# Machine Learning

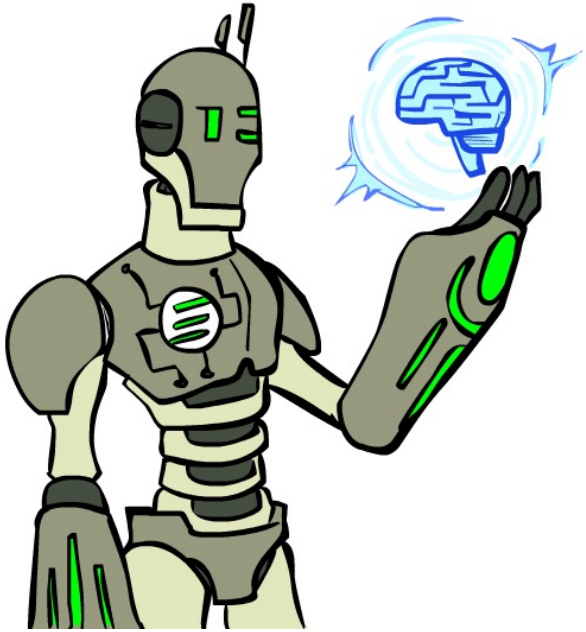
## Chapter 12: Sequence to Sequence Learning

**Fall 2021**

**Instructor: Xiaodong Gu**

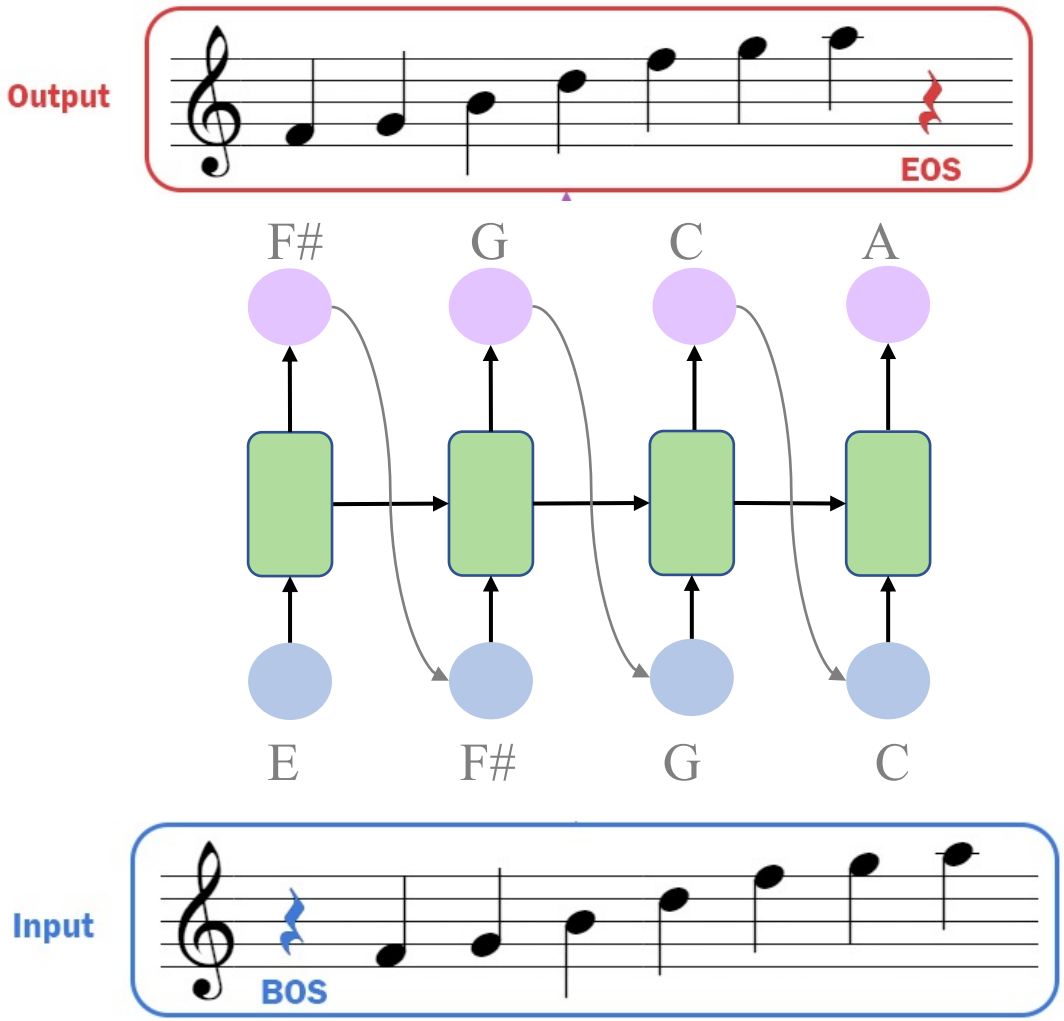
# Today

- RNN Encoder-Decoder
- Attention
- Transformer
- Pretrained Language Model




# Sequence-to-Sequence Learning

# Recall: Sequence Generation Using RNNs



# Conditional Sequence Generation

- generate a (**target**) sequence conditioned on a given **source** sequence.

**ASR**

Generator → How are you

**Translation**

机器学习

Generator → Machine Learning

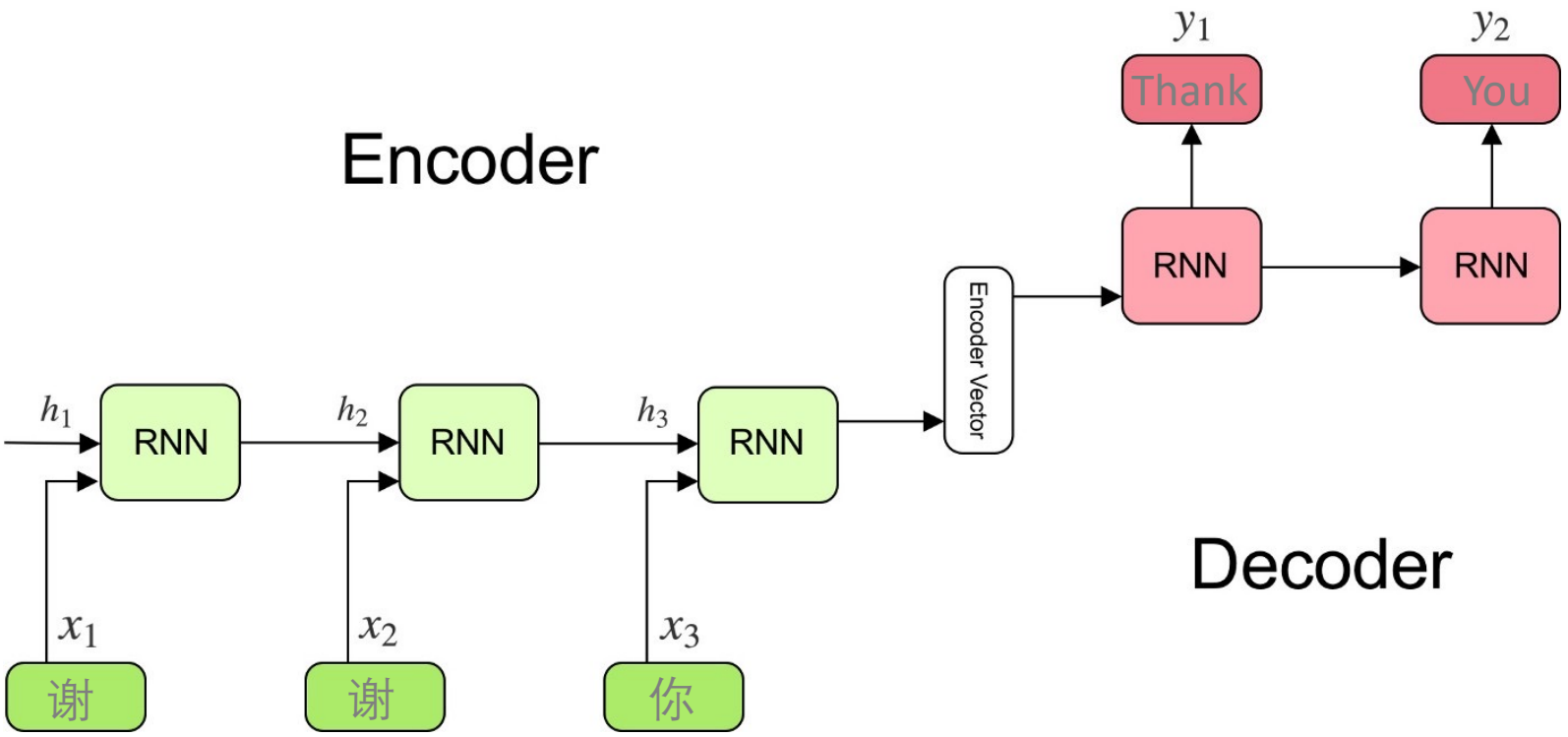
**Dialogue**

How are you

Generator → I am fine

# RNN Encoder-Decoder

- given x = $(x_1, \cdots, x_{T_x})$, generate y = $(y_1, \cdots, y_{T_y})$

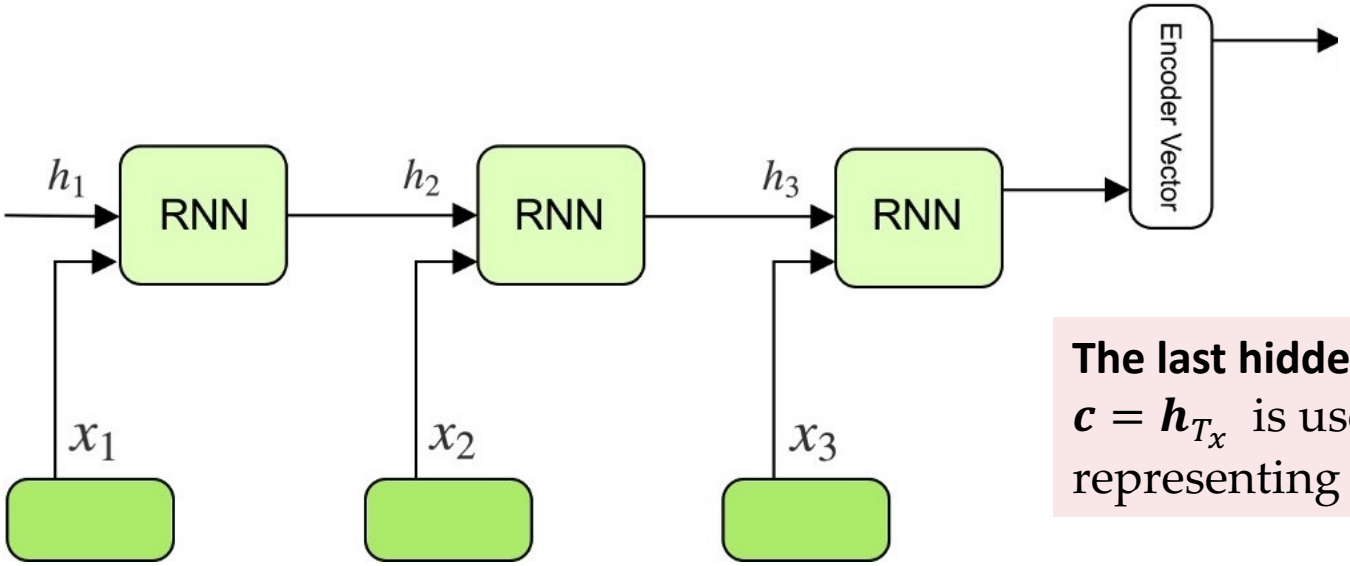


# Encoder

- An RNN which learns a dense representation of a sequence.
- Compresses a sequence of tokens into a context vector c.

$$\boldsymbol{h}_{\mathrm{t}} = \mathrm{RNN}_{\mathrm{in}}(x_t, \boldsymbol{h}_{\mathrm{t-1}})$$

e.g., $h_t = \sigma(W_{input}x_t + W_{rec}h_{t-1} + b)$




**The last hidden state** $\boldsymbol{c} = \boldsymbol{h}_{T_x}$ is used for representing $x$

# Decoder

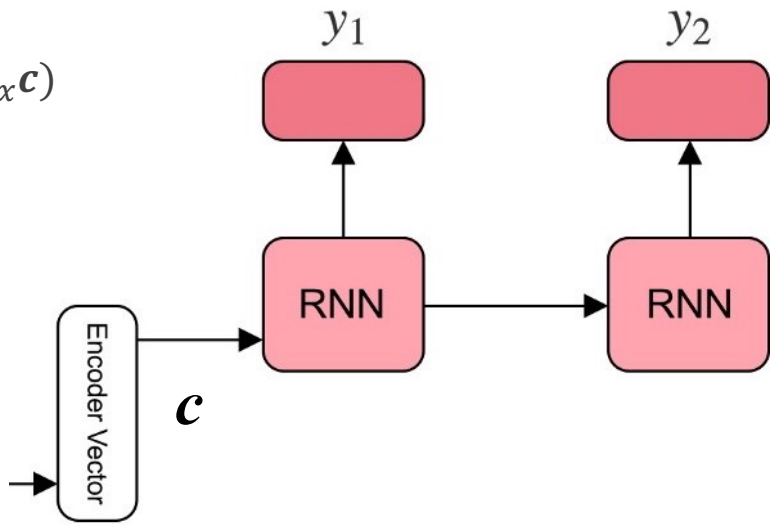
- An RNN which generates an sequence conditioned on the intermediate representation.
- Sequentially predicts the next token $y_i$ given the context vector $\boldsymbol{c}$ and the hidden state of past-generated sequence.

$$\boldsymbol{s}_i = \text{RNN}_{\text{out}}(y_{i-1}, \boldsymbol{s}_{i-1}, \boldsymbol{c})$$

e.g., $s_i = \sigma(W'_{input} y_{i-1} + W'_{rec} \boldsymbol{s}_{i-1} + W_{ctx} \boldsymbol{c})$

$$p(y_i \mid y_{<i}, \boldsymbol{c}) = \text{softmax}(g(\boldsymbol{s}_i))$$

e.g., $g = V^T(W_1 y_{t-1} + W_2 s_t + W_3 c)$



# Training

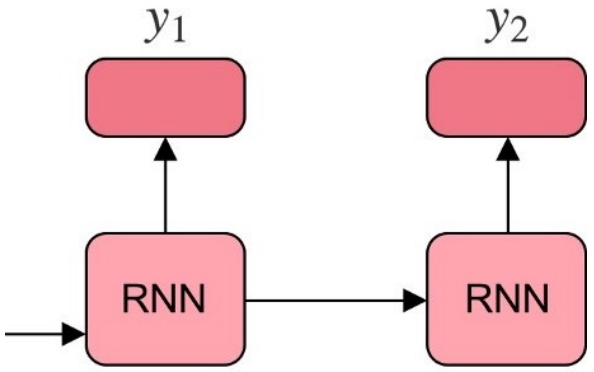
**Data:** $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$,

where $x^{(\ell)} = (x_1, \ldots x_{Tx})$ and $y^{(\ell)} = (y_1, \ldots y_{Ty})$.

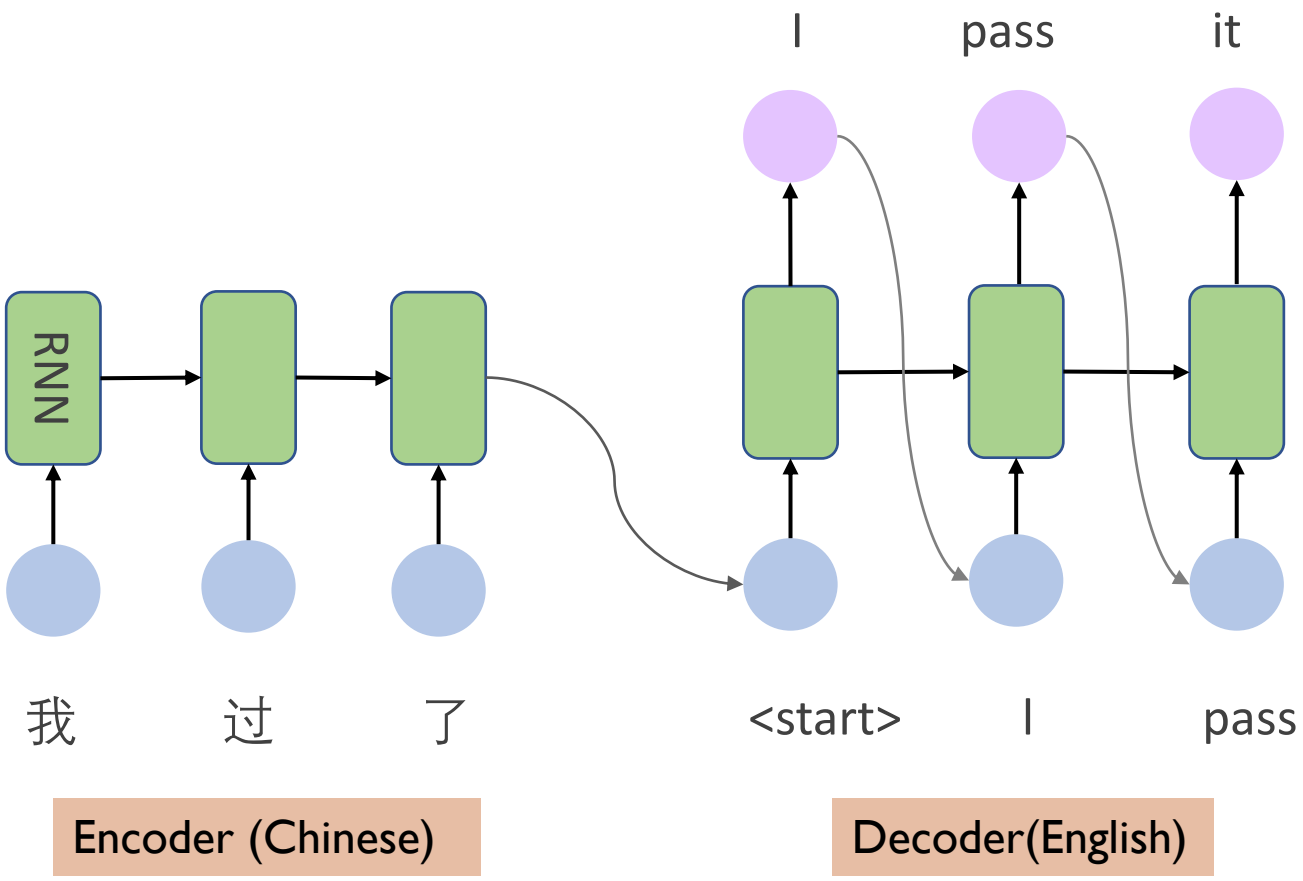
**Loss Function** – minimize the cross-entropy loss:

$$L(\theta) = -\frac{1}{N}\sum_{\ell=1}^{N}\sum_{t=1}^{Ty}\log p_\theta(y_t^{(\ell)}|x^{(\ell)})$$

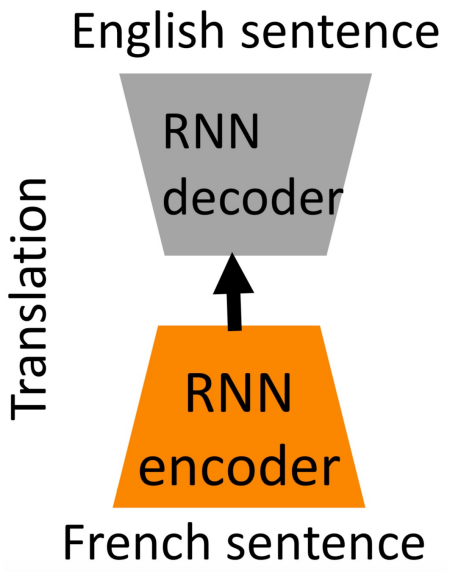
**Optimization** – gradient descend

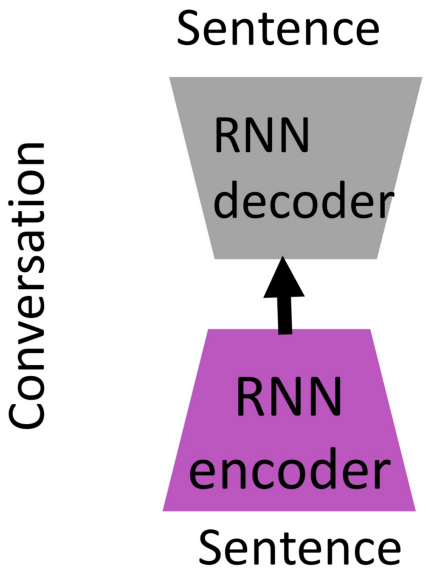


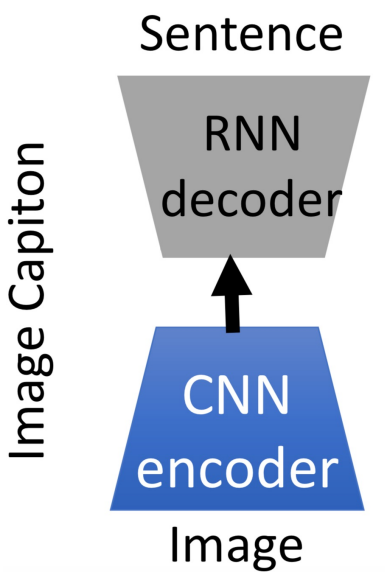
# Example: Machine Translation



# Applications



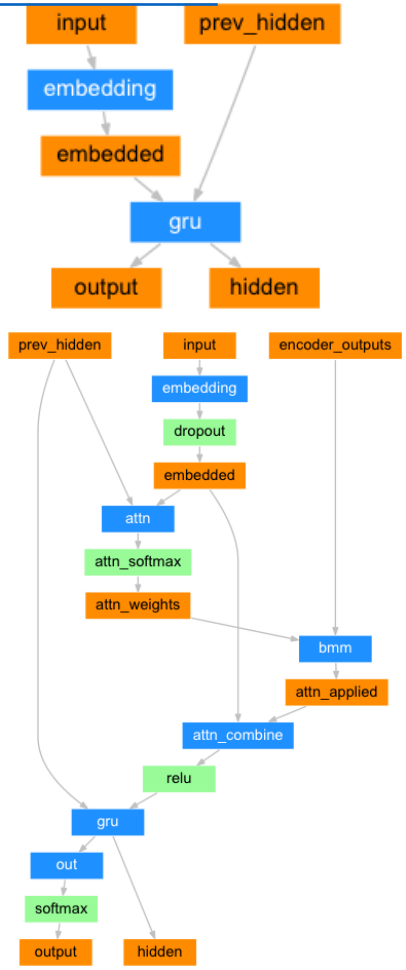



# Demo: Machine Translation by PyTorch

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

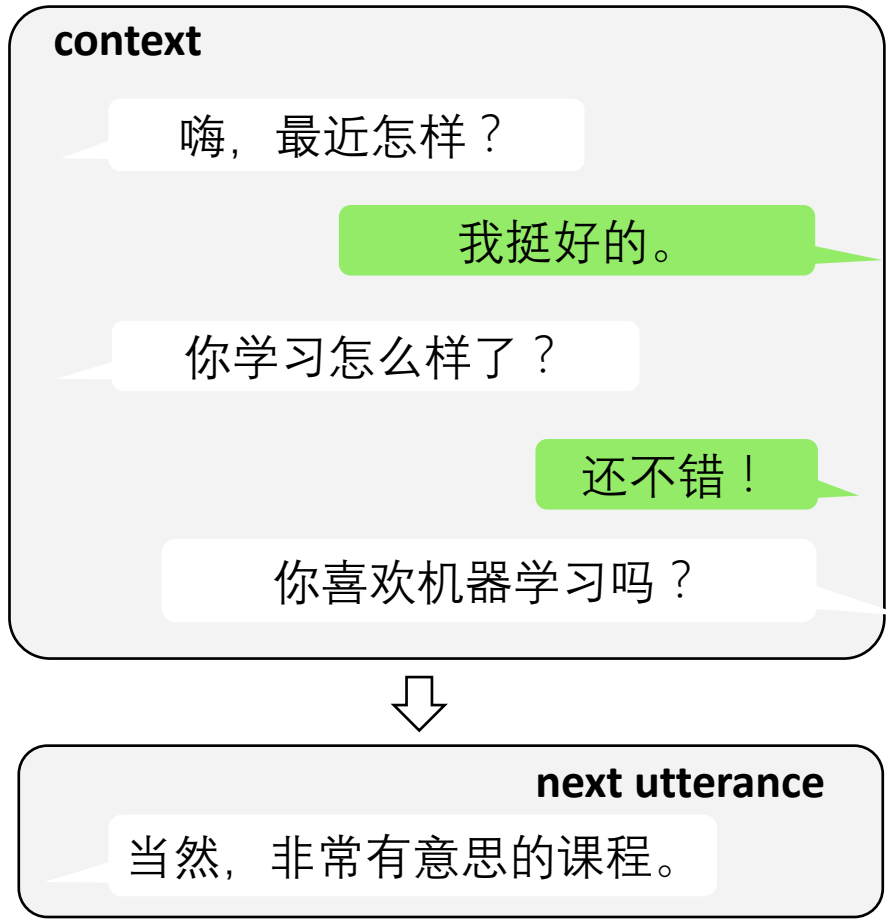
```
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size):
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.LSTM(hidden_size, hidden_size)
    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.rnn(embedded, hidden)
        return output, hidden
```

```
class Decoder(nn.Module):
    def __init__(self, hidden_size, vocab_size):
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.rnn = nn.LSTM(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, vocab_size)
    def forward(self, input, c, hidden):
        input = self.embedding(input).view(1, 1, -1)
        output, hidden = self.rnn([input,c], hidden)
        output = self.out(output[0]))
        return output, hidden
```



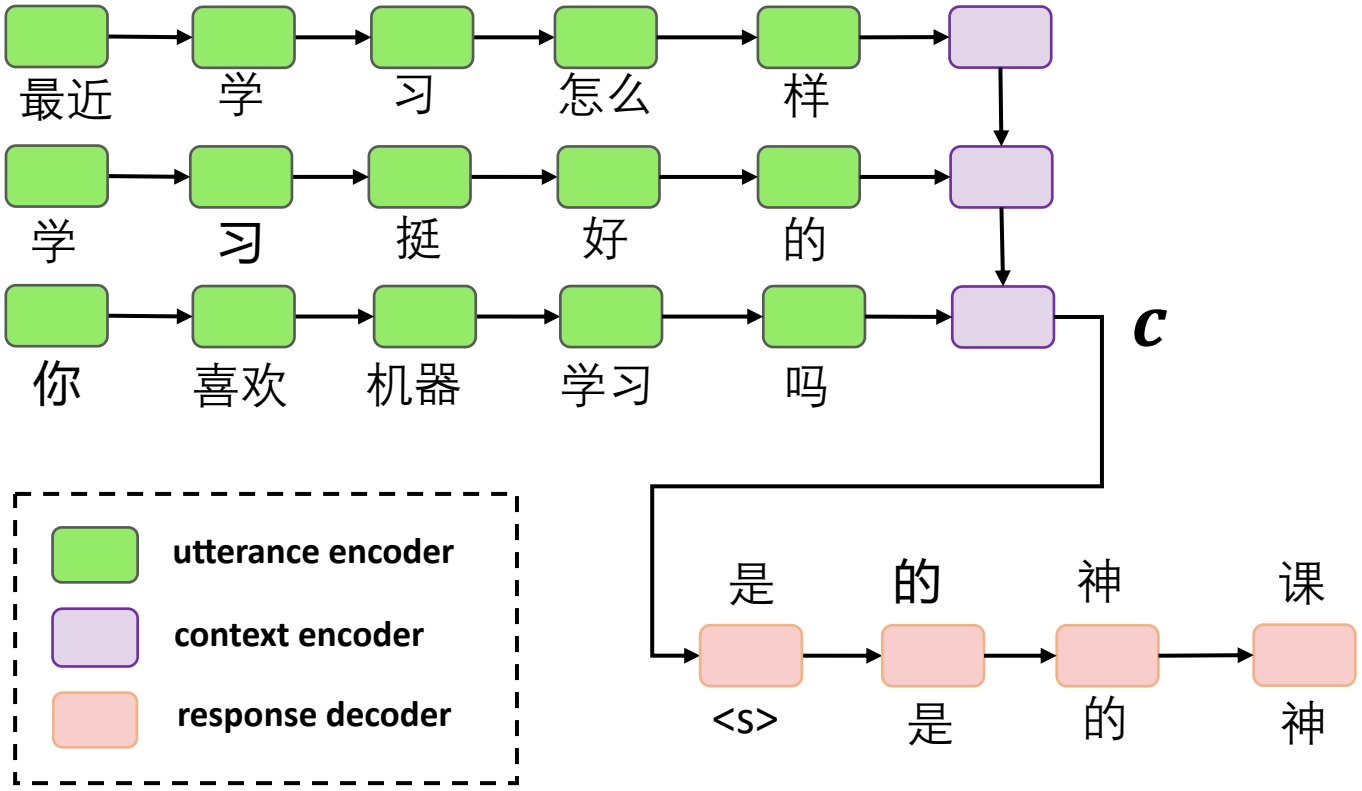
# Example: Chatbot

- Open-Domain Dialog Generation



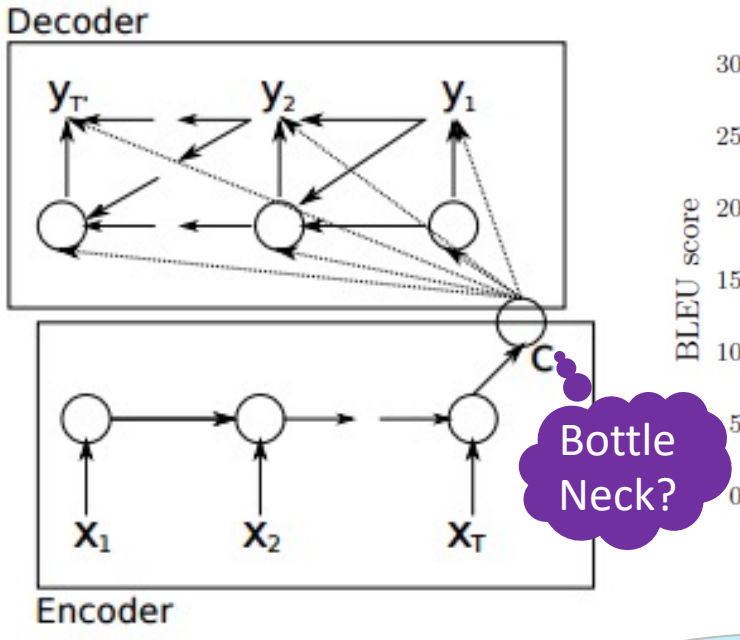
# Example: Chatbots

- Hierarchical RNN Encoder-Decoder

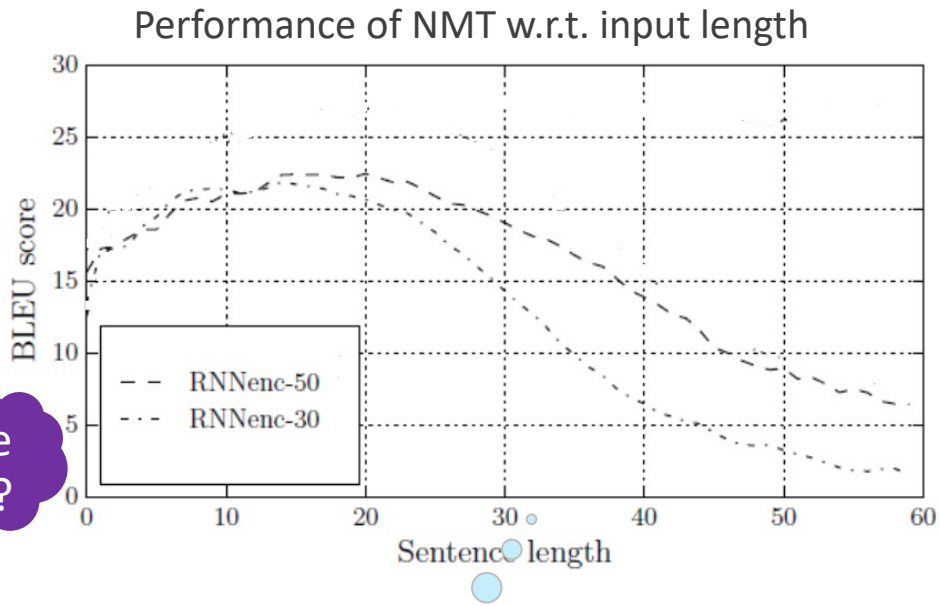


# Intermediate Representation: One Size Fits All?

**Limitation**: the entire sequence is compressed into **one** vector, regardless of the importance of each position.




Bottle Neck?

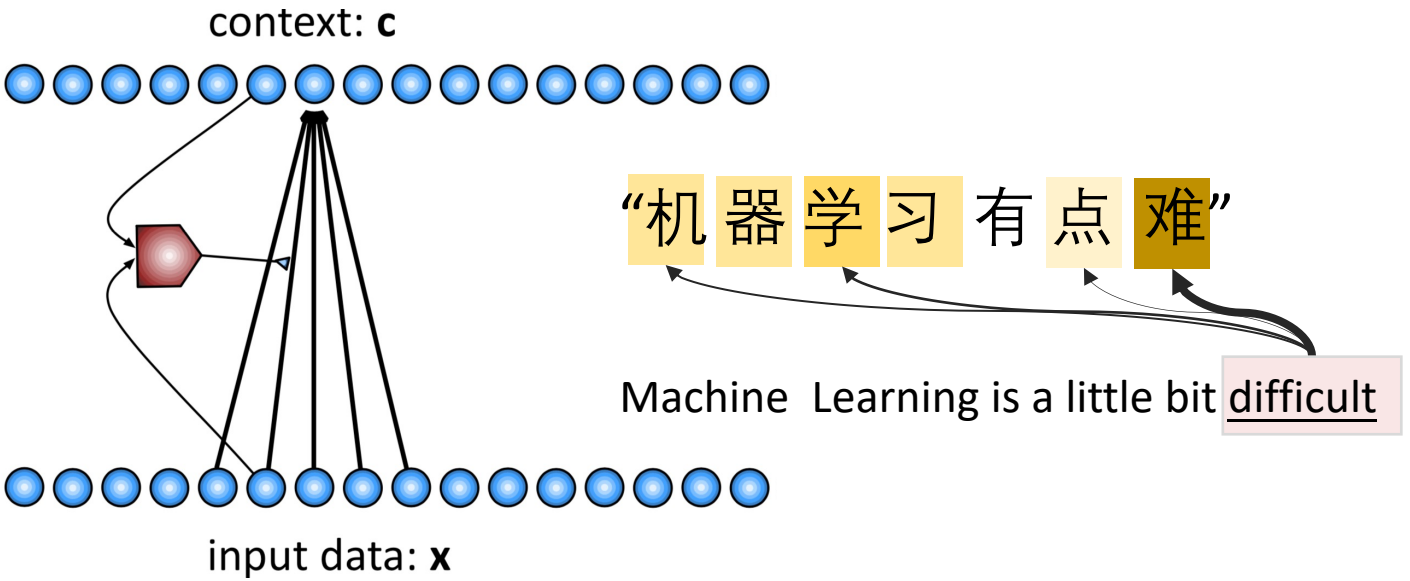



Performance decreases as the sequence becomes longer.

# How to Avoid "the Curse of Length"?

## Dynamic Context Vector

- do not rely on only a single fixed encoding when decoding the output sequence.
- select what to encode in a contextual manner.



# Sequence-to-Sequence with Attention


- When decoding each $y_i$ in $y = (y_1, \cdots, y_{T_y})$, we use a **dynamic context vector** $\boldsymbol{c}_i$ which corresponds to a linear combination of different positions in $x$.

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

**Without Attention**

- $p(y_i|y_{<i}, x) \propto g(y_i; y_{i-1}, s_i, c)$
- RNN hidden state
  $s_i = \text{RNN}_{\text{out}}(s_{i-1}, y_{i-1}, c)$

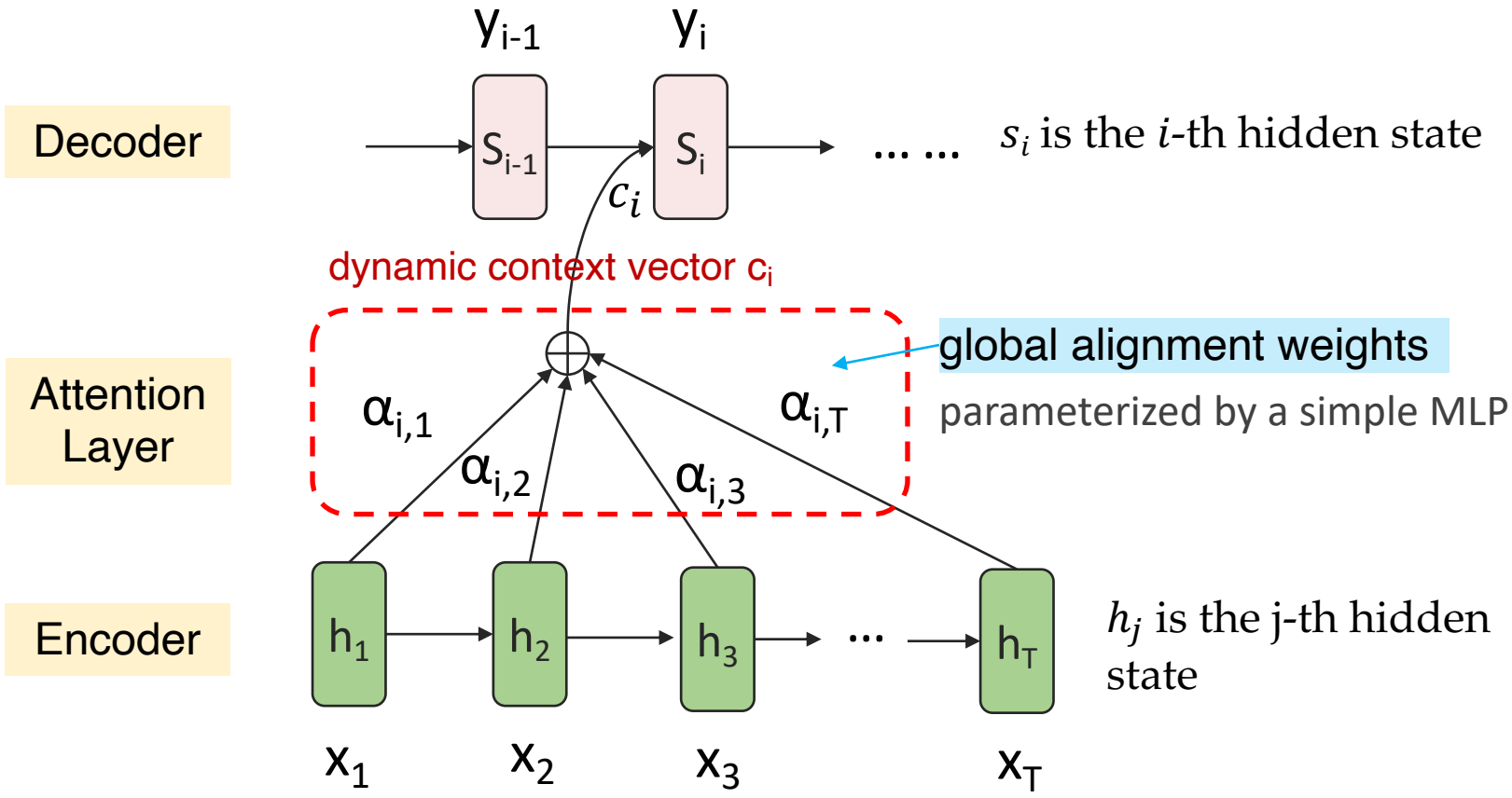
VS.

**With Attention**

- $p(y_i|y_{<i}, x) \propto g(y_i; y_{i-1}, s_i, \boldsymbol{c_i})$
- RNN hidden state
  $s_i = \text{RNN}_{\text{out}}(s_{i-1}, y_{i-1}, \boldsymbol{c_i})$

# Sequence-to-Sequence with Attention

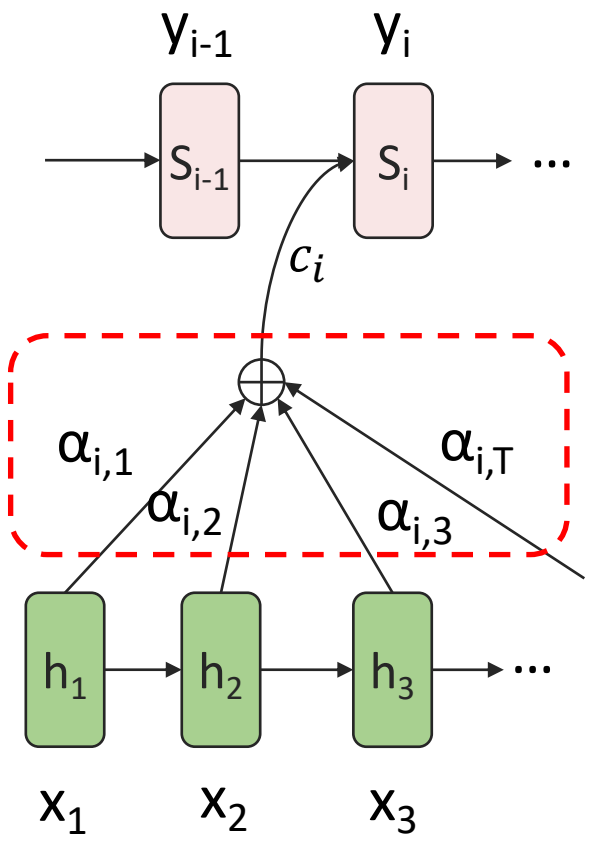
The decoder dynamically pays attention to different tokens in the source sentences during decoding.



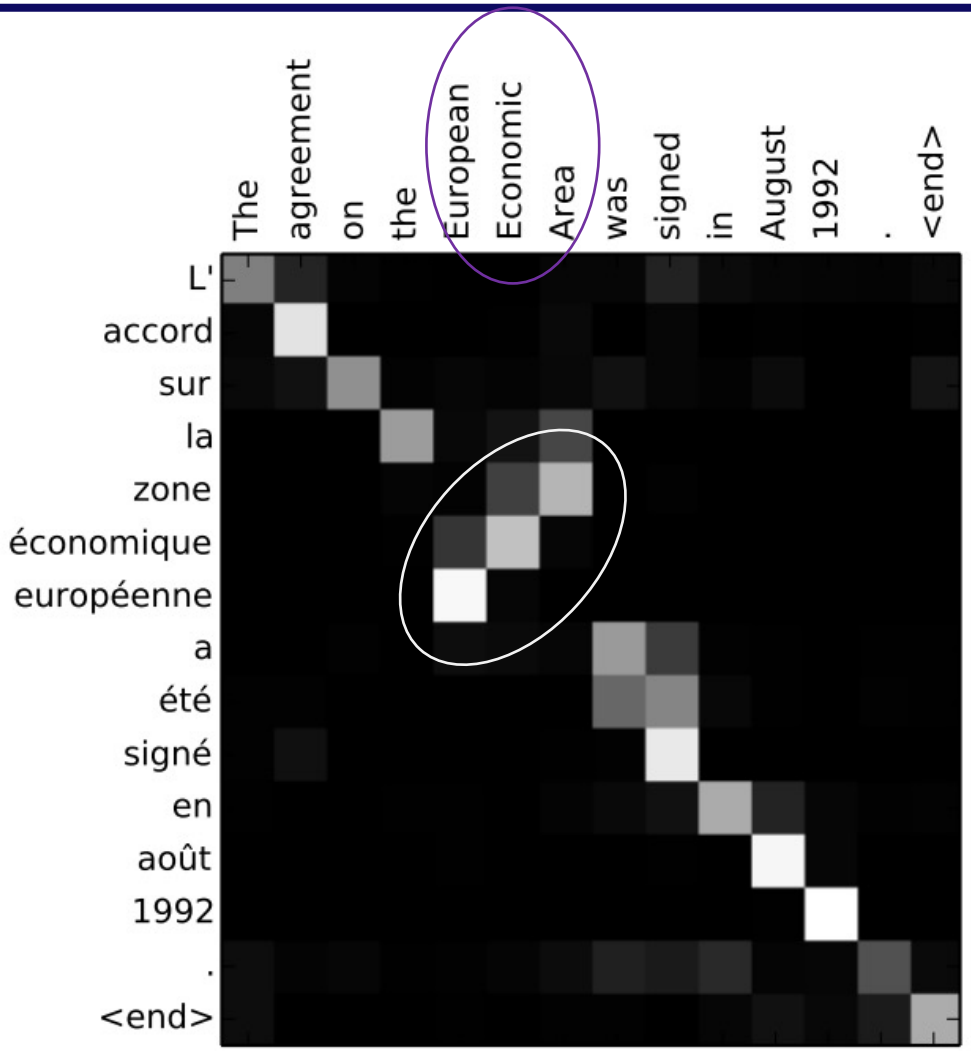
# Attention-based Model

$$\alpha_{ij} = \frac{\exp\left(a(s_{i-1}, h_j)\right)}{\sum_{k=1}^{T_x} \exp\left(a(s_{i-1}, h_k)\right)}$$

- scores how well the inputs around position $j$ and the output at position $i$ match.
- where a(·) denotes a neural network:
  e.g., $a(s_{i-1}, h_j) = \boldsymbol{v}_a^T \tanh\left(\boldsymbol{W}_a s_{i-1} + \boldsymbol{U}_a h_j\right)$



# Visualizing Attention




real $\alpha_{ij}$'s in machine translation

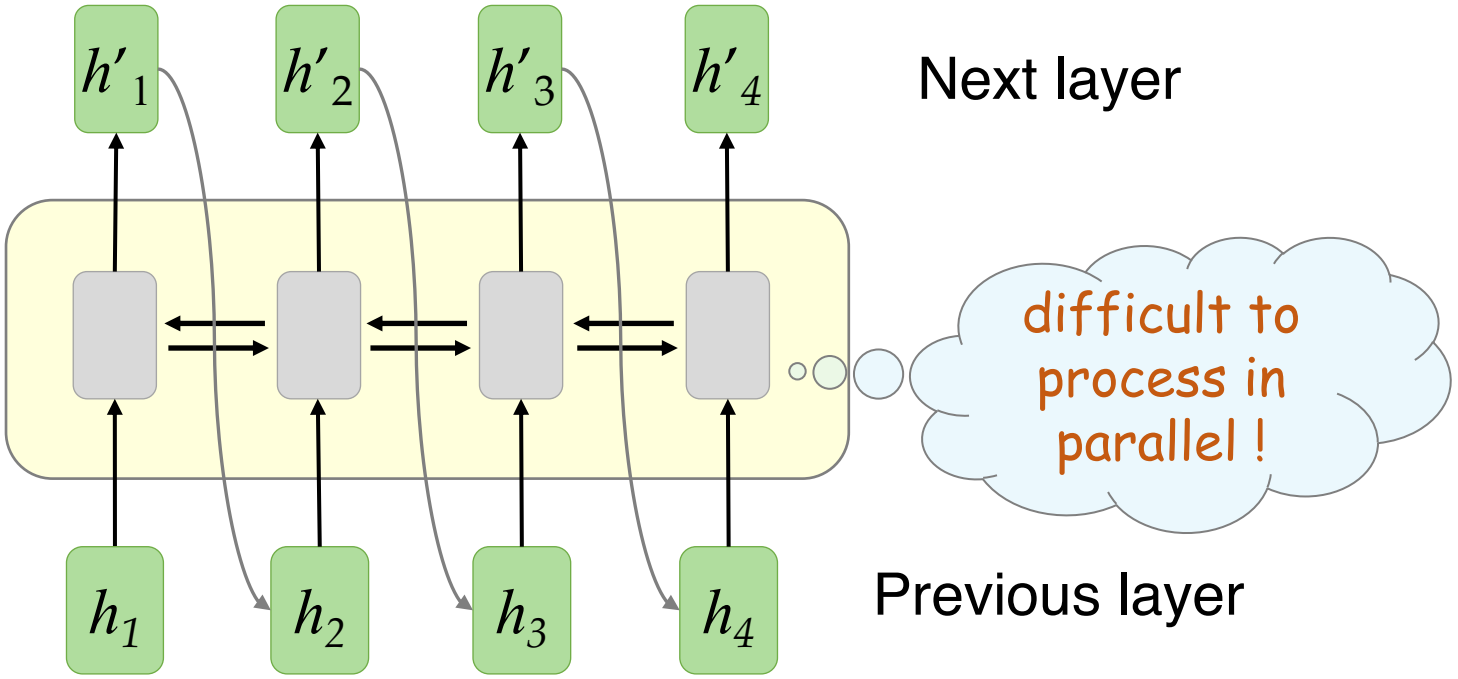
You can see how the model paid attention correctly when outputting "European Economic Area". In French, the **order of these words is reversed** ("européenne économique zone") as compared to English. Every other word in the sentence is in similar order.


# Transformer

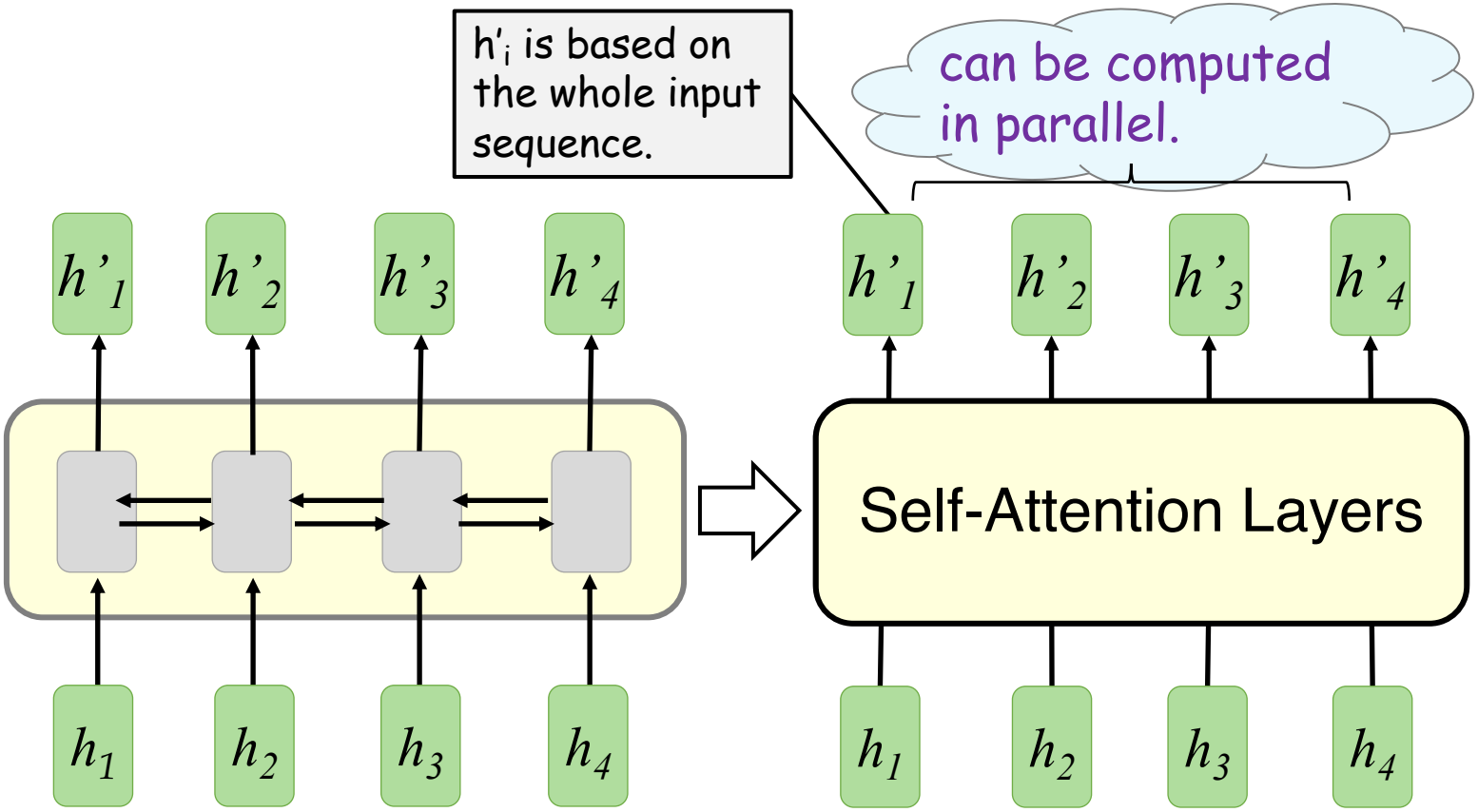
## Seq2seq model with "Self-attention"

# Auto-regressive Sequence Modeling

- **A problem for the RNN encoder/decoder:** the hidden state for one position is dependent on the computation of the preceding position.



# From RNN to Self-Attention



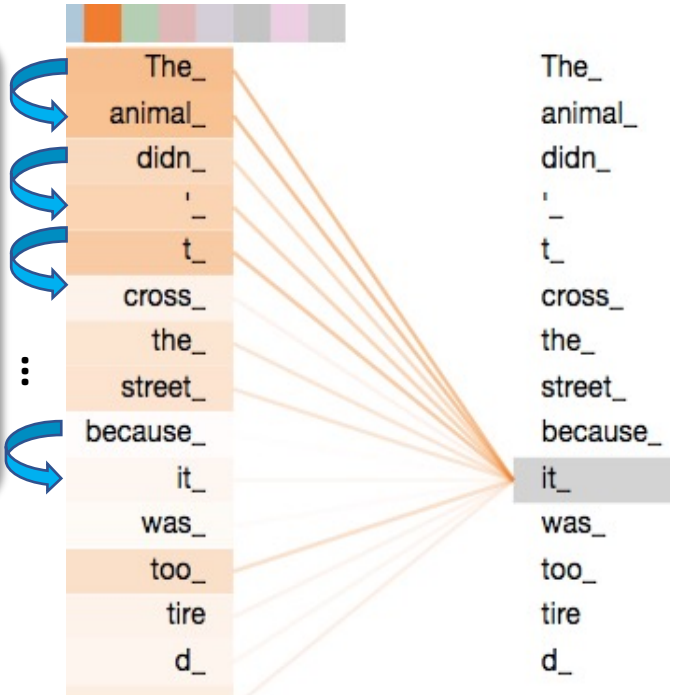

We can replace anything that can be processed by an RNN with a self-attention network.

# From RNN to Self-Attention

What does "**it**" in this sentence refer to?

**RNN:**

maintain a hidden state to sequentially incorporate previous words with the current one it's processing.

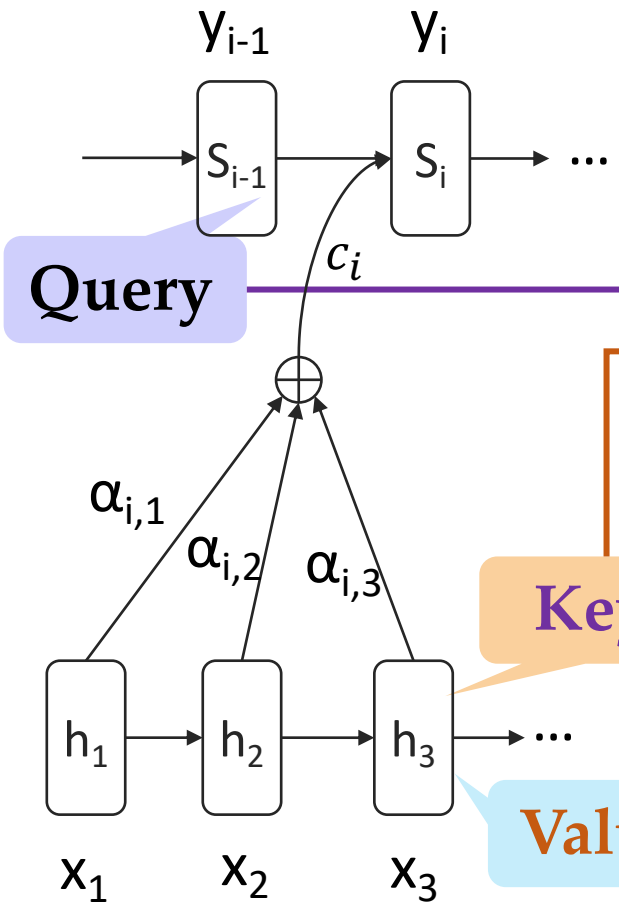



**Self-Attention:**

bake the "understanding" of other relevant words into the one we're currently processing simultaneously.

The animal didn't cross the street because **it** was too tiered.

# Attention Revisited




***q***: query (to match others)

***k***: key (to be matched)

***v***: content to be retrieved

Keys

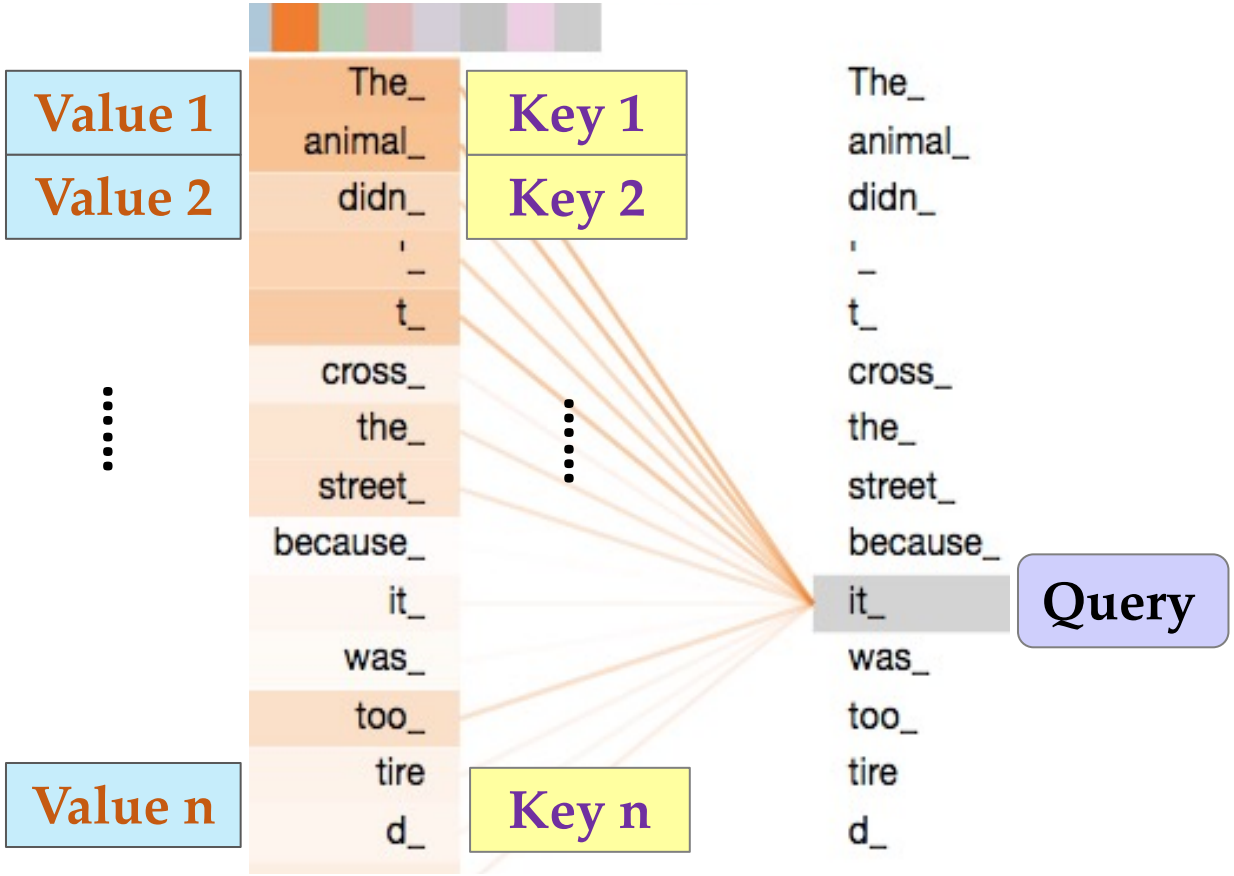
Values

$$\alpha_{ij} = \frac{\exp(a(s_{i-1}, h_j))}{\sum_{k=1}^{T_x} \exp(a(s_{i-1}, h_k))}$$

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

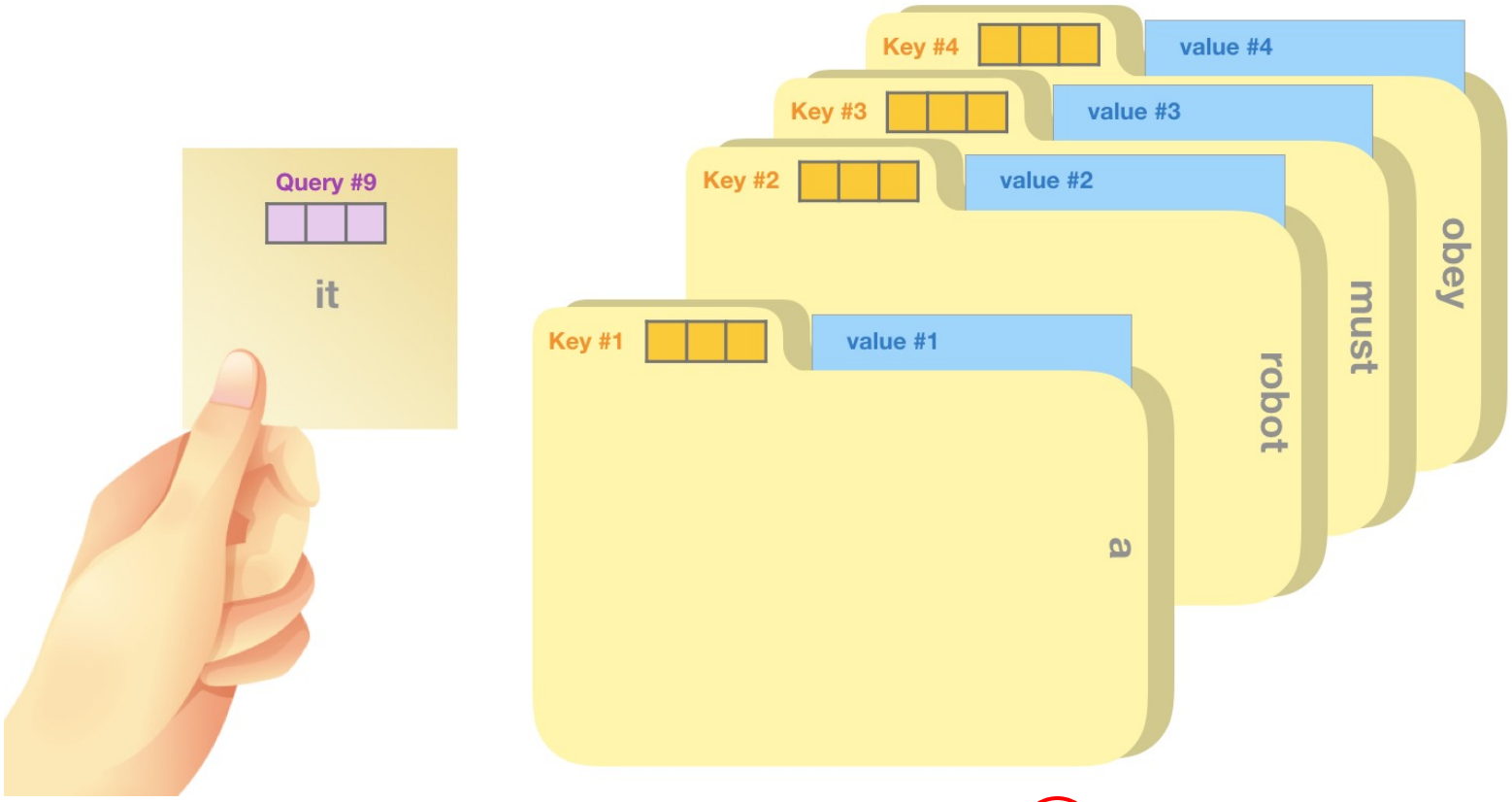
# Self-Attention: The Idea

- Let each word pay attention to all other words.



# Self-Attention: The Idea

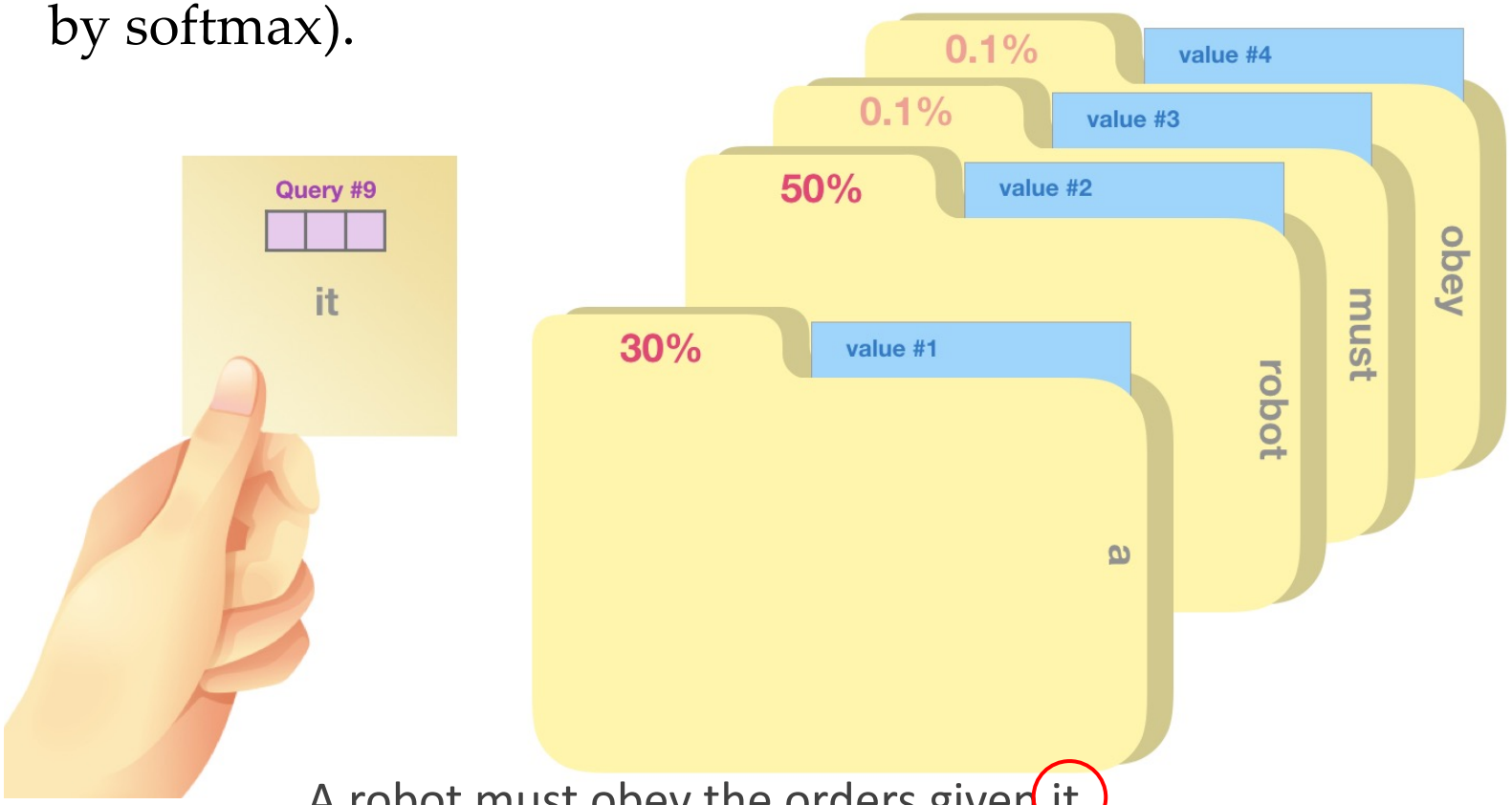
- Let each word pay attention to all other words.




A robot must obey the orders given it.

# Self-Attention: the Idea

- Multiplying the query vector by each key vector produces a score for each value (technically: dot product followed by softmax).



# Self-Attention: the Key Idea

- We multiply each value by its score and sum them up.

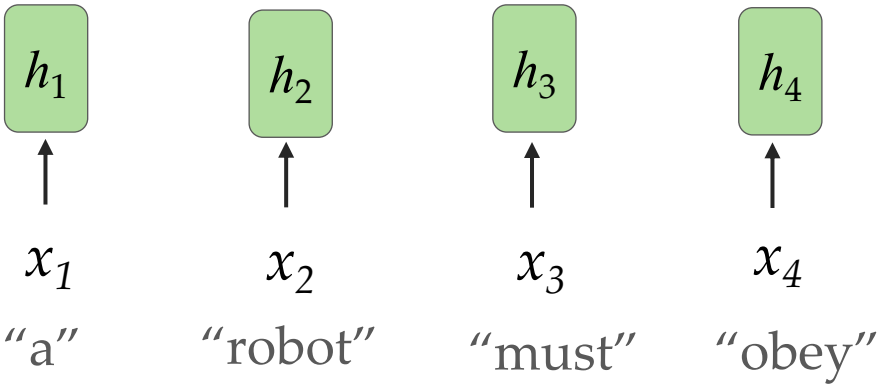
| Word | Value vector | Score | Value X Score |
|---|---|---|---|
| <s> |  | 0.001 |  |
| a |  | 0.3 |  |
| robot |  | 0.5 |  |
| must |  | 0.002 |  |
| obey |  | 0.001 |  |
| the |  | 0.0003 |  |
| orders |  | 0.005 |  |
| given |  | 0.002 |  |
| it |  | 0.19 |  |
| | | | |
| | | Sum: |  |

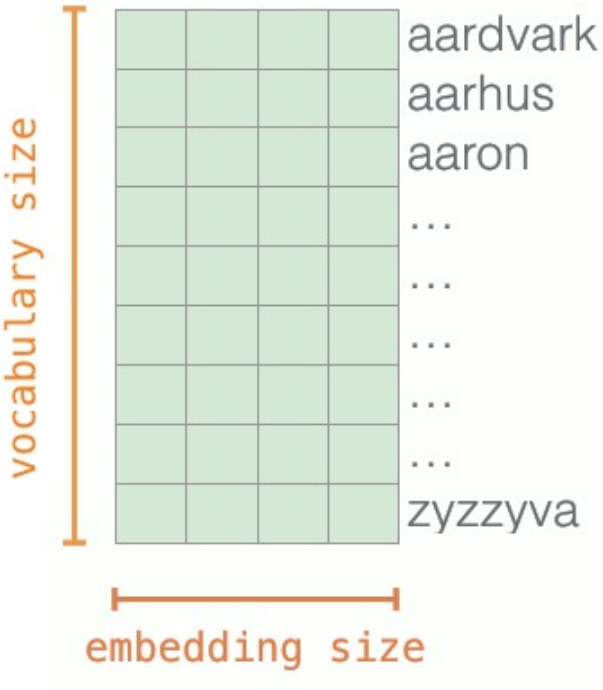
- The outcome vector represents the new state (refreshed memory) for the query word.

# Step 1: Token Embedding

- Embedding tokens (integer id) into vectors (hidden states).

$$\boldsymbol{h}_i = \boldsymbol{W} x_i$$

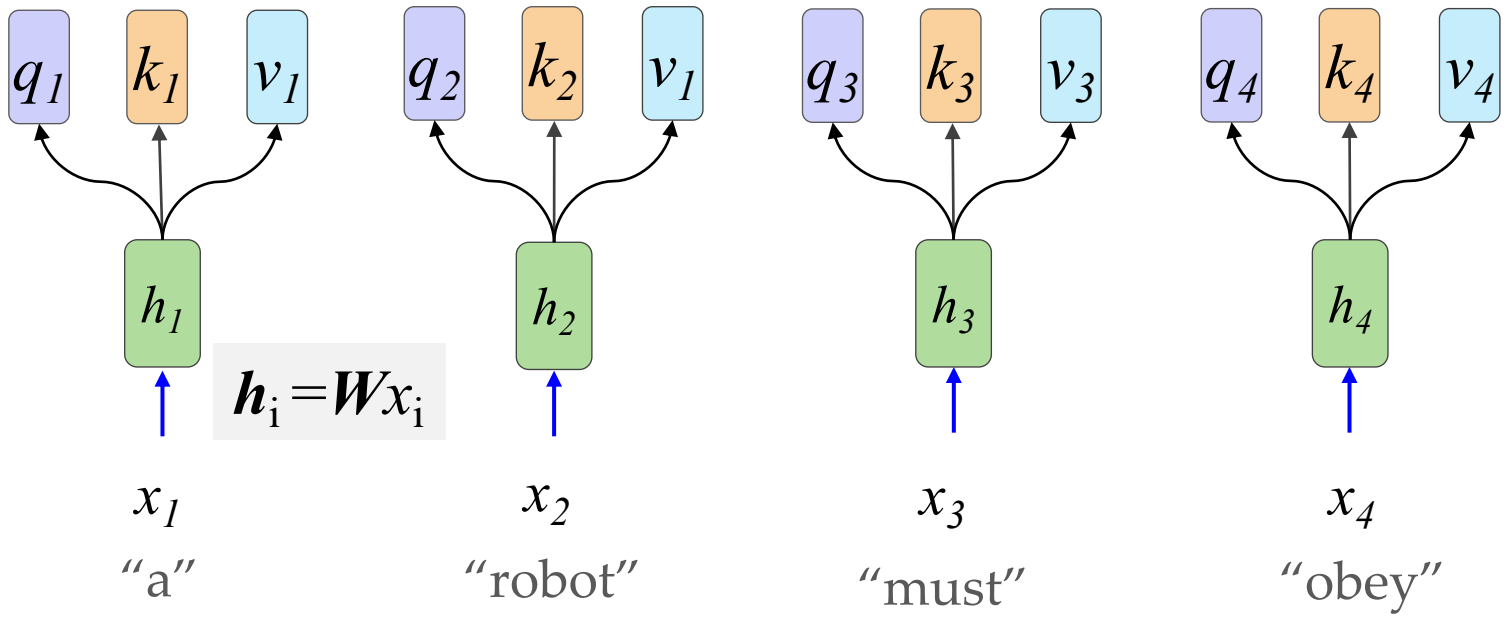


# Step 2 : Q, K, V vectors

- Transform each hidden state into **query/key/value** vectors:

$$q_i = W^q h_i \qquad k_i = W^k h_i \qquad v_i = W^v h_i$$

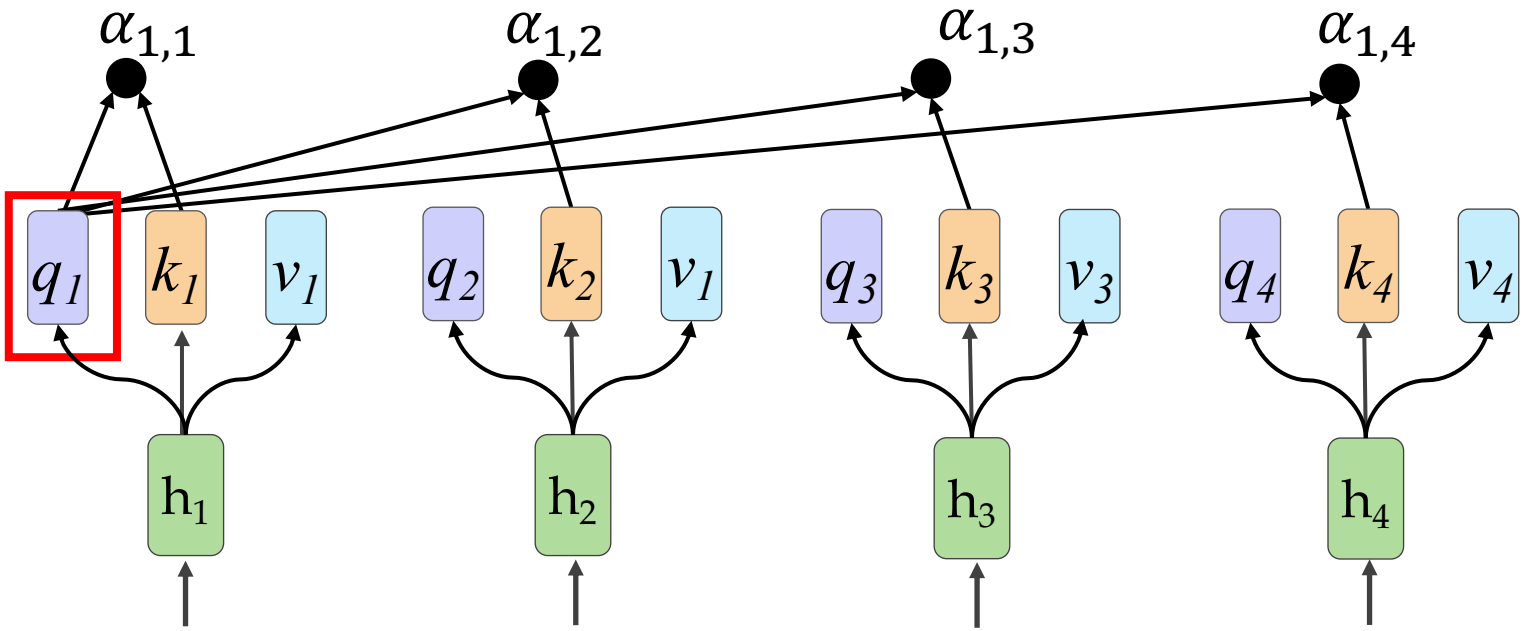


# Step 2: Calculate Attention Scores

- Calculate an attention score for each <query, key> pair using scaled dot-product.

$$\alpha_{1,i} = \frac{q_i^T k_i}{\sqrt{d}}$$

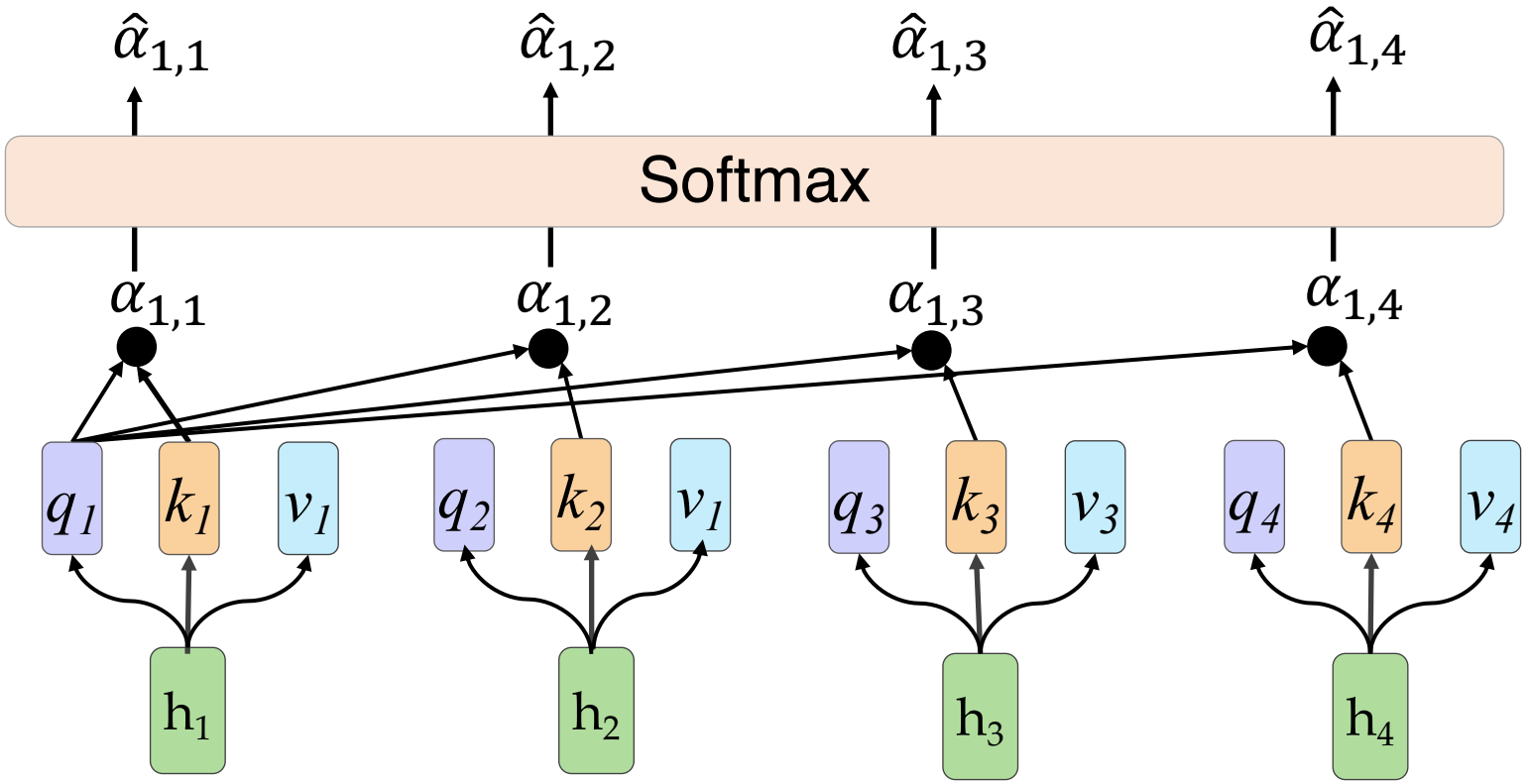
where d is the dim of $q$ and $k$



# Step 3: Normalize Attention Scores

- Normalize attention scores to attention weights using softmax.

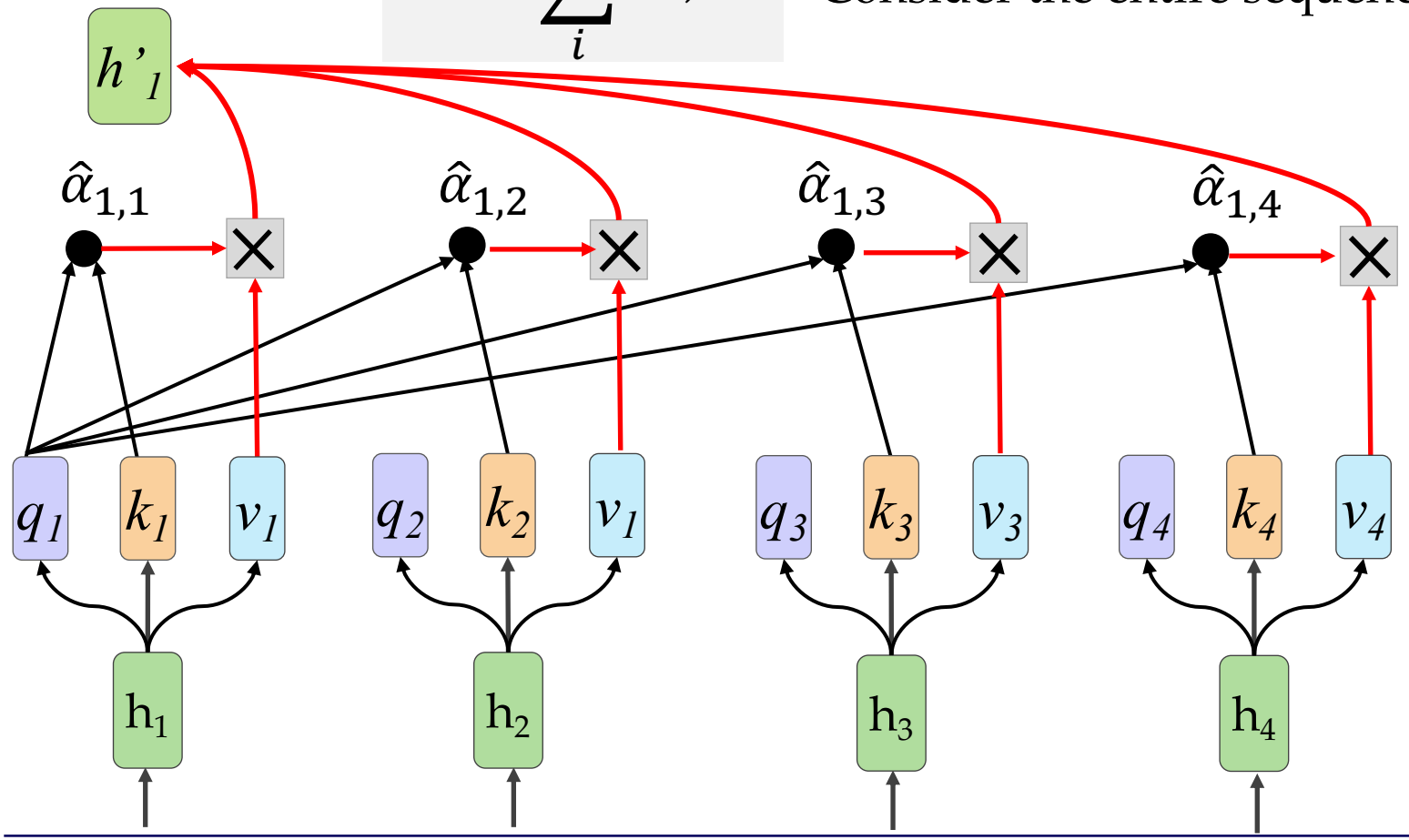
$$\hat{\alpha}_{1,i}=\exp(\alpha_{1,i})/\sum_j \exp(\alpha_{1,j})$$



# Step 4: Summarize Values According to Attention Weights

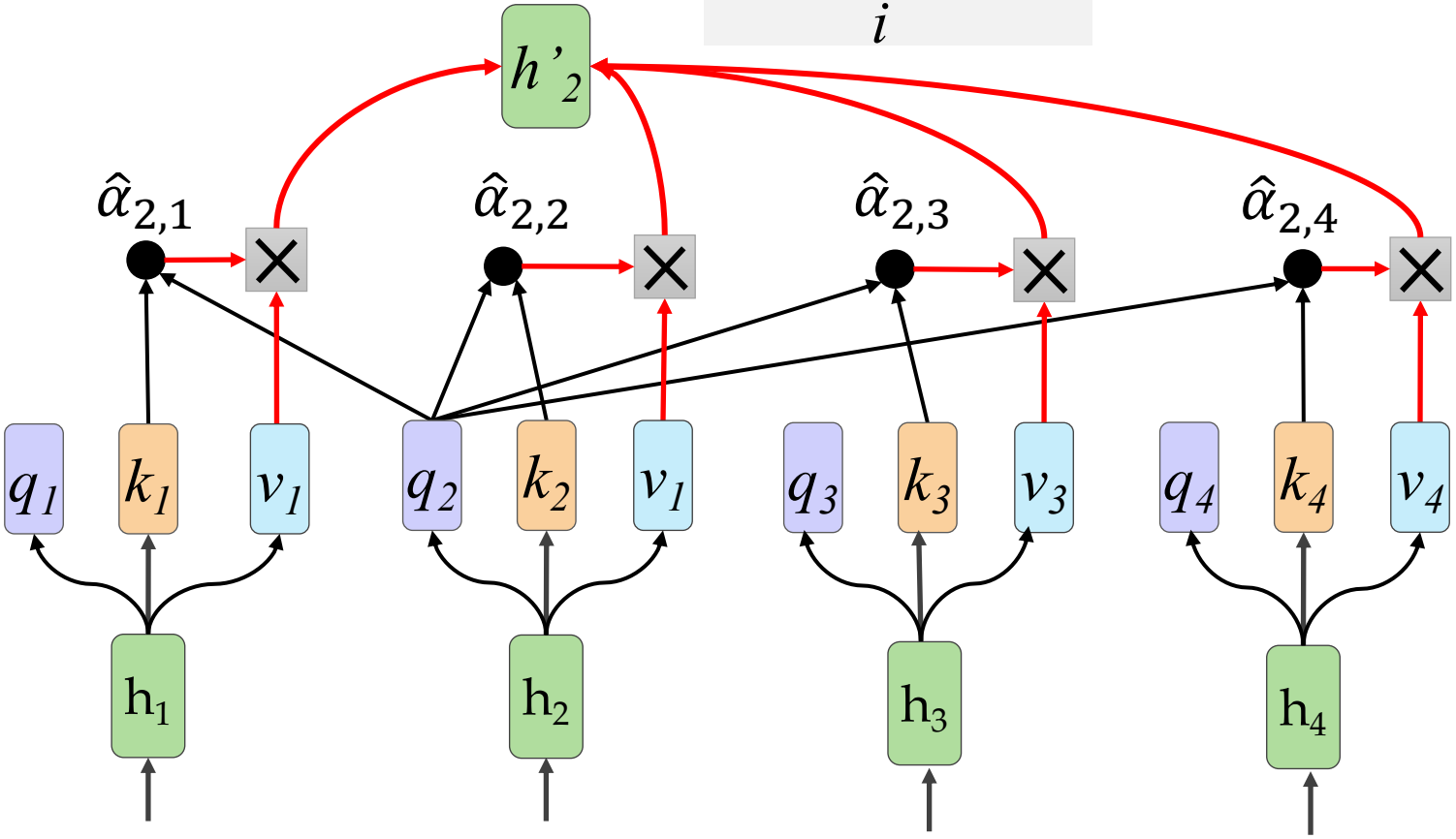
$$h'_1 = \sum_i \hat{\alpha}_{1,i} v_i$$

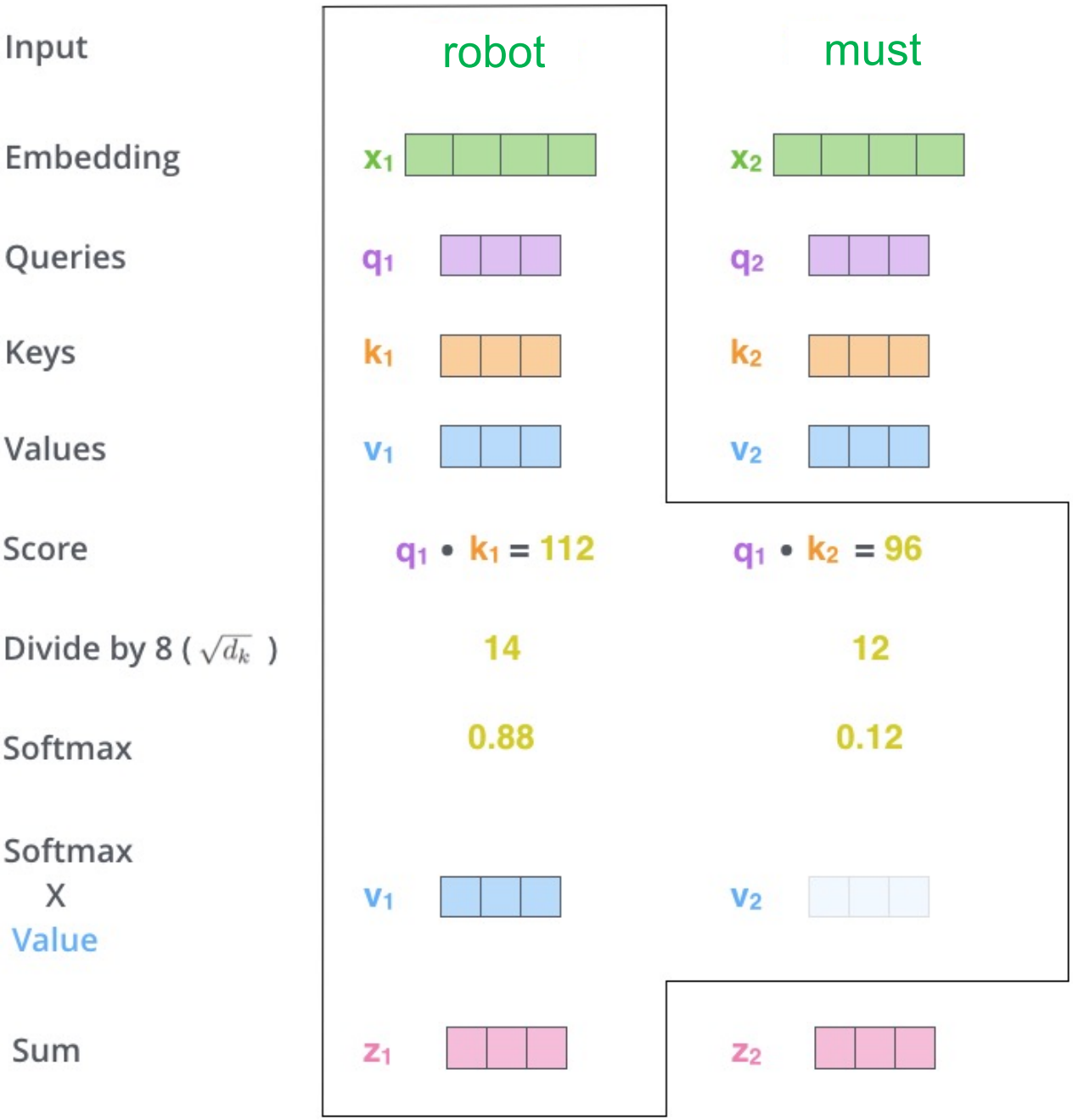
Consider the entire sequence



# Step 4: Summarize Values According to Attention Weights

$$h'_2 = \sum_i \hat{\alpha}_{2,i} v_i$$




Input
robot
must
Embedding
$x_1$
$x_2$
Queries
$q_1$
$q_2$
Keys
$k_1$
$k_2$
Values
$v_1$
$v_2$
Score
$q_1 \cdot k_1 = 112$
$q_1 \cdot k_2 = 96$
Divide by 8 ( $\sqrt{d_k}$ )
14
12
Softmax
0.88
0.12
Softmax X Value
$v_1$
$v_2$
Sum
$z_1$
$z_2$

**Queries**

| robot | must | obey | orders |
|---|---|---|---|

X

**Keys**

| robot | must | obey | orders |
|---|---|---|---|
| robot | must | obey | orders |
| robot | must | obey | orders |
| robot | must | obey | orders |

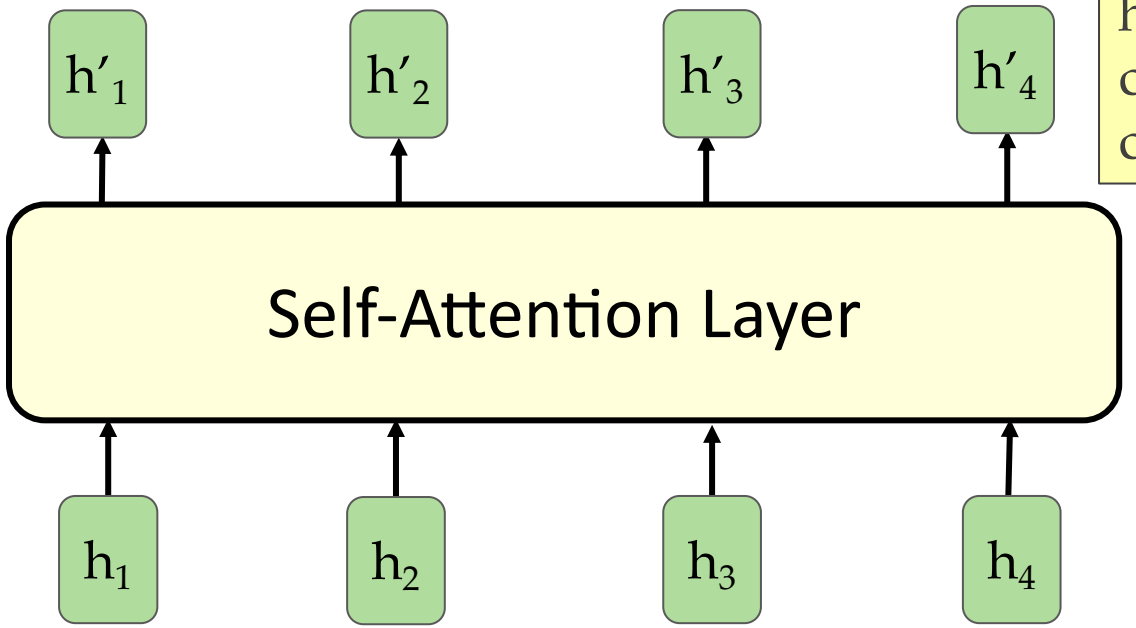
=

**Scores** (before softmax)

| | | | |
|---|---|---|---|
| 0.11 | 0.00 | 0.81 | 0.79 |
| 0.19 | 0.50 | 0.30 | 0.48 |
| 0.53 | 0.98 | 0.95 | 0.14 |
| 0.81 | 0.86 | 0.38 | 0.90 |

| Word | Value vector | Score | Value X Score |
|---|---|---|---|
| <s> |  | 0.001 |  |
| a |  | 0.3 |  |
| robot |  | 0.5 |  |
| must |  | 0.002 |  |
| obey |  | 0.001 |  |
| the |  | 0.0003 |  |
| orders |  | 0.005 |  |
| given |  | 0.002 |  |
| it |  | 0.19 |  |
| | | | |
| | | Sum: |  |

# Matrix Calculation?

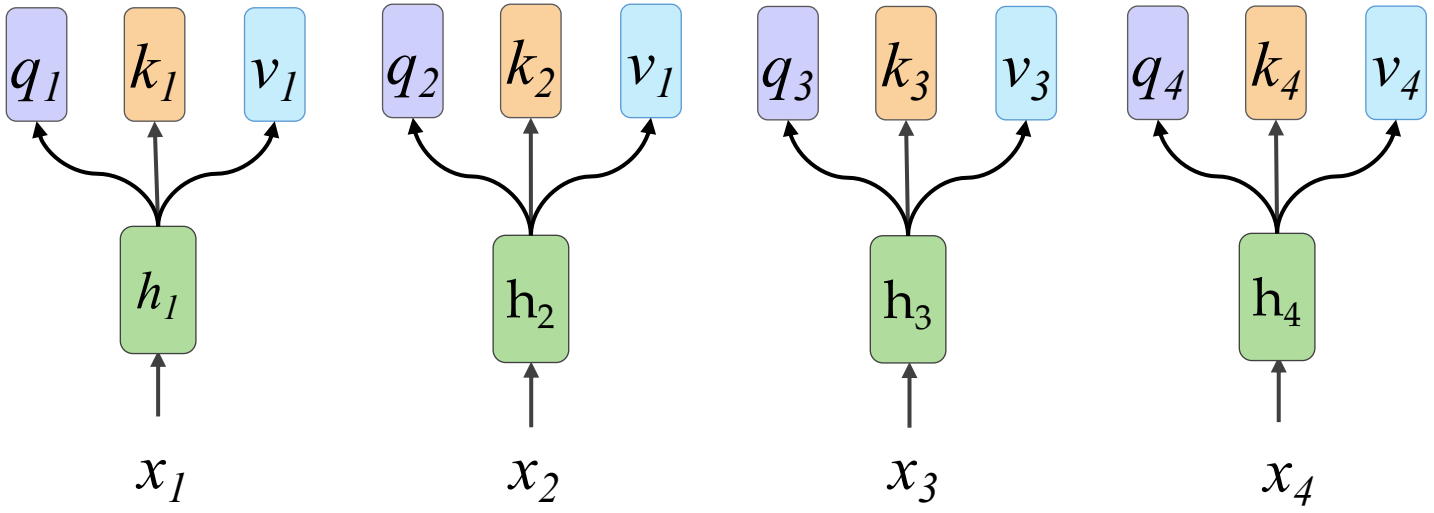



$h'_1, h'_2, h'_3, h'_4$ can be parallelly computed.

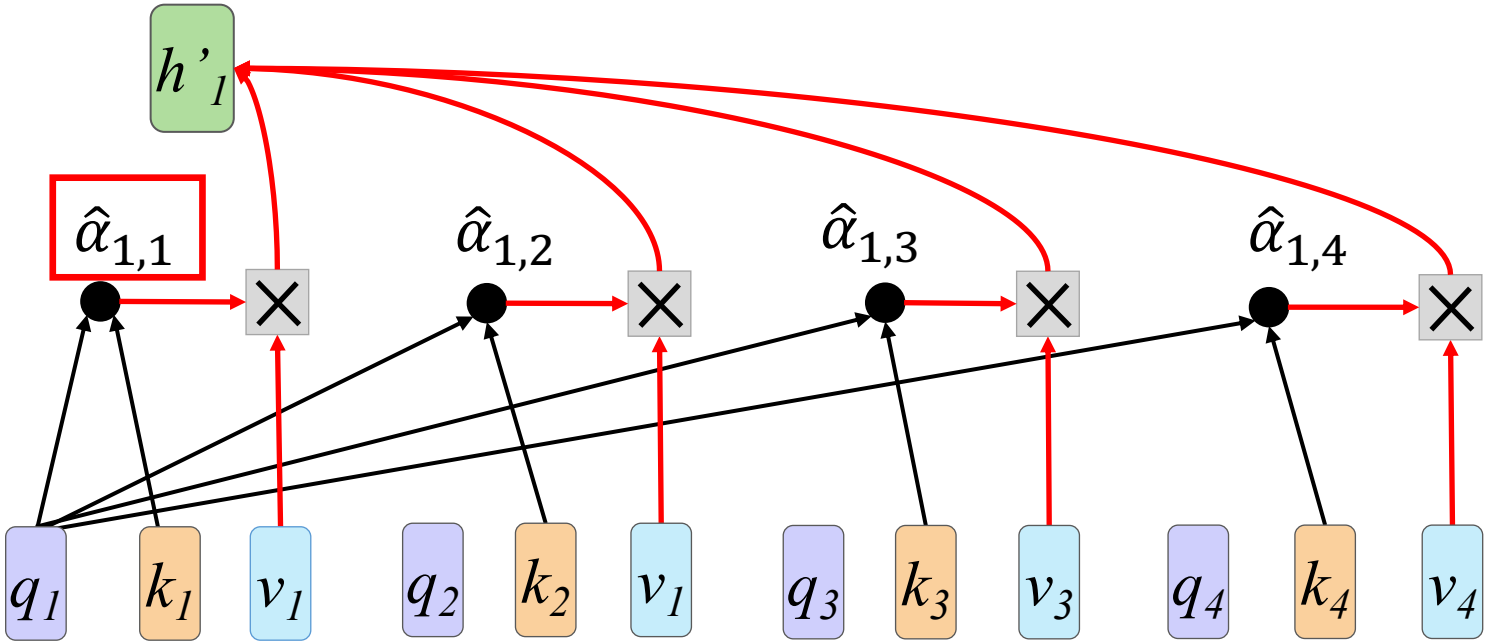
# Self-Attention as Matrix Multiplication

$$q_i = W^q h_i$$
$$k_i = W^k h_i$$
$$v_i = W^v h_i$$

$$Q \; [q_1\ q_2\ q_3\ q_4] = W^q \; [h_1\ h_2\ h_3\ h_4] \; H$$
$$K \; [k_1\ k_2\ k_3\ k_4] = W^k \; [h_1\ h_2\ h_3\ h_4] \; H$$
$$V \; [v_1\ v_2\ v_3\ v_4] = W^v \; [h_1\ h_2\ h_3\ h_4] \; H$$



# Self-Attention as Matrix Multiplication



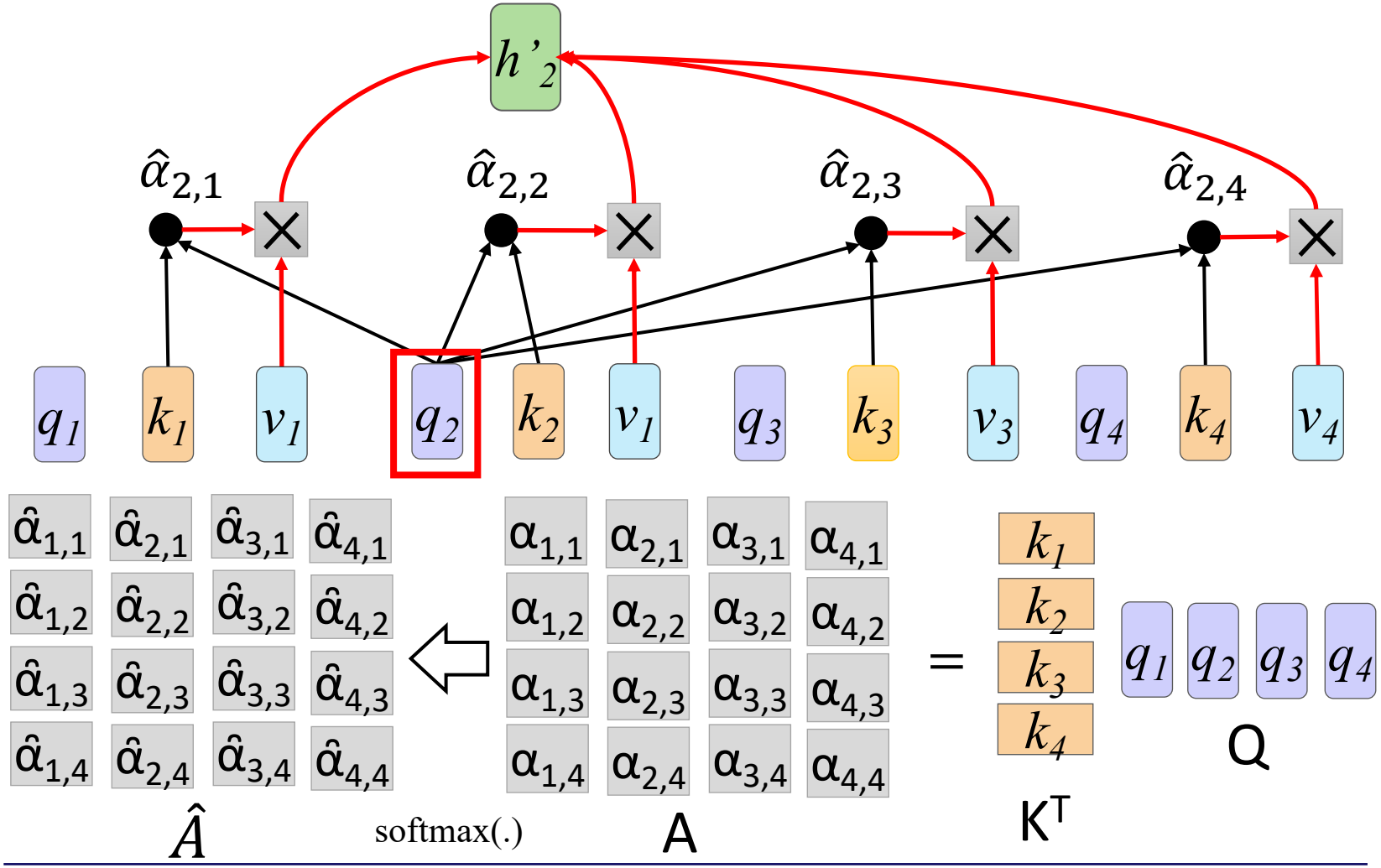

$\alpha_{1,1} = k_1 q_1$ $\alpha_{1,2} = k_2 q_1$

$\alpha_{1,3} = k_3 q_1$ $\alpha_{1,4} = k_4 q_1$

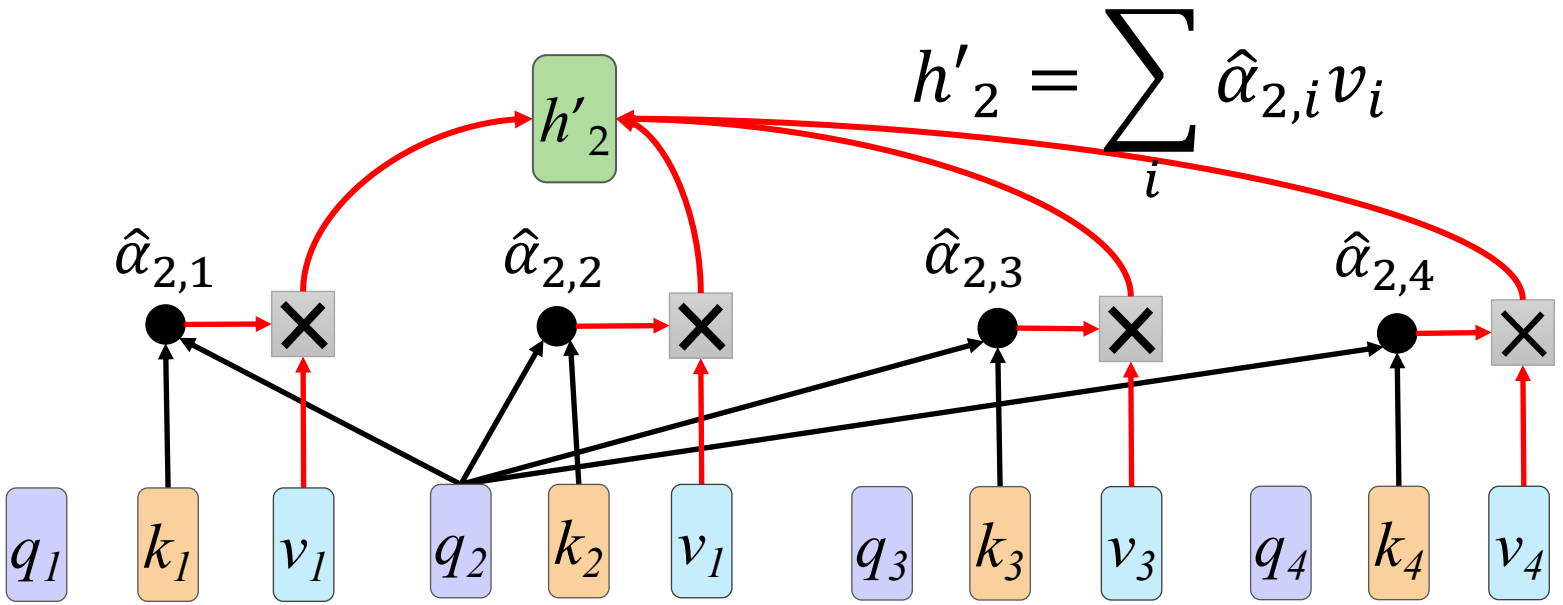
$$\begin{bmatrix} \alpha_{1,1} \\ \alpha_{1,2} \\ \alpha_{1,3} \\ \alpha_{1,4} \end{bmatrix} = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix} q_1$$

(ignore $\sqrt{d}$ for simplicity)

# Self-Attention as Matrix Multiplication



# Self-Attention as Matrix Multiplication



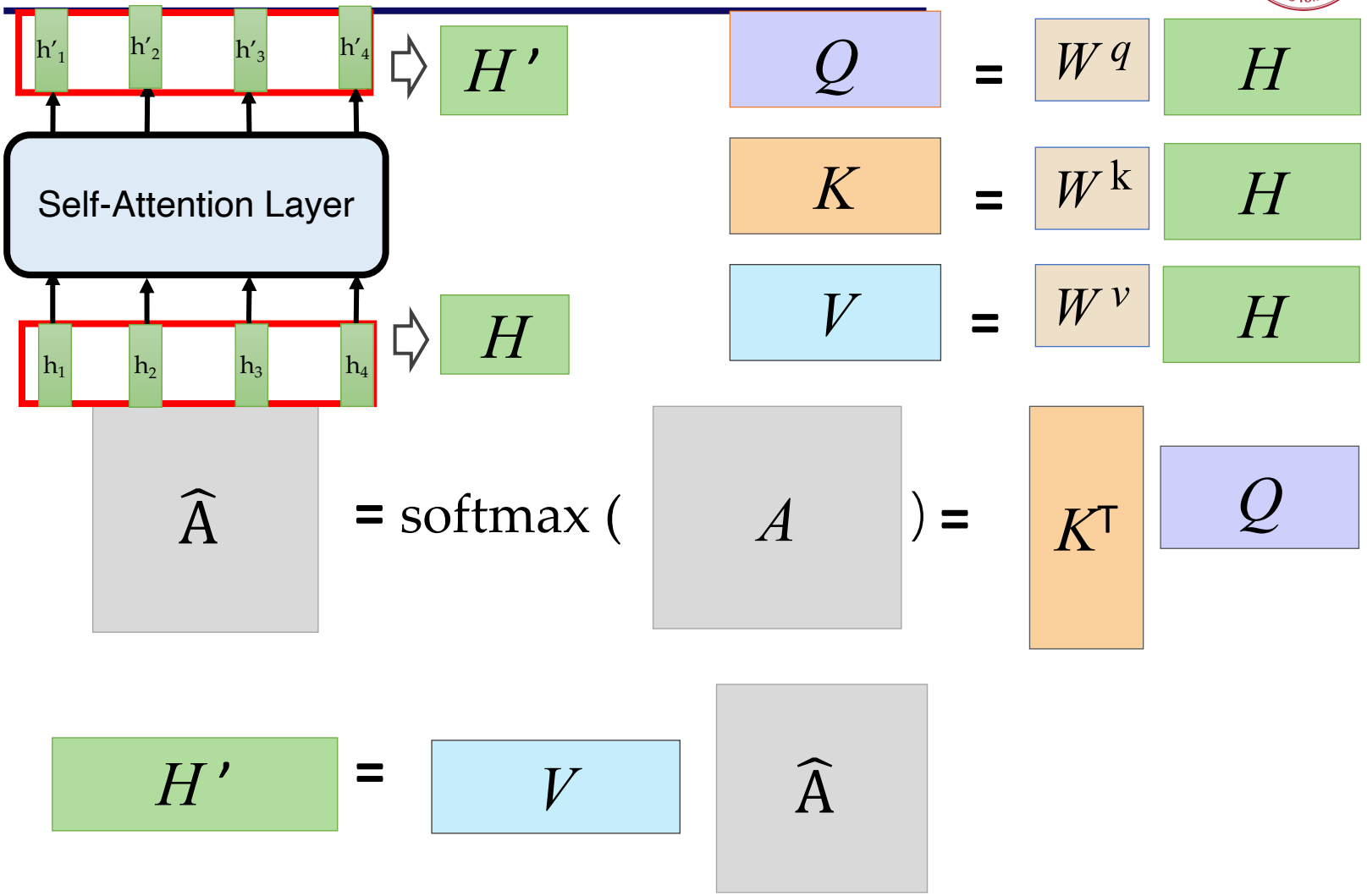

$h_1$ $h_2$ $h_3$ $h_4$ = $v_1$ $v_2$ $v_3$ $v_4$

H' V

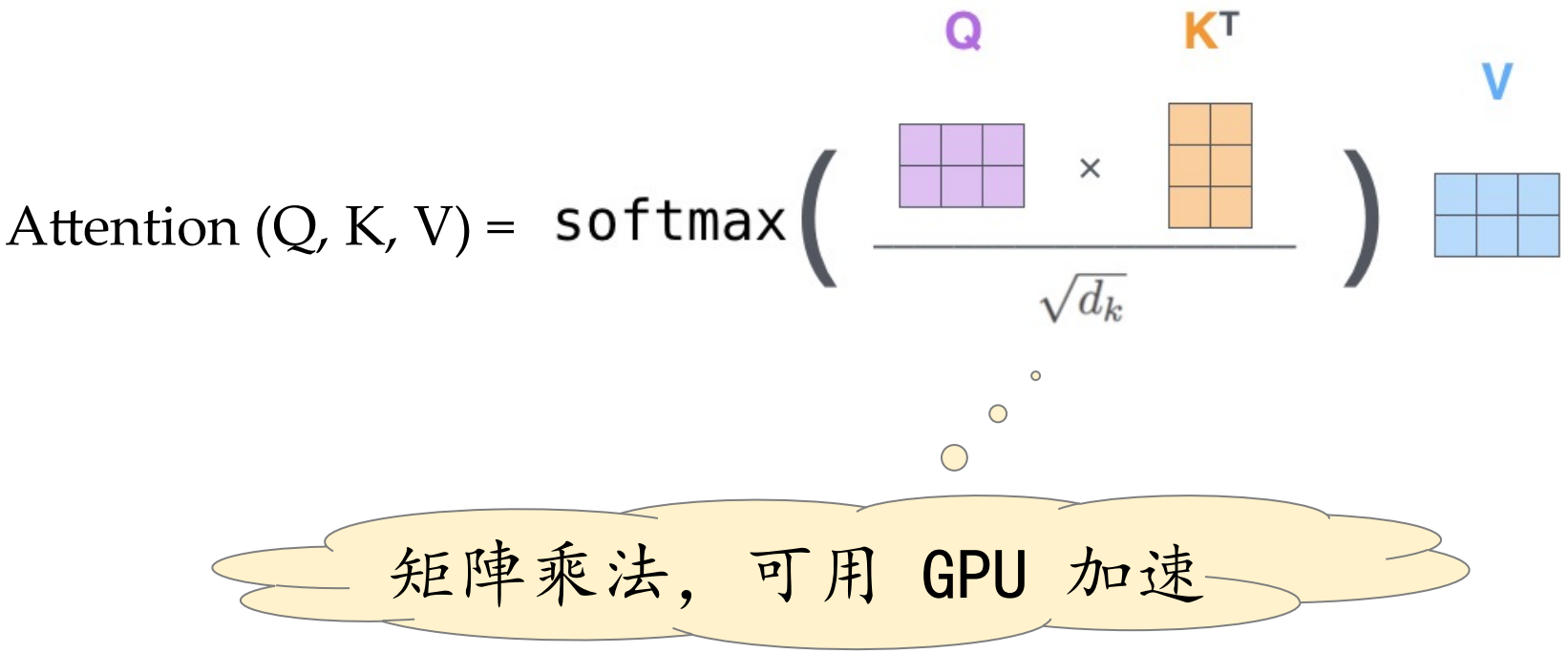
| | | | |
|---|---|---|---|
| $\hat{\alpha}_{1,1}$ | $\hat{\alpha}_{2,1}$ | $\hat{\alpha}_{3,1}$ | $\hat{\alpha}_{4,1}$ |
| $\hat{\alpha}_{1,2}$ | $\hat{\alpha}_{2,2}$ | $\hat{\alpha}_{3,2}$ | $\hat{\alpha}_{4,2}$ |
| $\hat{\alpha}_{1,3}$ | $\hat{\alpha}_{2,3}$ | $\hat{\alpha}_{3,3}$ | $\hat{\alpha}_{4,3}$ |
| $\hat{\alpha}_{1,4}$ | $\hat{\alpha}_{2,4}$ | $\hat{\alpha}_{3,4}$ | $\hat{\alpha}_{4,4}$ |

$\hat{A}$

# Self-Attention as Matrix Multiplication

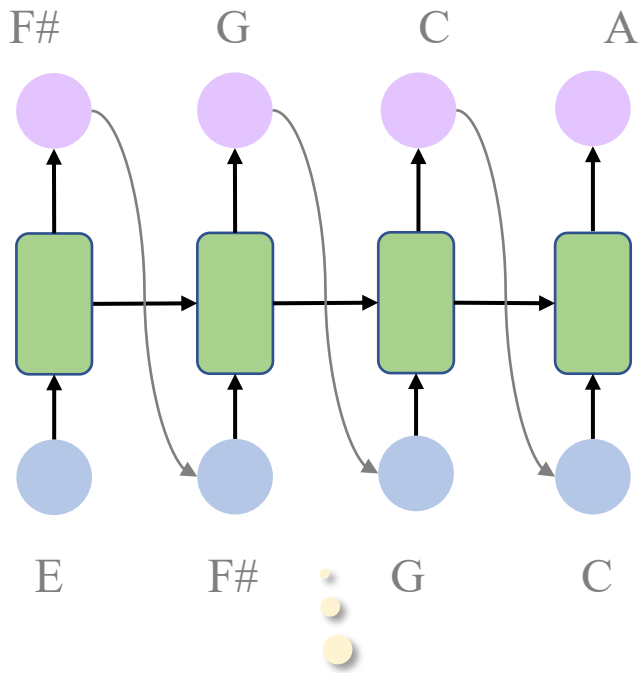


# Self-Attention as Matrix Multiplication



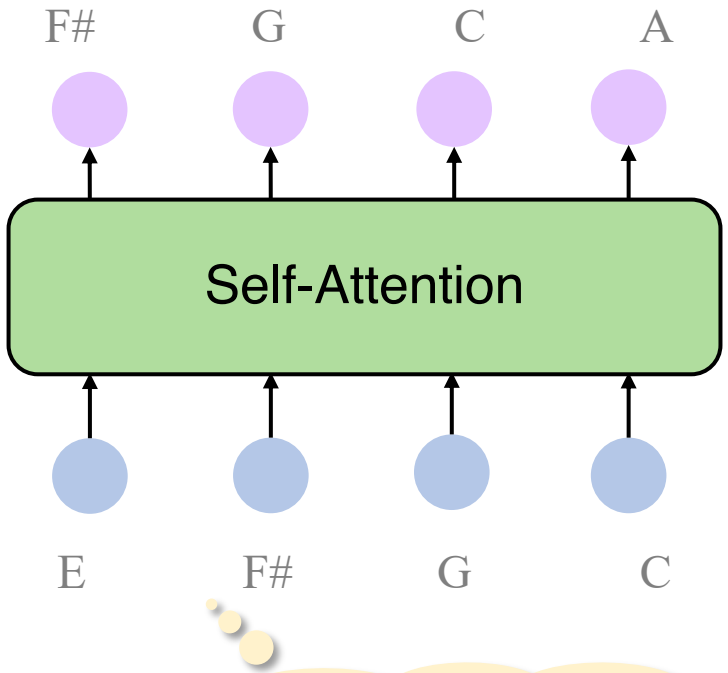
# Representing the Order?

RNN:




Maintain a hidden state to represent orders.

Self-Attention:

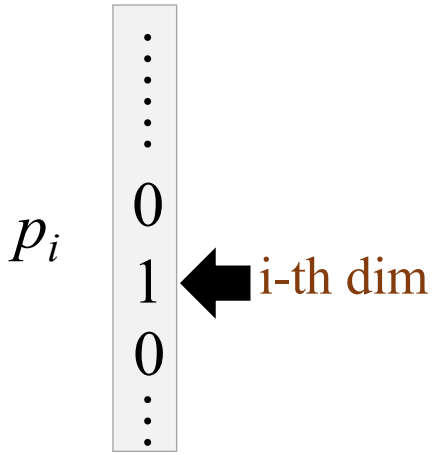


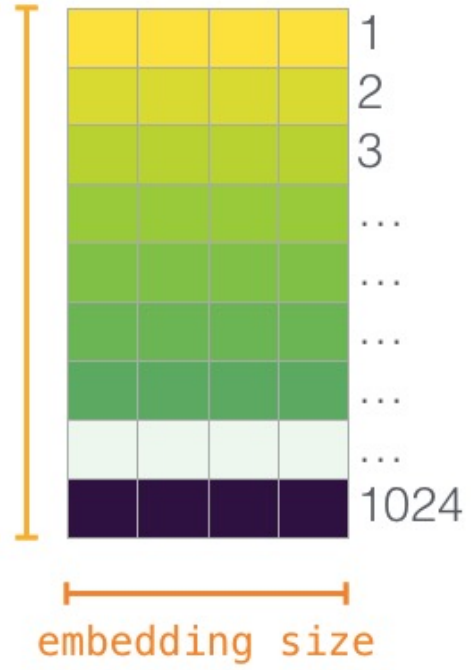

How to represent the order of input tokens?

# Representing the Order?

- Position Encoding: each position has a unique positional vector $e^i$ (typically learned from data)
- For each $x^i$ we append a one-hot vector $p^i$ indicating the position of the word in the sequence.

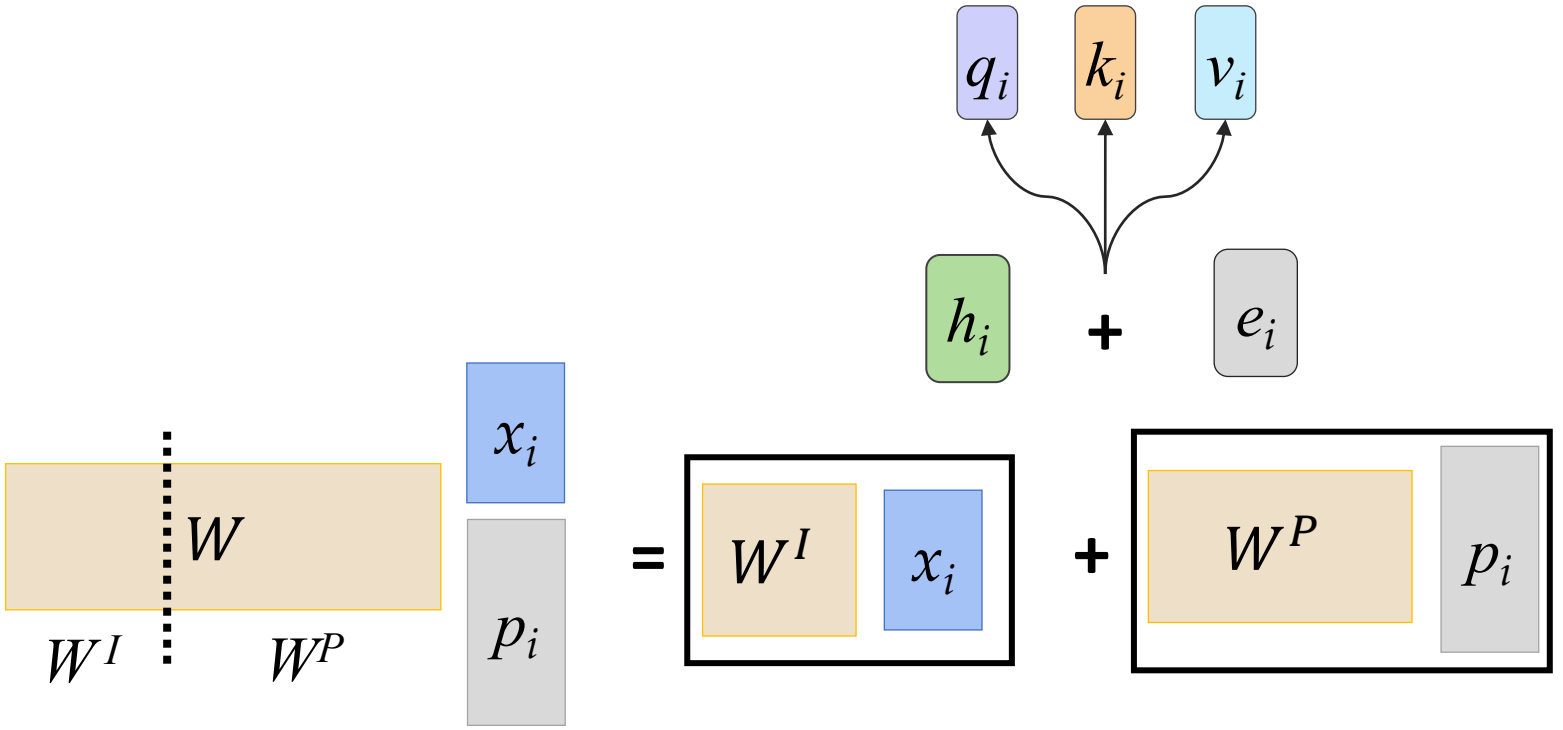
$$\boldsymbol{e}_i = W^p \, p_i$$



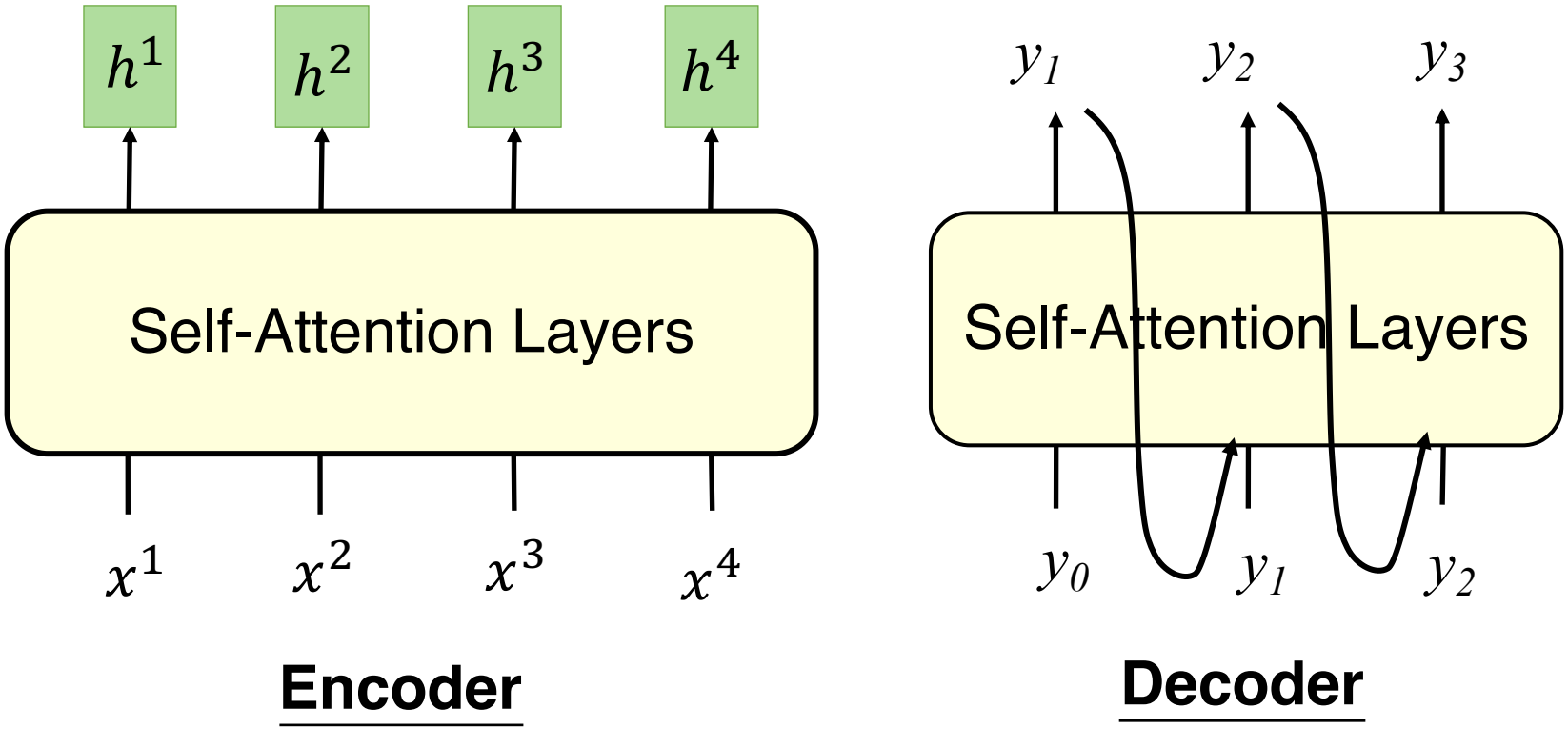
# Self-Attention with Position Encoding

- The position encoding $e_i$, combined with the word embedding $h_i$, is feed into the self-attention layer.

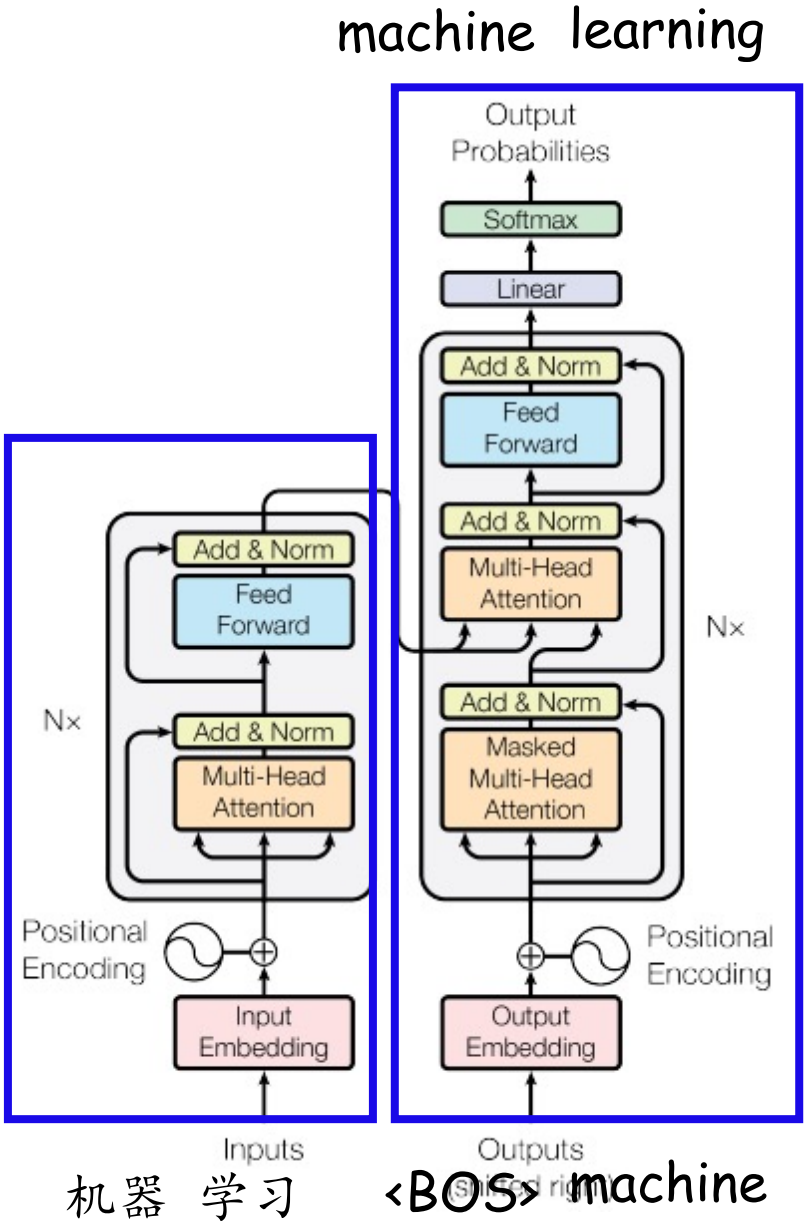


# Seq2seq ~~with~~Using Self-Attention

https://arxiv.org/abs/1706.03762



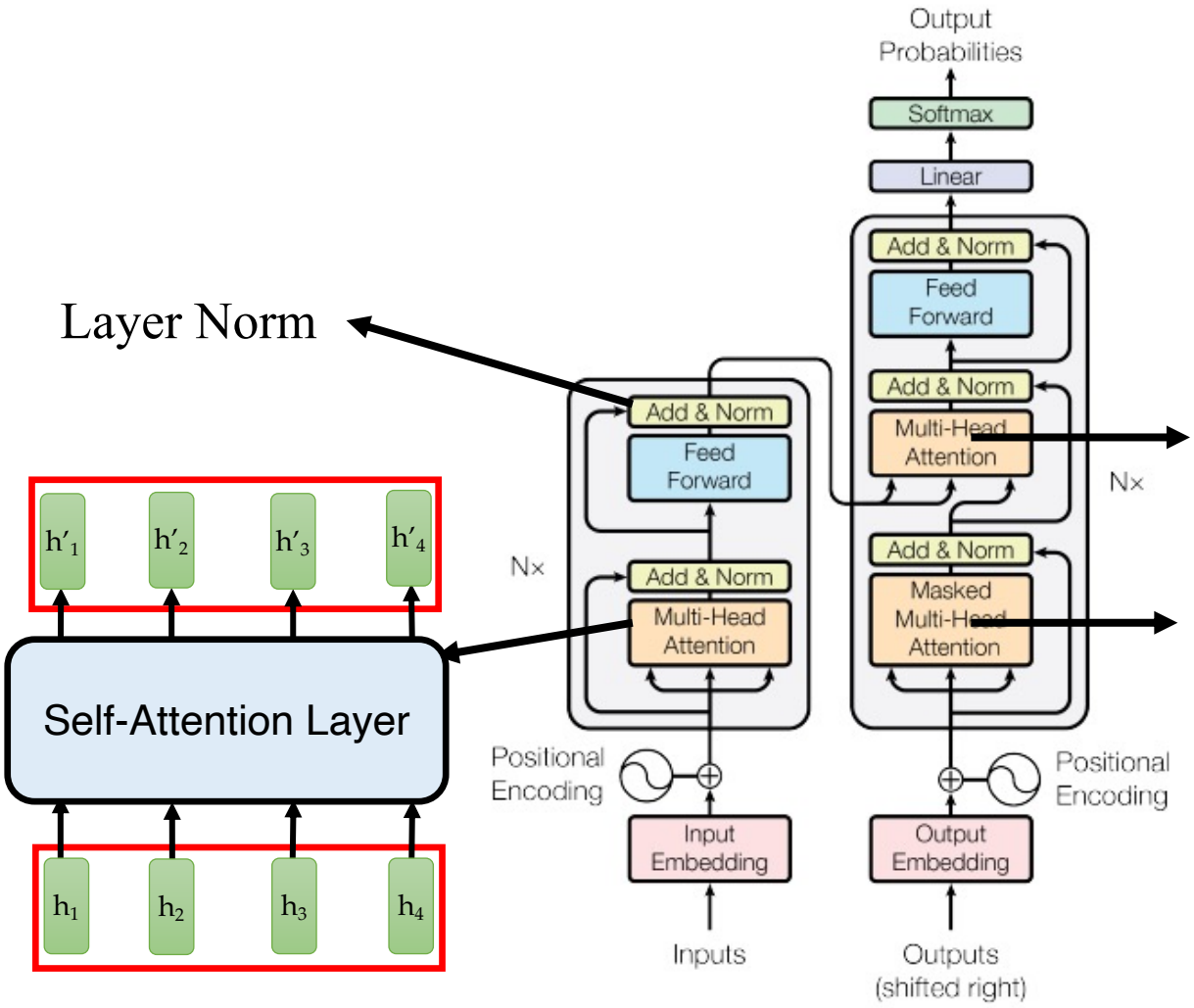
# Transformer




Encoder

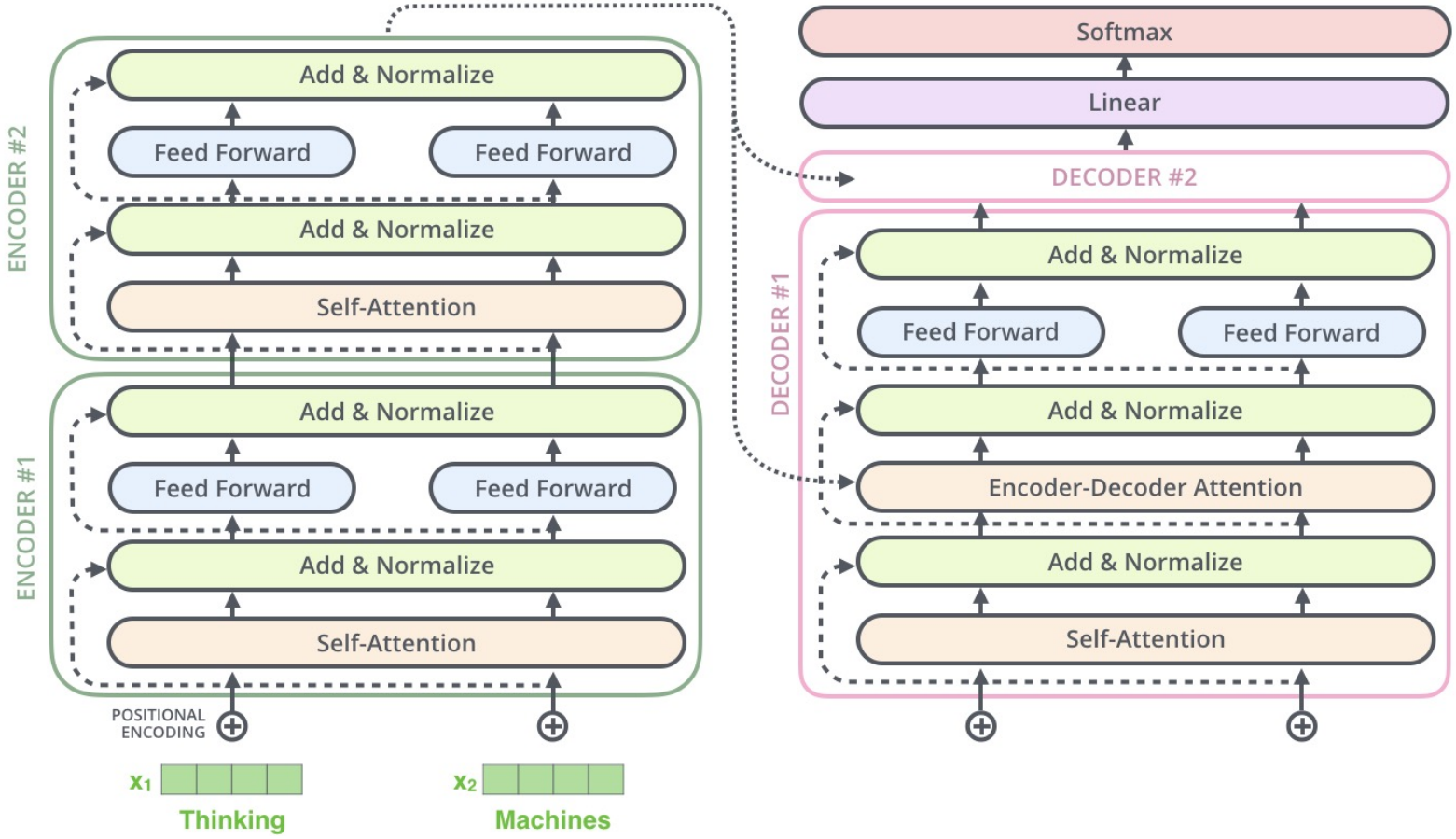
Decoder

# Architecture



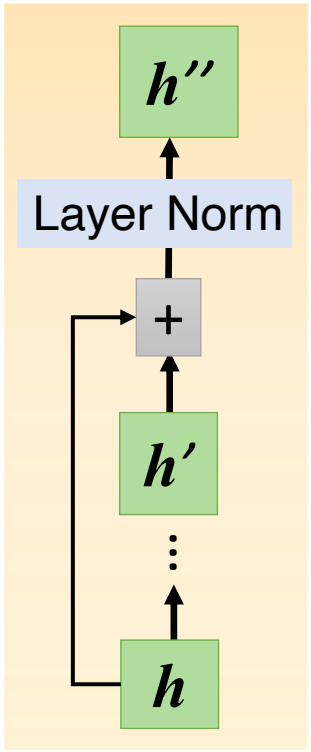
# Architecture

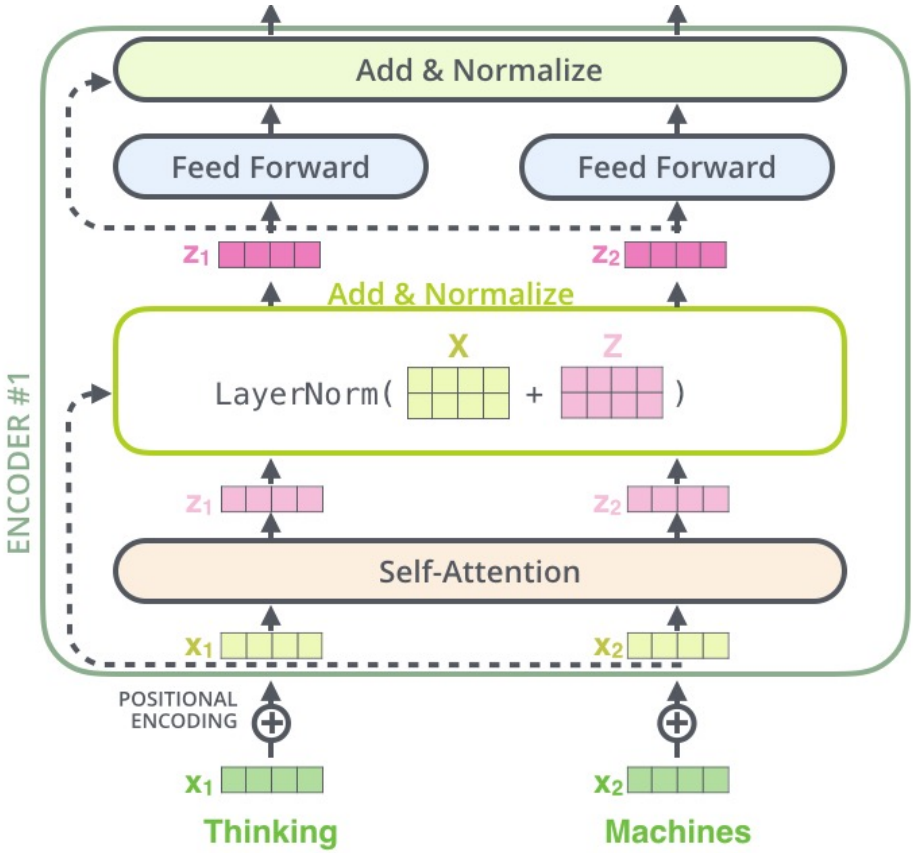
## Example: two words



# The Residuals*

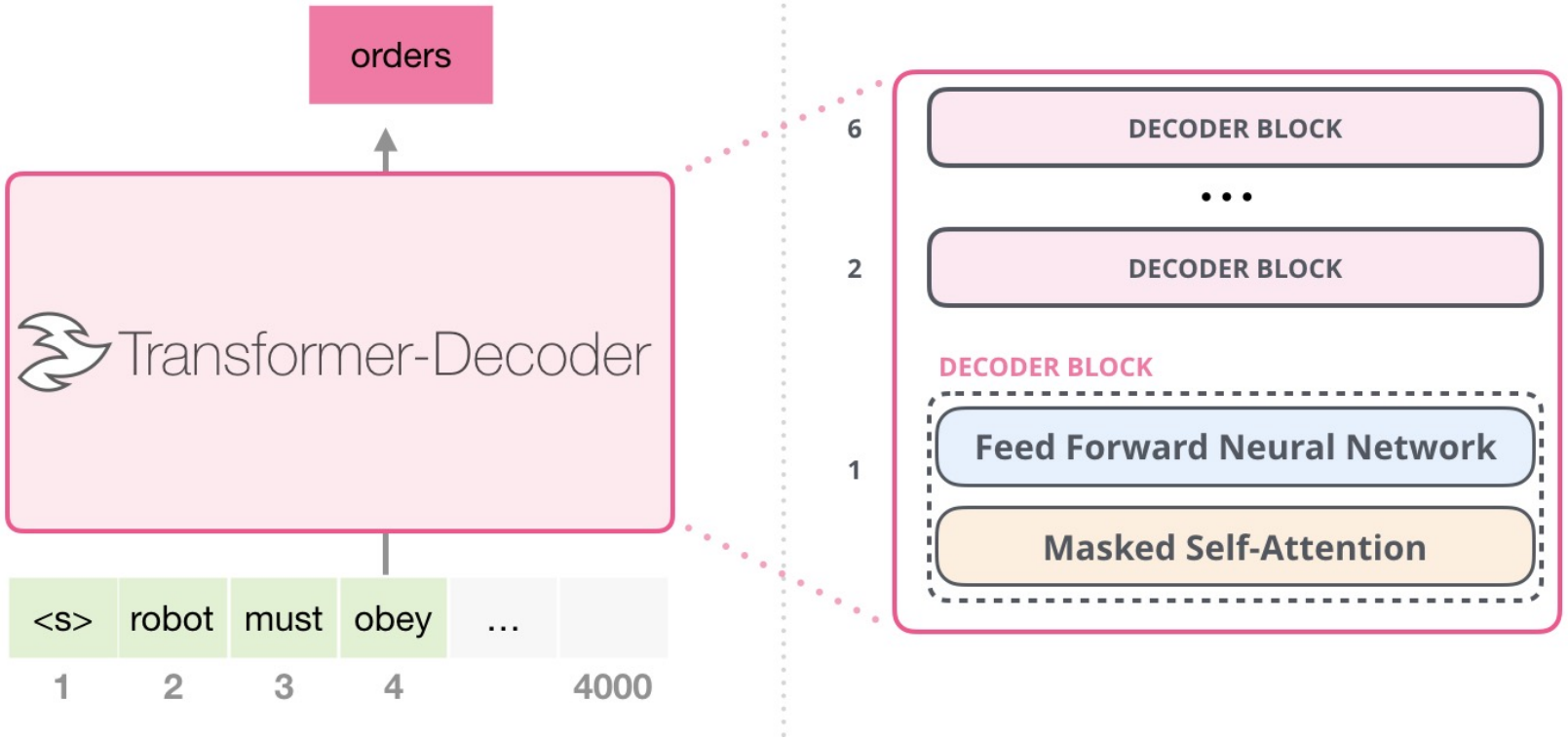
- Each sub-layer (self-attention, ffnn) has a residual connection around it, and is followed by a layer-normalization step.

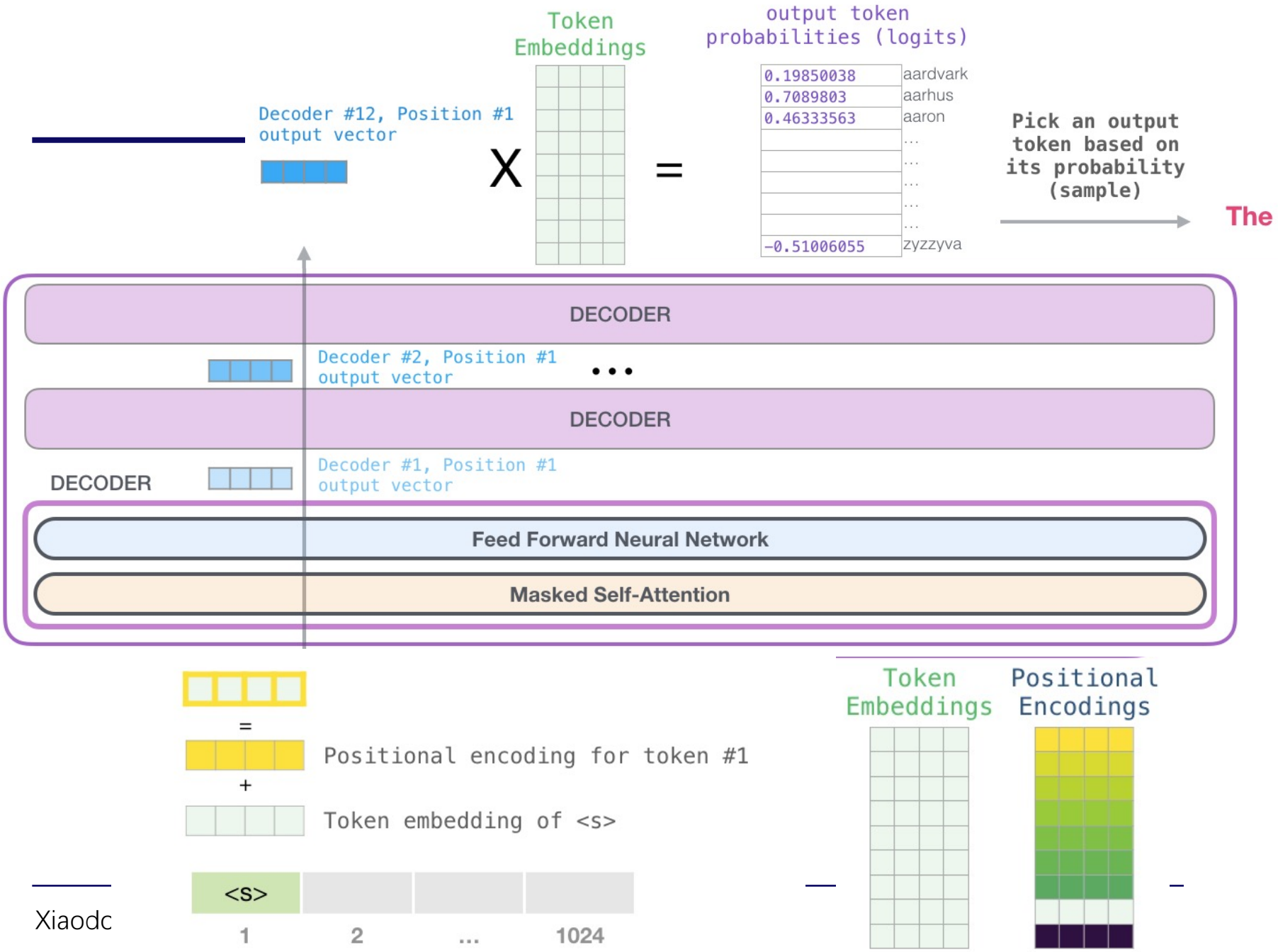



https://arxiv.org/abs/1607.06450

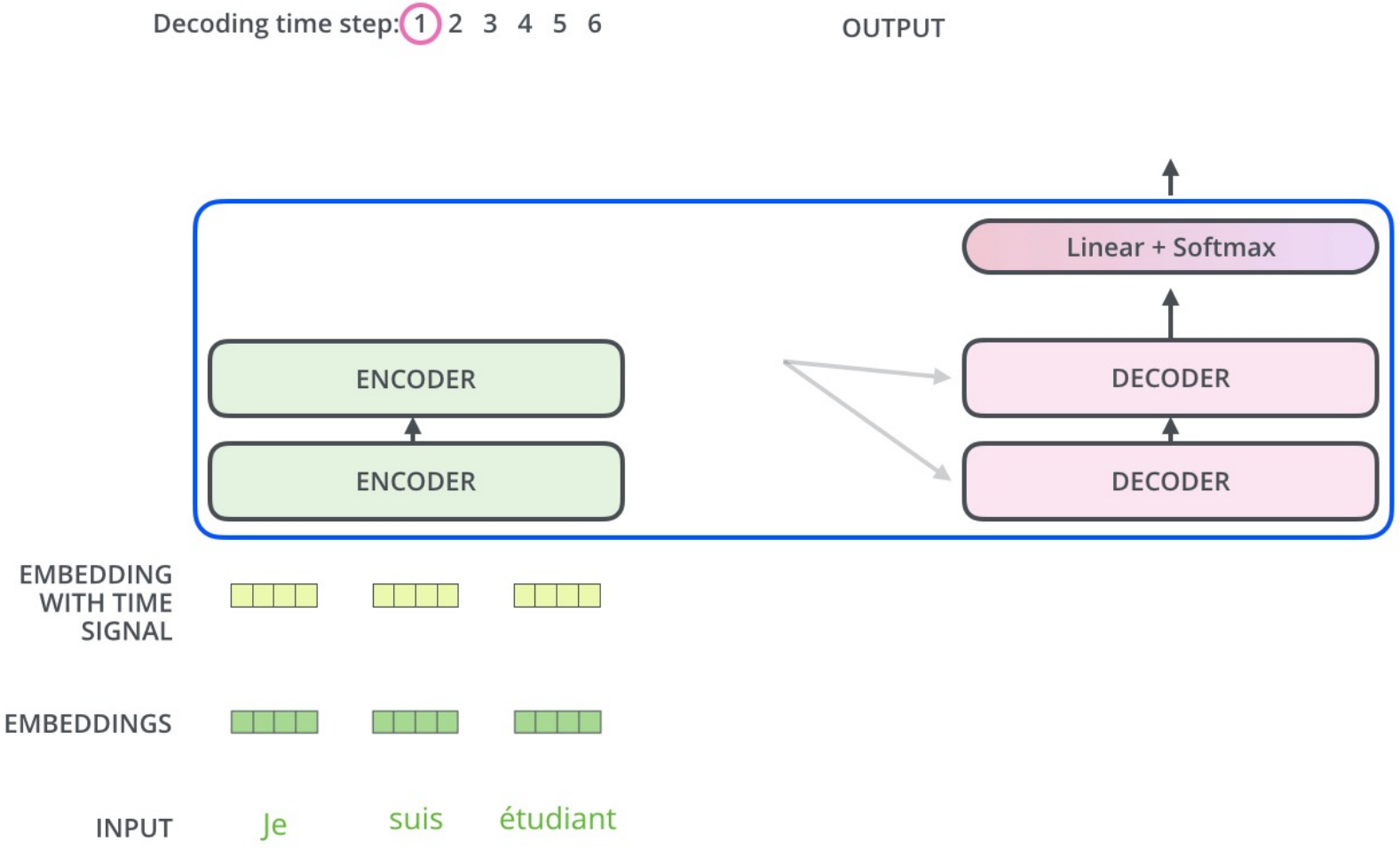
# The Decoder Side



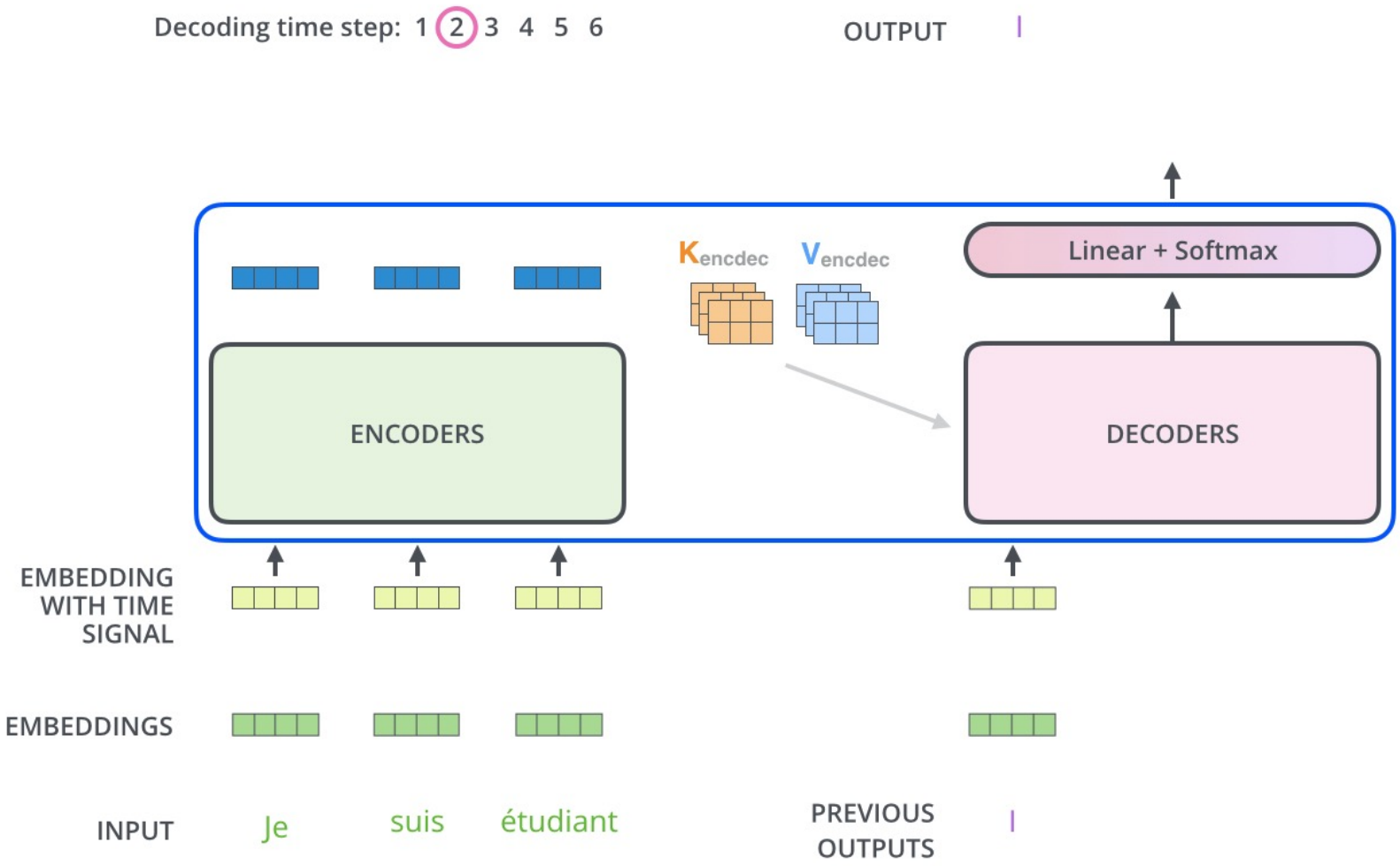

Token Embeddings
output token probabilities (logits)
Decoder #12, Position #1 output vector
X
=
0.19850038 aardvark
0.7089803 aarhus
0.46333563 aaron
-0.51006055 zyzzyva
Pick an output token based on its probability (sample)
The
DECODER
Decoder #2, Position #1 output vector
DECODER
Decoder #1, Position #1 output vector
DECODER
Feed Forward Neural Network
Masked Self-Attention
=
Positional encoding for token #1
+
Token embedding of <s>
<s>
1
2
...
1024
Token Embeddings
Positional Encodings

Xiaodc

# The Decoder Side

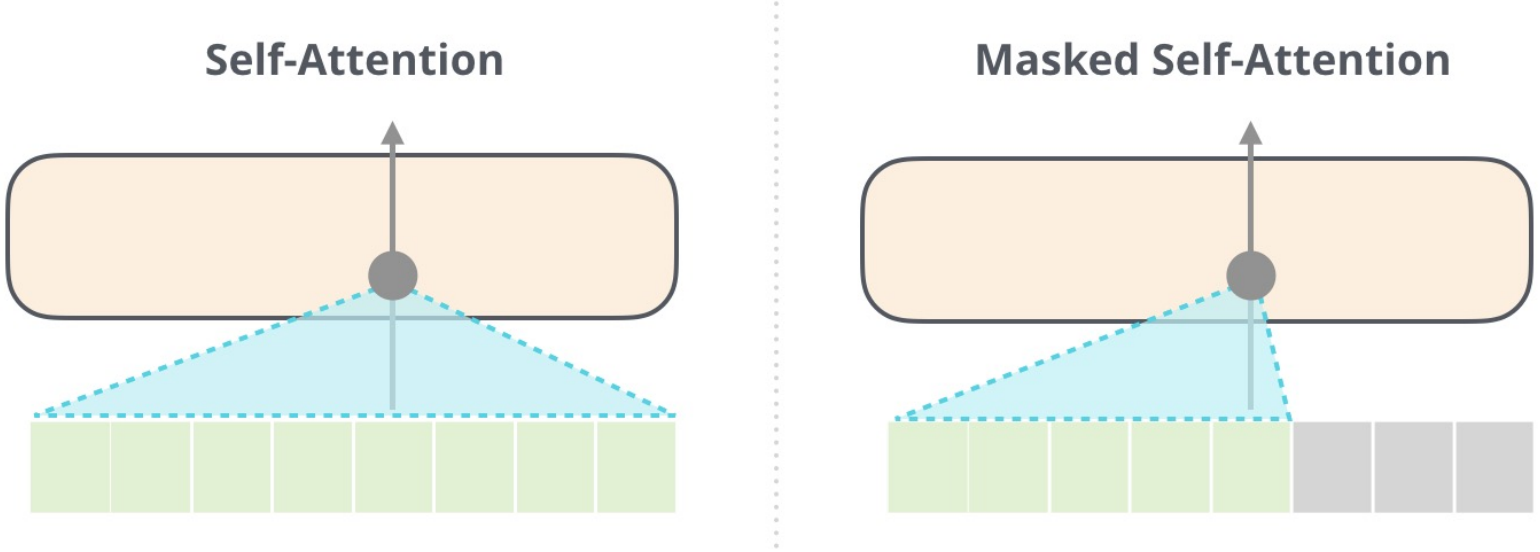


# The Decoder Side



# Attention Masks

- an L×L matrix that is applied to the attention scores
- avoid attention on specific tokens



# Attention Masks

**Scores**
(before softmax)

| | | | |
|---|---|---|---|
| 0.11 | 0.00 | 0.81 | 0.79 |
| 0.19 | 0.50 | 0.30 | 0.48 |
| 0.53 | 0.98 | 0.95 | 0.14 |
| 0.81 | 0.86 | 0.38 | 0.90 |

**Apply Attention Mask**

| | | | |
|---|---|---|---|
| 0 | -inf | -inf | -inf |
| 0 | 0 | -inf | -inf |
| 0 | 0 | 0 | -inf |
| 0 | 0 | 0 | 0 |

**Masked Scores**
(before softmax)

| | | | |
|---|---|---|---|
| 0.11 | -inf | -inf | -inf |
| 0.19 | 0.50 | -inf | -inf |
| 0.53 | 0.98 | 0.95 | -inf |
| 0.81 | 0.86 | 0.38 | 0.90 |

**Softmax (along rows)**

**Scores**

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0.48 | 0.52 | 0 | 0 |
| 0.31 | 0.35 | 0.34 | 0 |
| 0.25 | 0.26 | 0.23 | 0.26 |

# Attention Masks

## Self-Attention Mask

- for both encoder and decoder
- to ignore padding tokens


## Cross-Attention Mask

- for decoder only
- ignore padding tokens in the source sequence

## Causal Mask

- for decoder only
- attend on generated sequence

Output

| | | | | | |
|---|---|---|---|---|---|
| <EOS> | 1 | 1 | 1 | 1 | 1 |
| lunch | 1 | 1 | 1 | 1 | 0 |
| eating | 1 | 1 | 1 | 0 | 0 |
| love | 1 | 1 | 0 | 0 | 0 |
| I | 1 | 0 | 0 | 0 | 0 |

# A Live Demo

https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

# Training

## The Same as RNN Encoder-Decoder

**Data:** $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(N)}, y^{(N)})\}$,

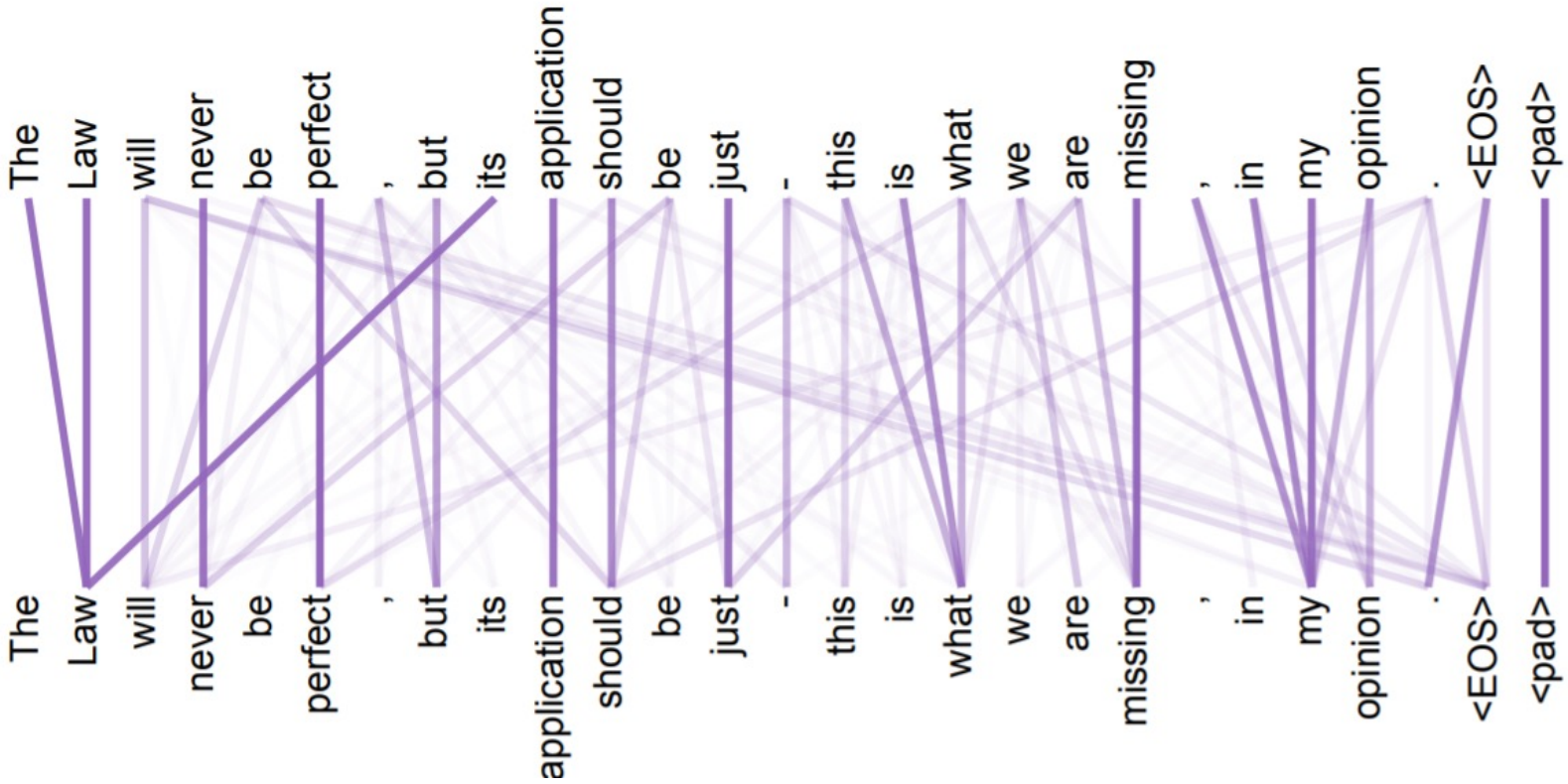
where $x^{(\ell)} = (x_1, \ldots x_{Tx})$ and $y^{(\ell)} = (y_1, \ldots y_{Ty})$.

**Loss Function** – minimize the cross-entropy loss:

$$L(\theta) = -\frac{1}{N} \sum_{\ell=1}^{N} \sum_{t=1}^{Ty} \log p_\theta(y_t^{(\ell)} | x^{(\ell)})$$

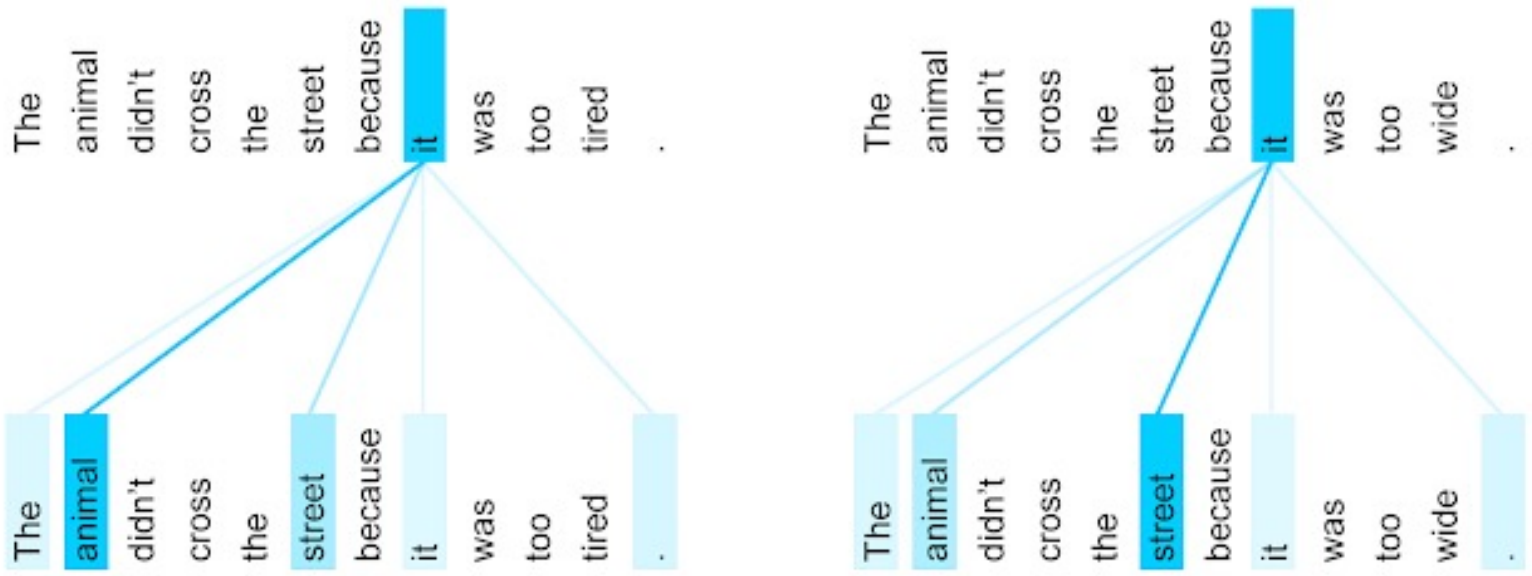
**Optimization** – gradient descend

# Visualizing Attention




https://arxiv.org/abs/1706.03762

# Visualizing Attention

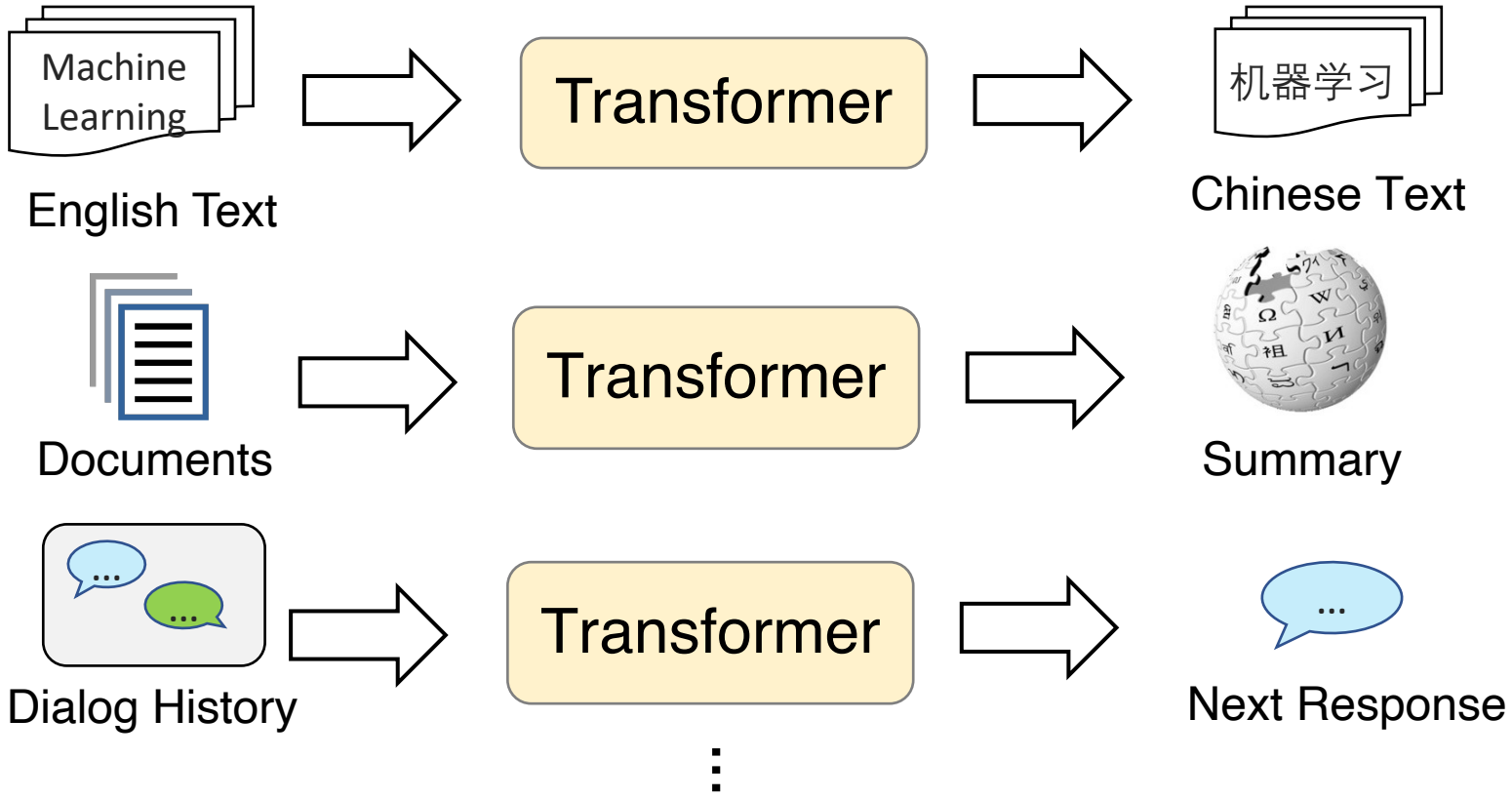



The encoder self-attention distribution for the word "it" from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

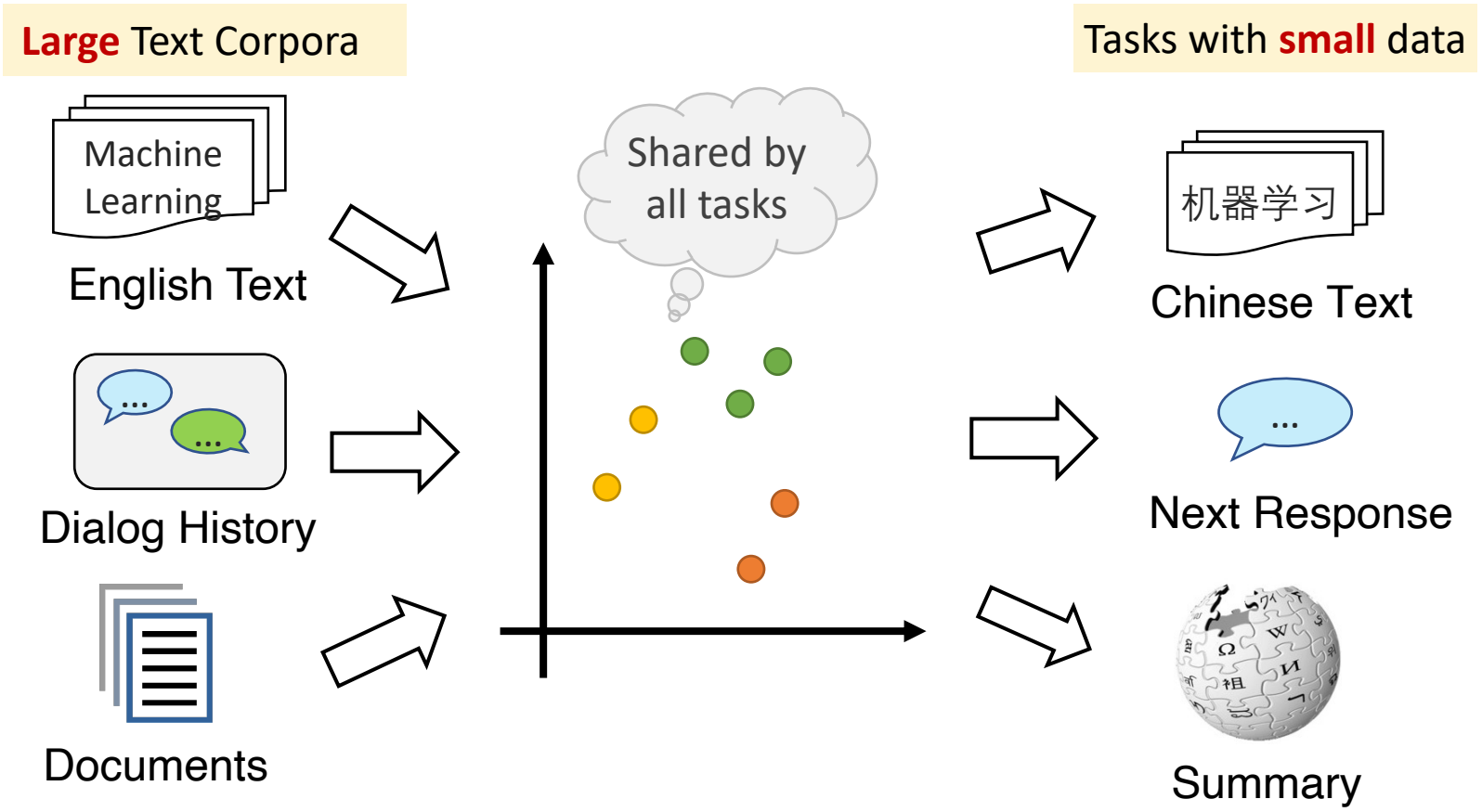
# Applications

Whenever we need sequence-to-sequence learning, we can always use Transformer.



# Thinking…

Is there a unified vector space for all texts?



# Pretrained Language Models

# Word Embedding Revisited

## 1-of-N Encoding

hotel = [ 1 0 0 0 0]

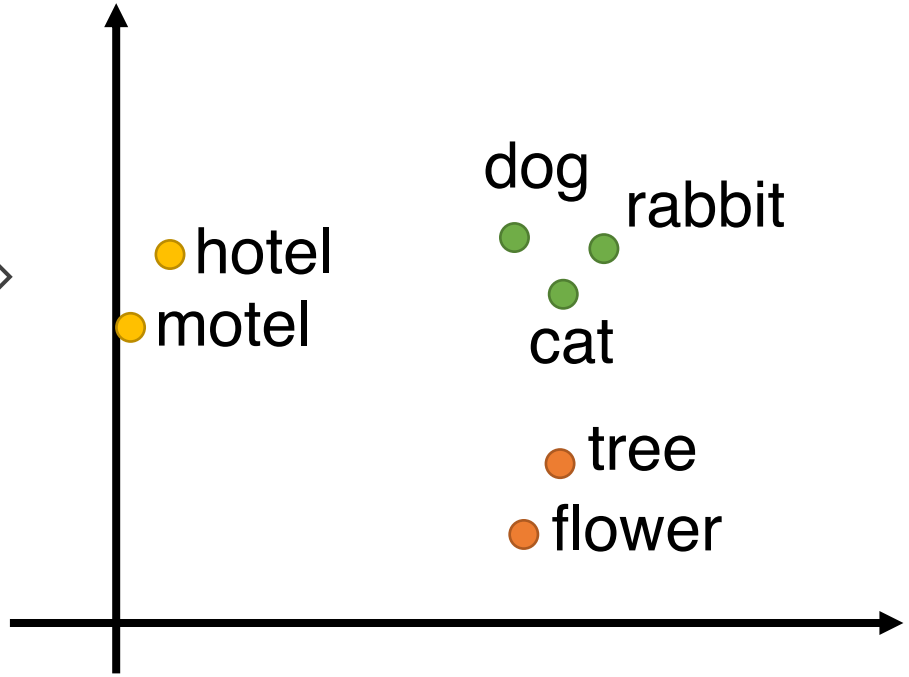
tree = [ 0 1 0 0 0]

apple = [ 0 0 1 0 0]

motel = [ 0 0 0 1 0]

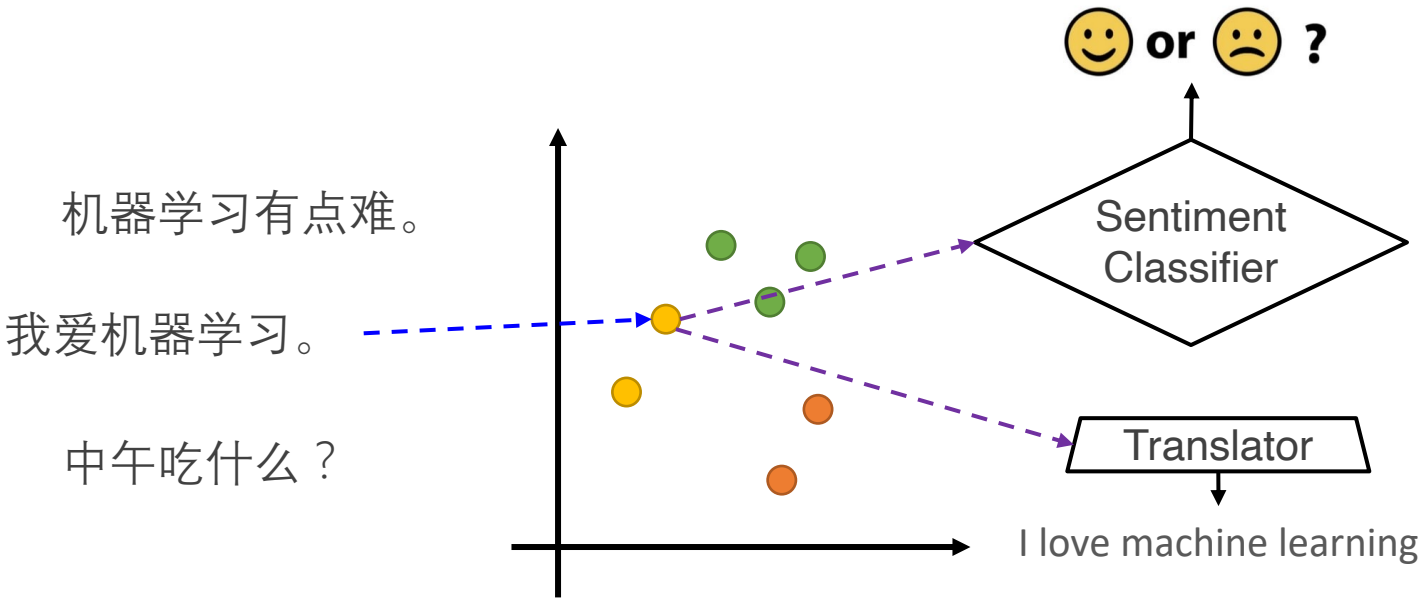
elephant = [ 0 0 0 0 1]

## Word Embedding



# Sentence Embeddings ?

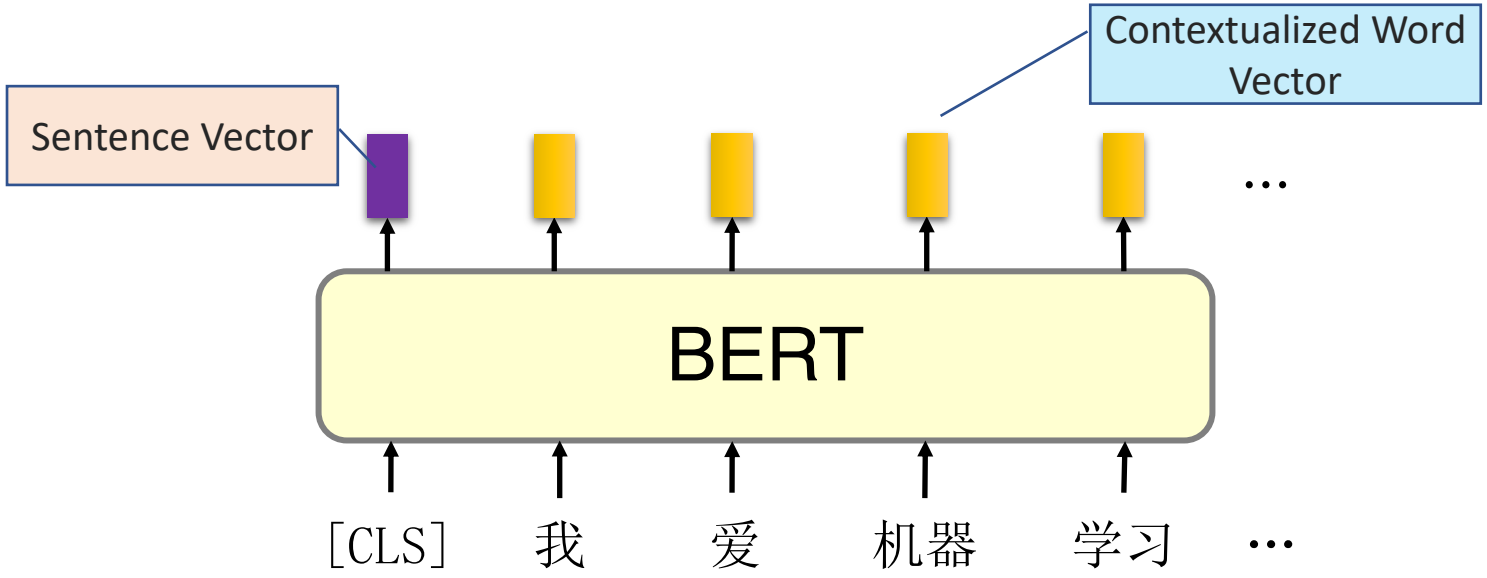
- a unified vector space for all sentences.
- can be trained from large-scale text corpus.
- can be reused for specific tasks with smaller data.



# Bidirectional Encoder Representations from Transformers (BERT)

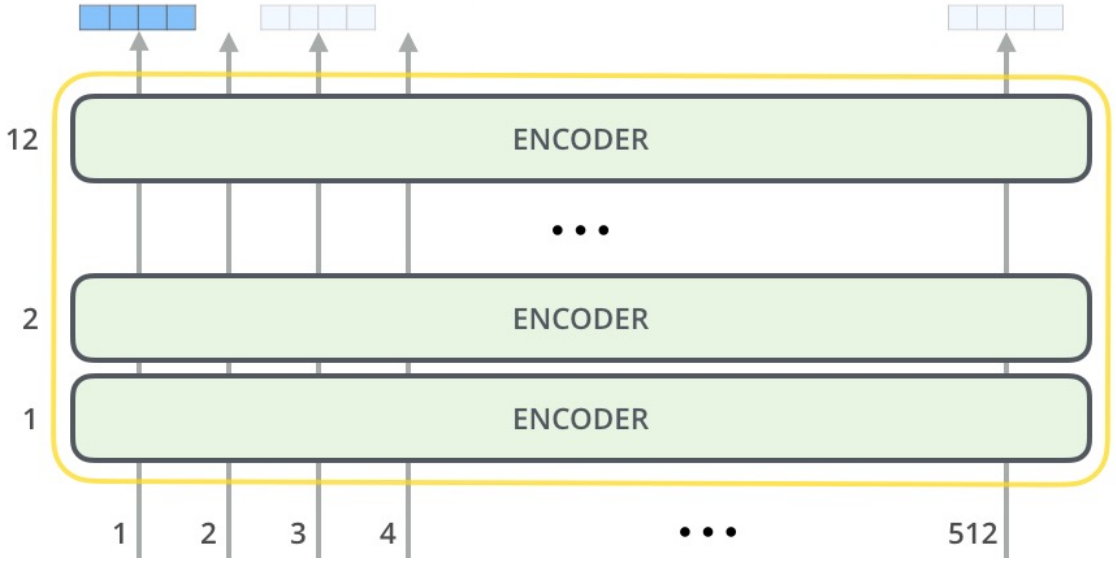
## A Transformer Encoder that

- allows for learning representations of words and sentences.
- pre-trained on large-scale text corpora and then;
- fine-tuned on smaller task-specific datasets (e.g., classification, QA.)

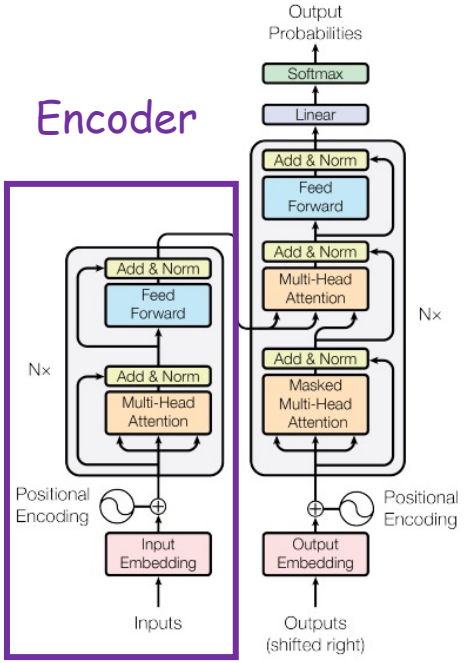


# Model Architecture

- Just like Transformer encoder, BERT takes a sequence of words as input which keep flowing up the stack.
- The first input token is always a special [CLS] token which stands for classification.

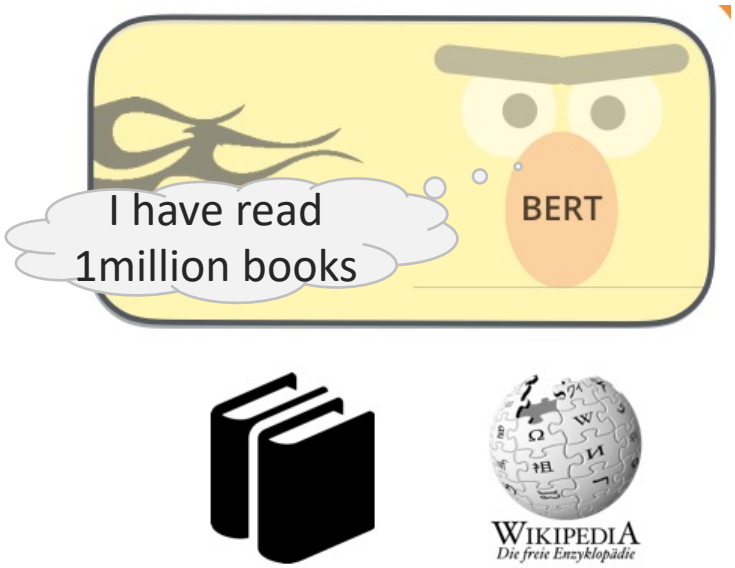



=



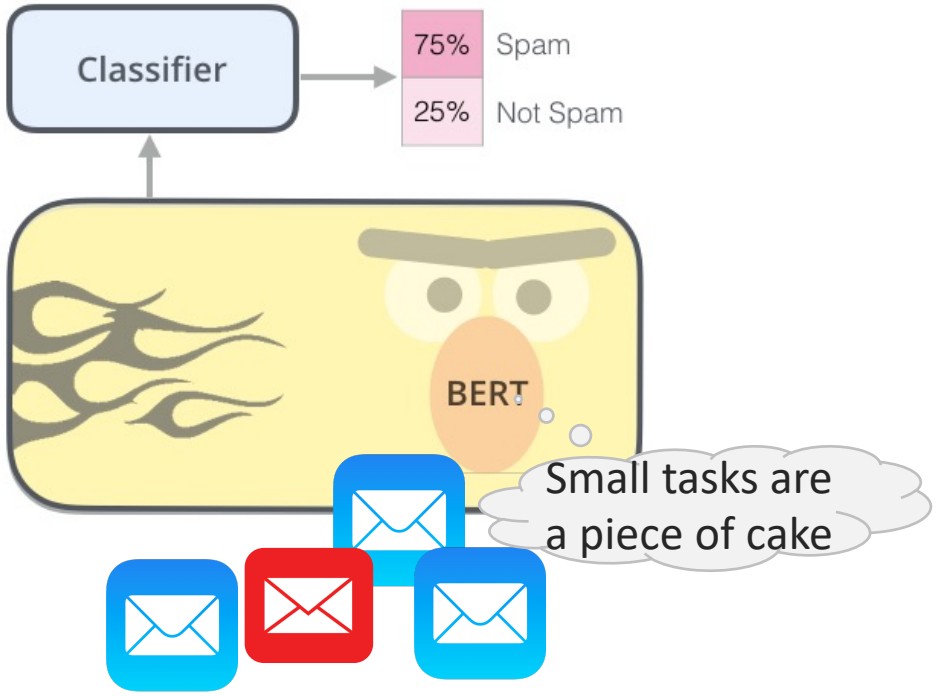
# The Pipeline

Unsupervised training on large amounts of text (e.g., books, Wikipedia, etc)




**Phase 1: Pre-Training**

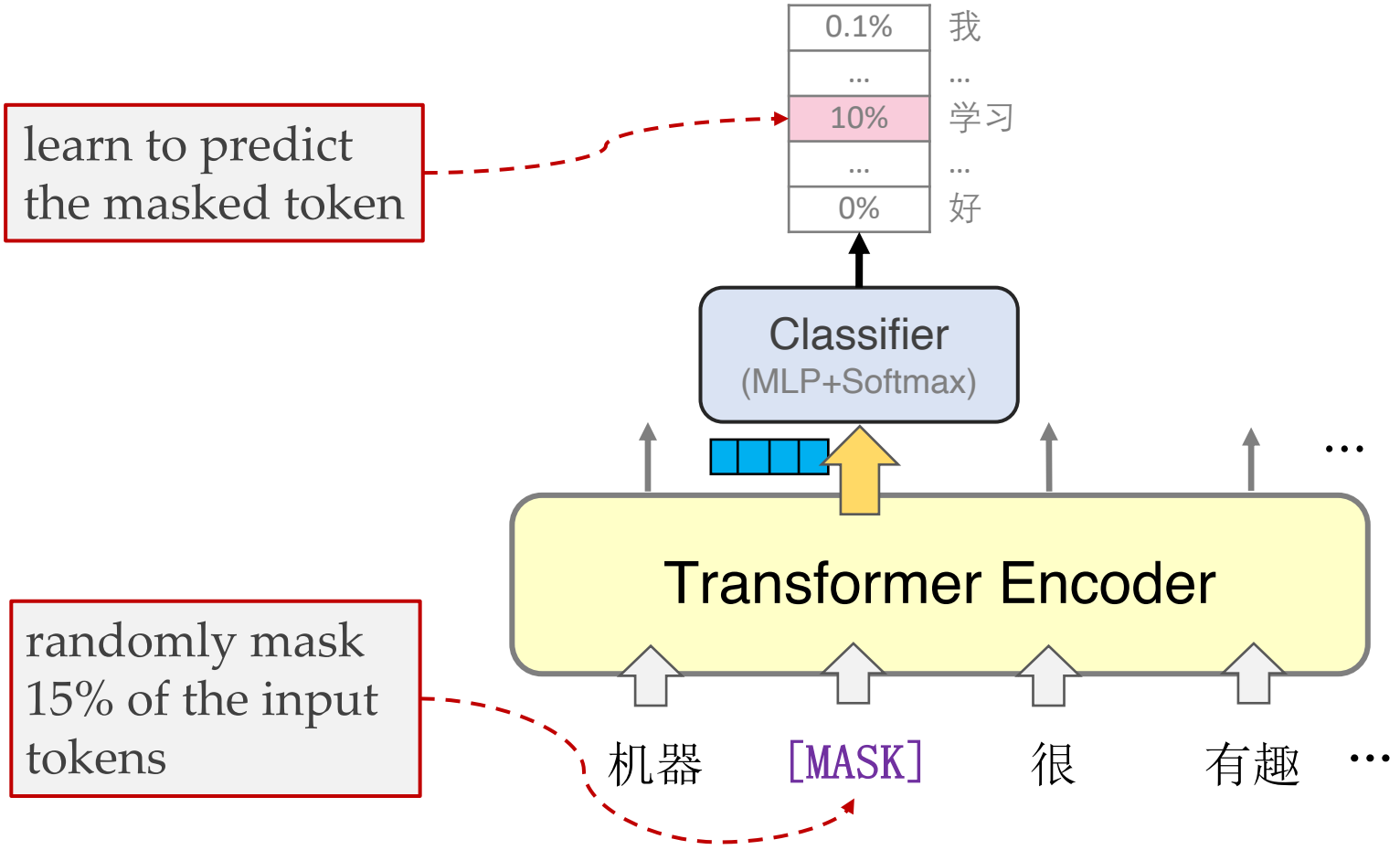
Supervised training on a specific task with a labeled dataset. (e.g., spam detection)




**Phase 2: Fine-Tuning**

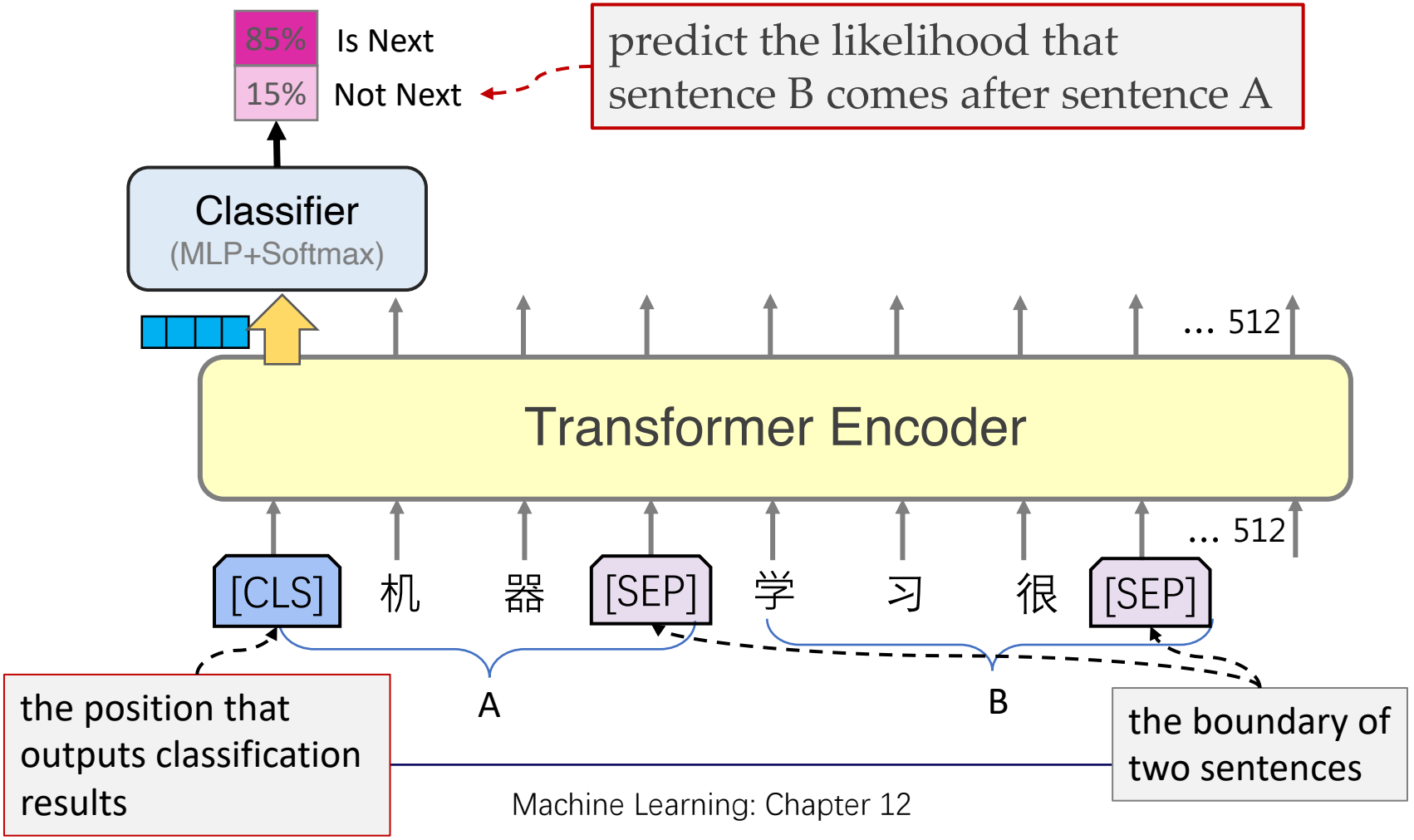
# Pre-Training

## Task 1: Masked Language Model



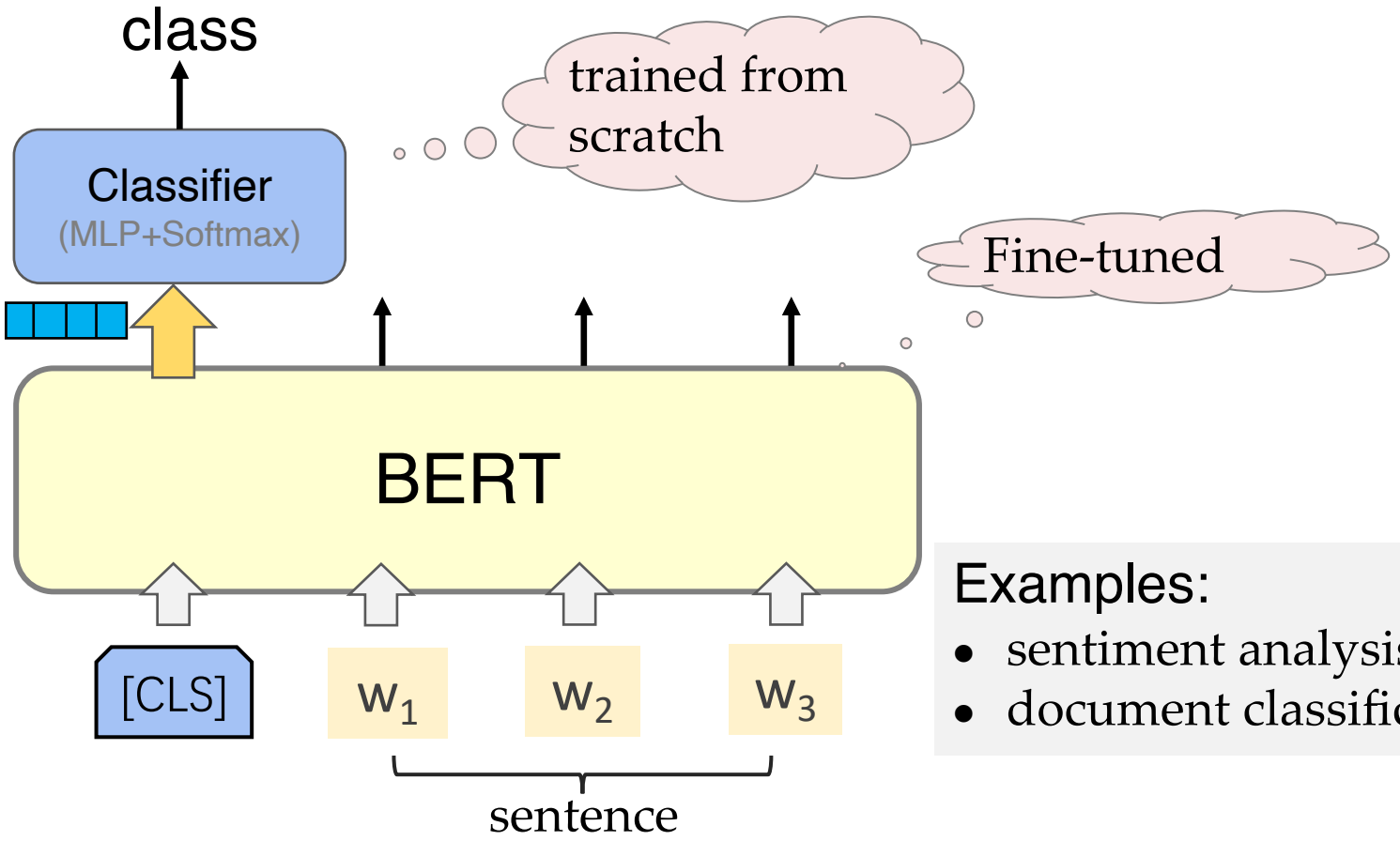
# Pre-Training

## Task 2: Next Sentence Prediction



# Finetuning: Sentence Classification

- **input**: a single sentence, **output**: class

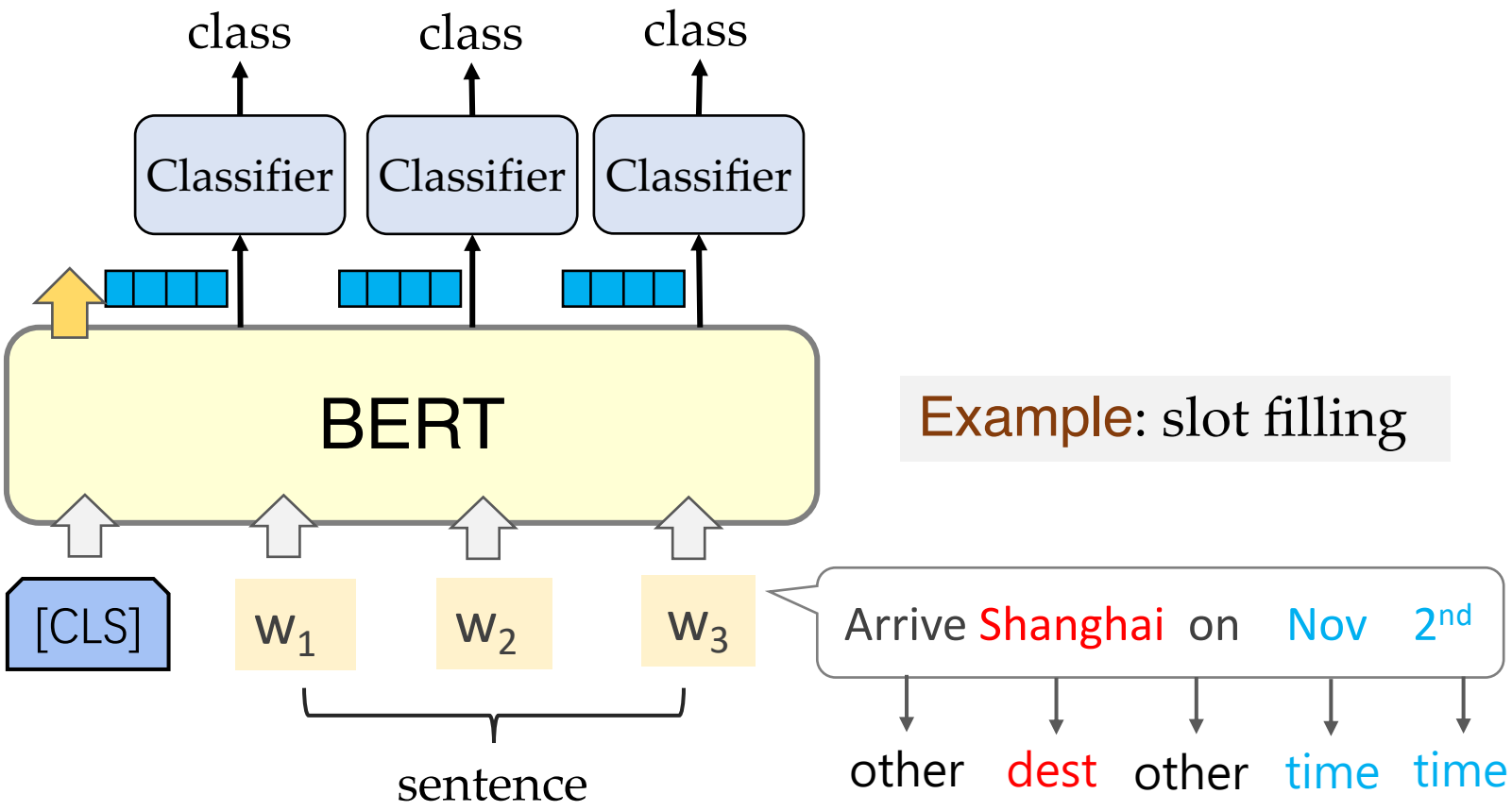



Examples:

- sentiment analysis
- document classification

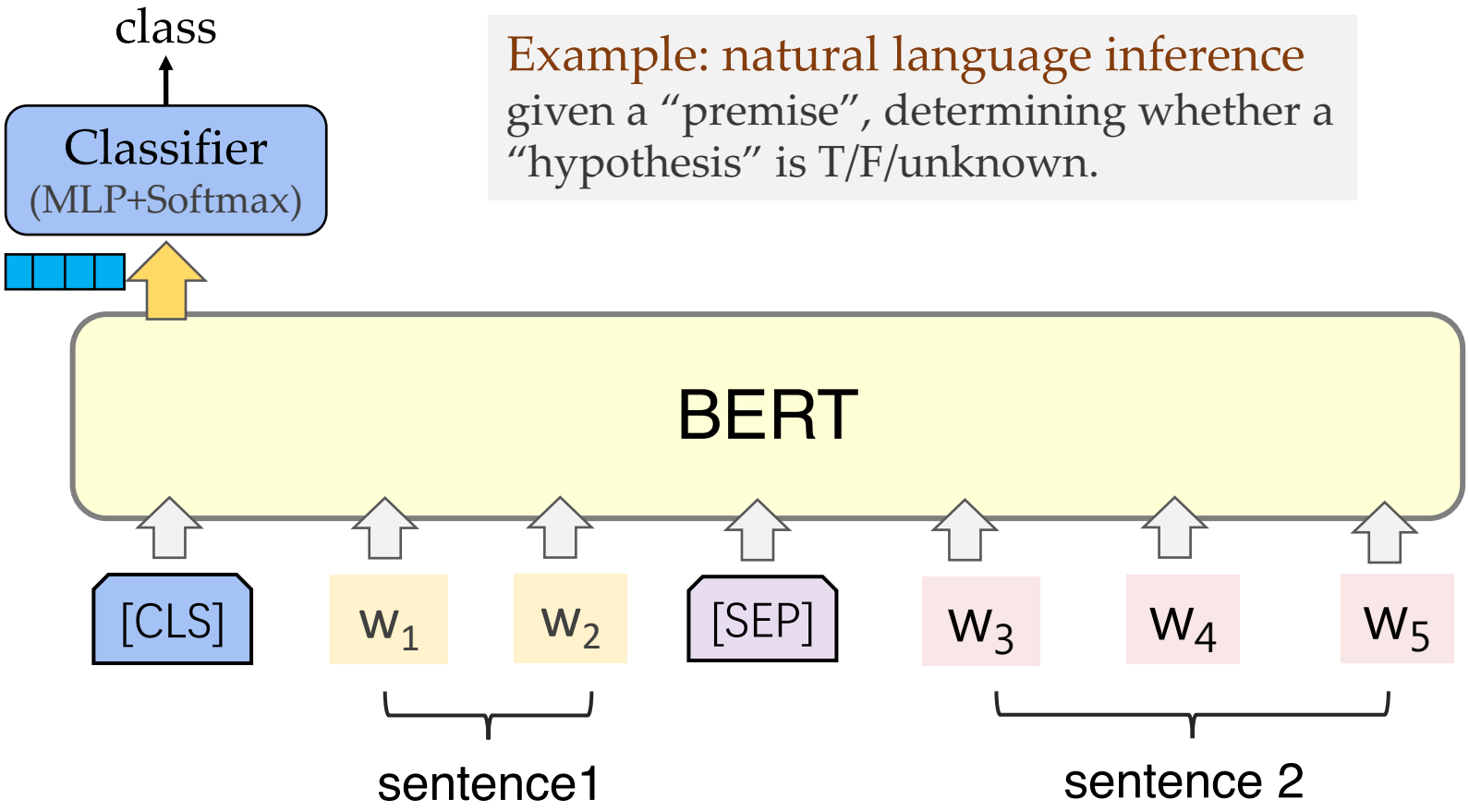
# Finetuning – Word Tagging

Input: a single sentence  Output: class of each word



# Finetuning – Classifying Sentence Pairs

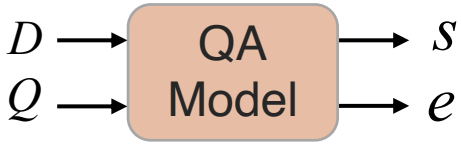
**input**: two sentences **output**: class




Example: natural language inference
given a "premise", determining whether a "hypothesis" is T/F/unknown.

# Finetuning – QA (Reading Comprehension)

**Task:** find the answer for a question from a document (e.g. SQuAD)




answer A = $\{d_s, \ldots, d_e\}$

**Input:**
a document D=$\{d_1, \ldots, d_N\}$
a question Q=$\{q_1,\ldots,q_N\}$

**Ouput:** two integers ($s$, $e$) which indicate the start and the end positions of the answer in the document.

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under gravity. The main forms of precipitation include drizzle, rain, sleet, snow, graupel and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals within a cloud. Short, intense periods of rain in scattered locations are called "showers".

17 77 79

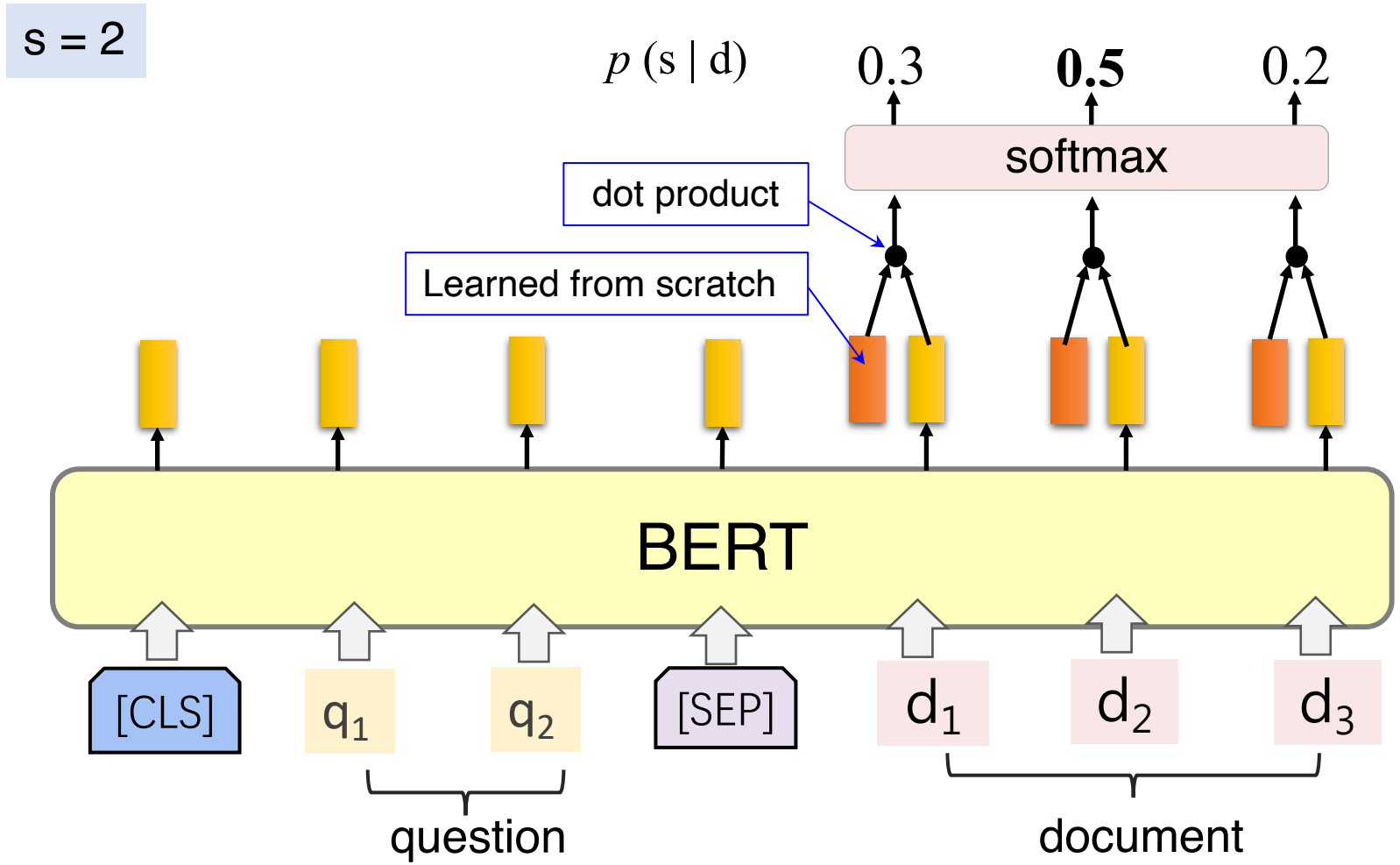
Q: what causes precipitations to fall?

A: gravity (s=17, e=17)

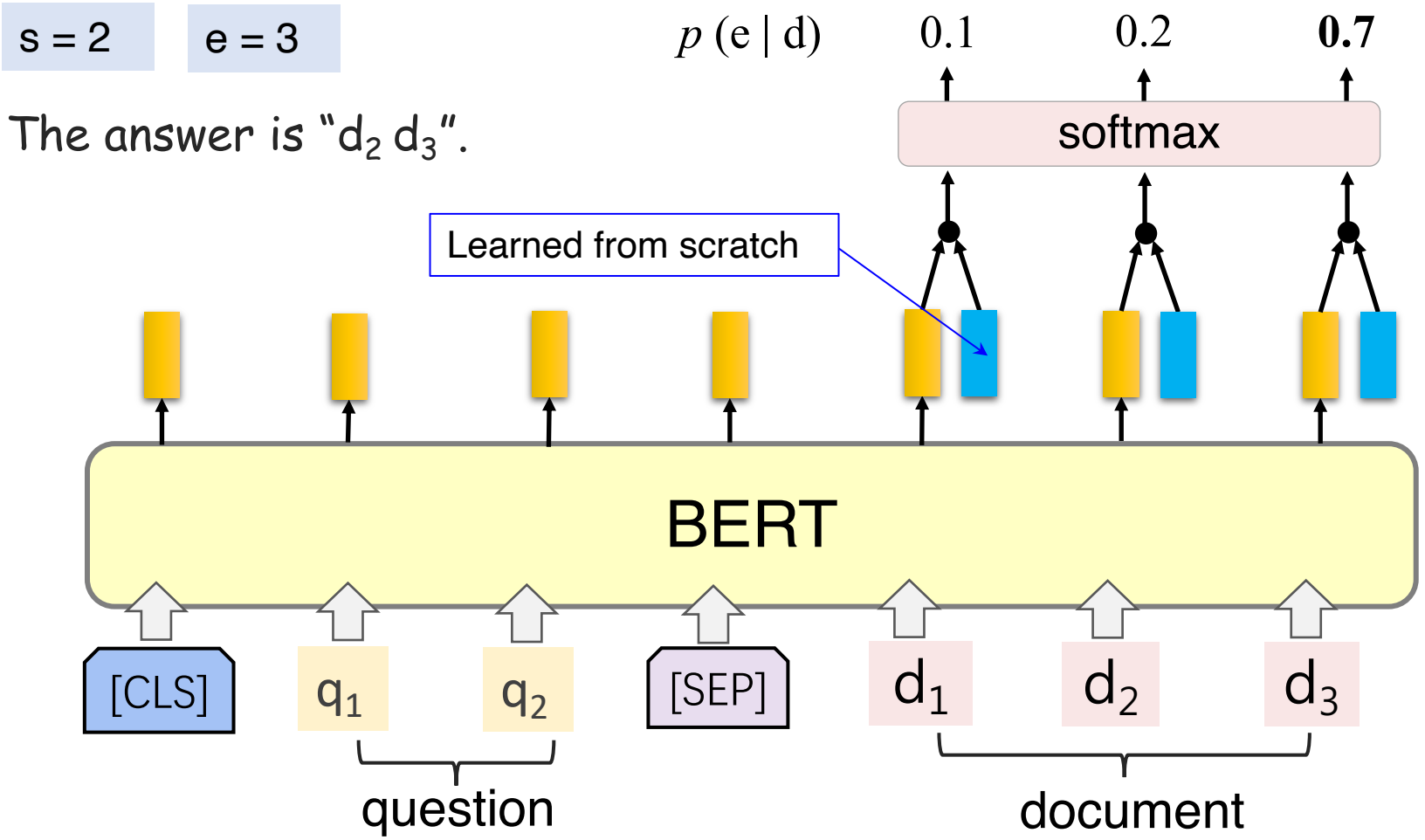
Q: where do water droplets collide with ice crystals to form precipitations?

A: within a cloud (s=77, e=79)

# Finetuning – QA (Reading Comprehension)



# Finetuning – QA (Reading Comprehension)

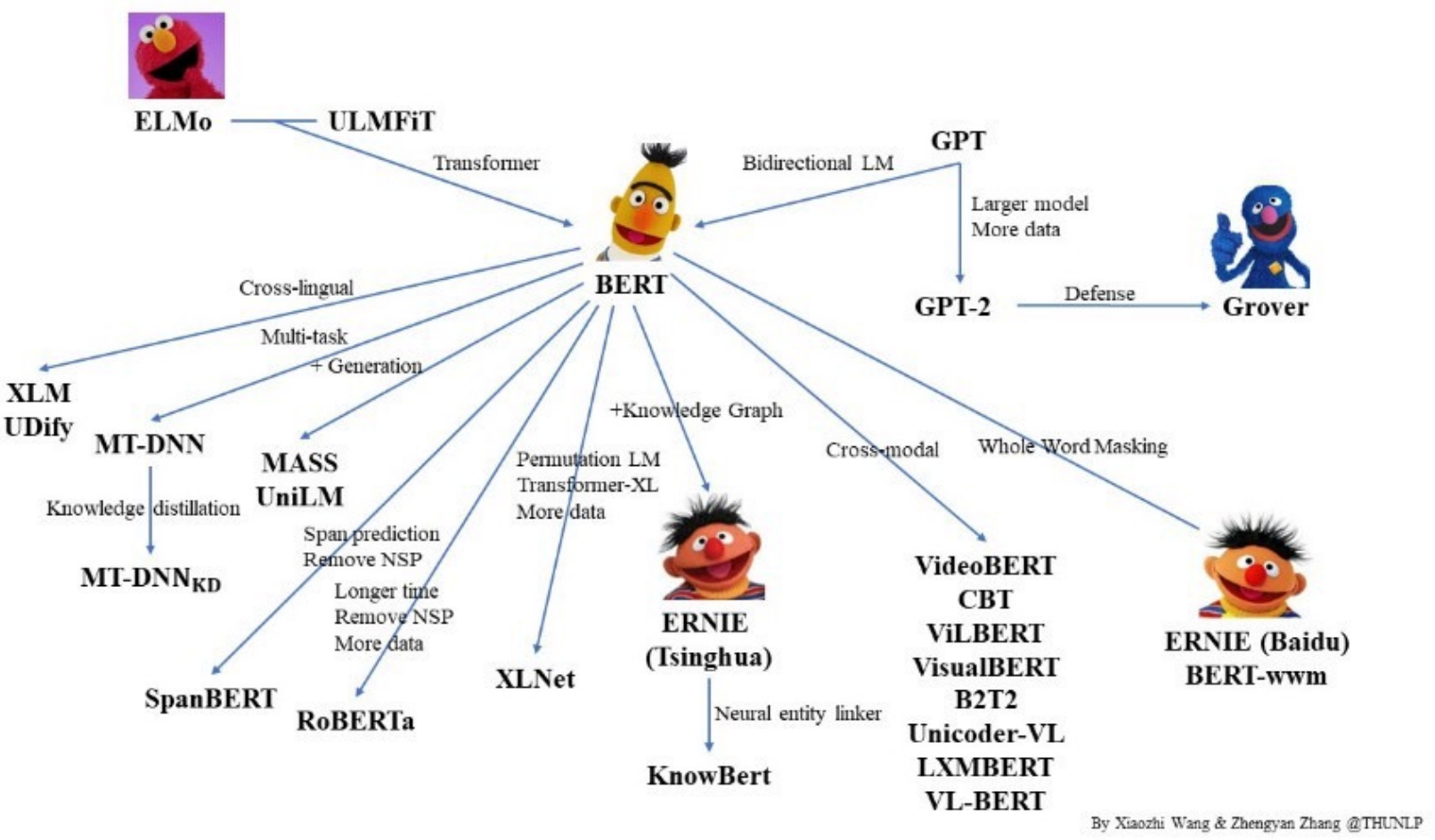


# BERT屠榜……

SQuAD 2.0

| Rank | Model | EM | F1 |
|---|---|---|---|
| | Human Performance<br>*Stanford University*<br>(Rajpurkar & Jia et al. '18) | 86.831 | 89.452 |
| 1<br>Mar 20, 2019 | BERT + DAE + AoA (ensemble)<br>*Joint Laboratory of HIT and iFLYTEK Research* | **87.147** | **89.474** |
| 2<br>Mar 15, 2019 | BERT + ConvLSTM + MTL + Verifier (ensemble)<br>*Layer 6 AI* | 86.730 | 89.286 |
| 3<br>Mar 05, 2019 | BERT + N-Gram Masking + Synthetic Self-Training (ensemble)<br>*Google AI Language*<br>https://github.com/google-research/bert | 86.673 | 89.147 |
| 4<br>May 21, 2019 | XLNet (single model)<br>*XLNet Team* | 86.346 | 89.133 |
| 5<br>Apr 13, 2019 | SemBERT(ensemble)<br>*Shanghai Jiao Tong University* | 86.166 | 88.886 |

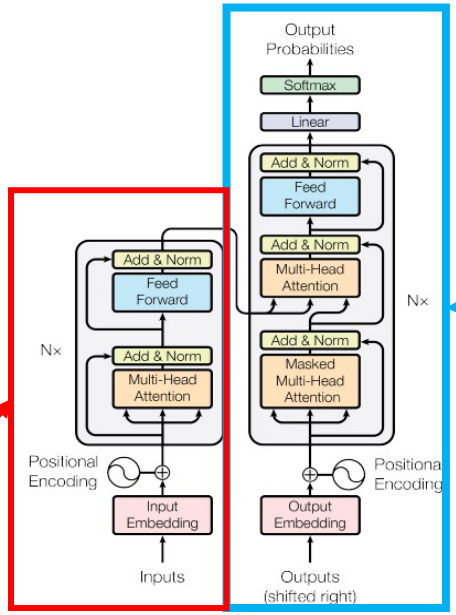
# The PLM Family



# Generative Pre-Training (GPT)

A pretrained language model that is built on top of a Transformer decoder.

BERT (340M)




GPT-2 (1542M)

Source of image: https://huaban.com/pins/1714071707/

# Pre-Training

Simply language model
(next token prediction)