Topics:

- Attention and Transformers

CS 4644-DL / 7643-A
ZSOLT KIRA

- **Assignment 2 extended**
    - Due **June 25th** (grace period June 27th)

- Meta office hours on Neural Machine Translation Friday 06/27 3pm ET

Lecture Outline

- Machine Translation with RNNs
- RNNs with Attention
- From Attention to Transformers
- What can Transformers do?

Slides from Justin Johnson, modified by Arjun Madjumdar

Sequence Modeling with RNNs

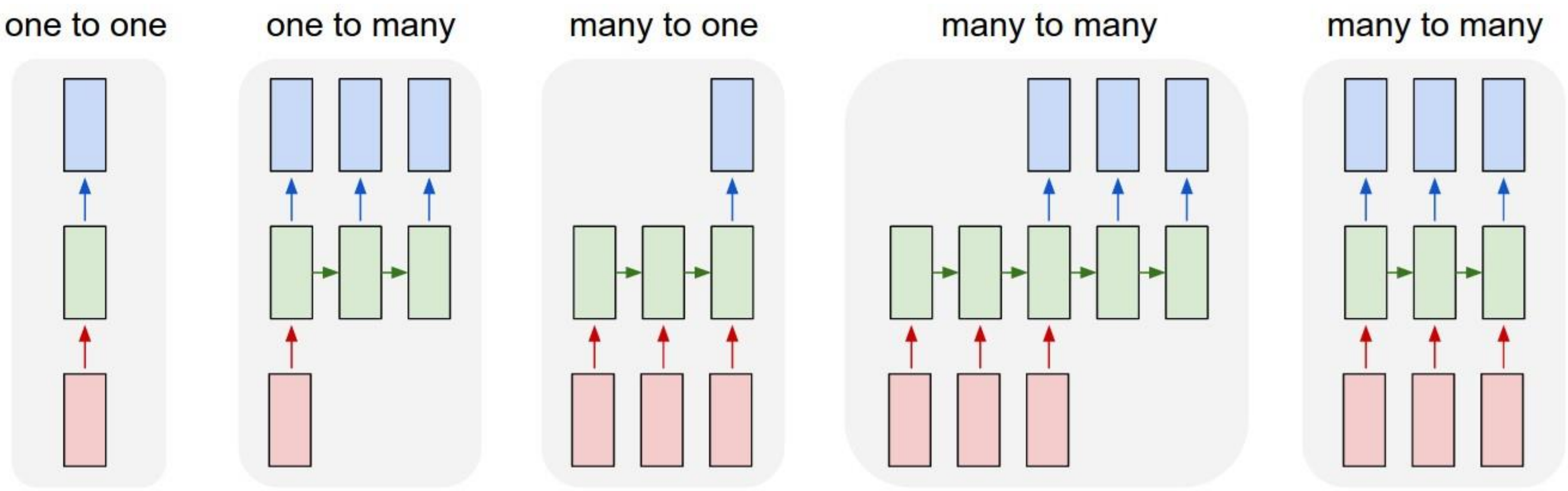
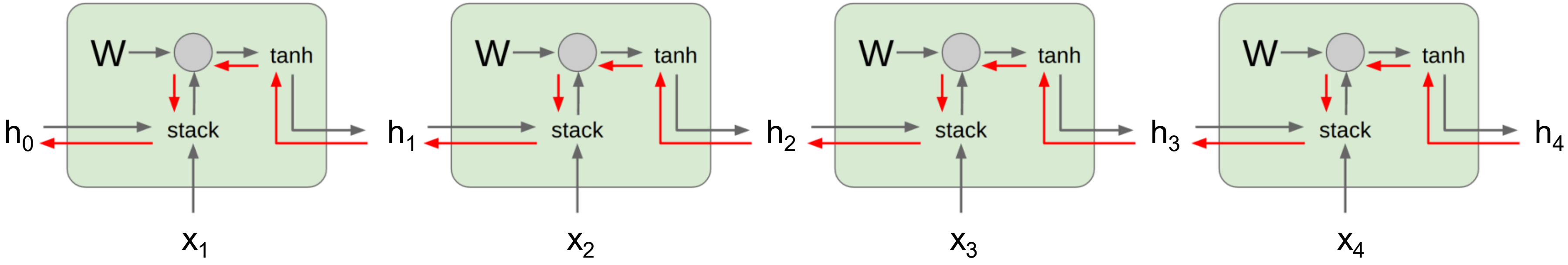


Image Credit: Andrej Karpathy

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



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left(\begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$
$$= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

How can we train this on language?

- Supervised Learning:
  - Sentiment analysis (sentence -> negative/neutral/positive) labeled by humans
  - Translation -> English and equivalent other language

- Self-supervised: Predict the next letter or word!
  - This is **extremely powerful!!**
  - In order to predict what's next, it needs to really understand not just language statistics but world knowledge!
    - Of course, we need scale for this level of loss reduction / understanding

- **Training:** A large corpus of text from the web
    - Note: No annotation required! It's just "the text"

- **Inference:** Just generate me new text
    - Can condition on some initial input (**prompt**)

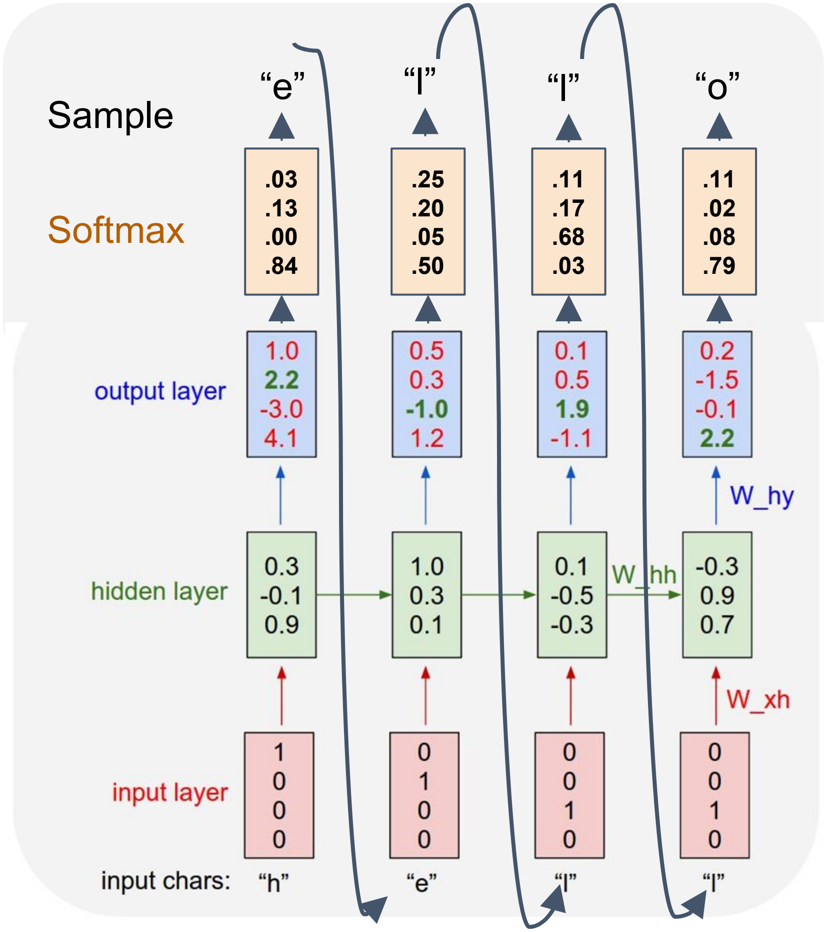
```
#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>

#define REG_PG    vesa_slot_addr_pack
#define PFM_NOCOMP  AFSR(0, load)
#define STACK_DDR(type)     (func)

#define SWAP_ALLOCATE(nr)     (e)
#define emulate_sigs()  arch_get_unaligned_child()
#define access_rw(TST)  asm volatile("movd %%esp, %0, %3" : : "r" (0));   \
  if (__type & DO_READ)

static void stat_PC_SEC __read_mostly offsetof(struct seq_argsqueue, \
          pC>[1]);

static void
os_prefix(unsigned long sys)
{
#ifdef CONFIG_PREEMPT
  PUT_PARAM_RAID(2, sel) = get_state_state();
  set_pid_sum((unsigned long)state, current_state_str(),
           (unsigned long)-1->lr_full; low;
}
```

Test Time: Sample / Argmax / Beam Search

Example: Character-level Language Model Sampling

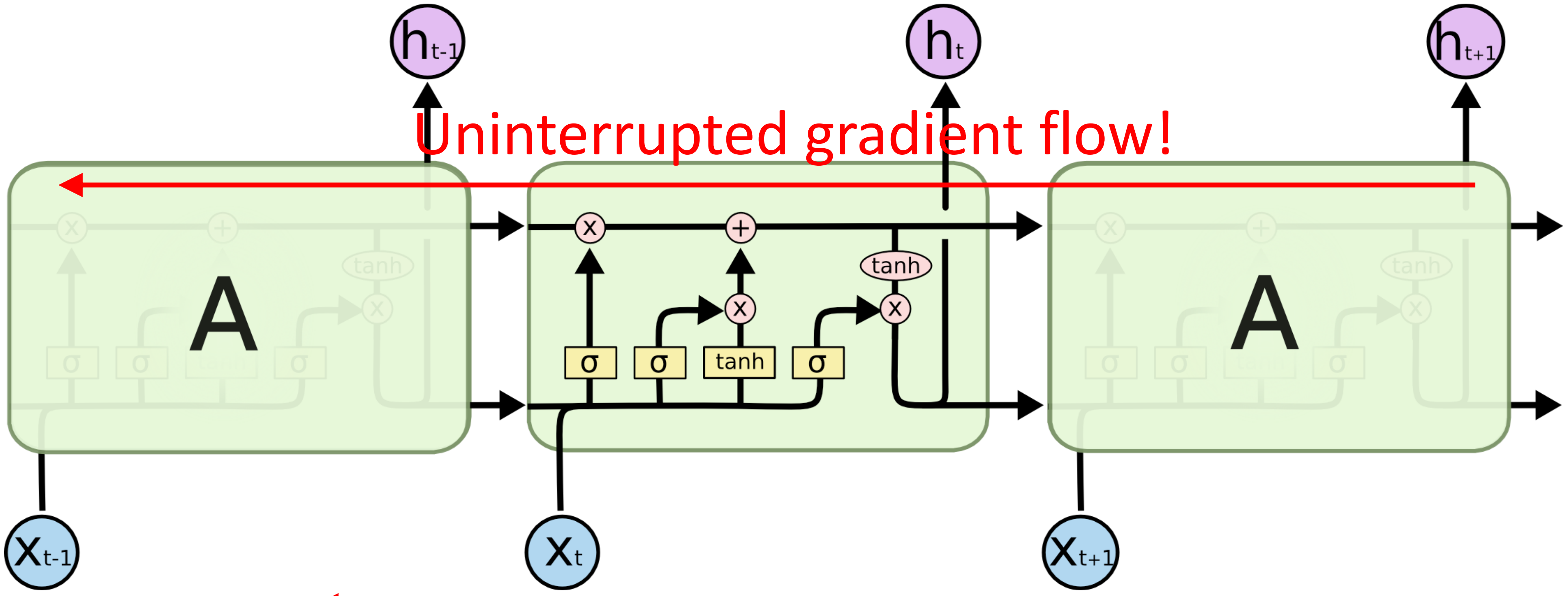
Vocabulary: [h,e,l,o]

At test-time sample characters one at a time, feed back to model



Can also feed in predictions during training (student forcing)

LSTMs Intuition: Additive Updates



Similar to ResNet!

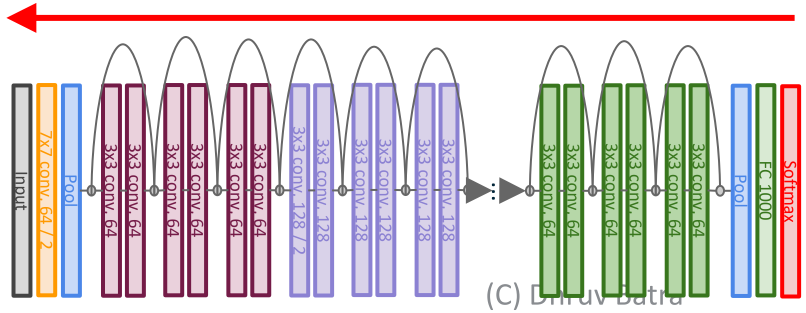
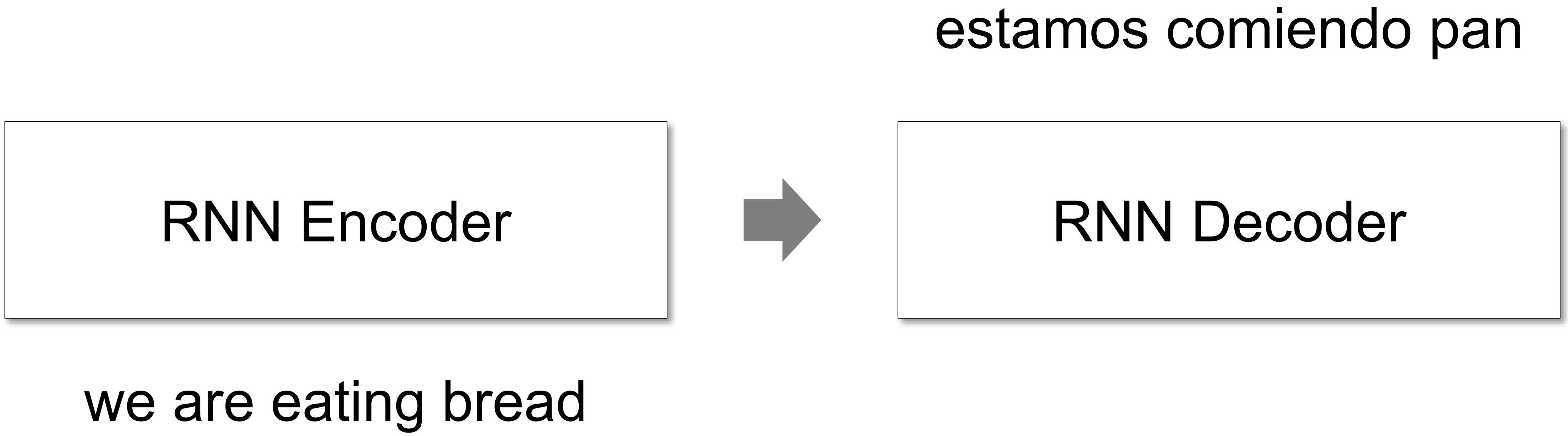


Image Credit: Christopher Olah (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Machine Translation

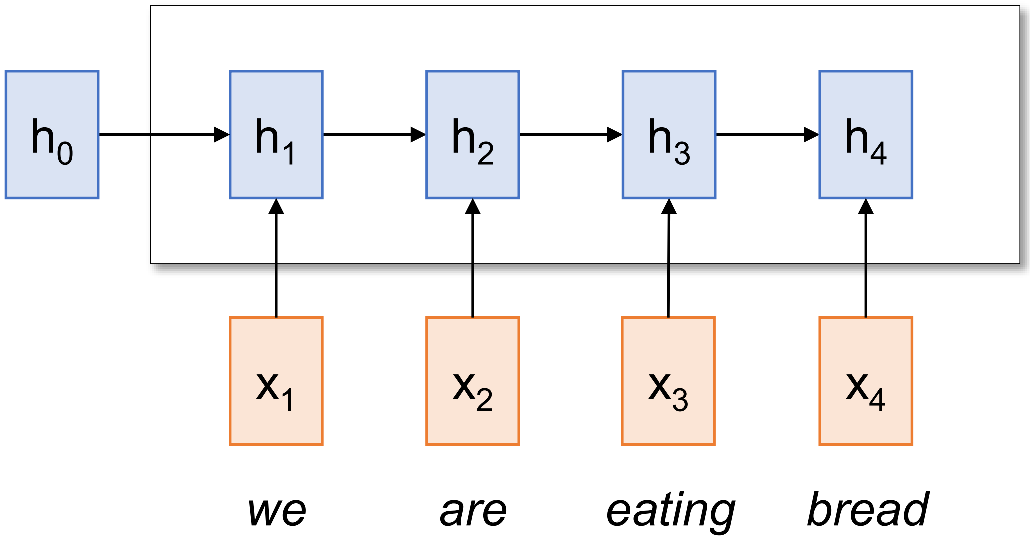
we are eating bread ➡ estamos comiendo pan

Machine Translation



Machine Translation with RNNs

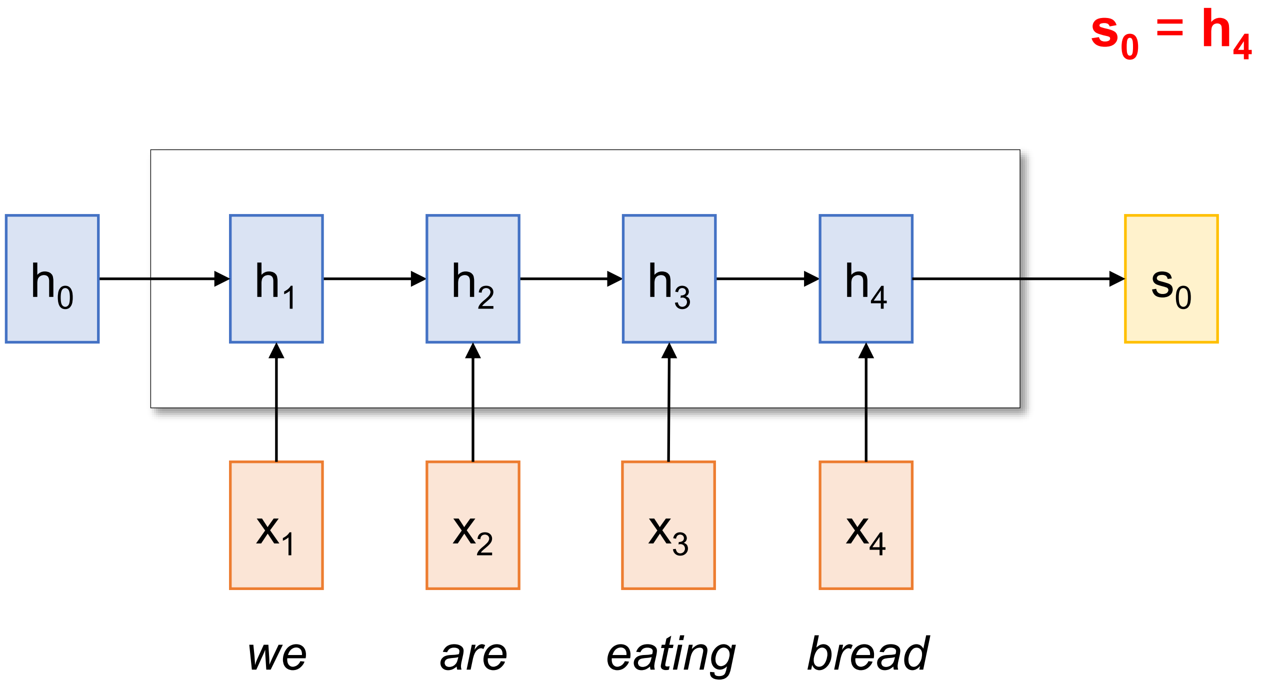
Encoder: $h_t = f_W(x_t, h_{t-1})$



Slide credit: Justin Johnson

Machine Translation with RNNs

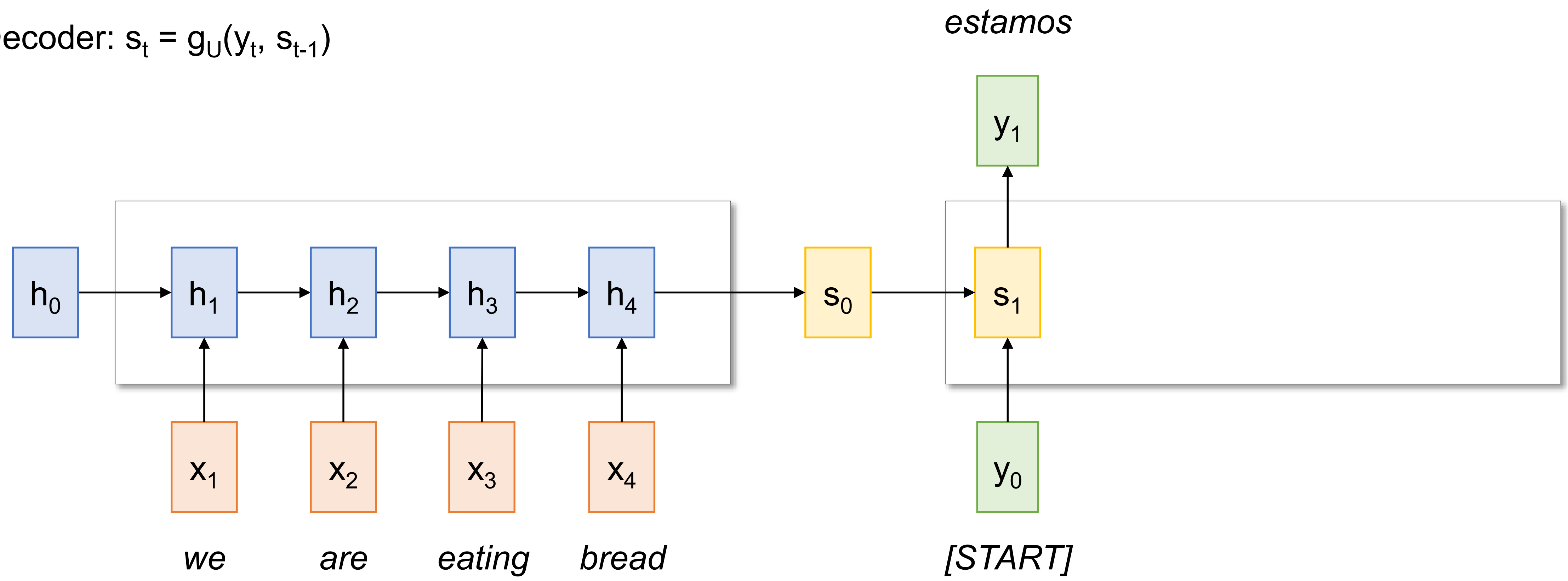
Encoder: $h_t = f_W(x_t, h_{t-1})$



Machine Translation with RNNs

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

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

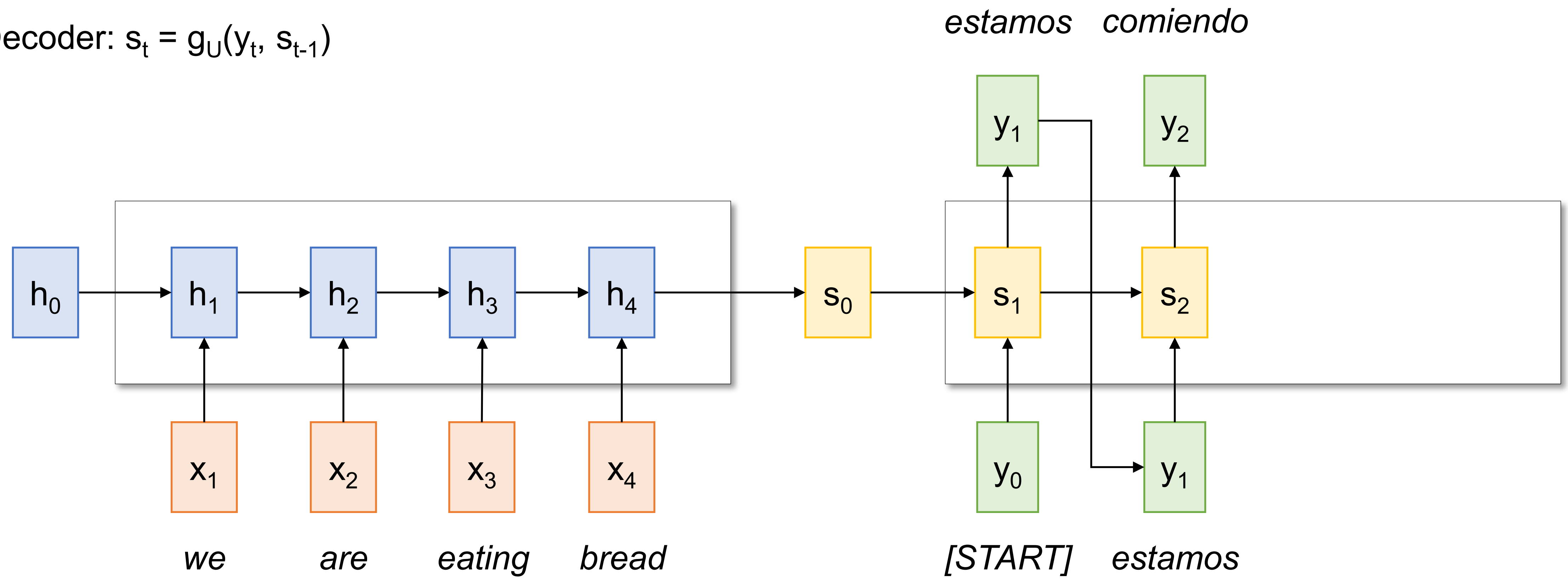


Slide credit: Justin Johnson

Machine Translation with RNNs

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

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



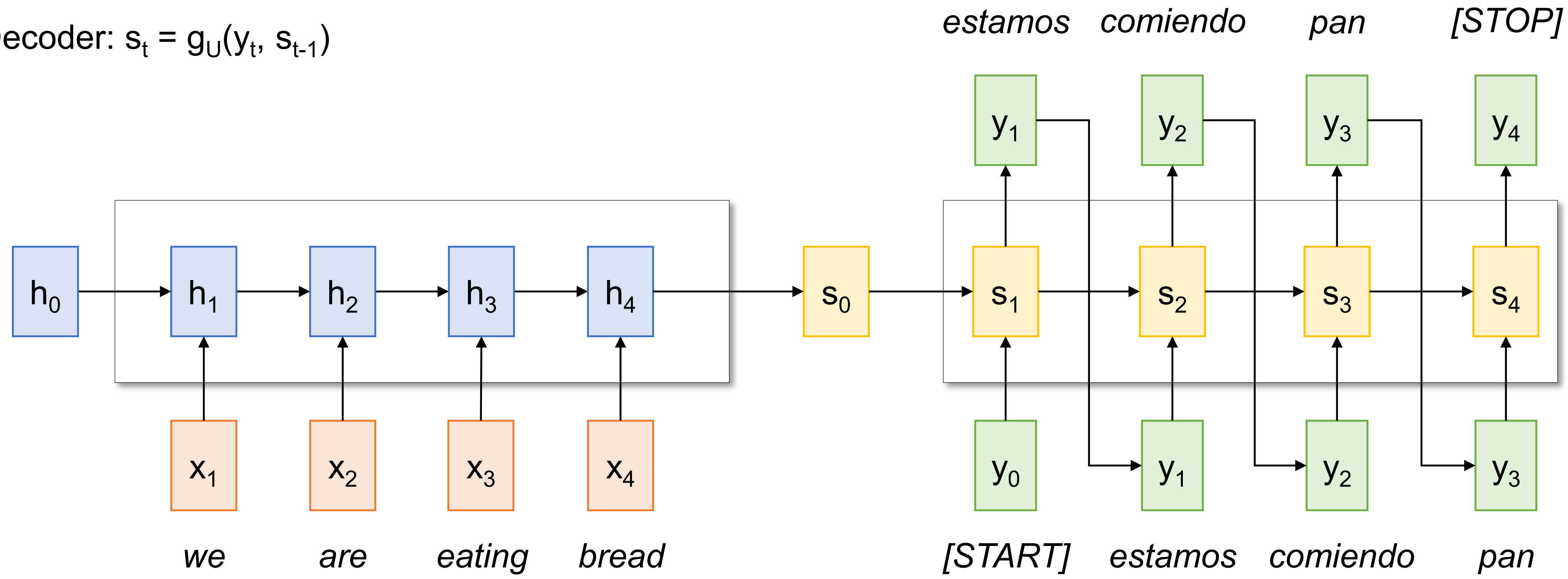
Slide credit: Justin Johnson

Machine Translation with RNNs

Note [START]/[STOP] words. This can be treated as representation for entire sentence

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

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



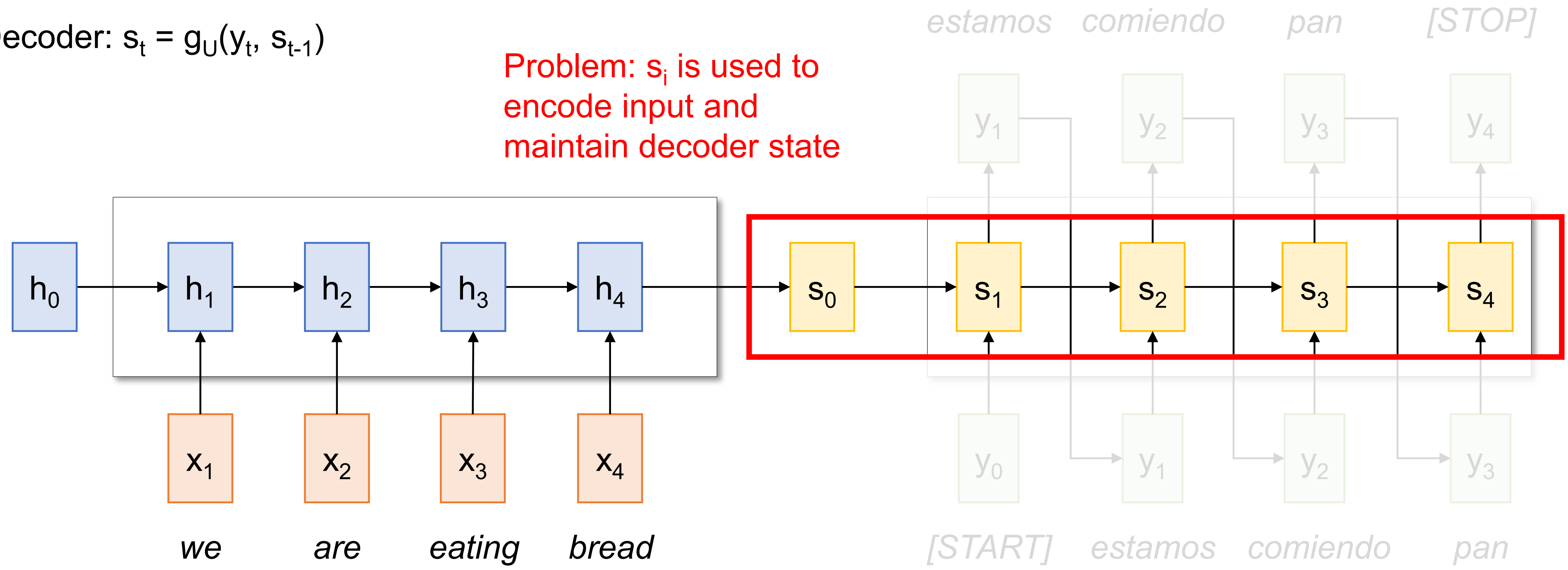
Slide credit: Justin Johnson

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 encode input and maintain decoder state



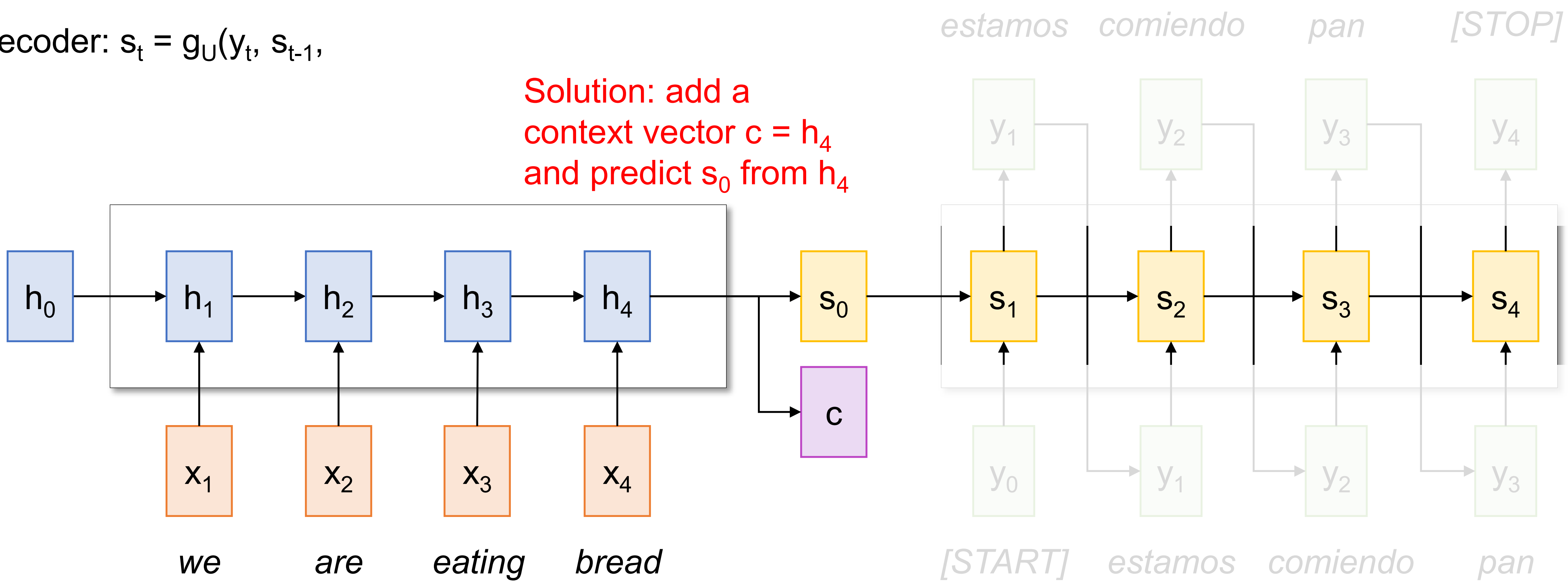
Slide credit: Justin Johnson

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 predict s_0 from h_4

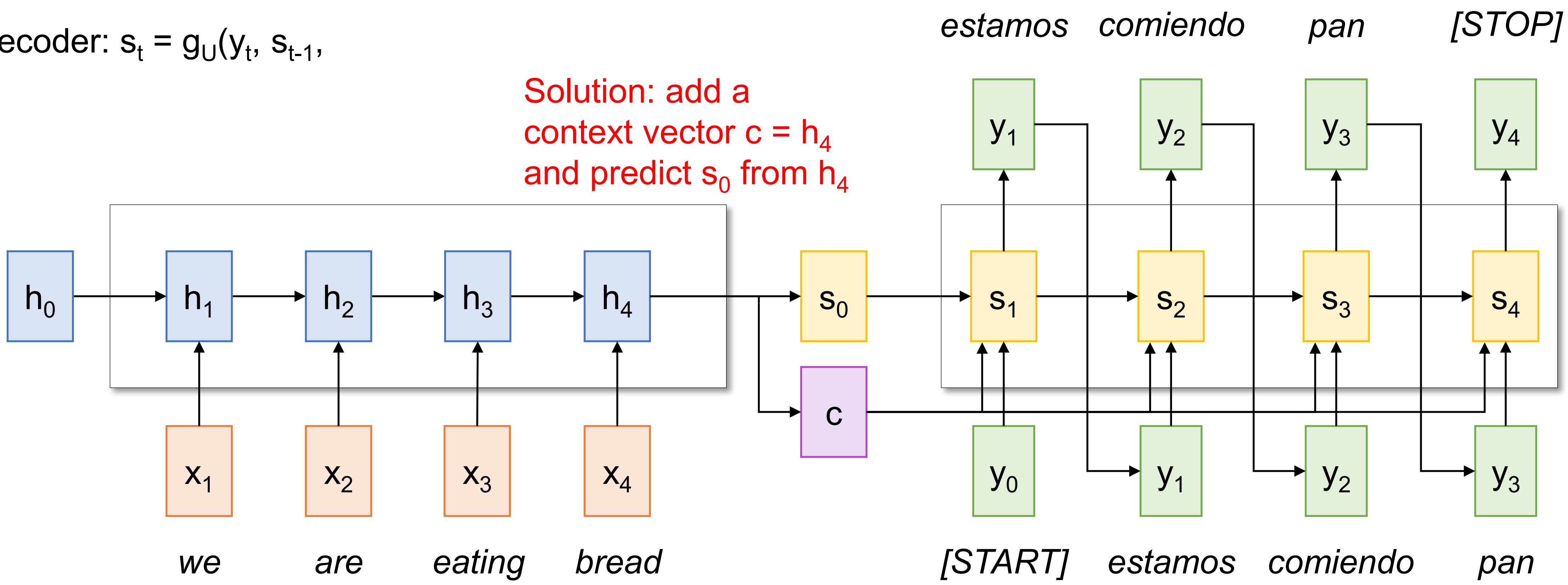


Slide credit: Justin Johnson

Machine Translation with RNNs

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

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

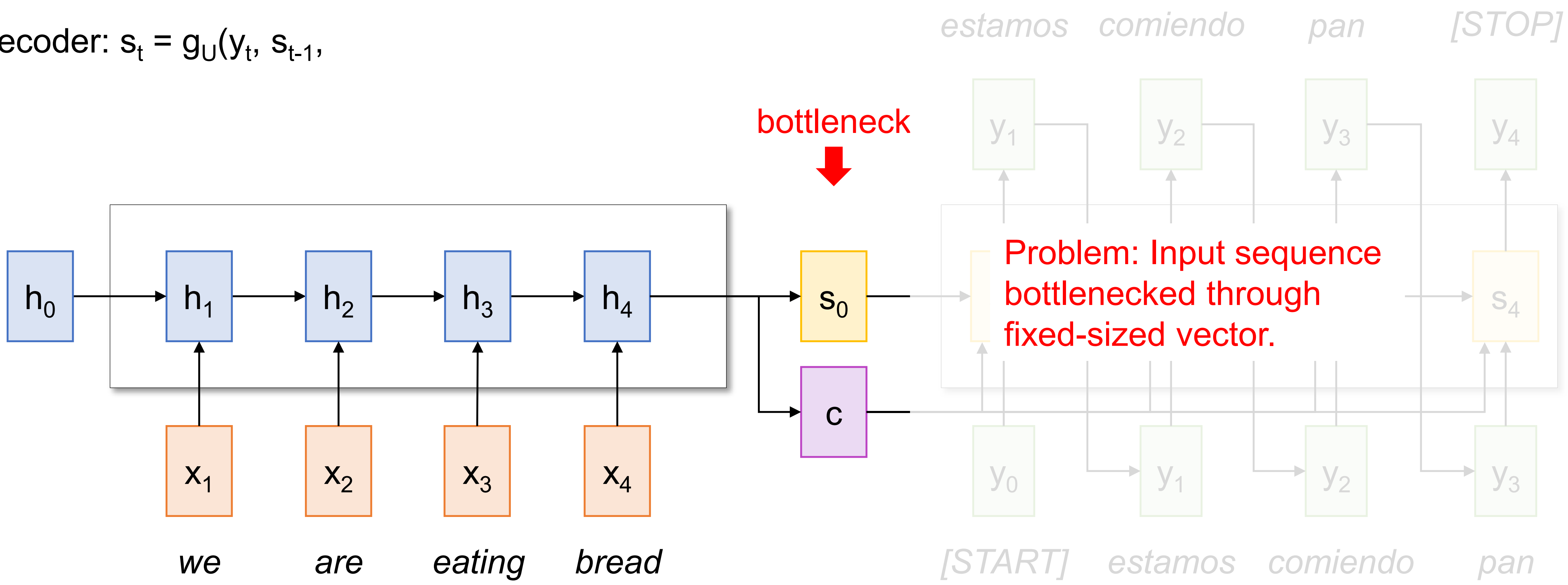


Slide credit: Justin Johnson

Machine Translation with RNNs

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

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

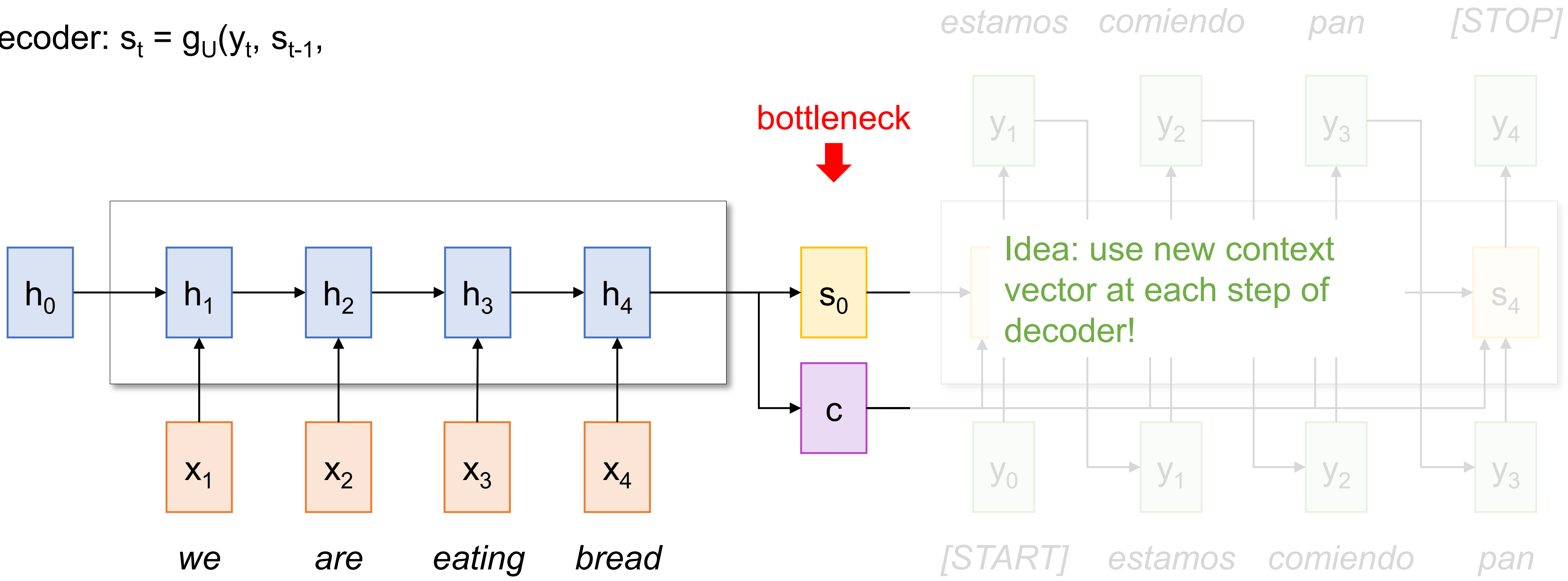


Slide credit: Justin Johnson

Machine Translation with RNNs

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

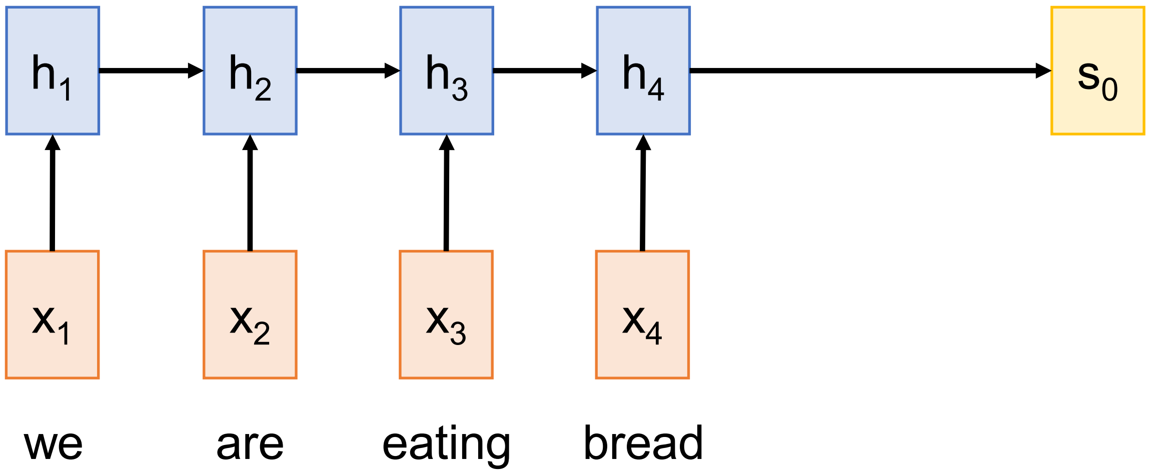
Decoder: $s_t = g_U(y_t, s_{t-1}, c)$



Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**

From final hidden state:
Initial decoder state s_0

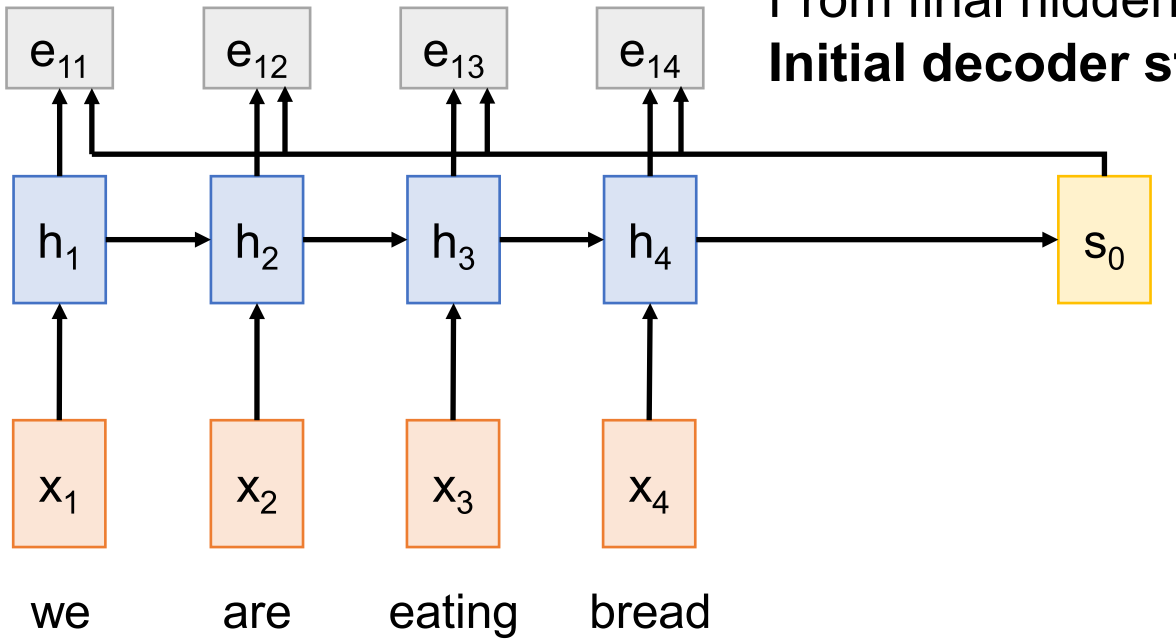


Bahdanau et al, “Neural machine translation by jointly learning to align and translate”, ICLR 2015

Machine Translation with RNNs **and Attention**

Compute **alignment scores**

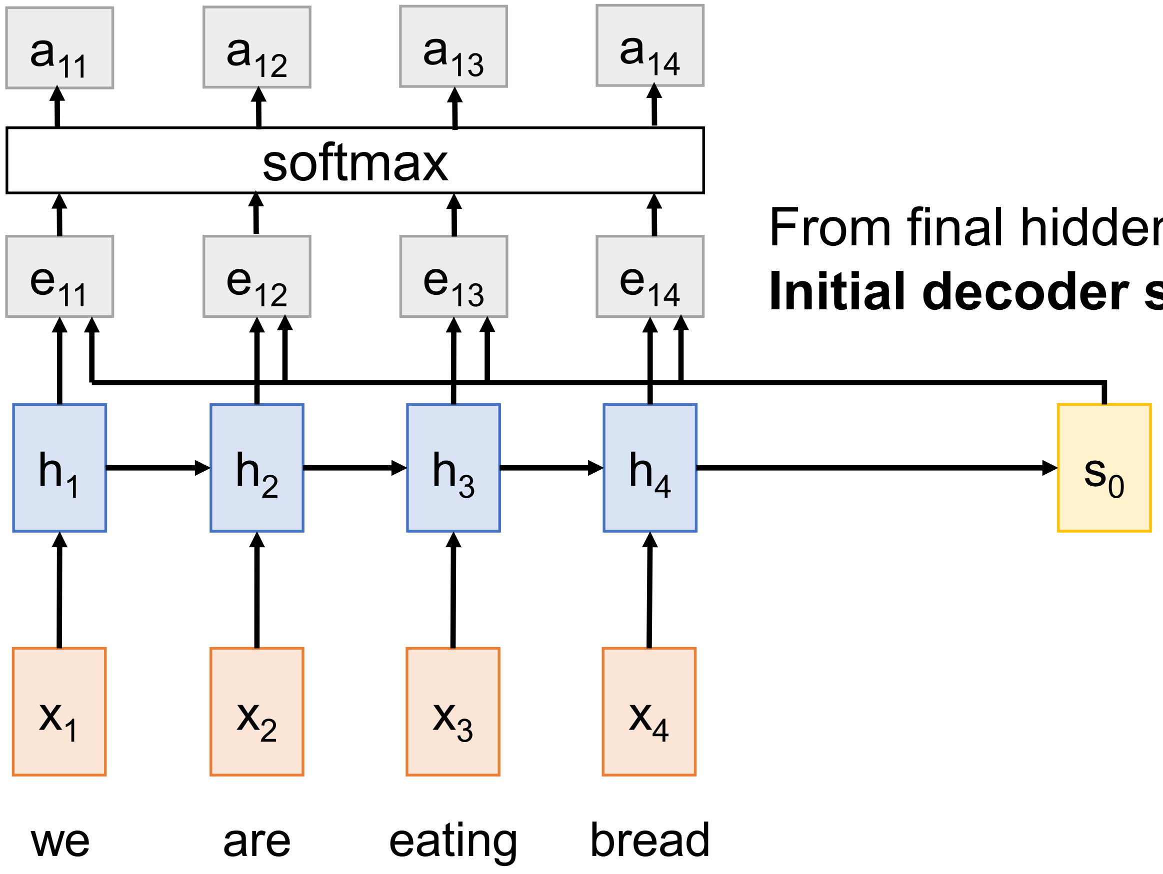
$e_{t,i} = f_{att}(s_{t-1}, h_i)$ **(f_{att} is an MLP)**



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

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

Machine Translation with RNNs **and Attention**



From final hidden state:
Initial decoder state s_0

Compute **alignment scores**

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

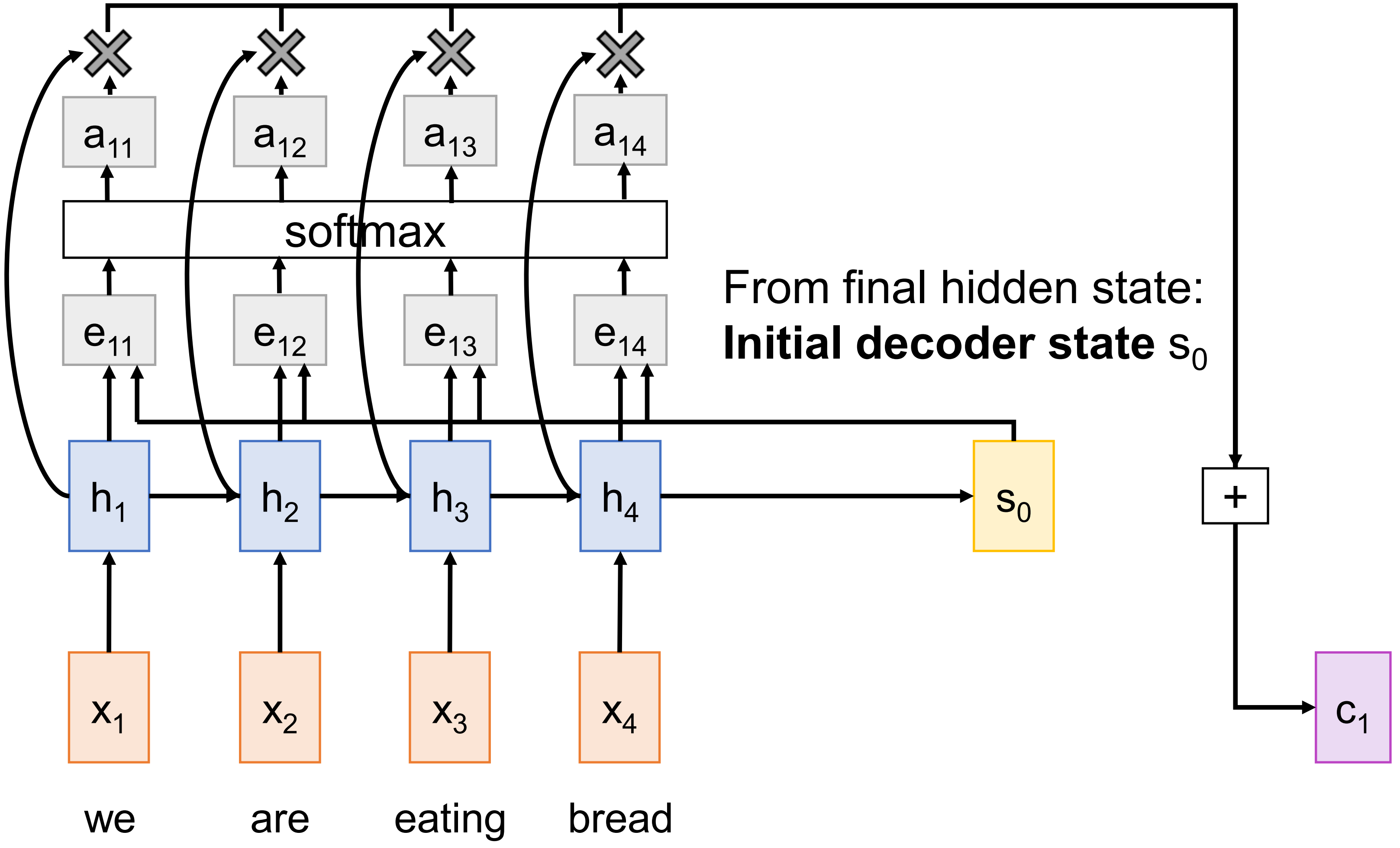
Normalize to get **attention weights**

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

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 **alignment scores**

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

Normalize to get **attention weights**

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

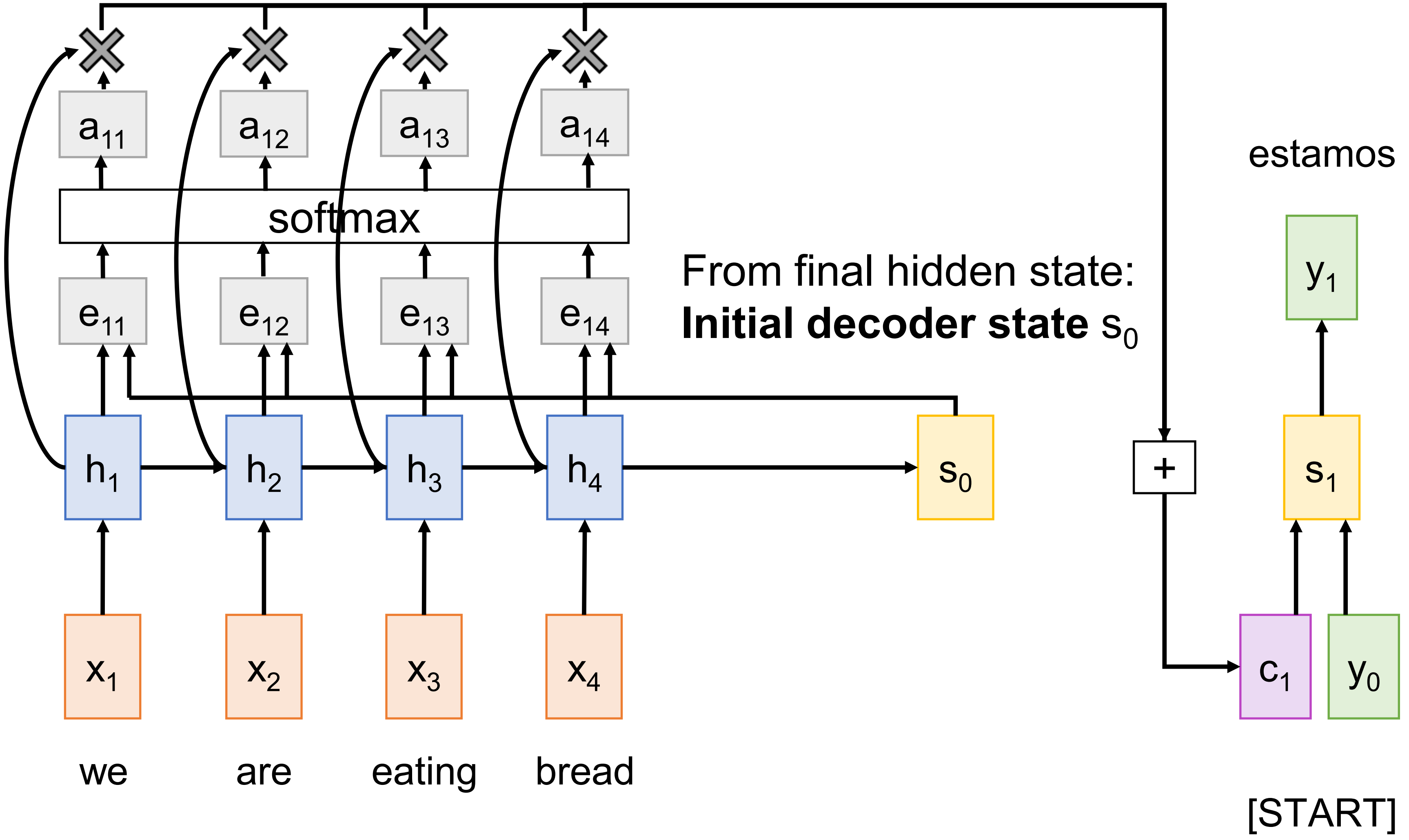
Set context vector **c** to a linear combination of hidden states

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

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**



Compute **alignment scores**

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

Normalize to get **attention weights**

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

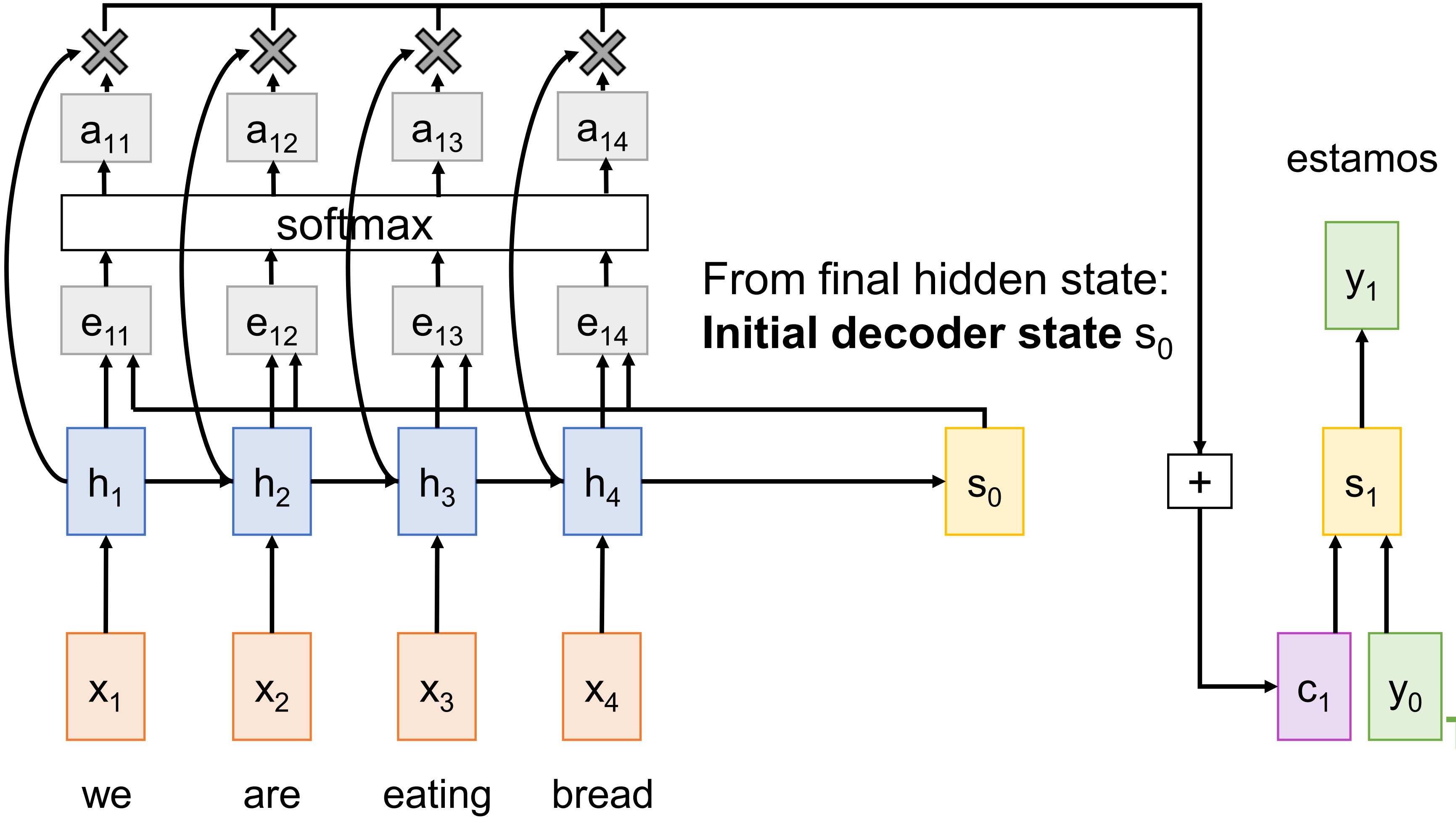
Set context vector **c** to a linear combination of hidden states

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

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**



Compute **alignment scores**

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

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

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

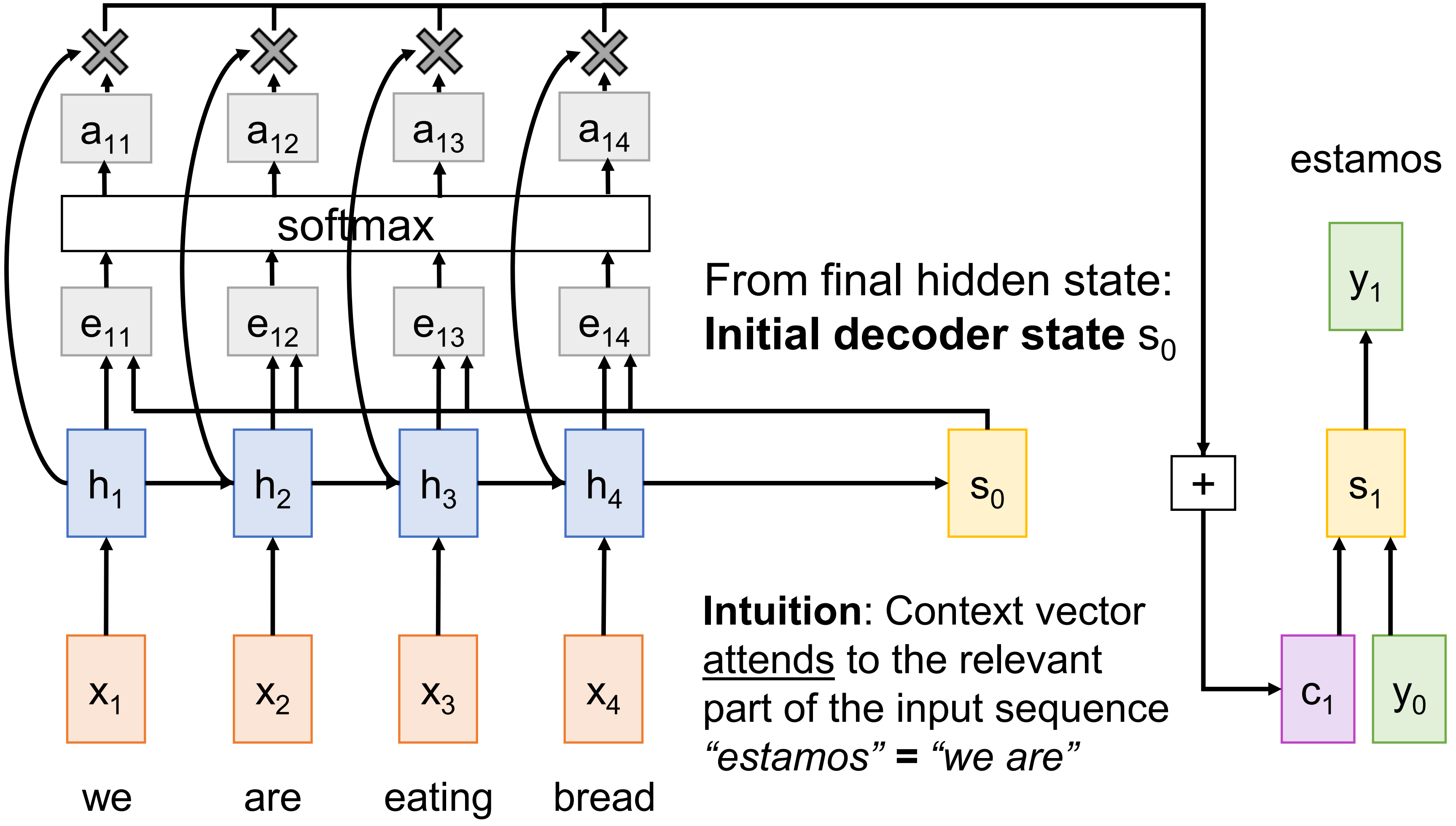
This is all differentiable! Do not supervise attention weights – backprop through everything

Can be seen as a input-dependent weighting (rather than MLP)

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

Intuition: Context vector <u>attends</u> to the relevant part of the input sequence *"estamos"* **=** *"we are"*

a_{11}=0.45, a_{12}=0.45, a_{13}=0.05, a_{14}=0.05

Compute **alignment scores**

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

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

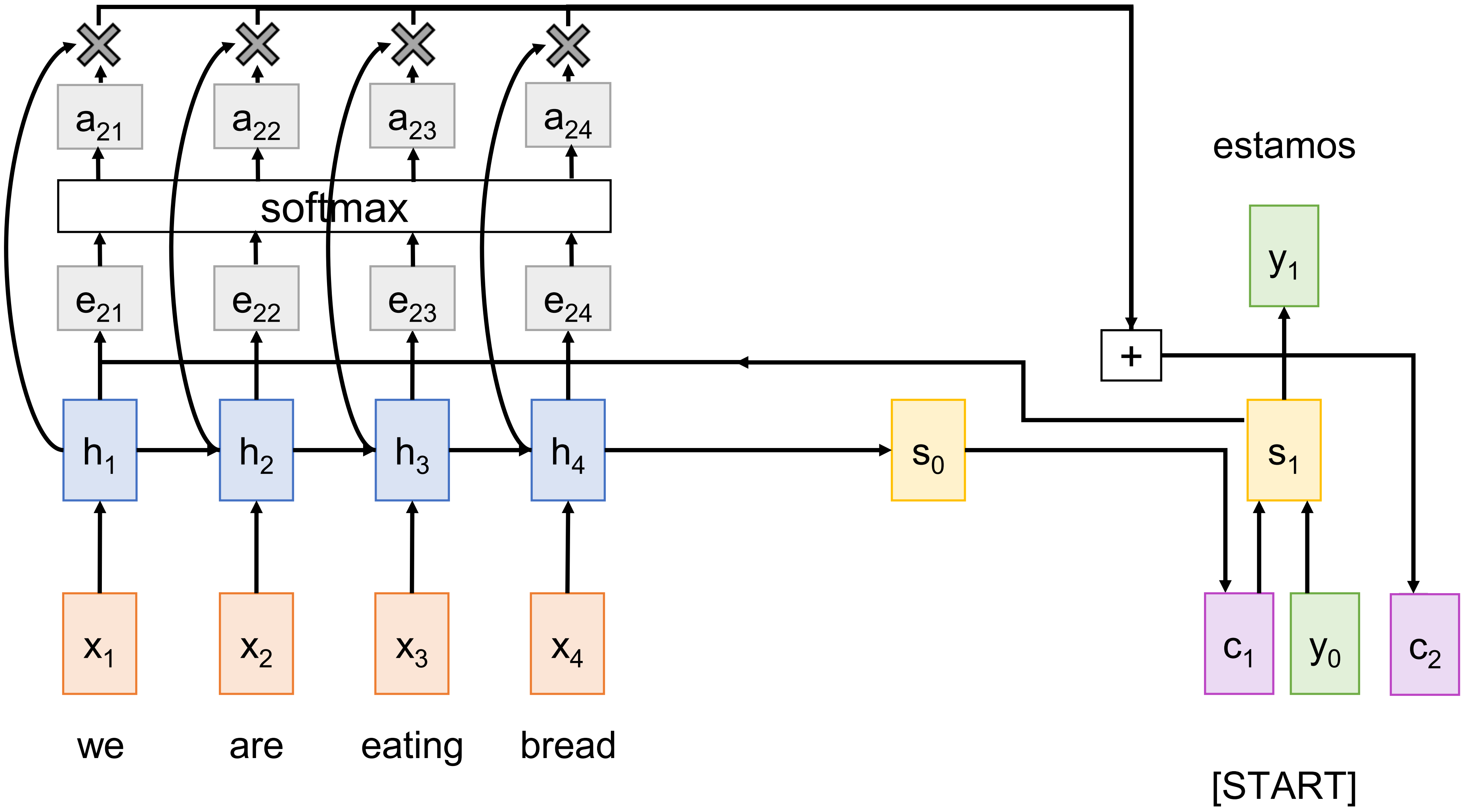
$c_t = \sum_i a_{t,i} h_i$

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

This is an inductive bias we think is reasonable for this task. Need to verify

Slide credit: Justin Johnson

Machine Translation with RNNs **and Attention**

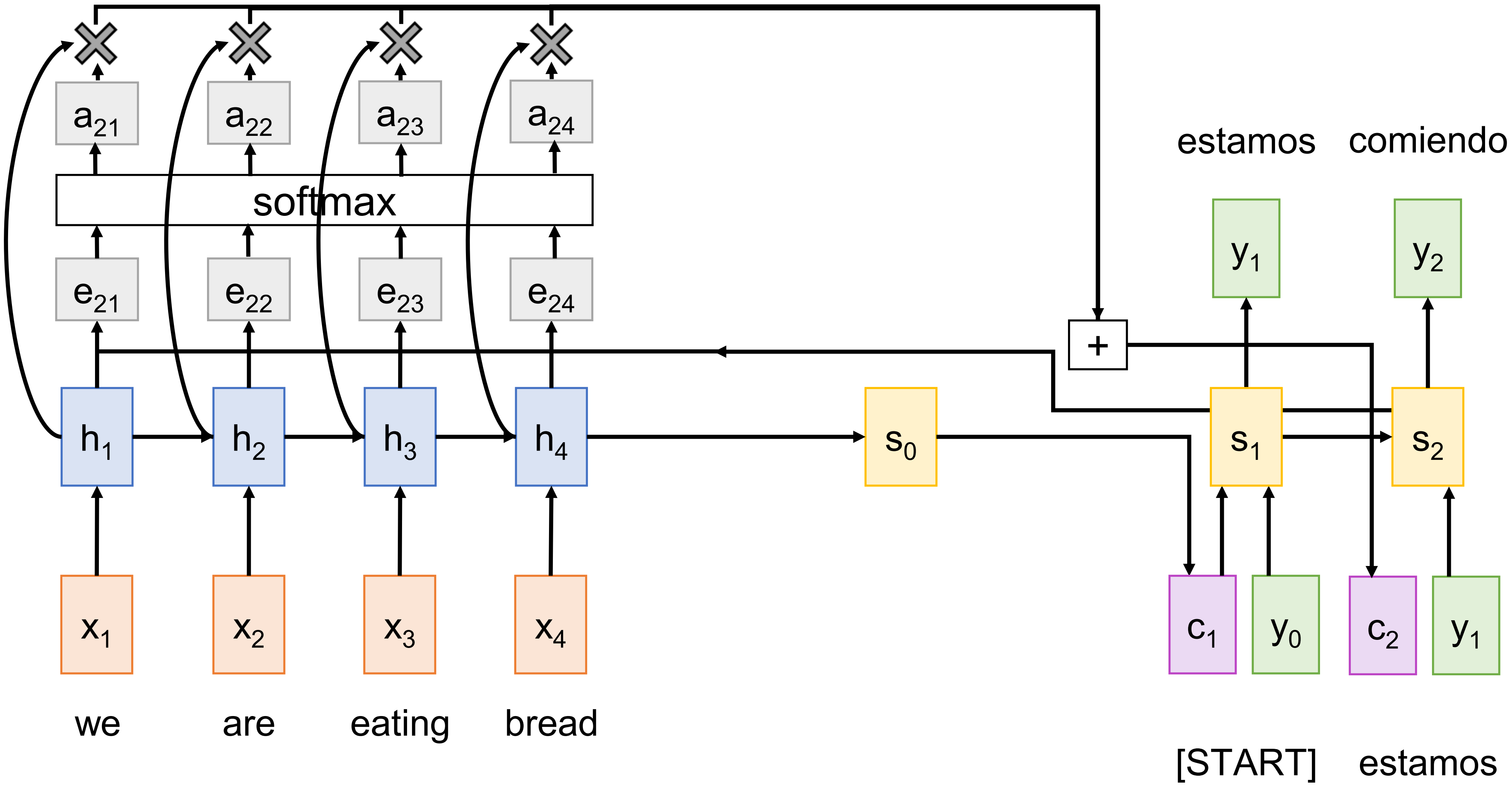


Repeat: Use s_1 to compute new context vector c_2

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**



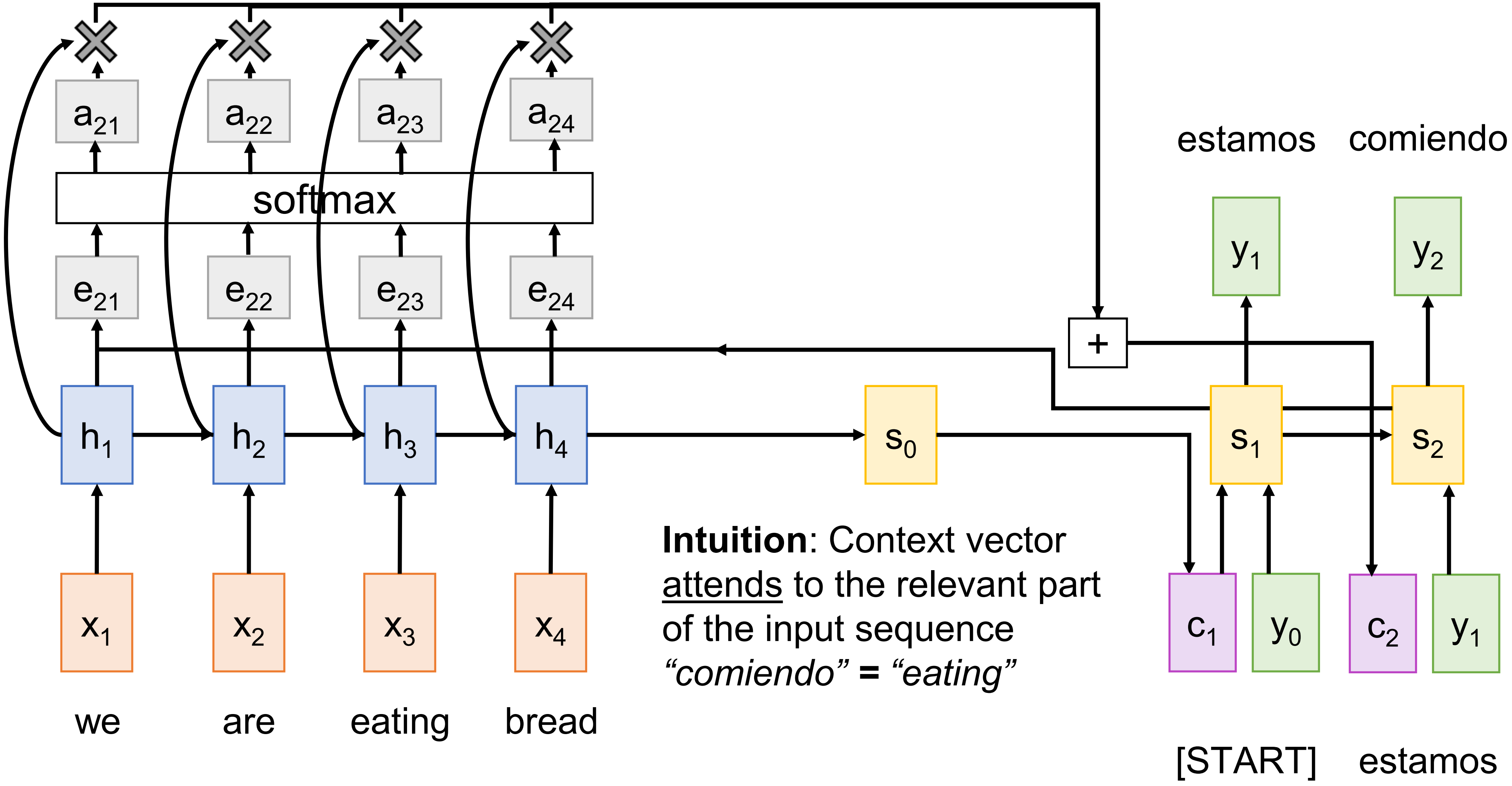
Repeat: Use s_1 to compute new context vector c_2

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**



Repeat: Use s_1 to compute new context vector c_2

Use c_2 to compute s_2, y_2

Intuition: Context vector attends to the relevant part of the input sequence *"comiendo"* **=** *"eating"*

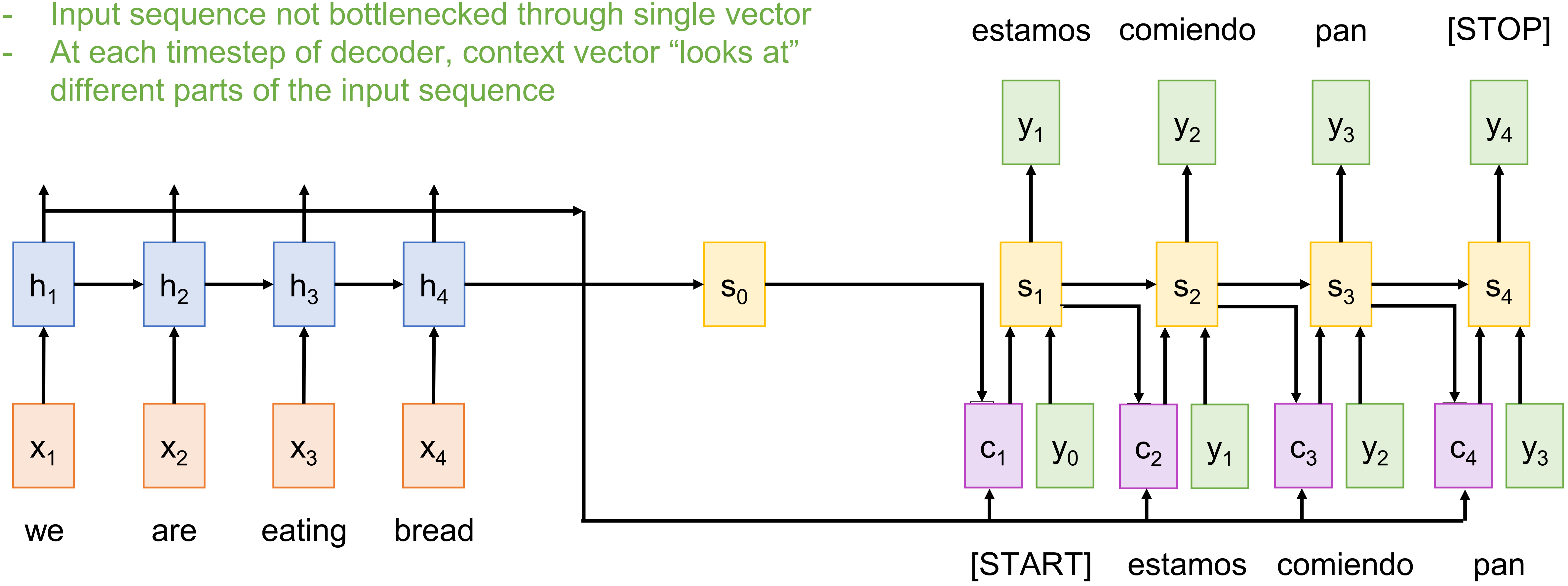
Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

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



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

Slide credit: Justin Johnson

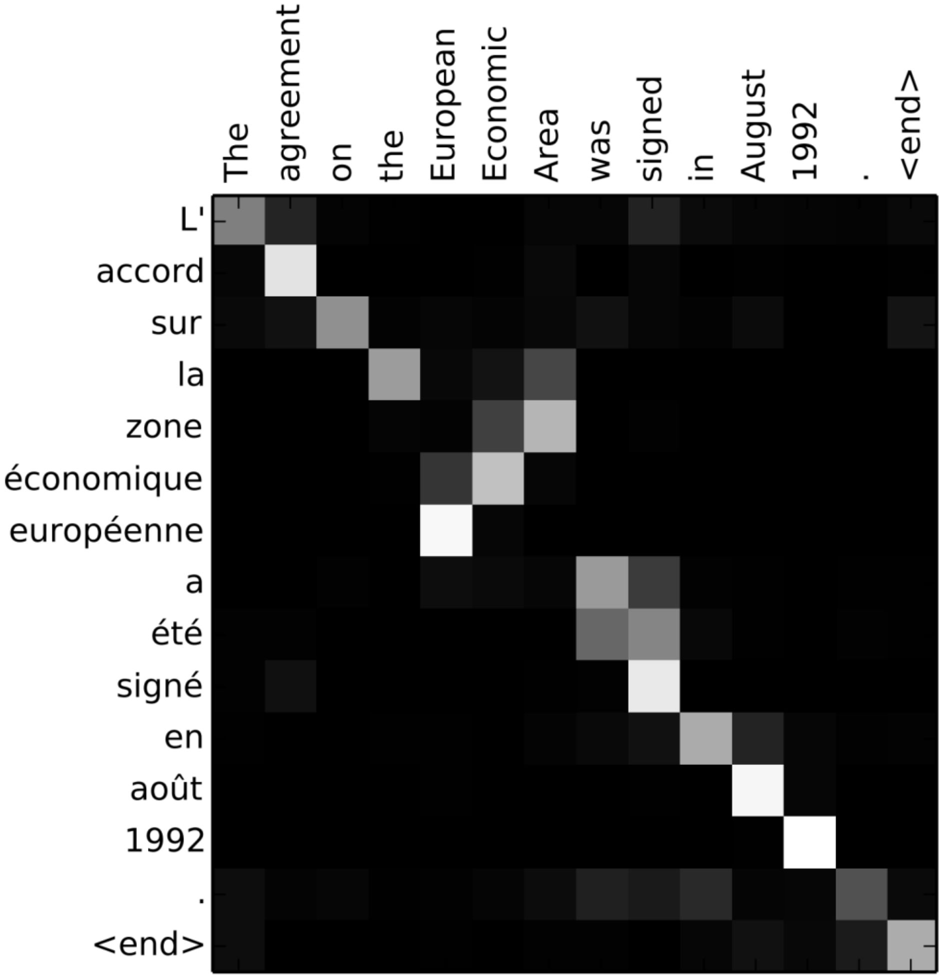
Machine Translation with RNNs **and Attention**

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

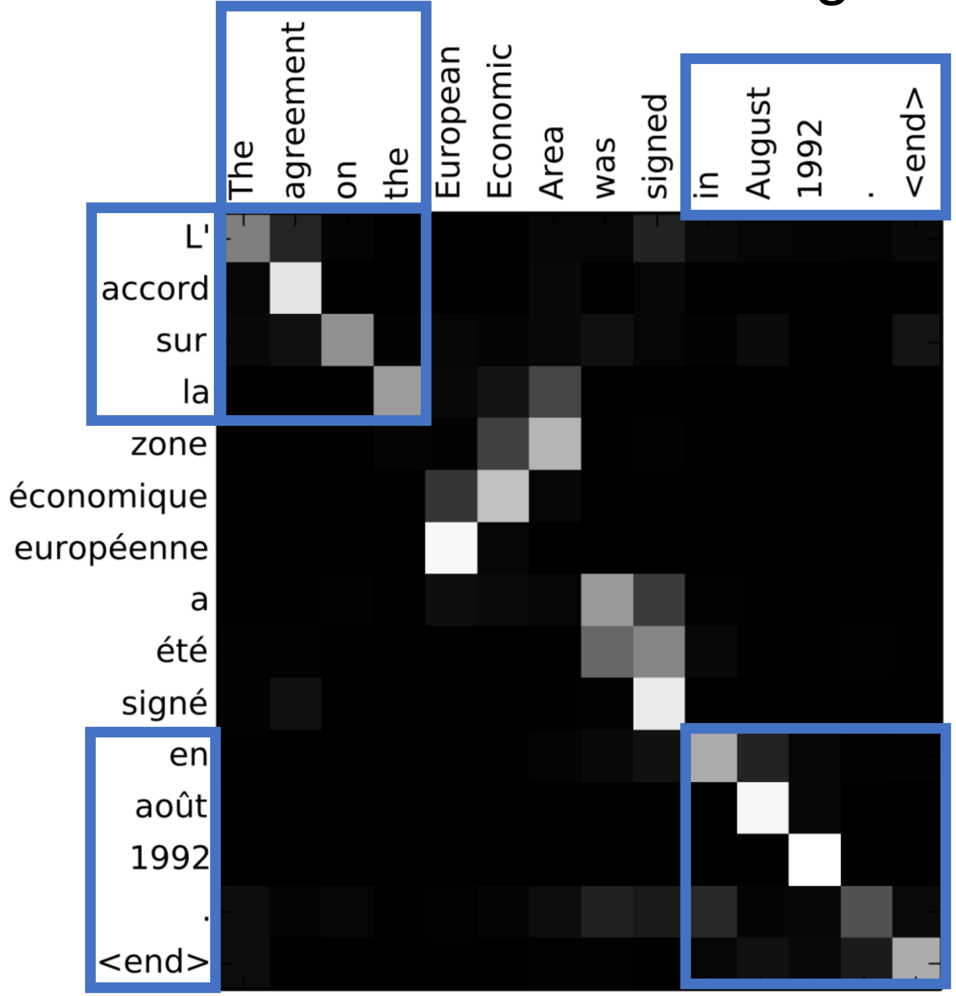
Input: “**The agreement on the** European Economic Area was signed **in August 1992**.”

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

Diagonal attention means words correspond in order

Diagonal attention means words correspond in order

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**."

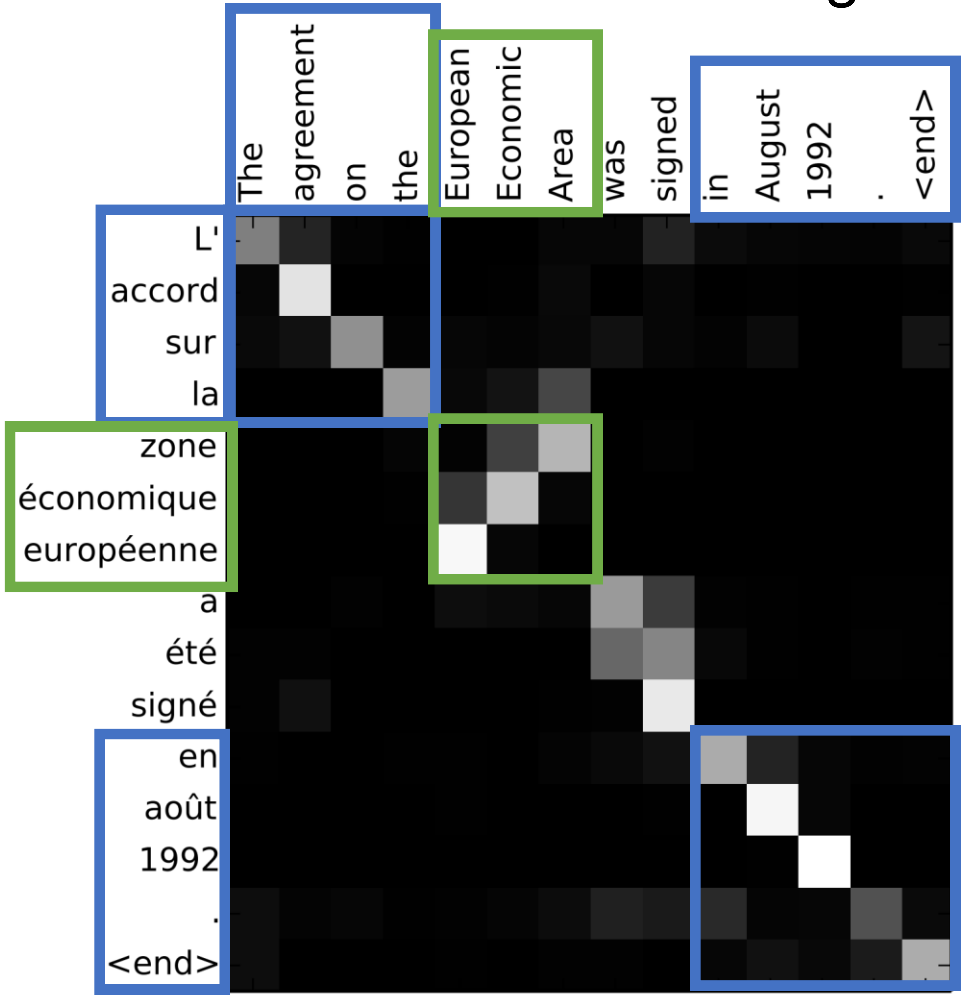
Output: "**L'accord sur la zone économique européenne** a été signé **en août 1992**."

Diagonal attention means words correspond in order

Attention figures out different word orders

Diagonal attention means words correspond in order

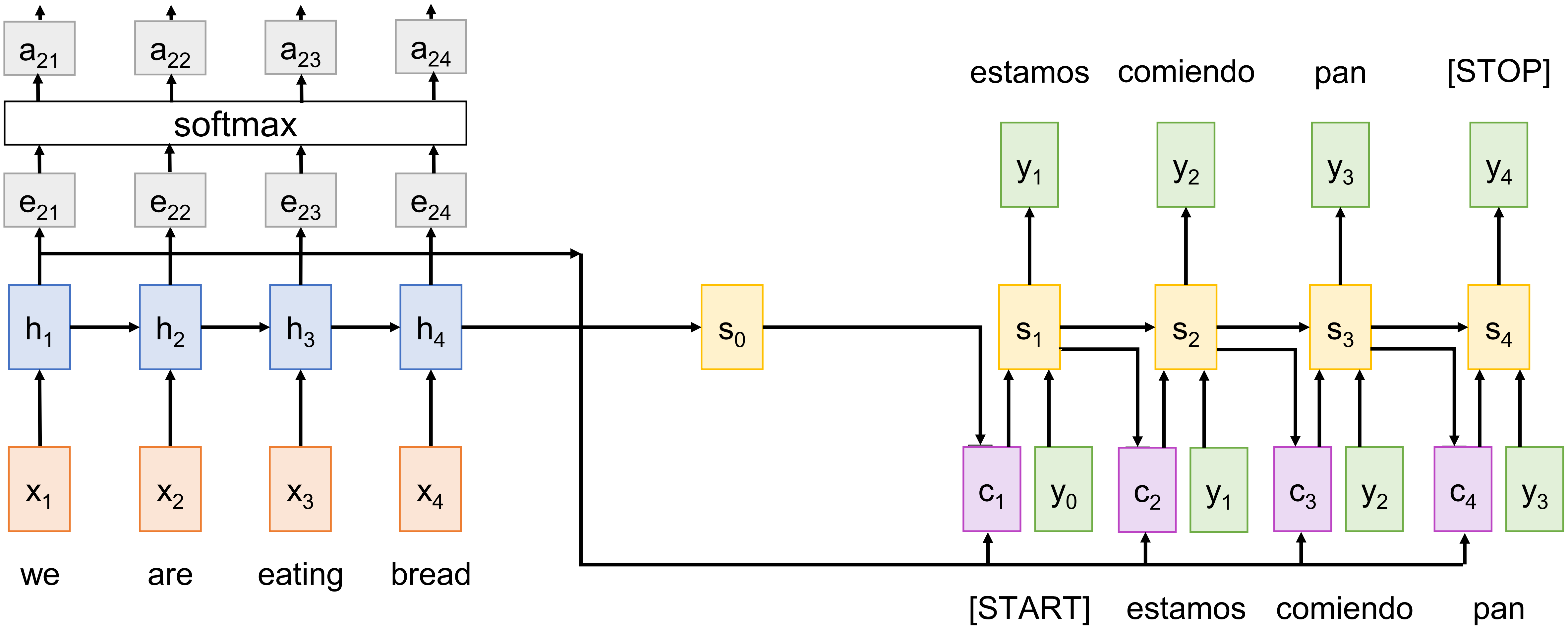
Visualize attention weights $a_{t,i}$



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

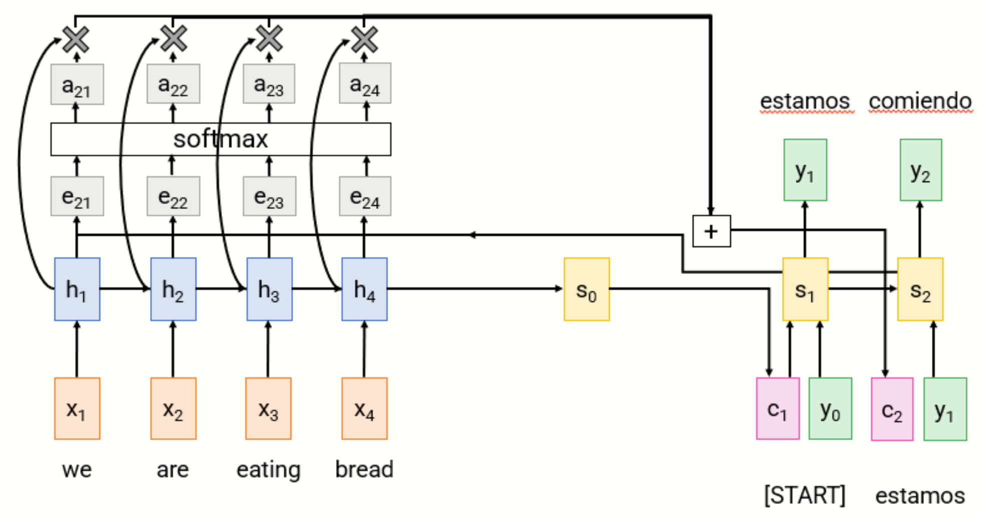
Slide credit: Justin Johnson

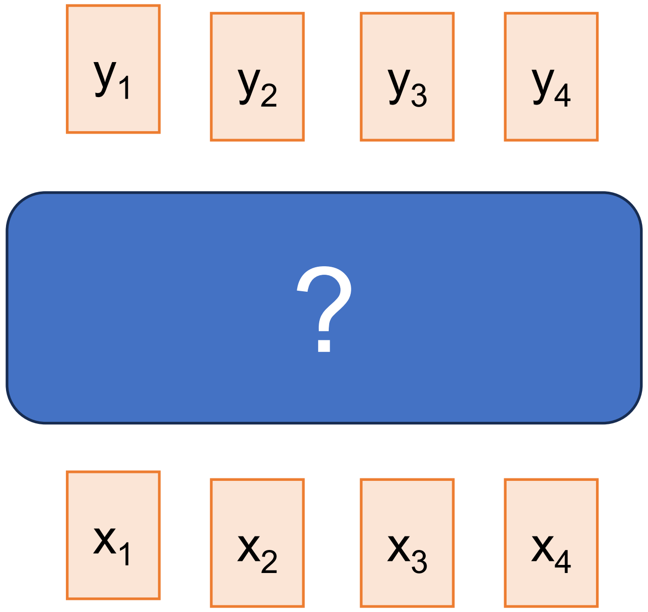
Machine Translation with RNNs **and Attention**



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

Slide credit: Justin Johnson





Idea: Can we use **attention** as a fundamental building block for a generic sequence (input) to sequence (output) layer?

Note: We just want a generic sequence-in, sequence-out model that will represent each input *contextualized* with rest of inputs, and encode meaning of entire sequence

We will progressively develop a generic mechanism using idea of attention.
Don't try to map to RNN translation example!

Attention Layer

Inputs:
State vector: $\mathbf{s_i}$ (Shape: D_Q)
Hidden vectors: $\mathbf{h_i}$ (Shape: N_X x D_H)
Similarity function: f_{att}

Computation:
Similarities: e (Shape: N_X) $e_i = f_{att}(\mathbf{s_{t-1}}, \mathbf{h_i})$
Attention weights: a = softmax(e) (Shape: N_X)
Output vector: $y = \sum_i a_i \mathbf{h_i}$ (Shape: D_X)

Slide credit: Justin Johnson

Attention Layer

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

Make the module generic:
Input (X), Query (q)
Output (Weighted sum of inputs)

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)

Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vector: $\mathbf{q}$ (Shape: D_Q)
Input vectors: $\mathbf{X}$ (Shape: N_X x D_Q)
Similarity function: dot product

Computation:
Similarities: e (Shape: N_X) $e_i = \mathbf{q} \cdot \mathbf{X_i}$
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

Inputs:
Query vector: **q** (Shape: D_Q)
Input vectors: **X** (Shape: N_X x D_Q)
Similarity function: scaled dot product

Computation:
Similarities: e (Shape: N_X) $e_i = \mathbf{q} \cdot \mathbf{X_i} / \text{sqrt}(D_Q)$
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

Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: **Q** (Shape: $N_Q \times D_Q$)
Input vectors: **X** (Shape: $N_X \times D_Q$)

Make the module generic:
Sequence Input (X), Sequence Query (Q)
Output: Sequence (Weighted sum/mixture of inputs)

Computation:
Similarities: $E = QX^T$ (Shape: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot X_j / \text{sqrt}(D_Q)$
Attention weights: $A = \text{softmax}(E, \text{dim}=1)$ (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 dot product for similarity
- Multiple **query** vectors

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)

Separate concerns:
1) *Matching* (similarity) -> Key,
2) Output given weighting -> Value

Computation:
Key vectors: **K** = **X**W_K (Shape: N_X x D_Q)
Value vectors: **V** = **X**W_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) $Y_i = \sum_j A_{i,j} V_j$

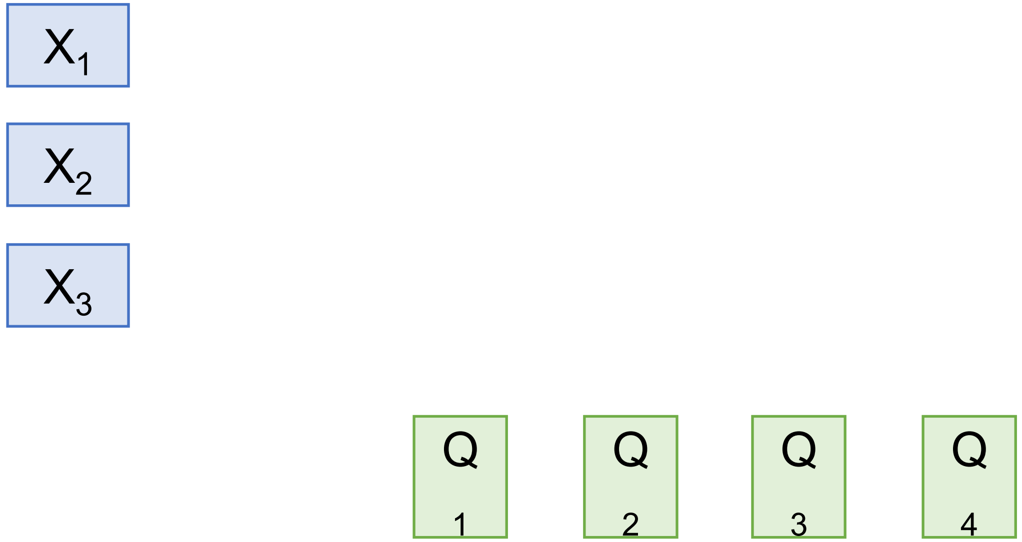
Changes:
- Use dot product for similarity
- Multiple **query** vectors
- Separate **key** and **value**

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: $N_Q \times N_X$) $E_{i,j} = Q_i \cdot K_j / \mathrm{sqrt}(D_Q)$
Attention weights: $A = \mathrm{softmax}(E, \mathrm{dim}=1)$ (Shape: $N_Q \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_Q \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

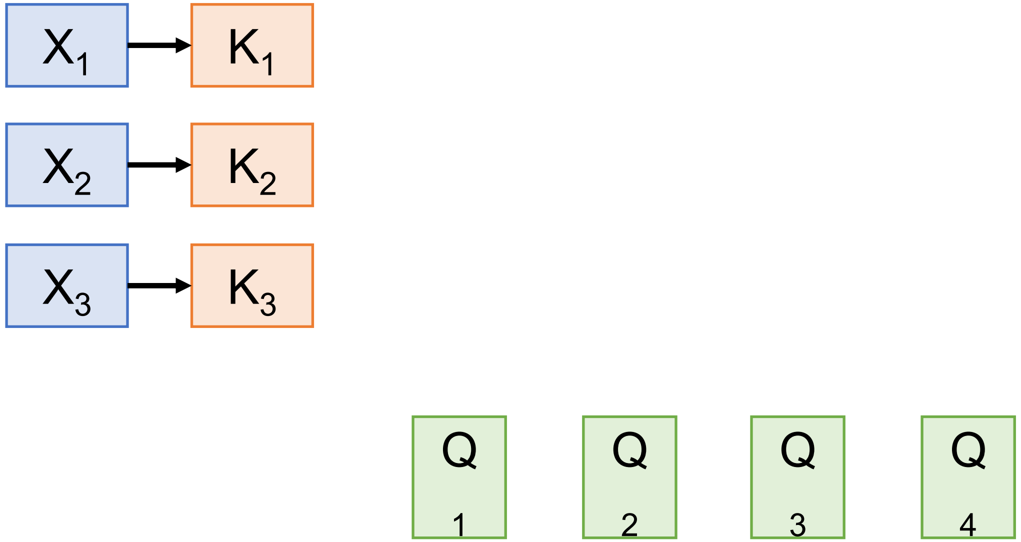


Slide credit: Justin Johnson

Attention Layer

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

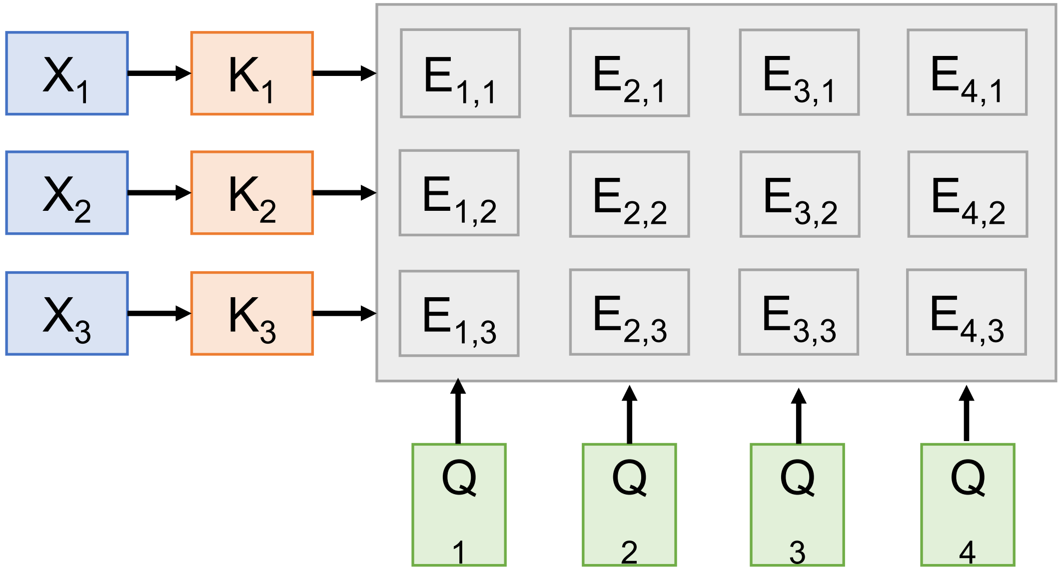


Slide credit: Justin Johnson

Attention Layer

Inputs:
Query vectors: **Q** (Shape: N_Q x D_Q)
Input vectors: **X** (Shape: N_X x D_X)
Key matrix: $\mathbf{W_K}$ (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) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \text{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = A**V** (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$

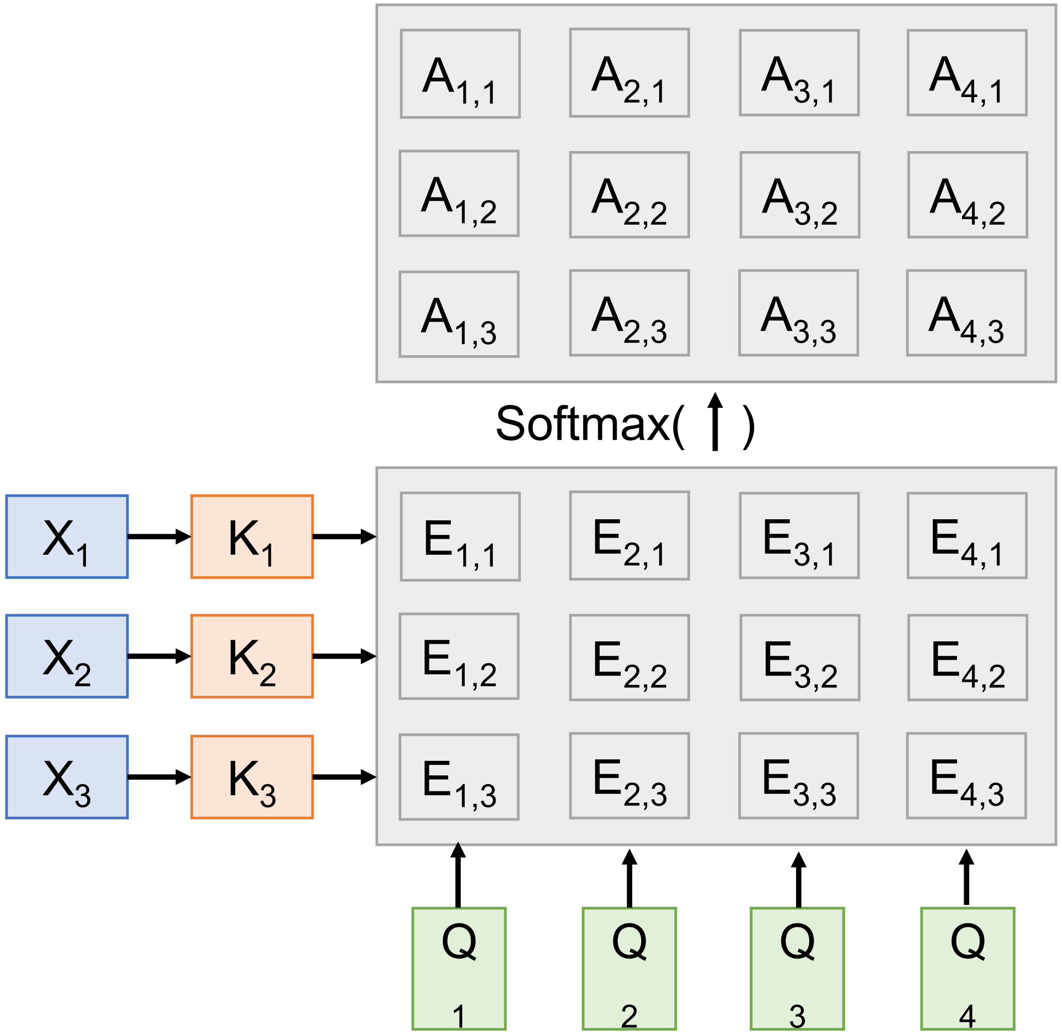


Slide credit: Justin Johnson

Attention 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) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$

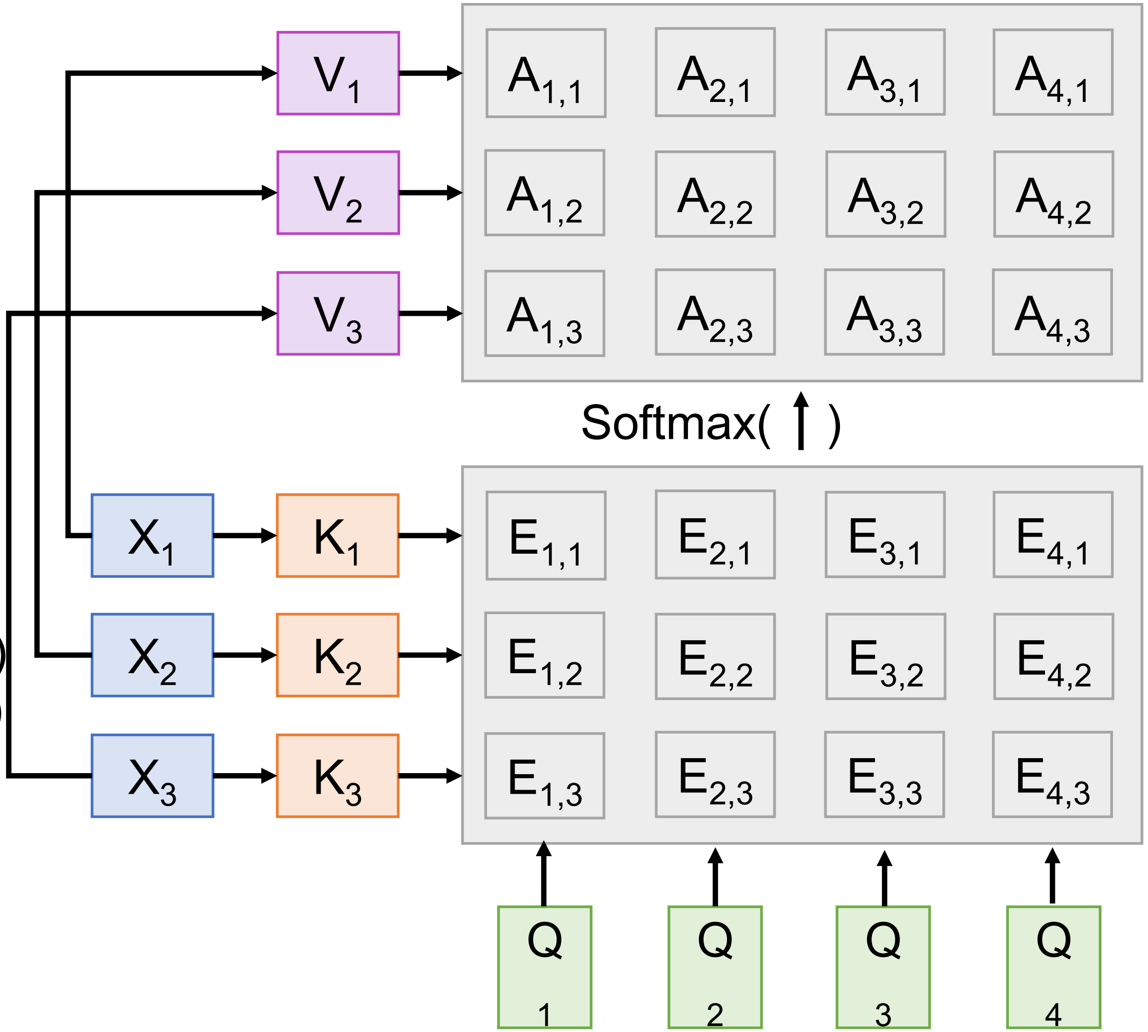


Slide credit: Justin Johnson

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) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \text{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: Y = A**V** (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$

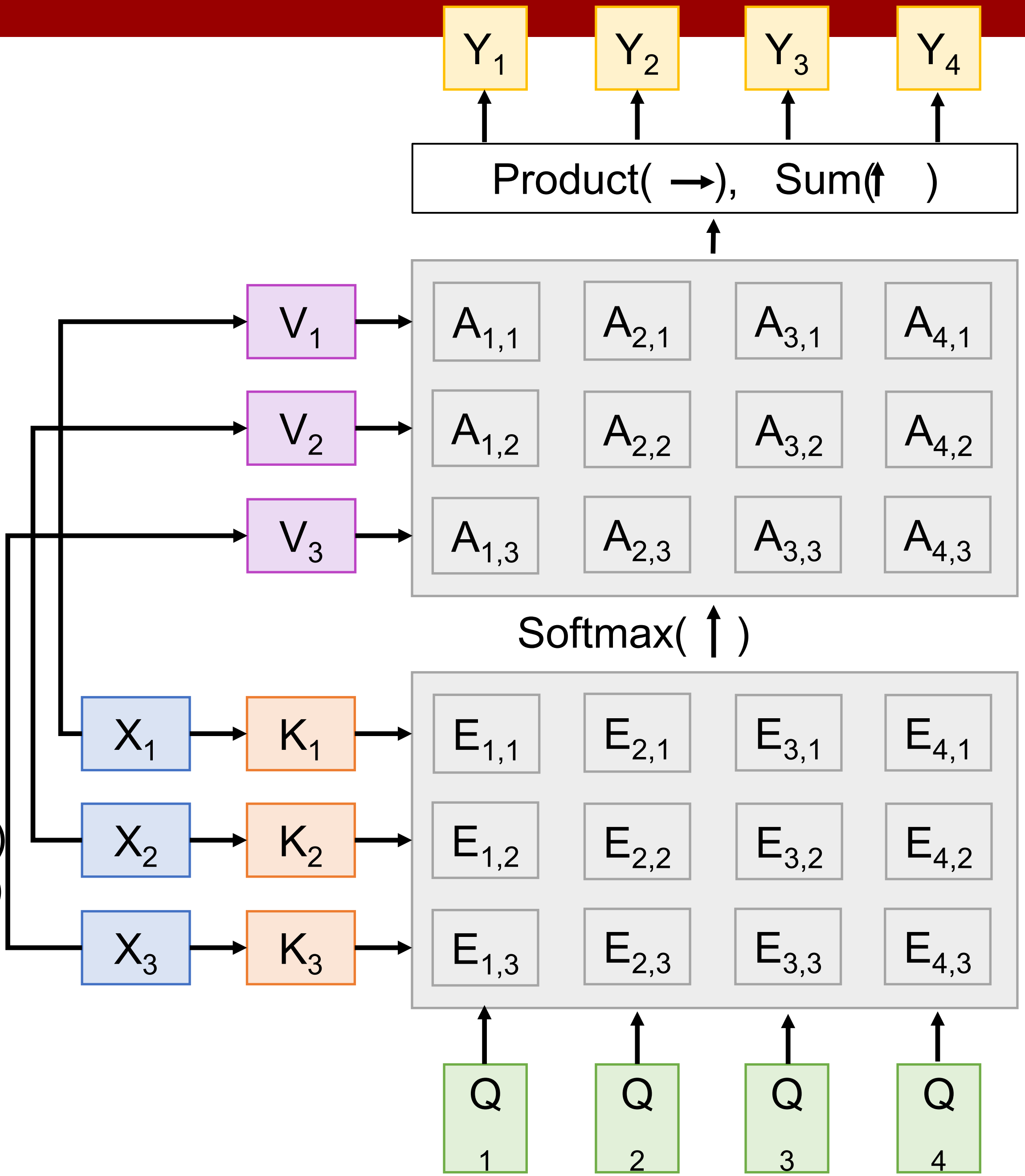


Slide credit: Justin Johnson

Attention 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: $\mathbf{K} = \mathbf{X}\mathbf{W_K}$ (Shape: N_X x D_Q)
Value vectors: $\mathbf{V} = \mathbf{X}\mathbf{W_V}$ (Shape: N_X x D_V)
Similarities: $E = \mathbf{Q}\mathbf{K}^T$ (Shape: N_Q x N_X) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \text{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_Q x N_X)
Output vectors: $Y = A\mathbf{V}$ (Shape: N_Q x D_V) $Y_i = \sum_j A_{i,j}\mathbf{V_j}$



Slide credit: Justin Johnson

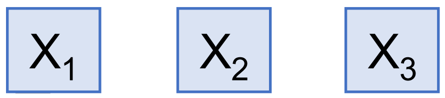
Self-Attention Layer

One **query** per **input vector**

Inputs:
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)
Query matrix: $\mathbf{W_Q}$ (Shape: D_X x D_Q)

Make the module generic:
Input: Sequence (X)
Output: Sequence (Weighted sum/mixture of inputs)

Computation:
Query vectors: $\mathbf{Q} = \mathbf{XW_Q}$
Key vectors: $\mathbf{K} = \mathbf{XW_K}$ (Shape: N_X x D_Q)
Value vectors: $\mathbf{V} = \mathbf{XW_V}$ (Shape: N_X x D_V)
Similarities: $E = \mathbf{QK}^T$ (Shape: N_X 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_X x N_X)
Output vectors: $Y = A\mathbf{V}$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j}\mathbf{V_j}$



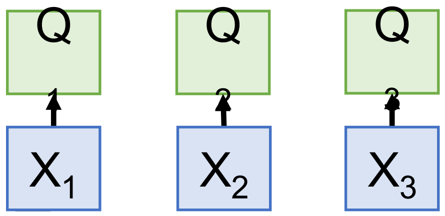
Slide credit: Justin Johnson

Self-Attention Layer

One **query** per **input vector**

Inputs:
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)
Query matrix: $\mathbf{W_Q}$ (Shape: D_X x D_Q)

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



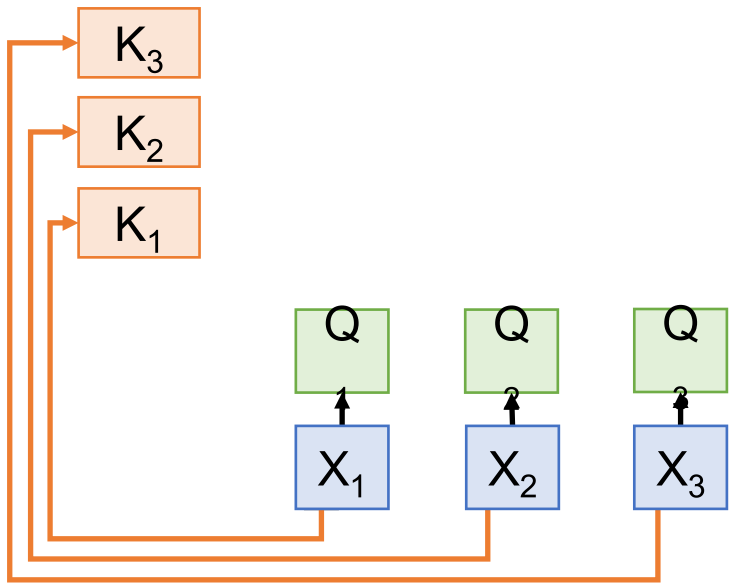
Slide credit: Justin Johnson

Self-Attention Layer

One **query** per **input vector**

Inputs:
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)
Query matrix: $\mathbf{W_Q}$ (Shape: D_X x D_Q)

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



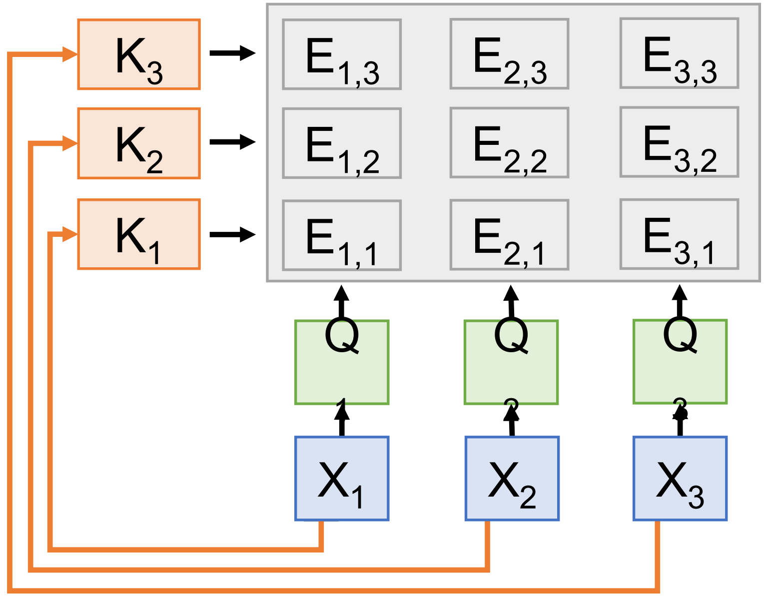
Slide credit: Justin Johnson

Self-Attention Layer

One **query** per **input vector**

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) $Y_i = \sum_j A_{i,j} V_j$



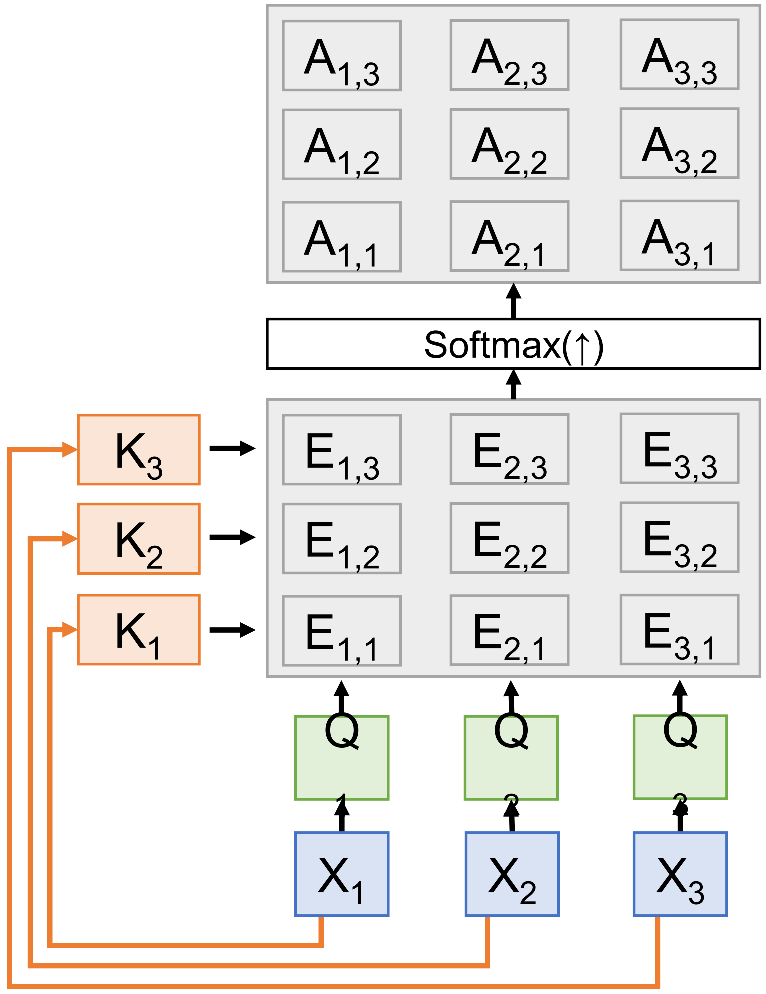
Slide credit: Justin Johnson

Self-Attention Layer

One **query** per **input vector**

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 = A**V** (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



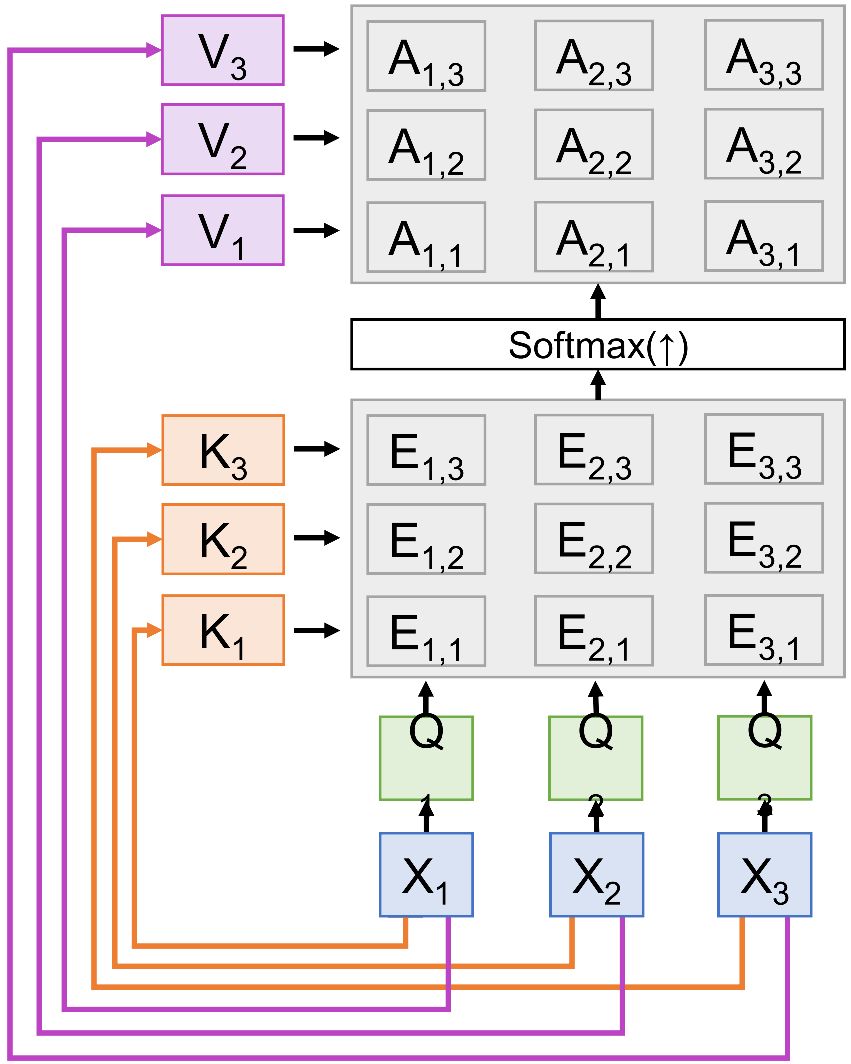
Slide credit: Justin Johnson

Self-Attention Layer

One **query** per **input vector**

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 = A**V** (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



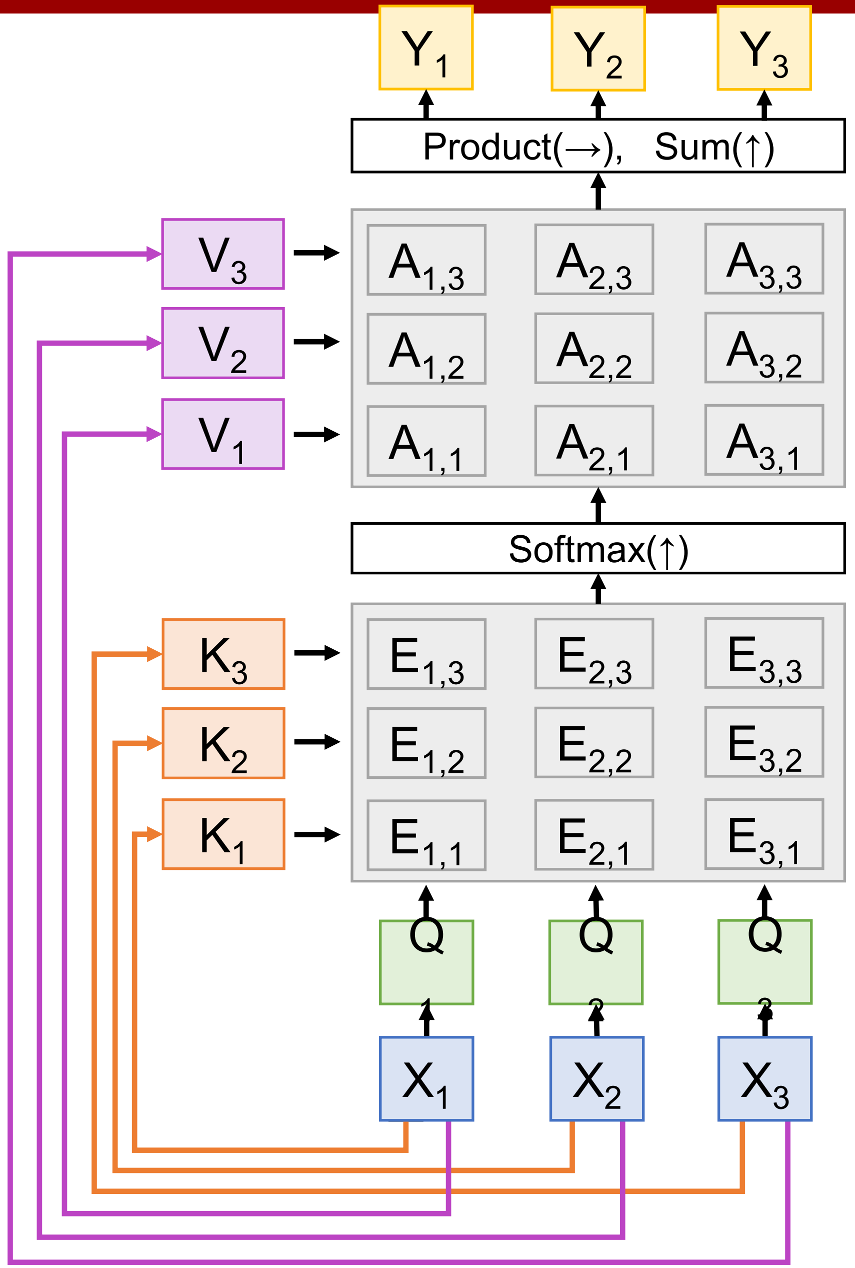
Slide credit: Justin Johnson

Self-Attention Layer

One query per input vector

Inputs:
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)
Query matrix: $\mathbf{W_Q}$ (Shape: D_X x D_Q)

Computation:
Query vectors: $\mathbf{Q} = \mathbf{XW_Q}$
Key vectors: $\mathbf{K} = \mathbf{XW_K}$ (Shape: N_X x D_Q)
Value vectors: $\mathbf{V} = \mathbf{XW_V}$ (Shape: N_X x D_V)
Similarities: $E = \mathbf{QK}^T$ (Shape: N_X 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_X x N_X)
Output vectors: $Y = A\mathbf{V}$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$



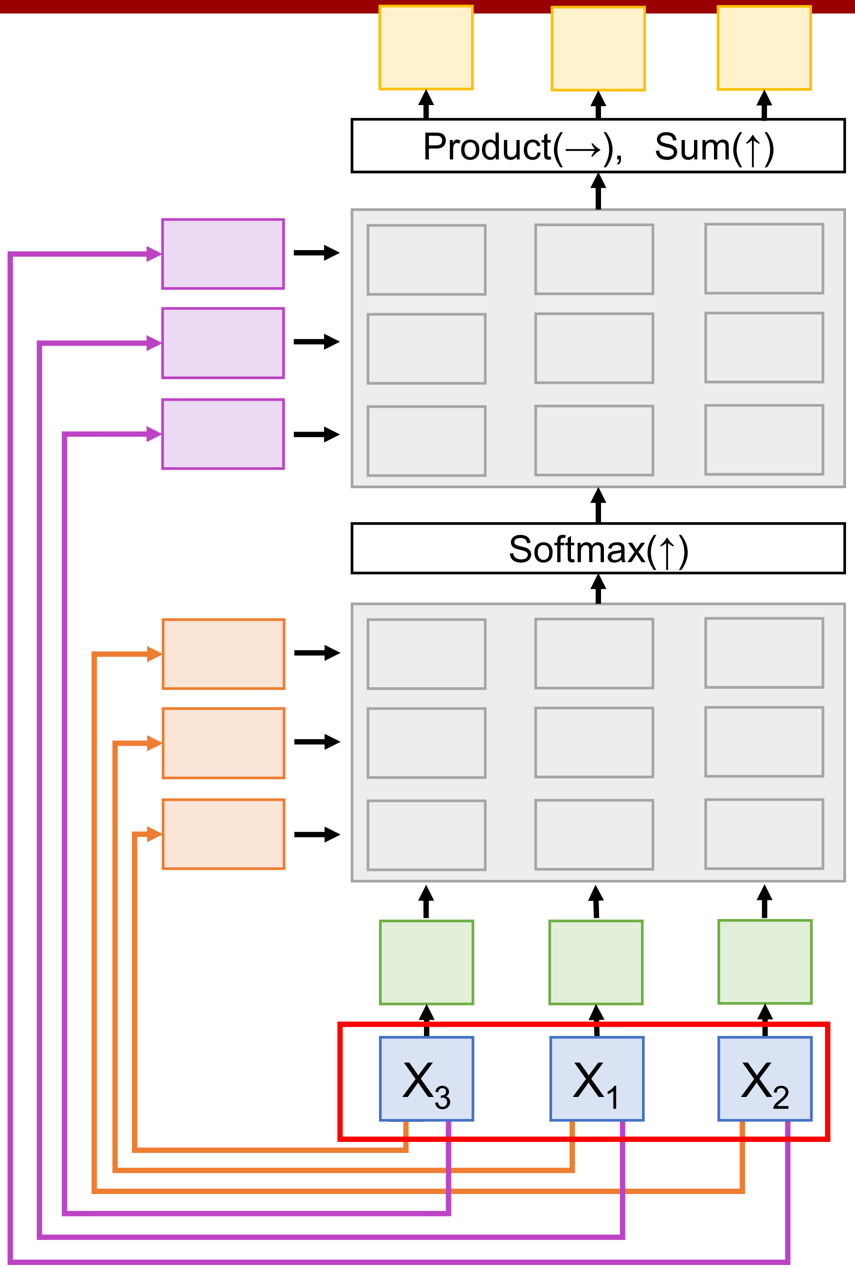
Slide credit: Justin Johnson

Self-Attention Layer

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

Computation:
Query vectors: **Q** = $\mathbf{XW_Q}$
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_X x N_X) $E_{i,j} = \mathbf{Q_i} \cdot \mathbf{K_j} / \text{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: Y = A**V** (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$

Consider **permuting** the input vectors:



Slide credit: Justin Johnson

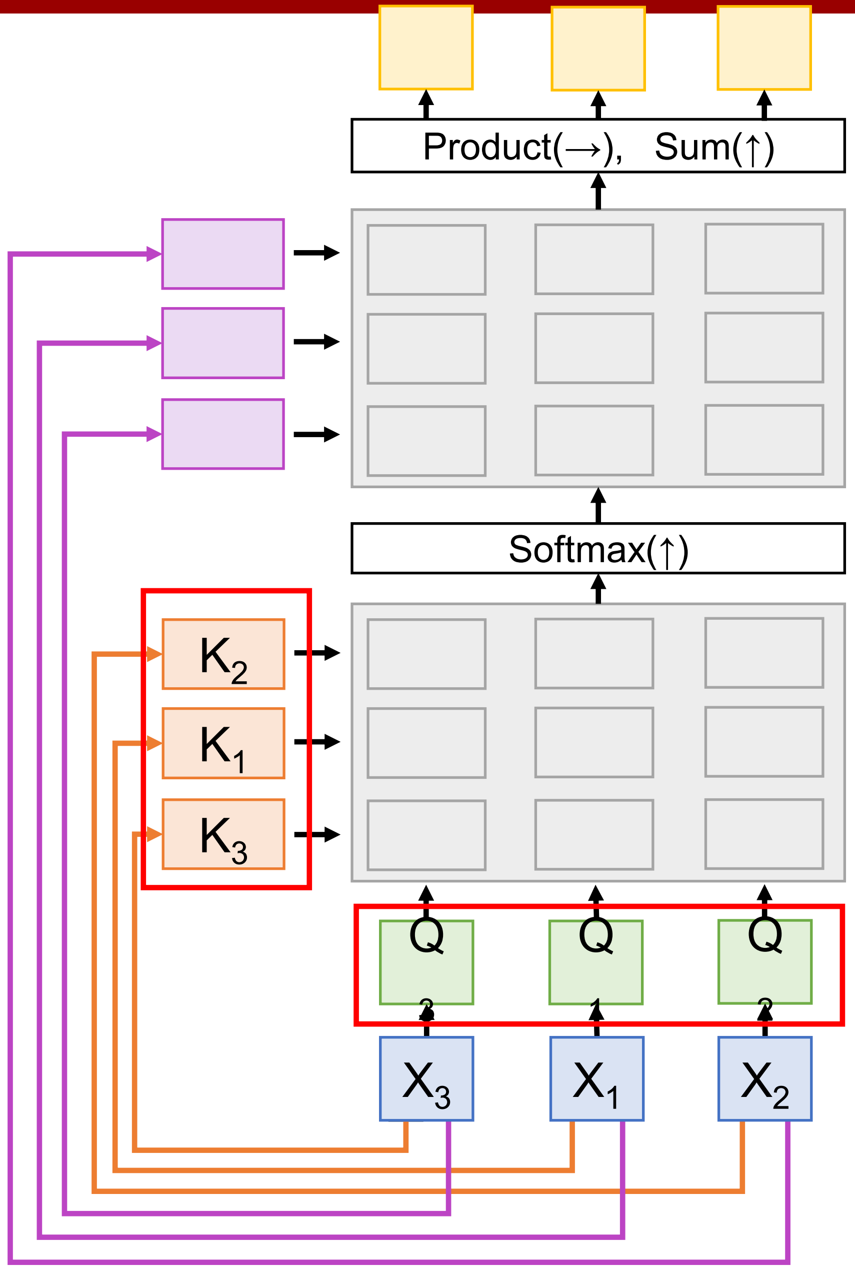
Self-Attention Layer

Inputs:
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)
Query matrix: $\mathbf{W_Q}$ (Shape: D_X x D_Q)

Consider **permuting** the input vectors:

Queries and Keys will be the same, but permuted

Computation:
Query vectors: $\mathbf{Q} = \mathbf{XW_Q}$
Key vectors: $\mathbf{K} = \mathbf{XW_K}$ (Shape: N_X x D_Q)
Value Vectors: $\mathbf{V} = \mathbf{XW_V}$ (Shape: N_X x D_V)
Similarities: $E = \mathbf{QK}^T$ (Shape: N_X 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_X x N_X)
Output vectors: $Y = A\mathbf{V}$ (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} \mathbf{V_j}$



Slide credit: Justin Johnson

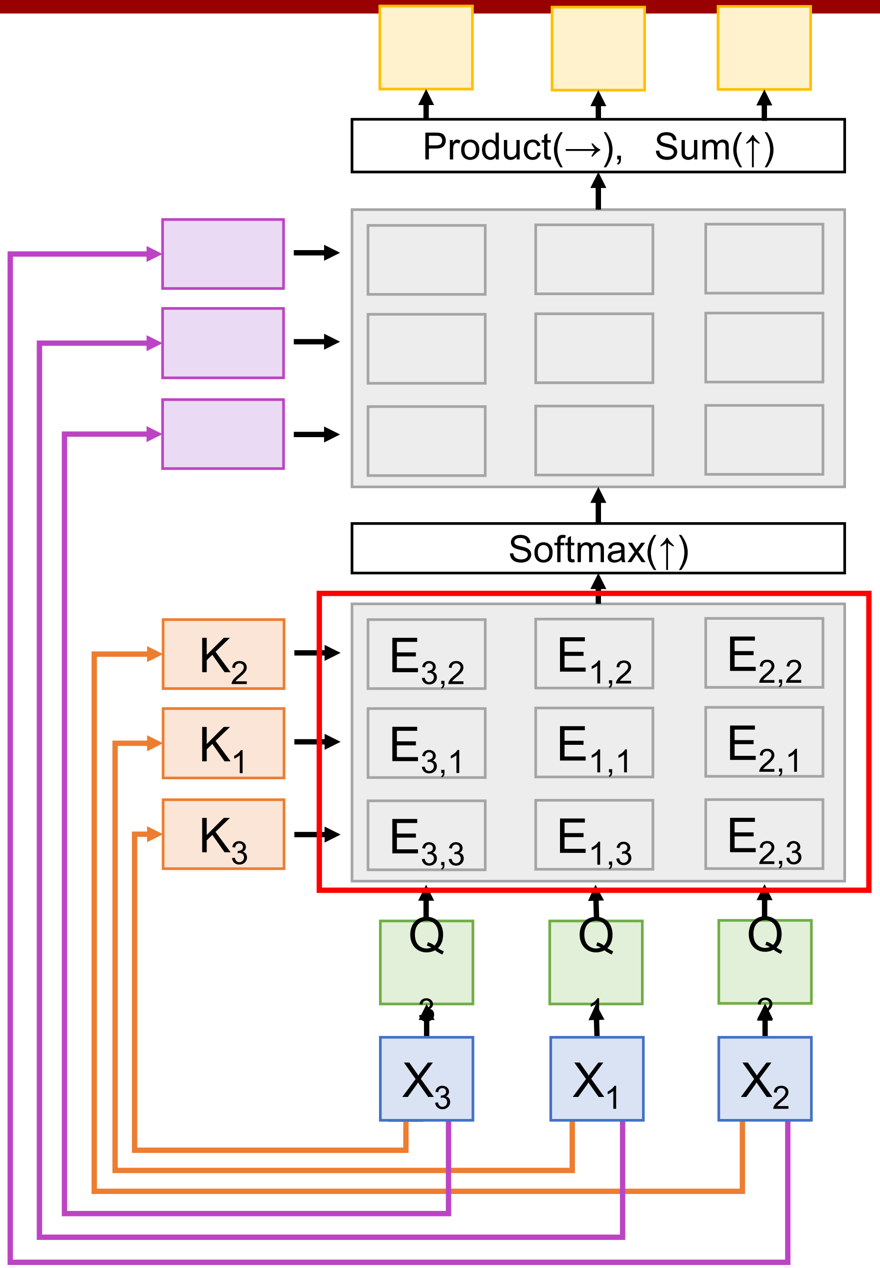
Self-Attention Layer

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)

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** · **K_j** / sqrt(D_Q)
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: Y = A**V** (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Slide credit: Justin Johnson

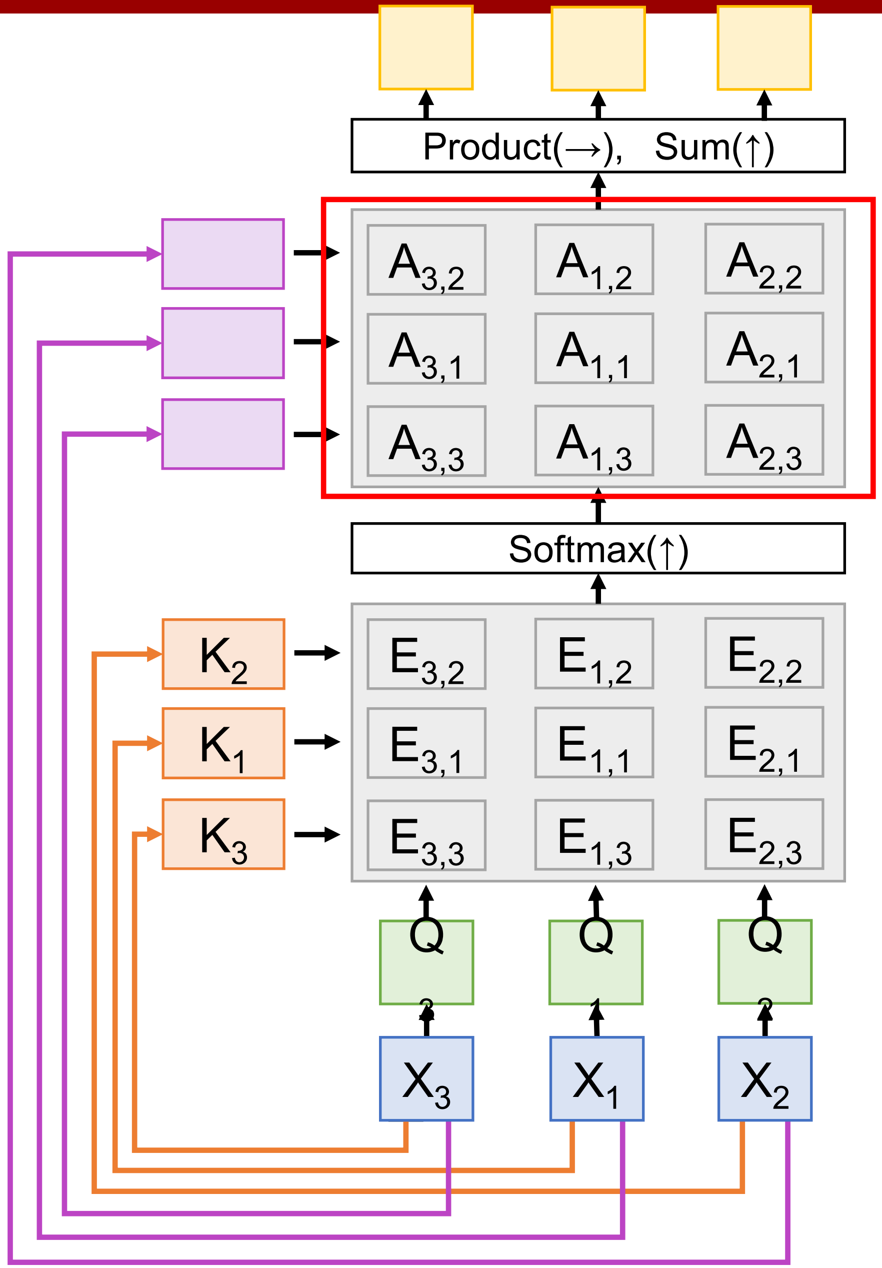
Self-Attention Layer

Inputs:
Input vectors: **X** (Shape: $N_X \times D_X$)
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 = softmax(E, dim=1) (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting** the input vectors:

Attention weights will be the same, but permuted



Slide credit: Justin Johnson

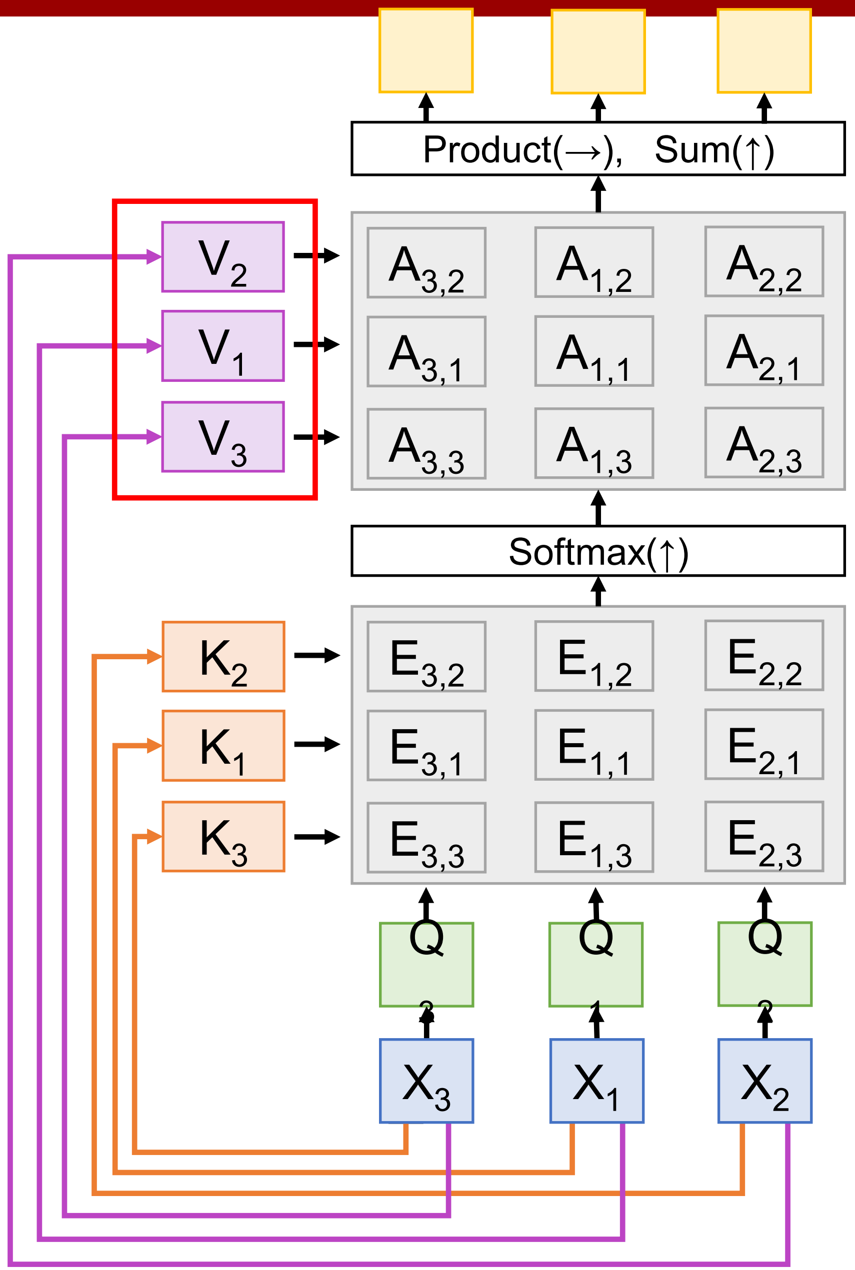
Self-Attention Layer

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)

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) $Y_i = \sum_j A_{i,j} V_j$

Consider **permuting** the input vectors:

Values will be the same, but permuted



Slide credit: Justin Johnson

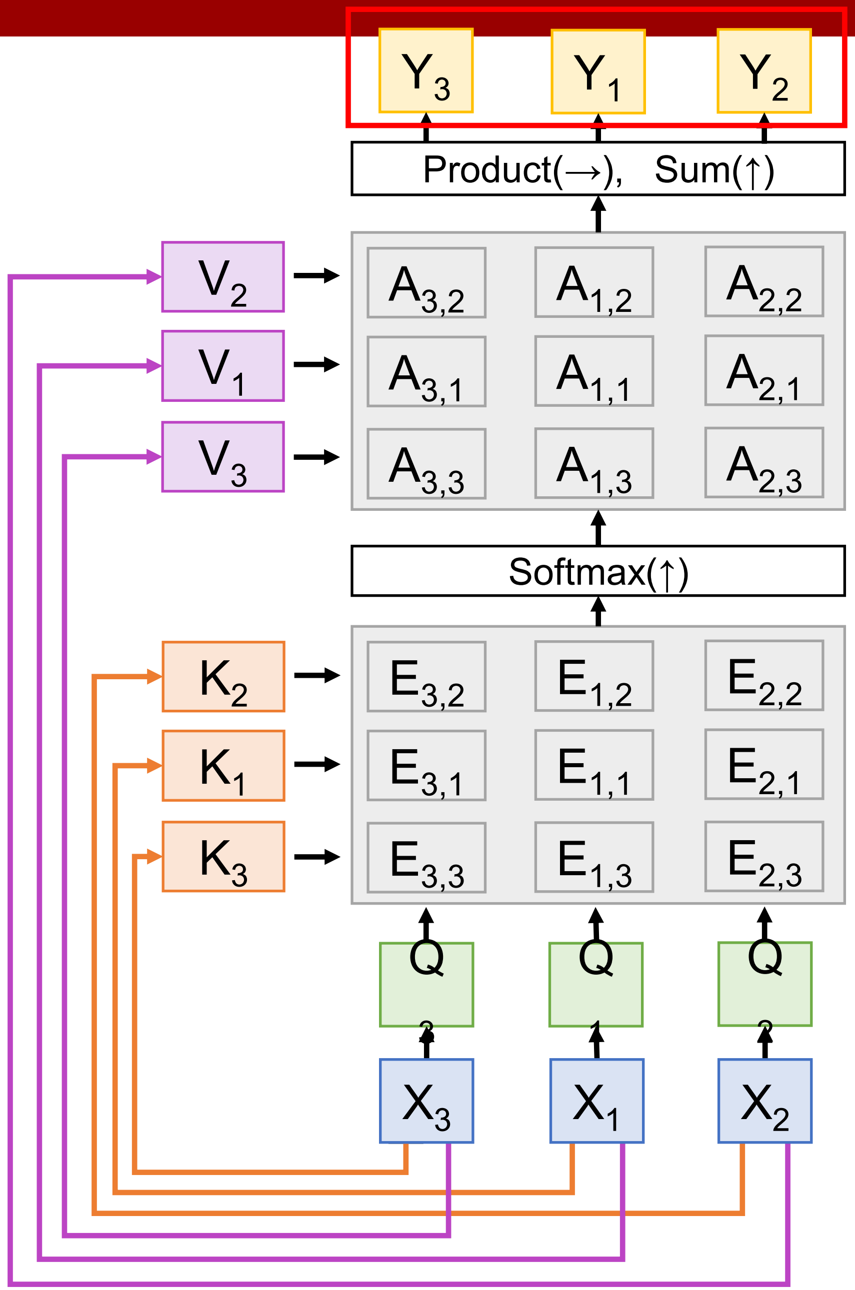
Self-Attention Layer

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:

Outputs 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 = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: Y = A**V** (Shape: N_X x D_V) $Y_i = \sum_j A_{i,j} V_j$



Slide credit: Justin Johnson

Self-Attention Layer

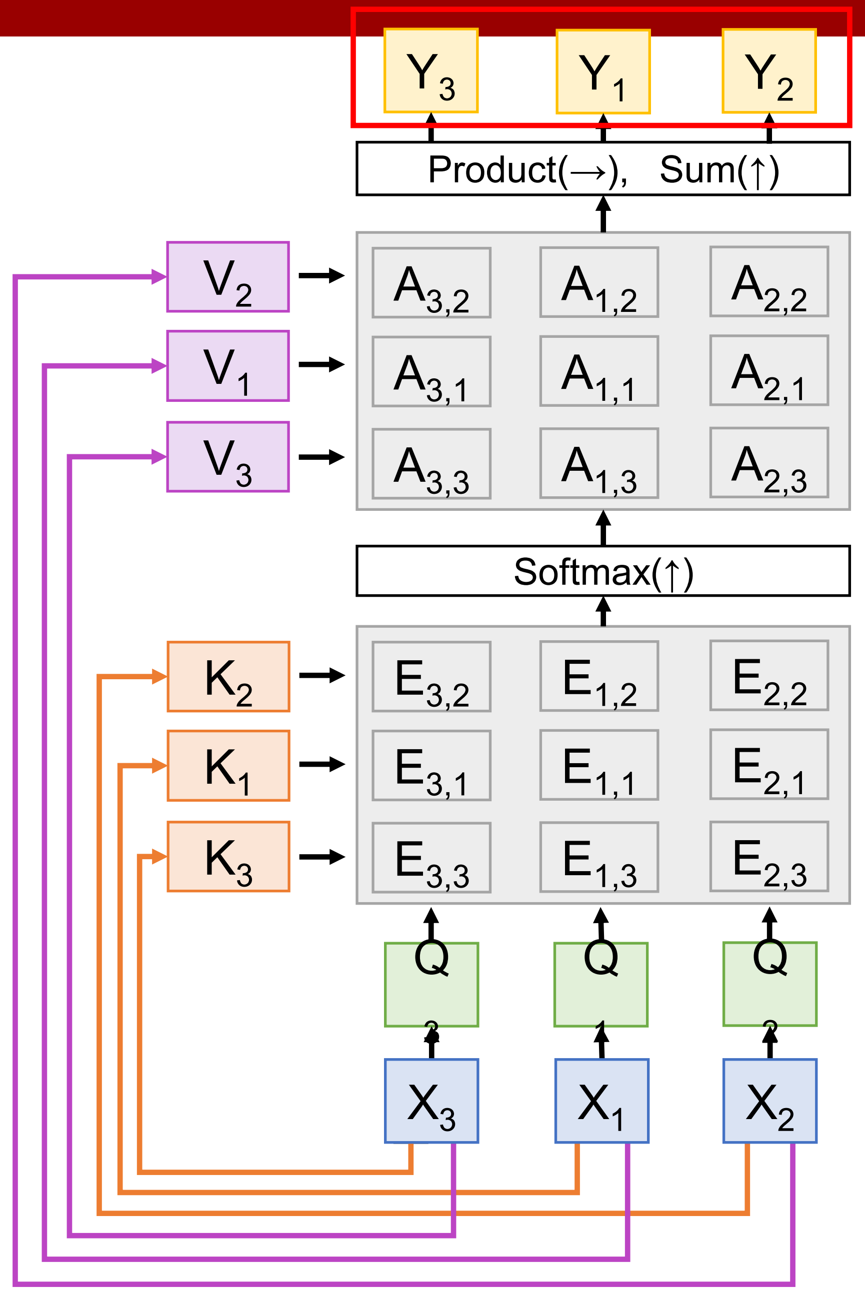
Inputs:
Input vectors: **X** (Shape: N_X x D_X)
Key matrix: **W_K** (Shape: D_X x D_Q)
Value matrix: **W_V** (Shape: D_X x D_V)
Query matrix: **W_Q** (Shape: D_X x D_Q)

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) $E_{i,j} = Q_i \cdot K_j / \text{sqrt}(D_Q)$
Attention weights: A = softmax(E, dim=1) (Shape: N_X x N_X)
Output vectors: Y = A**V** (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))



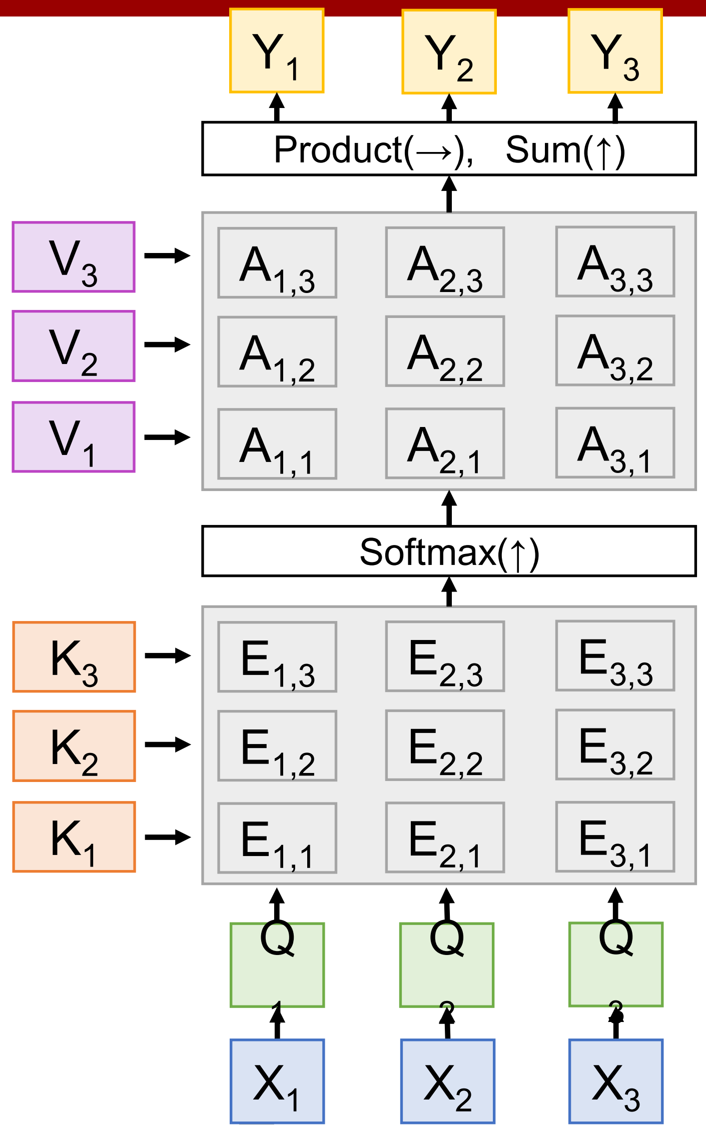
Slide credit: Justin Johnson

Self-Attention Layer

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

Self attention doesn't "know" the order of the vectors it is processing!

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



Slide credit: Justin Johnson

Self-Attention Layer

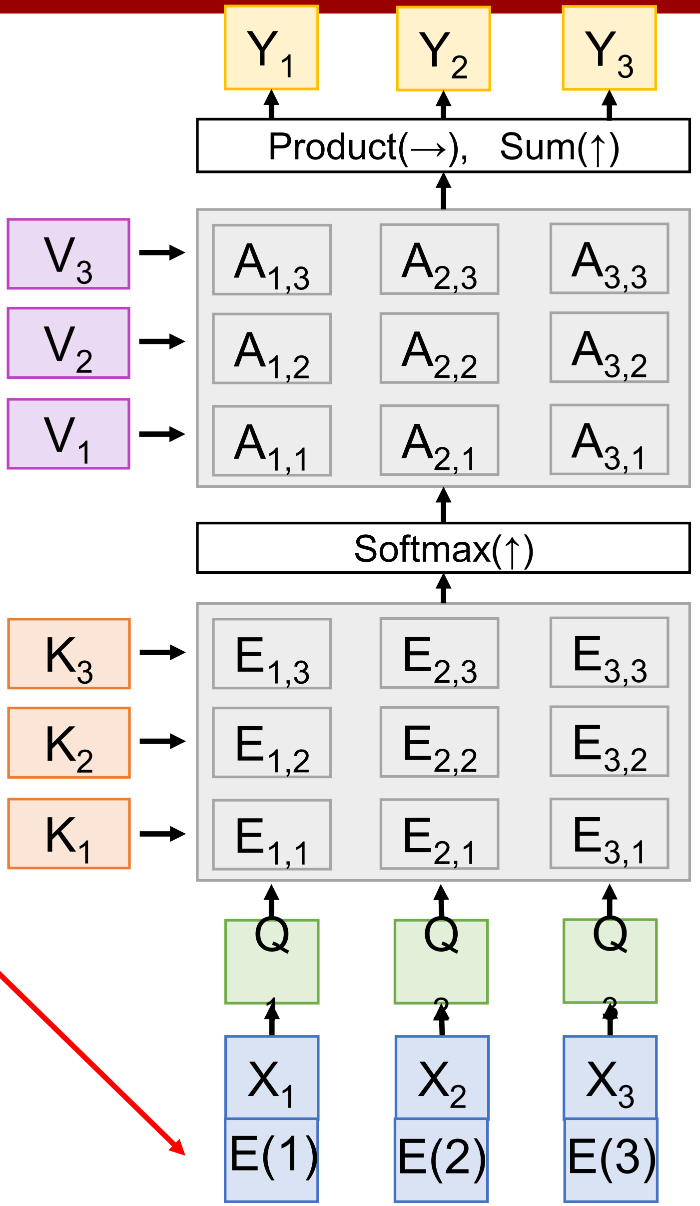
Inputs:
Input vectors: **X** (Shape: $N_X \times D_X$)
Key matrix: **W_K** (Shape: $D_X \times D_Q$)
Value matrix: **W_V** (Shape: $D_X \times D_V$)
Query matrix: **W_Q** (Shape: $D_X \times D_Q$)

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 = softmax(E, dim=1) (Shape: $N_X \times N_X$)
Output vectors: $Y = AV$ (Shape: $N_X \times D_V$) $Y_i = \sum_j A_{i,j} V_j$

Self attention doesn't "know" the order of the vectors it is processing!

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

E can be learned lookup table, or fixed function



Slide credit: Justin Johnson

Summary

- We have made a generic sequence-in to sequence-out layer
  - This is what we want for language processing!
  - Each output is a contextualized representation of the corresponding input word
  - Vector for stop word can be treated as representation of entire sentence (e.g. project its output to classifier and add loss)

- Unlike RNNs/LSTMs, it processes all inputs (e.g. entire sentence) **at once**
  - **Highly parallelizable**
  - **-> SCALE! -> Reduction of loss -> Magic**

- Next time: Entire transformer architecture that combines this new layer with other layers/concepts we know about (fully-connected, normalization, residual/skip connections)

Paper Discussion

Recurrent Neural Networks for Multivariate Time Series with Missing Values

Zhengping Che[1], Sanjay Purushotham[1], Kyunghyun Cho[2], David Sontag[3] & Yan Liu[1]

Multivariate time series data in practical applications, such as health care, geoscience, and biology, are characterized by a variety of missing values. In time series prediction and other related tasks, it has been noted that missing values and their missing patterns are often correlated with the target labels, a.k.a., *informative* missingness. There is very limited work on exploiting the missing patterns for effective imputation and improving prediction performance. In this paper, we develop novel deep learning models, namely GRU-D, as one of the early attempts. GRU-D is based on Gated Recurrent Unit (GRU), a state-of-the-art recurrent neural network. It takes two representations of missing patterns, i.e., *masking* and *time interval*, and effectively incorporates them into a deep model architecture so that it not only captures the long-term temporal dependencies in time series, but also utilizes the missing patterns to achieve better prediction results. Experiments of time series classification tasks on real-world clinical datasets (MIMIC-III, PhysioNet) and synthetic datasets demonstrate that our models achieve state-of-the-art performance and provide useful insights for better understanding and utilization of missing values in time series analysis.

Problem Statement

- What problem does this paper focus on?
  – Is this new or already explored?
  – Is this important?
  – What key applications this is relevant for?
  – What assumptions does this paper make about

Recurrent Neural Networks for Multivariate Time Series with Missing Values

Zhengping Che[1], Sanjay Purushotham[1], Kyunghyun Cho[2], David Sontag[3] & Yan Liu[1]

Multivariate time series data in practical applications, such as health care, geoscience, and biology, are characterized by a variety of missing values. In time series prediction and other related tasks, it has been noted that missing values and their missing patterns are often correlated with the target labels, a.k.a., *informative* missingness. There is very limited work on exploiting the missing patterns for effective imputation and improving prediction performance. In this paper, we develop novel deep learning models, namely GRU-D, as one of the early attempts. GRU-D is based on Gated Recurrent Unit (GRU), a state-of-the-art recurrent neural network. It takes two representations of missing patterns, i.e., *masking* and *time interval*, and effectively incorporates them into a deep model architecture so that it not only captures the long-term temporal dependencies in time series, but also utilizes the missing patterns to achieve better prediction results. Experiments of time series classification tasks on real-world clinical datasets (MIMIC-III, PhysioNet) and synthetic datasets demonstrate that our models achieve state-of-the-art performance and provide useful insights for better understanding and utilization of missing values in time series analysis.

Key idea – Informative missingness

Nice!

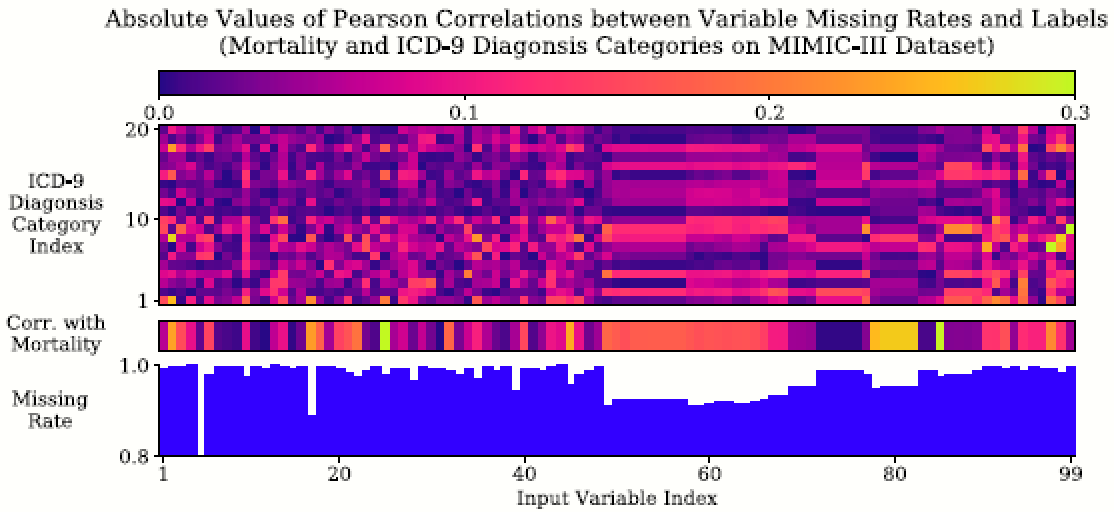


Figure 1. Demonstration of informative missingness on MIMIC-III dataset. The bottom figure shows the missing rate of each input variable. The middle figure shows the absolute values of Pearson correlation coefficients between missing rate of each variable and mortality. The top figure shows the absolute values of Pearson correlation coefficients between missing rate of each variable and each ICD-9 diagnosis category. More details can be found in supplementary information Section S1.

Related Work / Situation of Work

- What prior approaches exist to solve this problem?

In the past decades, various approaches have been developed to address missing values in time series[3]. A simple solution is to omit the missing data and to perform analysis only on the observed data, but it does not provide good performance when the missing rate is high and inadequate samples are kept. Another solution is to fill in the missing values with substituted values, which is known as data imputation. Smoothing, interpolation[4], and spline[5] methods are simple and efficient, thus widely applied in practice. However, these methods do not capture variable correlations and may not capture complex pattern to perform imputation. A variety of imputation methods have been developed to better estimate missing data. These include spectral analysis[6], kernel methods[7], EM algorithm[8], matrix completion[9] and matrix factorization[10]. Multiple imputation[11,12] can be further applied with these imputation methods to reduce the uncertainty, by repeating the imputation procedure multiple times and averaging the results. Combining the imputation methods with prediction models often results in a two-step process where imputation and prediction models are separated. By doing this, the missing patterns are not effectively explored in the prediction model, thus leading to suboptimal analyses results[13]. In addition, most imputation methods also have other requirements which may not be satisfied in real applications, for example, many of them work on data

Approach and Key Nugget

- What approach does this paper take?
- What is the key “golden nugget” – intuition, idea, etc. that leads to approach
  – Lower-level
  – Higher-level?

Paper Discussion

Recurrent Neural Networks for Multivariate Time Series with Missing Values

Zhengping Che[1], Sanjay Purushotham[1], Kyunghyun Cho[2], David Sontag[3] & Yan Liu[1]

Multivariate time series data in practical applications, such as health care, geoscience, and biology, are characterized by a variety of missing values. In time series prediction and other related tasks, it has been noted that missing values and their missing patterns are often correlated with the target labels, a.k.a., *informative* missingness. There is very limited work on exploiting the missing patterns for effective imputation and improving prediction performance. In this paper, we develop novel deep learning models, namely GRU-D, as one of the early attempts. GRU-D is based on Gated Recurrent Unit (GRU), a state-of-the-art recurrent neural network. It takes two representations of missing patterns, i.e., *masking* and *time interval*, and effectively incorporates them into a deep model architecture so that it not only captures the long-term temporal dependencies in time series, but also utilizes the missing patterns to achieve better prediction results. Experiments of time series classification tasks on real-world clinical datasets (MIMIC-III, PhysioNet) and synthetic datasets demonstrate that our models achieve state-of-the-art performance and provide useful insights for better understanding and utilization of missing values in time series analysis.

Method – masking & Time Interval

Methods

Notations. We denote a multivariate time series with D variables of length T as $X = (x_1, x_2, \ldots, x_T)^{\mathrm{T}} \in \mathbb{R}^{T \times D}$, where for each $t \in \{1, 2, \ldots, T\}$, $x_t \in \mathbb{R}^D$ represents the t-th observations (a.k.a., measurements) of all variables and x_t^d denotes the measurement of d-th variable of x_t. Let $s_t \in \mathbb{R}$ denote the time-stamp when the tth observation is obtained and we assume that the first observation is made at time-stamp 0 (i.e., $s_1 = 0$). A time series X could have missing values. We introduce a *masking vector* $m_t \in \{0, 1\}^D$ to denote which variables are missing at time step t, and also maintain the *time interval* $\delta_t^d \in \mathbb{R}$ for each variable d since its last observation. To be more specific, we have

$$m_t^d = \begin{cases} 1, & \text{if } x_t^d \text{ is observed} \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$\delta_t^d = \begin{cases} s_t - s_{t-1} + \delta_{t-1}^d, & t > 1,\, m_{t-1}^d = 0 \\ s_t - s_{t-1}, & t > 1,\, m_{t-1}^d = 1 \\ 0, & t = 1 \end{cases} \tag{2}$$

Three Methods for Imputing

$$x_t^d \leftarrow m_t^d x_t^d + (1 - m_t^d)\tilde{x}^d \tag{7}$$

where $\tilde{x}^d = \sum_{n=1}^{N}\sum_{t=1}^{T_n} m_{t,n}^d x_{t,n}^d / \sum_{n=1}^{N}\sum_{t=1}^{T_n} m_{t,n}^d$. $\tilde{x}^d$ is calculated on the training dataset and used for both training and testing datasets. We refer to this approach as **GRU-Mean**.

A second approach is to exploit the temporal structure. For example, we may assume any missing value is the same as its last measurement and use forward imputation (**GRU-Forward**), i.e.,

$$x_t^d \leftarrow m_t^d x_t^d + (1 - m_t^d)x_{t'}^d \tag{8}$$

where $t' < t$ is the last time the d-th variable was observed.

Instead of explicitly imputing missing values, the third approach simply indicates which variables are missing and how long they have been missing as a part of input by concatenating the measurement, masking and time interval vectors as

$$x_t^{(n)} \leftarrow [x_t^{(n)}; m_t^{(n)}; \delta_t^{(n)}] \tag{9}$$

where $x_t^{(n)}$ can be either from Equation (7) or (8). We later refer to this approach as **GRU-Simple**.

GRU

The structure of GRU is shown in Fig. 3(a). For each j-th hidden unit, GRU has a reset gate r_t^j and an update gate z_t^j to control the hidden state h_t^j at each time t. The update functions are as follows:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{3}$$

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{4}$$

$$\tilde{h}_t = \tanh(W x_t + U(r_t \odot h_{t-1}) + b) \tag{5}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{6}$$

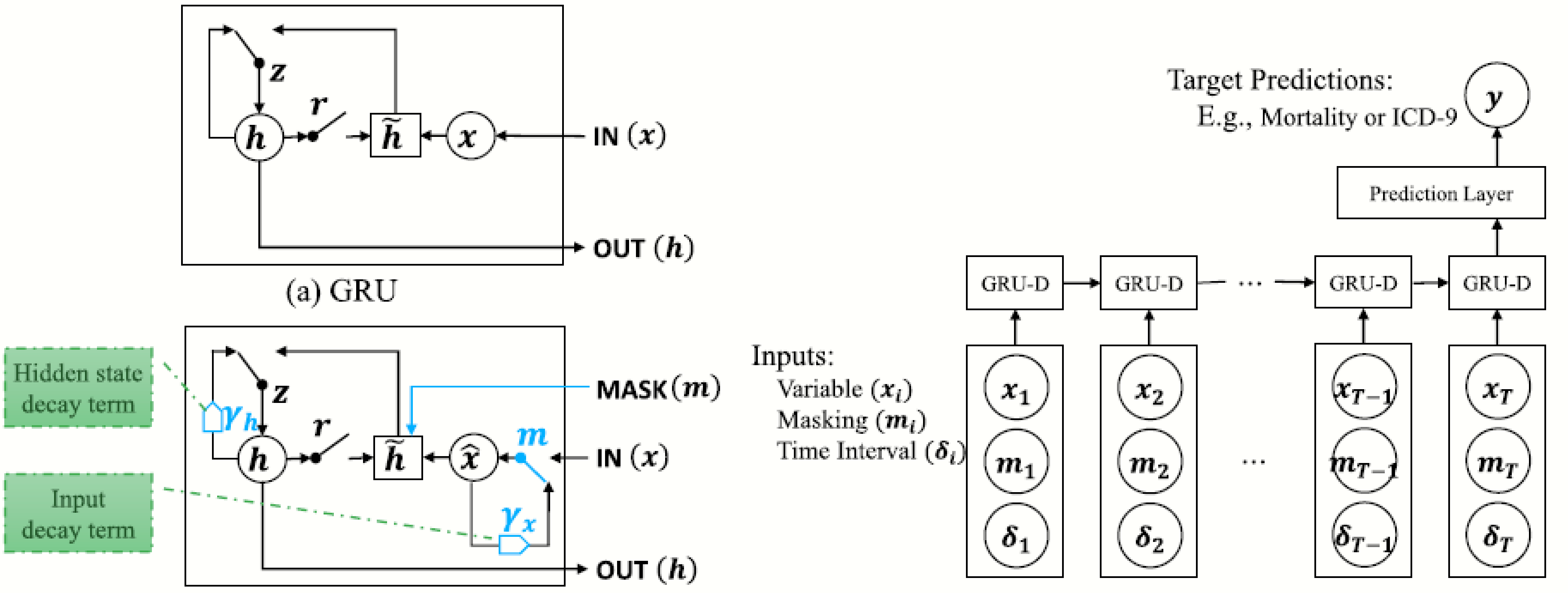
GRU-D

we aim at learning decay rates from the training data rather than fixed a priori. That is, we model a vector of decay rates γ as

$$\gamma_t = \exp\{-\max(0, W_\gamma \delta_t + b_\gamma)\} \tag{10}$$

$$\hat{x}_t^d = m_t^d x_t^d + (1 - m_t^d)\left(\gamma_{x_t}^d x_{t'}^d + (1 - \gamma_{x_t}^d)\tilde{x}^d\right) \tag{11}$$

where $x_{t'}^d$ is the last observation of the d-th variable ($t' < t$) and $\tilde{x}^d$ is the empirical mean of the d-th variable. When decaying the input variable directly, we constrain W_{γ_x} to be diagonal, which effectively makes the decay rate of each variable independent from the others.



(a) GRU

(b) GRU-D (Parts in cyan refer to the modifications.)

(c) Proposed prediction model architecture with GRU-D.

Figure 3. Graphical illustrations of the original GRU (top-left), the proposed GRU-D (bottom-left), and the whole network architecture (right).

Validation

- How do they validate their approach?
  - What data do they use?
  - What baselines do they compare against?

Paper Discussion

Recurrent Neural Networks for Multivariate Time Series with Missing Values

Zhengping Che[1], Sanjay Purushotham[1], Kyunghyun Cho[2], David Sontag[3] & Yan Liu[1]

Multivariate time series data in practical applications, such as health care, geoscience, and biology, are characterized by a variety of missing values. In time series prediction and other related tasks, it has been noted that missing values and their missing patterns are often correlated with the target labels, a.k.a., *informative* missingness. There is very limited work on exploiting the missing patterns for effective imputation and improving prediction performance. In this paper, we develop novel deep learning models, namely GRU-D, as one of the early attempts. GRU-D is based on Gated Recurrent Unit (GRU), a state-of-the-art recurrent neural network. It takes two representations of missing patterns, i.e., *masking* and *time interval*, and effectively incorporates them into a deep model architecture so that it not only captures the long-term temporal dependencies in time series, but also utilizes the missing patterns to achieve better prediction results. Experiments of time series classification tasks on real-world clinical datasets (MIMIC-III, PhysioNet) and synthetic datasets demonstrate that our models achieve state-of-the-art performance and provide useful insights for better understanding and utilization of missing values in time series analysis.

Models	**ICD-9 20 Tasks on MIMIC-III Dataset**	**All 4 Tasks on PhysioNet Dataset**
GRU-Mean	0.7070 ± 0.001	0.8099 ± 0.011
GRU-Forward	0.7077 ± 0.001	0.8091 ± 0.008
GRU-Simple	0.7105 ± 0.001	0.8249 ± 0.010
GRU-CubicSpline	0.6372 ± 0.005	0.7451 ± 0.011
GRU-MICE	0.6717 ± 0.005	0.7955 ± 0.003
GRU-MF	0.6805 ± 0.004	0.7727 ± 0.003
GRU-PCA	0.7040 ± 0.002	0.8042 ± 0.006
GRU-MissForest	0.7115 ± 0.003	0.8076 ± 0.009
Proposed GRU-D	$\mathbf{0.7123 \pm 0.003}$	$\mathbf{0.8370 \pm 0.012}$

Table 2. Model performances measured by average AUC score ($mean \pm std$) for multi-task predictions on real datasets.

Non-RNN Models						RNN Models	
Mortality Prediction On MIMIC-III Dataset						LSTM-Mean	0.8142 ± 0.014
LR-Mean	0.7589 ± 0.015	SVM-Mean	0.7908 ± 0.006	RF-Mean	0.8293 ± 0.004	GRU-Mean	0.8252 ± 0.011
LR-Forward	0.7792 ± 0.018	SVM-Forward	0.8010 ± 0.004	RF-Forward	0.8303 ± 0.003	GRU-Forward	0.8192 ± 0.013
LR-Simple	0.7715 ± 0.015	SVM-Simple	0.8146 ± 0.008	RF-Simple	0.8294 ± 0.007	GRU-Simple w/o δ[22]	0.8367 ± 0.009
LR-SoftImpute	0.7598 ± 0.017	SVM-SoftImpute	0.7540 ± 0.012	RF-SoftImpute	0.7855 ± 0.011	GRU-Simple w/o m[23,24]	0.8266 ± 0.009
LR-KNN	0.6877 ± 0.011	SVM-KNN	0.7200 ± 0.004	RF-KNN	0.7135 ± 0.015	GRU-Simple	0.8380 ± 0.008
LR-CubicSpline	0.7270 ± 0.005	SVM-CubicSpline	0.6376 ± 0.018	RF-CubicSpline	0.8339 ± 0.007	GRU-CubicSpline	0.8180 ± 0.011
LR-MICE	0.6965 ± 0.019	SVM-MICE	0.7169 ± 0.012	RF-MICE	0.7159 ± 0.005	GRU-MICE	0.7527 ± 0.015
LR-MF	0.7158 ± 0.018	SVM-MF	0.7266 ± 0.017	RF-MF	0.7234 ± 0.011	GRU-MF	0.7843 ± 0.012
LR-PCA	0.7246 ± 0.014	SVM-PCA	0.7235 ± 0.012	RF-PCA	0.7747 ± 0.009	GRU-PCA	0.8236 ± 0.007
LR-MissForest	0.7279 ± 0.016	SVM-MissForest	0.7482 ± 0.016	RF-MissForest	0.7858 ± 0.010	GRU-MissForest	0.8239 ± 0.006
						Proposed GRU-D	$\mathbf{0.8527 \pm 0.003}$
Mortality Prediction On PhysioNet Dataset						LSTM-Mean	0.8025 ± 0.013
LR-Mean	0.7423 ± 0.011	SVM-Mean	0.8131 ± 0.018	RF-Mean	0.8183 ± 0.015	GRU-Mean	0.8162 ± 0.014
LR-Forward	0.7479 ± 0.012	SVM-Forward	0.8140 ± 0.018	RF-Forward	0.8219 ± 0.017	GRU-Forward	0.8195 ± 0.004
LR-Simple	0.7625 ± 0.004	SVM-Simple	0.8277 ± 0.012	RF-Simple	0.8157 ± 0.014	GRU-Simple	0.8226 ± 0.010
LR-SoftImpute	0.7386 ± 0.007	SVM-SoftImpute	0.8057 ± 0.019	RF-SoftImpute	0.8100 ± 0.016	GRU-SoftImpute	0.8125 ± 0.005
LR-KNN	0.7146 ± 0.011	SVM-KNN	0.7644 ± 0.018	RF-KNN	0.7567 ± 0.012	GRU-KNN	0.8155 ± 0.004
LR-CubicSpline	0.6913 ± 0.022	SVM-CubicSpline	0.6364 ± 0.015	RF-CubicSpline	0.8151 ± 0.015	GRU-CubicSpline	0.7596 ± 0.020
LR-MICE	0.6828 ± 0.015	SVM-MICE	0.7690 ± 0.016	RF-MICE	0.7618 ± 0.007	GRU-MICE	0.8153 ± 0.013
LR-MF	0.6513 ± 0.014	SVM-MF	0.7515 ± 0.022	RF-MF	0.7355 ± 0.022	GRU-MF	0.7904 ± 0.012
LR-PCA	0.6890 ± 0.019	SVM-PCA	0.7741 ± 0.014	RF-PCA	0.7561 ± 0.025	GRU-PCA	0.8116 ± 0.007
LR-MissForest	0.7010 ± 0.018	SVM-MissForest	0.7779 ± 0.008	RF-MissForest	0.7890 ± 0.016	GRU-MissForest	0.8244 ± 0.012
						Proposed GRU-D	$\mathbf{0.8424 \pm 0.012}$

Table 1. Model performances measured by AUC score ($mean \pm std$) for mortality prediction.

Strengths / Weaknesses

- Strengths?
- Weakness / Limitations?