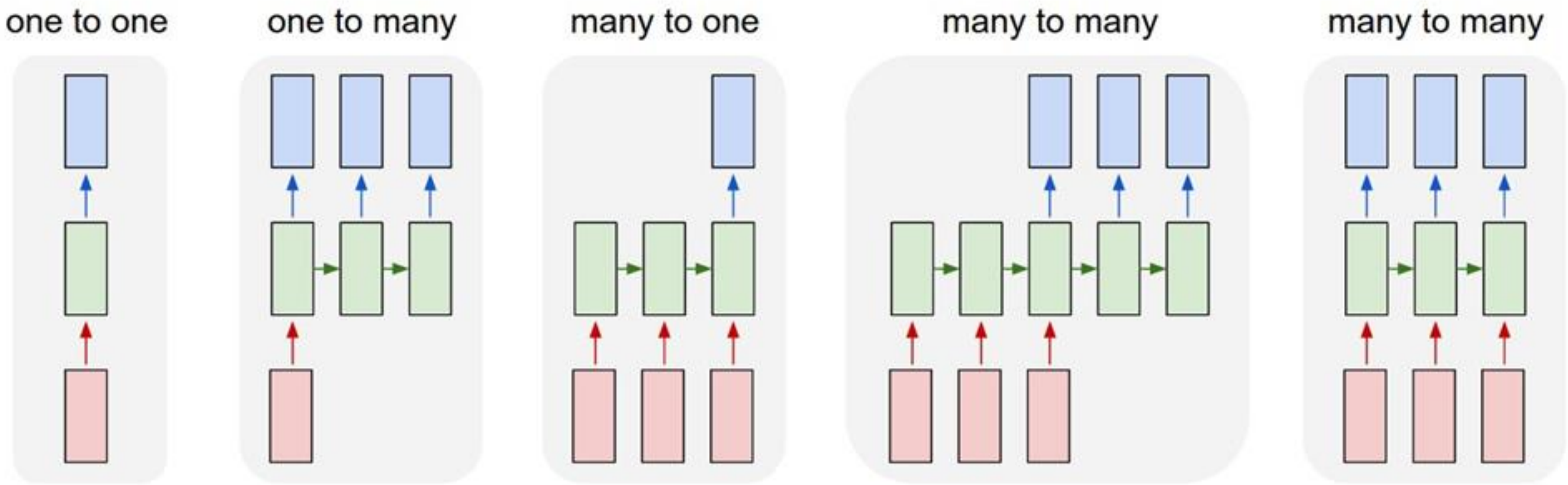
Lecture 8:
Attention and Transformers

Administrative

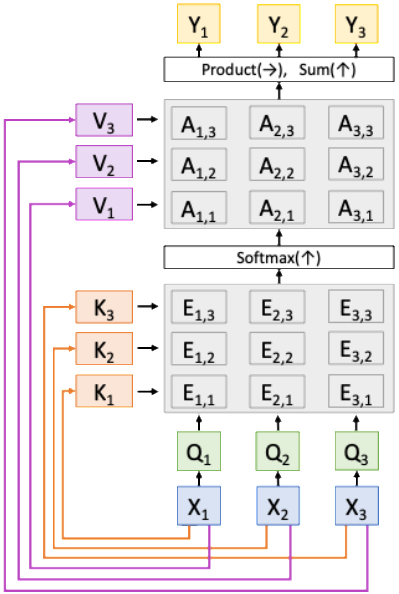
- Today: Project proposal due
- Today: Assignment 2 out, due 5/7
- Tomorrow 4/24: PyTorch review session

Last Time: Recurrent Neural Networks

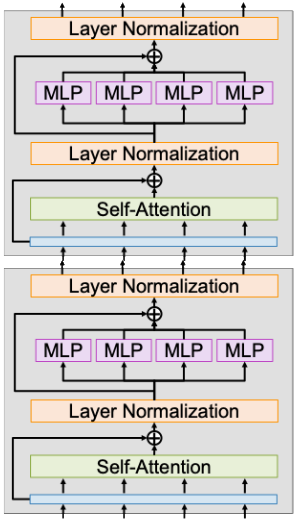


Today: Attention + Transformers

Attention: A new primitive that operates on sets of vectors

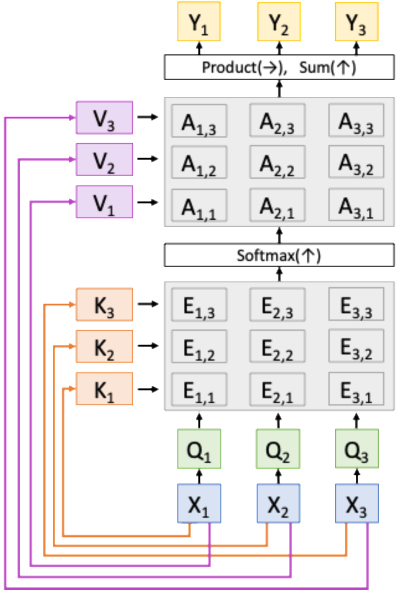


Transformer: A neural network architecture that uses attention everywhere

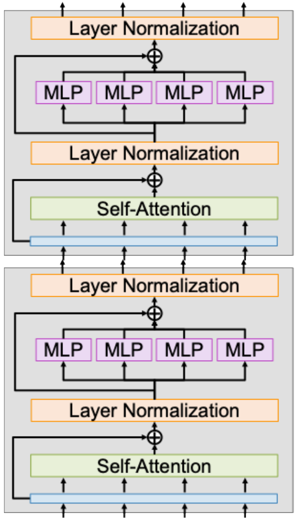


Today: Attention + Transformers

Attention: A new primitive that operates on sets of vectors



Transformer: A neural network architecture that uses attention everywhere



Transformers are used everywhere today!

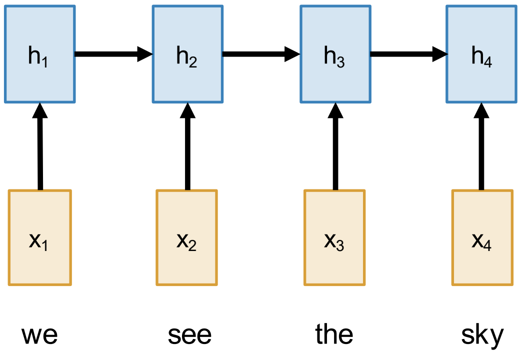
But they developed as an offshoot of RNNs so let's start there

Sequence to Sequence with RNNs: Encoder - Decoder

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

A motivating example for today's discussion – machine translation! English → Italian

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



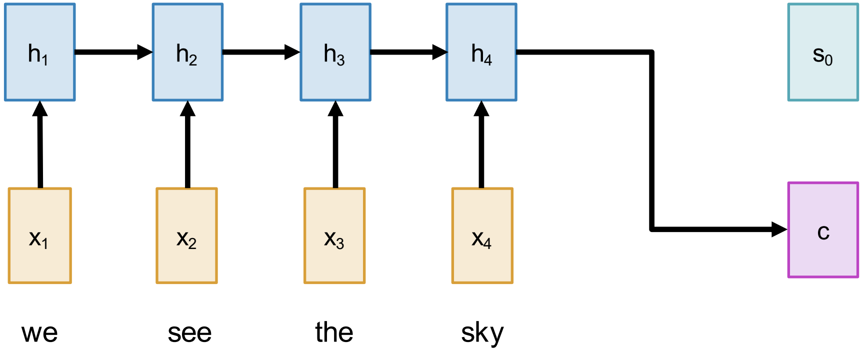
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

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

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



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

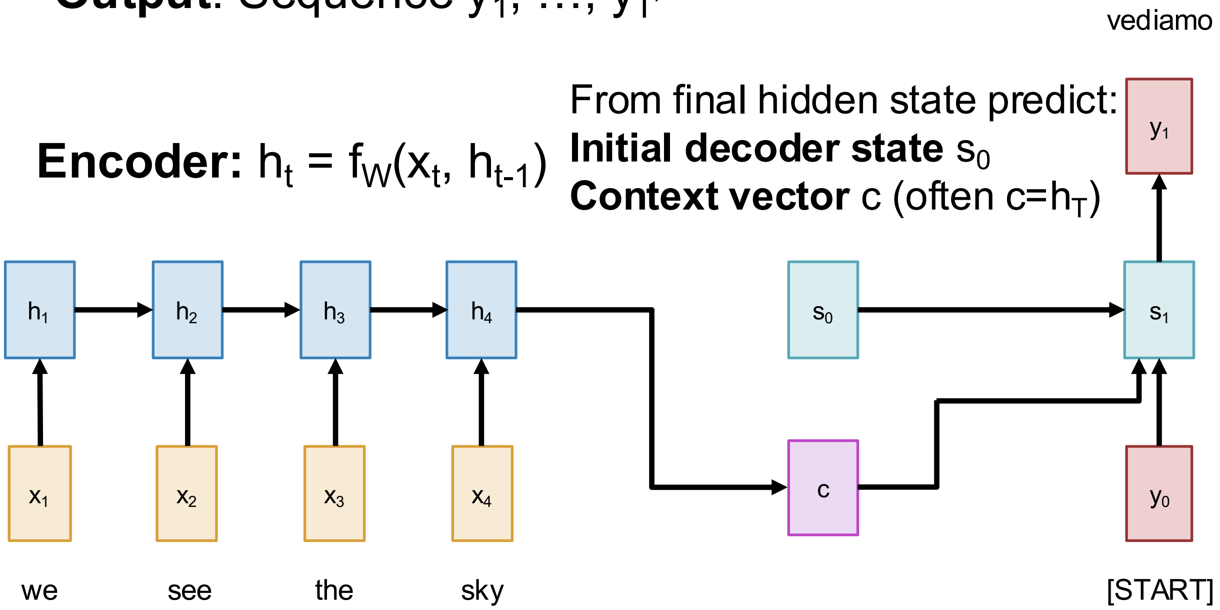
Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

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

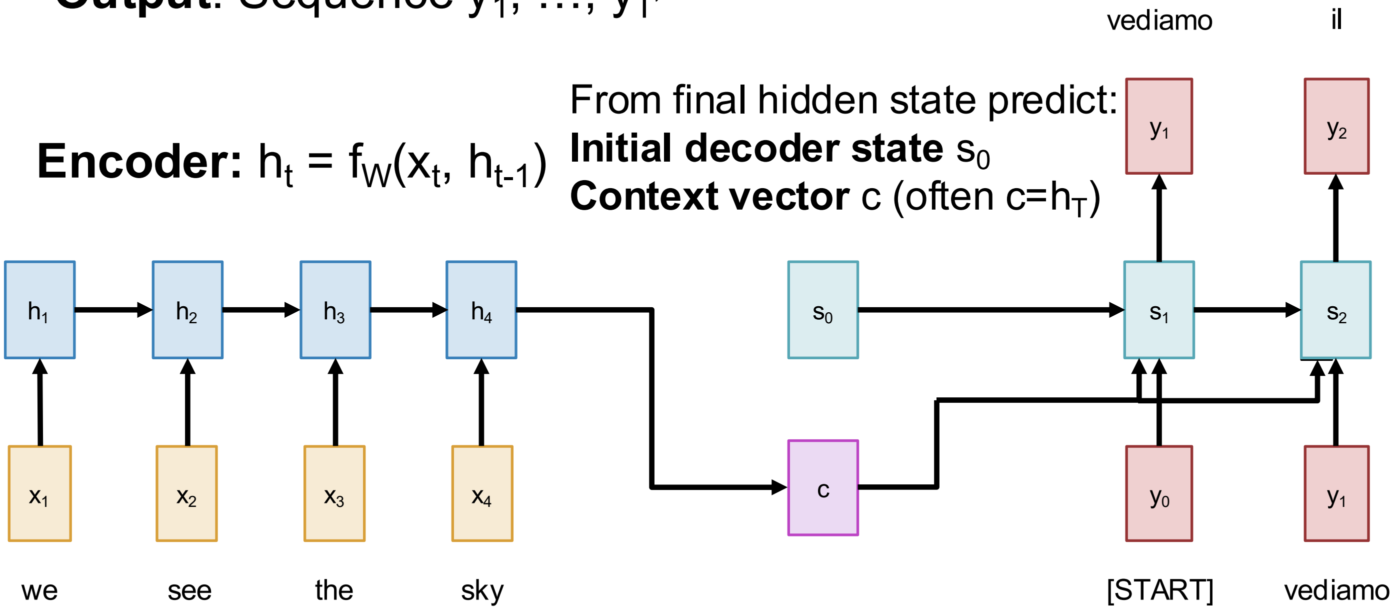
Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

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

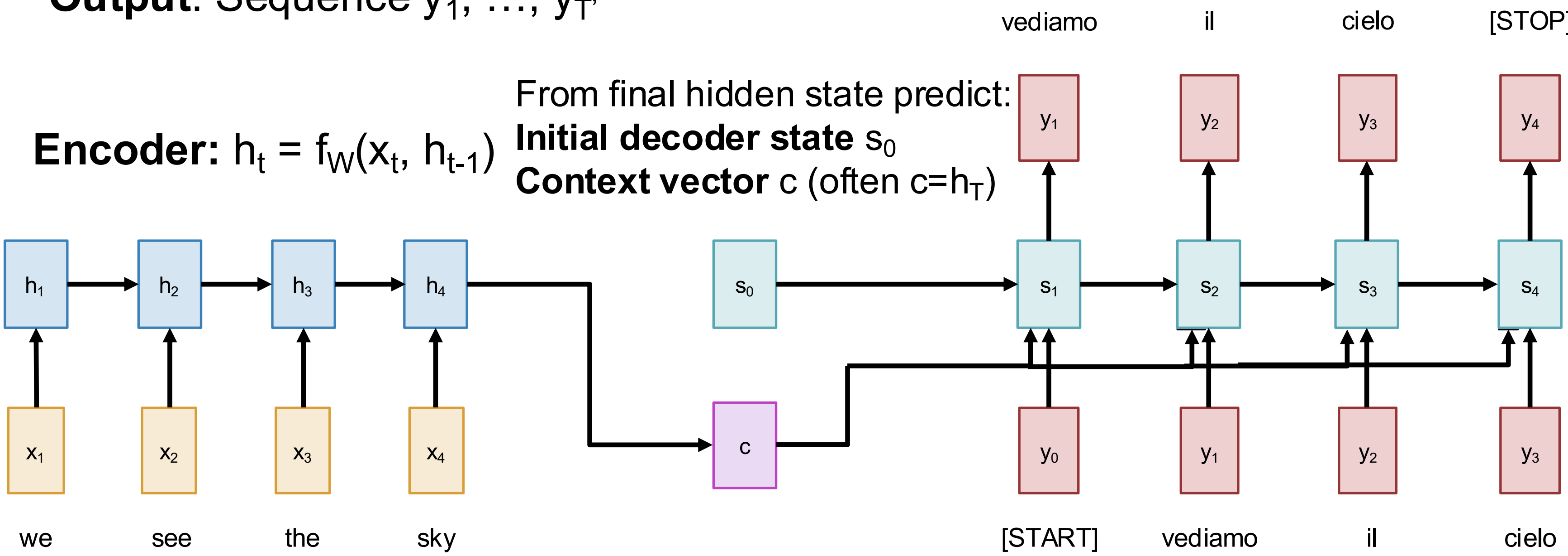
Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

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

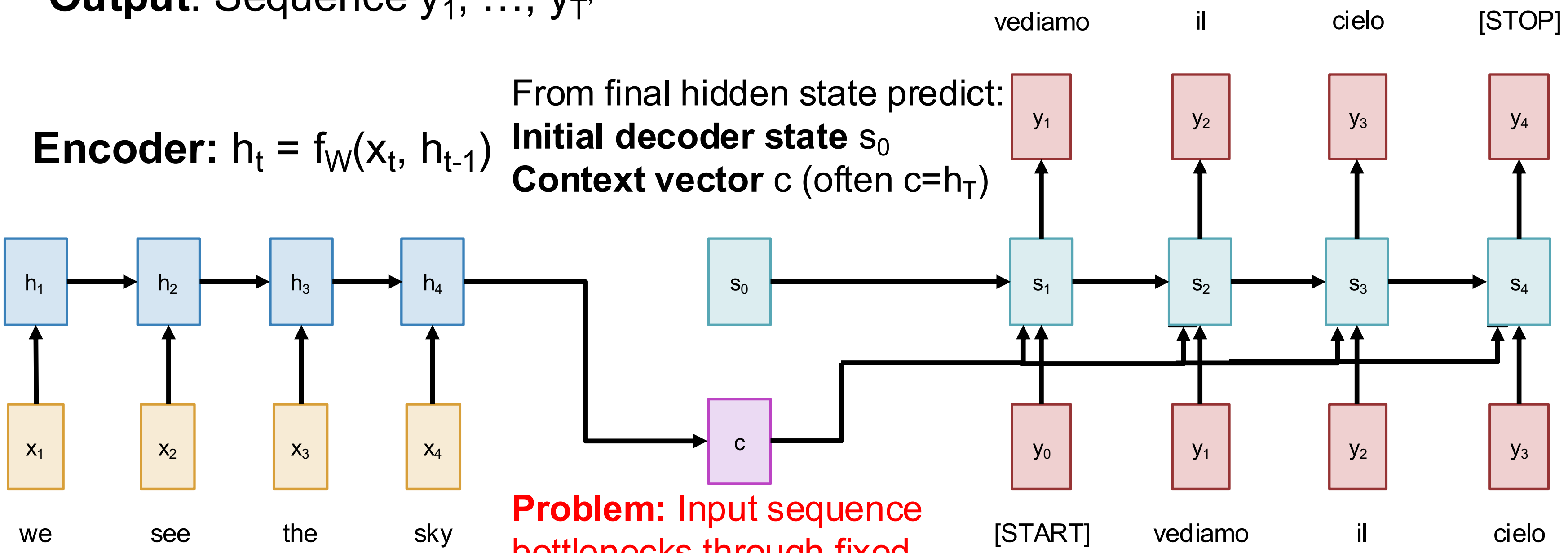
Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

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

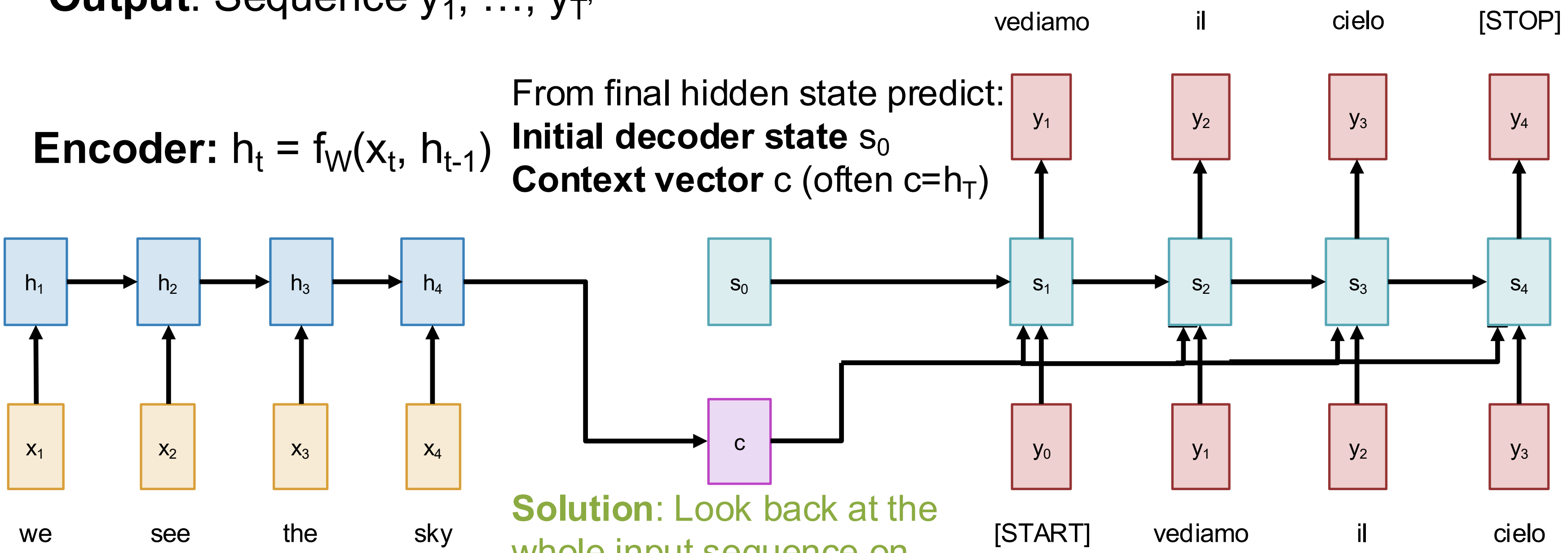
Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

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

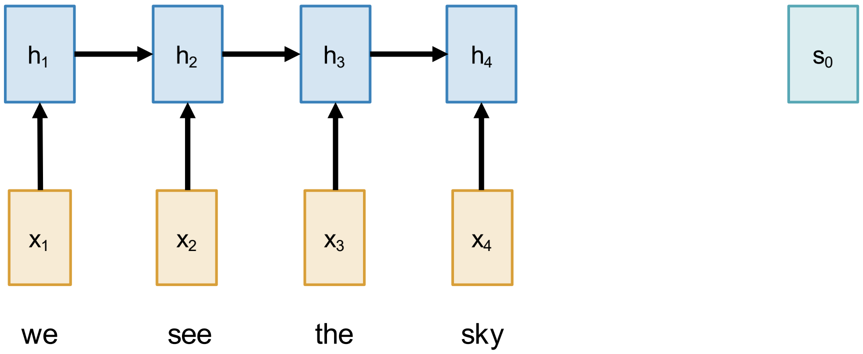
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs and Attention

Input: Sequence $x_1, \ldots x_T$
Output: Sequence $y_1, \ldots, y_{T'}$

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

From final hidden state:
Initial decoder state s_0

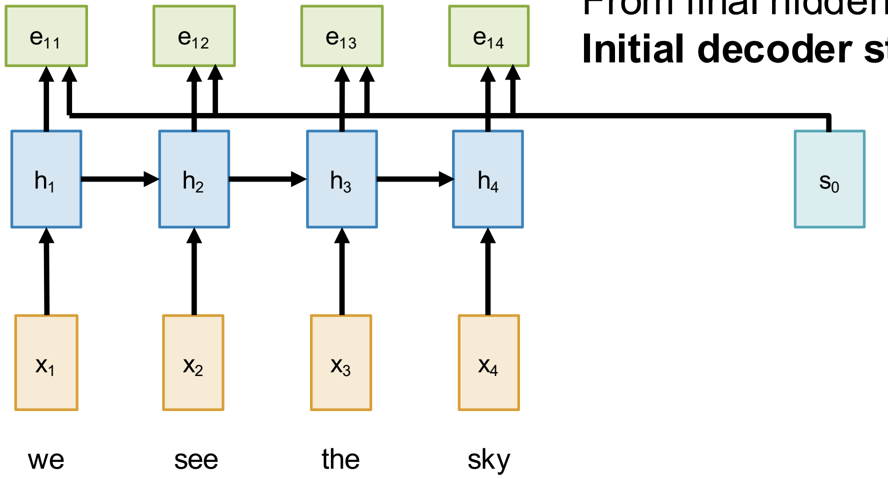


Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Sequence to Sequence with RNNs and Attention

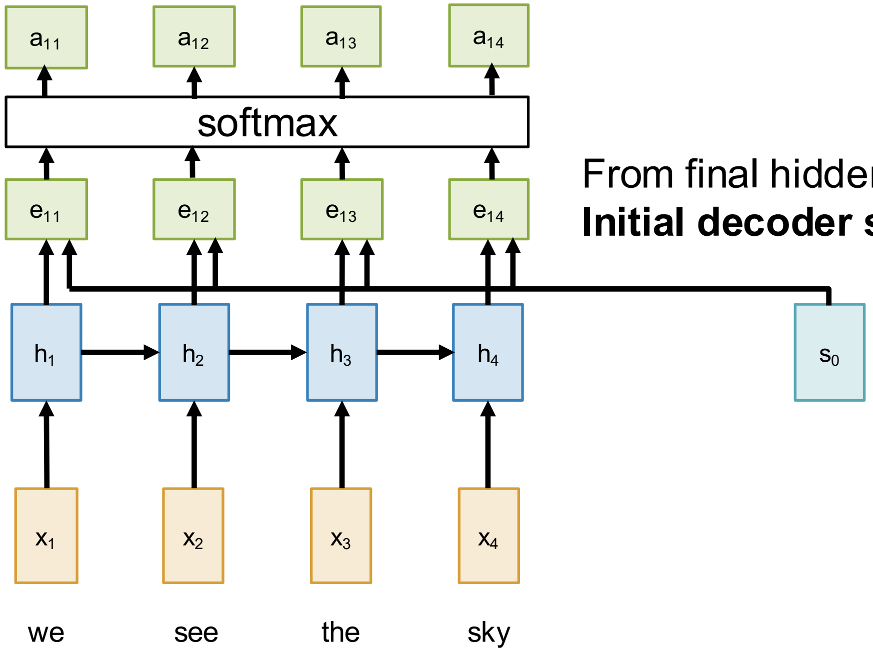
Compute (scalar) **alignment scores**
$e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)

From final hidden state:
Initial decoder state s_0



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

Sequence to Sequence with RNNs and Attention



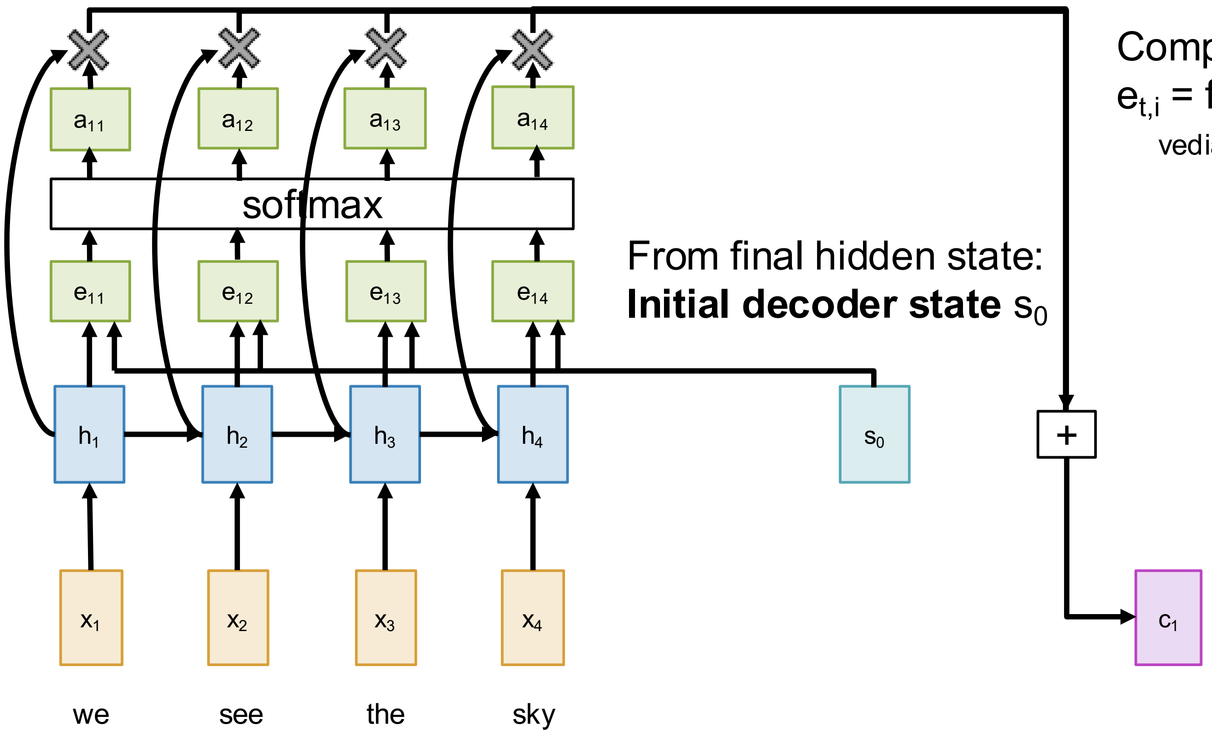
From final hidden state:
Initial decoder state s_0

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

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

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

Sequence to Sequence with RNNs and Attention



From final hidden state:
Initial decoder state s_0

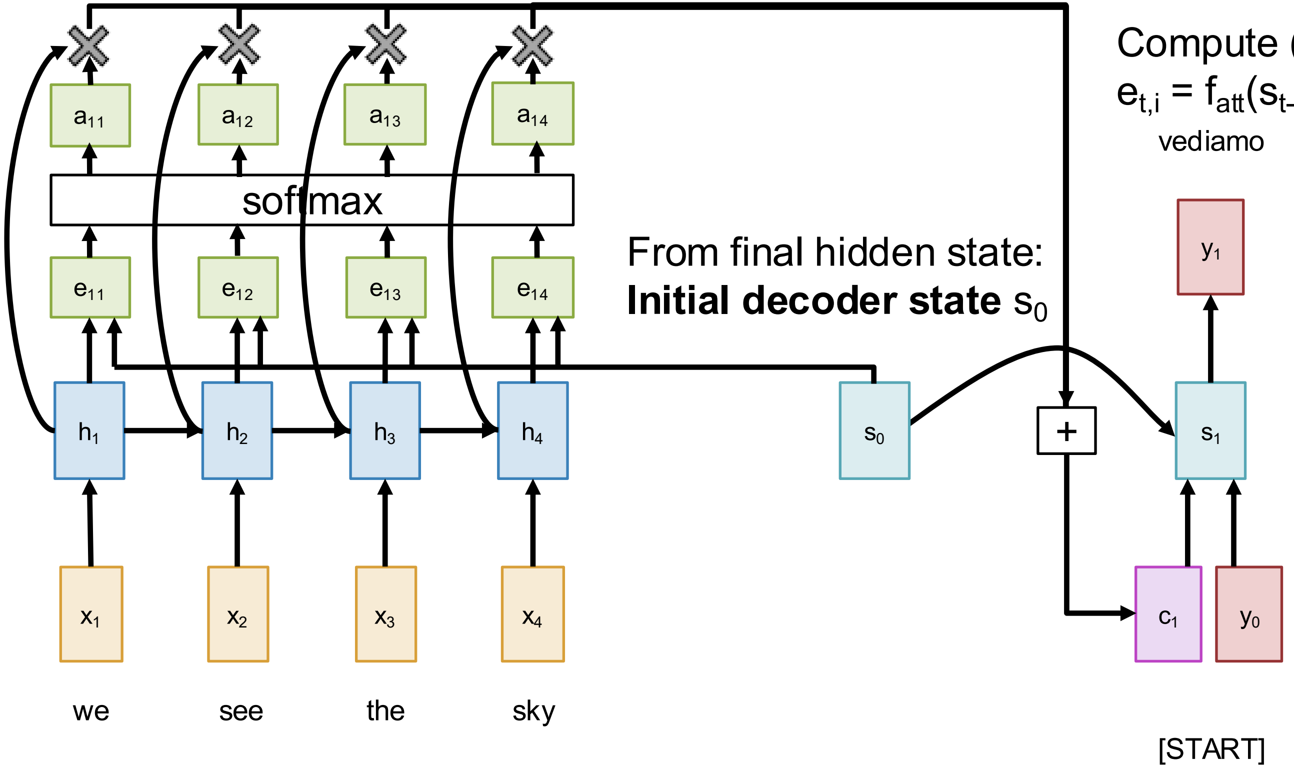
Compute (scalar) **alignment scores**
$e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)
vediamo

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**
$c_t = \sum_i a_{t,i} h_i$

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

Sequence to Sequence with RNNs and Attention



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

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

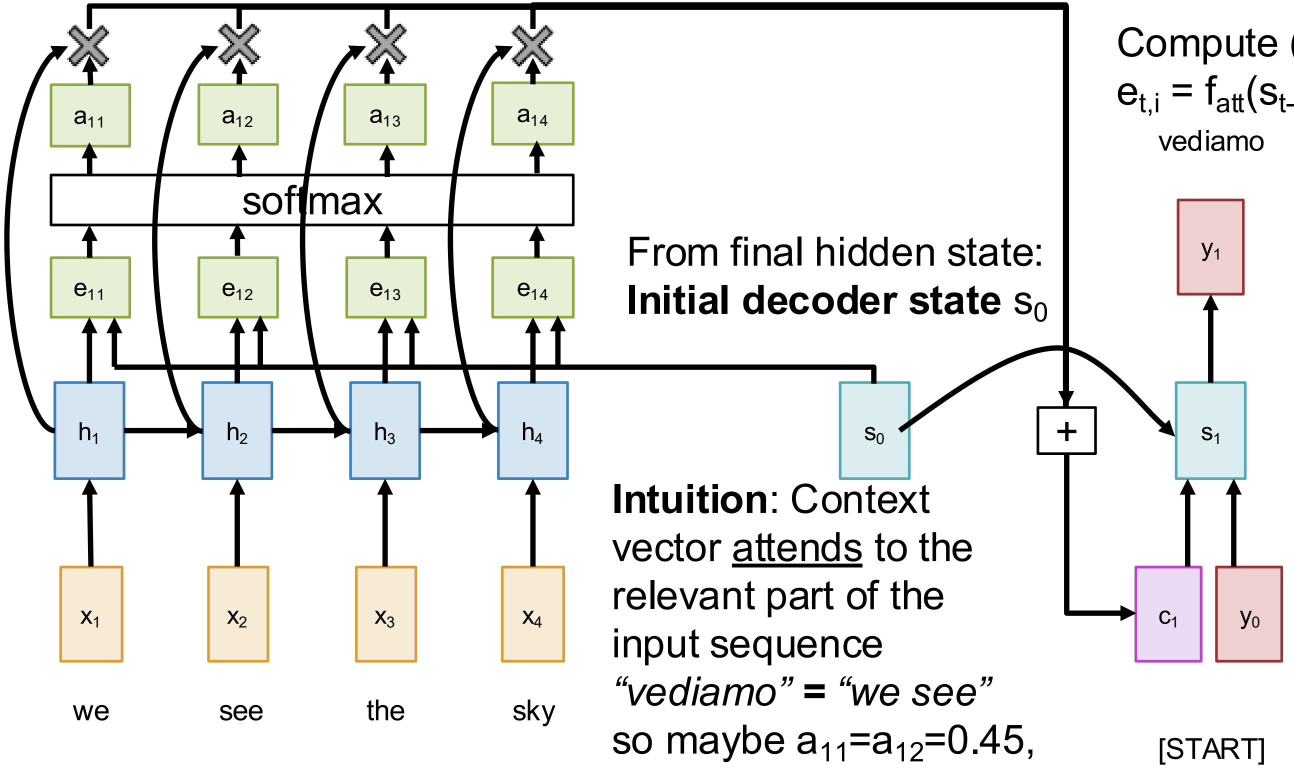
Compute **context vector** as weighted sum of hidden states
$c_t = \sum_i a_{t,i} h_i$

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

g_U is an RNN unit (e.g. LSTM, GRU)

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

Sequence to Sequence with RNNs and Attention



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

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

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

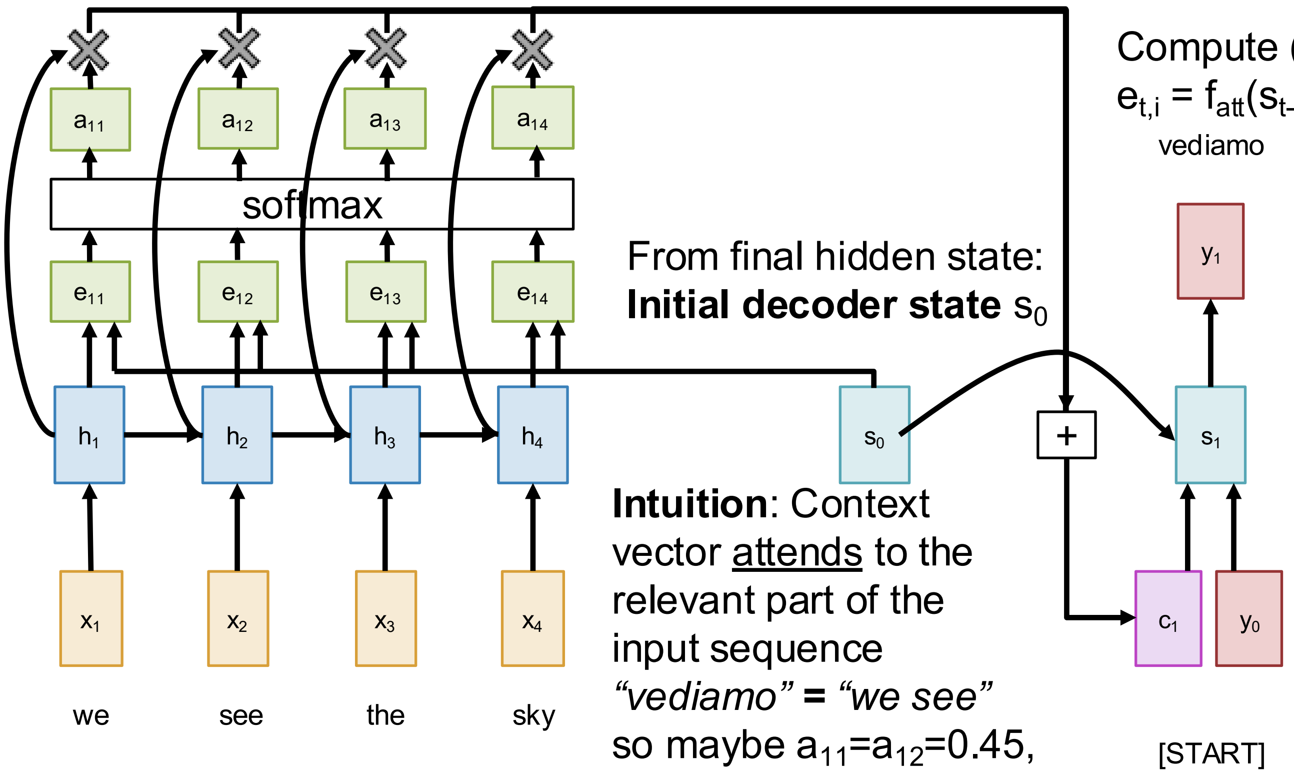
Use context vector in decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c_t)$

From final hidden state: **Initial decoder state** s_0

Intuition: Context vector attends to the relevant part of the input sequence
"vediamo" **=** *"we see"*
so maybe $a_{11}=a_{12}=0.45$, $a_{13}=a_{14}=0.05$

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

Sequence to Sequence with RNNs and Attention



From final hidden state: **Initial decoder state s_0**

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

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

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

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

Intuition: Context vector <u>attends</u> to the relevant part of the input sequence
"vediamo" **=** *"we see"*
so maybe a_{11}=a_{12}=0.45, a_{13}=a_{14}=0.05

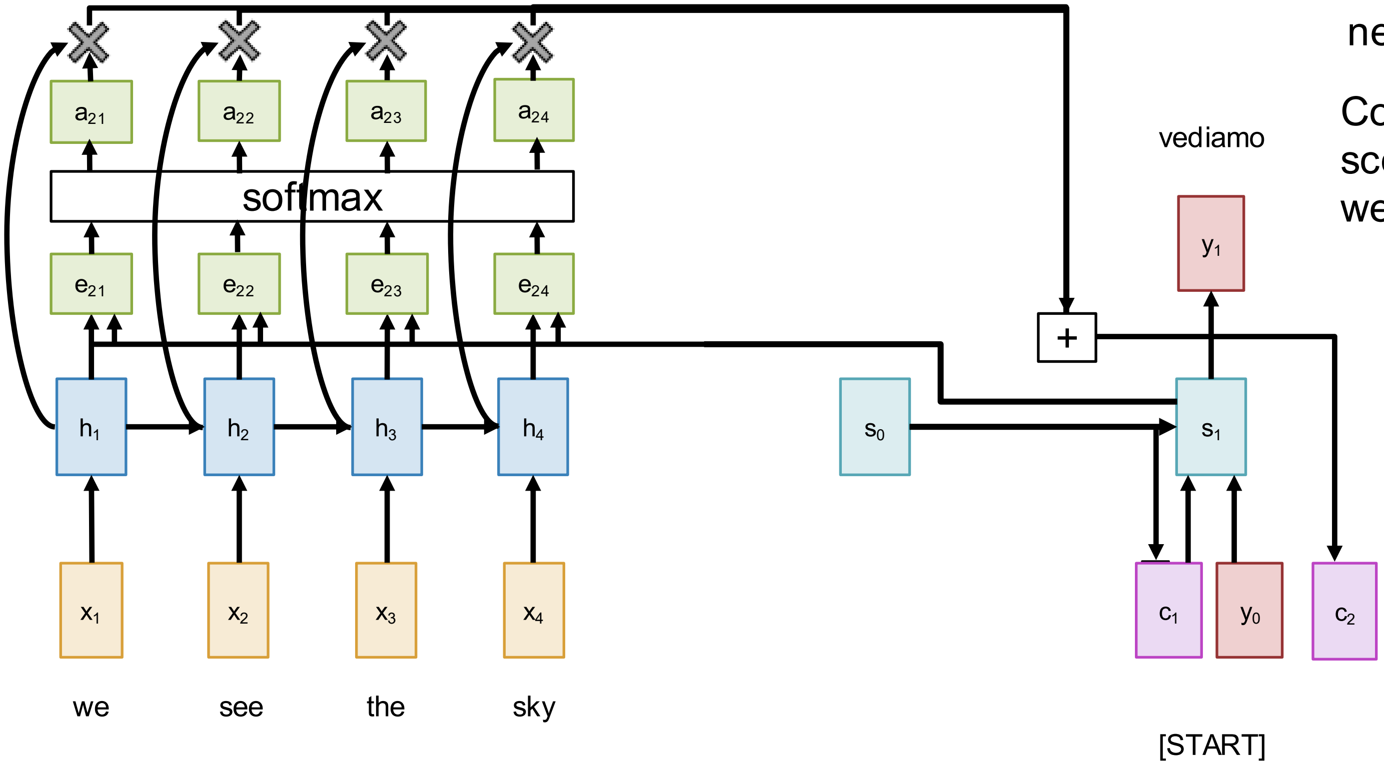
All differentiable! No supervision on attention weights. Backprop through everything

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

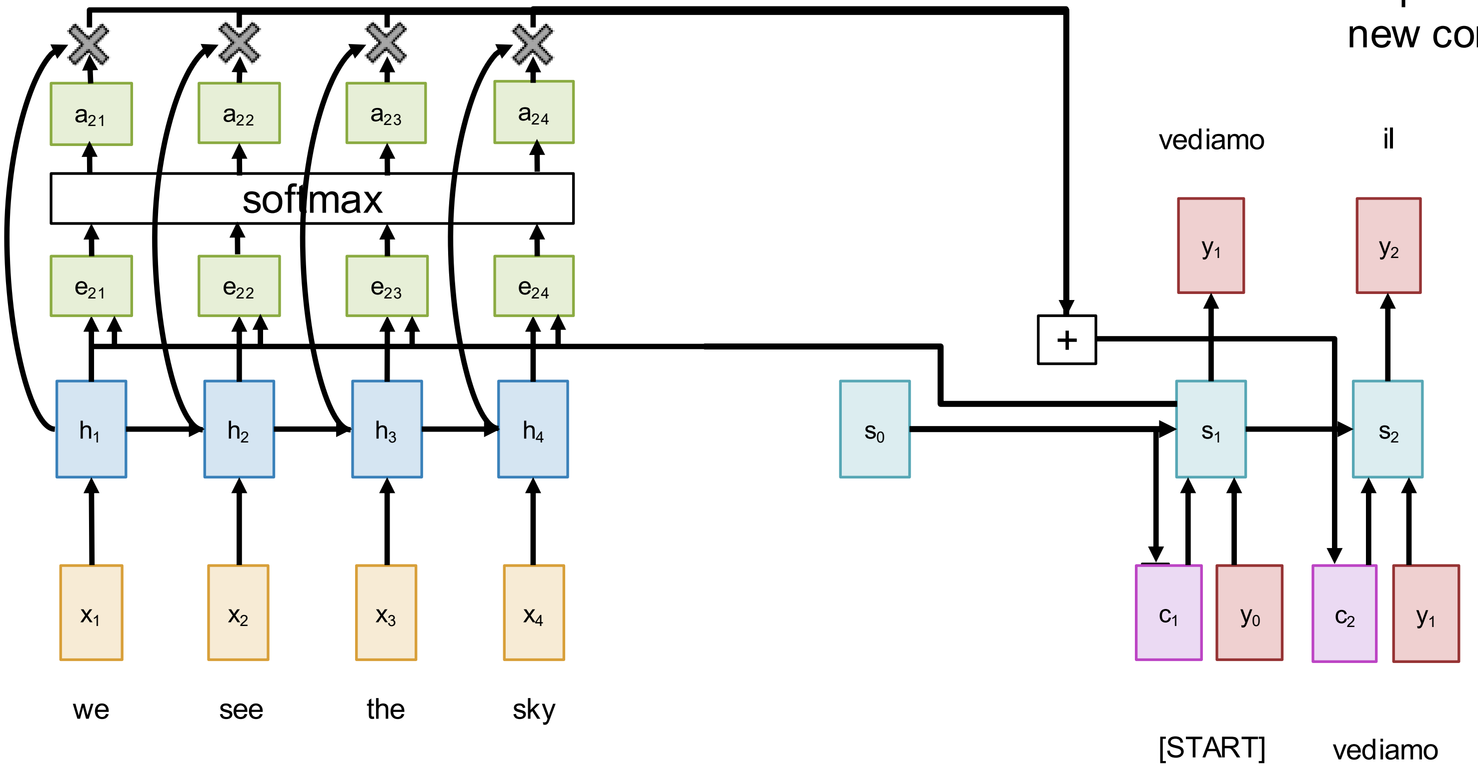
Compute new alignment scores $e_{2,i}$ and attention weights $a_{2,i}$



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

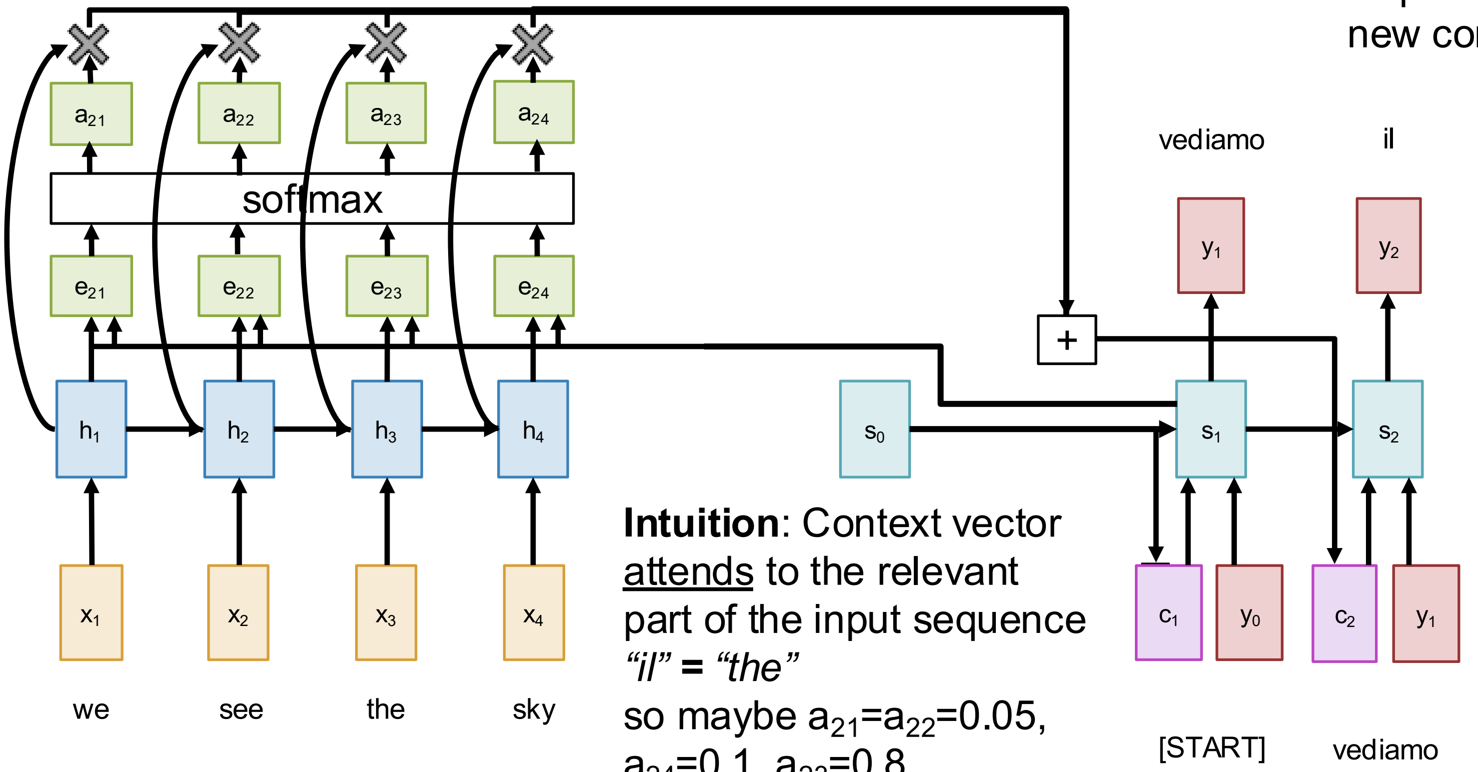


Use context vector in decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c_t)$

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

Sequence to Sequence with RNNs and Attention

Repeat: Use s_1 to compute new context vector c_2



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

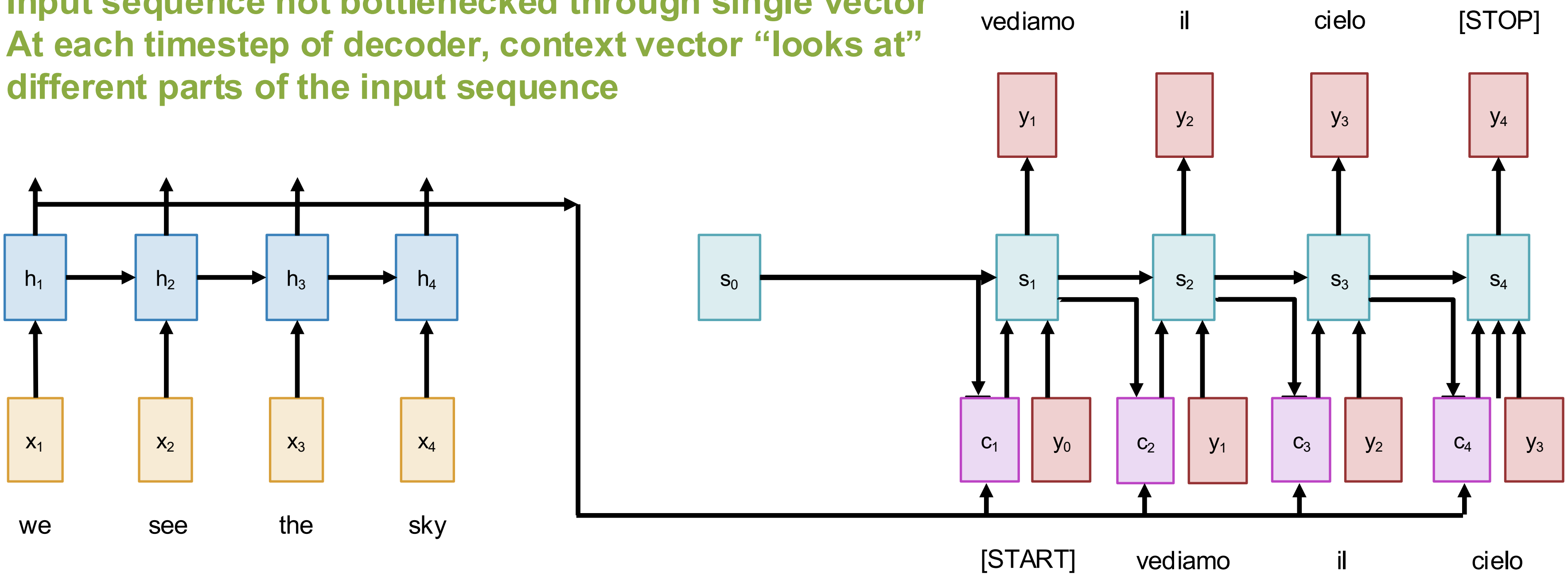
Intuition: Context vector attends to the relevant part of the input sequence
"il" = "the"
so maybe $a_{21}=a_{22}=0.05$, $a_{24}=0.1$, $a_{23}=0.8$

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

Sequence to Sequence with RNNs and Attention

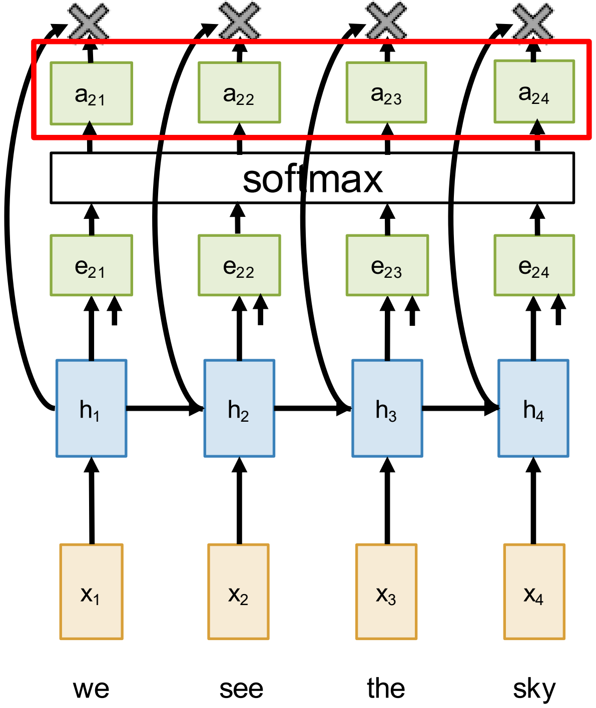
Use a different context vector in each timestep of decoder

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



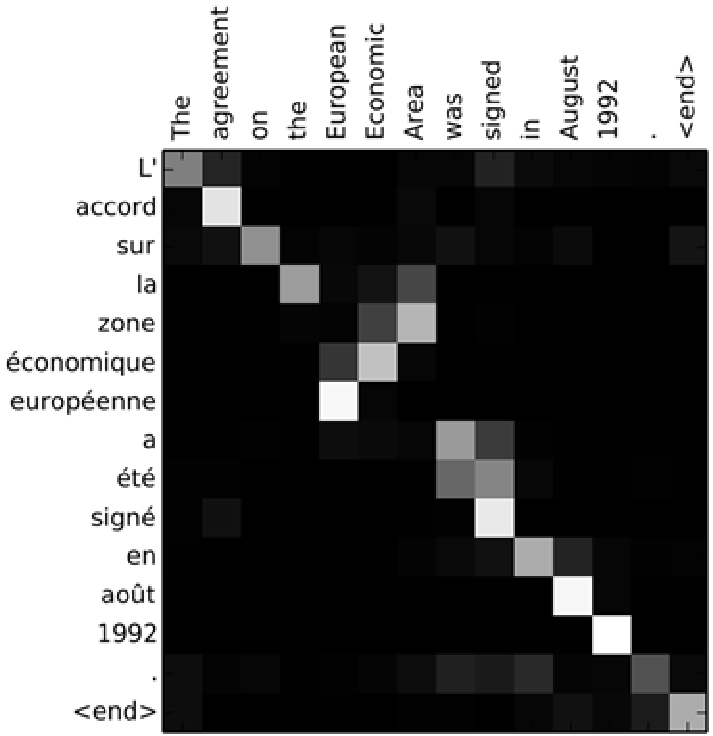
Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Sequence to Sequence with RNNs and Attention



Example: English to French translation

Visualize attention weights $a_{t,i}$



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

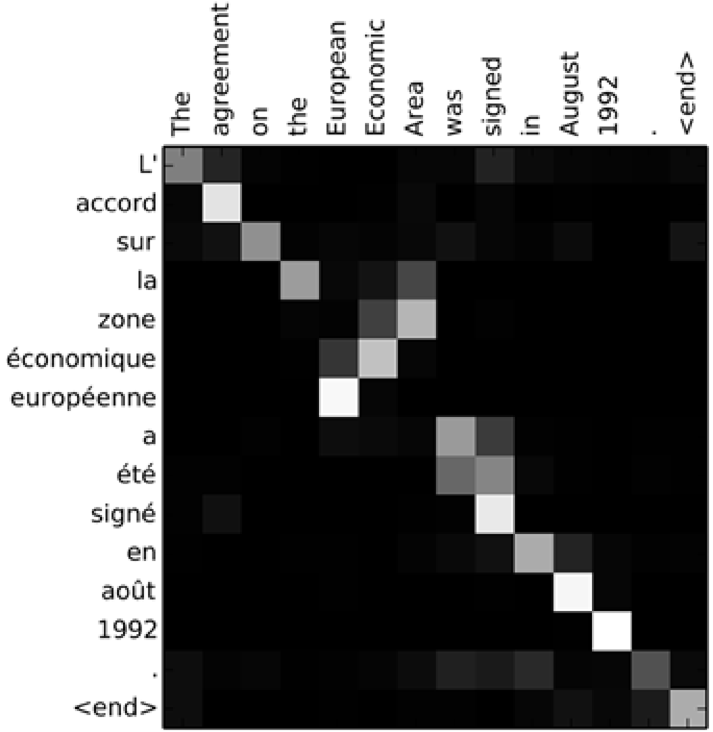
Sequence to Sequence with RNNs and Attention

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 $a_{t,i}$



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

Sequence to Sequence with RNNs and Attention

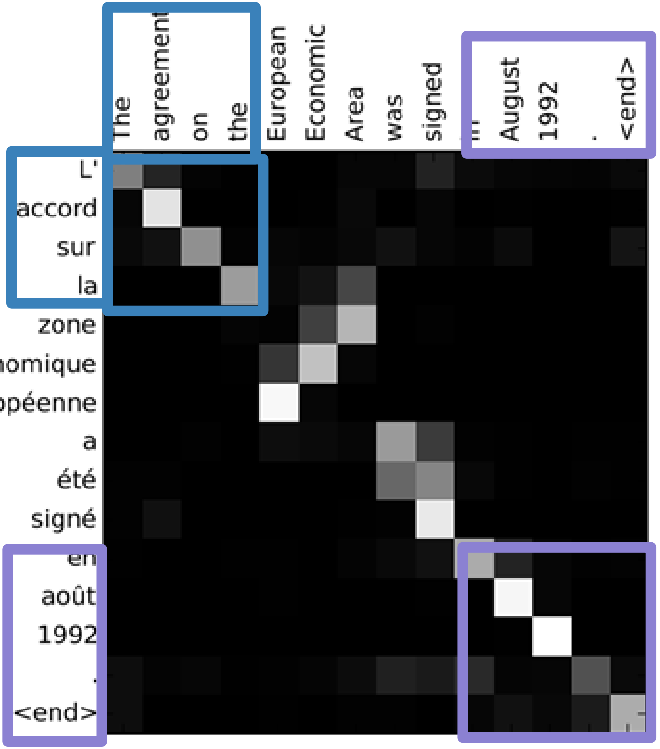
Example: English to French translation

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

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

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."

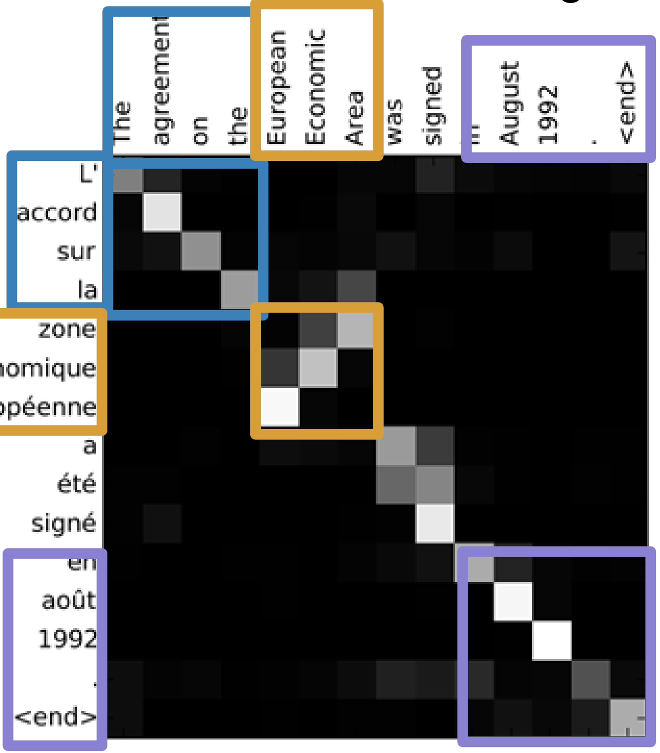
Output: "L'accord sur la zone économique européenne a été signé en août 1992."

Visualize attention weights $a_{t,i}$

Diagonal attention means words correspond in order

Attention figures out other word orders

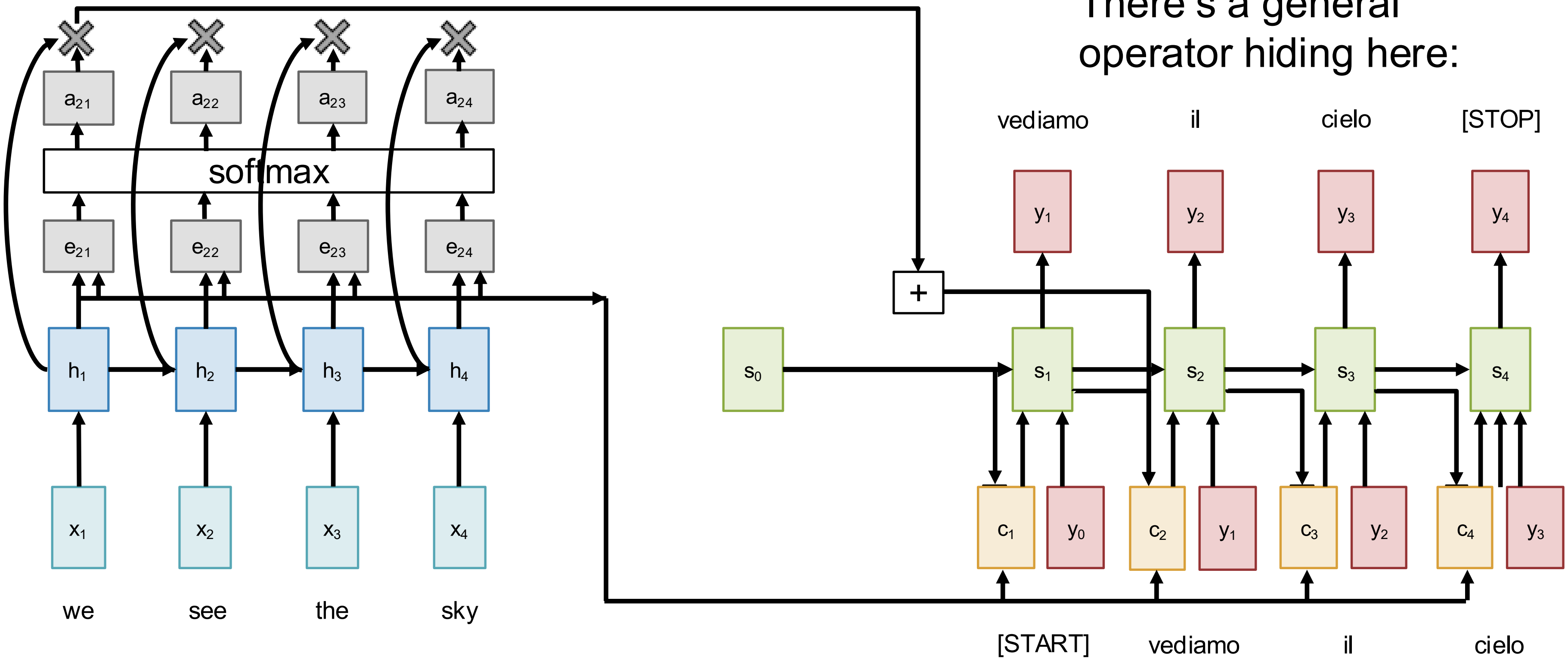
Diagonal attention means words correspond in order



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

Sequence to Sequence with RNNs and Attention

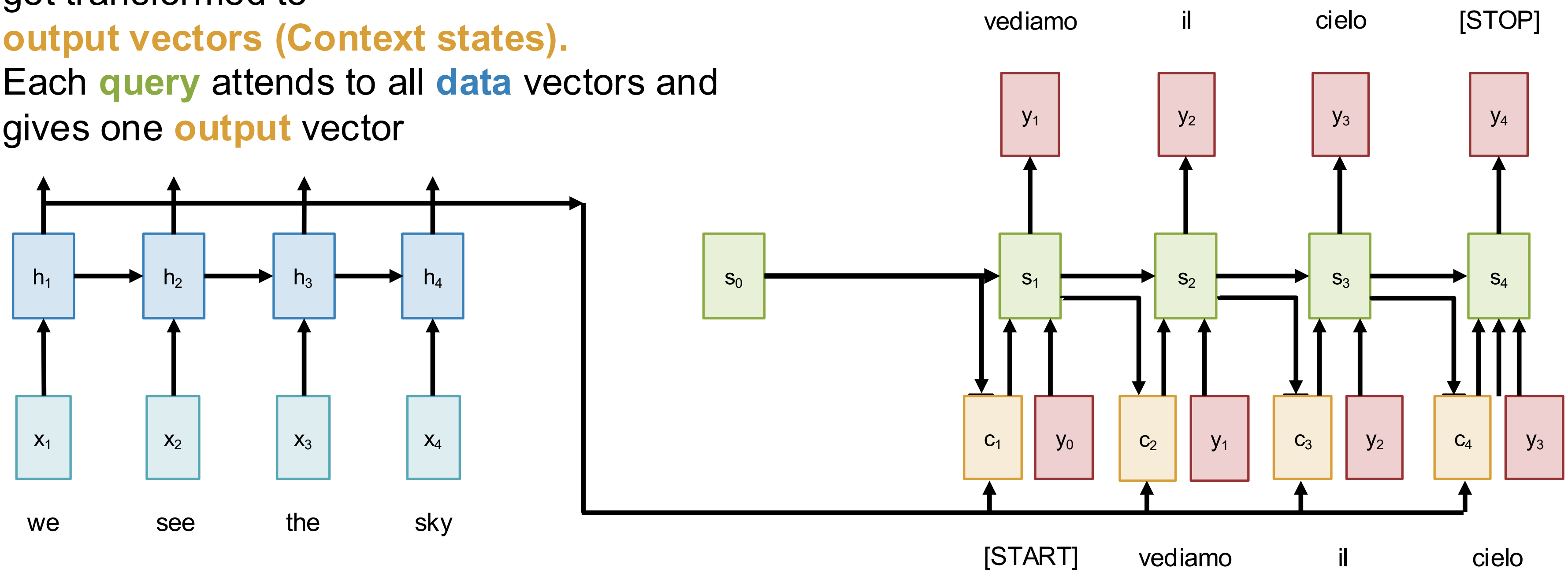
There's a general operator hiding here:



Sequence to Sequence with RNNs and Attention

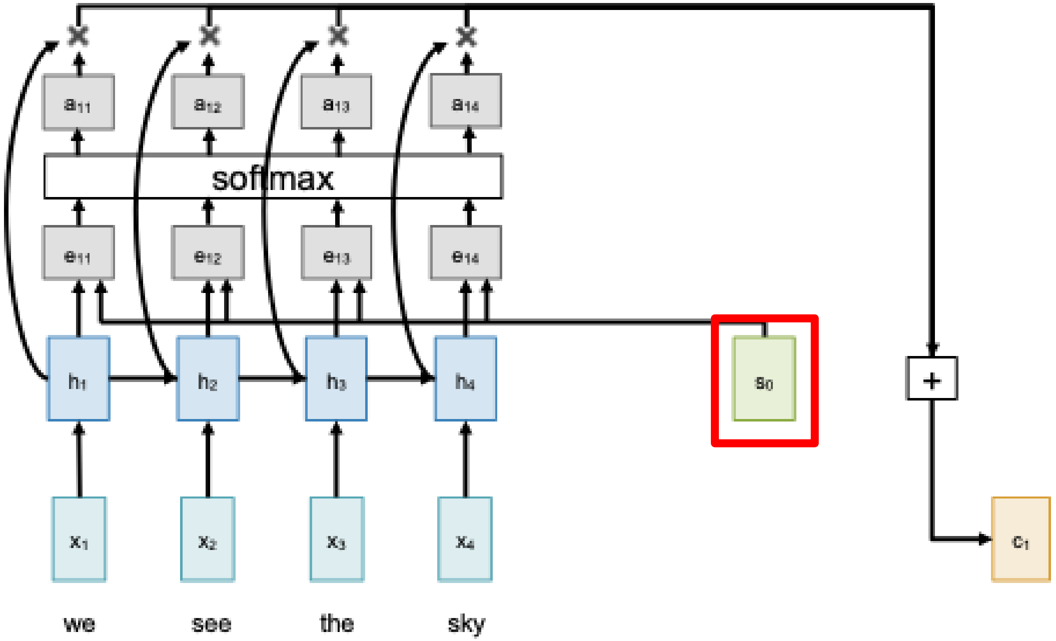
Query vectors (decoder RNN states) and
data vectors (encoder RNN states)
get transformed to
output vectors (Context states).
Each **query** attends to all **data** vectors and gives one **output** vector

There's a general operator hiding here:



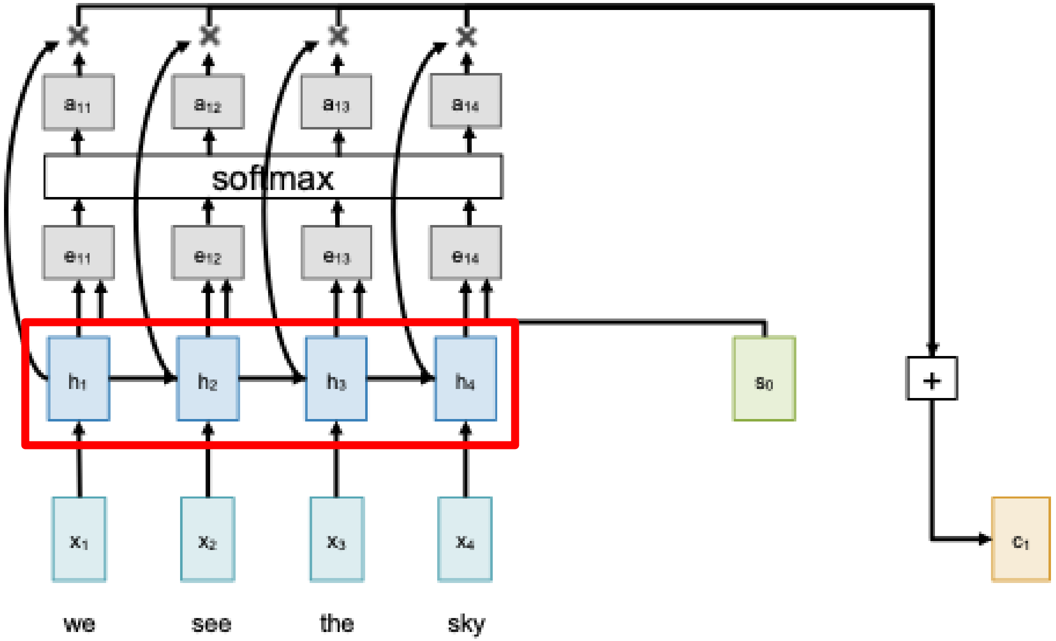
Attention Layer

Inputs:
Query vector: **q** [D_Q]



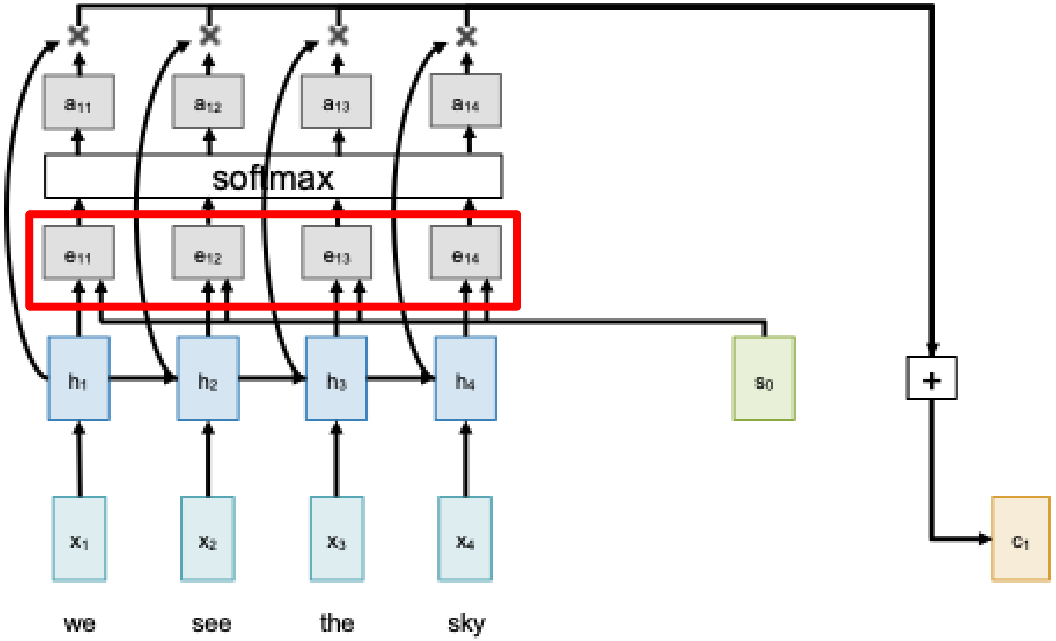
Attention Layer

Inputs:
Query vector: **q** [D_Q]
Data vectors: **X** [N_X x D_X]



Attention Layer

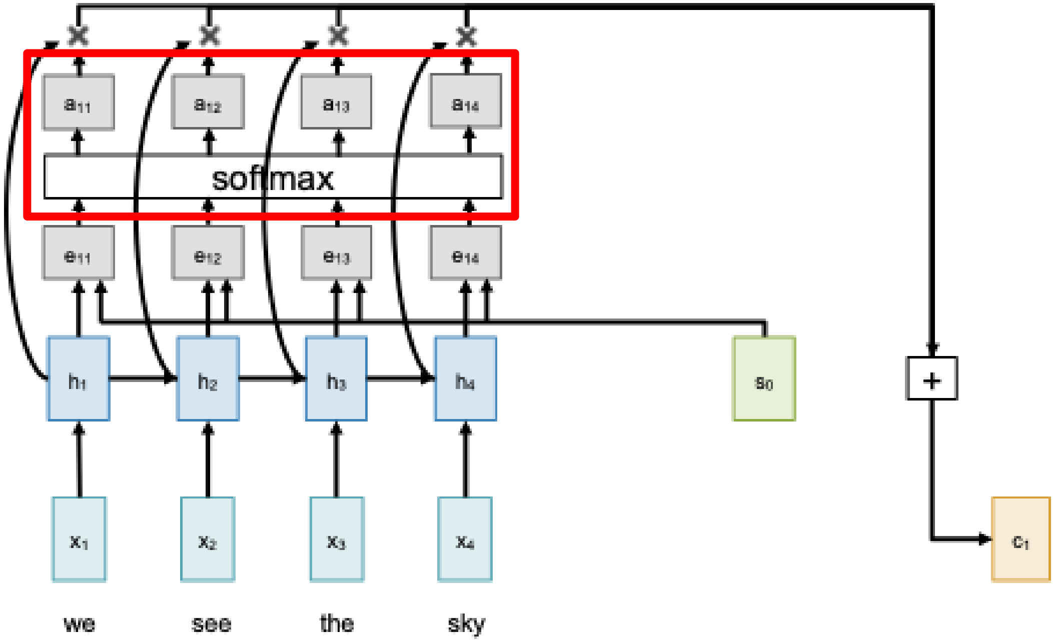
Inputs:
Query vector: $\mathbf{q}$ $[D_Q]$
Data vectors: $\mathbf{X}$ $[N_X \times D_X]$



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

Attention Layer

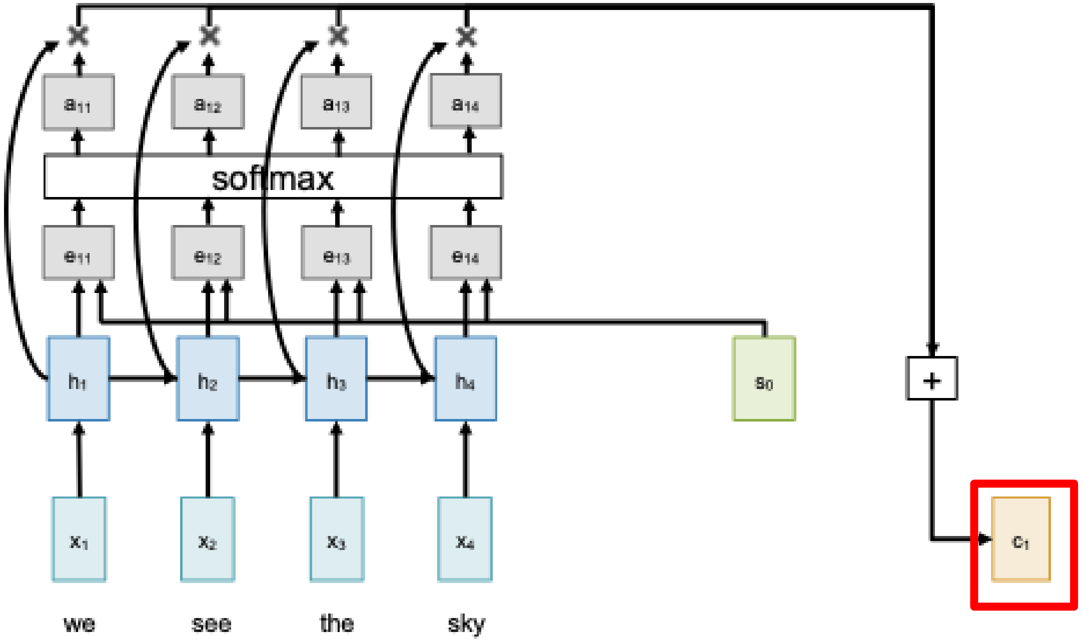
Inputs:
Query vector: **q** [D_Q]
Data vectors: **X** [N_X x D_X]



Computation:
Similarities: e [N_X] $e_i = f_{att}(\mathbf{q}, \mathbf{X}_i)$
Attention weights: a = softmax(e) [N_X]

Attention Layer

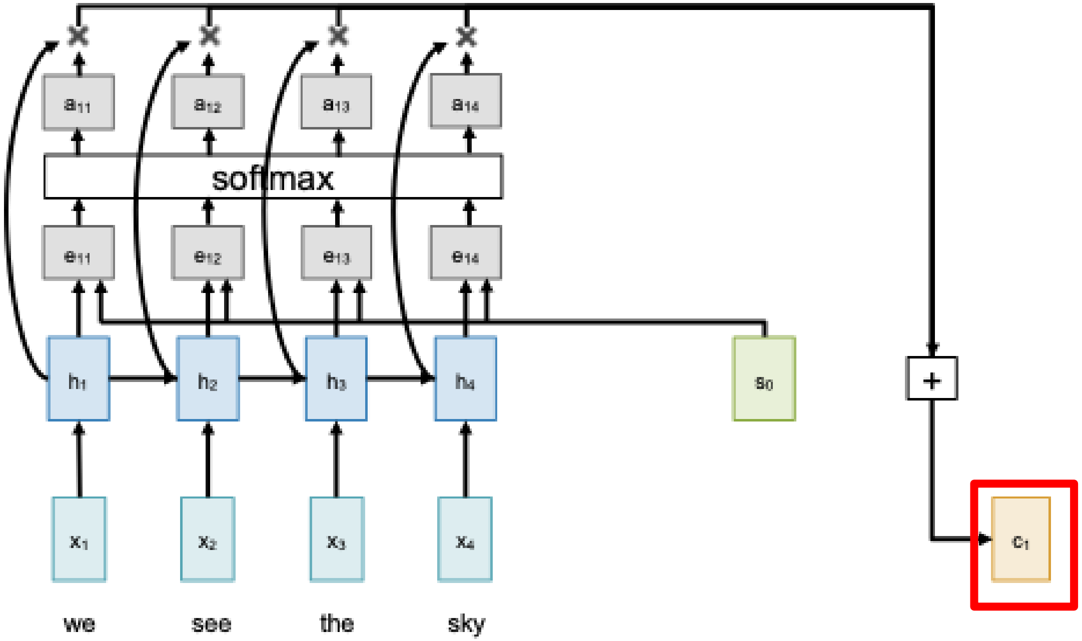
Inputs:
Query vector: $\mathbf{q}$ $[D_Q]$
Data vectors: $\mathbf{X}$ $[N_X \times D_X]$



Computation:
Similarities: e $[N_X]$ $e_i = f_{att}(\mathbf{q}, \mathbf{X}_i)$
Attention weights: a = softmax(e) $[N_X]$
Output vector: $\mathbf{y} = \sum_i a_i \mathbf{X}_i$ $[D_X]$

Attention Layer

Inputs:
Query vector: **q** [D_Q]
Data vectors: **X** [N_X x D_X]

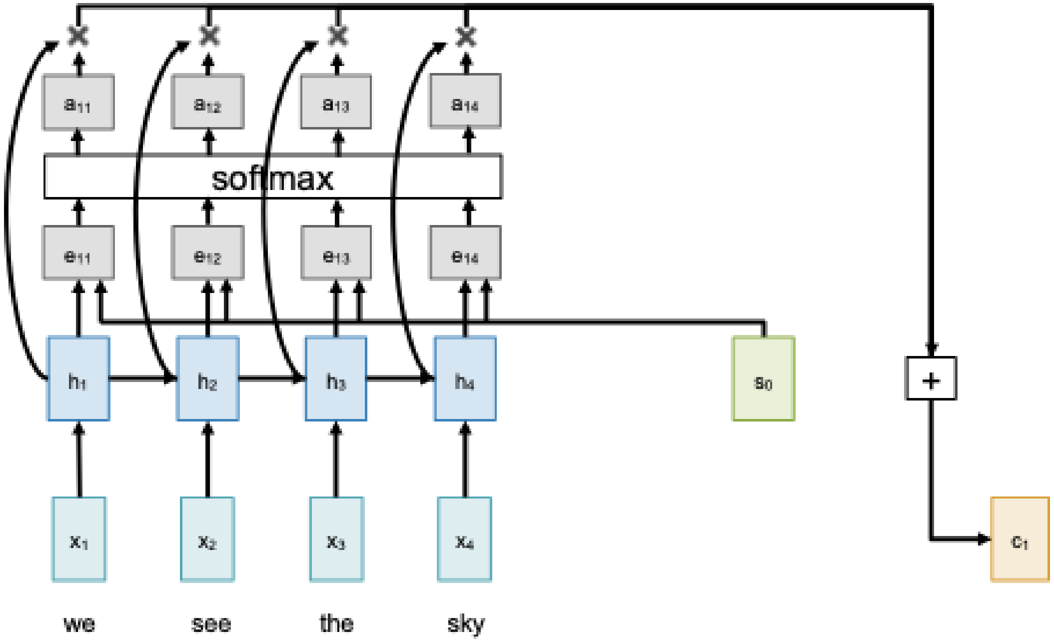


Computation:
Similarities: e [N_X] $e_i = f_{att}(\mathbf{q}, \mathbf{X}_i)$
Attention weights: a = softmax(e) [N_X]
Output vector: **y** = $\sum_i a_i \mathbf{X}_i$ [D_X]

Let's generalize this!

Attention Layer

Inputs:
Query vector: $\mathbf{q}$ $[D_X]$
Data vectors: $\mathbf{X}$ $[N_X \times D_X]$

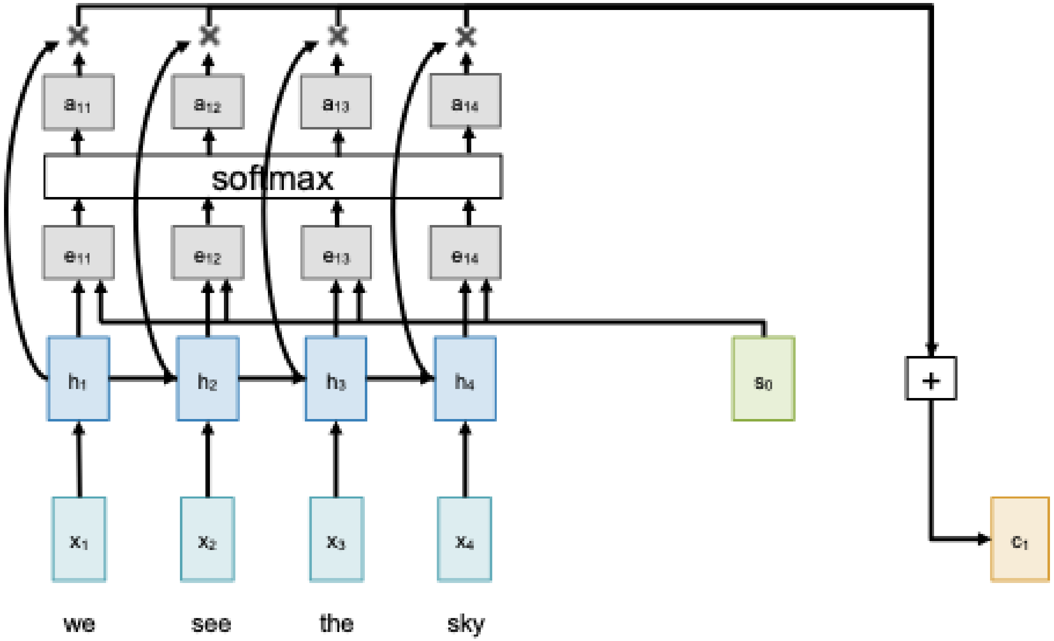


Computation:
Similarities: e $[N_X]$ $\boxed{e_i = \mathbf{q} \cdot \mathbf{X}_i}$
Attention weights: a = softmax(e) $[N_X]$
Output vector: $\mathbf{y} = \sum_i a_i \mathbf{X}_i$ $[D_X]$

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 $[N_X]$ $\boxed{e_i = \mathbf{q} \cdot \mathbf{X_i} / \sqrt{D_X}}$
Attention weights: a = softmax(e) $[N_X]$
Output vector: $\mathbf{y} = \sum_i a_i \mathbf{X_i}$ $[D_X]$

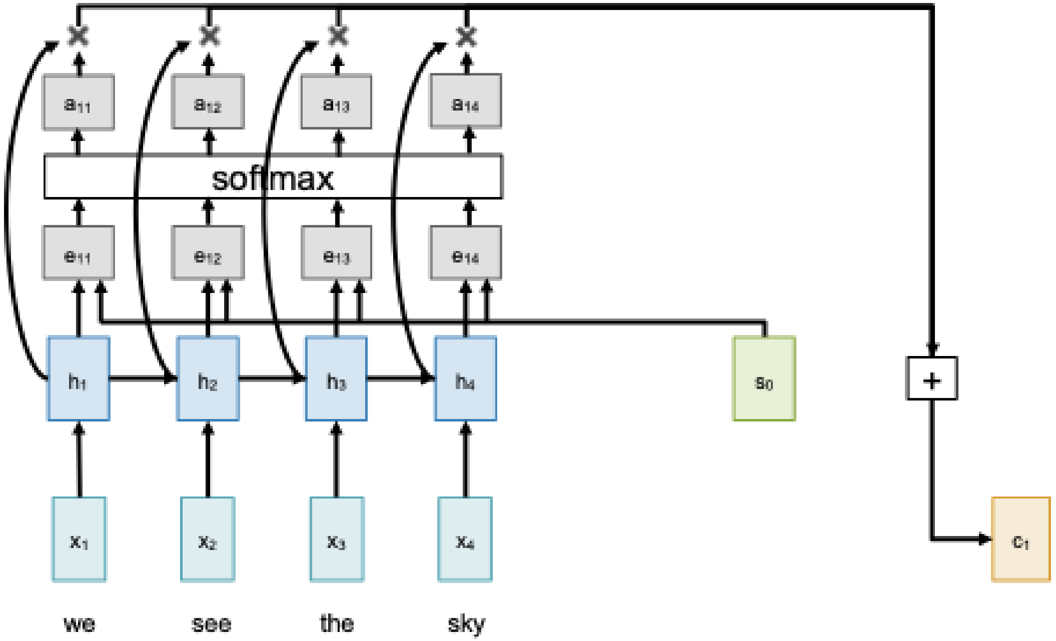
Changes
- Use **scaled** dot product for similarity

Attention Layer

Inputs:
Query vector: $\mathbf{q}$ [D_X]
Data vectors: $\mathbf{X}$ [N_X x D_X]

Large similarities will cause softmax to saturate and give vanishing gradients
Recall a · b = |a||b| cos(angle)
Suppose that a and b are constant vectors of dimension D
Then $|a| = (\sum_i a^2)^{1/2} = a\sqrt{D}$

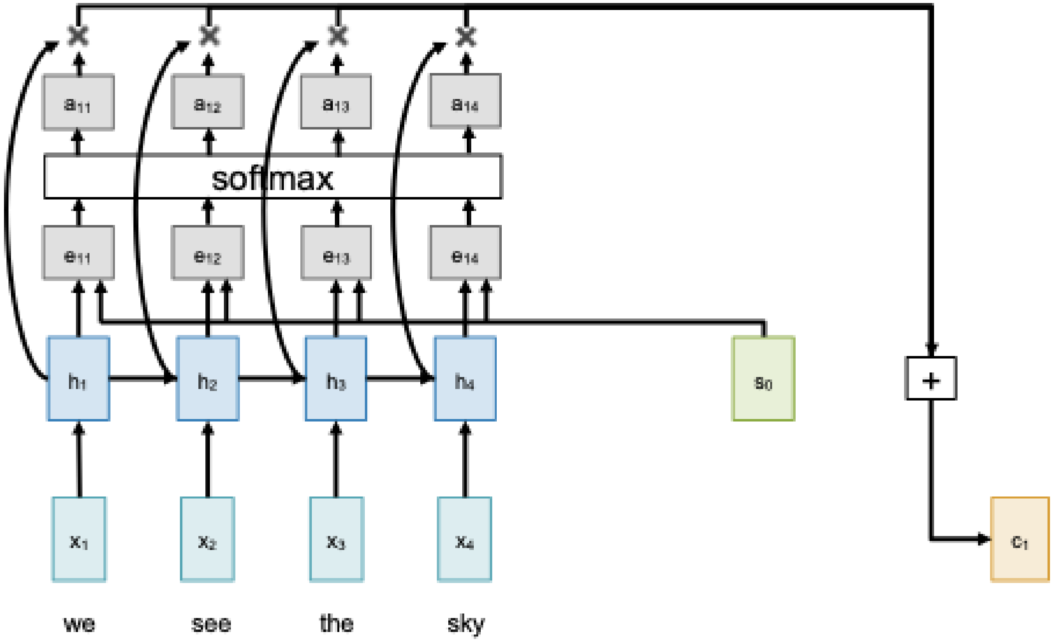
Computation:
Similarities: e [N_X] $\boxed{e_i = \mathbf{q} \cdot \mathbf{X_i} / \sqrt{D_X}}$
Attention weights: a = softmax(e) [N_X]
Output vector: $\mathbf{y} = \sum_i a_i \mathbf{X_i}$ [D_X]



Changes
- Use **scaled** dot product for similarity

Attention Layer

Inputs:
Query vector: $\boxed{\mathbf{Q}\ [N_Q \times D_X]}$
Data vectors: $\mathbf{X}$ $[N_X \times D_X]$

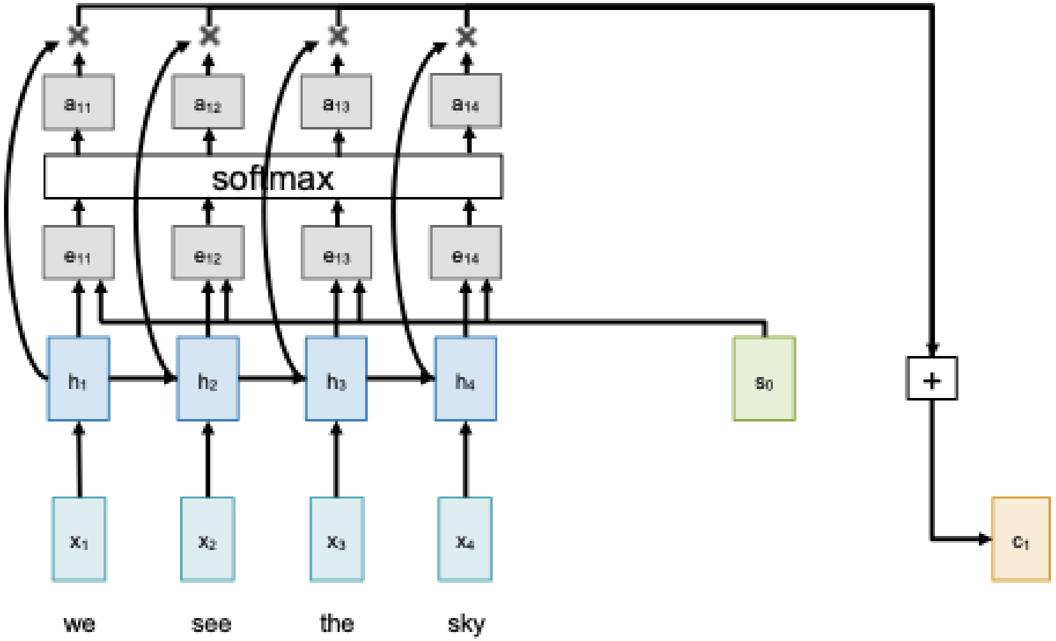


Computation:
Similarities: $E = \mathbf{Q}\mathbf{X}^T / \sqrt{D_X}$ $[N_Q \times N_X]$
$E_{ij} = \mathbf{Q}_i \cdot \mathbf{X}_j / \sqrt{D_X}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$
Output vector: $\mathbf{Y} = A\mathbf{X}$ $[N_Q \times D_X]$
$\mathbf{Y}_i = \sum_j A_{ij}\mathbf{X}_j$

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

Attention Layer

Inputs:
Query vector: **Q** $[N_Q \times D_Q]$
Data vectors: **X** $[N_X \times D_X]$
Key matrix: $\mathbf{W_K}$ $[D_X \times D_Q]$
Value matrix: $\mathbf{W_V}$ $[D_X \times D_V]$



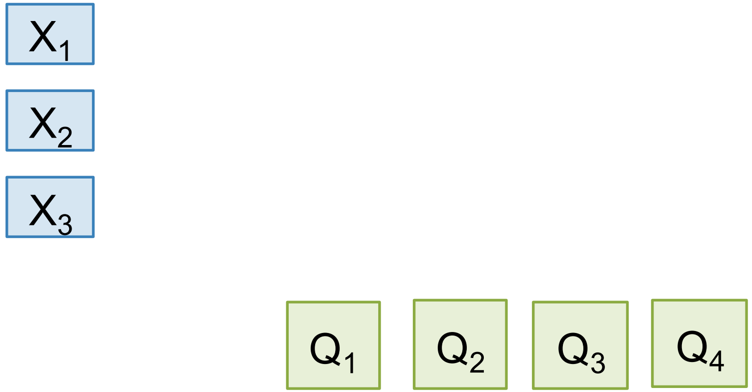
Computation:
Keys: $\mathbf{K} = \mathbf{XW_K}$ $[N_X \times D_Q]$
Values: $\mathbf{V} = \mathbf{XW_V}$ $[N_X \times D_V]$
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$
Output vector: $\mathbf{Y} = A\mathbf{V}$ $[N_Q \times D_V]$
$\mathbf{Y_i} = \sum_j A_{ij}\mathbf{V_j}$

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: **Q** $[N_Q \times D_Q]$
Data vectors: **X** $[N_X \times D_X]$
Key matrix: W_K $[D_X \times D_Q]$
Value matrix: W_V $[D_X \times D_V]$

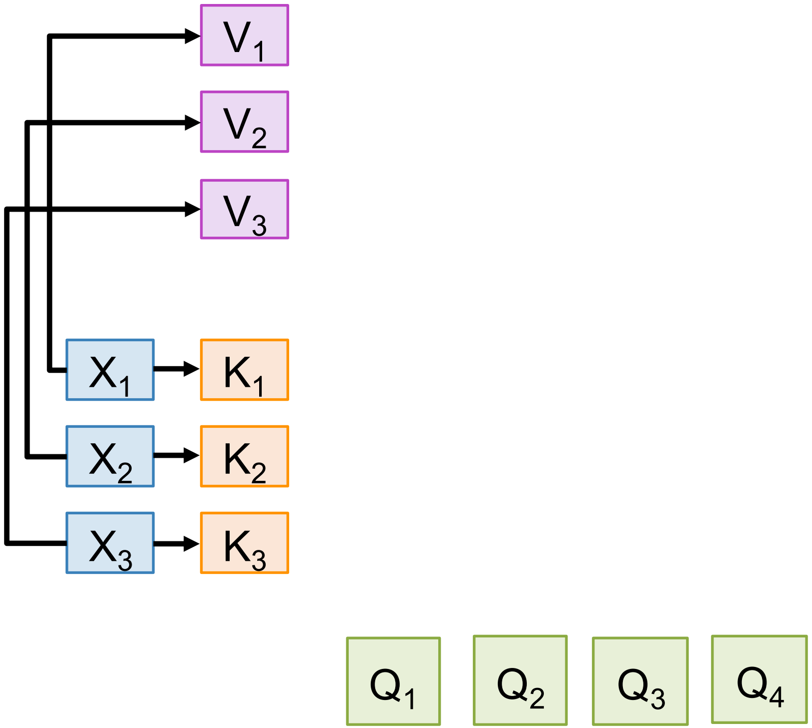
Computation:
Keys: $K = XW_K$ $[N_X \times D_Q]$
Values: $V = XW_V$ $[N_X \times D_V]$
Similarities: $E = QK^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$
Output vector: $Y = AV$ $[N_Q \times D_V]$
$Y_i = \sum_j A_{ij} V_j$



Attention Layer

Inputs:
Query vector: **Q** [N_Q x D_Q]
Data vectors: **X** [N_X x D_X]
Key matrix: $\mathbf{W_K}$ [D_X x D_Q]
Value matrix: $\mathbf{W_V}$ [D_X x D_V]

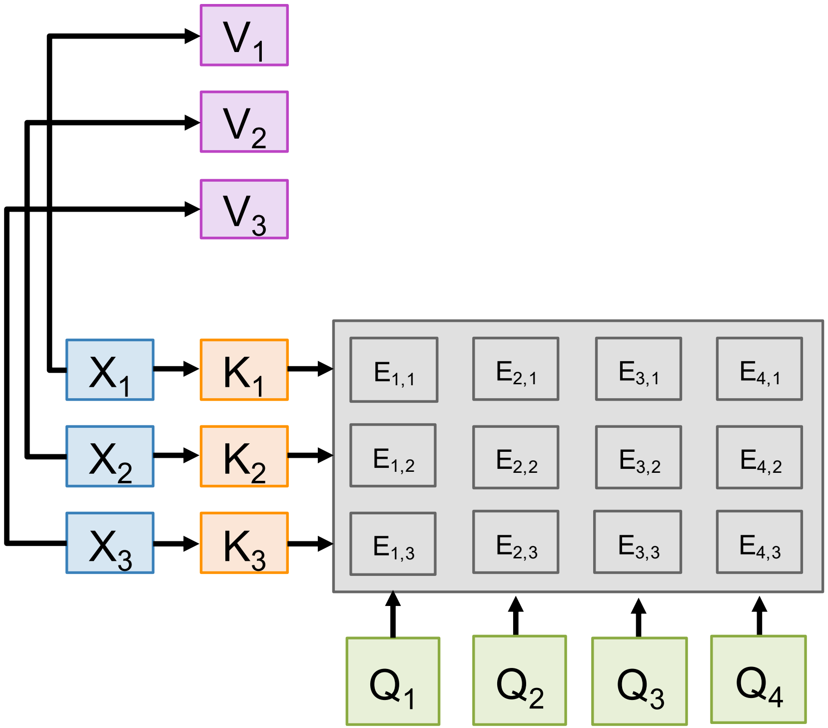
Computation:
Keys: **K** = $\mathbf{XW_K}$ [N_X x D_Q]
Values: **V** = $\mathbf{XW_V}$ [N_X x D_V]
Similarities: E = $QK^T / \sqrt{D_Q}$ [N_Q x N_X]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N_Q x N_X]
Output vector: Y = AV [N_Q x D_V]
$Y_i = \sum_j A_{ij} V_j$



Attention Layer

Inputs:
Query vector: **Q** $[N_Q \times D_Q]$
Data vectors: **X** $[N_X \times D_X]$
Key matrix: $\mathbf{W_K}$ $[D_X \times D_Q]$
Value matrix: $\mathbf{W_V}$ $[D_X \times D_V]$

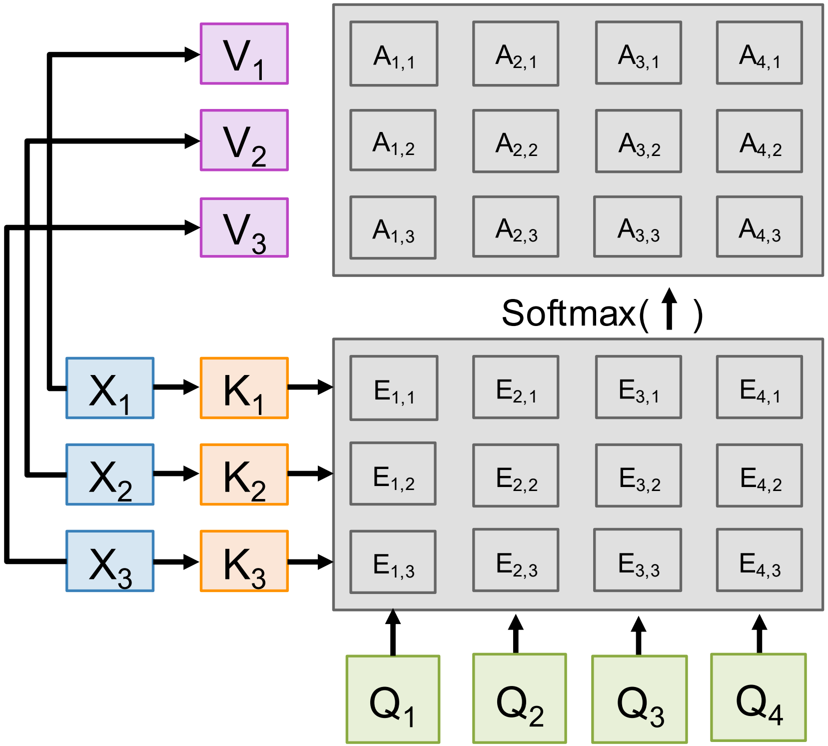
Computation:
Keys: $\mathbf{K} = \mathbf{XW_K}$ $[N_X \times D_Q]$
Values: $\mathbf{V} = \mathbf{XW_V}$ $[N_X \times D_V]$
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$
Output vector: $\mathbf{Y} = A\mathbf{V}$ $[N_Q \times D_V]$
$\mathbf{Y_i} = \sum_j A_{ij} \mathbf{V_j}$



Attention Layer

Inputs:
Query vector: **Q** $[N_Q \times D_Q]$
Data vectors: **X** $[N_X \times D_X]$
Key matrix: $\mathbf{W_K}$ $[D_X \times D_Q]$
Value matrix: $\mathbf{W_V}$ $[D_X \times D_V]$

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



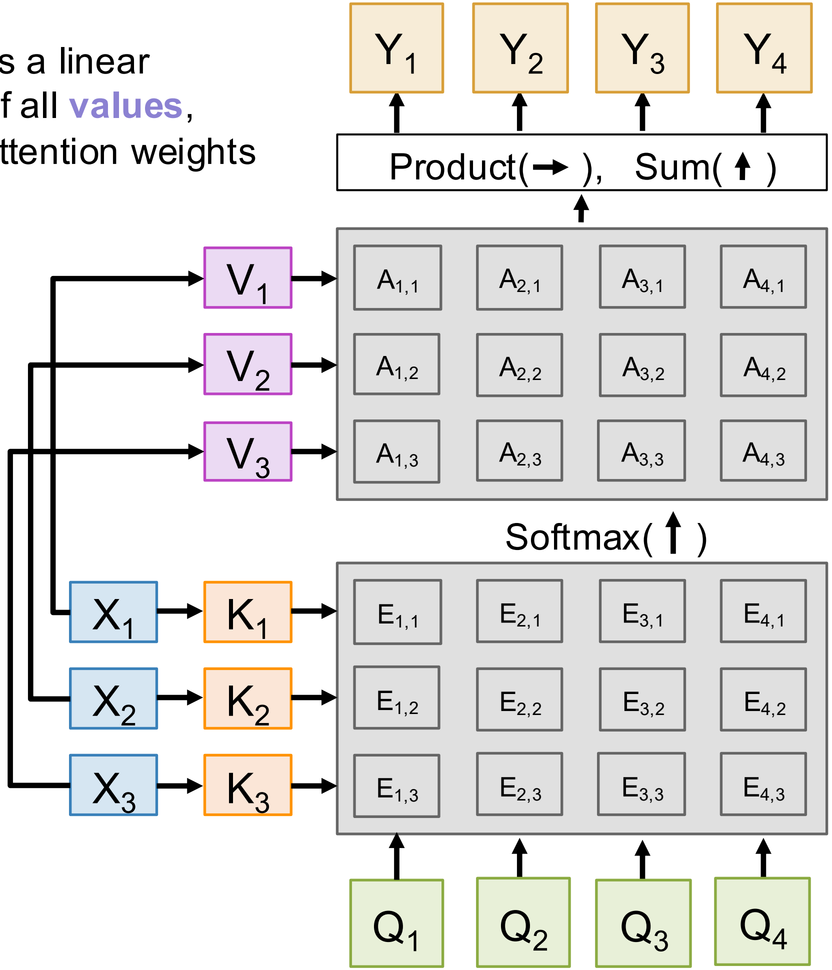
Computation:
Keys: $\mathbf{K} = \mathbf{XW_K}$ $[N_X \times D_Q]$
Values: $\mathbf{V} = \mathbf{XW_V}$ $[N_X \times D_V]$
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$
Output vector: $\mathbf{Y} = A\mathbf{V}$ $[N_Q \times D_V]$
$\mathbf{Y_i} = \sum_j A_{ij} \mathbf{V_j}$

Attention Layer

Inputs:
Query vector: **Q** $[N_Q \times D_Q]$
Data vectors: **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: $K = XW_K$ $[N_X \times D_Q]$
Values: $V = XW_V$ $[N_X \times D_V]$
Similarities: $E = QK^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$
Output vector: $Y = AV$ $[N_Q \times D_V]$
$Y_i = \sum_j A_{ij} V_j$

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

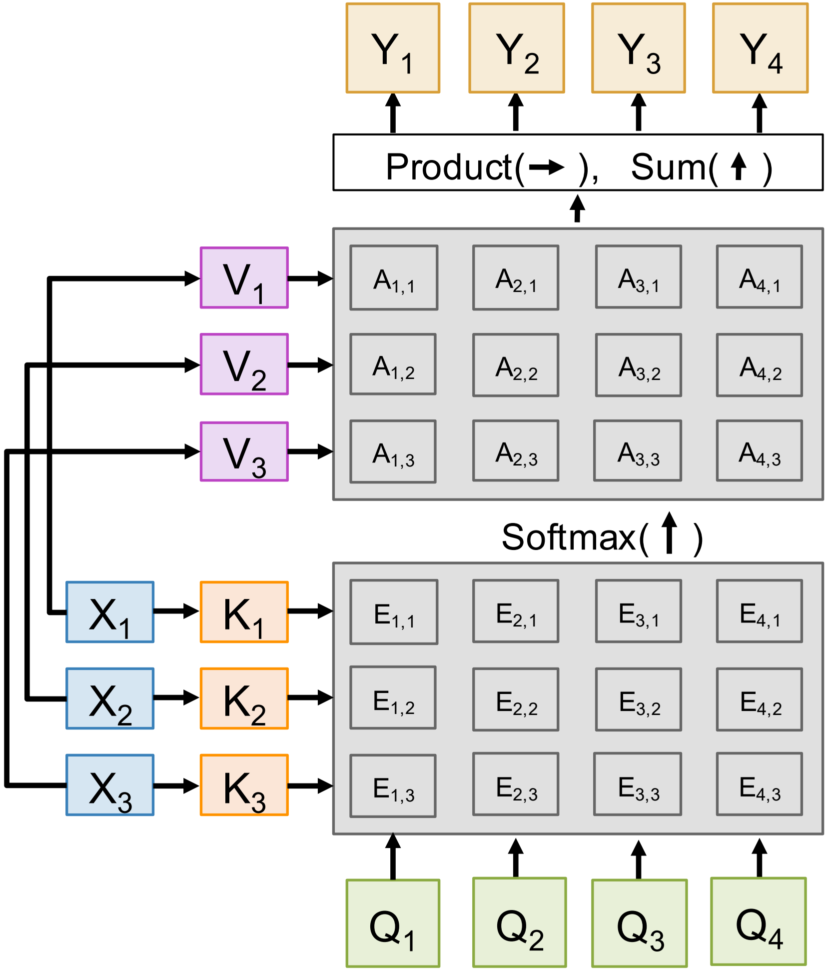


Cross-Attention Layer

Inputs:
Query vector: **Q** $[N_Q \times D_Q]$
Data vectors: **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: $\mathbf{V} = \mathbf{XW_V}$ $[N_X \times D_V]$
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ $[N_Q \times N_X]$
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ $[N_Q \times N_X]$
Output vector: $\mathbf{Y} = A\mathbf{V}$ $[N_Q \times D_V]$
$\mathbf{Y_i} = \sum_j A_{ij} \mathbf{V_j}$



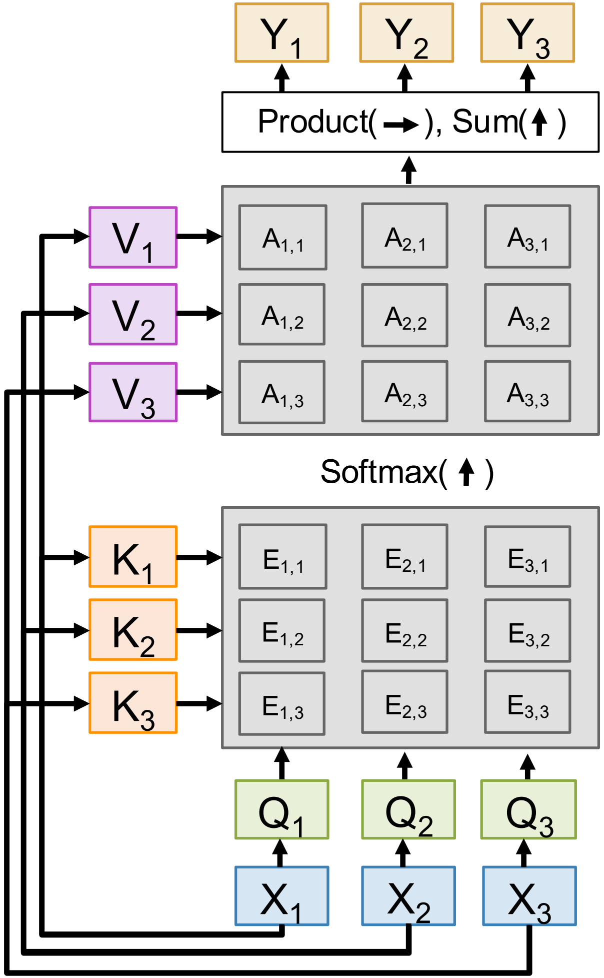
Self-Attention Layer

Inputs:
Input vectors: **X** [N x D_{in}]
Key matrix: **W_K** [D_{in} x D_{out}]
Value matrix: **W_V** [D_{in} x D_{out}]
Query matrix: **W_Q** [D_{in} x D_{out}]

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

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

Shapes get a little simpler:
- N input vectors, each D_{in}
- Almost always $D_Q = D_V = D_{out}$

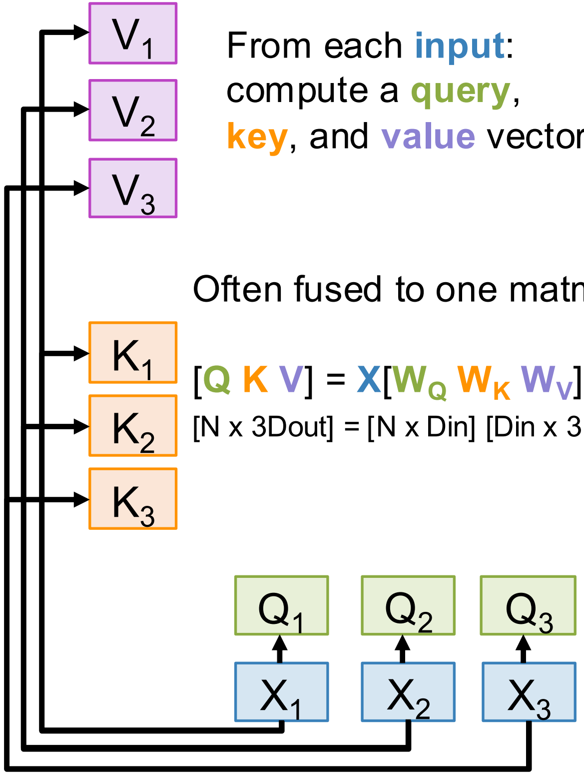


Self-Attention Layer

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

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

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$



From each **input**: compute a **query**, **key**, and **value** vector

Often fused to one matmul:

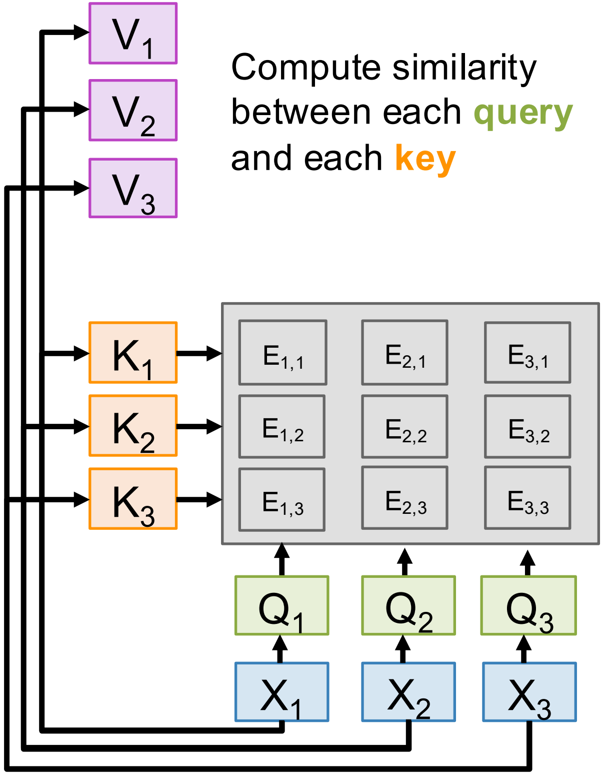
$[Q\ K\ V] = X[W_Q\ W_K\ W_V]$
[N x 3Dout] = [N x Din] [Din x 3Dout]

Self-Attention Layer

Inputs:
Input vectors: $\mathbf{X}$ [N x D_{in}]
Key matrix: $\mathbf{W_K}$ [D_{in} x D_{out}]
Value matrix: $\mathbf{W_V}$ [D_{in} x D_{out}]
Query matrix: $\mathbf{W_Q}$ [D_{in} x D_{out}]

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

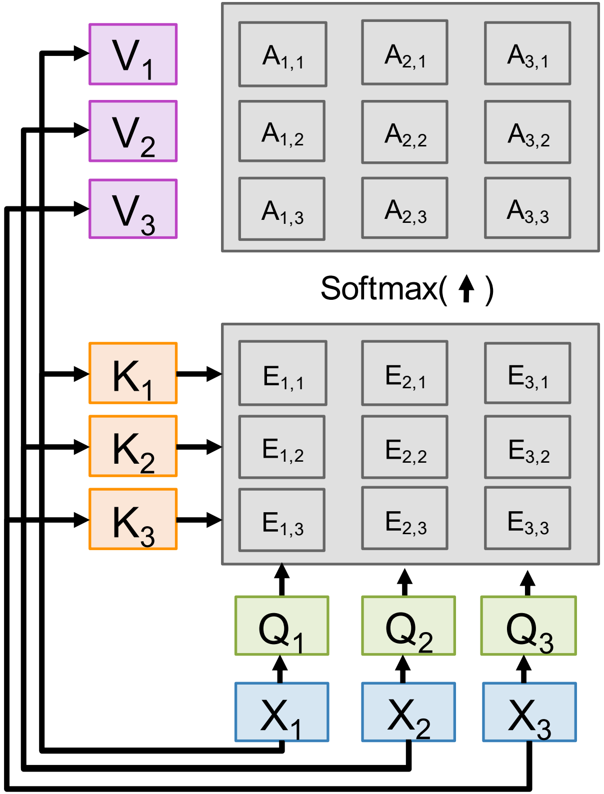
Computation:
Queries: $\mathbf{Q} = \mathbf{XW_Q}$ [N x D_{out}]
Keys: $\mathbf{K} = \mathbf{XW_K}$ [N x D_{out}]
Values: $\mathbf{V} = \mathbf{XW_V}$ [N x D_{out}]
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $\mathbf{Y} = A\mathbf{V}$ [N x D_{out}]
$\mathbf{Y_i} = \sum_j A_{ij} \mathbf{V_j}$



Compute similarity between each **query** and each **key**

Self-Attention Layer

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



Inputs:

Input vectors: **X** [N x D_{in}]

Key matrix: W_K [D_{in} x D_{out}]

Value matrix: W_V [D_{in} x D_{out}]

Query matrix: W_Q [D_{in} x D_{out}]

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

Computation:

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

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

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

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

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

Attention weights: A = softmax(E, dim=1) [N x N]

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

$Y_i = \sum_j A_{ij} V_j$

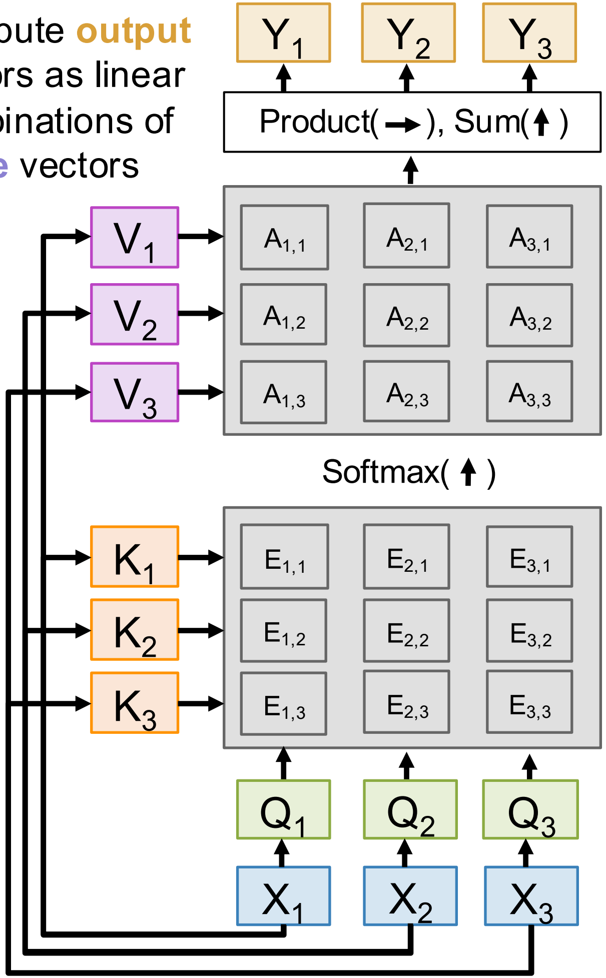
Self-Attention Layer

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

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

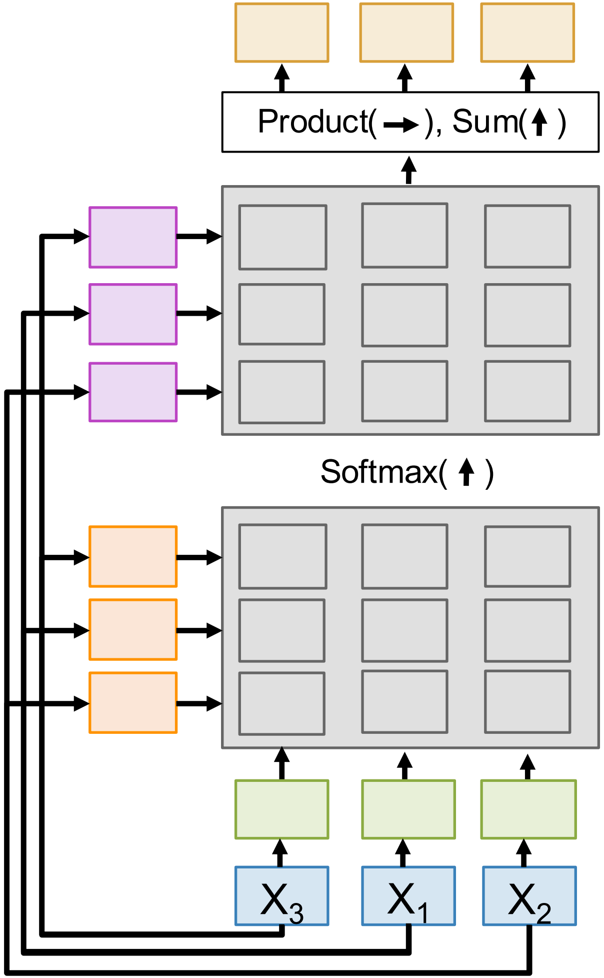
Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

Compute **output** vectors as linear combinations of **value** vectors



Self-Attention Layer

Consider permuting **inputs**:



Inputs:
Input vectors: **X** [N x D_{in}]
Key matrix: **W_K** [D_{in} x D_{out}]
Value matrix: **W_V** [D_{in} x D_{out}]
Query matrix: **W_Q** [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

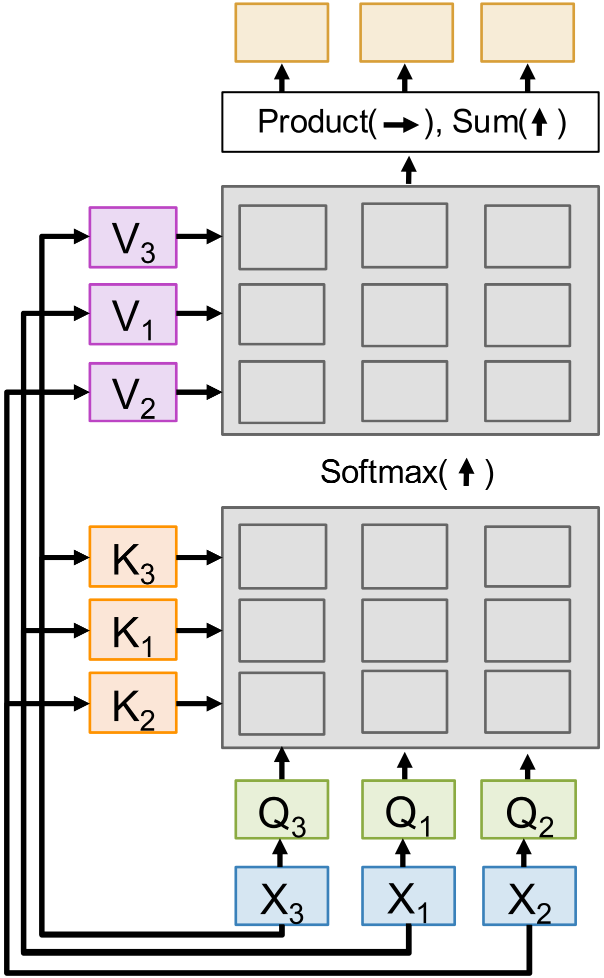
Self-Attention Layer

Consider permuting **inputs**:

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

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$



Self-Attention Layer

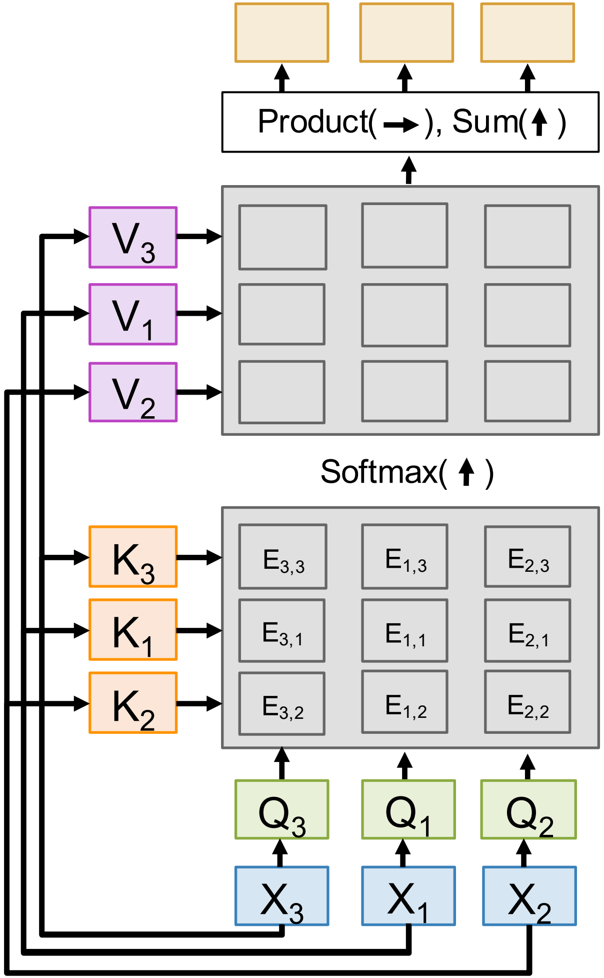
Inputs:
Input vectors: **X** [N x D_{in}]
Key matrix: **W_K** [D_{in} x D_{out}]
Value matrix: **W_V** [D_{in} x D_{out}]
Query matrix: **W_Q** [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $Y = AV$ [N x 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

<u>Inputs</u>:
Input vectors: $\mathbf{X}$ [N x D_{in}]
Key matrix: $\mathbf{W_K}$ [D_{in} x D_{out}]
Value matrix: $\mathbf{W_V}$ [D_{in} x D_{out}]
Query matrix: $\mathbf{W_Q}$ [D_{in} x D_{out}]

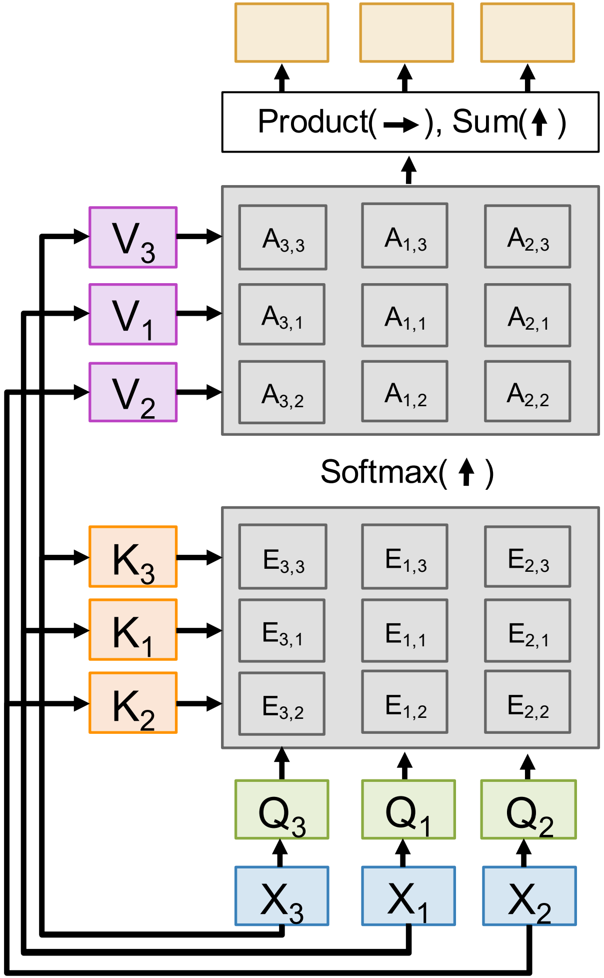
<u>Computation</u>:
Queries: $\mathbf{Q} = \mathbf{XW_Q}$ [N x D_{out}]
Keys: $\mathbf{K} = \mathbf{XW_K}$ [N x D_{out}]
Values: $\mathbf{V} = \mathbf{XW_V}$ [N x D_{out}]
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $\mathbf{Y} = A\mathbf{V}$ [N x D_{out}]
$\mathbf{Y_i} = \sum_j A_{ij} \mathbf{V_j}$

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 x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

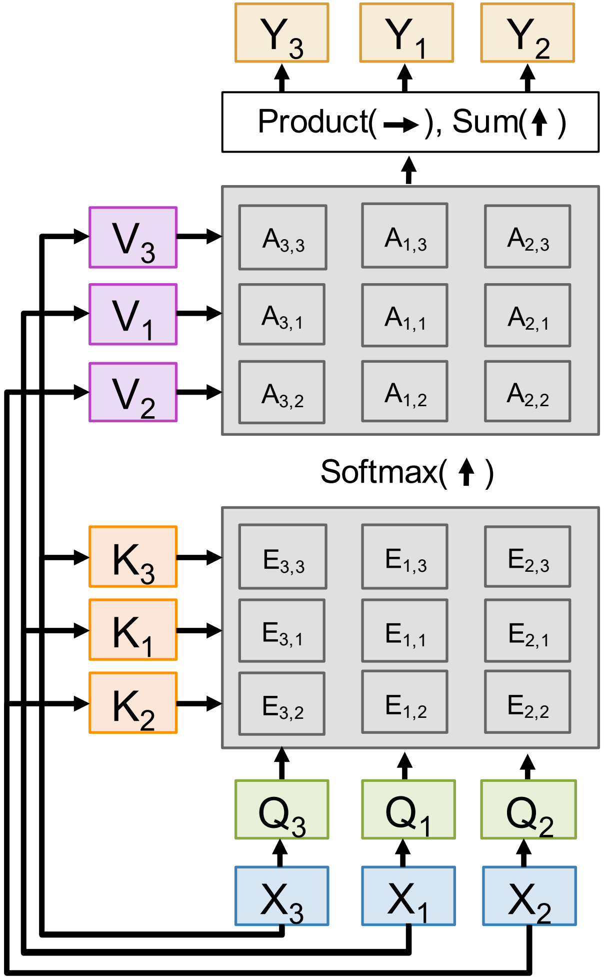
Consider permuting **inputs**:

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



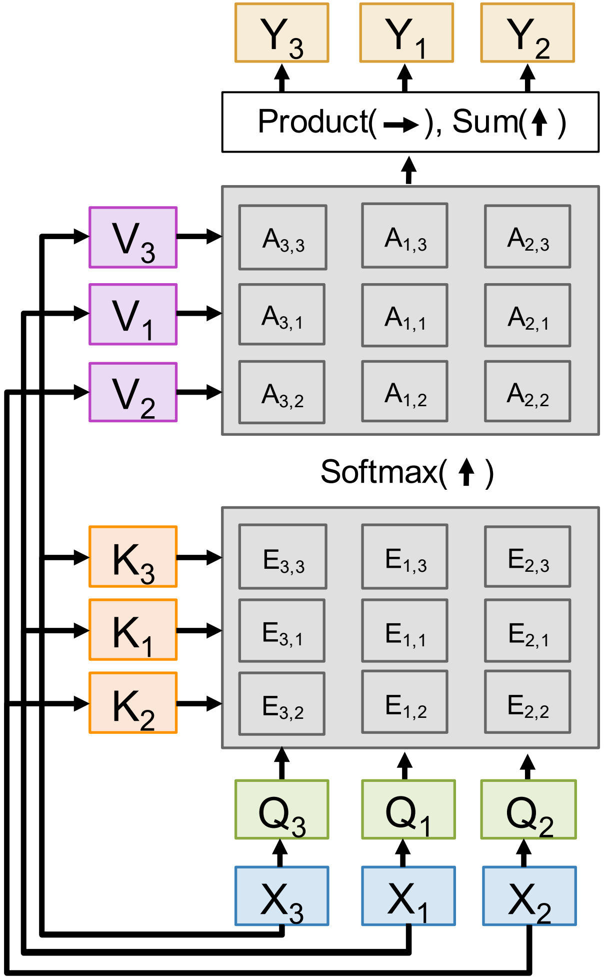
Self-Attention Layer

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x 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

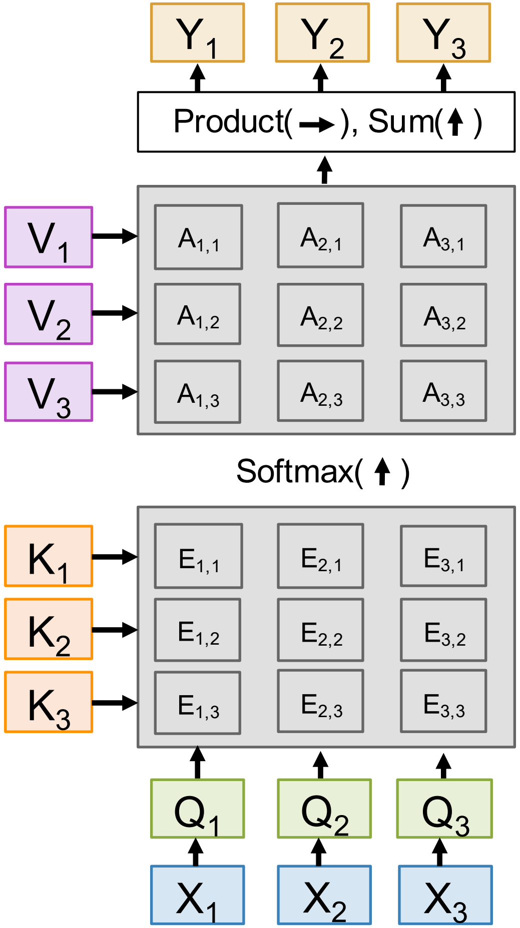


Self-Attention Layer

Inputs:
Input vectors: $\mathbf{X}$ [N x D_{in}]
Key matrix: $\mathbf{W_K}$ [D_{in} x D_{out}]
Value matrix: $\mathbf{W_V}$ [D_{in} x D_{out}]
Query matrix: $\mathbf{W_Q}$ [D_{in} x D_{out}]

Computation:
Queries: $\mathbf{Q} = \mathbf{XW_Q}$ [N x D_{out}]
Keys: $\mathbf{K} = \mathbf{XW_K}$ [N x D_{out}]
Values: $\mathbf{V} = \mathbf{XW_V}$ [N x D_{out}]
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $\mathbf{Y} = A\mathbf{V}$ [N x D_{out}]
$\mathbf{Y_i} = \sum_j A_{ij} \mathbf{V_j}$

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



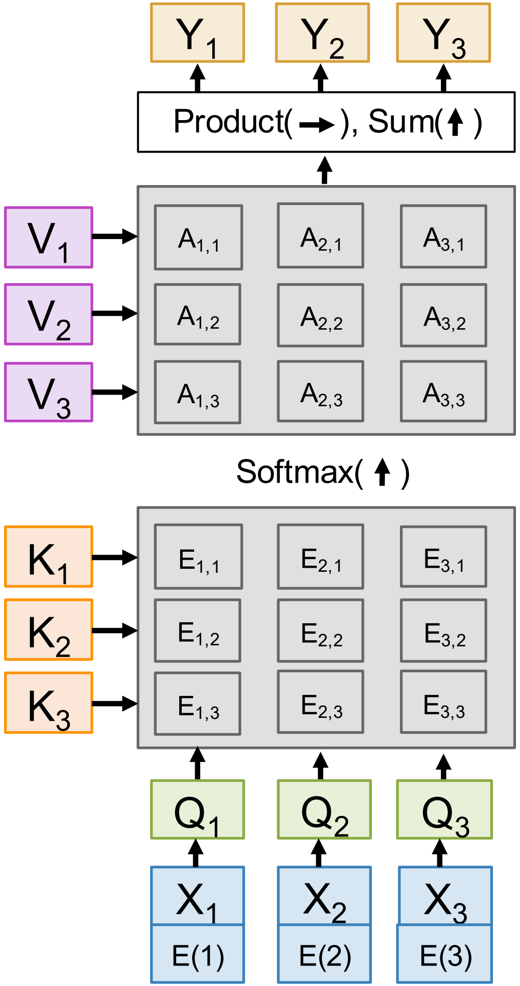
Self-Attention Layer

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x 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 x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

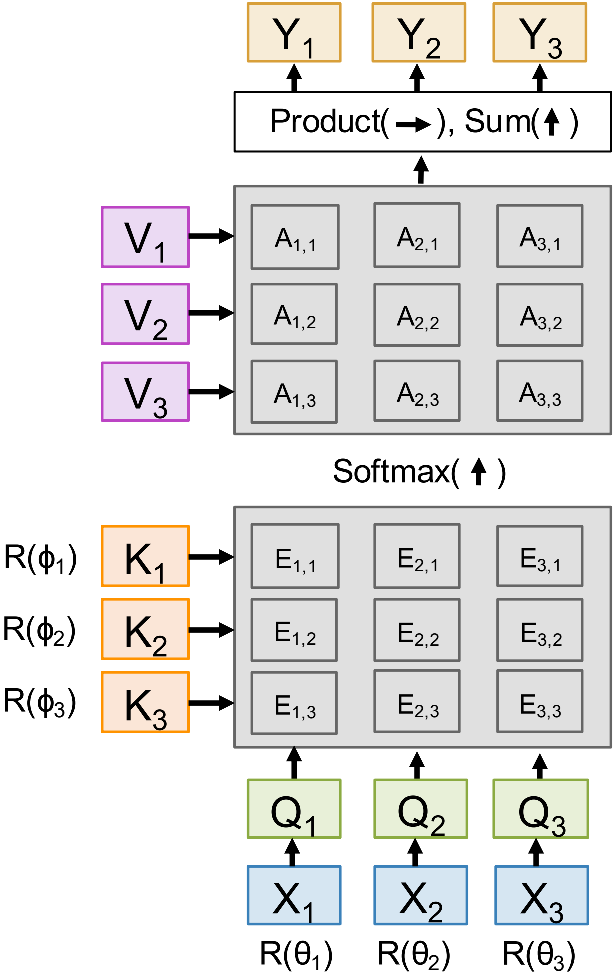
Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$

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)

$$(R(\theta_i)q_i)^T\left(R(\phi_j)k_j\right) = q_i^T R(\phi_j - \theta_i)k_j$$

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



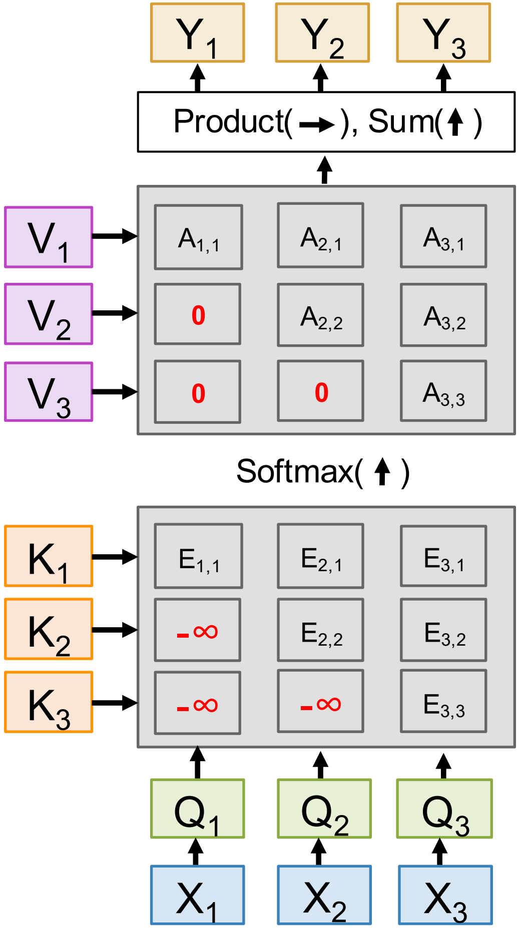
Masked Self-Attention Layer

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

Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

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

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AV$ [N x D_{out}]
$Y_i = \sum_j A_{ij} V_j$



Masked Self-Attention Layer

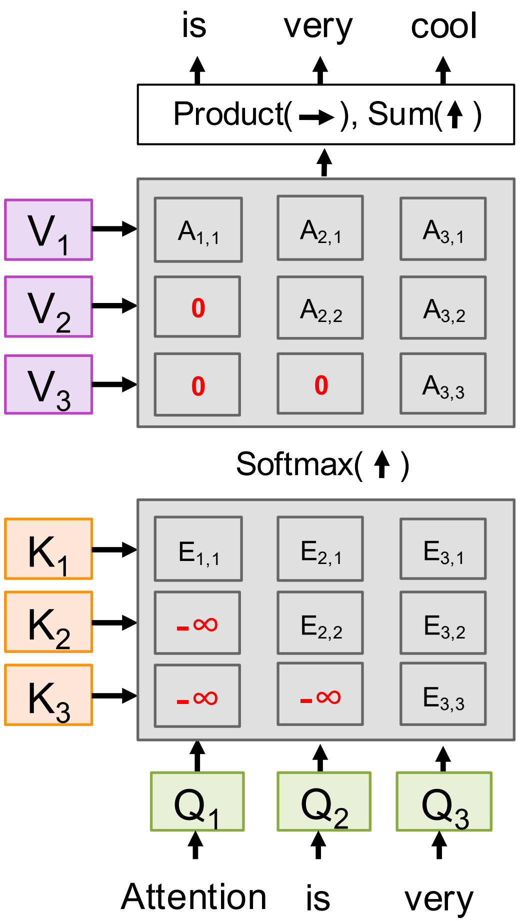
Don't let vectors "look ahead" in the sequence

Inputs:
Input vectors: $\mathbf{X}$ [N x D_{in}]
Key matrix: $\mathbf{W_K}$ [D_{in} x D_{out}]
Value matrix: $\mathbf{W_V}$ [D_{in} x D_{out}]
Query matrix: $\mathbf{W_Q}$ [D_{in} x D_{out}]

Computation:
Queries: $\mathbf{Q} = \mathbf{XW_Q}$ [N x D_{out}]
Keys: $\mathbf{K} = \mathbf{XW_K}$ [N x D_{out}]
Values: $\mathbf{V} = \mathbf{XW_V}$ [N x D_{out}]
Similarities: $E = \mathbf{QK}^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = \mathbf{Q_i} \cdot \mathbf{K_j} / \sqrt{D_Q}$
Attention weights: A = softmax(E, dim=1) [N x N]
Output vector: $\mathbf{Y} = A\mathbf{V}$ [N x D_{out}]
$\mathbf{Y_i} = \sum_j A_{ij} \mathbf{V_j}$

Override similarities with -inf; 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: $E = QK^T / \sqrt{D_Q}$ $[N \times N]$

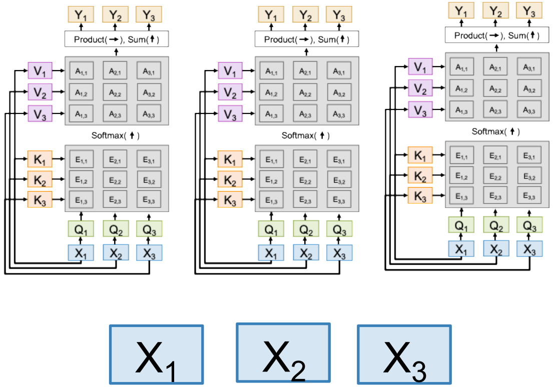
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$

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

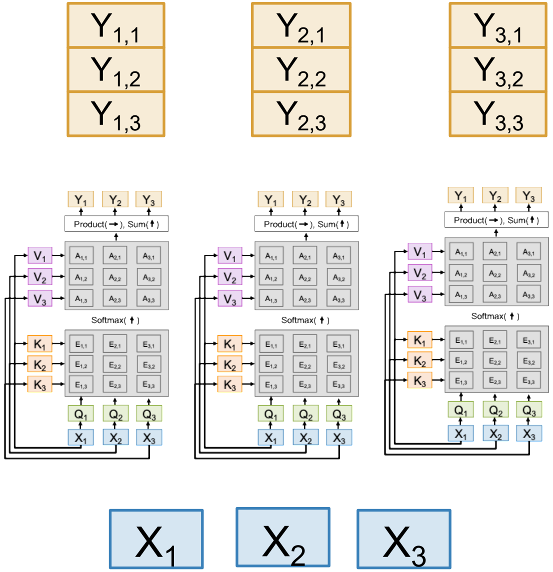
Run H copies of Self-Attention in parallel

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

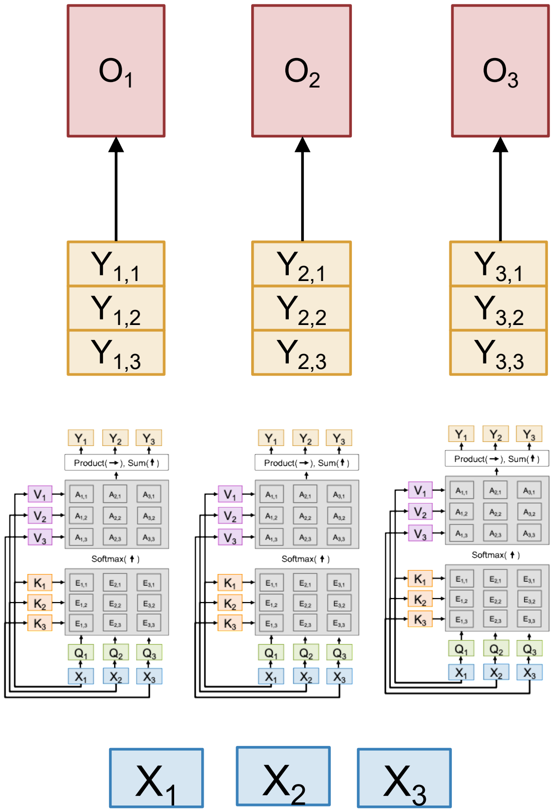
Inputs:
Input vectors: X [N x D_{in}]
Key matrix: W_K [D_{in} x D_{out}]
Value matrix: W_V [D_{in} x D_{out}]
Query matrix: W_Q [D_{in} x D_{out}]

Computation:
Queries: $Q = XW_Q$ [N x D_{out}]
Keys: $K = XW_K$ [N x D_{out}]
Values: $V = XW_V$ [N x D_{out}]
Similarities: $E = QK^T / \sqrt{D_Q}$ [N x N]
$E_{ij} = Q_i \cdot K_j / \sqrt{D_Q}$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ [N x N]
Output vector: $Y = AX$ [N x 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]

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

Computation:

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

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

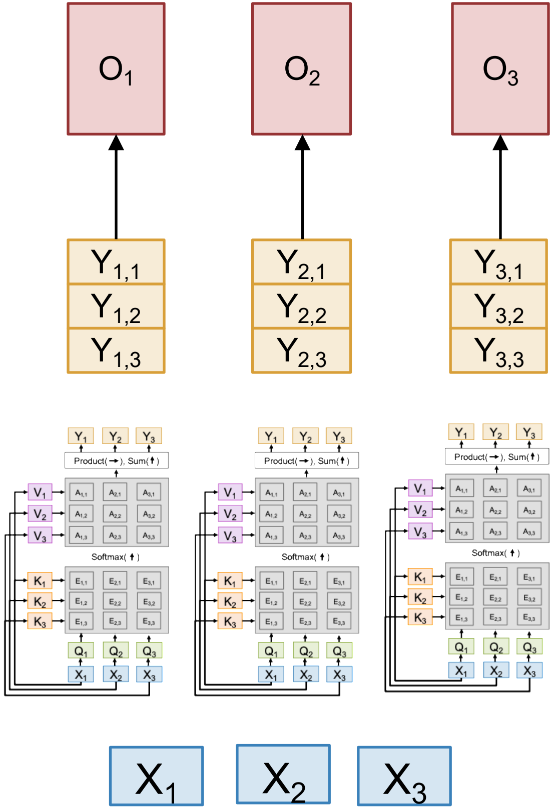
Values: $V = XW_V$ [H x N x D_H]

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

Attention weights: A = softmax(E, dim=2) [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]



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 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]

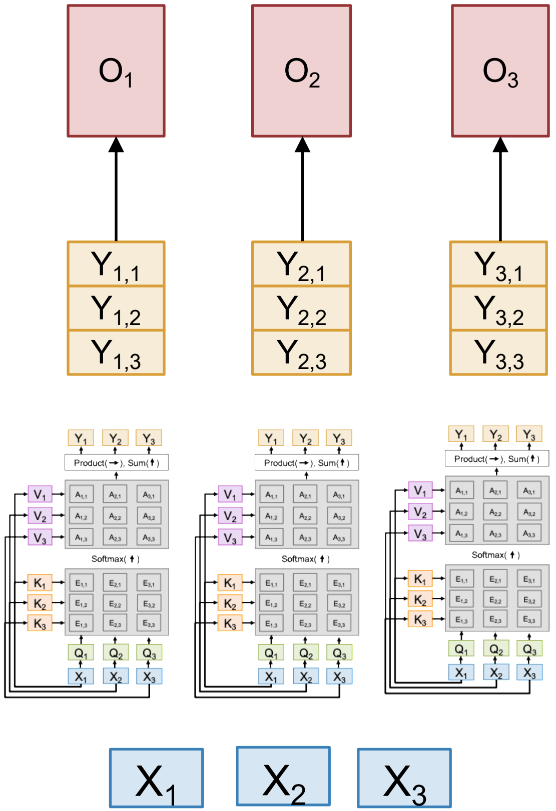
Attention weights: A = softmax(E, dim=2) [H x N x N]

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

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

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

Used everywhere in practice.



Self-Attention is Four Matrix Multiplies!

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 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 = softmax(E, dim=2) [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]

Self-Attention is Four Matrix Multiplies!

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]

Self-Attention is Four Matrix Multiplies!

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]

Self-Attention is Four Matrix Multiplies!

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 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: $A = \text{softmax}(E, \text{dim}=2)$ [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]

Self-Attention is Four Matrix Multiplies!

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x D]

Self-Attention is Four Matrix Multiplies!

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x D]

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

Self-Attention is Four Matrix Multiplies!

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x 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: 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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x D]

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

Self-Attention is Four Matrix Multiplies!

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x 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 HxNxN attention weights take 1.192 TB! GPUs don't have that much memory…

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get **Q**, **K**, **V** each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x 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 HxNxN attention weights take 1.192 TB! GPUs don't have that much memory…

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

Makes large N possible

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 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 = softmax(E, dim=2) [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]

1. QKV Projection
   [N x D] [D x $3HD_H$] => [N x $3HD_H$]
   Split and reshape to get Q, K, V each of shape [H x N x D_H]
2. QK Similarity
   [H x N x D_H] [H x N x D_H] => [H x N x N]
3. V-Weighting
   [H x N x N] [H x N x D_H] => [H x N x D_H]
   Reshape to [N x HD_H]
4. Output Projection
   [N x HD_H] [HD_H x D] => [N x D]

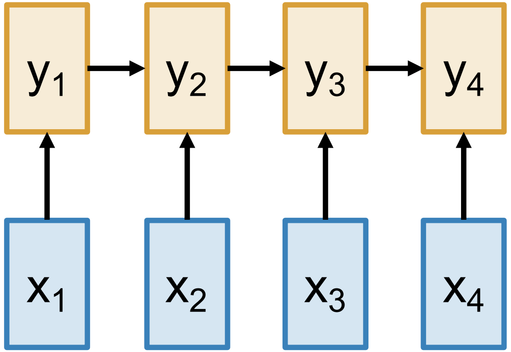
Q: How much memory does this take as the number of vectors N increases?
A: O(N) with Flash Attention

Dao et al, "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness", 2022

Three Ways of Processing Sequences

Three Ways of Processing Sequences

Recurrent Neural Network



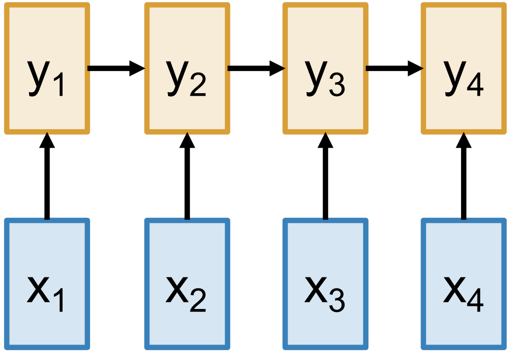
Works on **1D ordered sequences**

(+) 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

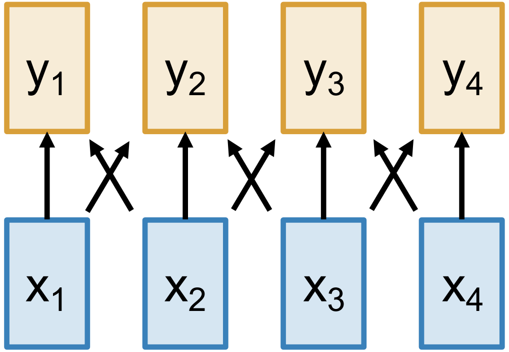


Works on **1D ordered sequences**

(+) 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



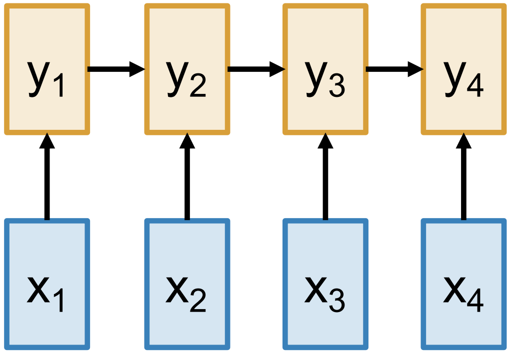
Works on **N-dimensional grids**

(-) 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

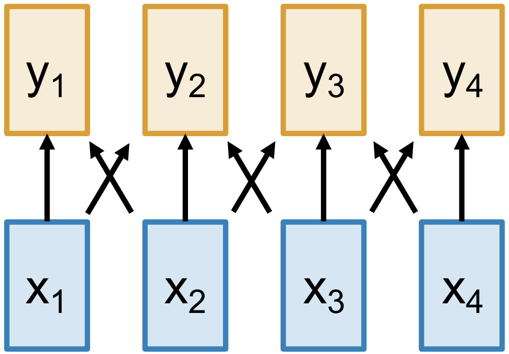


Works on **1D ordered sequences**

(+) 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

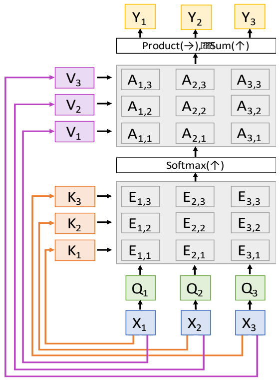


Works on **N-dimensional grids**

(-) 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

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

Convolution

receptive fields
(+) Parallelizable, outputs can be computed in parallel

Self-Attention

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

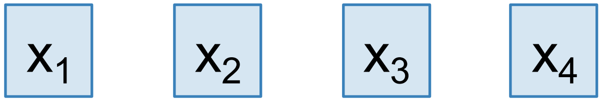
Attention is All You Need

Vaswani et al, NeurIPS 2017

The Transformer

Transformer Block

Input: Set of vectors x



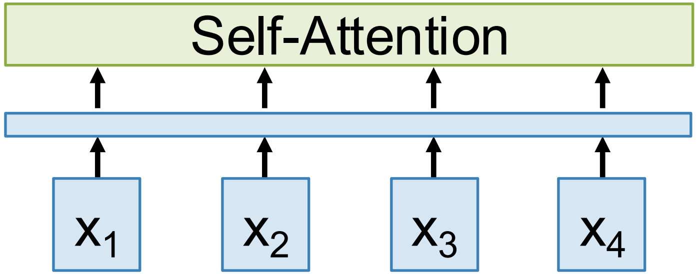
Vaswani et al, “Attention is all you need,” NeurIPS 2017

The Transformer

Transformer Block

Input: Set of vectors x

All vectors interact through (multiheaded) Self-Attention



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

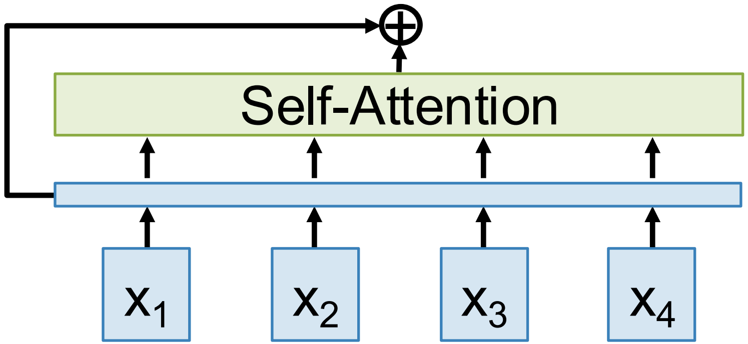
The Transformer

Transformer Block

Input: Set of vectors x

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

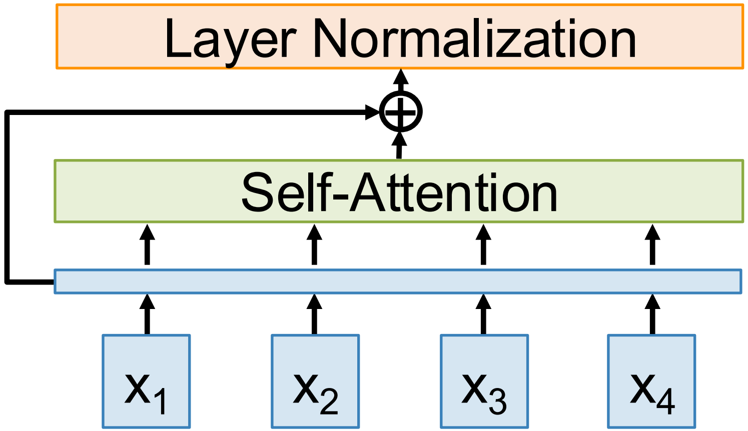
Recall **Layer Normalization**:
Given $h_1, \ldots, h_N$ (Shape: D)
scale: γ (Shape: D)
shift: β (Shape: D)
$\mu_i = (\sum_j h_{i,j})/D$ (scalar)
$\sigma_i = (\sum_j (h_{i,j} - \mu_i)^2/D)^{1/2}$ (scalar)
$z_i = (h_i - \mu_i) / \sigma_i$
$y_i = \gamma * z_i + \beta$

Ba et al, 2016

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

Usually a two-layer MLP; classic setup is
D => 4D => D

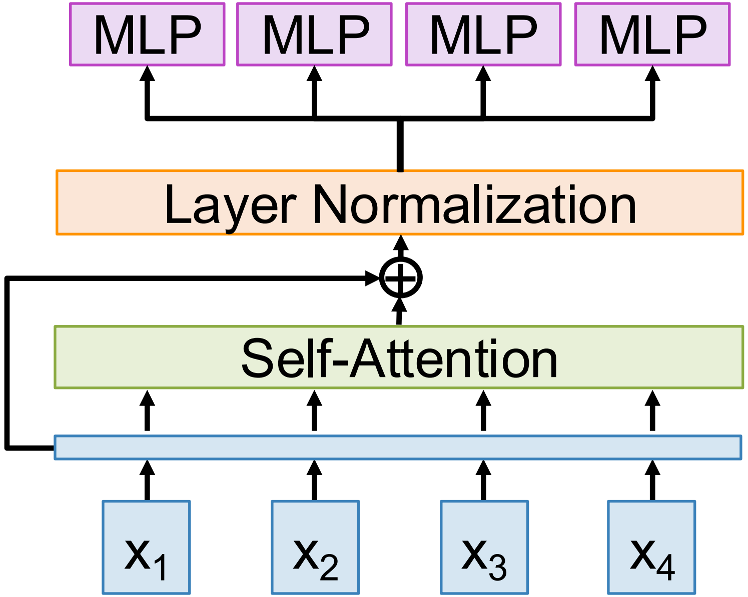
Also sometimes called FFN (Feed-Forward Network)

MLP independently on each vector

Layer normalization normalizes all vectors

Residual connection

All vectors interact through (multiheaded) Self-Attention

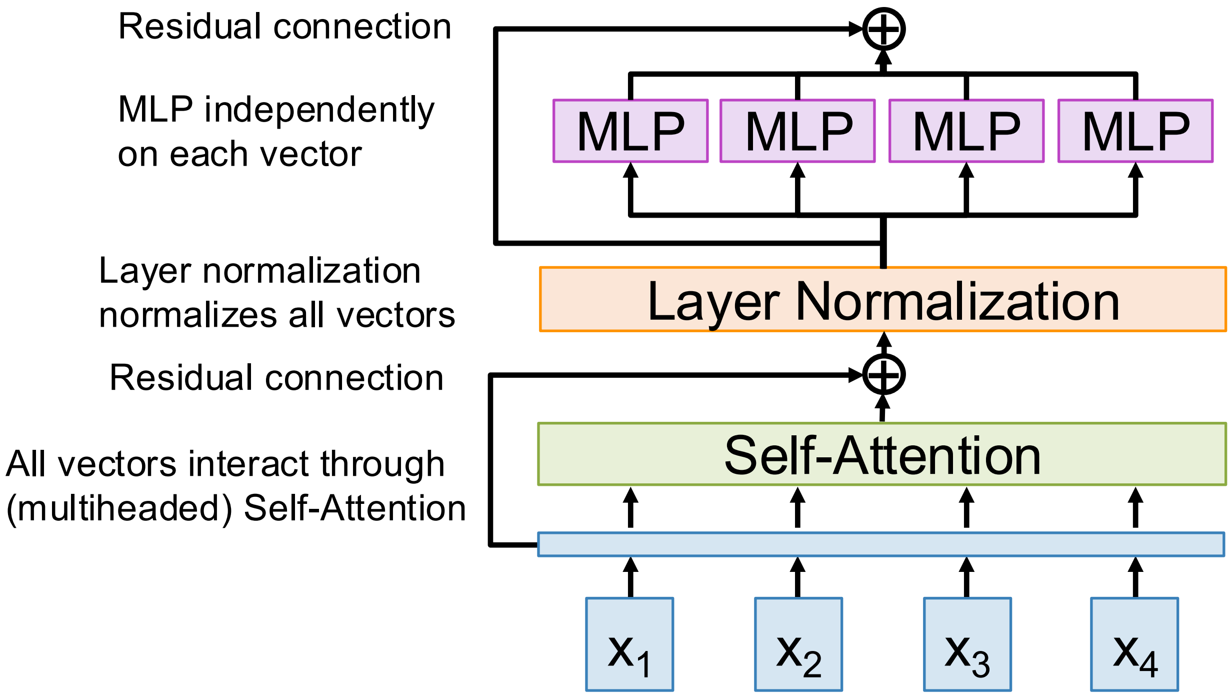


Vaswani et al, "Attention is all you need," NeurIPS 2017

The Transformer

Transformer Block

Input: Set of vectors x

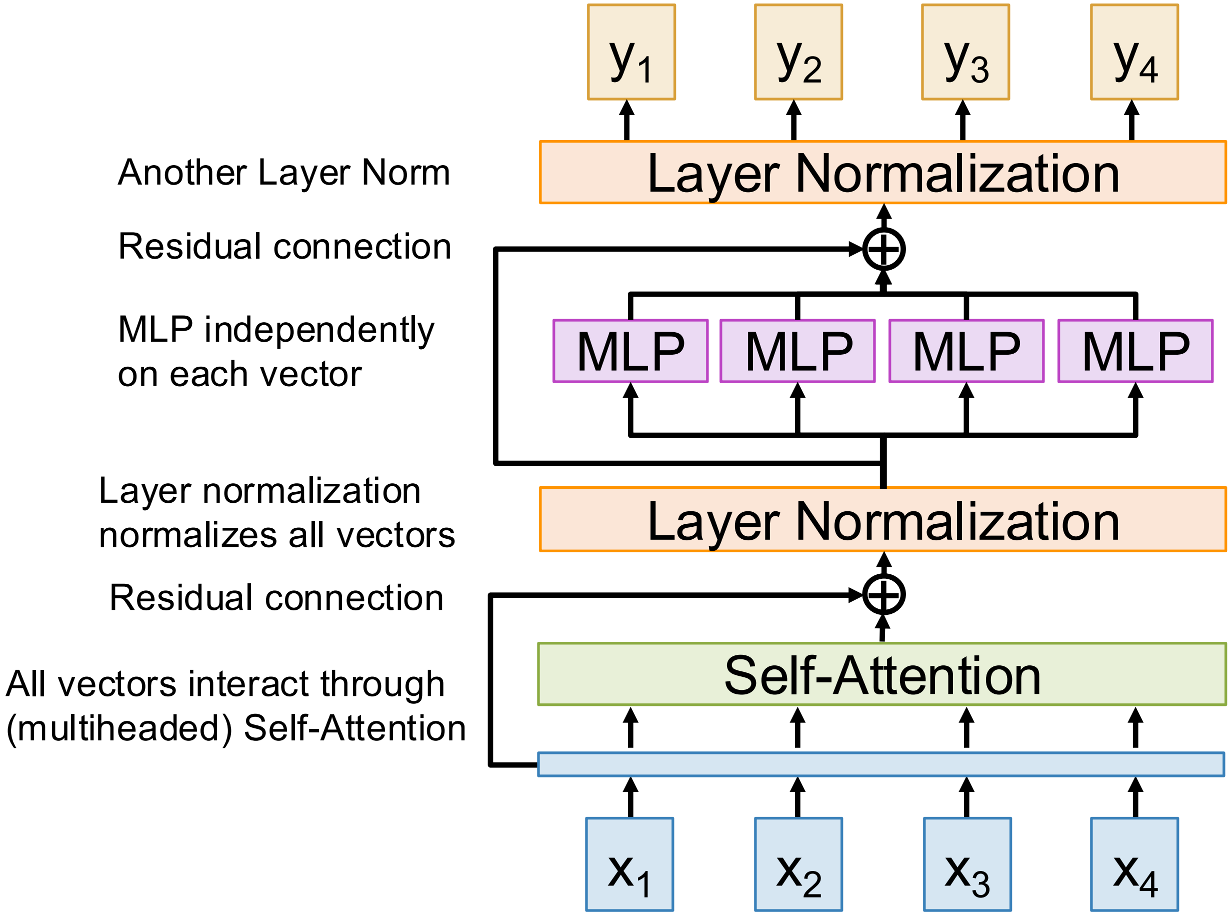


Vaswani et al, "Attention is all you need," NeurIPS 2017

The Transformer

Transformer Block

Input: Set of vectors x



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

The Transformer

Transformer Block

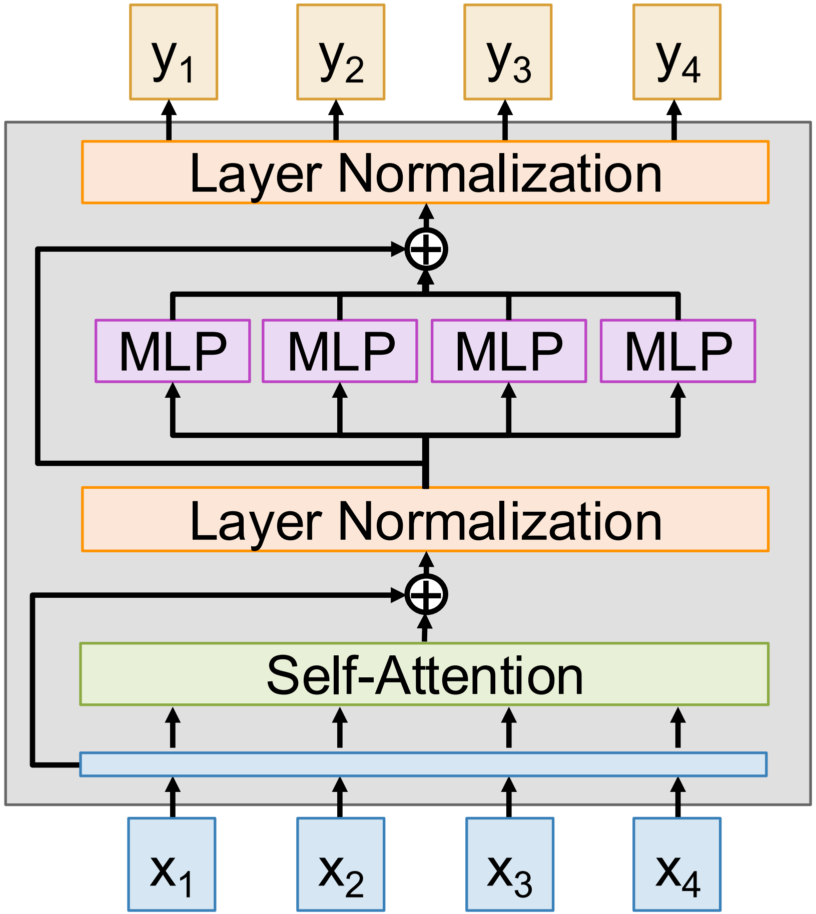
Input: Set of vectors x
Output: Set of vectors y

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



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

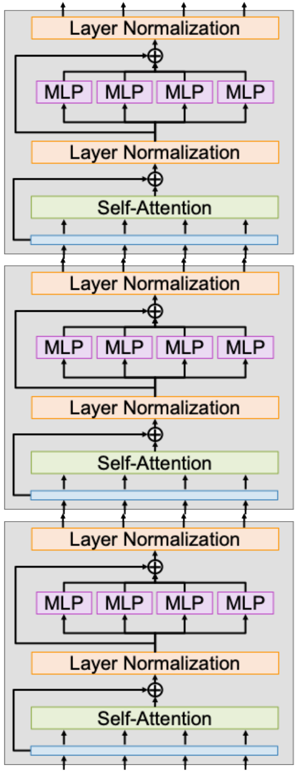
LayerNorm and MLP work on each vector independently

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

4 from Self-Attention
2 from MLP

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

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



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

The Transformer

<u>Transformer Block</u>

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

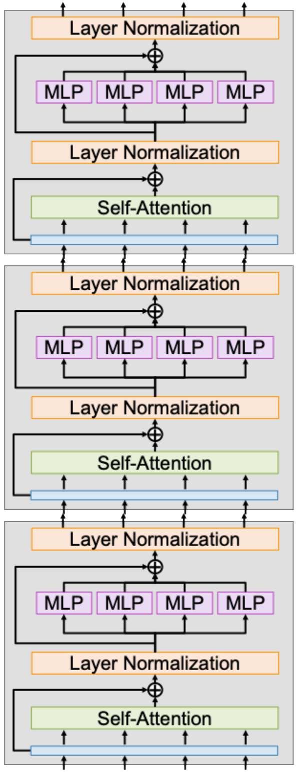
Highly scalable and parallelizable, most of the compute is just 6 matmuls:

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

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



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:

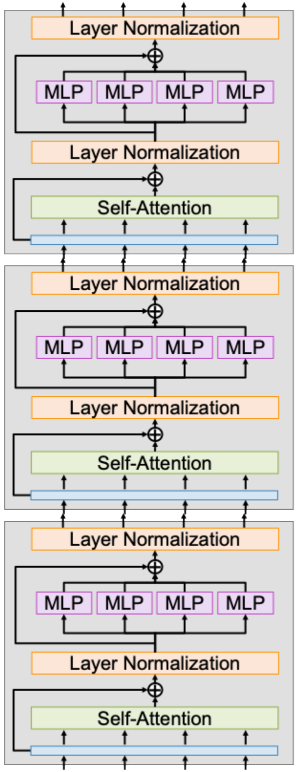
4 from Self-Attention
2 from MLP

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



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

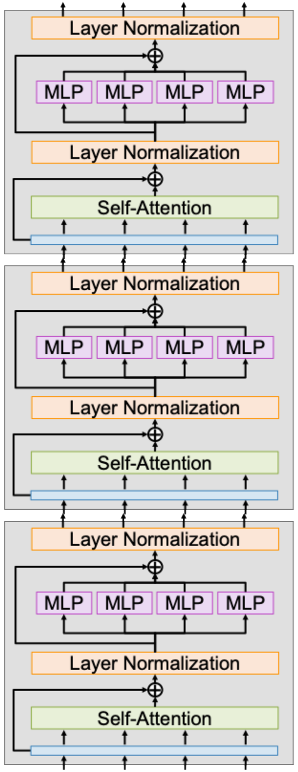
A **Transformer** is just a stack of identical Transformer blocks!

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

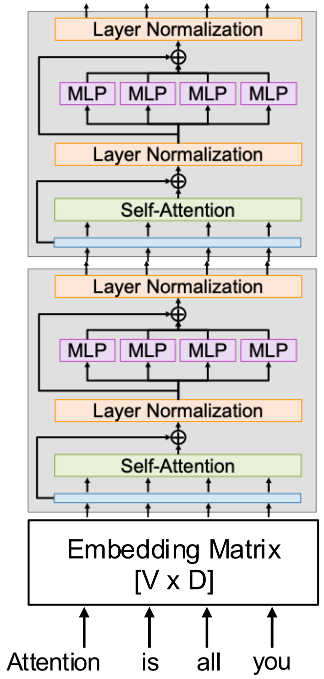


Vaswani et al, "Attention is all you need," NeurIPS 2017

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 x D]

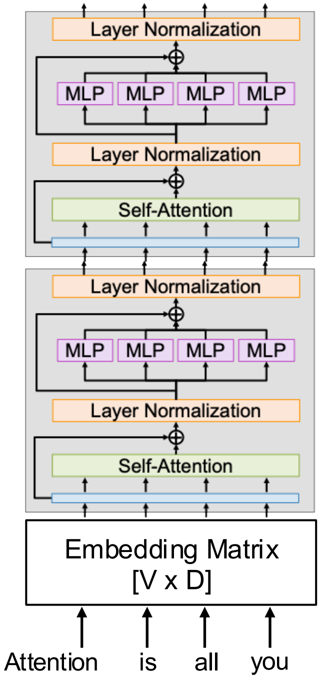


Transformers for Language Modeling (LLM)

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 x D]

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



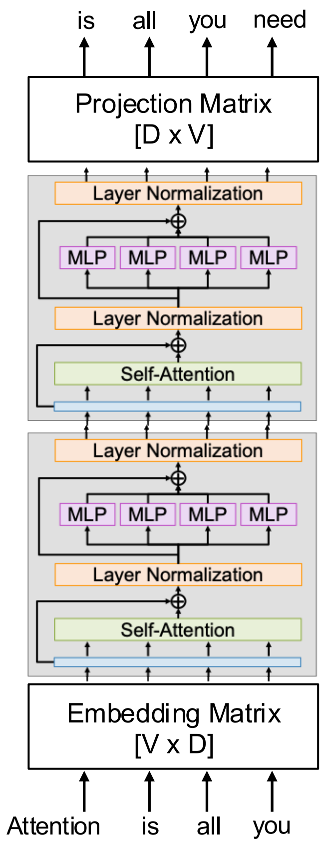
Transformers for Language Modeling (LLM)

Learn an <u>embedding matrix</u> 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 x D]

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

At the end, learn a <u>projection matrix</u> of shape [D x 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)

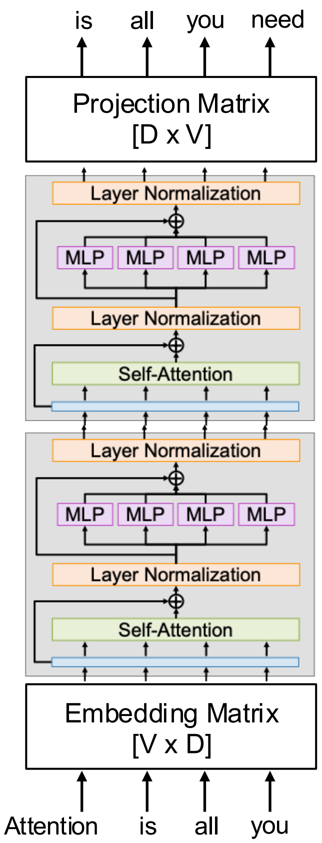
Learn an <u>embedding matrix</u> at the start of the model to convert words into vectors.

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

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

At the end, learn a <u>projection matrix</u> of shape [D x 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)

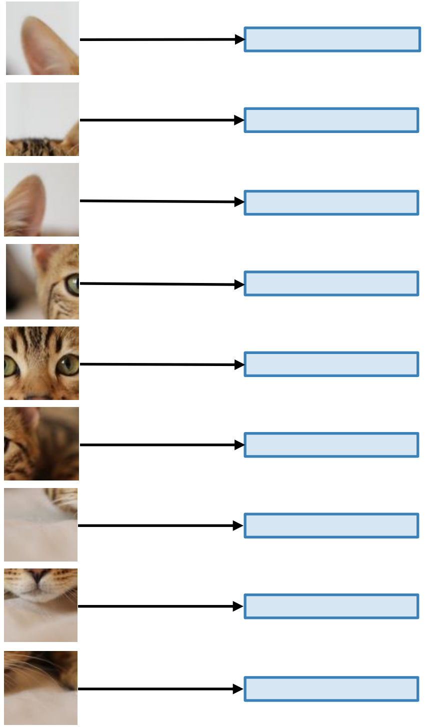
Input image:
e.g. 224x224x3

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



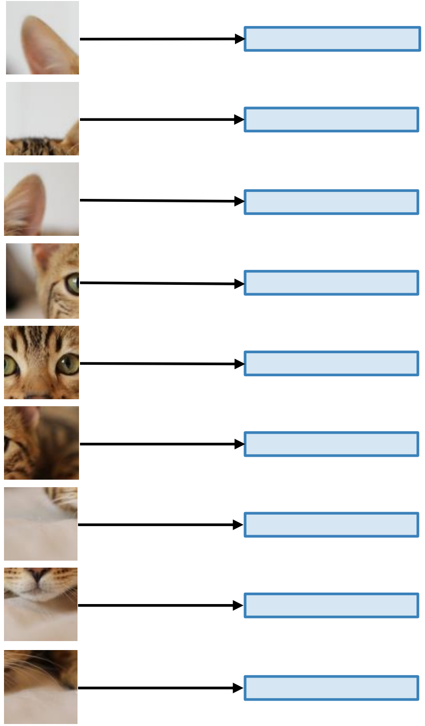
Break into patches
e.g. 16x16x3

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

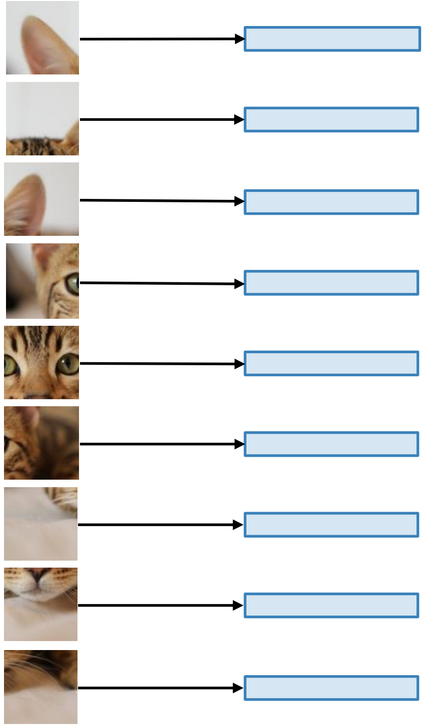
Flatten and apply a linear
transform 768 => D

Q: Any other way to
describe this operation?

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

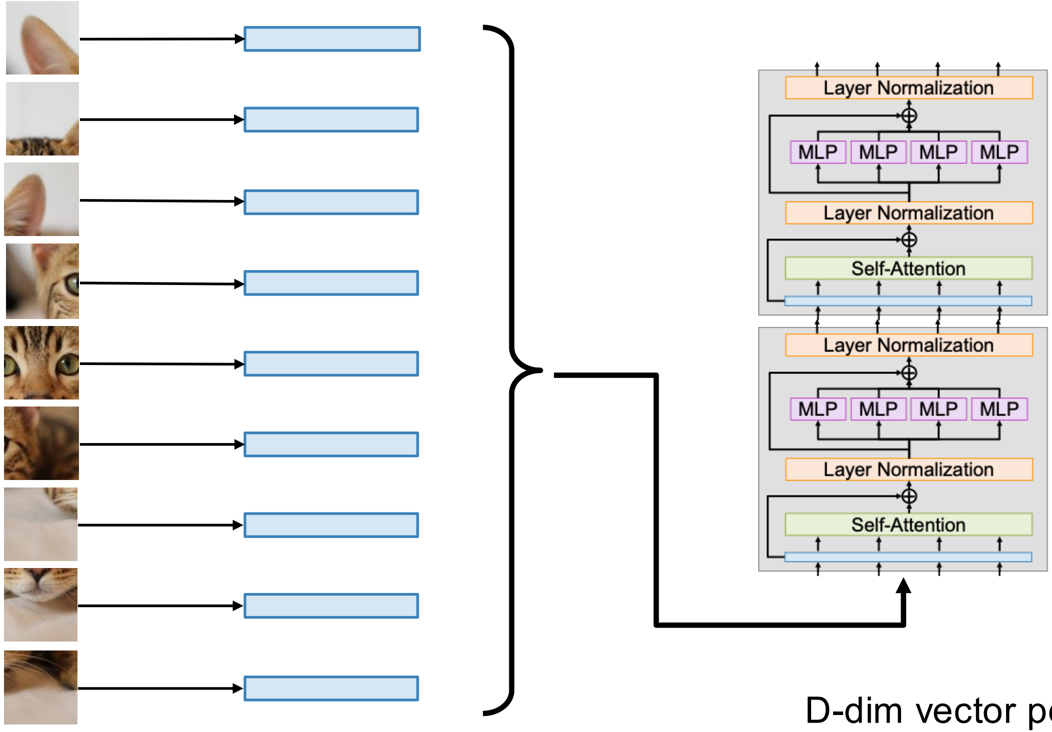
Q: Any other way to describe this operation?

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

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

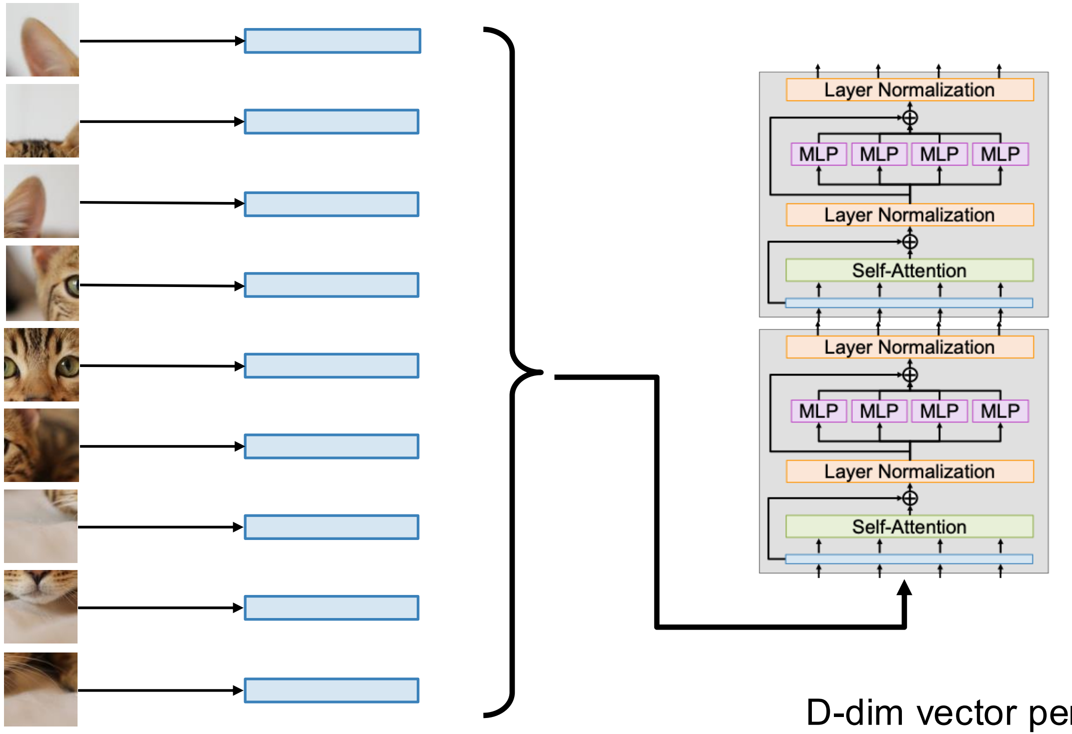
Flatten and apply a linear
transform 768 => D

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



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

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

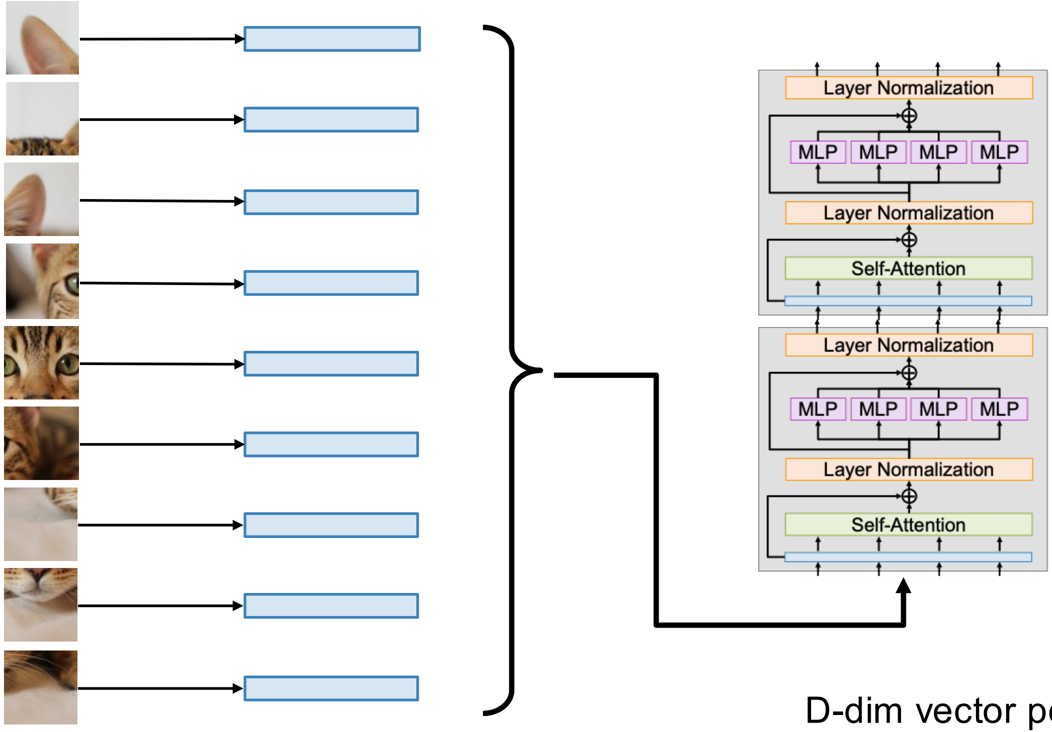
Break into patches
e.g. 16x16x3

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



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

Break into patches
e.g. 16x16x3

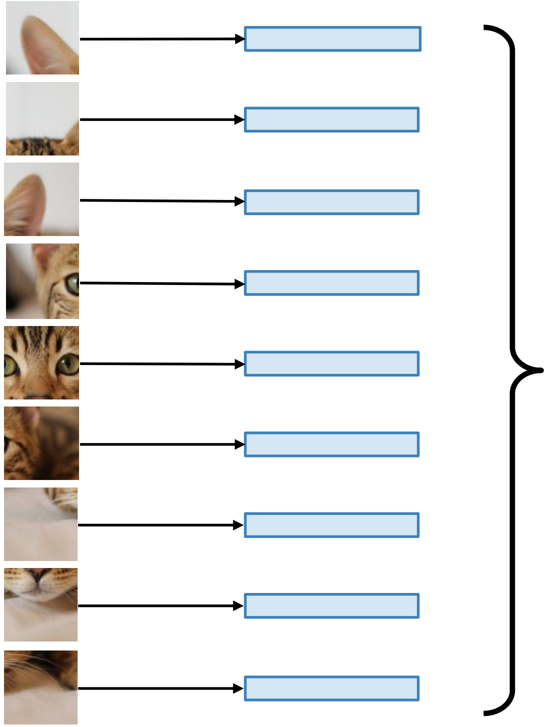
Flatten and apply a linear transform 768 => D

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

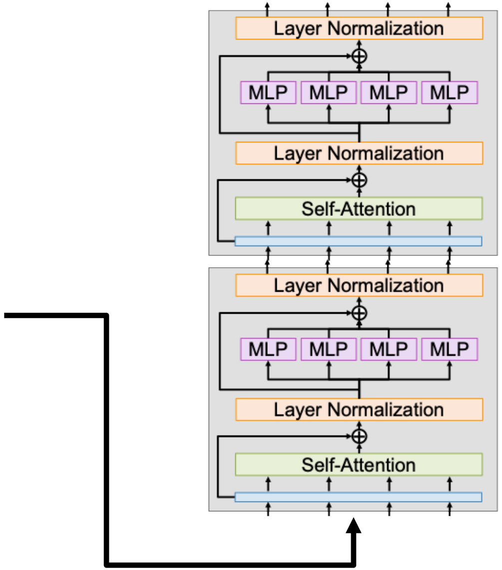
Vision Transformers (ViT)

Input image:
e.g. 224x224x3



Break into patches
e.g. 16x16x3

Flatten and apply a linear transform 768 => D



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

Transformer gives an output vector per patch

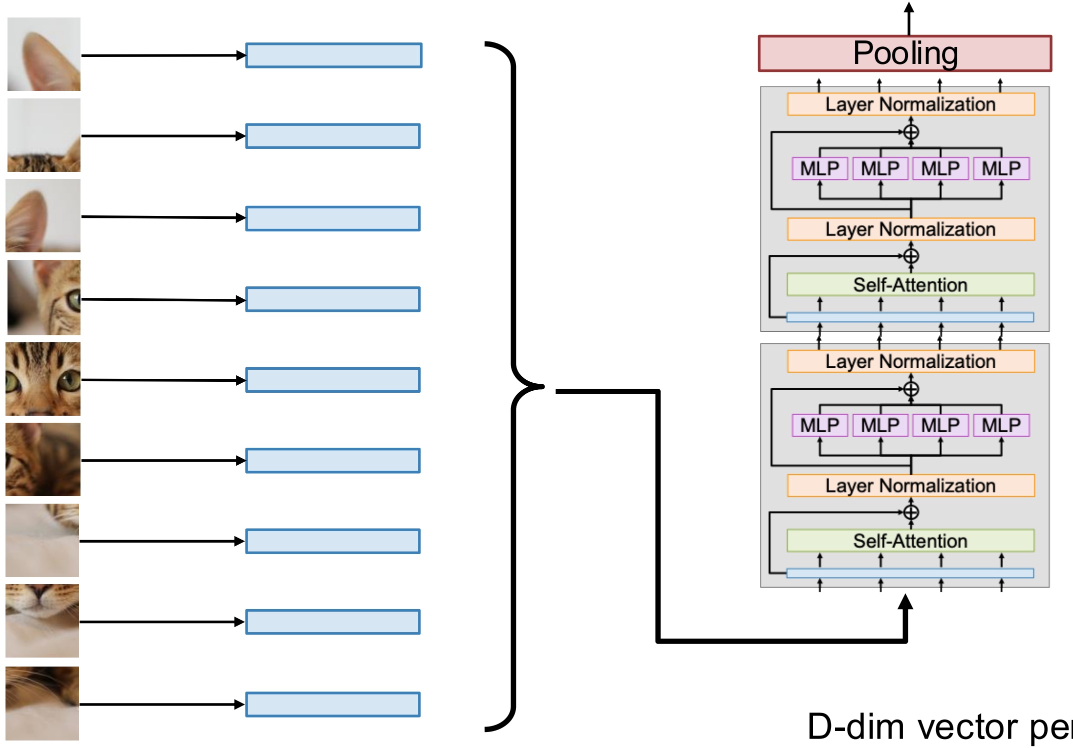
Don't use any masking; each image patch can look at all other image patches

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

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

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

Average pool NxD vectors to 1xD, apply a linear layer D=>C to predict class scores

Transformer gives an output vector per patch

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

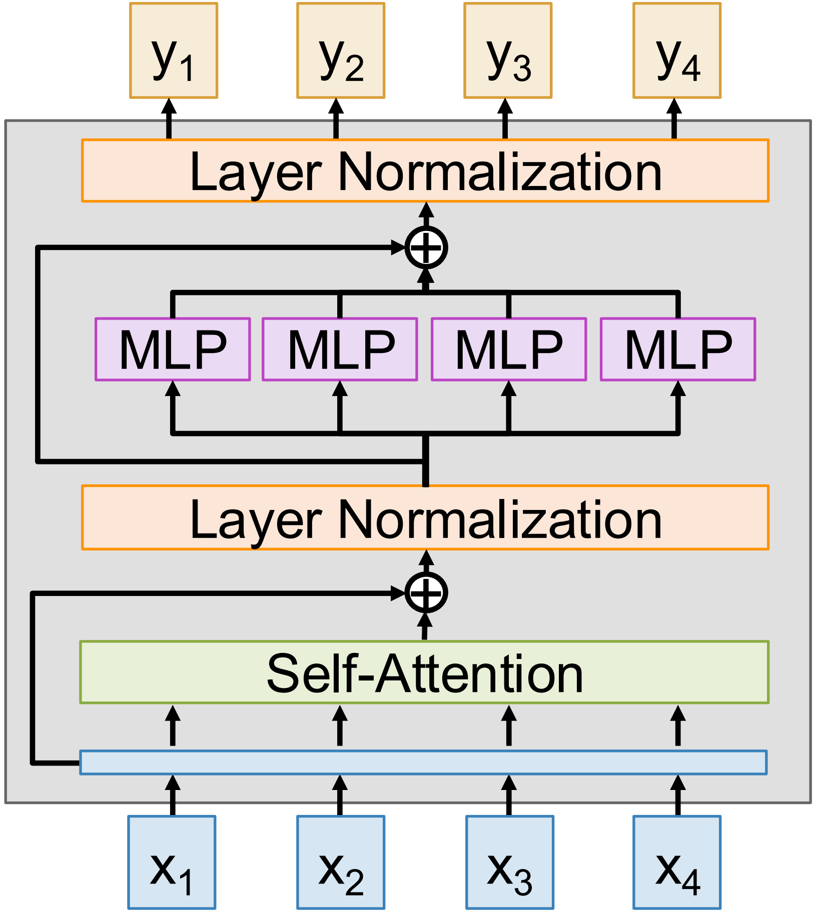
Use positional encoding to tell the transformer the 2D position of each patch

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

Tweaking Transformers

The Transformer architecture has not changed much since 2017.

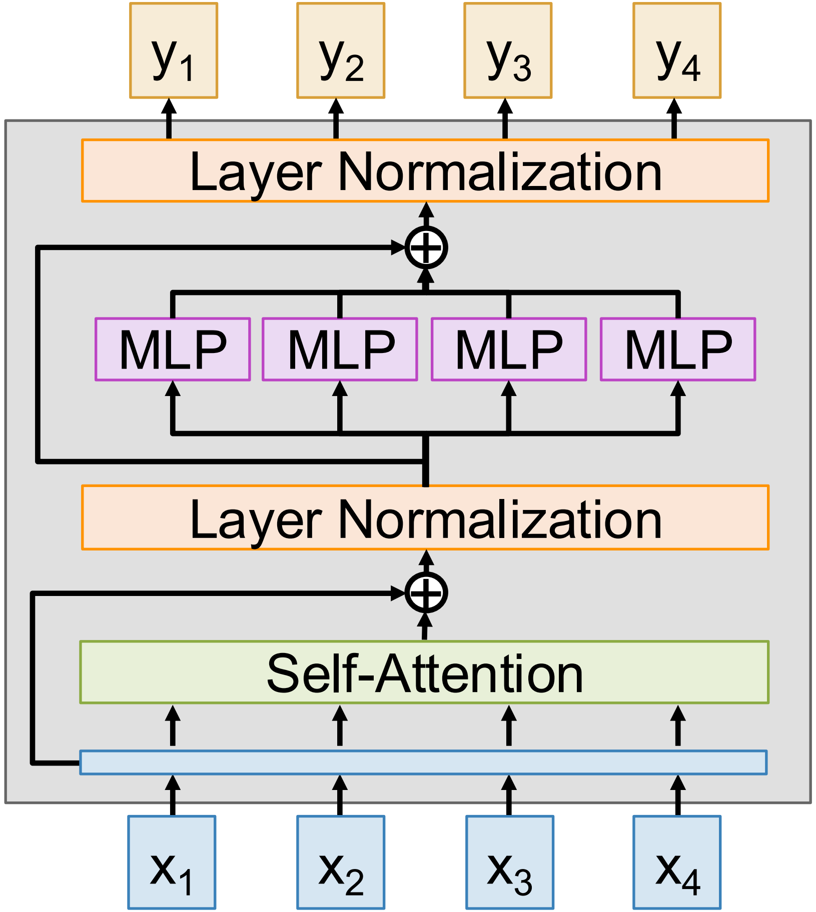
But a few changes have become common:



Pre-Norm Transformer

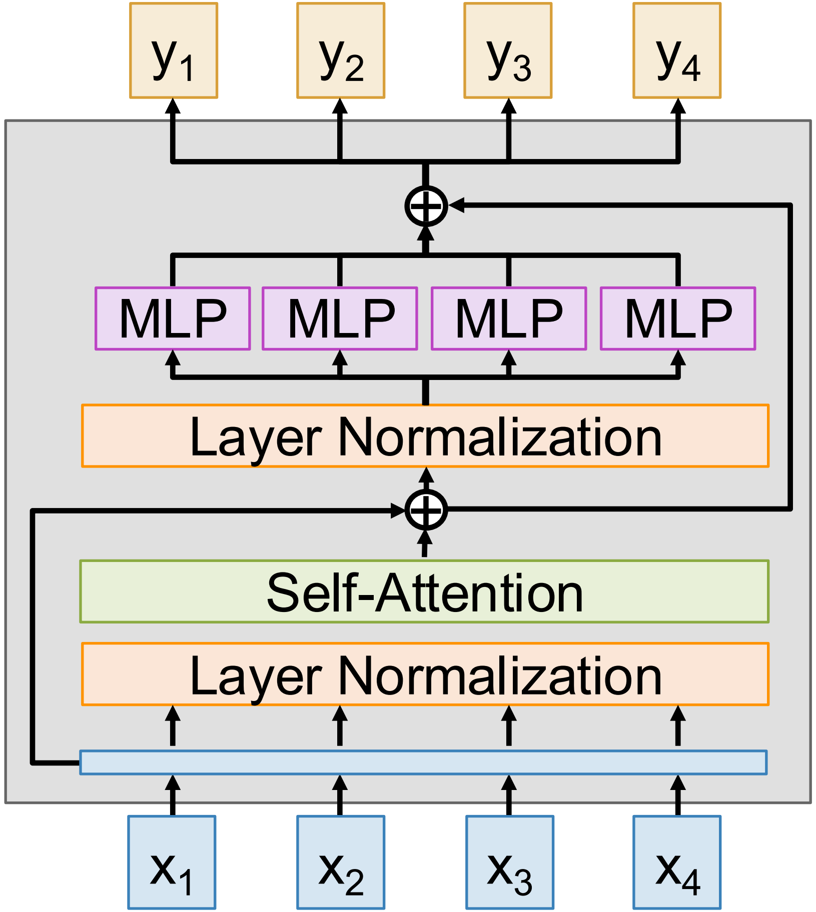
Layer normalization is outside the residual connections

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



Baevski & Auli, "Adaptive Input Representations for Neural Language Modeling", arXiv 2018

Pre-Norm Transformer



Layer normalization is outside the residual connections

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

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

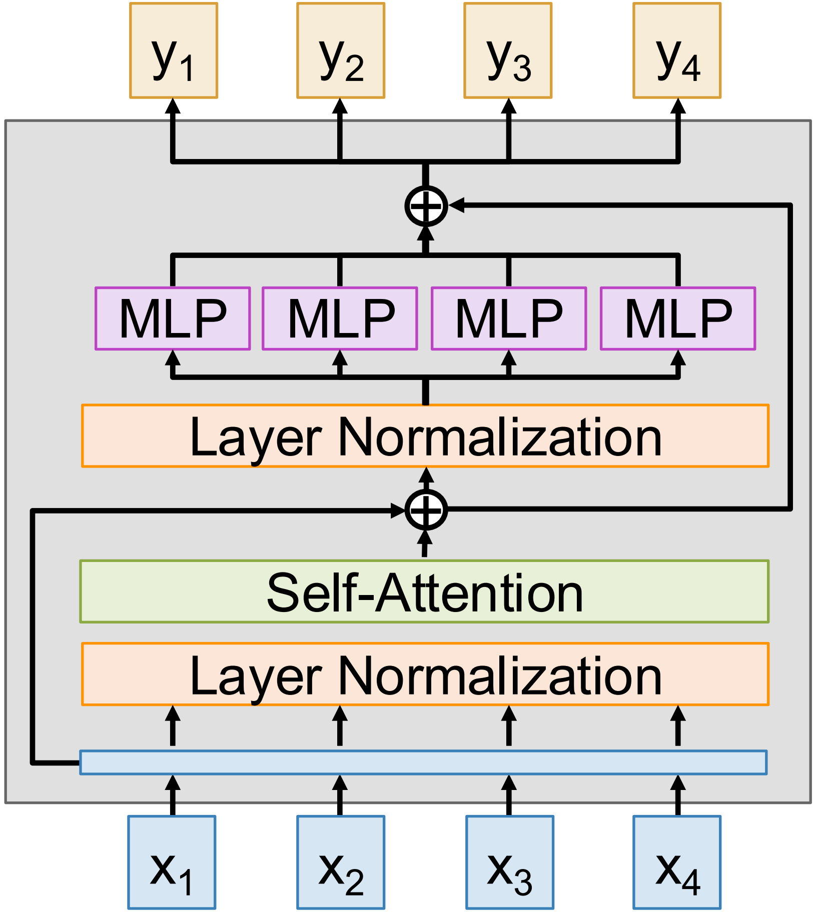
Baevski & Auli, "Adaptive Input Representations for Neural Language Modeling", arXiv 2018

QK-Norm

Normalize queries and keys before computing attention similarities

Queries: $\mathbf{Q}$ = normalize($\mathbf{XW_Q}$) [H x N x D_H]
Keys: $\mathbf{K}$ = normalize($\mathbf{XW_K}$) [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]

Prevents gradient spikes, stabilizes training



Zhang and Sennrich, "Root Mean Square Layer Normalization", NeurIPS 2019

QK-Norm

Normalize queries and keys before computing attention similarities

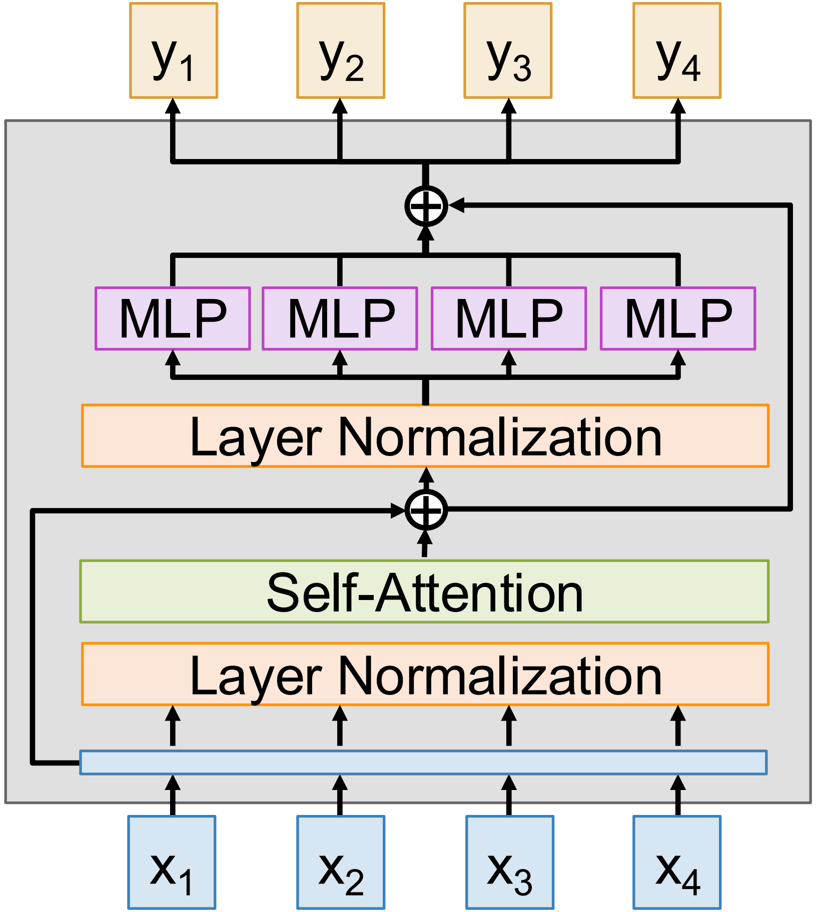
Queries: **Q** = normalize(**XW$_Q$**) [H x N x D_H]
Keys: **K** = normalize(**XW$_K$**) [H x N x D_H]
Values: **V** = **XW$_V$** [H x N x D_H]
Similarities: **E** = **QK^T** $/\sqrt{D_Q}$ [H x N x N]

Prevents gradient spikes, stabilizes training

Normalize with RMSNorm:

$$y_i = \frac{x_i}{RMS(x)} * \gamma_i$$

$$RMSNorm(x) = \sqrt{\varepsilon + \frac{1}{N}\sum_{i=1}^{N} x_i^2}$$

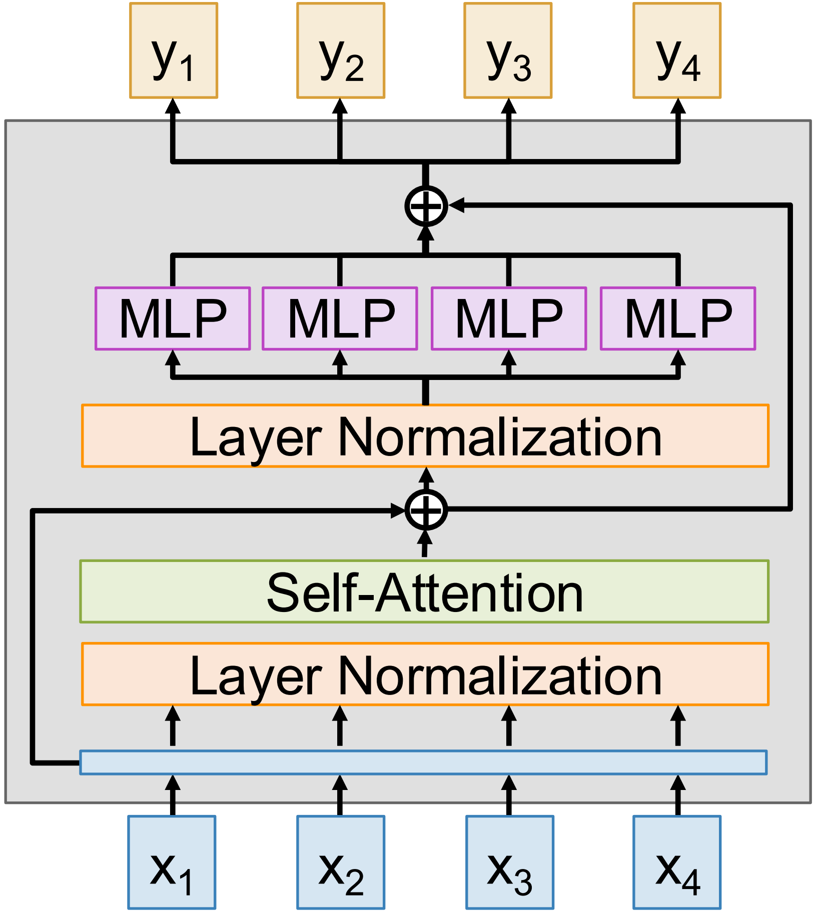


Zhang and Sennrich, "Root Mean Square Layer Normalization", NeurIPS 2019

SwiGLU MLP

Classic MLP:

Input: X [N x D]
Weights: W_1 [D x 4D]
W_2 [4D x D]
Output: $Y = \sigma(XW_1)W_2$ [N x D]



Shazeer, "GLU Variants Improve Transformers", 2020

SwiGLU MLP

Classic MLP:

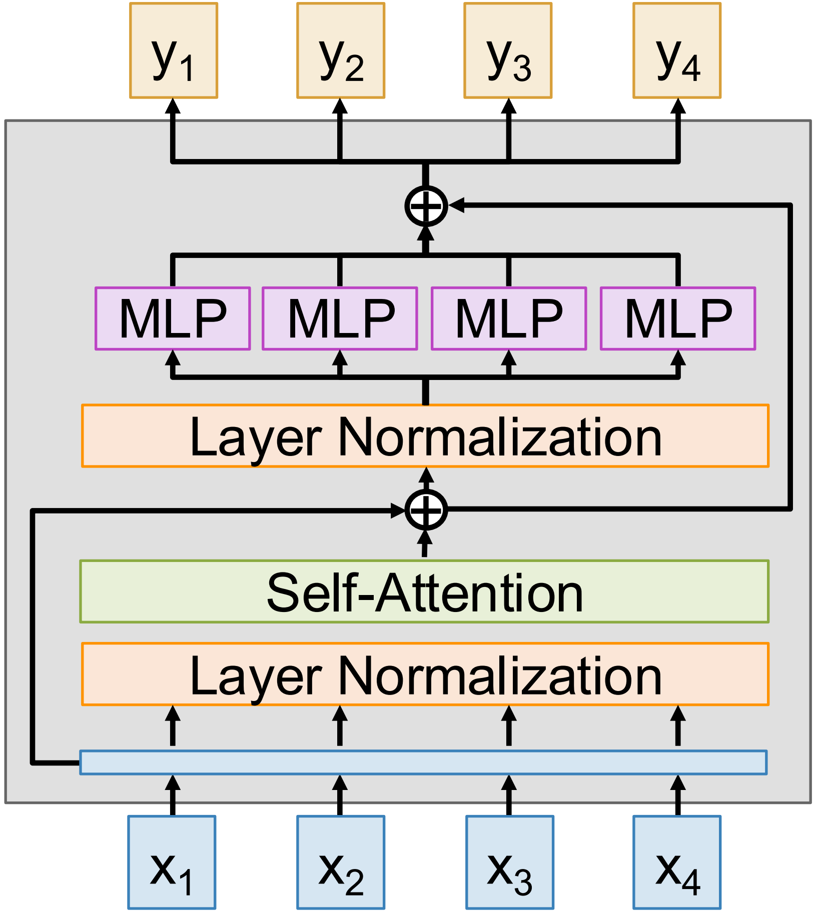
Input: X [N x D]
Weights: W_1 [D x 4D]
W_2 [4D x D]
Output: $Y = \sigma(XW_1)W_2$ [N x D]

SwiGLU MLP:

Input: X [N x D]
Weights: W_1 , W_2 [D x H]
W_3 [H x D]
Output:

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

Setting H = 8D/3 keeps same total params



Shazeer, "GLU Variants Improve Transformers", 2020

SwiGLU MLP

Classic MLP:

Input: X [N x D]
Weights: W_1 [D x 4D]
W_2 [4D x D]
Output: $Y = \sigma(XW_1)W_2$ [N x D]

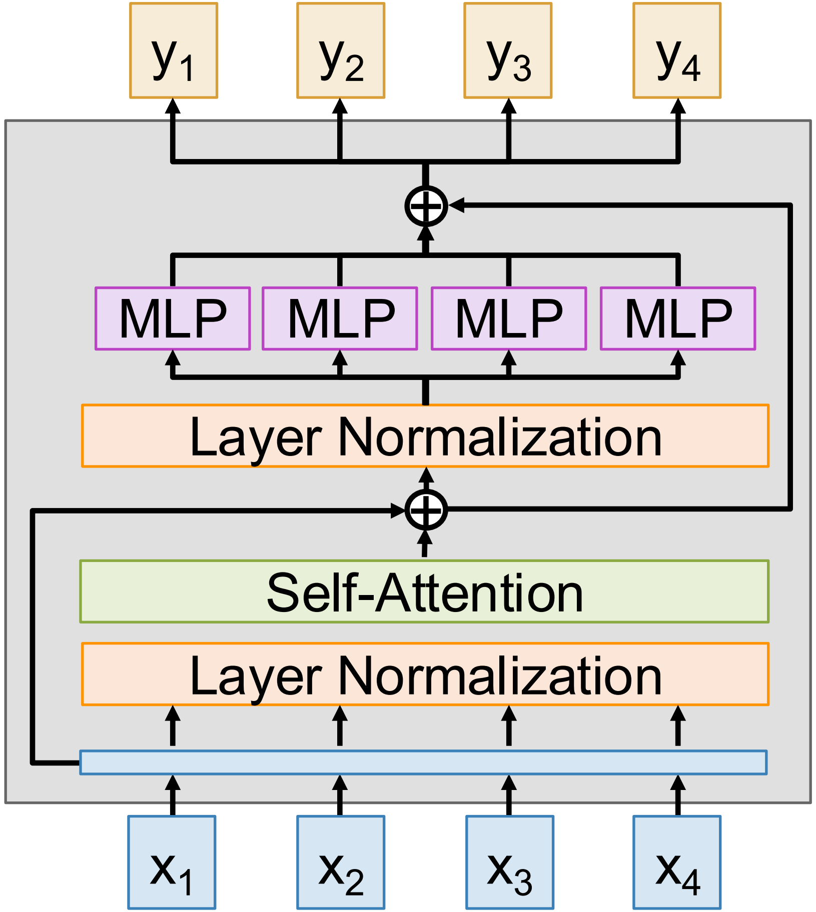
SwiGLU MLP:

Input: X [N x D]
Weights: W_1 , W_2 [D x H]
W_3 [H x 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.

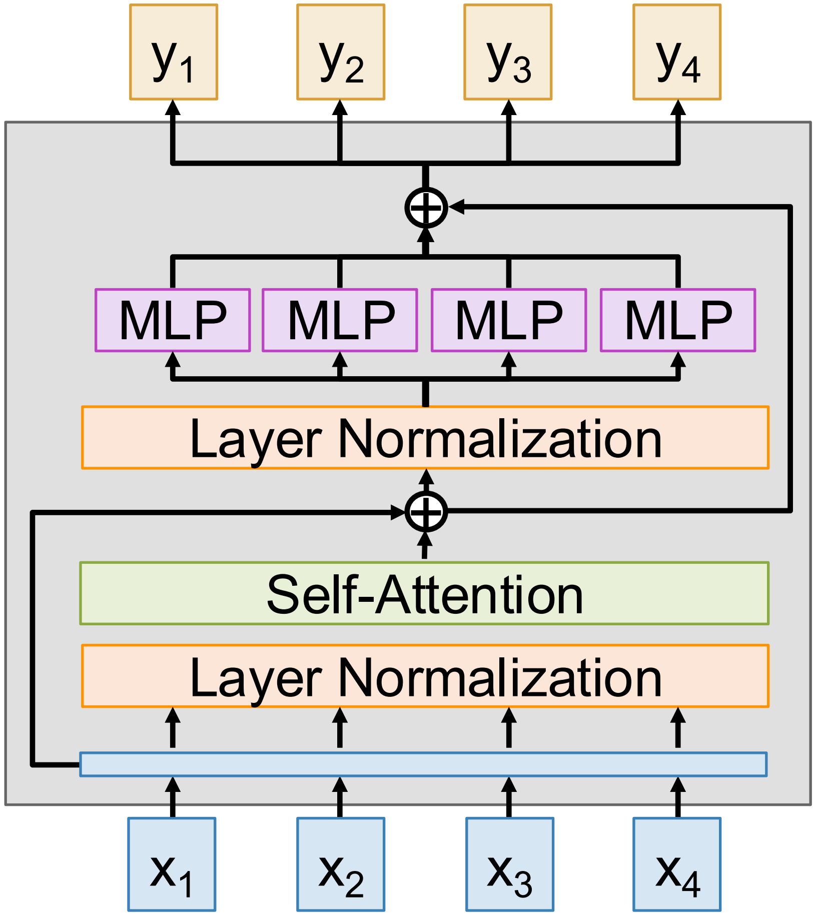


Shazeer, "GLU Variants Improve Transformers", 2020

Mixture of Experts (MoE)

Learn E separate sets of MLP weights in each block; each MLP is an *expert*

W_1: [D x 4D] => [E x D x 4D]
W_2: [4D x D] => [E x 4D x D]



Shazeer et al, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer", 2017

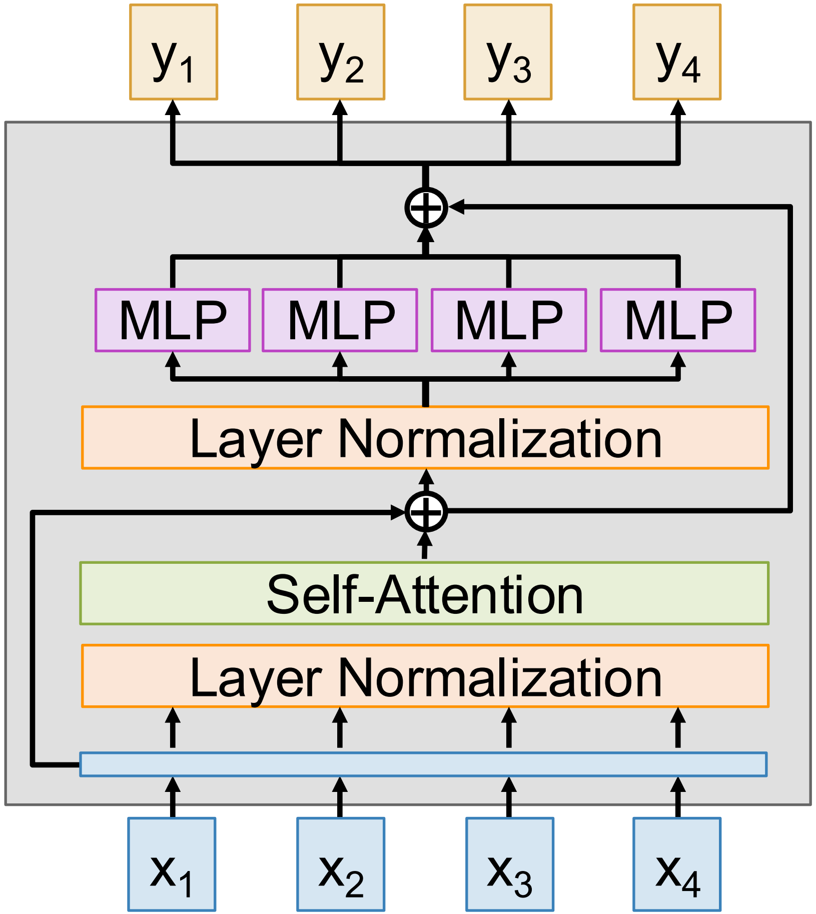
Mixture of Experts (MoE)

Learn E separate sets of MLP weights in each block; each MLP is an *expert*

W_1: [D x 4D] => [E x D x 4D]
W_2: [4D x D] => [E x 4D x 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



Shazeer et al, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer", 2017

Mixture of Experts (MoE)

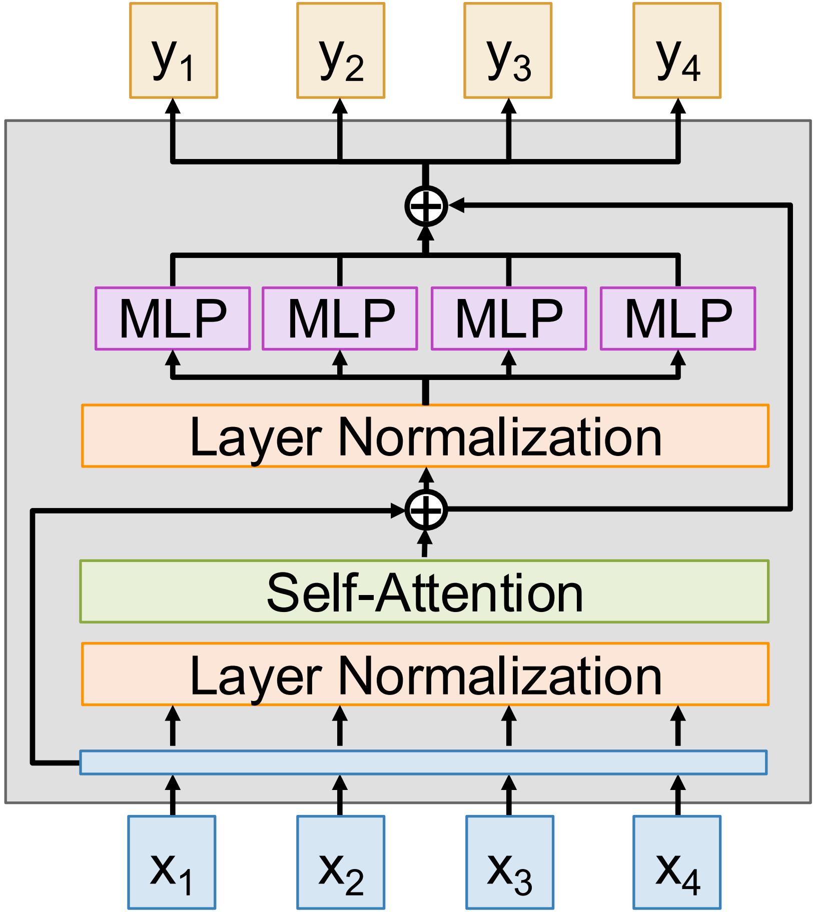
Learn E separate sets of MLP weights in each block; each MLP is an *expert*

W_1: [D x 4D] => [E x D x 4D]
W_2: [4D x D] => [E x 4D x 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

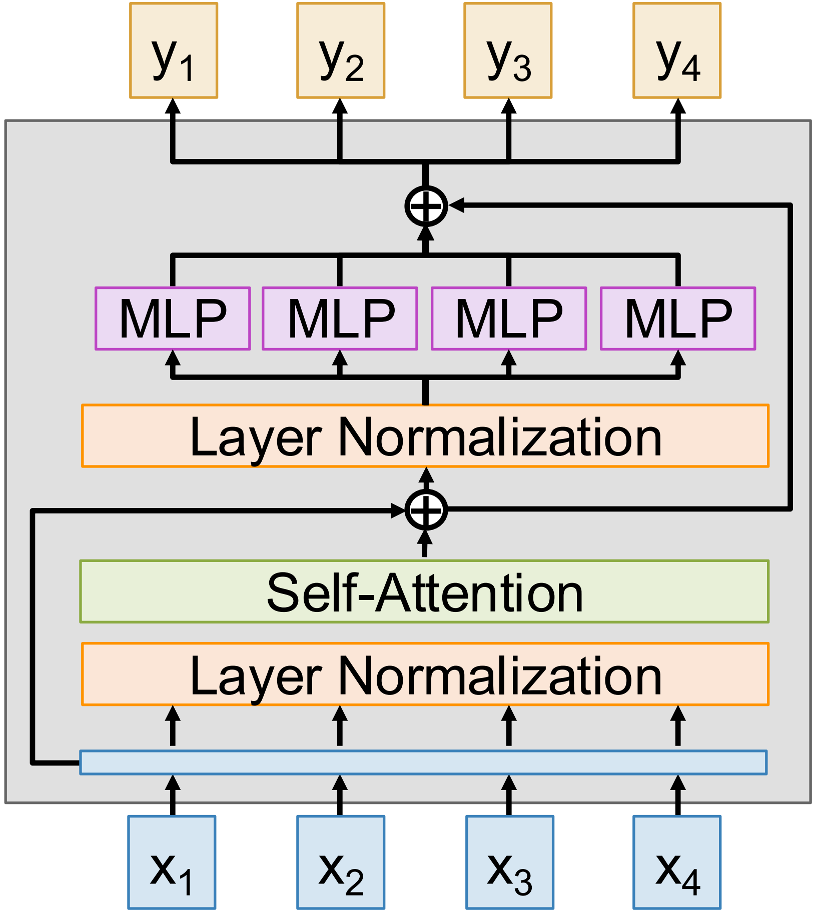


Tweaking Transformers

The Transformer architecture has not changed much since 2017.

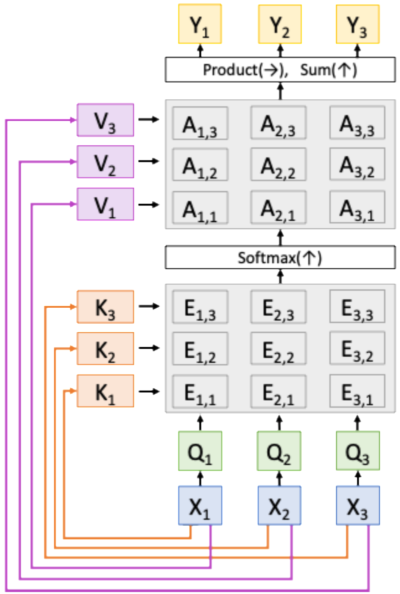
But a few changes have become common:

- **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

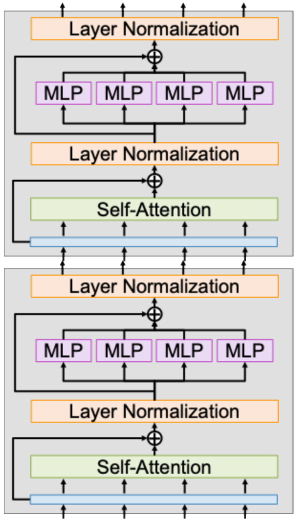
Attention: A new primitive that operates on sets of vectors



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