CS 4644-DL / 7643-A: LECTURE 13
DANFEI XU

Attention for Sequence Modeling

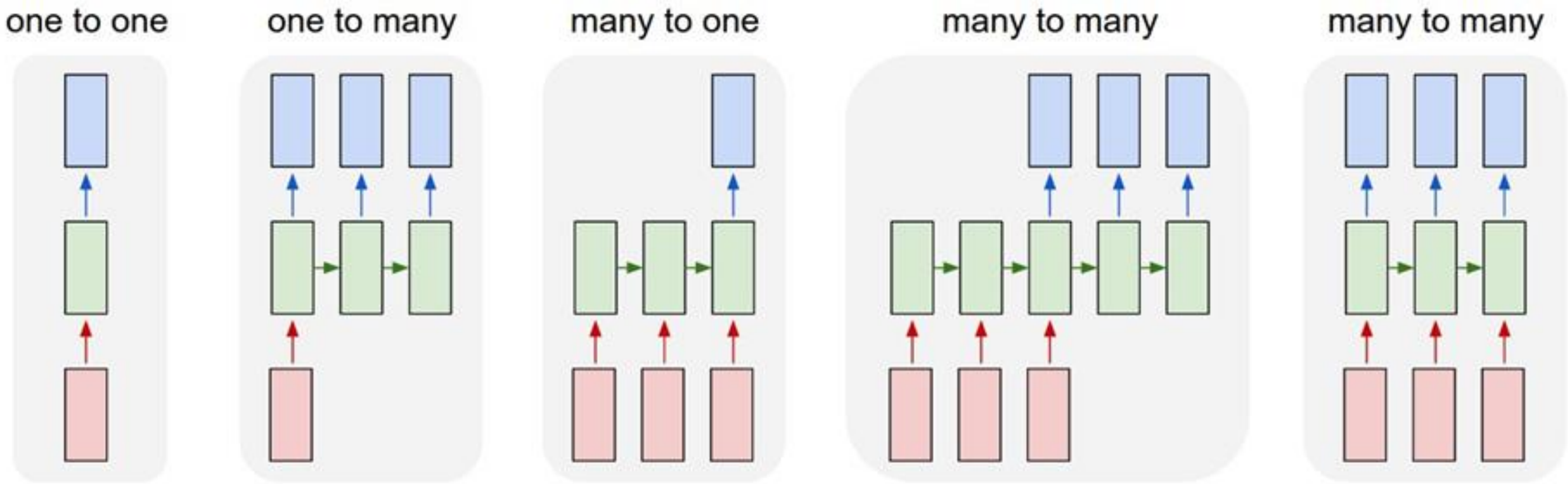
Attention is (Mostly) All you Need: Transformers

Administrative:

HW 3 due 10/20

Milestone report due by 11/3: Need baseline results (qualitative and quantitative)!

Recurrent Neural Networks: Process Sequences



RNN hidden state update

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

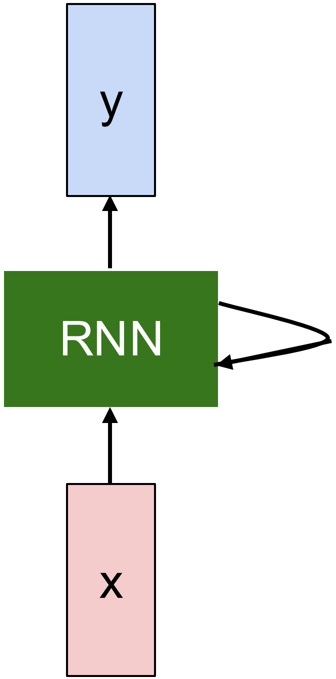
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state (vector)

old state (vector)

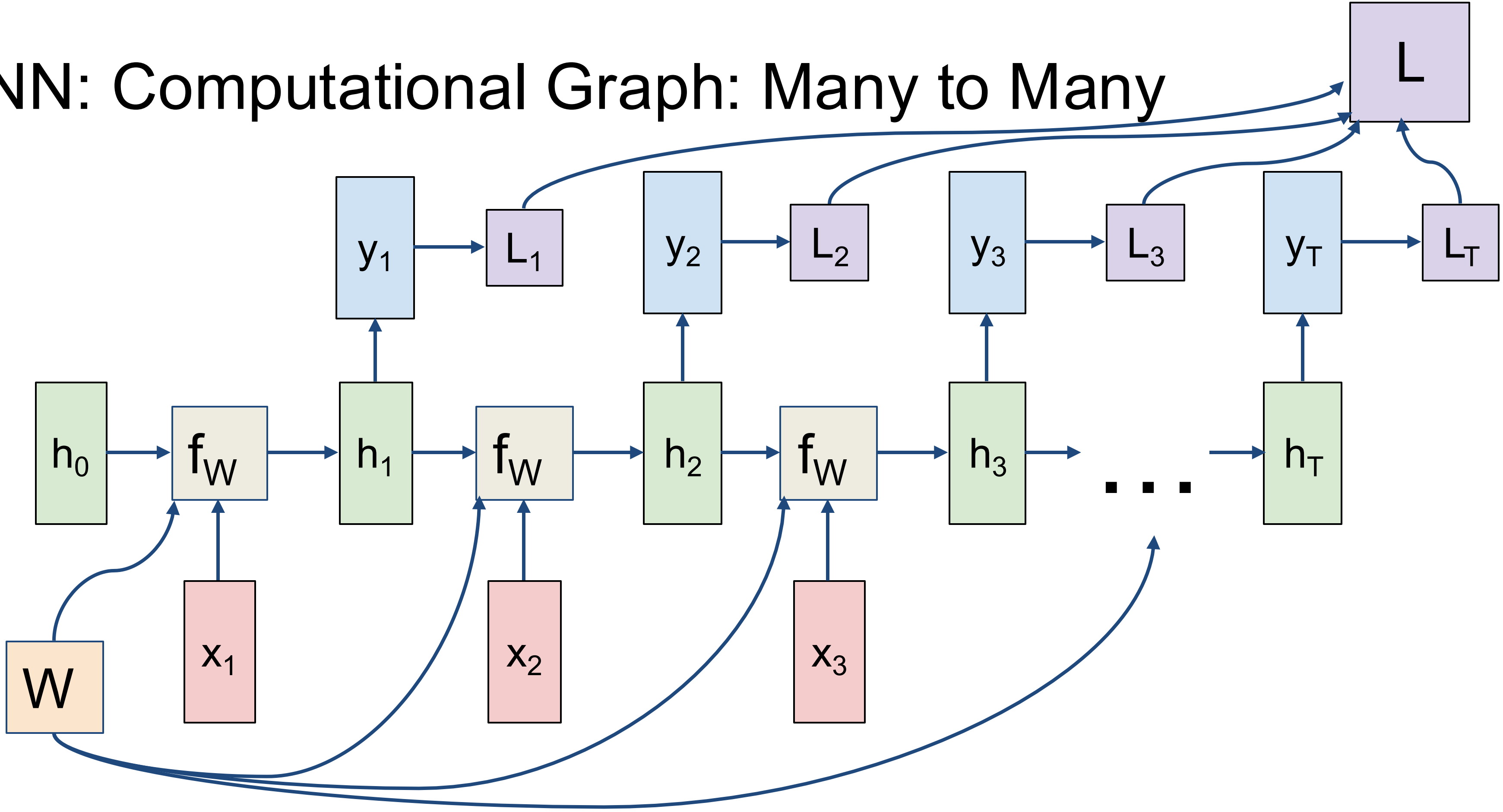
input vector at some time step

some function with parameters W

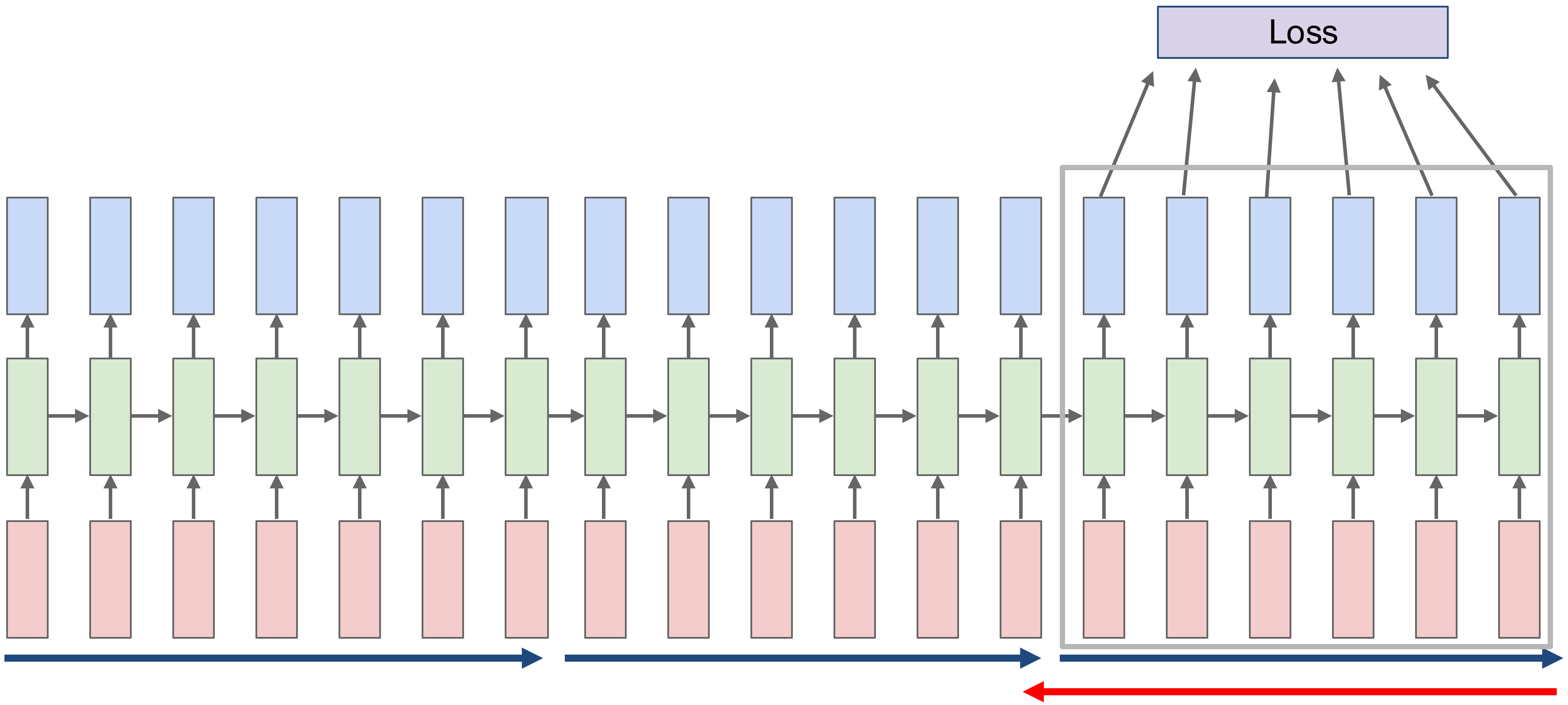


Can set initial state h_0 to all 0's

RNN: Computational Graph: Many to Many



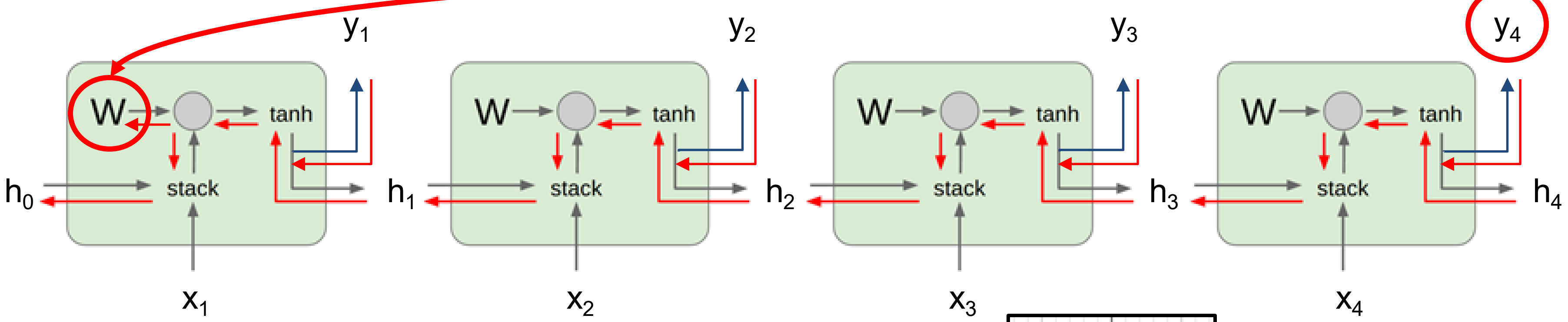
Truncated Backpropagation through time



Vanilla RNN Gradient Flow

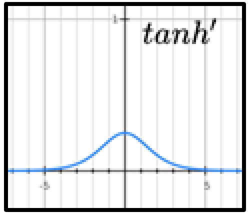
Gradients over multiple time steps:

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

Always < 1
Vanishing gradients

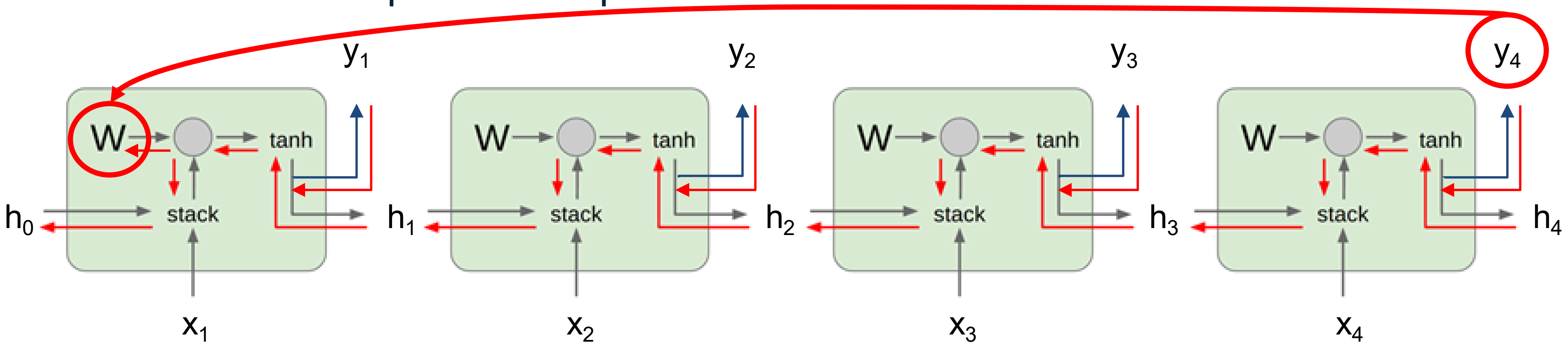


$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T}\left(\prod_{t=2}^{T} \boxed{tanh'(W_{hh}h_{t-1} + W_{xh}x_t)}\right)W_{hh}^{T-1}\frac{\partial h_1}{\partial W}$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Gradients over multiple time steps:



$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{hh}^{T-1}} \frac{\partial h_1}{\partial W}$$

What if we assumed no non-linearity?

Largest eigen value > 1:
Exploding gradients

Largest eigen value < 1:
Vanishing gradients

→ We need a new RNN architecture!

Long-Short Term Memory (Incomplete)

RNN directly updates h_t through **multiplying** with a weight matrix:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$\boxed{\frac{\partial h_t}{\partial h_{t-1}} = tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}}$$

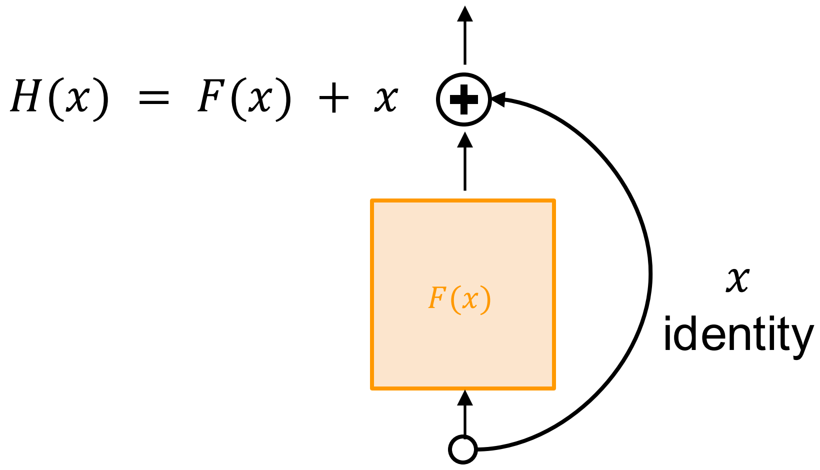
LSTM keeps track of a cell state c_t that's updated through **addition**

$$c_t = c_{t-1} + \tilde{c}$$
$$\tilde{c} = \tanh(W[h_{t-1}, x_t]), h_t = \tanh(c_t)$$

Gradient of cell state: $\frac{\partial c_t}{\partial c_{t-1}} = 1 + \frac{\partial \tilde{c}}{\partial c_{t-1}}$

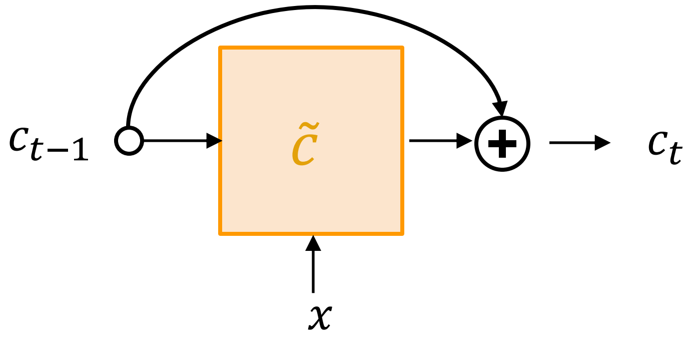
This should look familiar …
Residual connection!

Long-Short Term Memory (Incomplete)



Residual block

$$c_t = c_{t-1} + \tilde{c}$$



LSTM Cell Update

$$\tilde{c} = \tanh(W[h_{t-1}, x_t]), h_t = \tanh(c_t)$$

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

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

Learn to control information flow from previous state to the next state

Hochreiter and Schmidhuber, "Long Short Term Memory", Neural Computation 1997

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

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

Learn to control information flow from previous state to the next state

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997

Long Short Term Memory (LSTM)

Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$\boxed{\begin{aligned} c_t &= f \odot c_{t-1} + i \odot g \\ h_t &= o \odot \tanh(c_t) \end{aligned}}$$

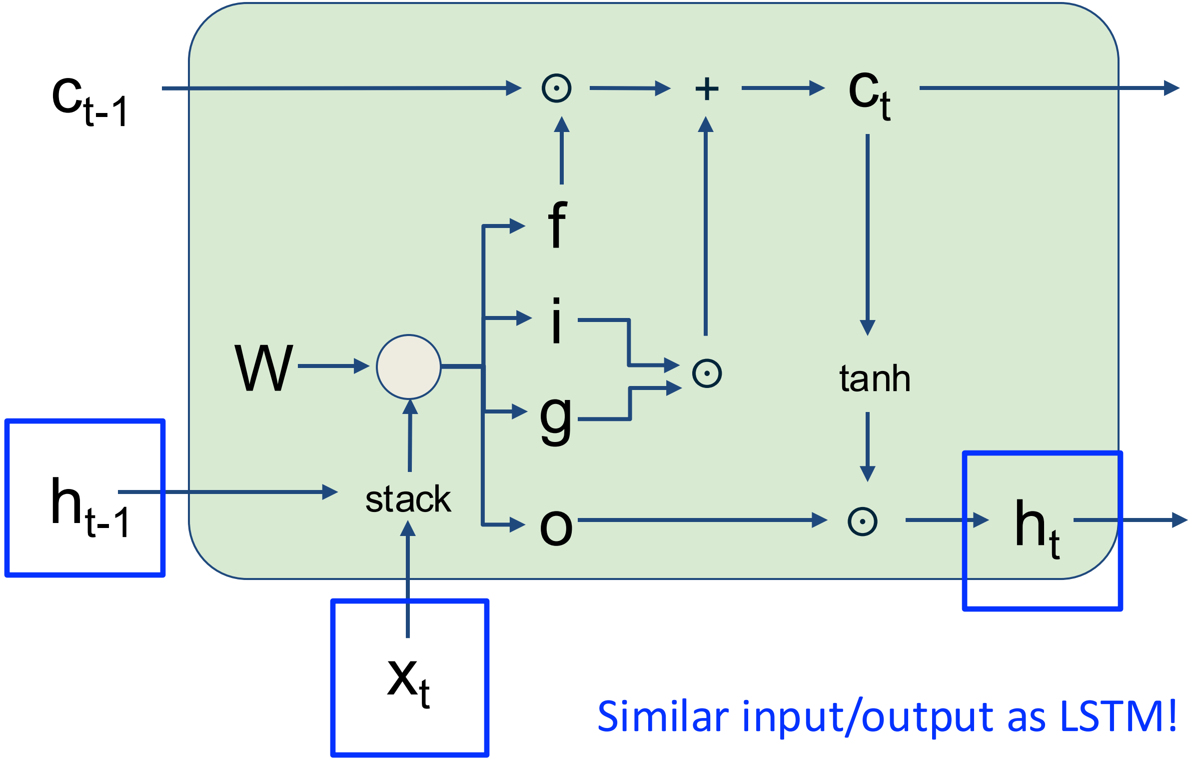
Long-term memory *c* determines how much information should go into the hidden state *h* (short-term memory)

Two “memory vectors”

Hochreiter and Schmidhuber, “Long Short Term Memory”, Neural Computation 1997

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]



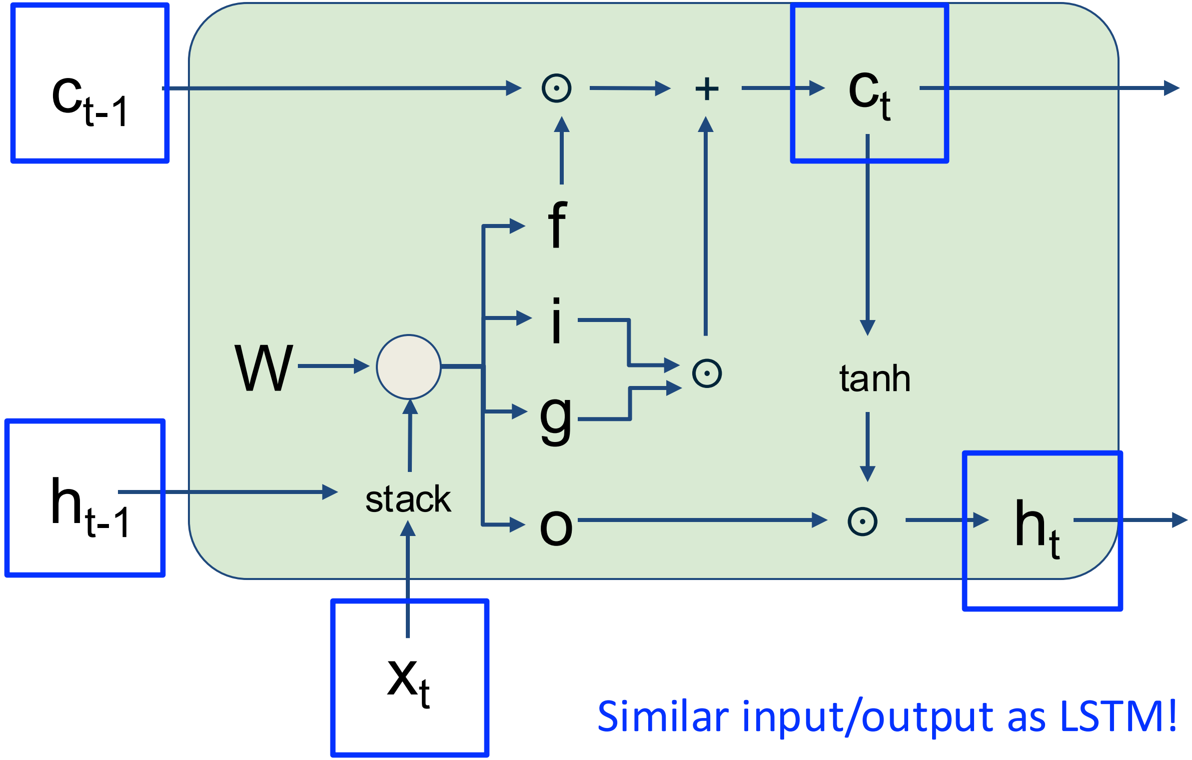
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

Similar input/output as LSTM!

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

Keep long-term memory cell c in addition to short term memory h

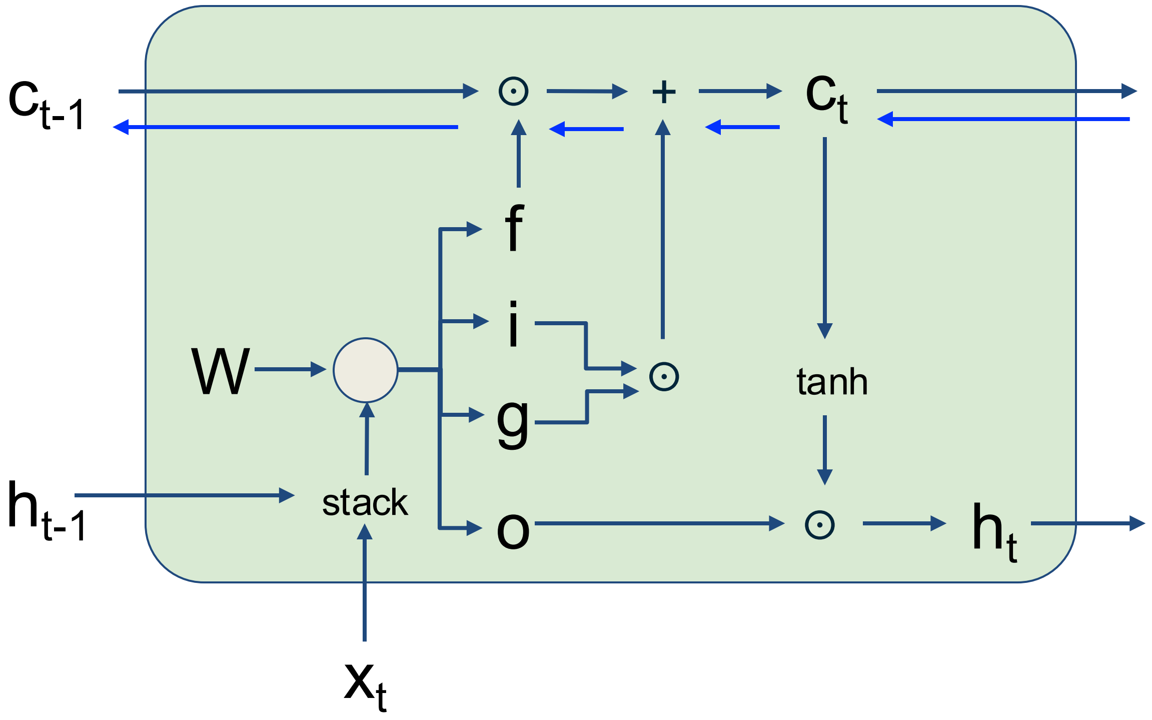


$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

Similar input/output as LSTM!

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]



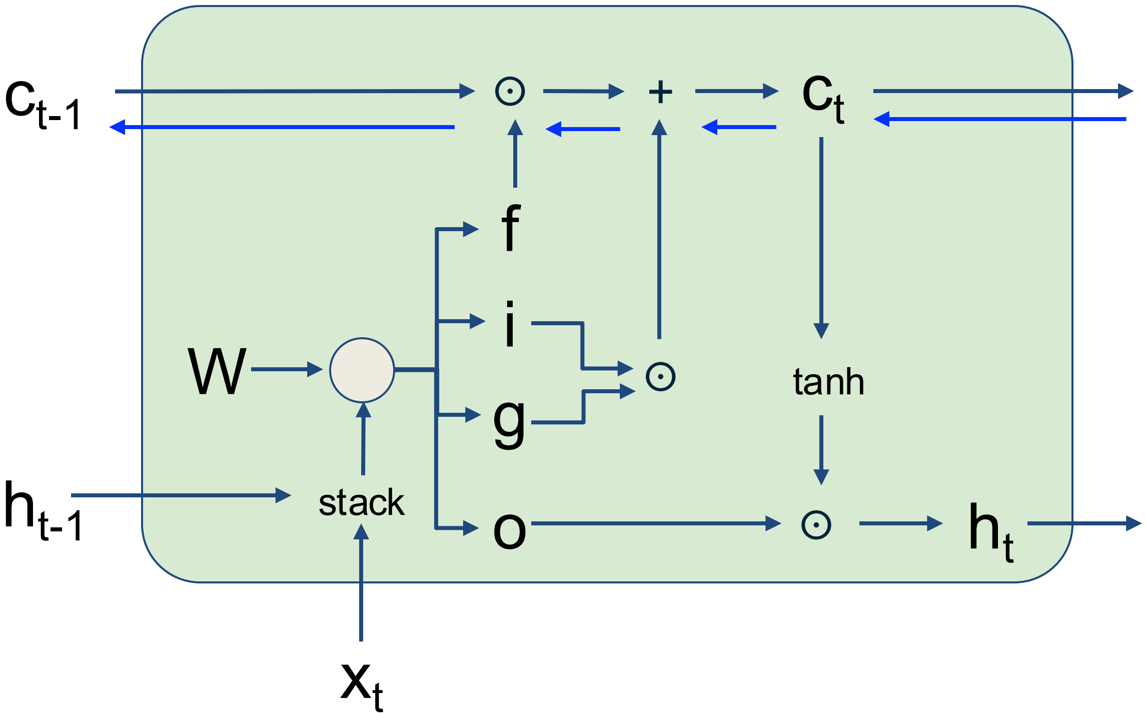
Backpropagation from c_t to c_{t-1} only elementwise multiplication by f (forget gate), no matrix multiply by W

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

$$\frac{\partial c_t}{\partial c_{t-1}} = ?$$

Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]



Backpropagation from c_t to c_{t-1} only elementwise multiplication by f (forget gate), no matrix multiply by W

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

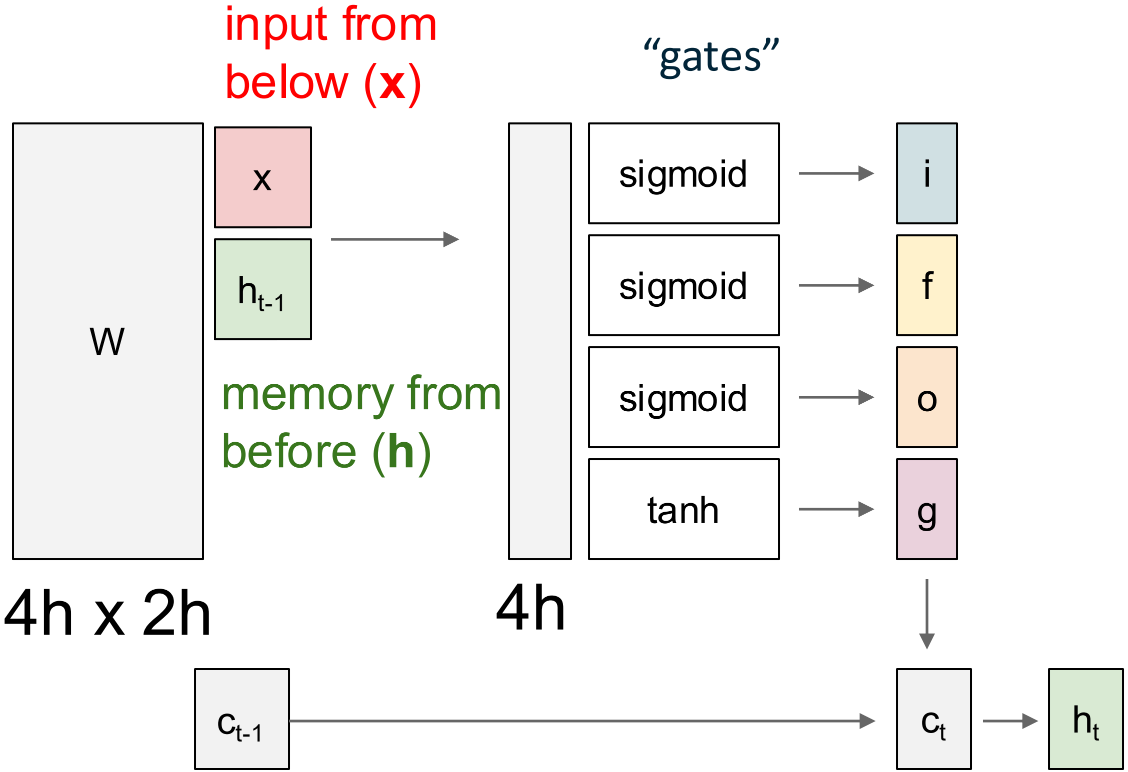
$$\frac{\partial c_t}{\partial c_{t-1}} = f_t + \cdots \quad \text{(forget gate)}$$

Different each step!

When f_t is close to 1, it allows gradient to flow back easily

Long Short Term Memory (LSTM)

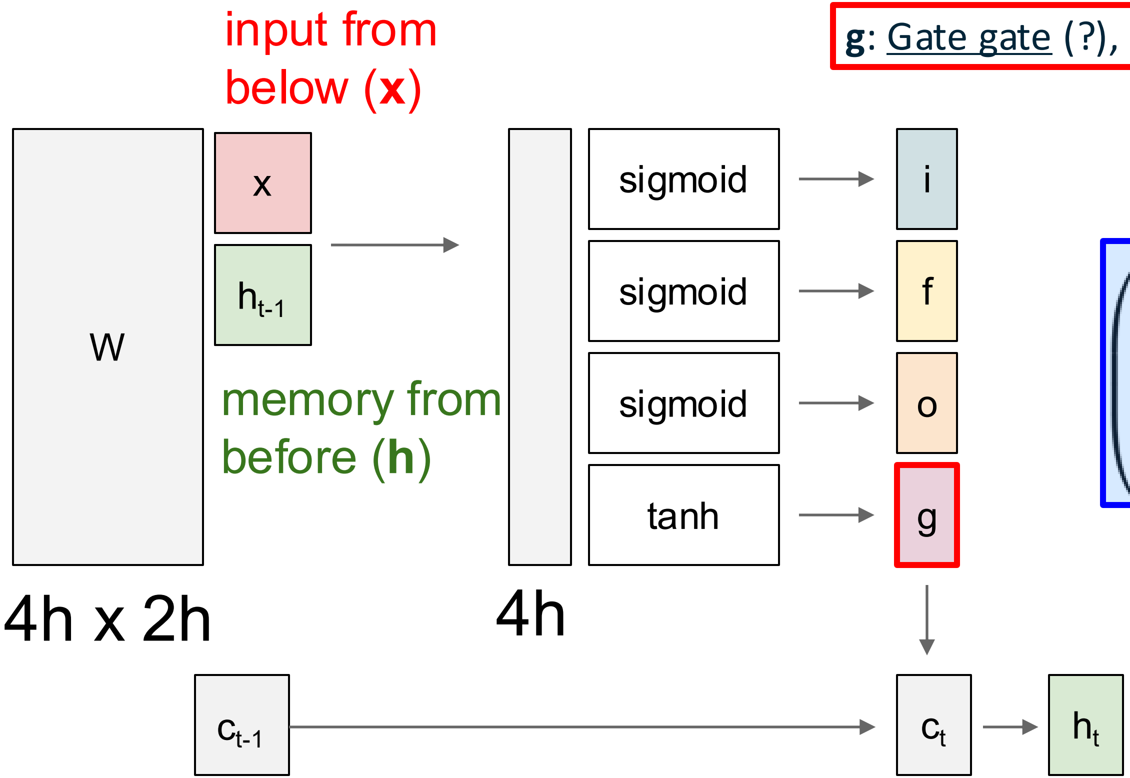
[Hochreiter et al., 1997]



Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

g: Gate gate (?), what to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \boxed{\tanh} \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot \boxed{g}$$

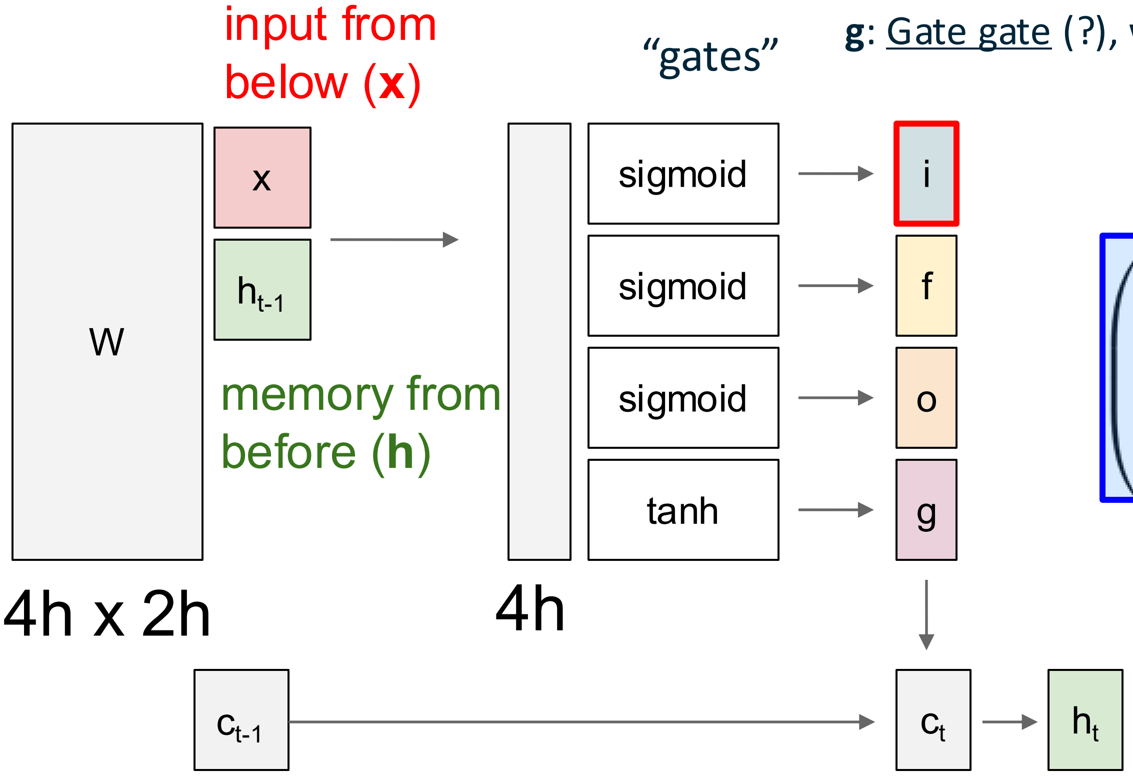
$$h_t = o \odot \tanh(c_t)$$

Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell

g: Gate gate (?), what to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \boxed{\sigma} \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + \boxed{i} \odot g$$

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

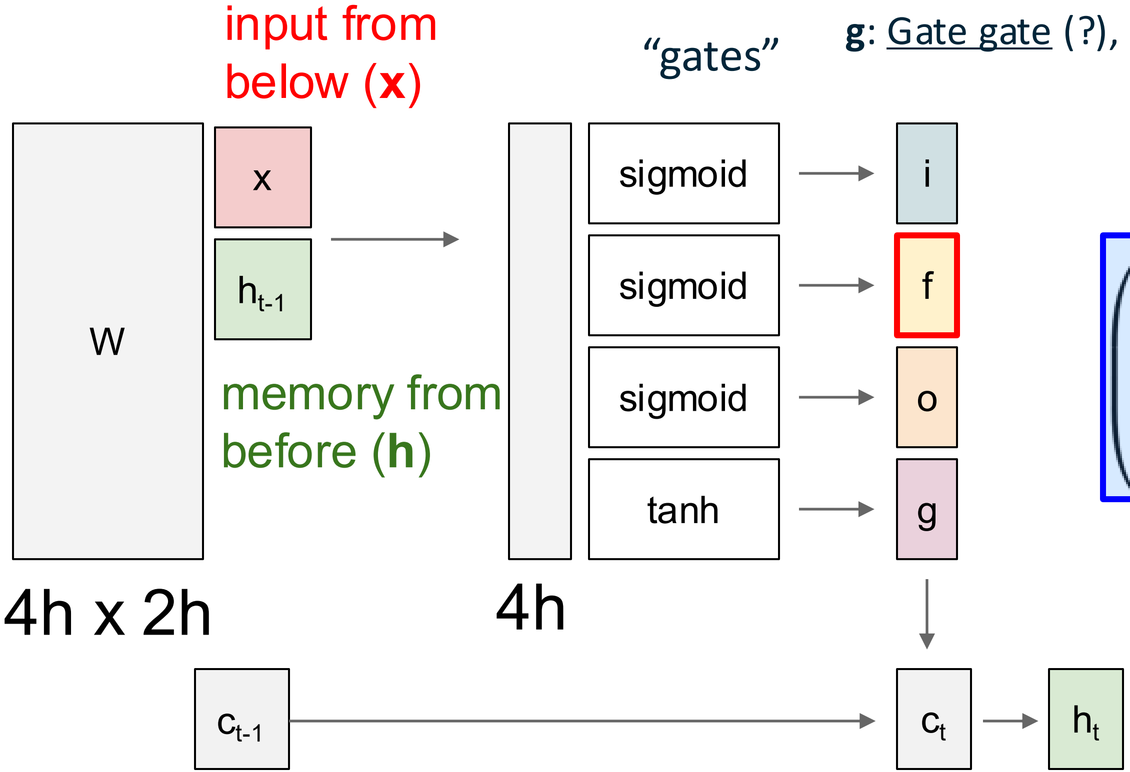
Long Short Term Memory (LSTM)

[Hochreiter et al., 1997]

i: Input gate, whether to write to cell

f: Forget gate, whether to erase cell

g: Gate gate (?), what to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \boxed{\sigma} \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = \boxed{f} \odot c_{t-1} + i \odot g$$

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

Long Short Term Memory (LSTM)

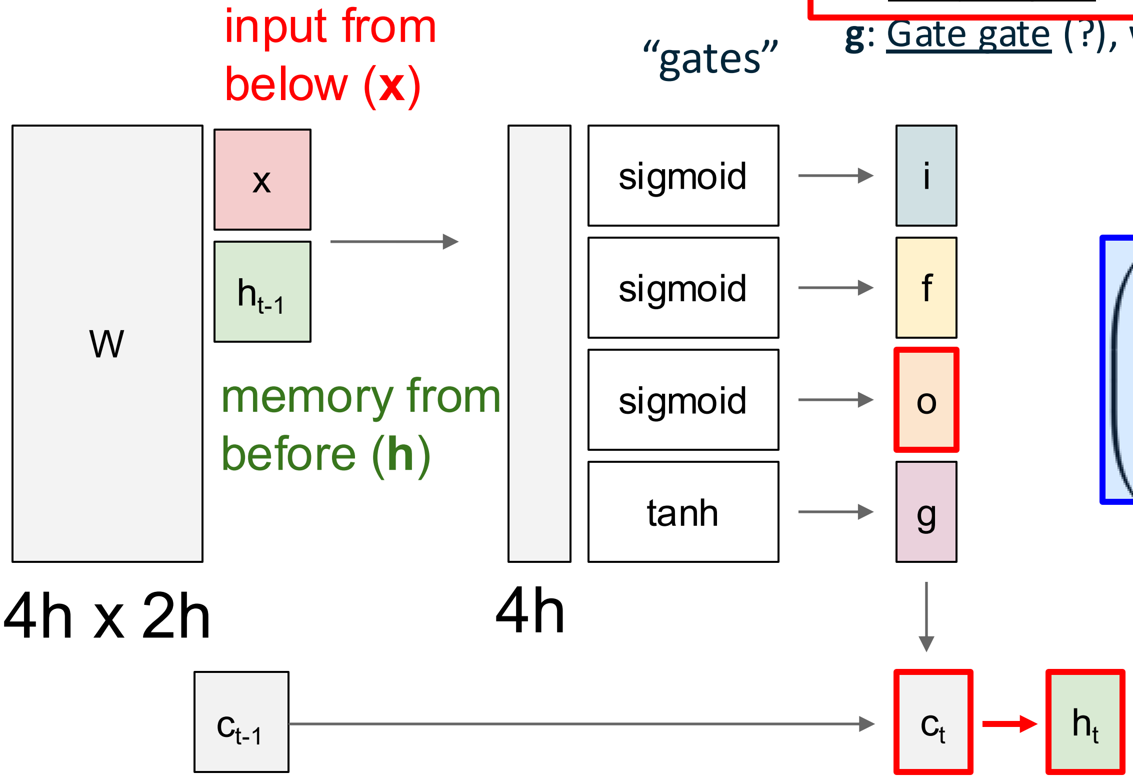
[Hochreiter et al., 1997]

i: Input gate, whether to write to cell

f: Forget gate, whether to erase cell

o: Output gate, how much to reveal cell

g: Gate gate (?), what to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \boxed{\sigma} \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = \boxed{o} \odot \tanh(c_t)$$

Do LSTMs solve the vanishing gradient problem?

The LSTM architecture makes it easier for the RNN to preserve information over many timesteps

- e.g. **if $\boldsymbol{f} = \mathbf{1}$ and $\boldsymbol{i} = \mathbf{0}$**, then the information of that cell is preserved indefinitely. Gradient flow back from cell c easily.
- By contrast, it's harder for vanilla RNN to learn a recurrent weight matrix Wh that preserves info in hidden state

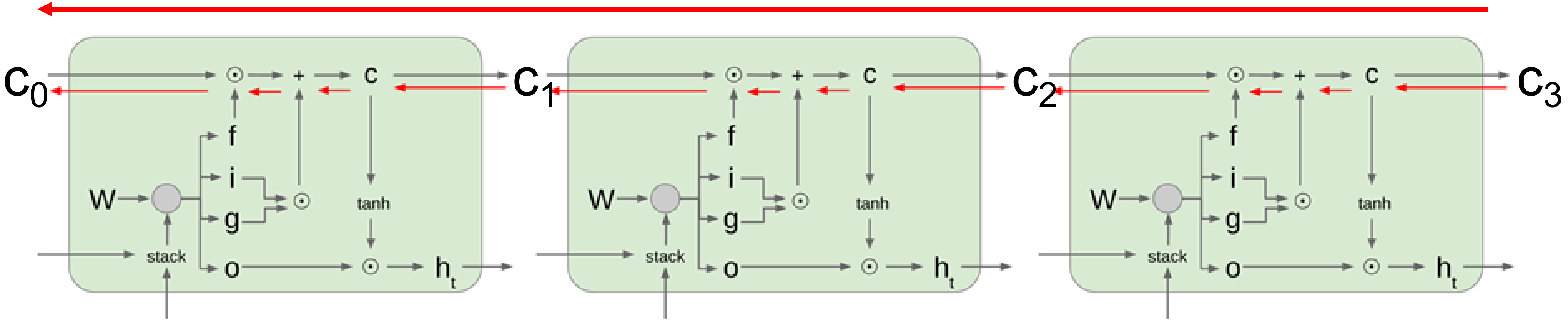
LSTM **doesn't guarantee** that there is no vanishing/exploding gradient, but it does provide an easier way for the model to learn long-distance dependencies.

It is possible to mitigate vanishing / exploding gradient by learning the correct f

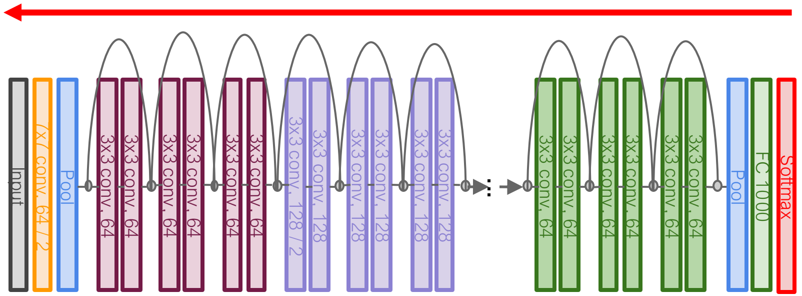
Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

Uninterrupted gradient flow!



Similar to ResNet!



Recommendations

- If you want to use RNN-like models, try LSTM
- Use variants like GRU if you want faster compute and less parameters
- New variants of RNNs are still active research topic. Example: RWKV (“Transformer-level performance but with RNN”)

Problem with Recurrent-style Models (RNN, LSTM, GRU, etc.)

Learning to memorize is still hard, especially for ultra-long sequences!

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

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

Essentially trying to tune W such that the memory cell c can retain **important information** for **arbitrary future prediction problems**.

Example (Q&A):

[… (20-page long transcript)]. Q: What did the CEO say about their competitor company? …

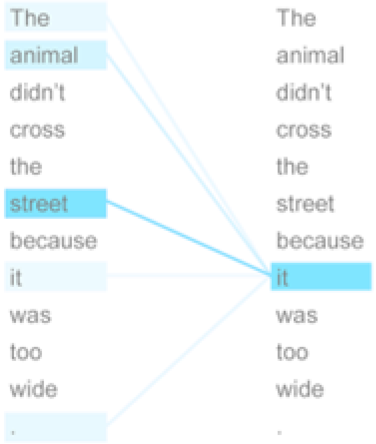
[… (same 20-page transcript)]. Q: How many times did the journalist use the word "interesting"? …

Very difficult learning problem!

Attention Mechanism

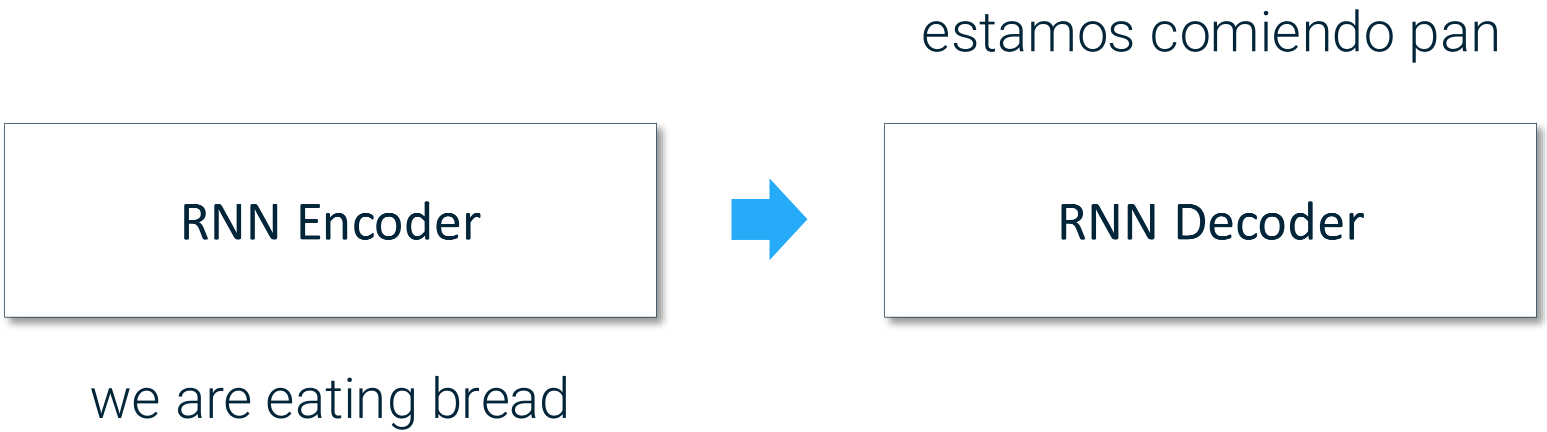
(What memory? Just show me the sequence again)

Attention Mechanism



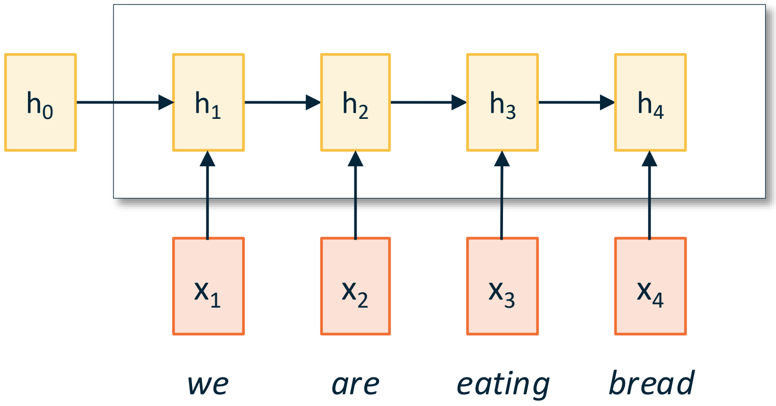
https://blog.research.google/2017/08/transformer-novel-neural-network.html?m=1

Example: Machine Translation



Machine Translation with RNNs

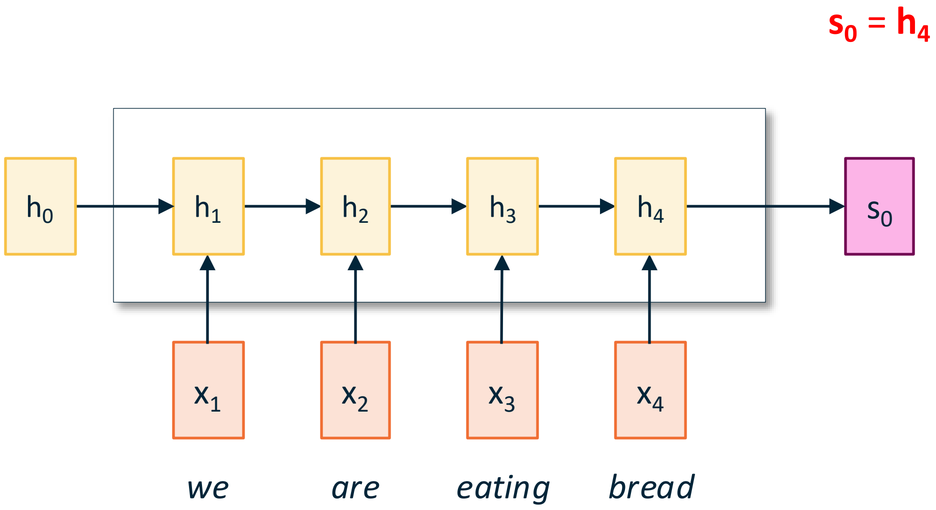
Encoder: $h_t = f_W(x_t, h_{t-1})$



Slide credit: Justin Johnson

Machine Translation with RNNs

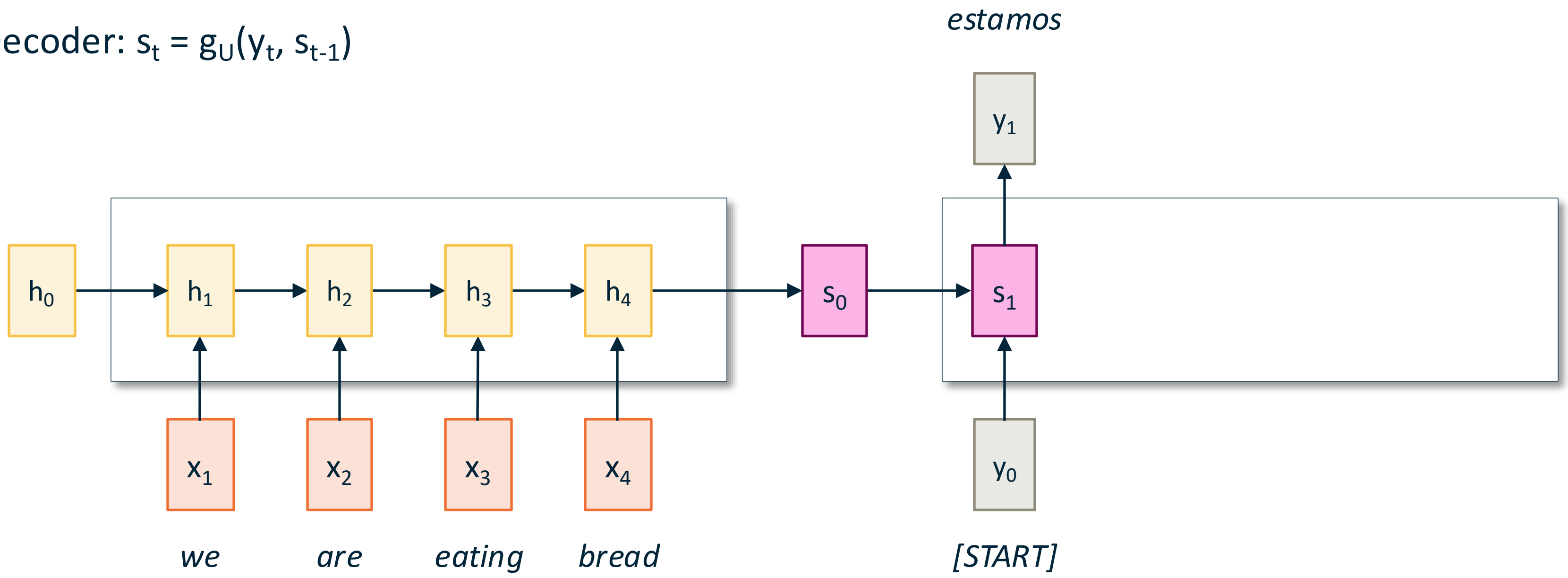
Encoder: $h_t = f_W(x_t, h_{t-1})$



Machine Translation with RNNs

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

Decoder: $s_t = g_U(y_t, s_{t-1})$

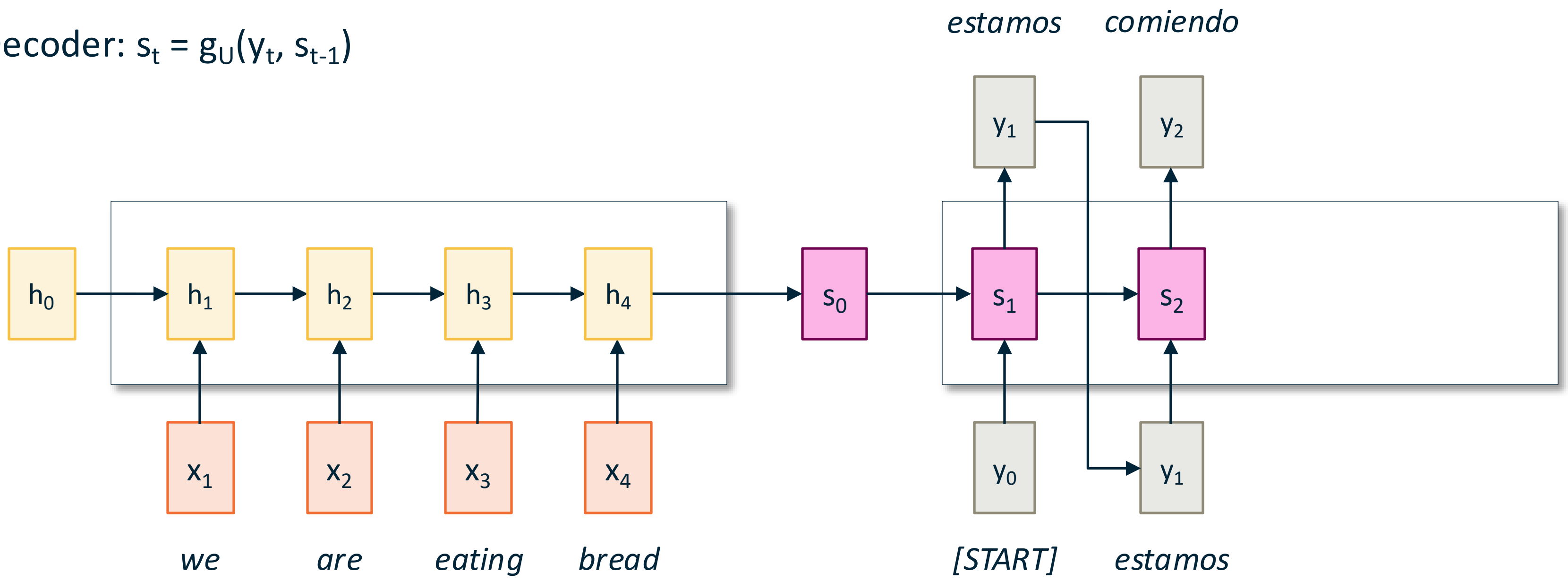


Slide credit: Justin Johnson

Machine Translation with RNNs

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

Decoder: $s_t = g_U(y_t, s_{t-1})$

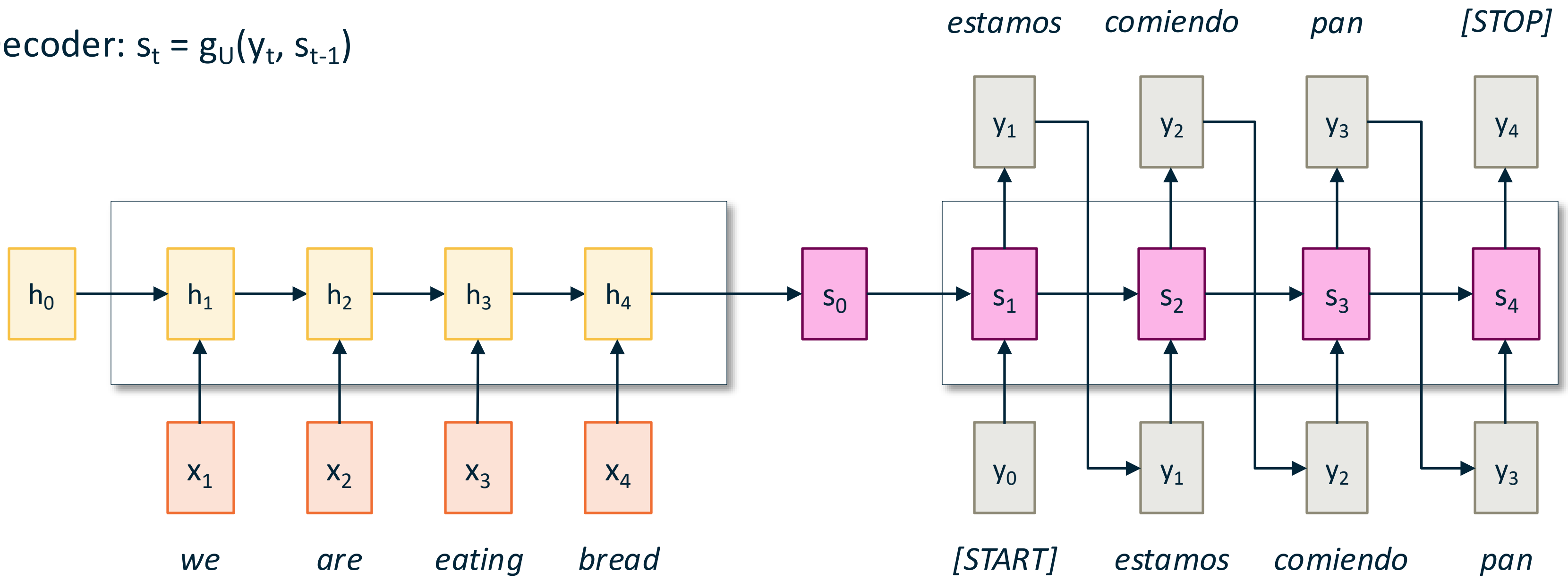


Slide credit: Justin Johnson

Machine Translation with RNNs

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

Decoder: $s_t = g_U(y_t, s_{t-1})$



Slide credit: Justin Johnson

Machine Translation with RNNs

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

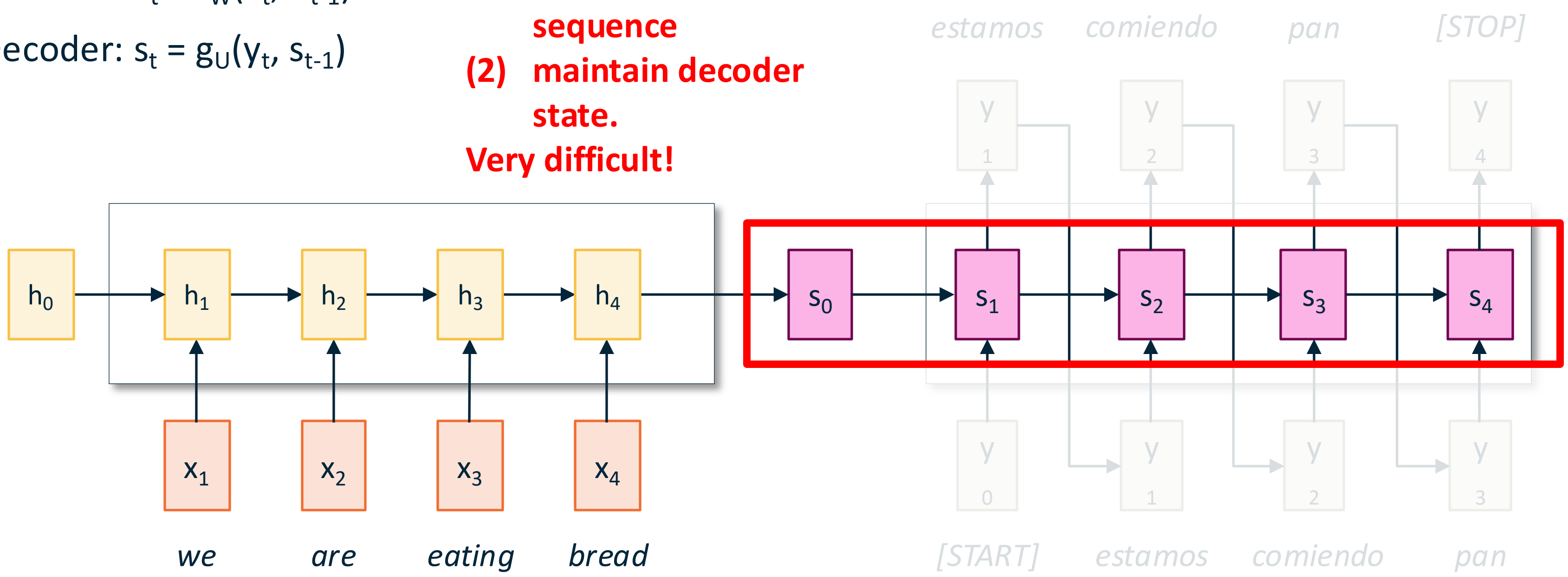
Decoder: $s_t = g_U(y_t, s_{t-1})$

Problem: s_i is used to both

(1) encode input sequence

(2) maintain decoder state.

Very difficult!



Slide credit: Justin Johnson

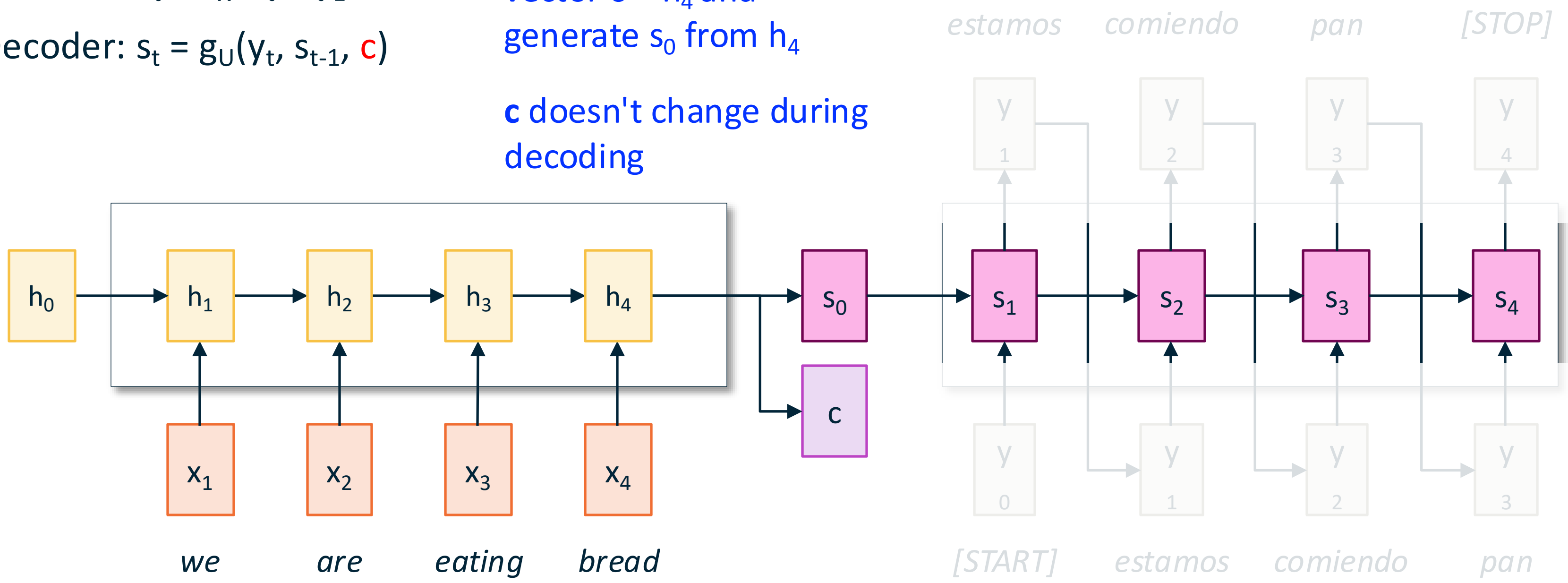
Machine Translation with RNNs

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

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

Solution: add a context vector **c** = h_4 and generate s_0 from h_4

c doesn't change during decoding



Slide credit: Justin Johnson

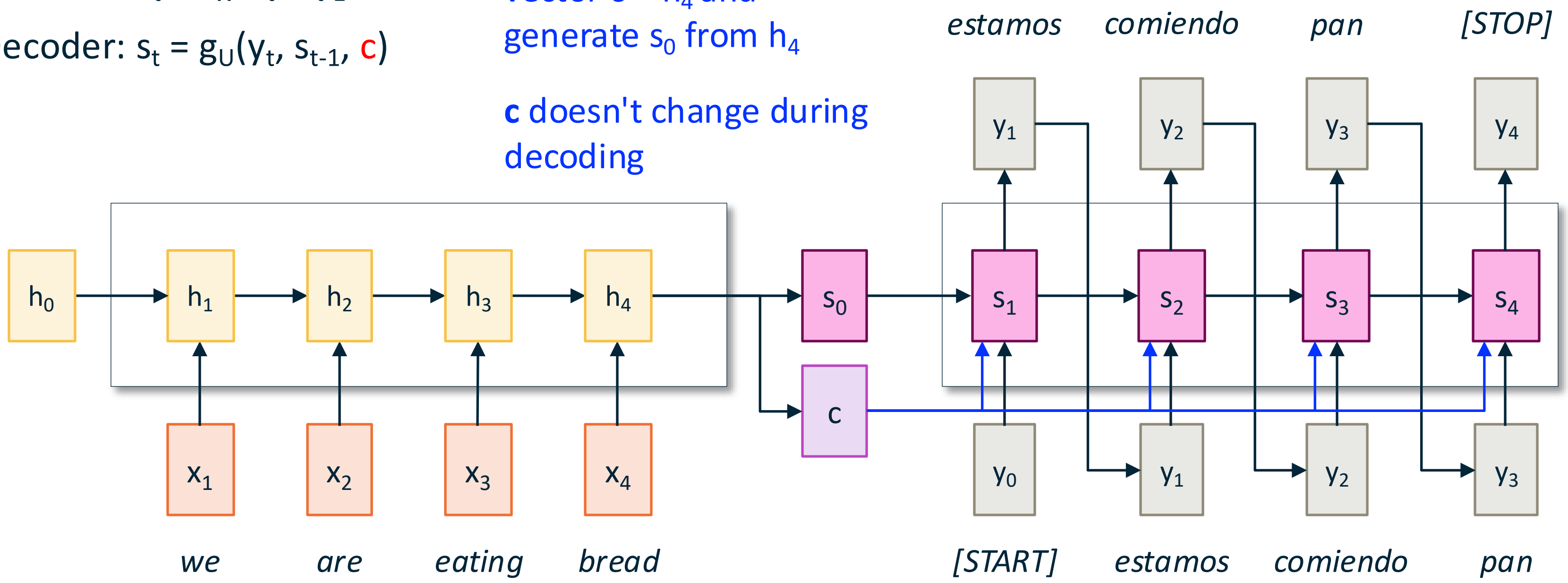
Machine Translation with RNNs

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

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

Solution: add a context vector **c** = h_4 and generate s_0 from h_4

c doesn't change during decoding

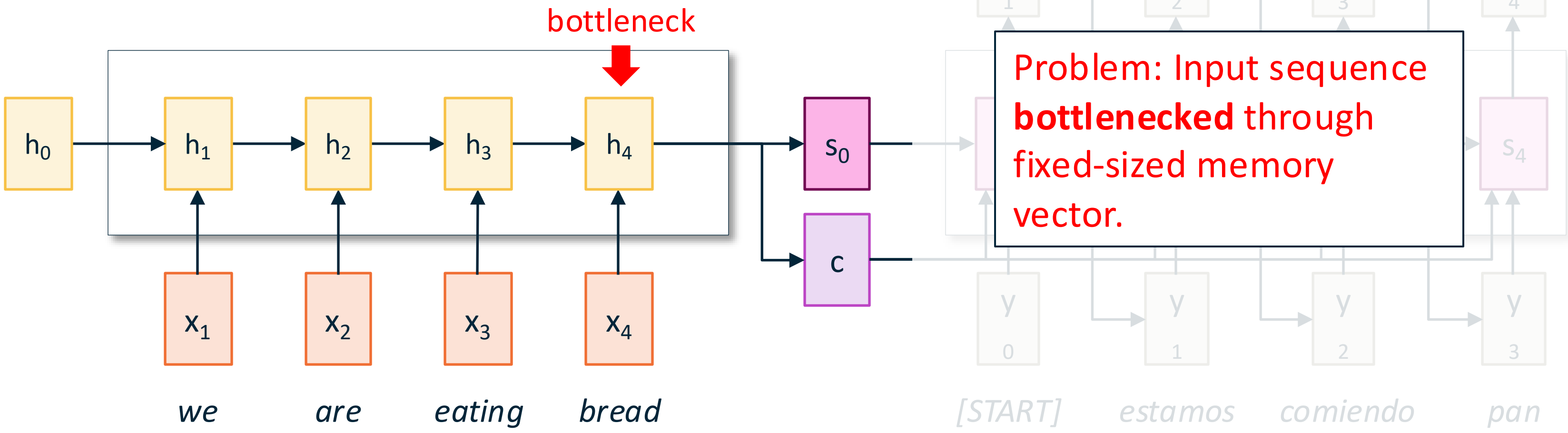


Slide credit: Justin Johnson

Machine Translation with RNNs

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

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



Slide credit: Justin Johnson

Problem with Recurrent-style Models (RNN, LSTM, GRU, etc.)

Learning to memorize is still hard, especially for ultra-long sequences!

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

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

Essentially trying to tune W such that the memory cell *c* can retain **important information** for **arbitrary future prediction problems**.

Example (Q&A):
[... (20-page long transcript)]. Q: What did the CEO say about their competitor company? ...

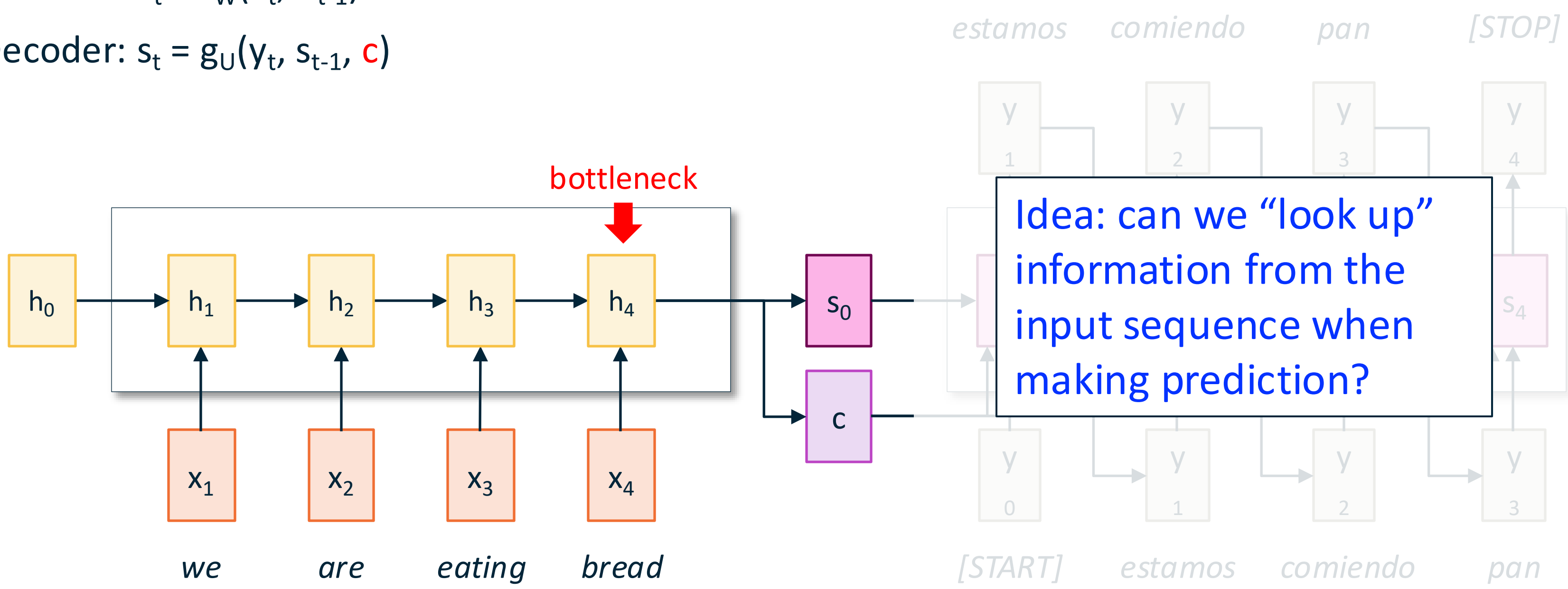
[... (same 20-page transcript)]. Q: How many times did the journalist use the word "interesting"? ...

Very difficult learning problem!

Machine Translation with RNNs

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

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



Idea: can we "look up" information from the input sequence when making prediction?

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**

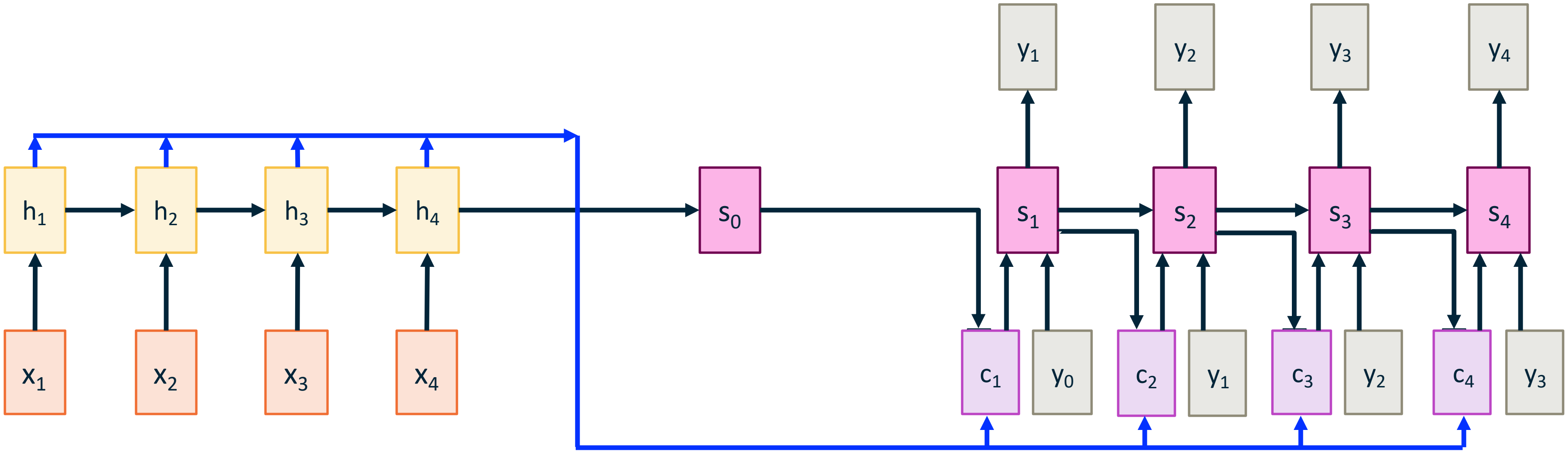
Conceptually, Attention is to **adaptively extract information** from input sequence based on the current decoding step

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**

Conceptually, Attention is to **adaptively extract information** from input sequence based on the current decoding step



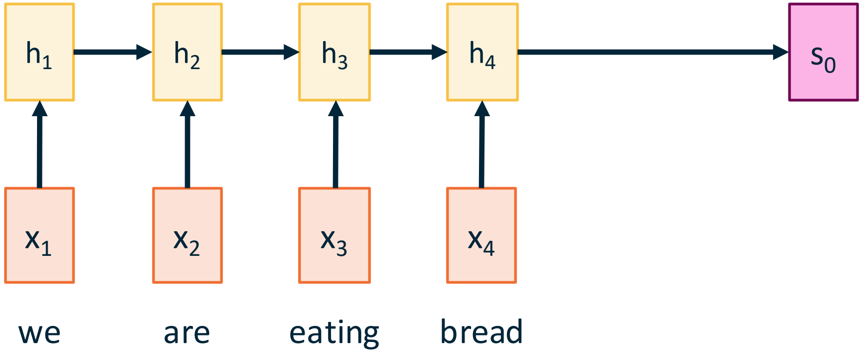
Goal: Adaptive context related to each prediction step

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**

From final hidden state:
Initial decoder state s_0



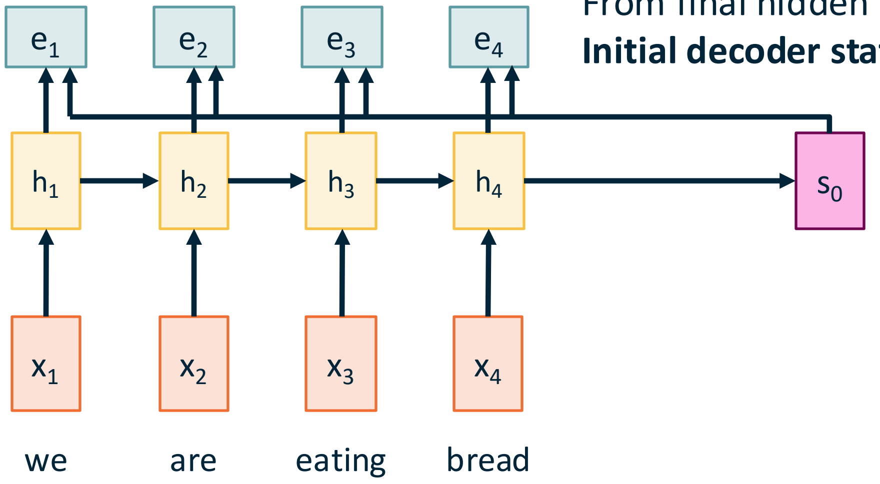
Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**

Compute **affinity scores**

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

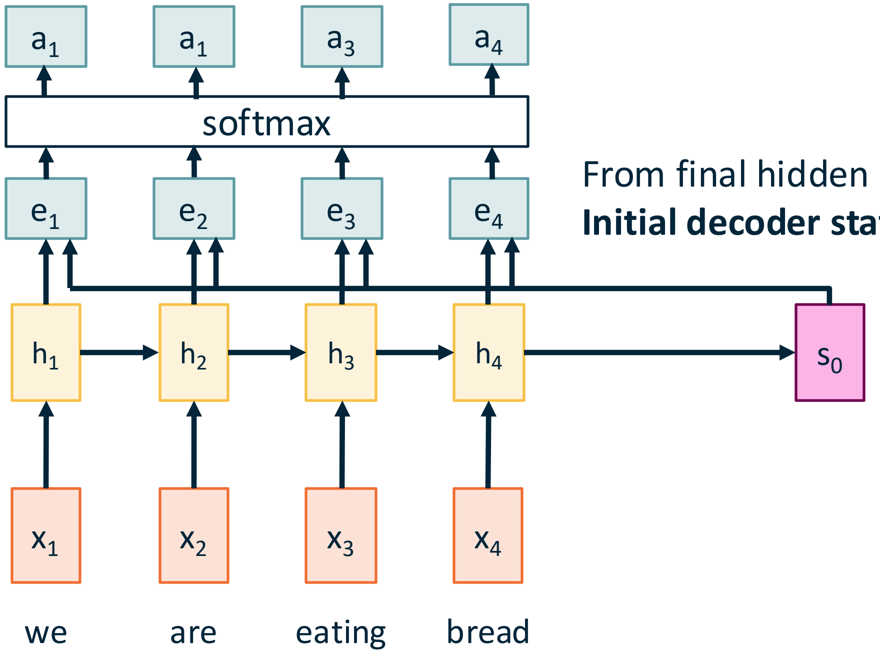


From final hidden state:
Initial decoder state s_0

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**



From final hidden state:
Initial decoder state s_0

Compute **affinity scores**

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

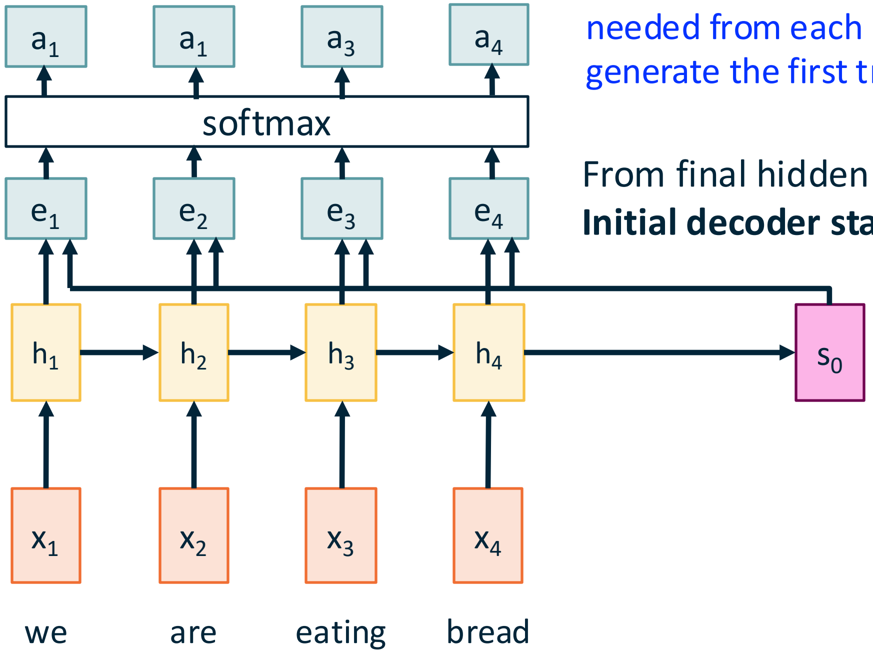
Normalize to get
attention weights

$$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$$

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**



Think of how much information needed from each input word to generate the first translated word

From final hidden state:
Initial decoder state s_0

Compute **affinity scores**

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

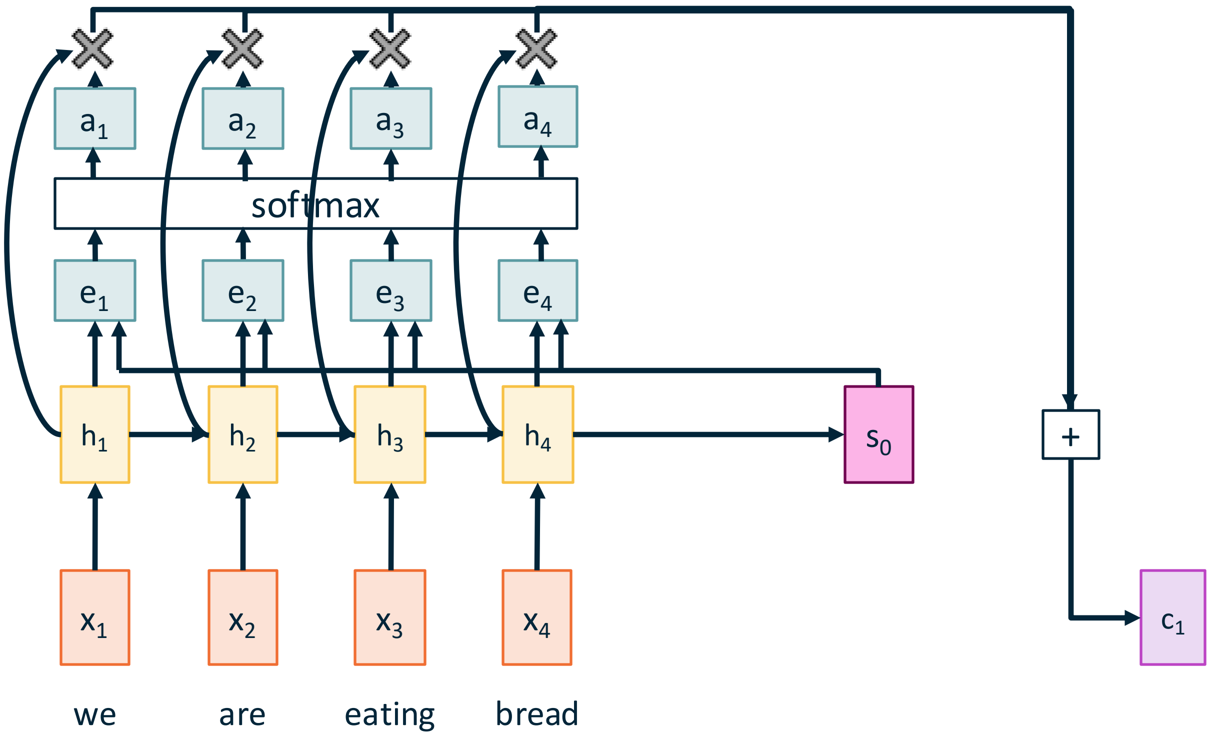
Normalize to get **attention weights**

$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**



Compute **affinity scores**

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

Normalize to get **attention weights**

$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

Set context vector **c** to a linear combination of hidden states

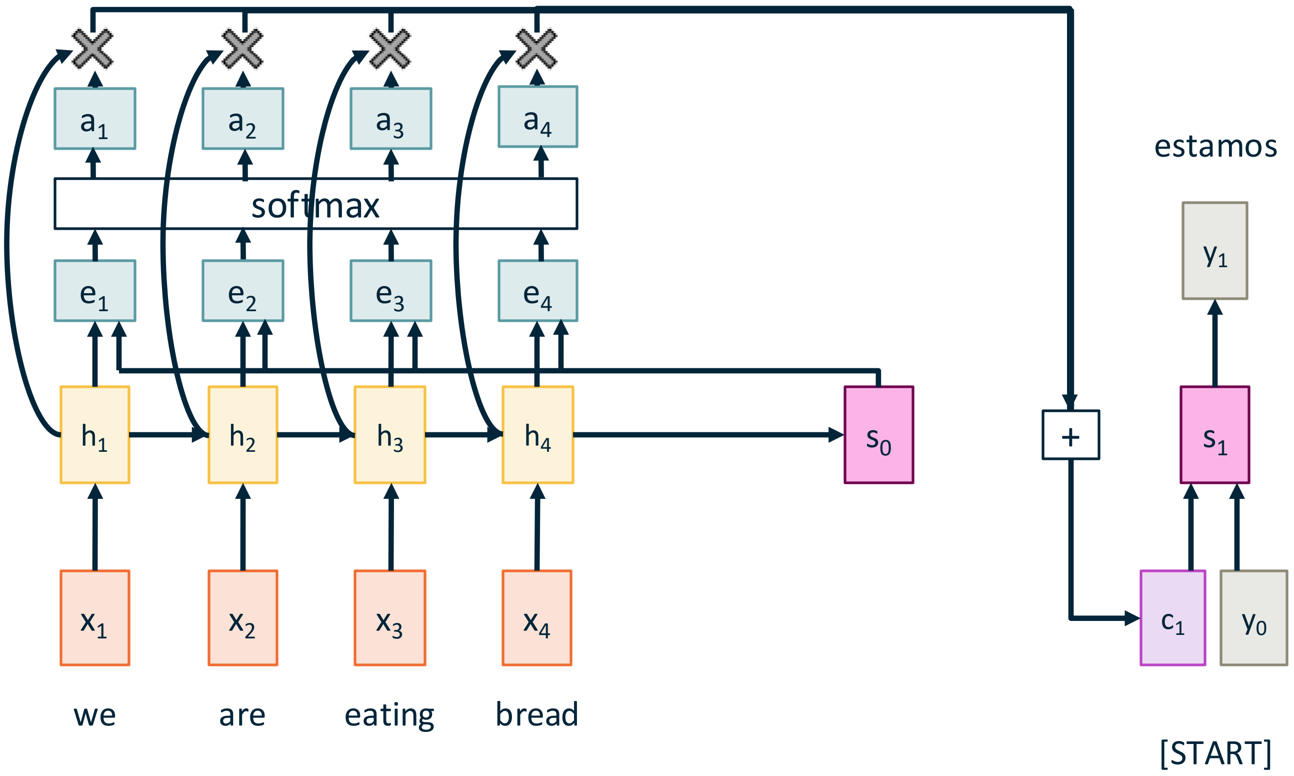
$c_t = \sum_i a_{t,i} h_i$

"Summarize the input sequence related to translating the *t-th* word"

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**



Compute **affinity scores**

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

Normalize to get **attention weights**

$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

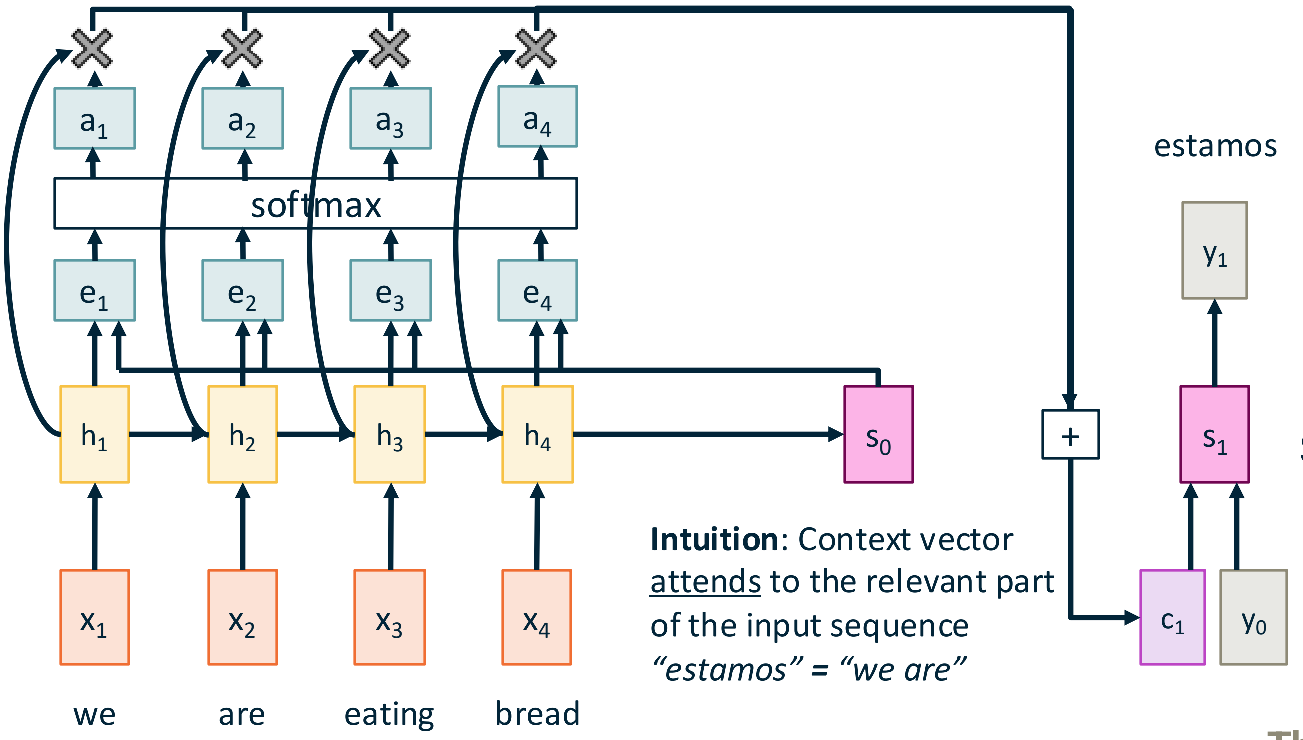
Set context vector **c** to a linear combination of hidden states

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

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**



Intuition: Context vector attends to the relevant part of the input sequence
"estamos" = "we are"

Compute **affinity scores**

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

Normalize to get **attention weights**

$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

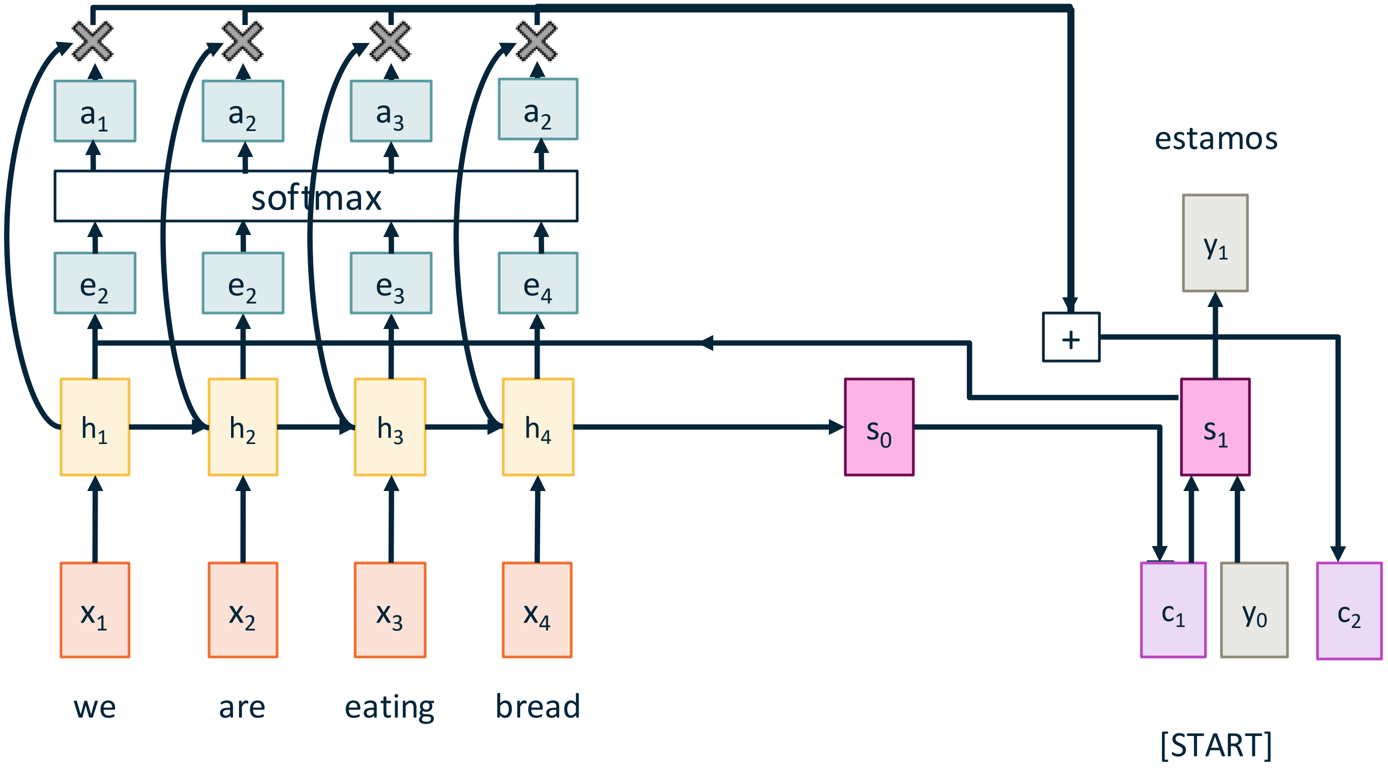
Set context vector **c** to a linear combination of hidden states

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

This is all differentiable! Do not supervise attention weights – backprop through everything

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**

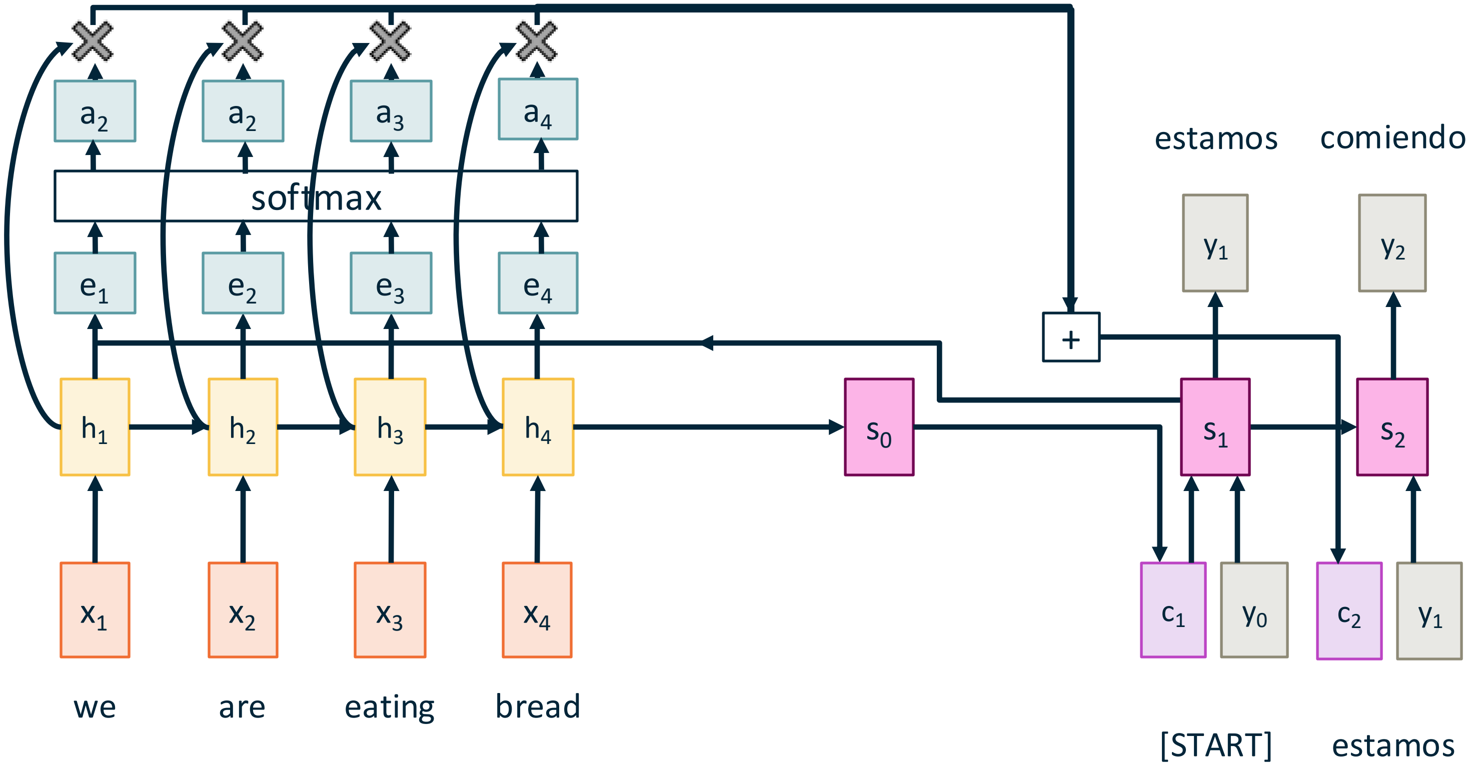


Repeat: Use s_1 to compute attention and get the new context vector c_2

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

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**



Repeat: Use s_1 to compute attention and get the new context vector c_2

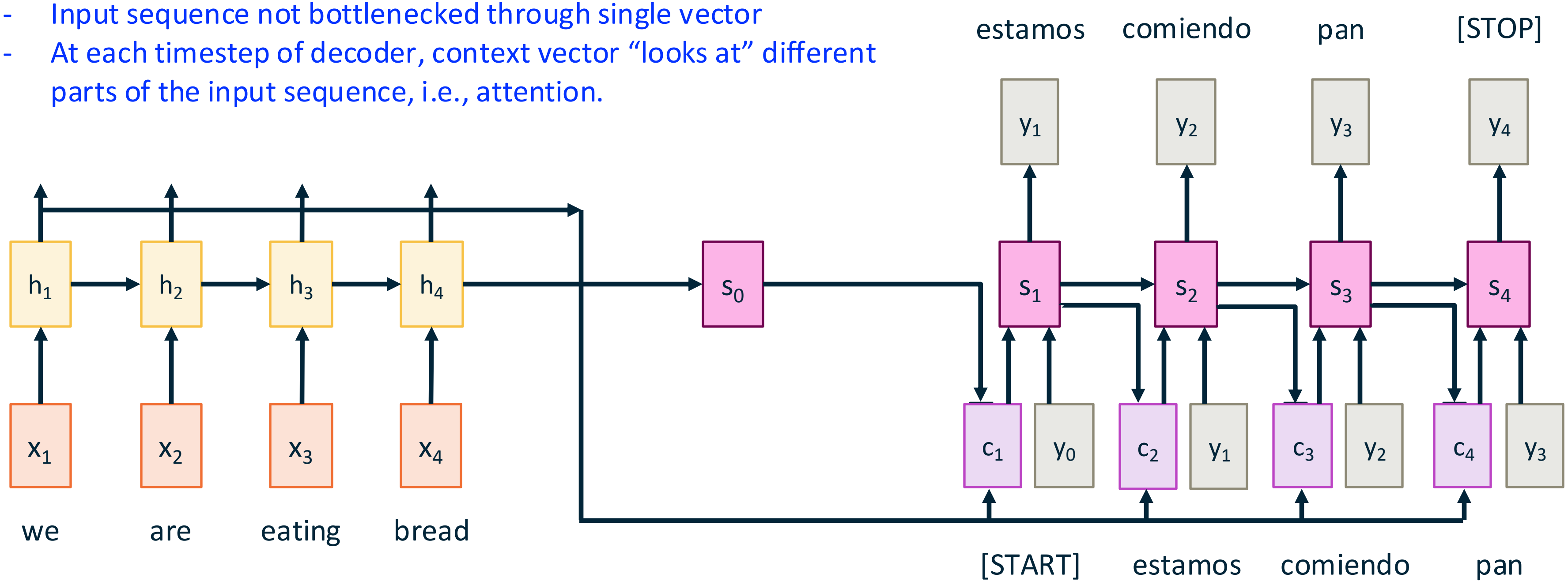
Use c_2 to compute s_2, y_2

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

Slide credit: Justin Johnson

Machine Translation 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, i.e., attention.



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

Slide credit: Justin Johnson

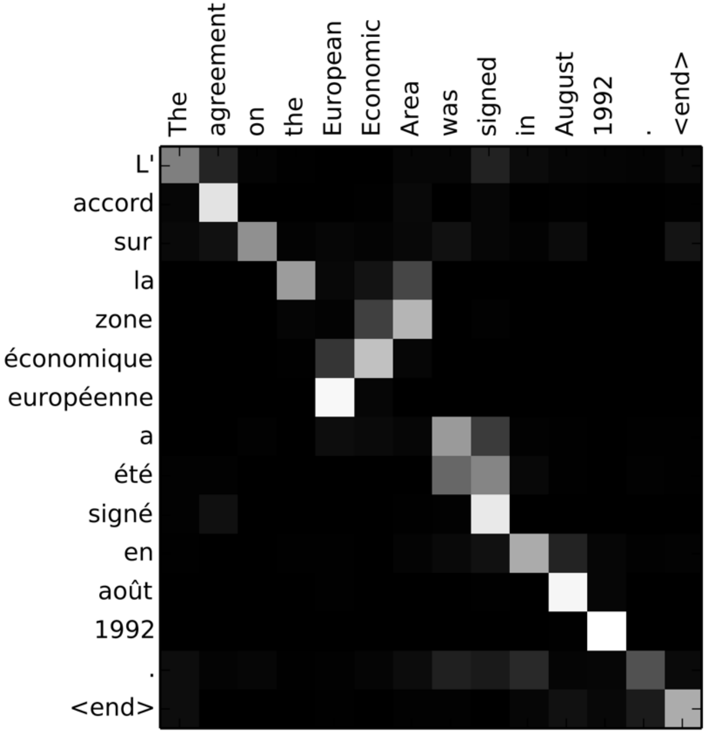
Machine Translation with RNNs **and Attention**

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

Slide credit: Justin Johnson

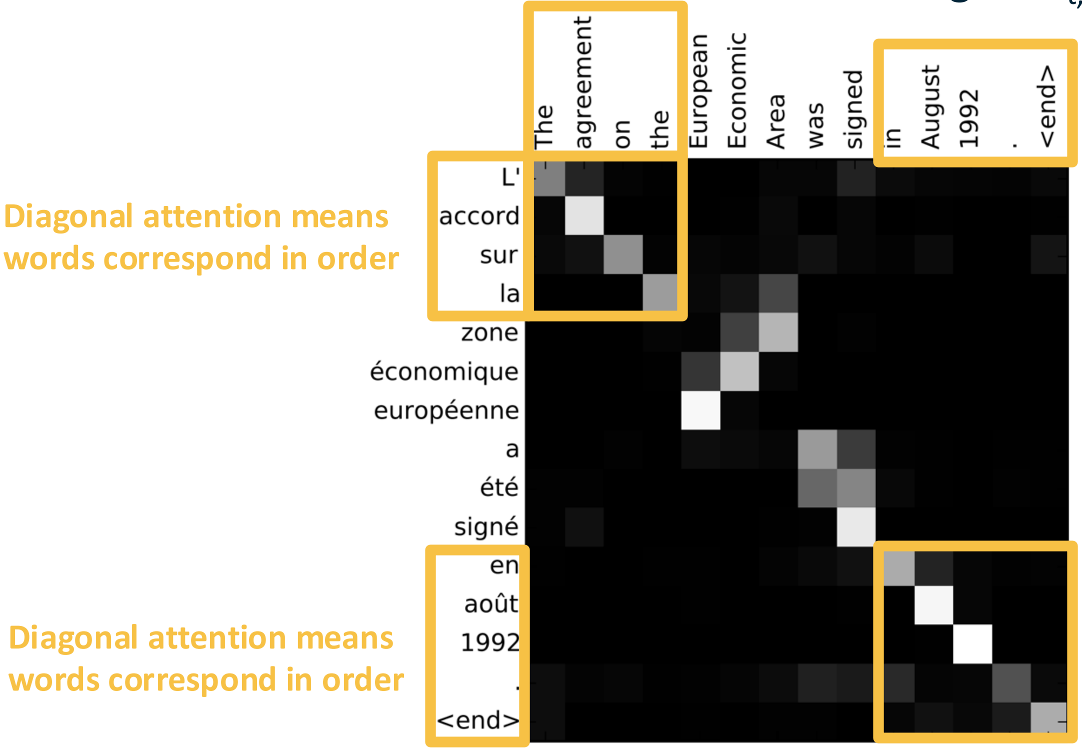
Machine Translation with RNNs **and Attention**

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

Slide credit: Justin Johnson

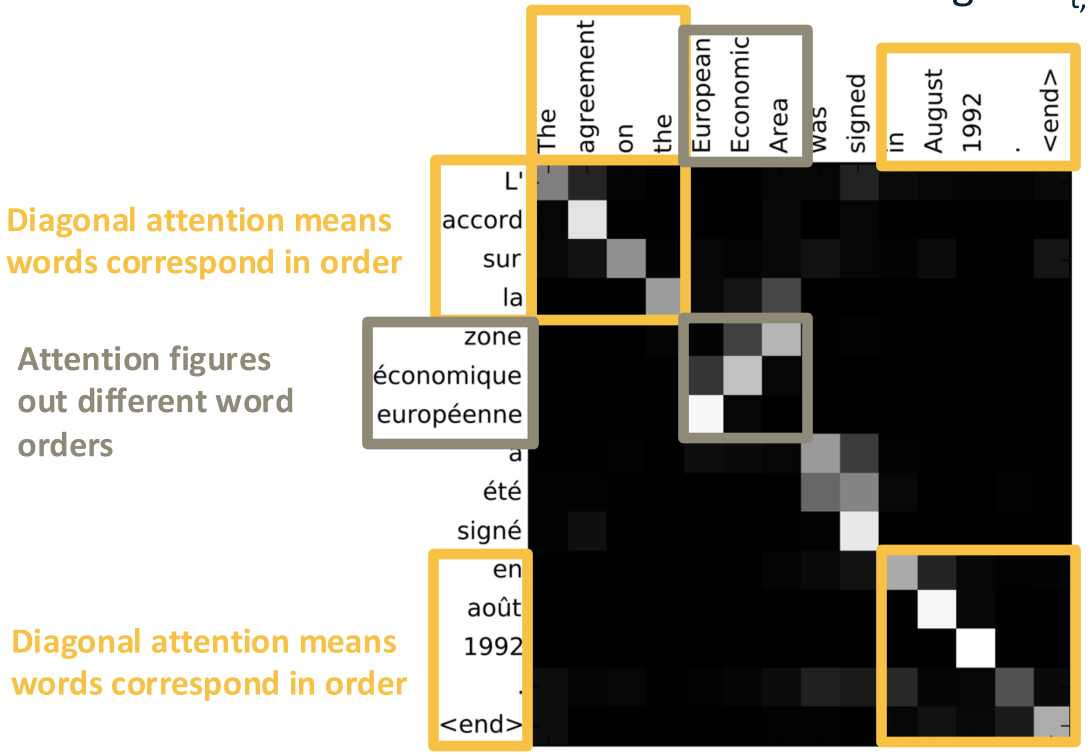
Machine Translation with RNNs **and Attention**

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

Slide credit: Justin Johnson

So … What *is* Attention?

The attention mechanism in allows a model to **focus** on **specific parts of an input sequence** when processing or generating output, **dynamically weighting** the **importance of different elements**.

A set of query vectors:

A set of input vectors:

An attention function:

An attention map:

0.7	0.3	0.0
0.8	0.1	0.1

An aggregation function:

Aside: Differentiable Neural Computer/ Neural Turing Machine

Article | Published: 12 October 2016

Hybrid computing using a neural network with dynamic external memory

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu & Demis Hassabis

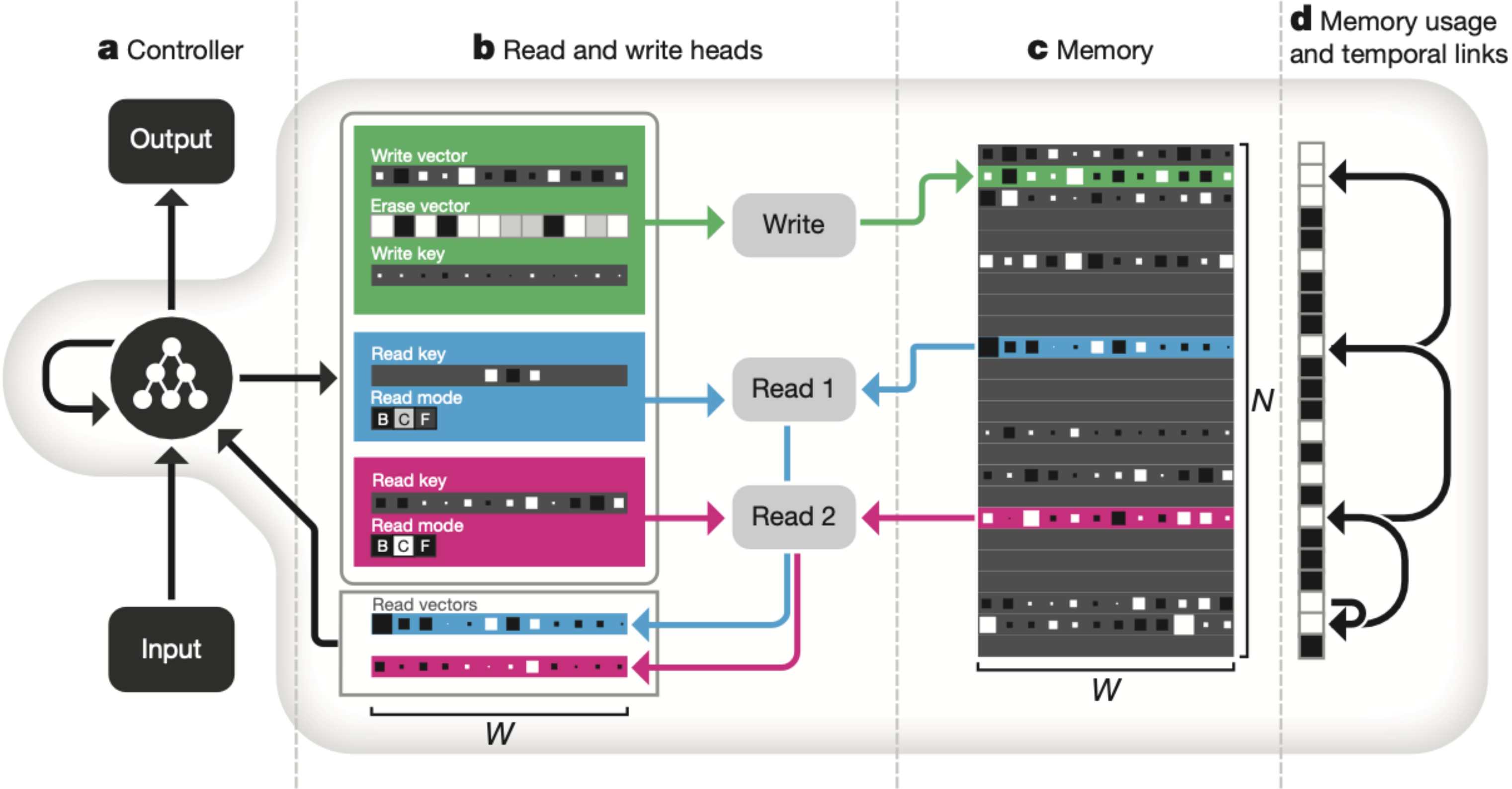
Nature **538**, 471–476 (2016) | Cite this article

Can we make a neural network that behaves like a computer?
Can we make a computer-like mechanism fully-differentiable and parameterized?

Aside: Differentiable Neural Computer/ Neural Turing Machine

Artificial neural networks are remarkably adept at sensory processing, sequence learning and reinforcement learning, but are limited in their ability to represent variables and data structures and to store data over long timescales, owing to the lack of an external memory. Here we introduce a machine learning model called a differentiable neural computer (DNC), which consists of a neural network that can read from and write to an external memory matrix, analogous to the random-access memory in a conventional computer. Like a conventional computer, it can use its memory to represent and manipulate complex data structures, but, like a neural network, it can learn to do so from data. When trained with supervised learning, we demonstrate that a DNC can successfully answer synthetic questions designed to emulate reasoning and inference problems in natural language. We show that it

Aside: Differentiable Neural Computer/ Neural Turing Machine



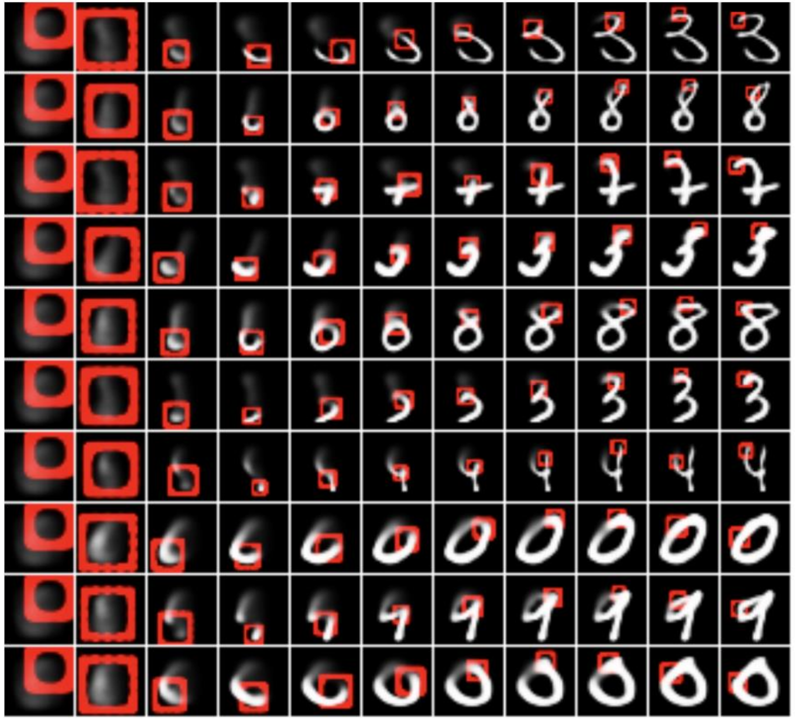
Aside: Attention for Generation

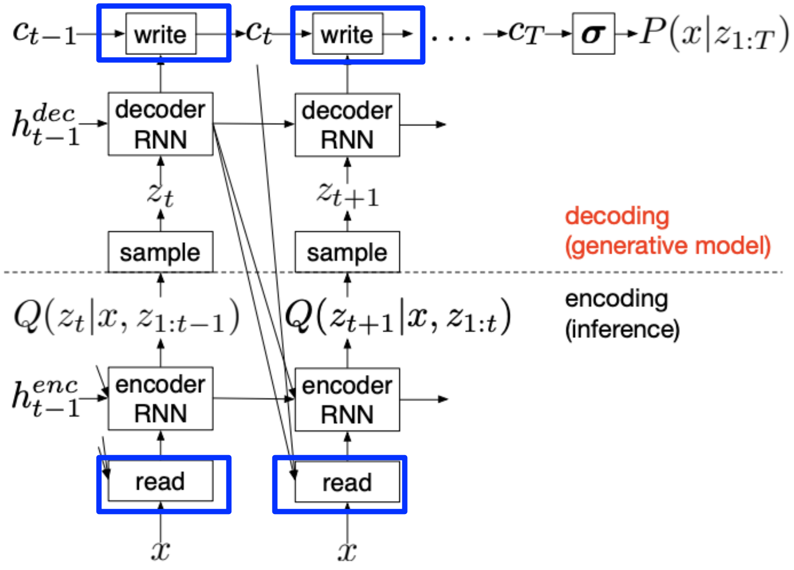
DRAW: A Recurrent Neural Network For Image Generation

Karol Gregor KAROLG@GOOGLE.COM
Ivo Danihelka DANIHELKA@GOOGLE.COM
Alex Graves GRAVESA@GOOGLE.COM
Danilo Jimenez Rezende DANILOR@GOOGLE.COM
Daan Wierstra WIERSTRA@GOOGLE.COM
Google DeepMind

DRAW: A Recurrent Neural Network For Image Generation, 2015

Aside: Attention for Generation





Optimize image-based attention (crop and blur) to learn to decide where to read and write (draw)

DRAW: A Recurrent Neural Network For Image Generation, 2015

So … What *is* Attention?

The attention mechanism in allows a model to **focus** on **specific parts of an input sequence** when processing or generating output, **dynamically weighting** the **importance of different elements**.

A set of query vectors:

A set of input vectors:

An attention function:

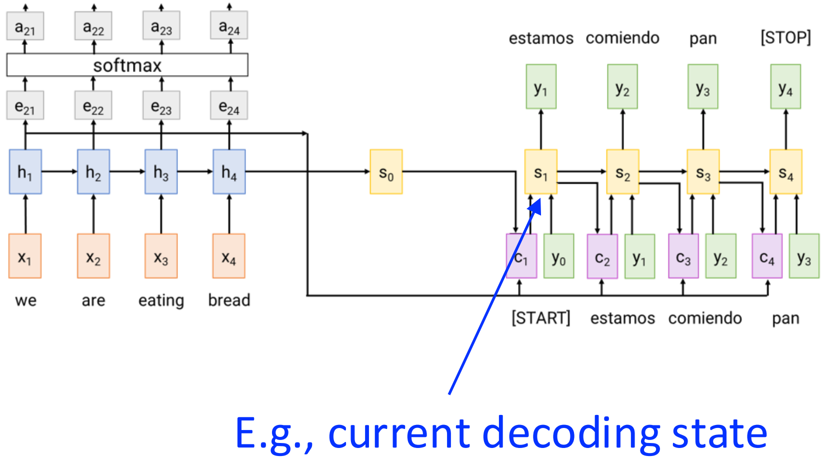
An attention map:

0.7	0.3	0.0
0.8	0.1	0.1

An aggregation function:

Attention Layer

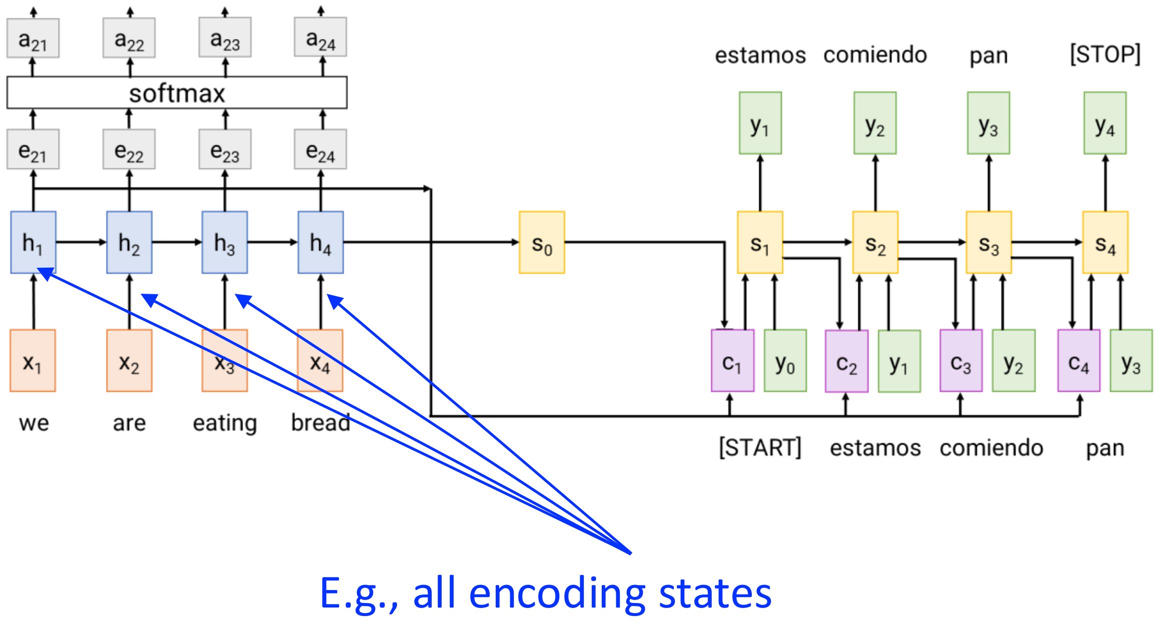
Inputs:
Query vector: **q** (Shape: D_Q)



Slide credit: Justin Johnson

Attention Layer

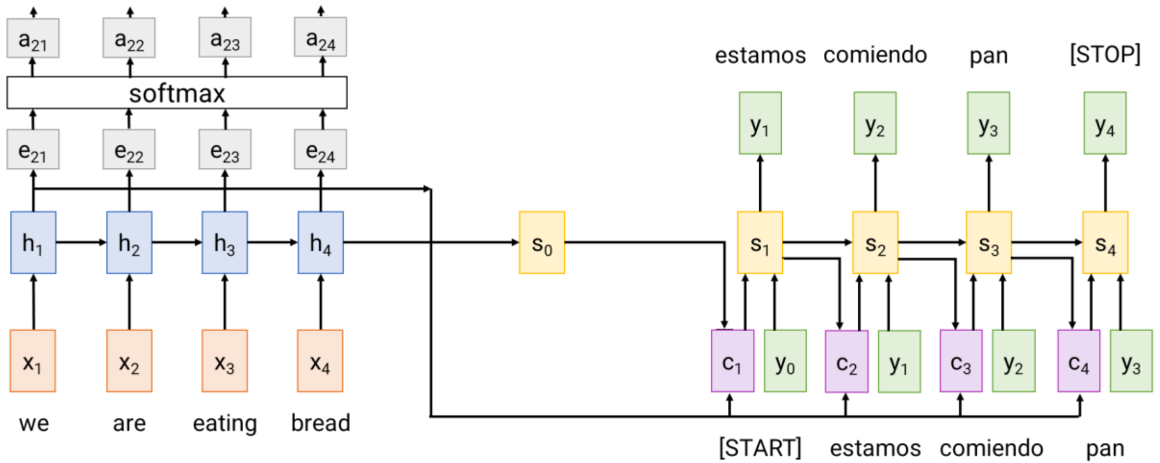
Inputs:
Query vector: q (Shape: D_Q)
Input vectors: X (Shape: N_X x D_Q)



Slide credit: Justin Johnson

Attention Layer

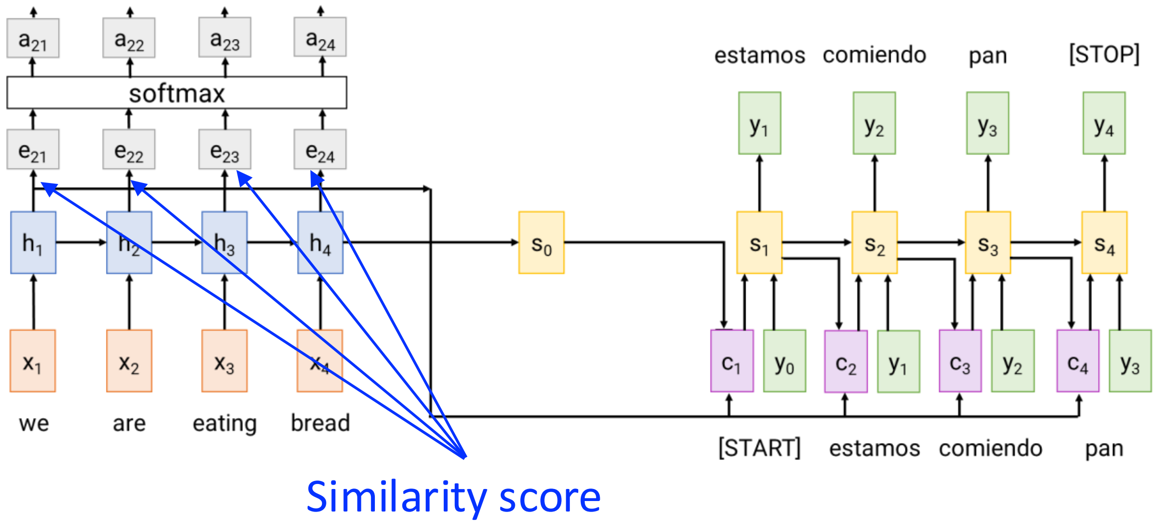
Inputs:
Query vector: **q** (Shape: D_Q)
Input vectors: **X** (Shape: N_X x D_Q)
Similarity function: f_{att}



Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vector: **q** (Shape: D_Q)
Input vectors: **X** (Shape: N_X x D_Q)
Similarity function: f_{att}

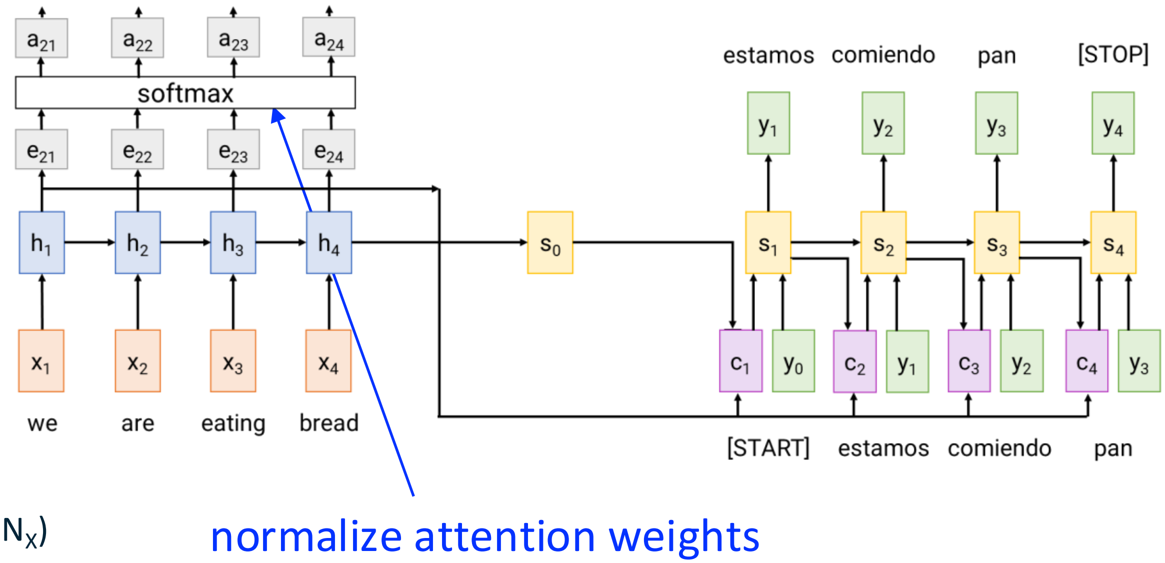


Computation:
Similarities: e (Shape: N_X) $e_i = f_{att}(\mathbf{q}, \mathbf{X}_i)$

Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vector: **q** (Shape: D_Q)
Input vectors: **X** (Shape: N_X x D_Q)
Similarity function: f_{att}

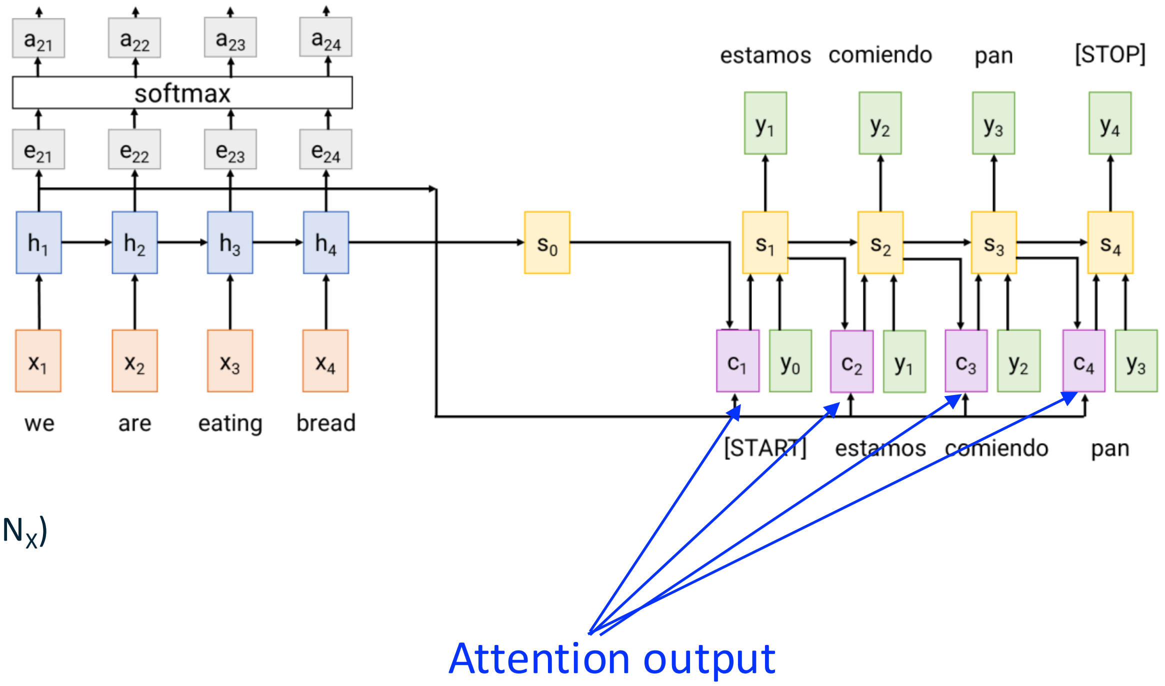


Computation:
Similarities: e (Shape: N_X) $e_i = f_{att}(\mathbf{q}, \mathbf{X}_i)$
Attention weights: a = softmax(e) (Shape: N_X)

Slide credit: Justin Johnson

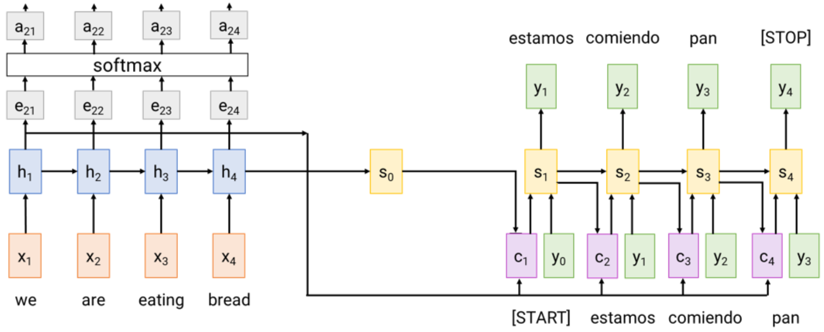
Attention Layer

Inputs:
Query vector: **q** (Shape: D_Q)
Input vectors: **X** (Shape: N_X x D_Q)
Similarity function: f_{att}



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

Attention Layer



<u>**Inputs**</u>:
Query vector: $\mathbf{q}$ (Shape: D_Q)
Input vectors: $\mathbf{X}$ (Shape: N_X x D_Q)
Similarity function: $\boxed{\text{dot product}}$

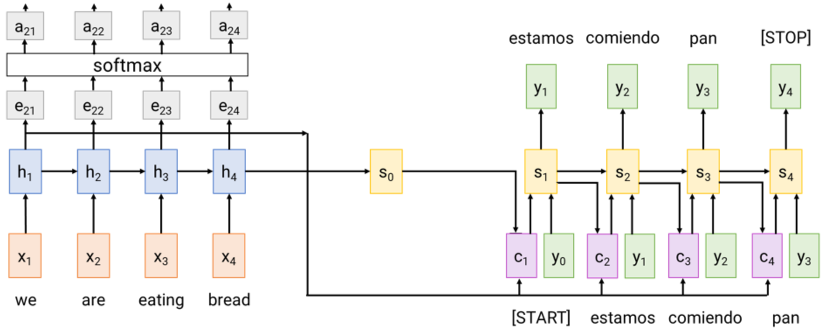
<u>**Computation**</u>:
Similarities: e (Shape: N_X) $\boxed{e_i = \mathbf{q} \cdot \mathbf{X}_i}$
Attention weights: a = softmax(e) (Shape: N_X)
Output vector: $y = \sum_i a_i \mathbf{X}_i$ (Shape: D_X)

Changes:
- Use dot product for similarity

Slide credit: Justin Johnson

Attention Layer

<u>**Inputs**</u>:
Query vector: **q** (Shape: D_Q)
Input vectors: **X** (Shape: N_X x D_Q)
Similarity function: $\boxed{\text{scaled dot product}}$



<u>**Computation**</u>:
Similarities: e (Shape: N_X) $\boxed{e_i = \mathbf{q} \cdot \mathbf{X}_i / \mathrm{sqrt}(D_Q)}$
Attention weights: a = softmax(e) (Shape: N_X)
Output vector: $y = \sum_i a_i \mathbf{X}_i$ (Shape: D_X)

Changes:
- Use **scaled** dot product for similarity to account for the input size D_Q

Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: $\boxed{Q \text{ (Shape: } N_Q \times D_Q)}$
Input vectors: X (Shape: $N_X \times D_Q$)

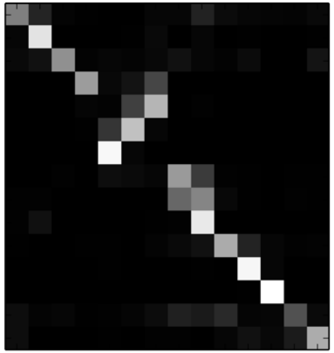
Computation:
Similarities: $E = QX^T / \mathrm{sqrt}(D_Q)$ (Shape: $N_Q \times N_X$)
Attention matrix: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_Q \times N_X$)
Output vectors: $Y = AX$ (Shape: $N_Q \times D_X$) $Y_i = \sum_j A_{i,j} X_j$

Changes:
- Use **matrix multiplication** for similarity (many-to-many dot product)
- Multiple **query** vectors

Attention Layer

<u>**Inputs**</u>:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_Q)

Attention matrix (A)
Each row sums up to 1



Computation:
Similarities: E = QX^T / sqrt(D_Q) (Shape: N_Q x N_X)
Attention matrix: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = AX (Shape: N_Q x D_X) $Y_i = \sum_j A_{i,j} X_j$

Changes:
- Use **matrix multiplication** for similarity (many-to-many dot product)
- Multiple **query** vectors

Attention Layer

Inputs:
Query vectors: **Q** (Shape: N_Q x D_Q)
Input vectors: **X** (Shape: N_X x D_Q)

Computation:
Similarities: E = **QX**^{T} / sqrt(D_Q) (Shape: N_Q x N_X)
Attention matrix: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = A**X** (Shape: N_Q x D_X) $Y_i = \sum_j A_{i,j} X_j$

Problem: use the same set of input vectors to compute both affinity and output.

Ideally, similarity/affinity should be based on some compact "signature vectors" that are efficient to compute.

Think of a key-value storage --- you typically use metadata (key) to look up an item (value), instead of using the item itself!

Changes:
- Use **matrix multiplication** for similarity (many-to-many dot product)
- Multiple **query** vectors

Slide credit: Justin Johnson

Attention Layer

Problem: use the same set of input vectors to compute both affinity and output

Solution: project input to two sets of vectors: Keys (K) and Values (V).

Q,K,V attention: Compute attention matrix using Queries (Q) and Keys (K). Then compute output using attention and Values (V).

Inputs:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)

Computation:
Similarities: $E = QX^T / \mathrm{sqrt}(D_Q)$ (Shape: N_Q x N_X)
Attention matrix: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: N_Q x N_X)
Output vectors: $Y = AX$ (Shape: N_Q x D_X) $Y_i = \sum_j A_{i,j} X_j$

Changes:
- Use **matrix multiplication** for similarity (many-to-many dot product)
- Multiple **query** vectors
- Separate **key** and **value**

Slide credit: Justin Johnson

Attention Layer

Problem: use the same set of input vectors to compute both affinity and output

Solution: project input to two sets of vectors: Keys (K) and Values (V).

Q,K,V attention: Compute attention matrix using Queries (Q) and Keys (K). Then compute output using attention and Values (V).

Inputs:
Query vectors: **Q** (Shape: N_Q x D_Q)
Input vectors: **X** (Shape: N_X x D_X)
Key matrix: $\mathbf{W_K}$ (Shape: D_X x D_Q)
Value matrix: $\mathbf{W_V}$ (Shape: D_X x D_V)

Computation:
Key vectors: **K** = $\mathbf{XW_K}$ (Shape: N_X x D_Q)
Value vectors: **V** = $\mathbf{XW_V}$ (Shape: N_X x D_V)
Similarities: E = $\mathbf{QK^T}$ (Shape: N_Q x N_X) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \mathrm{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = A**V** (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$

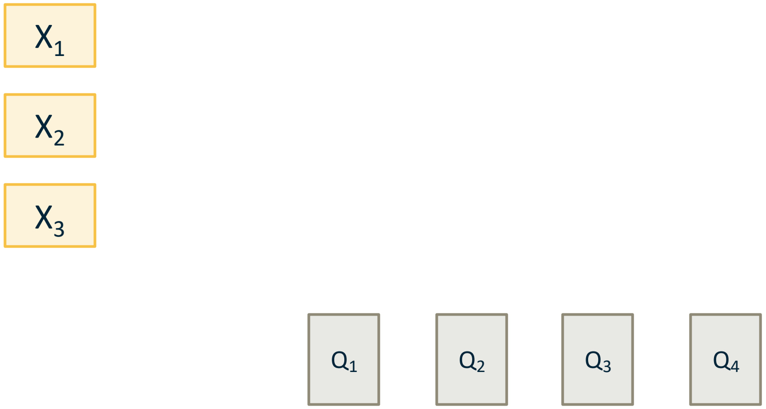
Changes:
- Use **matrix multiplication** for similarity (many-to-many dot product)
- Multiple **query** vectors
- Separate **key** and **value**

Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: **Q** (Shape: N_Q x D_Q)
Input vectors: **X** (Shape: N_X x D_X)
Key matrix: $\mathbf{W_K}$ (Shape: D_X x D_Q)
Value matrix: $\mathbf{W_V}$ (Shape: D_X x D_V)

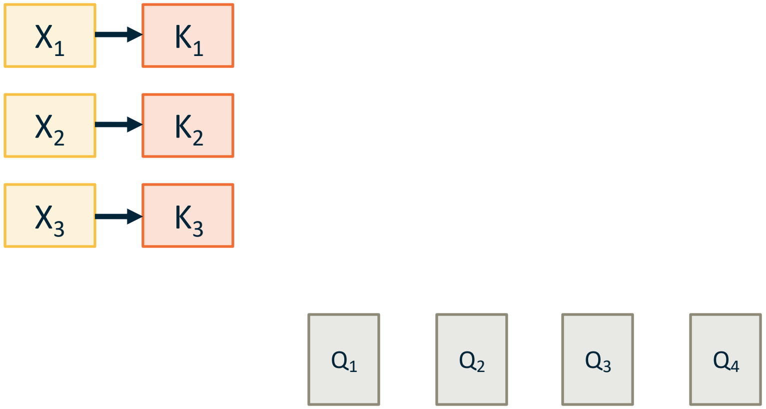
Computation:
Key vectors: **K** = $\mathbf{XW_K}$ (Shape: N_X x D_Q)
Value vectors: **V** = $\mathbf{XW_V}$ (Shape: N_X x D_V)
Similarities: E = $\mathbf{QK^T}$ (Shape: N_Q x N_X) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \mathrm{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = A**V** (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$



Attention Layer

Inputs:
Query vectors: Q (Shape: $N_Q \times D_Q$)
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)

Computation:
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_Q \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

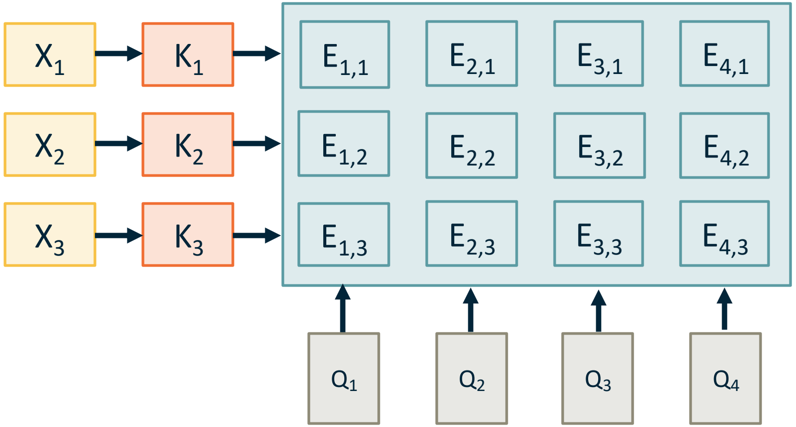


Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)

Computation:
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_Q x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: N_Q x N_X)
Output vectors: $Y = AV$ (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} V_j$

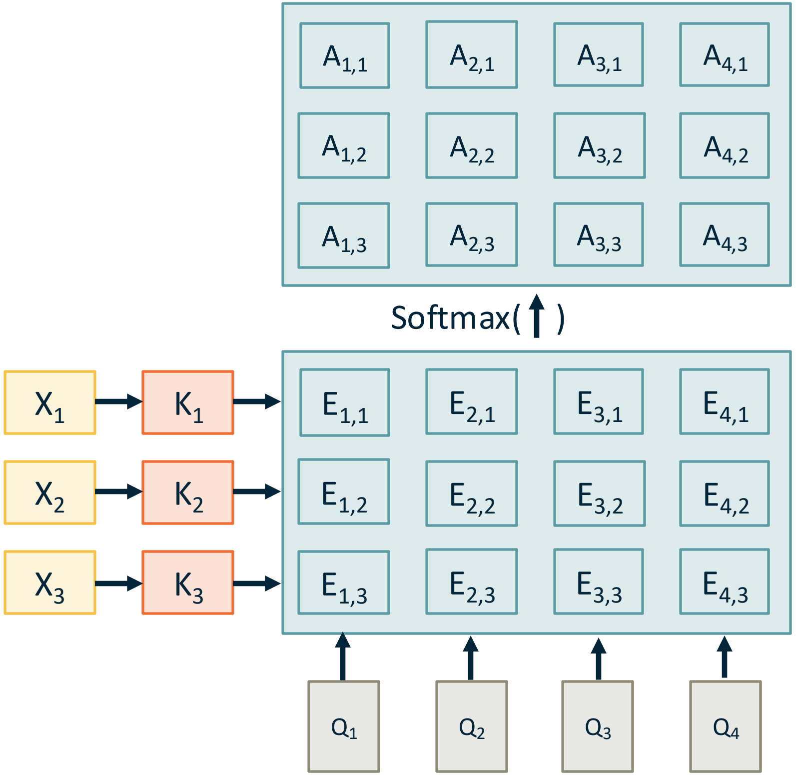


Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)

Computation:
Key vectors: K = XW_K (Shape: N_X x D_Q)
Value vectors: V = XW_V (Shape: N_X x D_V)
Similarities: E = QK^T (Shape: N_Q x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = AV (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} V_j$

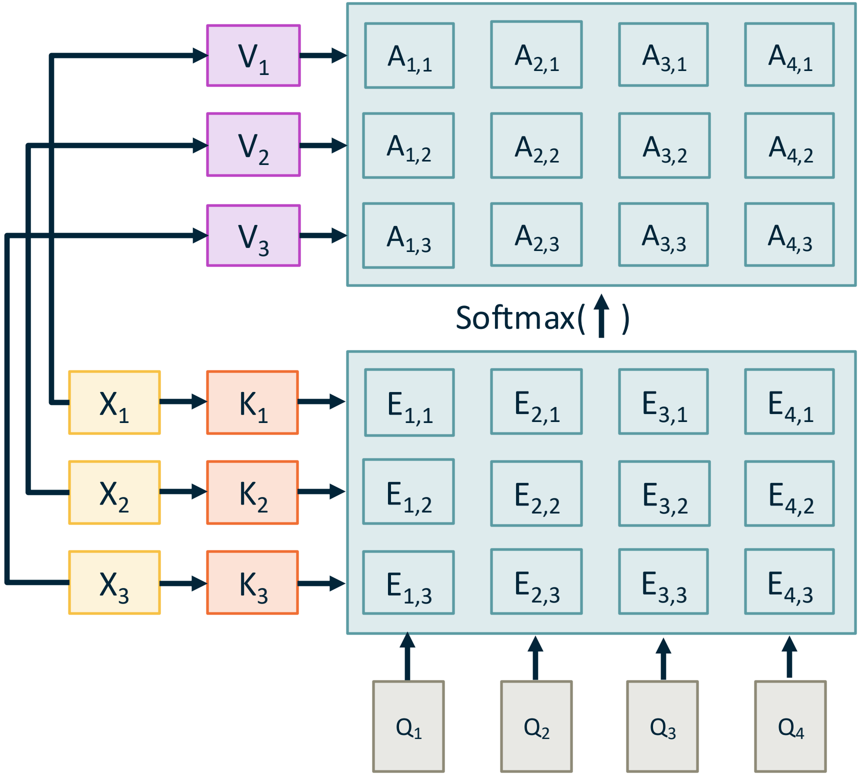


Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: Q (Shape: $N_Q \times D_Q$)
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)

Computation:
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$



Slide credit: Justin Johnson

Attention Layer

Inputs:

Query vectors: Q (Shape: N_Q x D_Q)

Input vectors: X (Shape: N_X x D_X)

Key matrix: W_K (Shape: D_X x D_Q)

Value matrix: W_V (Shape: D_X x D_V)

Computation:

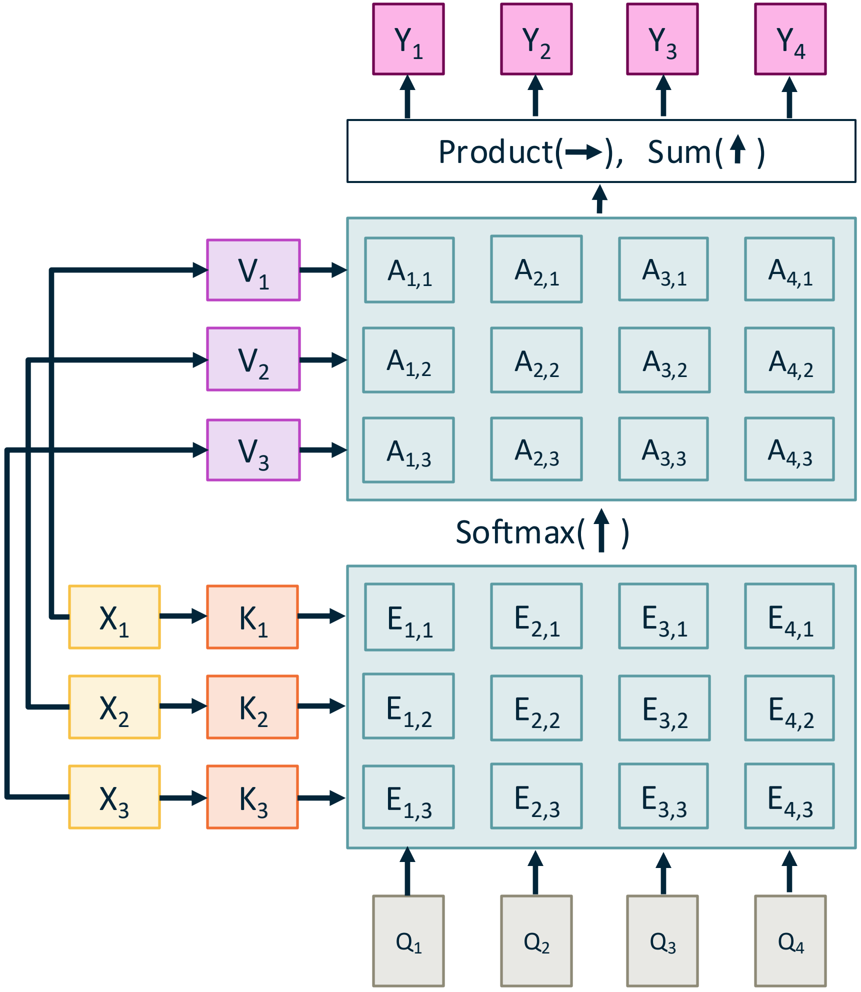
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)

Value vectors: $V = XW_V$ (Shape: N_X x D_V)

Similarities: $E = QK^T$ (Shape: N_Q x N_X) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$

Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: N_Q x N_X)

Output vectors: $Y = AV$ (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} V_j$

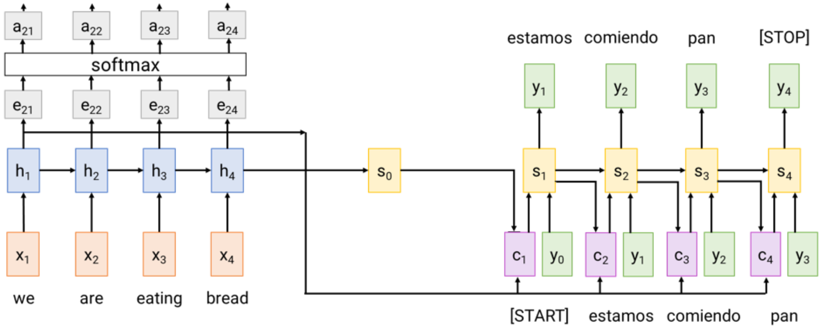


Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: Q (Shape: $N_Q \times D_Q$)
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)

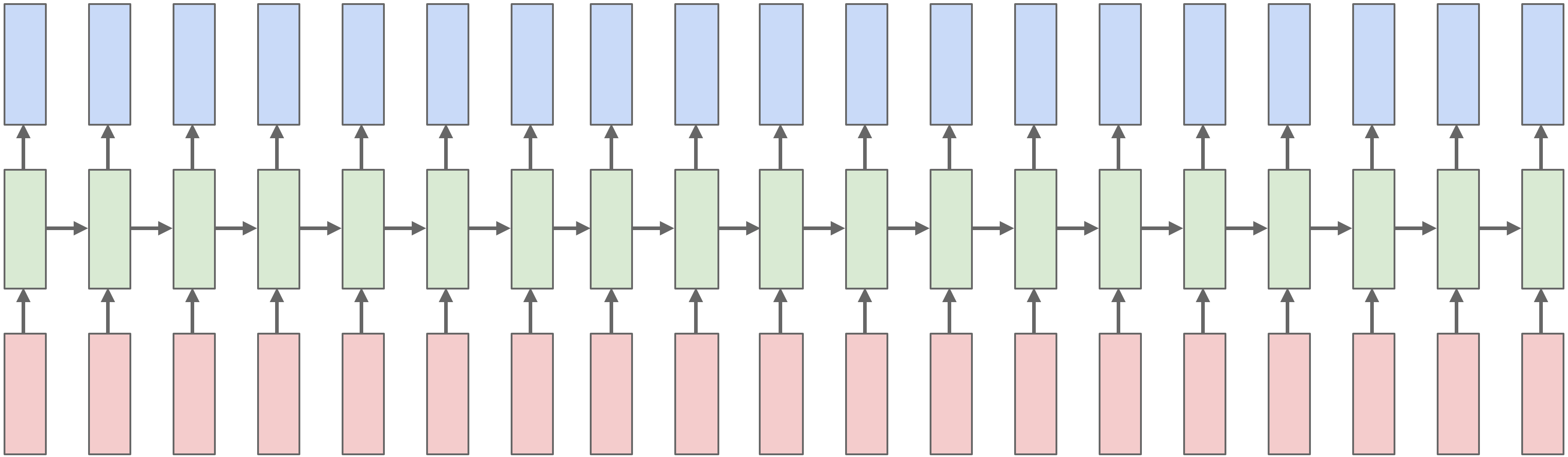
Computation:
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_Q \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$



Attention seems to be really powerful …
Do we still need RNN?

Slide credit: Justin Johnson

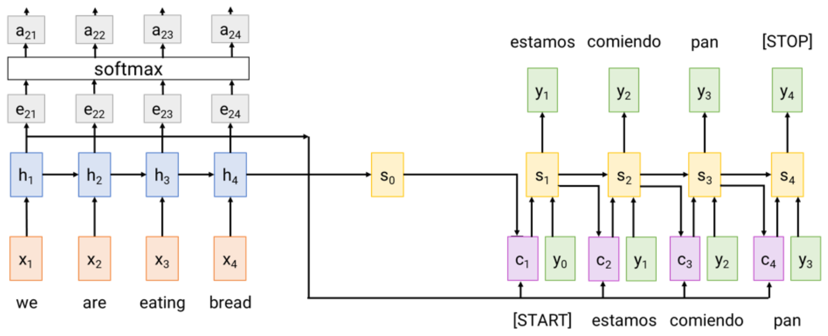
RNN is bad at encoding long-range relationships!



Recurrent update can easily "forget" information

Attention Layer

Inputs:
Query vectors: **Q** (Shape: N_Q x D_Q)
Input vectors: **X** (Shape: N_X x D_X)
Key matrix: $\mathbf{W_K}$ (Shape: D_X x D_Q)
Value matrix: $\mathbf{W_V}$ (Shape: D_X x D_V)



Attention seems to be really powerful …
Do we still need RNN?

Can we use **only attention layers** to encode an entire sequence?

Computation:
Key vectors: **K** = $\mathbf{XW_K}$ (Shape: N_X x D_Q)
Value vectors: **V** = $\mathbf{XW_V}$ (Shape: N_X x D_V)
Similarities: E = $\mathbf{QK}^T$ (Shape: N_Q x N_X) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \mathrm{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = A**V** (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$

Slide credit: Justin Johnson

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

"The Transformer Paper"

Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
Query vectors: Q (Shape: N_Q x D_Q)
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$

Without recurrent-based encoding, still need to somehow represent inter-token connection in the input sequence.

Goal: encode the input sequence with only attention, without a recurrent network.

X_1 X_2 X_3

Slide credit: Justin Johnson

Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
~~**Query vectors**: Q (Shape: N_Q x D_Q)~~
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$

Goal: encode the input sequence with only attention, without a recurrent network.

Encoding only -> no external queries
Use each element to query other elements in the same sequence, i.e., "self-attention"

X_1 X_2 X_3

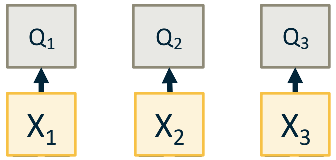
Slide credit: Justin Johnson

Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$



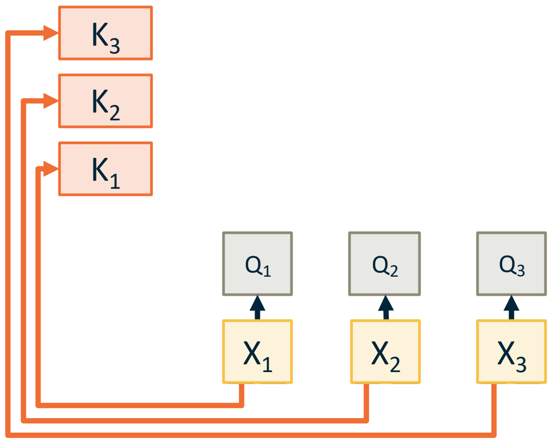
Slide credit: Justin Johnson

Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, dim=1)$ (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



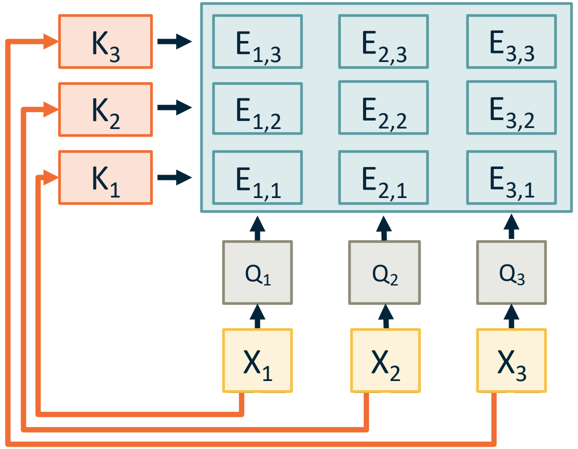
Slide credit: Justin Johnson

Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$



Slide credit: Justin Johnson

Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:

Input vectors: X (Shape: N_X x D_X)

Key matrix: W_K (Shape: D_X x D_Q)

Value matrix: W_V (Shape: D_X x D_V)

Query matrix: W_Q (Shape: D_X x D_Q)

Computation:

Query vectors: $Q = XW_Q$

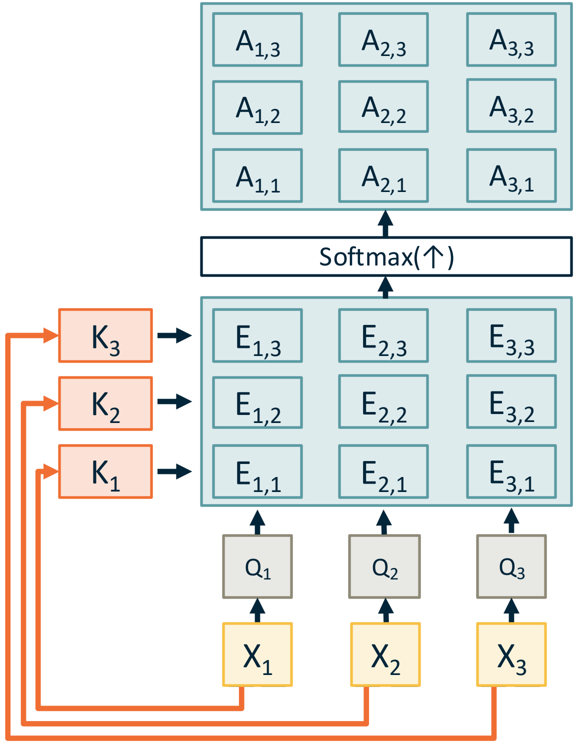
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)

Value vectors: $V = XW_V$ (Shape: N_X x D_V)

Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$

Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: N_X x N_X)

Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



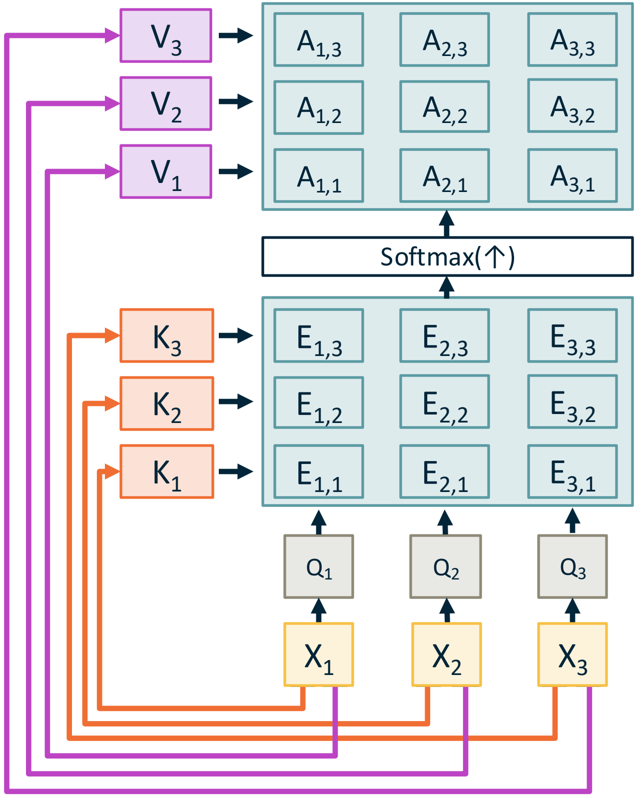
Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Q,K,V are all generated from X!



Slide credit: Justin Johnson

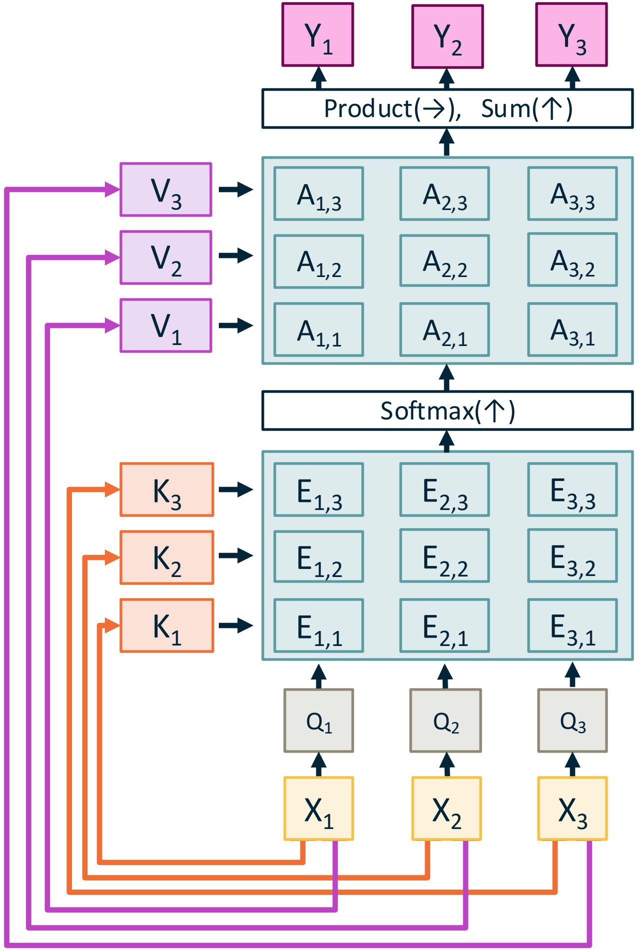
Self-Attention Layer

Sequence encode -> use each input element as query!

Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Q,K,V are all generated from X!



Slide credit: Justin Johnson

Self-Attention Layer

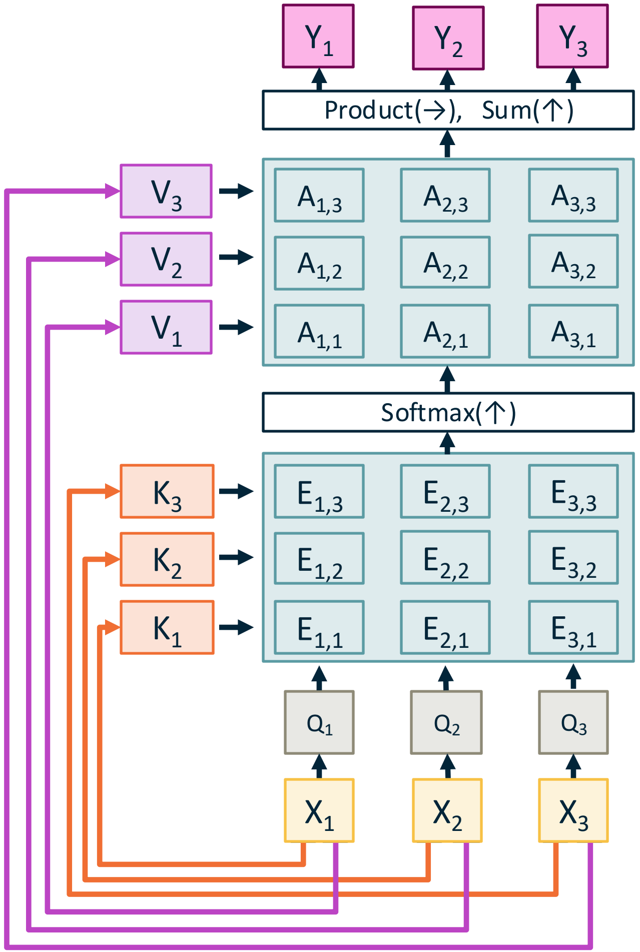
Sequence encode -> use each input element as query!

Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Q: Can we use self-attention to encode an input with specific sequential ordering?

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \dim=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Q,K,V are all generated from X!



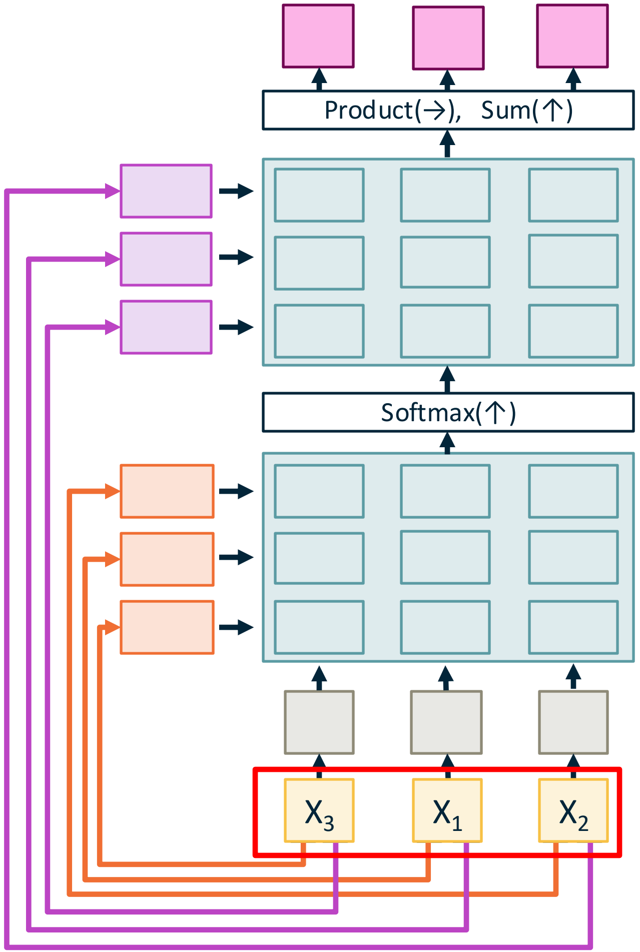
Slide credit: Justin Johnson

Self-Attention Layer

Consider **permuting** the input vectors:

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Slide credit: Justin Johnson

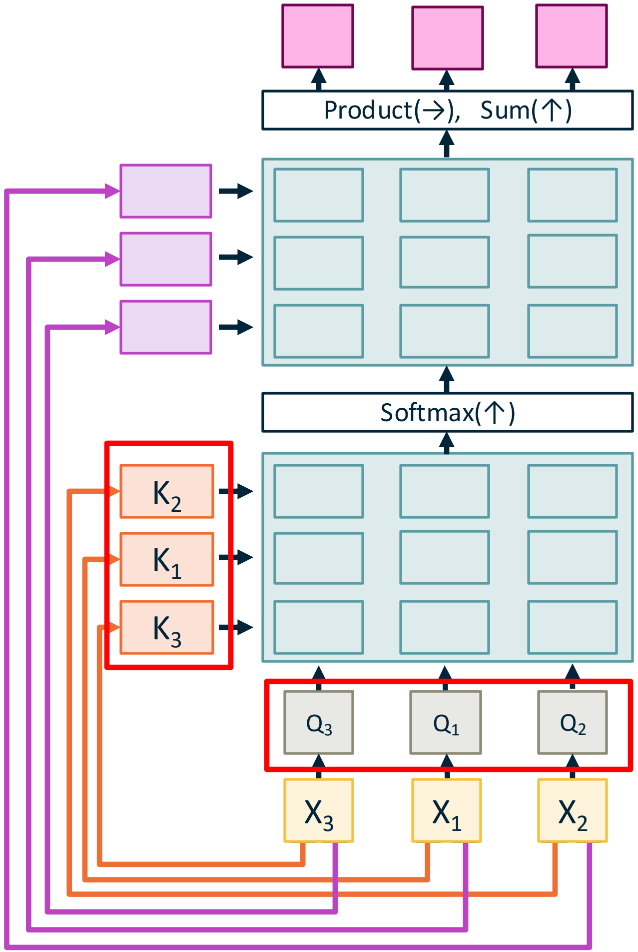
Self-Attention Layer

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value Vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting** the input vectors:

Queries and Keys will be the same, but permuted



Slide credit: Justin Johnson

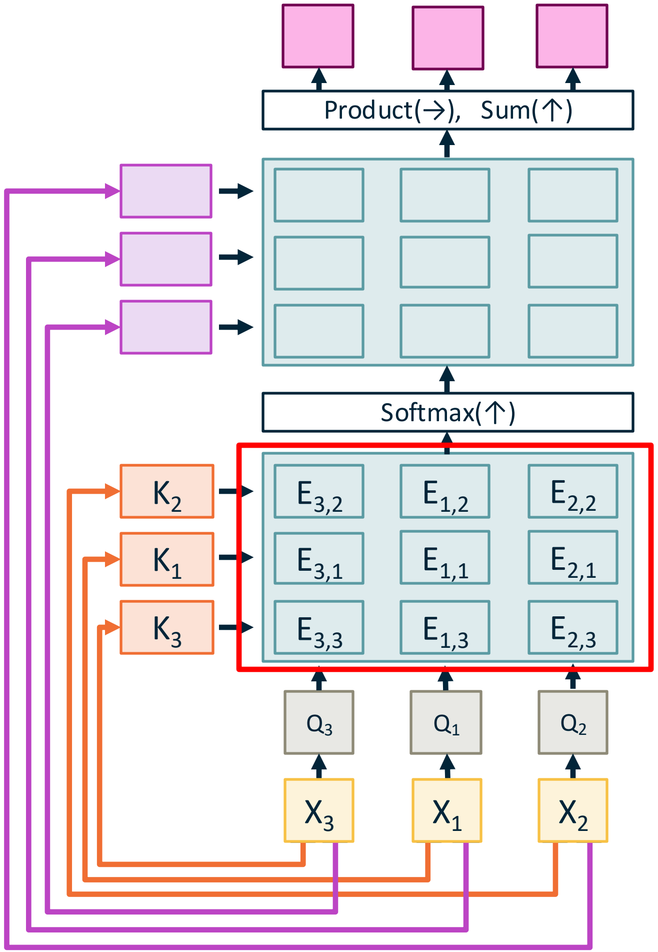
Self-Attention Layer

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Consider **permuting** the input vectors:

Similarities will be the same, but permuted

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$

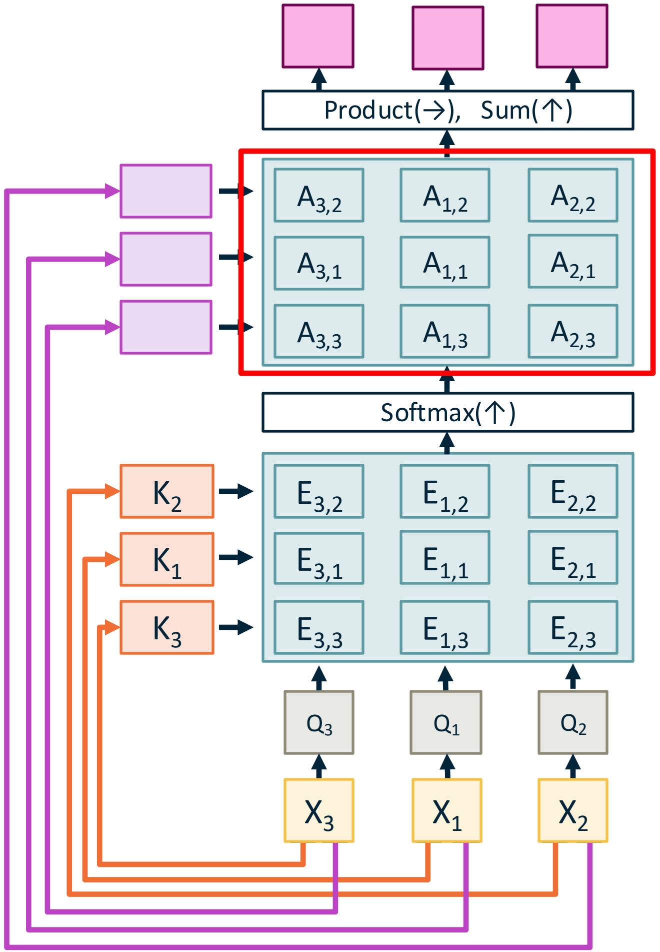


Slide credit: Justin Johnson

Self-Attention Layer

Consider **permuting** the input vectors:

Attention weights will be the same, but permuted



Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Slide credit: Justin Johnson

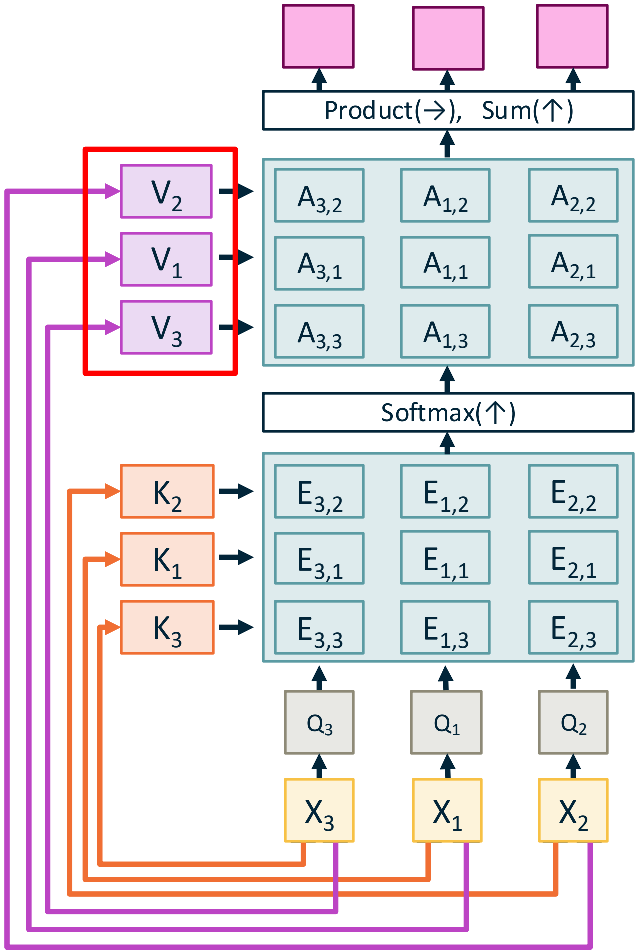
Self-Attention Layer

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Consider **permuting** the input vectors:

Values will be the same, but permuted

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Slide credit: Justin Johnson

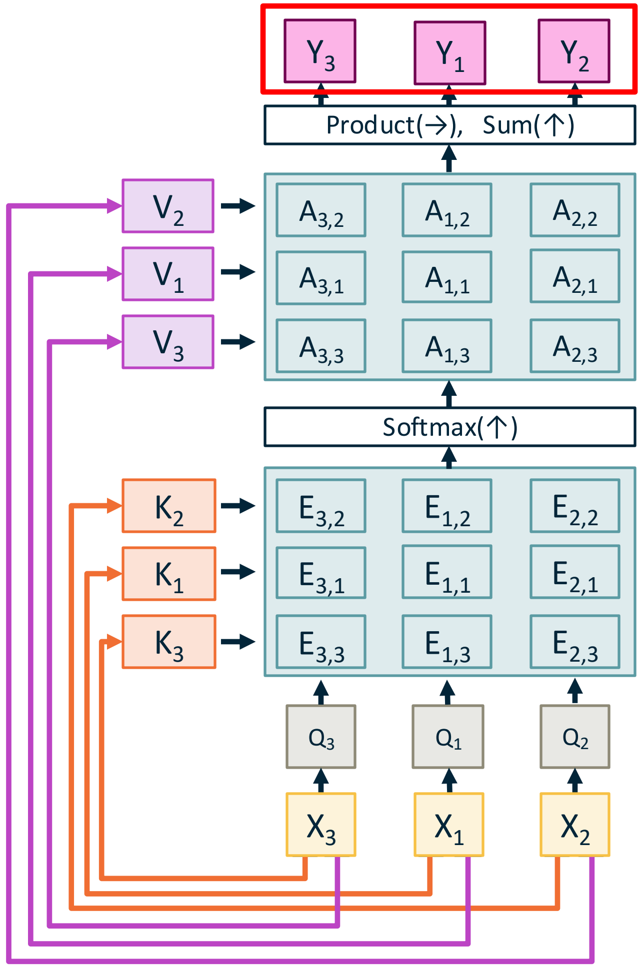
Self-Attention Layer

Consider **permuting** the input vectors:

Outputs will be the same, but permuted

<u>**Inputs**</u>:
Input vectors: **X** (Shape: $N_X \times D_X$)
Key matrix: $\mathbf{W_K}$ (Shape: $D_X \times D_Q$)
Value matrix: $\mathbf{W_V}$ (Shape: $D_X \times D_V$)
Query matrix: $\mathbf{W_Q}$ (Shape: $D_X \times D_Q$)

<u>**Computation**</u>:
Query vectors: $\mathbf{Q} = \mathbf{XW_Q}$
Key vectors: $\mathbf{K} = \mathbf{XW_K}$ (Shape: $N_X \times D_Q$)
Value vectors: $\mathbf{V} = \mathbf{XW_V}$ (Shape: $N_X \times D_V$)
Similarities: $E = \mathbf{QK^T}$ (Shape: $N_X \times N_X$) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = A\mathbf{V}$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j}\mathbf{V_j}$



Slide credit: Justin Johnson

Self-Attention Layer

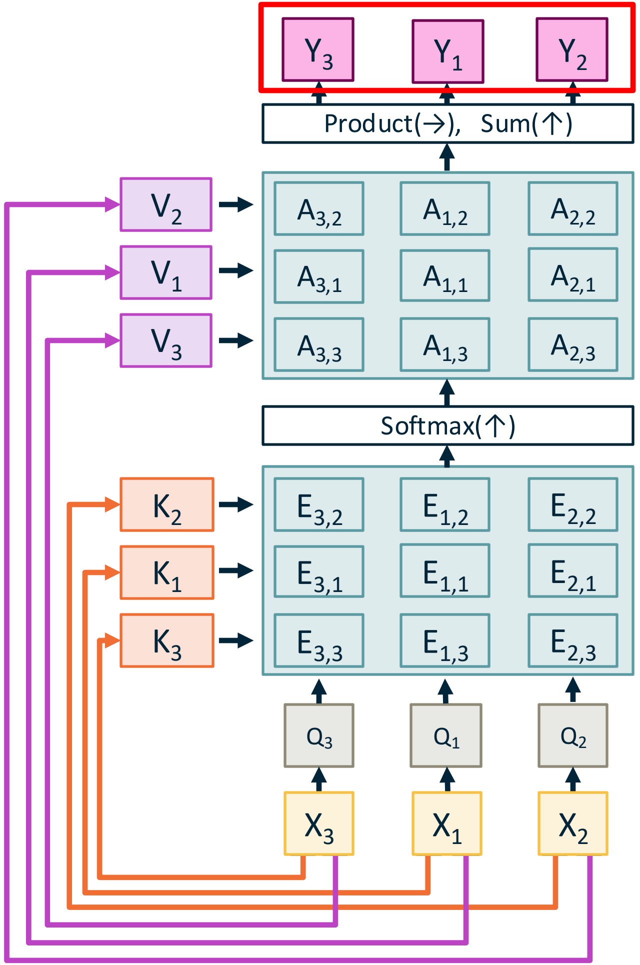
Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting** the input vectors:

Outputs will be the same, but permuted

Self-attention layer is **Permutation Equivariant**
f(s(x)) = s(f(x))



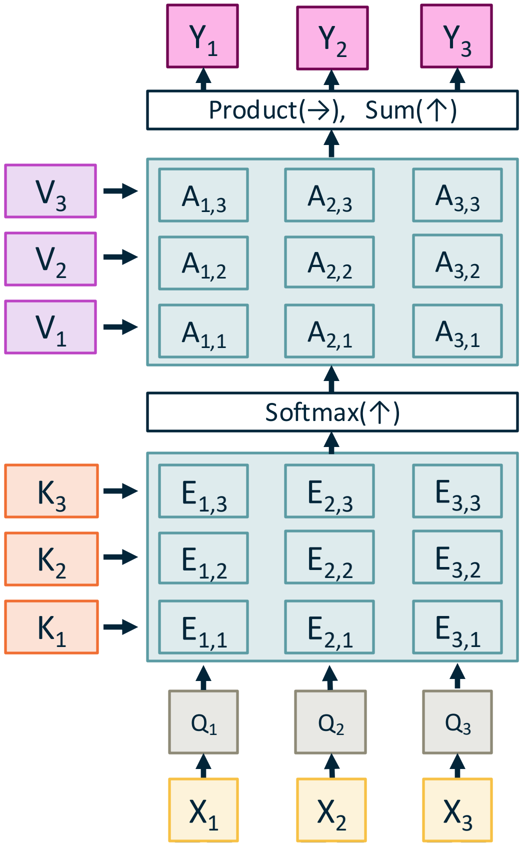
Slide credit: Justin Johnson

Self-Attention Layer

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Self attention doesn't "know" the order of the vectors it is processing! Not good for sequence encoding.

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Self-Attention Layer

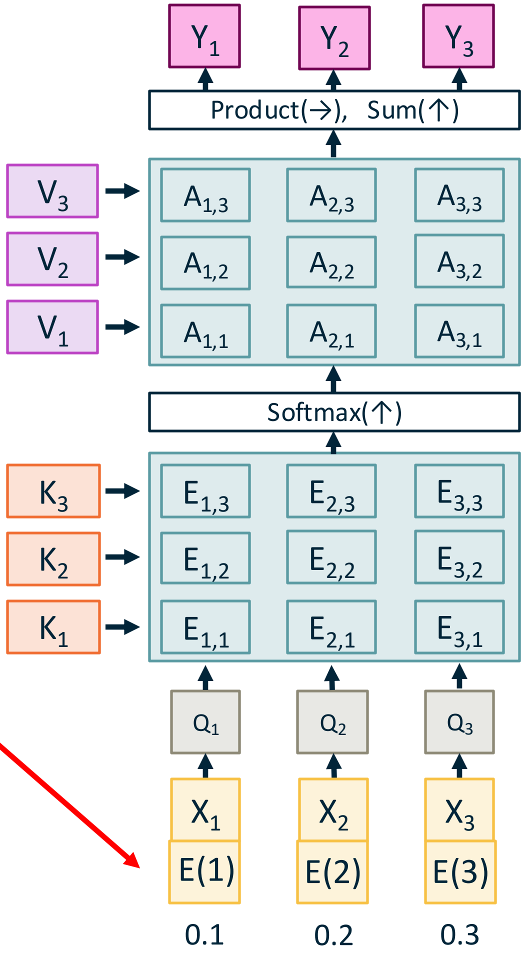
Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

In order to make processing position-aware, concatenate input with **positional encoding E**

E(i) encodes the position of the the i-th element in a sequence

E() can be a simple function (e.g., linear or sin functions) or a learned lookup table.



Slide credit: Justin Johnson

Aside: Positional Encoding (PE) for Self-Attention

Motivation: Maintain the order of input data since attention mechanisms are permutation invariant. PEs are shared across all input sequences.

Linear Positional Encoding: $PE(pos) = a \cdot pos + b.$
Problem: encoding increases with the sequence length, causing gradient problem for long sequences.

Sin/cos Positional Encoding (Default):

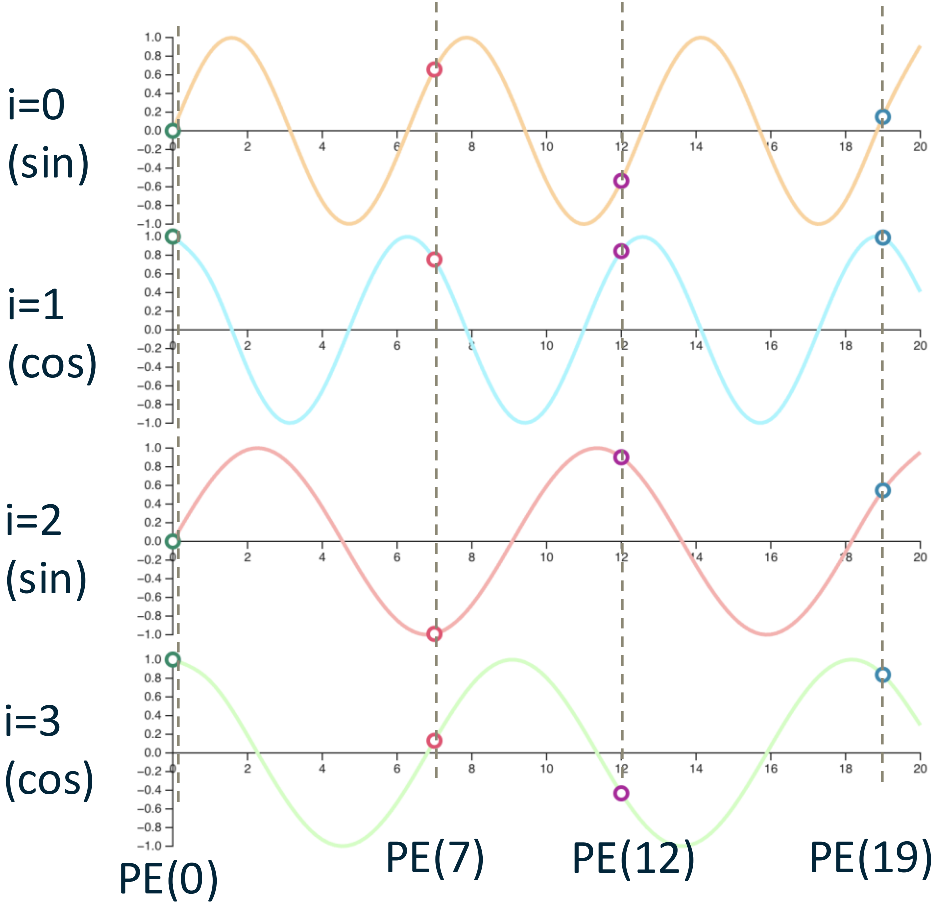
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

PE for each dimension (i) repeats periodically, combine different waveforms at each dimension to get a unique embedding.

Slide credit: Justin Johnson

Aside: Positional Encoding (PE) for Self-Attention



p0	p1	p2	p3	
0.000	0.657	-0.537	0.150	i=0
1.000	0.754	0.844	0.989	i=1
0.000	-0.992	0.901	0.547	i=2
1.000	0.130	-0.433	0.837	i=3

Positional Encoding

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

i determines the sin/cos period

d_{model}=50

https://erdem.pl/2021/05/understanding-positional-encoding-in-transformers

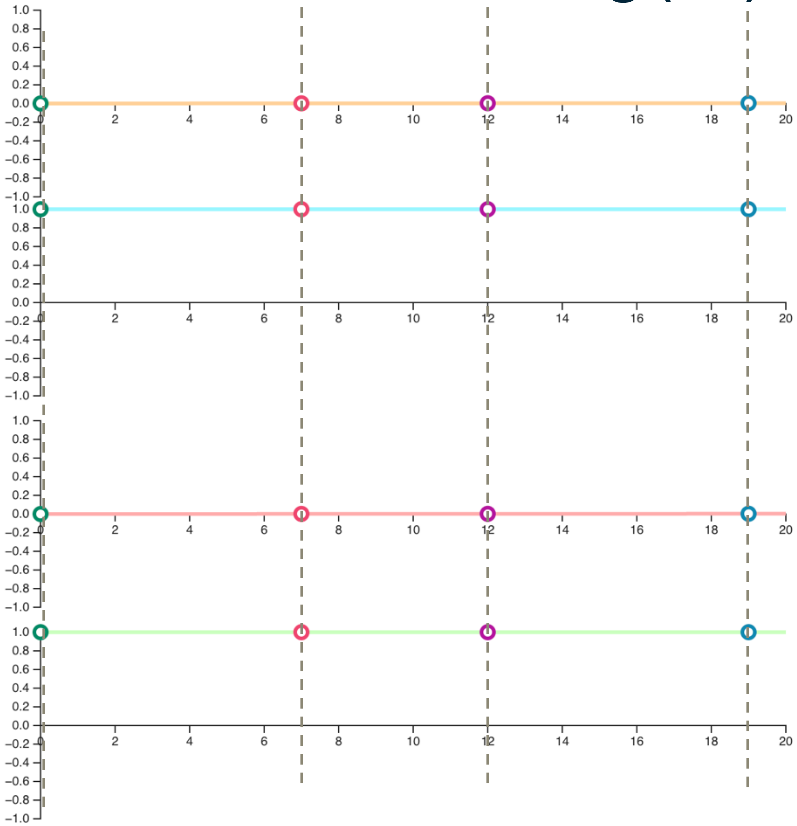
Aside: Positional Encoding (PE) for Self-Attention

i=46 (sin)

i=47 (cos)

i=48 (sin)

i=49 (cos)



PE(0) PE(7) PE(12) PE(19)

	p0	p1	p2	p3
i=46	0.000	0.001	0.003	0.004
i=47	1.000	1.000	1.000	1.000
i=48	0.000	0.001	0.002	0.003
i=49	1.000	1.000	1.000	1.000

Important! indexes has changed

Positional Encoding

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

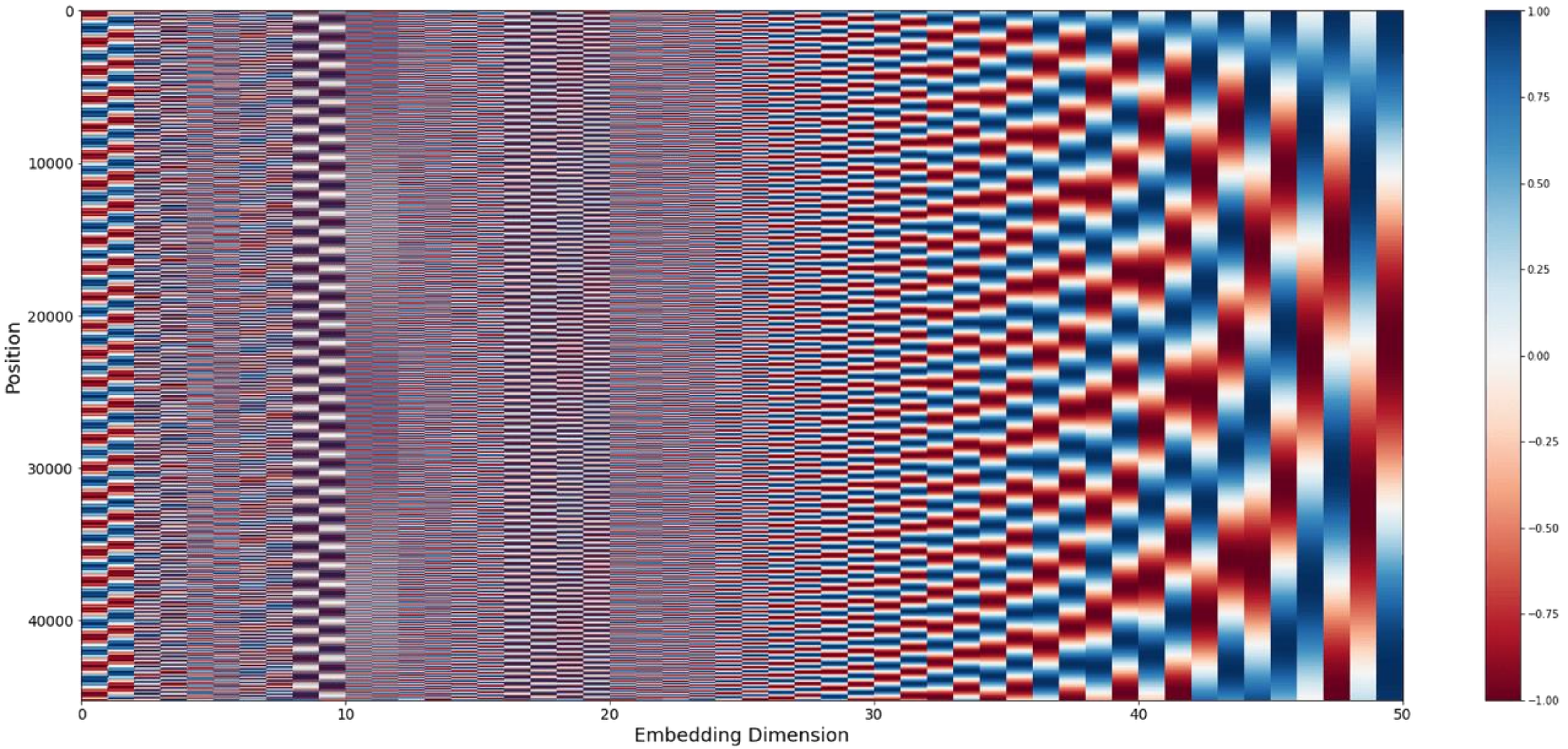
$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

i determines the sin/cos period

d_{model}=50

https://erdem.pl/2021/05/understanding-positional-encoding-in-transformers

Aside: Positional Encoding (PE) for Self-Attention



https://erdem.pl/2021/05/understanding-positional-encoding-in-transformers

Aside: Positional Encoding (PE) for Self-Attention

Motivation: Maintain the order of input data since attention mechanisms are permutation invariant. PEs are shared across all input sequences.

Linear Positional Encoding: $PE(pos) = \ a \cdot pos \ + \ b.$
Problem: encoding increases with the sequence length, causing gradient problem for long sequences.

Sin/cos Positional Encoding (Default):

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

PE for each dimension (i) repeats periodically, combine different waveforms at each dimension to get a unique embedding.

Learned Positional Encoding: $PE_\theta(pos, i)$.
Learn the most suitable position embedding for the training set.

Slide credit: Justin Johnson

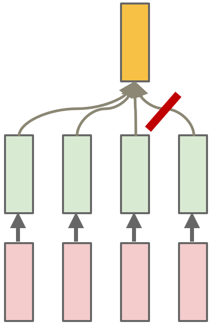
Masked Self-Attention Layer

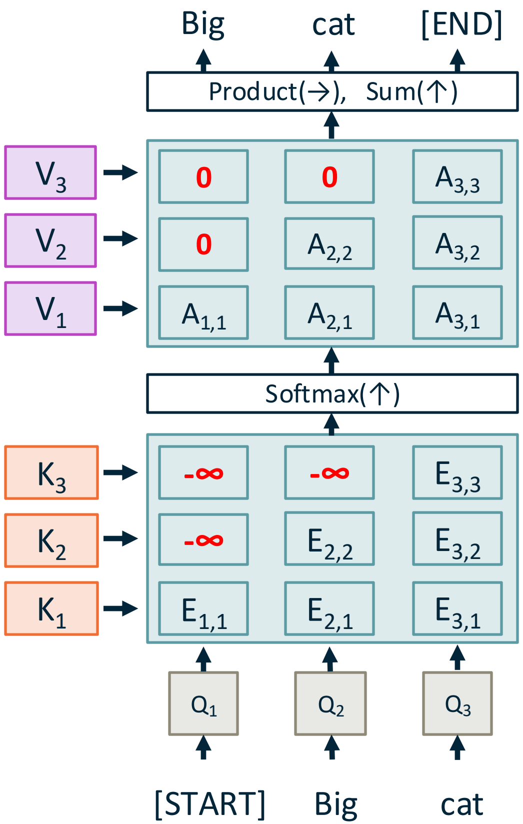
Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

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

Used for sequence decoding (predict next word)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$





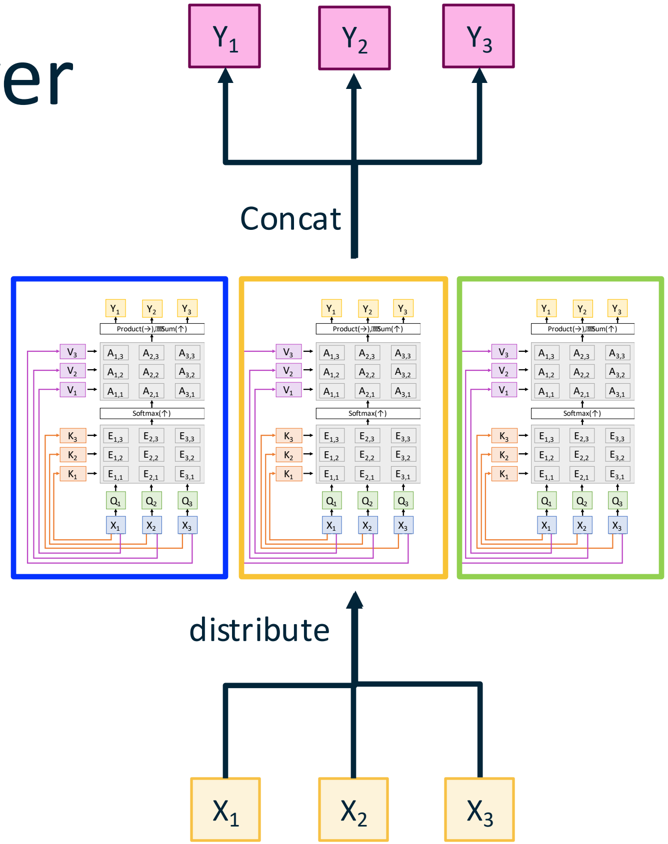
Slide credit: Justin Johnson

Multi-headed Self-Attention Layer

Inputs:
Input vectors: X (Shape: N_X x D_X)
Key matrix: W_K (Shape: D_X x D_Q)
Value matrix: W_V (Shape: D_X x D_V)
Query matrix: W_Q (Shape: D_X x D_Q)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: N_X x D_Q)
Value vectors: $V = XW_V$ (Shape: N_X x D_V)
Similarities: $E = QK^T$ (Shape: N_X x N_X) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (Shape: N_X x N_X)
Output vectors: $Y = AV$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$

Use H independent "Attention Heads" in parallel



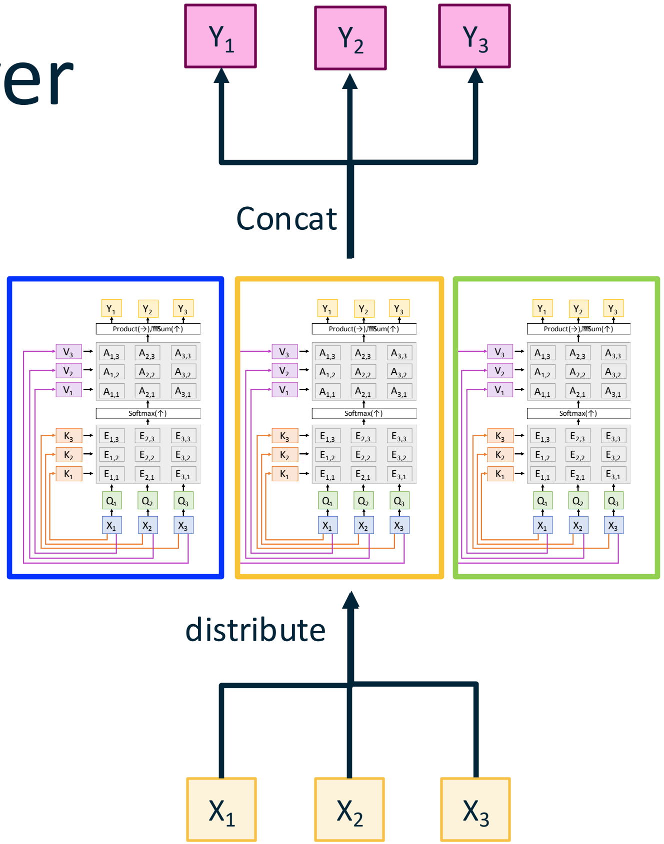
Slide credit: Justin Johnson

Multi-headed Self-Attention Layer

Inputs:
Input vectors: X (Shape: $N_X \times D_X$)
Key matrix: W_K (Shape: $D_X \times D_Q$)
Value matrix: W_V (Shape: $D_X \times D_V$)
Query matrix: W_Q (Shape: $D_X \times D_Q$)

Computation:
Query vectors: $Q = XW_Q$
Key vectors: $K = XW_K$ (Shape: $N_X \times D_Q$)
Value vectors: $V = XW_V$ (Shape: $N_X \times D_V$)
Similarities: $E = QK^T$ (Shape: $N_X \times N_X$) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Use H independent "Attention Heads" in parallel



Highly parallelizable: Can compute attentions for all input element from all head in parallel!

Slide credit: Justin Johnson

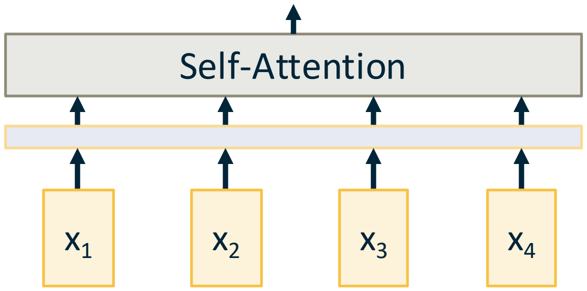
The Transformer Block

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

Slide credit: Justin Johnson

The Transformer Block

All vectors interact with each other



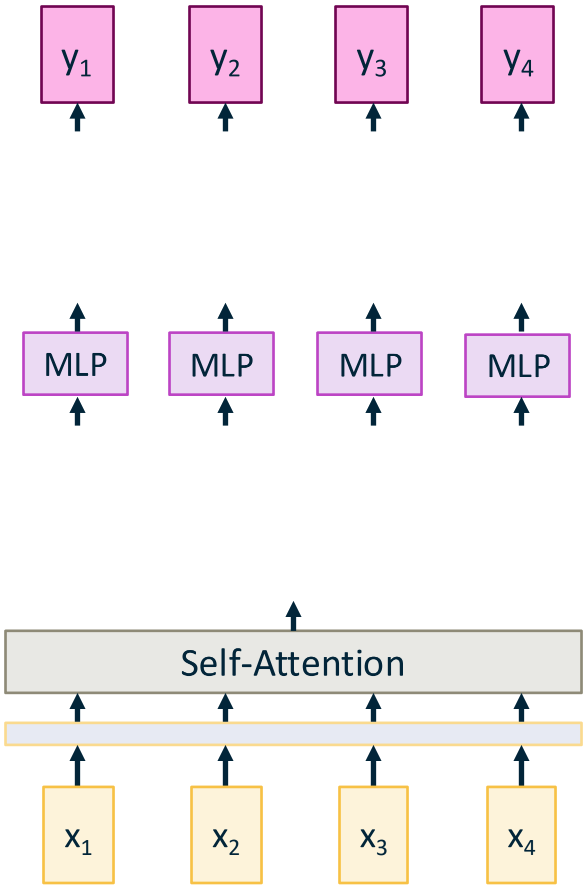
Vaswani et al, "Attention is all you need", NeurIPS 2017

Slide credit: Justin Johnson

The Transformer Block

MLP (same copy) independently on each vector

All vectors interact with each other



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

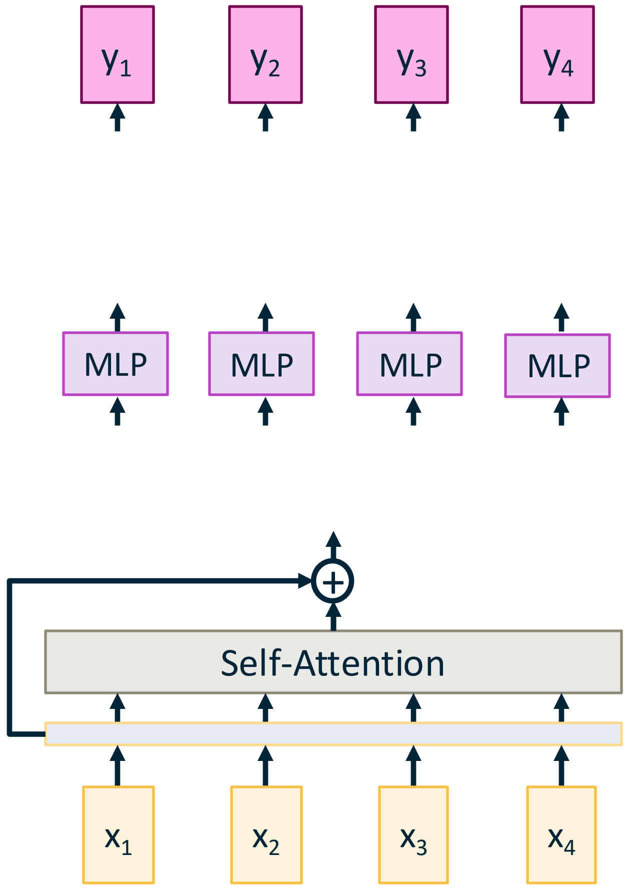
Slide credit: Justin Johnson

The Transformer Block

MLP (same copy) independently on each vector

Residual connection

All vectors interact with each other



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

Slide credit: Justin Johnson

The Transformer Block

Recall **Layer Normalization**:

Given $h_1, \ldots, h_N$ (shape: D)

scale: γ (shape: D)

shift: β (shape: D)

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

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

$z_i = (h_i - \mu_i) / \sigma_i$ (shape: D)

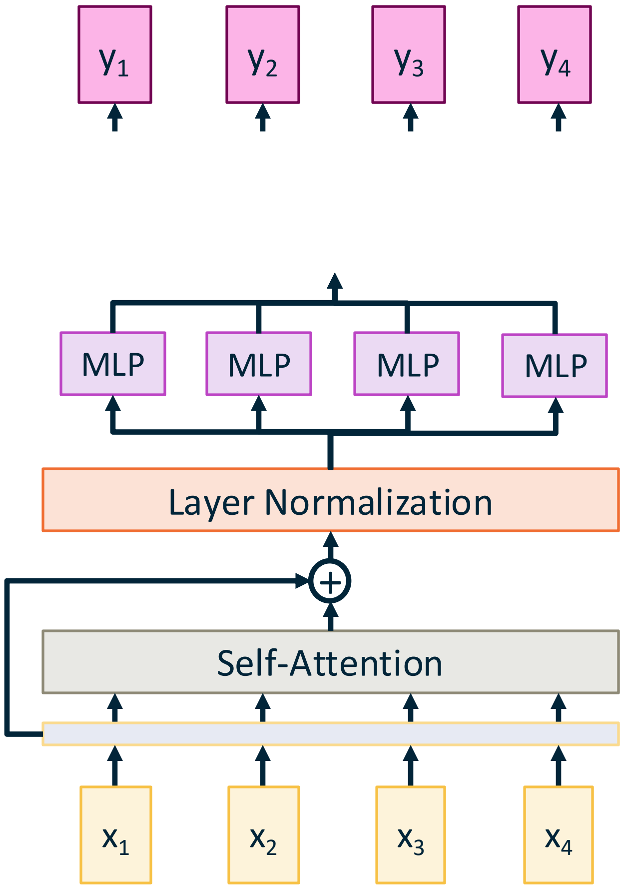
$y_i = \gamma * z_i + \beta$ (shape: D)

Applied **per dimension**, not across the sequence

MLP (same copy) independently on each vector

Residual connection

All vectors interact with each other



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

Slide credit: Justin Johnson

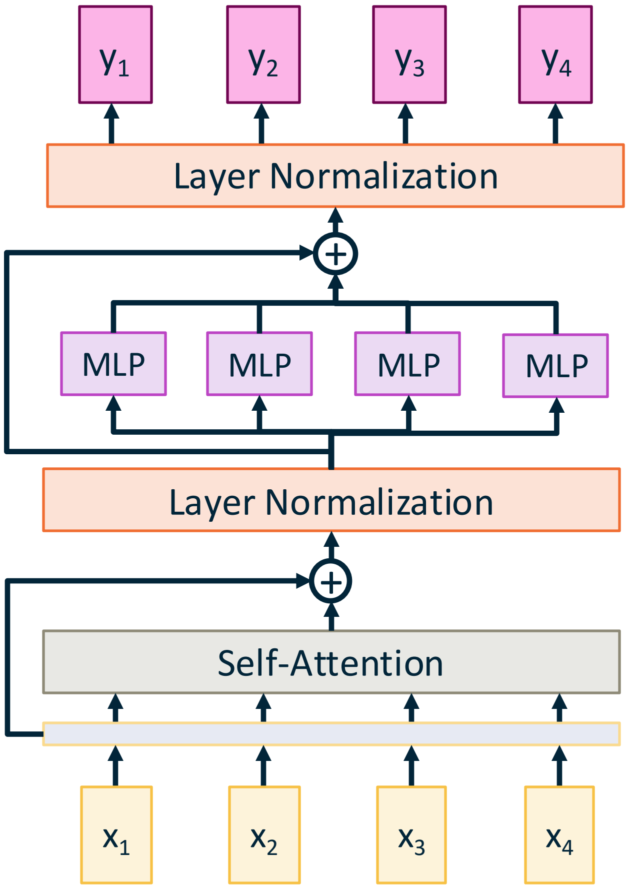
The Transformer Block

Residual connection

MLP (same copy) independently on each vector

Residual connection

All vectors interact with each other



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

Slide credit: Justin Johnson

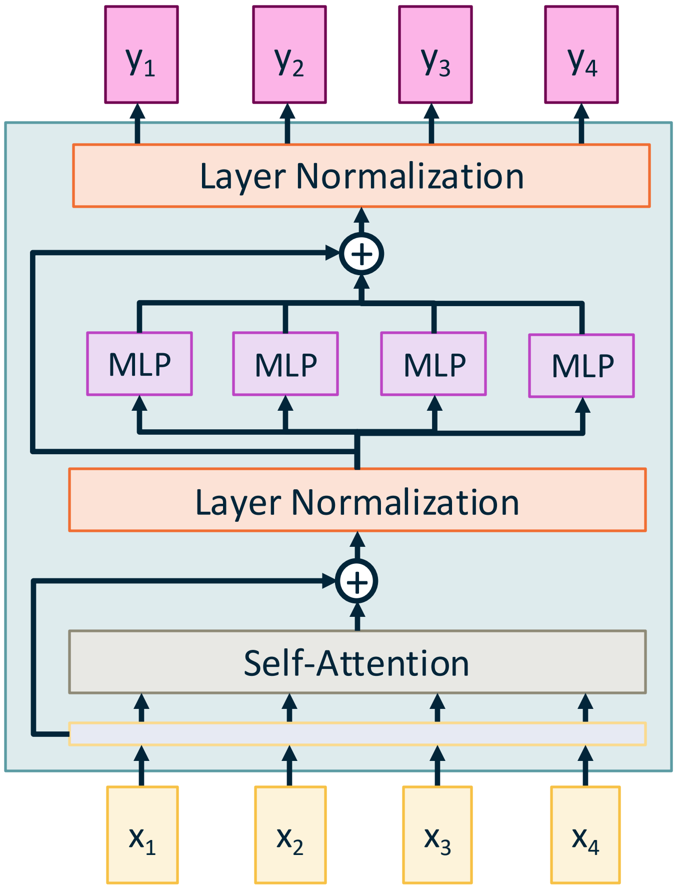
The Transformer Block

<u>**Transformer Block**</u>**:**
Input: Set of vectors x
Output: Set of vectors y

Self-attention is the only interaction among vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable



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

Slide credit: Justin Johnson

The Transformer

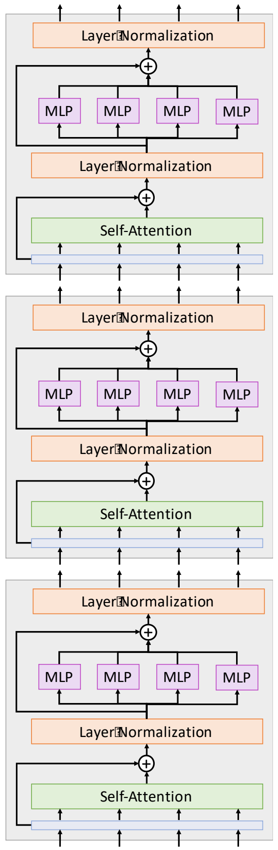
<u>**Transformer Block**</u>**:**
Input: Set of vectors x
Output: Set of vectors y

Self-attention is the only interaction among vectors!

Layer norm and MLP work independently per vector

Highly scalable, highly parallelizable

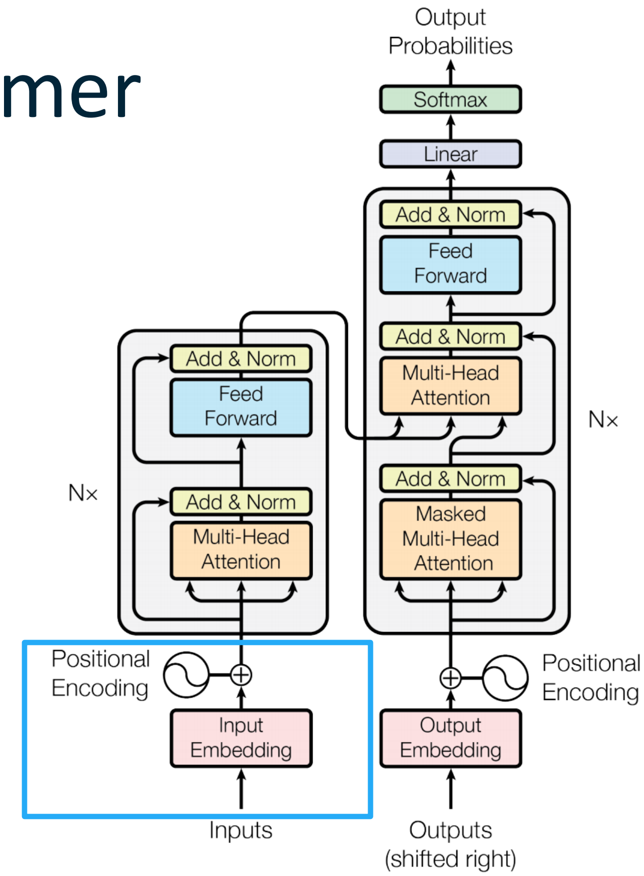
A **Transformer** is a sequence of transformer blocks



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

Slide credit: Justin Johnson

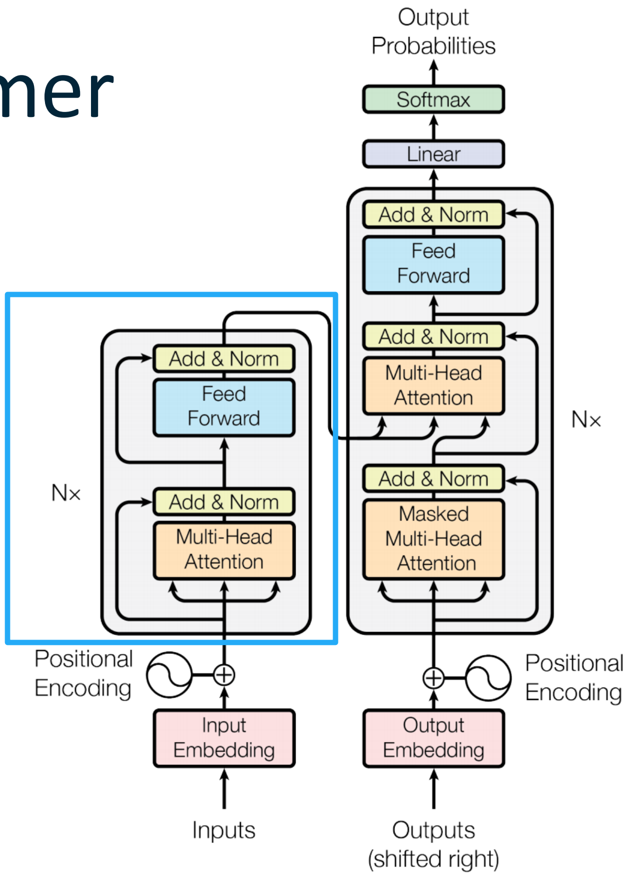
The Transformer



Encoder-Decoder

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

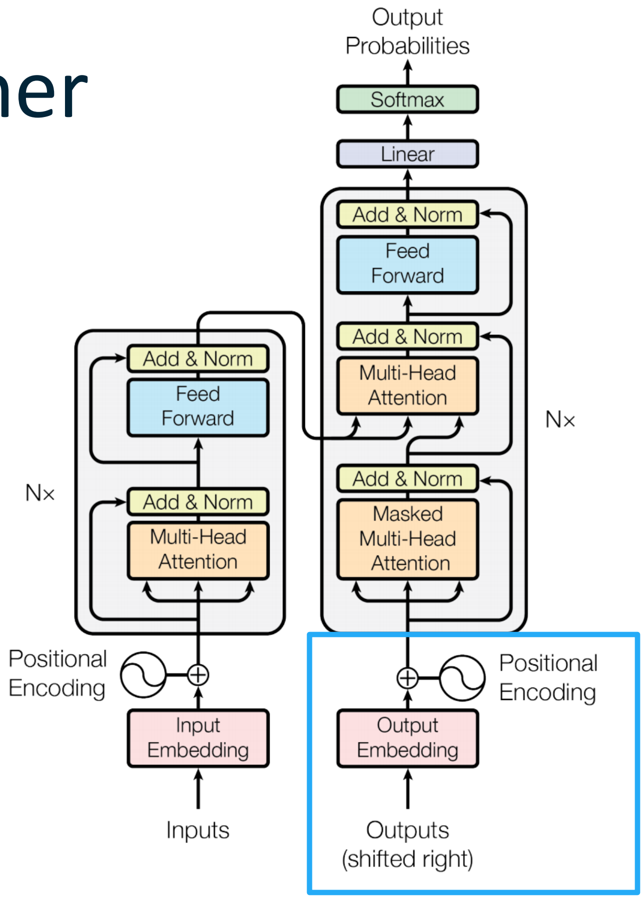
The Transformer

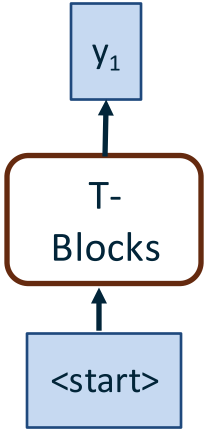


Encoder-Decoder

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

The Transformer

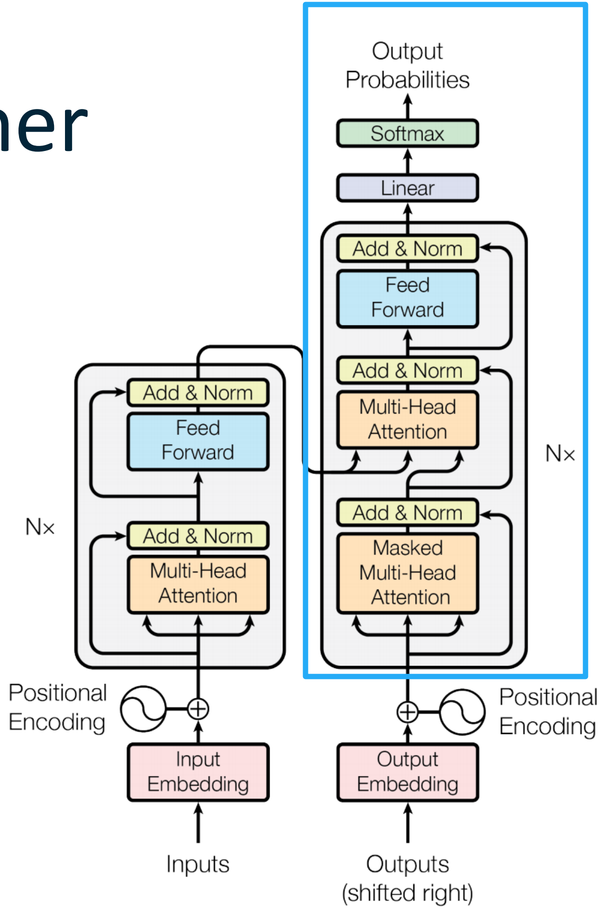




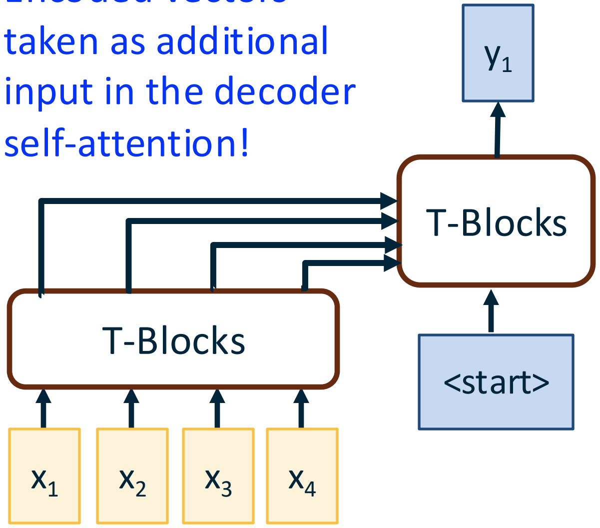
Encoder-Decoder

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

The Transformer



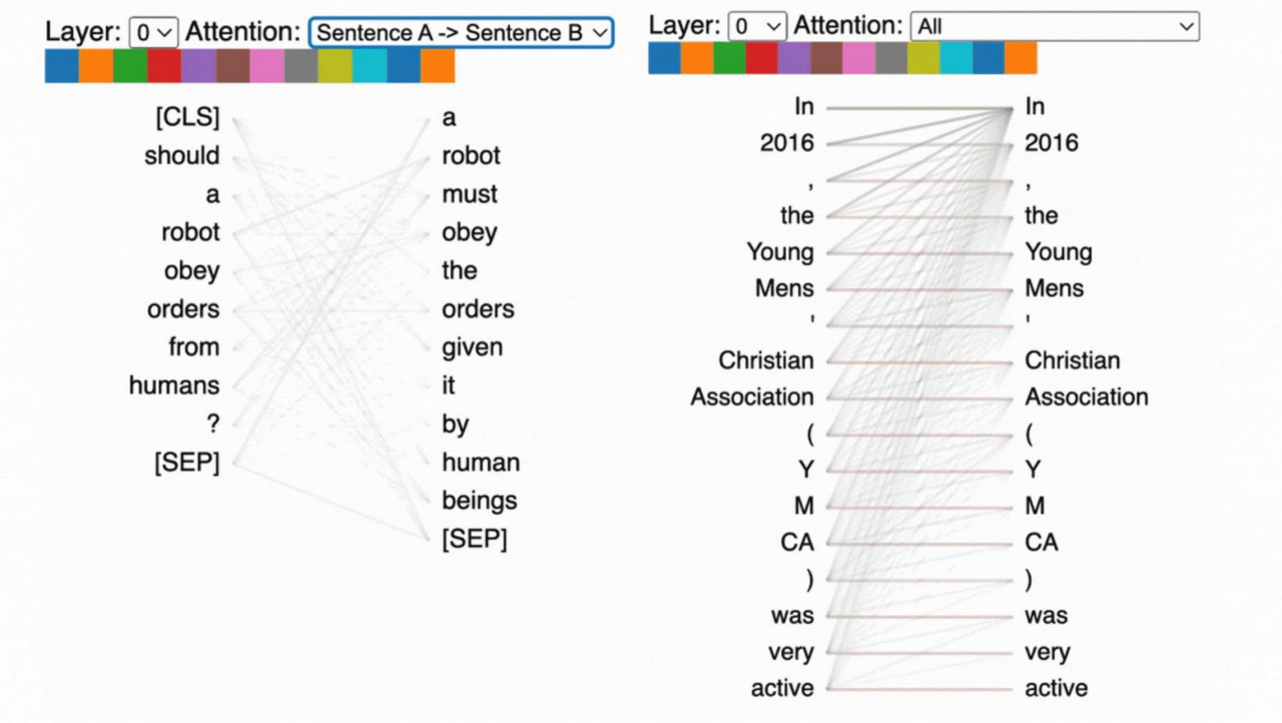
Encoded vectors taken as additional input in the decoder self-attention!



Encoder-Decoder

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

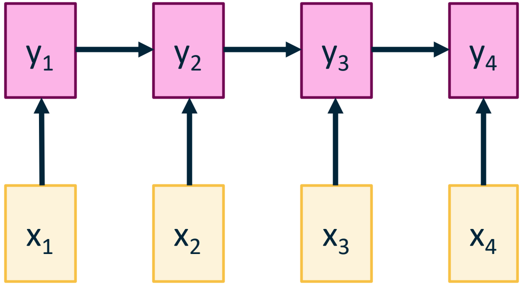
Visualizing Transformer Attentions



https://github.com/jessevig/bertviz

Three Ways of Processing Sequences

Recurrent Neural Network



Works on **Ordered Sequences**

(+) Natural sequential processing: "sees" the input sequence in its original ordering

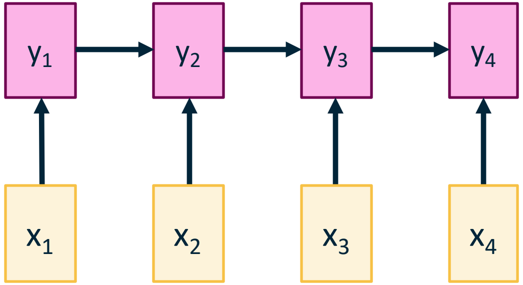
(-) Forgetful: difficult to handle long-range dependencies.

(-) Not parallelizable: need to compute hidden states sequentially

Slide credit: Justin Johnson

Three Ways of Processing Sequences

Recurrent Neural Network



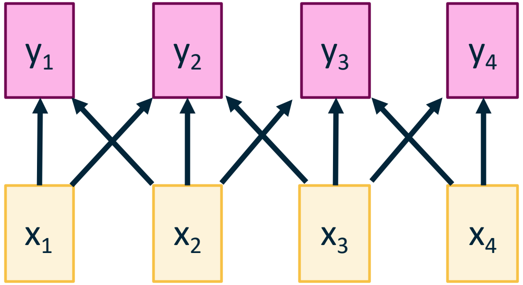
Works on **Ordered Sequences**

(+) Natural sequential processing: "sees" the input sequence in its original ordering

(-) Forgetful: difficult to handle long-range dependencies.

(-) Not parallelizable: need to compute hidden states sequentially

1D Convolution



Works on **Multidimensional Grids**

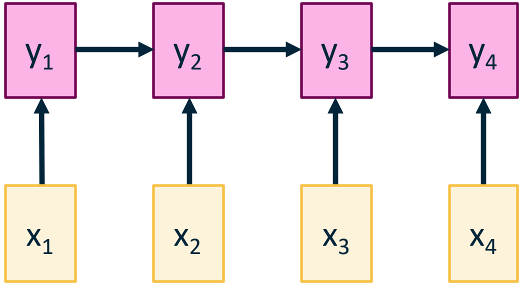
(-) Bad at long sequences: Need to stack many conv layers for outputs to "see" the whole sequence

(+) Highly parallel: Each output can be computed in parallel

Slide credit: Justin Johnson

Three Ways of Processing Sequences

Recurrent Neural Network



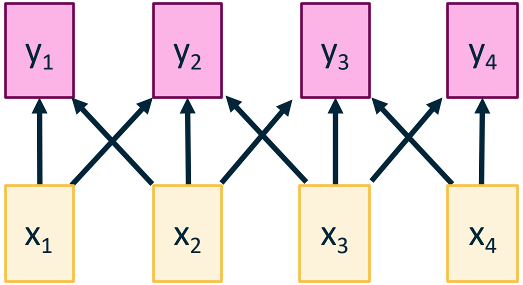
Works on **Ordered Sequences**

(+) Natural sequential processing: "sees" the input sequence in its original ordering

(-) Forgetful: difficult to handle long-range dependencies.

(-) Not parallelizable: need to compute hidden states sequentially

1D Convolution

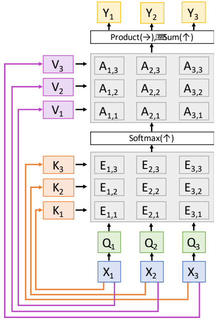


Works on **Multidimensional Grids**

(-) Bad at long sequences: Need to stack many conv layers for outputs to "see" the whole sequence

(+) Highly parallel: Each output can be computed in parallel

Self-Attention



Works on **Sets of Vectors**

(+) Good at long sequences: after one self-attention layer, each output "sees" all inputs!

(+) Highly parallel: Each output can be computed in parallel

(-) Very memory intensive

(-) Requires positional encoding

Slide credit: Justin Johnson

Some Recent Advances in Transformers

- **Compute Efficiency**: KV Cache, Grouped-query Attention
- **System-level optimization**: Paged Attention
- **Beyond language data**

Transformer Inference is Expensive!

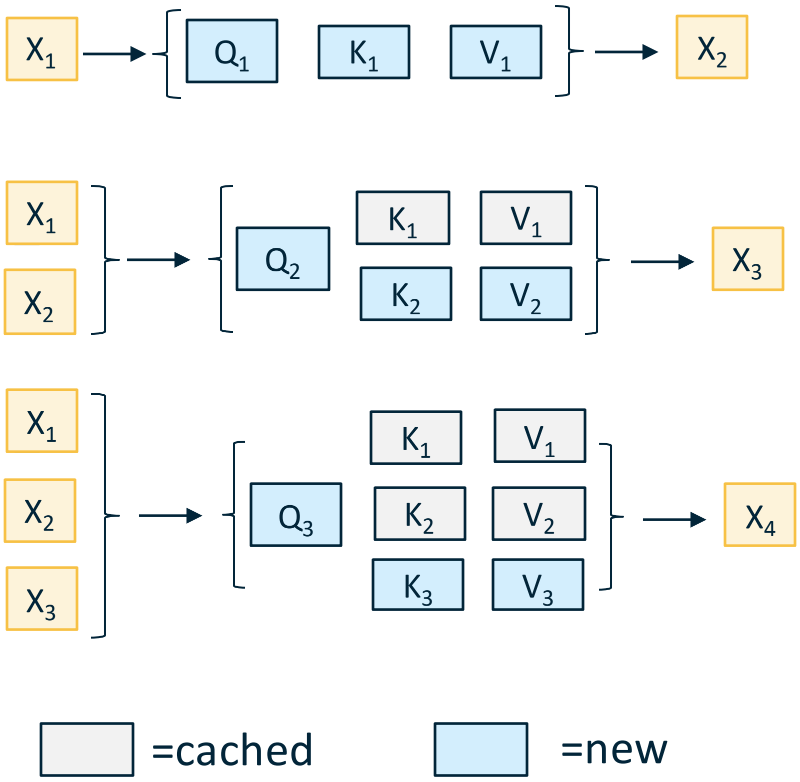
Q: What's the $O()$ complexity of generating a length-N sequence with vanilla RNN?
A: $O(N)$: you only process each generated token / item once

Q: What's the $O()$ complexity of generating a length-N sequence with Transformer?
A: $O(N^2)$. Generating a new token needs to attend to all existing tokens. Bad for long sequences!

Recall: $\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{KV^T}{\sqrt{d_k}})Q$

Observation: Only the query (Q) changes. No need to recompute K and V for the already-generated sequence!

KV Cache: Trading GPU Memory for Efficiency



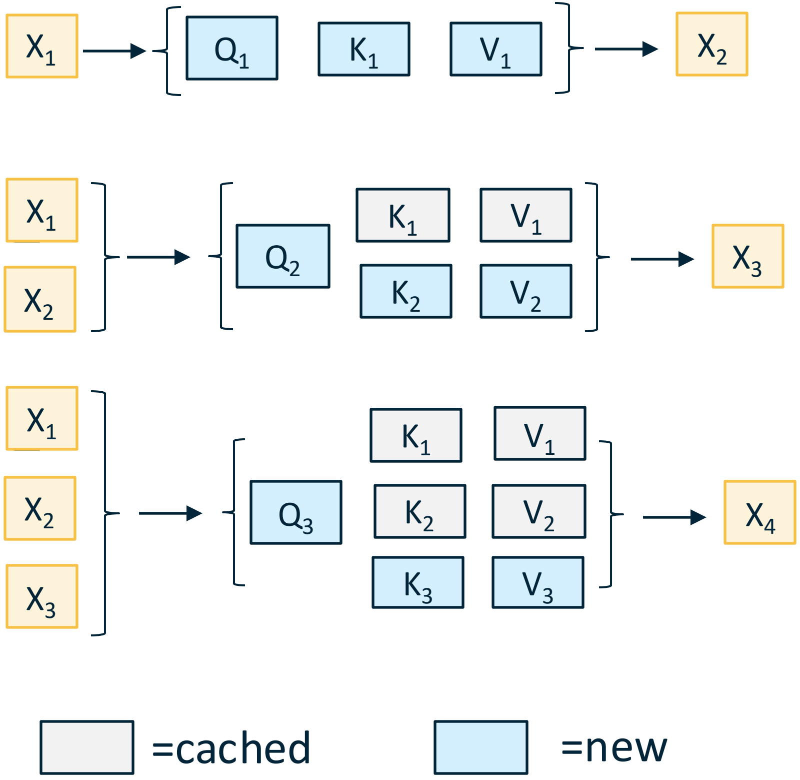
With KV Cache, complexity goes from quadratic to linear! $O(N)$

For ~1000 tokens, without KV cache requires 300 million attention operations per layer!

With KV cache: ~500,000 attention operations per layer.

600x speedup!

KV Cache: Trading GPU Memory for Efficiency



But we need to cache the KV for all of the intermediate attention layers

…

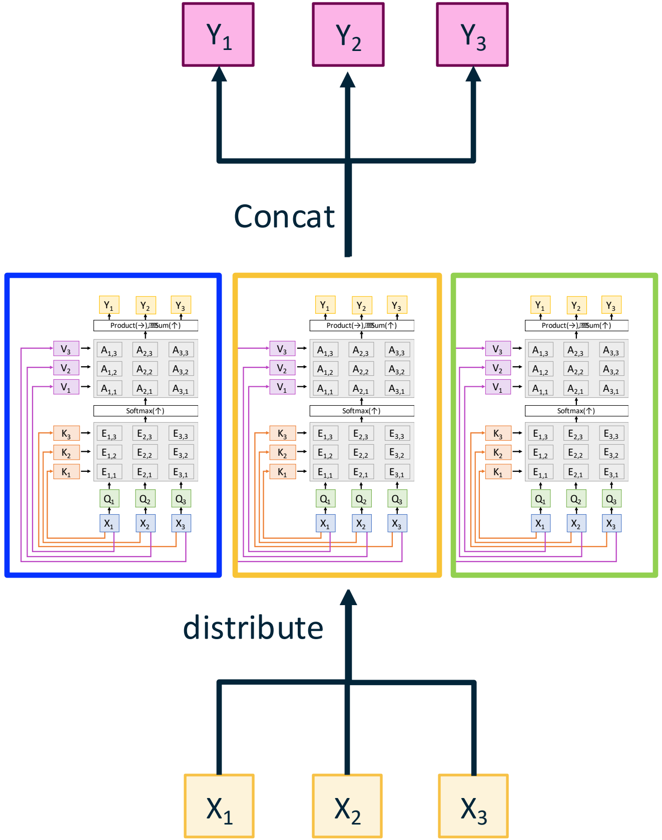
For a typical model: 32 layers X 4096 embedding dimension x 2 vectors

Takes **1GB of memory** to KV cache a **N=2048** sequence!

Optimizing KV Cache: Multi-Query Attention (MQA)

Recall Multi-headed Attention (MHA):

- Similar to having multiple kernels per layer in ConvNet, have multiple attention heads in each attention layer.
- Each head learns to attend to different patterns (syntax, semantics, long-range dependencies, etc.).
- The diversity comes from having separate initialization for each Q, K, V head.



Problem: For 32-head MHA, 32 K matrices + 32 V matrices = **64 matrices per layer.**

Big overhead for KV Cache!

Fast Transformer Decoding: One Write-Head is All You Need, Shazeer et al., 2019

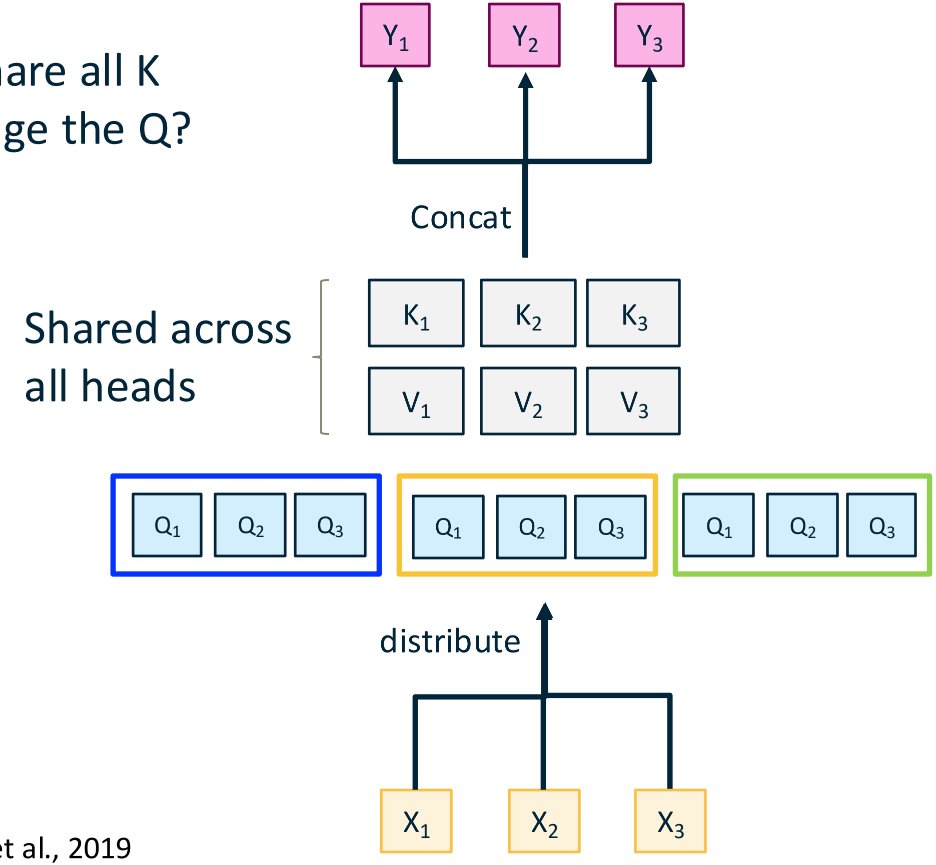
Optimizing KV Cache: Multi-Query Attention (MQA)

Multi-Query Attention (MQA): What if we share all K and V vectors across all heads and only change the Q?

Diversity comes from different Q.

1/32 KV cache size for 32-head MHA.
Not caching Q anyways

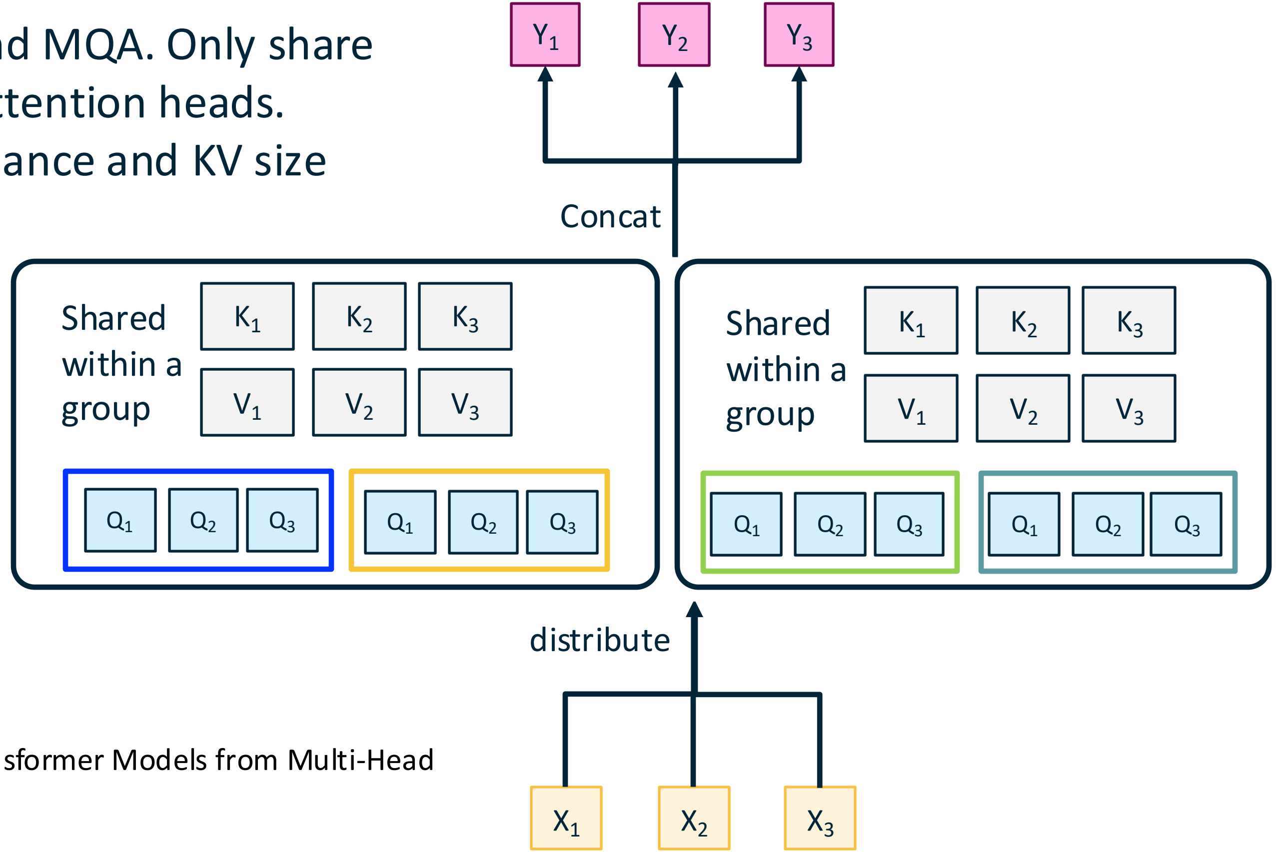
Slightly degraded performance (1%-2%), but massive save on KV cache!



Fast Transformer Decoding: One Write-Head is All You Need, Shazeer et al., 2019

Optimizing KV Cache: Group-Query Attention (GQA)

GQA: In-between MHA and MQA. Only share KV between a subset of attention heads. Balance between performance and KV size



GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints, Ainslie et al., 2023

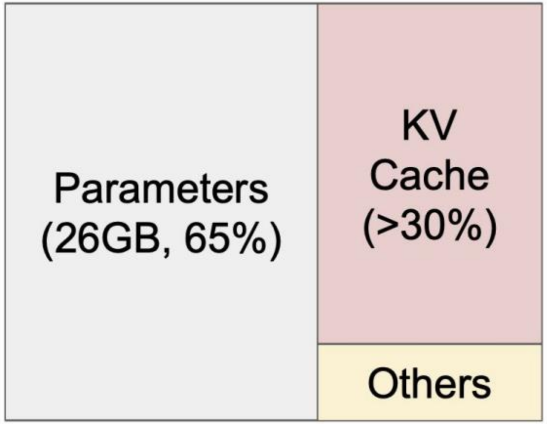
Can we do even better?

Need to go deeper than PyTorch/Python level …

Serving Transformers / LLMs on Modern GPUs

KV cache size in GPU memory determines:

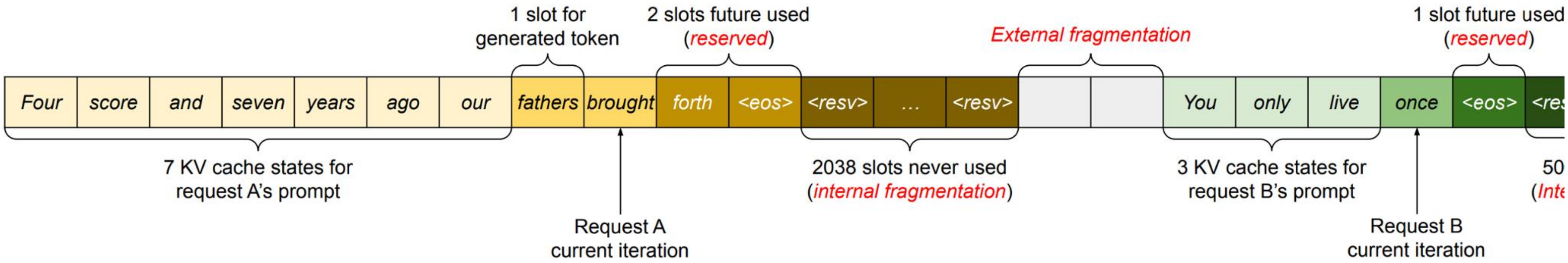
- How long of a sequence a system could generate
- How many requests a system could respond to in a batch (parallel inference instead of sequential)



NVIDIA A100 40GB

Efficient Memory Management for Large Language Model Serving with PagedAttention, Kwon et al., 2023

Naïve KV Caching is Wasteful



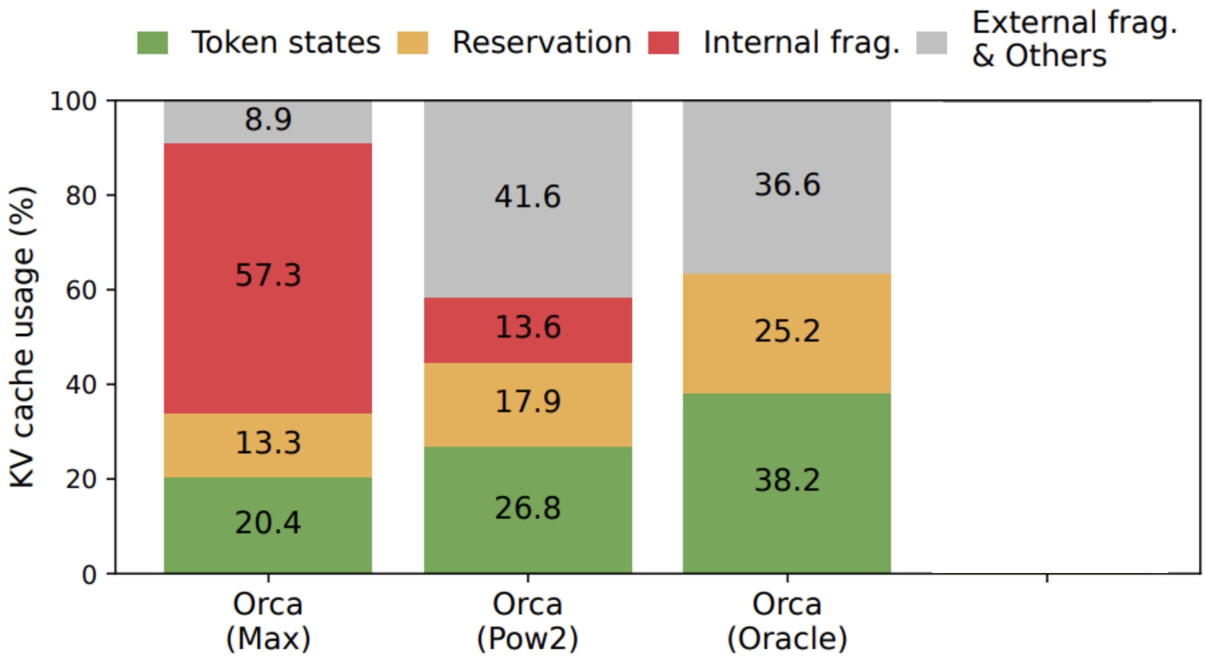
Internal fragmentation: memory reserved for a request but left unused because the final output length is shorter than the maximum length

Reservation: memory currently unused but may be use by the same request

External fragmentation: memory gaps between requested allocations (A and B) --- different requests can have different maximum length

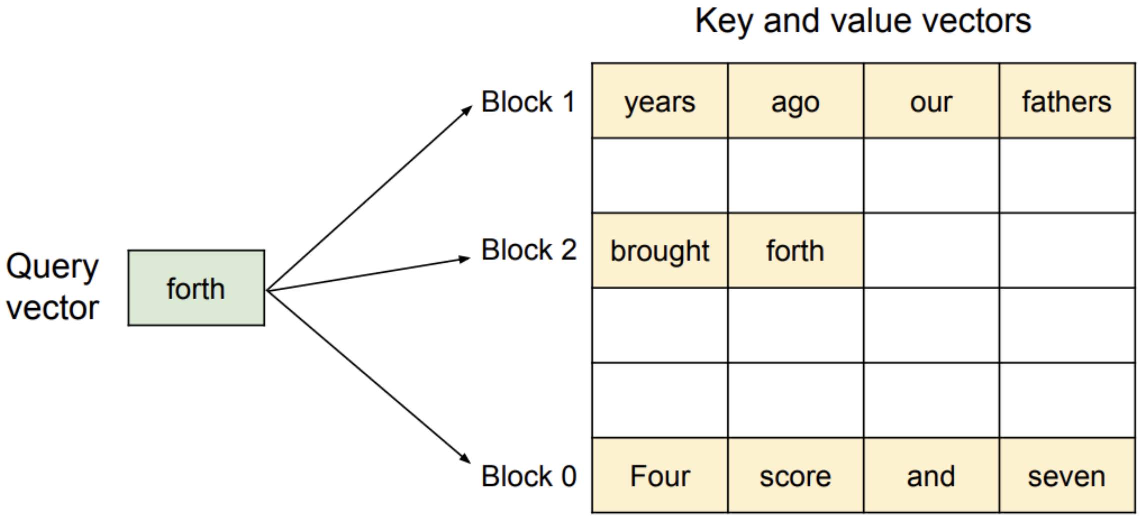
Efficient Memory Management for Large Language Model Serving with PagedAttention, Kwon et al., 2023

Naïve KV Caching is Wasteful



Efficient Memory Management for Large Language Model Serving with PagedAttention, Kwon et al., 2023

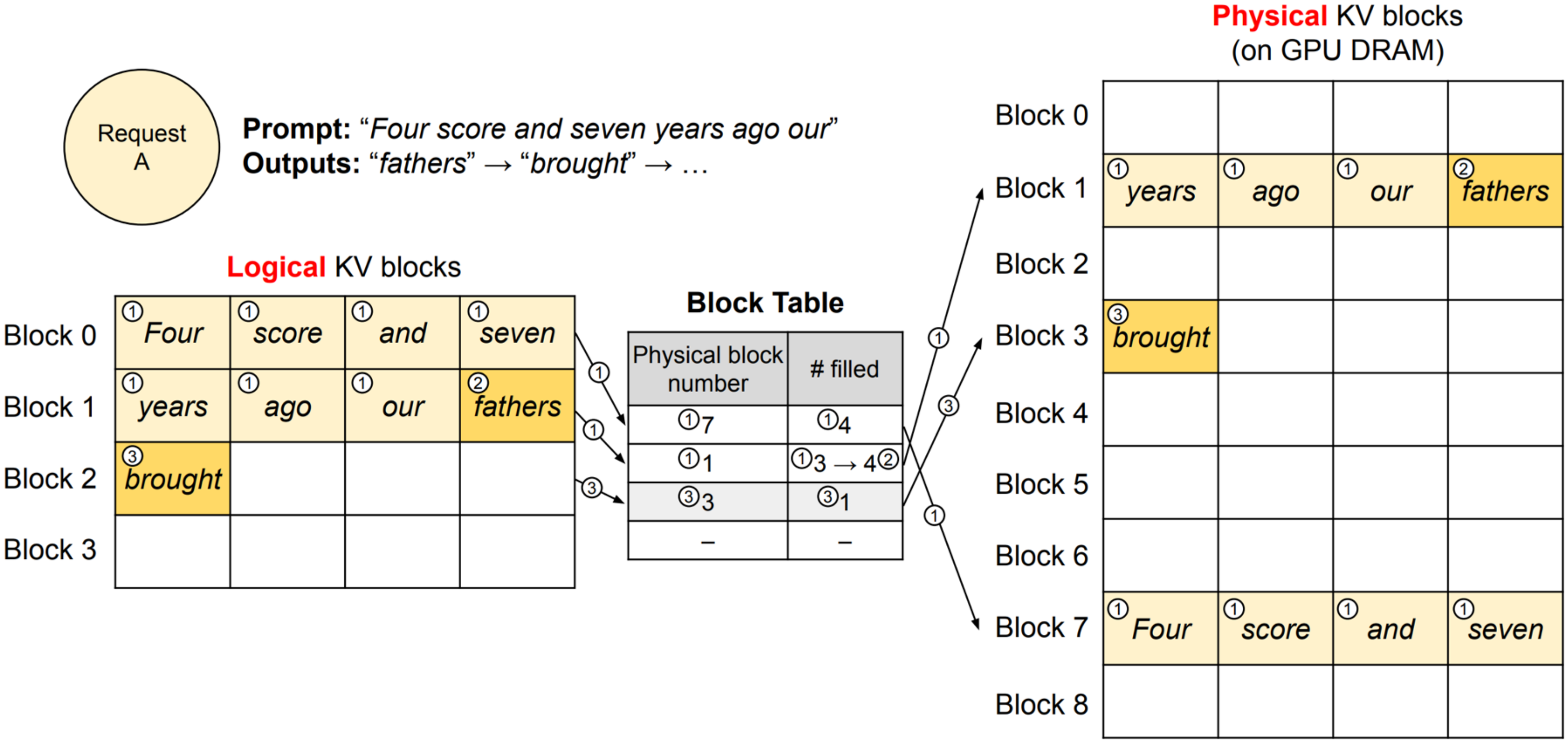
Idea: PagedAttention



- Store KV vectors in **non-contiguous, fixed-sized blocks** in the memory
- Compute attention across all relevant blocks

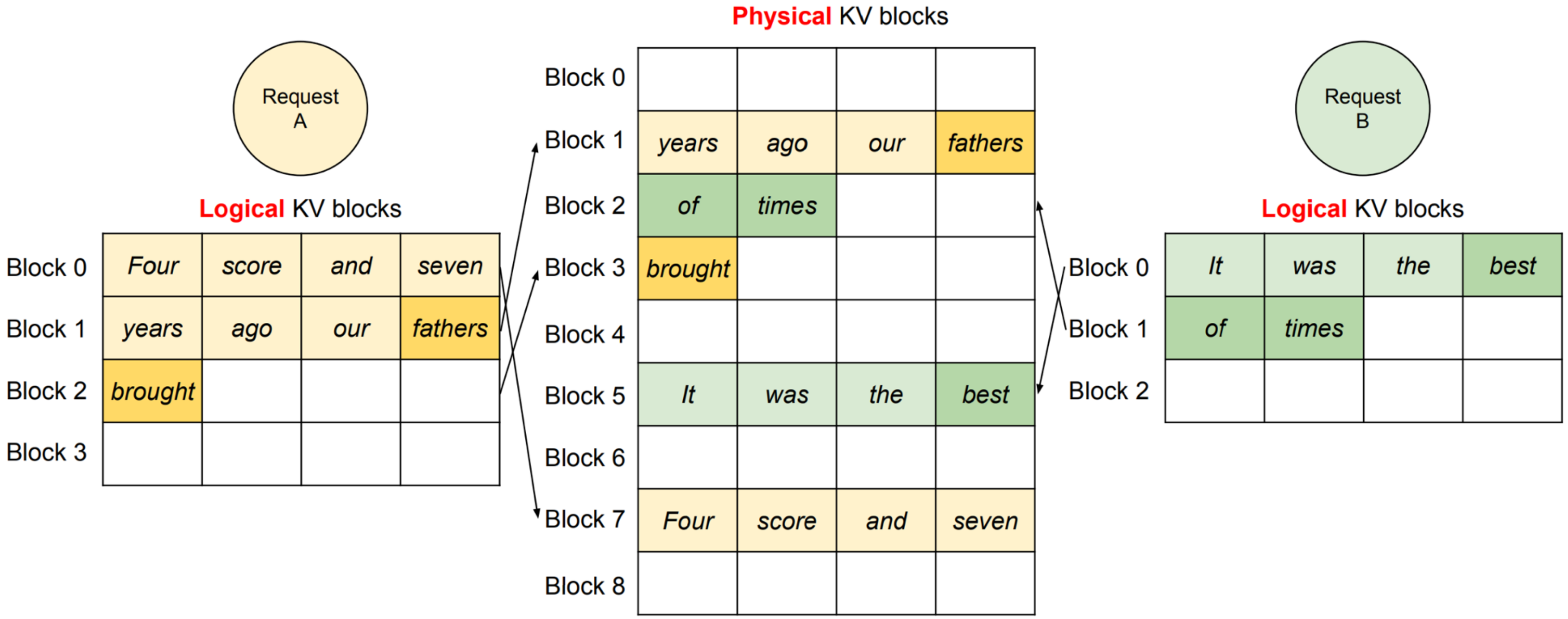
Efficient Memory Management for Large Language Model Serving with PagedAttention, Kwon et al., 2023

Logical vs. Physical KV Blocks



Efficient Memory Management for Large Language Model Serving with PagedAttention, Kwon et al., 2023

Serving Multiple Requests



Efficient Memory Management for Large Language Model Serving with PagedAttention, Kwon et al., 2023

System: vLLM

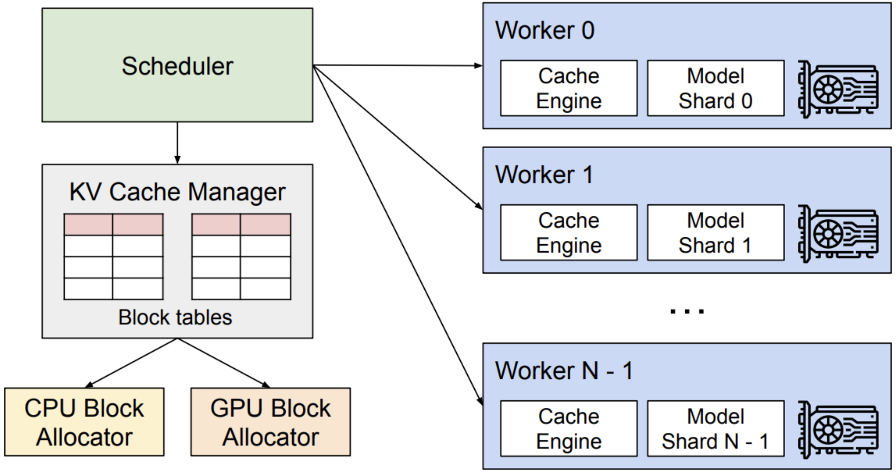
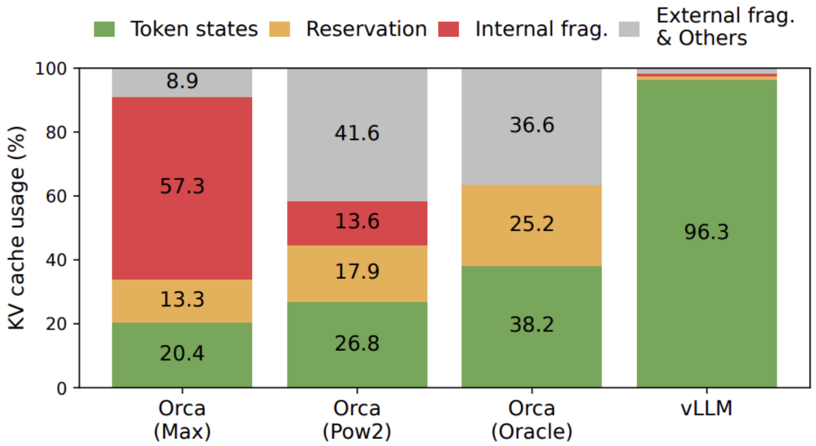


Figure 4. vLLM system overview.



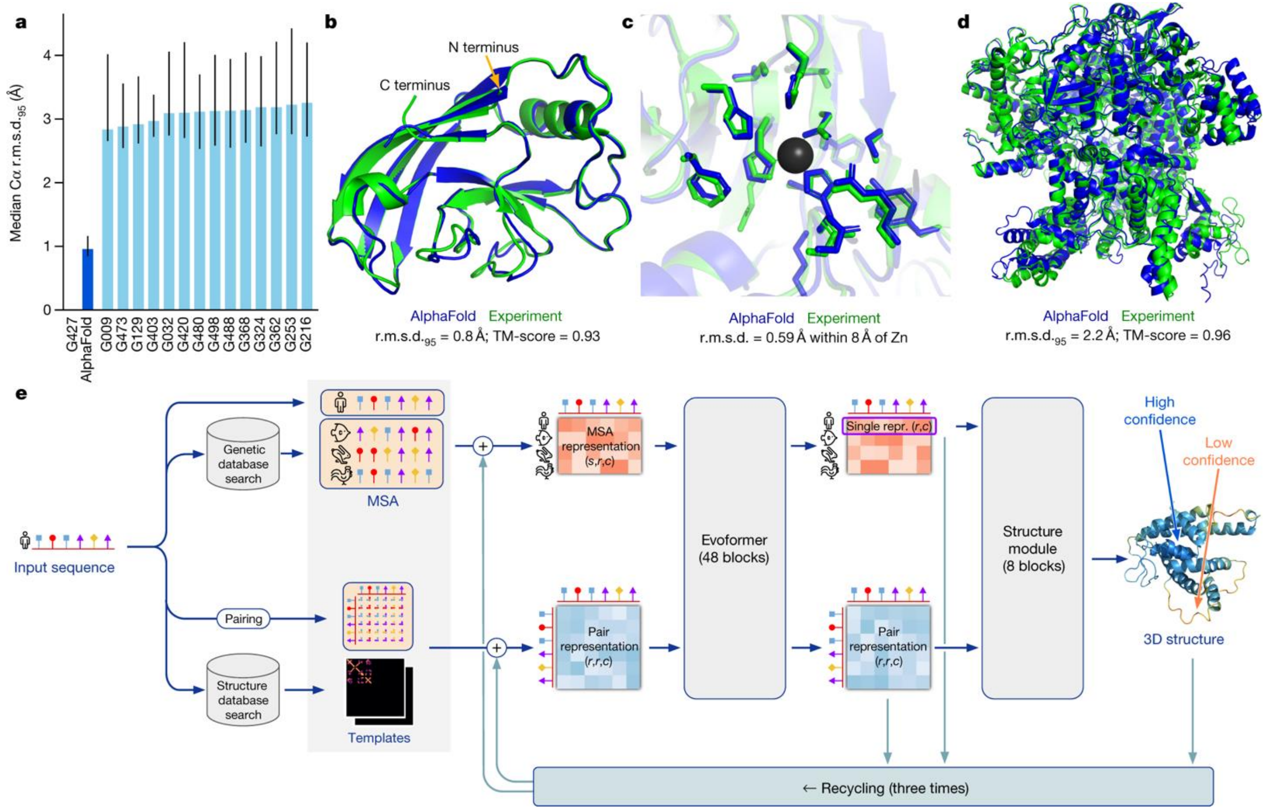
System: vLLM

vLLM is an autoregressive-style transformer serving system that:

Breaks KV cache into small fixed-size pages (like OS virtual memory) instead of large contiguous blocks per sequence
Allocates pages on-demand as sequences grow, eliminating wasted pre-allocated memory
Shares pages between sequences with common prefixes (same prompt, beam search branches, etc.)

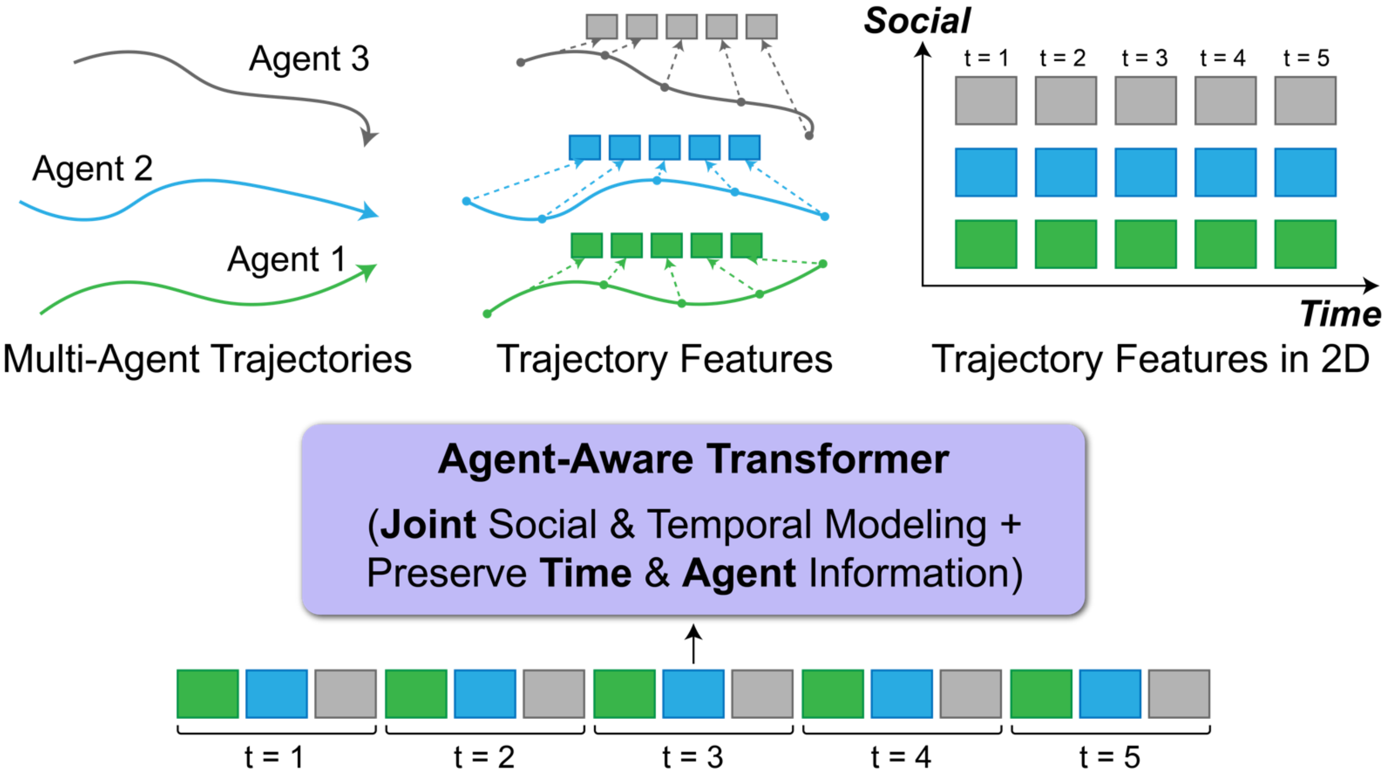
Can Attention/Transformers be used from more than text processing?

Encoding/Decoding Protein Structures (AlphaFold)



https://www.nature.com/articles/s41586-021-03819-2

Predicting Multi-agent Behaviors



Yuan et al., 2021 AgentFormer: Agent-Aware Transformers for Socio-Temporal Multi-Agent Forecasting

ViLBERT: A Visolinguistic Transformer

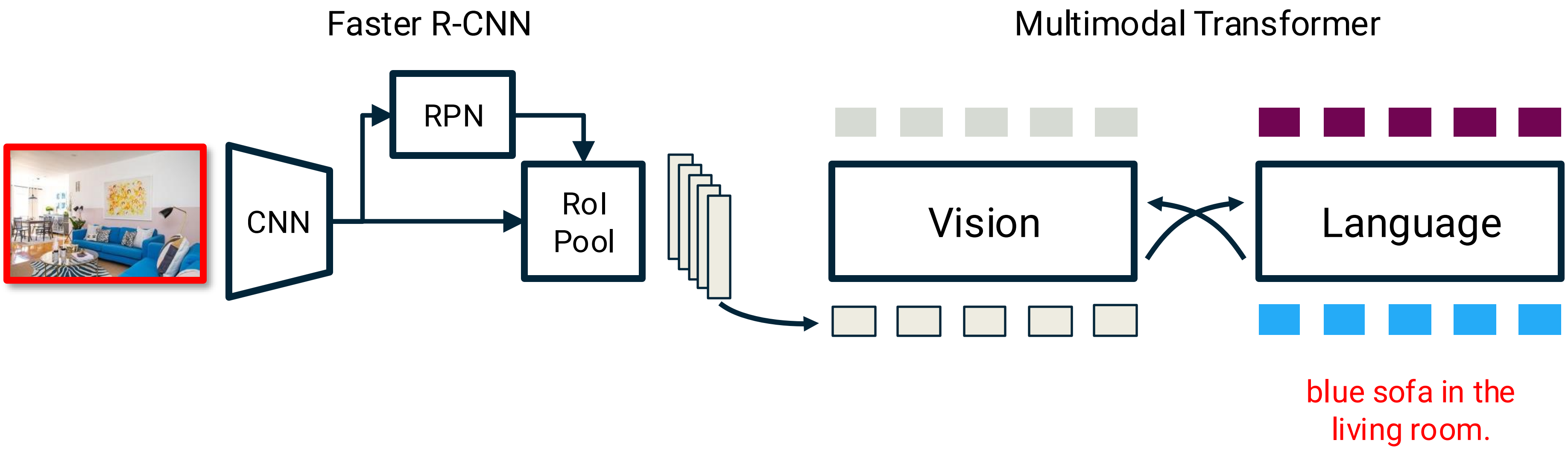
pop artist performs at the festival in a city.

a worker helps to clear the debris.

blue sofa in the living room.

Image and captions from: Sharma, Piyush, et al. "Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning." ACL. 2018.

ViLBERT: A Visolinguistic Transformer



Lu et al "Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks." *NeurIPS*. 2019.
Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks." *NeurIPS*. 2015.

What about for just image inputs? Without Convolution?

Preprint. Under review.

AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE

Alexey Dosovitskiy[*,†], Lucas Beyer[*], Alexander Kolesnikov[*], Dirk Weissenborn[*], Xiaohua Zhai[*], Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, Neil Houlsby[*,†]
[*]equal technical contribution, [†]equal advising
Google Research, Brain Team
{adosovitskiy, neilhoulsby}@google.com

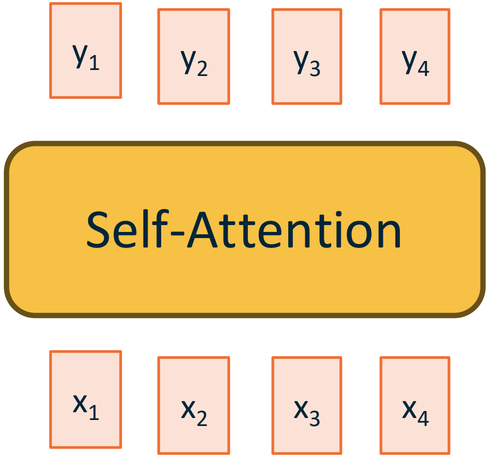
[cs.CV] 22 Oct 2020

ABSTRACT

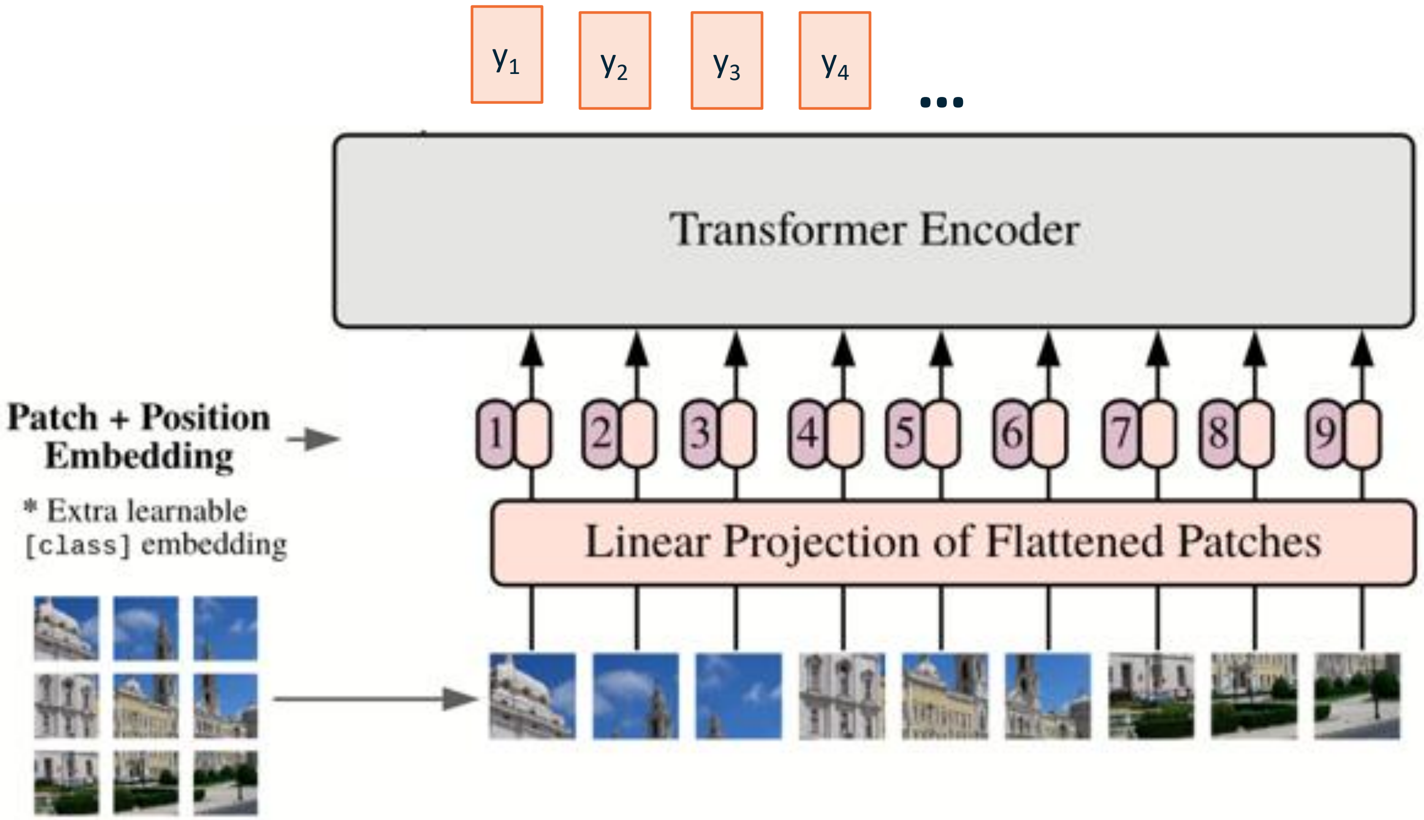
While the Transformer architecture has become the de-facto standard for natural language processing tasks, its applications to computer vision remain limited. In vision, attention is either applied in conjunction with convolutional networks, or used to replace certain components of convolutional networks while keeping their overall structure in place. We show that this reliance on CNNs is not necessary and a pure transformer applied directly to sequences of image patches can perform very well on image classification tasks. When pre-trained on large amounts of data and transferred to multiple mid-sized or small image recognition benchmarks (ImageNet, CIFAR-100, VTAB, etc.), Vision Transformer (ViT) attains excellent results compared to state-of-the-art convolutional networks while requiring substantially fewer computational resources to train.[1]

Slide progression inspired by Soheil Feizi

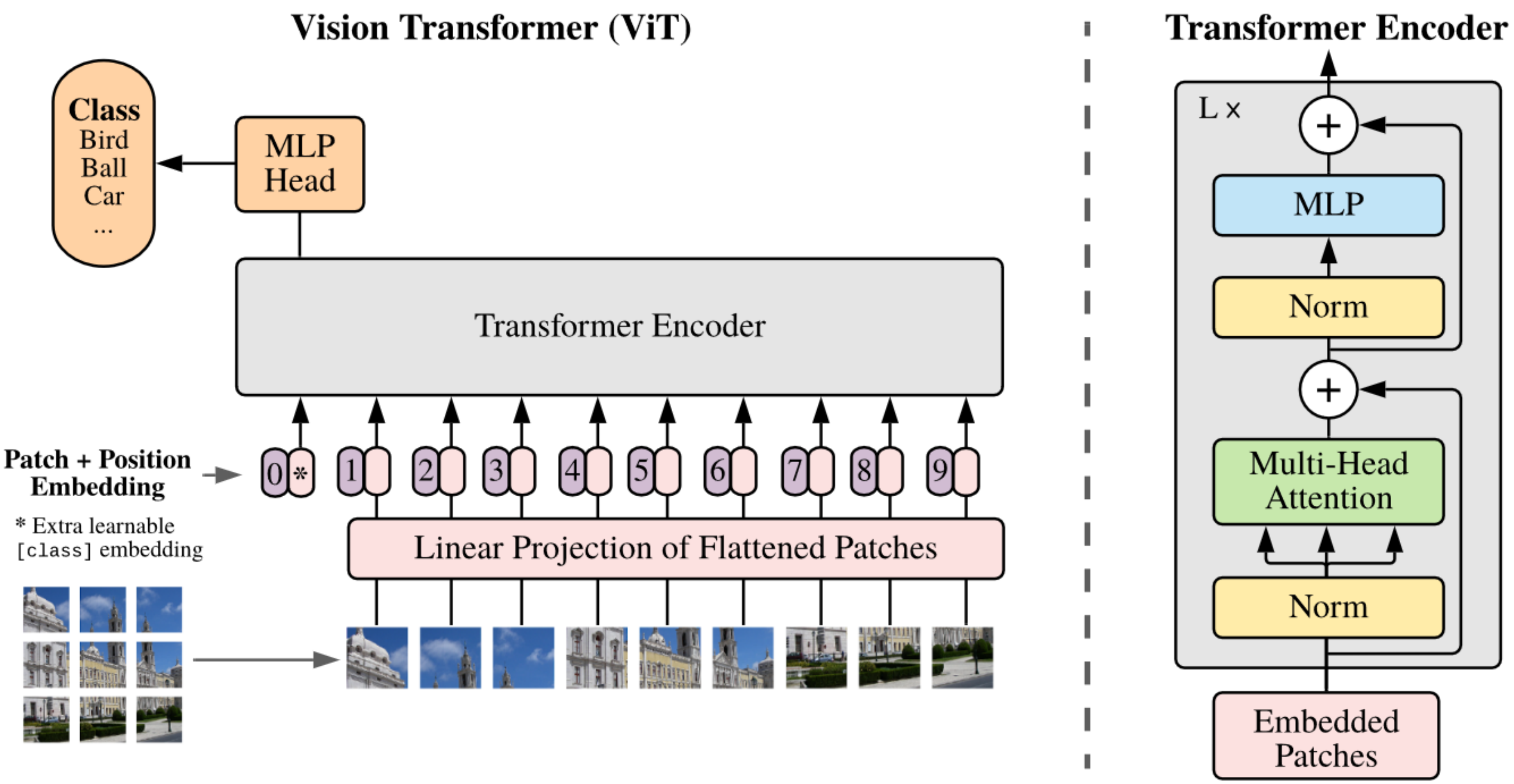
How should we "tokenize" images?



- Pixels? Too computationally intensive $O(n^2)$!
- Patches!



- **How do we do classification?**



Vision Transformer (ViT)

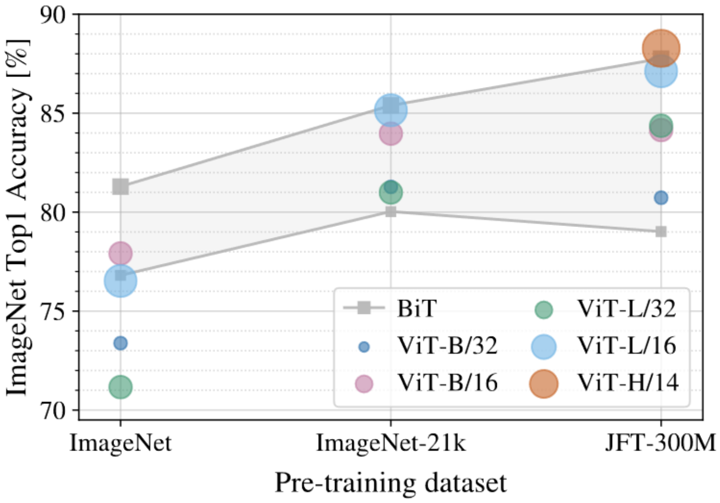


Figure 3: Transfer to ImageNet. While large ViT models perform worse than BiT ResNets (shaded area) when pre-trained on small datasets, they shine when pre-trained on larger datasets. Similarly, larger ViT variants overtake smaller ones as the dataset grows.

When trained on mid-sized datasets such as ImageNet, such models yield modest accuracies of a few percentage points below ResNets of comparable size.

Why?

Lacks some of the inductive biases:

- Spatial locality
- Translation equivariance

ViTs and Transfer Learning

https://paperswithcode.com/sota/image-classification-on-imagenet

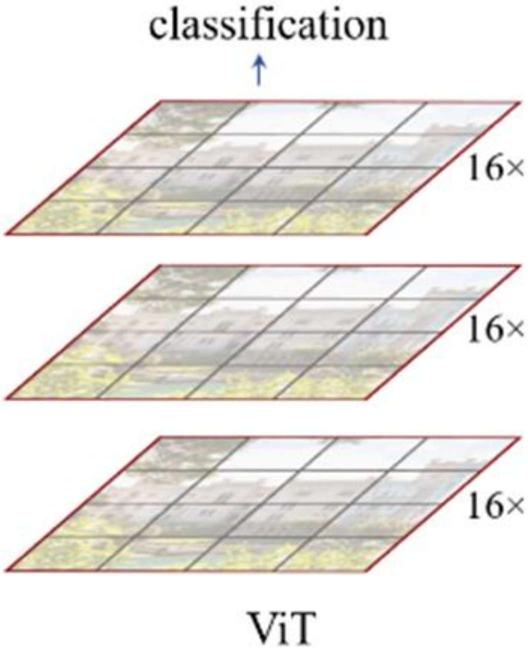
Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

Table 1: Details of Vision Transformer model variants.

However, the picture changes if the models are trained on larger datasets (14M-300M images). We find that large scale training trumps inductive bias.

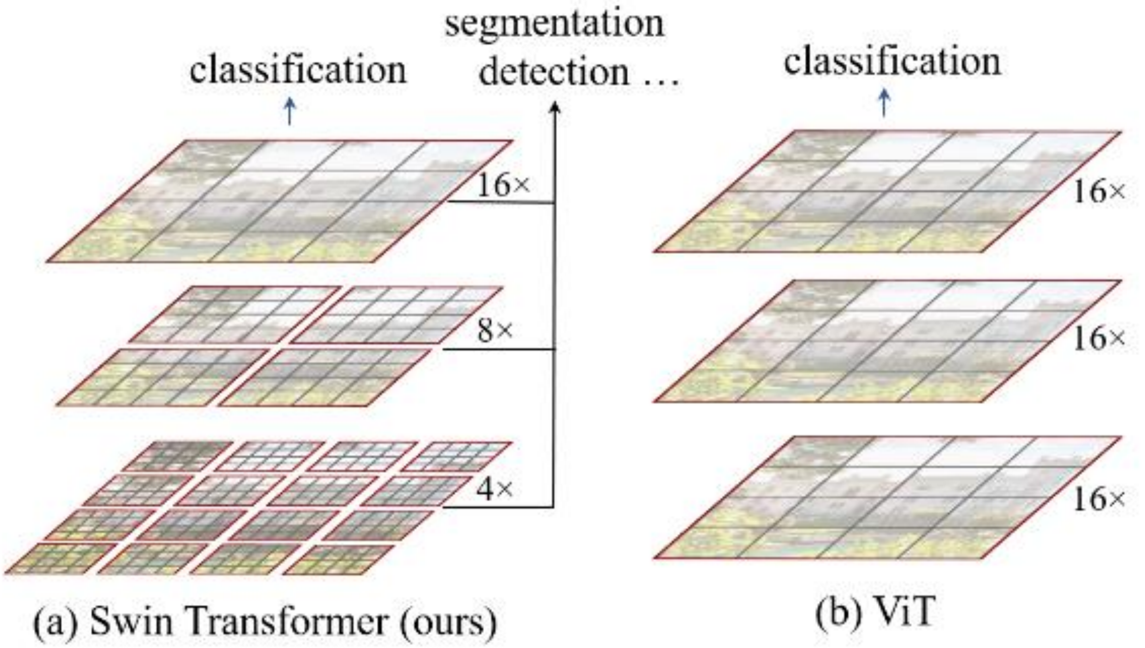
	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21K (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	**88.55** $\pm$ 0.04	87.76 $\pm$ 0.03	85.30 $\pm$ 0.02	87.54 $\pm$ 0.02	88.4/88.5*
ImageNet ReaL	**90.72** $\pm$ 0.05	90.54 $\pm$ 0.03	88.62 $\pm$ 0.05	90.54	90.55
CIFAR-10	**99.50** $\pm$ 0.06	99.42 $\pm$ 0.03	99.15 $\pm$ 0.03	99.37 $\pm$ 0.06	–
CIFAR-100	**94.55** $\pm$ 0.04	93.90 $\pm$ 0.05	93.25 $\pm$ 0.05	93.51 $\pm$ 0.08	–
Oxford-IIIT Pets	**97.56** $\pm$ 0.03	97.32 $\pm$ 0.11	94.67 $\pm$ 0.15	96.62 $\pm$ 0.23	–
Oxford Flowers-102	99.68 $\pm$ 0.02	**99.74** $\pm$ 0.00	99.61 $\pm$ 0.02	99.63 $\pm$ 0.03	–
VTAB (19 tasks)	**77.63** $\pm$ 0.23	76.28 $\pm$ 0.46	72.72 $\pm$ 0.21	76.29 $\pm$ 1.70	–
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k

ViT Results



What is wrong with this?

O($(HW)^2$) attention complexity!



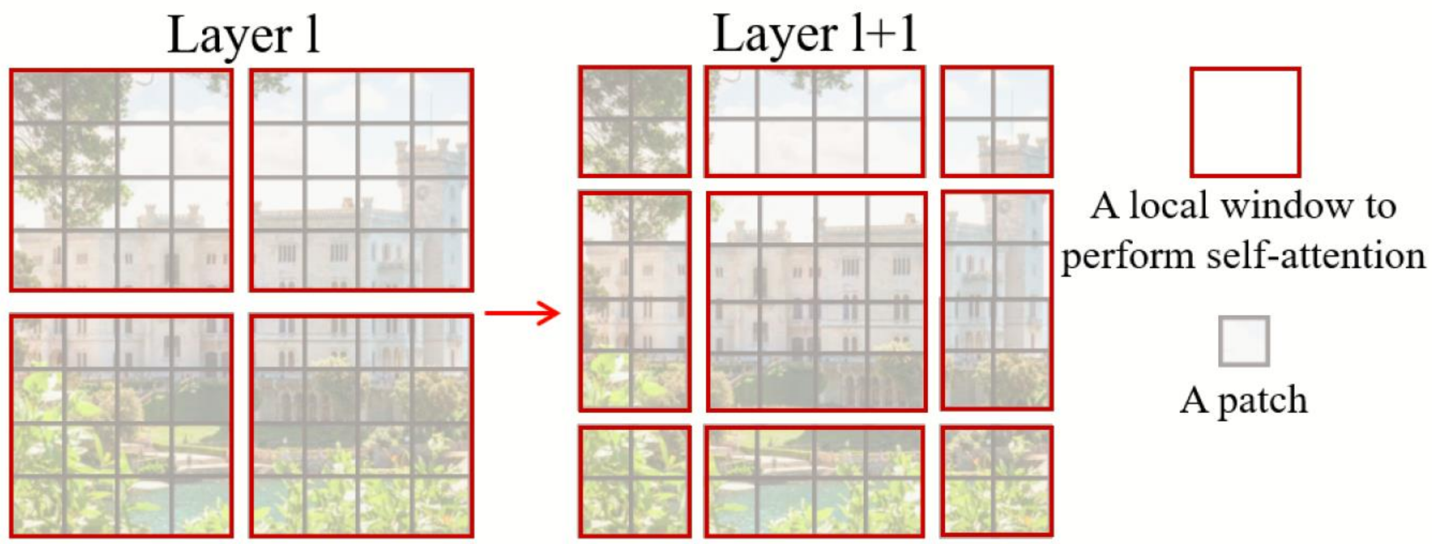
Swin Transformers key ideas:

- Feature pyramid!
- Self-attention only among patches within non-overlapping windows
- Shift windows across patches throughout the model (**S**hifted **WIN**dow)

Swin Transformer: Hierarchical Vision Transformer using Shifted Windows
Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, Baining Guo

Swin Transformers

https://paperswithcode.com/sota/instance-segmentation-on-coco

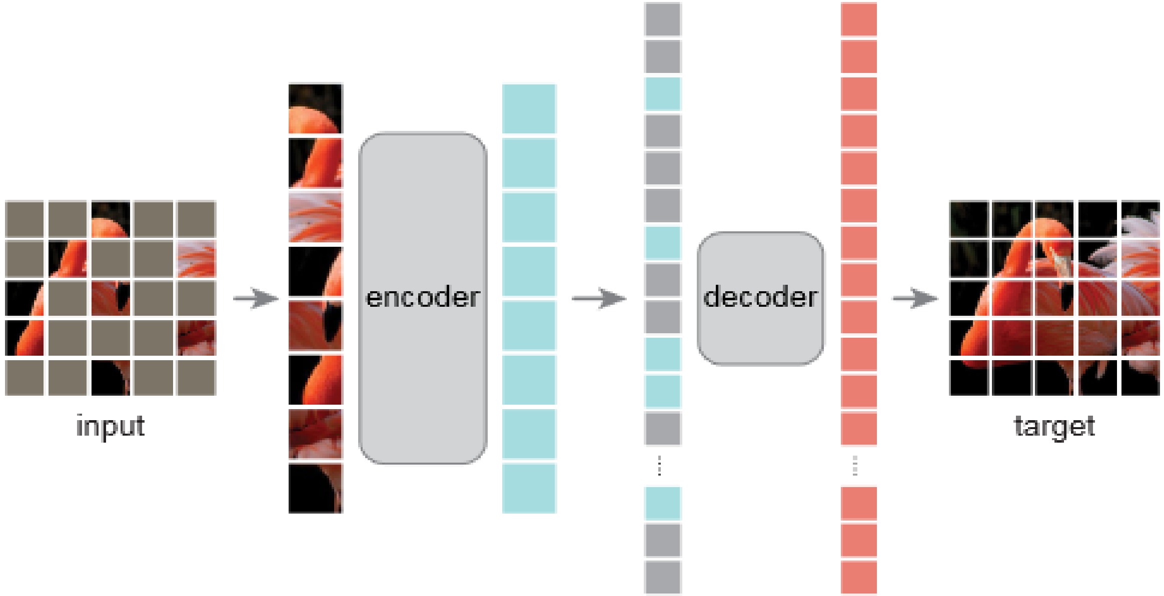


$O(HW)$ attention complexity
(number of windows is constant)

Swin Transformer: Hierarchical Vision Transformer using Shifted Windows
Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, Baining Guo

Shifted Window Attention

https://paperswithcode.com/sota/instance-segmentation-on-coco

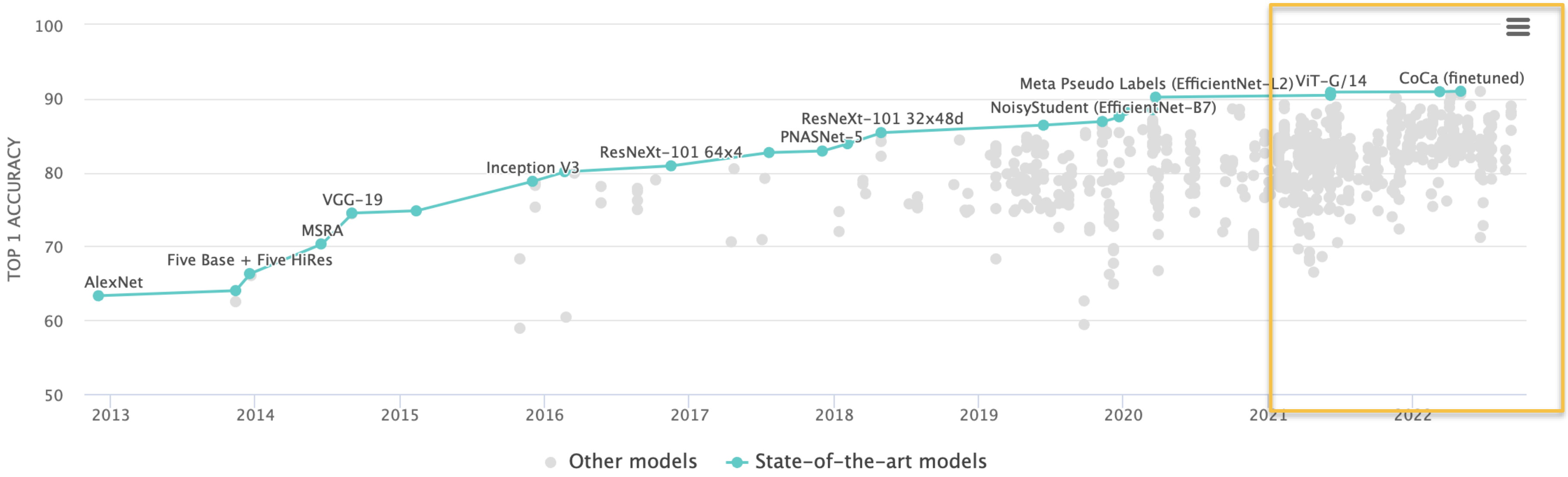


How can we learn unsupervised representations?

He et al., Masked Autoencoders Are Scalable Vision Learners

Masked Autoencoders

ViT: Vision Transformer



Generally more expensive to train and execute than ConvNets-based models

DeepMind

19 July 2022

Formal Algorithms for Transformers

Mary Phuong[1] **and Marcus Hutter**[1]
[1]DeepMind

This document aims to be a self-contained, mathematically precise overview of transformer architectures and algorithms (*not* results). It covers what transformers are, how they are trained, what they are used for, their key architectural components, and a preview of the most prominent models. The reader is assumed to be familiar with basic ML terminology and simpler neural network architectures such as MLPs.

Keywords: formal algorithms, pseudocode, transformers, attention, encoder, decoder, BERT, GPT, Gopher, tokenization, training, inference.

Contents

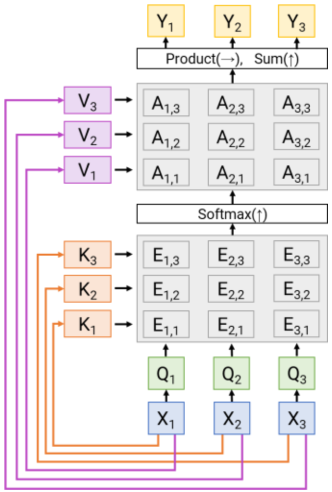
A famous colleague once sent an actually very well-written paper he was quite proud of to a famous complexity theorist. His answer: "I can't find

plete, precise and compact overview of transformer architectures and formal algorithms (but *not* results). It covers what Transformers are (Section 6), how they are trained (Section 7), what they're used for (Section 3), their key architectural components (Section 5), tokenization (Section 4), and a preview of practical considerations (Section 8) and the most prominent models.

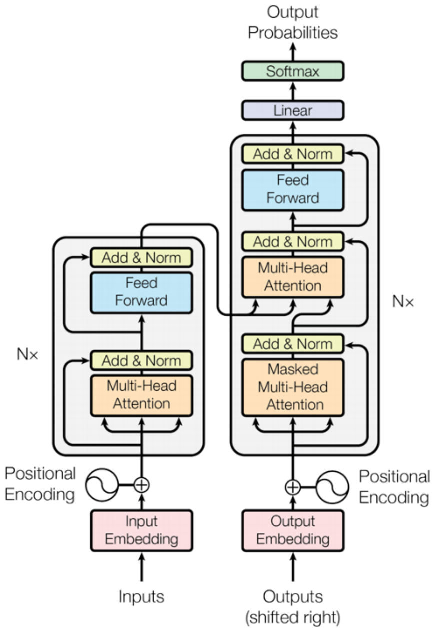
The essentially complete pseudocode is about 50 lines, compared to thousands of lines of actual real source code. We believe these formal algorithms will be useful for theoreticians who require compact, complete, and precise formulations, experimental researchers interested in

Summary

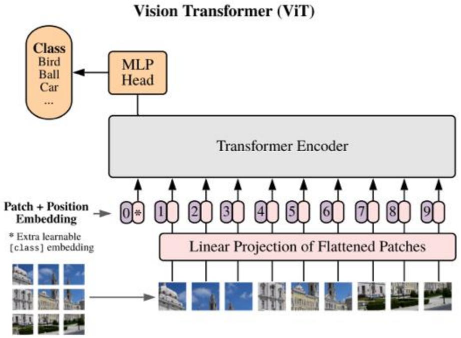
Self-Attention



Transformer Model



Beyond Language



Next Lecture (Virtual):

How to Train Your Large Language Models

Siddharth Karamcheti
(Incoming GT Prof.)