

# Lecture 8:

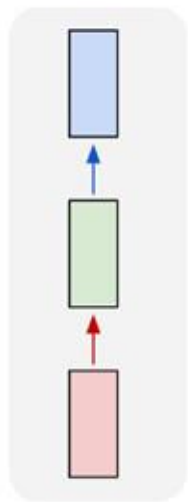
# Attention and Transformers

# Administrative

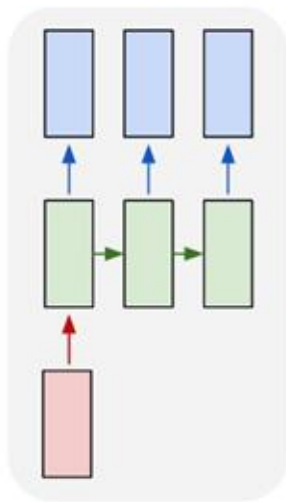
- Today: Project proposal due
- Today: Assignment 2 out, due 5/7
- Tomorrow 4/24: PyTorch review session

# Last Time: Recurrent Neural Networks

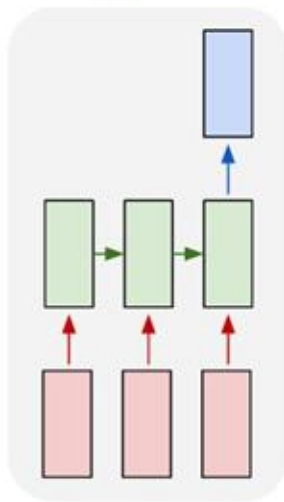
one to one



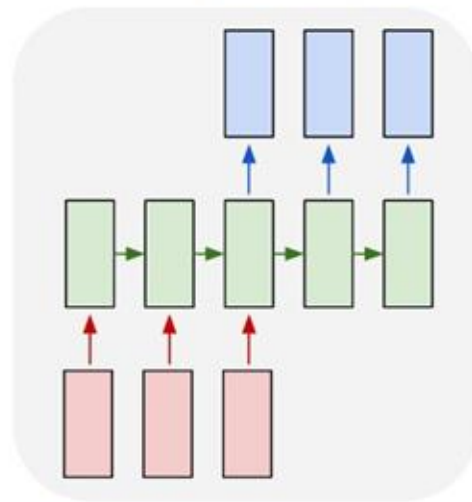
one to many



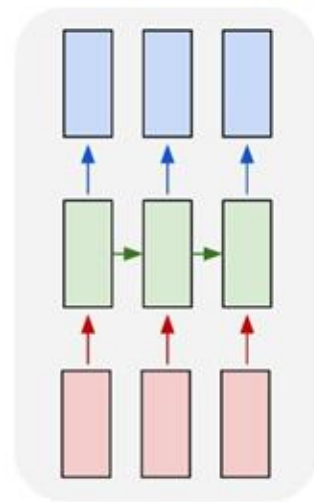
many to one



many to many

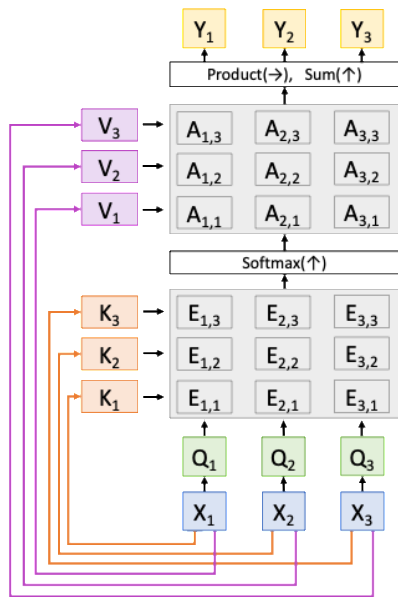


many to many

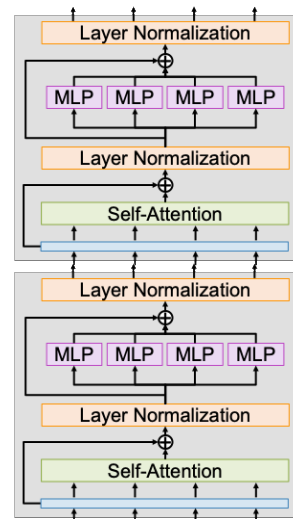


# Today: Attention + Transformers

**Attention:** A new primitive that operates on sets of vectors

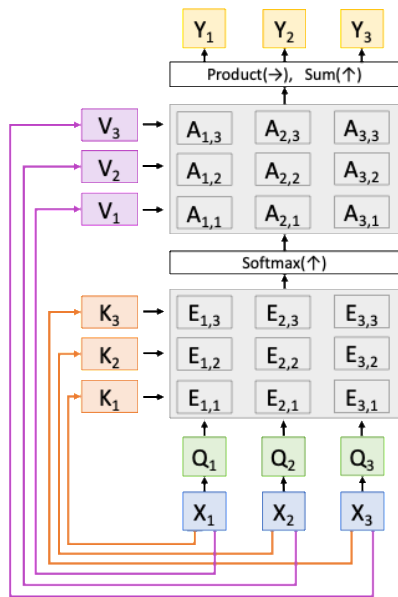


**Transformer:** A neural network architecture that uses attention everywhere

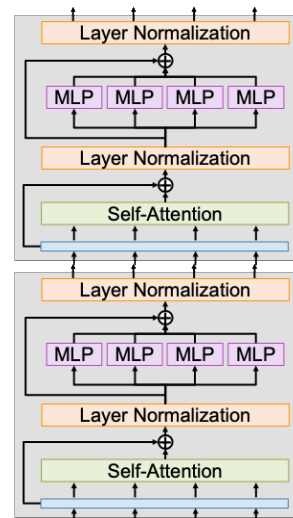


# Today: Attention + Transformers

**Attention:** A new primitive that operates on sets of vectors



**Transformer:** A neural network architecture that uses attention everywhere



Transformers are used everywhere today!

But they developed as an offshoot of RNNs so let's start there

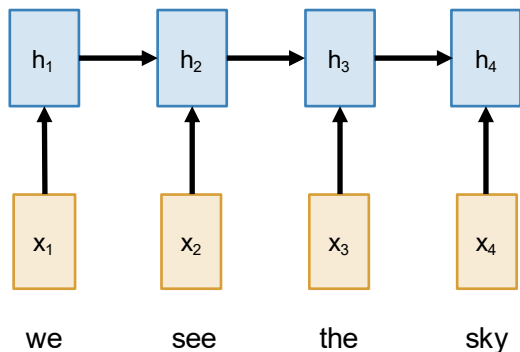
# Sequence to Sequence with RNNs: Encoder - Decoder

**Input:** Sequence  $x_1, \dots, x_T$

**Output:** Sequence  $y_1, \dots, y_T$

**A motivating example for today's discussion –  
machine translation! English  $\rightarrow$  Italian**

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$

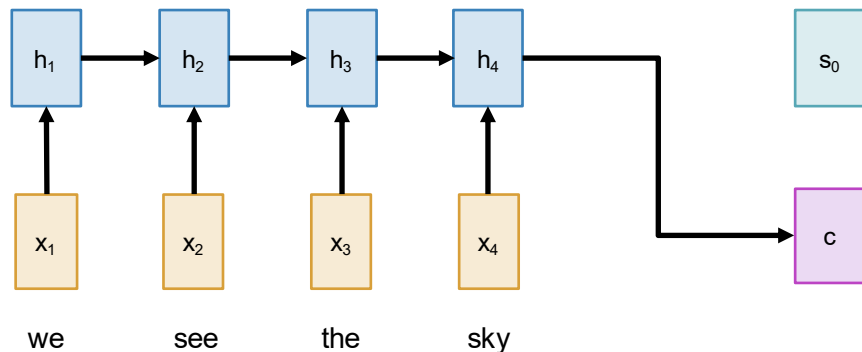


# Sequence to Sequence with RNNs

**Input:** Sequence  $x_1, \dots, x_T$

**Output:** Sequence  $y_1, \dots, y_T$

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$  From final hidden state predict:  
**Initial decoder state**  $s_0$   
**Context vector**  $c$  (often  $c=h_T$ )



# Sequence to Sequence with RNNs

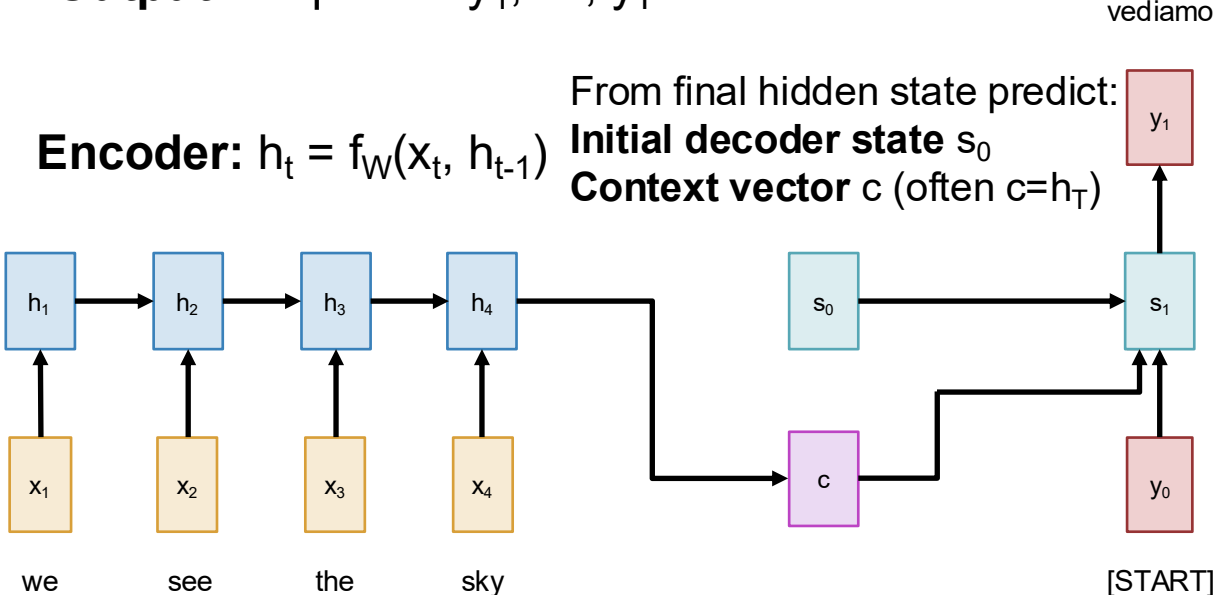
**Input:** Sequence  $x_1, \dots, x_T$

**Output:** Sequence  $y_1, \dots, y_T$

**Decoder:**  $s_t = g_U(y_{t-1}, s_{t-1}, c)$

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$

From final hidden state predict:  
**Initial decoder state**  $s_0$   
**Context vector**  $c$  (often  $c=h_T$ )



Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

# Sequence to Sequence with RNNs

**Input:** Sequence  $x_1, \dots, x_T$

**Output:** Sequence  $y_1, \dots, y_T$

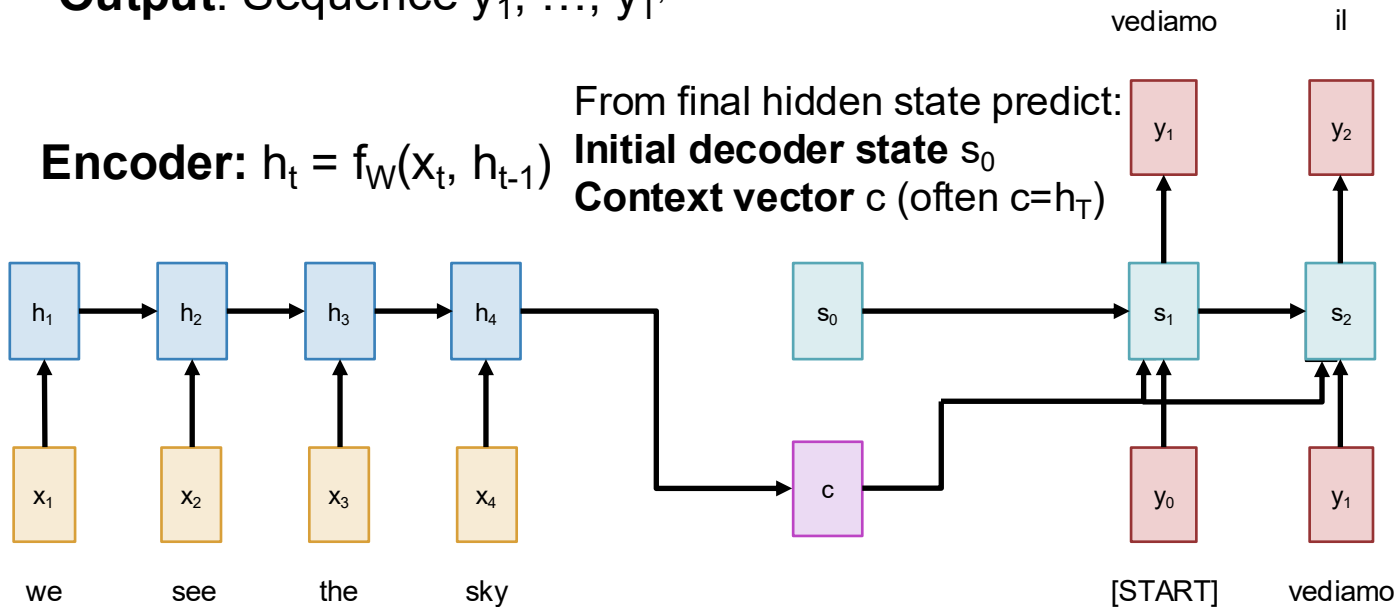
**Decoder:**  $s_t = g_U(y_{t-1}, s_{t-1}, c)$

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$

From final hidden state predict:

**Initial decoder state**  $s_0$

**Context vector**  $c$  (often  $c=h_T$ )



# Sequence to Sequence with RNNs

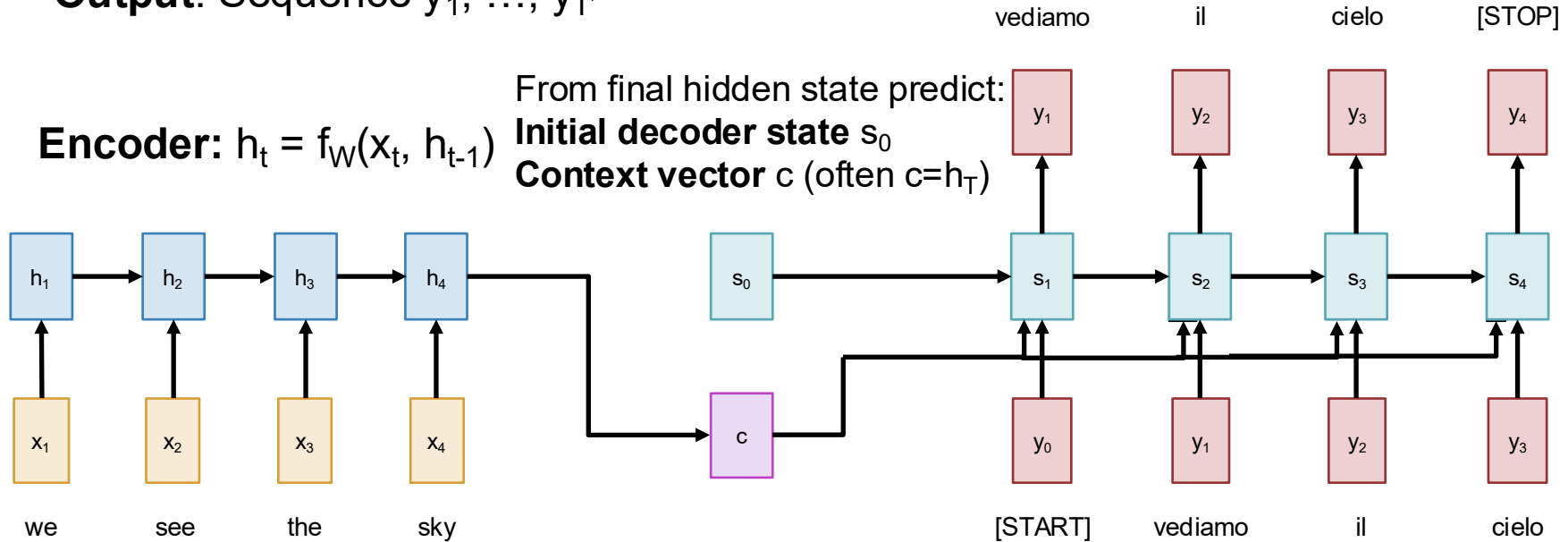
**Input:** Sequence  $x_1, \dots, x_T$

**Output:** Sequence  $y_1, \dots, y_T$

**Decoder:**  $s_t = g_U(y_{t-1}, s_{t-1}, c)$

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$

From final hidden state predict:  
**Initial decoder state**  $s_0$   
**Context vector**  $c$  (often  $c=h_T$ )



# Sequence to Sequence with RNNs

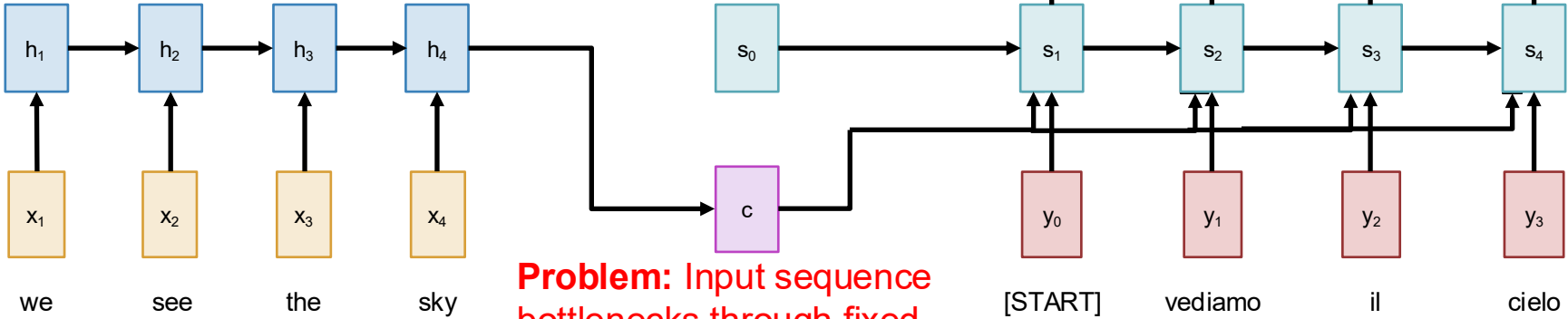
**Input:** Sequence  $x_1, \dots, x_T$

**Output:** Sequence  $y_1, \dots, y_T$

**Decoder:**  $s_t = g_U(y_{t-1}, s_{t-1}, c)$

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$

From final hidden state predict:  
**Initial decoder state**  $s_0$   
**Context vector**  $c$  (often  $c=h_T$ )



**Problem:** Input sequence bottlenecks through fixed sized  $c$ . What if  $T=1000$ ?

Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

# Sequence to Sequence with RNNs

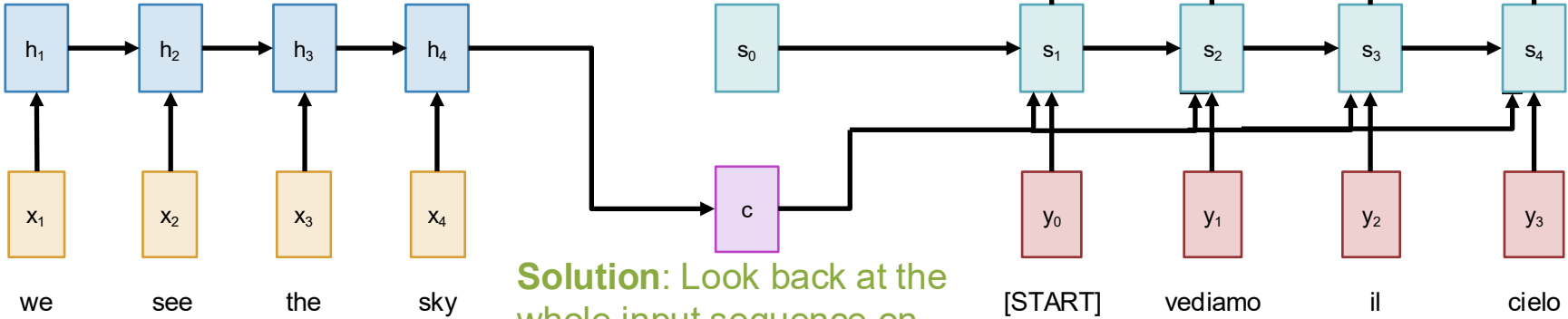
**Input:** Sequence  $x_1, \dots, x_T$

**Output:** Sequence  $y_1, \dots, y_T$

**Decoder:**  $s_t = g_U(y_{t-1}, s_{t-1}, c)$

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$

From final hidden state predict:  
**Initial decoder state**  $s_0$   
**Context vector**  $c$  (often  $c=h_T$ )



**Solution:** Look back at the whole input sequence on each step of the output

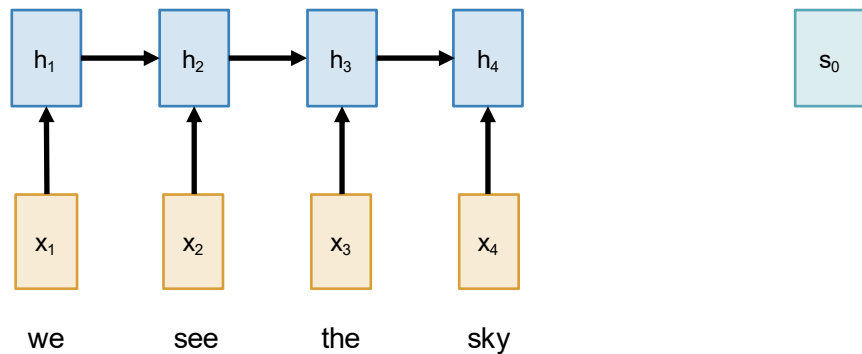
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

# Sequence to Sequence with RNNs and Attention

**Input:** Sequence  $x_1, \dots, x_T$

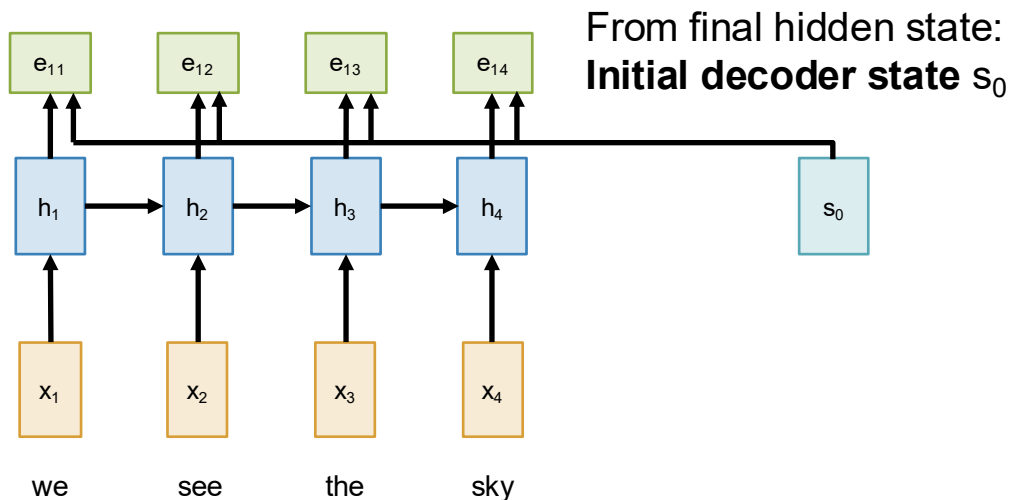
**Output:** Sequence  $y_1, \dots, y_T$

**Encoder:**  $h_t = f_W(x_t, h_{t-1})$  From final hidden state:  
**Initial decoder state**  $s_0$

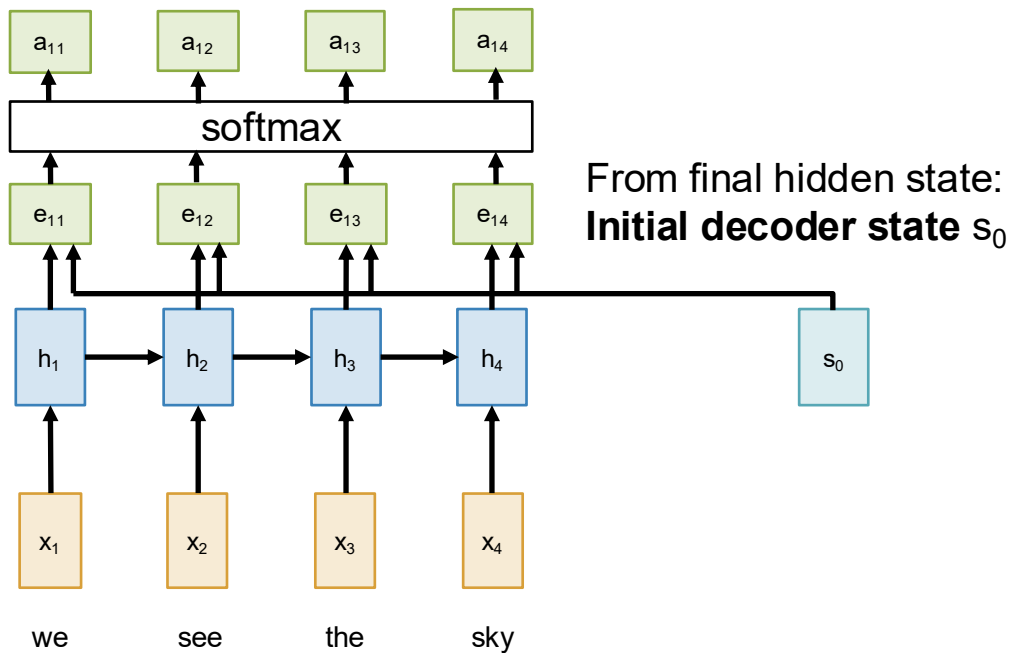


# Sequence to Sequence with RNNs and Attention

Compute (scalar) **alignment scores**  
 $e_{t,i} = f_{\text{att}}(s_{t-1}, h_i)$  ( $f_{\text{att}}$  is a Linear Layer)



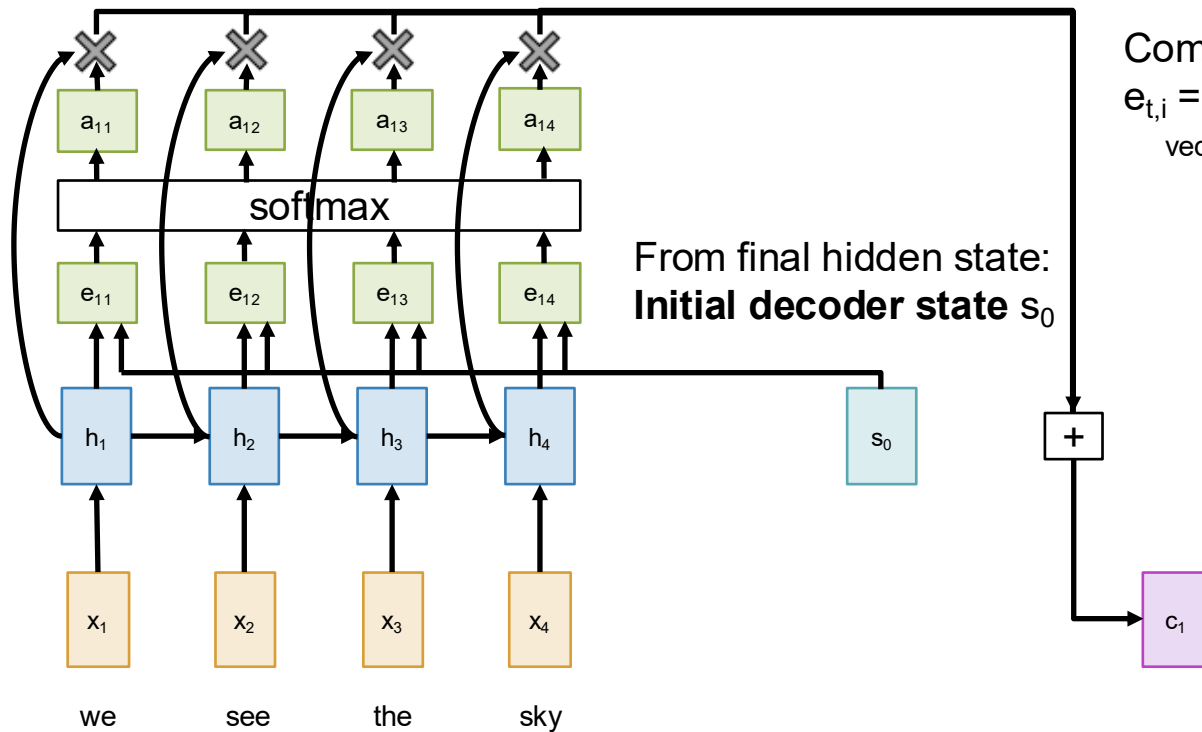
# Sequence to Sequence with RNNs and Attention



Compute (scalar) **alignment scores**  
 $e_{t,i} = f_{\text{att}}(s_{t-1}, h_i)$  ( $f_{\text{att}}$  is a Linear Layer)

Normalize alignment scores  
to get **attention weights**  
 $0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

# Sequence to Sequence with RNNs and Attention

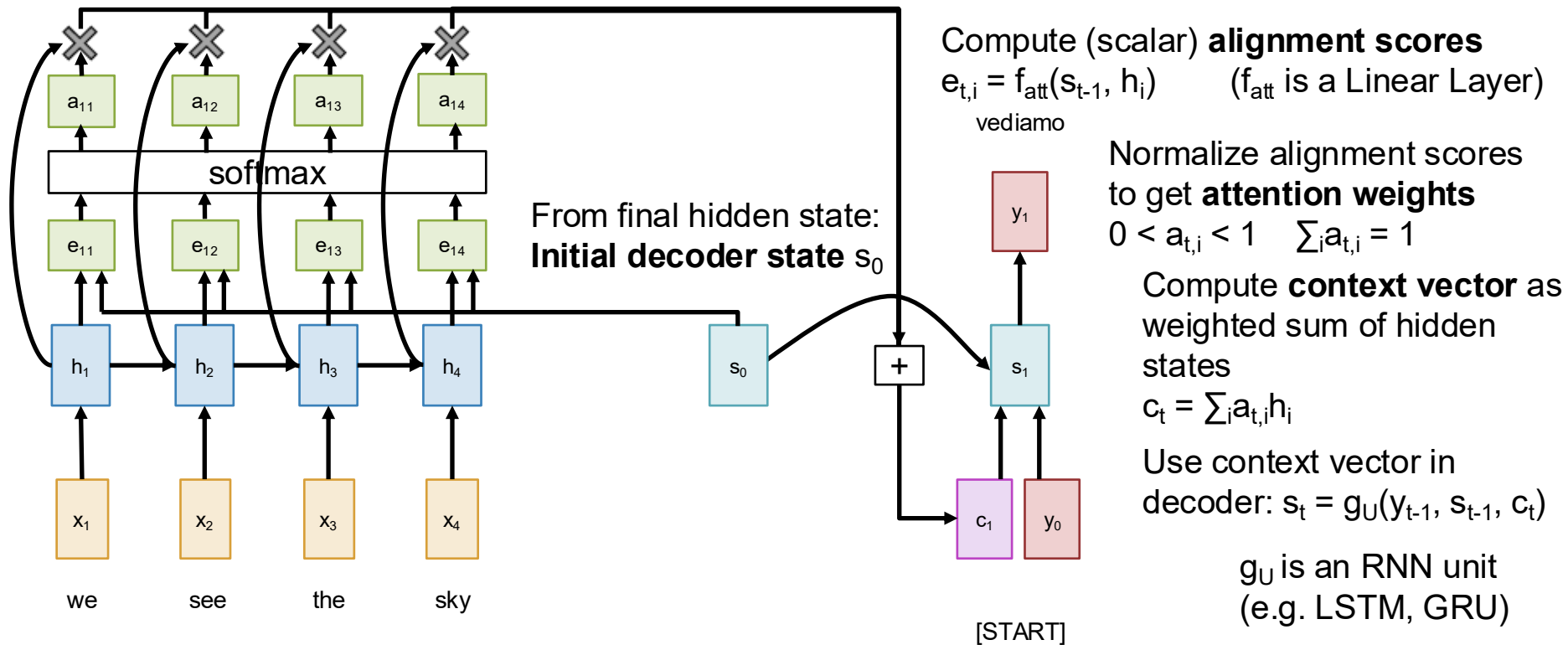


Compute (scalar) **alignment scores**  
 $e_{t,i} = f_{\text{att}}(s_{t-1}, h_i)$  ( $f_{\text{att}}$  is a Linear Layer)  
vediamo

Normalize alignment scores  
to get **attention weights**  
 $0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

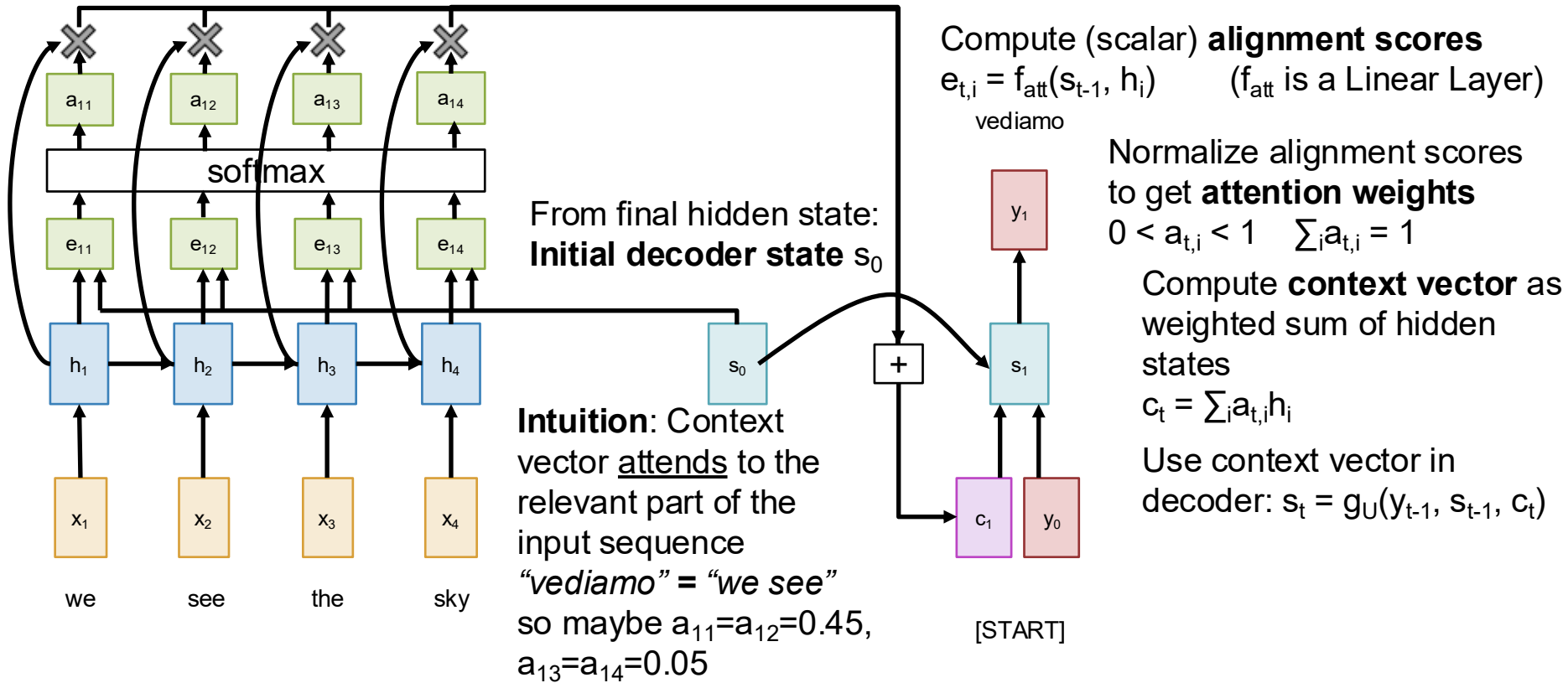
Compute context vector as  
**weighted sum of hidden  
states**  
 $c_t = \sum_i a_{t,i} h_i$

# Sequence to Sequence with RNNs and Attention



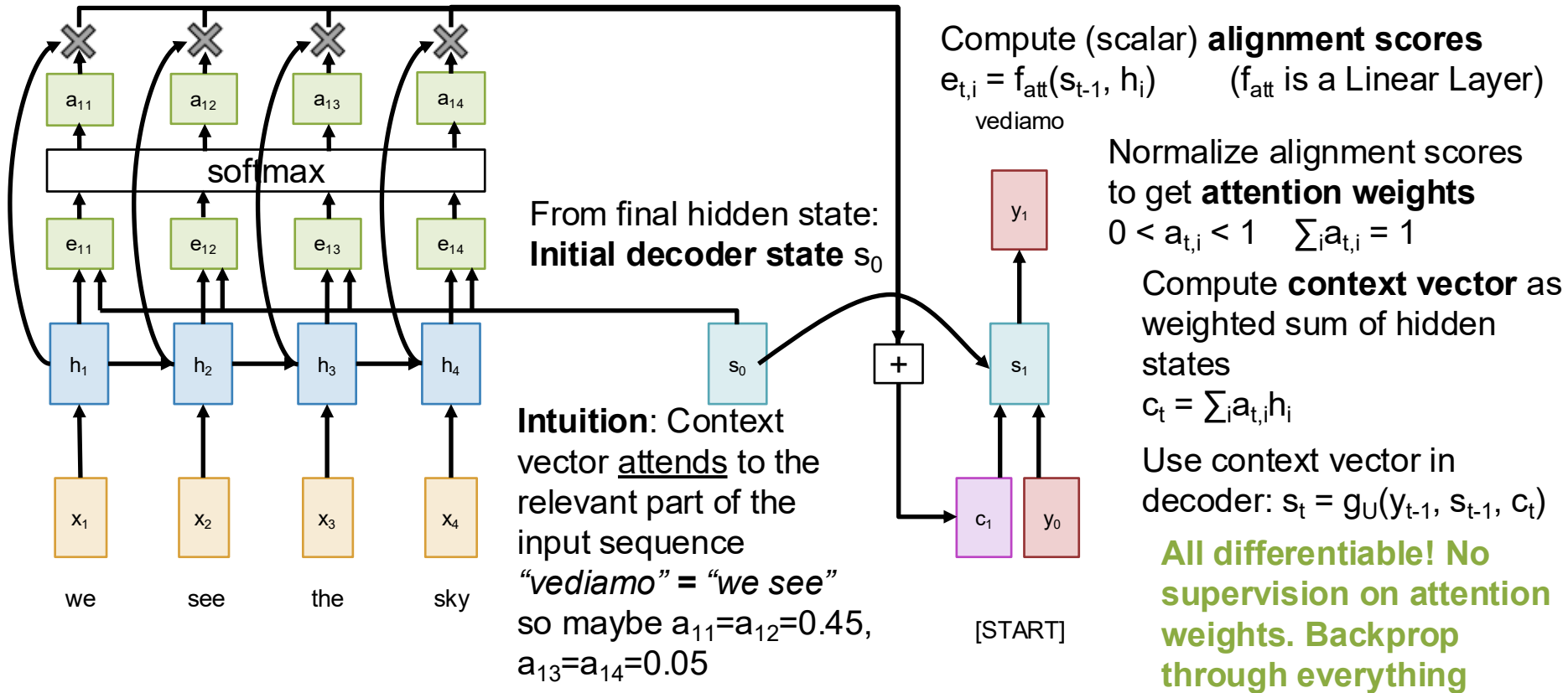
Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

# Sequence to Sequence with RNNs and Attention



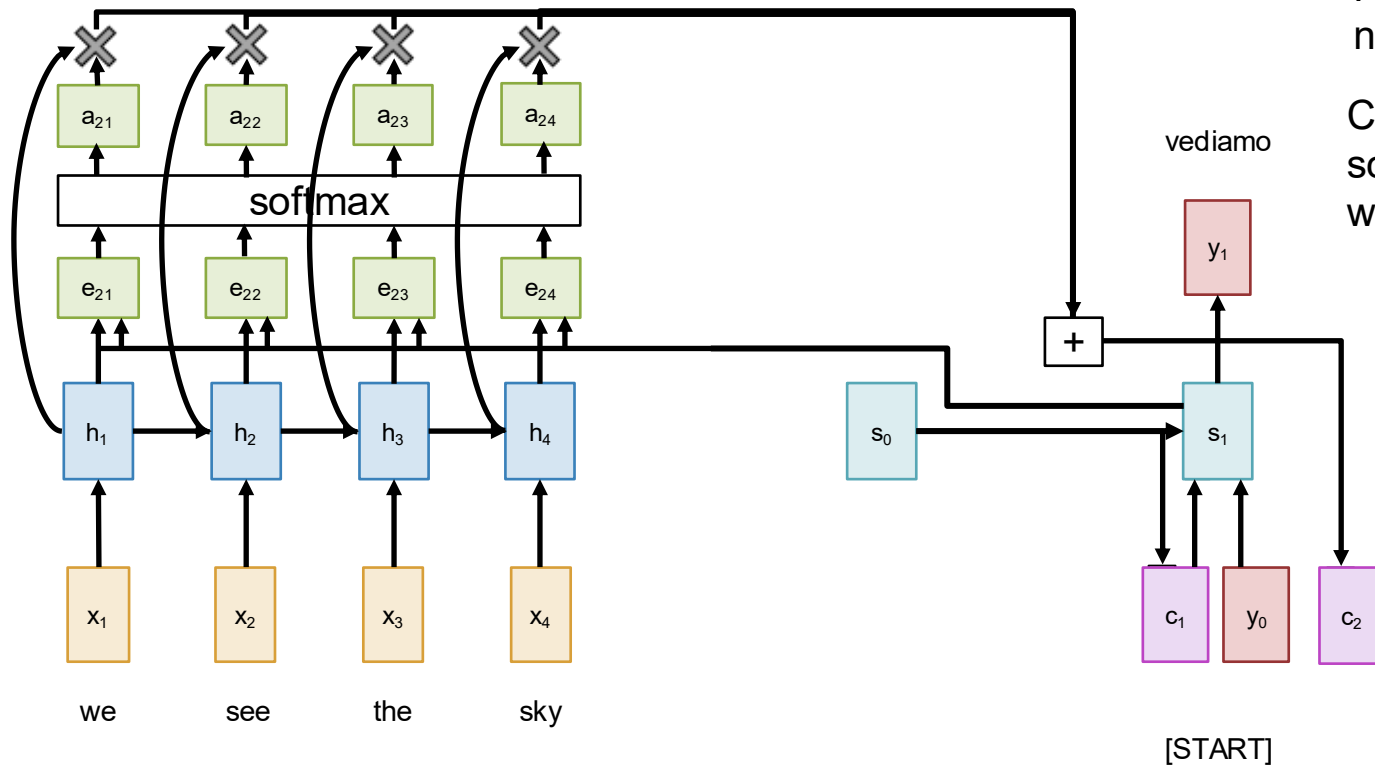
Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

# Sequence to Sequence with RNNs and Attention



Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

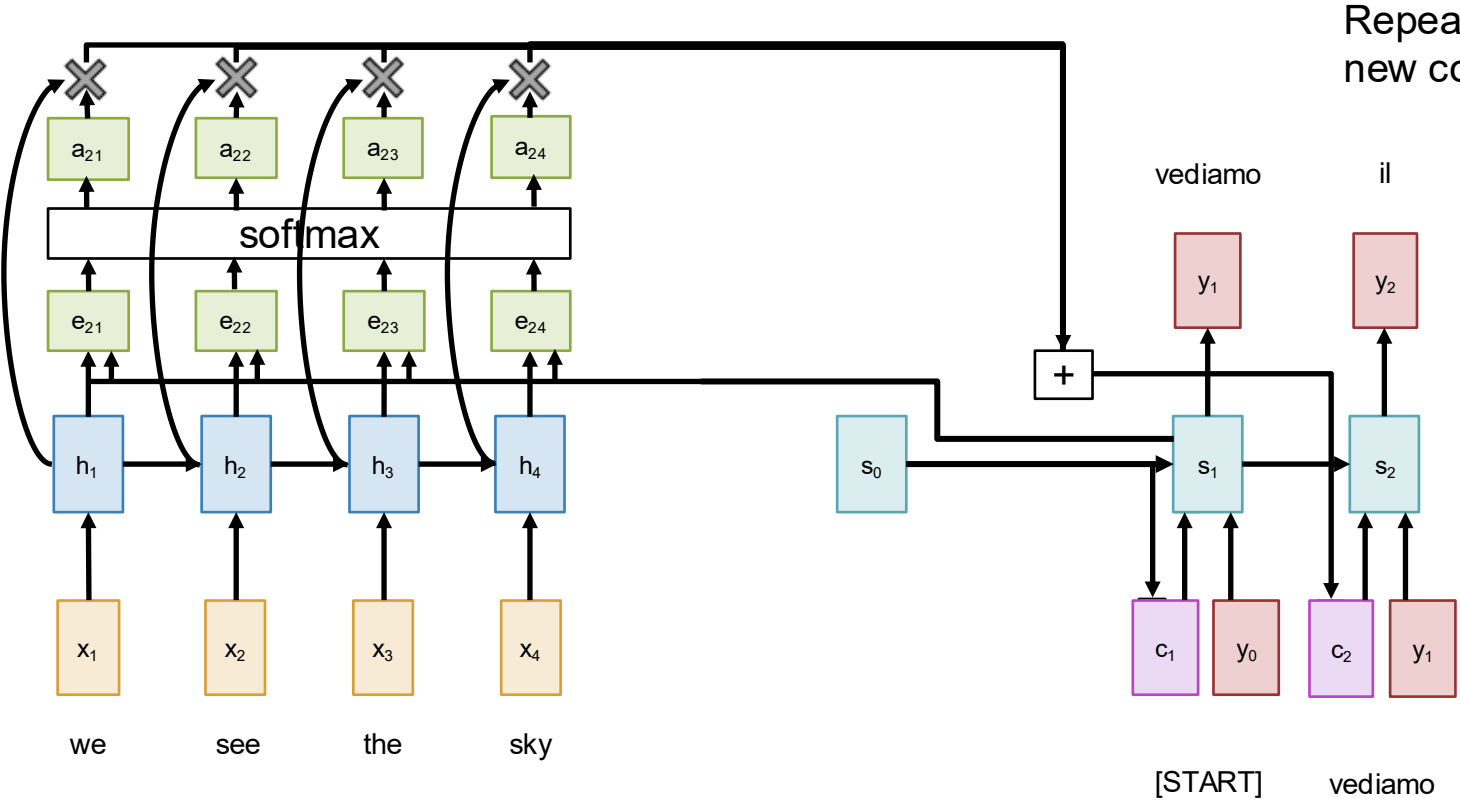
# Sequence to Sequence with RNNs and Attention



Repeat: Use  $s_1$  to compute new context vector  $c_2$

Compute new alignment scores  $e_{2,i}$  and attention weights  $a_{2,i}$

# Sequence to Sequence with RNNs and Attention

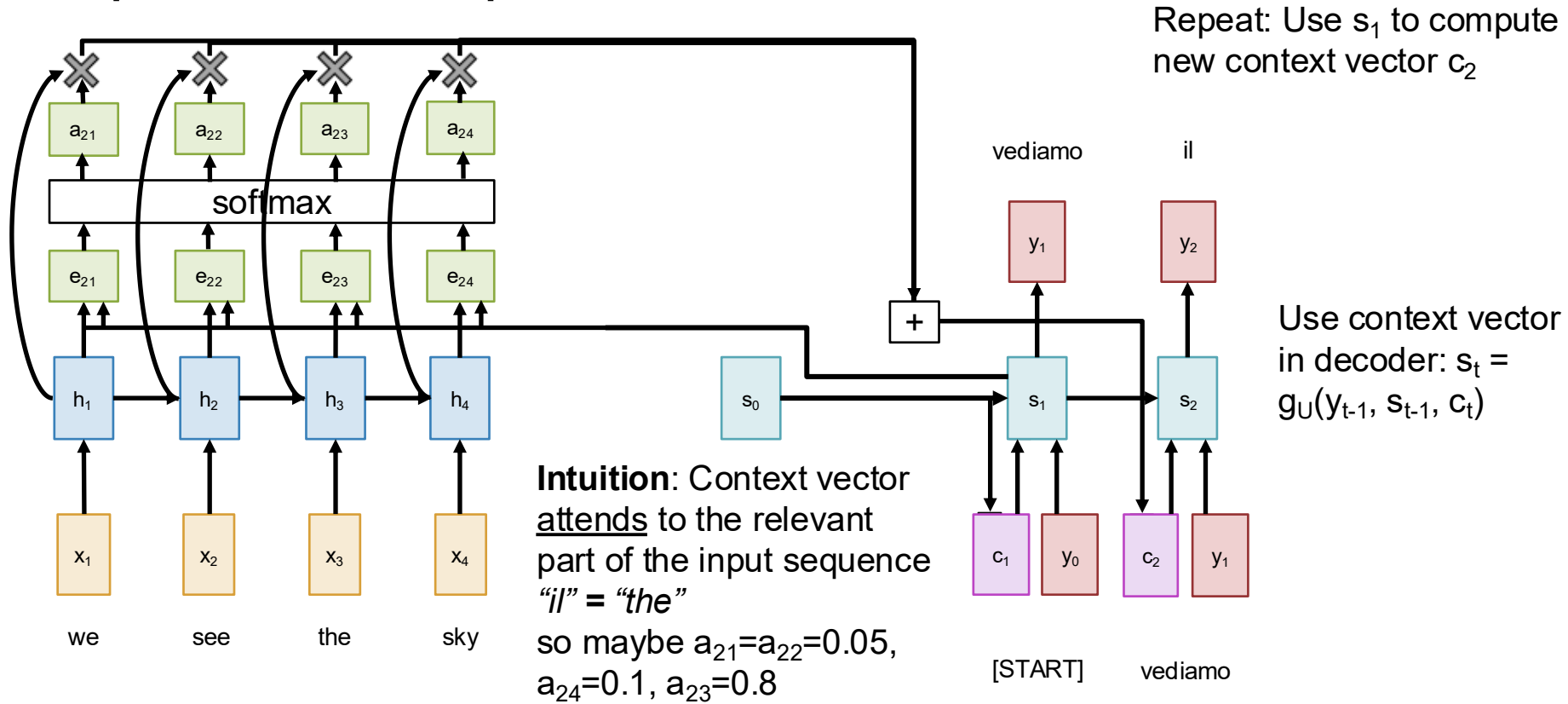


Repeat: Use  $s_1$  to compute new context vector  $c_2$

Use context vector in decoder:  $s_t = g_U(y_{t-1}, s_{t-1}, c_t)$

Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

# Sequence to Sequence with RNNs and Attention

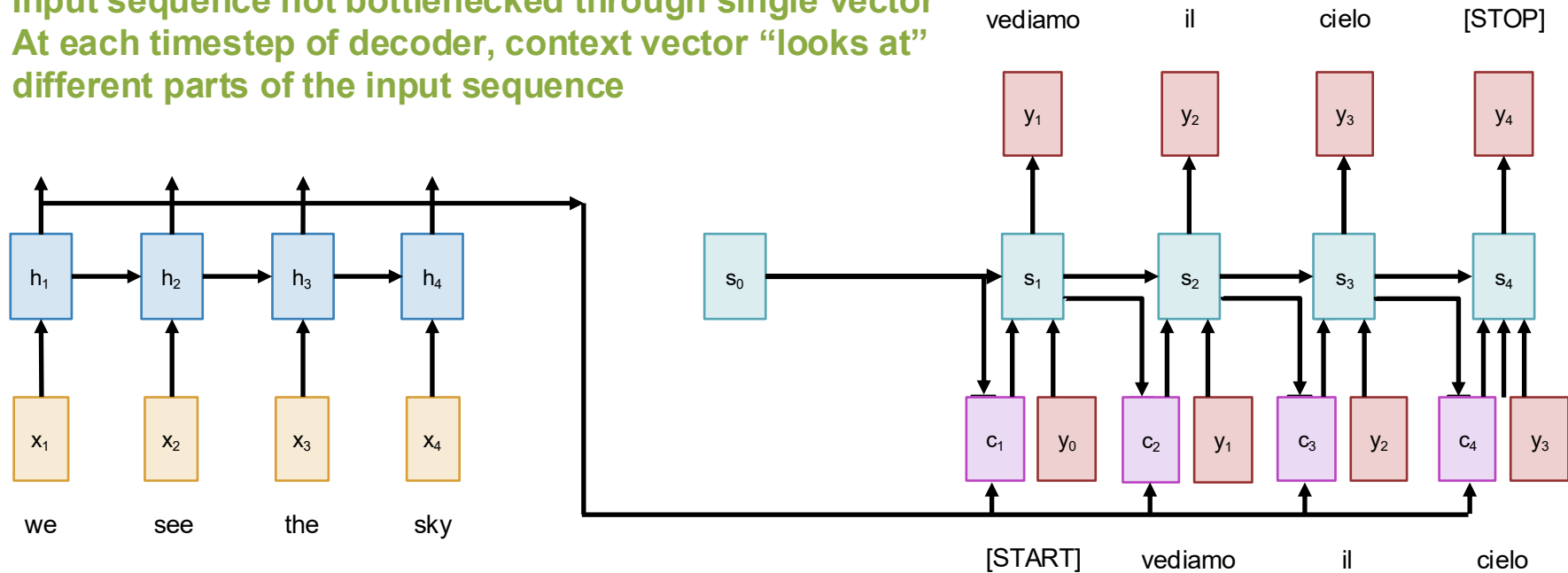


Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

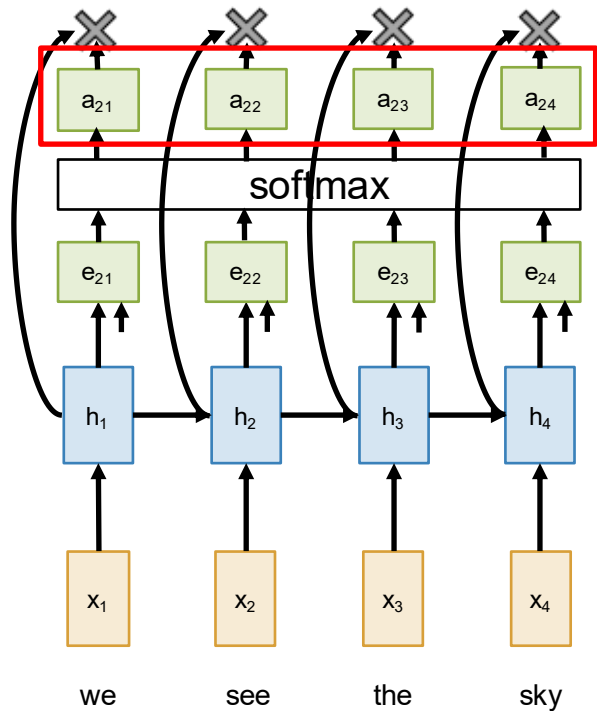
# Sequence to Sequence with RNNs and Attention

Use a different context vector in each timestep of decoder

- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector “looks at” different parts of the input sequence

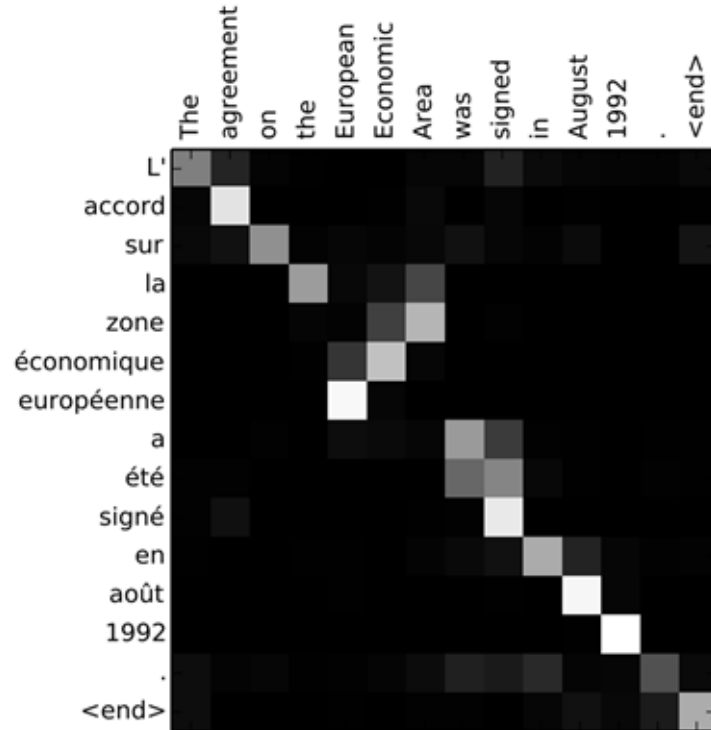


# Sequence to Sequence with RNNs and Attention



**Example:** English to French translation

Visualize attention weights  $a_{t,j}$



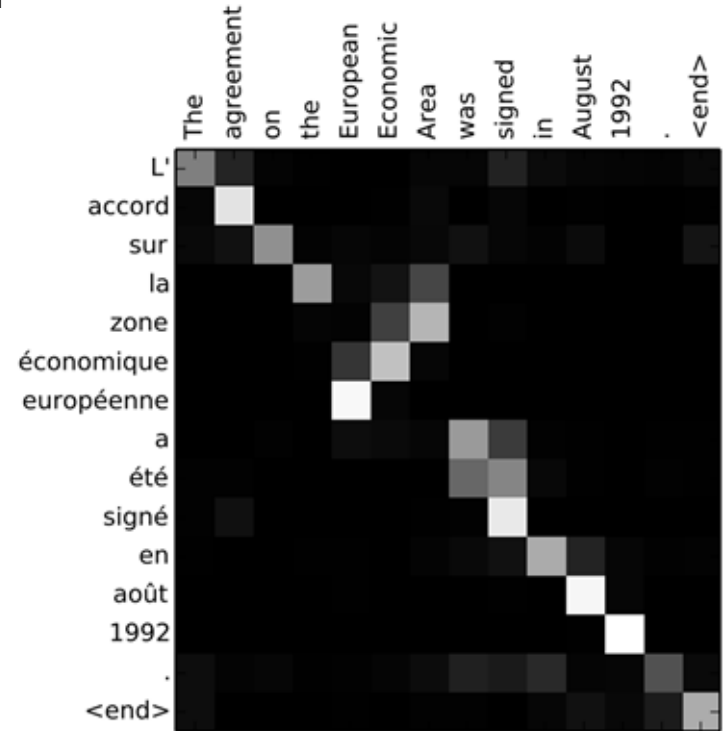
# Sequence to Sequence with RNNs and Attention

**Example:** English to French translation

Visualize attention weights  $a_{t,i}$

**Input:** “The agreement on the European Economic Area was signed in August 1992.”

**Output:** “L’accord sur la zone économique européenne a été signé en août 1992.”



# Sequence to Sequence with RNNs and Attention

**Example:** English to French translation

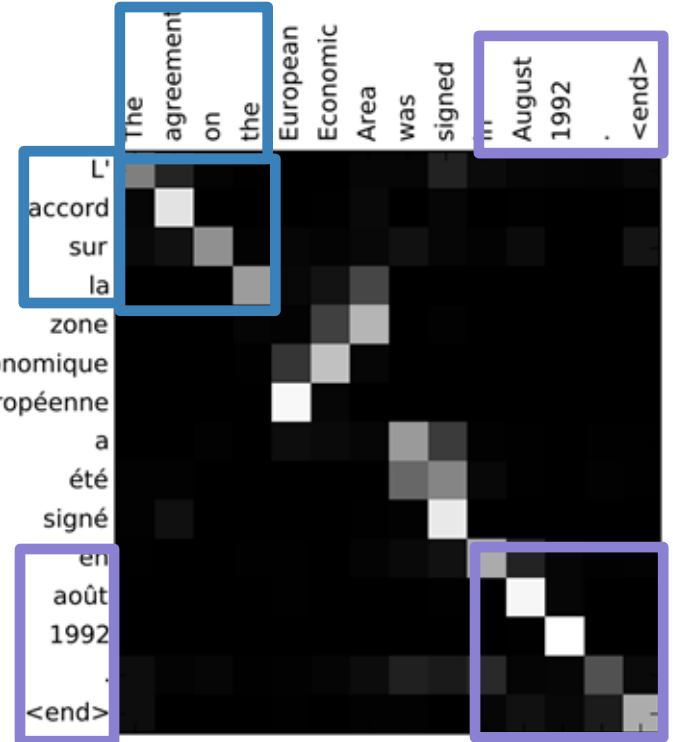
**Input:** “**The agreement on the** European Economic Area was signed in **August 1992.**”

**Output:** “**L’accord sur la** zone économique européenne a été signé **en août 1992.**”

Visualize attention weights  $a_{t,i}$

Diagonal attention means words correspond in order

Diagonal attention means words correspond in order



# Sequence to Sequence with RNNs and Attention

**Example:** English to French translation

**Input:** “The agreement on the European Economic Area was signed in August 1992.”

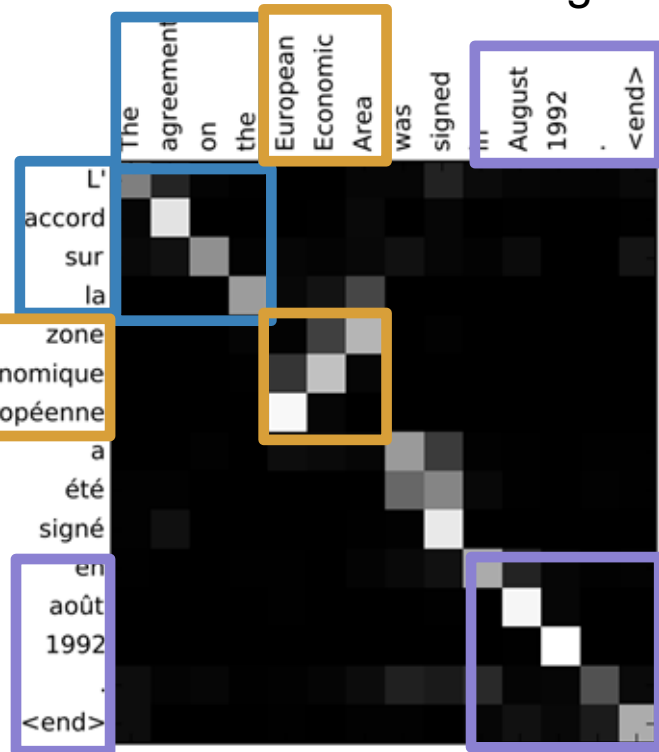
**Output:** “L'accord sur la zone économique européenne a été signé en août 1992.”

Visualize attention weights  $a_{t,i}$

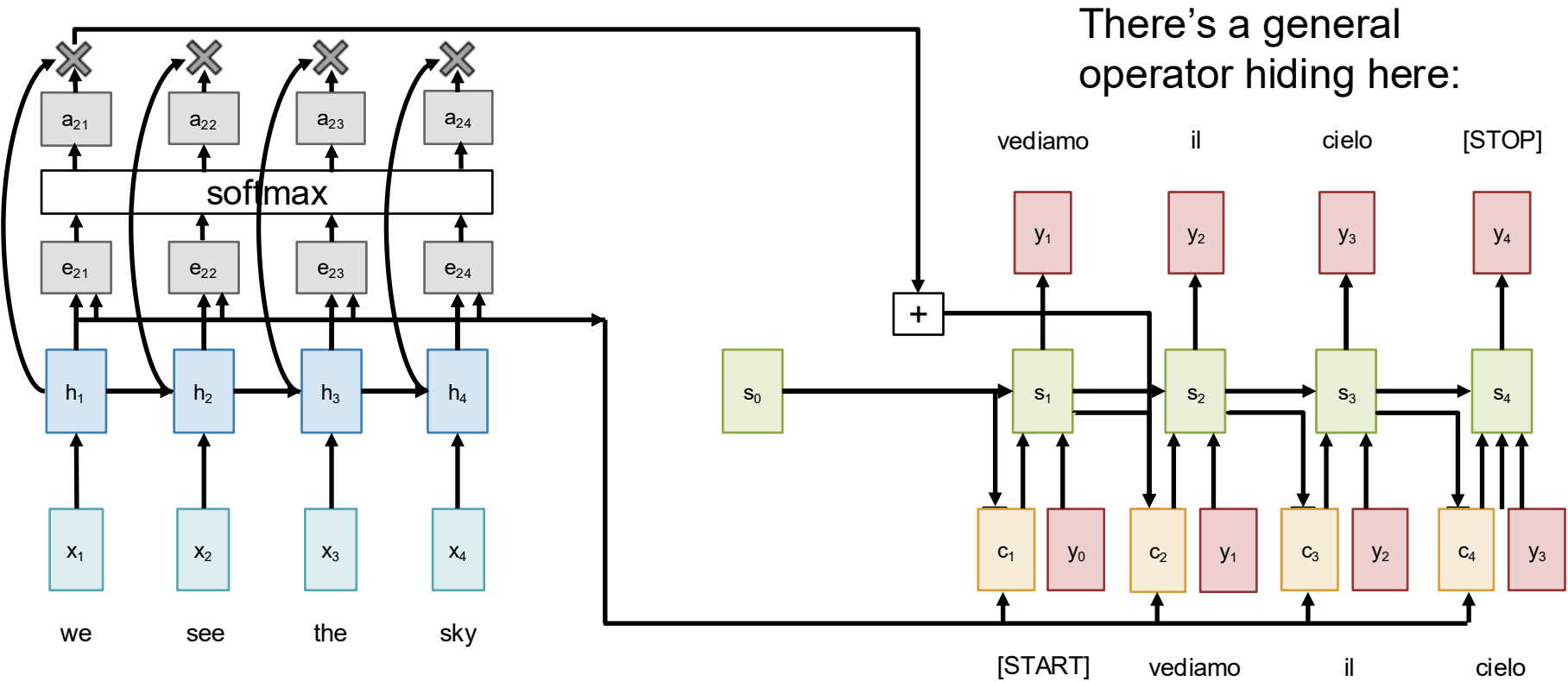
Diagonal attention means words correspond in order

Attention figures out other word orders

Diagonal attention means words correspond in order



# Sequence to Sequence with RNNs and Attention



# Sequence to Sequence with RNNs and Attention

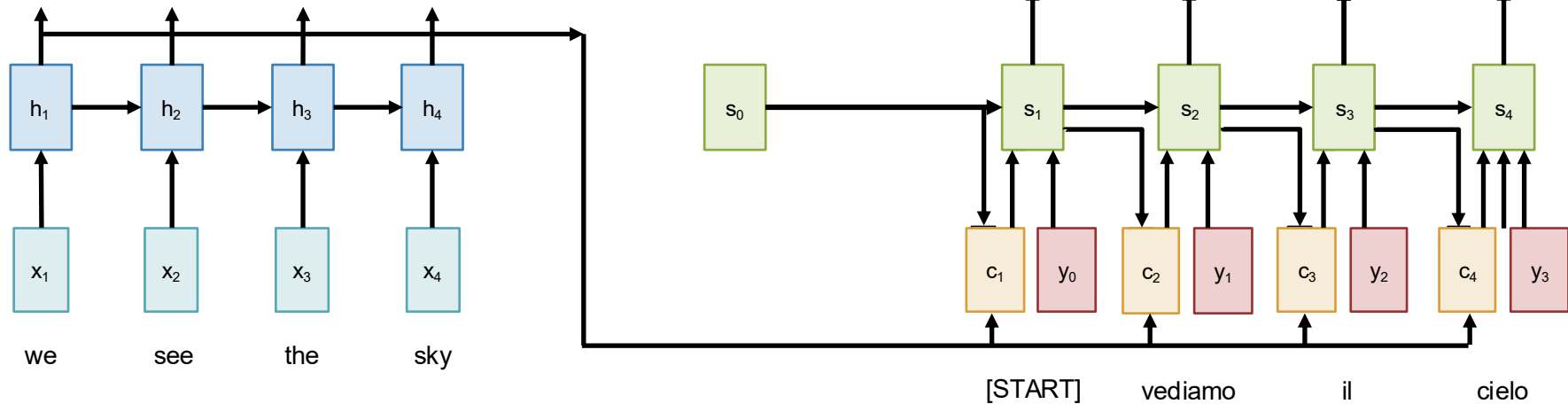
Query vectors (decoder RNN states) and data vectors (encoder RNN states)

get transformed to

output vectors (Context states).

Each query attends to all data vectors and gives one output vector

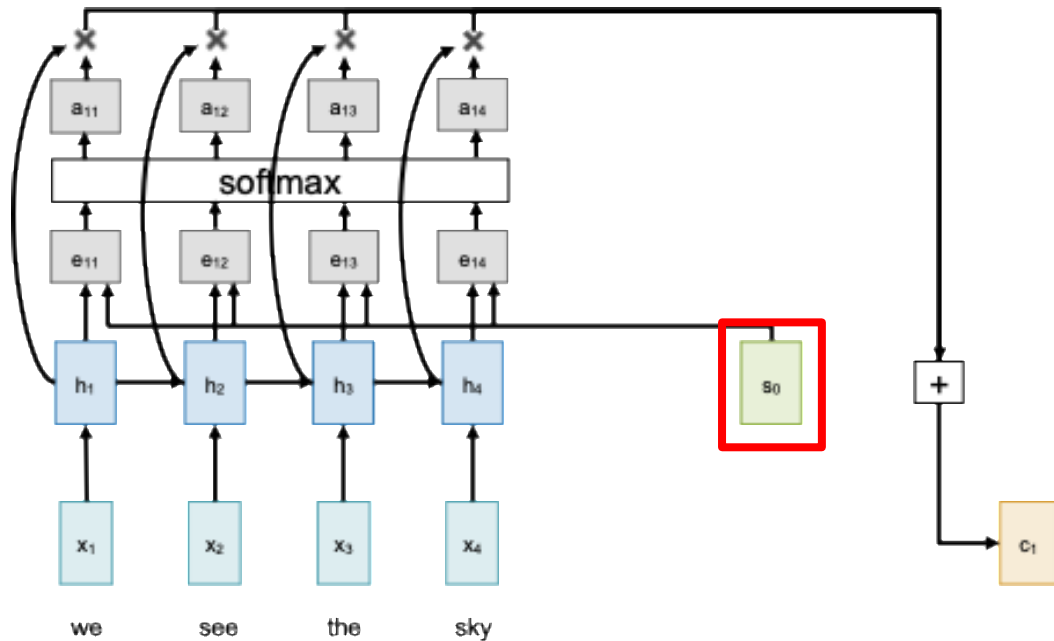
There's a general operator hiding here:



# Attention Layer

Inputs:

Query vector:  $\mathbf{q}$  [ $D_Q$ ]

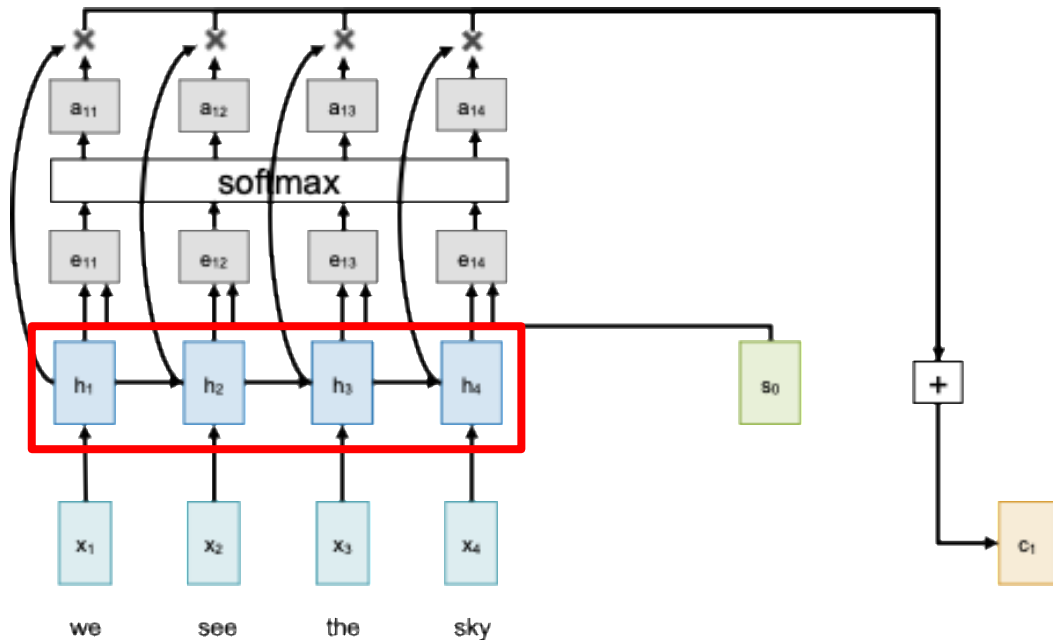


# Attention Layer

Inputs:

Query vector:  $\mathbf{q}$  [ $D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]



# Attention Layer

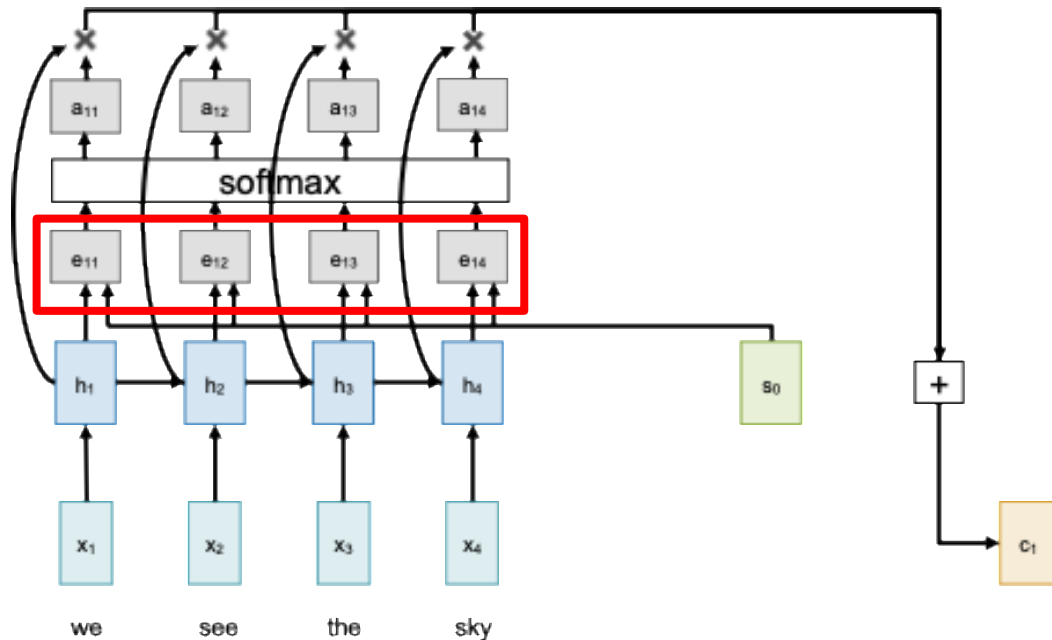
## Inputs:

Query vector:  $\mathbf{q}$  [ $D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

## Computation:

Similarities:  $e$  [ $N_X$ ]  $e_i = f_{\text{att}}(\mathbf{q}, \mathbf{X}_i)$

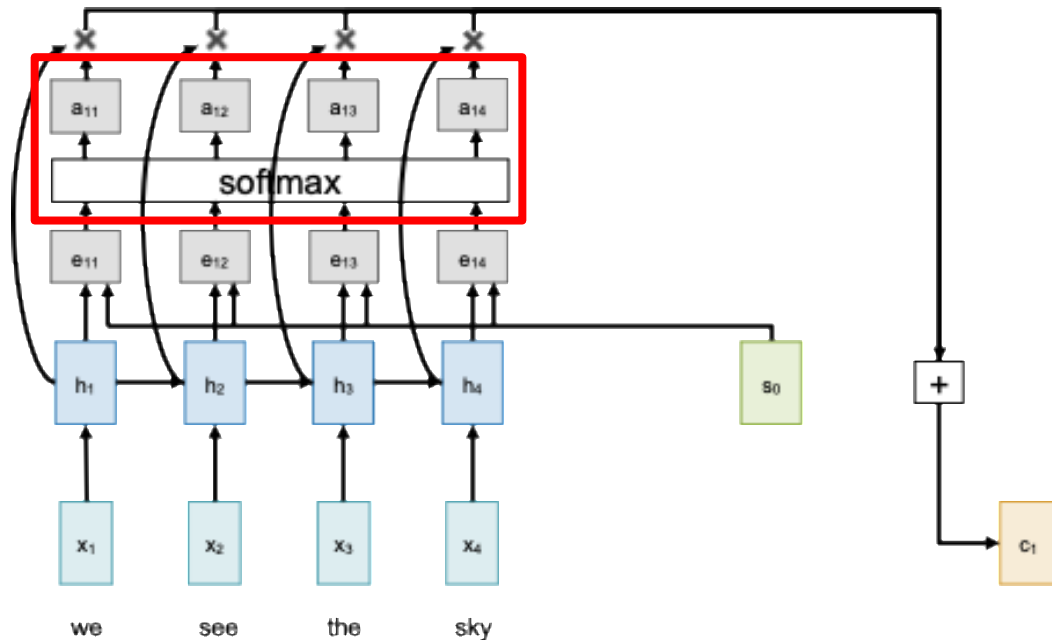


# Attention Layer

## Inputs:

Query vector:  $\mathbf{q}$  [ $D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]



## Computation:

Similarities:  $e$  [ $N_X$ ]  $e_i = f_{\text{att}}(\mathbf{q}, \mathbf{X}_i)$

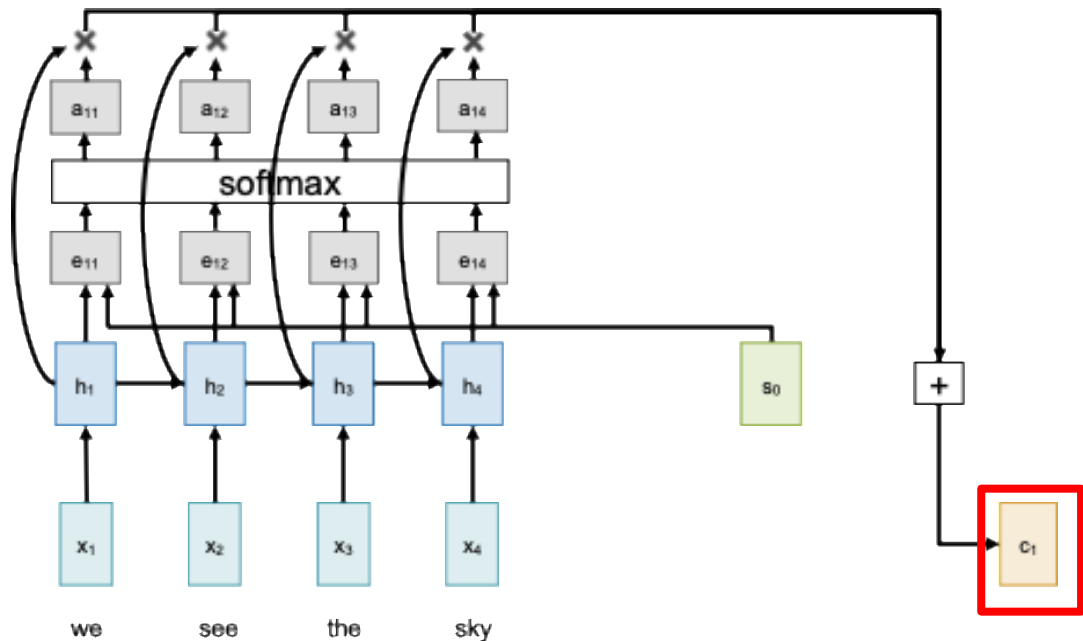
Attention weights:  $a = \text{softmax}(e)$  [ $N_X$ ]

# Attention Layer

## Inputs:

Query vector:  $\mathbf{q}$  [ $D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]



## Computation:

Similarities:  $e$  [ $N_X$ ]  $e_i = f_{\text{att}}(\mathbf{q}, \mathbf{X}_i)$

Attention weights:  $a = \text{softmax}(e)$  [ $N_X$ ]

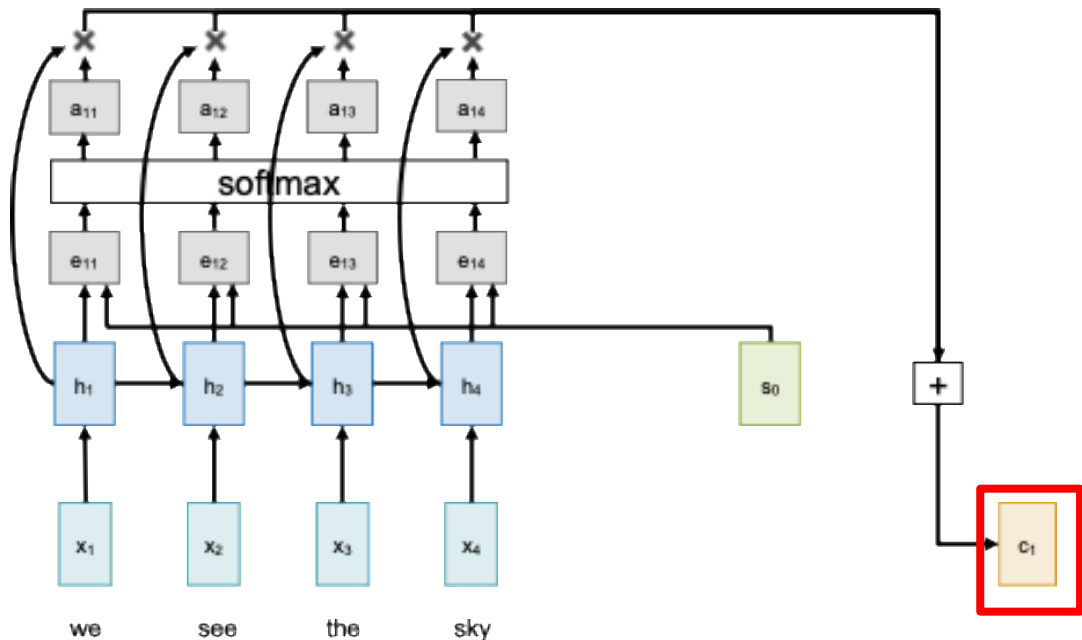
Output vector:  $\mathbf{y} = \sum_i a_i \mathbf{X}_i$  [ $D_X$ ]

# Attention Layer

## Inputs:

Query vector:  $\mathbf{q}$  [ $D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]



## Computation:

Similarities:  $e$  [ $N_X$ ]  $e_i = f_{\text{att}}(\mathbf{q}, \mathbf{X}_i)$

Attention weights:  $a = \text{softmax}(e)$  [ $N_X$ ]

Output vector:  $\mathbf{y} = \sum_i a_i \mathbf{X}_i$  [ $D_X$ ]

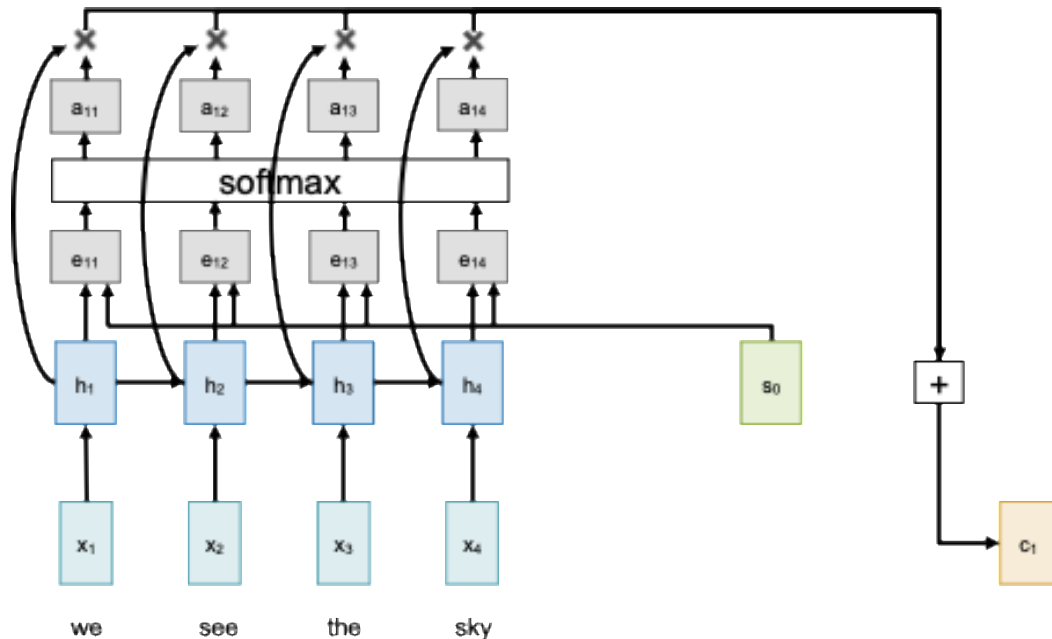
Let's generalize this!

# Attention Layer

## Inputs:

Query vector:  $\mathbf{q}$  [ $D_X$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]



## Computation:

Similarities:  $e$  [ $N_X$ ]  $e_i = \mathbf{q} \cdot \mathbf{X}_i$

Attention weights:  $\mathbf{a} = \text{softmax}(e)$  [ $N_X$ ]

Output vector:  $\mathbf{y} = \sum_i a_i \mathbf{X}_i$  [ $D_X$ ]

## Changes

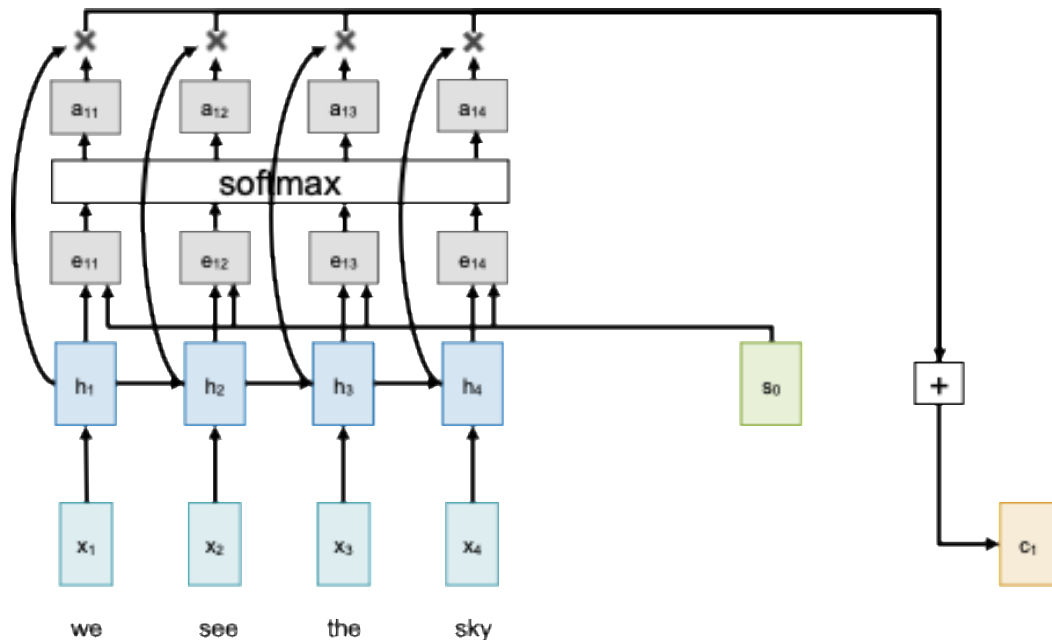
- Use dot product for similarity
- Requires dimensions to match

# Attention Layer

## Inputs:

Query vector:  $\mathbf{q}$  [ $D_X$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]



## Computation:

Similarities:  $e$  [ $N_X$ ]  $e_i = \mathbf{q} \cdot \mathbf{X}_i / \sqrt{D_X}$

Attention weights:  $\mathbf{a} = \text{softmax}(e)$  [ $N_X$ ]

Output vector:  $\mathbf{y} = \sum_i a_i \mathbf{X}_i$  [ $D_X$ ]

## Changes

- Use **scaled** dot product for similarity

# Attention Layer

## Inputs:

Query vector:  $\mathbf{q}$  [ $D_X$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Large similarities will cause softmax to saturate and give vanishing gradients

Recall  $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos(\text{angle})$

Suppose that  $\mathbf{a}$  and  $\mathbf{b}$  are constant vectors of dimension  $D$

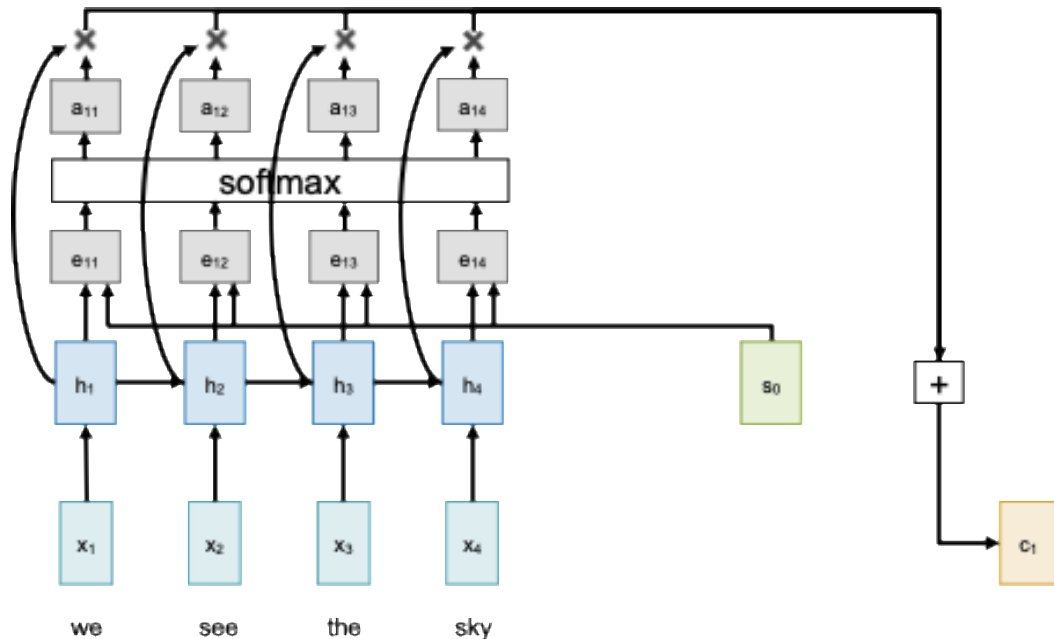
Then  $|\mathbf{a}| = (\sum_i a_i^2)^{1/2} = a \sqrt{D}$

## Computation:

Similarities:  $\mathbf{e}$  [ $N_X$ ]  $e_i = \mathbf{q} \cdot \mathbf{X}_i / \sqrt{D_X}$

Attention weights:  $\mathbf{a} = \text{softmax}(\mathbf{e})$  [ $N_X$ ]

Output vector:  $\mathbf{y} = \sum_i a_i \mathbf{X}_i$  [ $D_X$ ]



## Changes

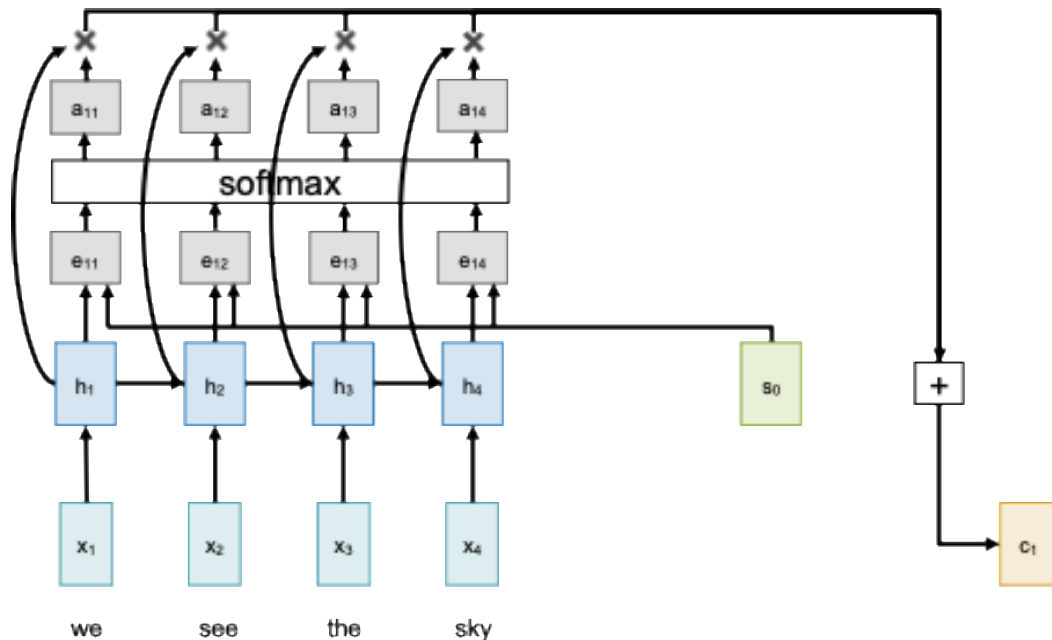
- Use **scaled** dot product for similarity

# Attention Layer

Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_X$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]



Computation:

Similarities:  $E = \mathbf{QX}^T / \sqrt{D_X}$  [ $N_Q \times N_X$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{X}_j / \sqrt{D_X}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AX}$  [ $N_Q \times D_X$ ]

$$Y_i = \sum_j A_{ij} X_j$$

**Changes**

- Use scaled dot product for similarity
- Multiple **query** vectors

# Attention Layer

## Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_X \times D_Q$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_X \times D_V$ ]

## Computation:

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N_X \times D_Q$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $D_X \times D_V$ ]

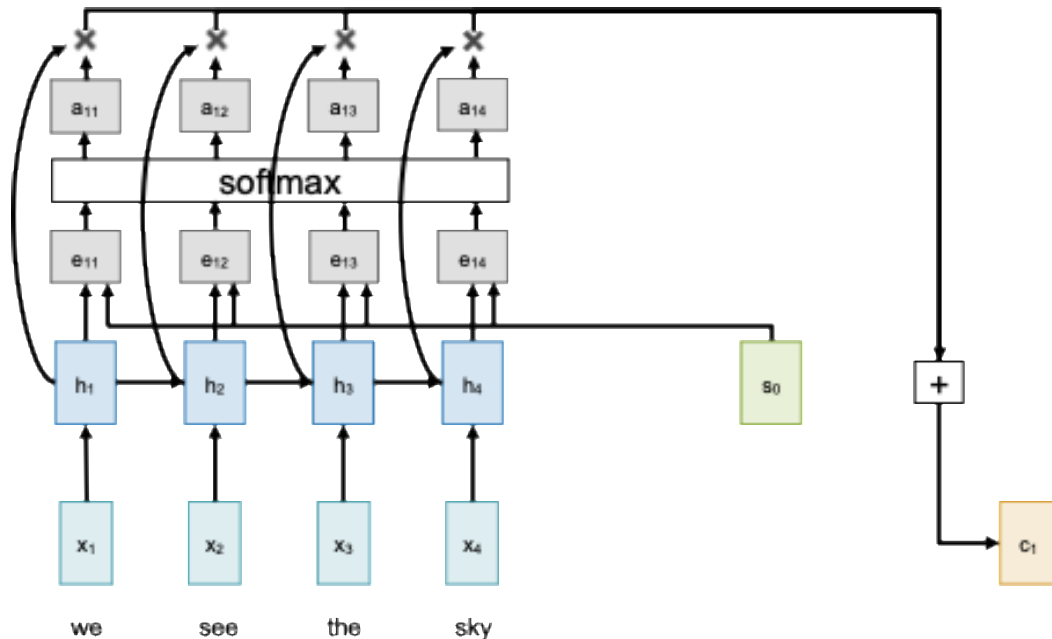
Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N_Q \times N_X$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N_Q \times D_X$ ]

$$Y_i = \sum_j A_{ij} V_j$$



## Changes

- Use scaled dot product for similarity
- Multiple **query** vectors
- Separate **key** and **query**
- No longer requires  $D_Q = D_X$

# Attention Layer

## Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_X \times D_Q$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_X \times D_V$ ]

## Computation:

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N_X \times D_Q$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $D_X \times D_V$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N_Q \times N_X$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N_Q \times D_X$ ]

$$Y_i = \sum_j A_{ij} V_j$$

$X_1$

$X_2$

$X_3$

$Q_1$

$Q_2$

$Q_3$

$Q_4$

# Attention Layer

## Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_X \times D_Q$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_X \times D_V$ ]

## Computation:

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N_X \times D_Q$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $D_X \times D_V$ ]

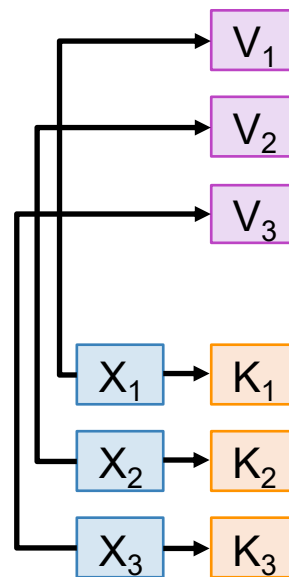
Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N_Q \times N_X$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N_Q \times D_V$ ]

$$Y_i = \sum_j A_{ij} V_j$$



$Q_1$

$Q_2$

$Q_3$

$Q_4$

# Attention Layer

## Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_X \times D_Q$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_X \times D_V$ ]

## Computation:

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N_X \times D_Q$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $D_X \times D_V$ ]

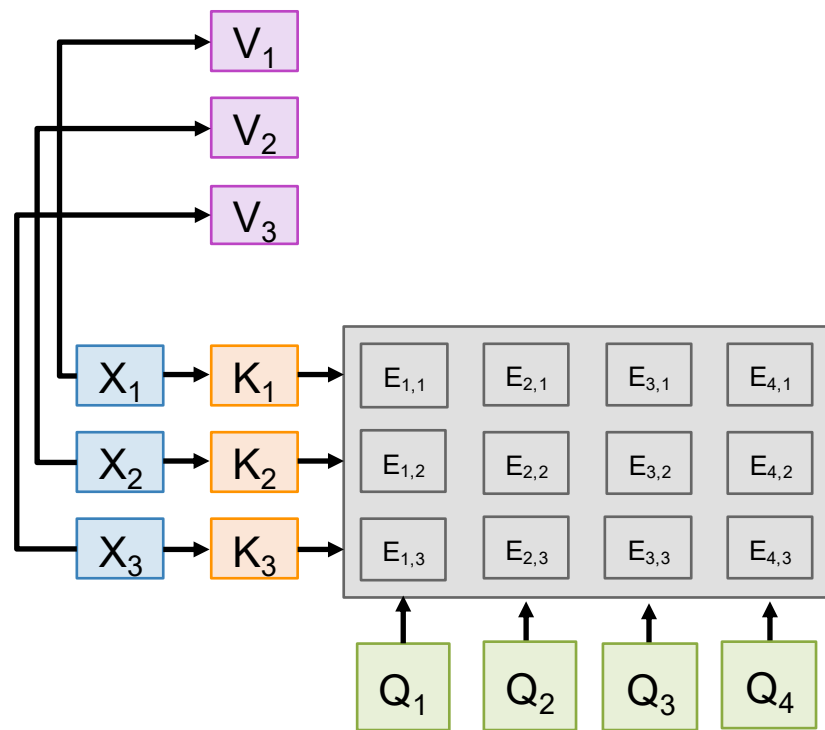
Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N_Q \times N_X$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N_Q \times D_V$ ]

$$Y_i = \sum_j A_{ij} V_j$$



# Attention Layer

## Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_X \times D_Q$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_X \times D_V$ ]

## Computation:

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N_X \times D_Q$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $D_X \times D_V$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N_Q \times N_X$ ]

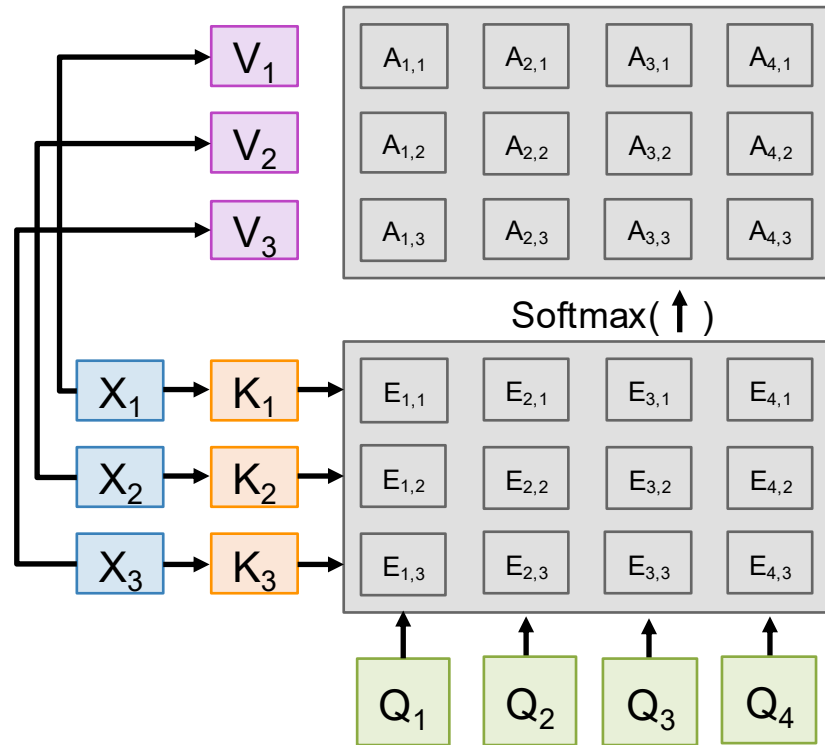
$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N_Q \times D_V$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Softmax normalizes each column: each **query** predicts a distribution over the **keys**



# Attention Layer

## Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_X \times D_Q$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_X \times D_V$ ]

## Computation:

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N_X \times D_Q$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $D_X \times D_V$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N_Q \times N_X$ ]

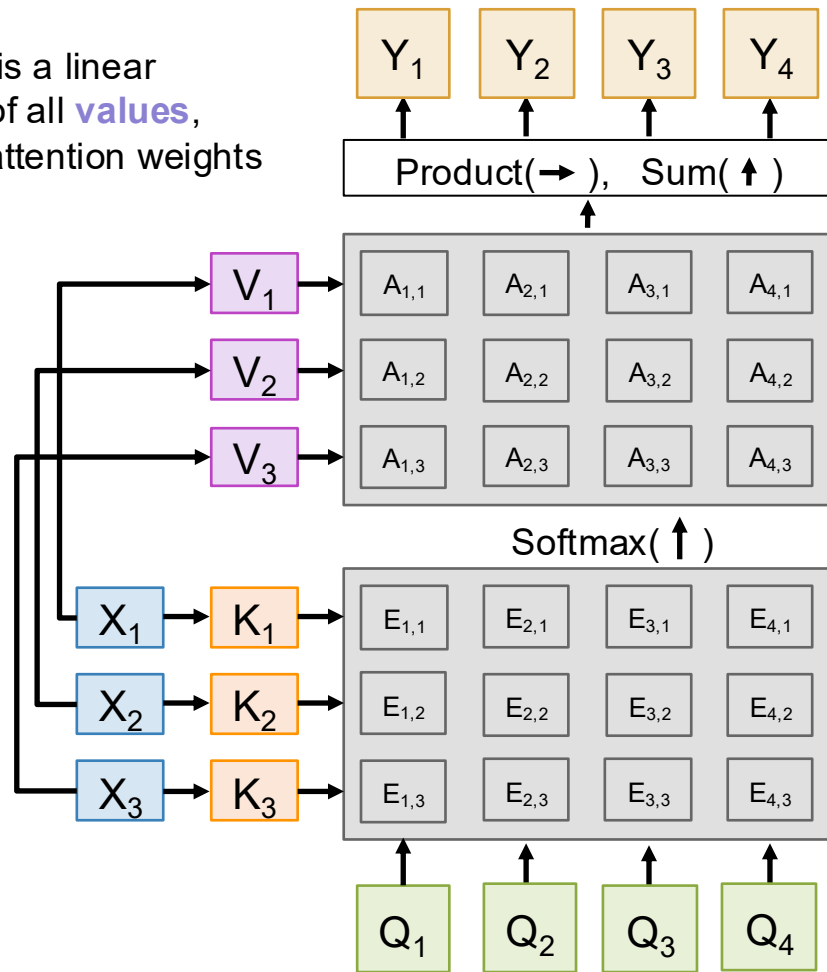
$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N_Q \times D_V$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Each **output** is a linear combination of all **values**, weighted by attention weights



# Cross-Attention Layer

## Inputs:

Query vector:  $\mathbf{Q}$  [ $N_Q \times D_Q$ ]

Data vectors:  $\mathbf{X}$  [ $N_X \times D_X$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_X \times D_Q$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_X \times D_V$ ]

Each **query** produces one **output**, which is a mix of information in the **data** vectors

## Computation:

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N_X \times D_Q$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $D_X \times D_V$ ]

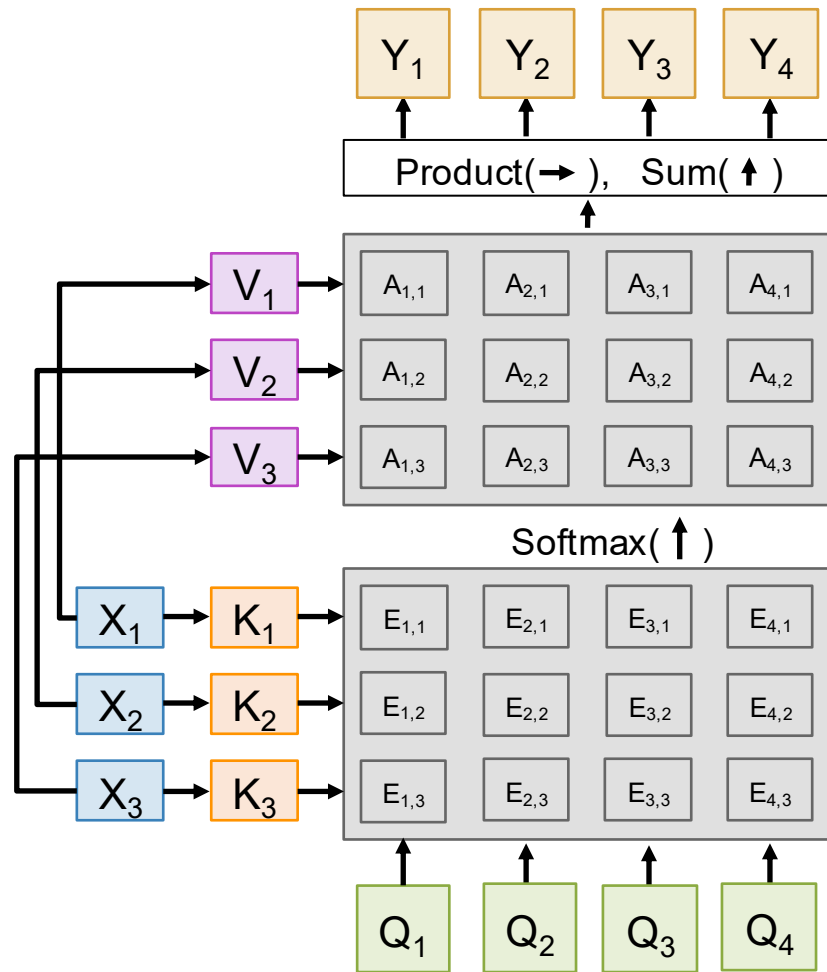
Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N_Q \times N_X$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N_Q \times N_X$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N_Q \times D_V$ ]

$$Y_i = \sum_j A_{ij} V_j$$



# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  [ $N \times D_{in}$ ]

**Key matrix:**  $W_K$  [ $D_{in} \times D_{out}$ ]

**Value matrix:**  $W_V$  [ $D_{in} \times D_{out}$ ]

**Query matrix:**  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

**Queries:**  $Q = XW_Q$  [ $N \times D_{out}$ ]

**Keys:**  $K = XW_K$  [ $N \times D_{out}$ ]

**Values:**  $V = XW_V$  [ $N \times D_{out}$ ]

**Similarities:**  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

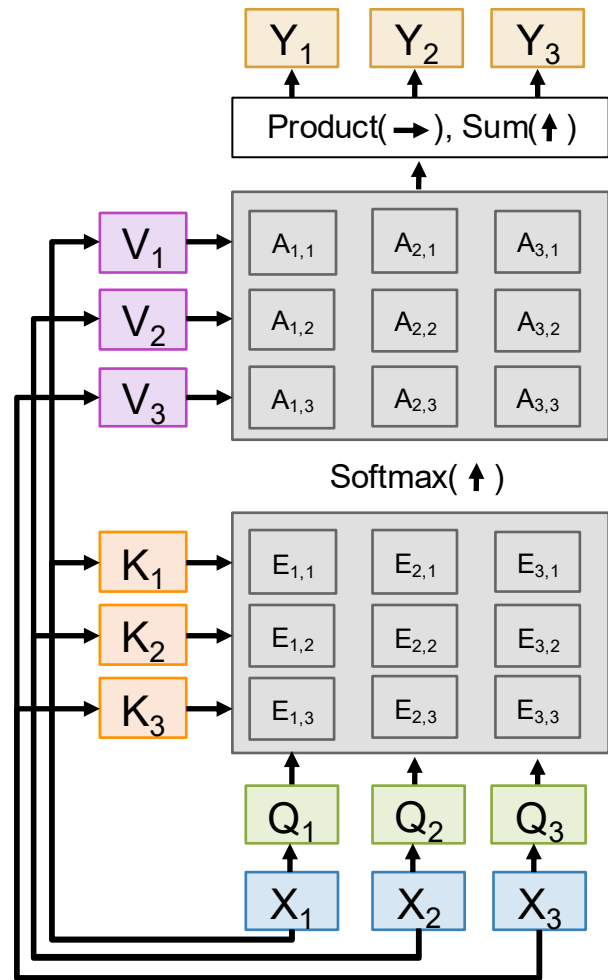
**Output vector:**  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Each **input** produces one **output**, which is a mix of information from all **inputs**

Shapes get a little simpler:

- $N$  input vectors, each  $D_{in}$
- Almost always  $D_Q = D_V = D_{out}$



# Self-Attention Layer

## Inputs:

**Input vectors:**  $\mathbf{X}$  [ $N \times D_{in}$ ]

**Key matrix:**  $\mathbf{W}_K$  [ $D_{in} \times D_{out}$ ]

**Value matrix:**  $\mathbf{W}_V$  [ $D_{in} \times D_{out}$ ]

**Query matrix:**  $\mathbf{W}_Q$  [ $D_{in} \times D_{out}$ ]

Each **input** produces one **output**, which is a mix of information from all **inputs**

## Computation:

**Queries:**  $\mathbf{Q} = \mathbf{XW}_Q$  [ $N \times D_{out}$ ]

**Keys:**  $\mathbf{K} = \mathbf{XW}_K$  [ $N \times D_{out}$ ]

**Values:**  $\mathbf{V} = \mathbf{XW}_V$  [ $N \times D_{out}$ ]

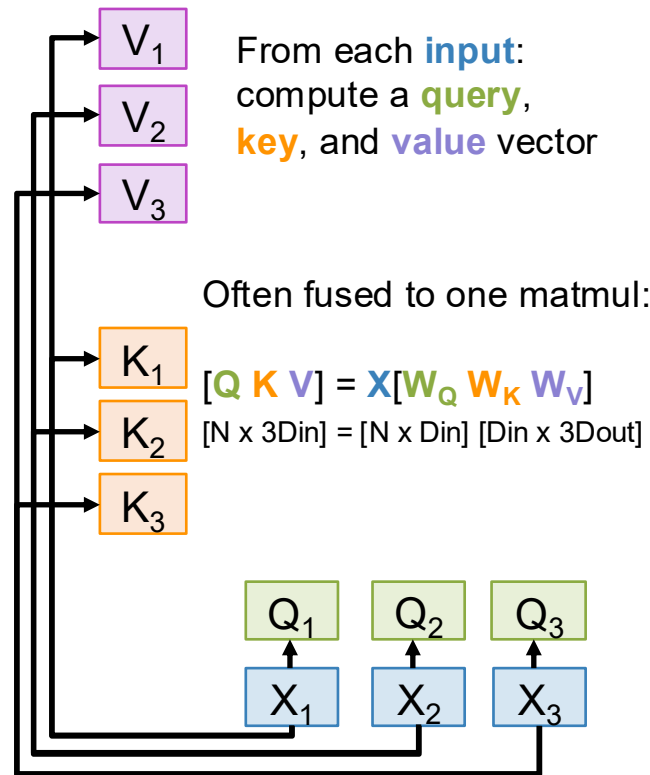
**Similarities:**  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

**Attention weights:**  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N \times N$ ]

**Output vector:**  $\mathbf{Y} = \mathbf{AV}$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$



# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  [ $N \times D_{in}$ ]

**Key matrix:**  $W_K$  [ $D_{in} \times D_{out}$ ]

**Value matrix:**  $W_V$  [ $D_{in} \times D_{out}$ ]

**Query matrix:**  $W_Q$  [ $D_{in} \times D_{out}$ ]

Each **input** produces one **output**, which is a mix of information from all **inputs**

## Computation:

**Queries:**  $Q = XW_Q$  [ $N \times D_{out}$ ]

**Keys:**  $K = XW_K$  [ $N \times D_{out}$ ]

**Values:**  $V = XW_V$  [ $N \times D_{out}$ ]

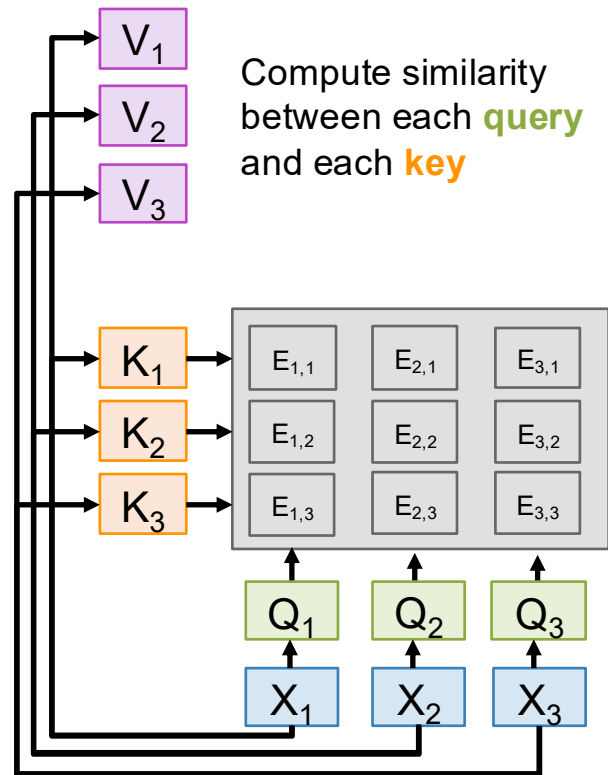
**Similarities:**  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

**Output vector:**  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  [ $N \times D_{in}$ ]

Key matrix:  $W_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $W_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $W_Q$  [ $D_{in} \times D_{out}$ ]

Each **input** produces one **output**, which is a mix of information from all **inputs**

## Computation:

Queries:  $Q = XW_Q$  [ $N \times D_{out}$ ]

Keys:  $K = XW_K$  [ $N \times D_{out}$ ]

Values:  $V = XW_V$  [ $N \times D_{out}$ ]

Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

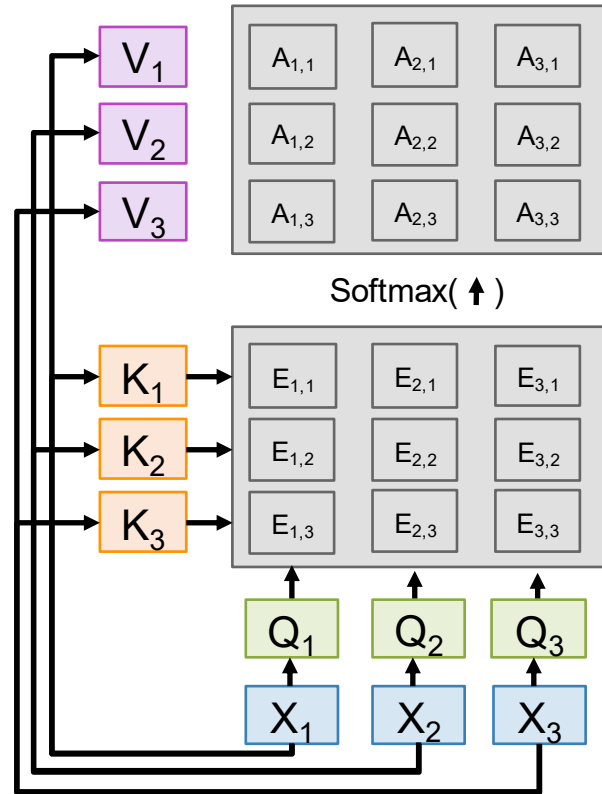
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Normalize over each column:  
each **query** computes a  
distribution over **keys**



# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  [ $N \times D_{in}$ ]

**Key matrix:**  $W_K$  [ $D_{in} \times D_{out}$ ]

**Value matrix:**  $W_V$  [ $D_{in} \times D_{out}$ ]

**Query matrix:**  $W_Q$  [ $D_{in} \times D_{out}$ ]

Each **input** produces one **output**, which is a mix of information from all **inputs**

## Computation:

**Queries:**  $Q = XW_Q$  [ $N \times D_{out}$ ]

**Keys:**  $K = XW_K$  [ $N \times D_{out}$ ]

**Values:**  $V = XW_V$  [ $N \times D_{out}$ ]

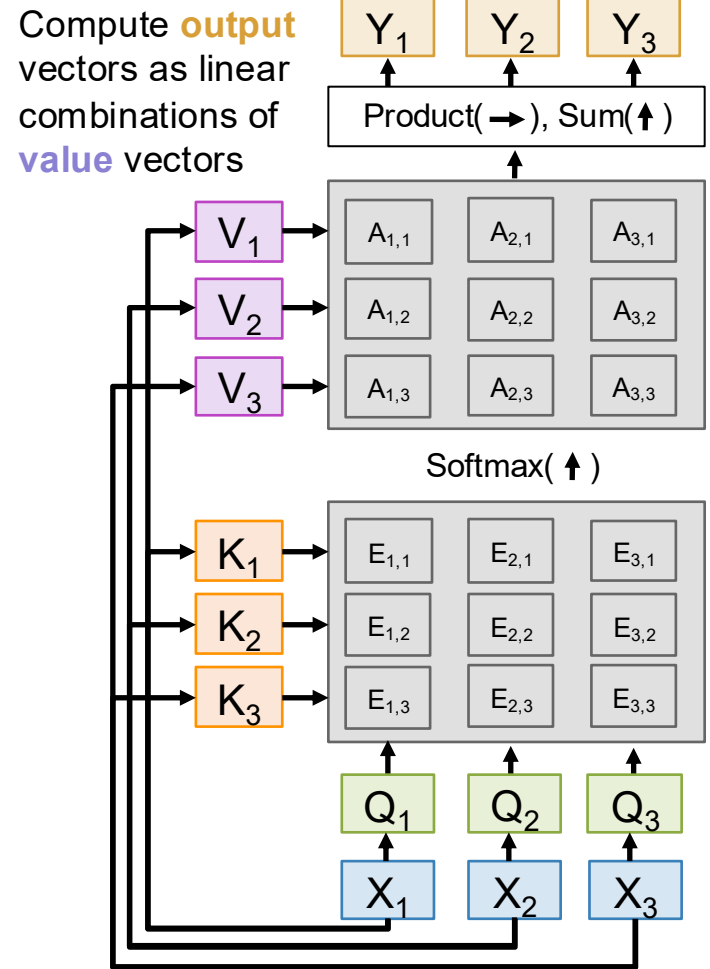
**Similarities:**  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

**Output vector:**  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$



# Self-Attention Layer

## Inputs:

Input vectors:  $\mathbf{X}$  [ $N \times D_{in}$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $\mathbf{W}_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$  [ $N \times D_{out}$ ]

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N \times D_{out}$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $N \times D_{out}$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N \times N$ ]

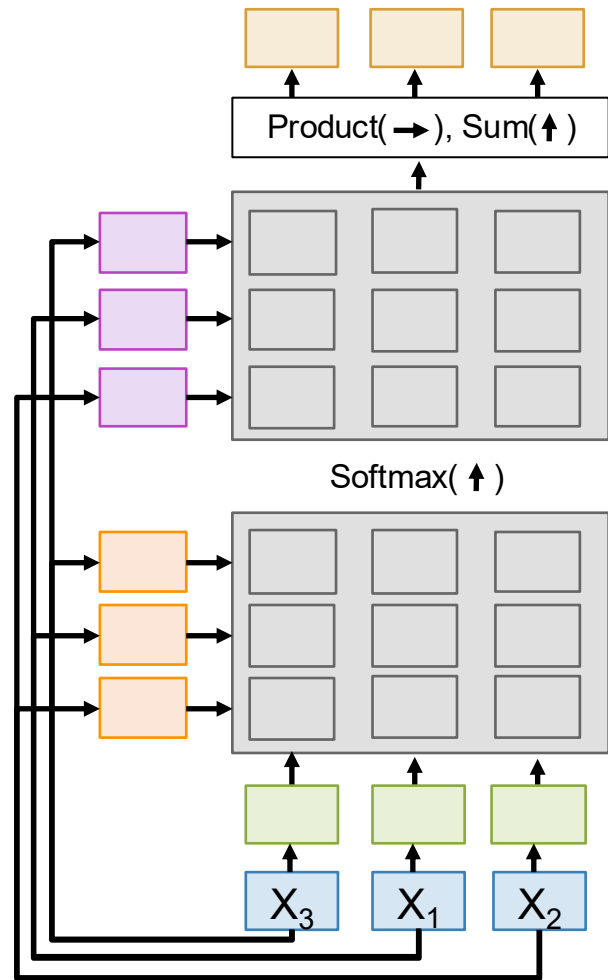
$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Consider permuting **inputs**:



# Self-Attention Layer

## Inputs:

Input vectors:  $\mathbf{X}$  [ $N \times D_{in}$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $\mathbf{W}_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$  [ $N \times D_{out}$ ]

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N \times D_{out}$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $N \times D_{out}$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

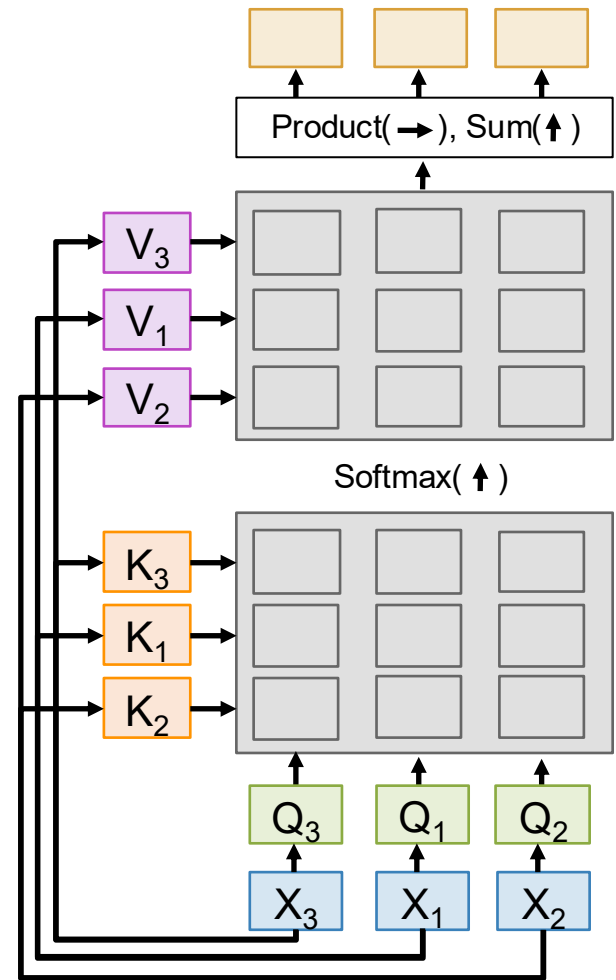
Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Consider permuting **inputs**:

**Queries**, **keys**, and **values** will be the same but permuted



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  [ $N \times D_{in}$ ]

Key matrix:  $W_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $W_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $Q = XW_Q$  [ $N \times D_{out}$ ]

Keys:  $K = XW_K$  [ $N \times D_{out}$ ]

Values:  $V = XW_V$  [ $N \times D_{out}$ ]

Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

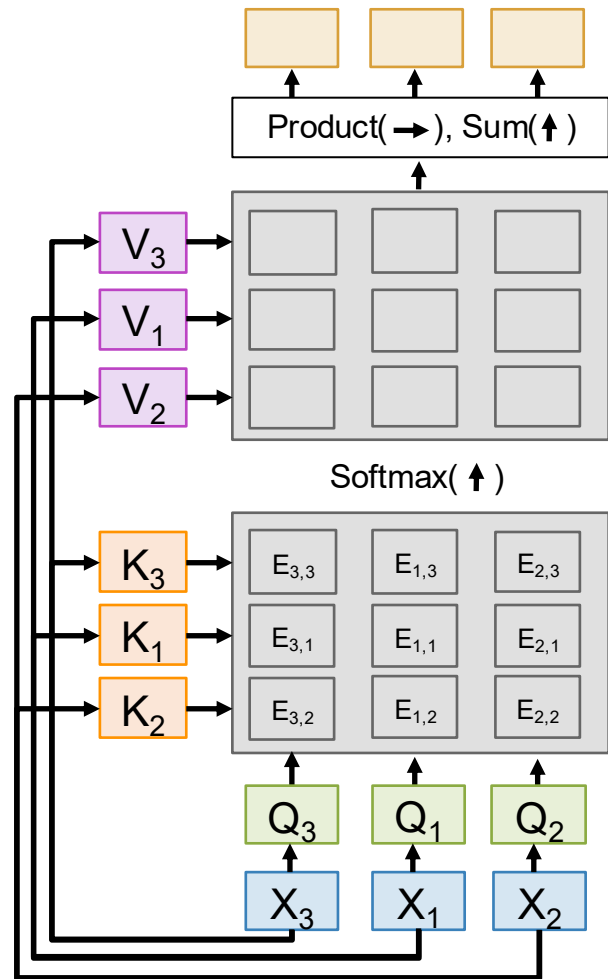
Output vector:  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Consider permuting **inputs**:

**Queries**, **keys**, and **values** will be the same but permuted

Similarities are the same but permuted



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  [ $N \times D_{in}$ ]

Key matrix:  $W_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $W_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $Q = XW_Q$  [ $N \times D_{out}$ ]

Keys:  $K = XW_K$  [ $N \times D_{out}$ ]

Values:  $V = XW_V$  [ $N \times D_{out}$ ]

Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $Y = AV$  [ $N \times D_{out}$ ]

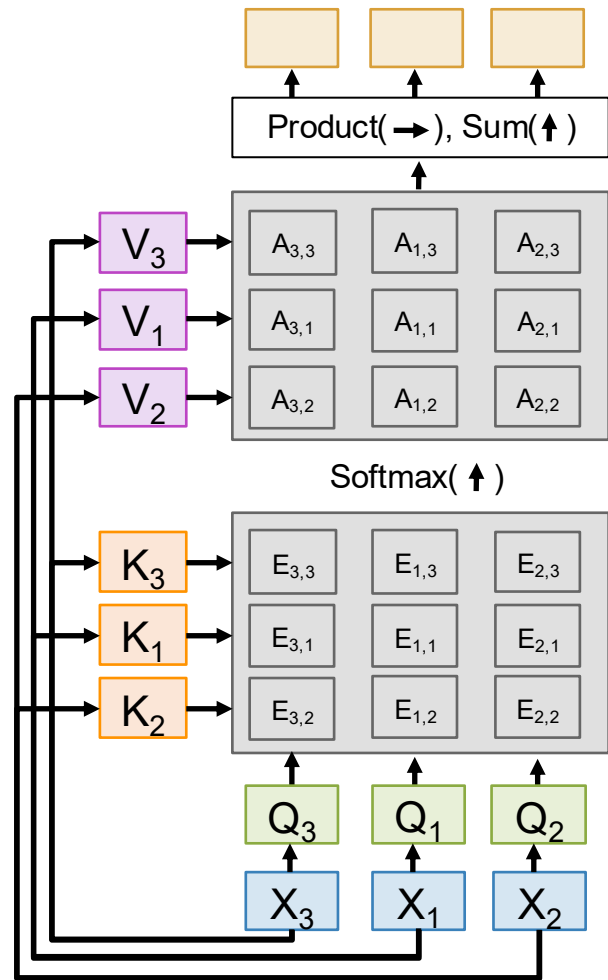
$$Y_i = \sum_j A_{ij} V_j$$

Consider permuting **inputs**:

**Queries**, **keys**, and **values** will be the same but permuted

Similarities are the same but permuted

Attention weights are the same but permuted



# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  [ $N \times D_{in}$ ]

**Key matrix:**  $W_K$  [ $D_{in} \times D_{out}$ ]

**Value matrix:**  $W_V$  [ $D_{in} \times D_{out}$ ]

**Query matrix:**  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

**Queries:**  $Q = XW_Q$  [ $N \times D_{out}$ ]

**Keys:**  $K = XW_K$  [ $N \times D_{out}$ ]

**Values:**  $V = XW_V$  [ $N \times D_{out}$ ]

**Similarities:**  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

**Output vector:**  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

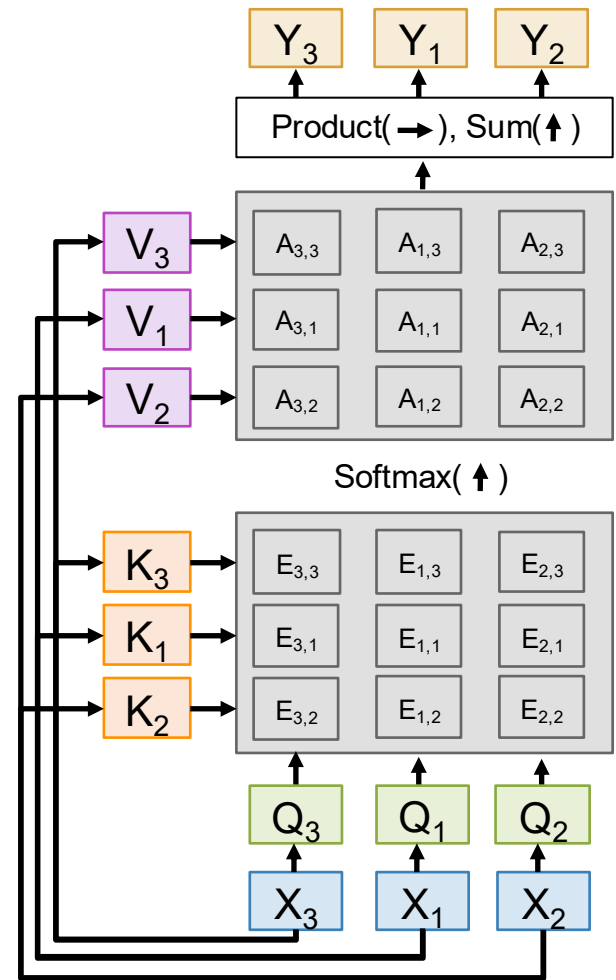
Consider permuting **inputs**:

**Queries**, **keys**, and **values** will be the same but permuted

Similarities are the same but permuted

Attention weights are the same but permuted

**Outputs** are the same but permuted



# Self-Attention Layer

## Inputs:

**Input vectors:**  $X$  [ $N \times D_{in}$ ]

**Key matrix:**  $W_K$  [ $D_{in} \times D_{out}$ ]

**Value matrix:**  $W_V$  [ $D_{in} \times D_{out}$ ]

**Query matrix:**  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

**Queries:**  $Q = XW_Q$  [ $N \times D_{out}$ ]

**Keys:**  $K = XW_K$  [ $N \times D_{out}$ ]

**Values:**  $V = XW_V$  [ $N \times D_{out}$ ]

**Similarities:**  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

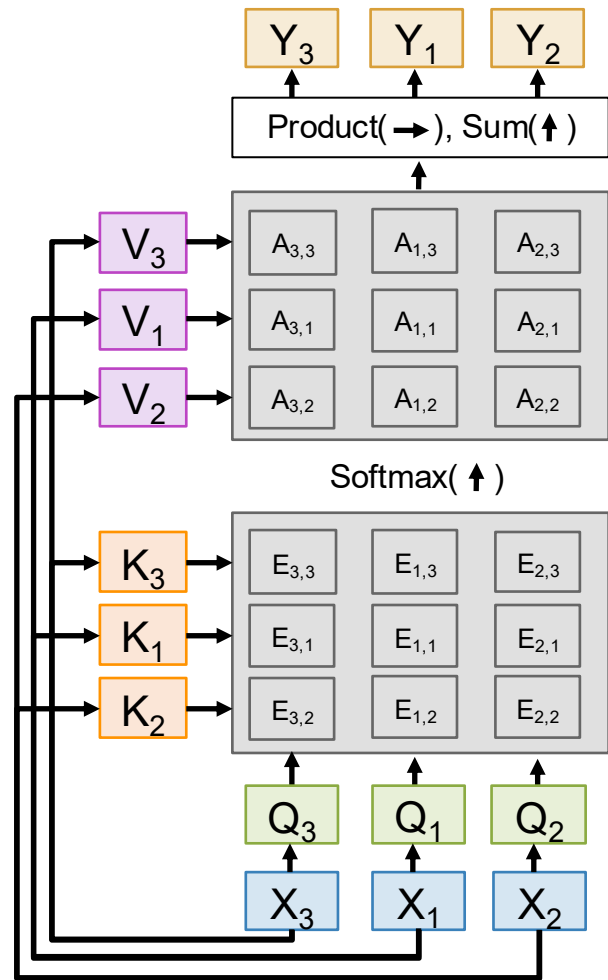
**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

**Output vector:**  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Self-Attention is permutation equivariant:  
 $F(\sigma(X)) = \sigma(F(X))$

This means that Self-Attention works on sets of vectors



# Self-Attention Layer

## Inputs:

Input vectors:  $\mathbf{X}$  [ $N \times D_{in}$ ]

Key matrix:  $\mathbf{W}_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $\mathbf{W}_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $\mathbf{W}_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$  [ $N \times D_{out}$ ]

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $N \times D_{out}$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $N \times D_{out}$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $N \times N$ ]

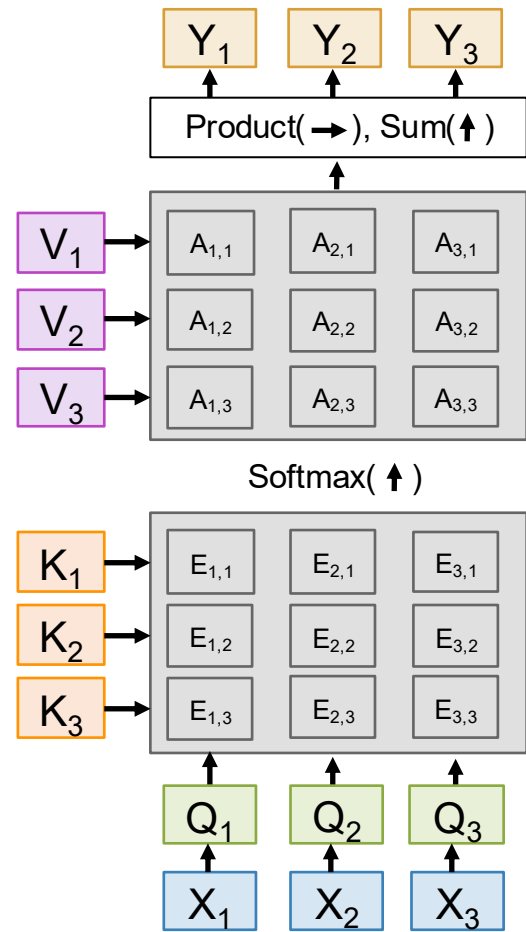
$$E_{ij} = \mathbf{Q}_i \cdot \mathbf{K}_j / \sqrt{D_Q}$$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $\mathbf{Y} = \mathbf{AV}$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

**Problem:** Self-Attention does not know the order of the sequence



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  [ $N \times D_{in}$ ]

Key matrix:  $W_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $W_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $Q = XW_Q$  [ $N \times D_{out}$ ]

Keys:  $K = XW_K$  [ $N \times D_{out}$ ]

Values:  $V = XW_V$  [ $N \times D_{out}$ ]

Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

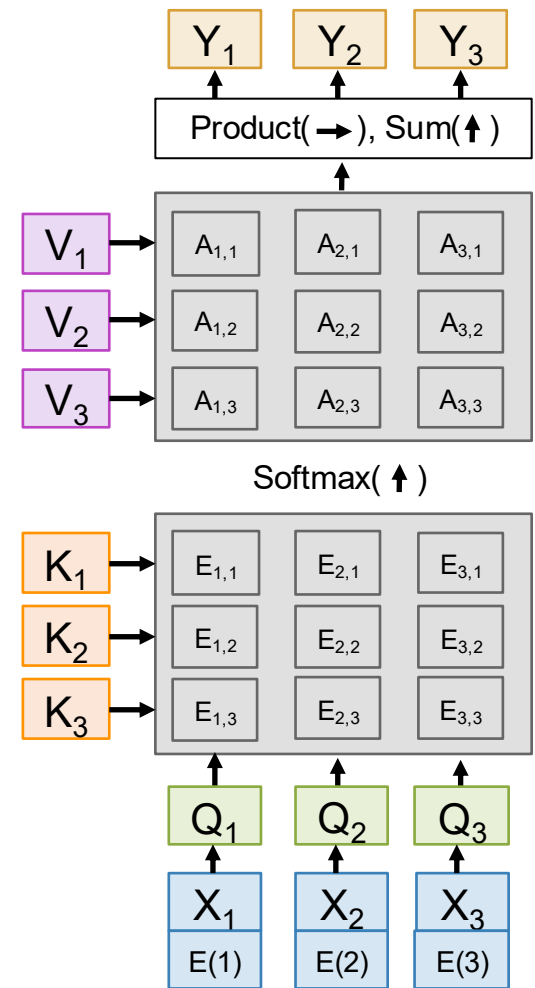
Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

**Problem:** Self-Attention does not know the order of the sequence

**Solution:** Add positional encoding to each input; this is a vector that is a fixed function of the index



# Self-Attention Layer

## Inputs:

Input vectors:  $X$  [ $N \times D_{in}$ ]

Key matrix:  $W_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $W_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $Q = XW_Q$  [ $N \times D_{out}$ ]

Keys:  $K = XW_K$  [ $N \times D_{out}$ ]

Values:  $V = XW_V$  [ $N \times D_{out}$ ]

Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $Y = AV$  [ $N \times D_{out}$ ]

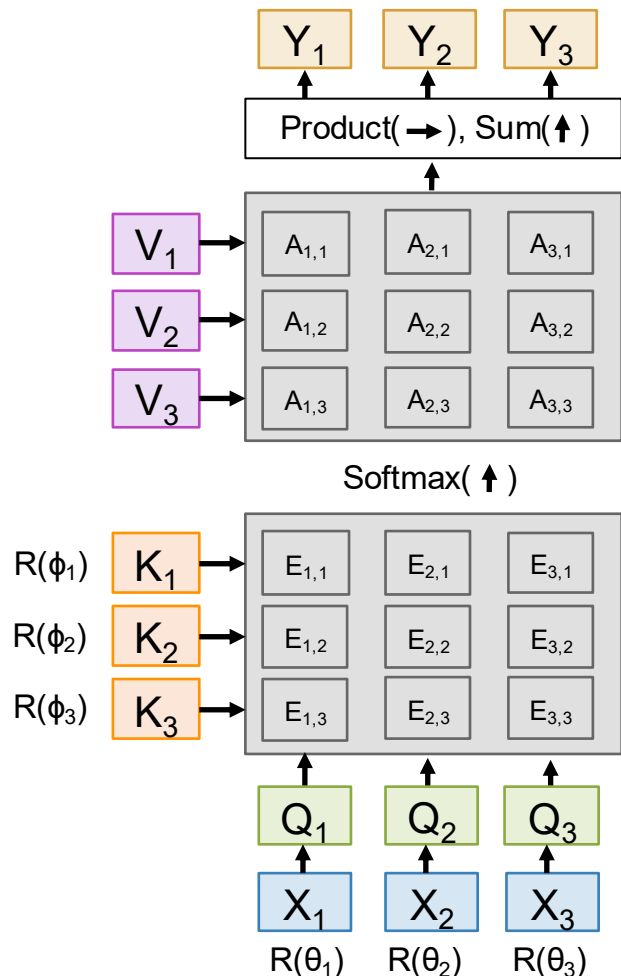
$$Y_i = \sum_j A_{ij} V_j$$

**Problem:** Self-Attention does not know the order of the sequence

**Solution:** Map positions to angles, and *rotate* queries and keys to encode relative positions (RoPE)

$$\begin{aligned} (R(\theta_i)q_i)^T (R(\phi_j)k_j) \\ = q_i^T R(\phi_j - \theta_i)k_j \end{aligned}$$

Su et al, "RoFormer: Enhanced Transformer with Rotary Position Embedding", arXiv 2021



# Masked Self-Attention Layer

Don't let vectors "look ahead" in the sequence

## Inputs:

Input vectors:  $X$  [ $N \times D_{in}$ ]

Key matrix:  $W_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $W_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $W_Q$  [ $D_{in} \times D_{out}$ ]

Override similarities with  $-\infty$ ; this controls which inputs each vector is allowed to look at.

## Computation:

Queries:  $Q = XW_Q$  [ $N \times D_{out}$ ]

Keys:  $K = XW_K$  [ $N \times D_{out}$ ]

Values:  $V = XW_V$  [ $N \times D_{out}$ ]

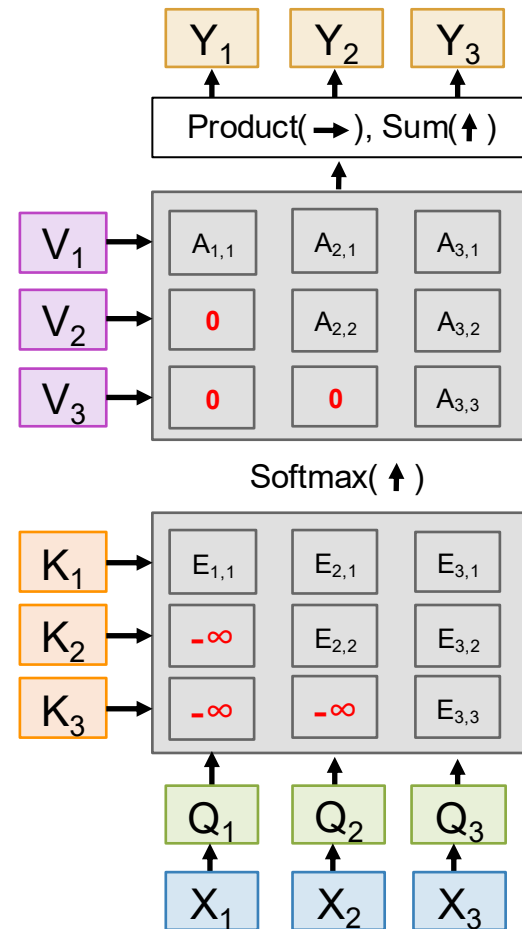
Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$



# Masked Self-Attention Layer

Don't let vectors "look ahead" in the sequence

## Inputs:

**Input vectors:**  $X$  [ $N \times D_{in}$ ]

**Key matrix:**  $W_K$  [ $D_{in} \times D_{out}$ ]

**Value matrix:**  $W_V$  [ $D_{in} \times D_{out}$ ]

**Query matrix:**  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

**Queries:**  $Q = XW_Q$  [ $N \times D_{out}$ ]

**Keys:**  $K = XW_K$  [ $N \times D_{out}$ ]

**Values:**  $V = XW_V$  [ $N \times D_{out}$ ]

**Similarities:**  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

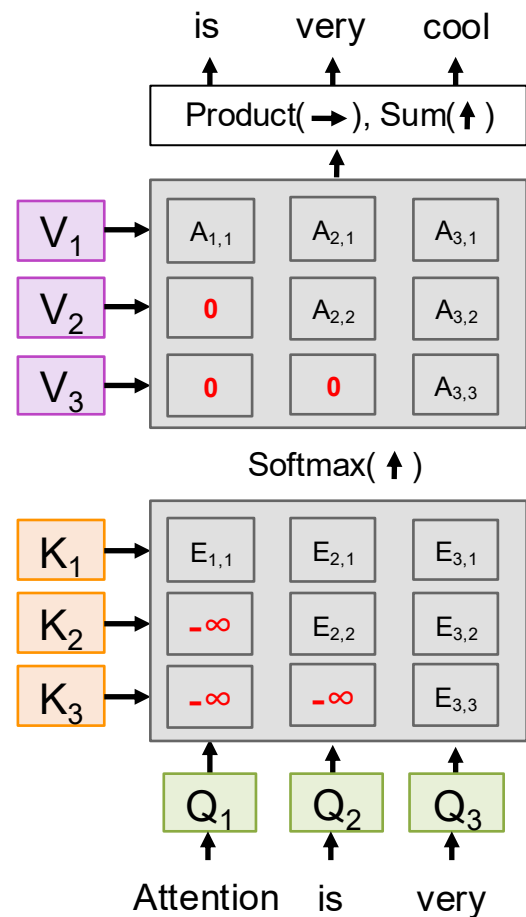
**Attention weights:**  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

**Output vector:**  $Y = AV$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

Override similarities with  $-\infty$ ; this controls which inputs each vector is allowed to look at.

Used for language modeling where you want to predict the next word



# Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

## Inputs:

Input vectors:  $X$  [ $N \times D_{in}$ ]

Key matrix:  $W_K$  [ $D_{in} \times D_{out}$ ]

Value matrix:  $W_V$  [ $D_{in} \times D_{out}$ ]

Query matrix:  $W_Q$  [ $D_{in} \times D_{out}$ ]

## Computation:

Queries:  $Q = XW_Q$  [ $N \times D_{out}$ ]

Keys:  $K = XW_K$  [ $N \times D_{out}$ ]

Values:  $V = XW_V$  [ $N \times D_{out}$ ]

Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $N \times N$ ]

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $N \times N$ ]

Output vector:  $Y = AX$  [ $N \times D_{out}$ ]

$$Y_i = \sum_j A_{ij} V_j$$

$X_1$

$X_2$

$X_3$

# Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

## Inputs:

Input vectors:  $X [N \times D_{in}]$

Key matrix:  $W_K [D_{in} \times D_{out}]$

Value matrix:  $W_V [D_{in} \times D_{out}]$

Query matrix:  $W_Q [D_{in} \times D_{out}]$

## Computation:

Queries:  $Q = XW_Q [N \times D_{out}]$

Keys:  $K = XW_K [N \times D_{out}]$

Values:  $V = XW_V [N \times D_{out}]$

Similarities:  $E = QK^T / \sqrt{D_Q} [N \times N]$

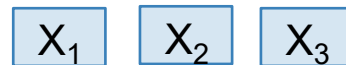
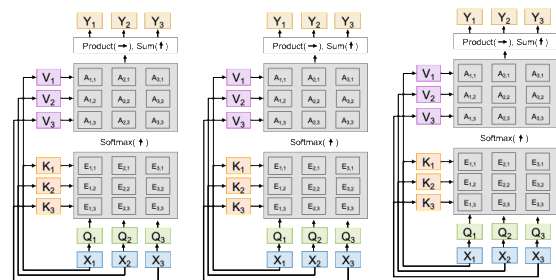
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1) [N \times N]$

Output vector:  $Y = AX [N \times D_{out}]$

$$Y_i = \sum_j A_{ij} V_j$$

H = 3 independent self-attention layers (called heads), each with their own weights



# Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

## Inputs:

Input vectors:  $X [N \times D_{in}]$

Key matrix:  $W_K [D_{in} \times D_{out}]$

Value matrix:  $W_V [D_{in} \times D_{out}]$

Query matrix:  $W_Q [D_{in} \times D_{out}]$

## Computation:

Queries:  $Q = XW_Q [N \times D_{out}]$

Keys:  $K = XW_K [N \times D_{out}]$

Values:  $V = XW_V [N \times D_{out}]$

Similarities:  $E = QK^T / \sqrt{D_Q} [N \times N]$

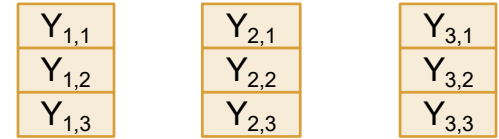
$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1) [N \times N]$

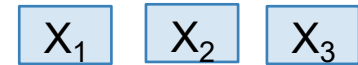
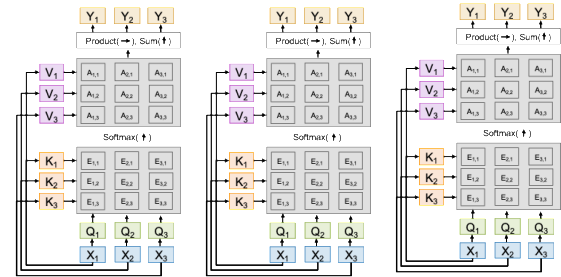
Output vector:  $Y = AX [N \times D_{out}]$

$$Y_i = \sum_j A_{ij} V_j$$

Stack up the H independent outputs for each input X



H = 3 independent self-attention layers (called heads), each with their own weights



# Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

## Inputs:

Input vectors:  $X [N \times D_{in}]$

Key matrix:  $W_K [D_{in} \times D_{out}]$

Value matrix:  $W_V [D_{in} \times D_{out}]$

Query matrix:  $W_Q [D_{in} \times D_{out}]$

## Computation:

Queries:  $Q = XW_Q [N \times D_{out}]$

Keys:  $K = XW_K [N \times D_{out}]$

Values:  $V = XW_V [N \times D_{out}]$

Similarities:  $E = QK^T / \sqrt{D_Q} [N \times N]$

$$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$$

Attention weights:  $A = \text{softmax}(E, \text{dim}=1) [N \times N]$

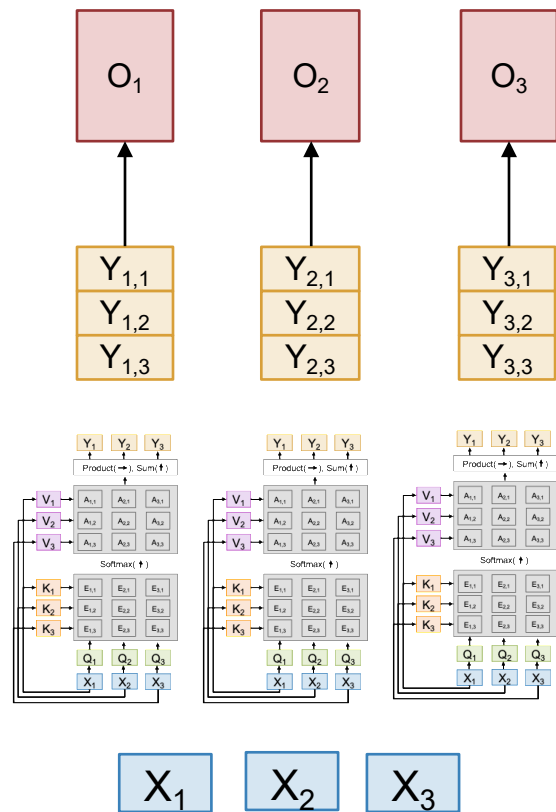
Output vector:  $Y = AX [N \times D_{out}]$

$$Y_i = \sum_j A_{ij} V_j$$

Output projection fuses data from each head

Stack up the H independent outputs for each input X

H = 3 independent self-attention layers (called heads), each with their own weights



# Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

## Inputs:

Input vectors:  $X$  [N x D]

Key matrix:  $W_K$  [D x  $HD_H$ ]

Value matrix:  $W_V$  [D x  $HD_H$ ]

Query matrix:  $W_Q$  [D x  $HD_H$ ]

Output matrix:  $W_O$  [ $HD_H$  x D]

## Computation:

Queries:  $Q = XW_Q$  [ $H \times N \times D_H$ ]

Keys:  $K = XW_K$  [ $H \times N \times D_H$ ]

Values:  $V = XW_V$  [ $H \times N \times D_H$ ]

Similarities:  $E = QK^T / \sqrt{D_Q}$  [ $H \times N \times N$ ]

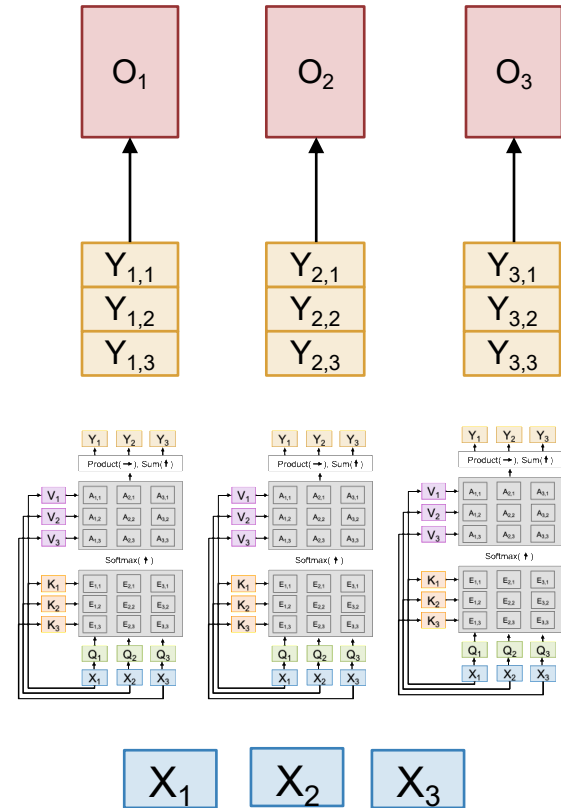
Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [ $H \times N \times N$ ]

Head outputs:  $Y = AV$  [ $H \times N \times D_H$ ]  $\Rightarrow$  [ $N \times HD_H$ ]

Outputs:  $O = YW_O$  [ $N \times D$ ]

Each of the H parallel layers use a qkv dim of  $D_H =$  "head dim"

Usually  $D_H = D / H$ , so inputs and outputs have the same dimension



# Multiheaded Self-Attention Layer

Run H copies of Self-Attention in parallel

## Inputs:

Input vectors:  $X$  [N x D]

Key matrix:  $W_K$  [D x  $HD_H$ ]

Value matrix:  $W_V$  [D x  $HD_H$ ]

Query matrix:  $W_Q$  [D x  $HD_H$ ]

Output matrix:  $W_O$  [ $HD_H$  x D]

In practice, compute all H heads in parallel using batched matrix multiply operations.

## Computation:

Queries:  $Q = XW_Q$  [H x N x  $D_H$ ]

Keys:  $K = XW_K$  [H x N x  $D_H$ ]

Values:  $V = XW_V$  [H x N x  $D_H$ ]

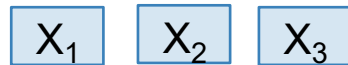
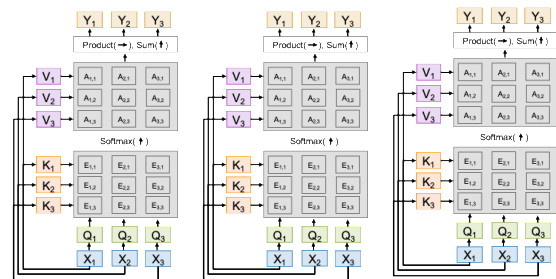
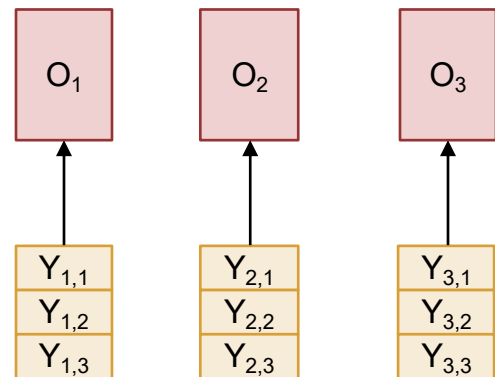
Similarities:  $E = QK^T / \sqrt{D_Q}$  [H x N x N]

Attention weights:  $A = \text{softmax}(E, \text{dim}=1)$  [H x N x N]

Head outputs:  $Y = AV$  [H x N x  $D_H$ ] => [N x  $HD_H$ ]

Outputs:  $O = YW_O$  [N x D]

Used everywhere in practice.



# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$  [N x D]

Key matrix:  $\mathbf{W}_K$  [D x  $HD_H$ ]

Value matrix:  $\mathbf{W}_V$  [D x  $HD_H$ ]

Query matrix:  $\mathbf{W}_Q$  [D x  $HD_H$ ]

Output matrix:  $\mathbf{W}_O$  [ $HD_H$  x D]

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$  [H x N x  $D_H$ ]

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [H x N x  $D_H$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [H x N x  $D_H$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [H x N x N]

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [H x N x N]

Head outputs:  $\mathbf{Y} = \mathbf{AV}$  [H x N x  $D_H$ ]  $\Rightarrow$  [N x  $HD_H$ ]

Outputs:  $\mathbf{O} = \mathbf{YW}_O$  [N x D]

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

## 2. QK Similarity

$[H \times N \times D_H]$   $[H \times N \times D_H] \Rightarrow [H \times N \times N]$

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

## 2. QK Similarity

$[H \times N \times D_H]$   $[H \times N \times D_H] \Rightarrow [H \times N \times N]$

## 3. V-Weighting

$[H \times N \times N]$   $[H \times N \times D_H] \Rightarrow [H \times N \times D_H]$

Reshape to  $[N \times HD_H]$

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

## 2. QK Similarity

$[H \times N \times D_H]$   $[H \times N \times D_H] \Rightarrow [H \times N \times N]$

## 3. V-Weighting

$[H \times N \times N]$   $[H \times N \times D_H] \Rightarrow [H \times N \times D_H]$

Reshape to  $[N \times HD_H]$

## 4. Output Projection

$[N \times HD_H]$   $[HD_H \times D] \Rightarrow [N \times D]$

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

## 2. QK Similarity

$[H \times N \times D_H]$   $[H \times N \times D_H] \Rightarrow [H \times N \times N]$

## 3. V-Weighting

$[H \times N \times N]$   $[H \times N \times D_H] \Rightarrow [H \times N \times D_H]$

Reshape to  $[N \times HD_H]$

## 4. Output Projection

$[N \times HD_H]$   $[HD_H \times D] \Rightarrow [N \times D]$

**Q:** How much compute does this take as the number of vectors  $N$  increases?

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

## 2. QK Similarity

$[H \times N \times D_H]$   $[H \times N \times D_H] \Rightarrow [H \times N \times N]$

## 3. V-Weighting

$[H \times N \times N]$   $[H \times N \times D_H] \Rightarrow [H \times N \times D_H]$

Reshape to  $[N \times HD_H]$

## 4. Output Projection

$[N \times HD_H]$   $[HD_H \times D] \Rightarrow [N \times D]$

**Q:** How much compute does this take as the number of vectors  $N$  increases?

**A:**  $O(N^2)$

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

## 2. QK Similarity

$[H \times N \times D_H]$   $[H \times N \times D_H] \Rightarrow [H \times N \times N]$

## 3. V-Weighting

$[H \times N \times N]$   $[H \times N \times D_H] \Rightarrow [H \times N \times D_H]$

Reshape to  $[N \times HD_H]$

## 4. Output Projection

$[N \times HD_H]$   $[HD_H \times D] \Rightarrow [N \times D]$

**Q:** How much memory does this take as the number of vectors  $N$  increases?

# Self-Attention is Four Matrix Multiplies!

## Inputs:

Input vectors:  $\mathbf{X}$   $[N \times D]$

Key matrix:  $\mathbf{W}_K$   $[D \times HD_H]$

Value matrix:  $\mathbf{W}_V$   $[D \times HD_H]$

Query matrix:  $\mathbf{W}_Q$   $[D \times HD_H]$

Output matrix:  $\mathbf{W}_O$   $[HD_H \times D]$

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$   $[H \times N \times D_H]$

Keys:  $\mathbf{K} = \mathbf{XW}_K$   $[H \times N \times D_H]$

Values:  $\mathbf{V} = \mathbf{XW}_V$   $[H \times N \times D_H]$

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$   $[H \times N \times N]$

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$   $[H \times N \times N]$

Head outputs:  $\mathbf{Y} = \mathbf{AV}$   $[H \times N \times D_H] \Rightarrow [N \times HD_H]$

Outputs:  $\mathbf{O} = \mathbf{YW}_O$   $[N \times D]$

## 1. QKV Projection

$[N \times D]$   $[D \times 3HD_H] \Rightarrow [N \times 3HD_H]$

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of shape  $[H \times N \times D_H]$

## 2. QK Similarity

$[H \times N \times D_H]$   $[H \times N \times D_H] \Rightarrow [H \times N \times N]$

## 3. V-Weighting

$[H \times N \times N]$   $[H \times N \times D_H] \Rightarrow [H \times N \times D_H]$

Reshape to  $[N \times HD_H]$

## 4. Output Projection

$[N \times HD_H]$   $[HD_H \times D] \Rightarrow [N \times D]$

**Q:** How much memory does this take as the number of vectors  $N$  increases?

**A:**  $O(N^2)$

# Self-Attention is Four Matrix Multiplies!

If  $N=100K$ ,  $H=64$  then  
 $H \times N \times N$  attention weights  
take 1.192 TB! GPUs don't  
have that much memory...

## Inputs:

Input vectors:  $\mathbf{X}$  [ $N \times D$ ]

Key matrix:  $\mathbf{W}_K$  [ $D \times HD_H$ ]

Value matrix:  $\mathbf{W}_V$  [ $D \times HD_H$ ]

Query matrix:  $\mathbf{W}_Q$  [ $D \times HD_H$ ]

Output matrix:  $\mathbf{W}_O$  [ $HD_H \times D$ ]

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$  [ $H \times N \times D_H$ ]

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $H \times N \times D_H$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $H \times N \times D_H$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $H \times N \times N$ ]

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $H \times N \times N$ ]

Head outputs:  $\mathbf{Y} = \mathbf{AV}$  [ $H \times N \times D_H$ ]  $\Rightarrow$  [ $N \times HD_H$ ]

Outputs:  $\mathbf{O} = \mathbf{YW}_O$  [ $N \times D$ ]

## 1. QKV Projection

$[N \times D]$  [ $D \times 3HD_H$ ]  $\Rightarrow$  [ $N \times 3HD_H$ ]

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of  
shape [ $H \times N \times D_H$ ]

## 2. QK Similarity

$[H \times N \times D_H]$  [ $H \times N \times D_H$ ]  $\Rightarrow$  [ $H \times N \times N$ ]

## 3. V-Weighting

$[H \times N \times N]$  [ $H \times N \times D_H$ ]  $\Rightarrow$  [ $H \times N \times D_H$ ]

Reshape to [ $N \times HD_H$ ]

## 4. Output Projection

$[N \times HD_H]$  [ $HD_H \times D$ ]  $\Rightarrow$  [ $N \times D$ ]

**Q:** How much memory does this take  
as the number of vectors  $N$  increases?

**A:**  $O(N^2)$

# Self-Attention is Four Matrix Multiplies!

If  $N=100K$ ,  $H=64$  then  
 $H \times N \times N$  attention weights  
take 1.192 TB! GPUs don't  
have that much memory...

## Inputs:

Input vectors:  $\mathbf{X}$  [ $N \times D$ ]

Key matrix:  $\mathbf{W}_K$  [ $D \times HD_H$ ]

Value matrix:  $\mathbf{W}_V$  [ $D \times HD_H$ ]

Query matrix:  $\mathbf{W}_Q$  [ $D \times HD_H$ ] Makes large  $N$

Output matrix:  $\mathbf{W}_O$  [ $HD_H \times D$ ] possible

## Computation:

Queries:  $\mathbf{Q} = \mathbf{XW}_Q$  [ $H \times N \times D_H$ ]

Keys:  $\mathbf{K} = \mathbf{XW}_K$  [ $H \times N \times D_H$ ]

Values:  $\mathbf{V} = \mathbf{XW}_V$  [ $H \times N \times D_H$ ]

Similarities:  $\mathbf{E} = \mathbf{QK}^T / \sqrt{D_Q}$  [ $H \times N \times N$ ]

Attention weights:  $\mathbf{A} = \text{softmax}(\mathbf{E}, \text{dim}=1)$  [ $H \times N \times N$ ]

Head outputs:  $\mathbf{Y} = \mathbf{AV}$  [ $H \times N \times D_H$ ]  $\Rightarrow$  [ $N \times HD_H$ ]

Outputs:  $\mathbf{O} = \mathbf{YW}_O$  [ $N \times D$ ]

Flash Attention  
algorithm computes  
2+3 at the same time  
without storing the  
full attention matrix!

## 1. QKV Projection

$[N \times D]$  [ $D \times 3HD_H$ ]  $\Rightarrow$  [ $N \times 3HD_H$ ]

Split and reshape to get  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$  each of  
shape [ $H \times N \times D_H$ ]

## 2. QK Similarity

$[H \times N \times D_H]$  [ $H \times N \times D_H$ ]  $\Rightarrow$  [ $H \times N \times N$ ]

## 3. V-Weighting

$[H \times N \times N]$  [ $H \times N \times D_H$ ]  $\Rightarrow$  [ $H \times N \times D_H$ ]

Reshape to [ $N \times HD_H$ ]

## 4. Output Projection

$[N \times HD_H]$  [ $HD_H \times D$ ]  $\Rightarrow$  [ $N \times D$ ]

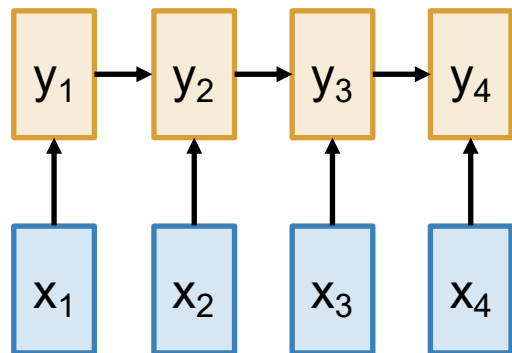
**Q:** How much memory does this take  
as the number of vectors  $N$  increases?

**A:**  $O(N)$  with Flash Attention

# Three Ways of Processing Sequences

# Three Ways of Processing Sequences

## Recurrent Neural Network



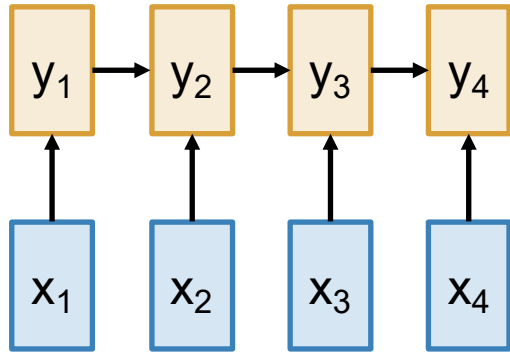
Works on **1D ordered sequences**

(+) Theoretically good at long sequences:  $O(N)$  compute and memory for a sequence of length  $N$

(-) Not parallelizable. Need to compute hidden states sequentially

# Three Ways of Processing Sequences

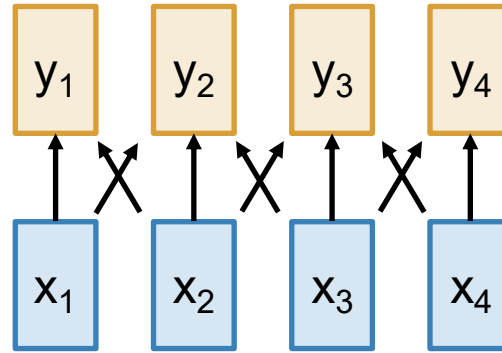
## Recurrent Neural Network



Works on **1D ordered sequences**

- (+) Theoretically good at long sequences:  $O(N)$  compute and memory for a sequence of length  $N$
- (-) Not parallelizable. Need to compute hidden states sequentially

## Convolution

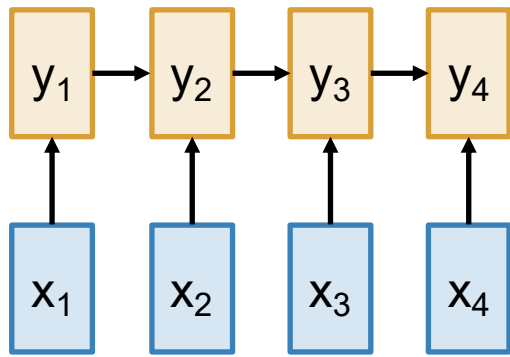


Works on **N-dimensional grids**

- (-) Bad for long sequences: need to stack many layers to build up large receptive fields
- (+) Parallelizable, outputs can be computed in parallel

# Three Ways of Processing Sequences

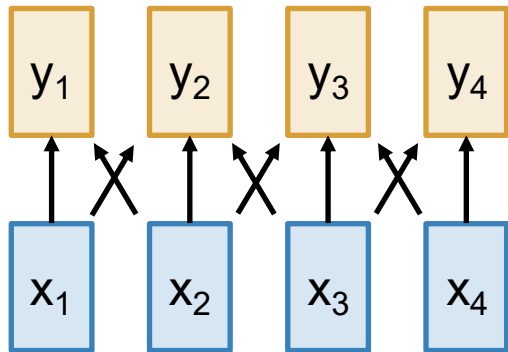
## Recurrent Neural Network



Works on **1D ordered sequences**

- (+) Theoretically good at long sequences:  $O(N)$  compute and memory for a sequence of length  $N$
- (-) Not parallelizable. Need to compute hidden states sequentially

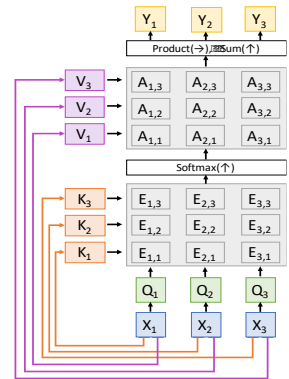
## Convolution



Works on **N-dimensional grids**

- (-) Bad for long sequences: need to stack many layers to build up large receptive fields
- (+) Parallelizable, outputs can be computed in parallel

## Self-Attention



Works on **sets of vectors**

- (+) Great for long sequences; each output depends directly on all inputs
- (+) Highly parallel, it's just 4 matmuls
- (-) Expensive:  $O(N^2)$  compute,  $O(N)$  memory for sequence of length  $N$

# Three Ways of Processing Sequences

Recurrent Neural Network

Convolution

Self-Attention

$Y_1$   $Y_2$   $Y_3$

## Attention is All You Need

Vaswani et al, NeurIPS 2017

sequences.  $O(N)$  compute and memory for a sequence of length  $N$   
(-) Not parallelizable. Need to compute hidden states sequentially

stack many layers to build up large receptive fields  
(+) Parallelizable, outputs can be computed in parallel

output depends directly on all inputs  
(+) Highly parallel, it's just 4 matmuls  
(-) Expensive:  $O(N^2)$  compute,  $O(N)$  memory for sequence of length  $N$

# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

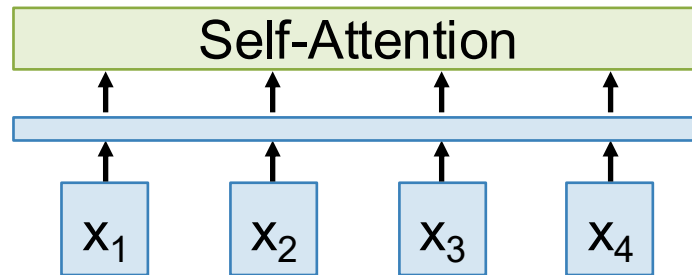


# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

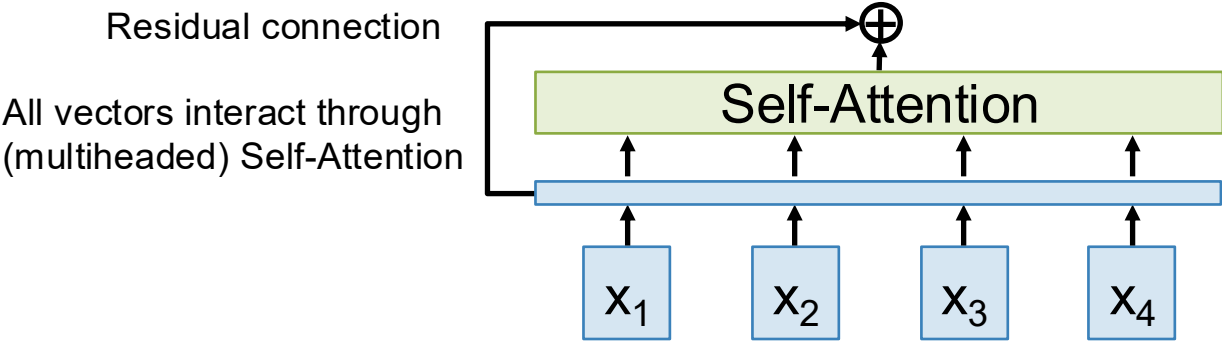
All vectors interact through  
(multiheaded) Self-Attention



# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$



Vaswani et al, "Attention is all you need," NeurIPS 2017

# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

Recall **Layer Normalization**:

Given  $h_1, \dots, h_N$  (Shape:  $D$ )

scale:  $\gamma$  (Shape:  $D$ )

shift:  $\beta$  (Shape:  $D$ )

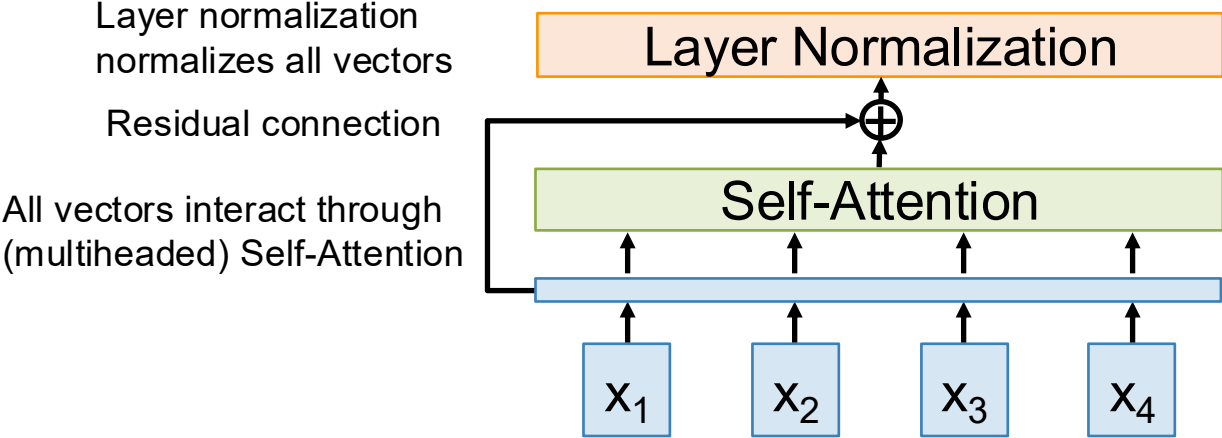
$\mu_i = (\sum_j h_{i,j})/D$  (scalar)

$\sigma_i = (\sum_j (h_{i,j} - \mu_i)^2/D)^{1/2}$  (scalar)

$z_i = (h_i - \mu_i) / \sigma_i$

$y_i = \gamma * z_i + \beta$

Ba et al, 2016



Vaswani et al, "Attention is all you need," NeurIPS 2017

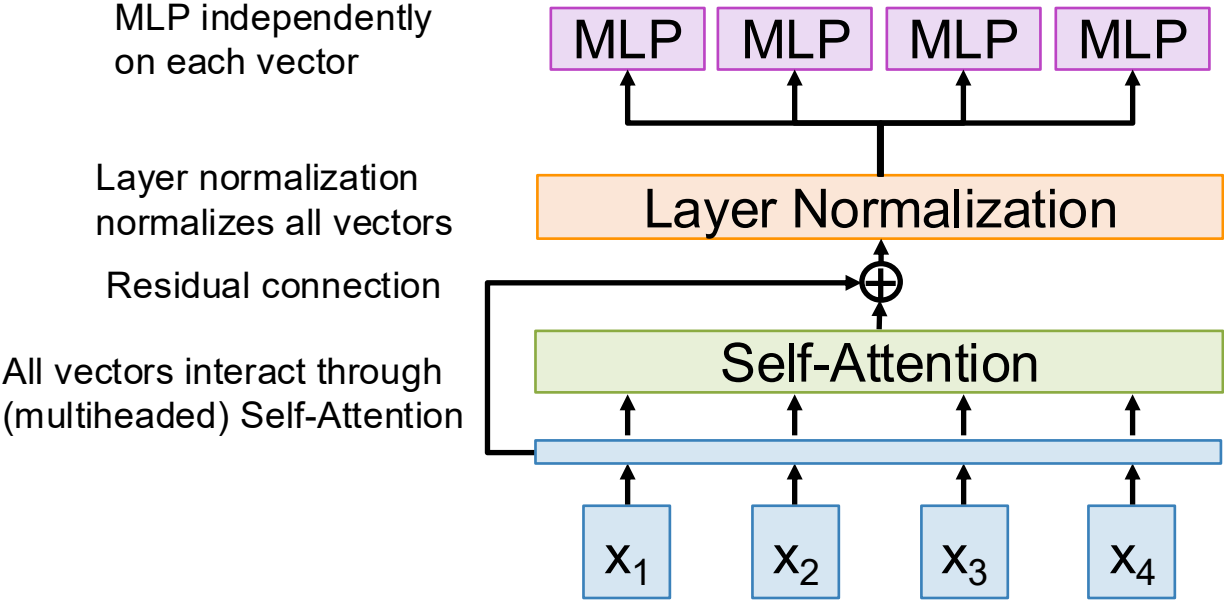
# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

Usually a two-layer MLP;  
classic setup is  
 $D \Rightarrow 4D \Rightarrow D$

Also sometimes called FFN  
(Feed-Forward Network)

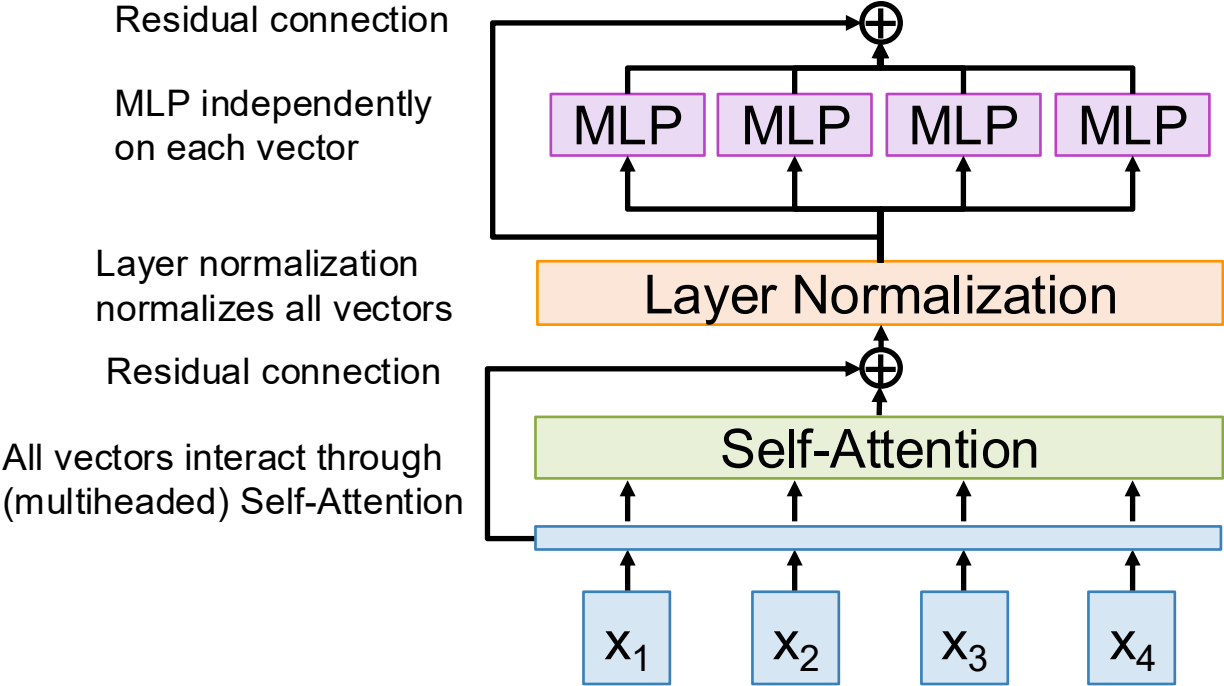


Vaswani et al, "Attention is all you need," NeurIPS 2017

# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

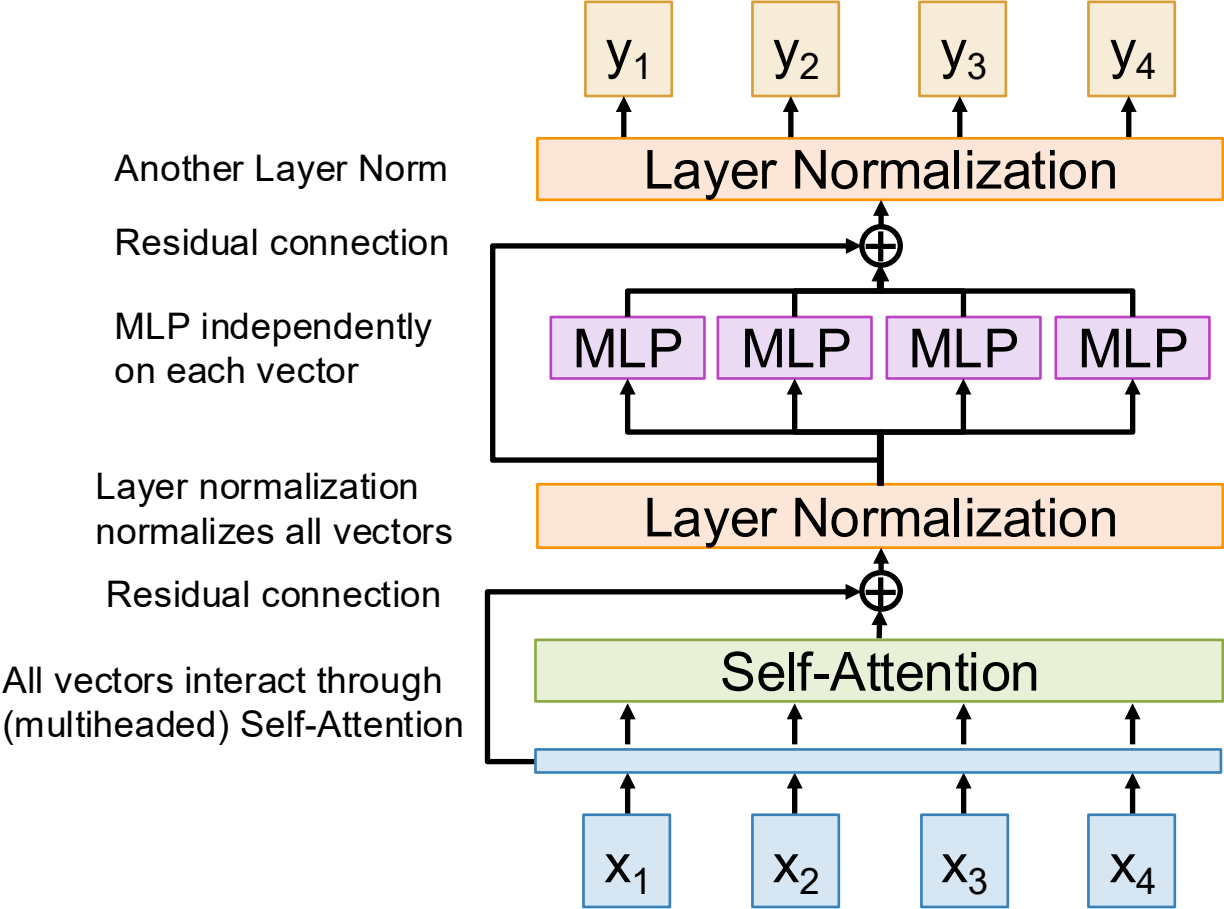


Vaswani et al, "Attention is all you need," NeurIPS 2017

# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$



Another Layer Norm

Residual connection

MLP independently on each vector

Layer normalization normalizes all vectors

Residual connection

All vectors interact through (multiheaded) Self-Attention

Vaswani et al, "Attention is all you need," NeurIPS 2017

# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

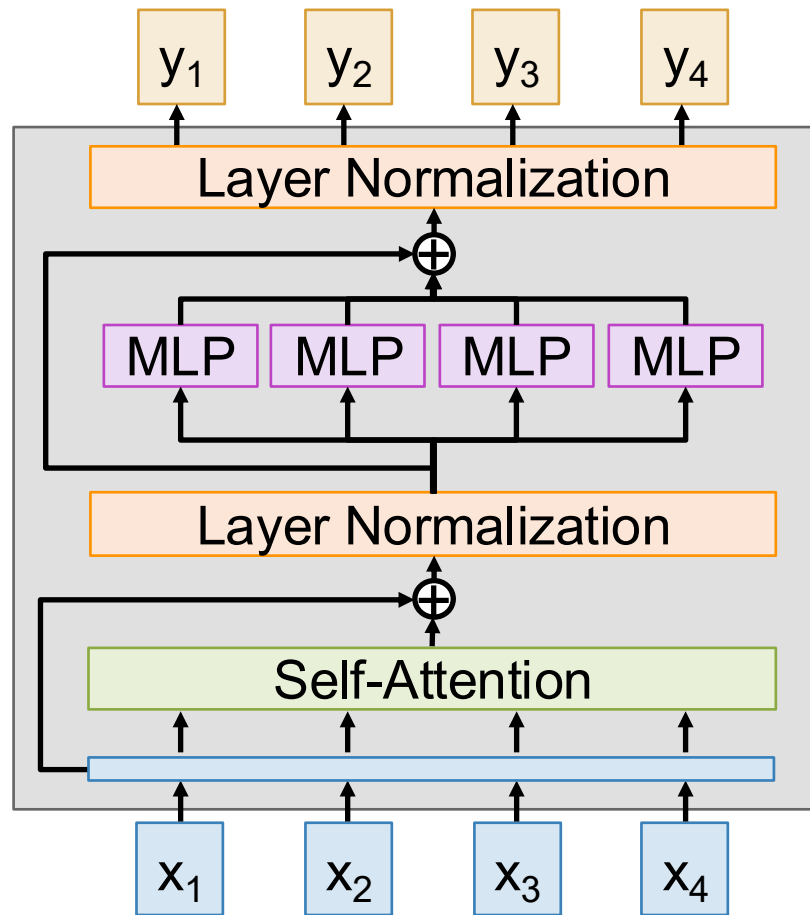
**Output:** Set of vectors  $y$

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention  
2 from MLP



# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

**Output:** Set of vectors  $y$

Self-Attention is the only interaction between vectors

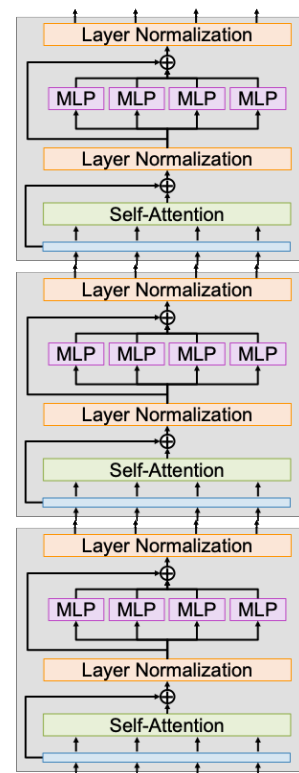
LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention  
2 from MLP

A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger



# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

**Output:** Set of vectors  $y$

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

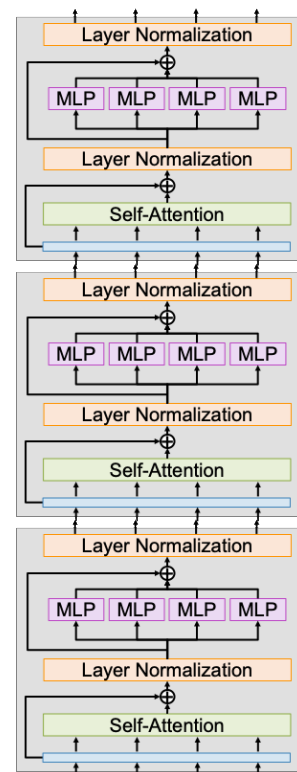
Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention  
2 from MLP

A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger

Original: [Vaswani et al, 2017]  
12 blocks,  $D=1024$ ,  $H=16$ ,  $N=512$   
213M params



# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

**Output:** Set of vectors  $y$

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

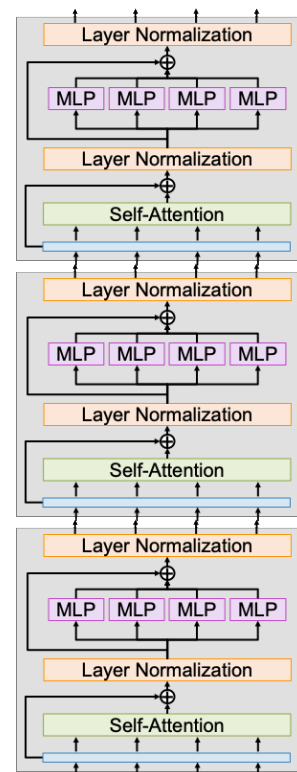
4 from Self-Attention  
2 from MLP

A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger

Original: [Vaswani et al, 2017]  
12 blocks,  $D=1024$ ,  $H=16$ ,  $N=512$   
213M params

GPT-2: [Radford et al, 2019]  
48 blocks,  $D=1600$ ,  $H=25$ ,  $N=1024$   
1.5B params



# The Transformer

## Transformer Block

**Input:** Set of vectors  $x$

**Output:** Set of vectors  $y$

Self-Attention is the only interaction between vectors

LayerNorm and MLP work on each vector independently

Highly scalable and parallelizable, most of the compute is just 6 matmuls:

4 from Self-Attention  
2 from MLP

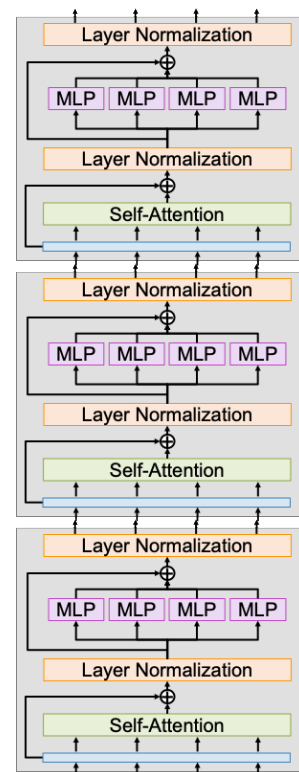
A **Transformer** is just a stack of identical Transformer blocks!

They have not changed much since 2017... but have gotten a lot bigger

Original: [Vaswani et al, 2017]  
12 blocks,  $D=1024$ ,  $H=16$ ,  $N=512$   
213M params

GPT-2: [Radford et al, 2019]  
48 blocks,  $D=1600$ ,  $H=25$ ,  $N=1024$   
1.5B params

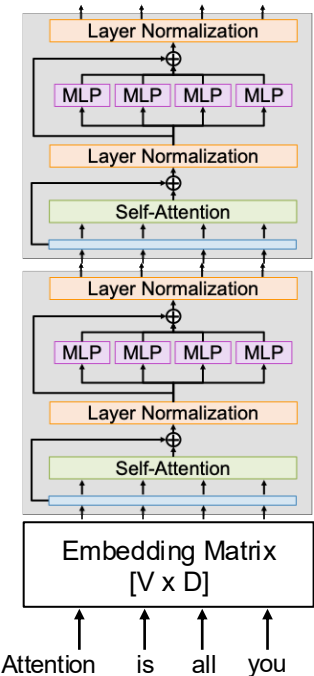
GPT-3: [Brown et al, 2020]  
96 blocks,  $D=12288$ ,  $H=96$ ,  $N=2048$   
175B params



# Transformers for Language Modeling (LLM)

Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size  $V$  and model dimension  $D$ , it's a lookup table of shape  $[V \times D]$

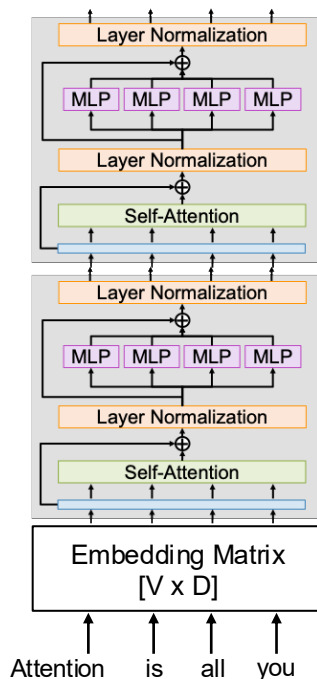


# Transformers for Language Modeling (LLM)

Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size  $V$  and model dimension  $D$ , it's a lookup table of shape  $[V \times D]$

Use masked attention inside each transformer block so each token can only see the ones before it



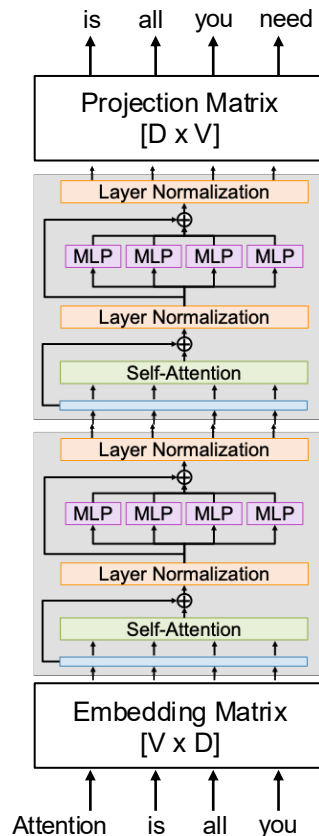
# Transformers for Language Modeling (LLM)

Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size  $V$  and model dimension  $D$ , it's a lookup table of shape  $[V \times D]$

Use masked attention inside each transformer block so each token can only see the ones before it

At the end, learn a projection matrix of shape  $[D \times V]$  to project each  $D$ -dim vector to a  $V$ -dim vector of scores for each element of the vocabulary.



# Transformers for Language Modeling (LLM)

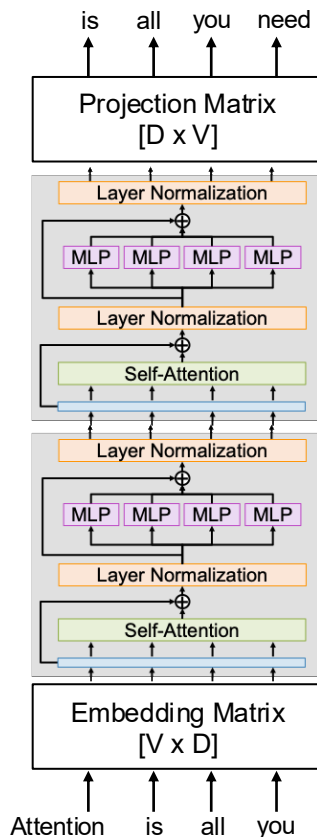
Learn an embedding matrix at the start of the model to convert words into vectors.

Given vocab size  $V$  and model dimension  $D$ , it's a lookup table of shape  $[V \times D]$

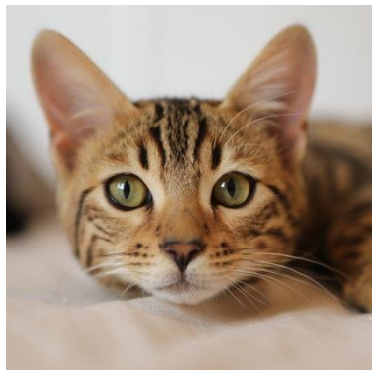
Use masked attention inside each transformer block so each token can only see the ones before it

At the end, learn a projection matrix of shape  $[D \times V]$  to project each  $D$ -dim vector to a  $V$ -dim vector of scores for each element of the vocabulary.

Train to predict next token using softmax + cross-entropy loss



# Vision Transformers (ViT)



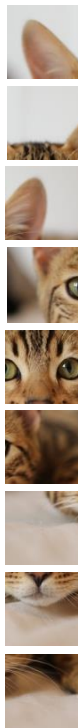
Input image:  
e.g. 224x224x3

Dosovitskiy et al, "An Image is Worth  
16x16 Words: Transformers for Image  
Recognition at Scale", ICLR 2021

# Vision Transformers (ViT)



Input image:  
e.g. 224x224x3



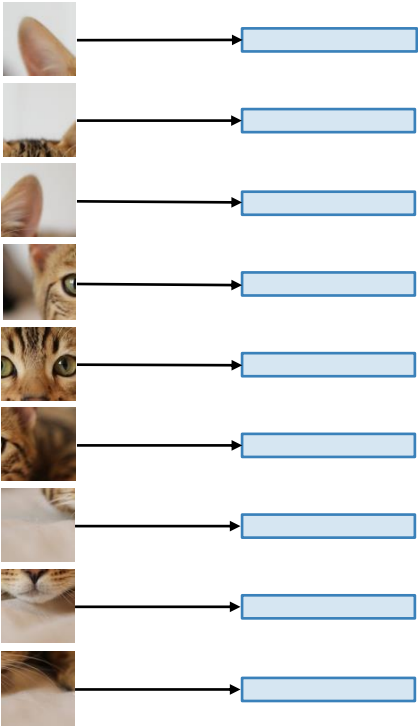
Break into patches  
e.g. 16x16x3

Dosovitskiy et al, "An Image is Worth  
16x16 Words: Transformers for Image  
Recognition at Scale", ICLR 2021

# Vision Transformers (ViT)



Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

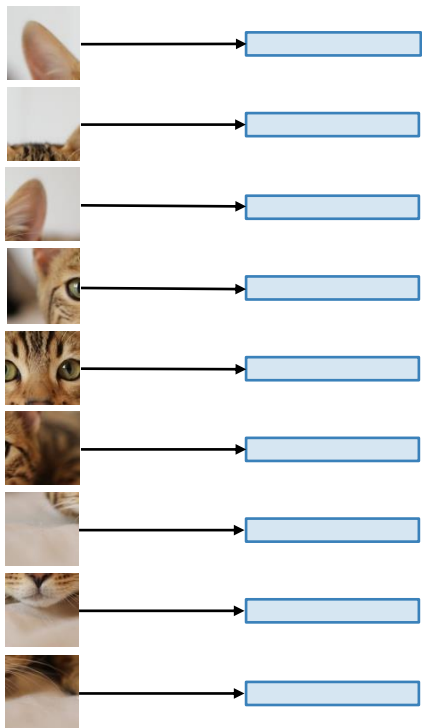
Flatten and apply a linear  
transform 768 => D

Dosovitskiy et al, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

# Vision Transformers (ViT)



Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

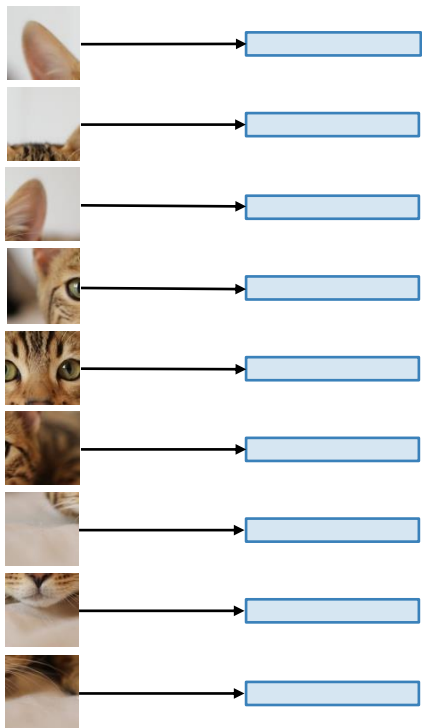
Flatten and apply a linear  
transform  $768 \Rightarrow D$

**Q:** Any other way to  
describe this operation?

# Vision Transformers (ViT)



Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

Flatten and apply a linear  
transform 768 => D

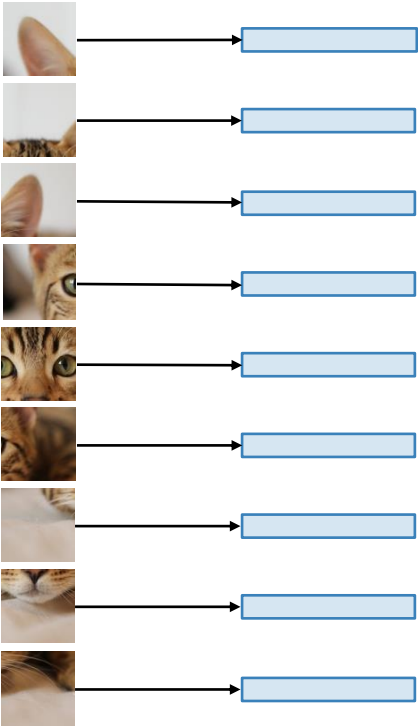
**Q:** Any other way to describe this operation?

**A:** 16x16 conv with stride 16, 3 input channels, D output channels

# Vision Transformers (ViT)

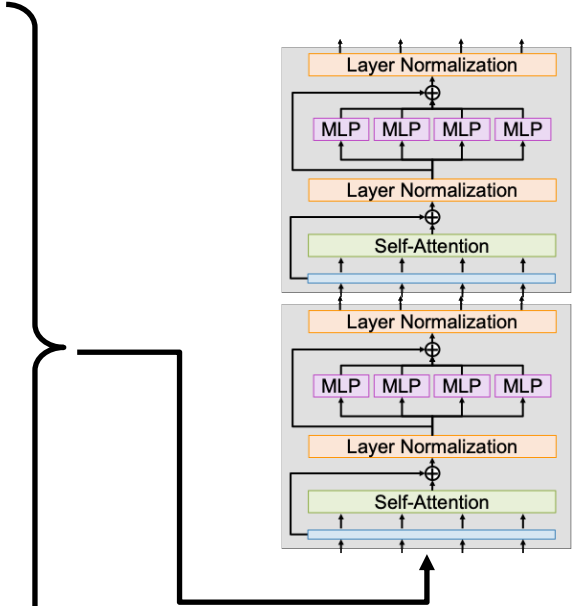


Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

Flatten and apply a linear  
transform 768 => D



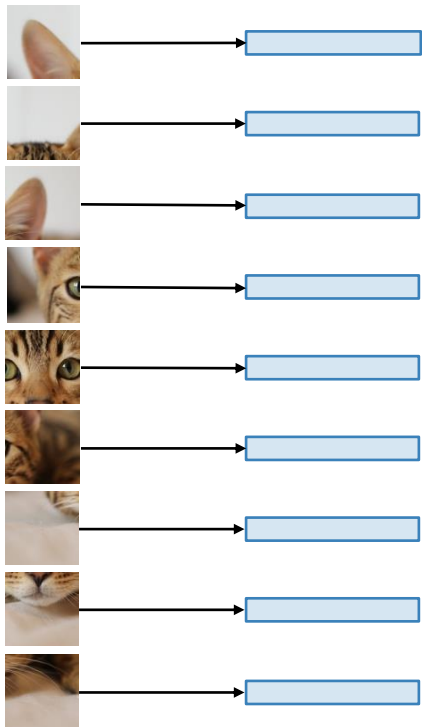
D-dim vector per patch  
are the input vectors to  
the Transformer

Dosovitskiy et al, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", ICLR 2021

# Vision Transformers (ViT)

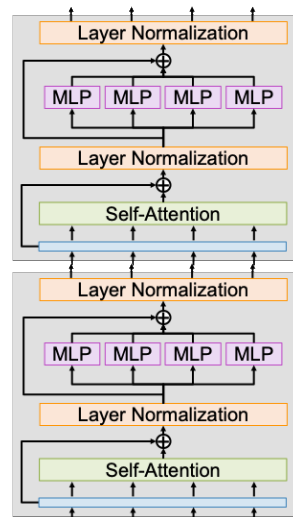


Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

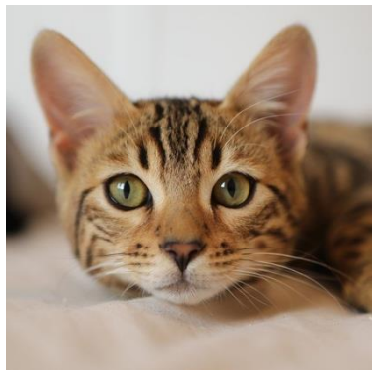
Flatten and apply a linear  
transform  $768 \Rightarrow D$



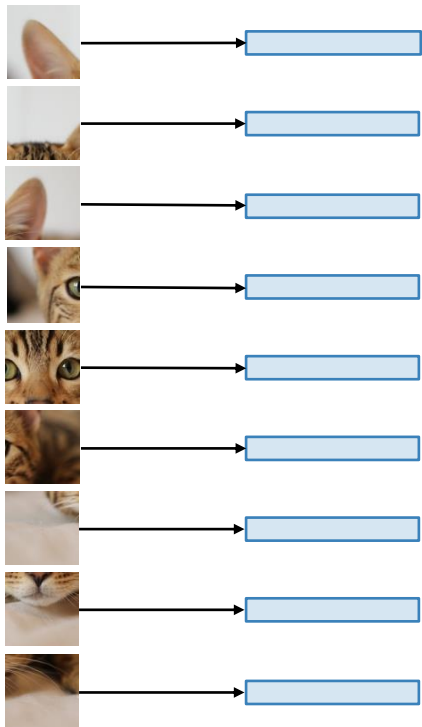
D-dim vector per patch  
are the input vectors to  
the Transformer

Use positional  
encoding to tell  
the transformer  
the 2D position  
of each patch

# Vision Transformers (ViT)

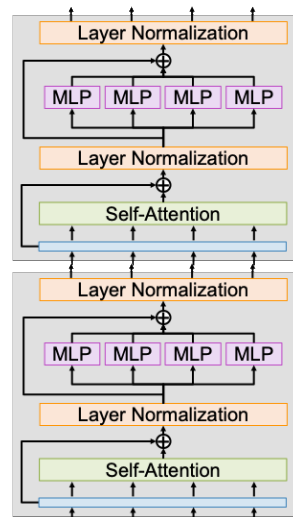


Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

Flatten and apply a linear  
transform  $768 \Rightarrow D$



D-dim vector per patch  
are the input vectors to  
the Transformer

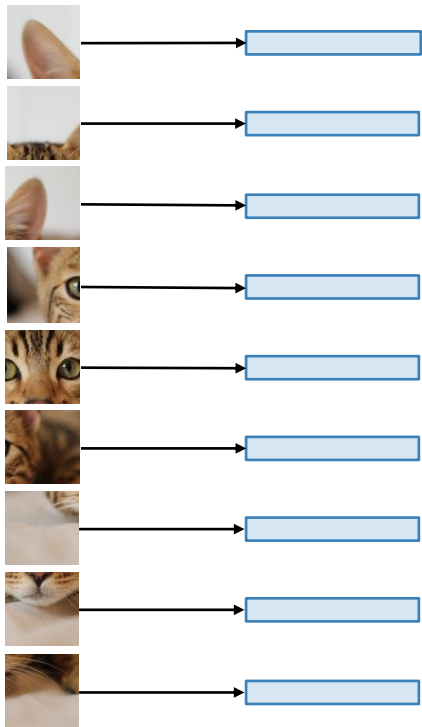
Don't use any  
masking; each  
image patch can  
look at all other  
image patches

Use positional  
encoding to tell  
the transformer  
the 2D position  
of each patch

# Vision Transformers (ViT)

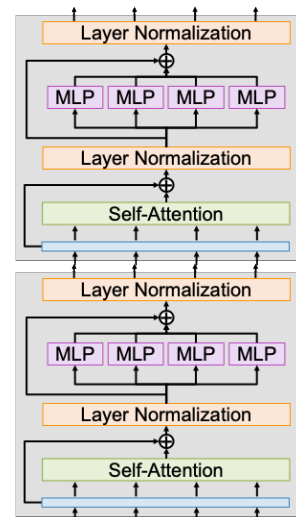


Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

Flatten and apply a linear  
transform  $768 \Rightarrow D$



D-dim vector per patch  
are the input vectors to  
the Transformer

Transformer  
gives an output  
vector per patch

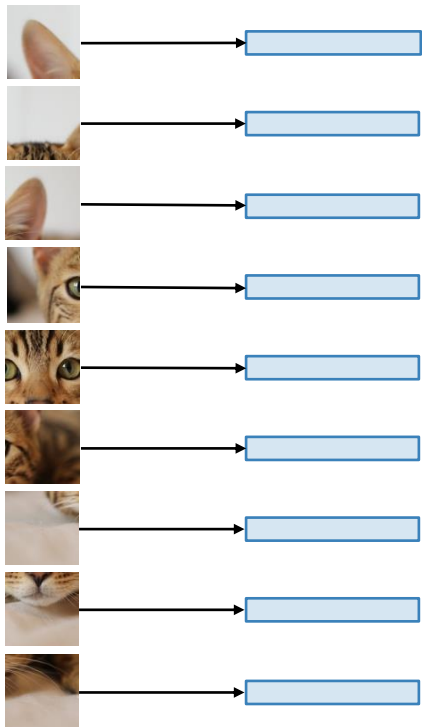
Don't use any  
masking; each  
image patch can  
look at all other  
image patches

Use positional  
encoding to tell  
the transformer  
the 2D position  
of each patch

# Vision Transformers (ViT)



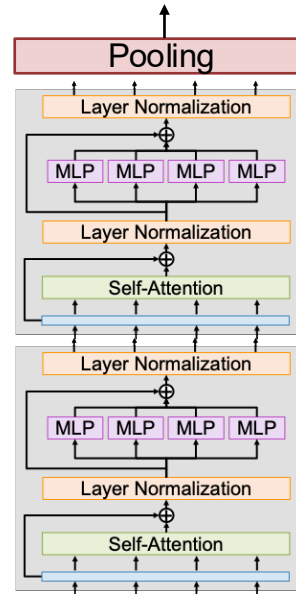
Input image:  
e.g. 224x224x3



Break into patches  
e.g. 16x16x3

Flatten and apply a linear  
transform  $768 \Rightarrow D$

Average pool  $N \times D$  vectors to  
 $1 \times D$ , apply a linear layer  
 $D \Rightarrow C$  to predict class scores



$D$ -dim vector per patch  
are the input vectors to  
the Transformer

Transformer  
gives an output  
vector per patch

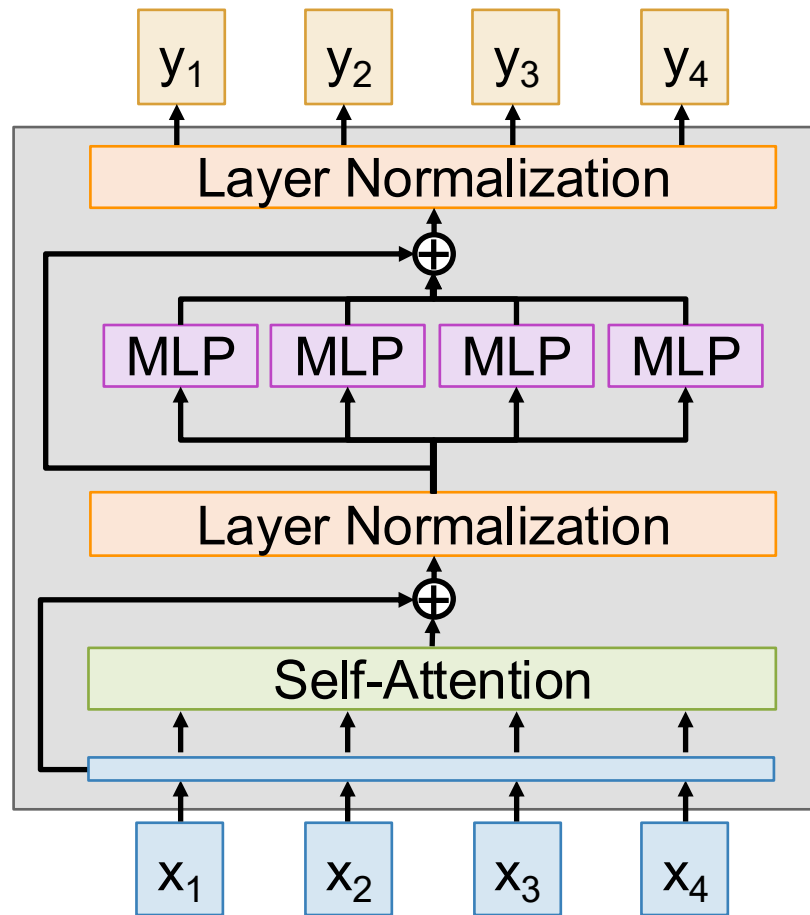
Don't use any  
masking; each  
image patch can  
look at all other  
image patches

Use positional  
encoding to tell  
the transformer  
the 2D position  
of each patch

# Tweaking Transformers

The Transformer architecture has not changed much since 2017.

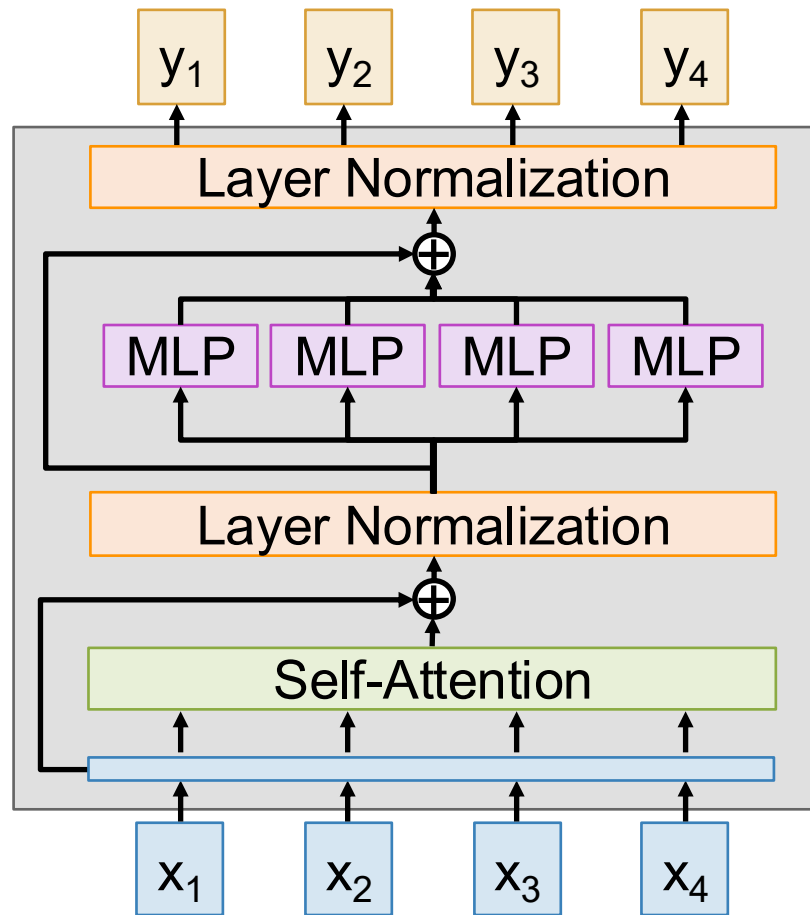
But a few changes have become common:



# Pre-Norm Transformer

**Layer normalization** is outside the residual connections

Kind of weird, the model can't actually learn the identity function

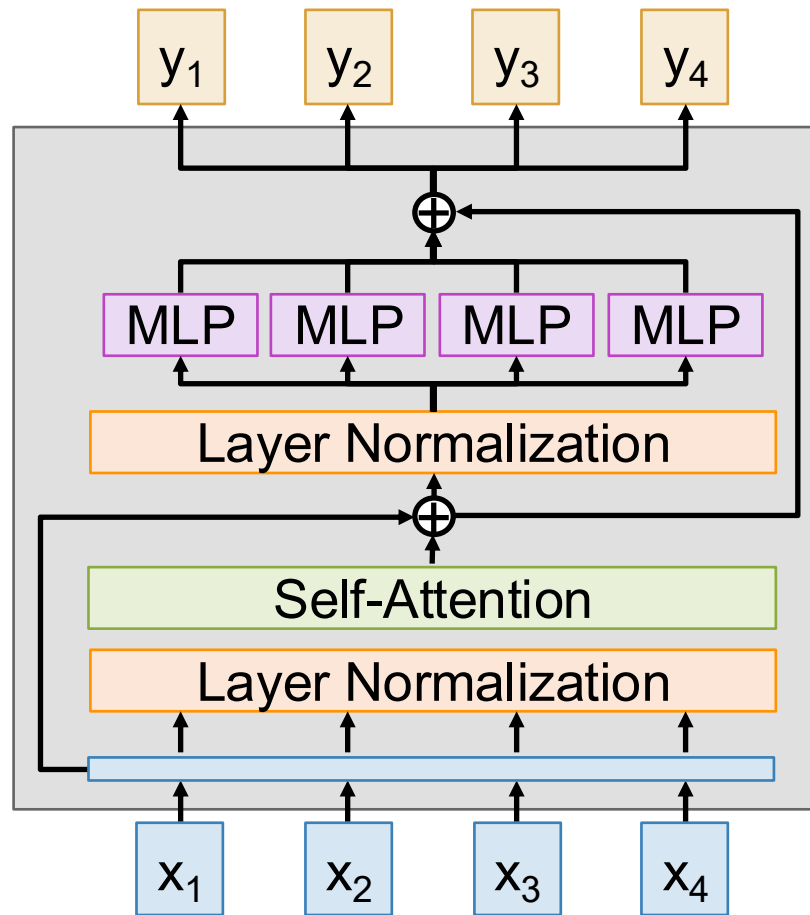


# Pre-Norm Transformer

**Layer normalization** is outside the residual connections

Kind of weird, the model can't actually learn the identity function

**Solution:** Move layer normalization before the Self-Attention and MLP, inside the residual connections. Training is more stable.



# QK-Norm

Normalize queries and keys before computing attention similarities

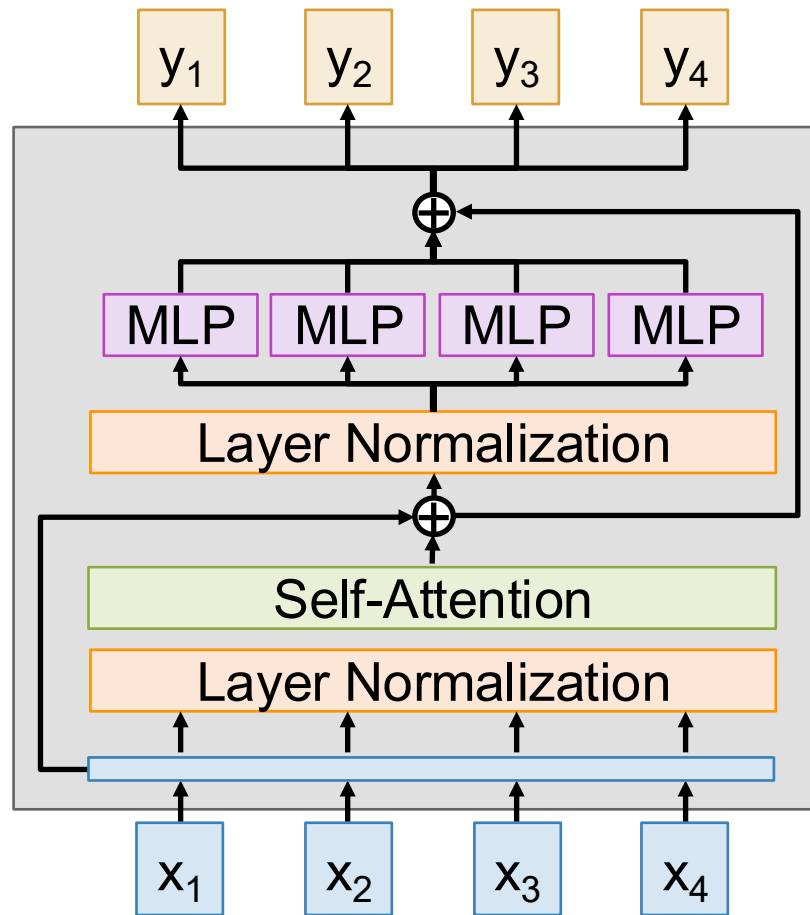
**Queries:**  $Q = \text{normalize}(XW_Q)$   $[H \times N \times D_H]$

**Keys:**  $K = \text{normalize}(XW_K)$   $[H \times N \times D_H]$

**Values:**  $V = XW_V$   $[H \times N \times D_H]$

**Similarities:**  $E = QK^T / \sqrt{D_Q}$   $[H \times N \times N]$

Prevents gradient spikes, stabilizes training



# QK-Norm

Normalize queries and keys before computing attention similarities

**Queries:**  $Q = \text{normalize}(XW_Q)$   $[H \times N \times D_H]$

**Keys:**  $K = \text{normalize}(XW_K)$   $[H \times N \times D_H]$

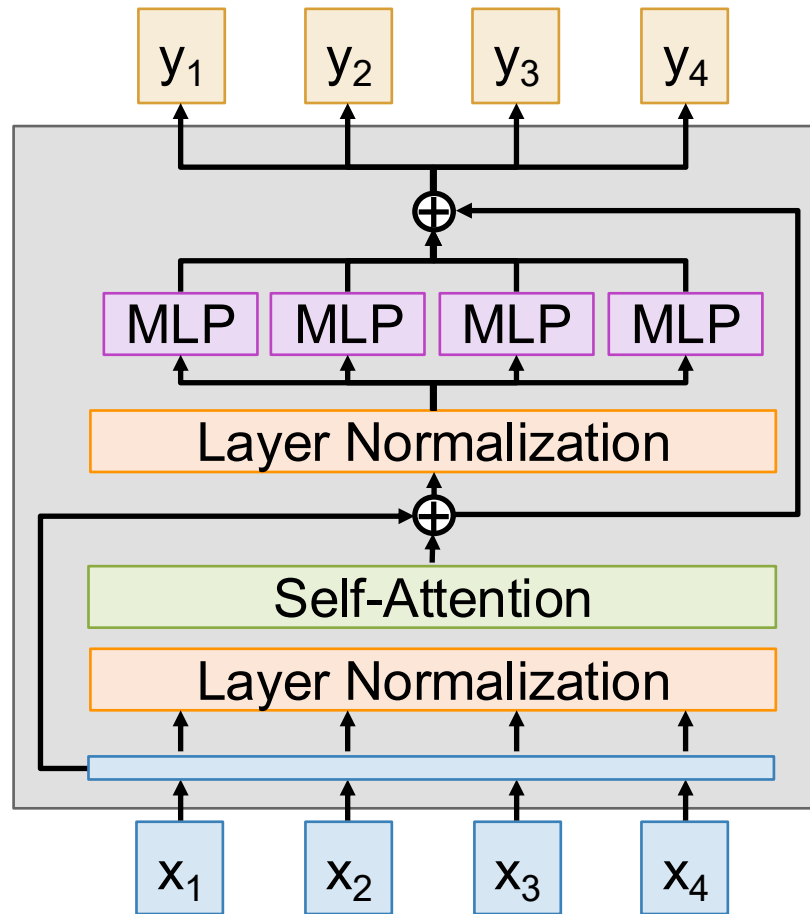
**Values:**  $V = XW_V$   $[H \times N \times D_H]$

**Similarities:**  $E = QK^T / \sqrt{D_Q}$   $[H \times N \times N]$

Prevents gradient spikes, stabilizes training

Normalize with RMSNorm:

$$y_i = \frac{x_i}{RMS(x)} * \gamma_i$$
$$RMSNorm(x) = \sqrt{\epsilon + \frac{1}{N} \sum_{i=1}^N x_i^2}$$



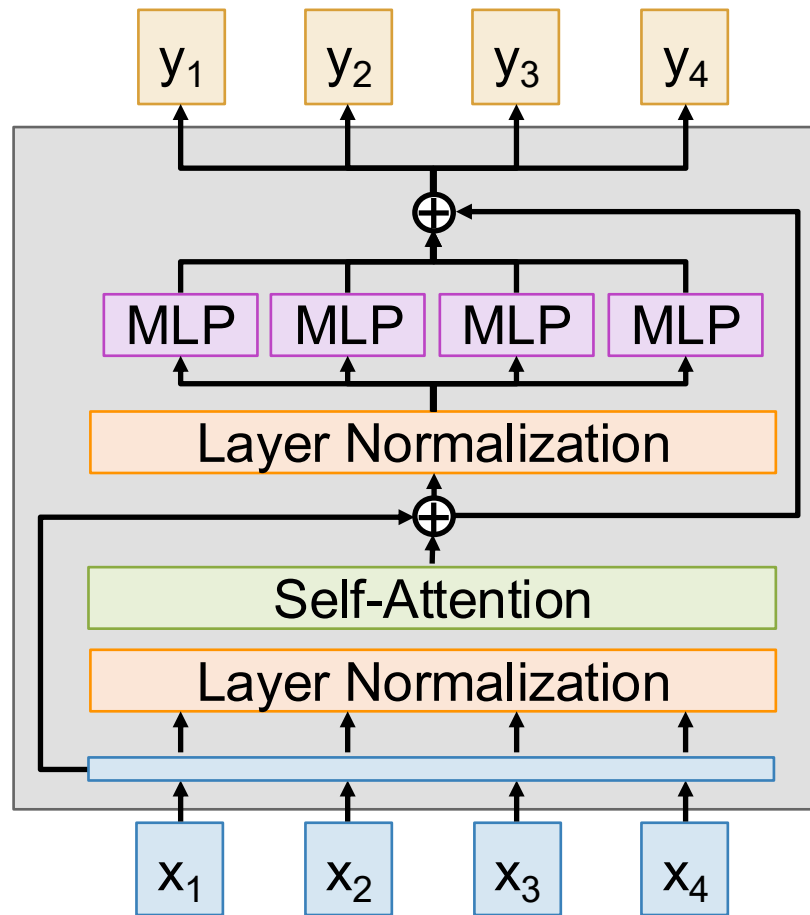
# SwiGLU MLP

## Classic MLP:

**Input:**  $X$  [ $N \times D$ ]

**Weights:**  $W_1$  [ $D \times 4D$ ]  
 $W_2$  [ $4D \times D$ ]

**Output:**  $Y = \sigma(XW_1)W_2$  [ $N \times D$ ]



# SwiGLU MLP

## Classic MLP:

**Input:**  $X [N \times D]$

**Weights:**  $W_1 [D \times 4D]$   
 $W_2 [4D \times D]$

**Output:**  $Y = \sigma(XW_1)W_2 [N \times D]$

## SwiGLU MLP:

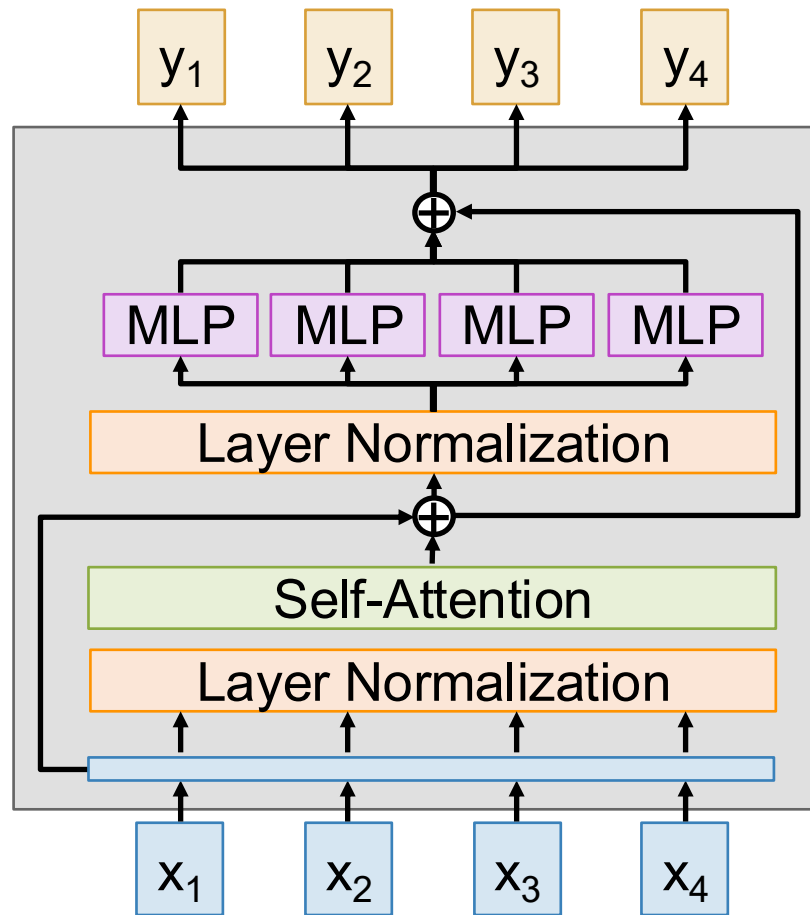
**Input:**  $X [N \times D]$

**Weights:**  $W_1, W_2 [D \times H]$   
 $W_3 [H \times D]$

**Output:**

$$Y = (\sigma(XW_1) \odot XW_2)W_3$$

Setting  $H = 8D/3$  keeps  
same total params



# SwiGLU MLP

## Classic MLP:

**Input:**  $X [N \times D]$

**Weights:**  $W_1 [D \times 4D]$   
 $W_2 [4D \times D]$

**Output:**  $Y = \sigma(XW_1)W_2 [N \times D]$

## SwiGLU MLP:

**Input:**  $X [N \times D]$

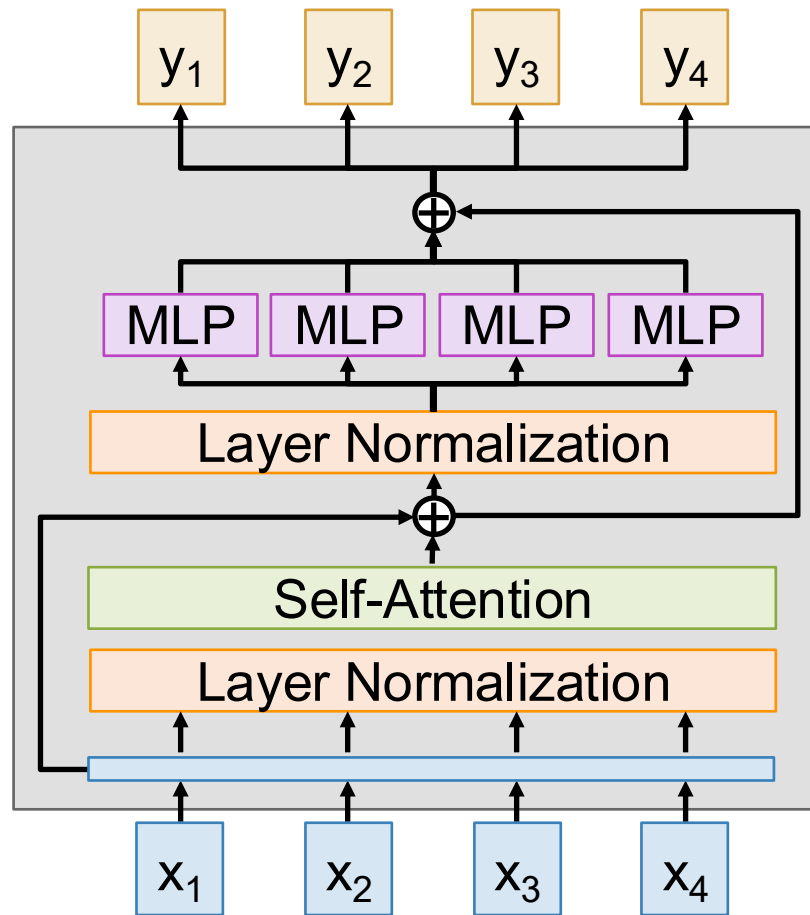
**Weights:**  $W_1, W_2 [D \times H]$   
 $W_3 [H \times D]$

**Output:**

$$Y = (\sigma(XW_1) \odot XW_2)W_3$$

Setting  $H = 8D/3$  keeps  
same total params

*We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.*

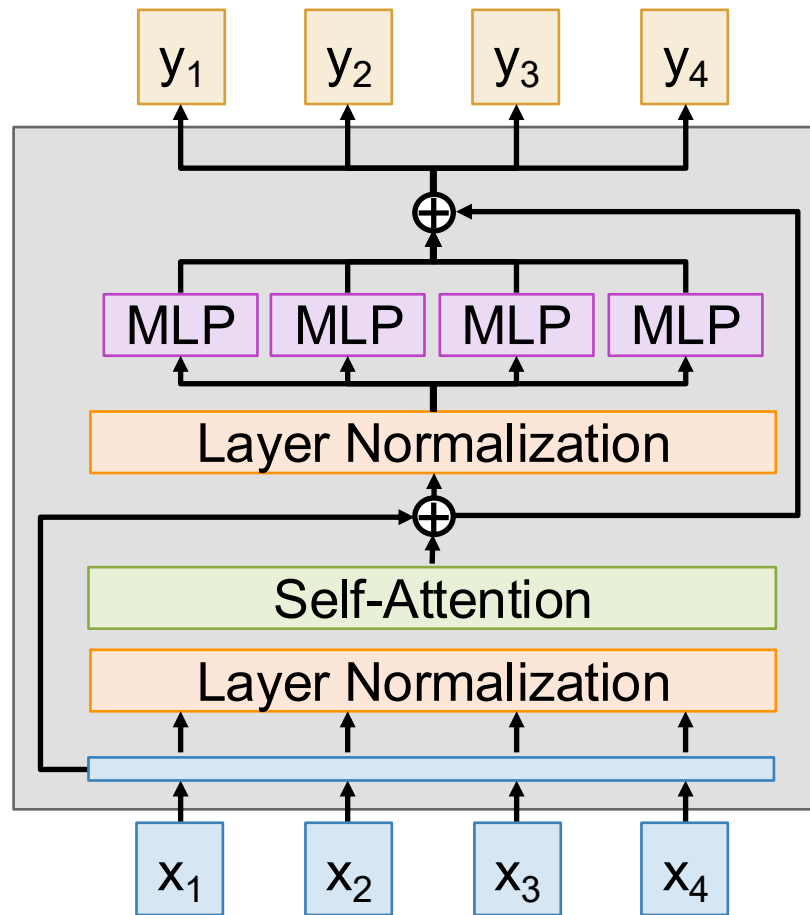


# Mixture of Experts (MoE)

Learn  $E$  separate sets of MLP weights in each block; each MLP is an *expert*

$$W_1: [D \times 4D] \Rightarrow [E \times D \times 4D]$$

$$W_2: [4D \times D] \Rightarrow [E \times 4D \times D]$$



# Mixture of Experts (MoE)

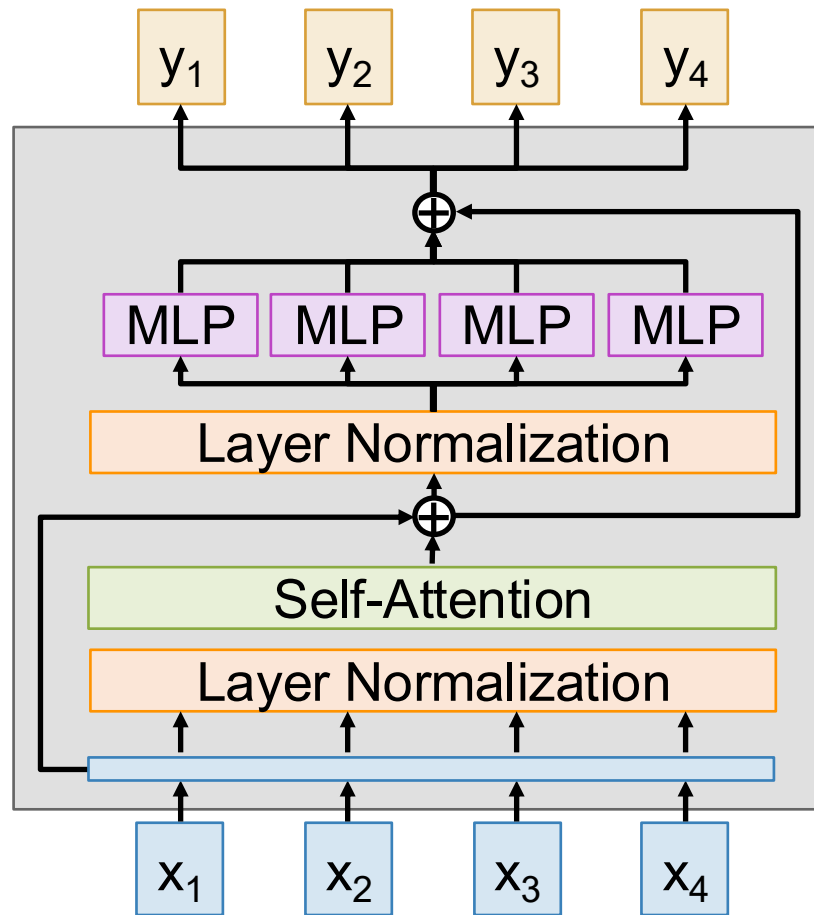
Learn  $E$  separate sets of MLP weights in each block; each MLP is an *expert*

$W_1: [D \times 4D] \Rightarrow [E \times D \times 4D]$

$W_2: [4D \times D] \Rightarrow [E \times 4D \times D]$

Each token gets *routed* to  $A < E$  of the experts. These are the *active experts*.

Increases params by  $E$ ,  
But only increases compute by  $A$



# Mixture of Experts (MoE)

Learn  $E$  separate sets of MLP weights in each block; each MLP is an *expert*

$W_1: [D \times 4D] \Rightarrow [E \times D \times 4D]$

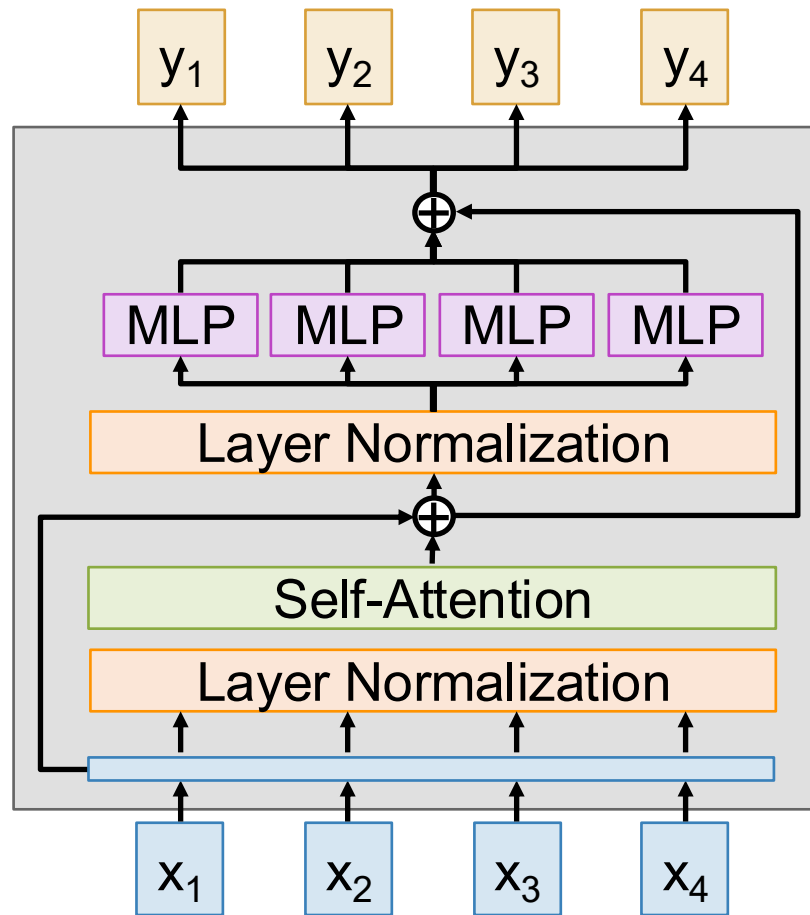
$W_2: [4D \times D] \Rightarrow [E \times 4D \times D]$

Each token gets *routed* to  $A < E$  of the experts. These are the *active experts*.

Increases params by  $E$ ,  
But only increases compute by  $A$

Example: Gemma4 26B-A4B (4/2/2026)

- 1 “shared expert” processes all tokens
- 128 “routed experts”; each token selects 8/128 to process it
- 26B total params, but each token only “activates” 4B params

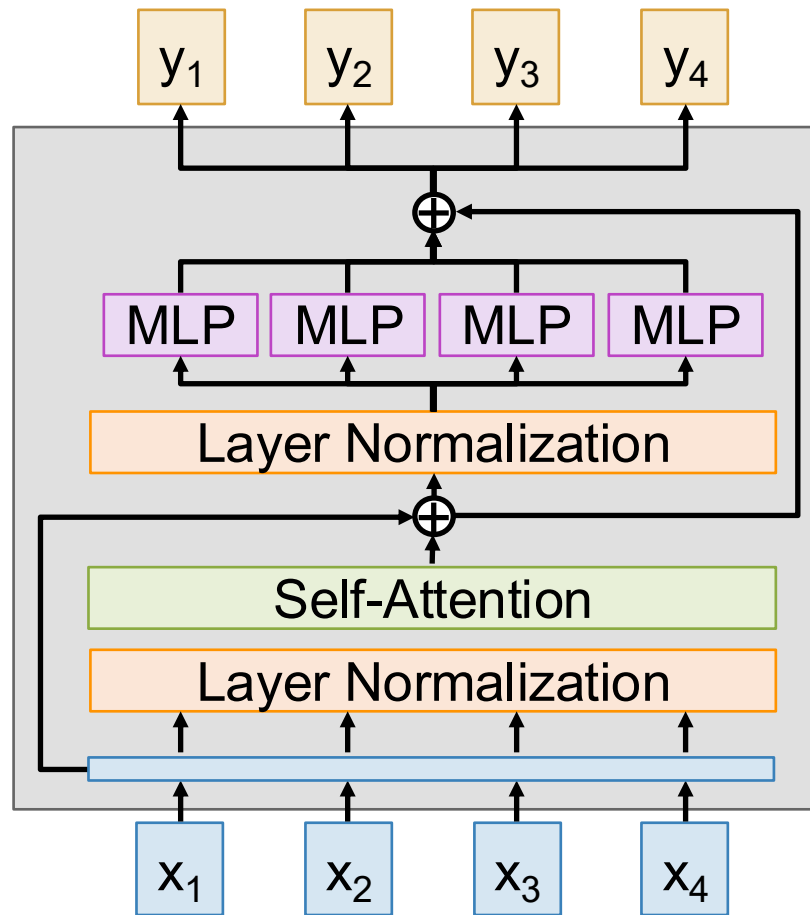


# Tweaking Transformers

The Transformer architecture has not changed much since 2017.

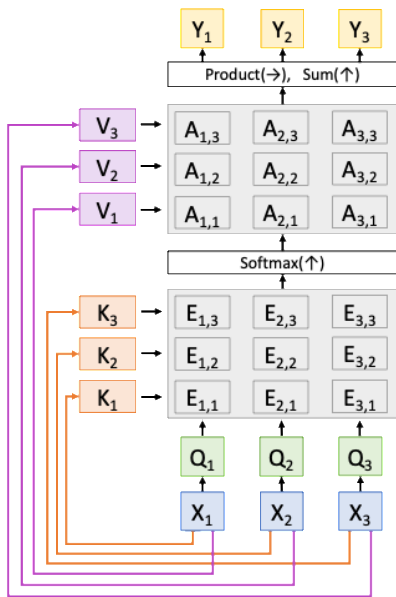
But a few changes have become common:

- **Pre-Norm**: Move normalization inside residual
- **QK-Norm**: Normalize keys and queries
- **SwiGLU**: Different MLP architecture
- **Mixture of Experts (MoE)**: Learn  $E$  different MLPs, use  $A < E$  of them per token. Massively increase params, modest increase to compute cost.



# Summary: Attention + Transformers

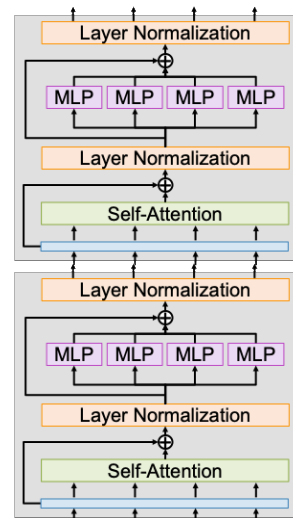
**Attention:** A new primitive that operates on sets of vectors



Transformers are the backbone of all large AI models today!

Used for language, vision, speech, ...

**Transformer:** A neural network architecture that uses attention everywhere



Next Time:  
Detection, Segmentation,  
Visualization