Attention and Transformers

M. Soleymani
Sharif University of Technology
Spring 2024

Most slides have been adopted from Fei Fei Li and colleagues lectures cs231n, Stanford and Manning & Hewitt lectures cs224n, Stanford

Sequence to Sequence with RNNs

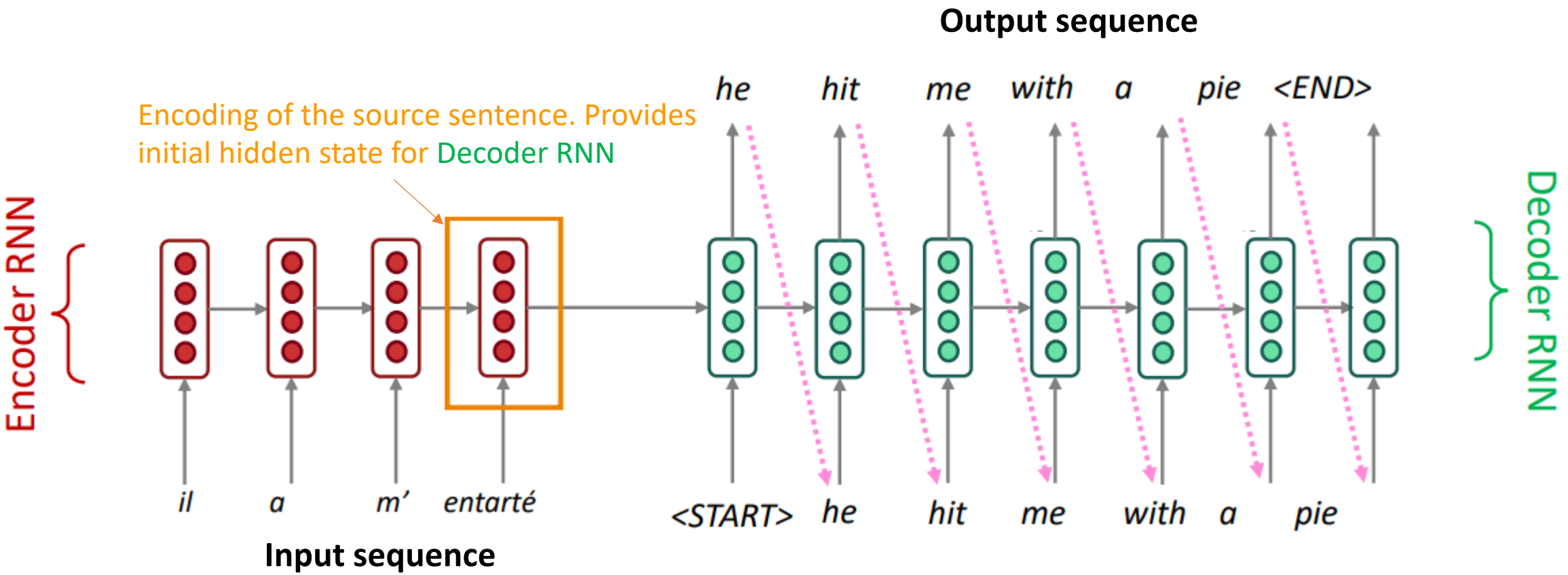
- The seq2seq model was born in the field of language modeling (Sutskever, et al. 2014).
- It aims to transform an input sequence (source) to a new one (target)
  - Both sequences can be of arbitrary lengths

- Many NLP tasks can be phrased as sequence-to-sequence:
  - Summarization (long text → short text)
  - Dialogue (previous utterances → next utterance)
  - Parsing (input text → output parse as sequence)
  - Code generation (natural language → Python code)

Sequence to Sequence with RNNs

- The seq2seq model normally has an encoder-decoder architecture, composed of:
  - An **encoder** processes the input sequence and compresses the information into a context vector (also known as sentence embedding or "thought" vector) of a *fixed length*
  - A **decoder** is initialized with the context vector
    - The early work only used the last state of the encoder network as the decoder initial state
  - Both the encoder and decoder are recurrent neural networks, i.e. LSTM or GRU units

Seq-2-Seq model: Machine Translation example



Encoder RNN produces an encoding of the source sentence.

Decoder RNN is a Language Model that generates target sentence, conditioned on encoding

Note: This diagram shows test time behaviour: decoder output is fed in - - - - -> as next step's input

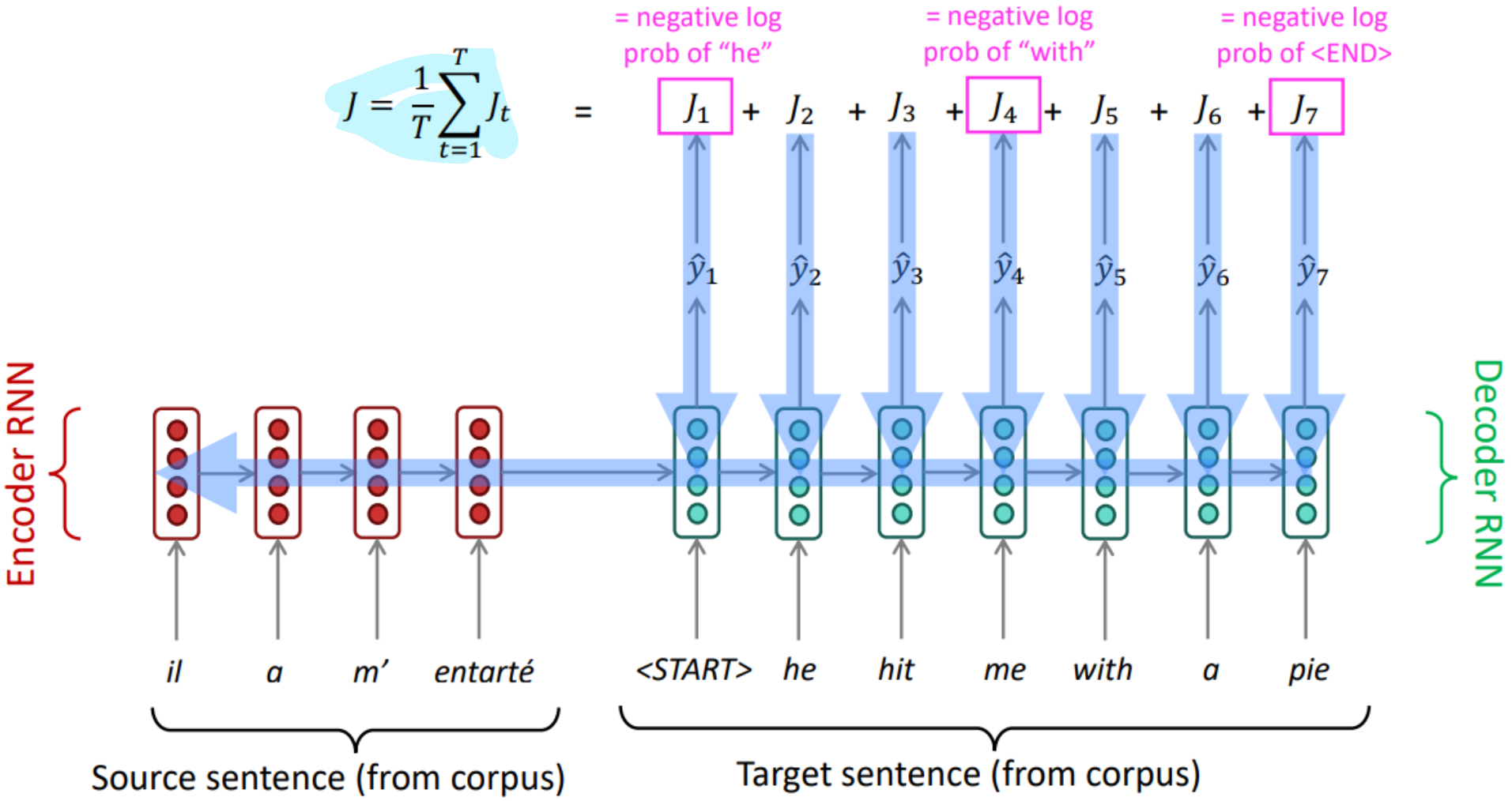
Seq-2-Seq model as Conditional Language Model

- The sequence-to-sequence model is an example of a Conditional Language Model
  - Language Model because the decoder is predicting the next word of the target sentence y
  - Conditional because its predictions are also conditioned on the source sentence x

$$p(y|x) = p(y_1|x)p(y_2|y_1, x) \dots p(y_T|y_1, y_2, \dots y_{T-1}, x)$$

Seq-2-Seq model: Machine Translation example

- The sequence-to-sequence model is an example of a Conditional Language Model (conditioned on the source sentence x)

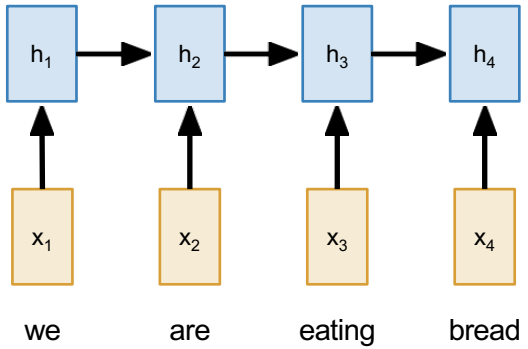


Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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



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

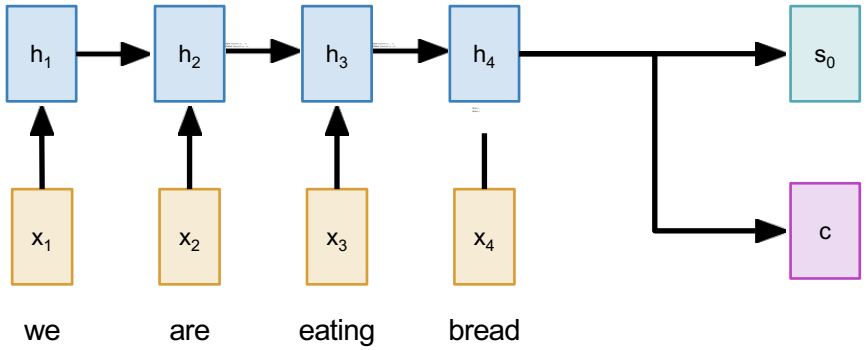
Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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

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



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

Sequence to Sequence with RNNs

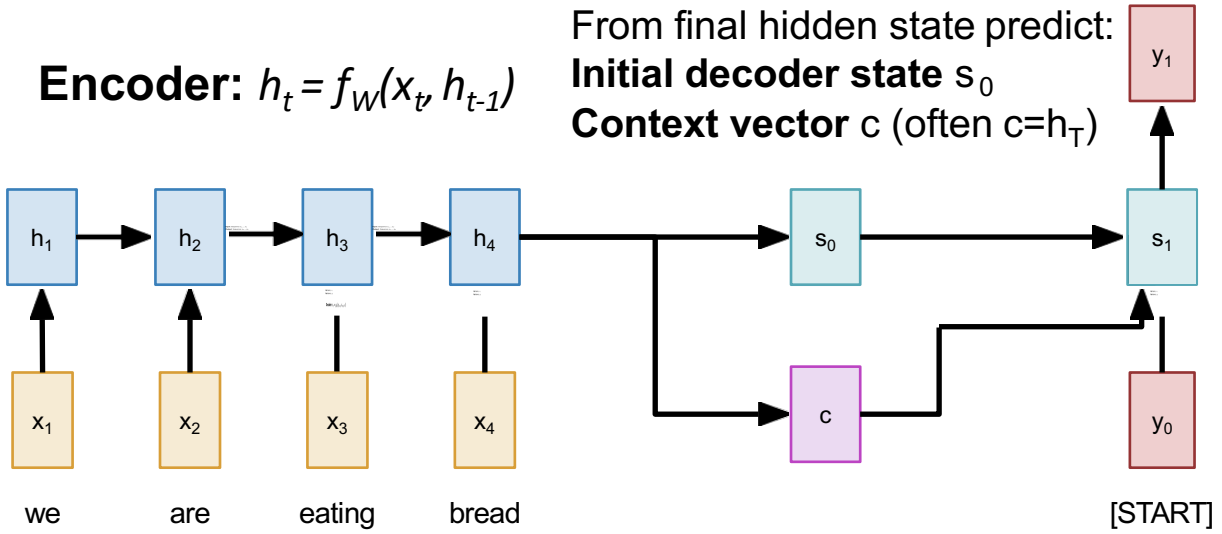
Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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

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

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



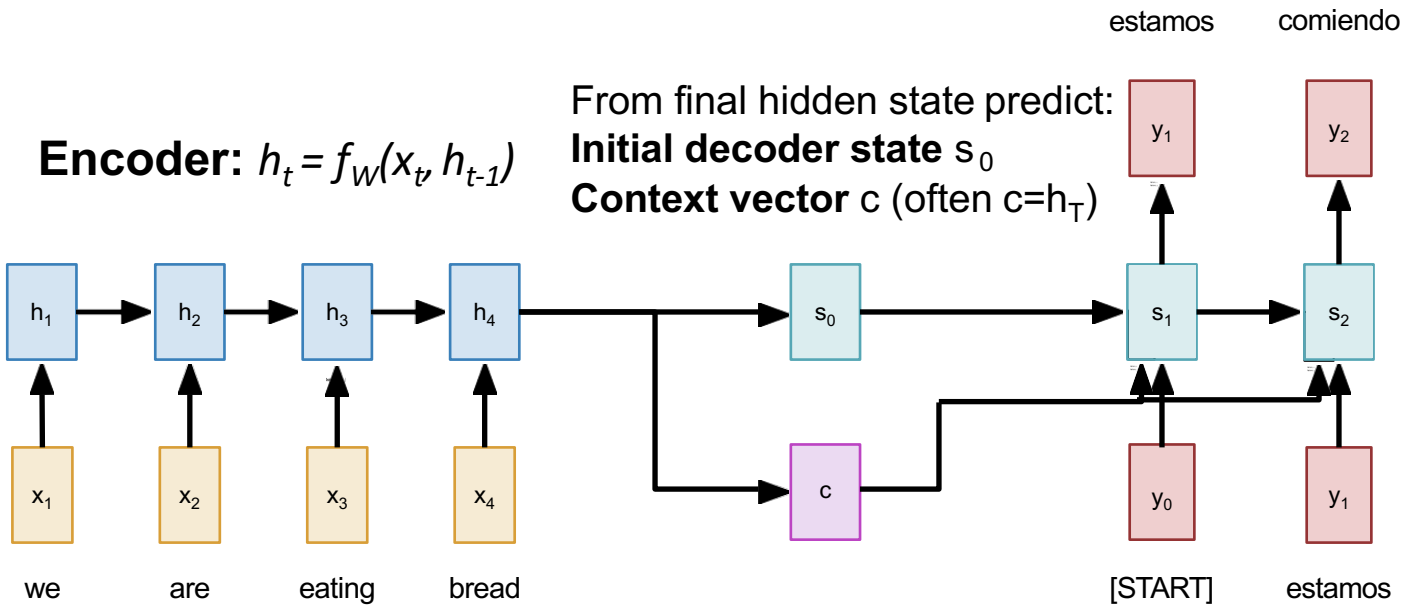
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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



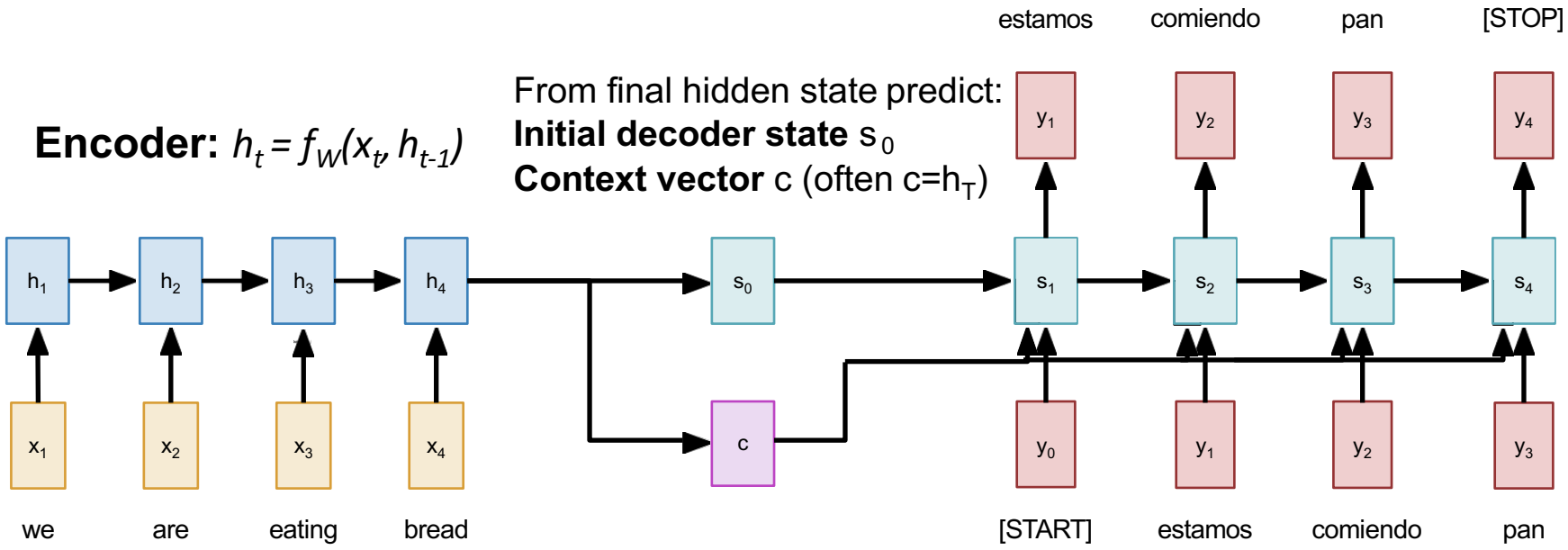
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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



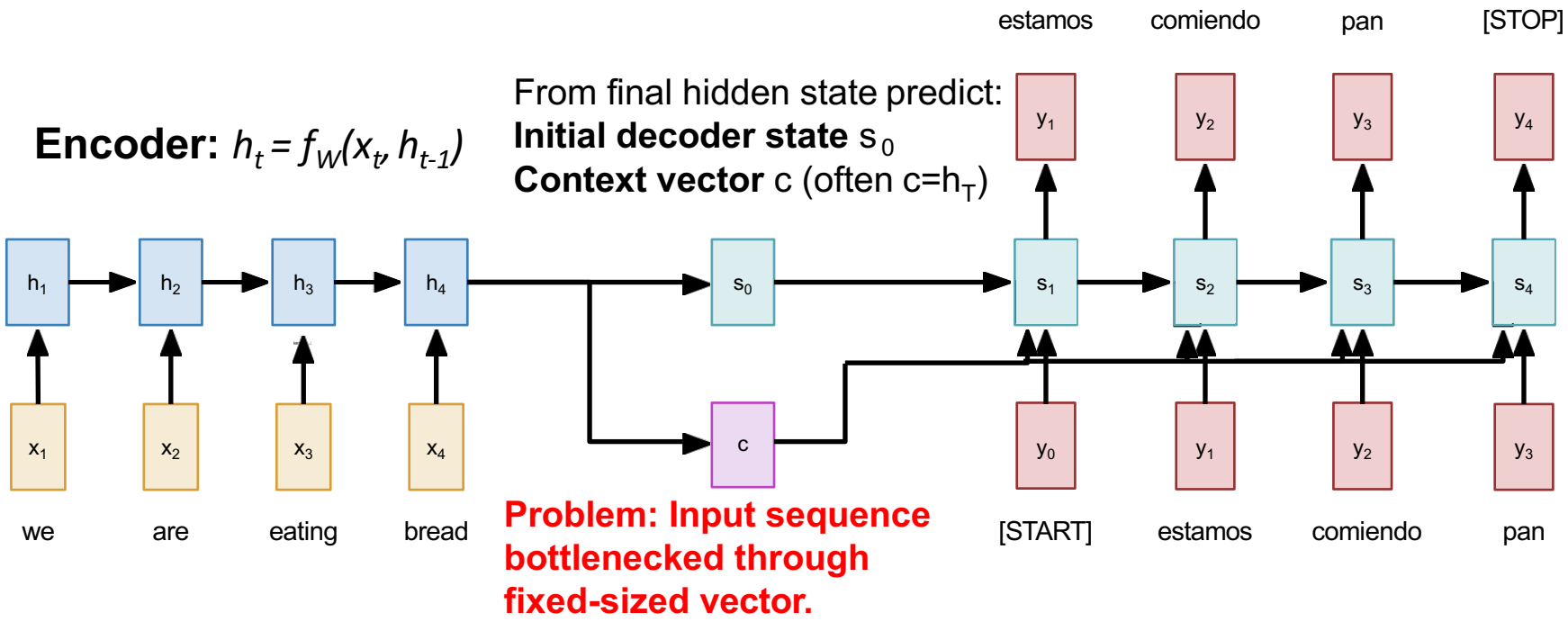
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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



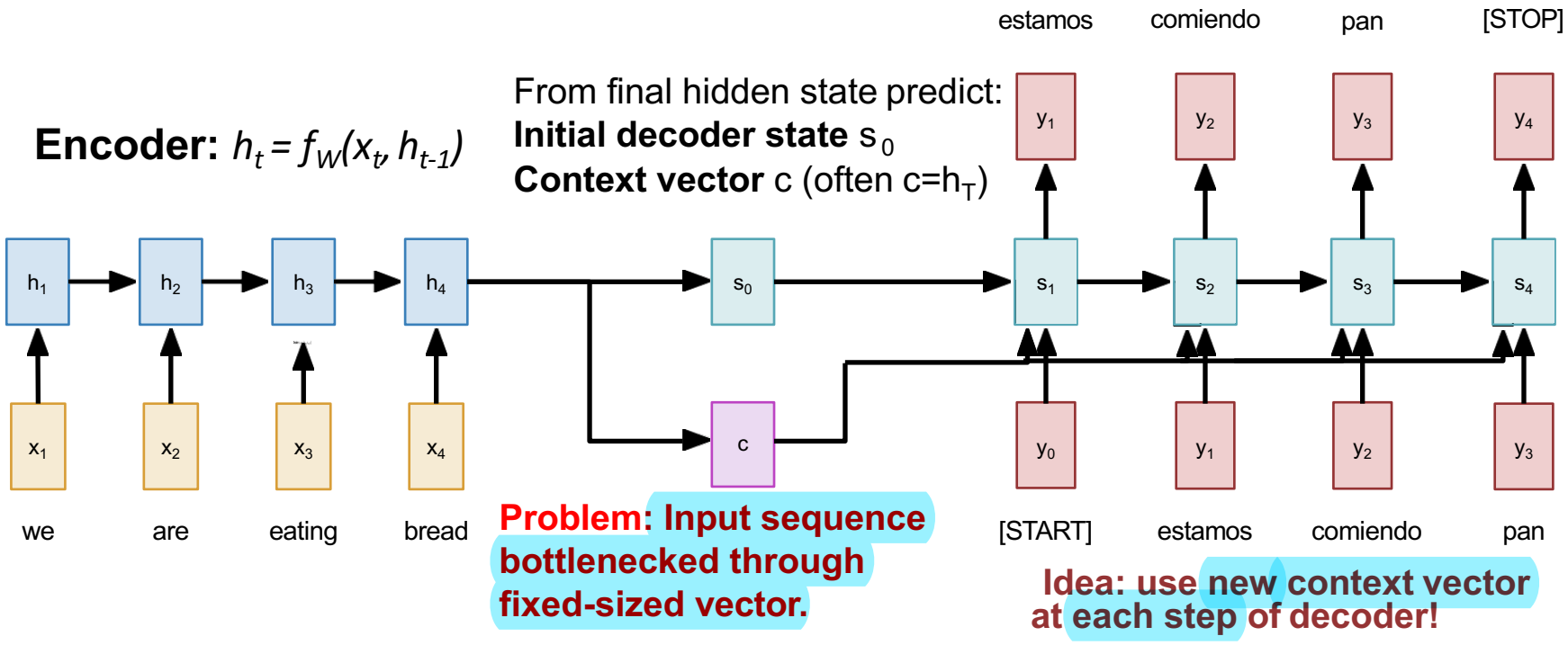
Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs

Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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

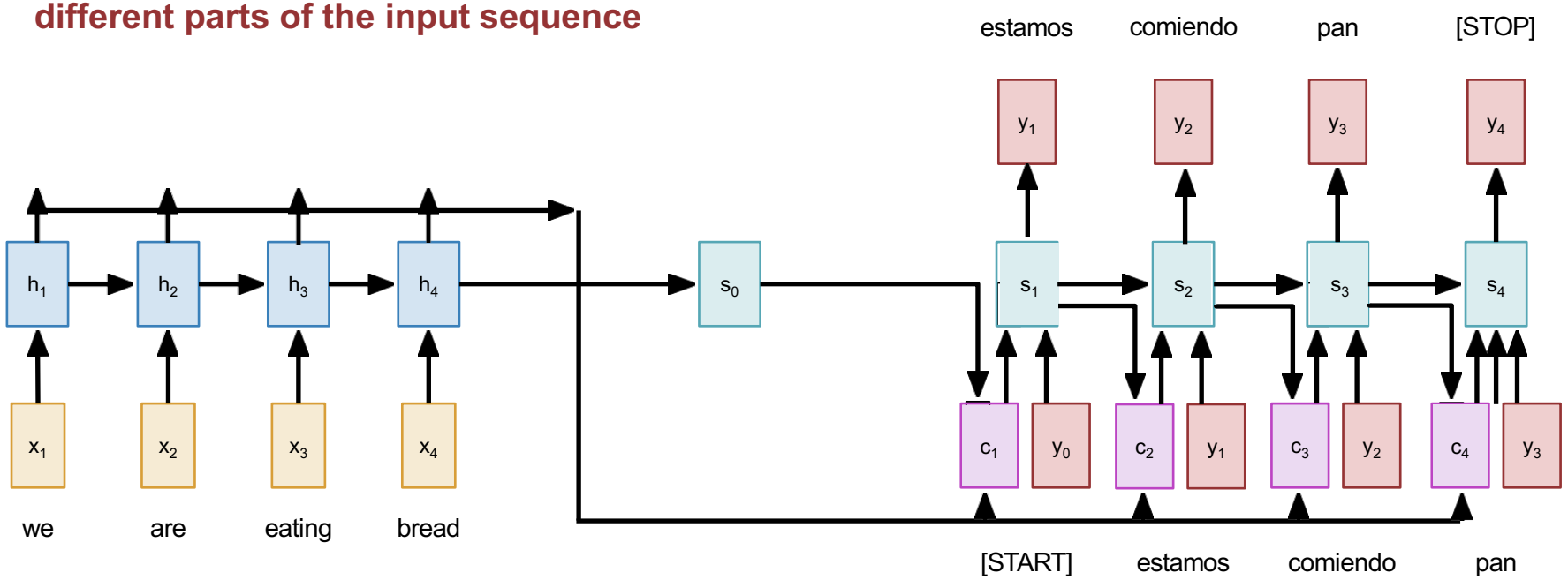


Sutskever et al, "Sequence to sequence learning with neural networks", NeurIPS 2014

Sequence to Sequence with RNNs and Attention

Use a different context vector in each timestep of decoder

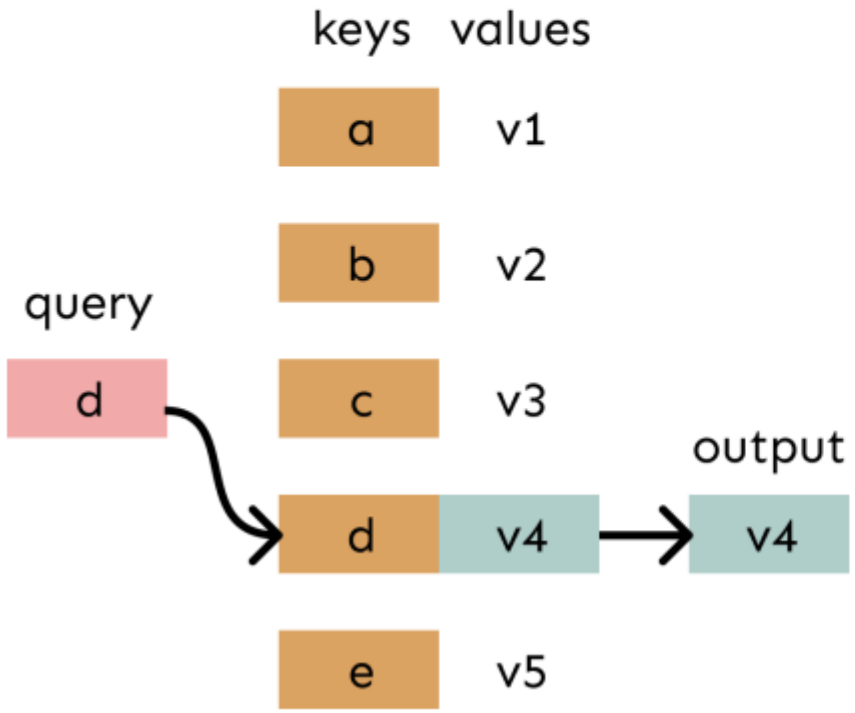
- Input sequence not bottlenecked through single vector
- At each timestep of decoder, context vector "looks at" different parts of the input sequence



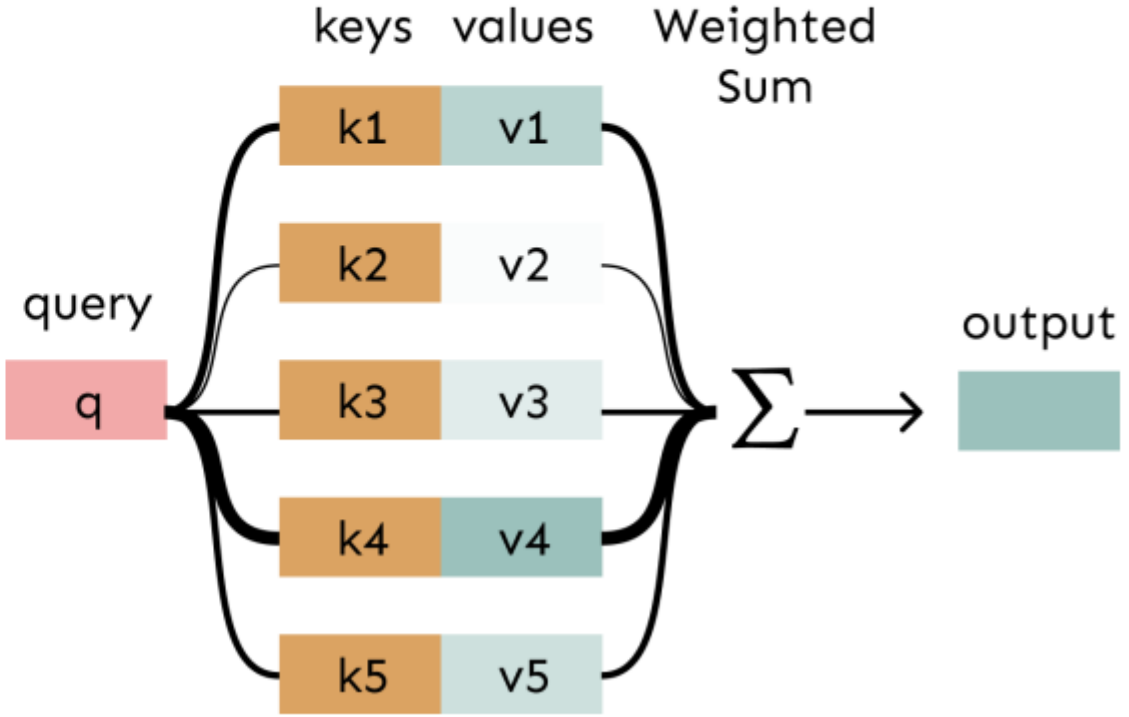
Bahdanau et al, "Neural machine translation by jointly learning to align and translate", ICLR 2015

Attention as a soft, averaging lookup table

In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.



In **attention**, the query matches all keys *softly*, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



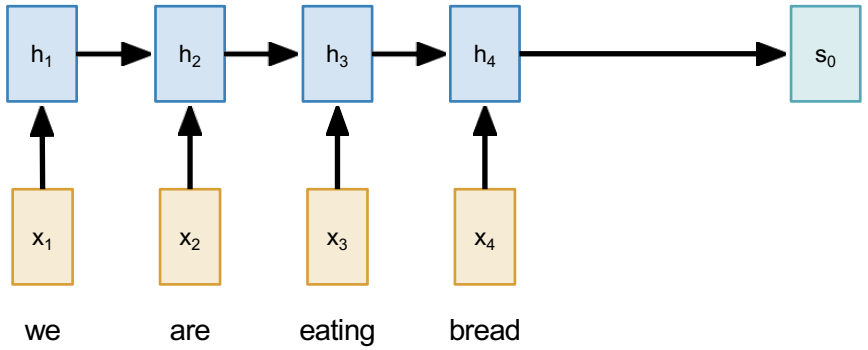
Sequence to Sequence with RNNs and Attention

Input: Sequence $x_1, \ldots x_T$

Output: Sequence $y_1, \ldots y_T$

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

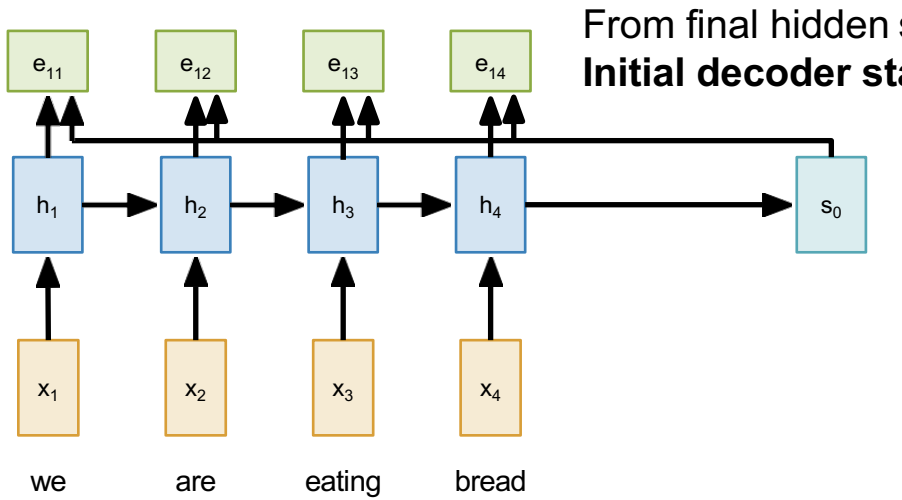
From final hidden state:
Initial decoder state s_0



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

Sequence to Sequence with RNNs and Attention

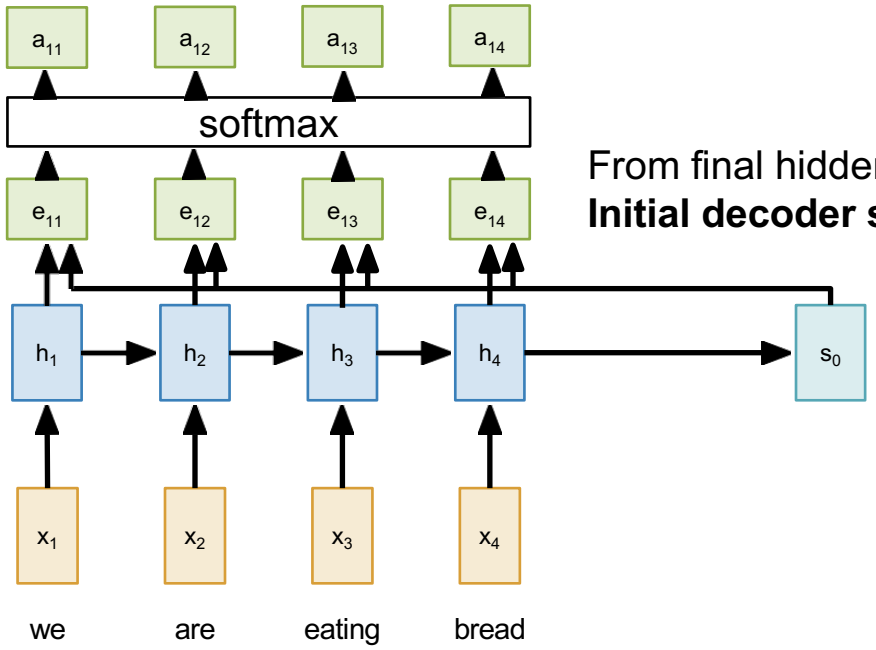
Compute (scalar) **alignment scores** :
$e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is an MLP)



From final hidden state:
Initial decoder state s_0

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

Sequence to Sequence with RNNs and Attention



From final hidden state:
Initial decoder state s_0

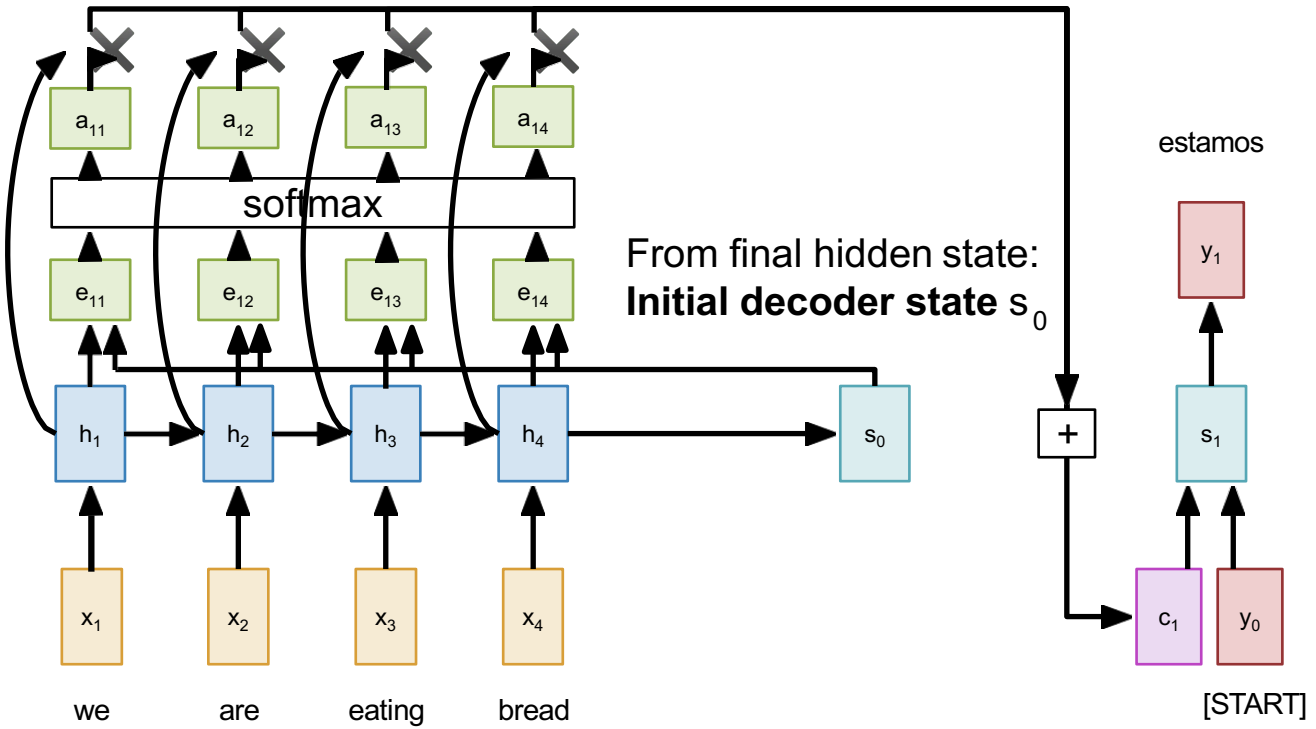
Compute (scalar) **alignment scores** :
$e_{t,i} = f_{att}(s_{t-1}, h_i)$ (f_{att} is an MLP)

Normalize alignment scores to get **attention weights**

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

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 an MLP)

Normalize alignment scores to get **attention weights**

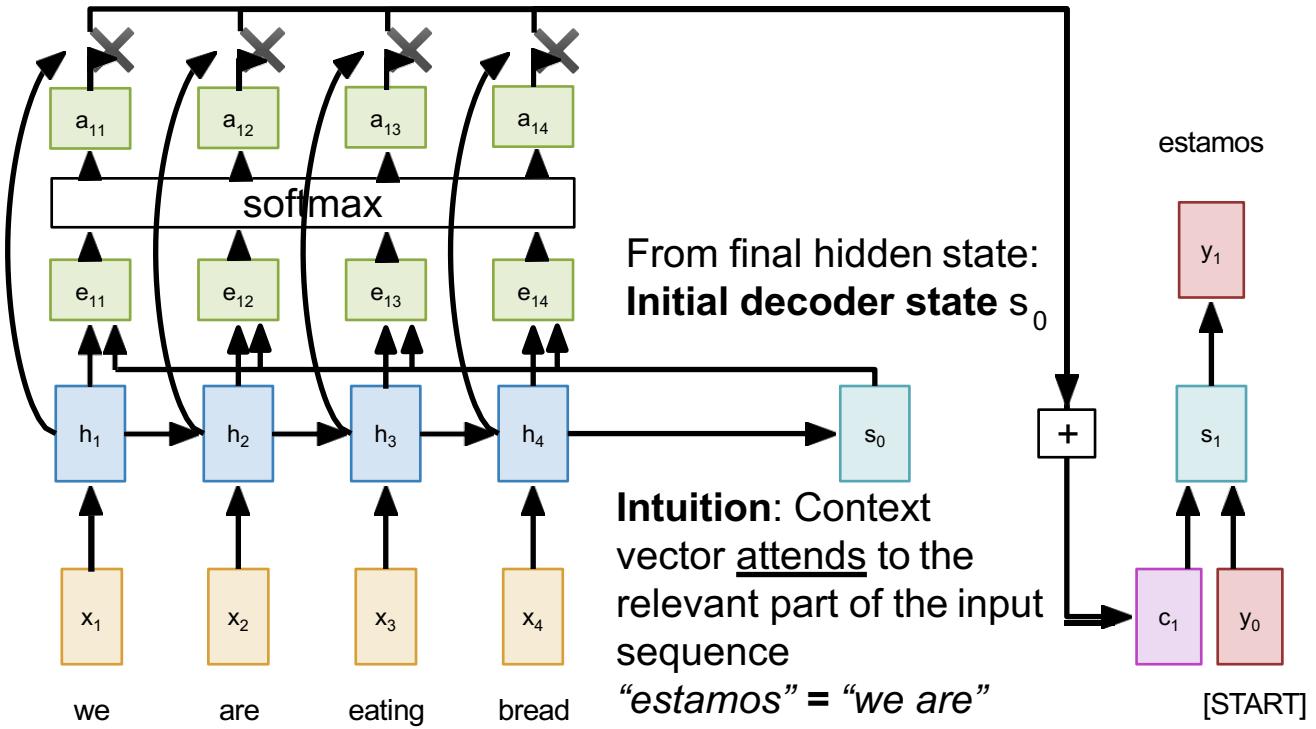
$$0 < a_{t,i} < 1 \qquad \sum_i a_{t,i} = 1$$

Compute context vector as 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

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
"estamos" = *"we are"*
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 an MLP)

Normalize alignment scores to get **attention weights**

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

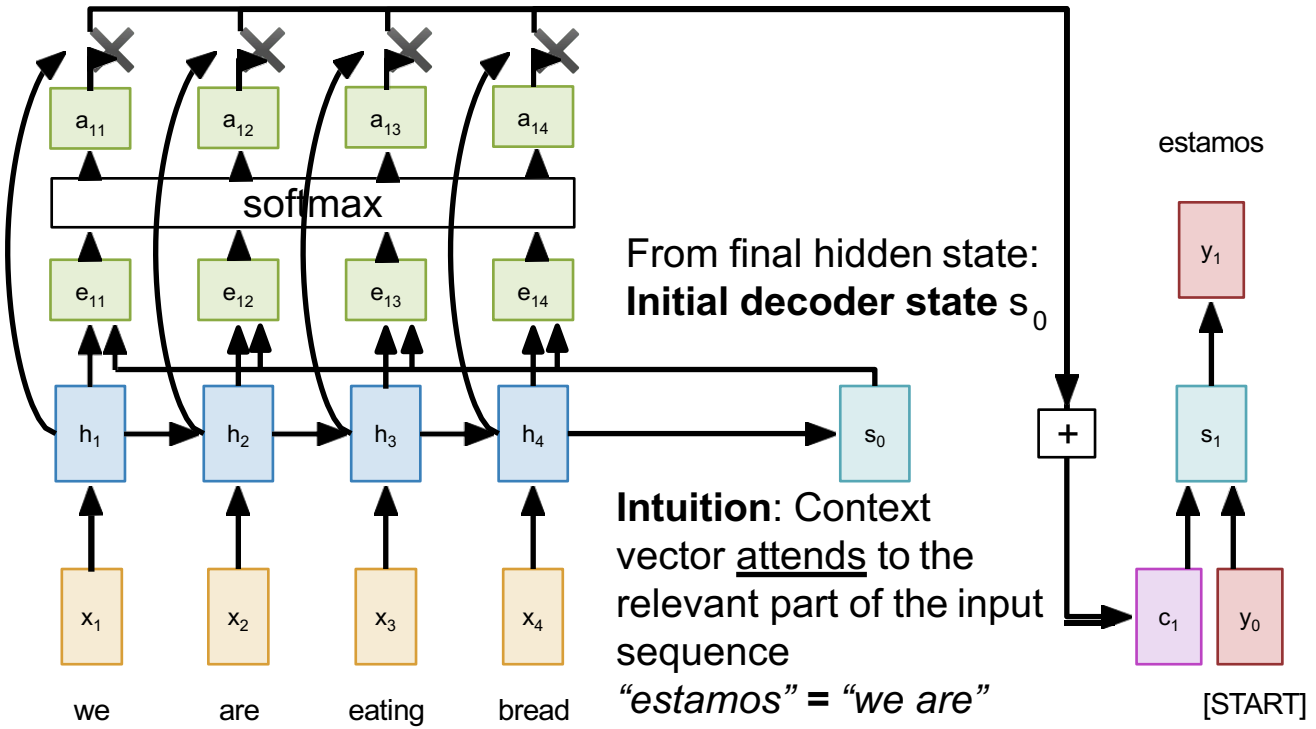
Compute context vector as linear combination of hidden states

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

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

"estamos" = *"we are"*

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 an MLP)

Normalize alignment scores to get **attention weights**

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

Compute context vector as linear combination of hidden states

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

Use context vector in decoder:

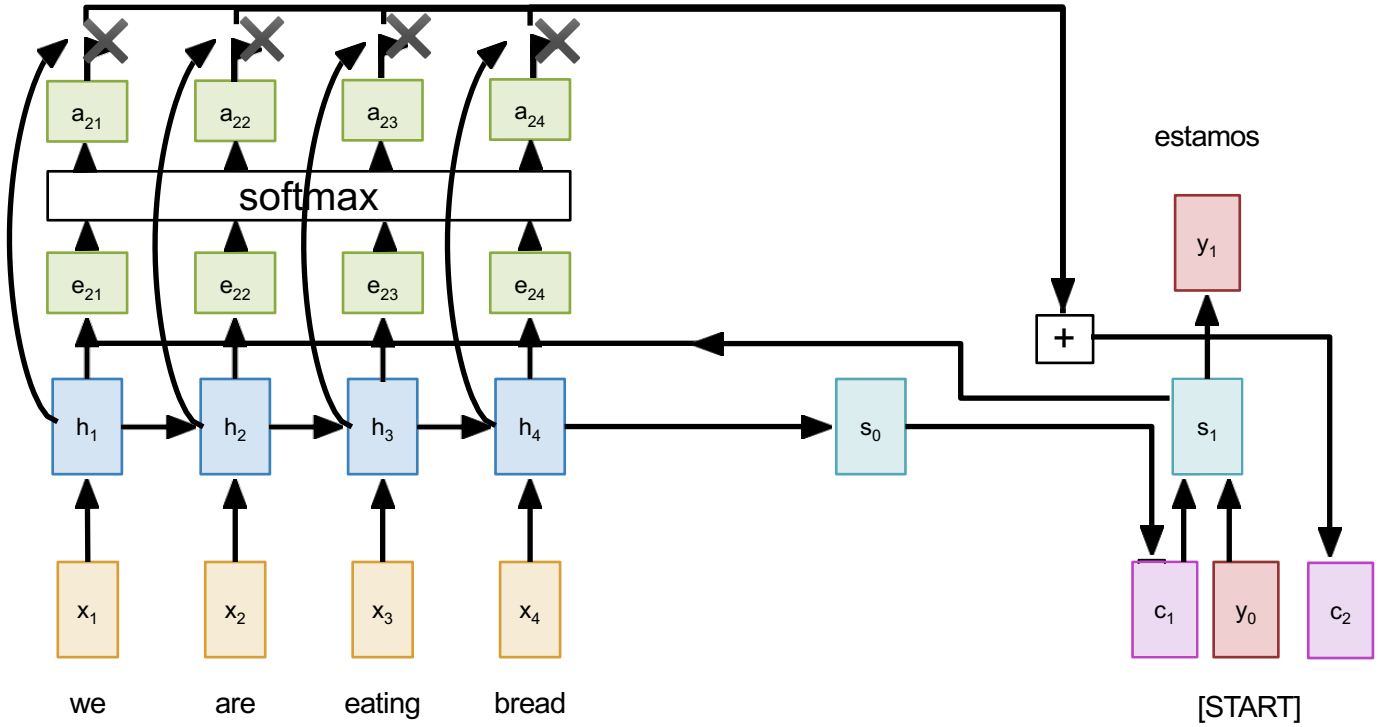
$s_t = g_U(y_{t-1}, s_{t-1}, c_t)$

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

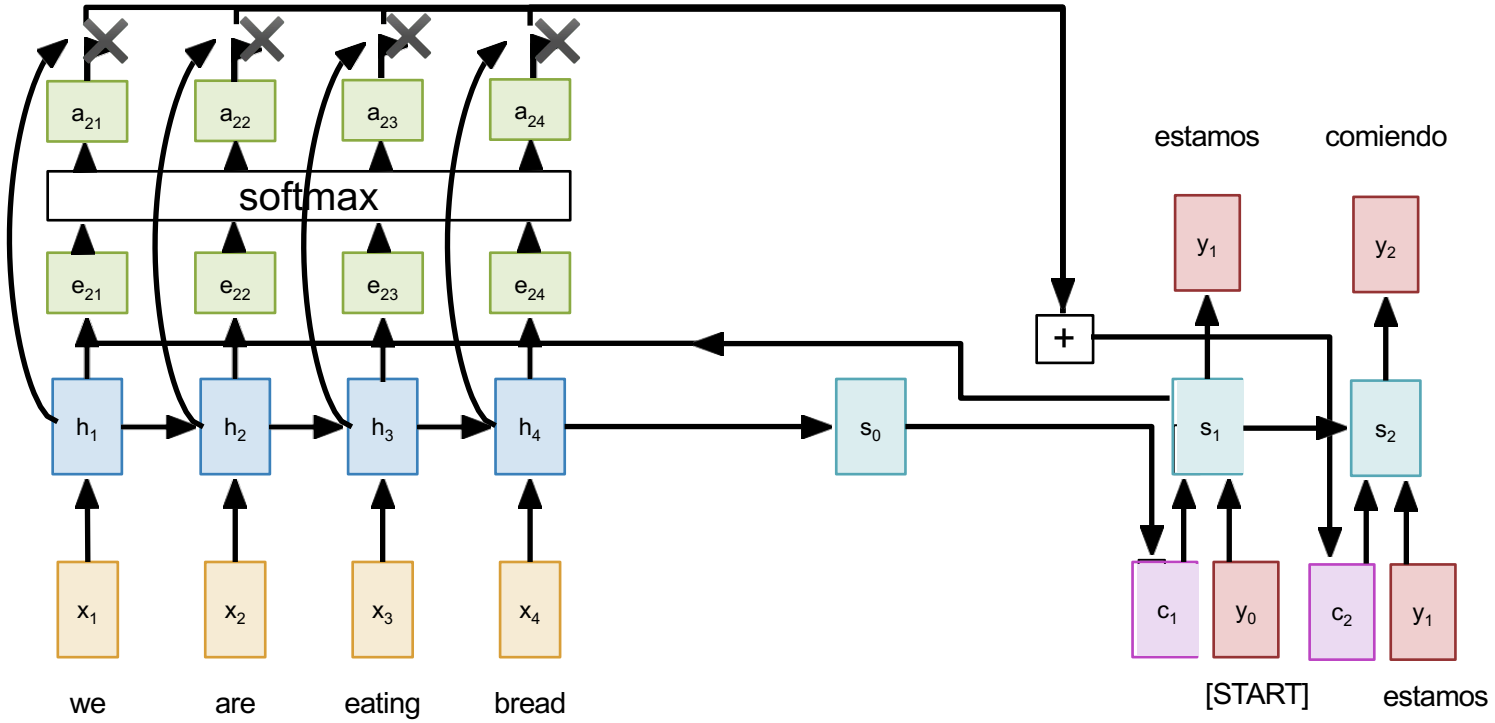
Sequence to Sequence with RNNs and Attention

Repeat: Use s_1 to compute new context vector c_2



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

Sequence to Sequence with RNNs and Attention

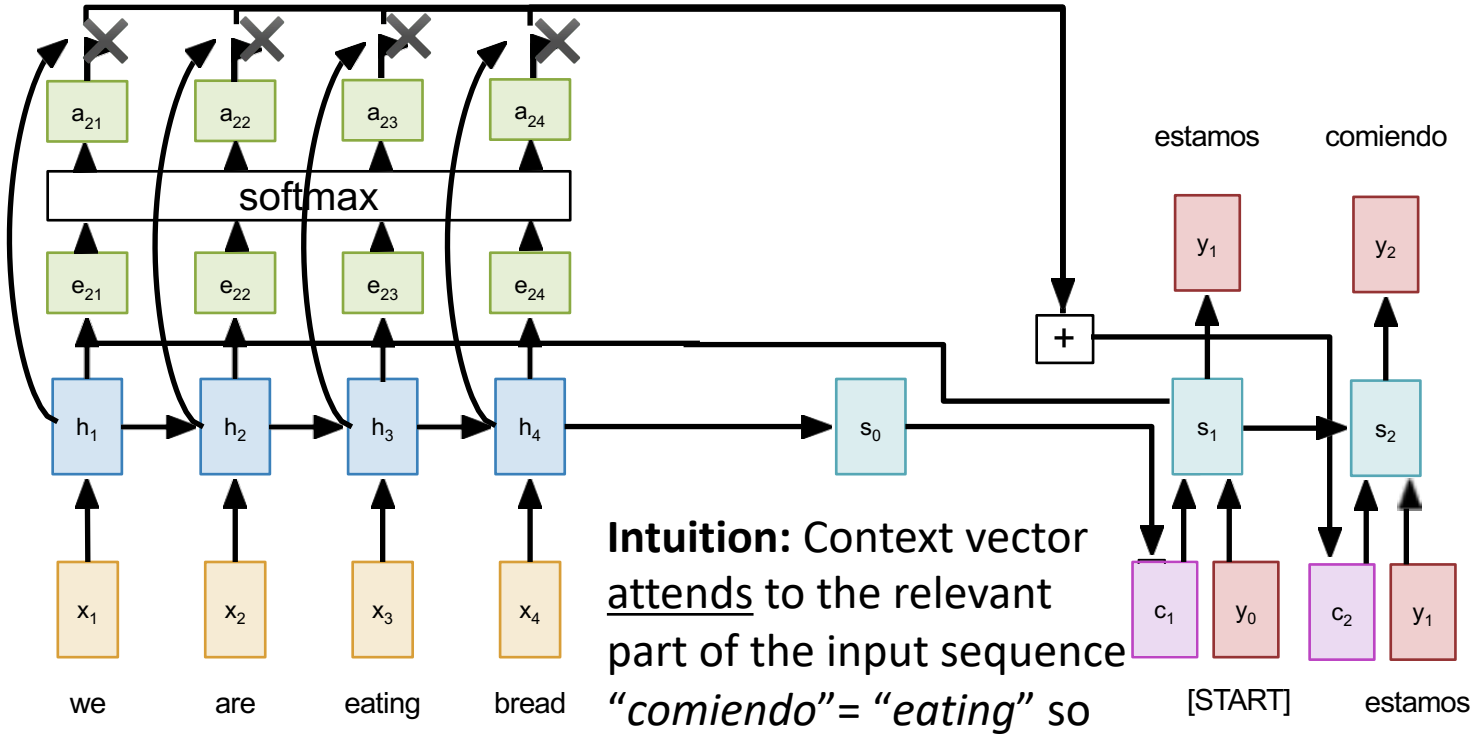


Repeat: Use s_1 to compute new context vector c_2

Use c_2 to compute s_2, y_2

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

Sequence to Sequence with RNNs and Attention



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

Repeat: Use s_1 to compute new context vector c_2

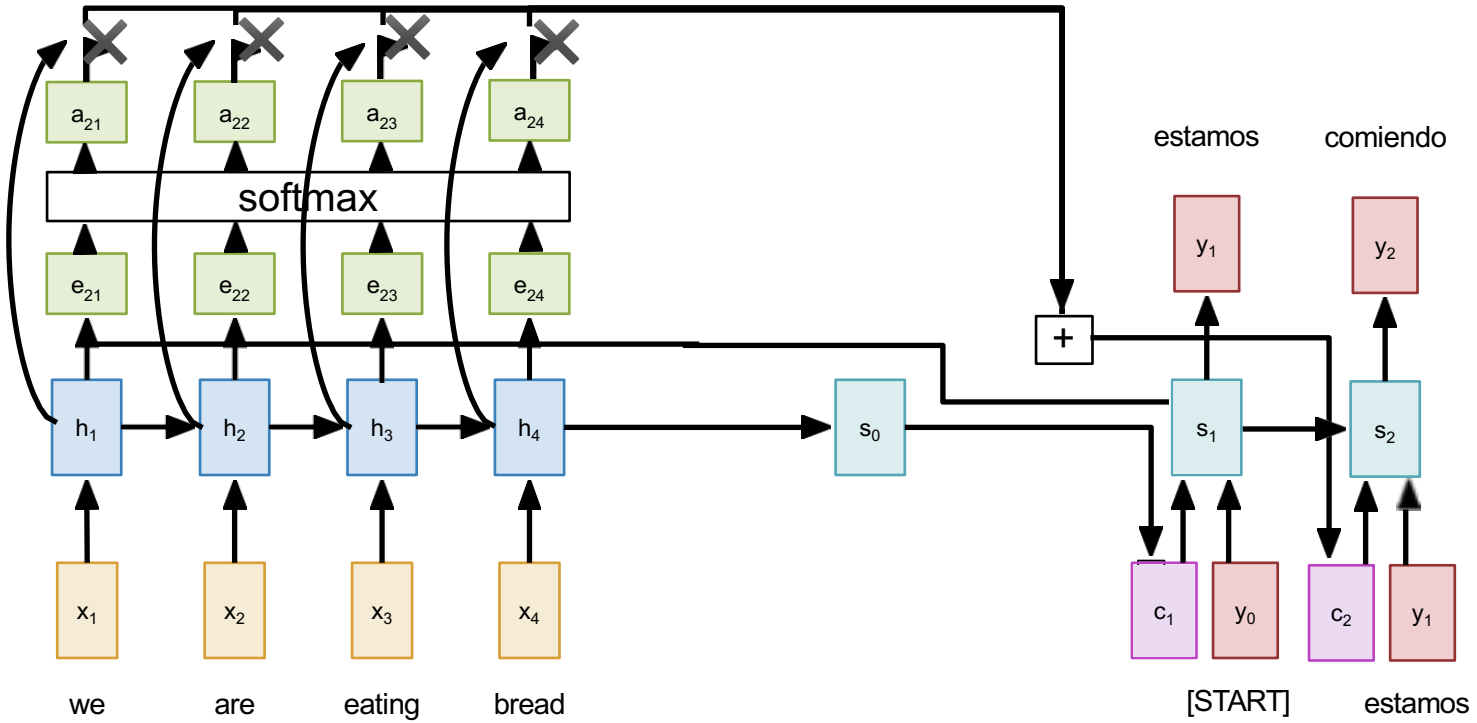
Use c_2 to compute s_2, y_2

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

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

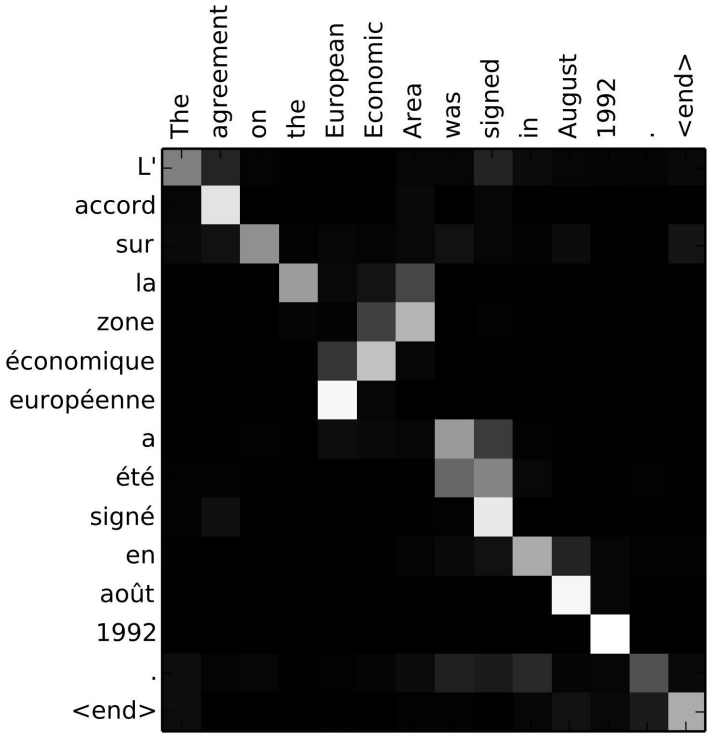
Sequence to Sequence with RNNs and Attention

Example: English to French translation

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

Output: “L'accord sur la zone économique européen 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

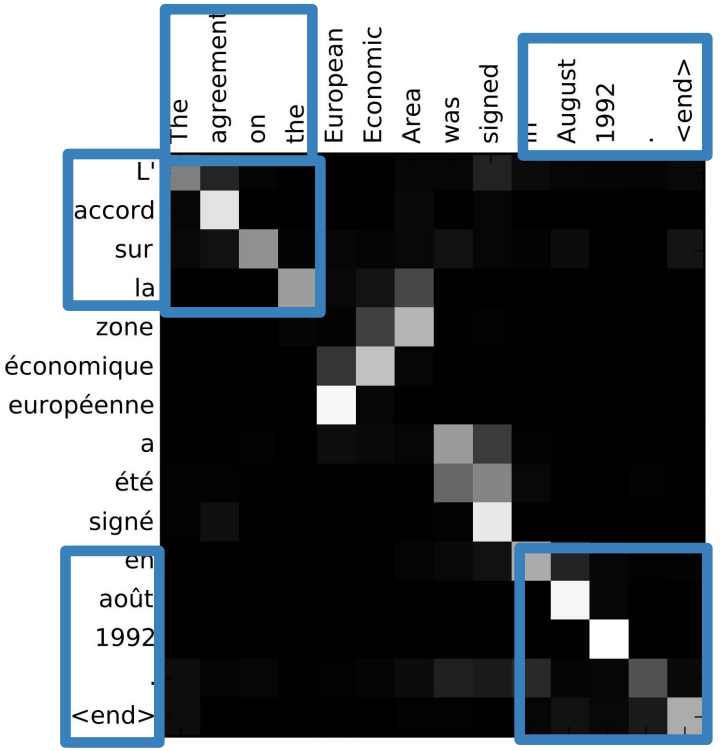
Visualize attention weights $a_{t,i}$

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

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

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

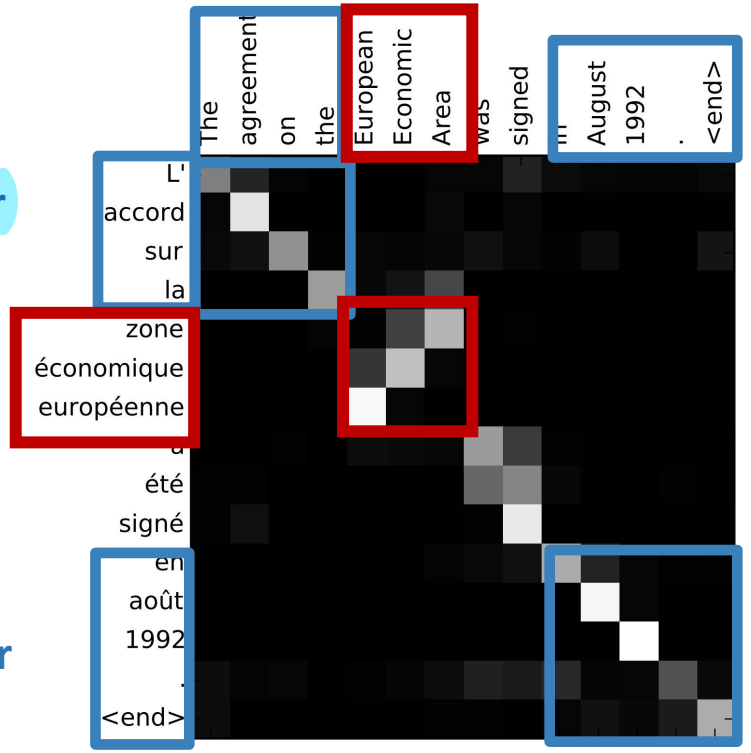
Sequence to Sequence with RNNs and Attention

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 européen 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

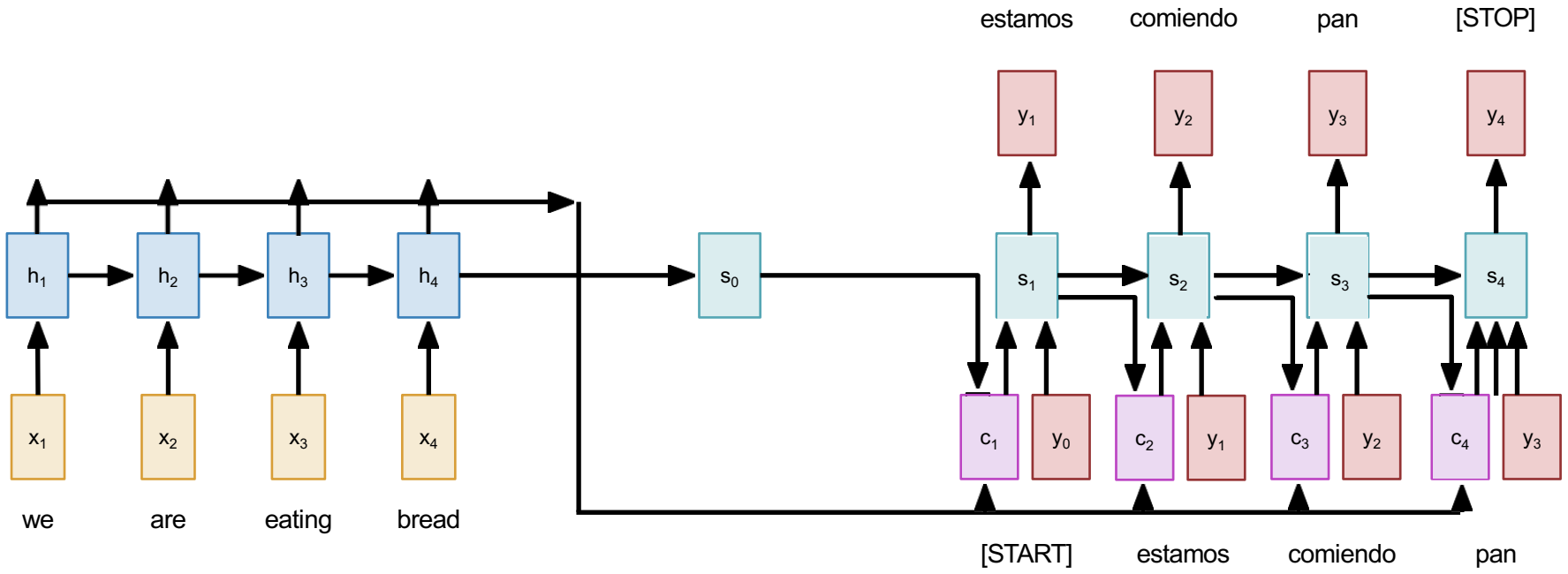
Diagonal attention means words correspond in order

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

Sequence to Sequence with RNNs and Attention

The decoder doesn't use the fact that h_i form an ordered sequence – it just treats them as an unordered set $\{h_i\}$

Can use similar architecture given any set of input hidden vectors $\{h_i\}$!



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

Attention is great

- Attention significantly improves NMT performance
  - It's very useful to allow decoder to focus on certain parts of the source
- Attention provides a more "human-like" model of the MT process
  - You can look back at the source sentence while translating, rather than needing to remember it all
- Attention solves the bottleneck problem
  - Attention allows decoder to look directly at source; bypass bottleneck
- Attention helps with the vanishing gradient problem
  - Provides shortcut to faraway states
- Attention provides some interpretability
  - By inspecting attention distribution, we see what the decoder was focusing on
  - We get (soft) alignment for free!
  - This is cool because we never explicitly trained an alignment system
  - The network just learned alignment by itself

Attention: Summary

- We have some values $h_1, \ldots, h_T \in \mathbb{R}^{d_1}$ and a query $s \in \mathbb{R}^{d_2}$:
- Attention always involves:
  1. Computing the attention scores $e \in \mathbb{R}^T$
  2. Taking softmax to get attention distribution $\alpha = \text{softmax}(e) \in \mathbb{R}^T$
  3. Using attention distribution to take weighted sum of values: $a = \sum_{t=1}^{T} \alpha_t h_t$
     thus obtaining the attention output a (sometimes called the context vector)

Attention variants

- There are several ways you can compute $e \in \mathbb{R}^T$ from $h_1, \ldots, h_T \in \mathbb{R}^{d_1}$ and $s \in \mathbb{R}^{d_2}$:
  - Basic dot-product attention: $e_t = s^T h_t$
    - Note: this assumes $d_1 = d_2$. This is the version we saw earlier.
  - Multiplicative attention: $e_t = s^T W h_t$ [Luong, Pham, and Manning 2015]
    - Where $W \in \mathbb{R}^{d_1 \times d_2}$ is a weight matrix. Perhaps better called "bilinear attention"
  - Reduced-rank multiplicative attention: $e_t = s^T (U^T V) h_t = (Us)^T (V h_t)$
    - • For low rank matrices $U \in \mathbb{R}^{k \times d_2}, V \in \mathbb{R}^{k \times d_1}, k \ll d_1, d_2$
  - Additive attention: $e_t = v^T \tanh(W_1 h_t + W_2 s)$ [Bahdanau, Cho, and Bengio 2014]
    - Where $W_1 \in \mathbb{R}^{d_3 \times d_1}, W_1 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $v \in \mathbb{R}^{d_3}$ is a weight vector.
    - d3 (the attention dimensionality) is a hyperparameter
    - "Additive" is a weird/bad name. It's really using a feed-forward neural net layer

More information: "Deep Learning for NLP Best Practices", Ruder, 2017
"Massive Exploration of Neural Machine Translation Architectures", Britz et al, 2017

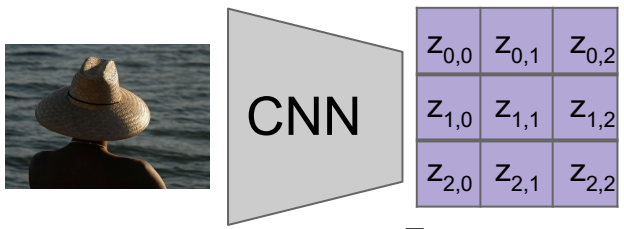
Attention is a general Deep Learning technique

- General definition of attention: Given a set of vector values and a vector query, attention is a technique to compute a weighted sum of the vector values, dependent on the query.
  - We sometimes say that the query attends to the values.
  - For example, in the seq2seq + attention model, each decoder hidden state (query) attends to all the encoder hidden states (values)
- Intuition:
  - The weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on.
  - Attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query).
- Upshot:
  - Attention has become the powerful, flexible, general way pointer and memory manipulation in all deep learning models. A new idea from after 2010! From NMT!

Image Captioning using spatial features

Input: Image **I**

Output: Sequence $\boldsymbol{y} = y_1, y_2, \ldots y_T$



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

Image Captioning using spatial features

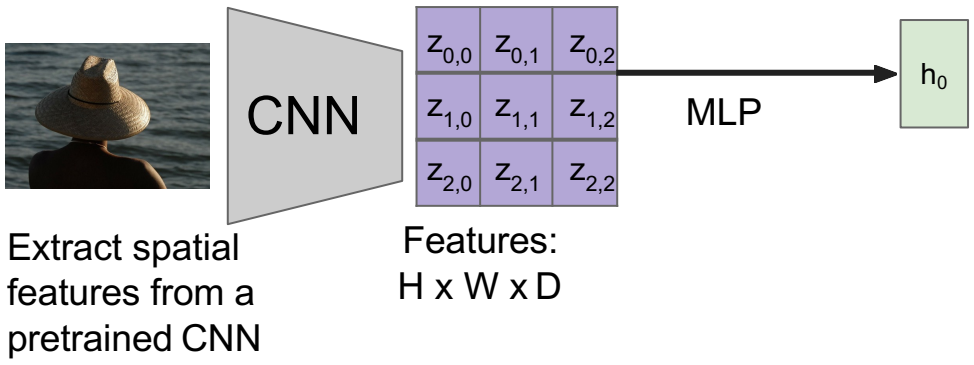
Input: Image **I**

Output: Sequence $\boldsymbol{y} = y_1, y_2, \ldots y_T$

Encoder: $h_0 = f_{\boldsymbol{w}}(\boldsymbol{z})$

where **z** is spatial CNN features

$f_{\boldsymbol{w}}(.)$ is an MLP



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

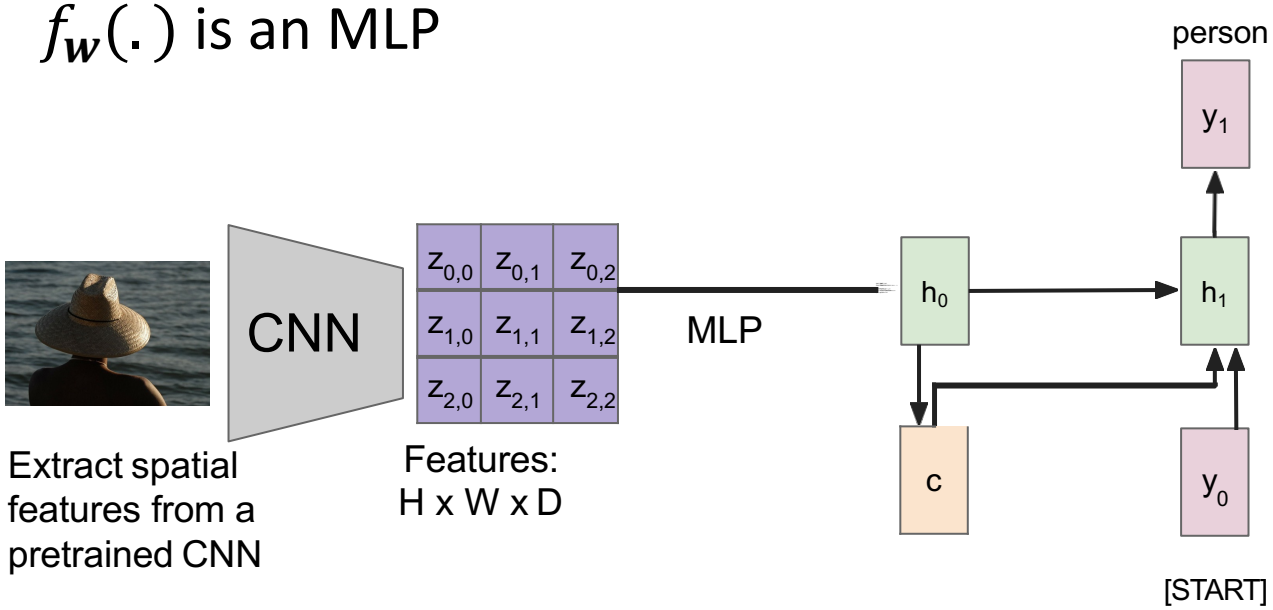
Image Captioning using spatial features

Input: Image **I**

Output: Sequence $\boldsymbol{y} = y_1, y_2, \ldots y_T$

Decoder: $y_t = g_v(y_{t-1}, h_{t-1}, c)$
Where context vector c is often $c = h_0$

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



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

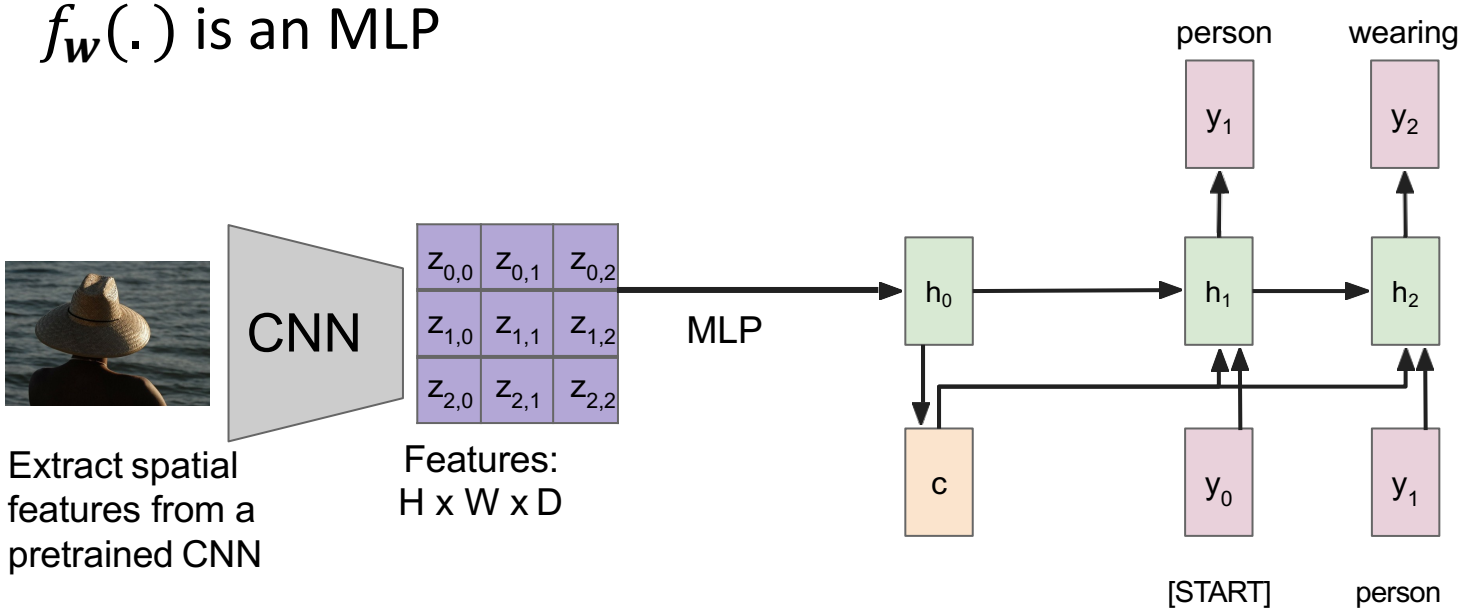
Image Captioning using spatial features

Input: Image **I**

Output: Sequence $\boldsymbol{y} = y_1, y_2, \ldots y_T$

Decoder: $y_t = g_v(y_{t-1}, h_{t-1}, c)$
Where context vector c is often $c = h_0$

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



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

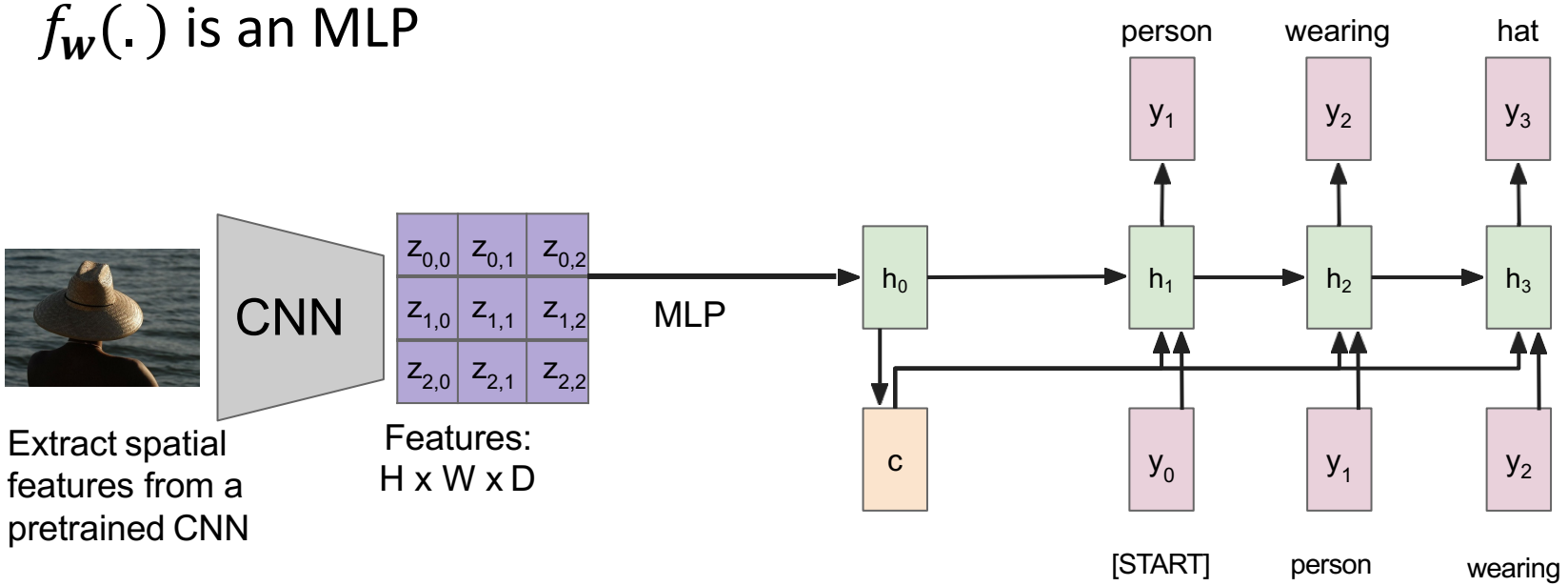
Image Captioning using spatial features

Input: Image **I**

Output: Sequence $\boldsymbol{y} = y_1, y_2, \ldots y_T$

Decoder: $y_t = g_v(y_{t-1}, h_{t-1}, c)$
Where context vector c is often $c = h_0$

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



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

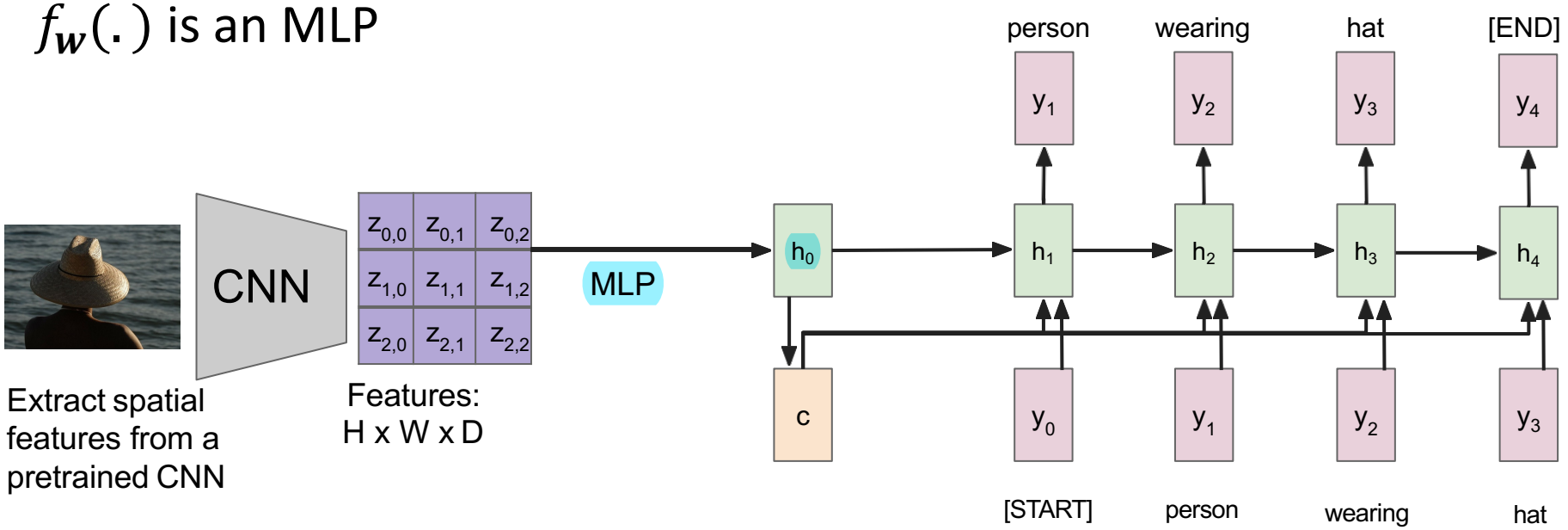
Image Captioning using spatial features

Input: Image **I**

Output: Sequence $\boldsymbol{y} = y_1, y_2, \ldots y_T$

Decoder: $y_t = g_v(y_{t-1}, h_{t-1}, c)$
Where context vector c is often $c = h_0$

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



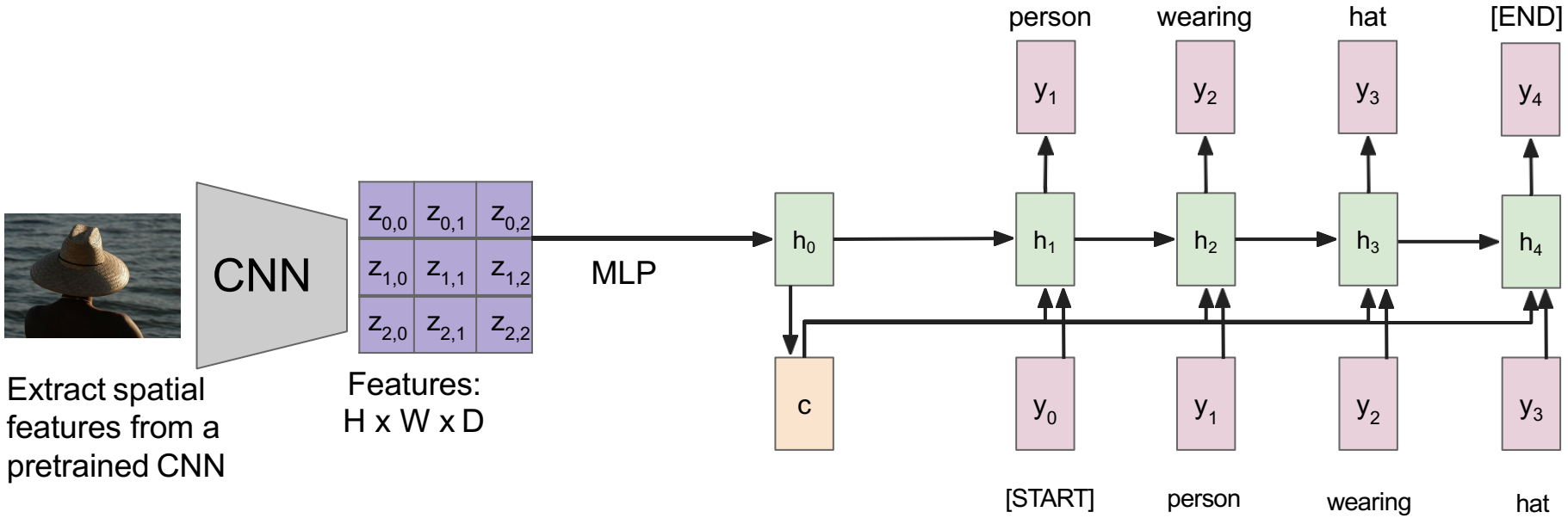
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning using spatial features

Problem: 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

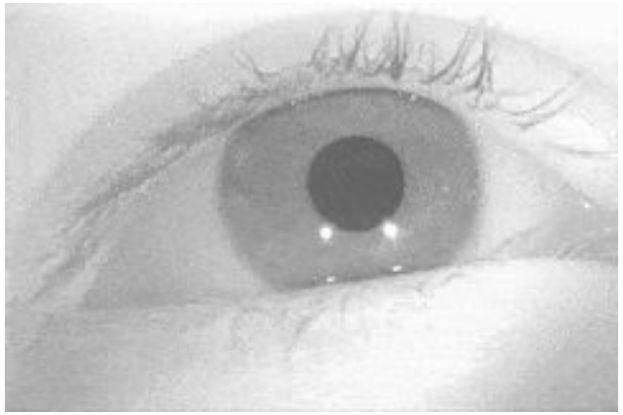


Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

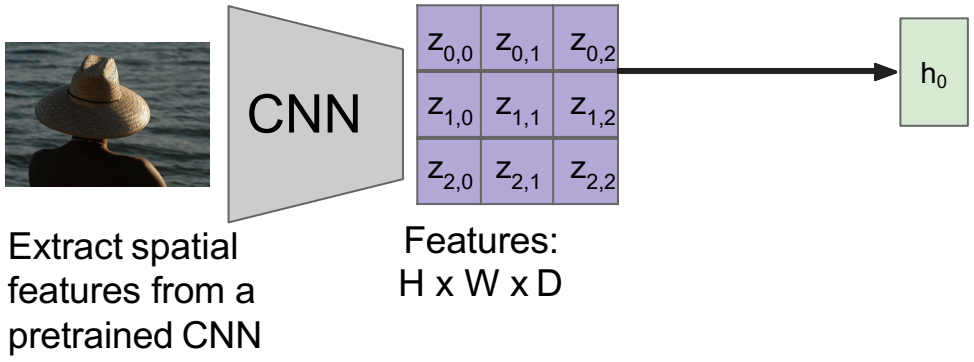
Image Captioning with RNNs and Attention

Attention idea: New context vector at every time step.

Each context vector will attend to different image regions

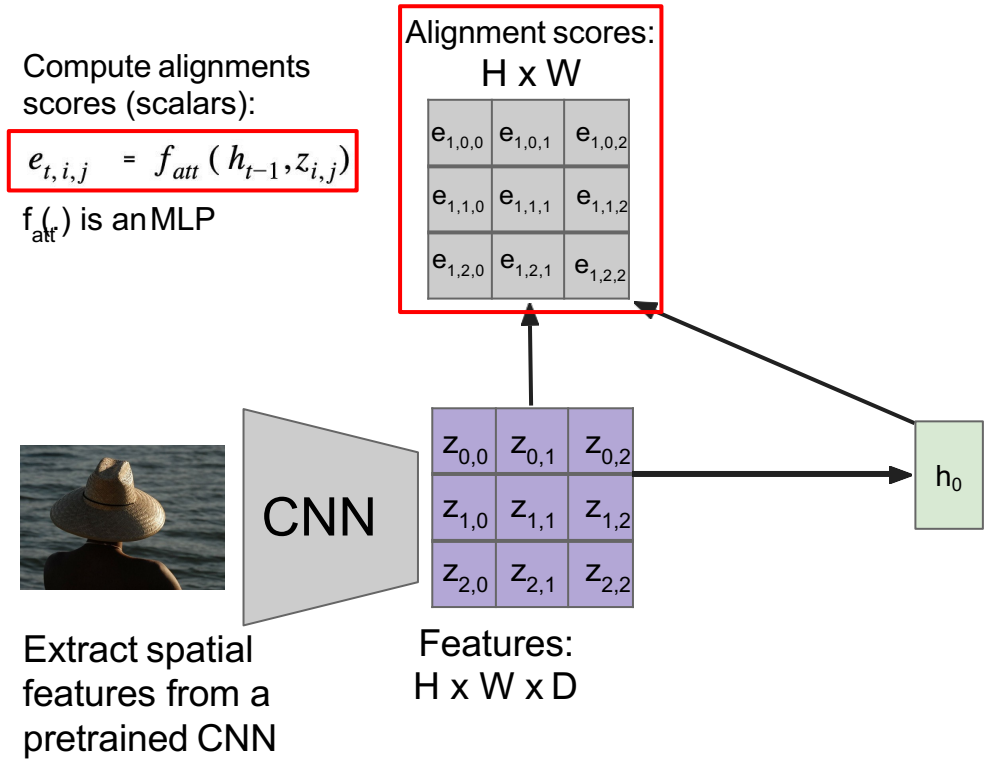


Attention Saccades in humans



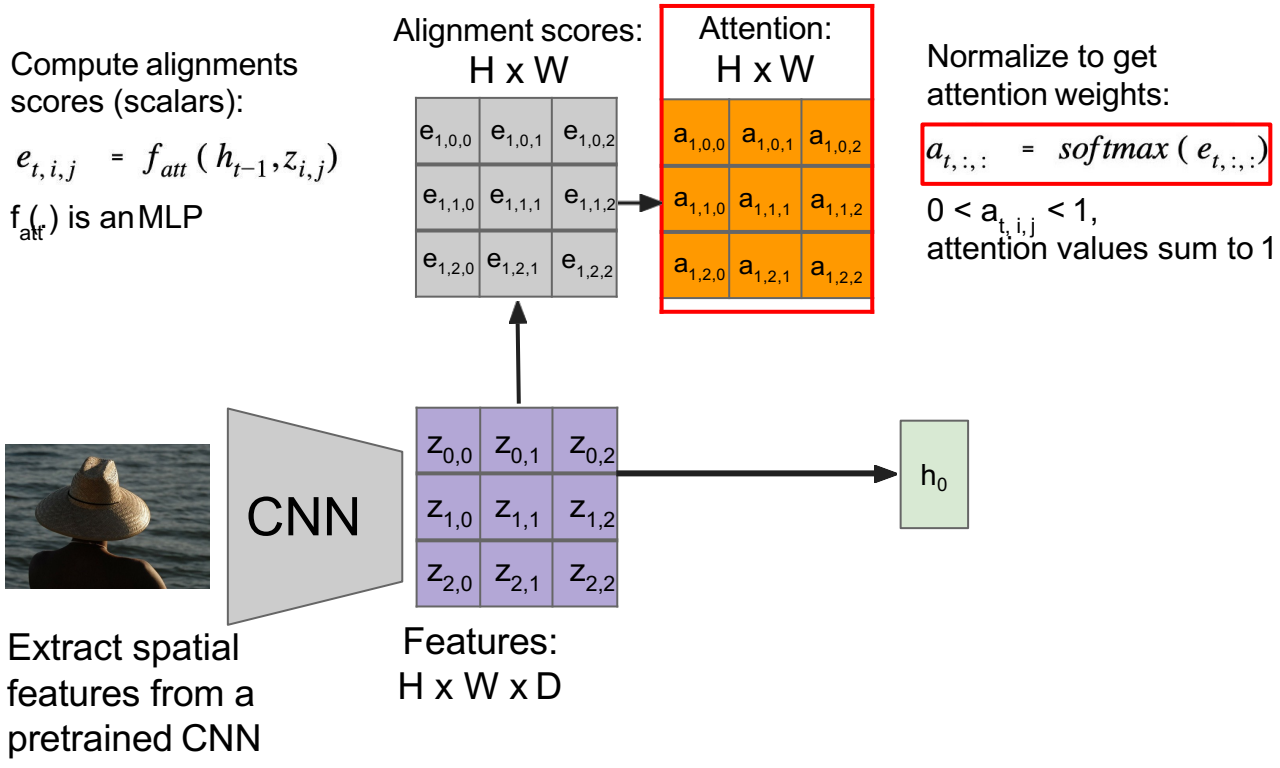
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention



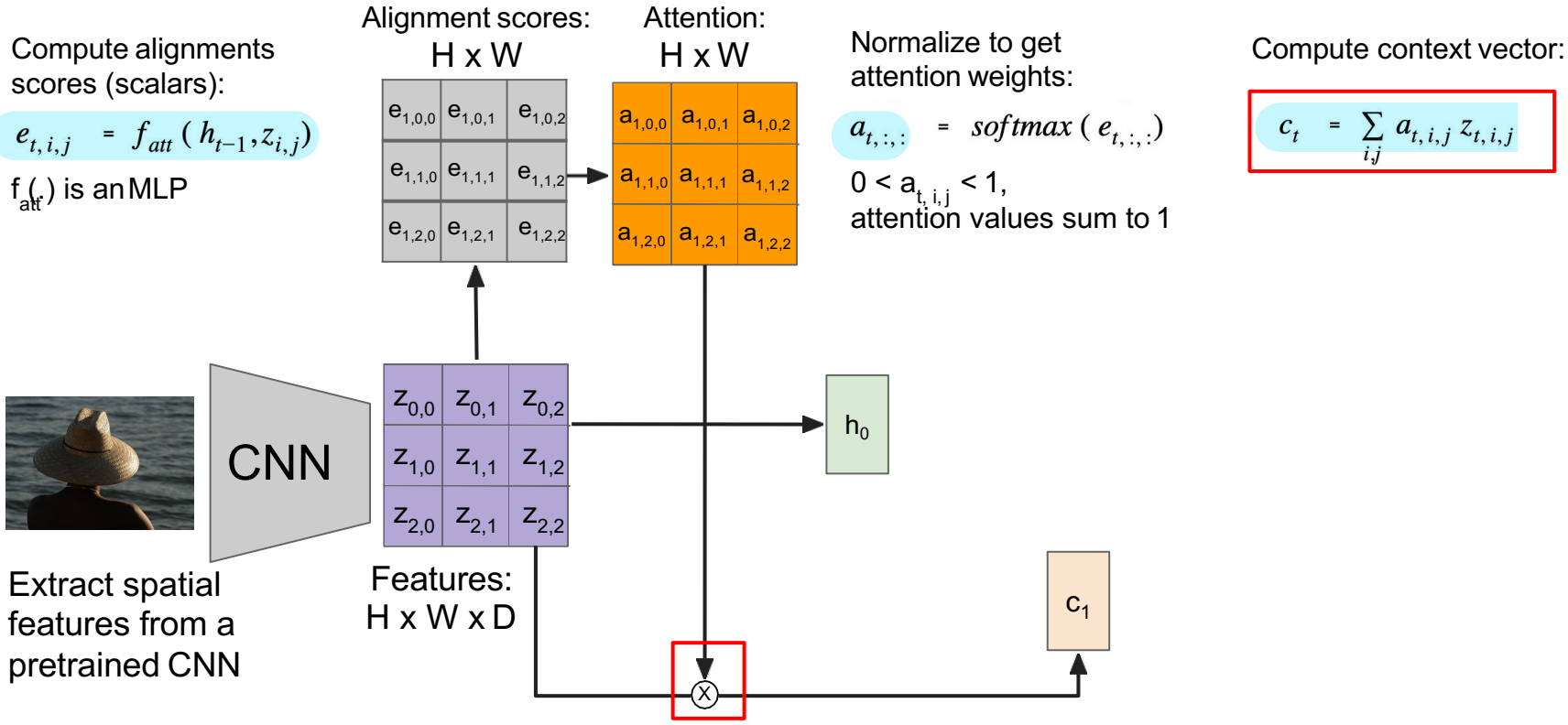
Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

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



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

Image Captioning with RNNs and Attention

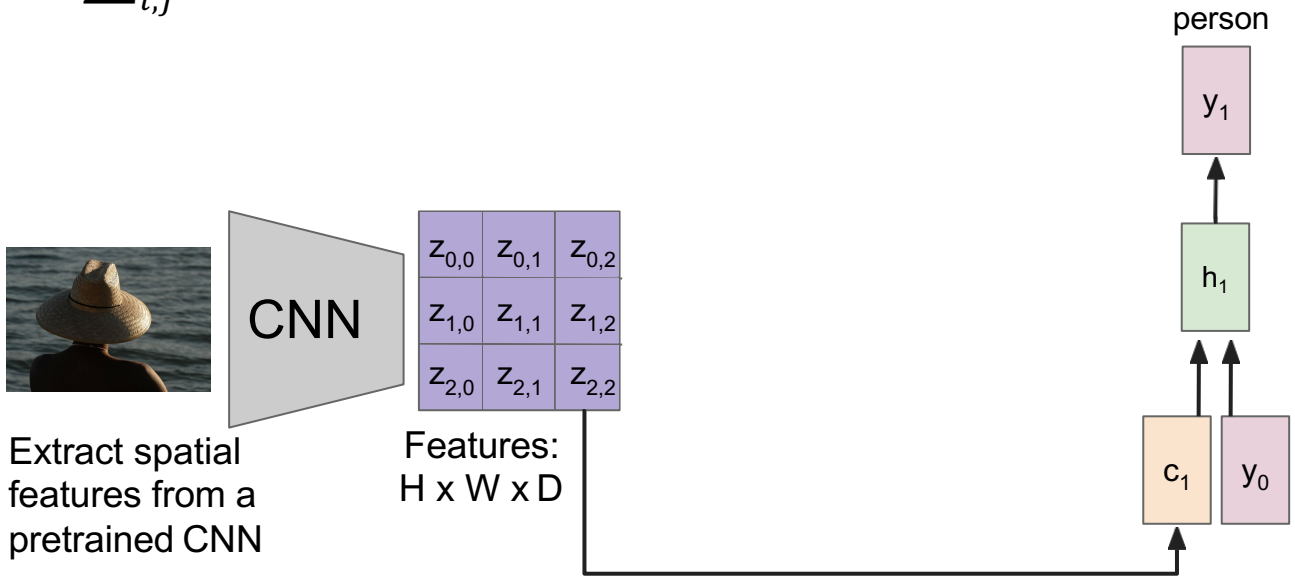
Each timestep of decoder uses a different context vector that looks at different parts of the input image

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

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

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

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

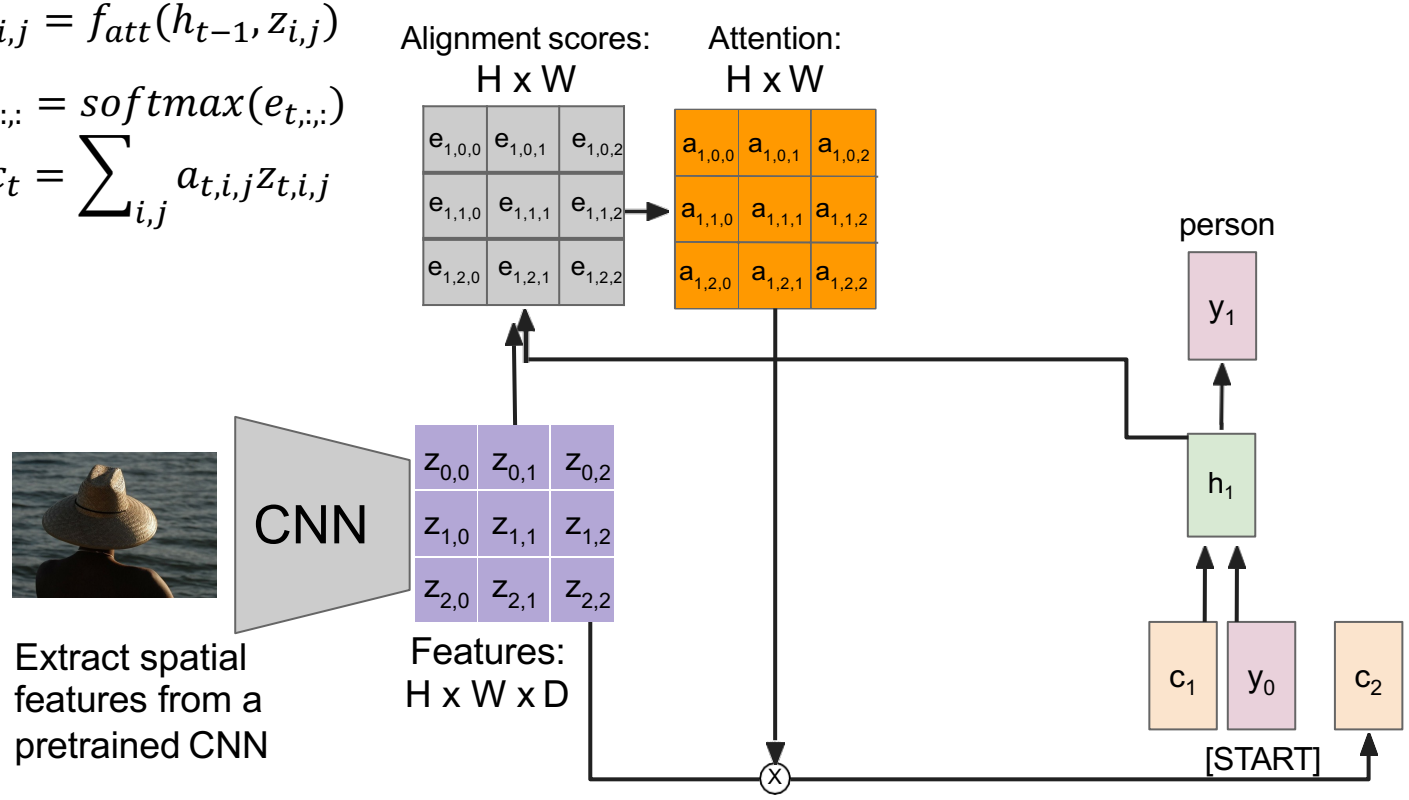


Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

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

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



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

Image Captioning with RNNs and Attention

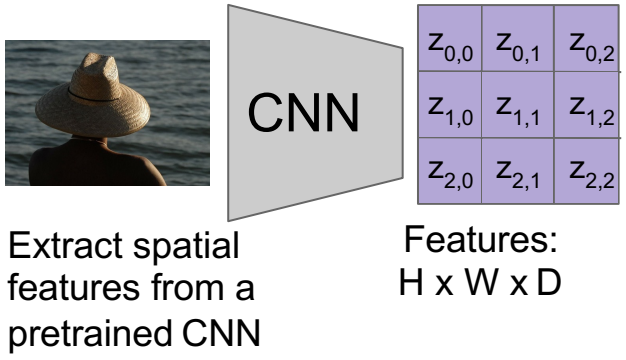
Each timestep of decoder uses a different context vector that looks at different parts of the input image

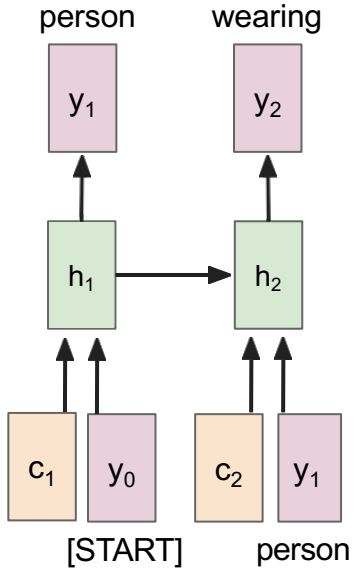
$$e_{t,i,j} = f_{att}(h_{t-1}, z_{i,j})$$

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

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

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



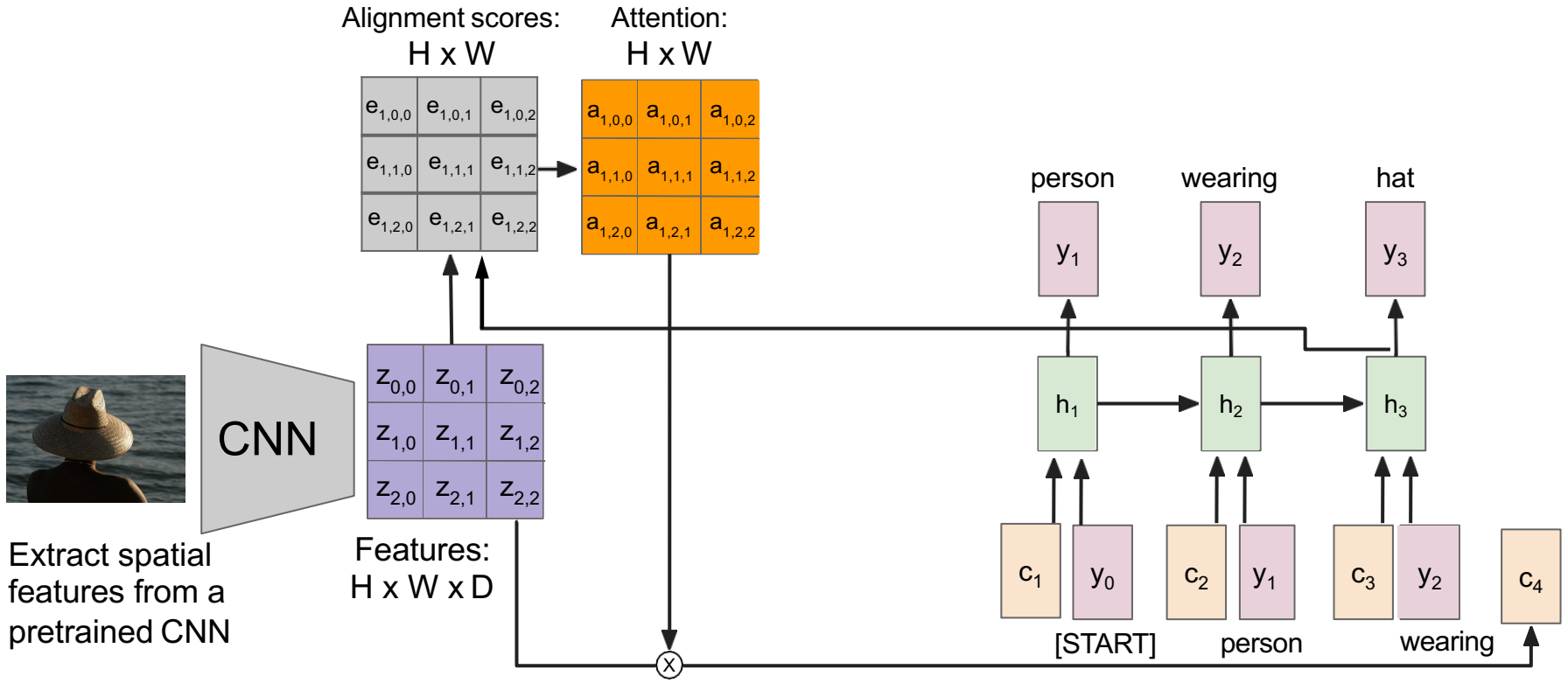


Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

This entire process is differentiable.

- **Model chooses its own attention weights. No attention supervision is required**

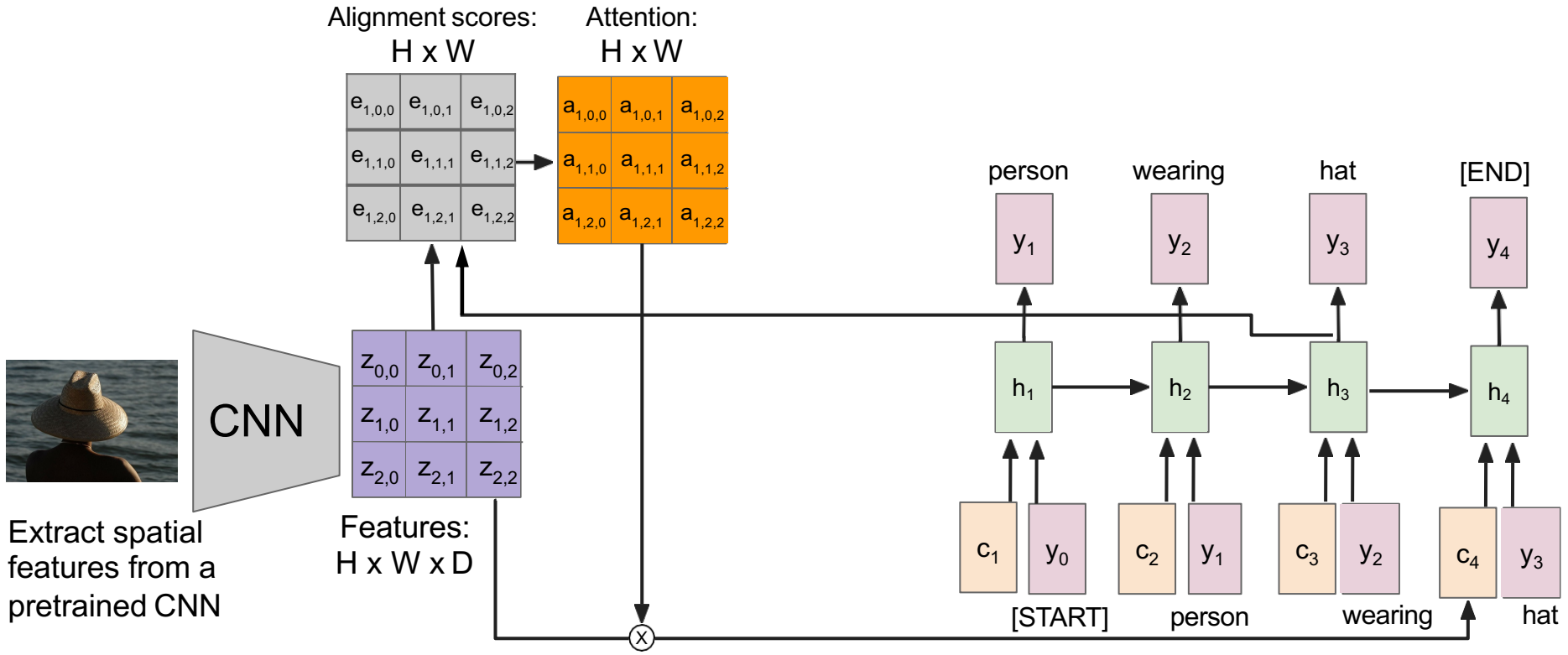


Xu et al, "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", ICML 2015

Image Captioning with RNNs and Attention

This entire process is differentiable.

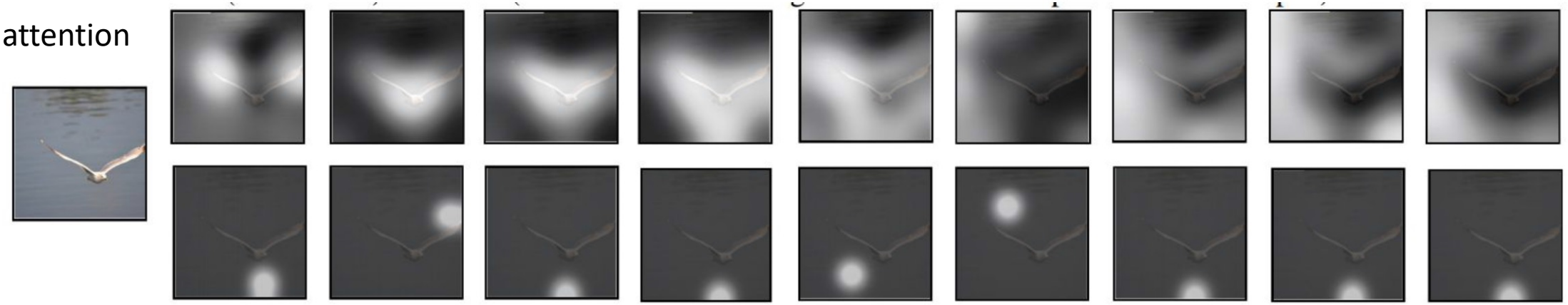
- **Model chooses its own attention weights. No attention supervision is required**



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

Image Captioning with Attention

Soft attention



Hard attention
(requires RL)

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

Image Captioning with Attention

A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

A little girl sitting on a bed with a teddy bear.

A group of people sitting on a boat in the water.

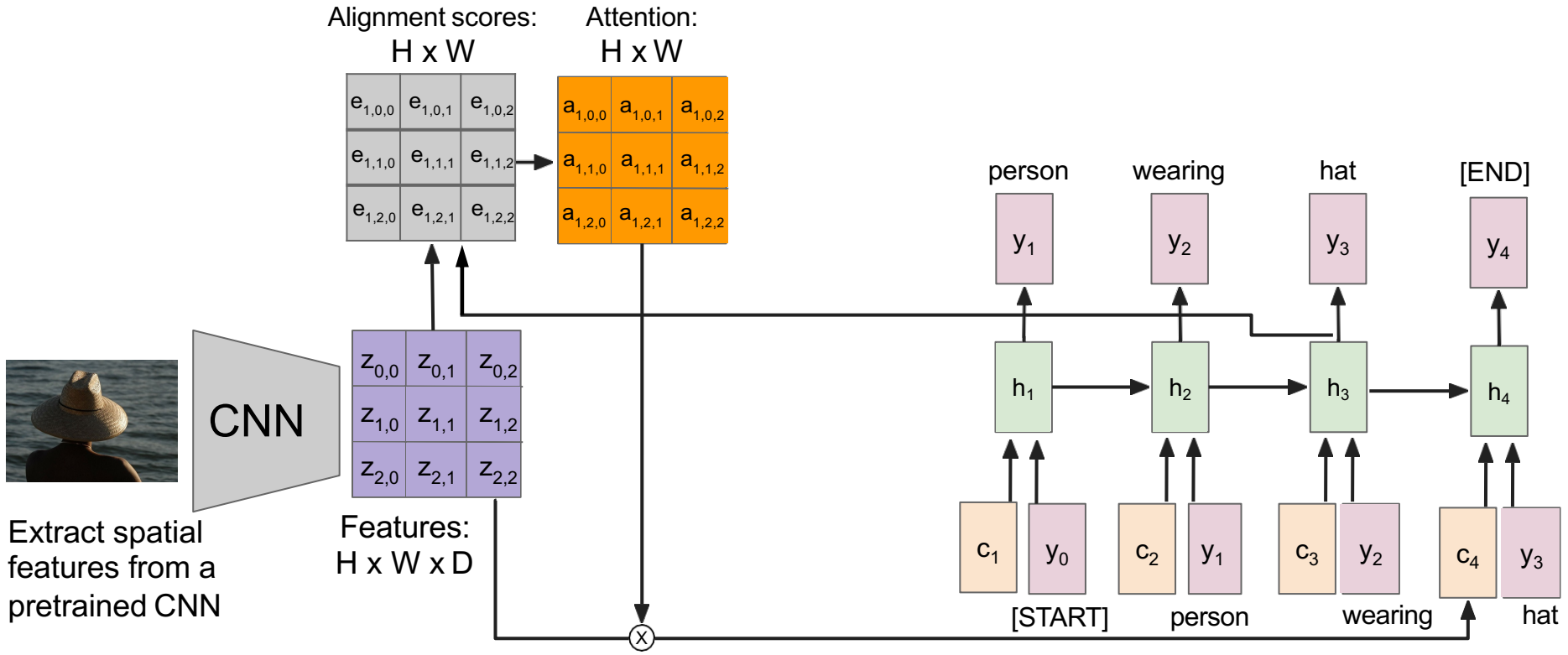
A giraffe standing in a forest with trees in the background.

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

Image Captioning with RNNs and Attention

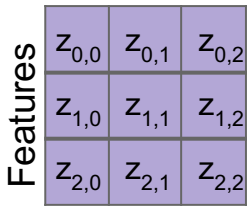
This entire process is differentiable.

- **Model chooses its own attention weights. No attention supervision is required**



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:

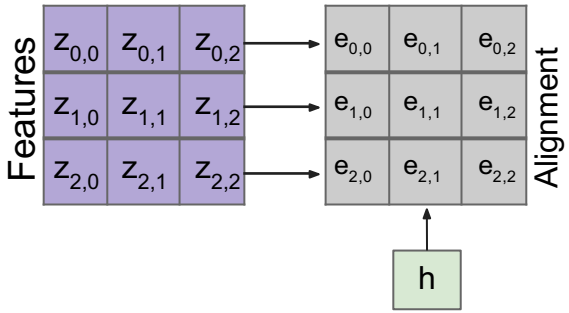
Features: z $(shape:\ H \times W \times D)$

Query: h $(shape:\ D)$

Attention we just saw in image captioning

Operations:

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

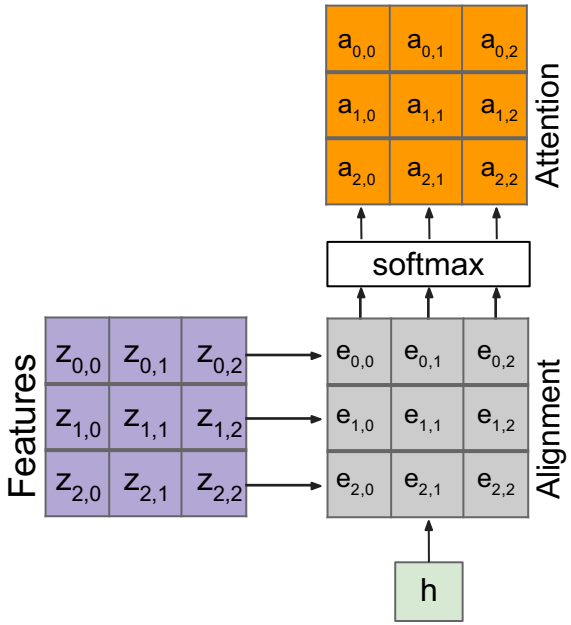


Inputs:

Features: z $(shape:\ H \times W \times D)$

Query: h $(shape:\ D)$

Attention we just saw in image captioning



Operations:

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

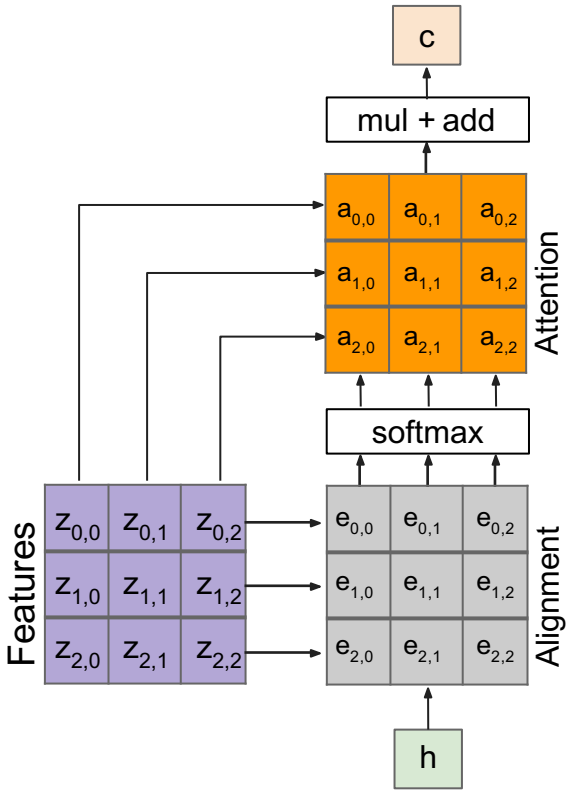
Attention: a $= softmax(e)$

Inputs:

Features: z $(shape:\ H \times W \times D)$

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: $a = softmax(e)$

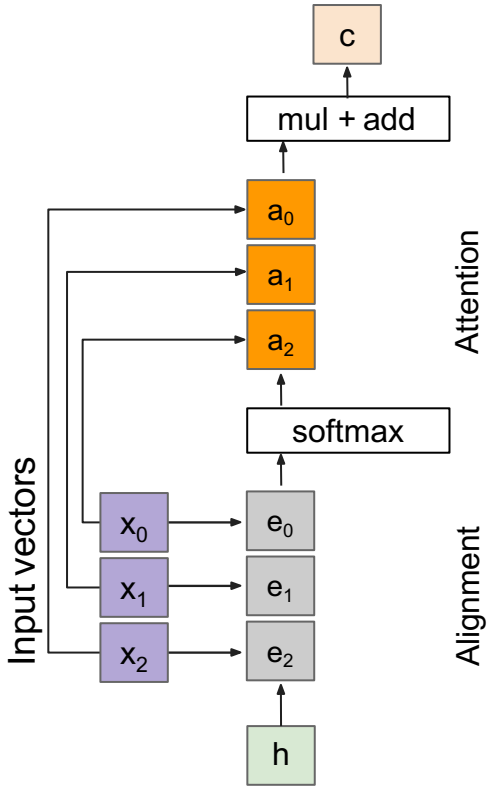
Output: $c = \sum_{i,j} a_{i,j} z_{i,j}$

Inputs:

Features: z ($shape$: $H \times W \times D$)

Query: h ($shape$: D)

General attention layer



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:

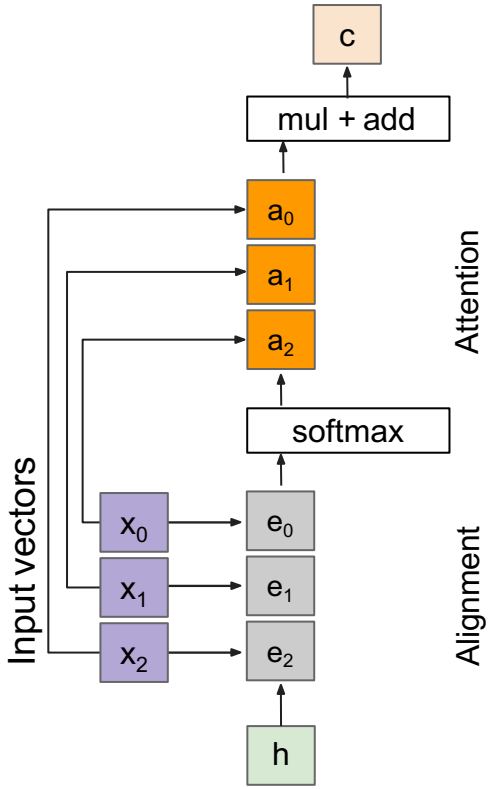
Input vectors: x ($shape$: $N \times D$)

Query: h ($shape$: D)

Attention operation is permutation invariant.

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

General attention layer



Outputs:

Context vector: c $(shape: D)$

Operations:

Alignment: $e_i = h.x_i$

Attention: a $= softmax(e)$

Output: c $= \sum_i a_i x_i$

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

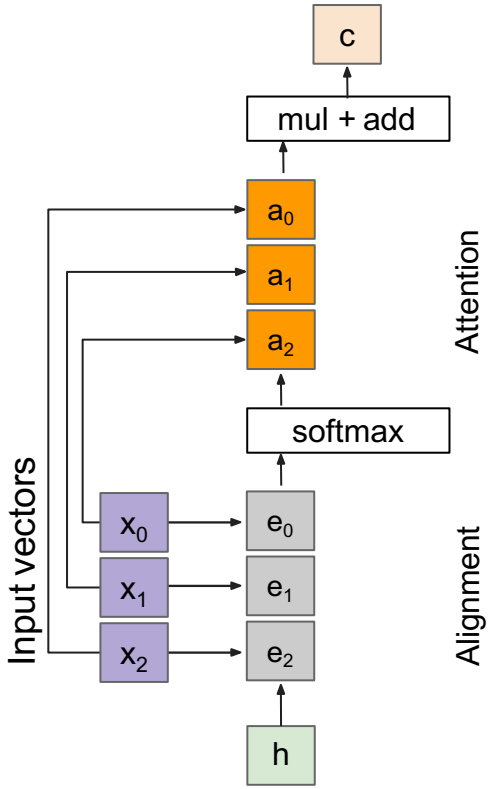
- Only works well with key & value transformation trick (will mention in a few slides)

Inputs:

Input vectors: x $(shape: N \times D)$

Query: h $(shape: D)$

General attention layer



Outputs:

Context vector: c $(shape:\ D)$

Operations:

Alignment: $e_i = h.x_i/\sqrt{D}$

Attention: a $= softmax(e)$

Output: c $= \sum_i a_i x_i$

Inputs:

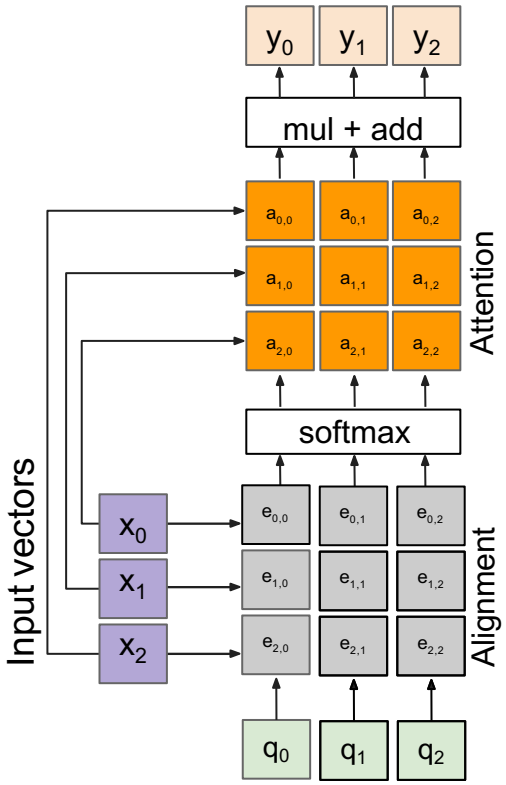
Input vectors: x $(shape: N \times D)$

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.
- The post-softmax distribution has lower-entropy, assuming logits are IID.
- Ultimately, it causes softmax to peak
- Larger inputs to the softmax function making the gradients small.
- Divide by $\sqrt{D}$ to reduce effect of large magnitude vectors

General attention layer



Outputs:

Context vectors: y $(shape:\ D)$

Operations:

Alignment: $e_{i,j} = q_j . x_i / \sqrt{D}$

Attention: a $= softmax(e)$

Output: $y_j = \sum_i a_{i,j} x_i$

Multiple query vectors

– Each query creates a new output context vector

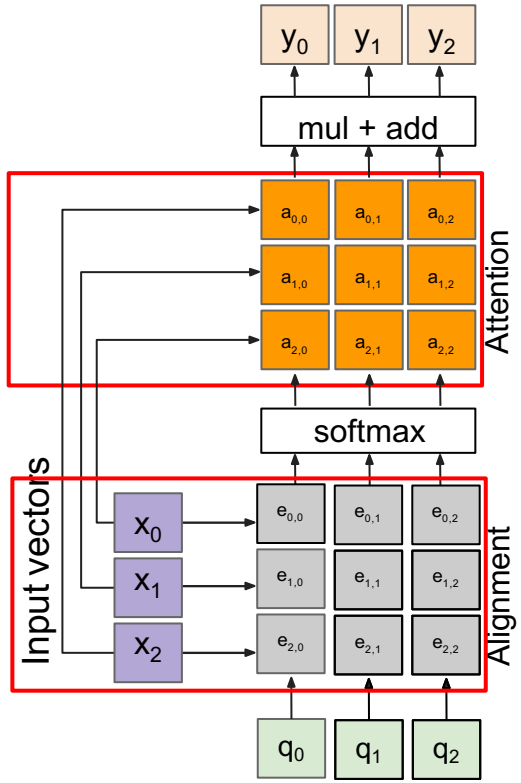
Inputs:

Input vectors: x $(shape: N \times D)$

Query: q $(shape: M \times D)$

Multiple query vectors

General attention layer



Outputs:

Context vectors: y $(shape:\ D)$

Operations:

Alignment: $e_{i,j} = q_j . x_i / \sqrt{D}$

Attention: $a = softmax(e)$

Output: $y_j = \sum_i a_{i,j} x_i$

Inputs:

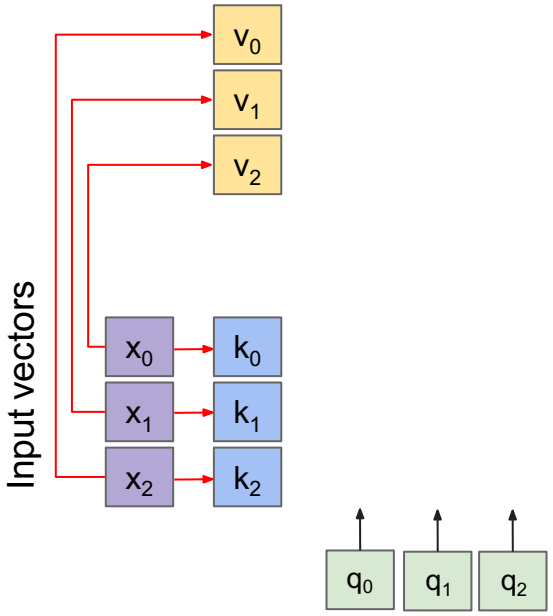
Input vectors: x $(shape: N \times D)$

Query: q $(shape: M \times D)$

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: $\mathrm{k} = W_k^T x$

Value vectors: $\mathrm{v} = W_v^T x$

Inputs:

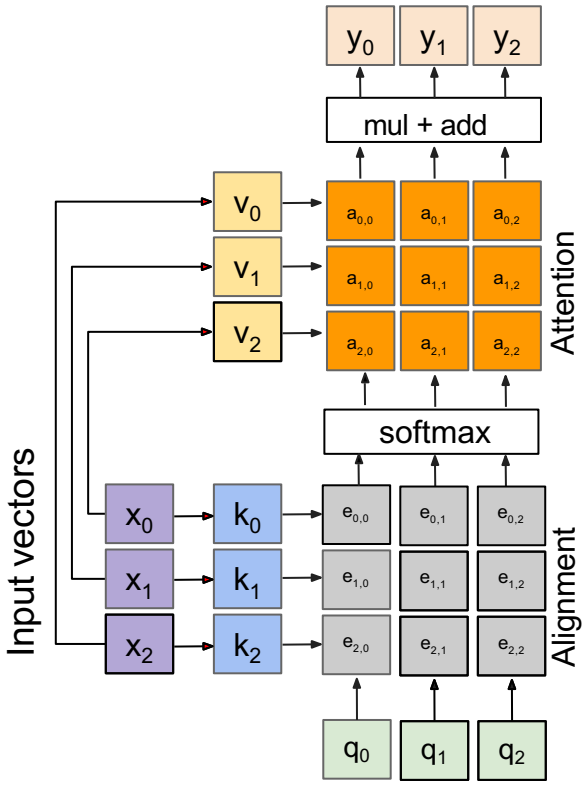
Input vectors: x ($shape$: $N \times D$)

Queries: q ($shape$: $M \times D_k$)

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: $k = W_k^T x$

Value vectors: $v = W_v^T x$

Alignment: $e_{i,j} = q_j . k_i / \sqrt{D}$

Attention: $a = softmax(e)$

Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:

Input vectors: x ($shape$: $N \times D$)

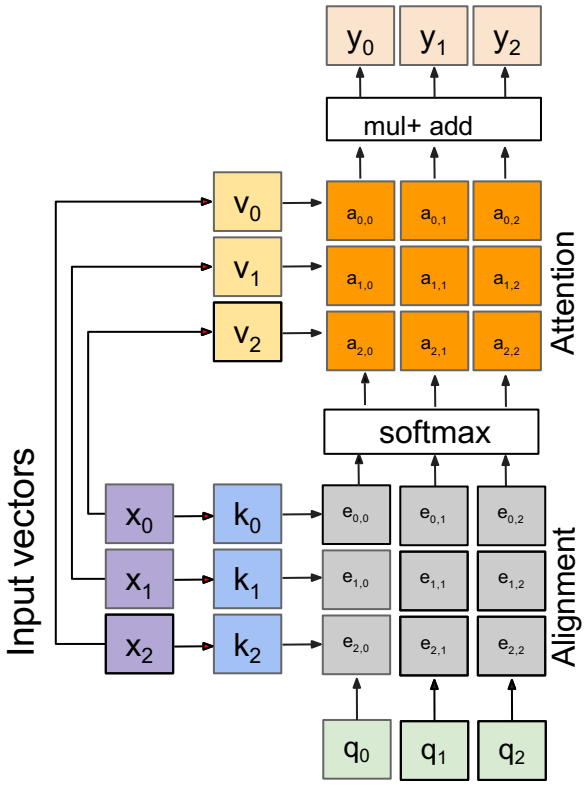
Queries: q ($shape$: $M \times D_k$)

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

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: $\mathrm{k} = W_k^T x$

Value vectors: $\mathrm{v} = W_v^T x$

Alignment: $e_{i,j} = q_j . k_i / \sqrt{D}$

Attention: $\mathrm{a} = softmax(e)$

Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:

Input vectors: x ($shape$: $N \times D$)

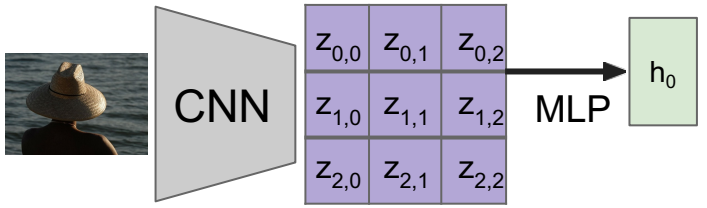
Queries: q ($shape$: $M \times D_k$)

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

Encoder: $h_0 = f_w(z)$

where z is spatial CNN features

$f_w(.)$ is an MLP



ImageNet Moment for Natural Language Processing

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

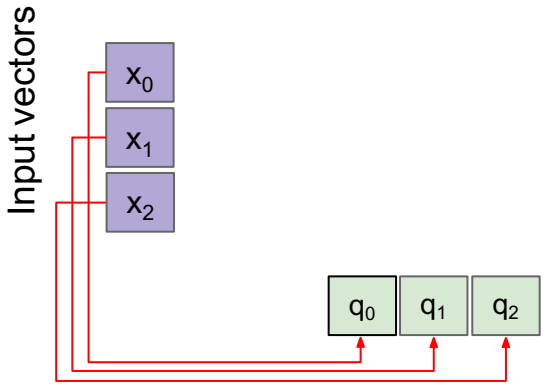
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

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

Self attention layer



Operations:

Key vectors: $\mathrm{k} = W_k^T x$

Value vectors: $\mathrm{v} = W_v^T x$

Query vectors: $\mathrm{q} = W_q^T x$

Alignment: $e_{i,j} = q_j . k_i / \sqrt{D}$

Attention: $\mathrm{a} = softmax(e)$

Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:

Input vectors: x $(shape: N \times D)$

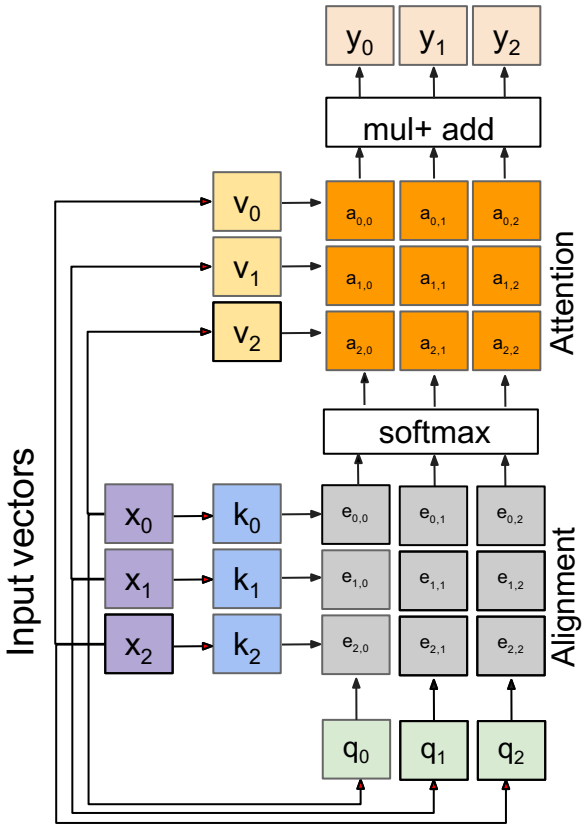
~~Queries: q $(shape: M \times D_k)$~~

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.

No Input query vectors anymore

Self attention layer



Outputs:

context vectors: y ($shape$: D_v)

Operations:

Key vectors: $\mathrm{k} = W_k^T x$

Value vectors: $\mathrm{v} = W_v^T x$

Query vectors: $\mathrm{q} = W_q^T x$

Alignment: $e_{i,j} = q_j . k_i / \sqrt{D}$

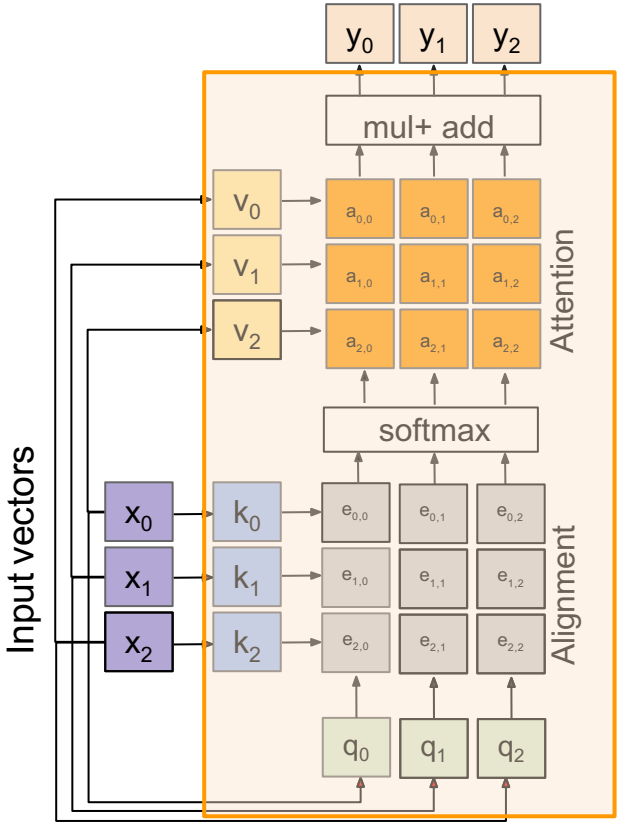
Attention: $\mathrm{a} = softmax(e)$

Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:

Input vectors: x ($shape$: $N \times D$)

Self attention layer – attends over sets of inputs



Outputs:

context vectors: y ($shape$: D_v)

Operations:

Key vectors: $k = W_k^T x$

Value vectors: $v = W_v^T x$

Query vectors: $q = W_q^T x$

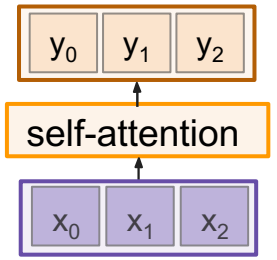
Alignment: $e_{i,j} = q_j . k_i / \sqrt{D}$

Attention: $a = softmax(e)$

Output: $y_j = \sum_i a_{i,j} v_i$

Inputs:

Input vectors: x ($shape$: $N \times D$)



Self-Attention: keys, queries, values

- Let $w_1, \ldots, w_n$ be a sequence of words in vocabulary V
- For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_i$, where $E \in \mathbb{R}^d \times |V|$ is an embedding matrix.

1. Transform each word embedding with weight matrices W_q, W_k, W_v each in $\mathbb{R}^{d \times d}$

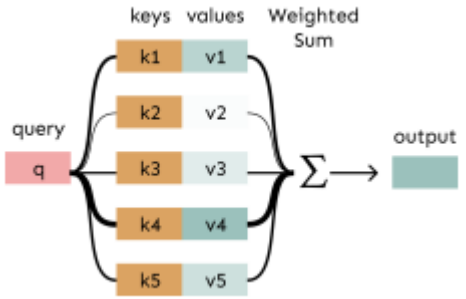
   Key vectors: $\mathrm{k} = W_k^T x$ Value vectors: $\mathrm{v} = W_v^T x$ Query vectors: $\mathrm{q} = W_q^T x$

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$e_{i,j} = q_j^T k_i / \sqrt{D} \qquad a_{i,j} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$y_j = \sum_i a_{i,j} v_i$$

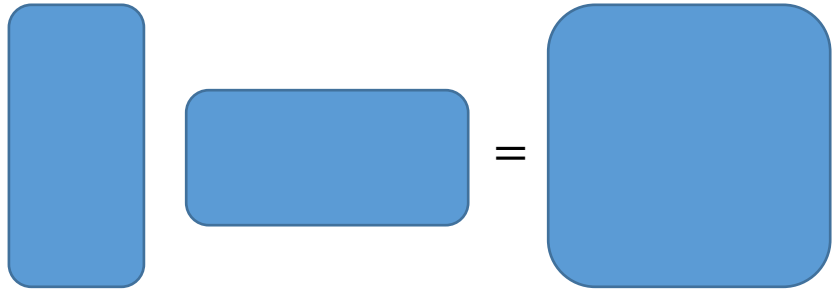


Self attention: Vector notation

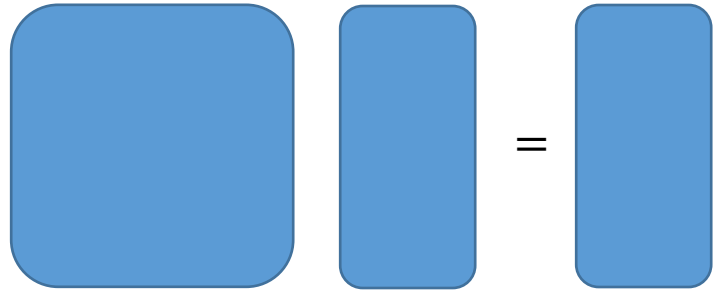
$$X \in \mathbb{R}^{T \times d}$$
$$W_k, W_q, W_v \in \mathbb{R}^{d \times d}$$

$$XW_k = K$$
$$XW_q = Q$$
$$XW_v = V$$

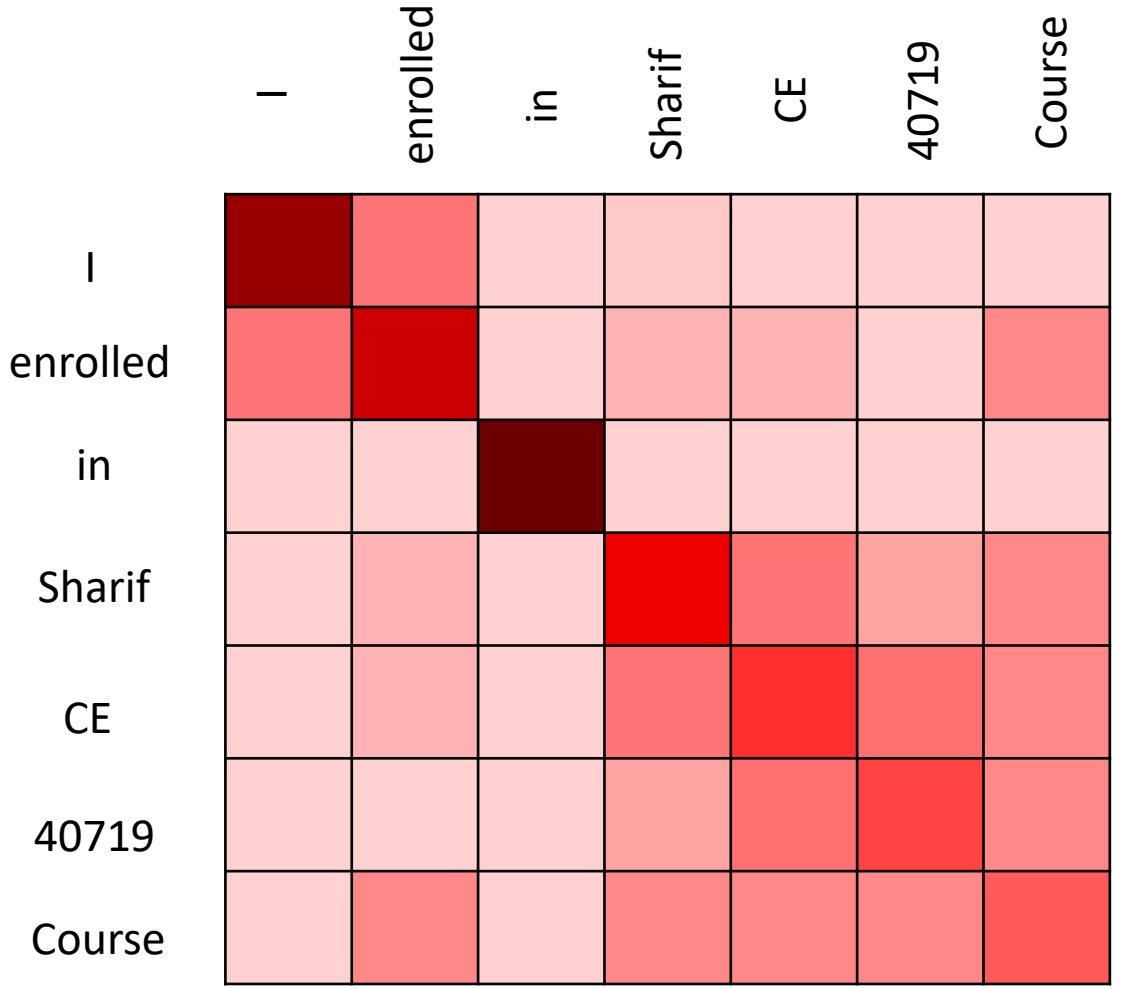
Attention map: $softmax(\frac{QK^T}{scaling})$



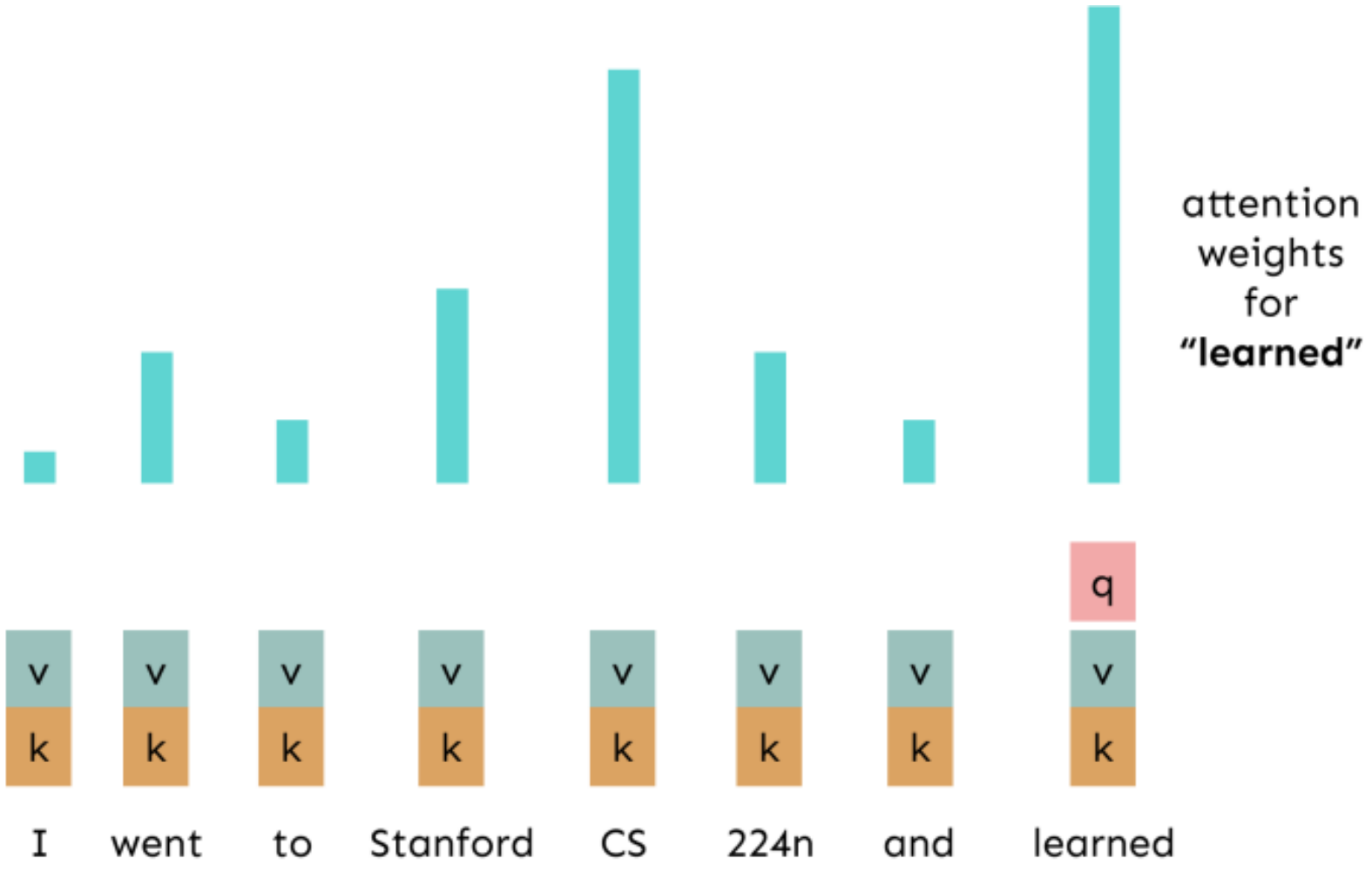
Output: $softmax\left(\frac{QK^T}{scaling}\right)V$



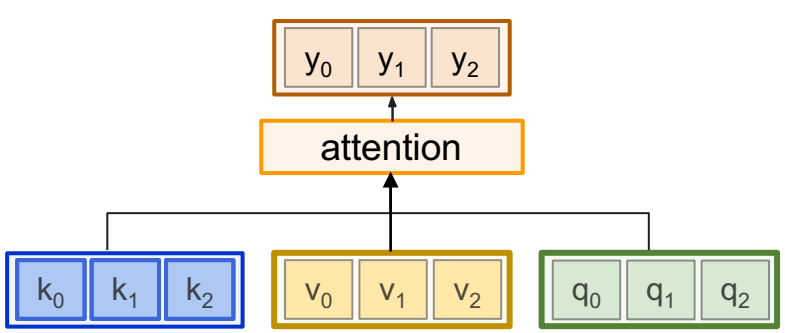
Self attention: Hypothetical Example

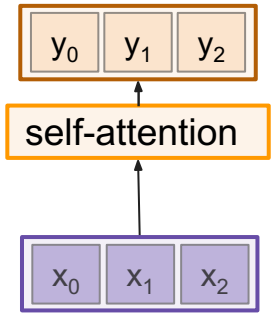


Self-Attention Hypothetical Example



General attention versus self-attention



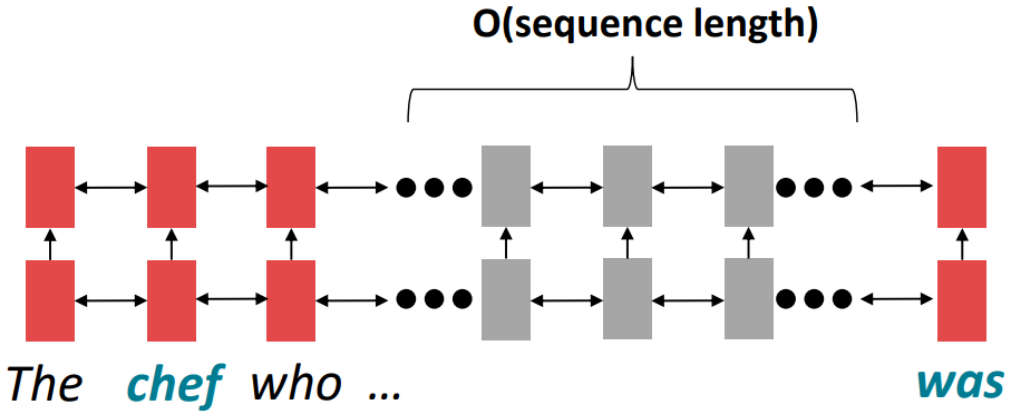


Issues with RNNs

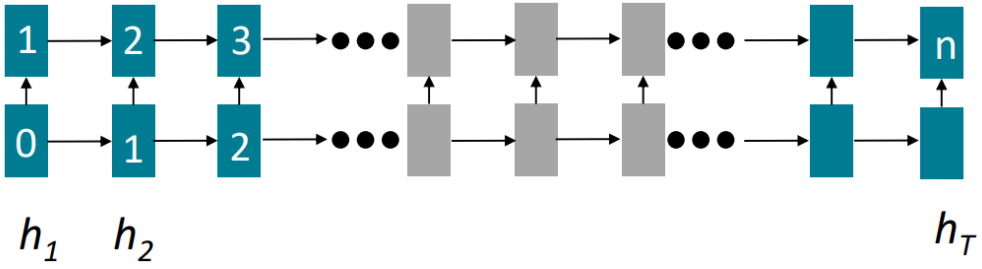
- RNNs are unrolled "left-to-right".
  - This encodes linear locality: a useful inductive bias!
    - Nearby words often affect each other's meanings

- **Encoding bottleneck**
  - **Whole sequence into a hidden state**
  - **Linear interaction distance**: RNNs take O(seq length) steps for distant pairs to interact.

- **Lack of parallelizability**: Forward and backward passes have O(seq length) unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before computing past RNN hidden states
  - Inhibits training on very large datasets!



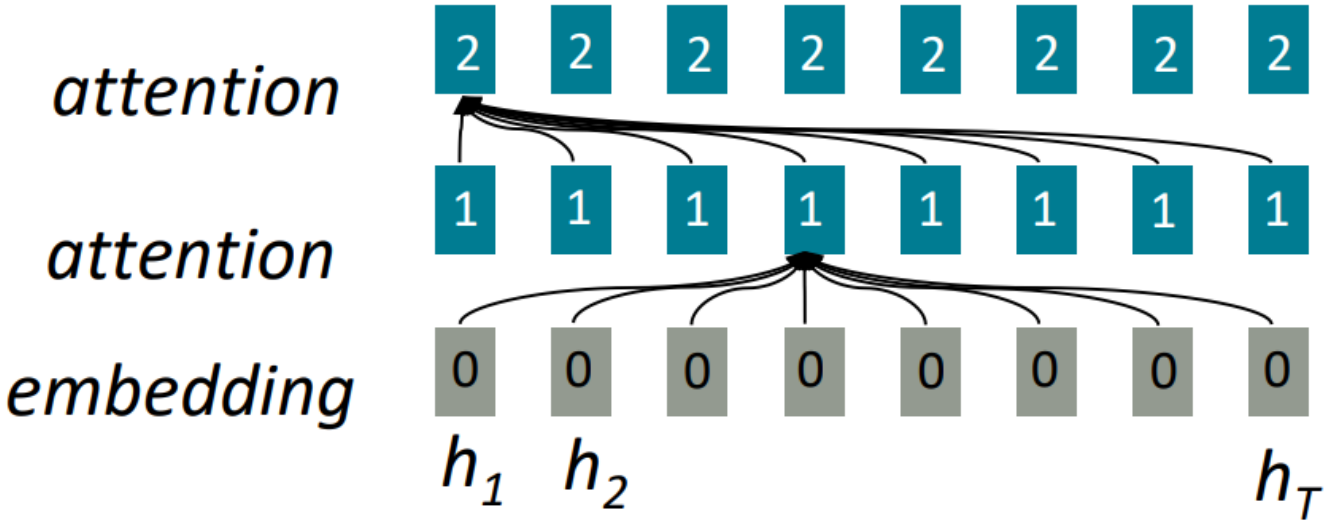
Info of chef has gone through O(sequence length) many layers!



Numbers indicate min # of steps before a state can be computed

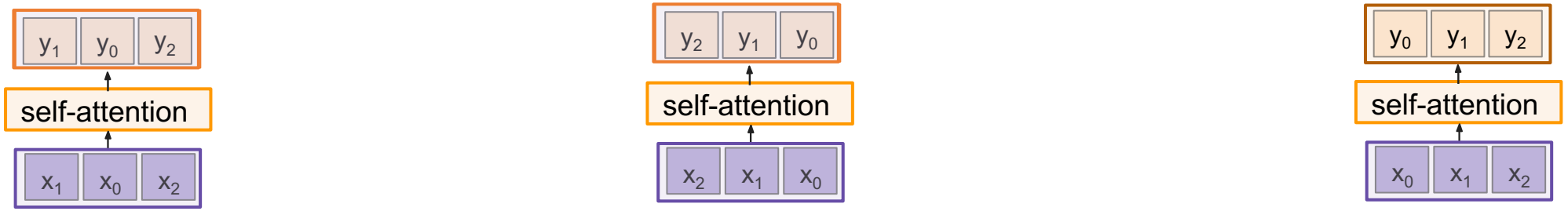
Self-attention

- Self-attention treats each word's representation as a query to access and incorporate information from a set of values.
- Number of unparallelizable operations does not increase with sequence length.
- Maximum interaction distance: O(1), since all words interact at every layer!



All words attend to all words in previous layer; most arrows here are omitted

Self attention layer – attends over sets of inputs

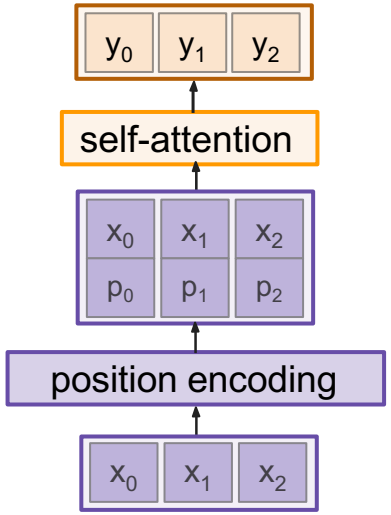


Permutation equivariant

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



Concatenate/add special positional encoding p_j to each input vector x_j

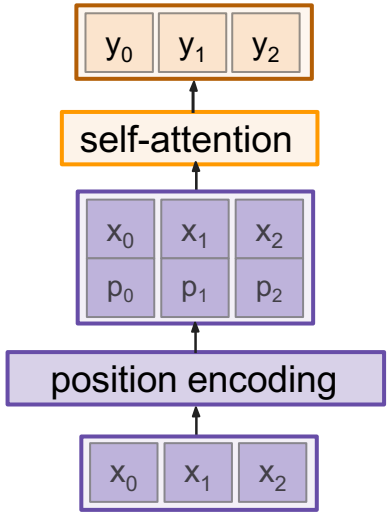
We use a function $pos: N \rightarrow R^d$ to process the position j of the vector into a d-dimensional vector

So, $p_j = pos(j)$

Desiderata 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 $pos(.)$:

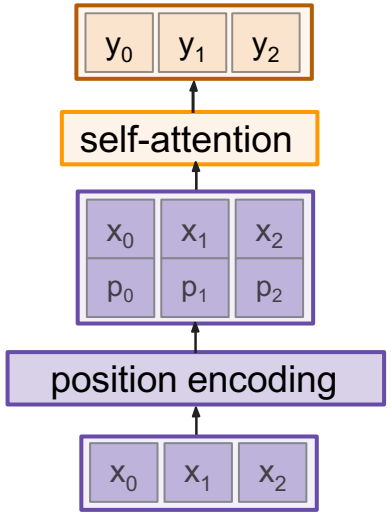
1. Learn a lookup table:
   - Learn parameters to use for $pos(t)$ for $t \in [0, T)$
   - Lookup table contains T x d parameters.

Desiderata 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 $t \in [0, T)$
   - Lookup table contains T x d parameters.
2. **Design a fixed function with the desiderata**

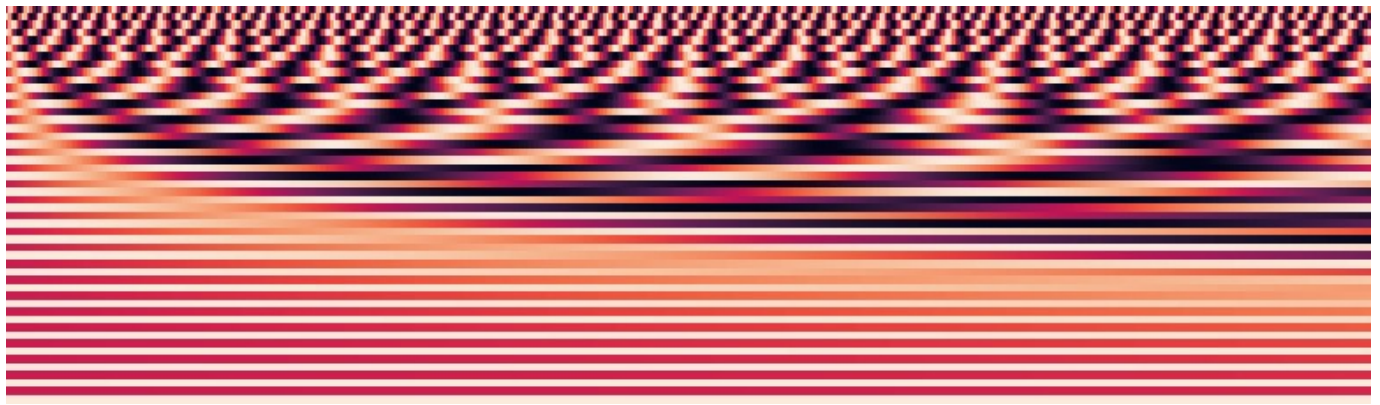
$$\begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \vdots \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_d \quad \text{where } \omega_k = \frac{1}{10000^{2k/d}}$$

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

Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

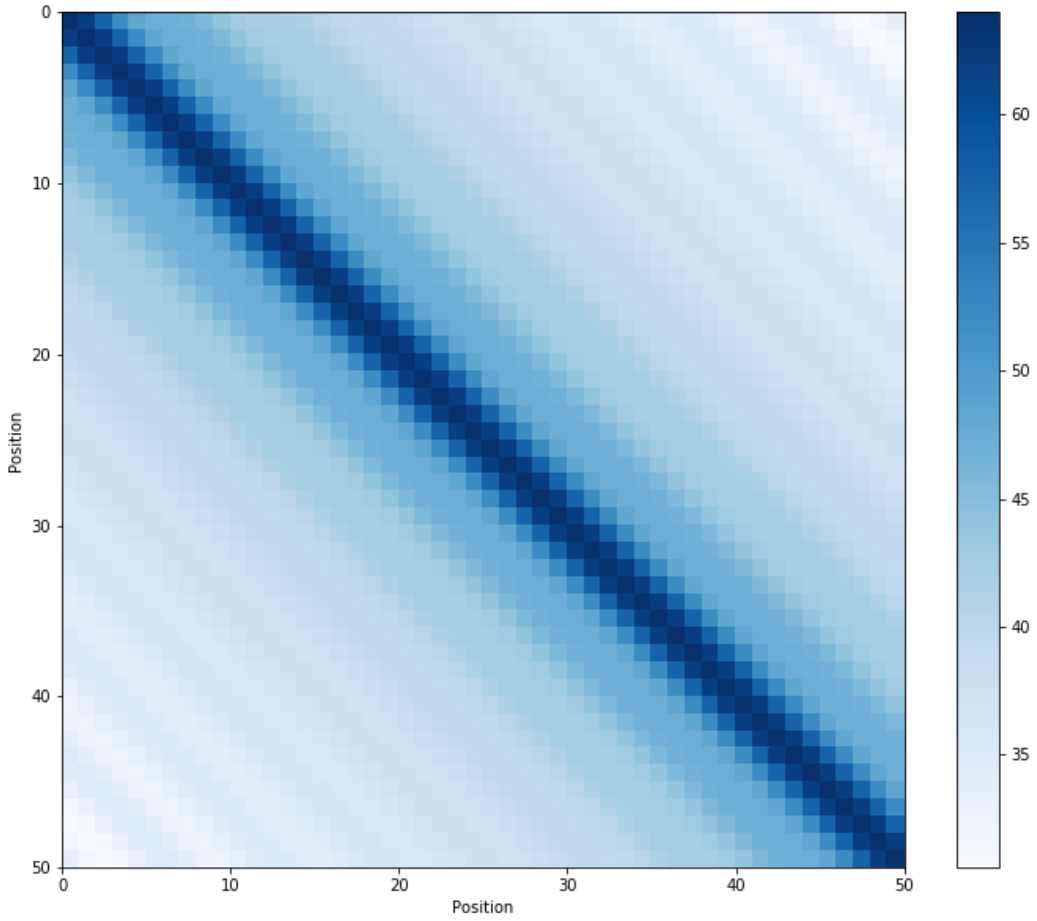
$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



- Pros:
  - Periodicity indicates that maybe "absolute position" isn't as important
  - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
  - Not learnable; also the extrapolation doesn't really work!

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

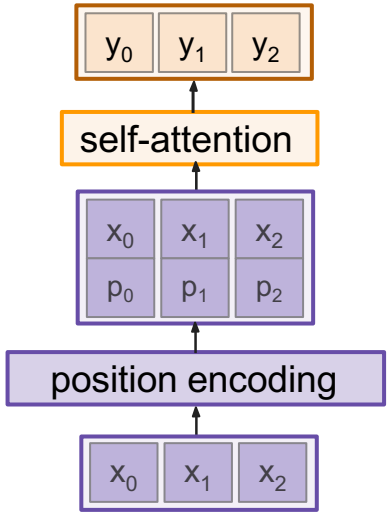
Dot product of position embeddings



Dot product of position embeddings for all time-steps

Source: Kazemnejad's blog

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 $t \in [0, T)$
   - Lookup table contains T x d parameters.
2. **Design a fixed function with the desiderata**

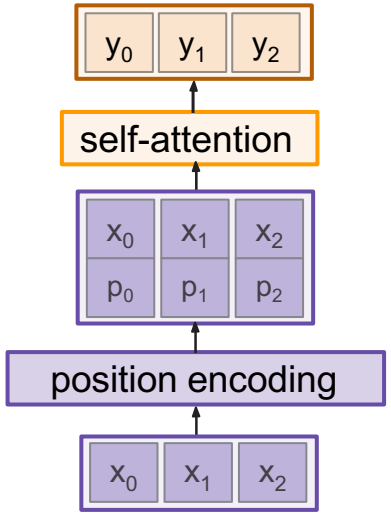
$$p(t) = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \vdots \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_d \quad \text{where } \omega_k = \frac{1}{10000^{2k/d}}$$

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

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

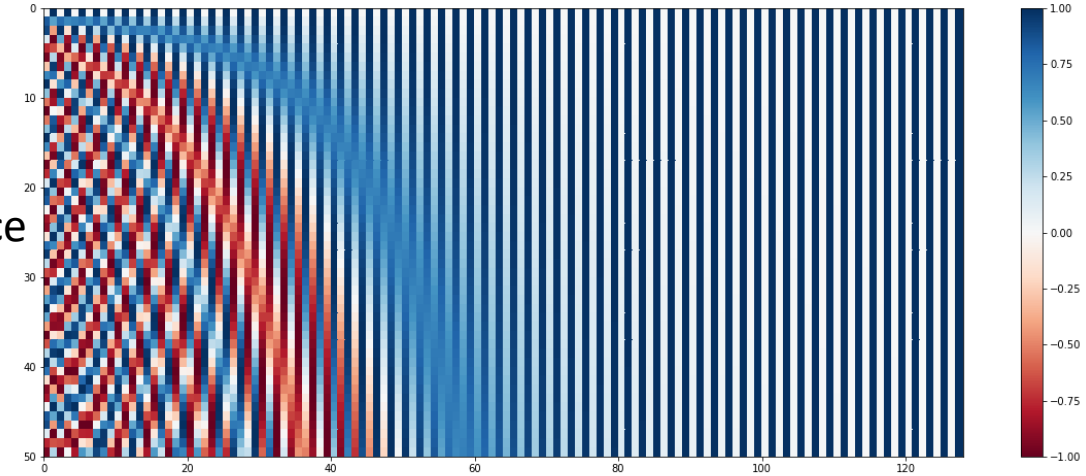
Positional encoding



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, $p_j = pos(j)$



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

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

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

Fixing the first self-attention problem: Sequence order

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$\boldsymbol{p}_t \in \mathbb{R}^d, \text{ for } t \in \{1,2,\ldots,T\} \text{ are position vectors}$$

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!

$$\tilde{\boldsymbol{x}}_t = \boldsymbol{x}_t + \boldsymbol{p}_t$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

Fixing the first self-attention problem: Sequence order

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$$p_t \in \mathbb{R}^d, \text{ for } t \in \{1,2,\dots,T\} \text{ are position vectors}$$

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!

Let $\tilde{v}_i$ $\tilde{k}_i, \tilde{q}_i$ be our old values, keys, and queries.

$$v_i = \tilde{v}_i + p_i$$
$$q_i = \tilde{q}_i + p_i$$
$$k_i = \tilde{k}_i + p_i$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add…

Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters!
  Learn a matrix $p \in \mathbb{R}^{d \times T}$, *and let each* p_i *be a column of that matrix!*

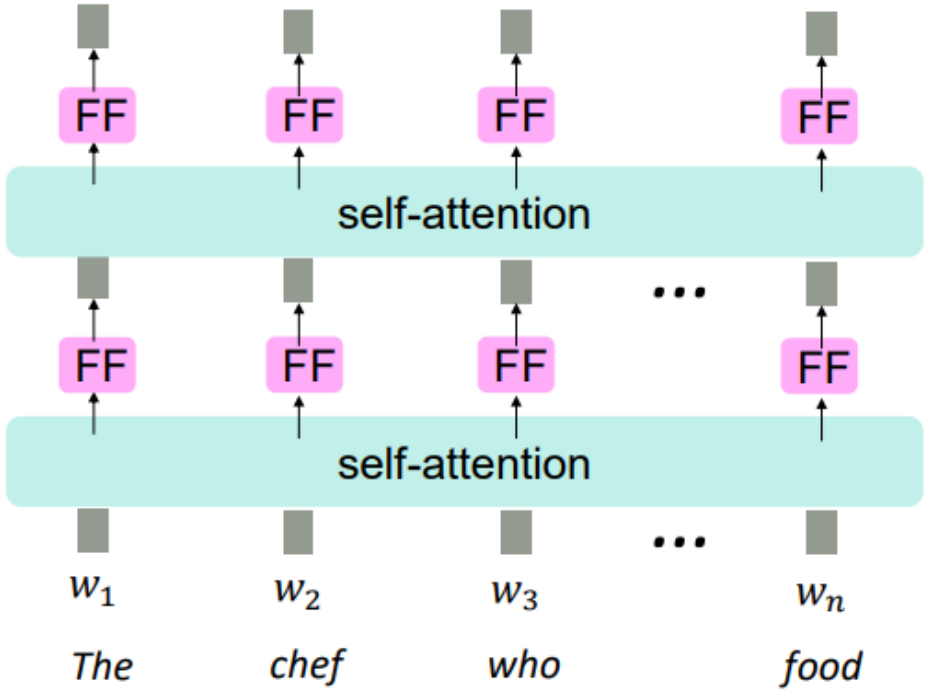
- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside $1, \ldots, T$.
- Most systems use this!

- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

Adding nonlinearities after self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors (Why? Look at the notes!)
- Easy fix: add a feed-forward network to post-process each output vector

$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \, \text{output}_i + b_1) + b_2$$

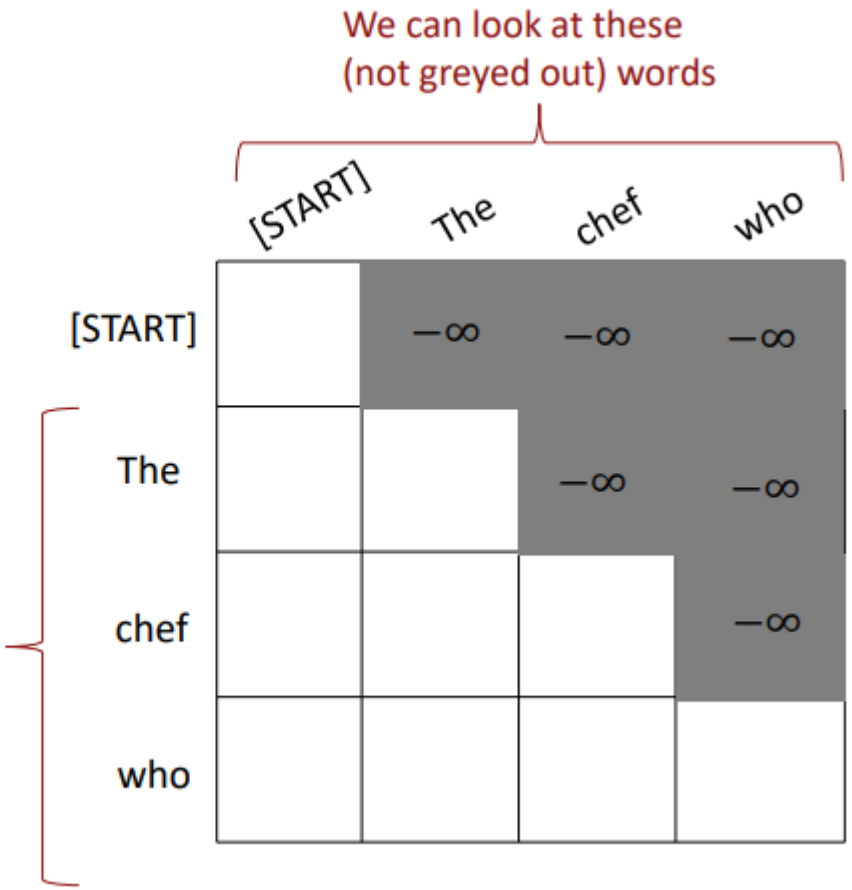


Intuition: the FF network processes the result of attention

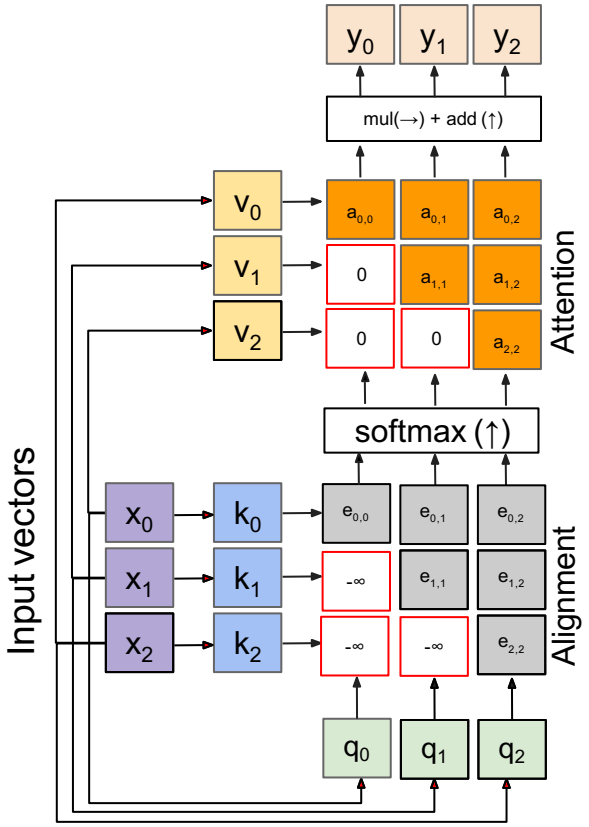
Masking the future in self-attention

- To use self-attention in decoders, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of keys and queries to include only past words. (Inefficient!)
- To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$

$$e_{ij} = \begin{cases} q_i^\top k_j, j \leq i \\ -\infty, j > i \end{cases}$$



Masked self-attention layer



Outputs:

context vectors: y ($shape$: D_v)

Operations:

Key vectors: $\mathrm{k} = W_k^T x$

Value vectors: $\mathrm{v} = W_v^T x$

Query vectors: $\mathrm{q} = W_q^T x$

Alignment: $e_{i,j} = q_j . k_i / \sqrt{D}$

Attention: $\mathrm{a} = softmax(e)$

Output: $y_j = \sum_i a_{i,j} v_v$

Inputs:

Input vectors: x ($shape$: $N \times D$)

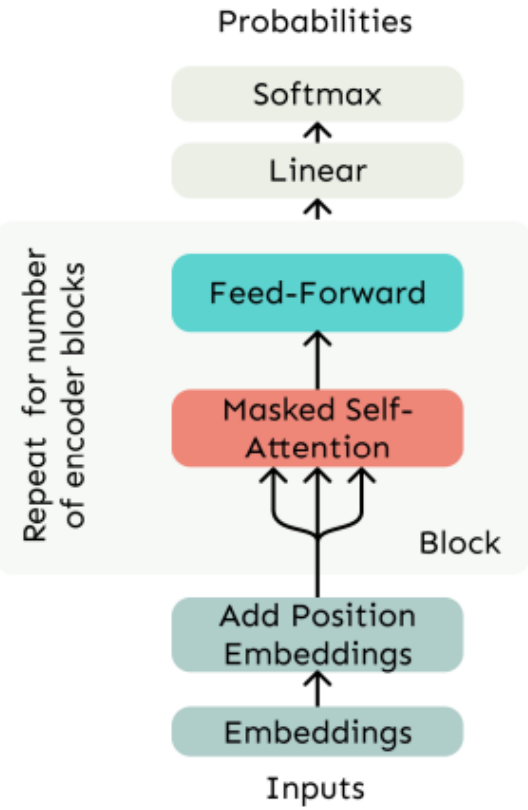
- Prevent vectors from looking at future vectors.
- Manually set alignment scores to -infinity

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

Barriers and solutions for Self-Attention as a building block

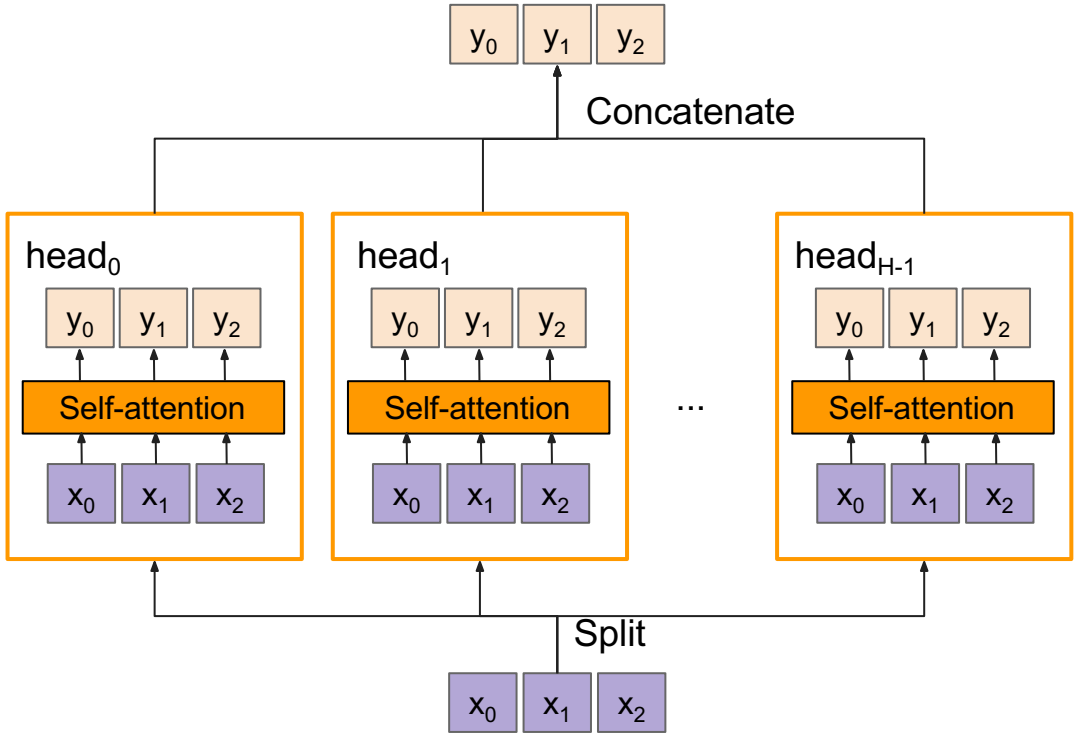
Barriers	Solutions
• Doesn't have an inherent notion of order!	• Add position representations to the inputs
• No nonlinearities for deep learning magic! It's all just weighted averages	• Easy fix: apply the same feedforward network to each self-attention output.

• Need to ensure we don't "look at the future" when predicting a sequence
  • Like in machine translation
  • Or language modeling
	• Mask out the future by artificially setting attention weights to 0!

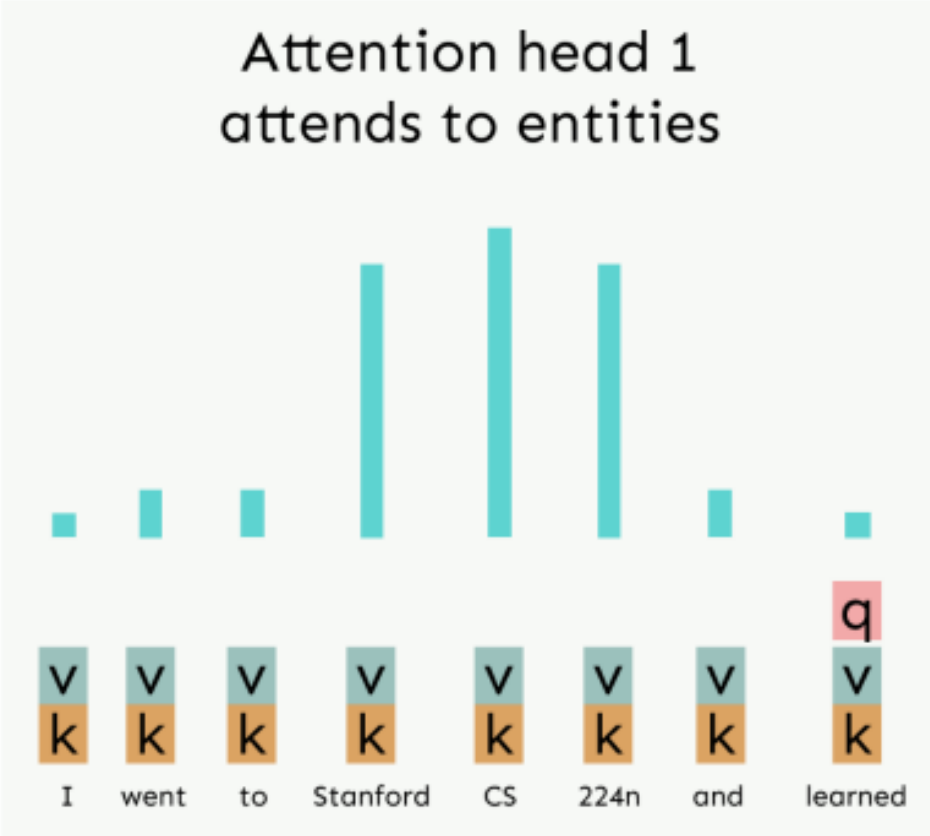


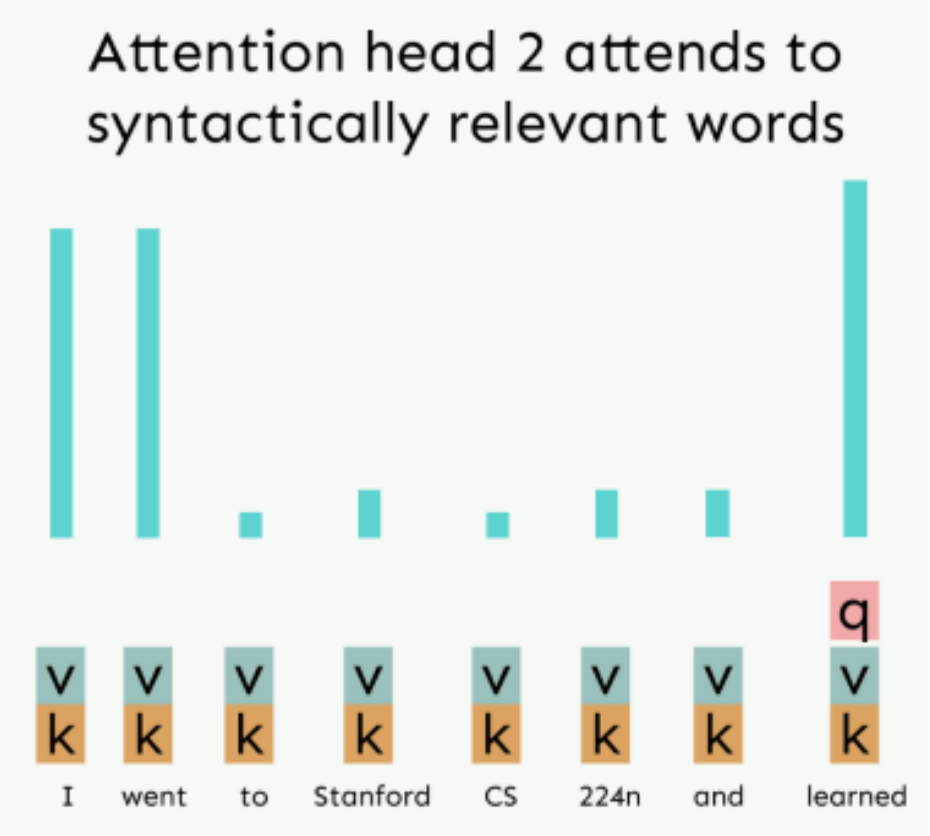
Multi-head self-attention layer

- Multiple self-attention heads in parallel



Hypothetical example of multi-head attention





I went to Stanford CS 224n and learned

Multi-headed attention

- We'll define multiple attention "heads" through multiple W_q, W_k, and W_v matrices
- $W_q^l, W_k^l, W_v^l \in \mathbb{R}^{d \times \frac{d}{h}} \qquad l = 1, \ldots, h$
- Each attention head performs attention independently:

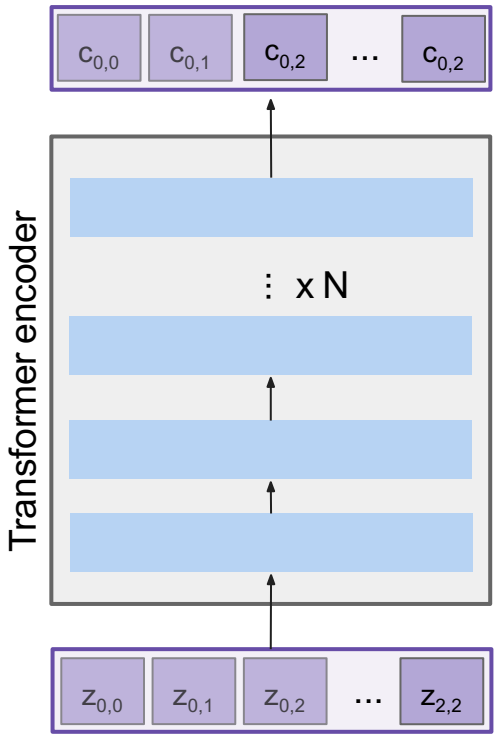
$$output_l = softmax(\frac{XW_q^l(XW_k^l)^T}{\sqrt{d/h}})XW_v^l \qquad output_l \in \mathbb{R}^{T \times \frac{d}{h}}$$

$$Y \in \mathbb{R}^{d \times d}$$

- $output = [output_1 \ldots output_h]Y$
- Each head gets to "look" at different things, and construct value vectors differently.

For scaling, we divide the attention scores by $\sqrt{d/h}$

The Transformer encoder block

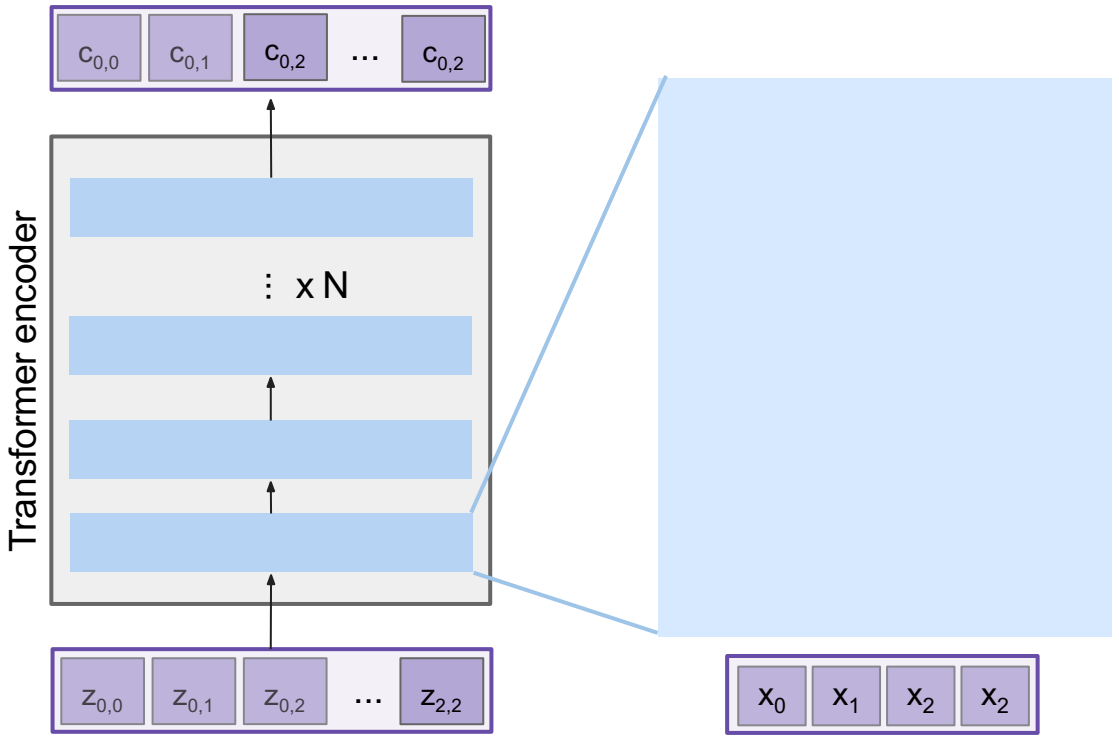


Made up of N encoder blocks.

In Vaswani et al. $\mathrm{N} = 6, D_q(.) = 512$

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

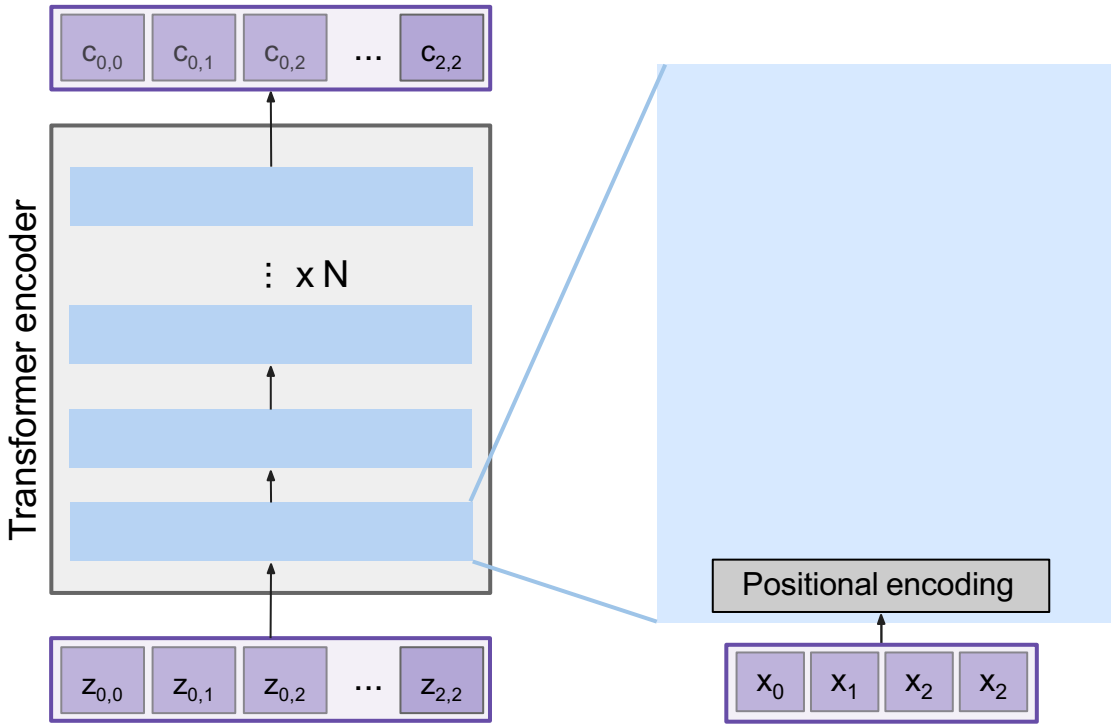
The Transformer encoder block



Let's dive into one encoder block

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

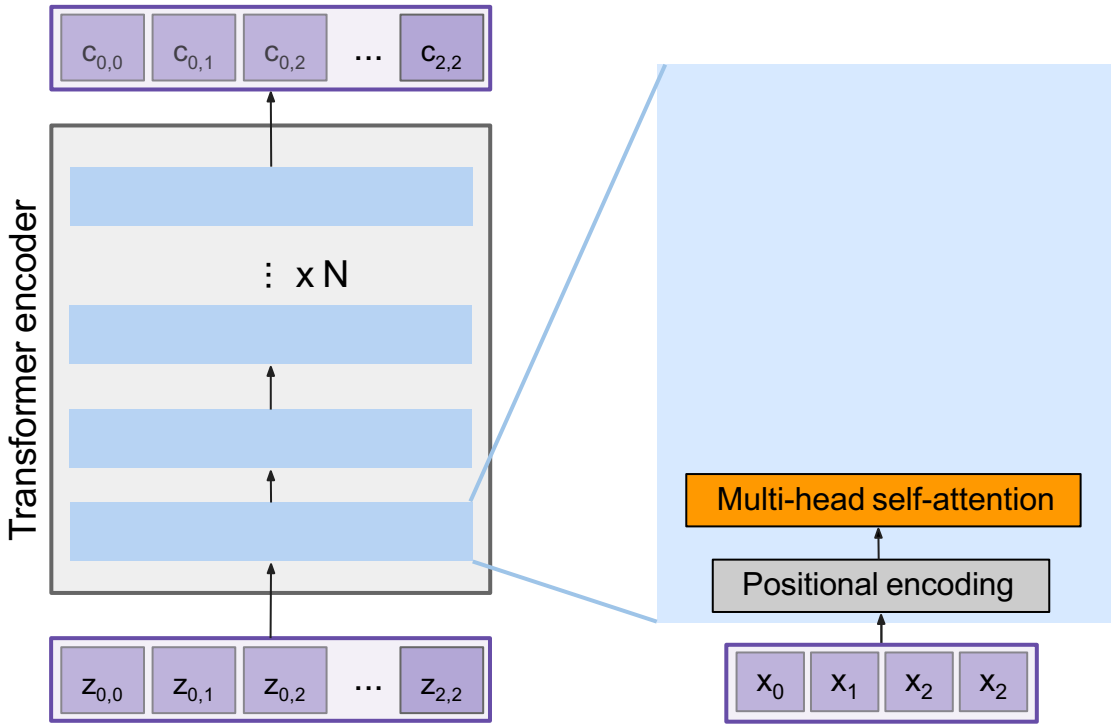
The Transformer encoder block



Add positional encoding

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

The Transformer encoder block



Attention attends over all the vectors

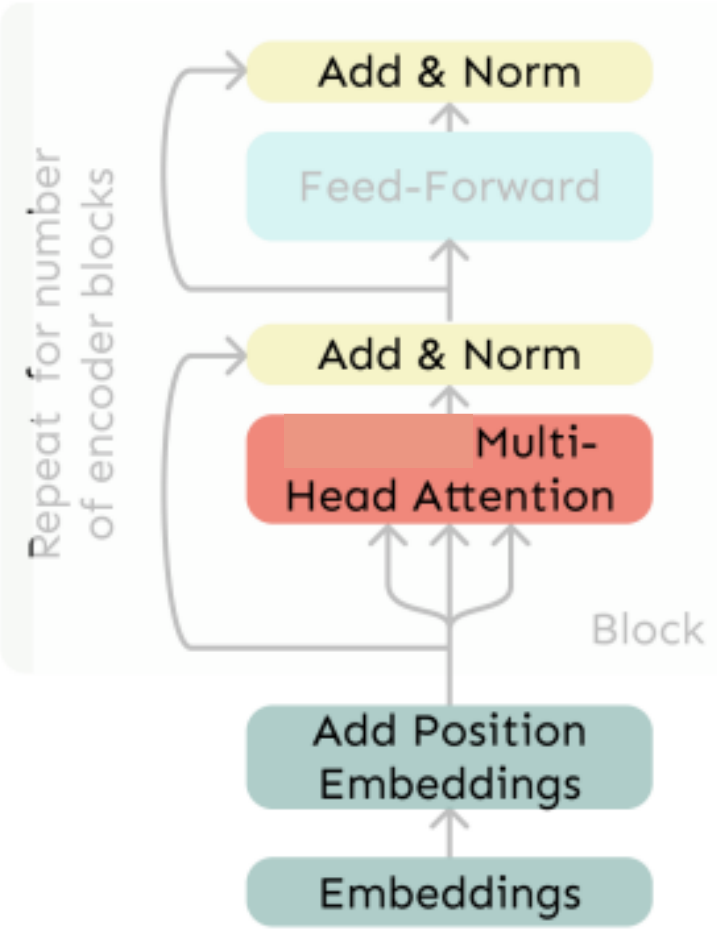
Add positional encoding

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

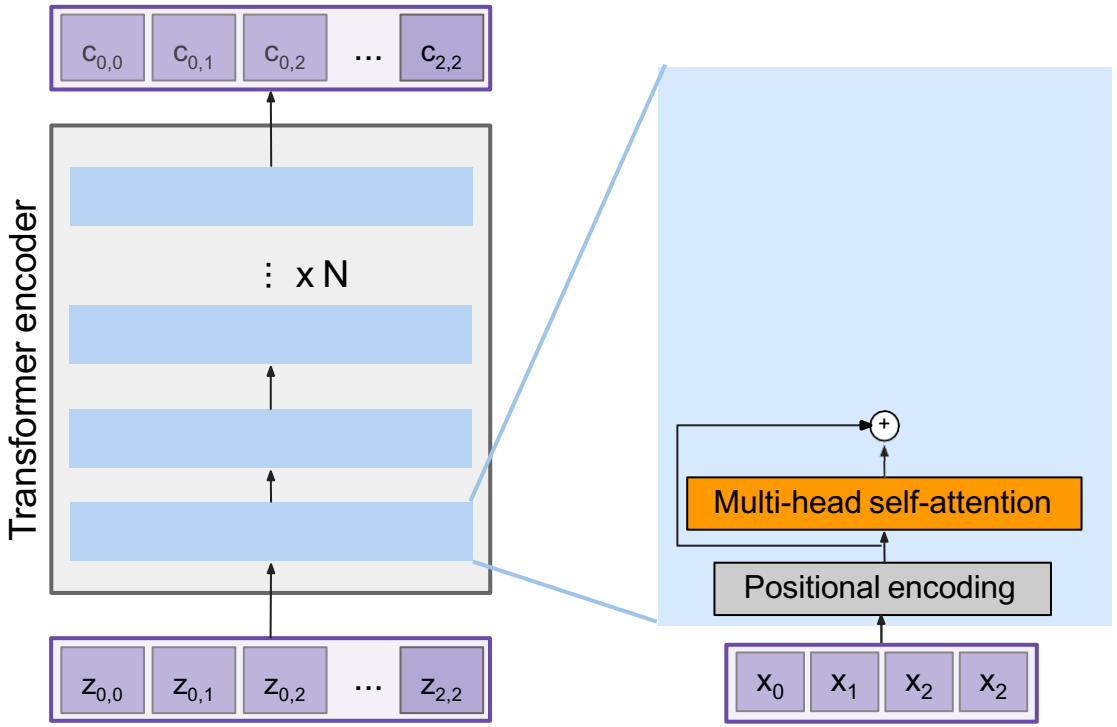
Transformer architecture

- Now that we've replaced self-attention with multi-head self-attention, we'll go through two optimization tricks that end up being :
  - Residual Connections
  - Layer Normalization

- In most Transformer diagrams, these are often written together as "Add & Norm"



The Transformer encoder block



Residual connection

Attention attends over all the vectors

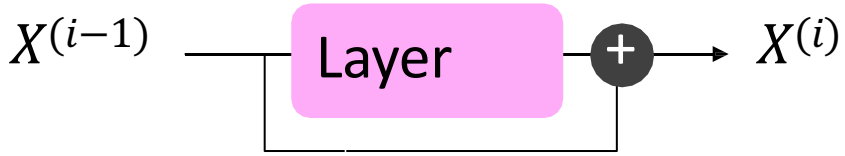
Add positional encoding

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

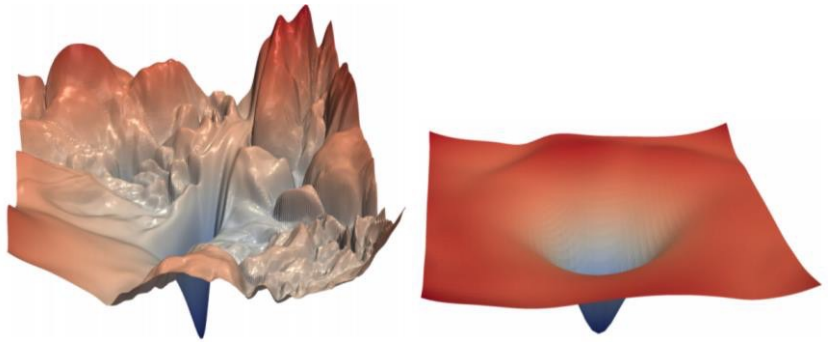
Residual connections [He et al., 2016]

- **Residual connections** are a trick to help models train better.
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)

  - We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn "the residual" from the previous layer)



  - Residual connections are thought to make the loss landscape considerably smoother (thus easier training!)



no residuals residuals

[Loss landscape visualization, Li et al., 2018, on a ResNet]

Layer normalization [Ba et al., 2016]

- **Layer normalization** is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
  - LayerNorm's success may be due to its normalizing gradients [Xu et al., 2019]

Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.

Let $\mu = \sum_{j=1}^{d} x_j$

Let $\sigma = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2}$

Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)

Then layer normalization computes:

$$output = \frac{x - \mu}{\sqrt{\sigma + \epsilon}} * \gamma + \beta$$

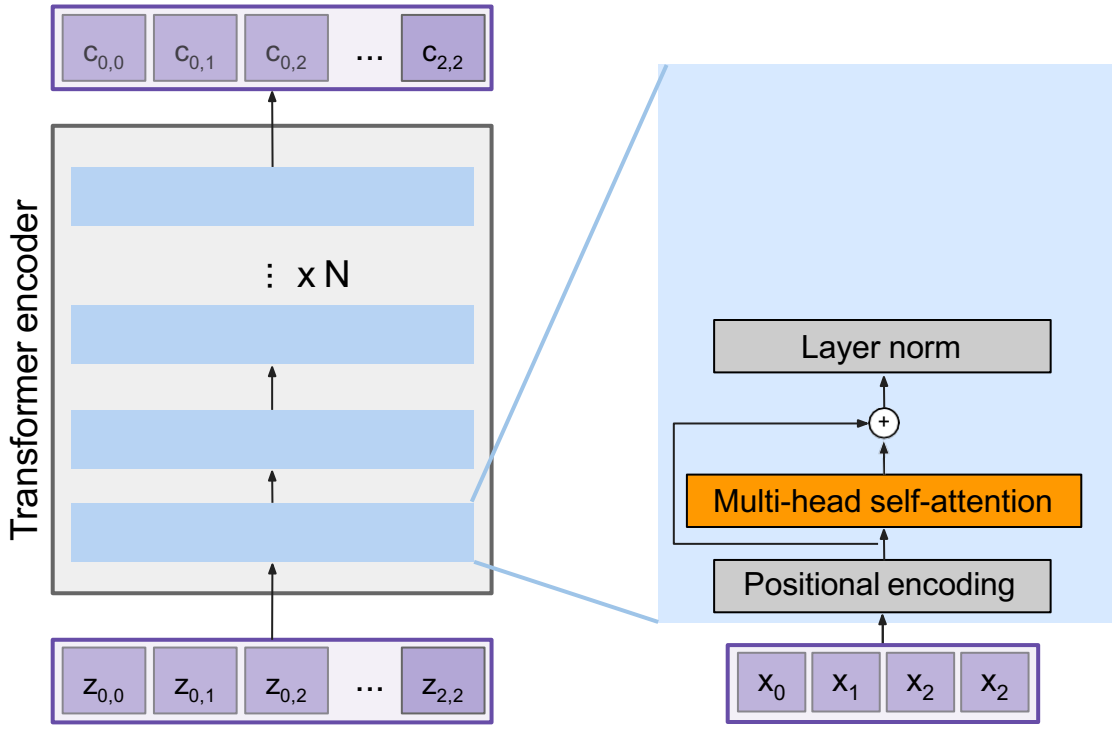
Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias

Layer Normalization vs. Batch Normalization

- Batch normalization normalizes features independently across the mini-batch.
- Layer normalization normalizes each of the samples in the batch independently across all features.

The Transformer encoder block



LayerNorm over each vector individually

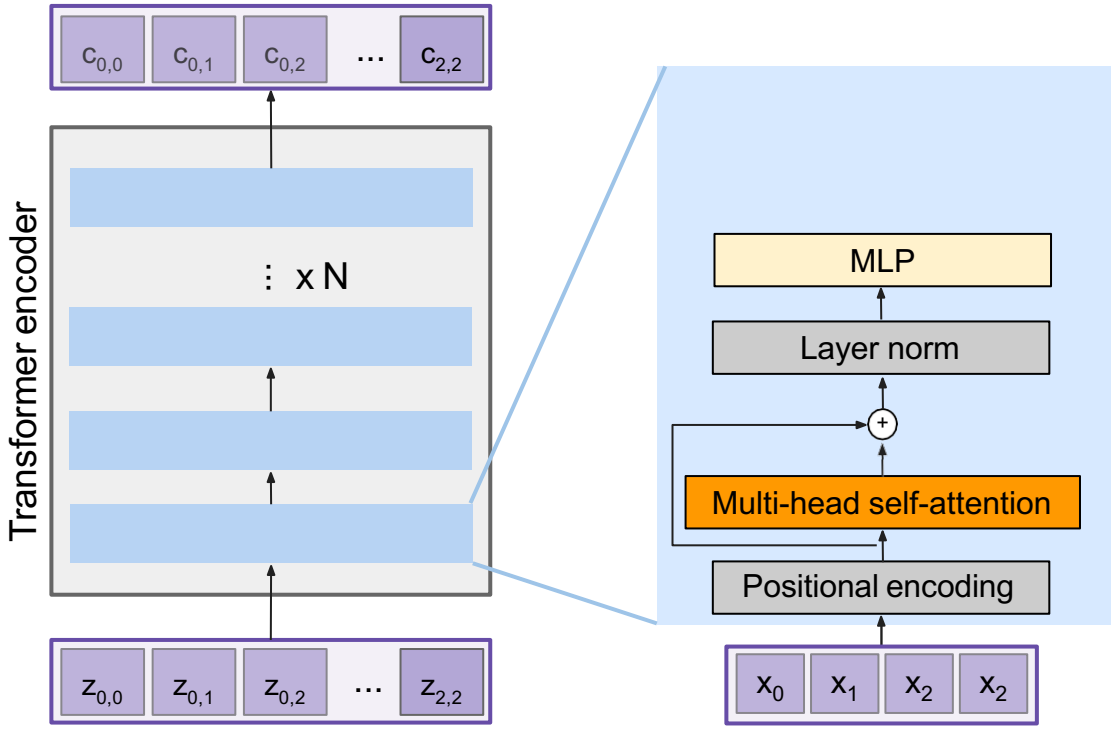
Residual connection

Attention attends over all the vectors

Add positional encoding

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

The Transformer encoder block



MLP over each vector individually

LayerNorm over each vector individually

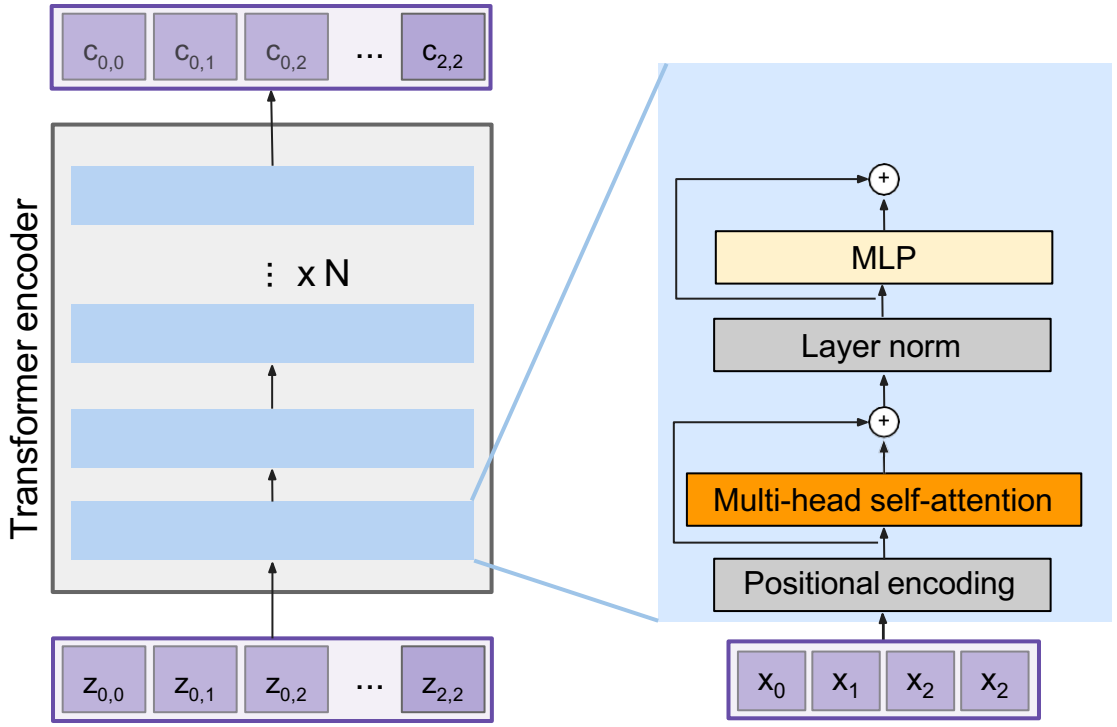
Residual connection

Attention attends over all the vectors

Add positional encoding

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

The Transformer encoder block



Residual connection

MLP over each vector individually

LayerNorm over each vector individually

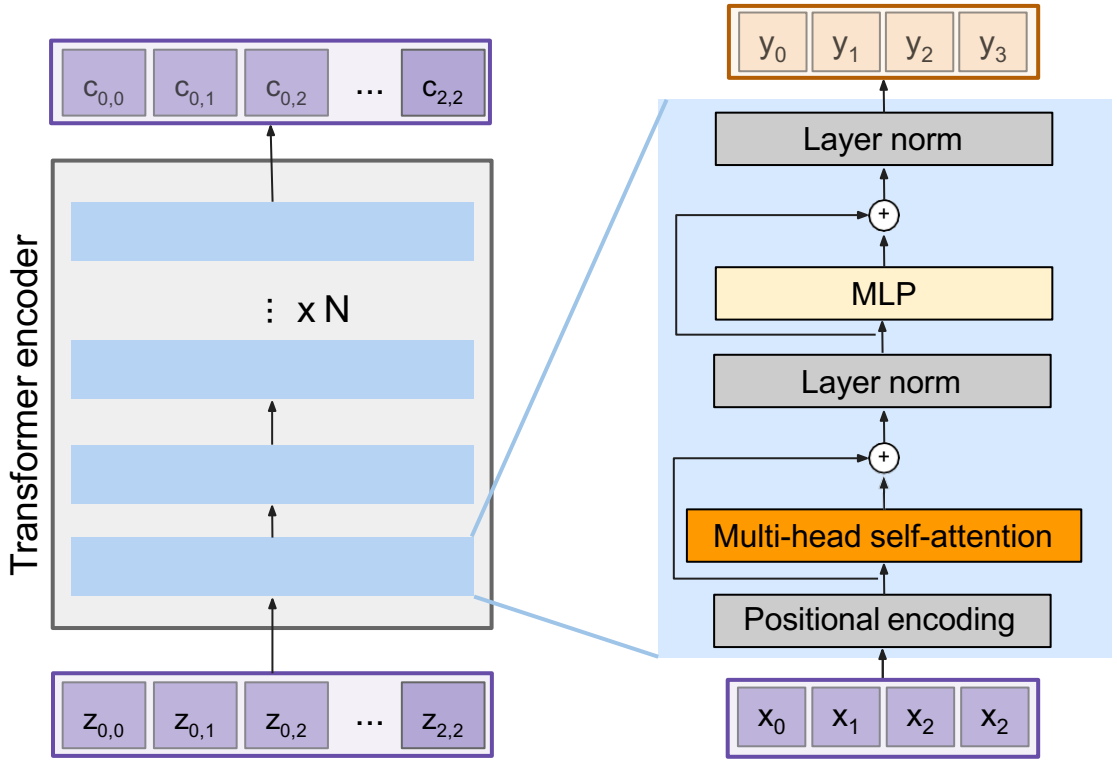
Residual connection

Attention attends over all the vectors

Add positional encoding

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

The Transformer encoder block



Transformer Encoder Block:

Inputs: Set of vectors **x**
Outputs: Set of vectors **y**

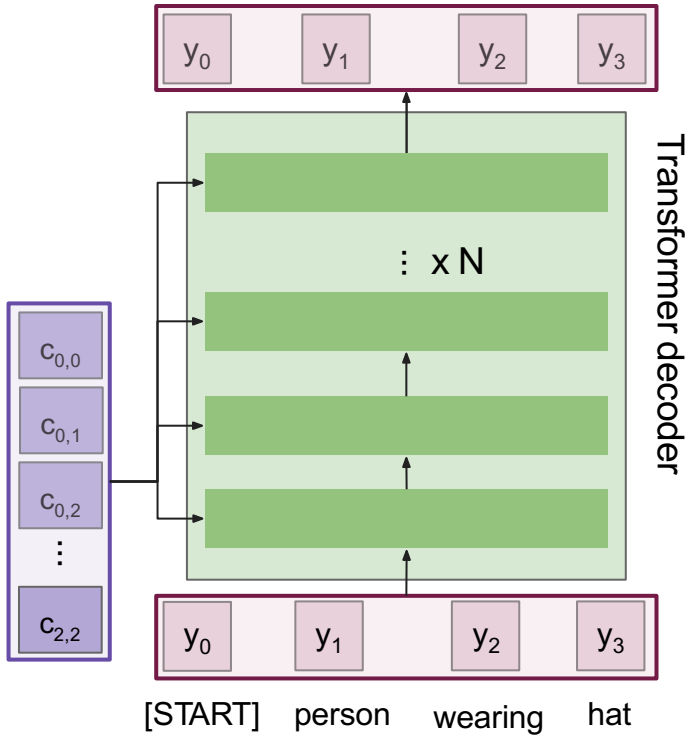
Self-attention is the only interaction between vectors.

Layer norm and MLP operate independently per vector.

Highly scalable, highly parallelizable, but high memory usage.

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

The Transformer decoder block

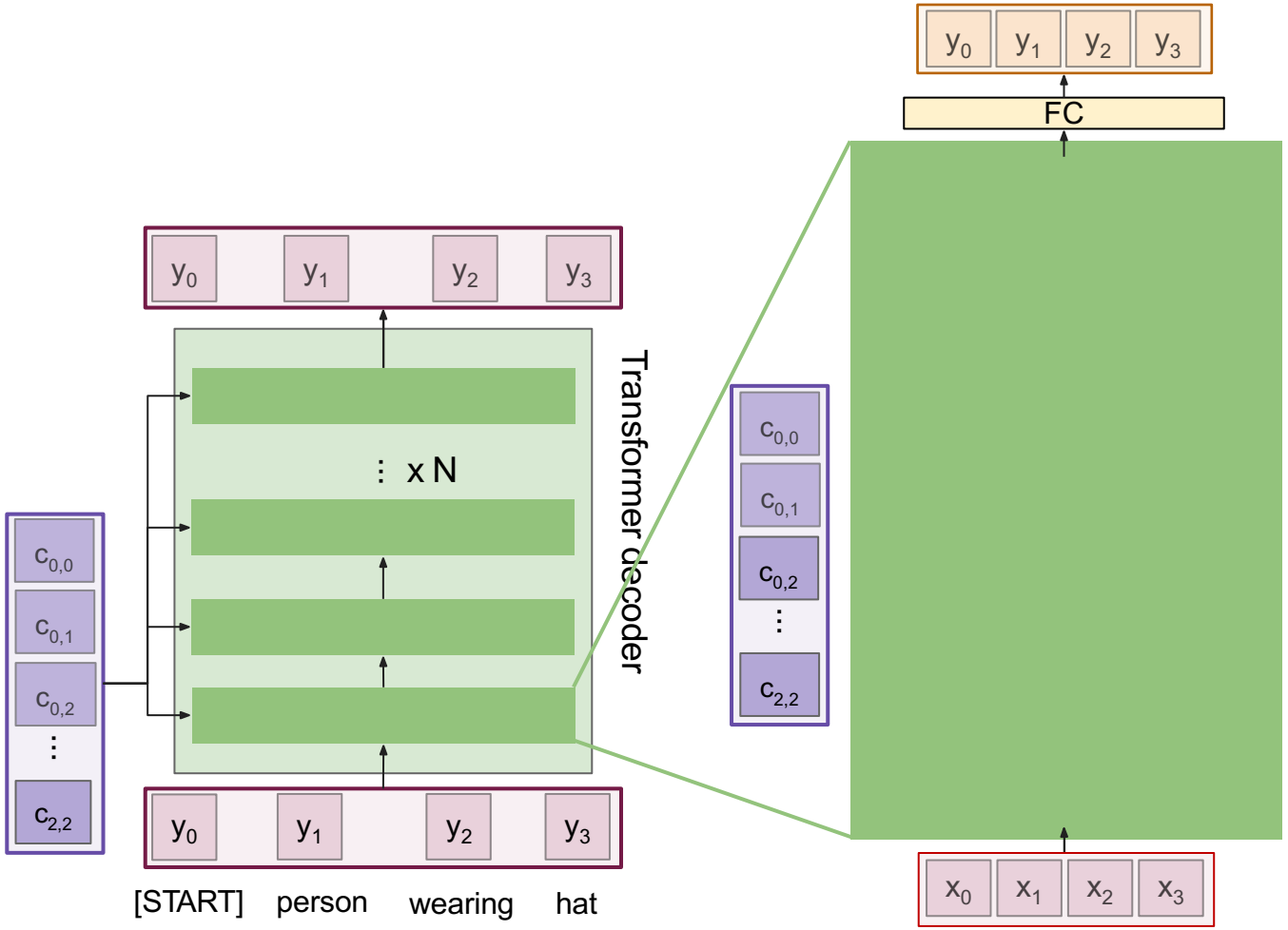


Made up of N decoder blocks.

In vaswani et al. $\mathrm{N} = 6, D_q(.) = 512$

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

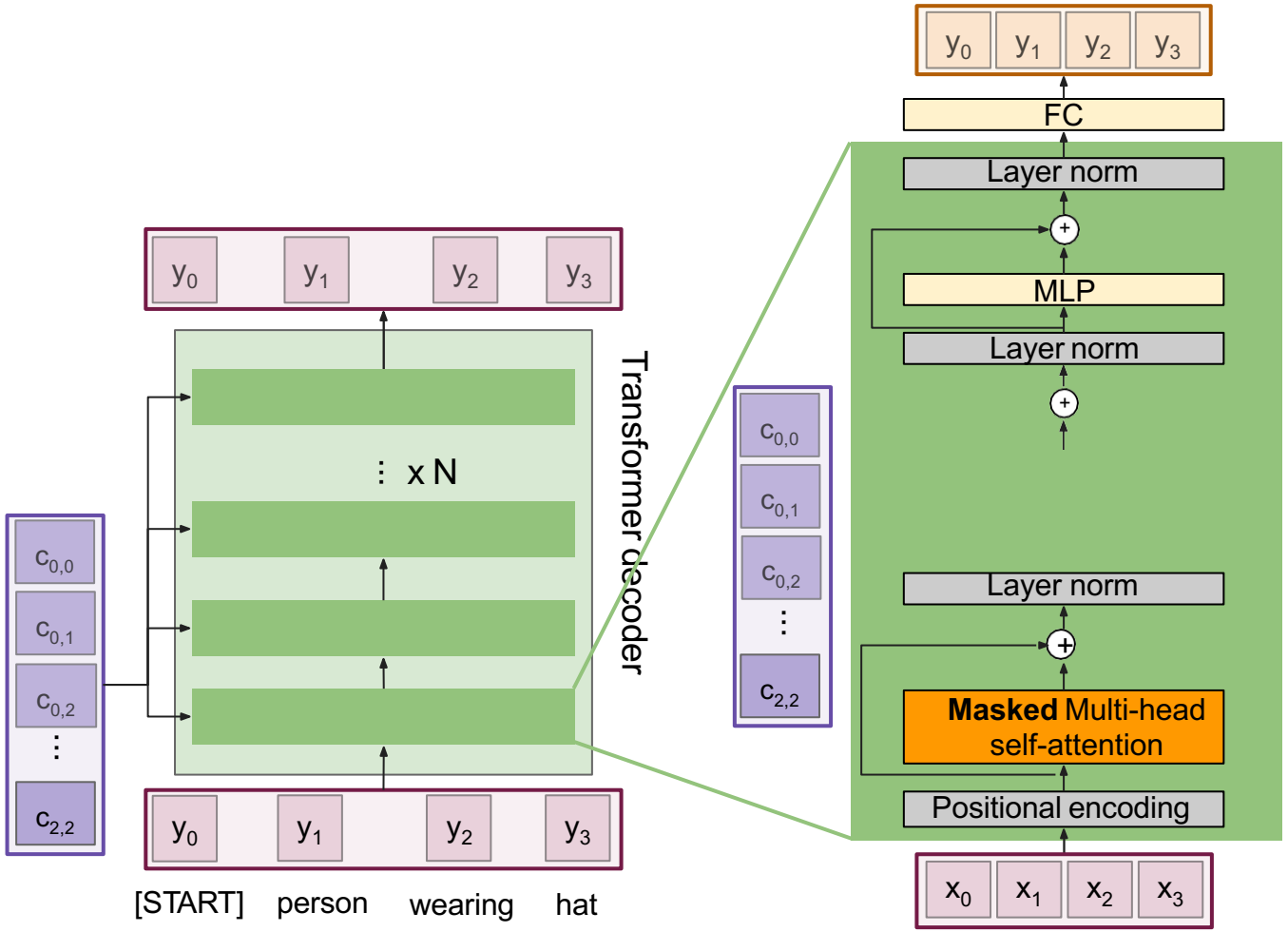
The Transformer decoder block



Let's dive into the transformer decoder block

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

The Transformer decoder block

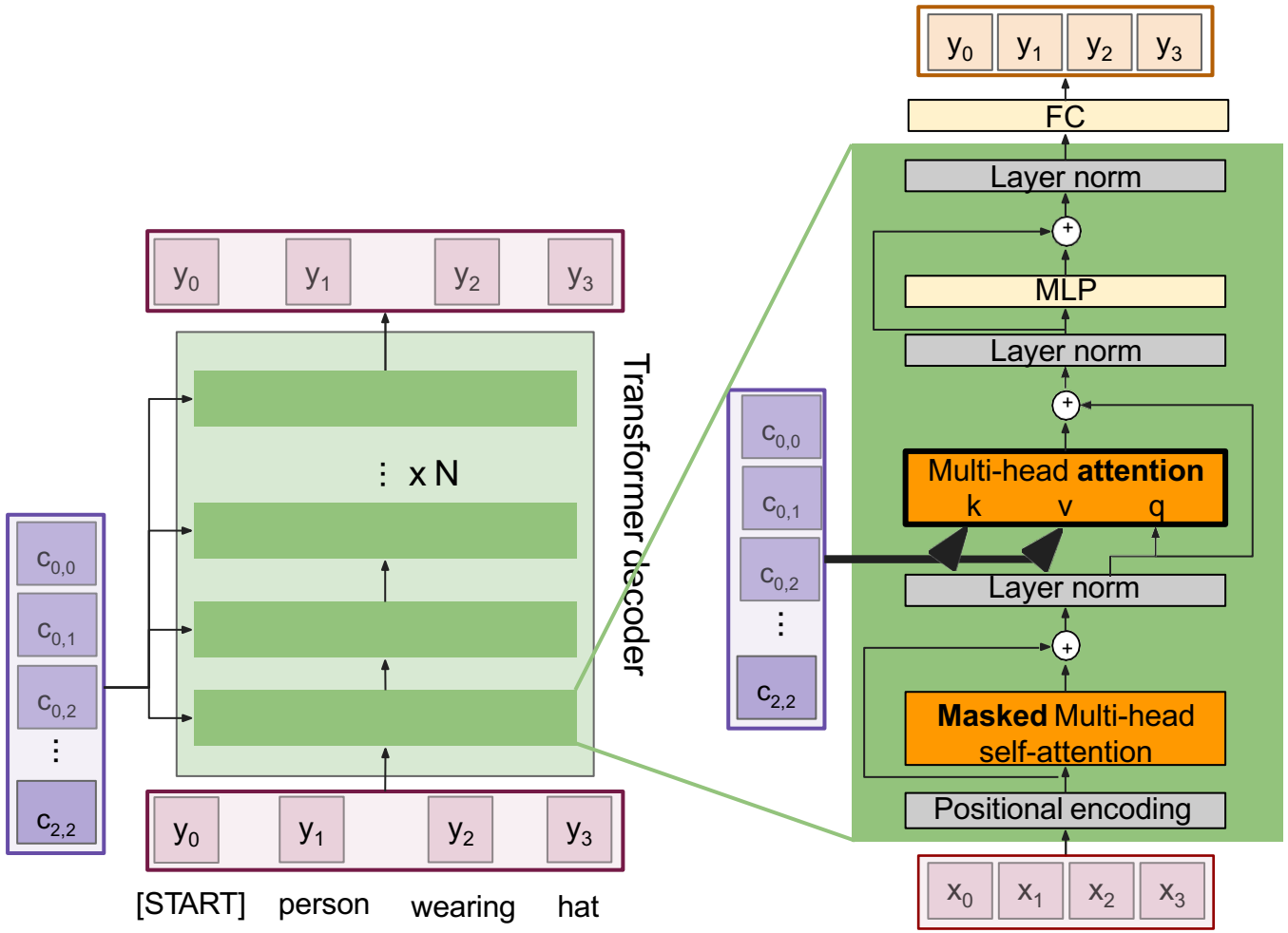


Most of the network is the same the transformer encoder.

The Transformer Decoder constrains to unidirectional context, as for language models.

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

The Transformer decoder block



Multi-head attention block attends over the transformer encoder outputs.

For image captions, this is how we inject image features into the decoder.

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

The Transformer Encoder-Decoder

- Transformer Decoder is modified to perform cross-attention to the output of the Encoder.

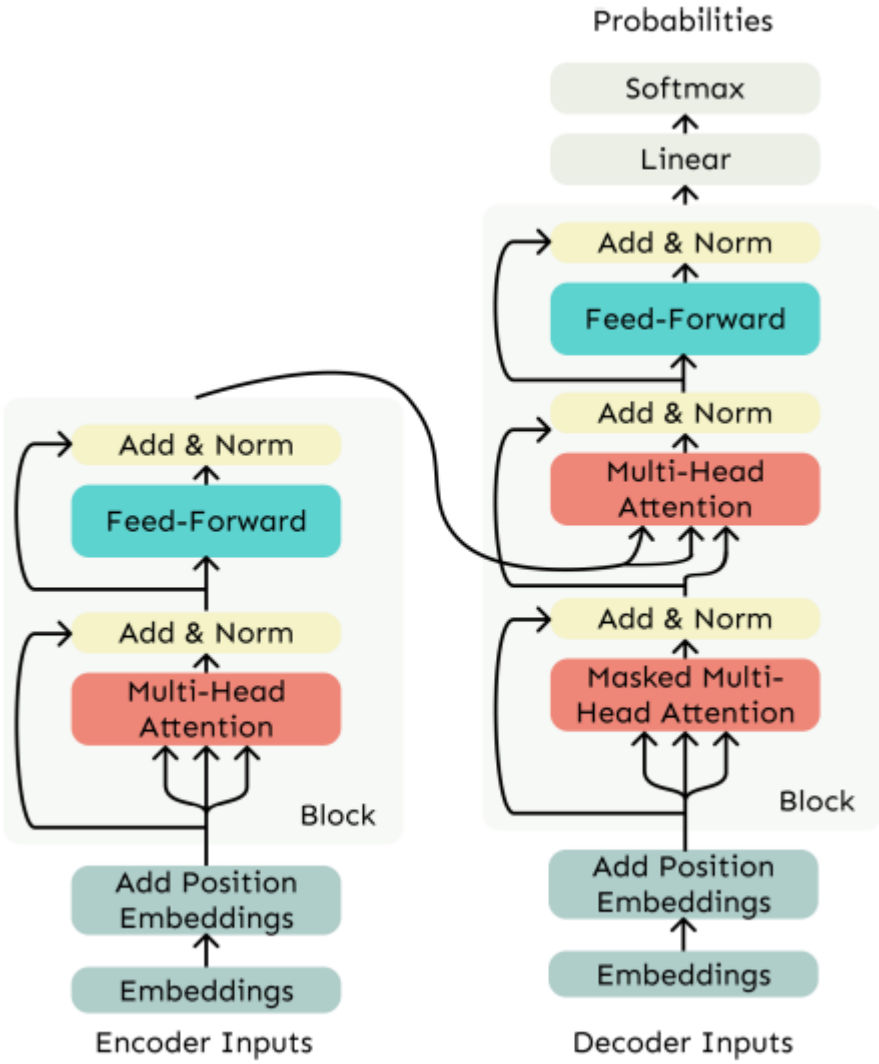
- Let $h_1, \ldots, h_n$ be output vectors from the Transformer encoder
- Let $z_1, \ldots, z_n$ be input vectors from the Transformer decoder, $z_i \in \mathbb{R}^d$

- Then keys and values are drawn from the encoder (like a memory):

$$k_i = W_k h_i \qquad v_i = W_v h_i$$

- And the queries are drawn from the decoder

$$q_i = W_Q z_i$$



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

Necessities for a self-attention building block:

- Self-attention:
  - the basis of the method.
- Position representations:
  - Specify the sequence order, since self-attention is an unordered function of its inputs.
- Nonlinearities:
  - At the output of the self-attention block
  - Frequently implemented as a simple feedforward network.
- Masking (in the decoder):
  - In order to parallelize operations while not looking at the future.
  - Keeps information about the future from “leaking” to the past.

The Transformer Encoder-Decoder

- Transformer Decoder is modified to perform cross-attention to the output of the Encoder.

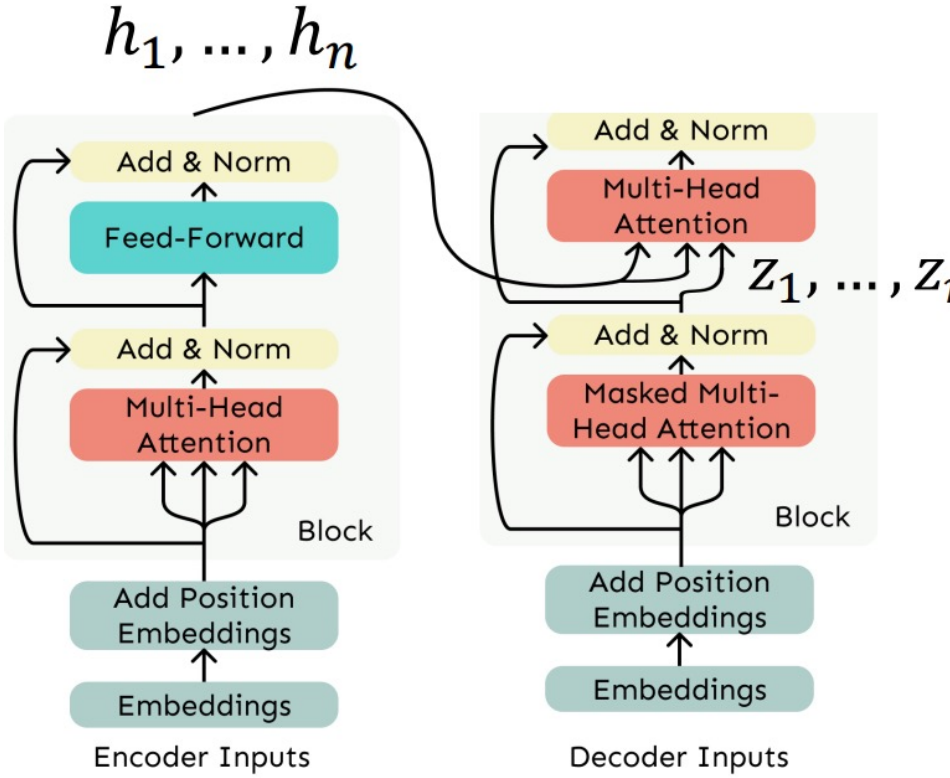
- Let $h_1, \dots, h_n$ be output vectors from the Transformer encoder
- Let $z_1, \dots, z_n$ be input vectors from the Transformer decoder, $z_i \in \mathbb{R}^d$

- Then keys and values are drawn from the encoder (like a memory):

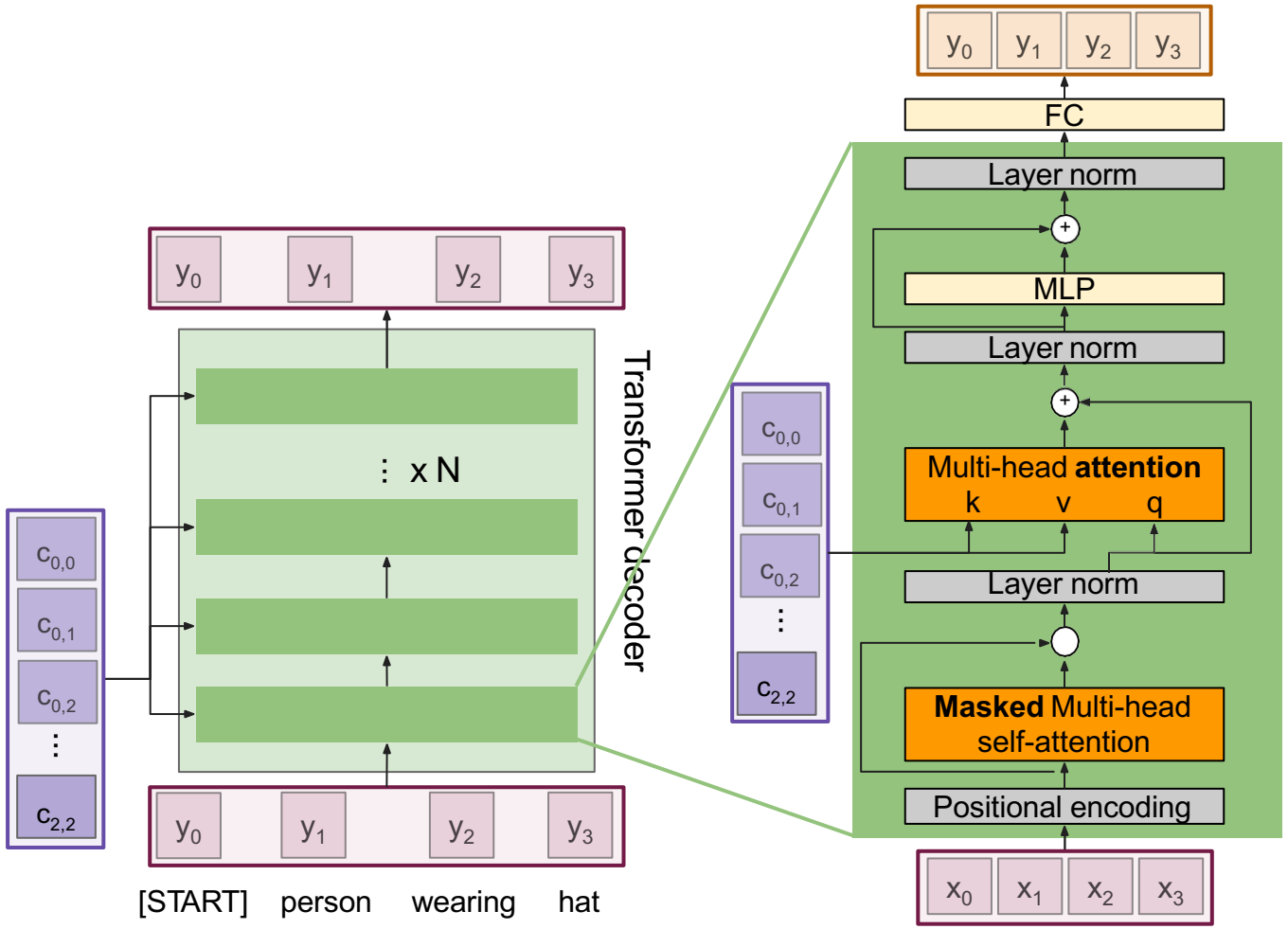
$$k_i = W_k^T h_i \qquad v_i = W_v^T h_i$$

- And the queries are drawn from the decoder

$$q_i = W_Q^T z_i$$



The Transformer decoder block



Transformer Decoder Block:

Inputs: Set of vectors **x** and Set of context vectors **c.**
Outputs: Set of vectors **y.**

Masked Self-attention only interacts with past inputs.

Multi-head attention block is NOT self-attention. It attends over encoder outputs.

Highly scalable, highly parallelizable, but high memory usage.

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

Transfomer Encoder-Decoder

- The Transformer Decoder constrains to unidirectional context, as for language models.
- What if we want bidirectional context, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we remove the masking in the self-attention.

Great Results with Transformers

- First, Machine Translation from the original Transformers paper!

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

[Test sets: WMT 2014 English-German and English-French]

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

Great Results with Transformers

- Most Transformers results also included pretraining.
- Transformers' parallelizability allows for efficient pretraining, and have made them the de-facto standard.
- On the popular aggregate benchmark GLUE, for example:
  - All top models are Transformer (and pretraining)-based.

Do Transformer Modifications Transfer?

- "Surprisingly, we find that most modifications do not meaningfully improve performance."

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223M	11.1T	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
GeLU	223M	11.1T	3.58	2.179 ± 0.003	1.838	**75.79**	**17.86**	**25.13**	26.47
Swish	223M	11.1T	3.62	2.186 ± 0.003	1.847	**73.77**	17.74	**24.34**	**26.75**
ELU	223M	11.1T	3.56	2.270 ± 0.007	1.932	67.83	16.73	23.02	26.08
GLU	223M	11.1T	3.59	2.174 ± 0.003	**1.814**	**74.20**	**17.42**	24.34	**27.12**
GeGLU	223M	11.1T	3.55	2.130 ± 0.006	**1.792**	**75.96**	**18.27**	**24.87**	**26.87**
ReGLU	223M	11.1T	3.57	2.145 ± 0.004	**1.803**	**76.17**	**18.36**	**24.87**	**27.02**
SeLU	223M	11.1T	3.55	2.315 ± 0.004	1.948	68.76	16.76	22.75	25.99
SwiGLU	223M	11.1T	3.53	2.127 ± 0.003	**1.789**	**76.00**	**18.20**	**24.34**	**27.02**
LiGLU	223M	11.1T	3.59	2.149 ± 0.005	**1.798**	**75.34**	**17.97**	**24.34**	26.53
Sigmoid	223M	11.1T	3.63	2.291 ± 0.019	1.867	**74.31**	17.51	23.02	26.30
Softplus	223M	11.1T	3.47	2.207 ± 0.011	1.850	**72.45**	17.65	**24.34**	**26.89**
RMS Norm	223M	11.1T	3.68	2.167 ± 0.008	**1.821**	**75.45**	**17.94**	**24.07**	**27.14**
Rezero	223M	11.1T	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Rezero + LayerNorm	223M	11.1T	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Rezero + RMS Norm	223M	11.1T	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	223M	11.1T	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31
24 layers, $d_{ff} = 1536, H = 6$	224M	11.1T	3.33	2.200 ± 0.007	1.843	**74.89**	17.75	**25.13**	**26.89**
18 layers, $d_{ff} = 2048, H = 8$	223M	11.1T	3.38	2.185 ± 0.005	**1.831**	**76.45**	16.83	**24.34**	**27.10**
8 layers, $d_{ff} = 4608, H = 18$	223M	11.1T	3.69	2.190 ± 0.005	1.847	**74.58**	17.69	**23.28**	**26.85**
6 layers, $d_{ff} = 6144, H = 24$	223M	11.1T	3.70	2.201 ± 0.010	1.857	**73.55**	17.59	**24.60**	**26.66**
Block sharing	65M	11.1T	3.91	2.497 ± 0.037	2.164	64.50	14.53	21.96	25.48
+ Factorized embeddings	45M	9.4T	4.21	2.631 ± 0.305	2.183	60.84	14.00	19.84	25.27
+ Factorized & shared embeddings	20M	9.1T	4.37	2.907 ± 0.313	2.385	53.95	11.37	19.84	25.19
Encoder only block sharing	170M	11.1T	3.68	2.298 ± 0.023	1.929	69.60	16.23	23.02	26.23
Decoder only block sharing	144M	11.1T	3.70	2.352 ± 0.029	2.082	67.93	16.13	**23.81**	26.08
Factorized Embedding	227M	9.4T	3.80	2.208 ± 0.006	1.855	70.41	15.92	22.75	26.50
Factorized & shared embeddings	202M	9.1T	3.92	2.320 ± 0.010	1.952	68.69	16.33	22.22	26.44
Tied encoder/decoder input embeddings	248M	11.1T	3.55	2.192 ± 0.002	1.840	**71.70**	17.72	**24.34**	26.49
Tied decoder input and output embeddings	248M	11.1T	3.57	2.187 ± 0.007	**1.827**	**74.86**	17.74	**24.87**	**26.67**
Untied embeddings	273M	11.1T	3.53	2.195 ± 0.005	**1.834**	**72.99**	17.58	**23.28**	26.48
Adaptive input embeddings	204M	9.2T	3.55	2.250 ± 0.002	1.899	66.57	16.21	**24.07**	**26.66**
Adaptive softmax	204M	9.2T	3.60	2.364 ± 0.005	1.982	**72.91**	16.67	21.16	25.56
Adaptive softmax without projection	223M	10.8T	3.43	2.229 ± 0.009	1.914	**71.82**	17.10	23.02	25.72
Mixture of softmaxes	232M	16.3T	2.24	2.227 ± 0.017	**1.821**	**76.77**	17.62	22.75	**26.82**
Transparent attention	223M	11.1T	3.33	2.181 ± 0.014	1.874	54.31	10.40	21.16	**26.80**
Dynamic convolution	257M	11.8T	2.65	2.403 ± 0.009	2.047	58.30	12.67	21.16	17.03
Lightweight convolution	224M	10.4T	4.07	2.370 ± 0.010	1.989	63.07	14.86	23.02	24.73
Evolved Transformer	217M	9.9T	3.09	2.220 ± 0.003	1.863	**73.67**	10.76	**24.07**	26.58
Synthesizer (dense)	224M	11.4T	3.47	2.334 ± 0.021	1.962	61.03	14.27	16.14	**26.63**
Synthesizer (dense plus)	243M	12.6T	3.22	2.191 ± 0.010	1.840	**73.98**	16.96	**23.81**	**26.71**
Synthesizer (dense plus alpha)	243M	12.6T	3.01	2.180 ± 0.007	**1.828**	**74.25**	17.02	**23.28**	26.61
Synthesizer (factorized)	207M	10.1T	3.94	2.341 ± 0.017	1.968	62.78	15.39	**23.55**	26.42
Synthesizer (random)	254M	10.1T	4.08	2.326 ± 0.012	2.009	54.27	10.35	19.56	26.44
Synthesizer (random plus)	292M	12.0T	3.63	2.189 ± 0.004	1.842	**73.32**	17.04	**24.87**	26.43
Synthesizer (random plus alpha)	292M	12.0T	3.42	2.186 ± 0.007	**1.828**	**75.24**	17.08	**24.08**	26.39
Universal Transformer	84M	40.0T	0.88	2.406 ± 0.036	2.053	70.13	14.09	19.05	23.91
Mixture of experts	648M	11.7T	3.20	2.148 ± 0.006	**1.785**	**74.55**	**18.13**	**24.08**	**26.94**
Switch Transformer	1100M	11.7T	3.18	2.135 ± 0.007	**1.758**	**75.38**	**18.02**	**26.19**	**26.81**
Funnel Transformer	223M	1.9T	4.30	2.288 ± 0.008	1.918	67.34	16.26	22.75	23.20
Weighted Transformer	280M	71.0T	0.59	2.378 ± 0.021	1.989	69.04	16.98	23.02	26.30
Product key memory	421M	386.6T	0.25	2.155 ± 0.003	**1.798**	**75.16**	17.04	**23.55**	**26.73**

Do Transformer Modifications Transfer Across Implementations and Applications?

Sharan Narang* Hyung Won Chung Yi Tay William Fedus

Thibault Fevry† Michael Matena† Karishma Malkan† Noah Fiedel

Noam Shazeer Zhenzhong Lan† Yanqi Zhou Wei Li

Nan Ding Jake Marcus Adam Roberts Colin Raffel†

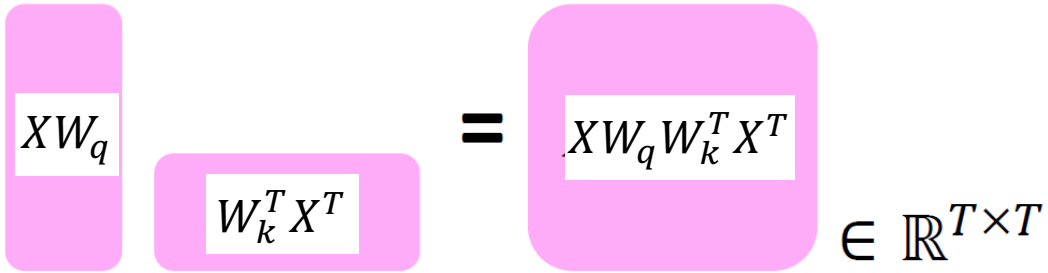
What would we like to fix about the Transformer?

- Quadratic compute in self-attention:
  - Computing all pairs of interactions means our computation grows quadratically with the sequence length!
  - For recurrent models, it only grew linearly!

- Position representations:
  - Are simple absolute indices the best we can do to represent position?
  - Relative linear position attention [Shaw et al., 2018]

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable
- However, its total number of operations grows as $O(T^2 d)$, where T is the sequence length, and d is the dimensionality



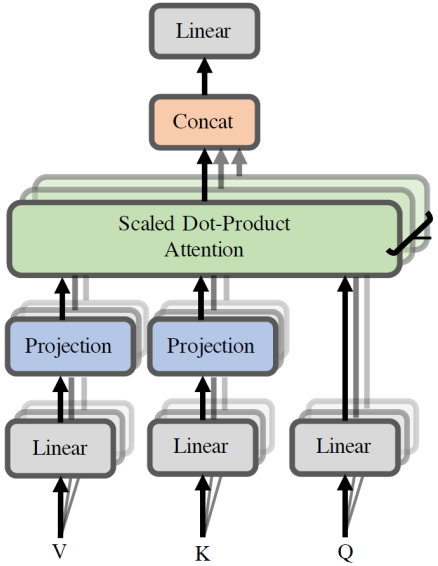
Need to compute all pairs of interactions!
$O(T^2 d)$

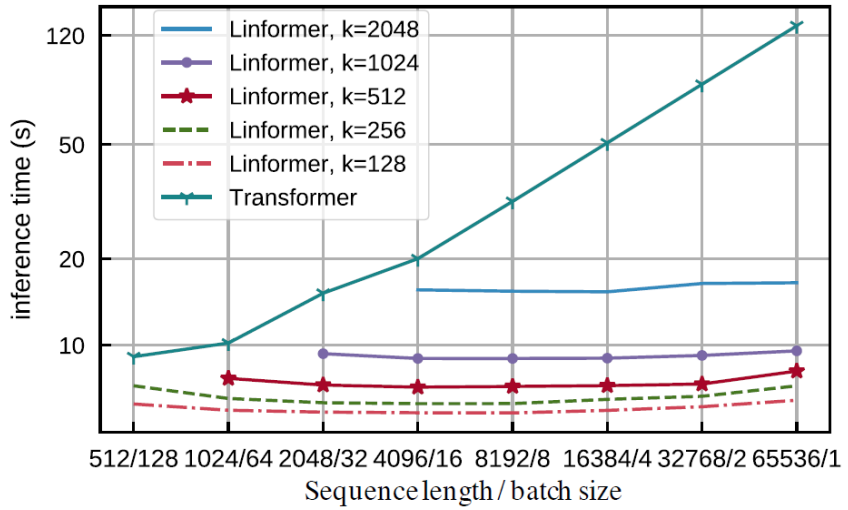
- Think of d as around 1,000
  - So, for a single short sentence, $T \leq 30; T^2 \leq 900$
  - In practice, we set a bound like T = 512
  - But what if we'd like $T \geq 10{,}000$? For example, to work on long documents?

Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, Linformer [Wang et al., 2020]

Key idea: map the sequence length dimension to a lower-dimensional space for values and keys

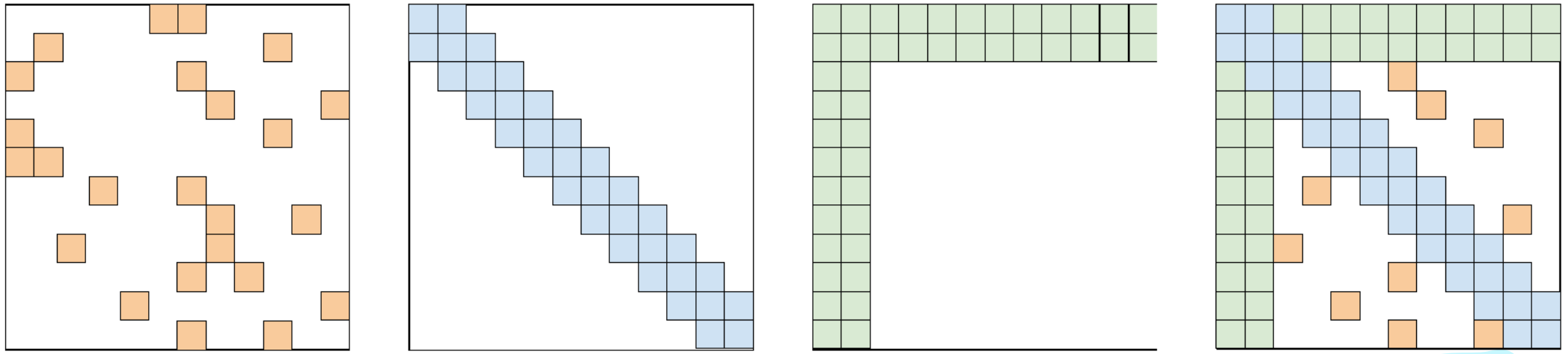




Recent work on improving on quadratic self-attention cost

- And a more recent work is: BigBird [Zaheer et al., 2021]

Key idea: replace all-pairs interactions with a family of other interactions, like random interactions, local windows and looking at everything.



(a) Random attention (b) Window attention (c) Global Attention (d) BIGBIRD

(a) random attention, (b) sliding window attention, (c) global attention, (d) the combined BIGBIRD model.

White color indicates absence of attention.

Do we even need to remove the quadratic cost of attention?

- As Transformers grow larger, a larger and larger percent of compute is outside the self-attention portion, despite the quadratic cost.
- In practice, almost no large Transformer language models use anything but the quadratic cost attention we've presented here.
- The cheaper methods tend not to work as well at scale.
- So, is there no point in trying to design cheaper alternatives to self-attention?
- Or would we unlock much better models with much longer contexts (>100k tokens?) if we were to do it right?

Summary

- Adding **attention** to RNNs allows them to "attend" to different parts of the input at every time step
- The **general attention layer** is a new type of layer that can be used to design new neural network architectures
- **Transformers** are a type of layer that uses **self-attention** and layer norm.
  - It is highly **scalable** and highly **parallelizable**
  - **Faster** training, **larger** models, **better** performance across language tasks
  - They are quickly replacing RNNs and LSTMs

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)