Attention and Transformers

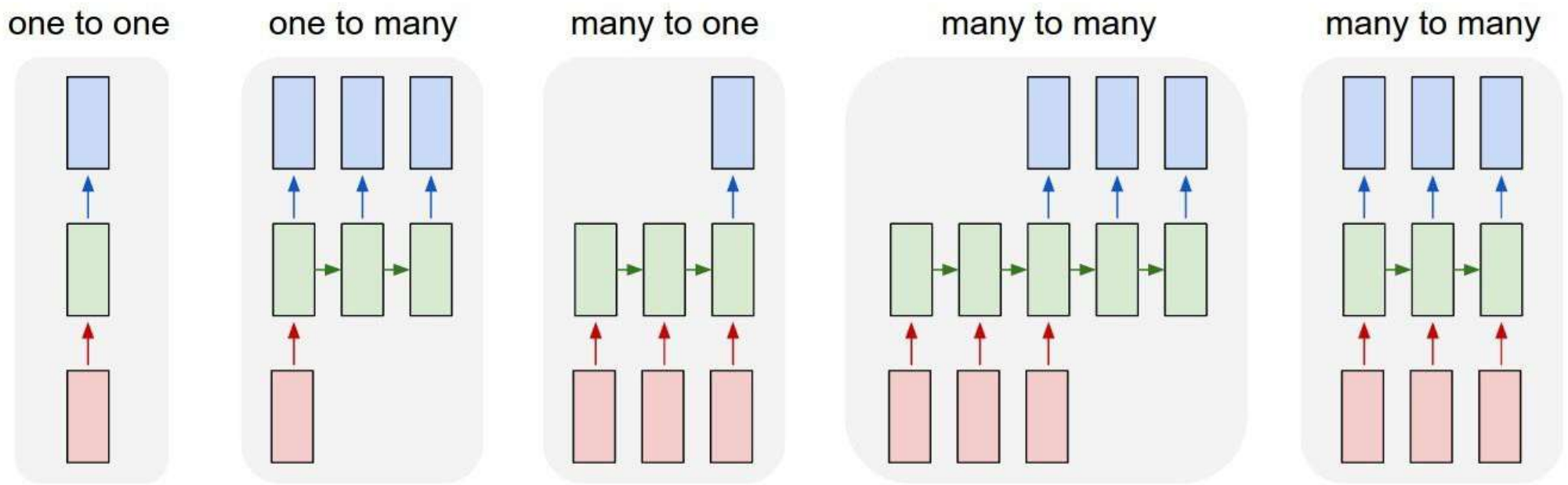
Prof. Jianguo Zhang
SUSTech

Slides were adapted from various sources

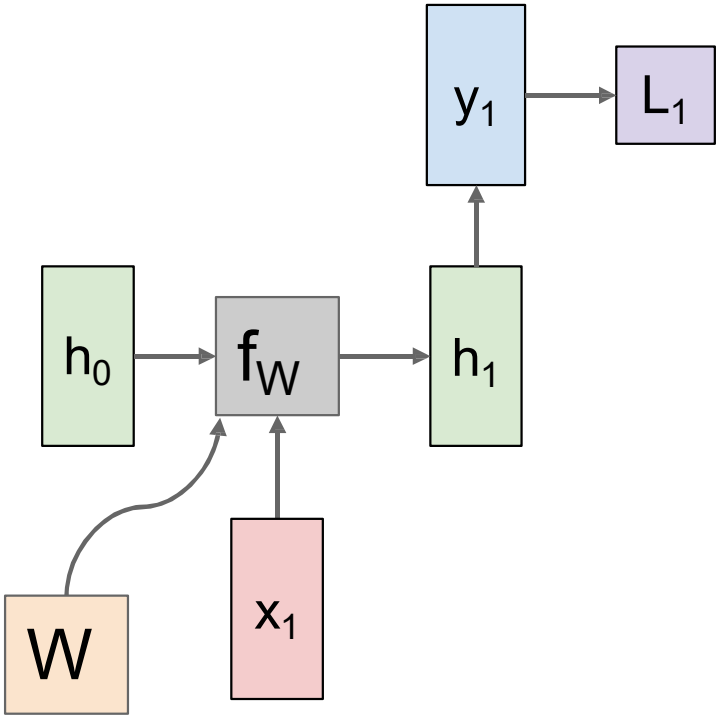
1. From RNN to Attention Layers

Prof. Jianguo Zhang
SUSTech

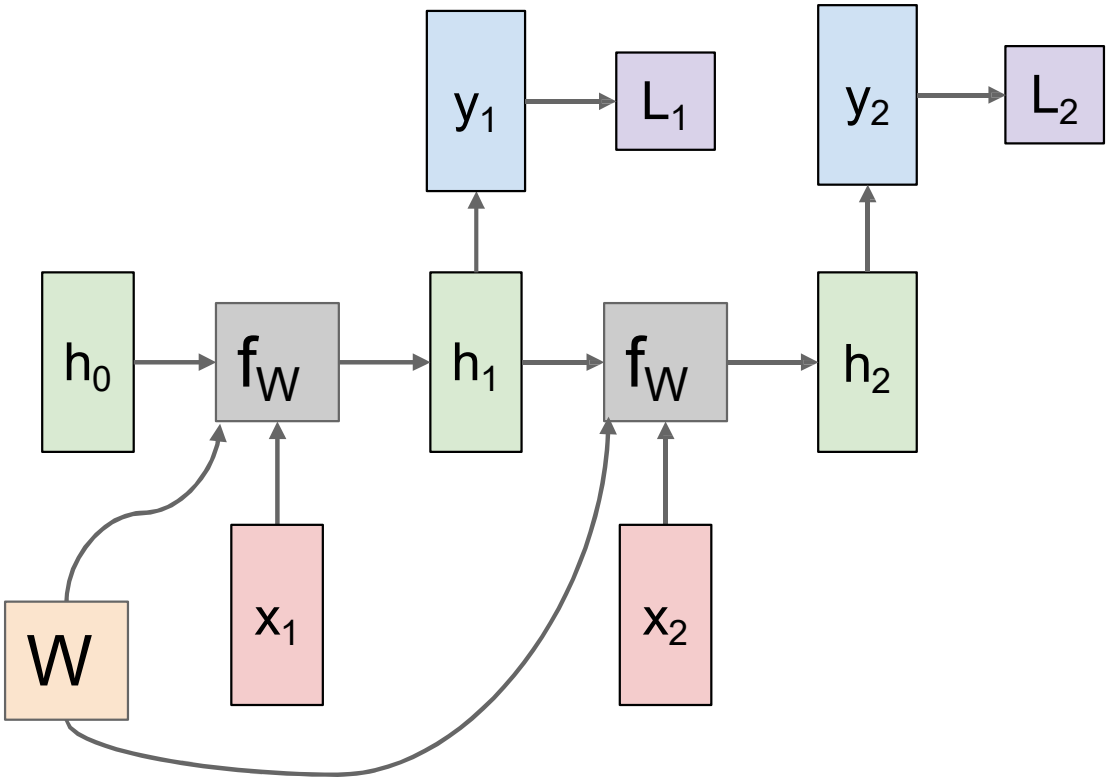
Last Time: Recurrent Neural Networks



Last Time: Variable length computation graph with shared weights

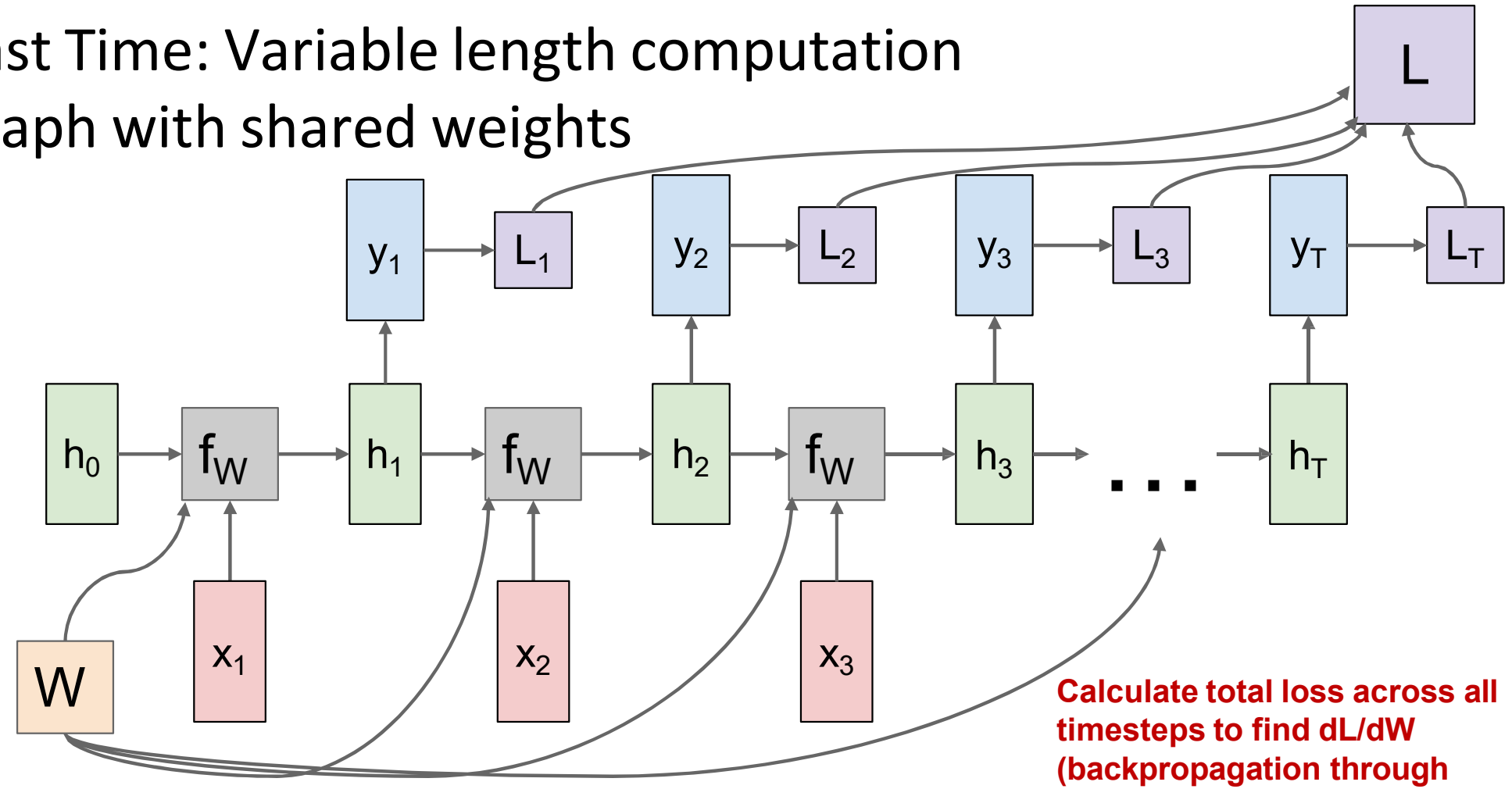


Last Time: Variable length computation graph with shared weights



W is reused (recurrently)!

Last Time: Variable length computation graph with shared weights



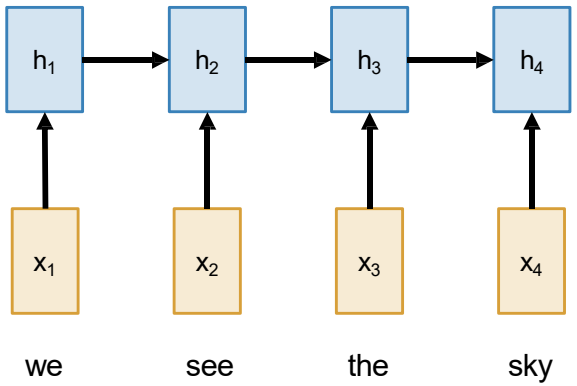
Calculate total loss across all timesteps to find dL/dW (backpropagation through time)!

Sequence to Sequence with RNNs: Encoder - Decoder

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

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

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



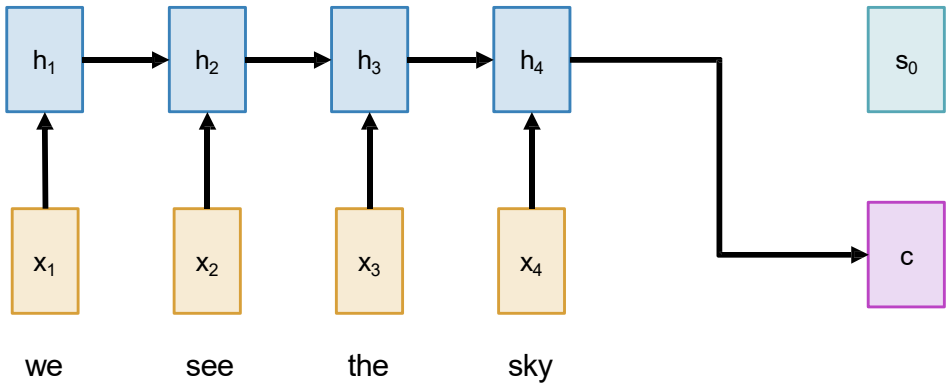
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs

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

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

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



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

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

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

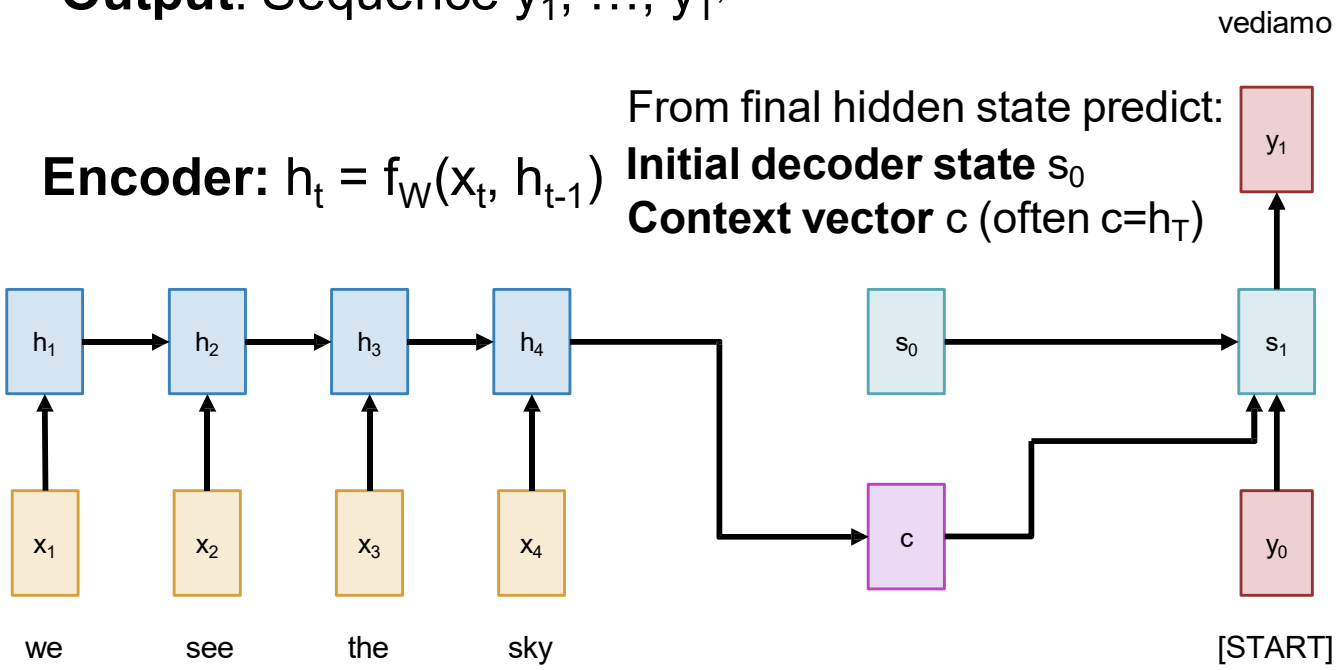
Encoder: $h_t = f_W(x_t, h_{t-1})$

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

From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)



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

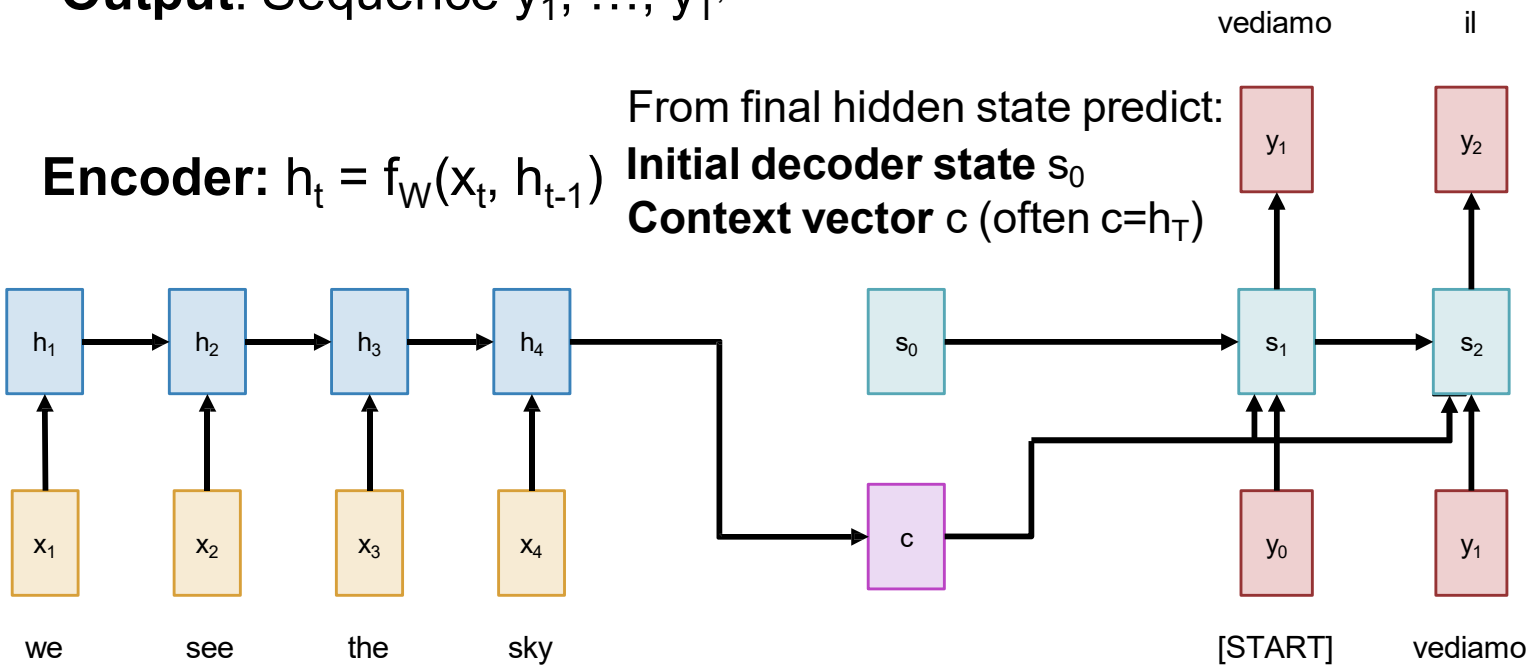
Sequence to Sequence with RNNs

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

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

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

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



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

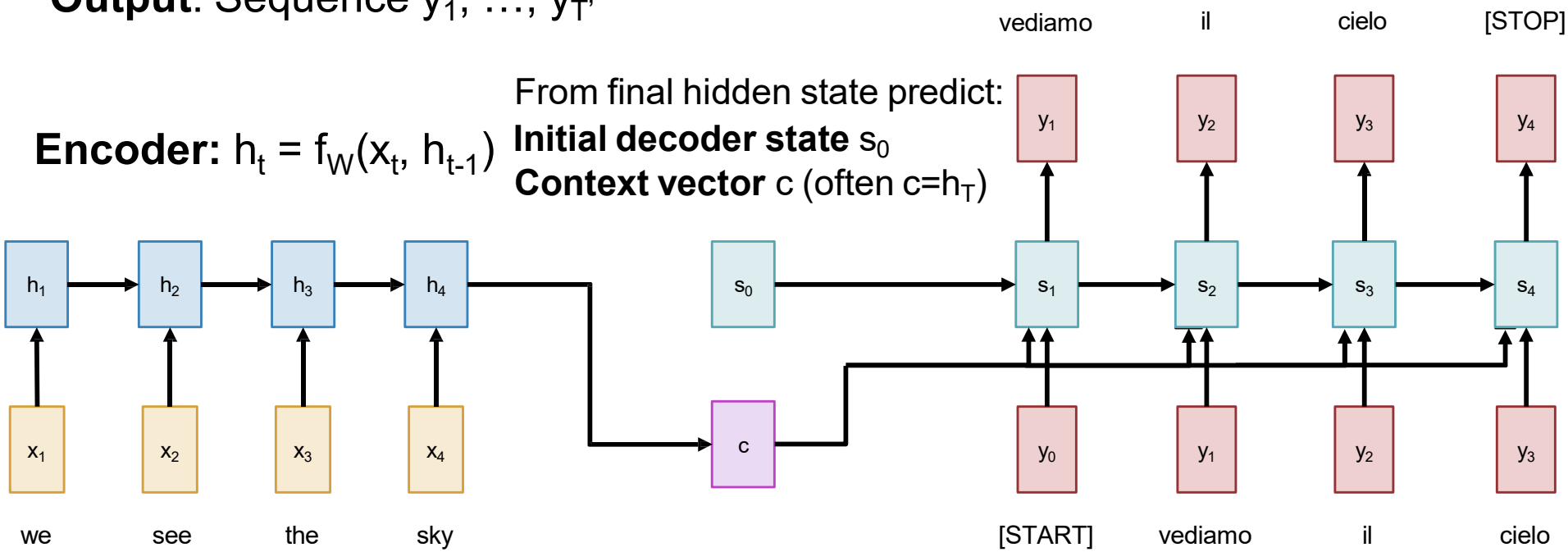
Sequence to Sequence with RNNs

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

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

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

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



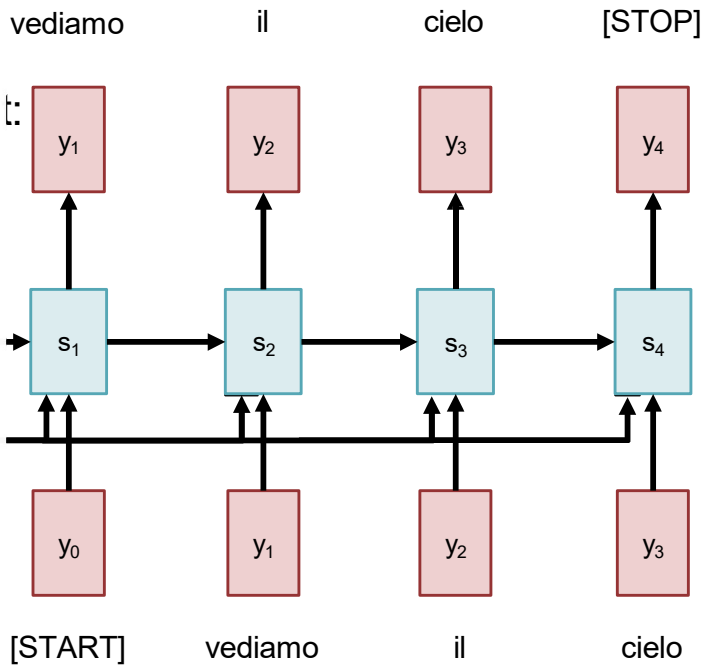
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Remember:

During Training:
Often, we use the "correct" token even if the model is wrong. Called **teacher forcing**

During Test-time:
We sample from the model's outputs until we sample [STOP]

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



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

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

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

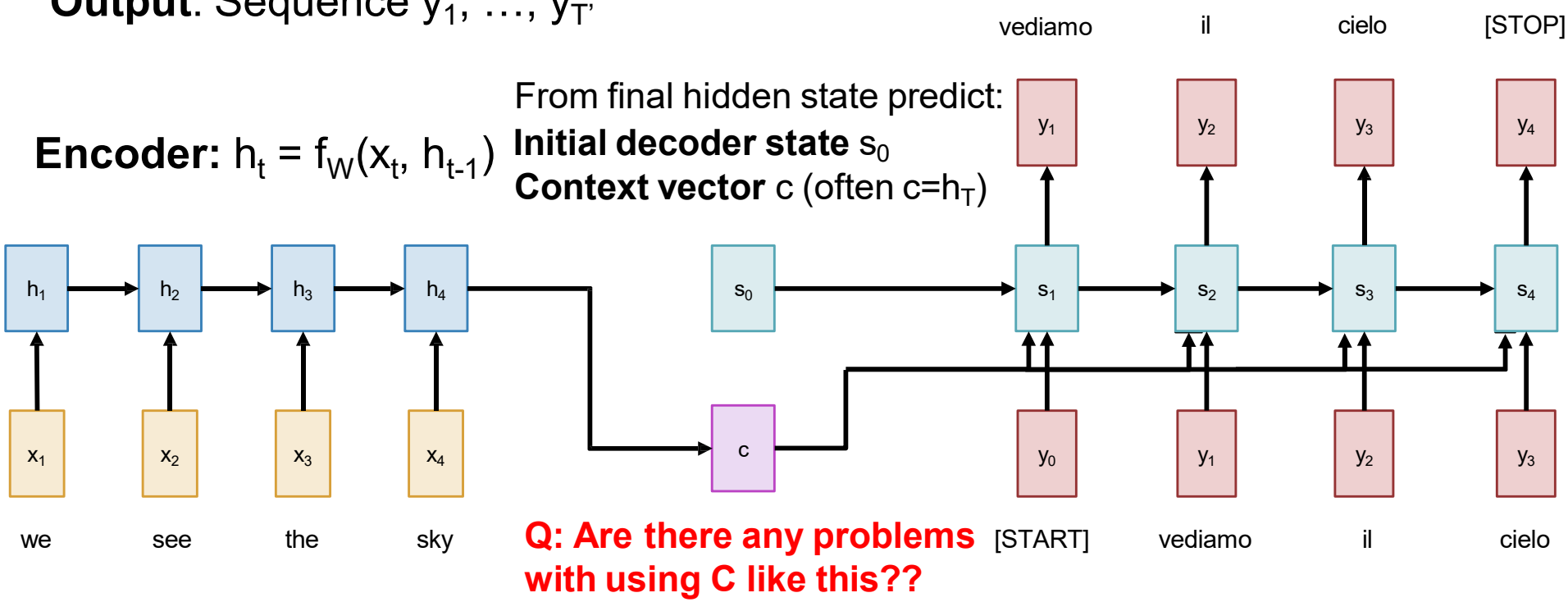
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

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

From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)



Q: Are there any problems with using C like this??

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

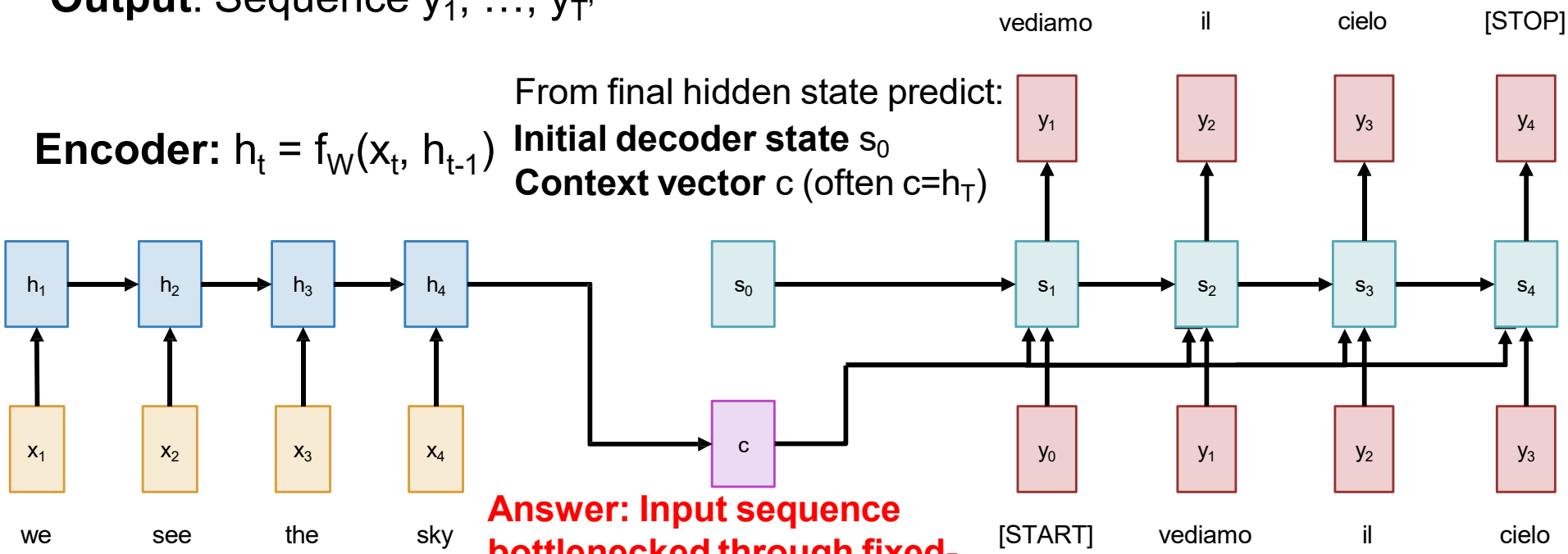
Sequence to Sequence with RNNs

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

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

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

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



Answer: Input sequence bottlenecked through fixed-sized vector. What if T=1000?

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

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

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

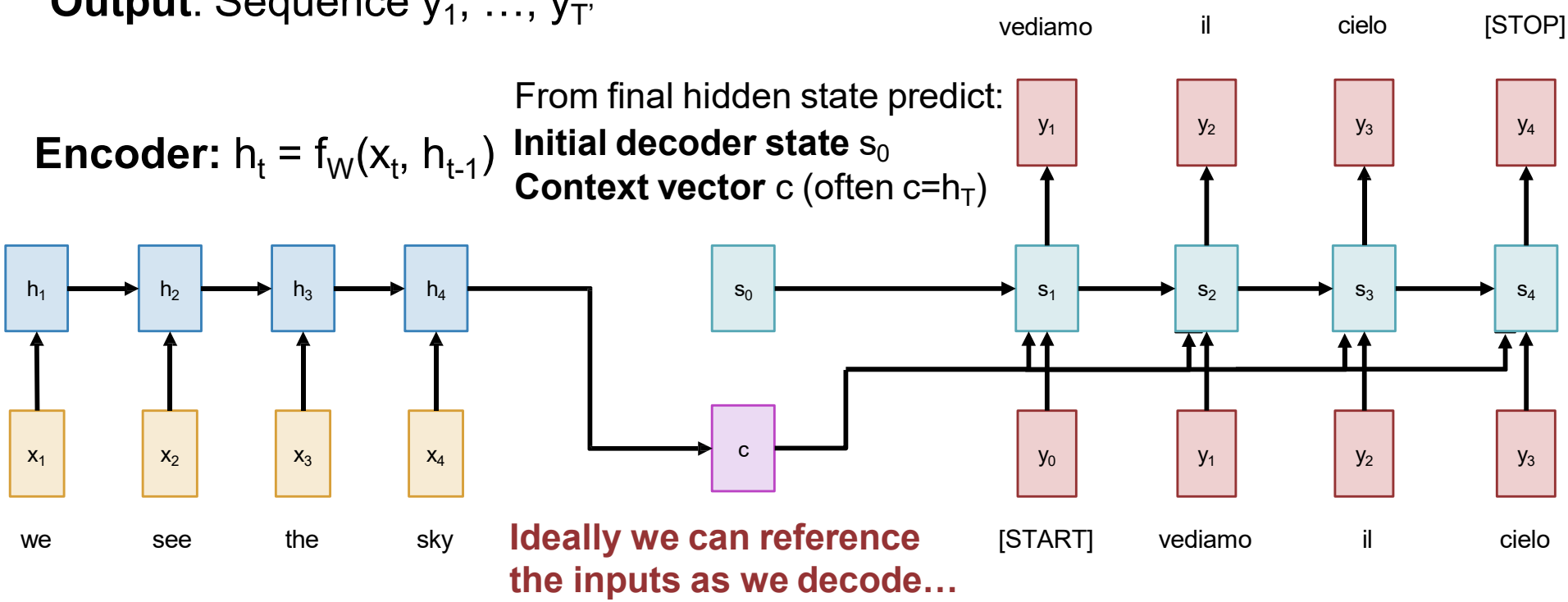
Decoder: $s_t = g_U(y_{t-1}, s_{t-1}, c)$

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

From final hidden state predict:

Initial decoder state s_0

Context vector c (often $c=h_T$)



Ideally we can reference the inputs as we decode…

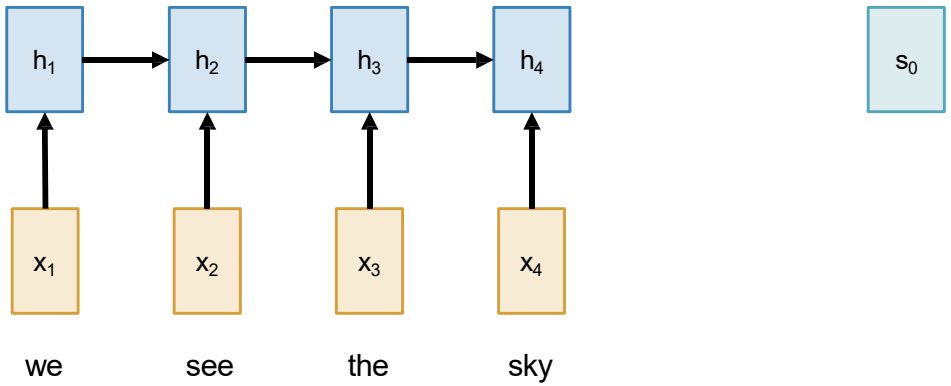
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs and Attention

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

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

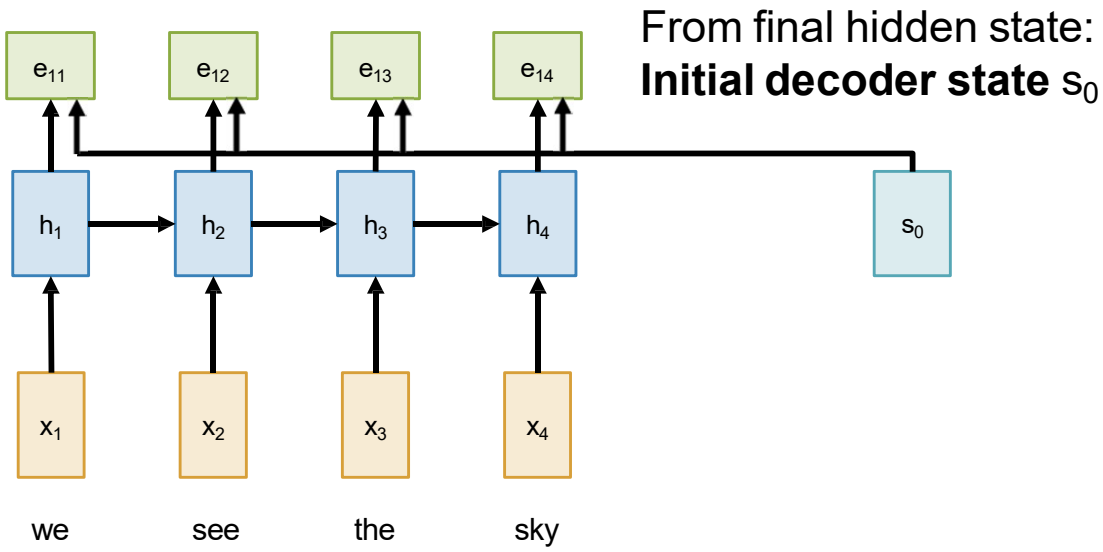
From final hidden state:
Initial decoder state s_0



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

Sequence to Sequence with RNNs and Attention

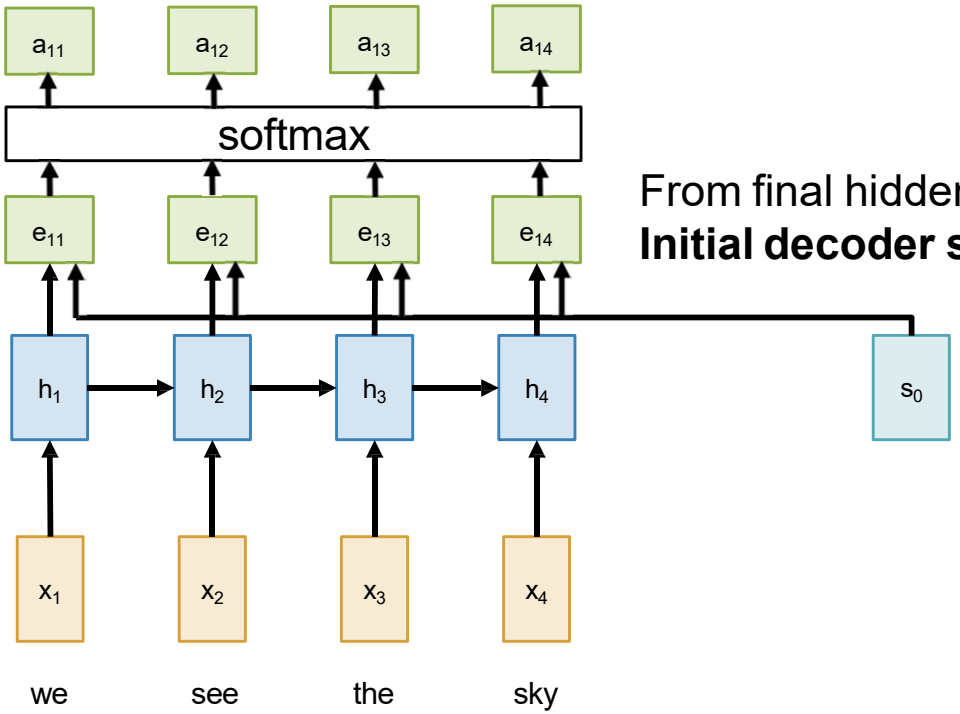
Compute (scalar) **alignment scores**
$e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is a Linear Layer)



From final hidden state:
Initial decoder state s_0

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

Sequence to Sequence with RNNs and Attention



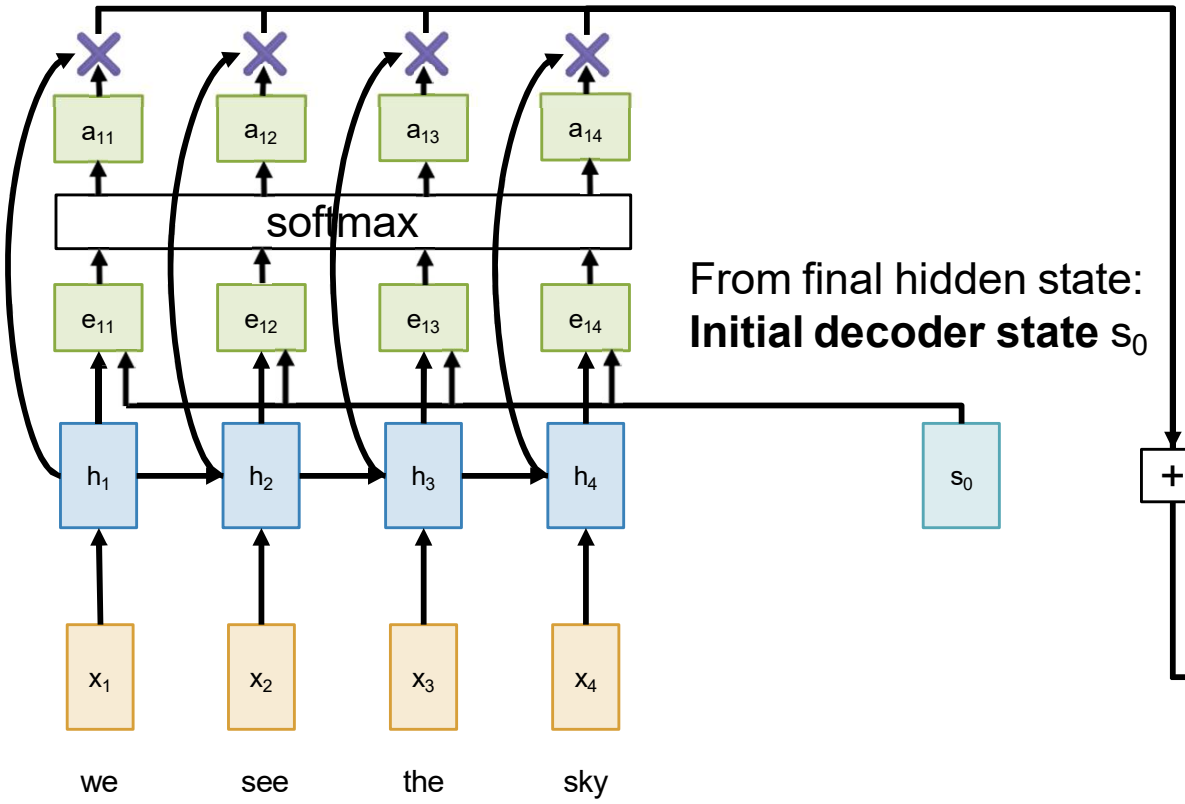
From final hidden state:
Initial decoder state s_0

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

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

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

Sequence to Sequence with RNNs and Attention



Compute (scalar) **alignment scores**

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

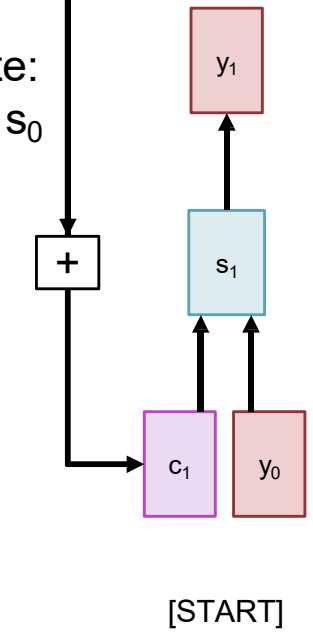
vediamo

Normalize alignment scores to get **attention weights**

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

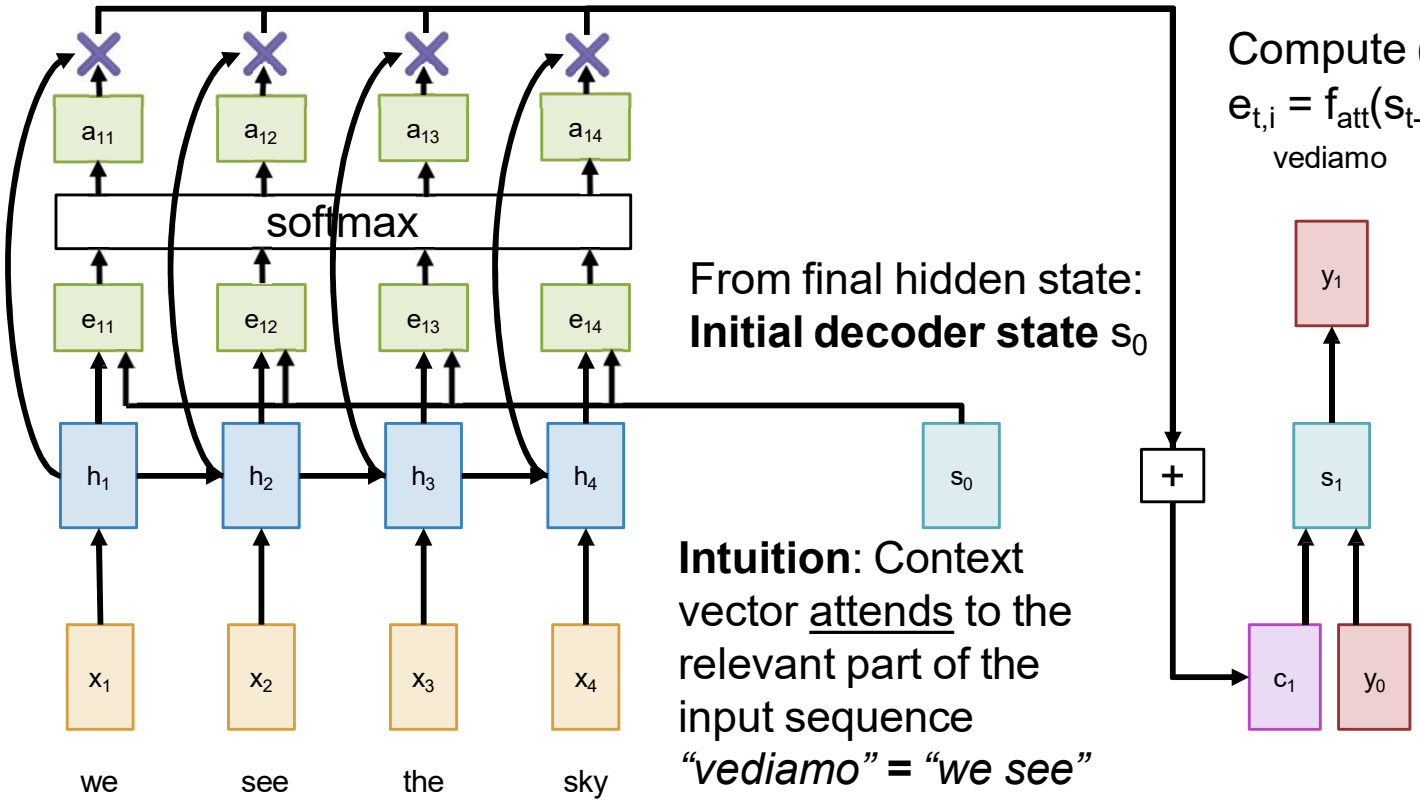
Compute context vector as **weighted sum of hidden states**

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



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

Sequence to Sequence with RNNs and Attention



From final hidden state:
Initial decoder state s_0

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

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

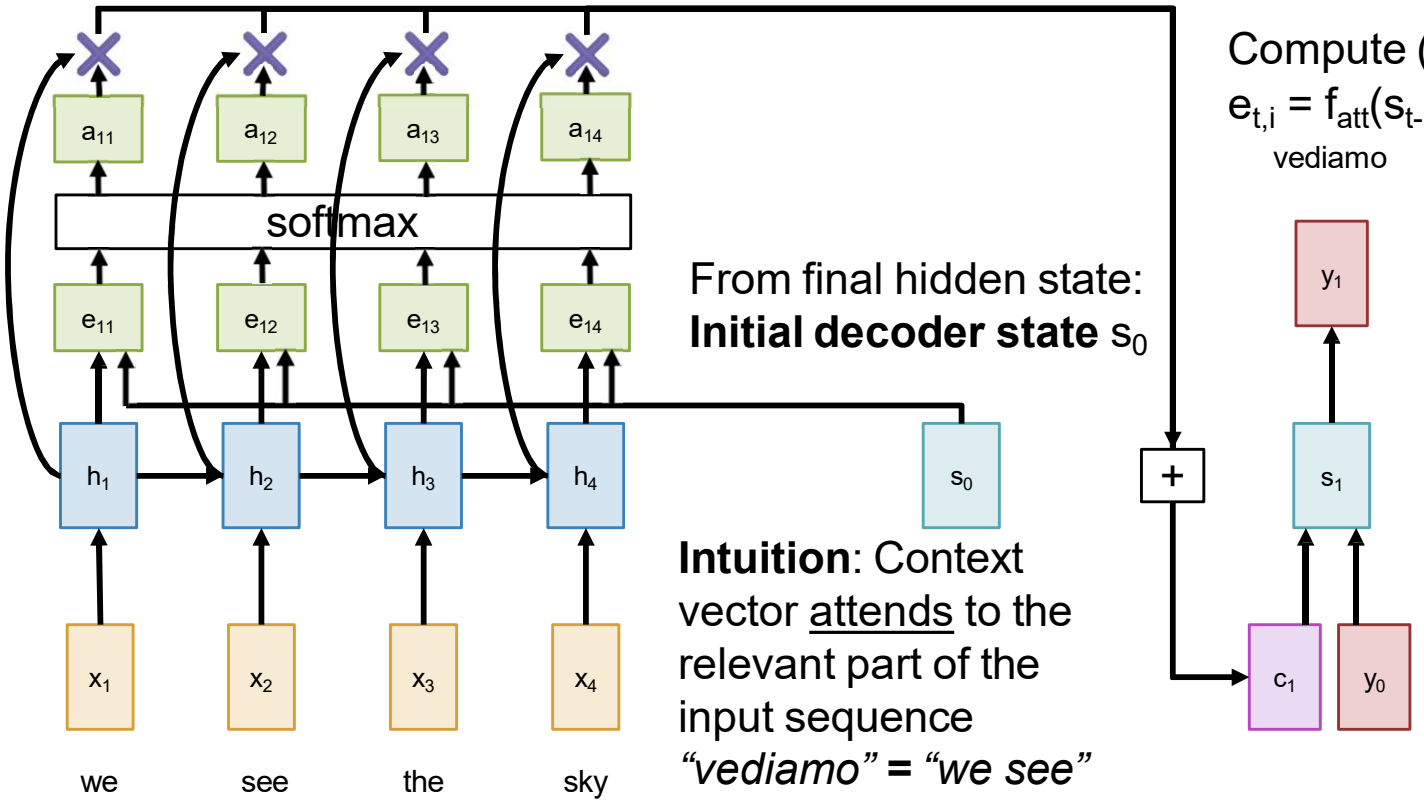
Normalize alignment scores to get **attention weights**
$0 < a_{t,i} < 1 \quad \sum_i a_{t,i} = 1$

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

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

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

Sequence to Sequence with RNNs and Attention



From final hidden state:
Initial decoder state s_0

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

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

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

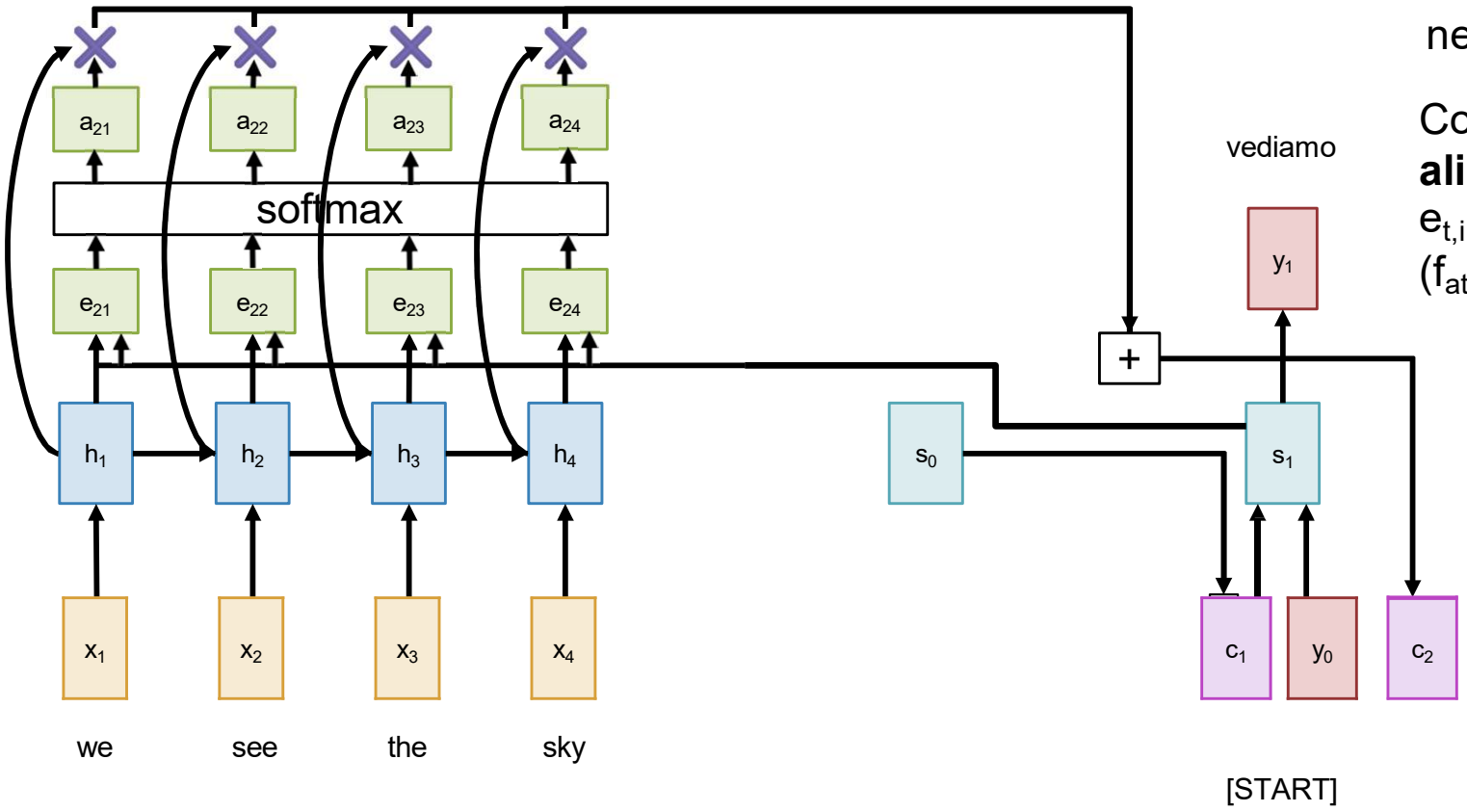
Compute **context vector** as weighted sum of hidden states
$c_t = \sum_i a_{t,i} h_i$

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

This is all differentiable! No supervision on attention weights – backprop through everything

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

Sequence to Sequence with RNNs and Attention

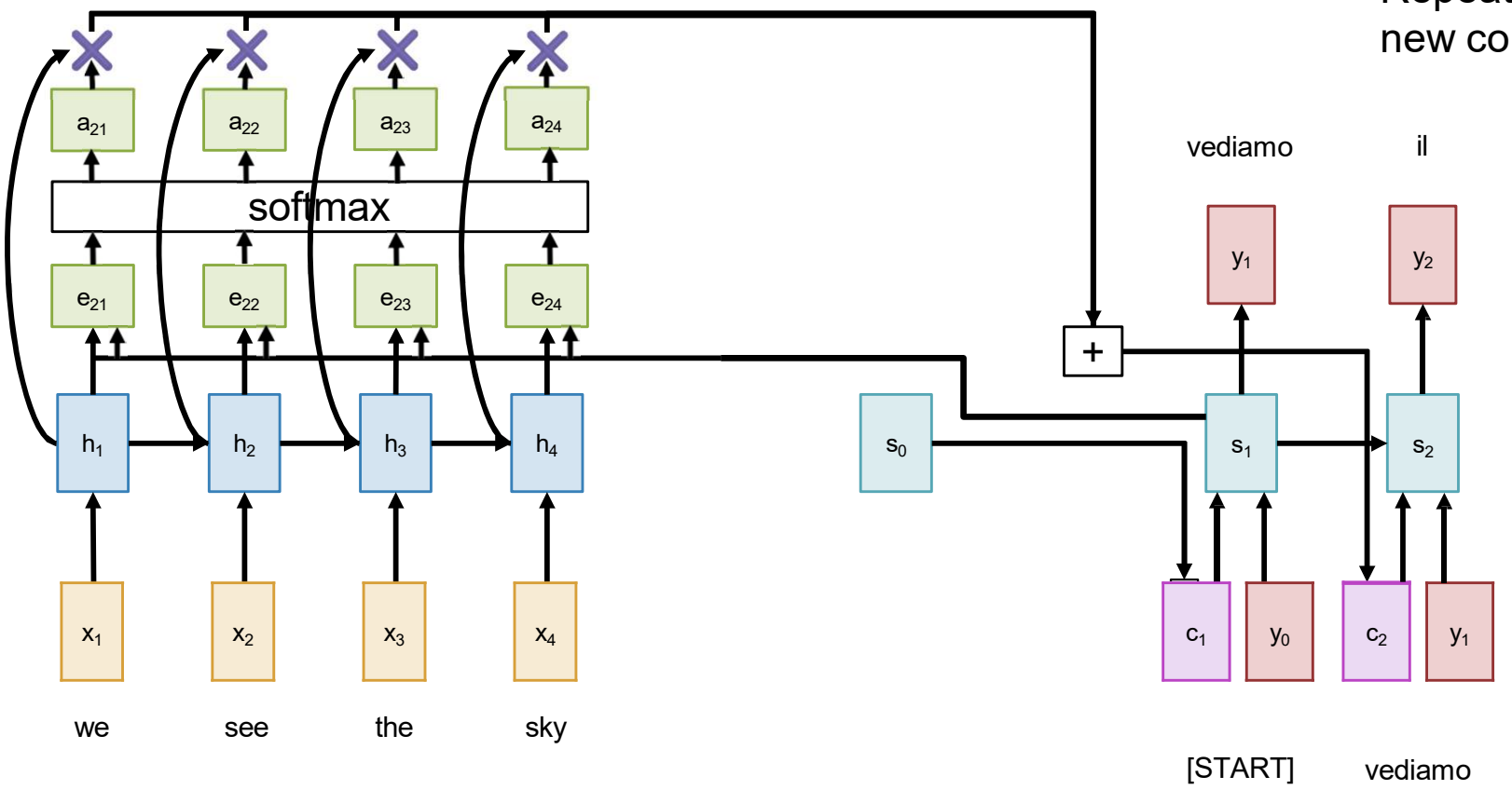


Repeat: Use s_1 to compute new context vector c_2

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

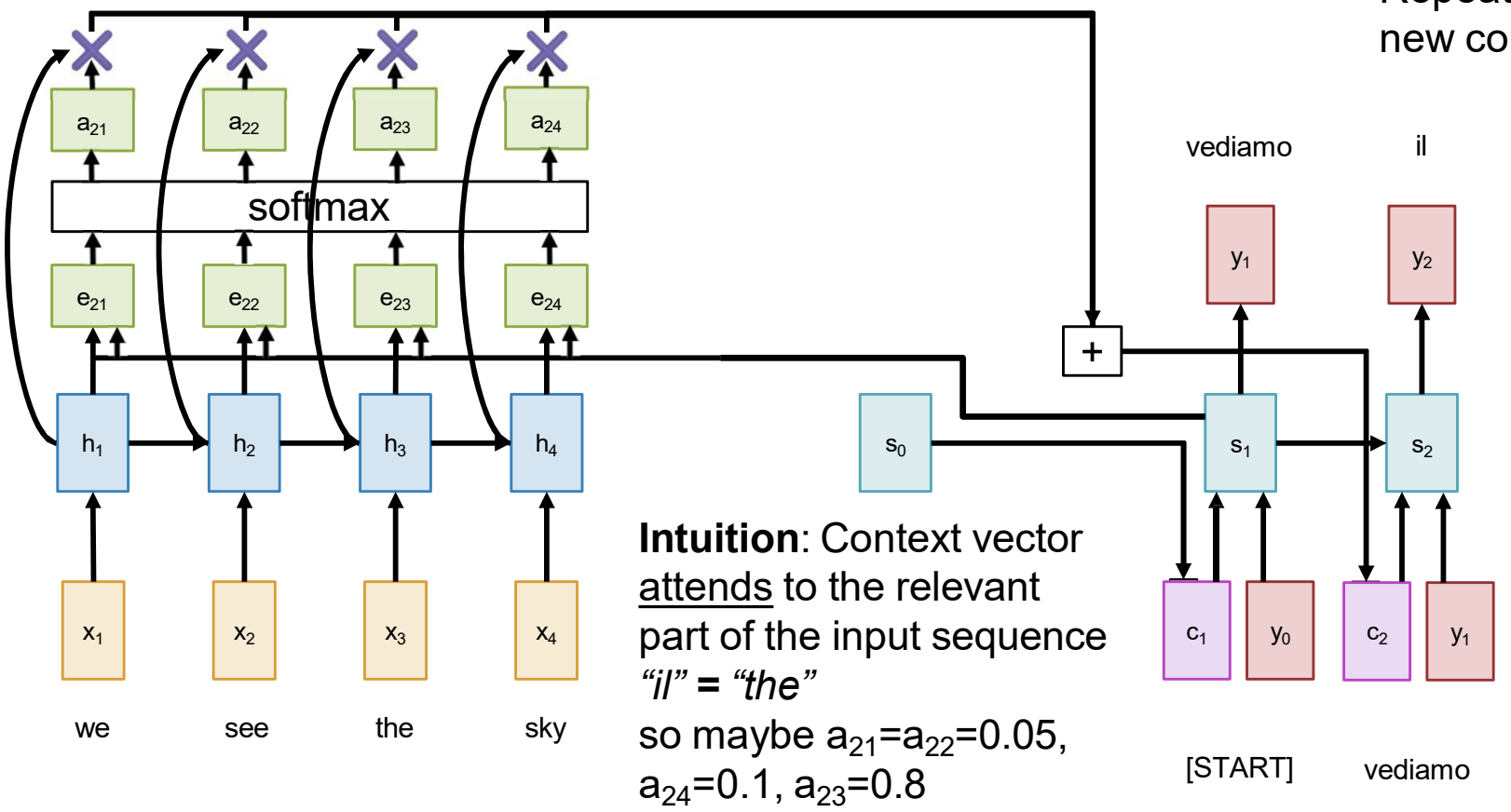
Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Sequence to Sequence with RNNs and Attention



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

Sequence to Sequence with RNNs and Attention

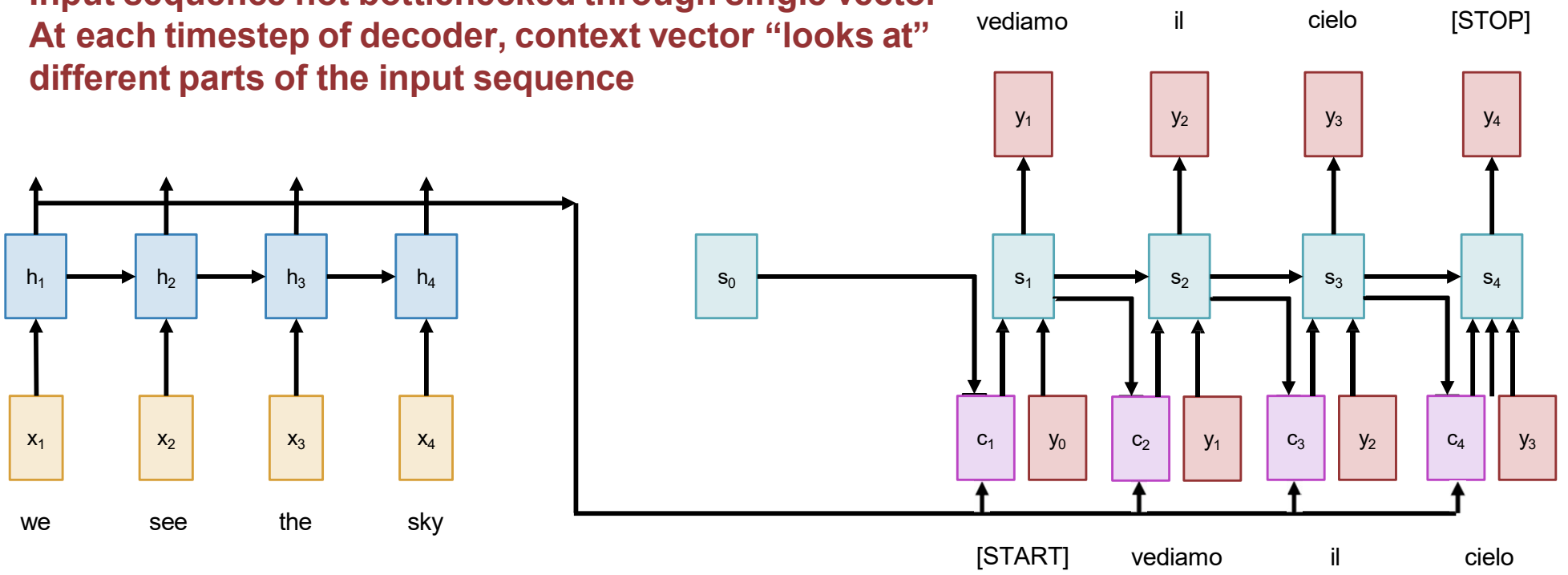


Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Sequence to Sequence with RNNs and Attention

Use a different context vector in each timestep of decoder

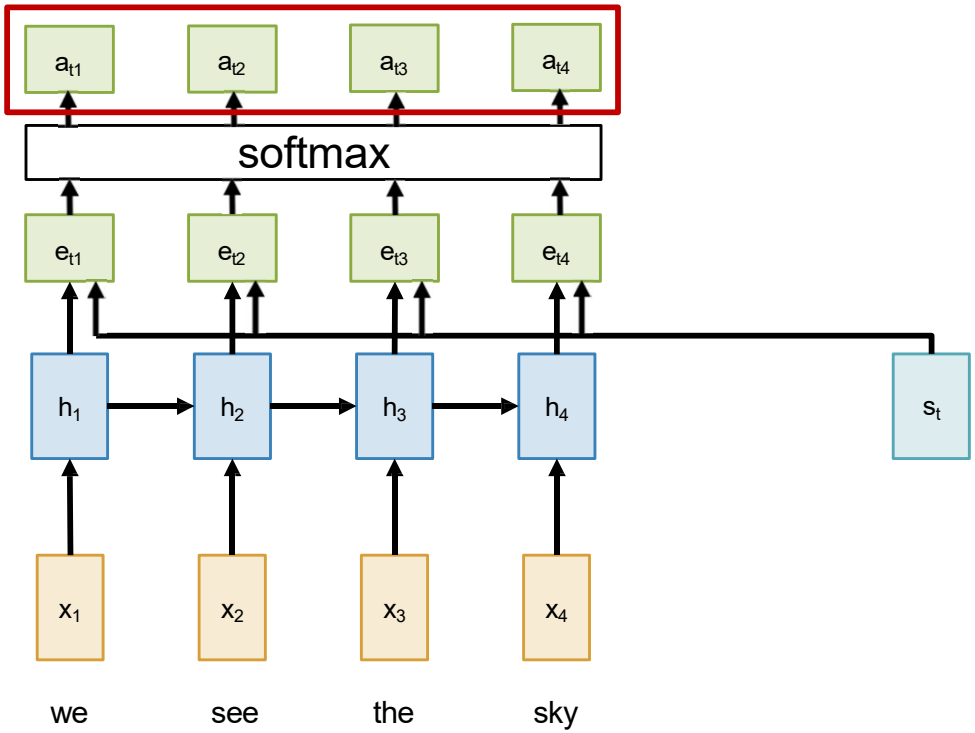
- **Input sequence not bottlenecked through single vector**
- **At each timestep of decoder, context vector "looks at" different parts of the input sequence**



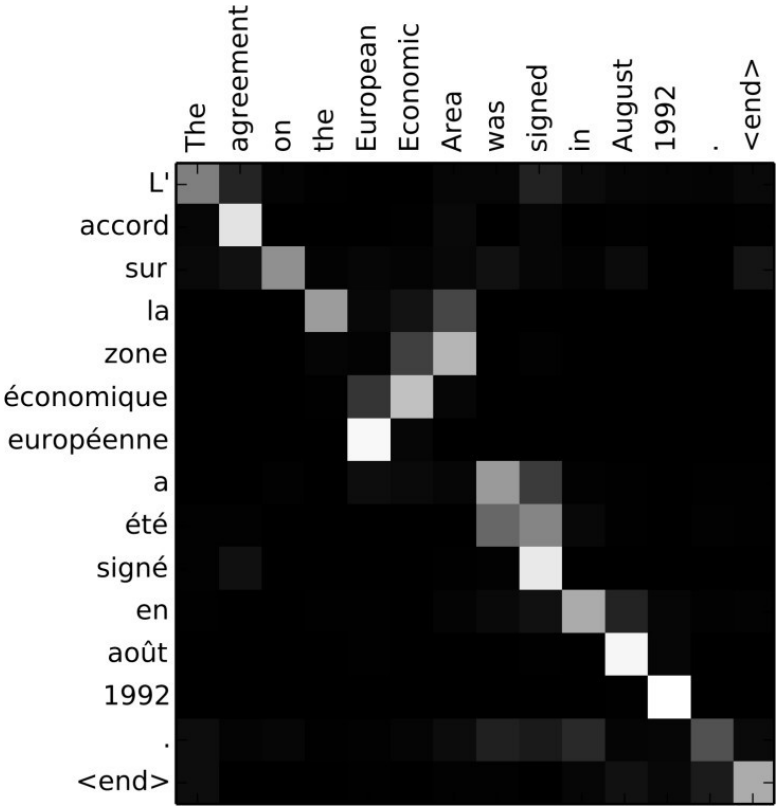
Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Sequence to Sequence with RNNs and Attention

Example: English to French translation



Visualize attention weights $a_{t,i}$



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

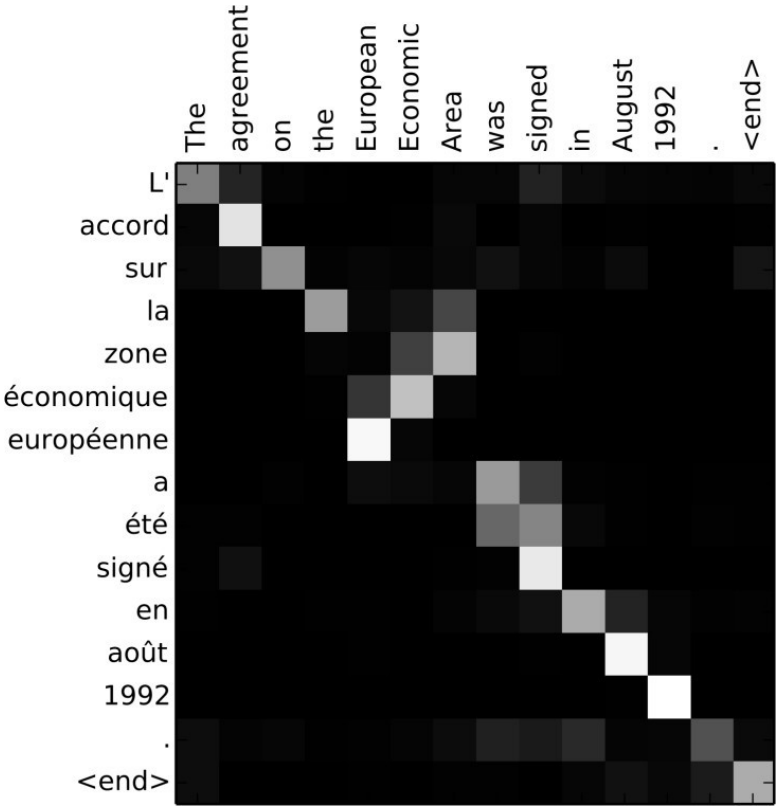
Sequence to Sequence with RNNs and Attention

Example: English to French translation

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

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

Visualize attention weights $a_{t,i}$



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

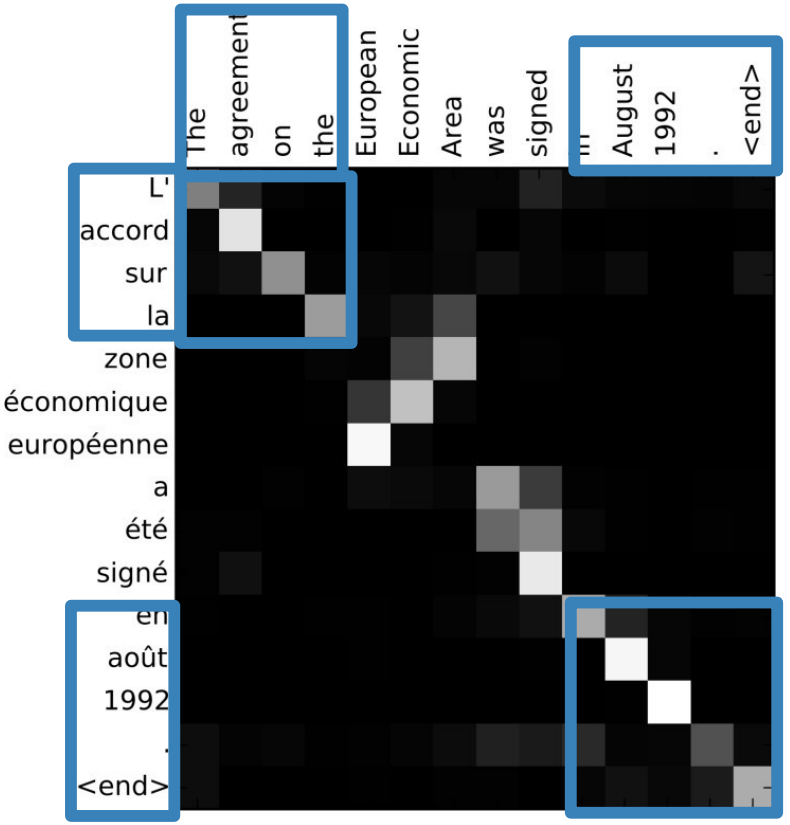
Sequence to Sequence with RNNs and Attention

Example: English to French translation

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

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

Visualize attention weights $a_{t,i}$



Diagonal attention means words correspond in order

Diagonal attention means words correspond in order

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

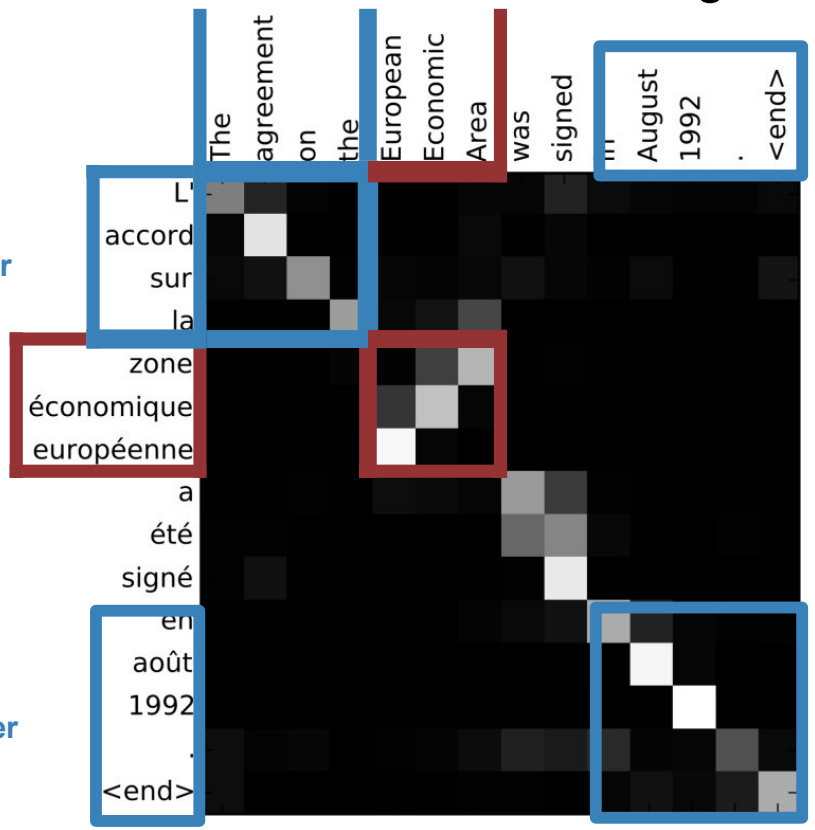
Sequence to Sequence with RNNs and Attention

Example: English to French translation

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

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

Visualize attention weights $a_{t,i}$



Diagonal attention means words correspond in order

Attention figures out different word orders

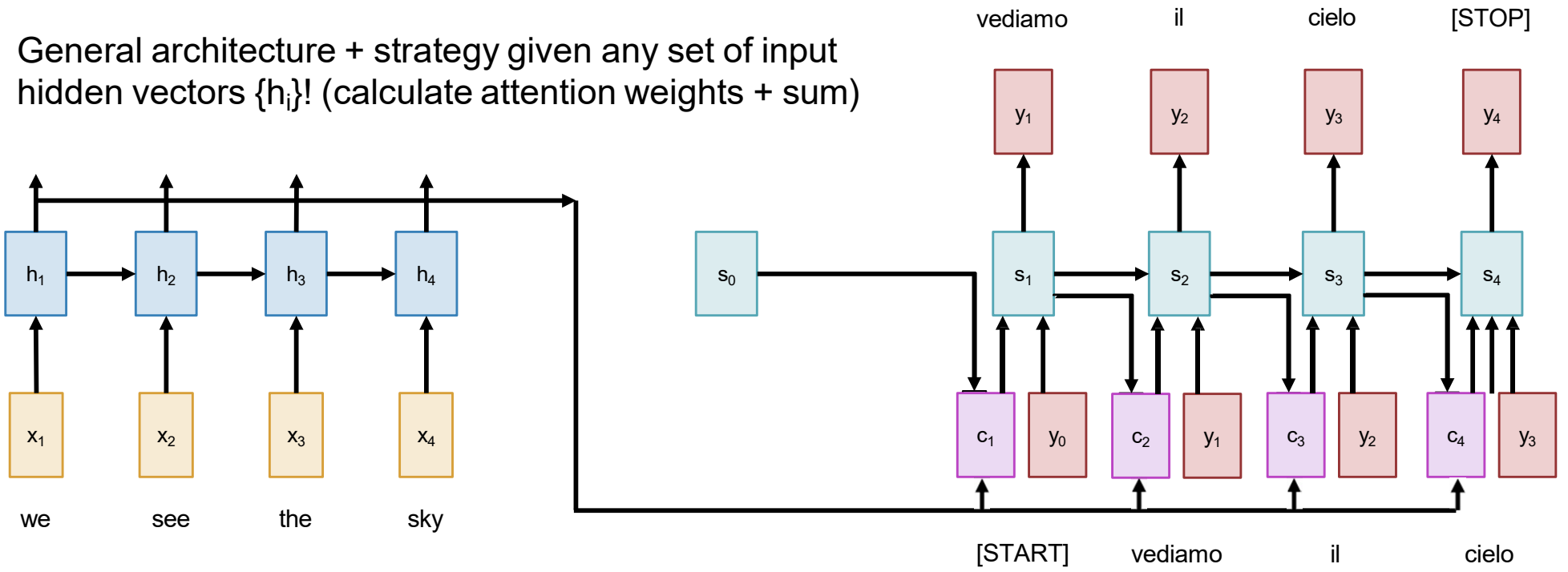
Diagonal attention means words correspond in order

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

Sequence to Sequence with RNNs and Attention

Context vectors don't use the fact that h_i form an ordered sequence – it just treats them as an unordered set $\{h_i\}$

General architecture + strategy given any set of input hidden vectors $\{h_i\}$! (calculate attention weights + sum)



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

Image Captioning using spatial features

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, ..., y_T$

An example network for image captioning
without attention

CNN

$z_{0,0}$	$z_{0,1}$	$z_{0,2}$
$z_{1,0}$	$z_{1,1}$	$z_{1,2}$
$z_{2,0}$	$z_{2,1}$	$z_{2,2}$

Extract spatial features from a pretrained CNN

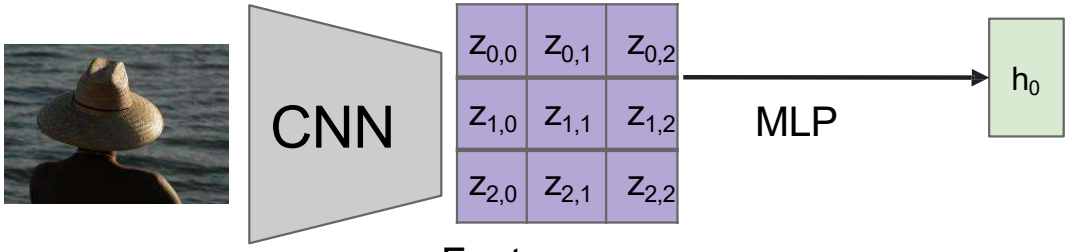
Features:
H x W x D

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, \ldots, y_T$

Encoder: $h_0 = f_{\mathbf{W}}(\mathbf{z})$
where **z** is spatial CNN features
$f_{\mathbf{W}}(.)$ is an MLP



Extract spatial features from a pretrained CNN

Features: H x W x D

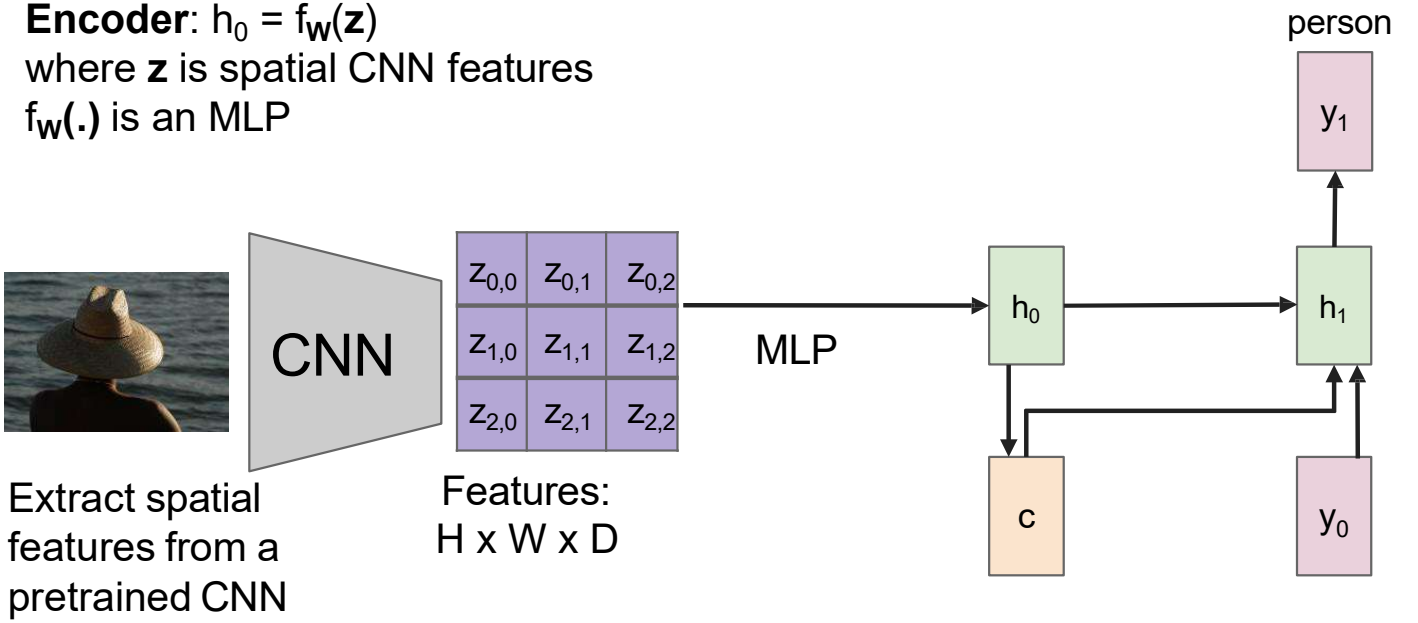
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, ..., y_T$

Encoder: $h_0 = f_{\mathbf{W}}(\mathbf{z})$
where **z** is spatial CNN features
$f_{\mathbf{W}}(.)$ is an MLP

Decoder: $h_t = g_{\mathbf{V}}(y_{t-1}, h_{t-1}, c)$
where context vector c is often $c = h_0$
and output $y_t = T(h_t)$



Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, ..., y_T$

Encoder: $h_0 = f_{\mathbf{W}}(\mathbf{z})$
where **z** is spatial CNN features
$f_{\mathbf{W}}(.)$ is an MLP

Decoder: $h_t = g_{\mathbf{V}}(y_{t-1}, h_{t-1}, c)$
where context vector c is often $c = h_0$
and output $y_t = T(h_t)$

CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

MLP

h_0 h_1 h_2

c

y_0 y_1

y_1 y_2

person wearing

[START] person

Extract spatial features from a pretrained CNN

Features: H x W x D

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, ..., y_T$

Encoder: $h_0 = f_{\mathbf{W}}(\mathbf{z})$
where **z** is spatial CNN features
$f_{\mathbf{W}}(.)$ is an MLP

Decoder: $h_t = g_{\mathbf{V}}(y_{t-1}, h_{t-1}, c)$
where context vector c is often $c = h_0$
and output $y_t = T(h_t)$

CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

MLP

h_0 h_1 h_2 h_3

c

y_0 y_1 y_2

y_1 y_2 y_3

person wearing hat

[START] person wearing

Extract spatial features from a pretrained CNN

Features: H x W x D

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, \ldots, y_T$

Encoder: $h_0 = f_{\mathbf{W}}(\mathbf{z})$
where **z** is spatial CNN features
$f_{\mathbf{W}}(.)$ is an MLP

Decoder: $h_t = g_{\mathbf{V}}(y_{t-1}, h_{t-1}, c)$
where context vector c is often $c = h_0$
and output $y_t = T(h_t)$

CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

Extract spatial features from a pretrained CNN

Features: H x W x D

MLP

h_0 h_1 h_2 h_3 h_4

c

y_0 y_1 y_2 y_3

[START] person wearing hat

y_1 y_2 y_3 y_4

person wearing hat [END]

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Input: Image **I**
Output: Sequence **y** = $y_1, y_2, \ldots, y_T$

Encoder: $h_0 = f_W(\mathbf{z})$
where **z** is spatial CNN features
$f_W(.)$ is an MLP

Decoder: $h_t = g_V(y_{t-1}, h_{t-1}, c)$
where context vector c is often $c = h_0$
and output $y_t = T(h_t)$

Q: What is the problem with this setup? Think back to last time…

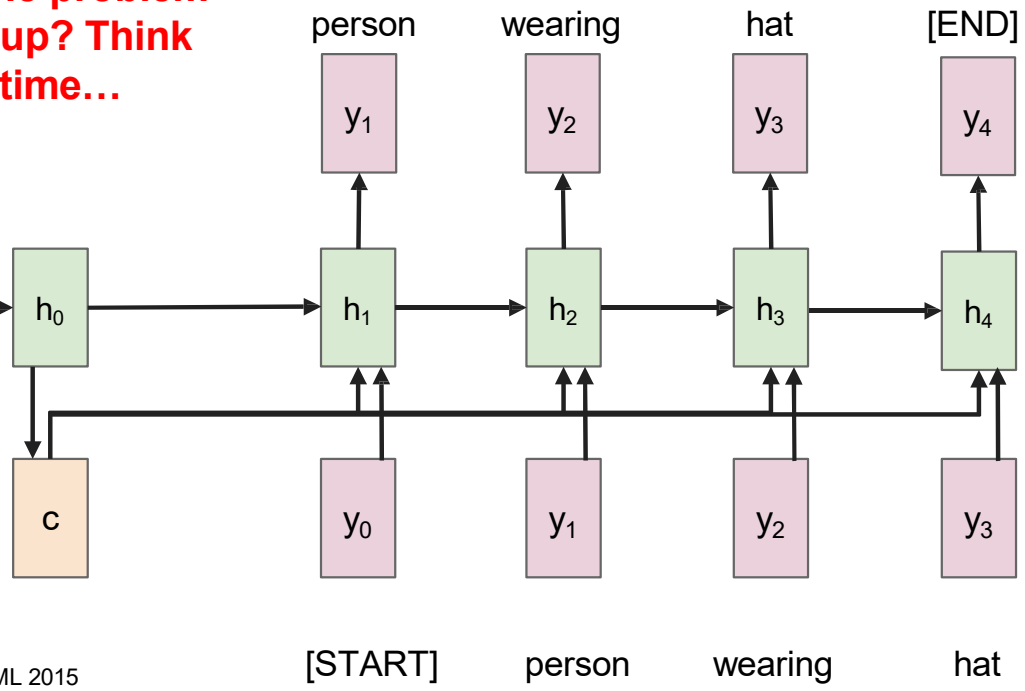
CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

Extract spatial features from a pretrained CNN

Features: H x W x D

MLP



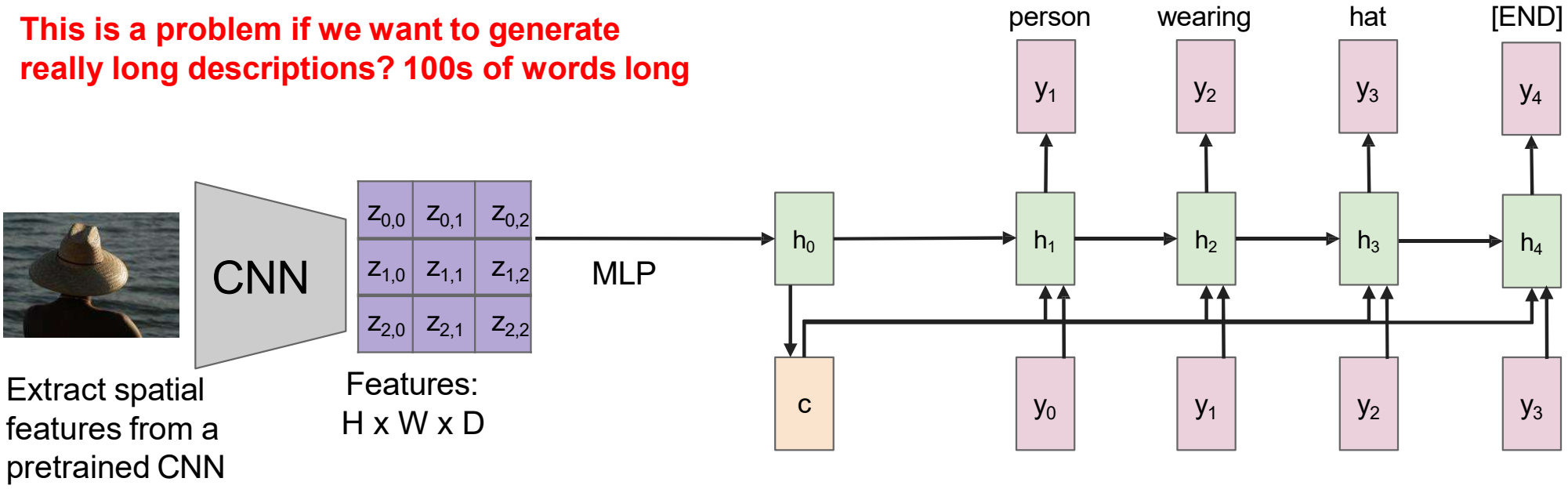
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Answer: Input is "bottlenecked" through c

- Model needs to encode everything it wants to say within c

This is a problem if we want to generate really long descriptions? 100s of words long



Extract spatial features from a pretrained CNN

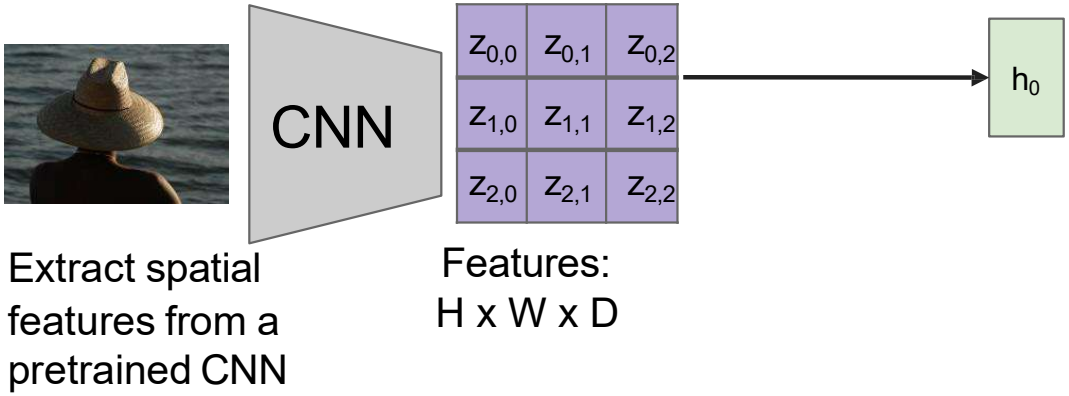
Features: H x W x D

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

Attention idea: New context vector at every time step.

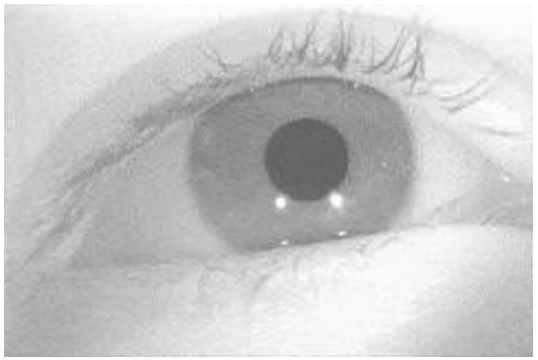
Each context vector will attend to different image regions



Extract spatial features from a pretrained CNN

Features: H x W x D

gif source



Attention Saccades in humans

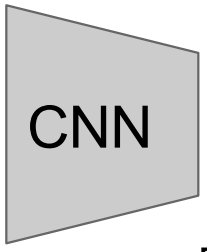
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

Compute alignments scores (scalars):

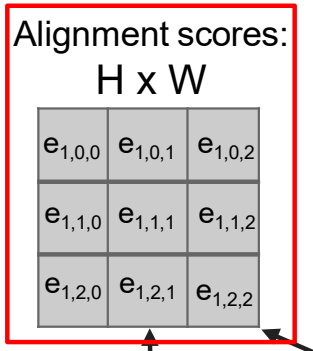
$$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$$

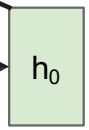
$f_{att}(.)$ is an MLP



Extract spatial features from a pretrained CNN

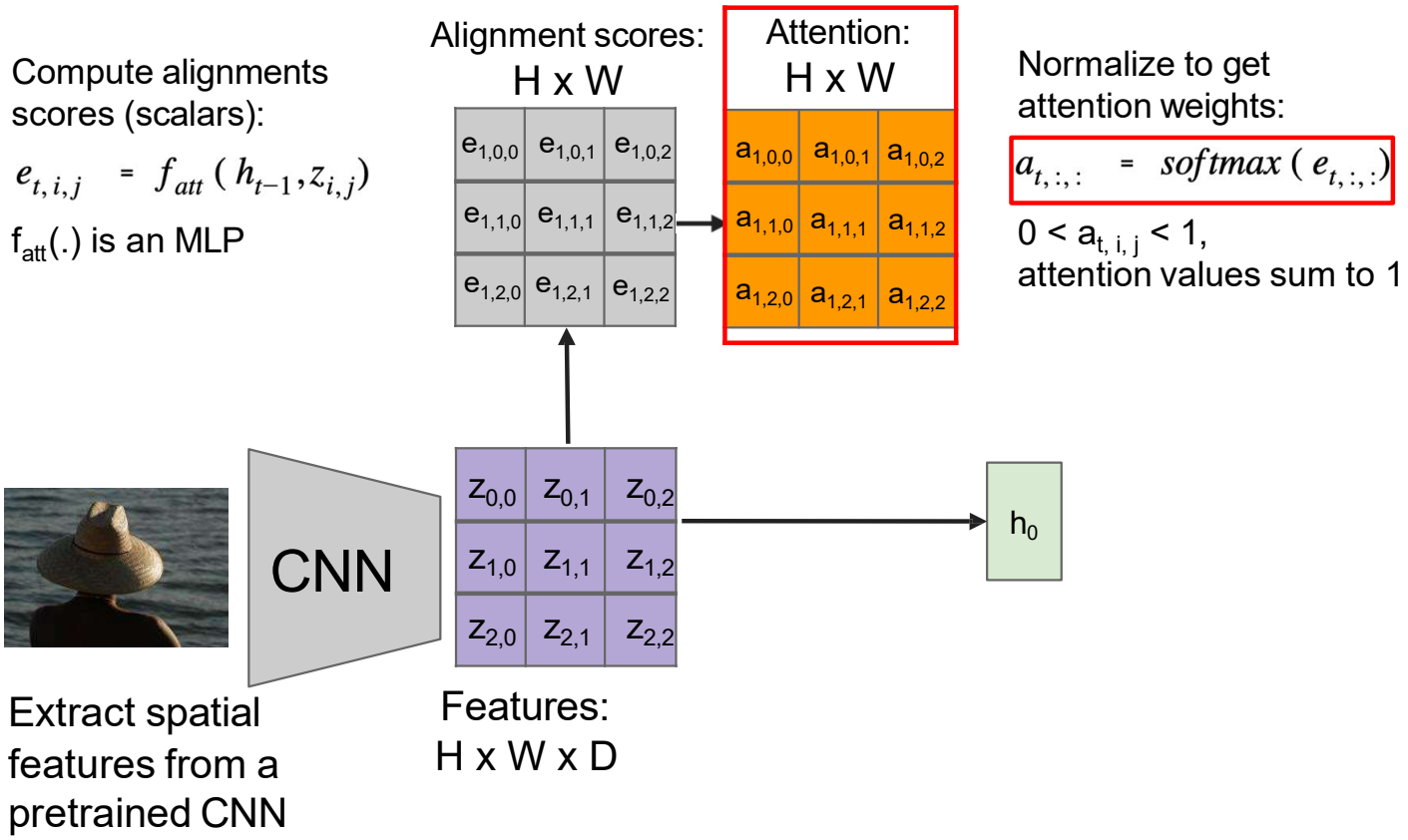
Features: H x W x D





Xu et al, “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

Image Captioning with RNNs and Attention



Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

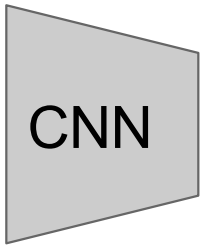
Image Captioning with RNNs and Attention

Compute alignments scores (scalars):

$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$

$f_{att}(.)$ is an MLP

Extract spatial features from a pretrained CNN



Features: H x W x D

Alignment scores: H x W

$e_{1,0,0}$ $e_{1,0,1}$ $e_{1,0,2}$
$e_{1,1,0}$ $e_{1,1,1}$ $e_{1,1,2}$
$e_{1,2,0}$ $e_{1,2,1}$ $e_{1,2,2}$

Attention: H x W

$a_{1,0,0}$ $a_{1,0,1}$ $a_{1,0,2}$
$a_{1,1,0}$ $a_{1,1,1}$ $a_{1,1,2}$
$a_{1,2,0}$ $a_{1,2,1}$ $a_{1,2,2}$

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

h_0

c_1

Normalize to get attention weights:

$a_{t,:,:} = softmax(e_{t,:,:})$

$0 < a_{t,i,j} < 1$,
attention values sum to 1

Compute context vector:

$$c_t = \sum_{i,j} a_{t,i,j} z_{t,i,j}$$

Q: How many context vectors are computed?

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

Each timestep of decoder uses a different context vector that looks at different parts of the input image

Decoder: $y_t = g_V(y_{t-1}, h_{t-1}, c_t)$
New context vector at every time step

$$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$$
$$a_{t,:,:} = softmax(e_{t,:,:})$$
$$c_t = \sum_{i,j} a_{t,i,j} z_{t,i,j}$$

CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

Features:
H x W x D

Extract spatial features from a pretrained CNN

h_0

person

y_1

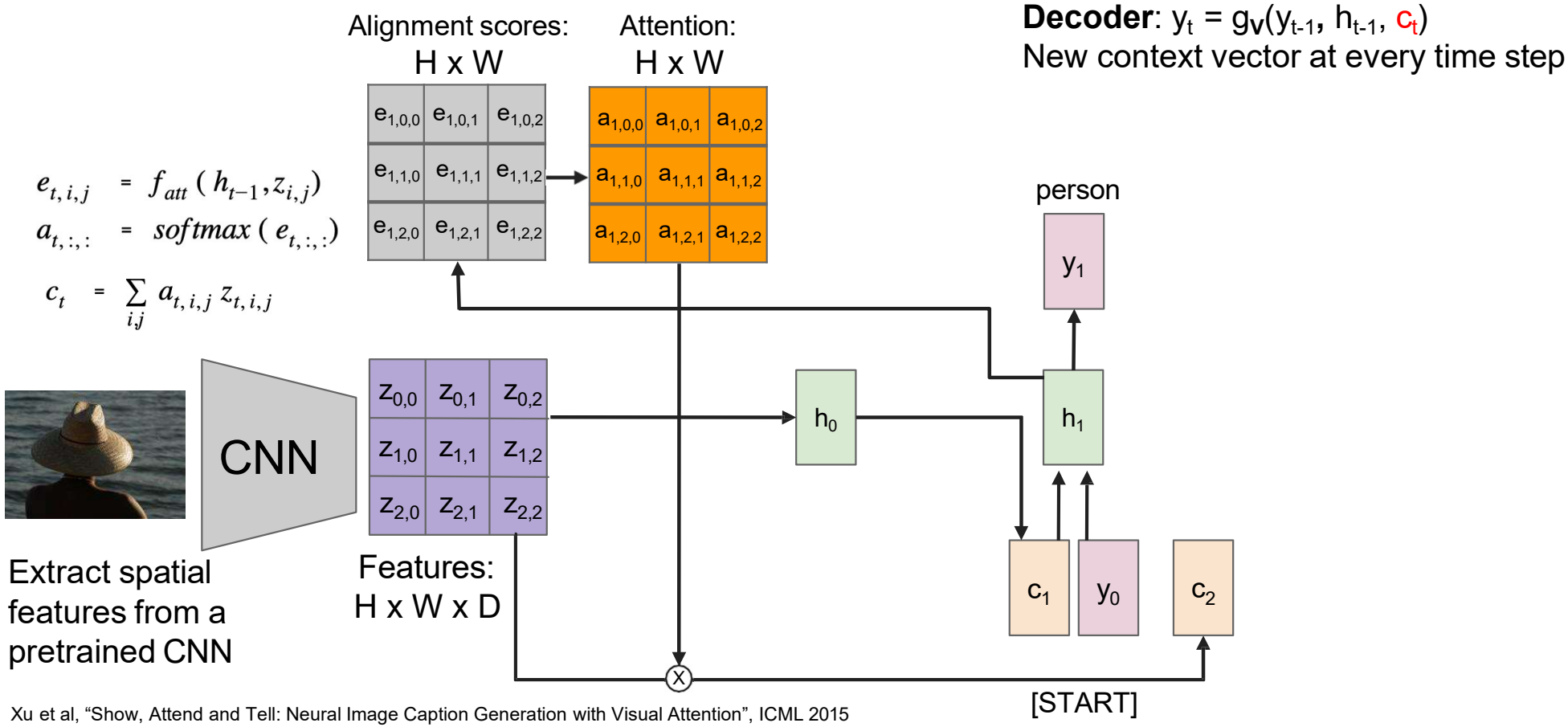
h_1

c_1 y_0

[START]

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention



Decoder: $y_t = g_V(y_{t-1}, h_{t-1}, c_t)$
New context vector at every time step

Image Captioning with RNNs and Attention

Each timestep of decoder uses a different context vector that looks at different parts of the input image

Decoder: $y_t = g_V(y_{t-1}, h_{t-1}, c_t)$
New context vector at every time step

$$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$$
$$a_{t,:,:} = softmax(e_{t,:,:})$$
$$c_t = \sum_{i,j} a_{t,i,j} z_{t,i,j}$$

CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

Extract spatial features from a pretrained CNN

Features: H x W x D

h_0 h_1 h_2

y_1 y_2

person wearing

c_1 y_0 c_2 y_1

[START] person

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

Each timestep of decoder uses a different context vector that looks at different parts of the input image

Decoder: $y_t = g_V(y_{t-1}, h_{t-1}, c_t)$
New context vector at every time step

$$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$$
$$a_{t,:,:} = softmax(e_{t,:,:})$$
$$c_t = \sum_{i,j} a_{t,i,j} z_{t,i,j}$$

CNN

Extract spatial features from a pretrained CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

Features: H x W x D

h_0 h_1 h_2 h_3

c_1 y_0 c_2 y_1 c_3 y_2

person wearing hat

y_1 y_2 y_3

[START] person wearing

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

Each timestep of decoder uses a different context vector that looks at different parts of the input image

Decoder: $y_t = g_V(y_{t-1}, h_{t-1}, c_t)$
New context vector at every time step

$$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$$
$$a_{t,:,:} = softmax(e_{t,:,:})$$
$$c_t = \sum_{i,j} a_{t,i,j} z_{t,i,j}$$

CNN

Extract spatial features from a pretrained CNN

$z_{0,0}$ $z_{0,1}$ $z_{0,2}$
$z_{1,0}$ $z_{1,1}$ $z_{1,2}$
$z_{2,0}$ $z_{2,1}$ $z_{2,2}$

Features: H x W x D

h_0 h_1 h_2 h_3 h_4

c_1 y_0 c_2 y_1 c_3 y_2 c_4 y_3

person wearing hat [END]

y_1 y_2 y_3 y_4

[START] person wearing hat

Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

CNN

Extract spatial features from a pretrained CNN

Features: H x W x D

Alignment scores: H x W

Attention: H x W

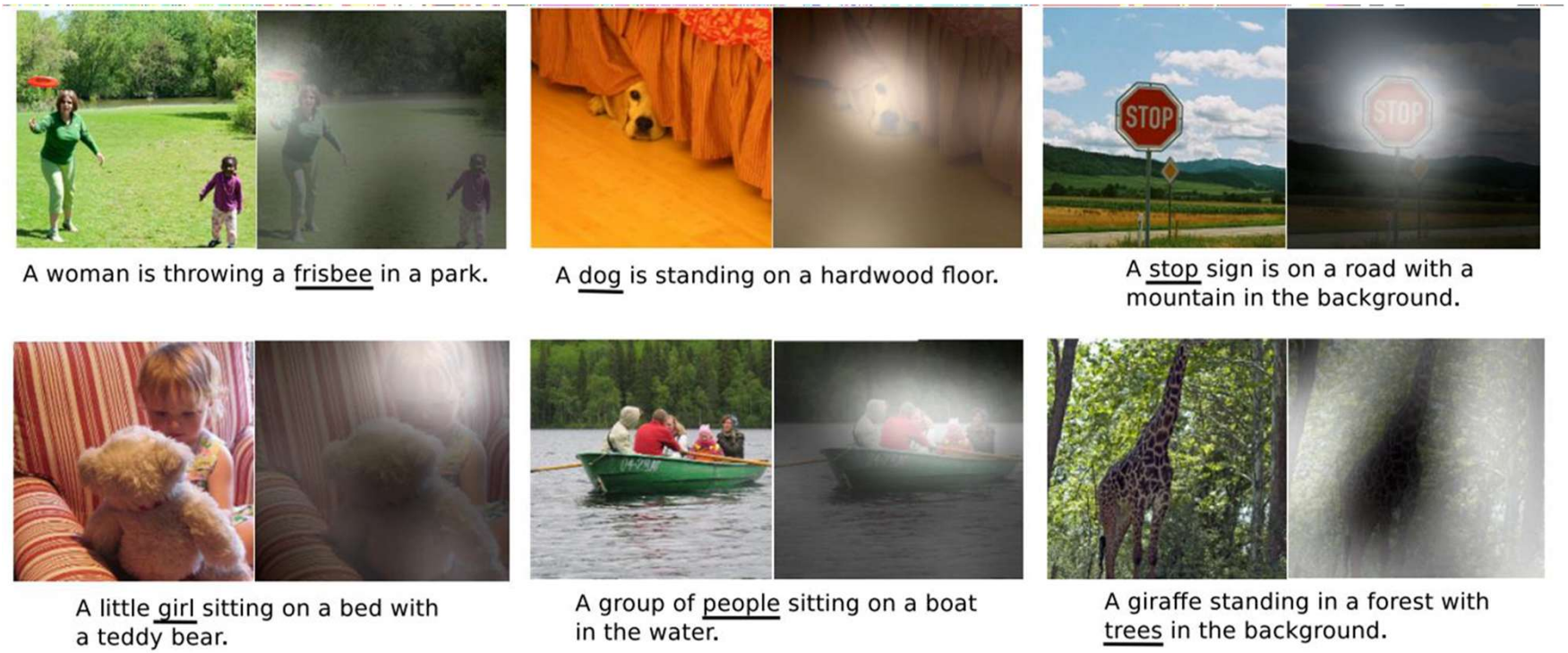
This entire process is differentiable.
- model chooses its own attention weights. No attention supervision is required

person wearing hat [END]

[START] person wearing hat

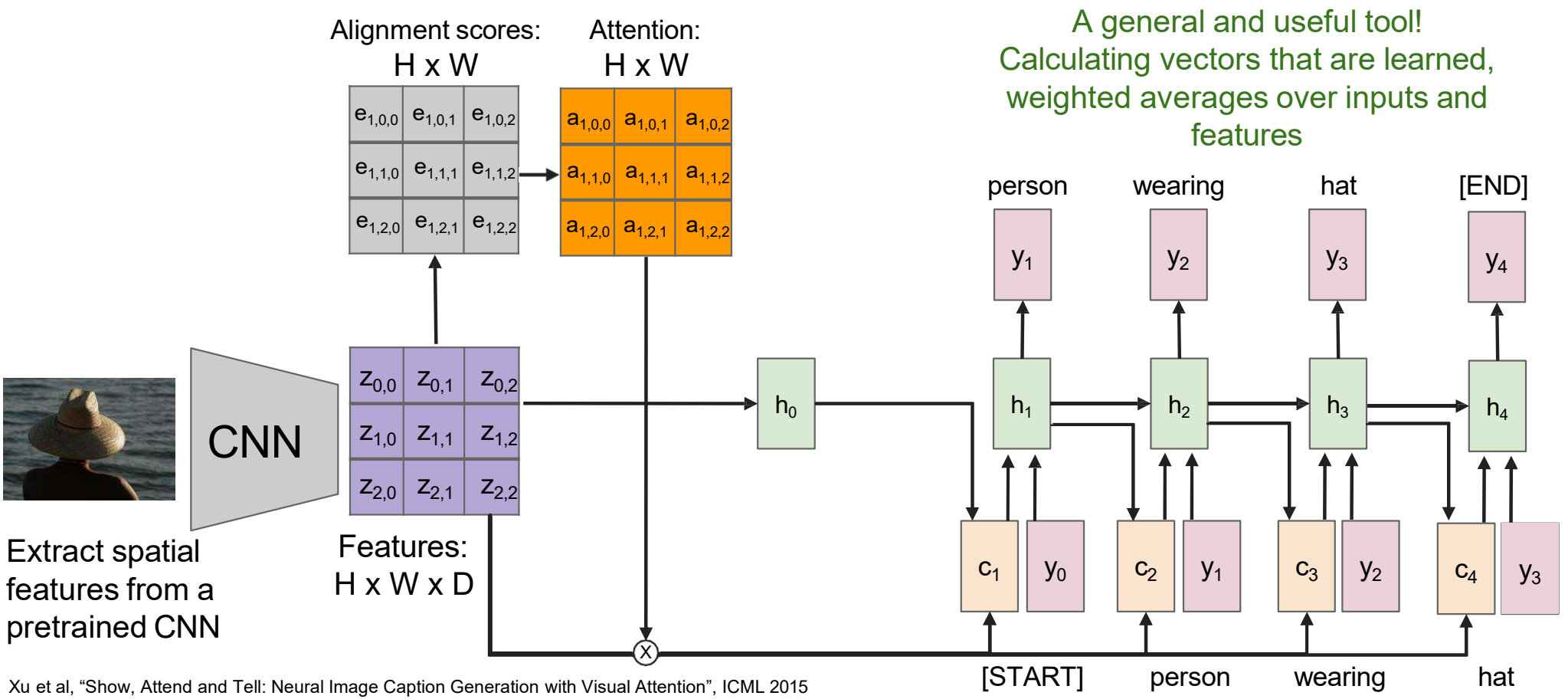
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with Attention



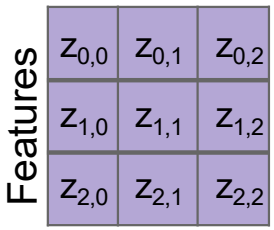
Xu et al, “Show, Attend, and Tell: Neural Image Caption Generation with Visual Attention”, ICML 2015

Image Captioning with RNNs and Attention



Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Attention we just saw in image captioning

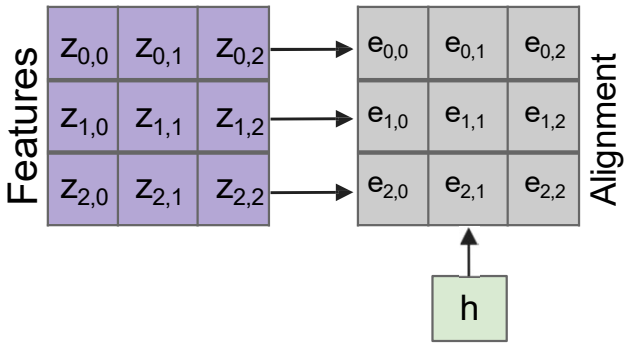


h

Inputs:
Features: **z** (shape: H x W x D)
Query: **h** (shape: D) ← “query” refers to a vector used to calculate a corresponding context vector.

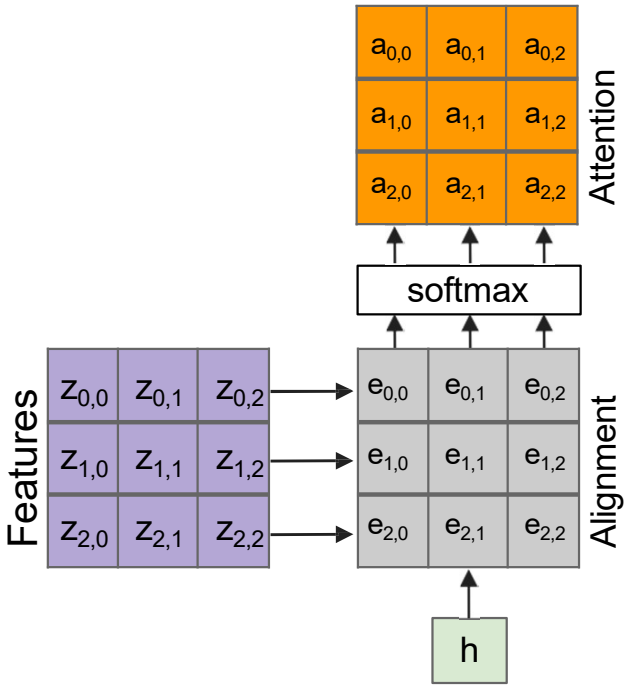
Attention we just saw in image captioning

Operations:
Alignment: $e_{i,j} = f_{att}(h, z_{i,j})$



Inputs:
Features: **z** (shape: H x W x D)
Query: **h** (shape: D)

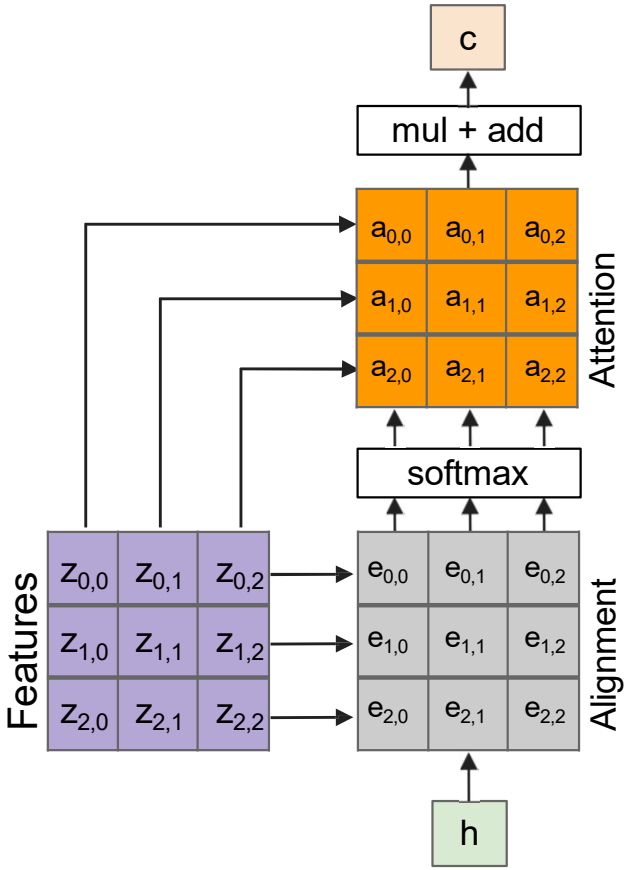
Attention we just saw in image captioning



Operations:
Alignment: $e_{i,j} = f_{att}(h, z_{i,j})$
Attention: **a** = softmax(**e**)

Inputs:
Features: **z** (shape: H x W x D)
Query: **h** (shape: D)

Attention we just saw in image captioning

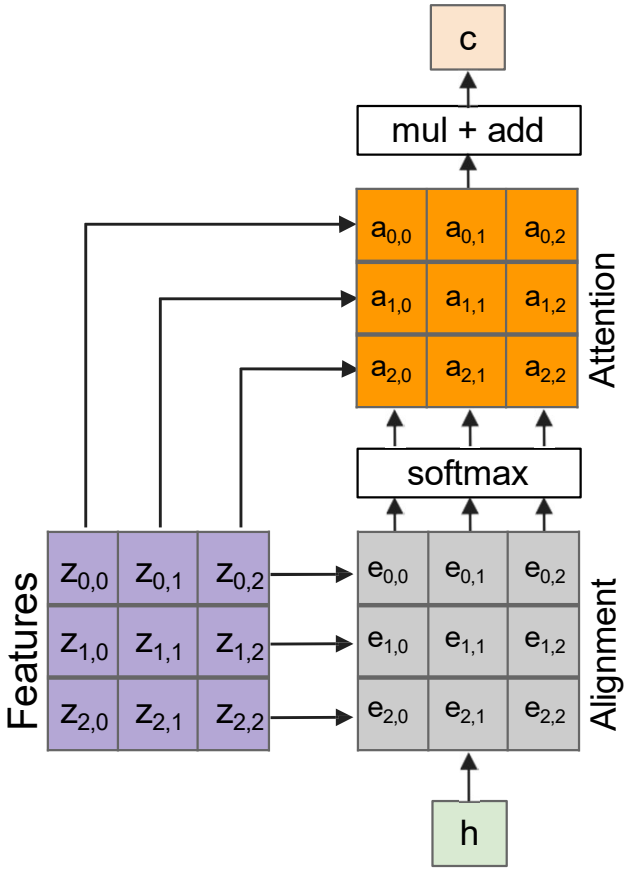


Outputs:
context vector: **c** (shape: D)

Operations:
Alignment: $e_{i,j} = f_{att}(h, z_{i,j})$
Attention: **a** = softmax(**e**)
Output: **c** = $\sum_{i,j} a_{i,j} z_{i,j}$

Inputs:
Features: **z** (shape: H x W x D)
Query: **h** (shape: D)

Attention we just saw in image captioning



Outputs:
context vector: **c** (shape: D)

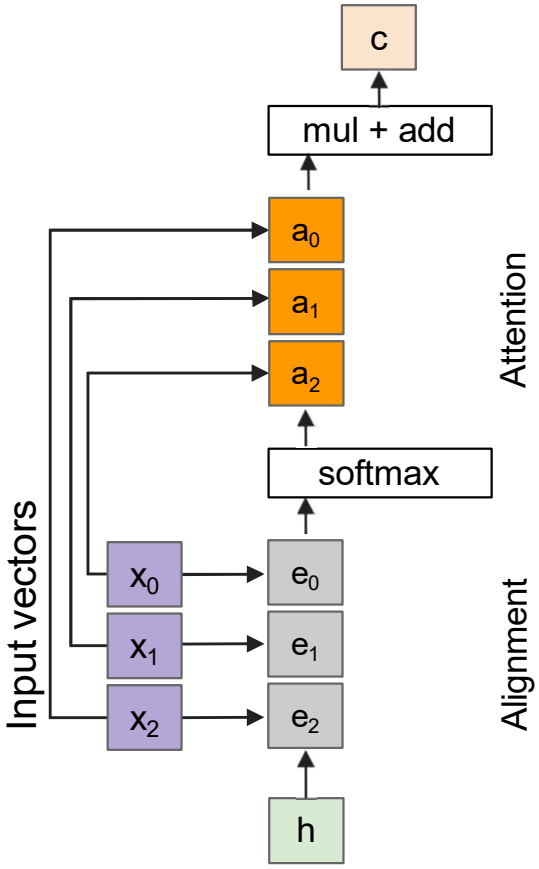
Operations:
Alignment: $e_{i,j} = f_{att}(h, z_{i,j})$
Attention: **a** = softmax(**e**)
Output: $\mathbf{c} = \sum_{i,j} a_{i,j} z_{i,j}$

Inputs:
Features: **z** (shape: H x W x D)
Query: **h** (shape: D)

How is this different from the attention mechanism in transformers?

We'll go into that next, any questions?

General attention layer – used in LLMs + beyond



Outputs:
context vector: **c** (shape: D)

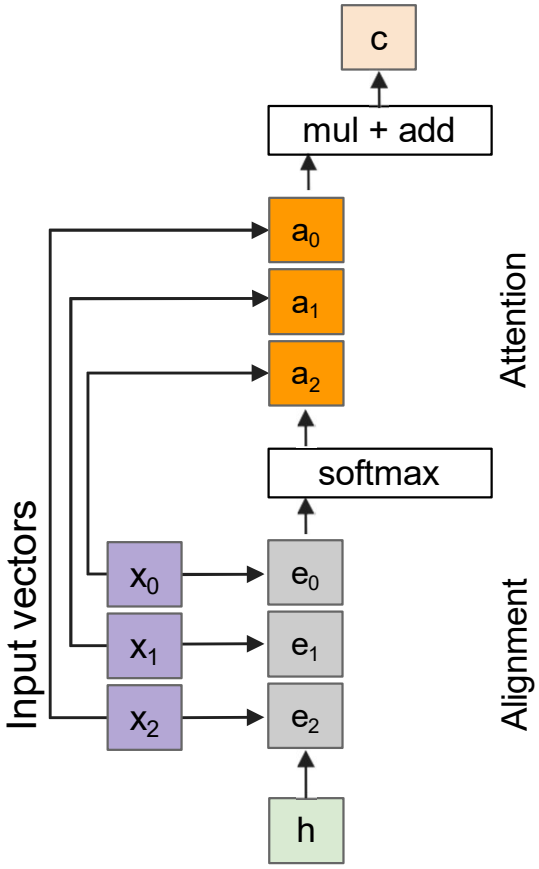
Operations:
Alignment: $e_i = f_{att}(h, x_i)$
Attention: **a** = softmax(**e**)
Output: **c** = $\sum_i a_i x_i$

Inputs:
Input vectors: **x** (shape: N x D)
Query: **h** (shape: D)

Attention operation is **permutation invariant.**

- Doesn't care about ordering of the features
- Stretch into **N = H x W** vectors

General attention layer



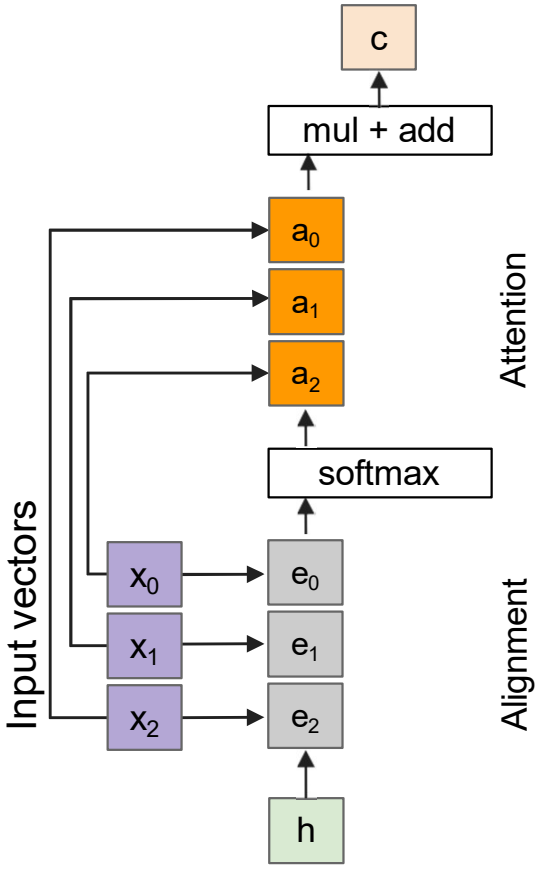
Outputs:
context vector: **c** (shape: D)

Operations:
Alignment: $e_i = h \cdot x_i$
Attention: **a** = softmax(**e**)
Output: **c** = $\sum_i a_i x_i$

Change $f_{att}(.)$ to a dot product, this actually can work well in practice, but a simple dot product can have some issues…

Inputs:
Input vectors: **x** (shape: N x D)
Query: **h** (shape: D)

General attention layer



Outputs:
context vector: **c** (shape: D)

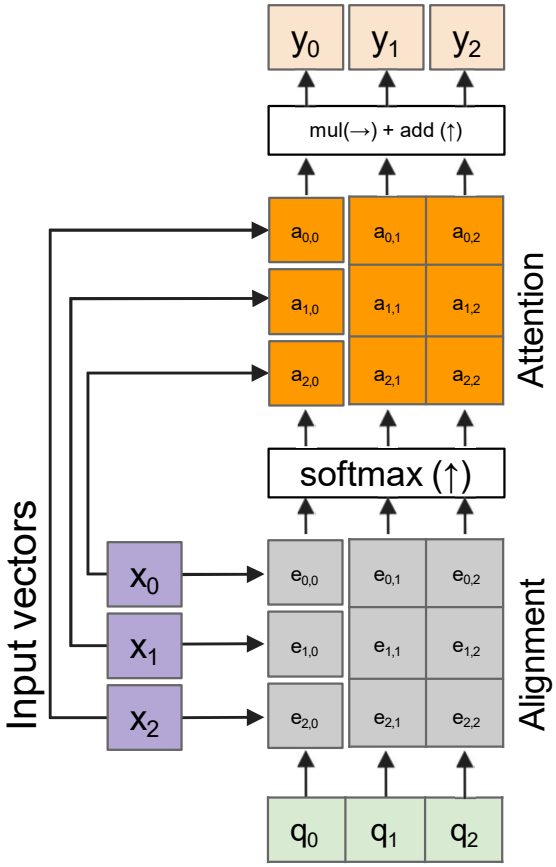
Operations:
Alignment: $e_i = h \cdot x_i / \sqrt{D}$
Attention: **a** = softmax(**e**)
Output: **c** = $\sum_i a_i x_i$

Inputs:
Input vectors: **x** (shape: N x D)
Query: **h** (shape: D)

Change $f_{att}(.)$ to a scaled simple dot product

- Larger dimensions means more terms in the dot product sum.
- So, the variance of the logits is higher. Large magnitude vectors will produce much higher logits.
- So, the post-softmax distribution has lower-entropy, assuming logits are IID.
- Ultimately, these large magnitude vectors will cause softmax to peak and assign very little weight to all others
- Divide by $\sqrt{D}$ to reduce effect of large magnitude vectors
- Similar to Xavier and Kaiming Initialization!

General attention layer



Outputs:
context vectors: **y** (shape: D)

Operations:
Alignment: $e_{i,j} = q_j \cdot x_i / \sqrt{D}$
Attention: **a** = softmax(**e**)
Output: $y_j = \sum_i a_{i,j} x_i$

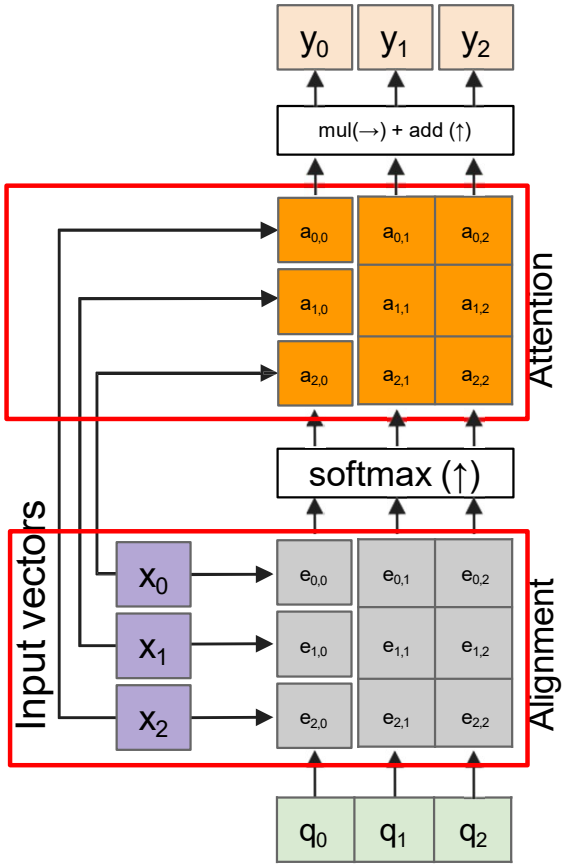
Inputs:
Input vectors: **x** (shape: N x D)
Queries: **q** (shape: M x D)

Multiple query vectors
- each query creates a new, corresponding output context vector

Allows us to compute multiple attention context vectors at once
Will go into more details in future slides, but this allows us to compute context vectors for multiple timesteps in parallel

Multiple query vectors

General attention layer



Outputs:
context vectors: **y** (shape: D)

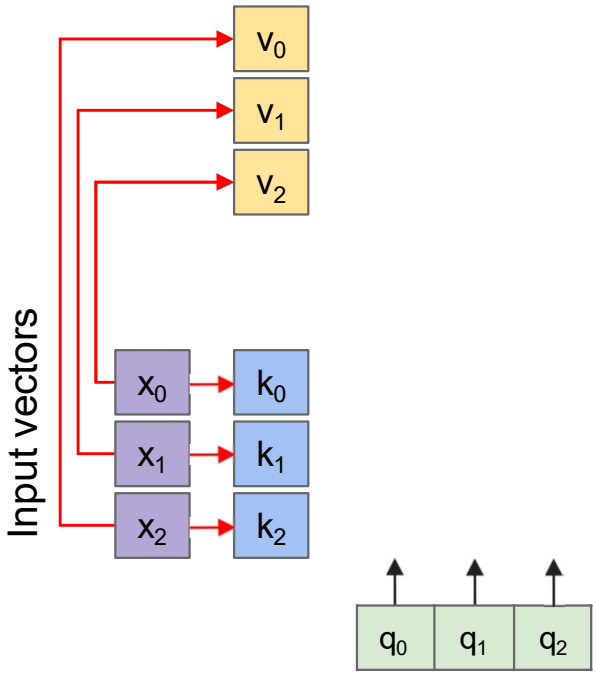
Operations:
Alignment: $e_{i,j} = q_j \cdot x_i / \sqrt{D}$
Attention: **a** = softmax(**e**)
Output: $y_j = \sum_i a_{i,j} x_i$

Inputs:
Input vectors: **x** (shape: N x D)
Queries: **q** (shape: M x D)

Notice that the input vectors are used for **both the alignment as well as the attention** calculations.

- We can add more expressivity to the layer by adding a different FC layer before each of the two steps.

General attention layer



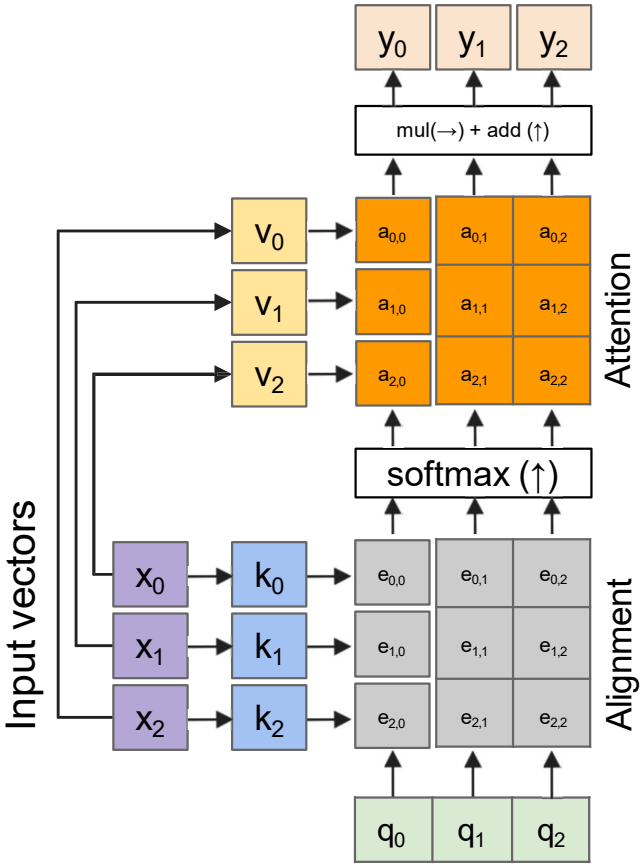
Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$

Inputs:
Input vectors: **x** (shape: N x D)
Queries: **q** (shape: M x D_k)

Notice that the input vectors are used for both the alignment as well as the attention calculations.

- We can add more expressivity to the layer by adding a different FC layer before each of the two steps.

General attention layer



Outputs:
context vectors: **y** (shape: D_v)

Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

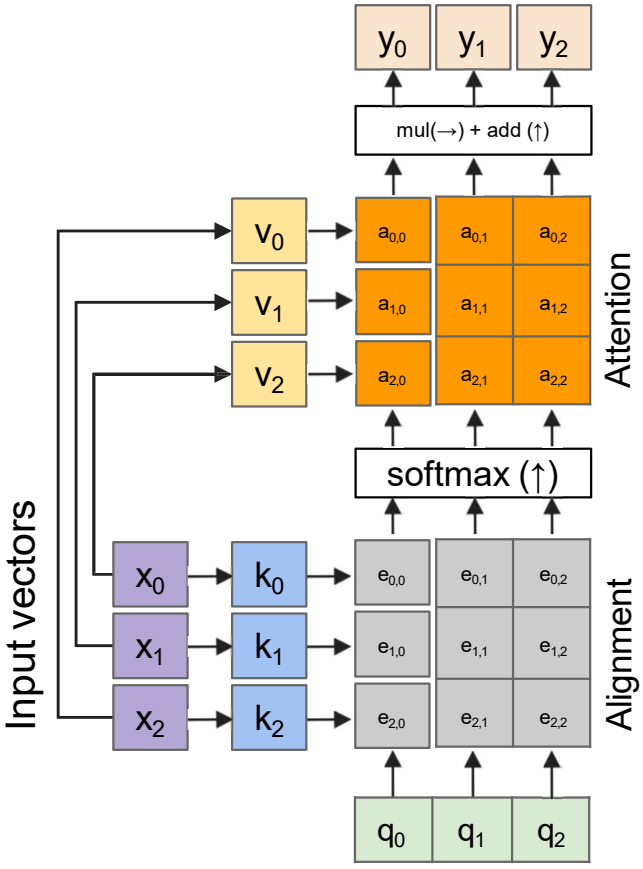
Inputs:
Input vectors: **x** (shape: N x D)
Queries: **q** (shape: M x D_k)

The input and output dimensions can now change depending on the key and value FC layers

Since the alignment scores are just scalars, the value vectors can be any dimension we want

General attention layer

This is a working example of how we could use an attention layer + CNN encoder for image captioning



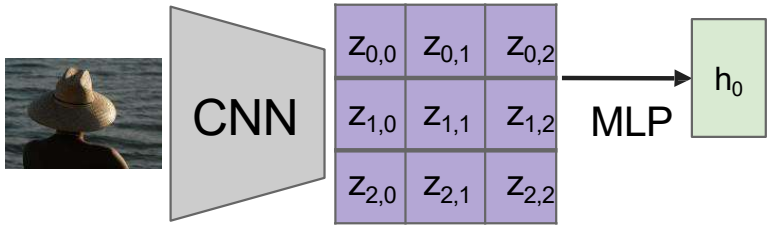
Outputs:
context vectors: **y** (shape: D_v)

Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:
Input vectors: **x** (shape: N x D)
Queries: **q** (shape: M x D_k)

Recall that the query vector was a function of the input vectors

Encoder: $h_0 = f_W(\mathbf{z})$
where **z** is spatial CNN features
$f_W(.)$ is an MLP

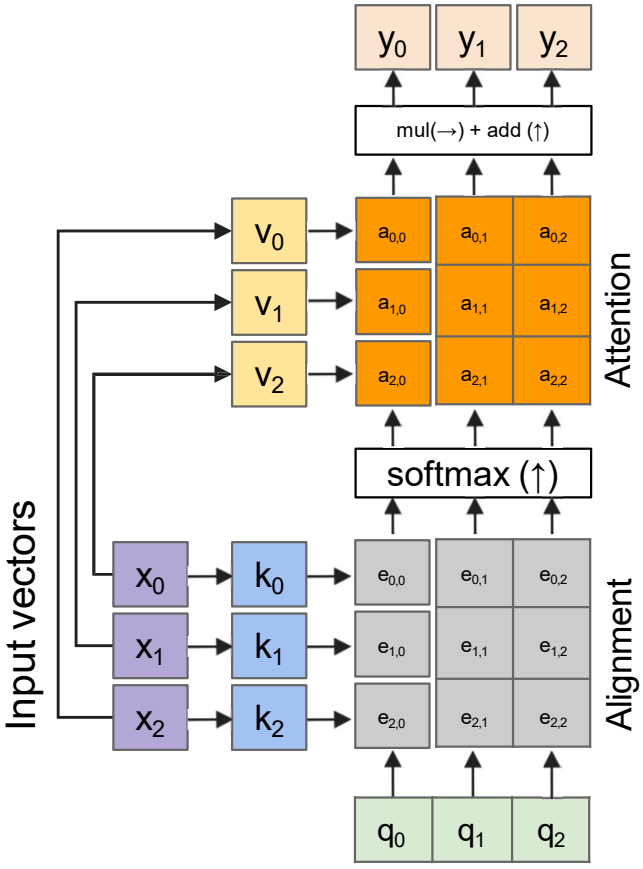


We used h_0 as q_0 previously

2. From Attention Layers to Transformers

Prof. Jianguo Zhang
SUSTech

Next: The Self-attention Layer



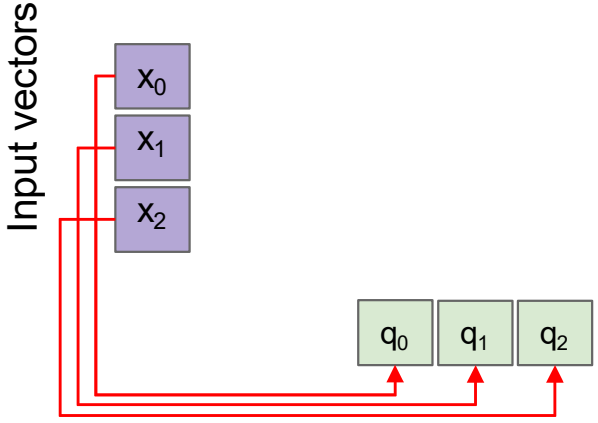
Outputs:
context vectors: **y** (shape: D_v)

Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:
Input vectors: **x** (shape: N x D)
Queries: **q** (shape: M x D_k)

Idea: leverages the strengths of attention layers without the need for separate query vectors.

Self attention layer



Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$
Query vectors: $\mathbf{q} = \mathbf{x}W_q$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

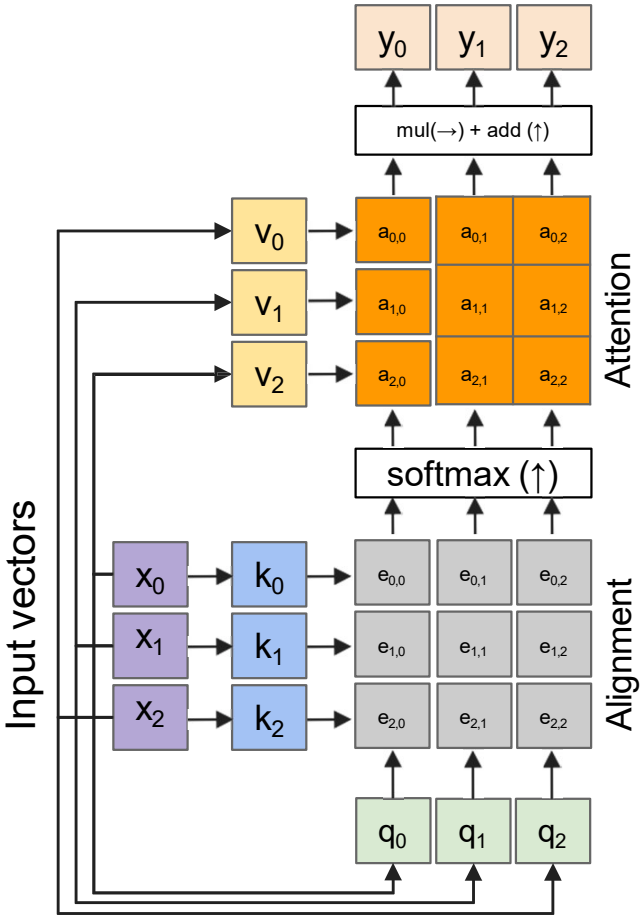
We can calculate the query vectors from the input vectors, therefore, defining a "self-attention" layer.

Instead, query vectors are calculated using a FC layer.

Inputs:
Input vectors: **x** (shape: N x D)
~~Queries: **q** (shape: M x D_k)~~

No input query vectors anymore

Self attention layer

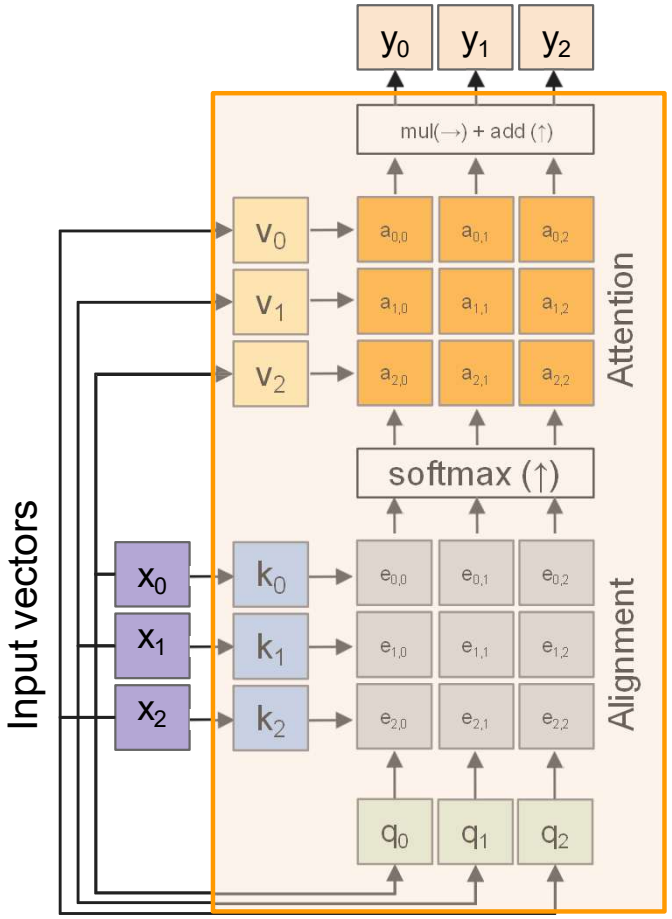


Outputs:
context vectors: **y** (shape: D_v)

Operations:
Key vectors: $\mathbf{k} = \mathbf{x}\mathbf{W}_k$
Value vectors: $\mathbf{v} = \mathbf{x}\mathbf{W}_v$
Query vectors: $\mathbf{q} = \mathbf{x}\mathbf{W}_q$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:
Input vectors: **x** (shape: N x D)

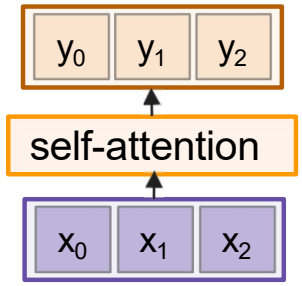
Self attention layer - attends over sets of inputs



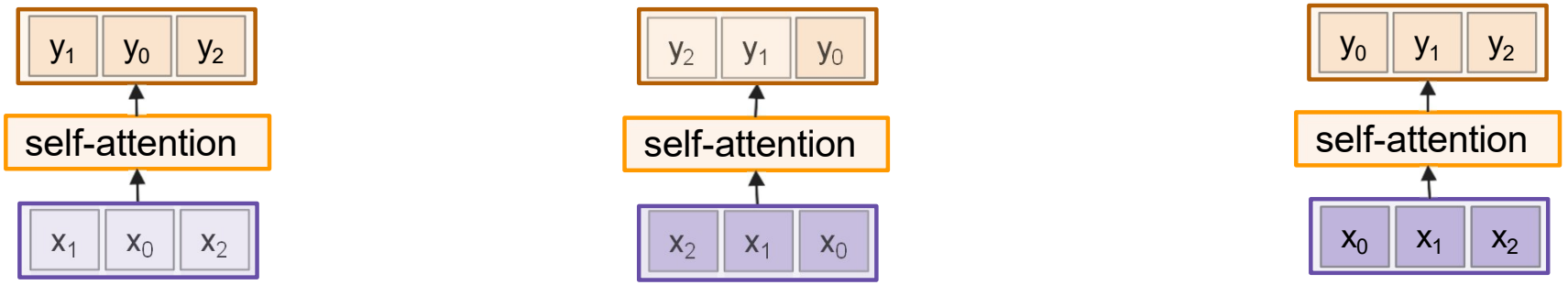
Outputs:
context vectors: **y** (shape: D_v)

Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$
Query vectors: $\mathbf{q} = \mathbf{x}W_q$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:
Input vectors: **x** (shape: N x D)



Self attention layer - attends over sets of inputs



Permutation equivariant

Self-attention layer doesn't care about the orders of the inputs!

Problem: How can we encode ordered sequences like language or spatially ordered image features?

Positional encoding

- The transformer so far ignores the sequential order information of the input sequence (i.e., the order of the words in a sentence)
- However: position and order of words are the essential parts of any language. They define grammar and semantics.
- The attention block in the transformer is invariant to the permutation of Key/Values for a give query. It can't tell the difference between the sequence of words of different ordering.
- *E.g., I eat apple* versus *Apple eat me*. Given the same query, the output of transformer will be the same.

Image source

Positional encoding

- Need to insert position information
- Positional embedding: a function mapping positions (of words) into vectors
- Ideally, the function should have the **following properties**:
  - It should output **a unique encoding** for each time-step (word's position in a sentence)
  - Distance between any two time-steps should be consistent across sentences with different lengths.
  - The model should generalize to longer sentences without any efforts. Its values should be bounded.
  - It must be deterministic.

Positional encoding

- To give transformer information about ordering of tokens, add function of position (based on sines and cosines) to every input

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

where

$$\omega_k = \frac{1}{10000^{2k/d}}$$

- Note: this sin and cos functions satisfy all the requirements of the positional encoding function.

The intuition

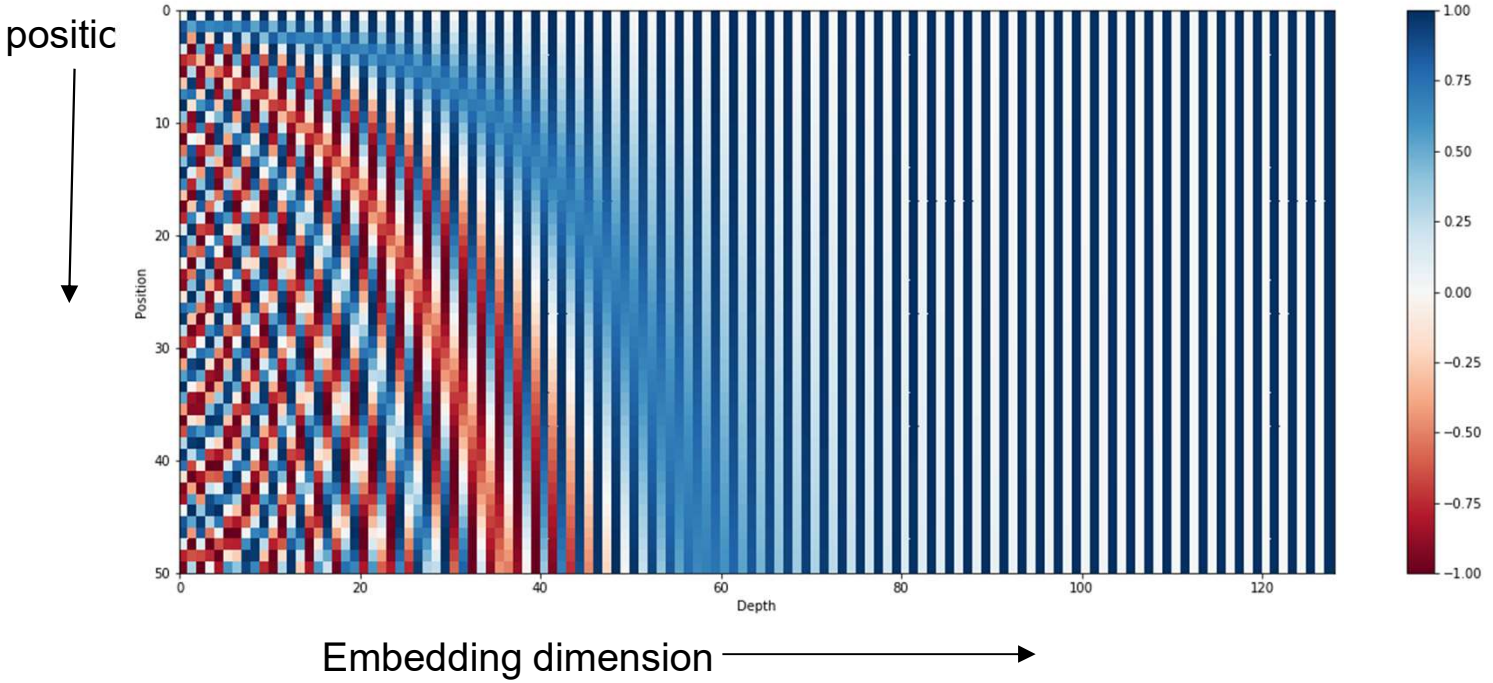
- You may wonder how this combination of sines and cosines could ever represent a position/order? It is actually quite simple,

0 :	0	0	0	0	8 :	1	0	0	0
1 :	0	0	0	1	9 :	1	0	0	1
2 :	0	0	1	0	10 :	1	0	1	0
3 :	0	0	1	1	11 :	1	0	1	1
4 :	0	1	0	0	12 :	1	1	0	0
5 :	0	1	0	1	13 :	1	1	0	1
6 :	0	1	1	0	14 :	1	1	1	0
7 :	0	1	1	1	15 :	1	1	1	1

But using binary values would be a waste of space in the world of floats!!

Positional encoding

- To give transformer information about ordering of tokens, add function of position (based on sines and cosines) to every input



The 128-dimensional positonal encoding for a sentence with the maximum lenght of 50. Each row represents the embedding vector

Image source

Positional encoding

- **Observation**: Frequencies decreasing along depth for the give position. This is consistent with the trend in the positional binary coding.

- **Another property**: It allows the model to attend relative positions effortlessly. since for any fixed offset k, PEpos+k can be represented as a linear function of PEpos.
- Refer to https://blog.timodenk.com/linear-relationships-in-the-transformers-positional-encoding/ for more details and proof.

Image source

Positional encoding

How to fuse the positional embedding to the token (e.g. words) embedding?

1. Add the positional embedding to the token embedding, requires the same number of the dimensions; but could save memories
2. Concatenate the positional embedding to the token embedding, dimensions can be different, however, require more memory spaces

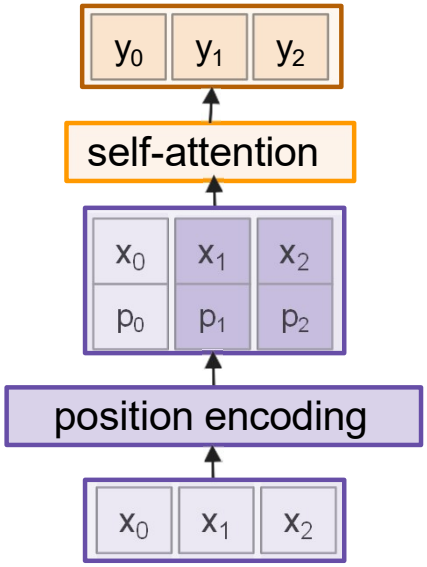
Positional embeding

Positional encoding is crucial for good performance of both text-transformer and vision-transformers.

1. So far we have introduced the **pre-defined position encoding** technique, which sometimes is called: **absolute positional** encoding. I.e., the mapping function (sin and cos) is fixed.
2. Positional embedding are also **learnable** (see the original Transformer paper) . Once learned, it is then fixed; **cause great difficulties in handling sequences longer than that seen in training**.
3. Other positional encodings:
    1. **Relative positional encodings** for text (by Shaw et al 2018) and for image (Bello etal 2019).
    2. **Complex-value encodings** (wang et al 2019). Rotary encodings in Roformer (Su et al 2021)
    3. **Conditional positional encoding** (Chu et al, ICLR2023).

       For details see: Chu et al, Conditional Positional Encodings for Vision Transformers, ICLR 2023

Positional encoding



Concatenate or **add** special positional encoding p_j to each input vector x_j

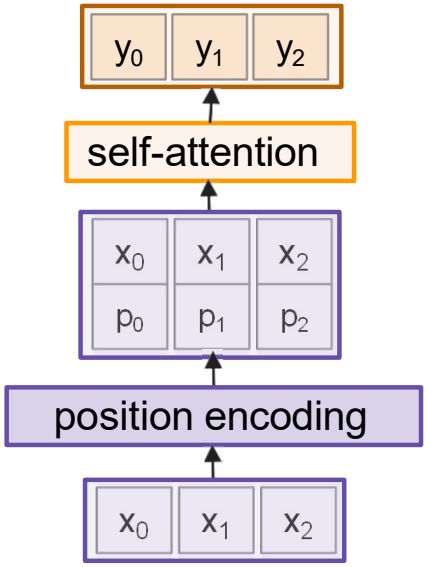
We use a function *pos*: $N \rightarrow R^d$
to process the position j of the vector
into a d-dimensional vector

So, $p_j = pos(j)$

Possible desirable properties of *pos*(.) :

1. It should output a **unique** encoding for each time-step (word's position in a sentence)
2. **Distance** between any two time-steps should be consistent across sentences with different lengths.
3. Our model should generalize to **longer** sentences without any efforts. Its values should be bounded.
4. It must be **deterministic**.

Positional encoding



Concatenate special positional encoding p_j to each input vector x_j

We use a function *pos*: N $\rightarrow R^d$ to process the position j of the vector into a d-dimensional vector

So, p_j = *pos*(j)

Options for *pos*(.)

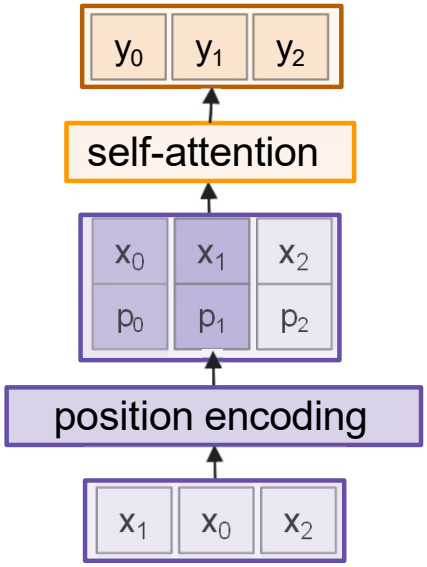
1. Learn a lookup table:
   - Learn parameters to use for *pos*(t) for t ε [0, T)
   - Lookup table contains T x d parameters.

Possible desirable properties of *pos*(.) :

1. It should output a **unique** encoding for each time-step (word's position in a sentence)
2. **Distance** between any two time-steps should be consistent across sentences with different lengths.
3. Our model should generalize to **longer** sentences without any efforts. Its values should be bounded.
4. It must be **deterministic**.

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

Positional encoding



Concatenate special positional encoding p_j to each input vector x_j

We use a function *pos*: N $\rightarrow R^d$ to process the position j of the vector into a d-dimensional vector

So, p_j = *pos*(j)

Options for *pos*(.)

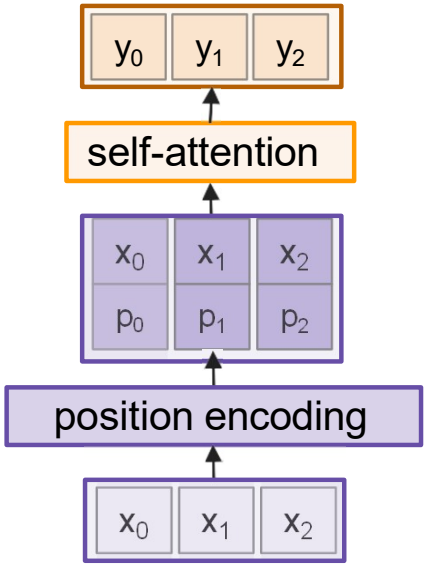
1. Learn a lookup table:
   - Learn parameters to use for *pos*(t) for t ε [0, T)
   - Lookup table contains T x d parameters.

2. Design a fixed function with the desired properties

$$\mathbf{p(t)} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_d \quad \text{where } \omega_k = \frac{1}{10000^{2k/d}}$$

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

Positional encoding



Concatenate special positional encoding p_j to each input vector x_j

We use a function *pos*: N $\rightarrow \mathbb{R}^d$ to process the position j of the vector into a d-dimensional vector

So, p_j = *pos*(j)

Options for *pos*(.)

1. Learn a lookup table:
   - Learn parameters to use for *pos*(t) for t ε [0, T)
   - Lookup table contains T x d parameters.

2. Design a fixed function with the desired properties

$$\mathbf{p(t)} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \vdots \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_d$$

Intuition:

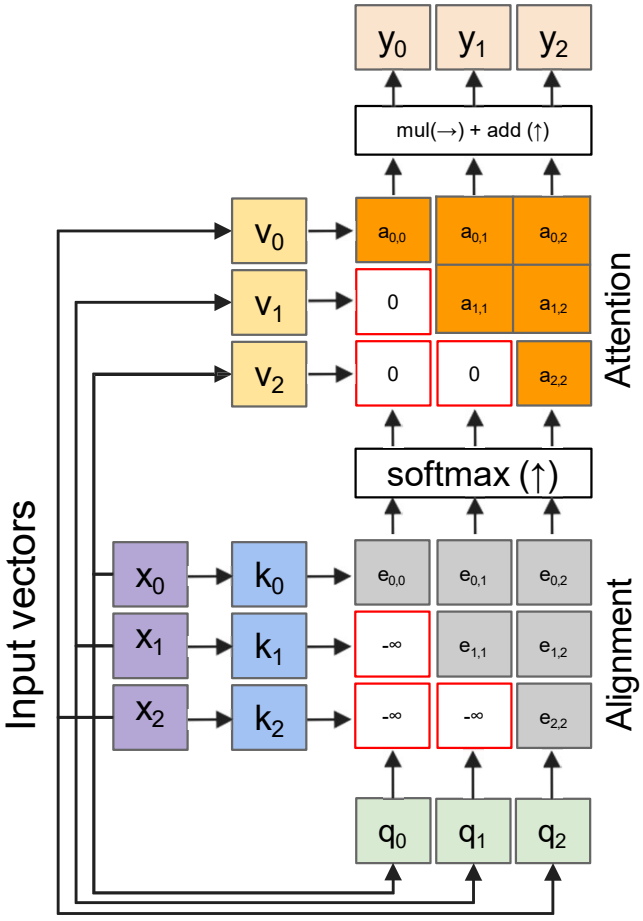
0:	0	0	0	0	8:	1	0	0	0
1:	0	0	0	1	9:	1	0	0	1
2:	0	0	1	0	10:	1	0	1	0
3:	0	0	1	1	11:	1	0	1	1
4:	0	1	0	0	12:	1	1	0	0
5:	0	1	0	1	13:	1	1	0	1
6:	0	1	1	0	14:	1	1	1	0
7:	0	1	1	1	15:	1	1	1	1

where $\omega_k = \frac{1}{10000^{2k/d}}$

image source

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

Masked self-attention layer



Outputs:
context vectors: **y** (shape: D_v)

Operations:
Key vectors: $\mathbf{k} = \mathbf{x}W_k$
Value vectors: $\mathbf{v} = \mathbf{x}W_v$
Query vectors: $\mathbf{q} = \mathbf{x}W_q$
Alignment: $e_{i,j} = q_j \cdot k_i / \sqrt{D}$
Attention: $\mathbf{a} = \text{softmax}(\mathbf{e})$
Output: $y_j = \sum_i a_{i,j} v_i$

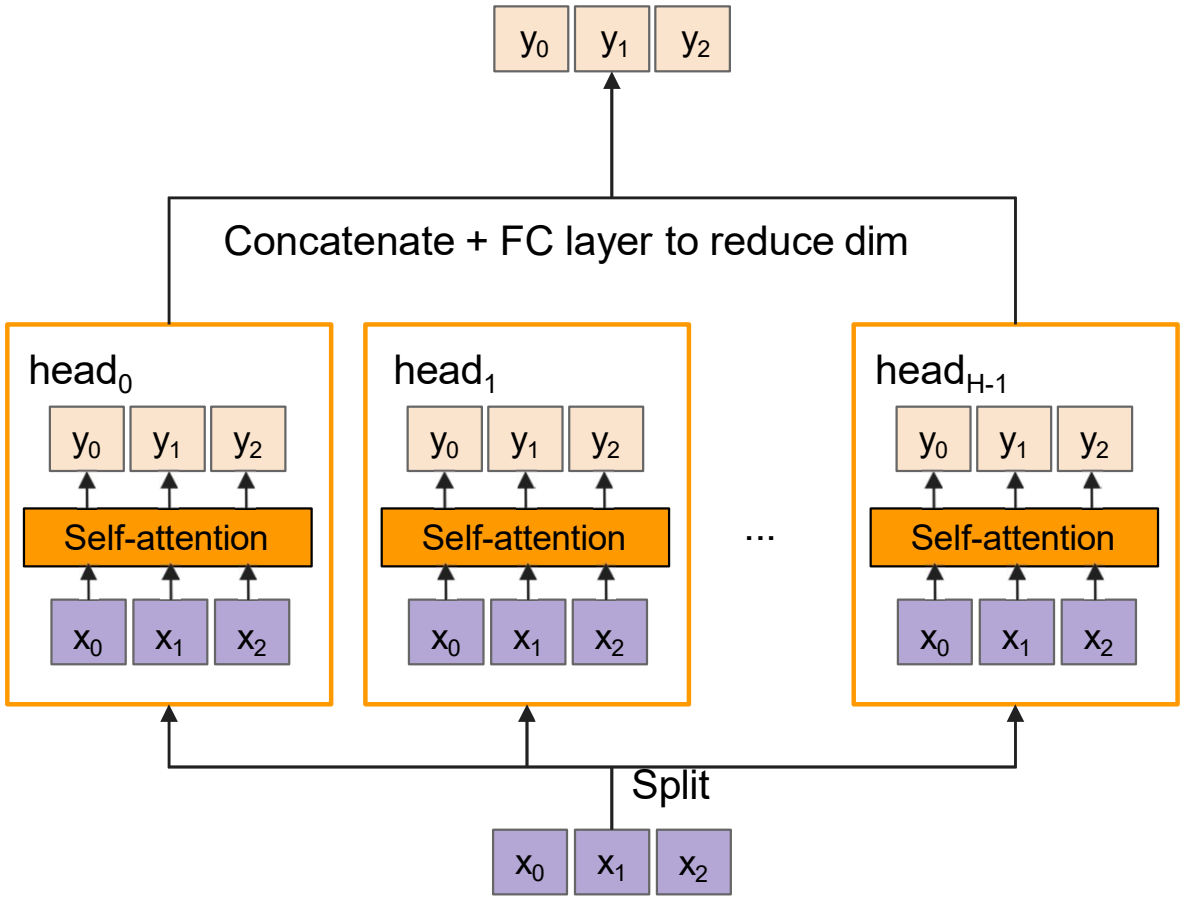
Inputs:
Input vectors: **x** (shape: N x D)

- Allows us to parallelize attention across time
- Don't need to calculate the context vectors from the previous timestep first!

- Prevent vectors from looking at future vectors.
- Manually set alignment scores to –infinity (-nan)

Multi-head self-attention layer

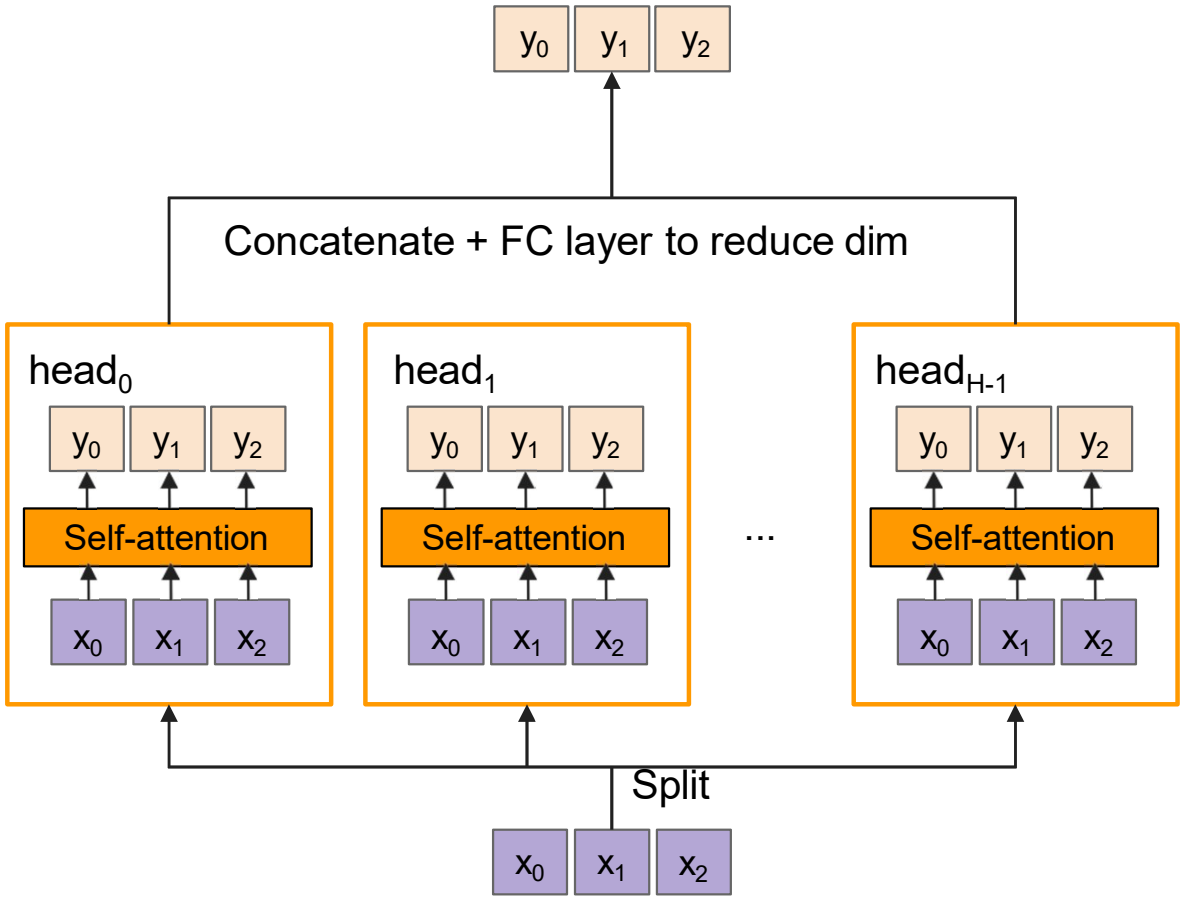
- Multiple self-attention “heads” in parallel



Q: Why do this?

Multi-head self-attention layer

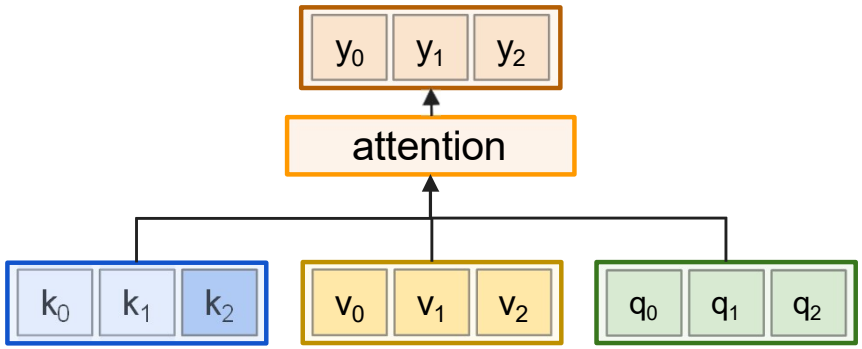
- Multiple self-attention “heads” in parallel

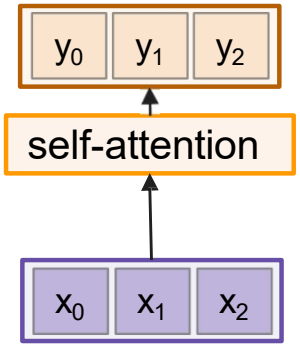


A: We may want to have multiple sets of queries/keys/values calculated in the layer. This is a similar idea to having multiple conv filters learned in a layer

General attention versus self-attention

Transformer models rely on many, stacked self-attention layers





Comparing RNNs to Transformer

RNNs

(+) LSTMs work reasonably well for long sequences. (-) Expects an ordered sequences of inputs

(-) Sequential computation: subsequent hidden states can only be computed after the previous ones are done.

Transformer:

(+) Good at long sequences. Each attention calculation looks at all inputs.

(+) Can operate over unordered sets or ordered sequences with positional encodings.

(+) Parallel computation: All alignment and attention scores for all inputs can be done in parallel. (-) Requires a lot of memory: N x M alignment and attention scalers need to be calculated and stored for a single self-attention head. (but GPUs are getting bigger and better)

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

"ImageNet Moment for Natural Language Processing"

Pretraining:
Download a lot of text from the internet

Train a giant Transformer model for language modeling

Finetuning:
Fine-tune the Transformer on your own NLP task