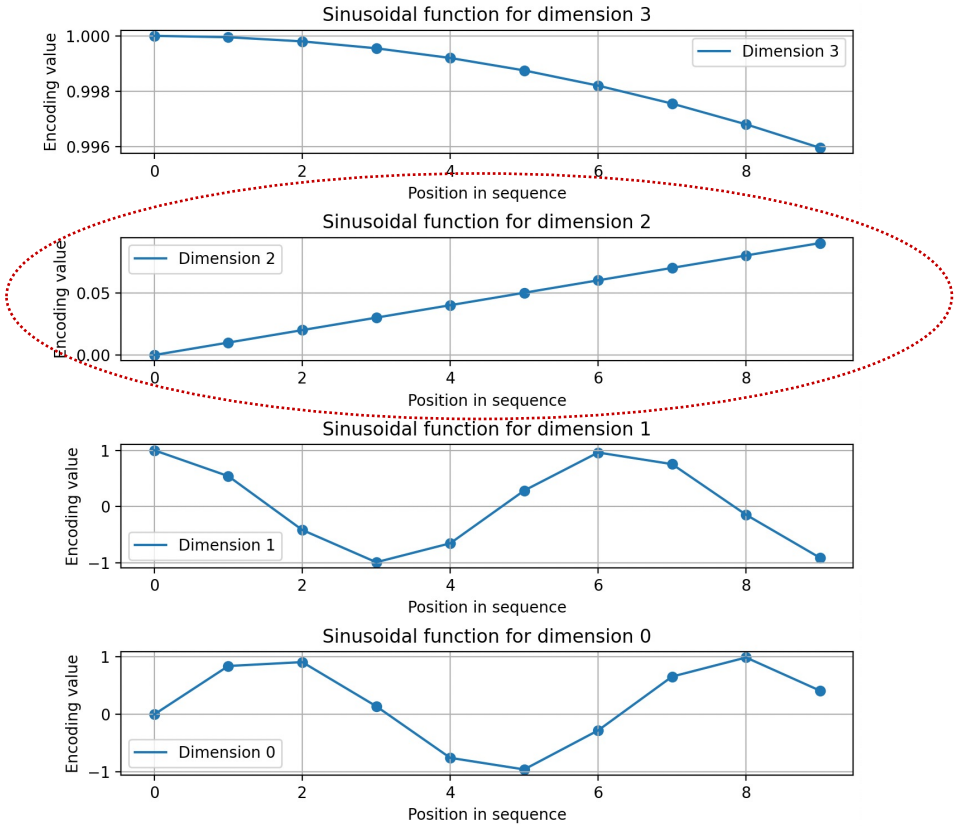


Position of the token

i[th] dimension

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{n^{2i/D}}\right) \end{cases}$$

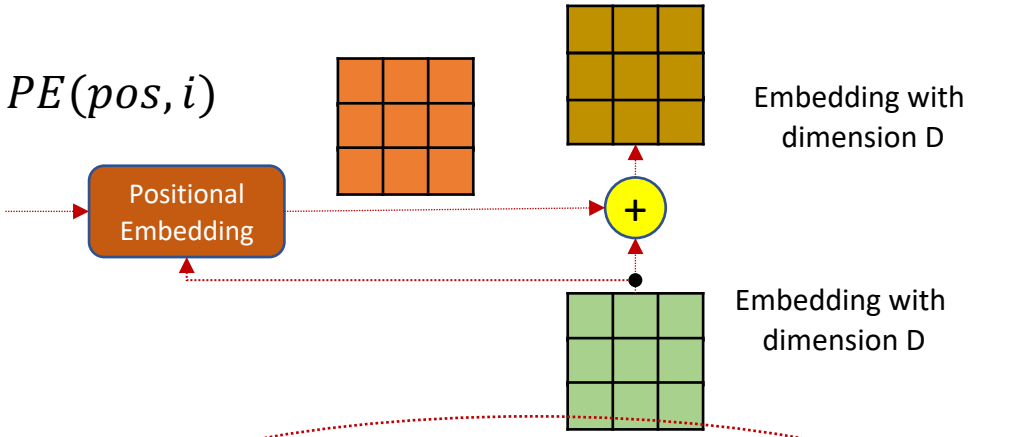
$$n = 10000$$

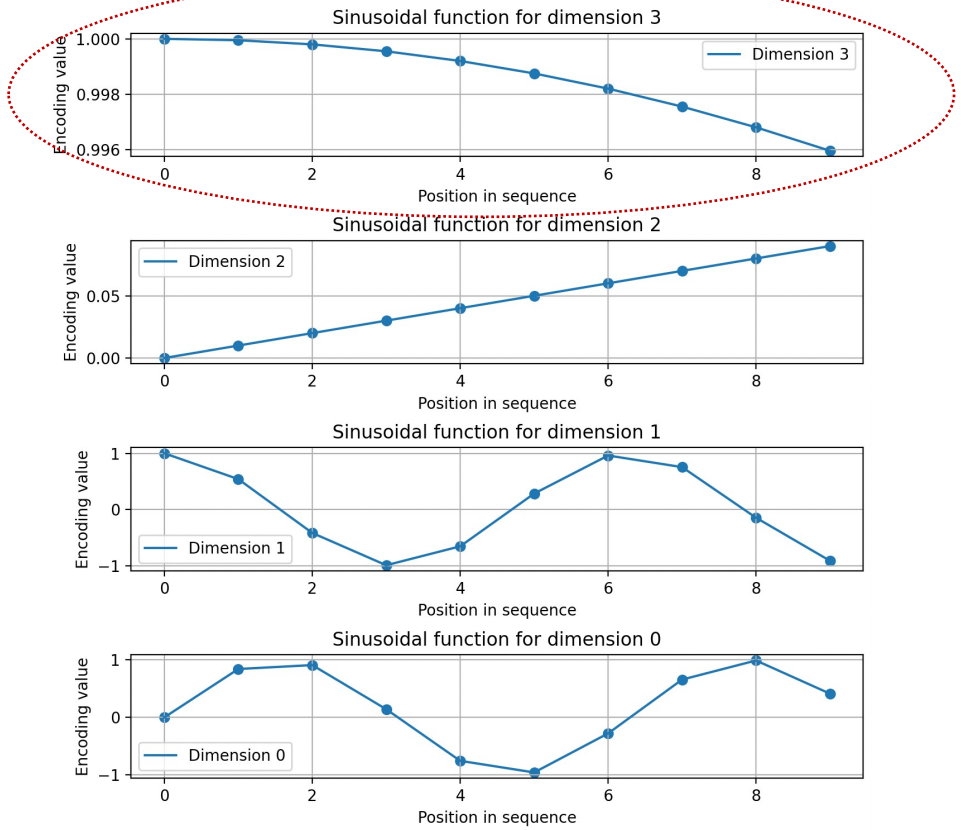


Pos	Word	i=0 $sin\left(\frac{Pos}{1}\right)$	i=0 $\cos\left(\frac{pos}{100^{2/4}}\right)$	i=1 $\sin\left(\frac{pos}{100^{4/4}}\right)$	i=1 $\cos\left(\frac{pos}{100^{6/4}}\right)$
0	The	0.00	1.00	0.00	1.00
1	cat	0.84	0.54	0.10	1.00
2	sat	0.91	-0.42	0.20	0.98
3	on	0.14	-0.99	0.30	0.96
4	the	-0.76	-0.65	0.39	0.92
5	mat	-0.96	0.28	0.48	0.88

Positional embeddings with D=4 and n=100

Positional Encoding As a Matrix





Position of the token

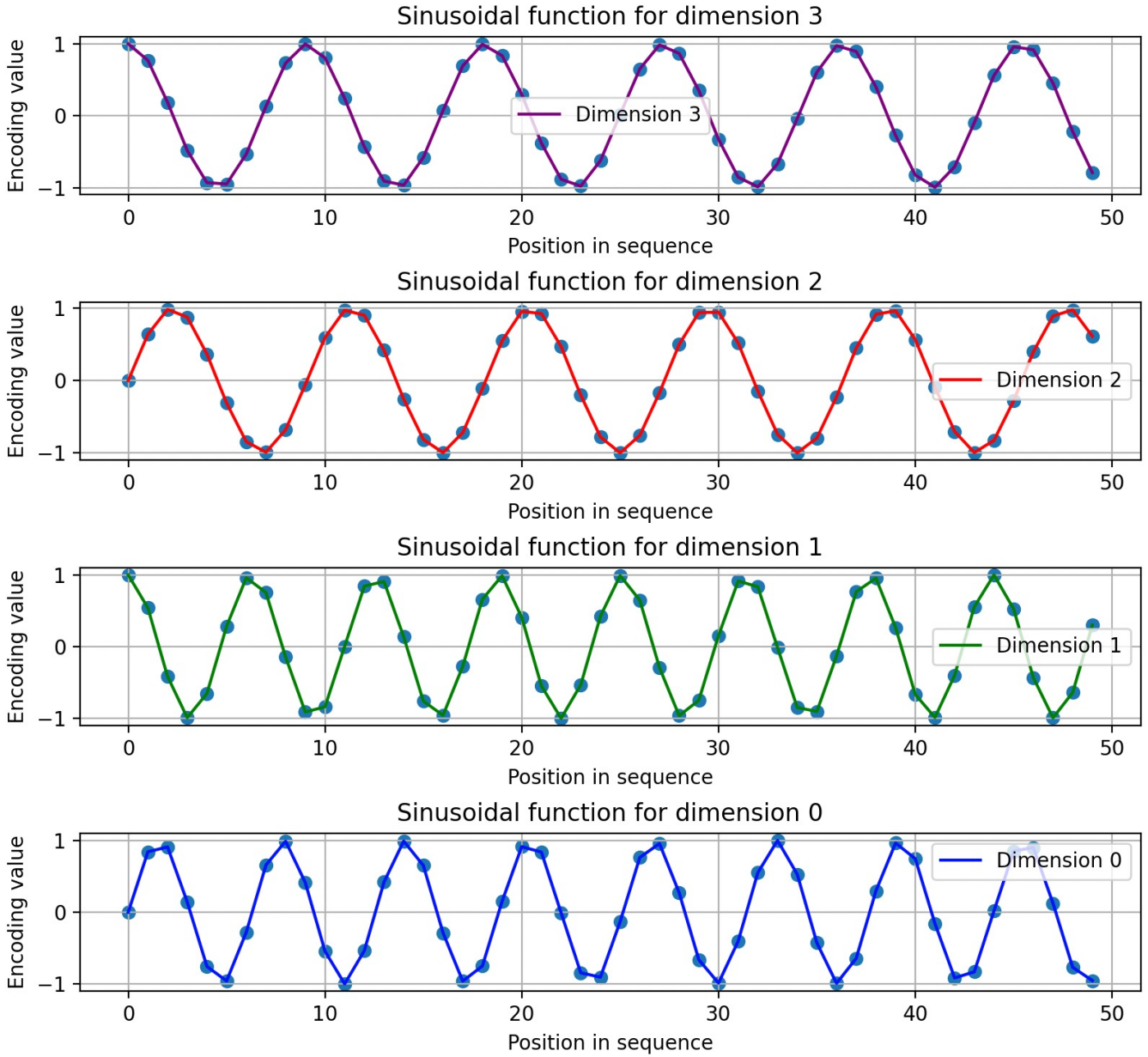
i[th] dimension

$$\begin{cases} PE(pos, 2i) = \sin\left(\dfrac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\dfrac{pos}{n^{2i/D}}\right) \end{cases}$$

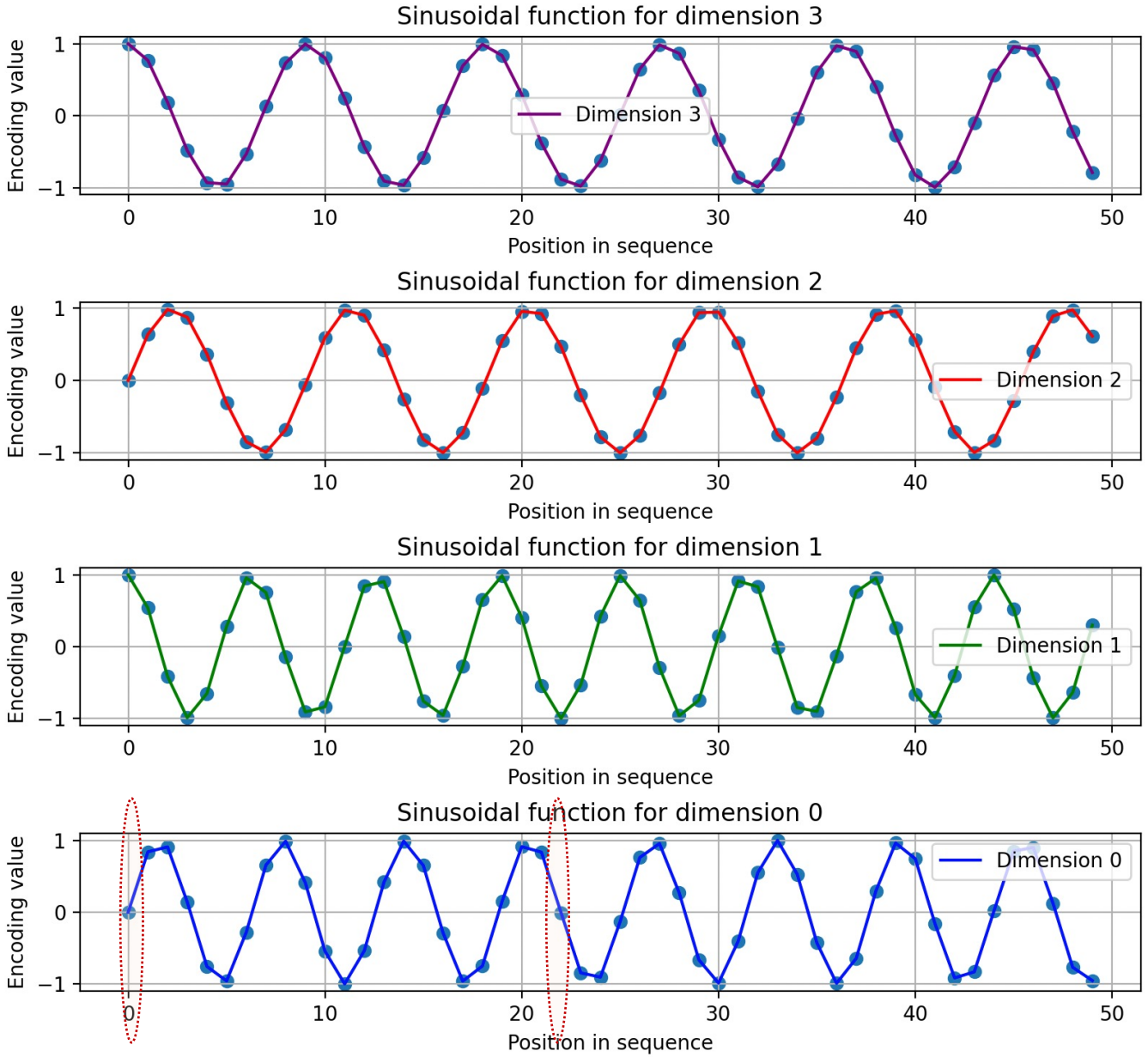
$$n = 10000$$

Pos	Word	i=0 $\sin\left(\frac{Pos}{1}\right)$	i=0 $\cos\left(\frac{pos}{100^{2/4}}\right)$	i=1 $\sin\left(\frac{pos}{100^{4/4}}\right)$	i=1 $\cos\left(\frac{pos}{100^{6/4}}\right)$
0	The	0.00	1.00	0.00	1.00
1	cat	0.84	0.54	0.10	1.00
2	sat	0.91	-0.42	0.20	0.98
3	on	0.14	-0.99	0.30	0.96
4	the	-0.76	-0.65	0.39	0.92
5	mat	-0.96	0.28	0.48	0.88

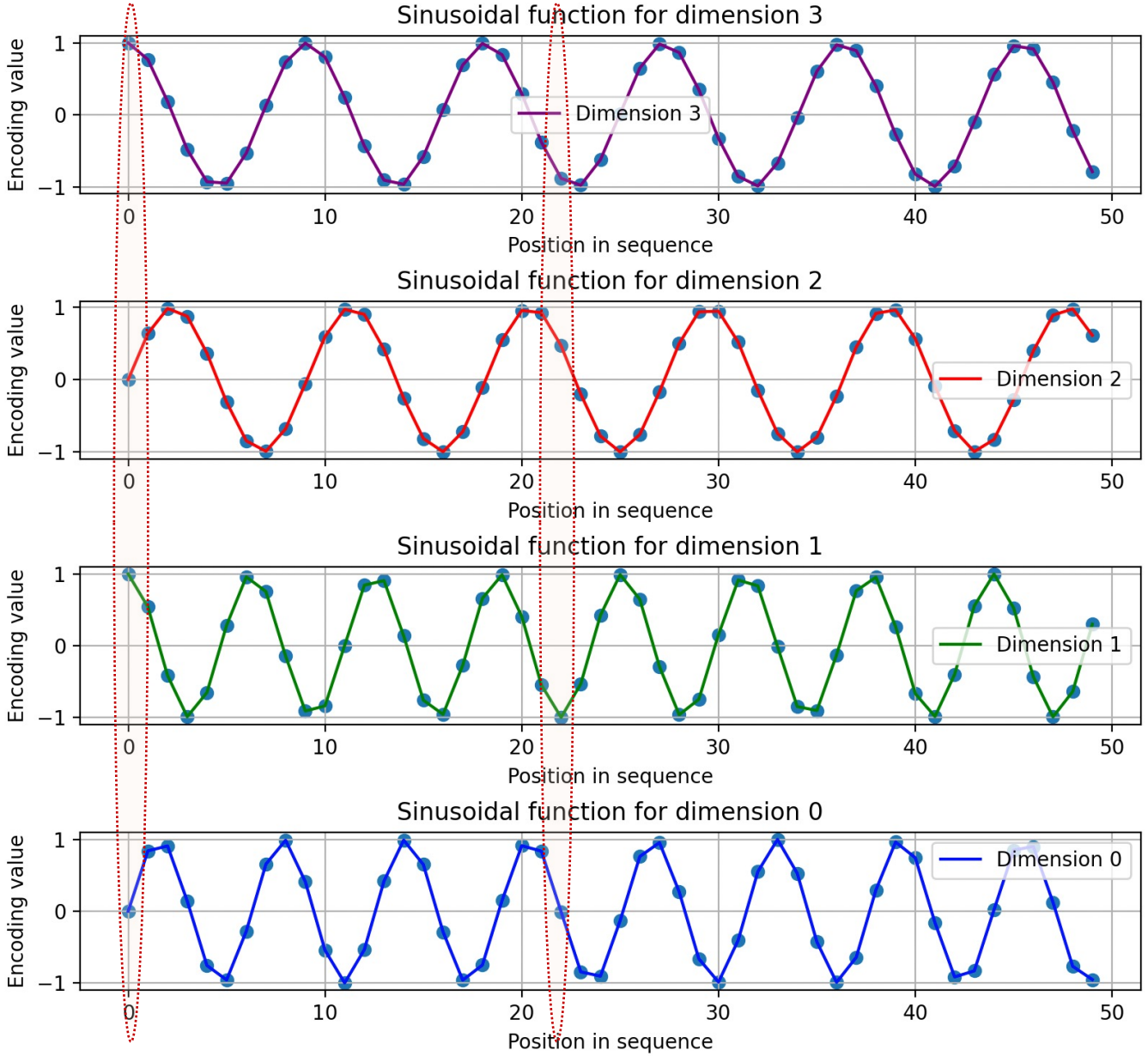
Positional embeddings with D=4 and n=100


Sinusoidal function for dimension 3
Encoding value
Position in sequence
Dimension 3
Sinusoidal function for dimension 2
Encoding value
Position in sequence
Dimension 2
Sinusoidal function for dimension 1
Encoding value
Position in sequence
Dimension 1
Sinusoidal function for dimension 0
Encoding value
Position in sequence
Dimension 0

D=50, n=100

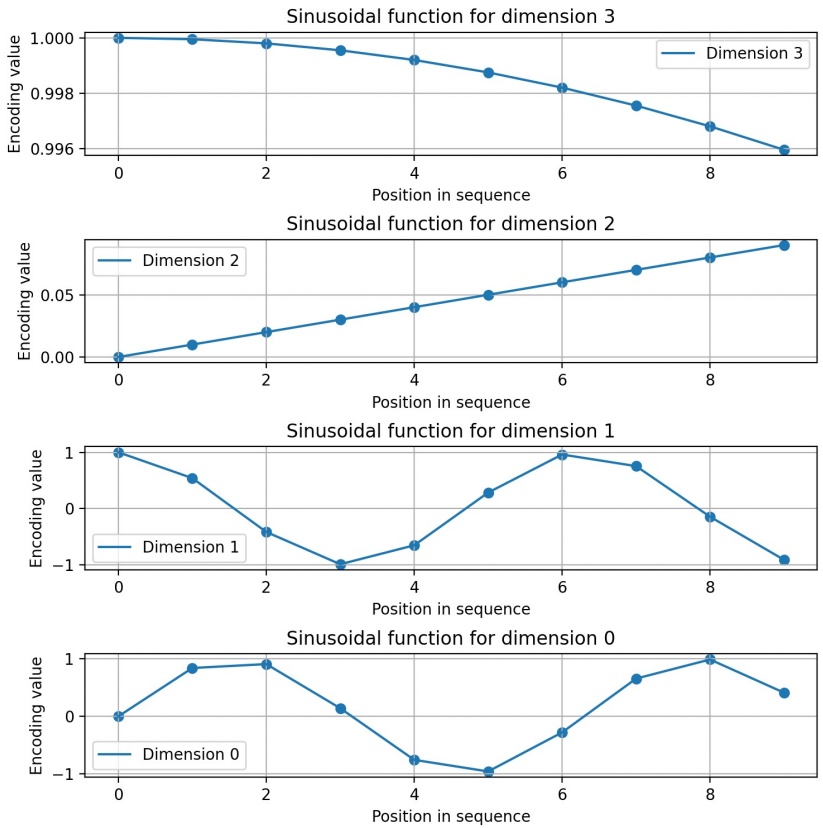

Sinusoidal function for dimension 3
Encoding value
Position in sequence
Dimension 3
Sinusoidal function for dimension 2
Encoding value
Position in sequence
Dimension 2
Sinusoidal function for dimension 1
Encoding value
Position in sequence
Dimension 1
Sinusoidal function for dimension 0
Encoding value
Position in sequence
Dimension 0

D=50, n=100


Sinusoidal function for dimension 3
Encoding value
Position in sequence
Dimension 3
Sinusoidal function for dimension 2
Encoding value
Position in sequence
Dimension 2
Sinusoidal function for dimension 1
Encoding value
Position in sequence
Dimension 1
Sinusoidal function for dimension 0
Encoding value
Position in sequence
Dimension 0

D=50, n=100

Pos	Word	i=0	i=1	i=2	i=3
0	The	0.00	1.00	0.00	1.00
1	cat	0.84	0.54	0.10	1.00
2	sat	0.91	-0.42	0.20	0.98
3	on	0.14	-0.99	0.30	0.96
4	the	-0.76	-0.65	0.39	0.92
5	mat	-0.96	0.28	0.48	0.88



Where are we in the sequence? (low resolution)

Distinguishes consecutive tokens more (Higher resolution)

Higher dimensions add more information about where we are in the sequence with higher resolution

Effect of n

Suppose Max(pos) = 50

Position of the token

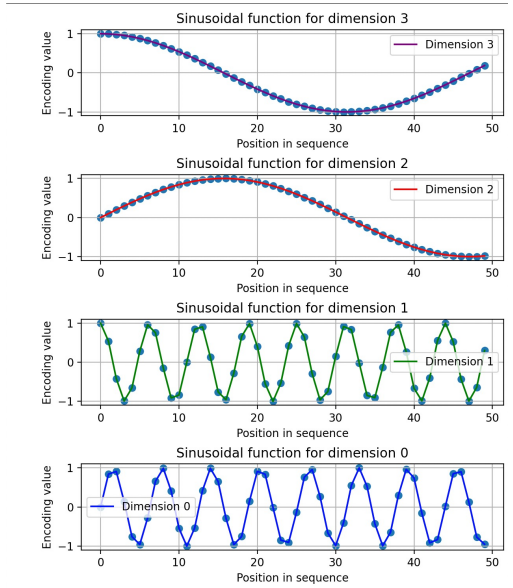
i^{th} dimension

$0 \leq i \leq \frac{D}{2} - 1$

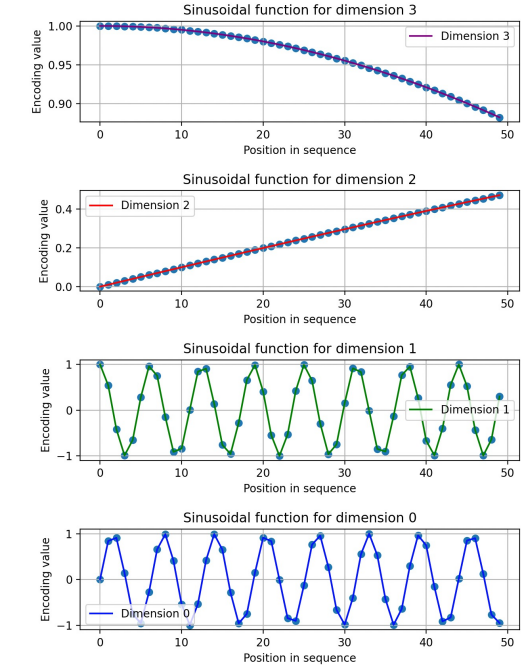
Length of embedding i.e. each token is a vector of D numbers

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{n^{2i/D}}\right) \end{cases}$$

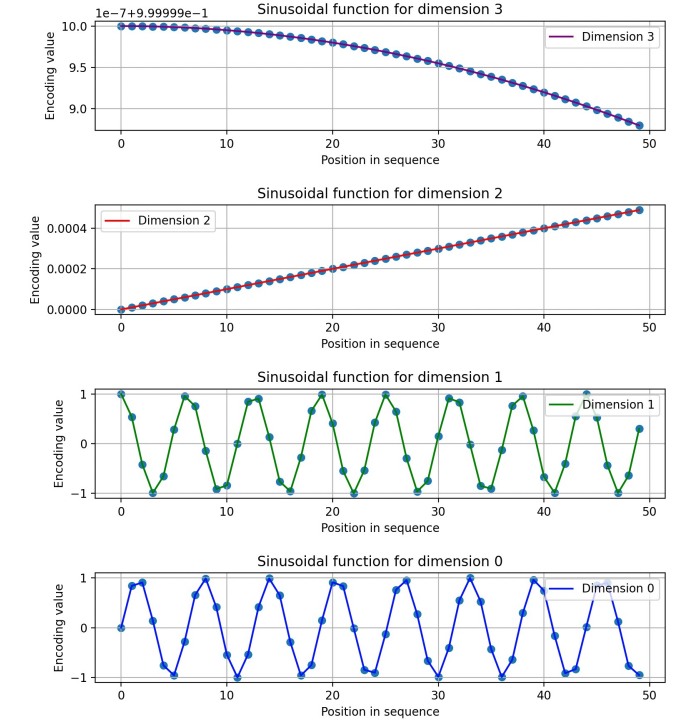
n = 10



n = 100



n = 100000



Effect of n

Suppose Max(pos) = 50

Position of the token

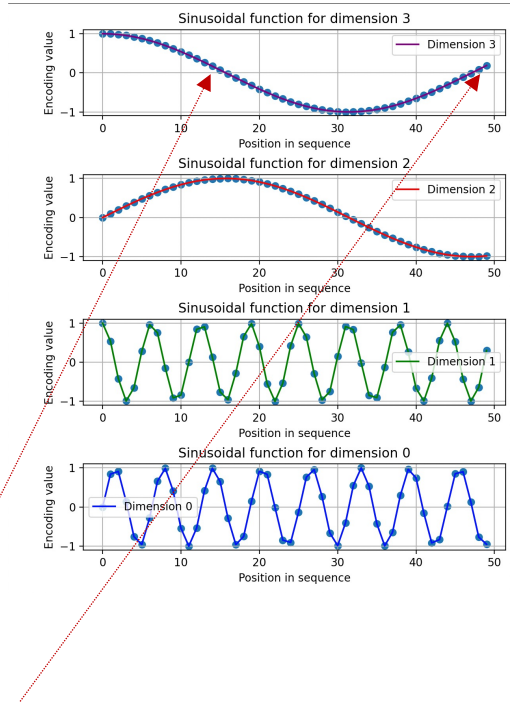
i^{th} dimension

$0 \le i \le \frac{D}{2} - 1$

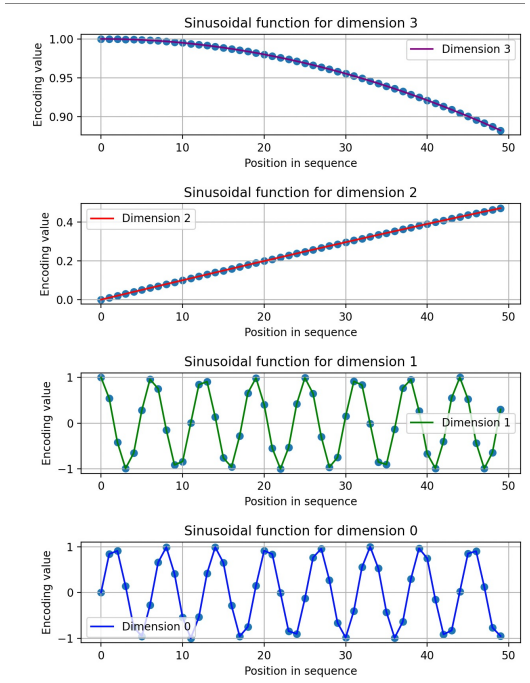
Length of embedding i.e. each token is a vector of D numbers

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{n^{2i/D}}\right) \end{cases}$$

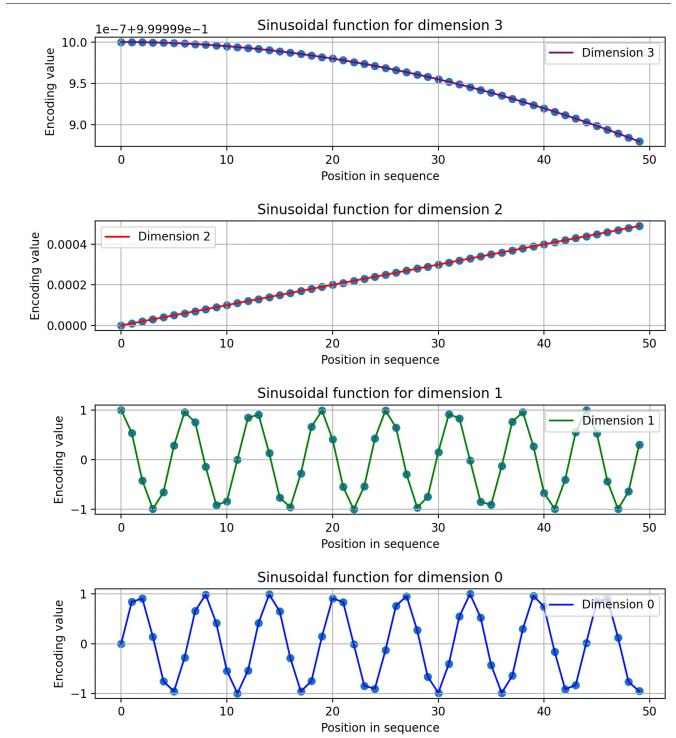
n = 10



n = 100



n = 100000



Periodic values show up

Effect of n

Suppose Max(pos) = 50

Position of the token

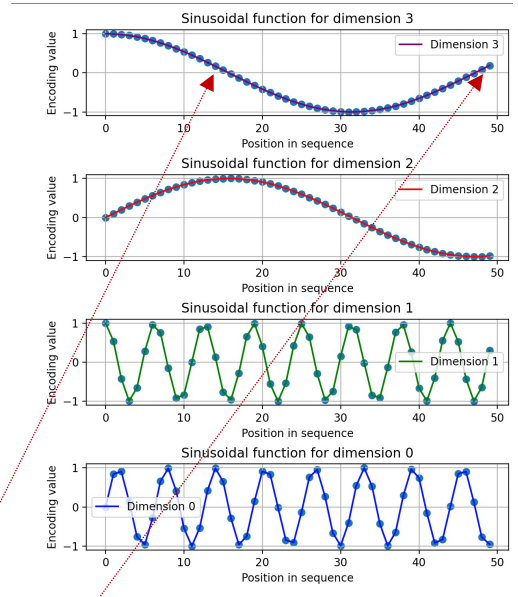
i^{th} dimension

$0 \le i \le \frac{D}{2} - 1$

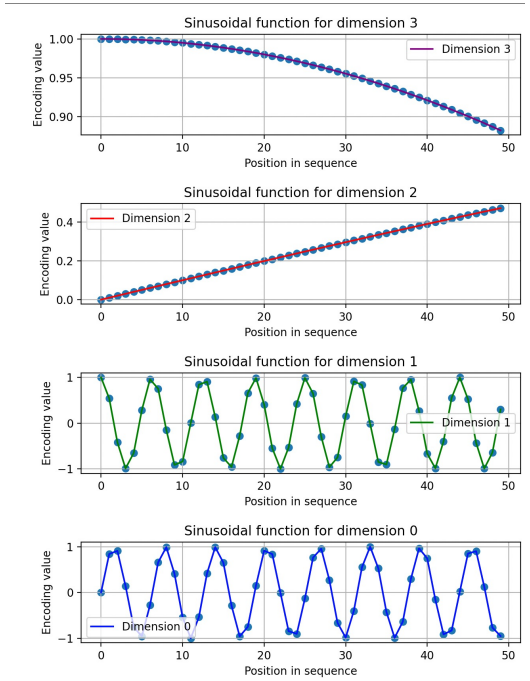
Length of embedding i.e. each token is a vector of D numbers

$$\begin{cases} PE(pos, 2i) = \sin\left(\frac{pos}{n^{2i/D}}\right) \\ PE(pos, 2i+1) = \cos\left(\frac{pos}{n^{2i/D}}\right) \end{cases}$$

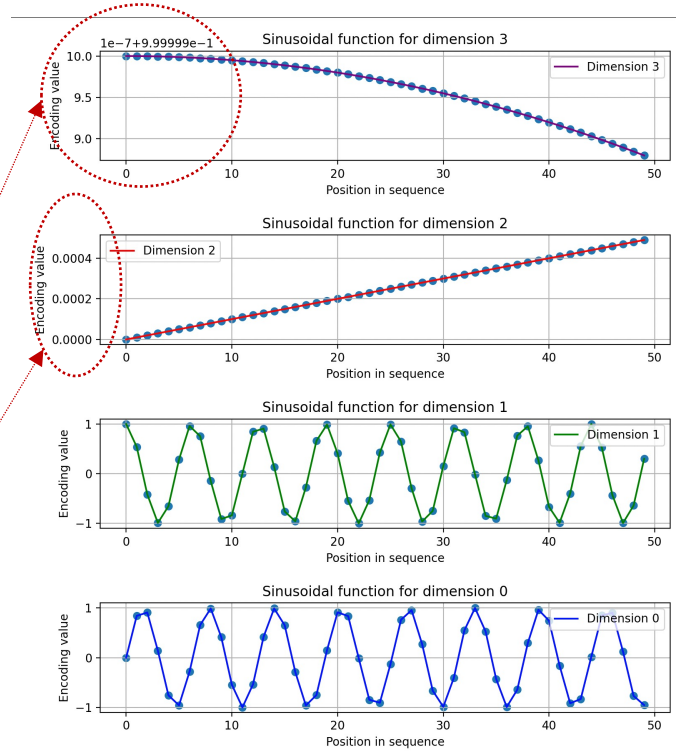
n = 10



n = 100



n = 100000

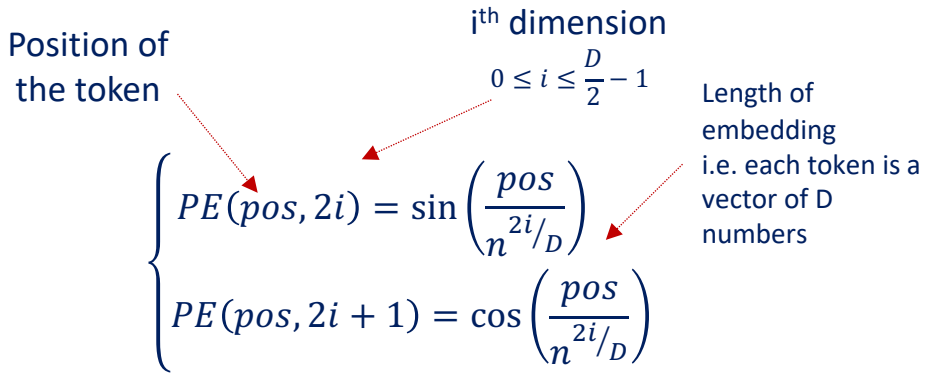


Periodic values show up

Differences are smaller

Effect of n

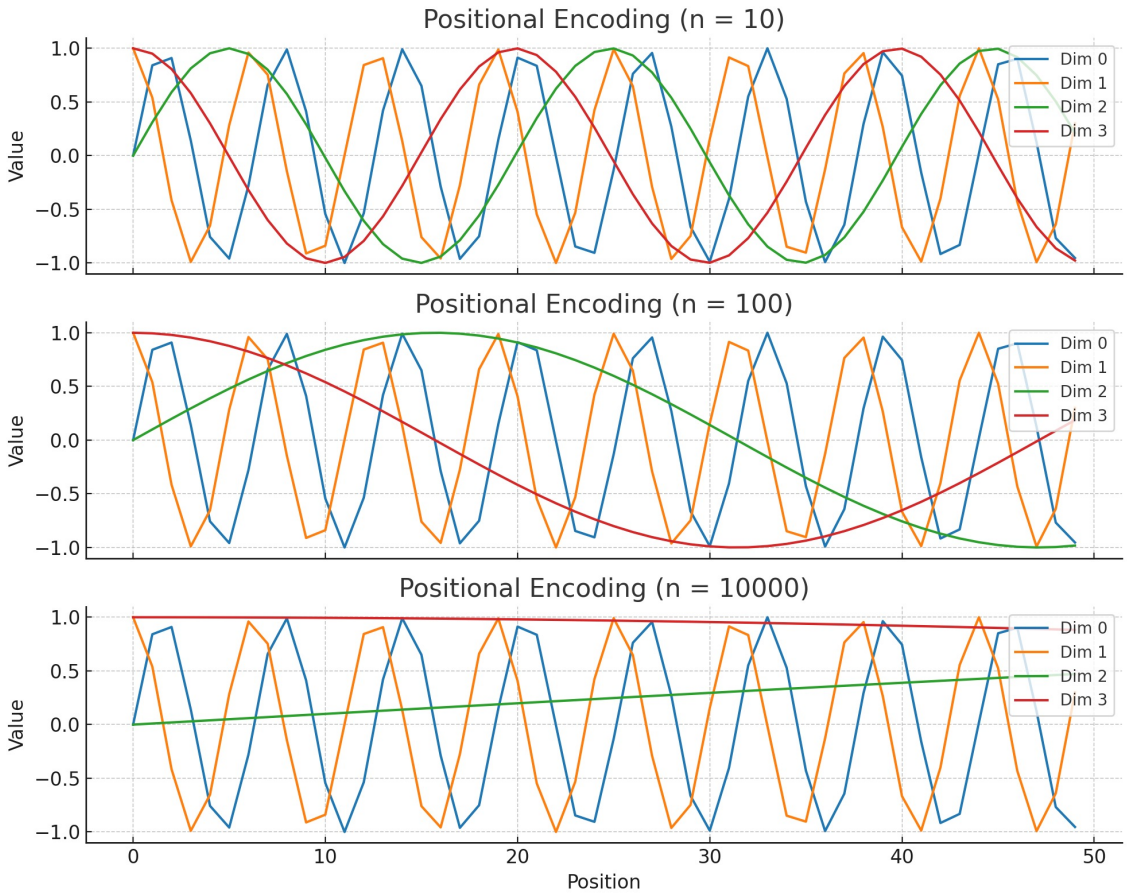
Suppose Max(pos) = 50



- **larger n → slower frequency decay**
  - More dimensions change **slowly** → good for encoding **global position**
- **Smaller n → faster frequency increase** across dimensions
  - More dimensions change **quickly** → focus on **local positio**

- **Why n=10,000 is a good default?**
  - Gives a **smooth spectrum** of low-to-high frequencies
  - Works well for sequence lengths up to **several thousand tokens**
  - Ensures each position gets a **unique and distinguishable** encoding

Effect of n



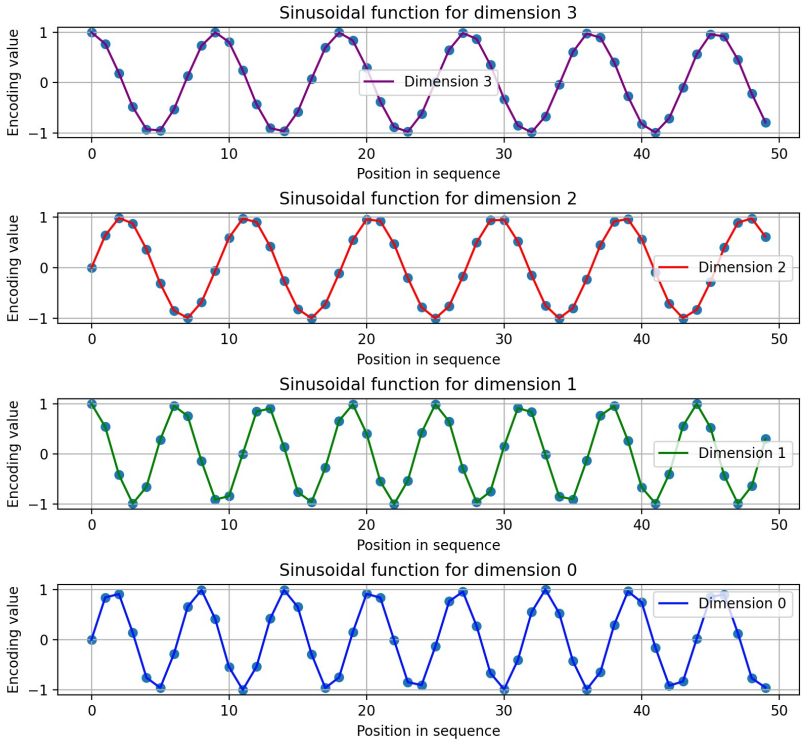
- **Top (n = 10):** Frequencies are very high across all dimensions → local patterns dominate
- **Middle (n = 100):** A mix of moderate and low frequencies
- **Bottom (n = 10000):** Slow-changing dimensions appear — captures global structure better

This shows how increasing n spreads the frequency spectrum across dimensions, enabling both **local and global position encoding**.

Why Sin and Cos?

- Continuous and Differentiable
  - Range between -1, 1
- Linear Transformations:
  - The encoding for position pos+k can be calculated as a linear function of the encoding for position pos.
  - This allows the model to detect how far apart tokens are in the sequence.

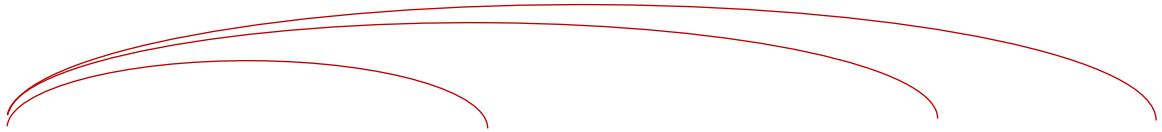
$$\sin(x + k) = \sin(x)\cos(k) - \cos(x)\sin(k)$$



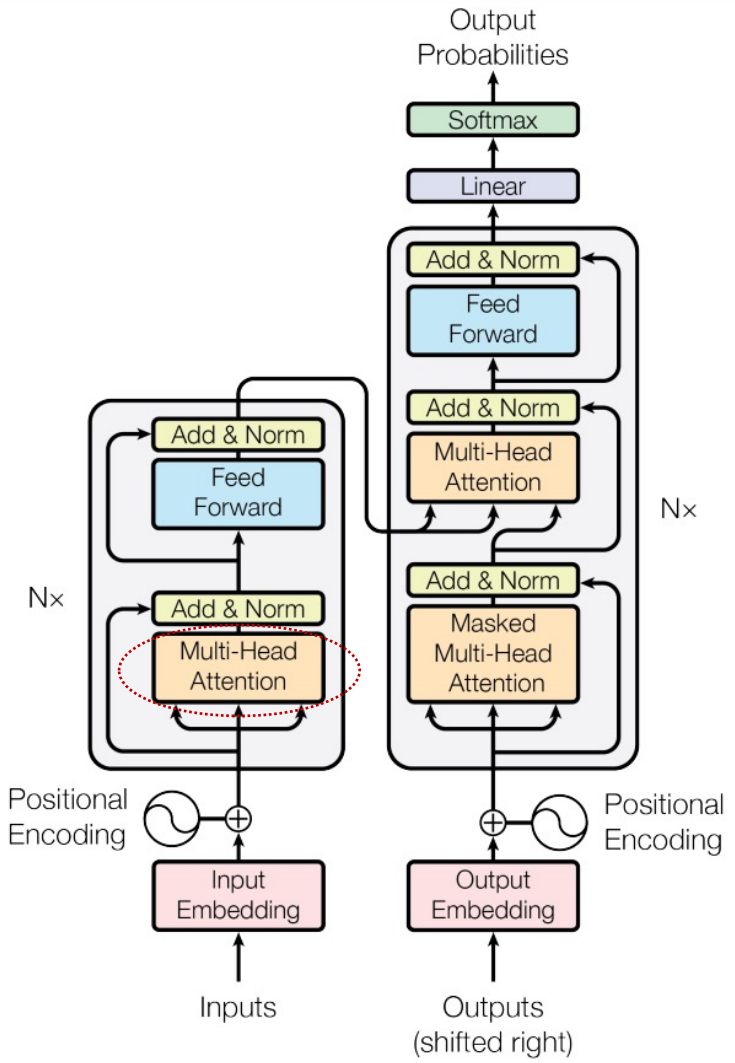
Self Attention

Self Attention

- Associating the related words together
  - Captures dependencies regardless of distance.
  - Parallelizable computation enhances training efficiency.

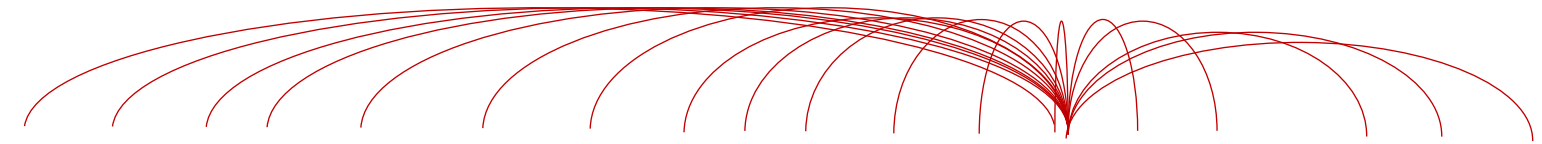


The **cat**, the cute little **white one**, sat on the mat and **it** then **jumped** on the couch.

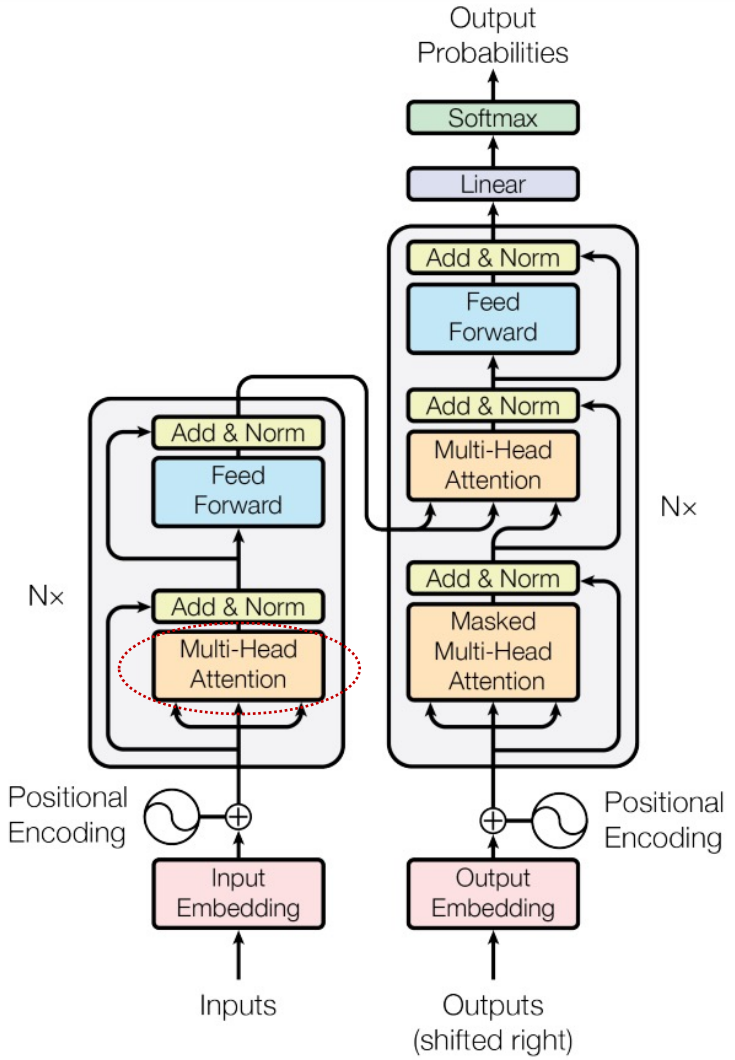


Self Attention

- Associating the related words together
  - Captures dependencies regardless of distance.
  - Parallelizable computation enhances training efficiency.



The **cat**, the cute little **white one**, sat on the mat and **it** then **jumped** on the couch.



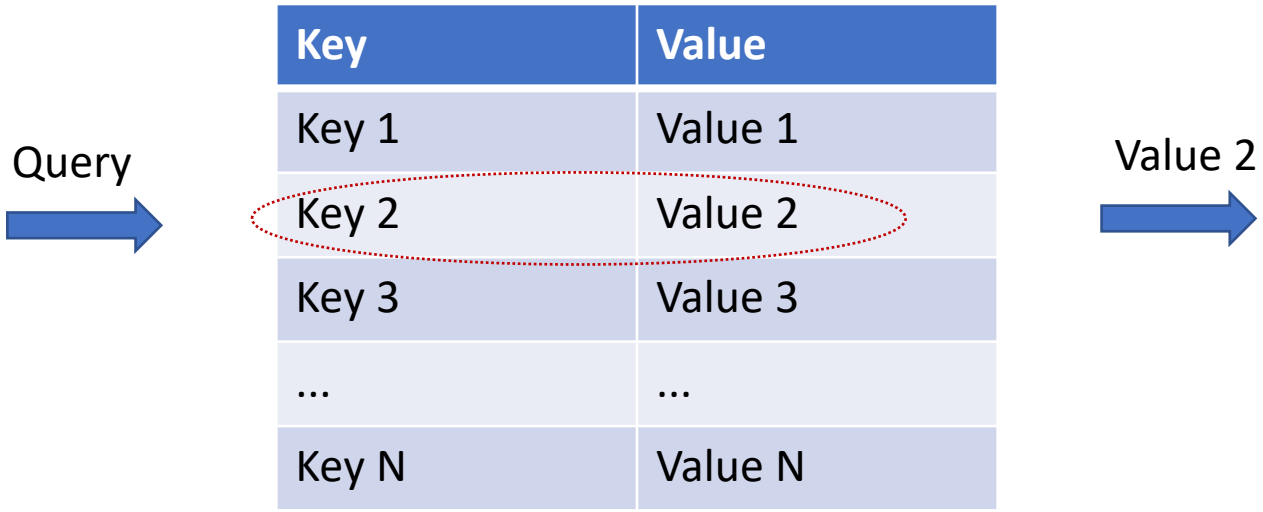
Attention: Database Analogy

Query

Key	Value
Key 1	Value 1
Key 2	Value 2
Key 3	Value 3
...	...
Key N	Value N

Database Analogy **(Q, K, V)**: A search query **Q** is matched against multiple **(K,V)** entries.
If **Q** matches **K**, the value **V** is retrieved.

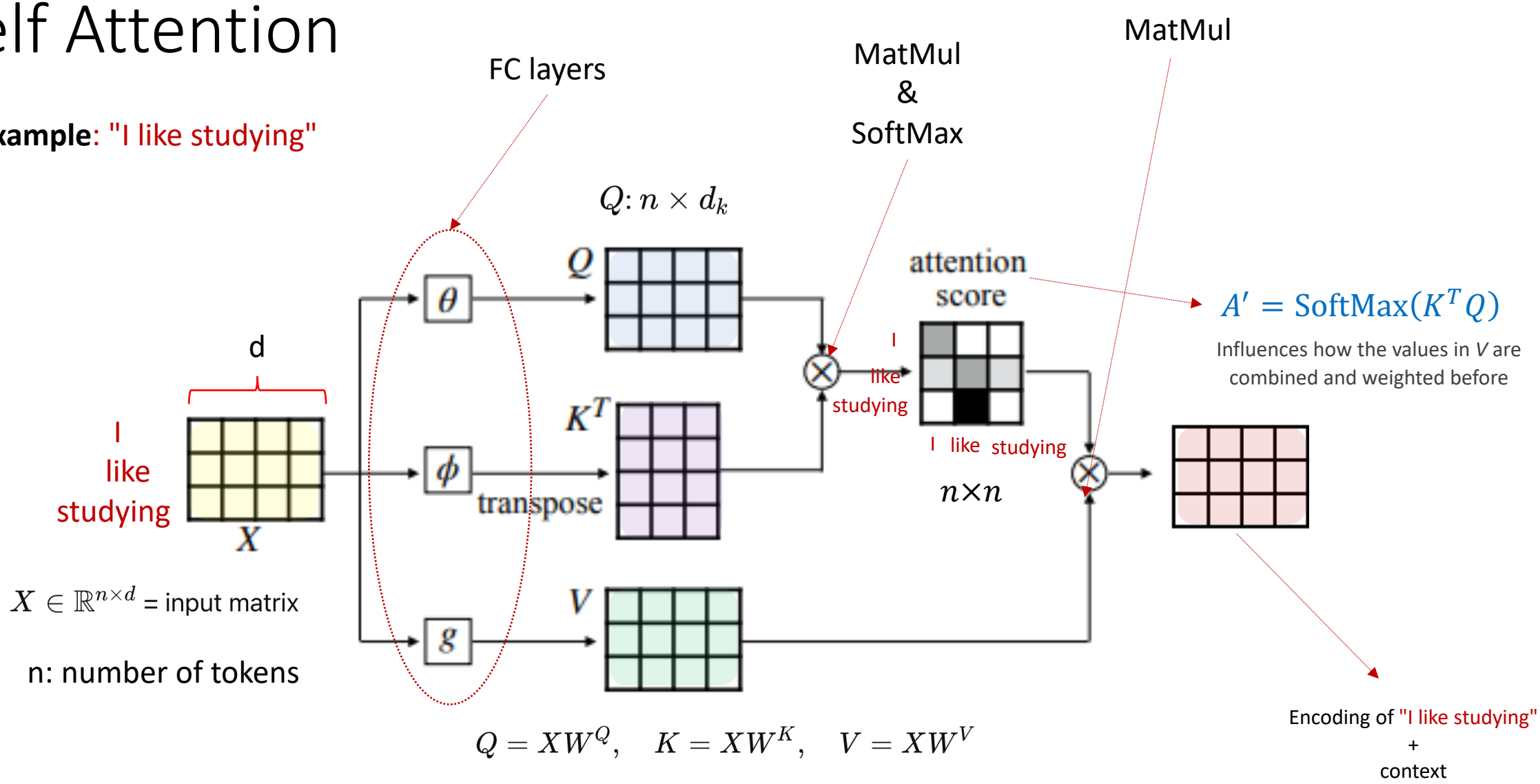
Attention: Database Analogy



Database Analogy **(Q, K, V)**: A search query **Q** is matched against multiple **(K,V)** entries.
If **Q** matches **K**, the value **V** is retrieved.

Self Attention

Example: "I like studying"

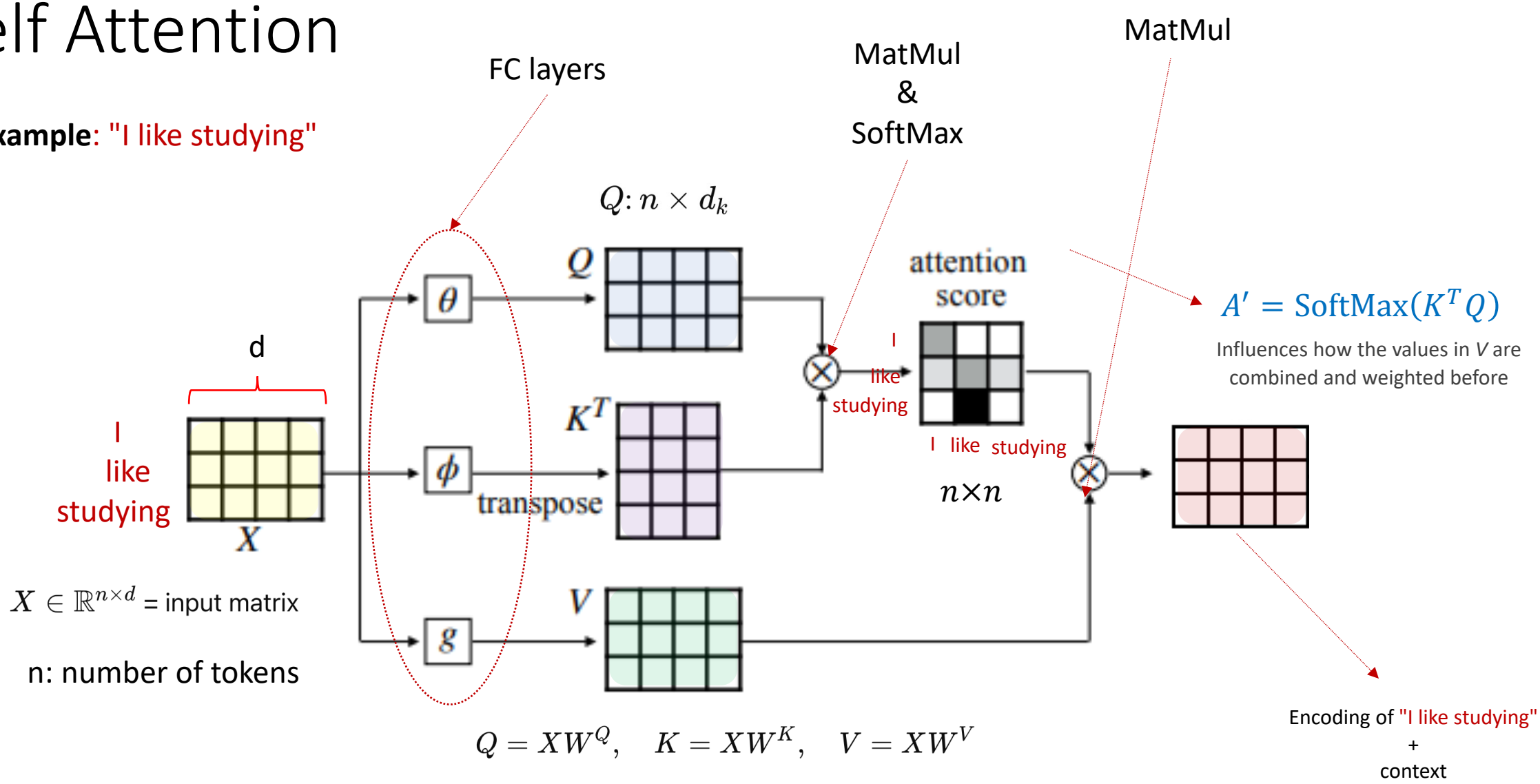


$W^Q, W^K, W^K \in R^{d \times d_k}$: learned weight matrices for Query, Key, Value

- d is the **embedding dimension** of the input tokens — also called the **model dimension** or d_model.
- dk is the **dimension of the Key and Query vectors**.

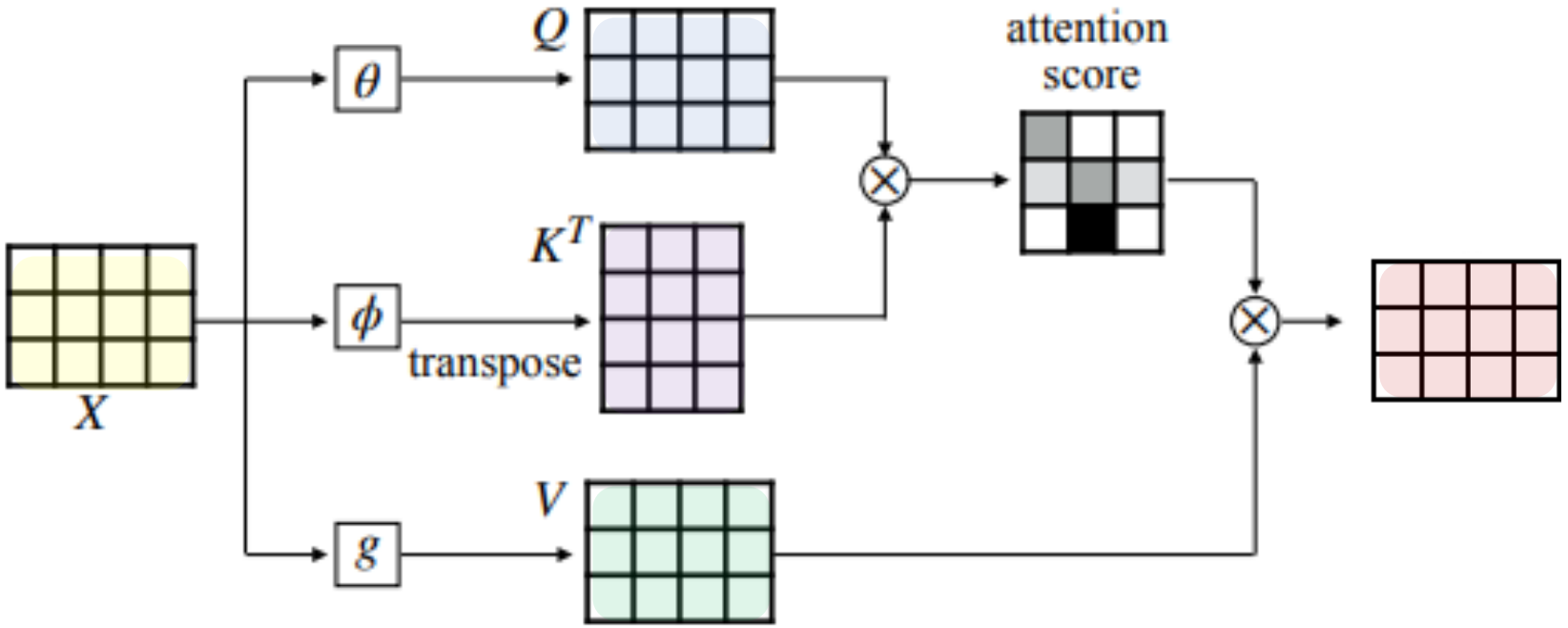
Self Attention

Example: "I like studying"

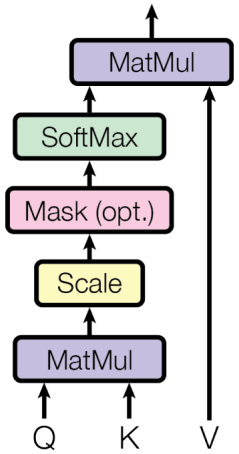


- Database Analogy **(Q, K, V)**: A search query **Q** is matched against multiple **(K,V)** entries.
- If **Q** matches **K**, the value **V** is retrieved.
- **(Q, K, V)**: Different "views" of X requiring **learned projections** from X → Q, K, V.

Scaled Dot-Product Attention



Scaled Dot-Product Attention

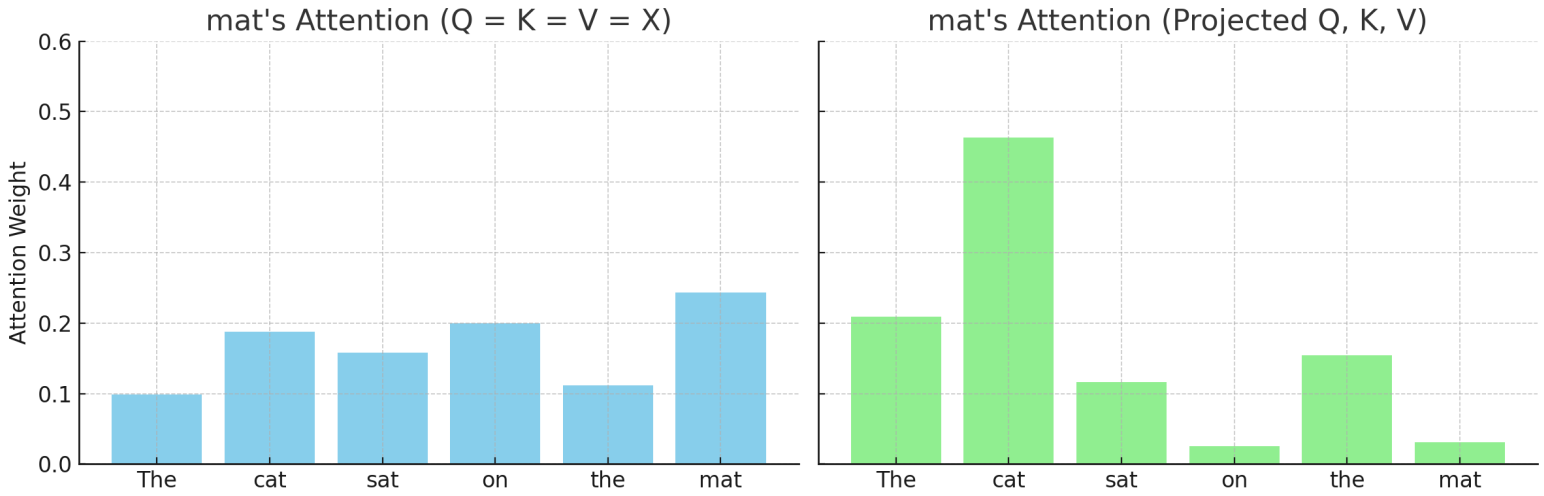


$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{K^T Q}{\sqrt{d_k}}\right) V$$

Dimension of K

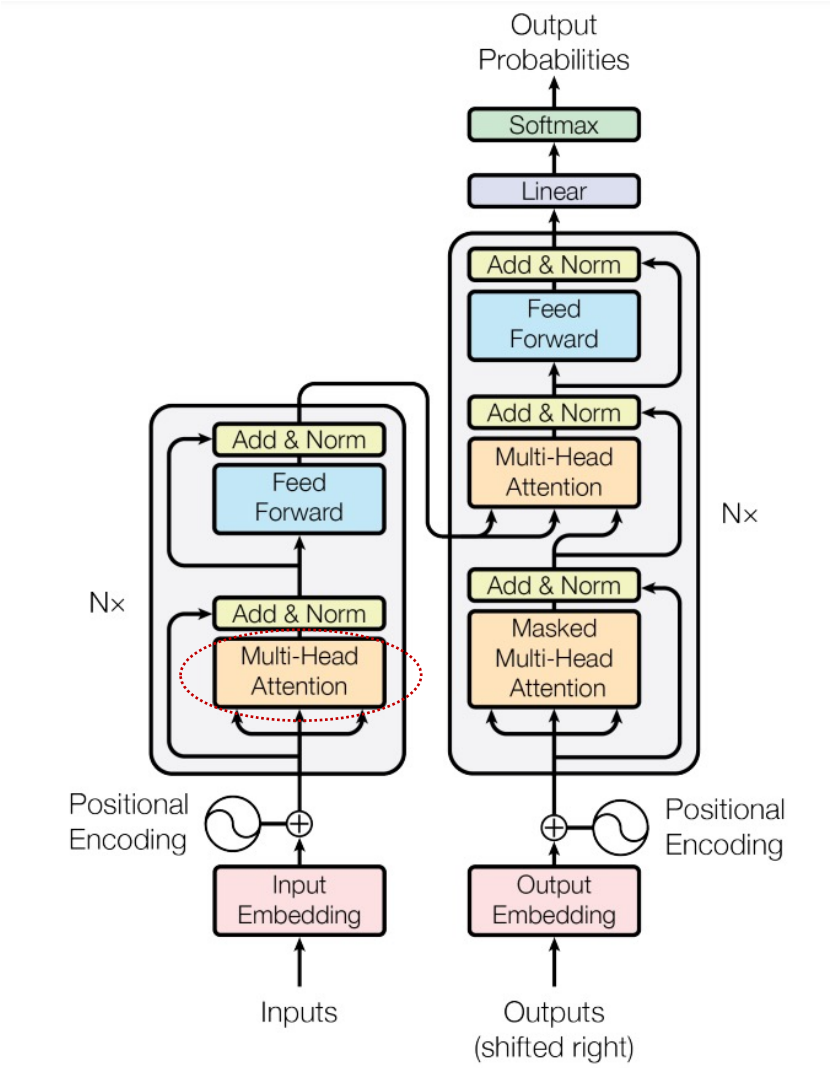
Why Do We Create Query, Key, and Value Vectors?

- **Why Not Just Use X Directly?**
  - Adds **learning capacity**
  - Separates **matching** from **content passing**
    - **Q/K decide *who to listen to*, and V determines *what they say*.**
  - Essential for **multi-head attention**:
    - Different Q/K/V projections in each head
    - Each head focuses on different relationships



Multi-Head Attention

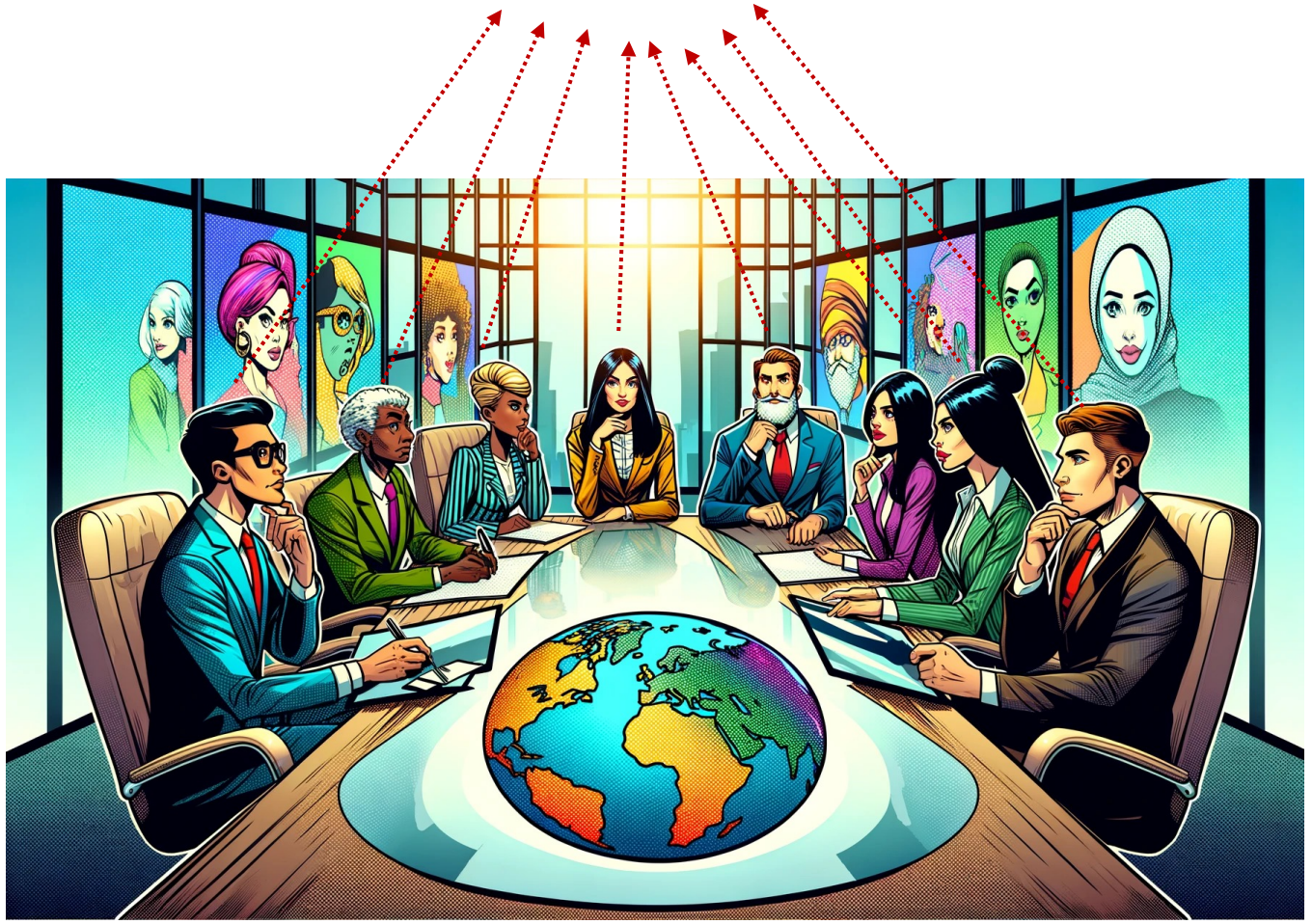
Multi-Head Attention



Multi-Head Attention Intuition

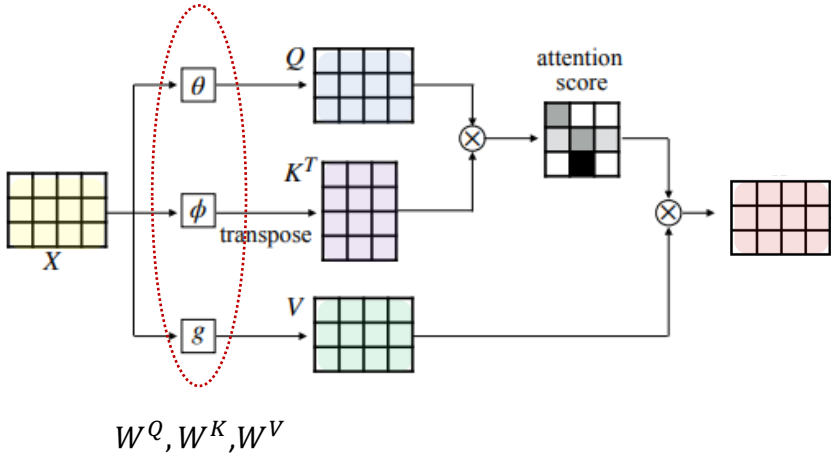
Multi-Head Attention Intuition

Combine these to get a more **comprehensive** understanding of the topic



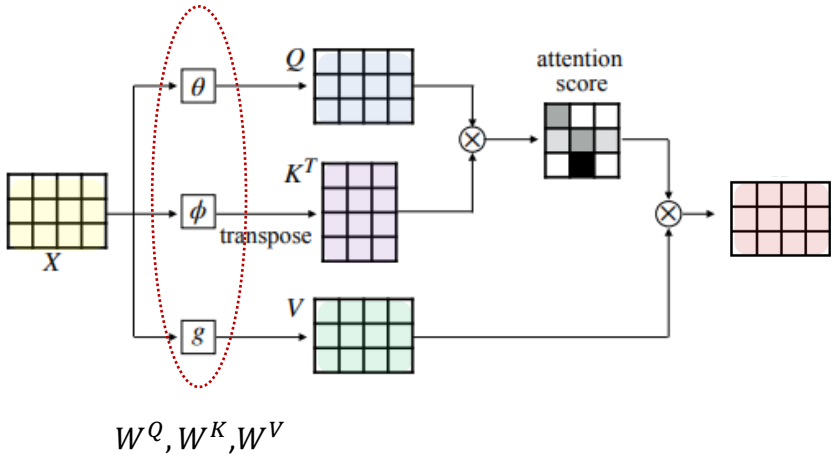
Analogy: multiple people have different understanding from the same topic

Multi-head Attention



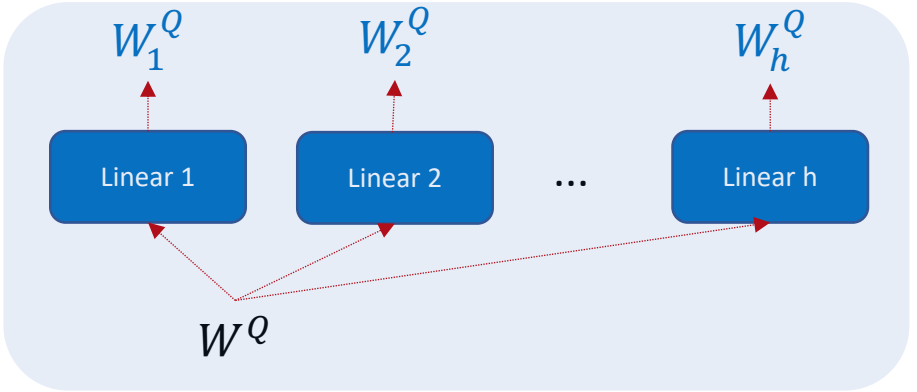
Single head: A single weight matrix for each of Q, K, and V:
W^Q, W^K, W^V

Multi-head Attention

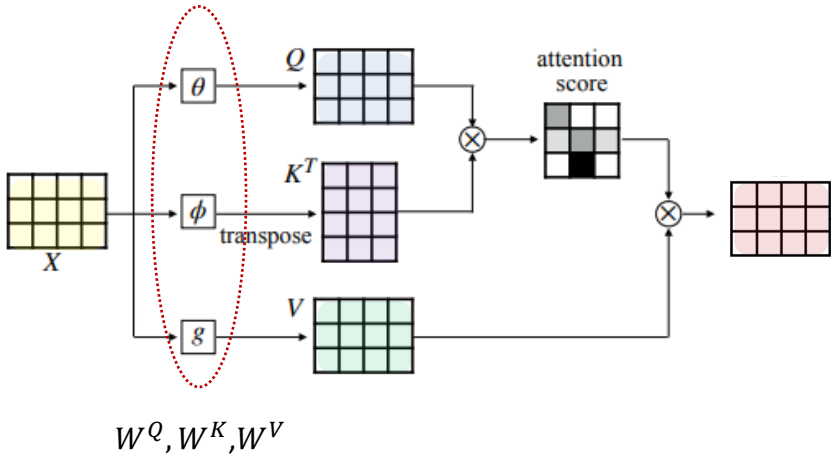


W^Q, W^K, W^V

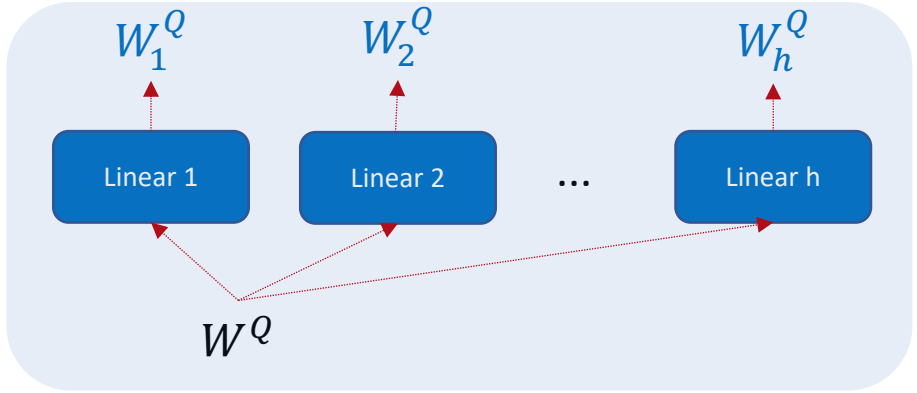
Single head: A single weight matrix for each of Q, K, and V:
W^Q, W^K, W^V

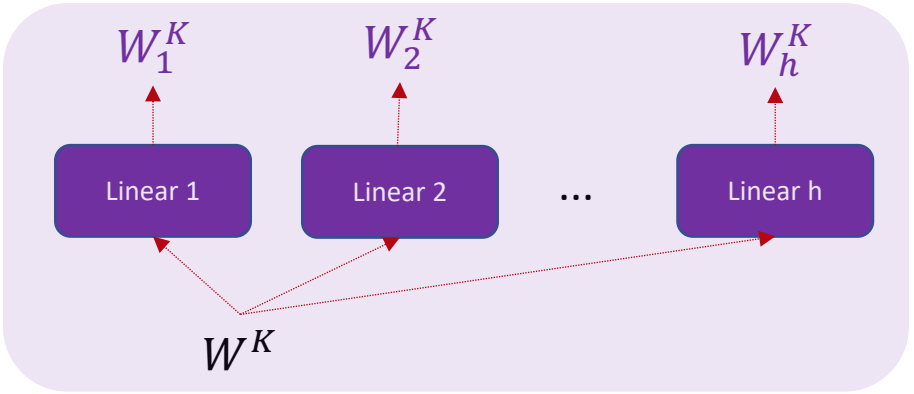


Multi-head Attention

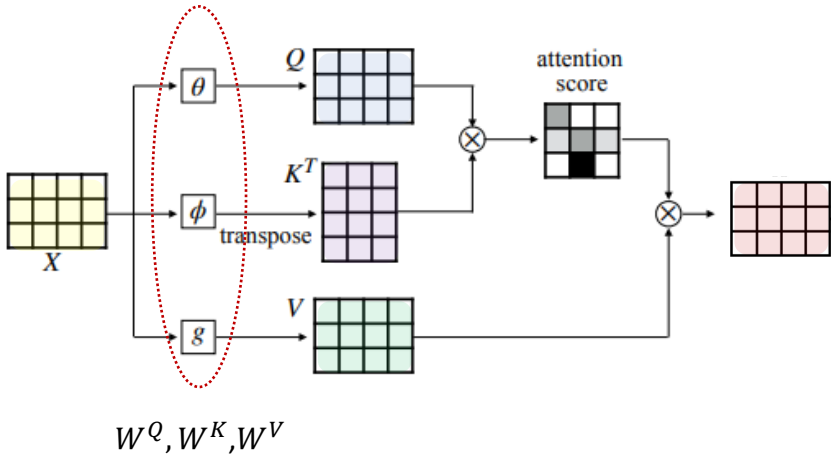


Single head: A single weight matrix for each of Q, K, and V:
W^Q, W^K, W^V

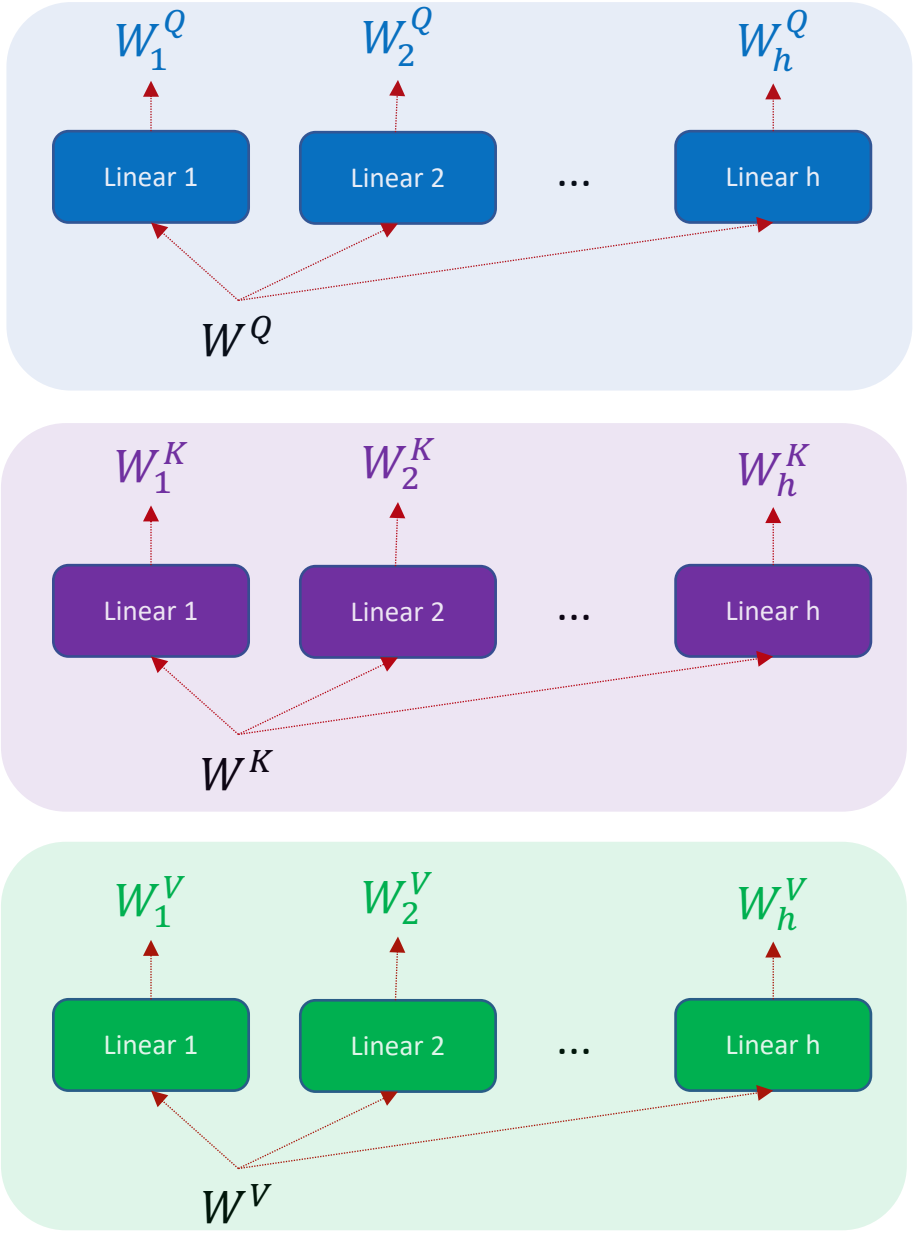




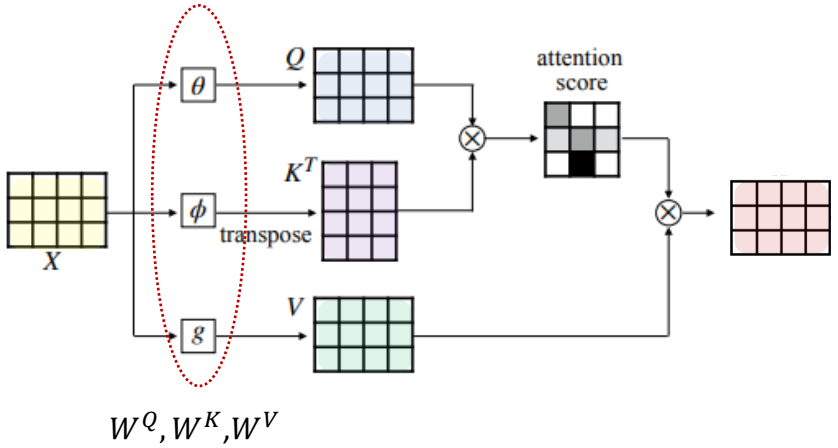
Multi-head Attention



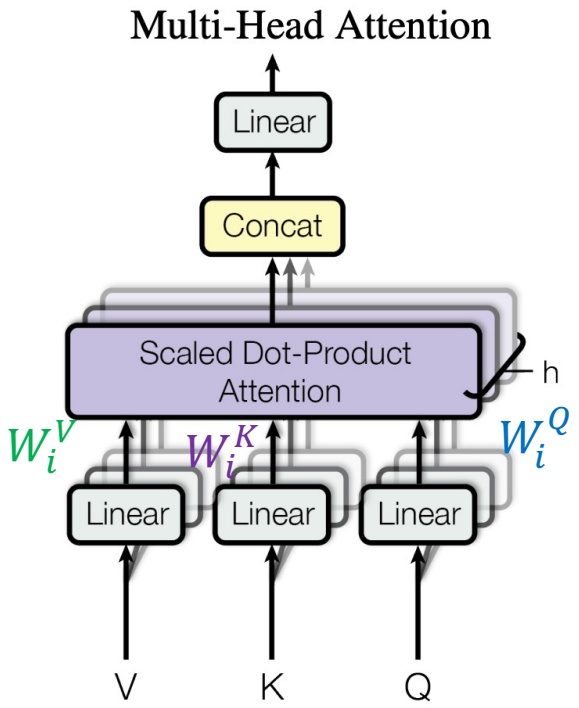
Single head: A single weight matrix for each of Q, K, and V:
W^Q, W^K, W^V



Multi-head Attention



Single head: A single weight matrix for each of Q, K, and V:
W^Q, W^K, W^V

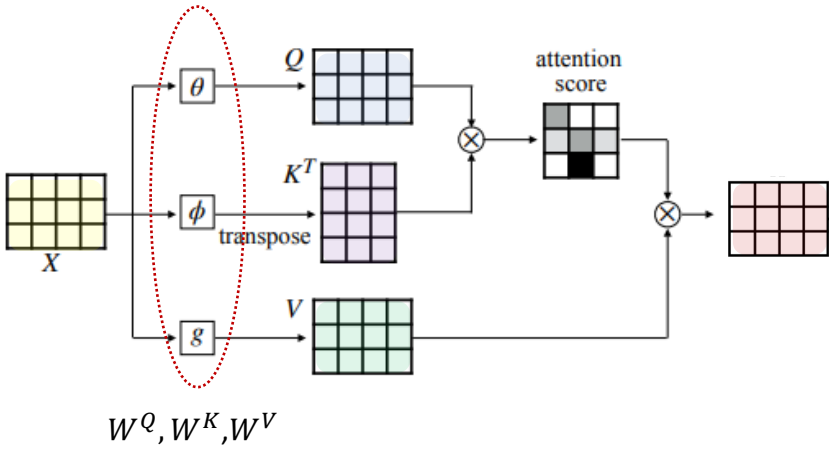


Multi head: a single weight matrix for each of **Q, K**, and **V** per head h_i

W_i^V W_i^K W_i^Q

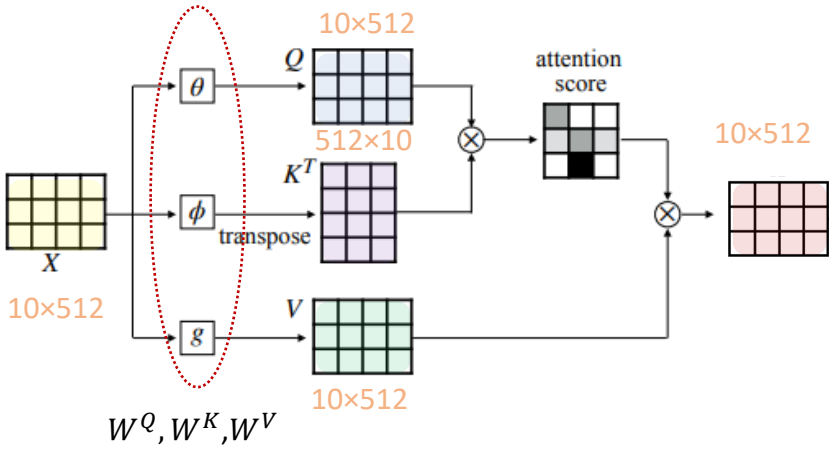
Multi-head Attention

- **Example**: Suppose $\boldsymbol{d_{model} = 512}$ and the input sequence has **10** tokens:



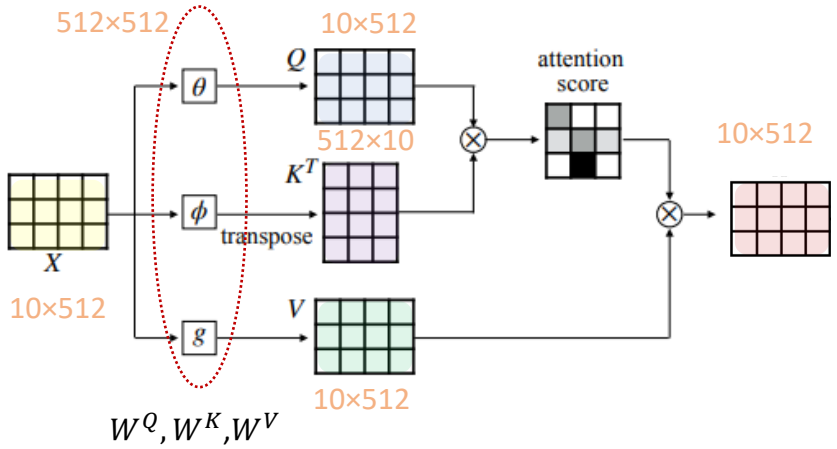
Multi-head Attention

- **Example**: Suppose $\boldsymbol{d_{model} = 512}$ and the input sequence has **10** tokens:
  - Input embedding, Q,K, and V each has a dimension of **10×512**



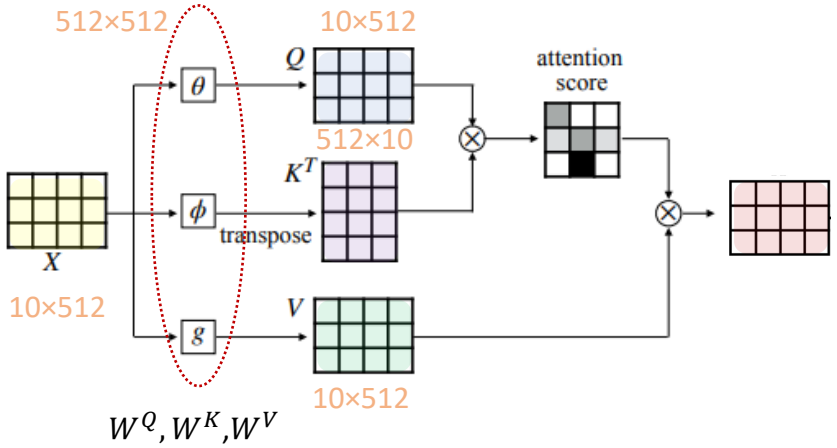
Multi-head Attention

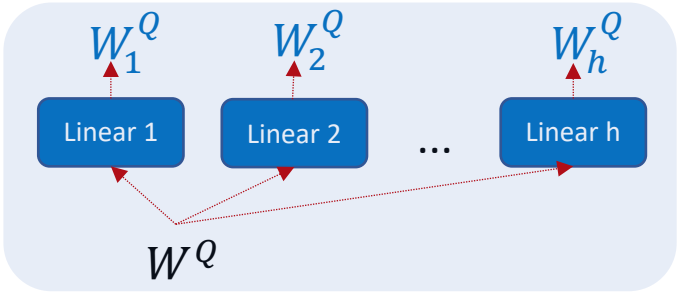
- **Example**: Suppose $\boldsymbol{d_{model} = 512}$ and the input sequence has **10** tokens:
  - Input embedding, Q,K, and V each has a dimension of **10×512**
  - With a single head:
    - W^Q, W^K, W^V have dimension of 512×512.

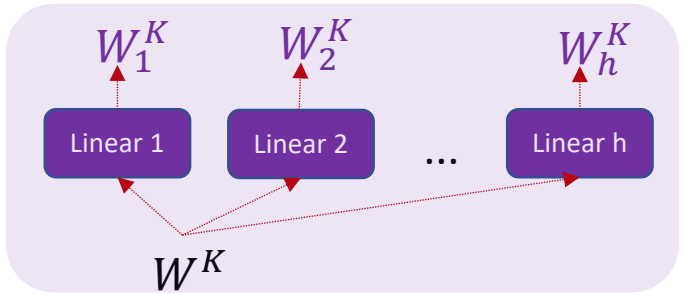


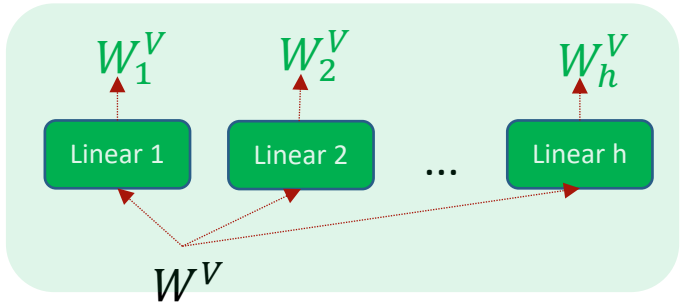
Multi-head Attention

- **Example**: Suppose $\boldsymbol{d_{model} = 512}$ and the input sequence has **10** tokens:
  - Input embedding, Q,K, and V each has a dimension of **10×512**
  - With a single head:
    - W^Q, W^K, W^V have dimension of 512×512 $= d_{model} \times d_{model}$.
  - With 8 heads:
    - $d_Q = d_K = d_V = \frac{d_{model}}{h} = \frac{512}{8}$
    - $W_i^Q = W_i^K = W_i^V = 512 \times 64 = d_{model} \times \frac{d_{model}}{h}$









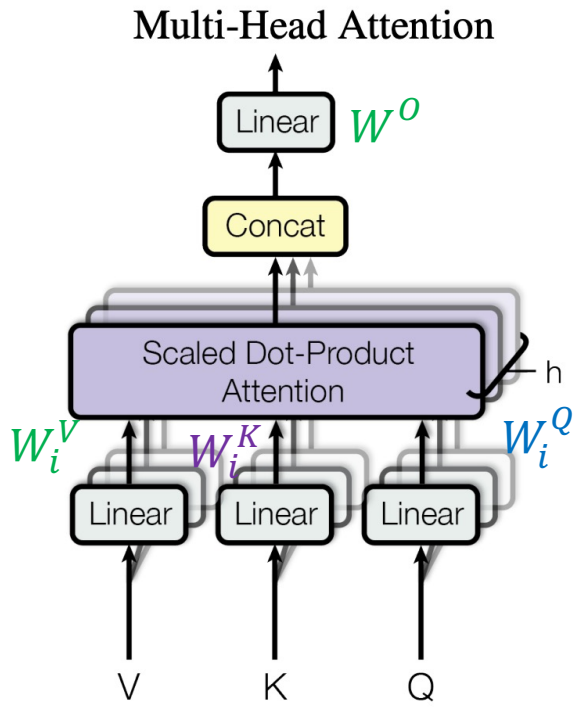
Multi-head Attention

- **Example**: Suppose $\boldsymbol{d_{model} = 512}$ and the input sequence has **10** tokens:
  - Input embedding, Q,K, and V each has a dimension of **10×512**
  - With a single head:
    - W^Q, W^K, W^V have dimension of 512×512 = $d_{model} \times d_{model}$.
  - With 8 heads:
    - $d_Q = d_K = d_V = \frac{d_{model}}{h} = \frac{512}{8}$
    - $W_i^Q = W_i^K = W_i^V = 512 \times 64 = d_{model} \times \frac{d_{model}}{h}$

d_{model} is divided by $\boldsymbol{h}$ to keep the computational cost roughly the same as that of single-head attention.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.



Multi head: a single weight matrix for each of **Q**, **K**, and **V** per head h_i

W_i^V W_i^K W_i^Q

Source: "Attention is all you need", Vaswani et al.

Interpretation of Heads

While not guaranteed that head functionality is human interpretable, in many popular architectures they do map onto linguistic concepts:

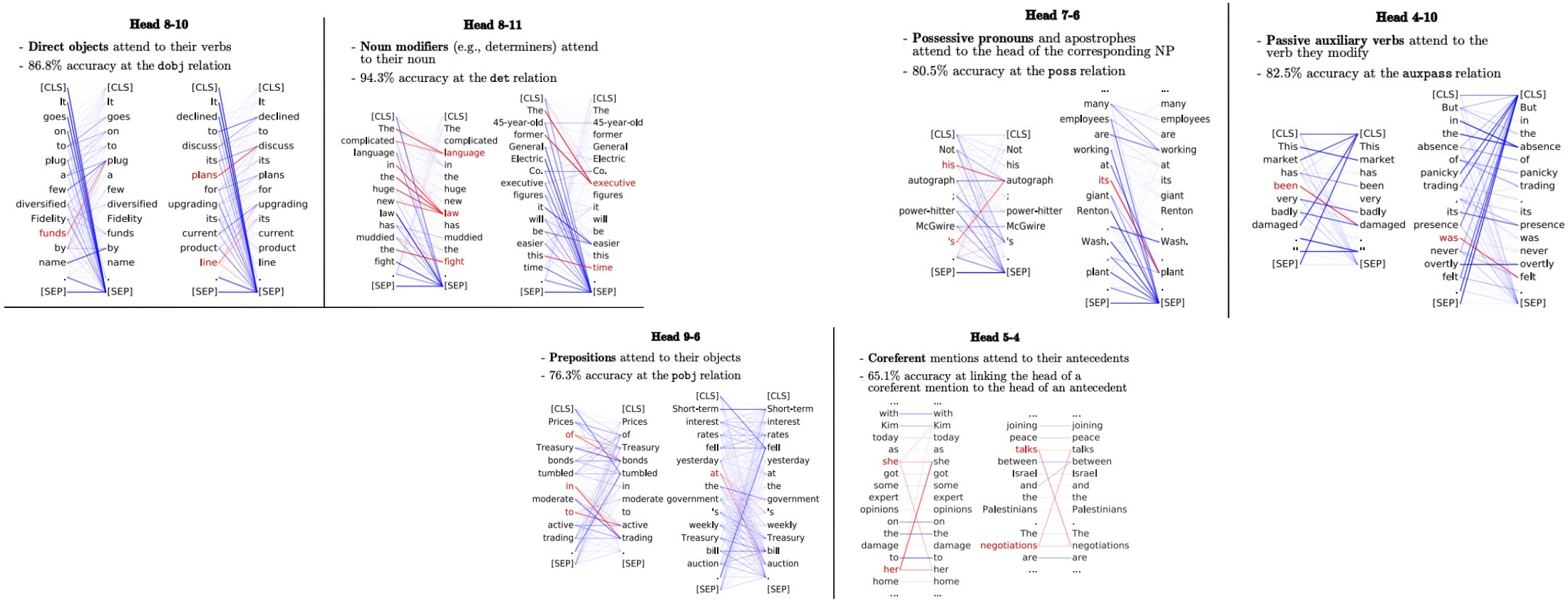
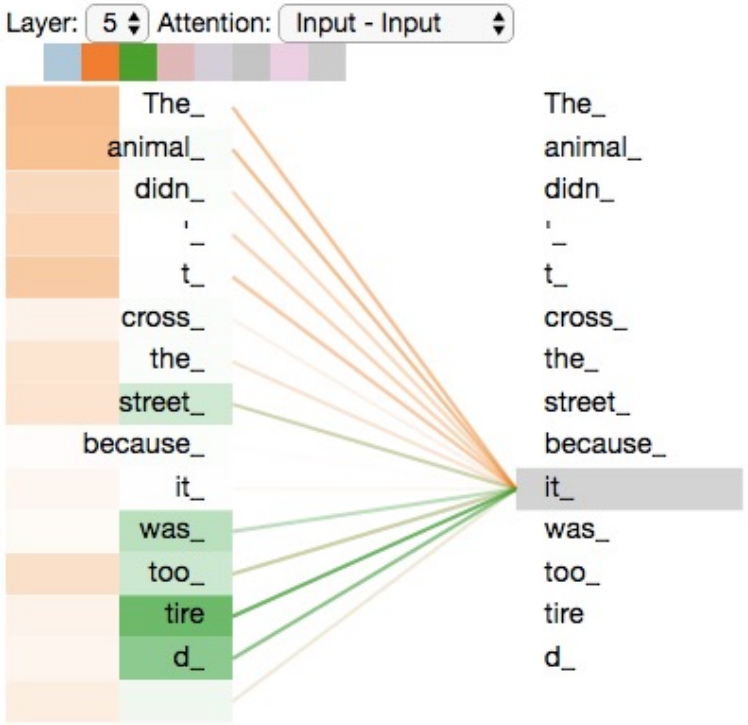


Figure 5: BERT attention heads that correspond to linguistic phenomena. In the example attention maps, the darkness of a line indicates the strength of the attention weight. All attention to/from red words is colored red; these colors are there to highlight certain parts of the attention heads' behaviors. For Head 9-6, we don't show attention to [SEP] for clarity. Despite not being explicitly trained on these tasks, BERT's attention heads perform remarkably well, illustrating how syntax-sensitive behavior can emerge from self-supervised training alone.

Source: What Does BERT Look at? An Analysis of BERT's Attention (2019)

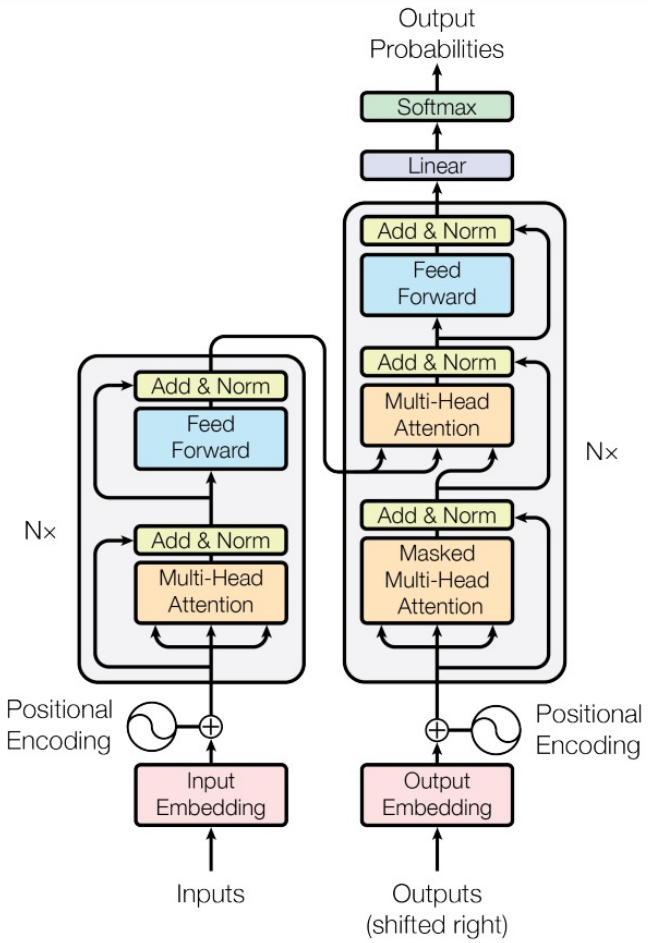
Interpretation of Heads

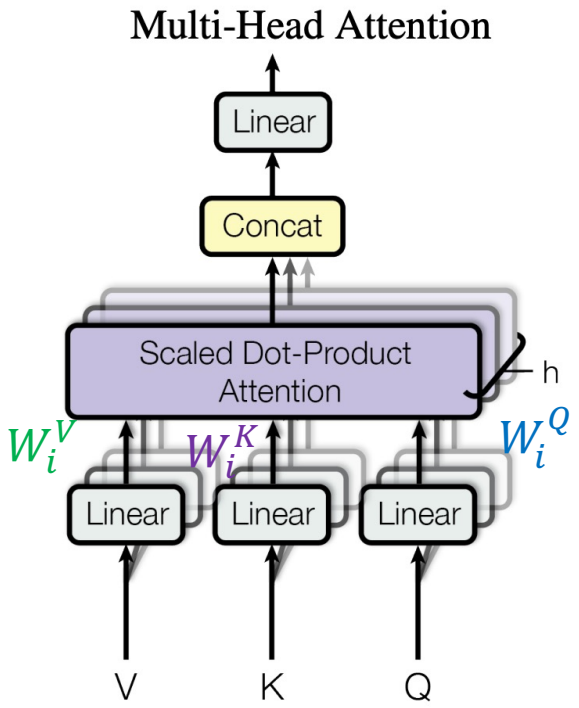


Source: *Jay Alammar's The Illustrated Transformer*

If the Query word is **it**, the first head focuses more on the words **the animal,** and the second head focuses more on the word **tired**. Hence, the final context representation will be focusing on all the words **the, animal** and **tired,** and thus is a superior representation as compared to the traditional way.

Multi-head Attention





Multi head: a single weight matrix for each of **Q**, **K**, and **V** per head h_i

W_i^V W_i^K W_i^Q

Note: The calculations of each head is done in parallel.

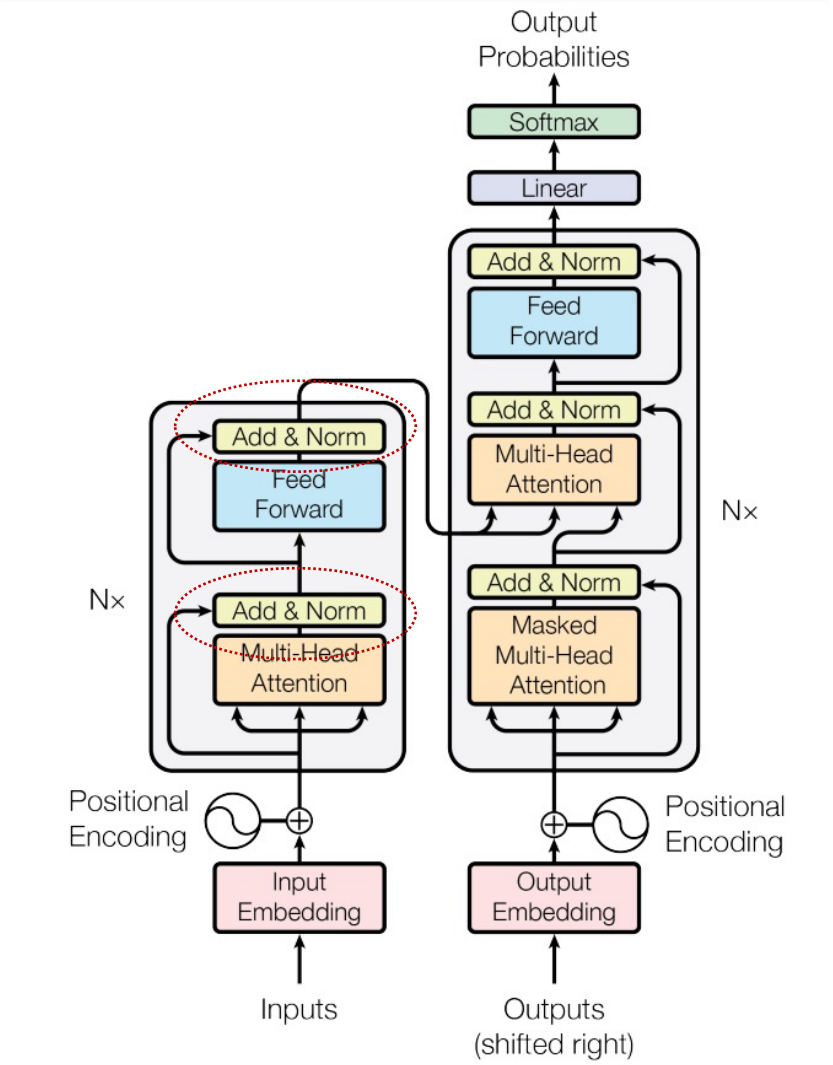
Why Not Use a Single Large Attention Head Instead of Multi-Head?

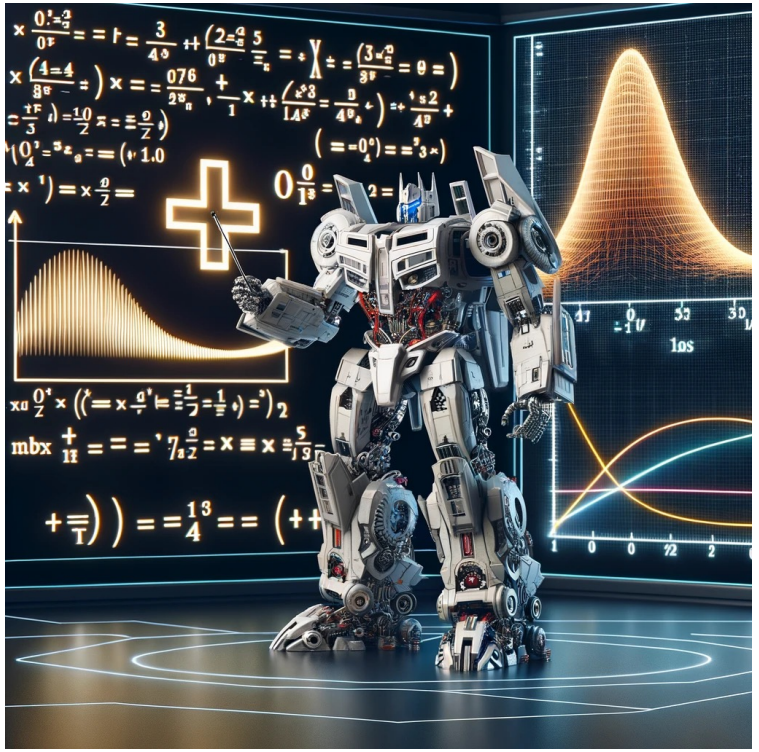
- **In Theory:**
  - A single head with full dimension d_model **can** learn attention patterns just like multiple heads.
  - But in **practice**, multi-head attention works **better and faster**.
- **Why Multi-Head Is Preferred**
  - **Better Optimization**
  - Smaller projections per head →
    Faster, more stable learning due to **fewer parameters per subspace**
- **Nonlinear Mixing**
  - Output of all heads is **concatenated and linearly transformed**,
    allowing for **richer combinations** than one big head.
- **Parallelization Efficiency**
  - All heads:
    - Operate independently, are easy to batch, and map well to **GPU/TPU architecture**

More Heads Doesn't Always Mean Better

1. **"Are Sixteen Heads Really Better than One?"** by Paul Michel, Omer Levy, and Graham Neubig (2019)
    1. This study found that many attention heads can be removed after training without significantly affecting performance, suggesting redundancy among heads.
2. **"Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned"** by Elena Voita et al. (2019):
    1. The authors observed that only a few attention heads are crucial for model performance, with specialized heads playing significant roles, while others can be pruned with minimal impact.
3. **"Finding the Pillars of Strength for Multi-Head Attention"** by Jinjie Ni et al. (2023):
    1. This research proposed methods to identify and retain the most representative and distinctive attention heads, addressing issues of redundancy and over-parameterization in multi-head attention.

Add & Norm


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)



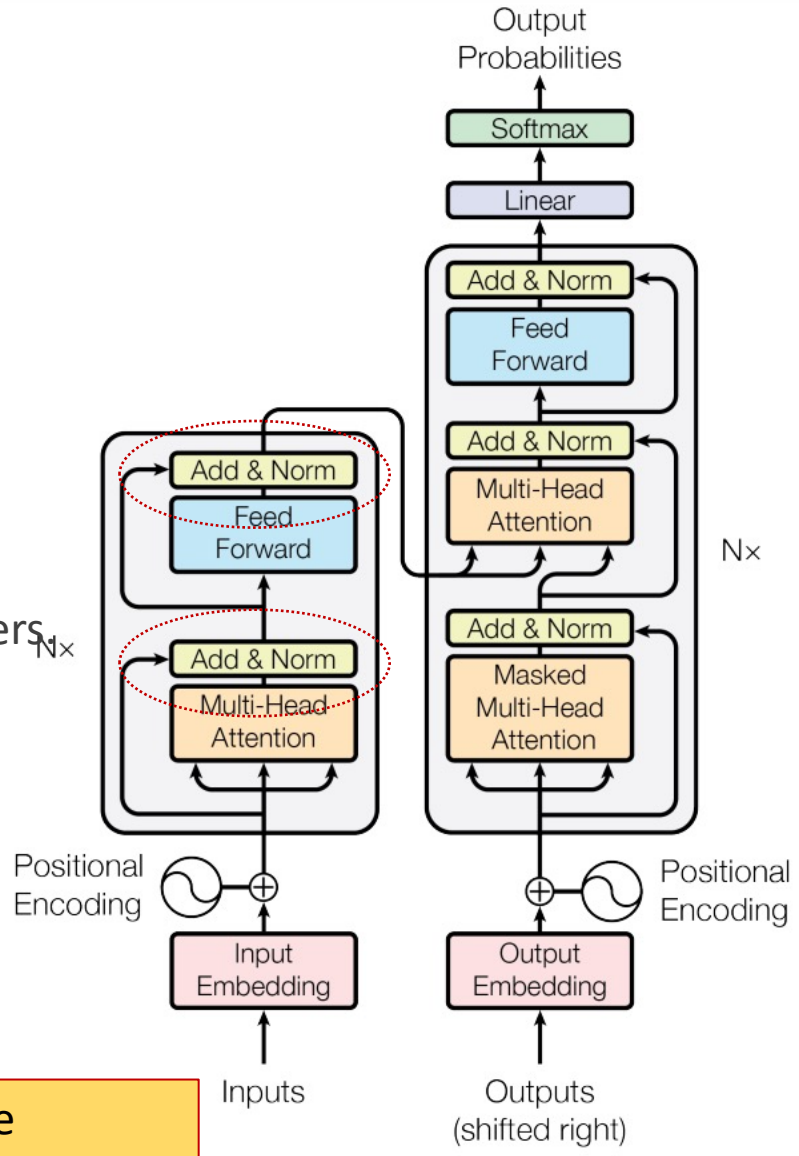
Add & Norm

- Residual connection ("**Add**")
  - helps **gradients** flow through the network without **diminishing** over layers
  - Encourages **feature reuse**.
- Layer normalization ("**Norm**")
  - Normalizes the outputs across the **feature dimension** (not batch)..
  - Helps with training stability and convergence speed.
  - Standardization ensures the gradients do not become **too dependent** on the **scale** of inputs or weights.
  - Mitigates **internal covariate shift**.
  - Layer normalization makes the network **less sensitive** to the scale of parameters. Otherwise, very large or very small weights can lead to very large or small activations, respectively, which can adversely affect the gradients.

Standardization $\hat{x_i} = \frac{x_i - \mu}{\sigma}$

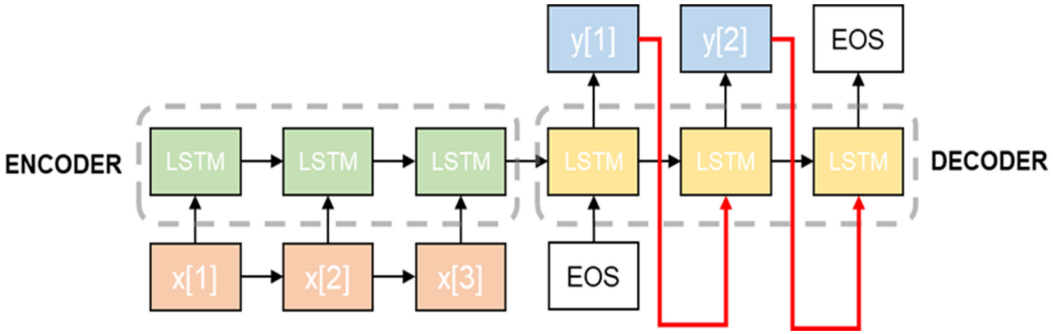
Scale and shift $y_i = \gamma \hat{x_i} + \beta$

New mean is 0 and variance is 1

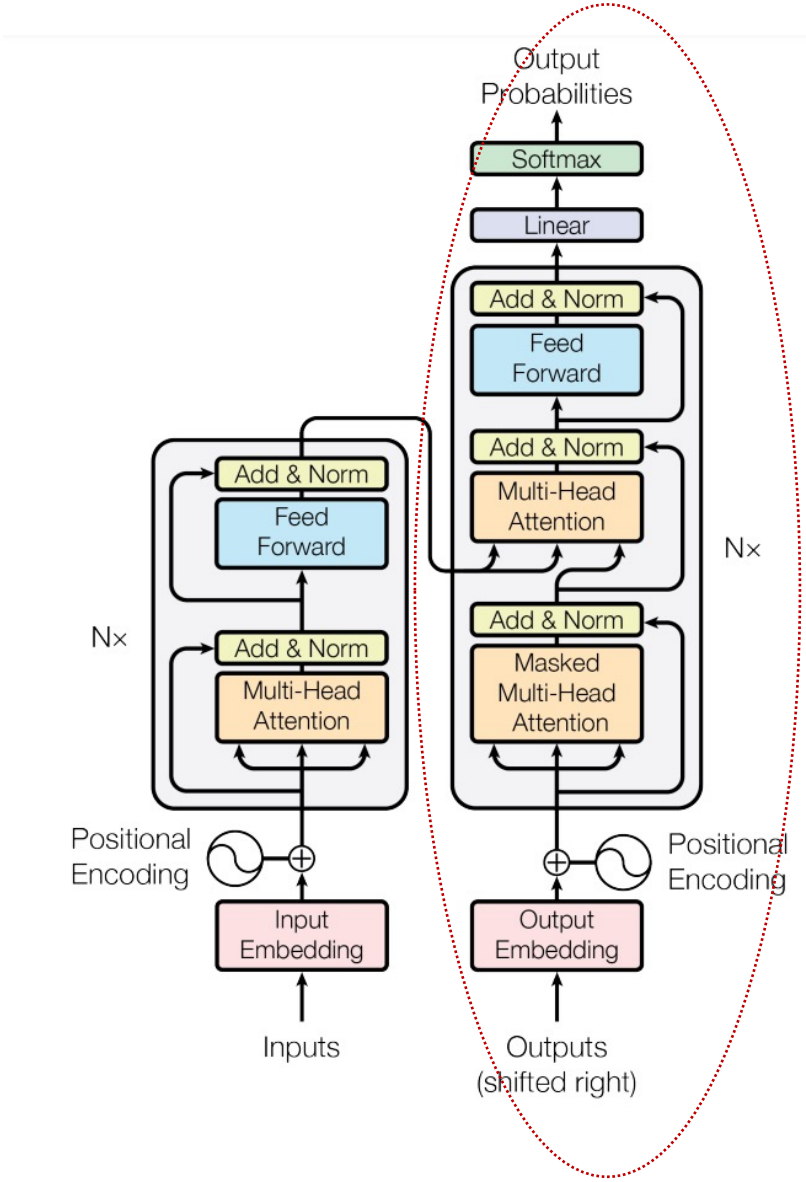


- Together, "Add and Norm" helps the Transformer effectively learn and propagate gradients throughout the network's depth, leading to more stable and faster training.
- This design pattern is applied consistently after each sub-layer (like after Multi-Head Attention and after the Feed Forward layer) before passing the output to the next layer.

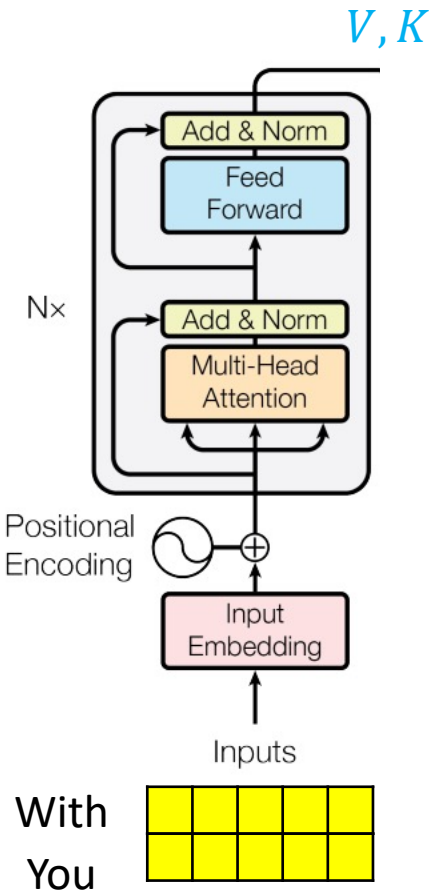
Decoder



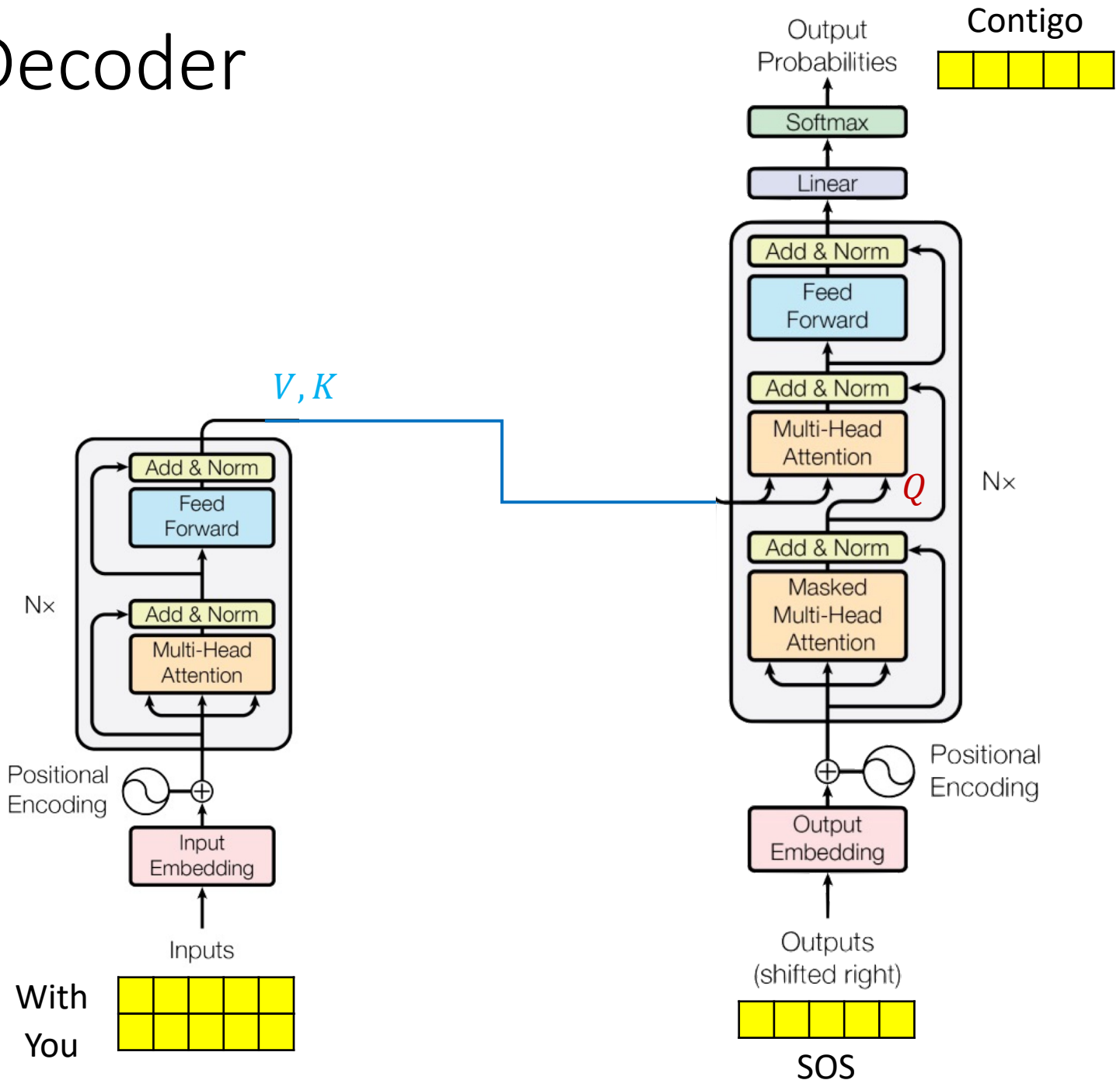
Source: "Electricity load forecasting using advanced feature selection and optimal deep learning model for the variable refrigerant flow systems", Woohyun Kim et al.



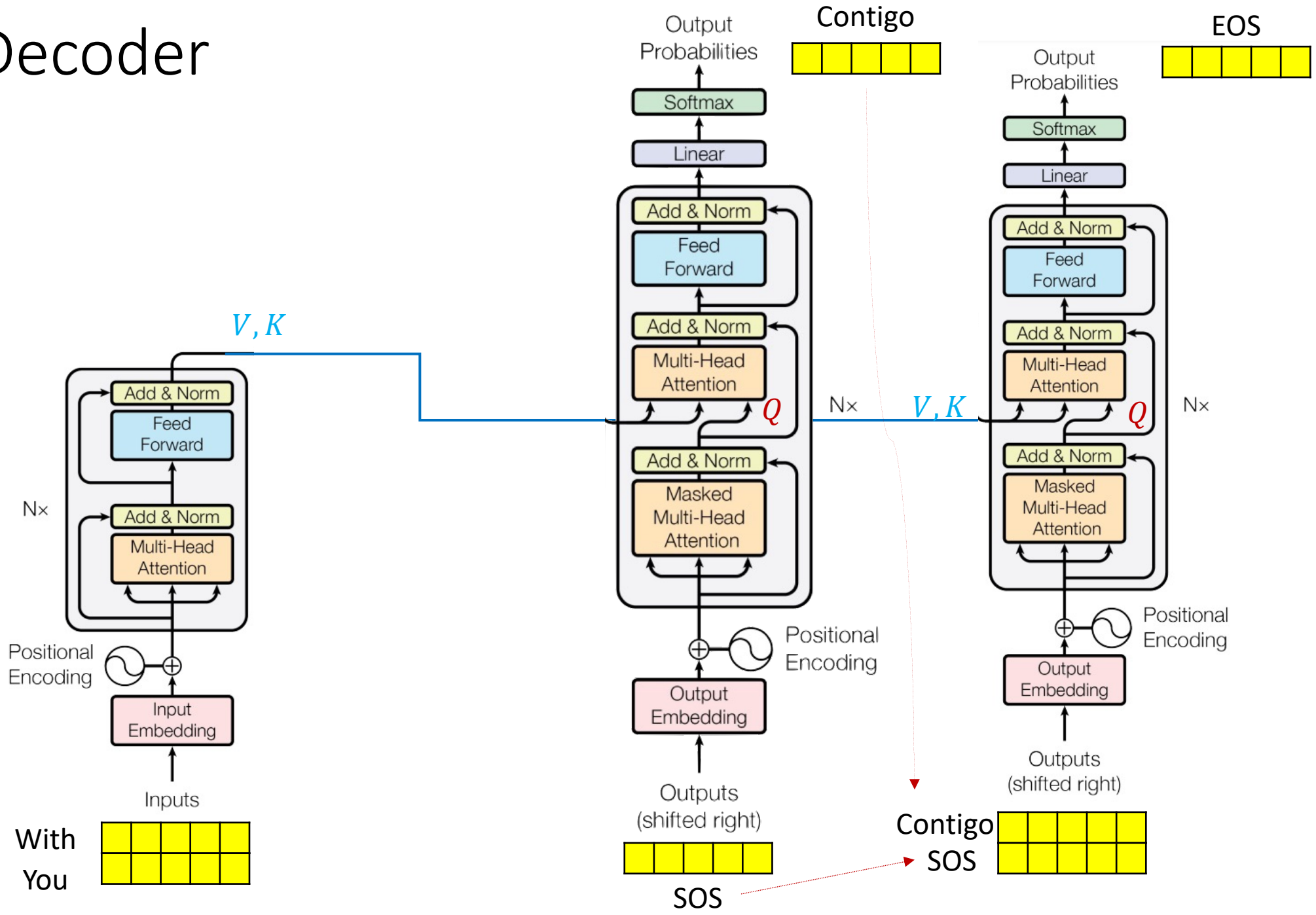
Decoder


V, K
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Input Embedding
Inputs
With
You

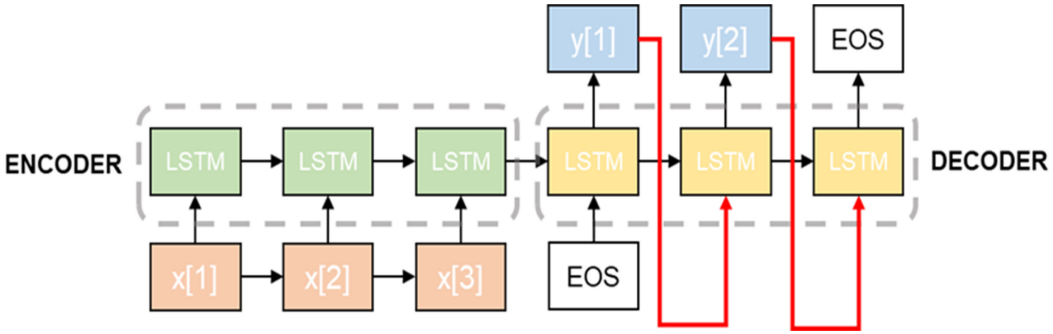
Decoder


Output Probabilities
Contigo
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Q
Add & Norm
Masked Multi-Head Attention
Nx
Positional Encoding
Output Embedding
Outputs (shifted right)
SOS
V, K
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Input Embedding
Inputs
With
You

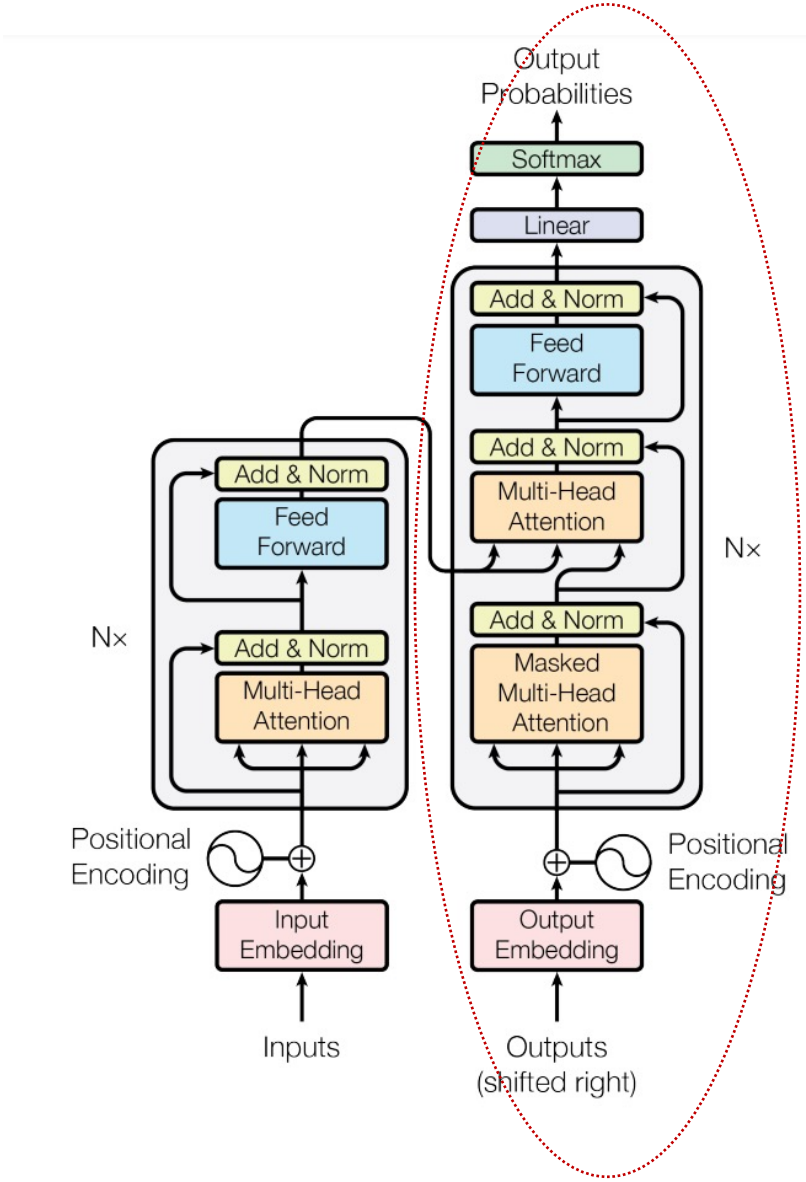
Decoder


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Positional Encoding
Output Embedding
Outputs (shifted right)
Nx
V, K
Q
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Positional Encoding
Output Embedding
Outputs (shifted right)
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Positional Encoding
Input Embedding
Inputs
Nx
With
You
Contigo
SOS
Contigo
EOS
SOS

Decoder



Source: "Electricity load forecasting using advanced feature selection and optimal deep learning model for the variable refrigerant flow systems", Woohyun Kim et al.



Non-Auto Regressive Decoding

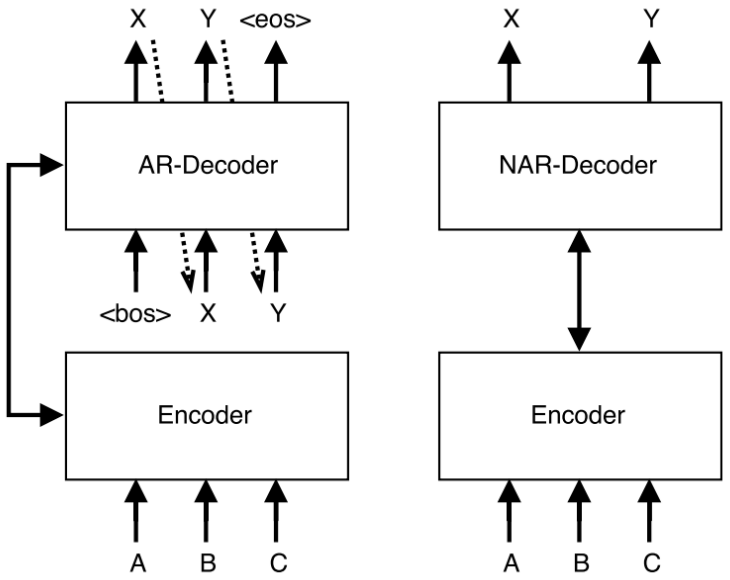
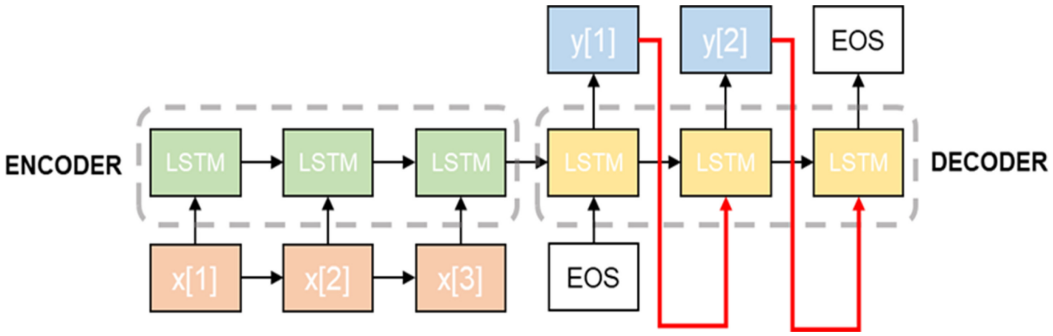


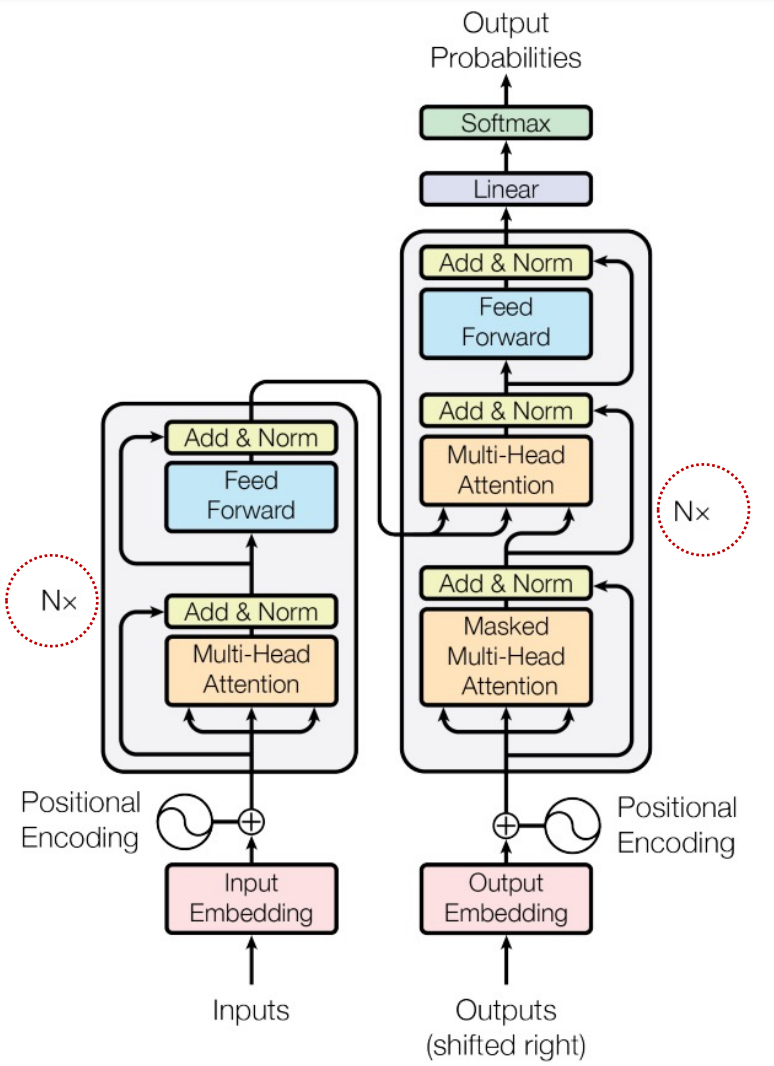
Figure 1: Translating "A B C" to "X Y" using autoregressive and non-autoregressive neural MT architectures. The latter generates all output tokens in parallel.

Source: "*Non-Autoregressive Neural Machine Translation", Jiatao Gu* et al.

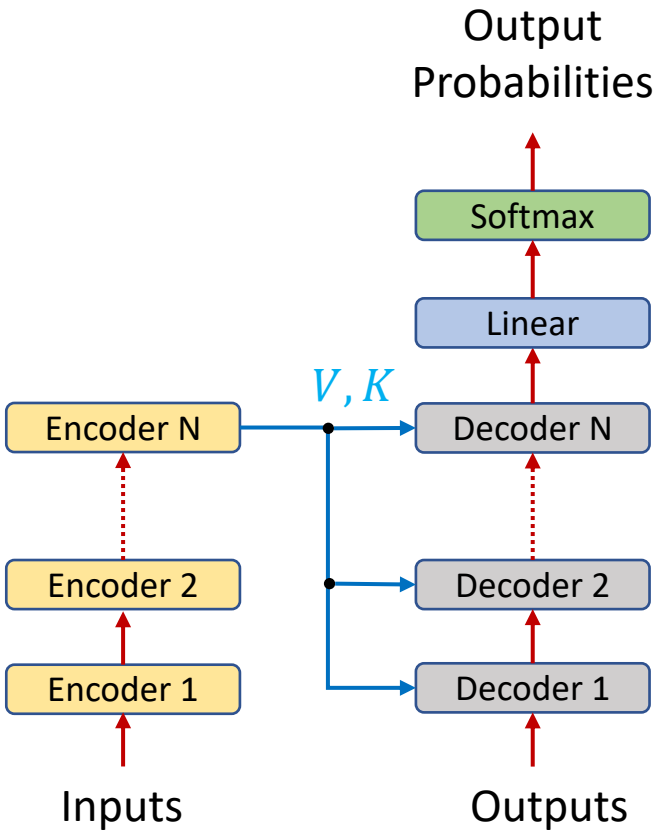
Stacked Layers

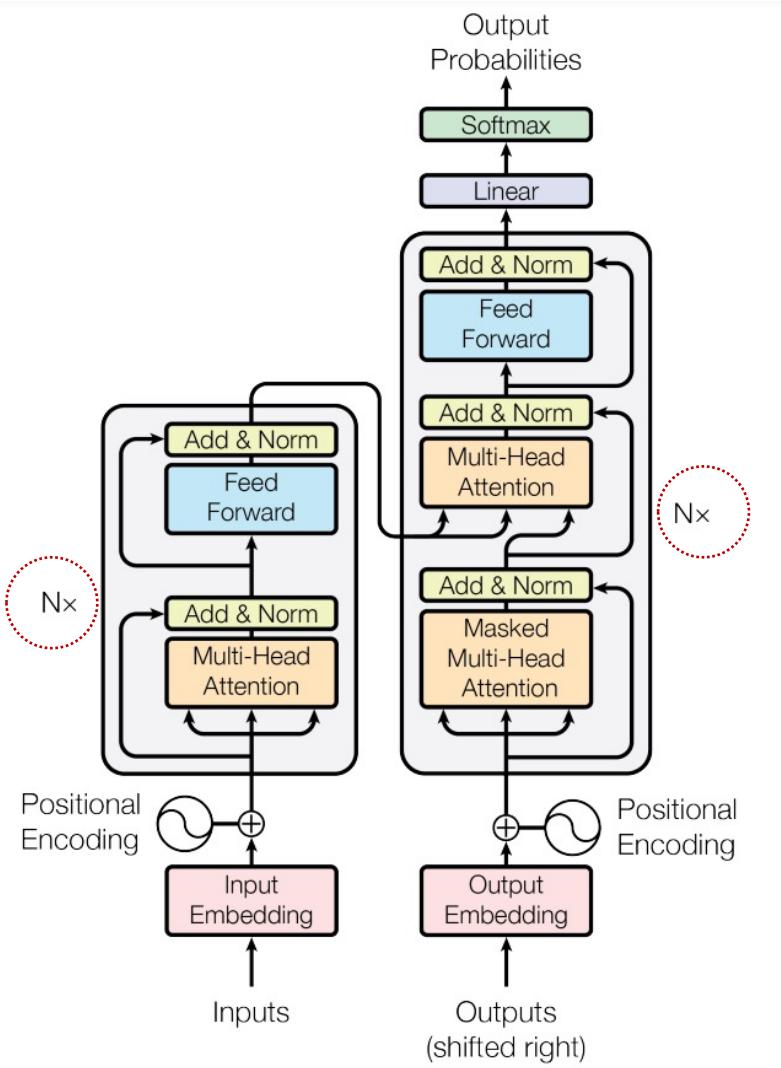


Source: "Electricity load forecasting using advanced feature selection and optimal deep learning model for the variable refrigerant flow systems", Woohyun Kim et al.



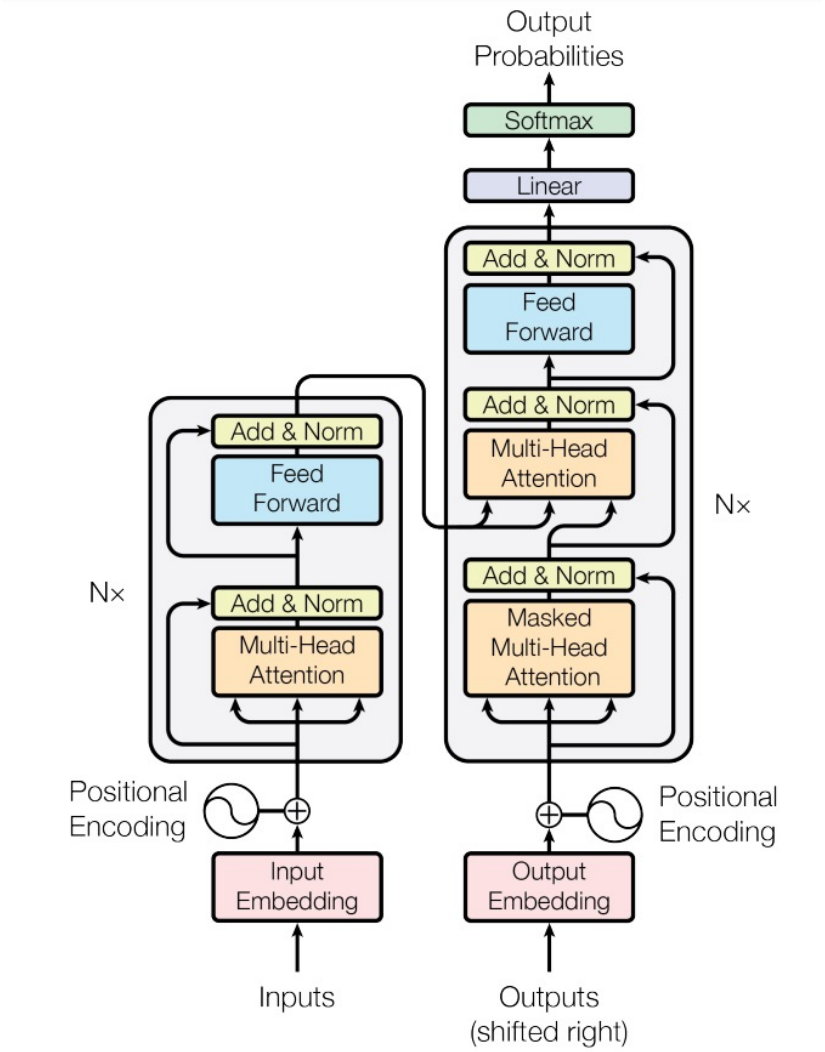
Stacked Layers





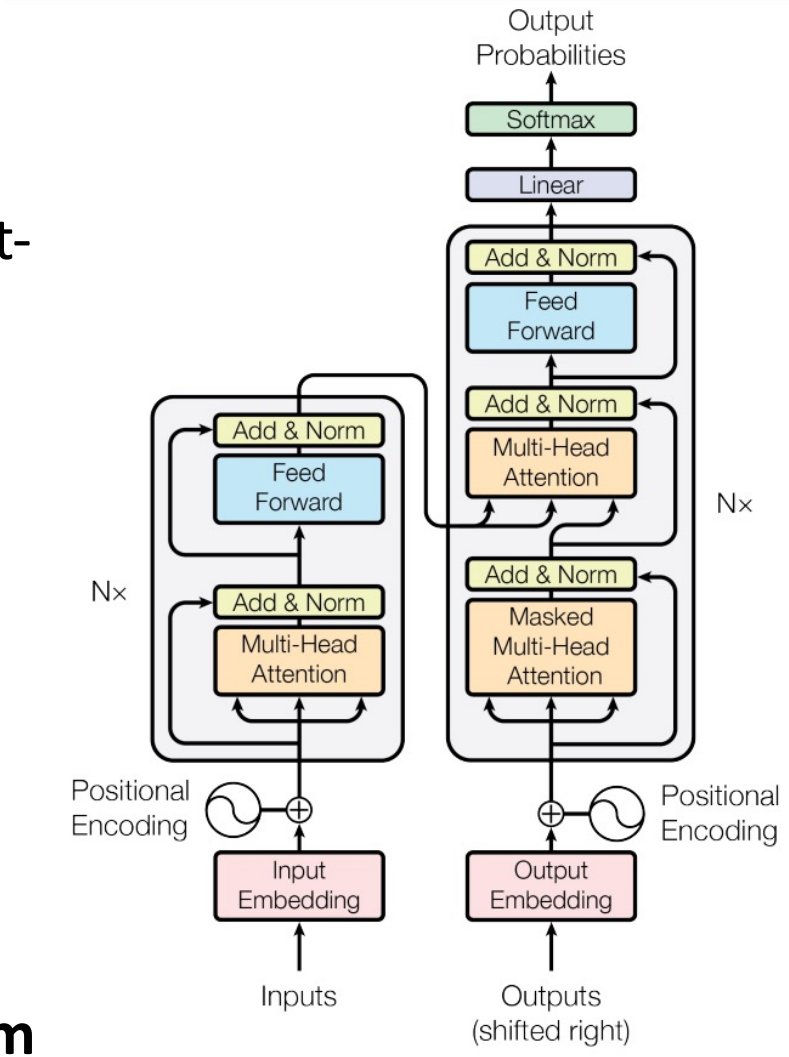
Parallelization in Transformers

- **Why Transformers Parallelize Well?**
  - No recurrence (unlike RNNs)
  - All tokens attend to each other simultaneously
  - Highly optimized for **modern hardware**



Parallelization in Transformers

- **Attention Head Parallelism**
  - Each attention head computes its own **Q/K/V**, dot-products, and output independently.
  - Heads operate on **disjoint subspaces** → fully parallelizable.
  - Efficient on GPUs/TPUs via **batched matrix operations**.
- **Sequence Parallelism**
  - Long input sequences can be **split into chunks**.
  - Each chunk can be processed in **parallel** on separate devices (GPUs).
  - Often used in **pipeline** or **tensor model parallelism** strategies.



The Problem — Long Sequences

- Transformers have **quadratic time & memory** complexity: $O(n^2)$
  - Sequences like **books, code, or genomes** can be **10k+ tokens**
  - Solution: **Split into chunks** to fit in memory and allow parallelization
- **Naive Chunking:**
  - Input split into non-overlapping blocks : [Token 0–999], [1000–1999], etc.
  - **Problem:**
    - **No attention across chunks**
    - Tokens **lose access to global context**
    - Model can't relate distant concepts

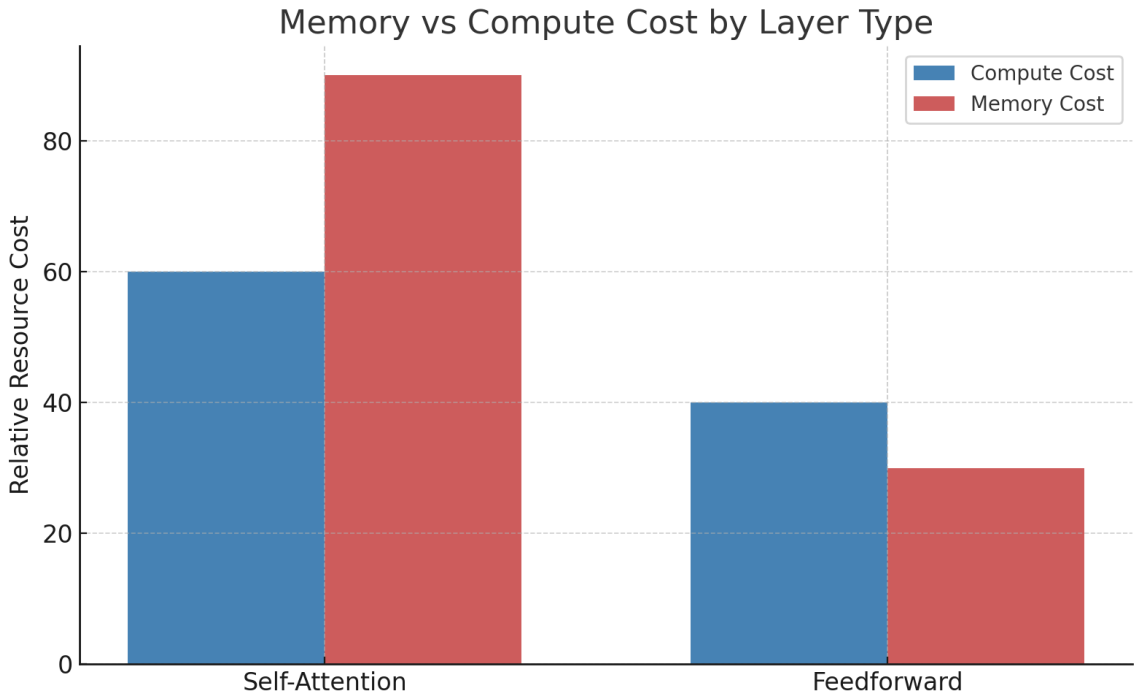
Solutions to Preserve Context

- **Overlapping Chunks**
  - Add **token overlap** between chunks
  - Example: [0–999], [950–1949]
  - Preserves partial context
- **Recurrent Memory**
  - Pass **summary states** across chunks (Transformer-XL)
  - Maintains long-term dependencies
- **Sparse or Global Attention**
  - Use **special tokens** that attend across chunks (e.g., [CLS] tokens in Longformer/BigBird)
  - Global views with local efficiency

Memory & Compute Bottlenecks in Transformers

- **Why Memory Becomes a Bottleneck**
  - **Attention Complexity:**
  - Self-Attention:
    - n: sequence length
    - d: embedding dimension
  - Memory grows **quadratically** with sequence length
    - **Backpropagation Requires:**
      - Storing activations from:
        - Q, K, V
        - Softmax scores
        - Intermediate outputs

$$\text{Self-Attention: } O(n^2 \cdot d)$$



- **Memory cost (red)** is especially high for self-attention due to storing Q, K, V, softmax maps, and intermediate outputs.
- **Compute cost (blue)** is high for both layers but more efficient for feedforward because of simple dense ops.

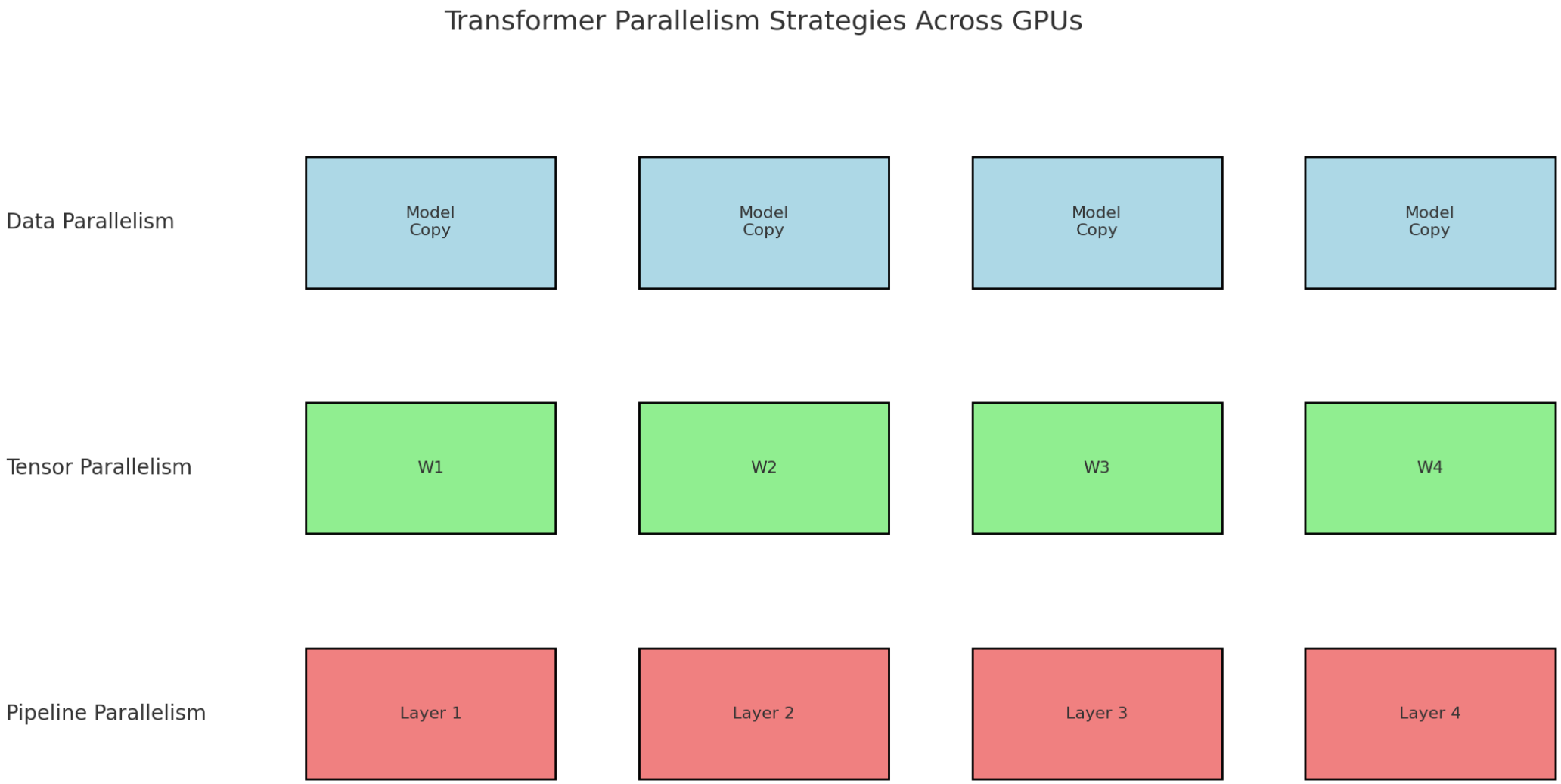
Parallelism Strategies in Transformers

- **Why Parallelism Is Needed**
  - Transformer models are **large** (hundreds of millions to billions of parameters)
  - Training must be **distributed across multiple devices**
  - Parallelism enables scaling and faster training
- **Data Parallelism**
  - Each GPU gets a **different batch of data**
  - All GPUs have the **same model**
  - Gradients are **averaged** across devices
    - Easy to implement
    - Doesn’t reduce model size per device
- **Tensor Parallelism**
  - Split large **matrices (weights)** across GPUs
  - Each device computes a **slice** of Q, K, V or feedforward ops
    - Memory efficient
      Requires synchronized matmuls and communications

Parallelism Strategies in Transformers

- **Pipeline Parallelism**
  - Split model **layers** across GPUs (vertical partitioning)
  - One batch **flows through** devices like an assembly line
  - Good for very deep models
    Requires careful scheduling, may have idle time

Parallelism Strategies in Transformers



The Problem with Standard Attention

- Standard dot-product attention:
  - Requires building the full **attention matrix** of size n×n
  - This uses **O(n^2)** memory
  - Makes long sequences very expensive to train

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Component	Memory Cost
Q, K, V	$O(n \cdot d)$
$QK^\top$	$O(n^2)$
Softmax map	$O(n^2)$
Output	$O(n \cdot d)$

Backpropagation needs to **store all of these** — very inefficient.

Flash Attention: The Solution

- Introduced in 2022
  - “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”
- **Reformulates attention** to:
  - Avoid storing the full QK^T
  - Fuse softmax + matmul into one CUDA kernel
  - Compute attention **block-by-block** on GPU registers

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$