Understanding Large Language Models

Carsten Eickhoff, Michael Franke and Polina Tsvilodub

Session 03: Transformers

Main learning goals

1. Recap
   - ANNs & language models
   - contextualized embeddings
   - LSTMs
2. Attention
   - intuition, architecture, early transparency
3. Self-Attention
   - contextualized word embeddings, architecture
4. Transformers
   - transformer encoders
   - multi-headed self-attention
   - decoders

Slides adapted from: Chris Tanner CS287r

Recap

Language model

left-to-right / autoregressive / causal model

- an **autoregressive language model** is defined as a function that maps an initial token sequence to a next-token distribution:

  $$LM : w_{1:n} \mapsto \Delta(\mathscr{V})$$

  - we write $P_{LM}(w_{n+1} \mid w_{1:n})$ for the **next-token probability**
  - the **surprisal** of w_{n+1} after sequence $w_{1:n}$ is $-\log\left(P_{LM}(w_{n+1} \mid w_{1:n})\right)$
- the **sequence probability** follows from the chain rule:

  $$P_{LM}(w_{1:n}) = \prod_{i=1}^{n} P_{LM}(w_i \mid w_{1:i-1})$$
- measures of **goodness of fit** for observed sequence $w_{1:n}$:
  - **perplexity**:

    $$\mathrm{PP}_{LM}(w_{1:n}) = P_{LM}(w_{1:n})^{-\frac{1}{n}}$$
  - **average surprisal**:

    $$\text{Avg-Surprisal}_{LM}(w_{1:n}) = -\frac{1}{n}\log P_{LM}(w_{1:n})$$

$$\log \mathrm{PP}_{M}(w_{1:n}) = \text{Avg-Surprisal}_{M}(w_{1:n})$$

Recap: Language Modeling

An auto-regressive LM is one that only has access to the previous tokens (and the outputs become the inputs).

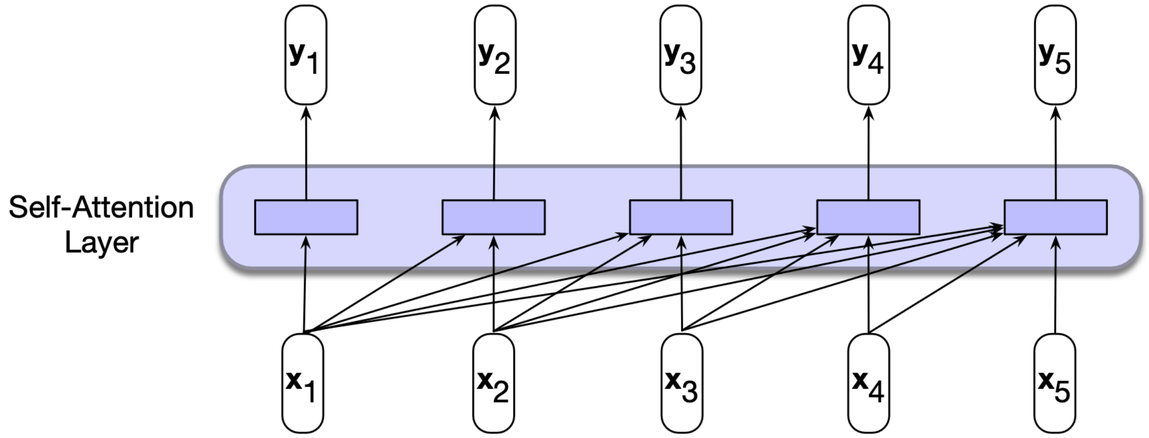
Evaluation: Perplexity

A masked LM can peak ahead, too. It "masks" a word within the context (e.g., the center word).

Evaluation: downstream NLP tasks that use the learned embeddings.

Both of these can produce useful word embeddings!

Autoregressive / left-to-right

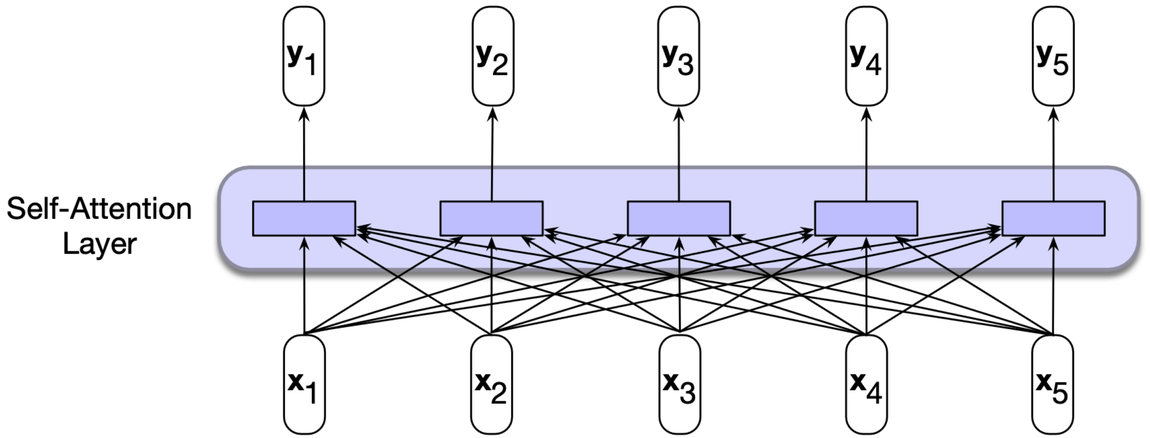


computation for input $\mathbf{x}_1, \ldots, \mathbf{x}_3$ blind to $\mathbf{x}_4$ and $\mathbf{x}_5$

$\mathbf{y}_5$ is embedding for input $\mathbf{x}_1, \ldots, \mathbf{x}_5$

$\mathbf{y}_5$ is a “left-contextual embedding”

Bidirectional



computation for input $\mathbf{x}_1, \ldots, \mathbf{x}_3$ sees $\mathbf{x}_4$ and $\mathbf{x}_5$

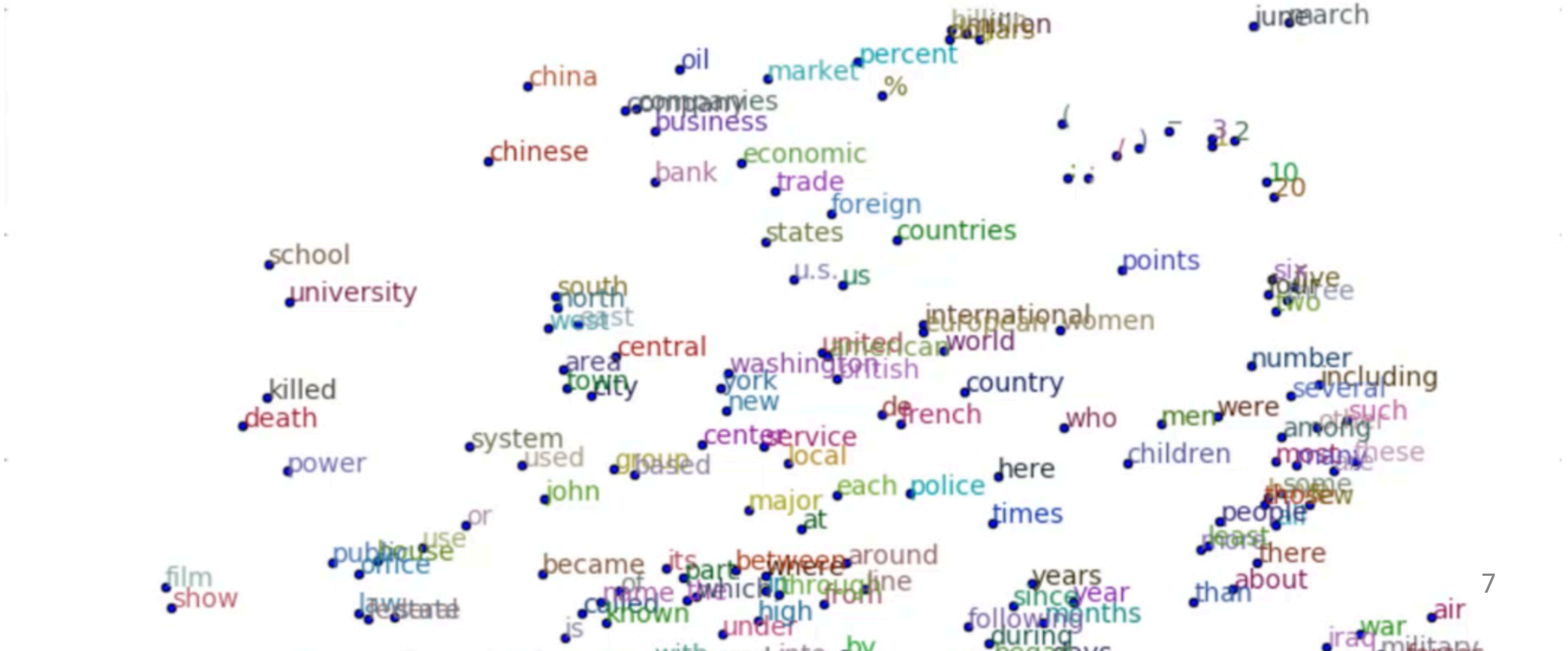
$\mathbf{y}_1, \ldots, \mathbf{y}_5$ is embedding for input $\mathbf{x}_1, \ldots, \mathbf{x}_5$

$\mathbf{y}_i$ are bidirectional “contextual embeddings”

Recap: Embeddings

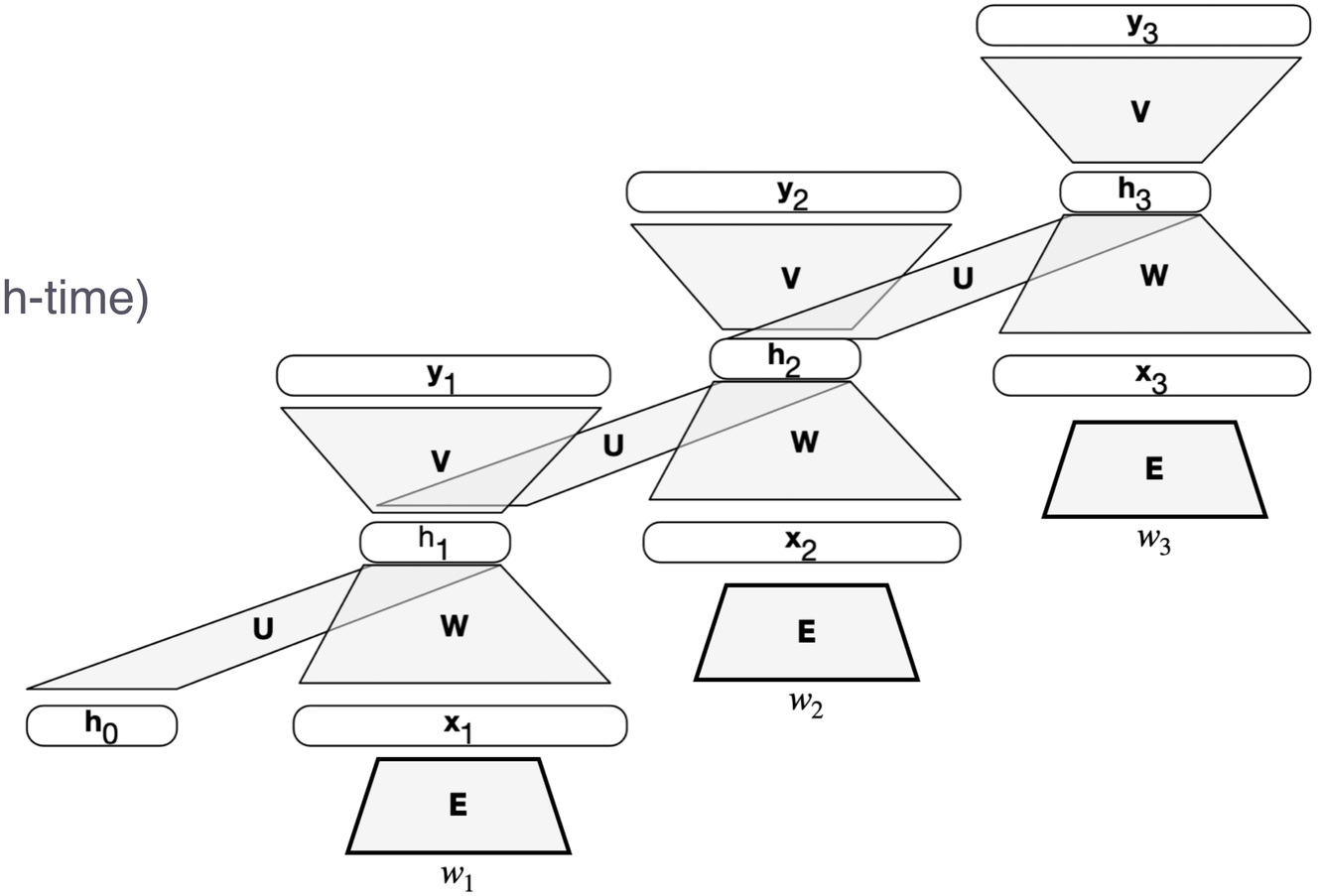
Distributed Representations: dense vectors (aka embeddings) that aim to convey the meaning of tokens:

- "word embeddings" refer to when you have type-based representations
- "contextualized embeddings" refer to when you have token-based representations



Problems with RNNs

- **conceptual**
  - two-fold role of hidden state:
    - memory for past sequence
    - recommend what to do now
- **technical**
  - difficult to train (backpropagation-through-time)
  - vanishing & exploding gradients

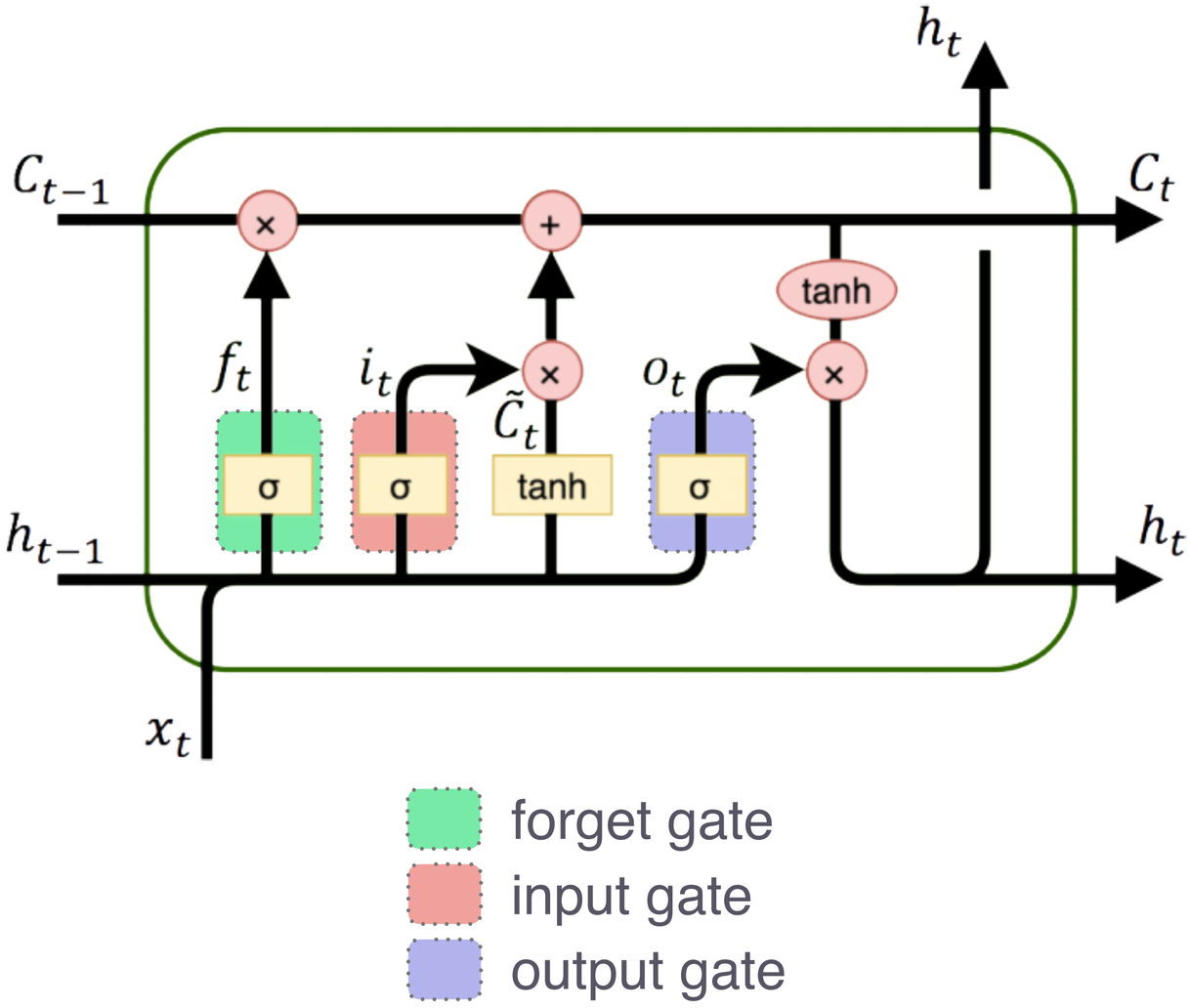


LSTM Motivation

- A type of RNN that is designed to better handle long-range dependencies
- In ”vanilla” RNNs, the hidden state is perpetually being rewritten
- In addition to a traditional hidden state h, let’s have a dedicated memory cell c for long-term events. More power to relay sequence information.

LSTM cell

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$



LSTM Summary

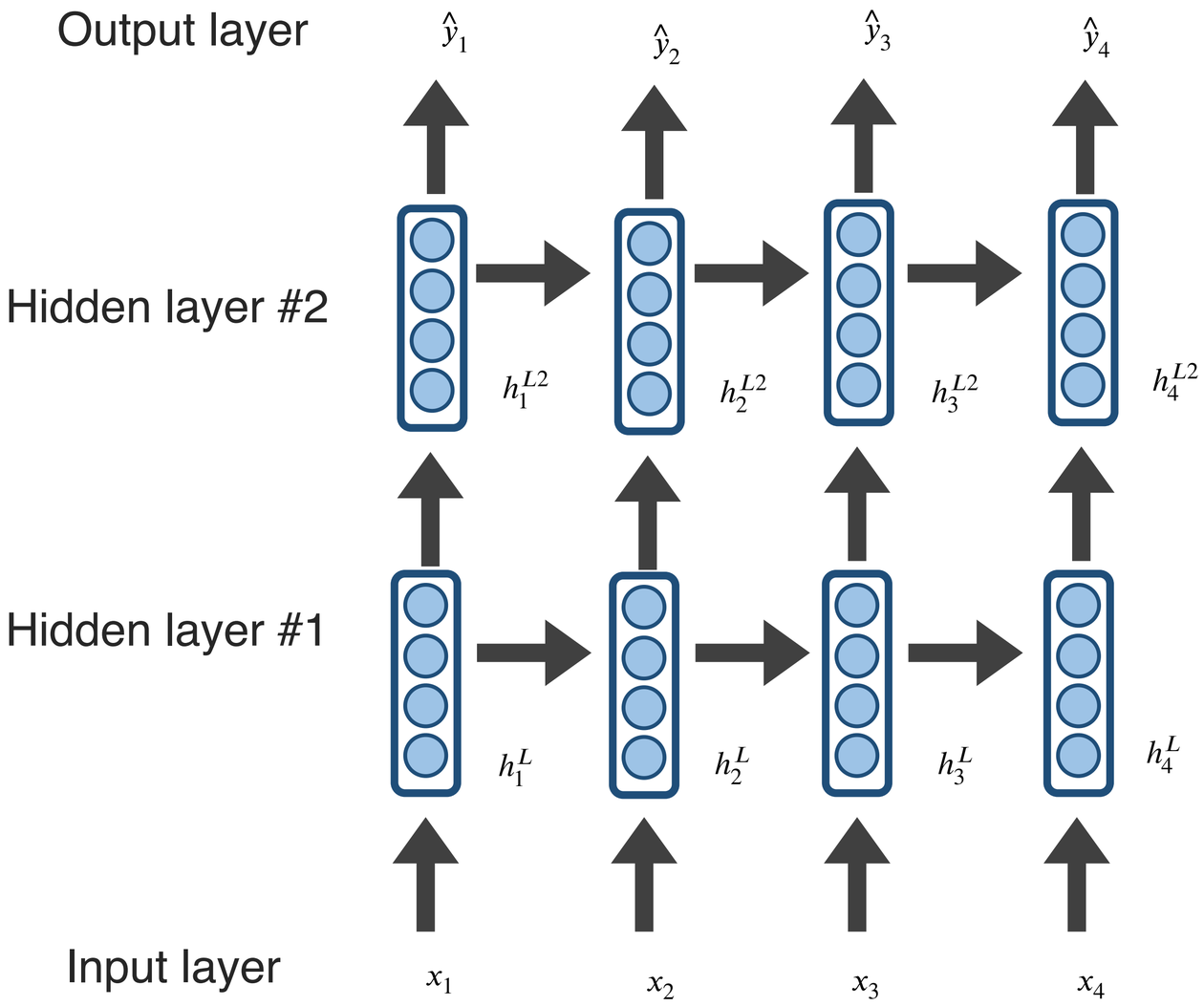
Strengths

- Almost always outperforms RNNs
- Captures long-range dependencies very well

Issues?

- More weights to learn, higher demand on training data
- Can still suffer from vanishing/exploding gradients

Stacked LSTMs



Hidden layers provide an abstraction (holds “meaning”).

Stacking hidden layers provides increased abstractions.

Embedding of “stick” in “Let’s stick to” - Step #2

1- Concatenate hidden layers

2- Multiply each vector by a weight based on the task

x s_2

x s_1

x s_0

3- Sum the (now weighted) vectors

ELMo embedding of “stick” for this task in this context

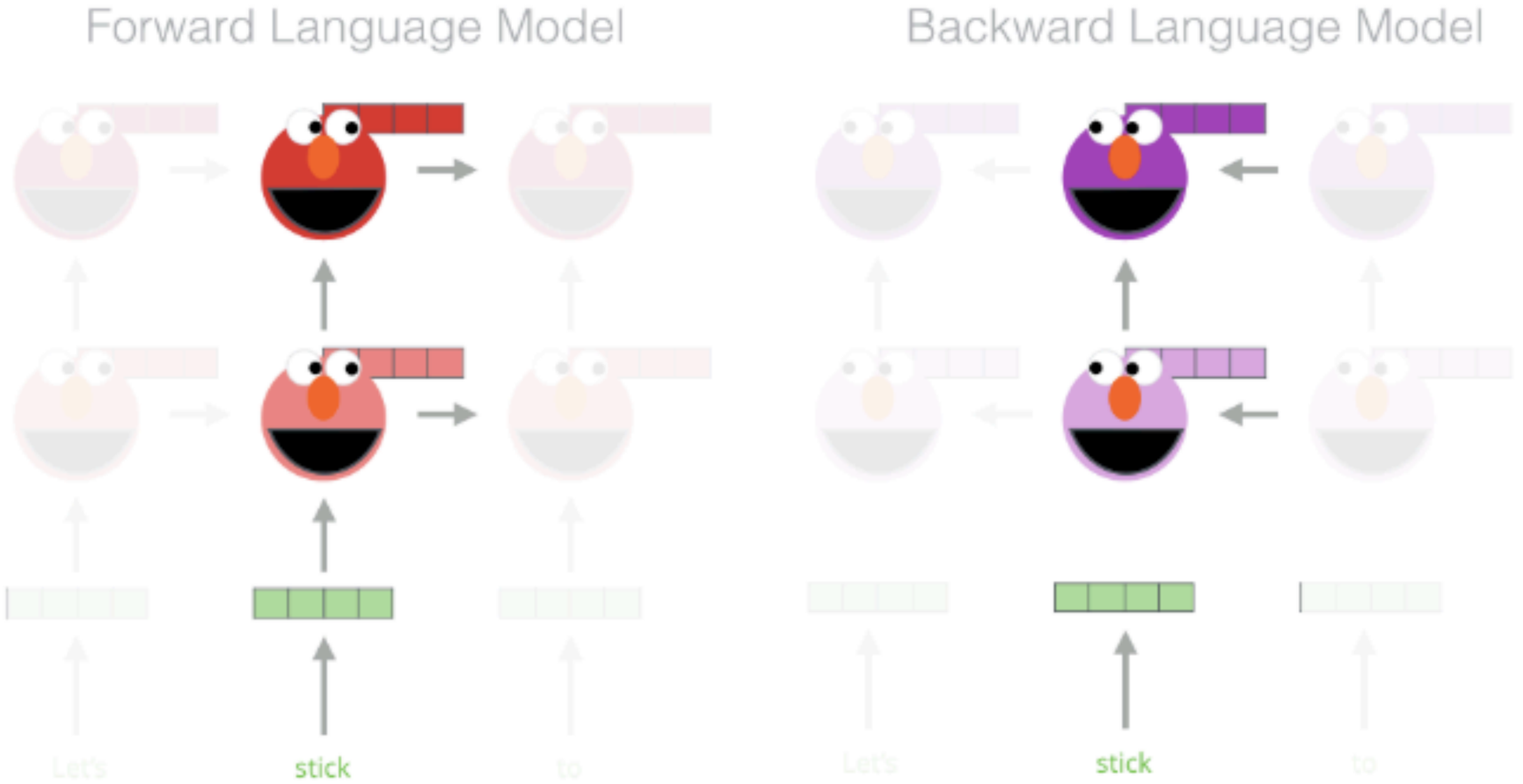


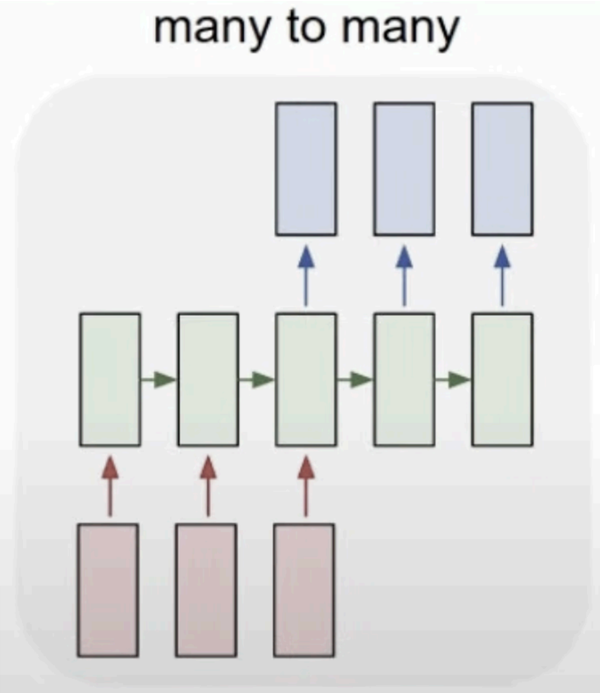
Illustration: http://jalammar.github.io/illustrated-bert/

ELMo

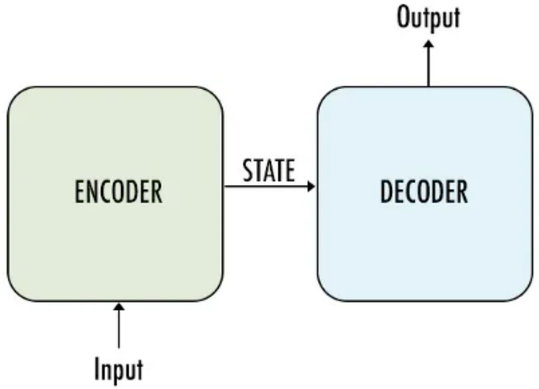
- ELMo yielded very good contextualized embeddings, which re-defined SOTA when applied to many NLP tasks.
- Main ELMo takeaway: given enough training data, having tons of explicit connections between your vectors is useful (the system can determine how to best use context)

ELMo Slides: https://www.slideshare.net/shuntaroy/a-review-of-deep-contextualized-word-representations-peters-2018

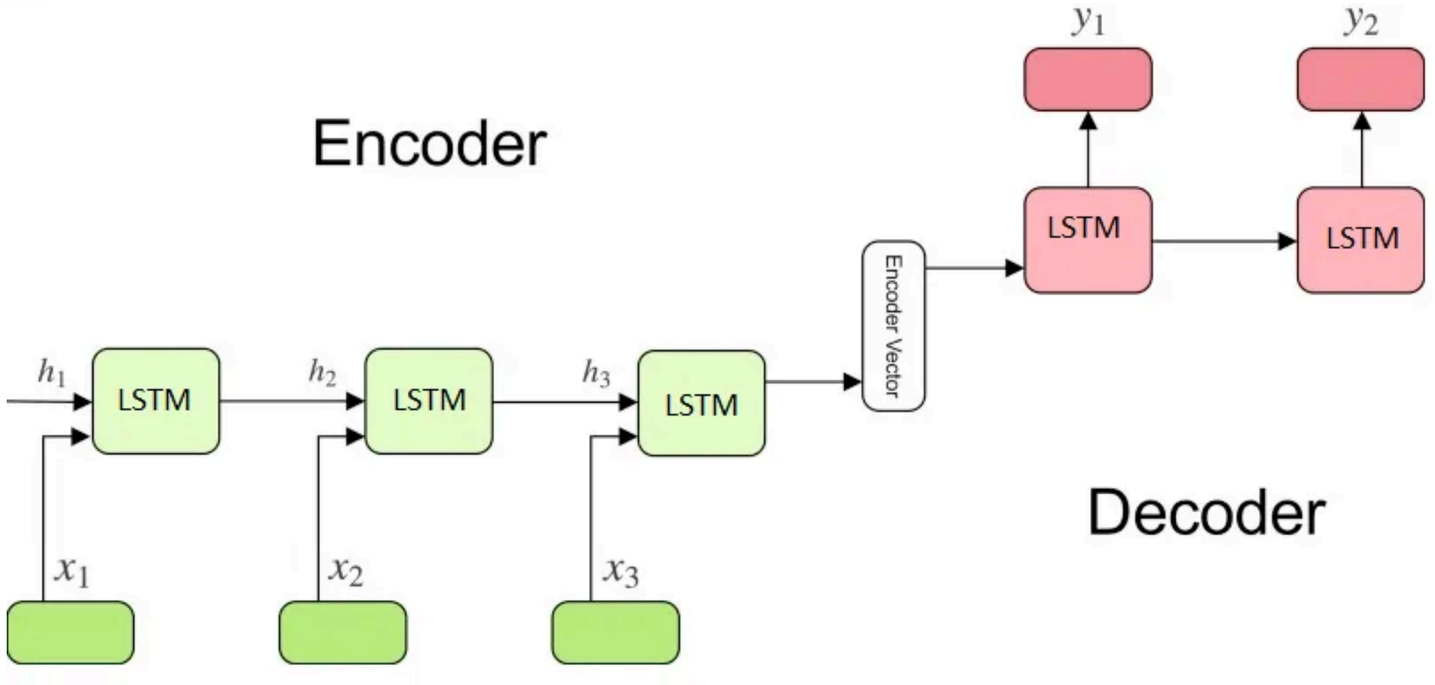
LSTM Machine Translation



Seq2Seq architecture



Bottleneck problem



source

Attention

Attention Motivation

What if the decoder (component that produces outputs), at each step, pays attention to a distribution of all of the encoder's (component that receives inputs) hidden states?

Attention Motivation

What if the decoder (component that produces outputs), at each step, pays attention to a distribution of all of the encoder's (component that receives inputs) hidden states?

Intuition: when we (humans) translate a sentence, we don't just consume the original sentence, reflect on the meaning of the last word, then regurgitate in a new language; we continuously think back at the original sentence while focusing on different parts.

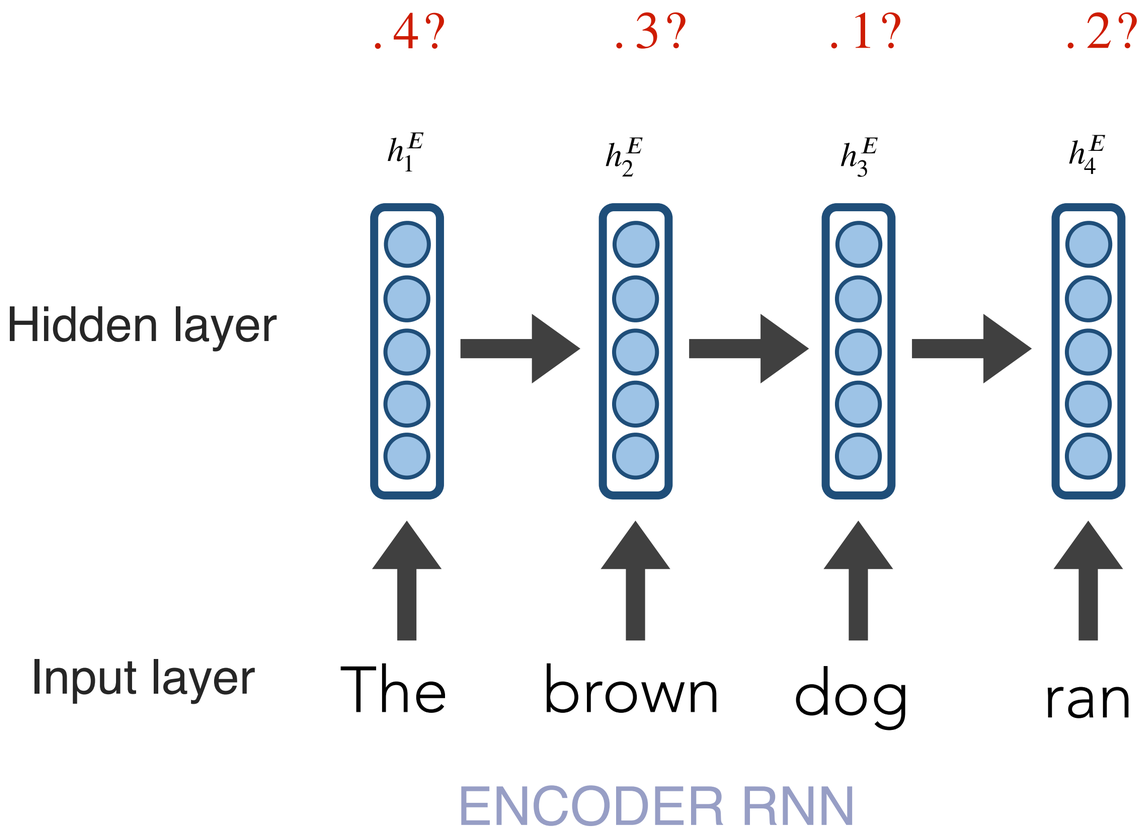
Attention Motivation

The concept of attention within cognitive neuroscience and psychology dates back to the 1800s. [William James, 1890].

Nadaray-Watson kernel regression proposed in 1964. It locally weighted its predictions.

Attention Architecture

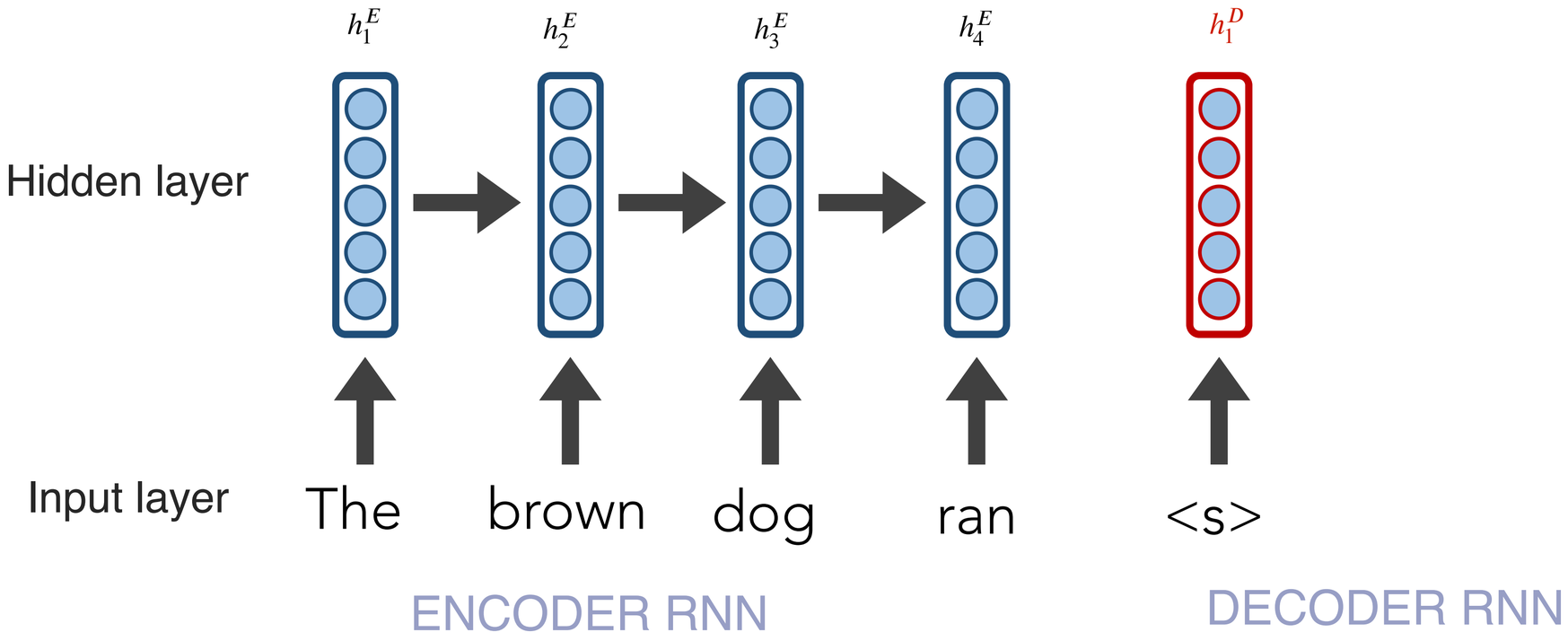
Q: How do we determine how much to pay attention to each of the encoder's hidden layers?



Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

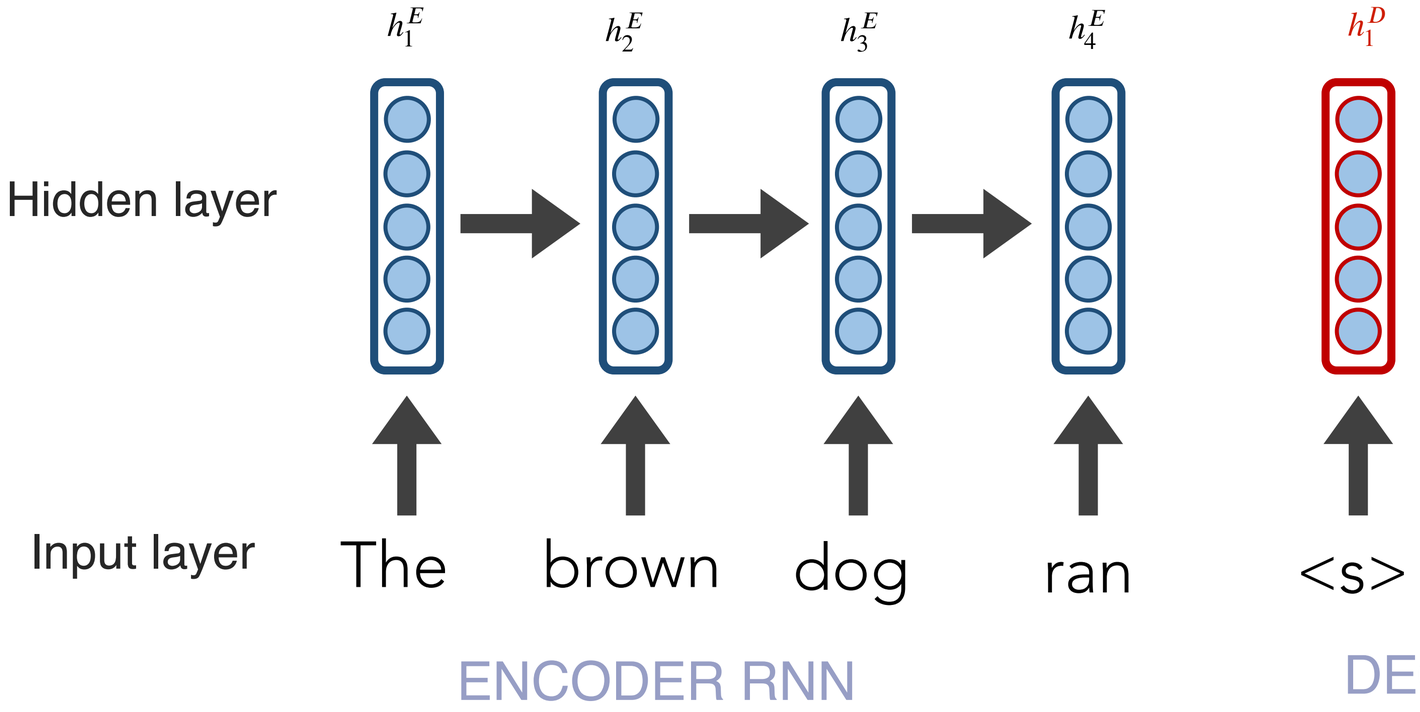
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!

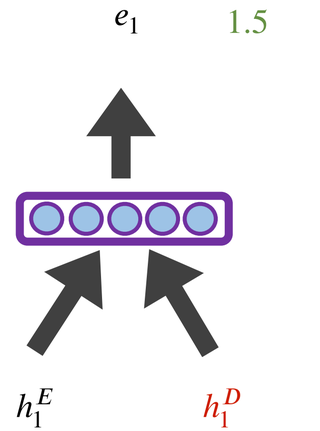


Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



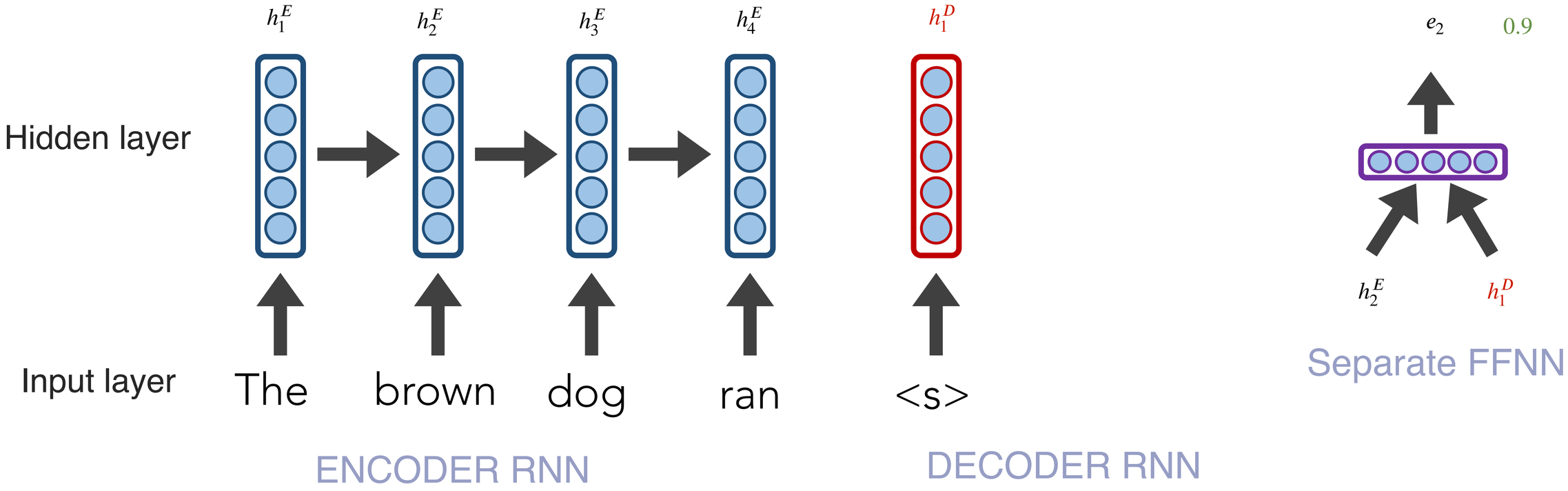


Separate FFNN

Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

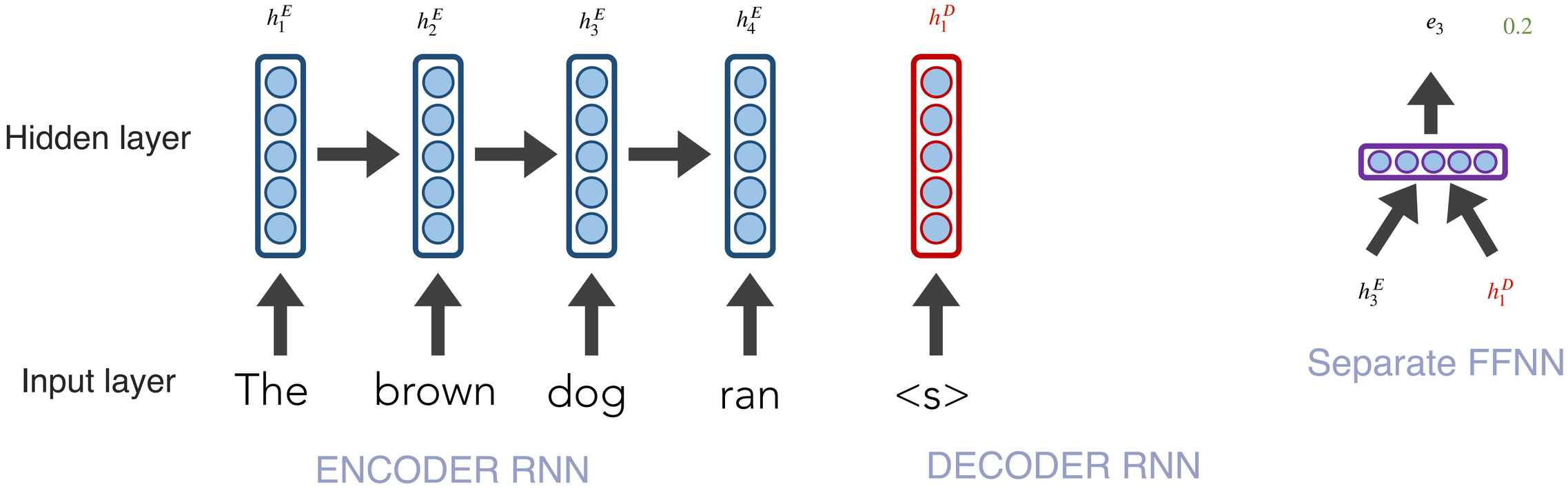
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

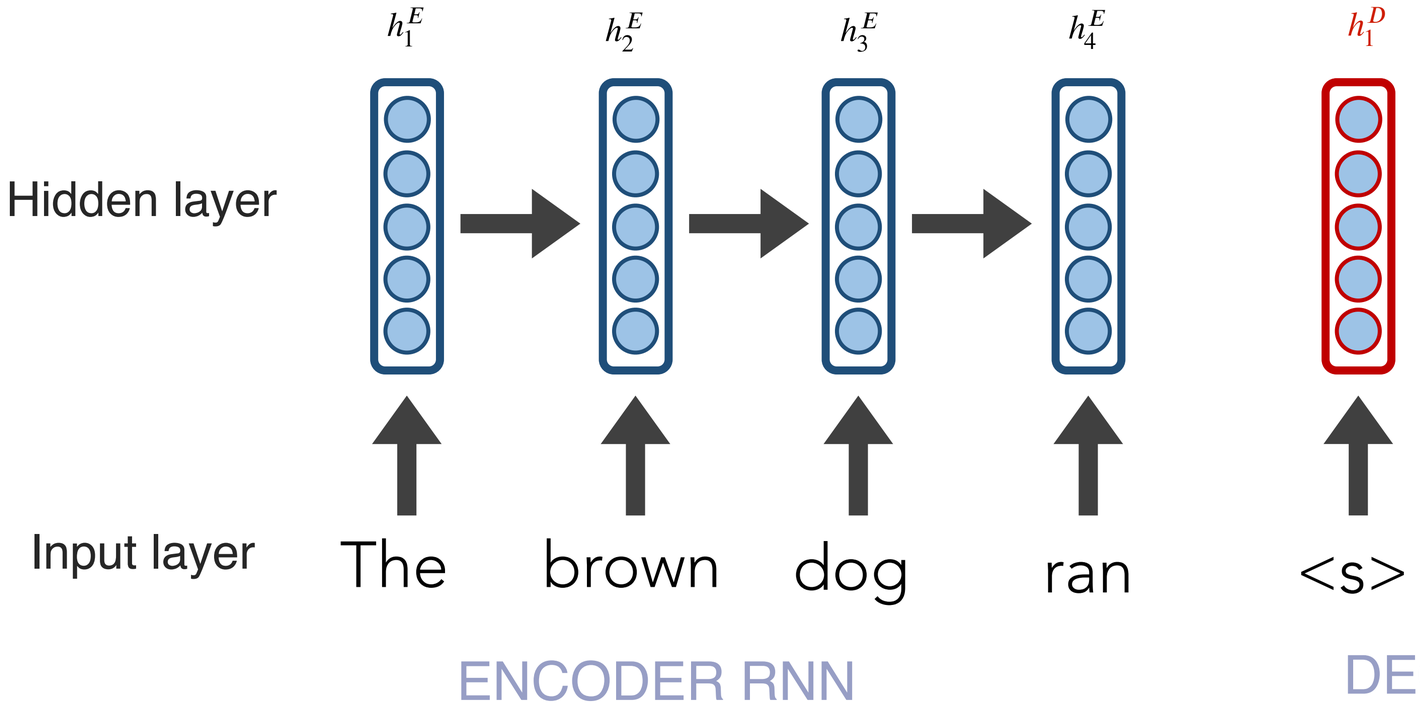
A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!

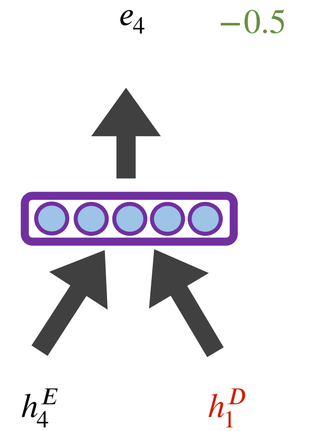


Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



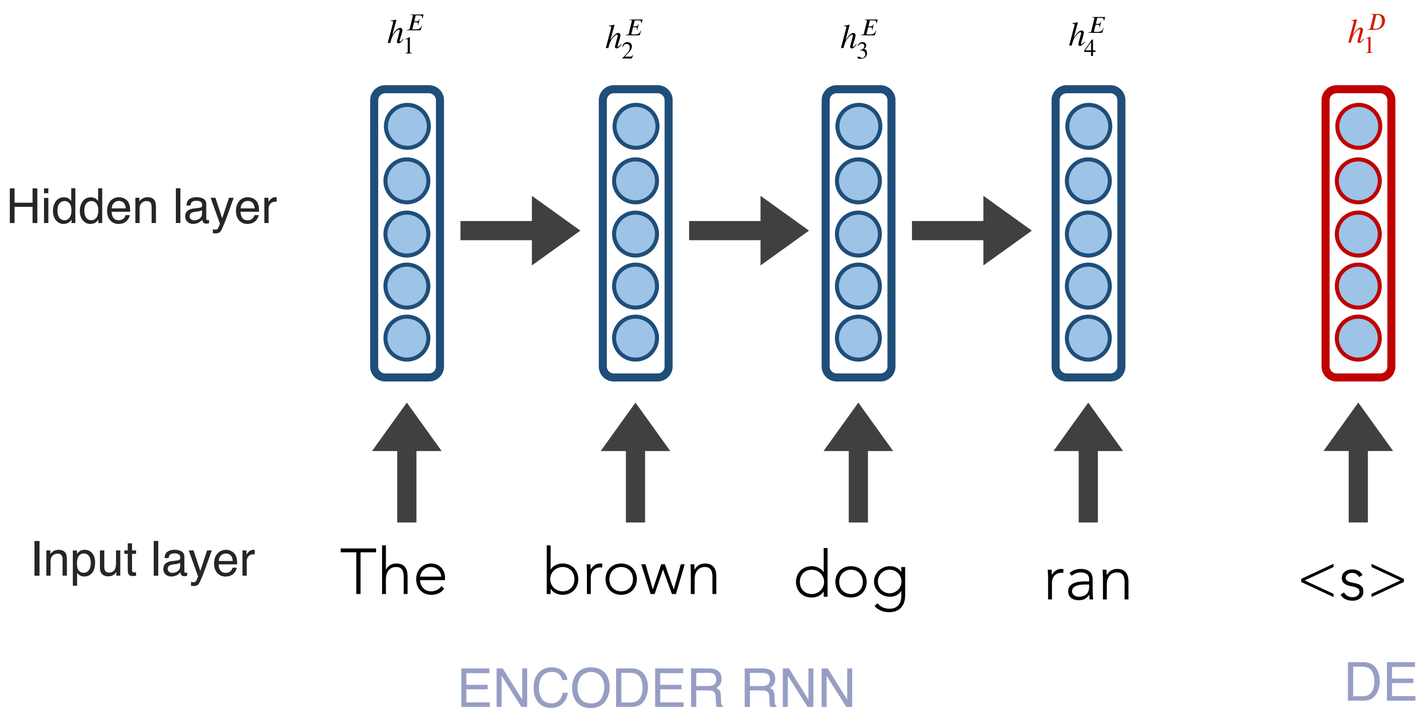


Separate FFNN

Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



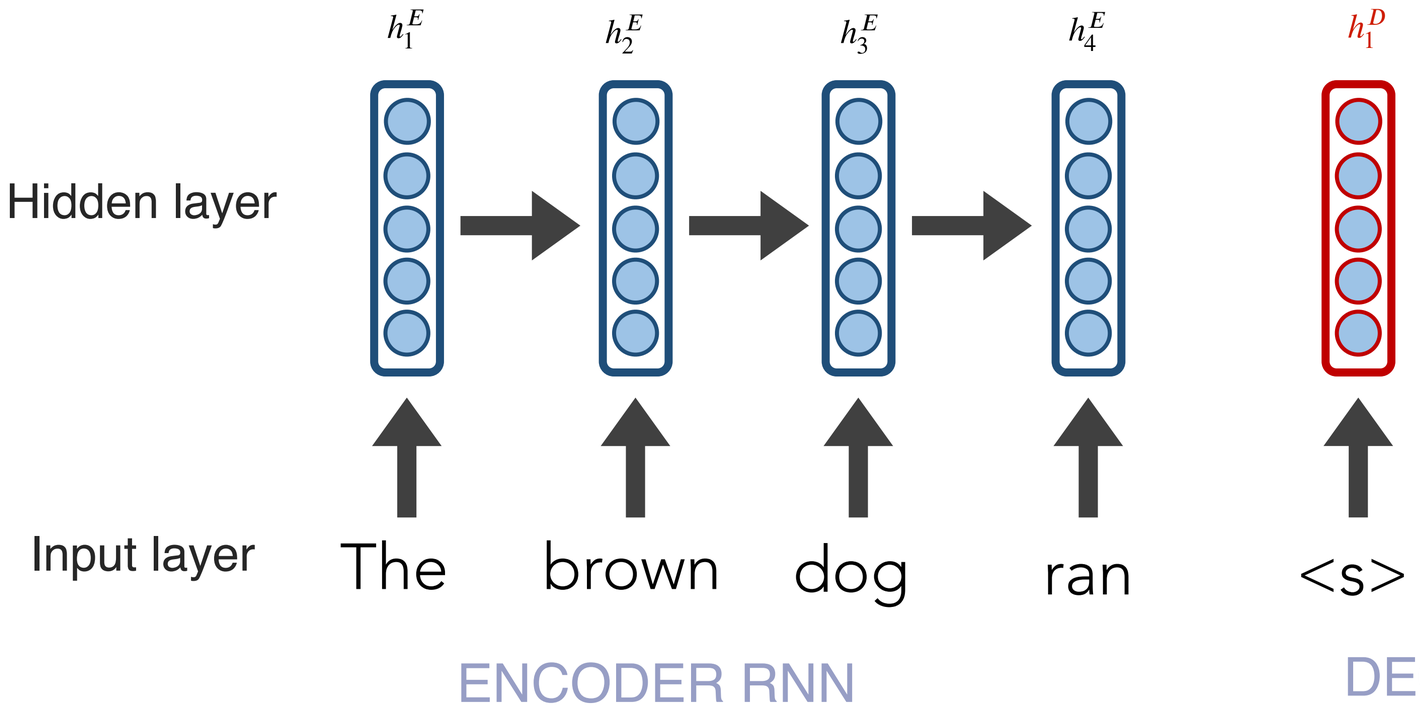
Attention (raw scores)

e_1	1.5
e_2	0.9
e_3	0.2
e_4	−0.5

Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



Attention (raw scores)

e_1	1.5
e_2	0.9
e_3	0.2
e_4	−0.5

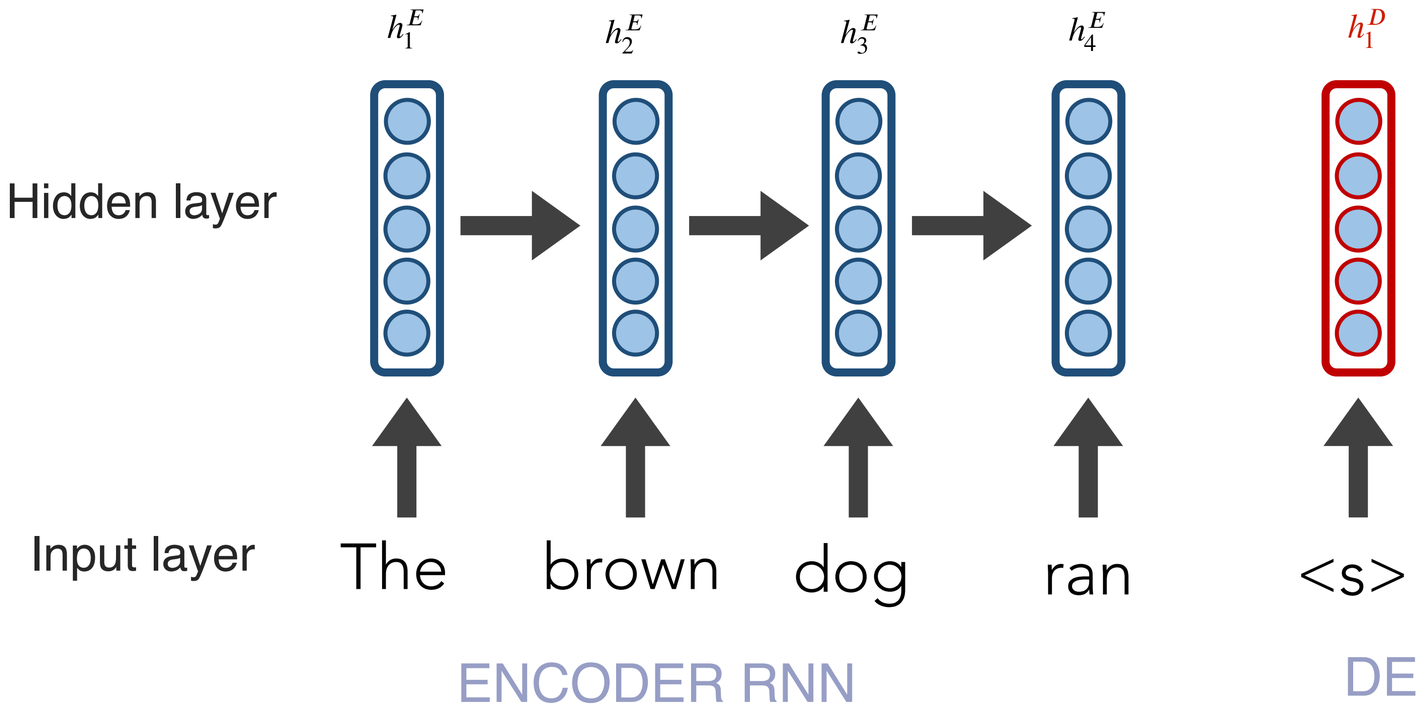
Attention (softmax'd)

$$a_i^1 = \frac{\exp(e_i)}{\sum_i^N \exp(e_1)}$$

Attention Architecture

Q: How do we determine how much to pay attention to each of the encoder's hidden layers?

A: Let's base it on our decoder's current hidden state (our current representation of meaning) and all of the encoder's hidden layers!



Attention (raw scores)

e_1 1.5

e_2 0.9

e_3 0.2

e_4 −0.5

Attention (softmax'd)

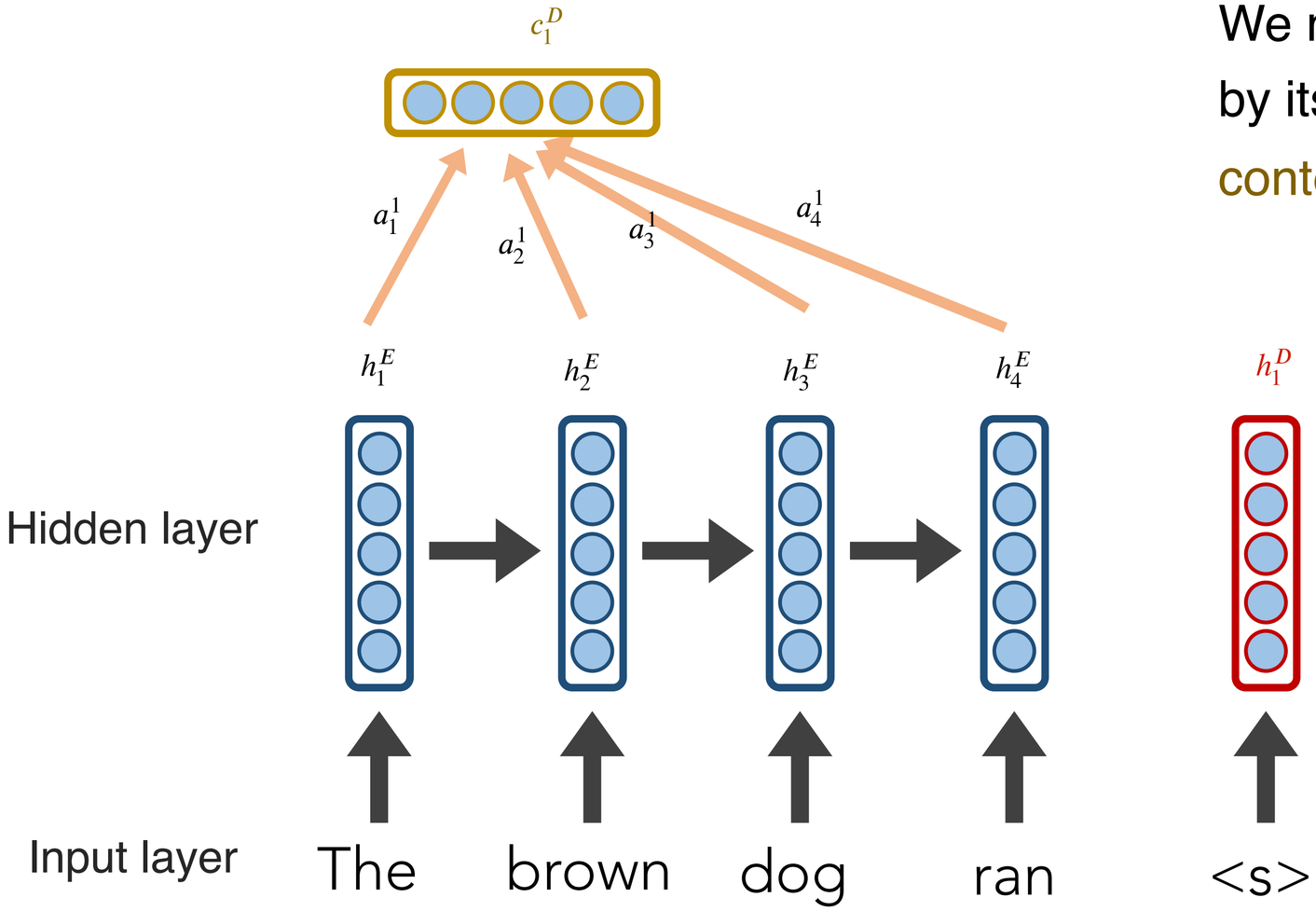
$a_1^1 = 0.51$

$a_2^1 = 0.28$

$a_3^1 = 0.14$

$a_3^1 = 0.07$

Attention Architecture



We multiply each encoder's hidden layer by its a_i^1 attention weights to create a context vector c_1^D

Attention (softmax'd)

$a_1^1 = 0.51$

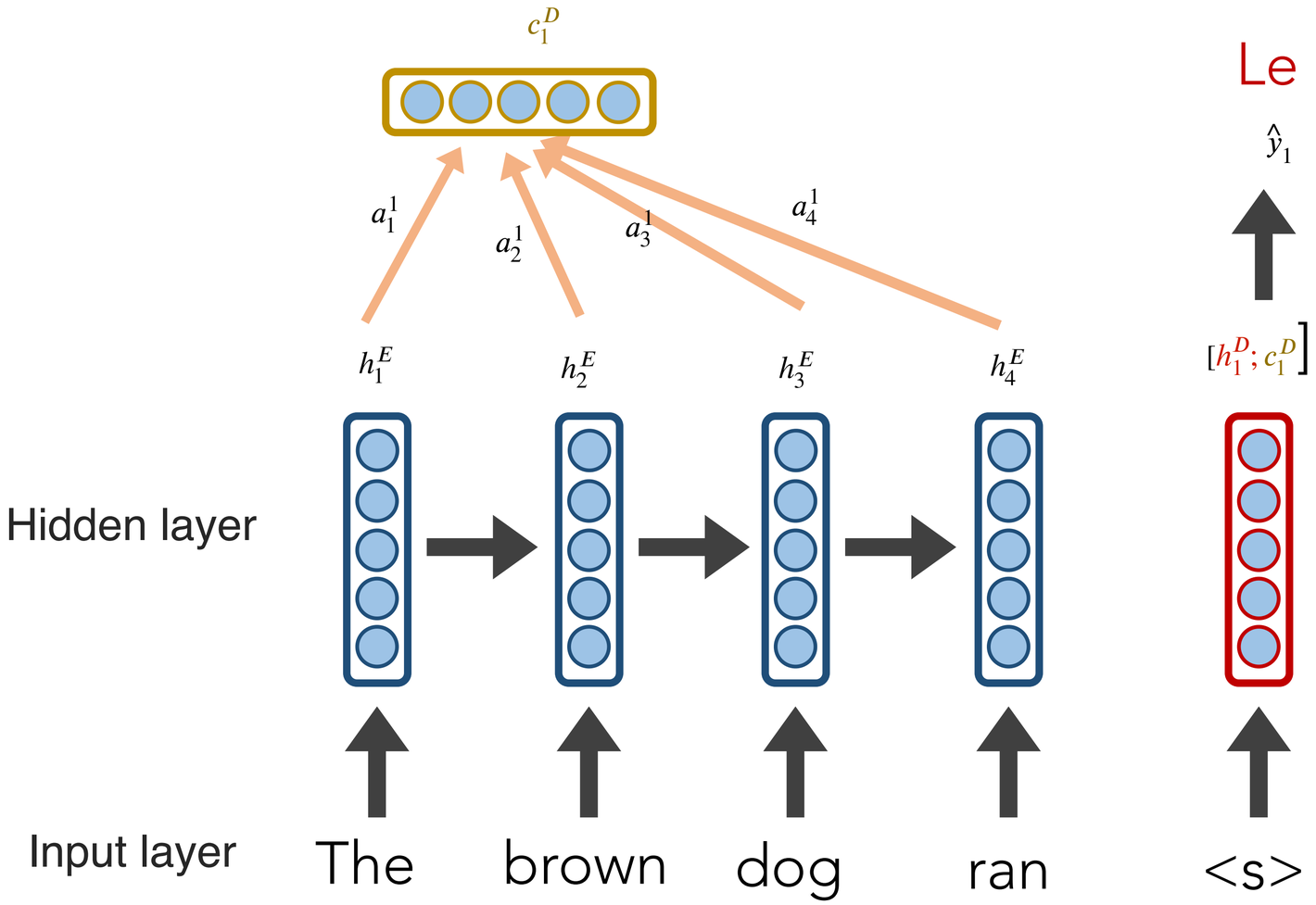
$a_2^1 = 0.28$

$a_3^1 = 0.14$

$a_3^1 = 0.07$

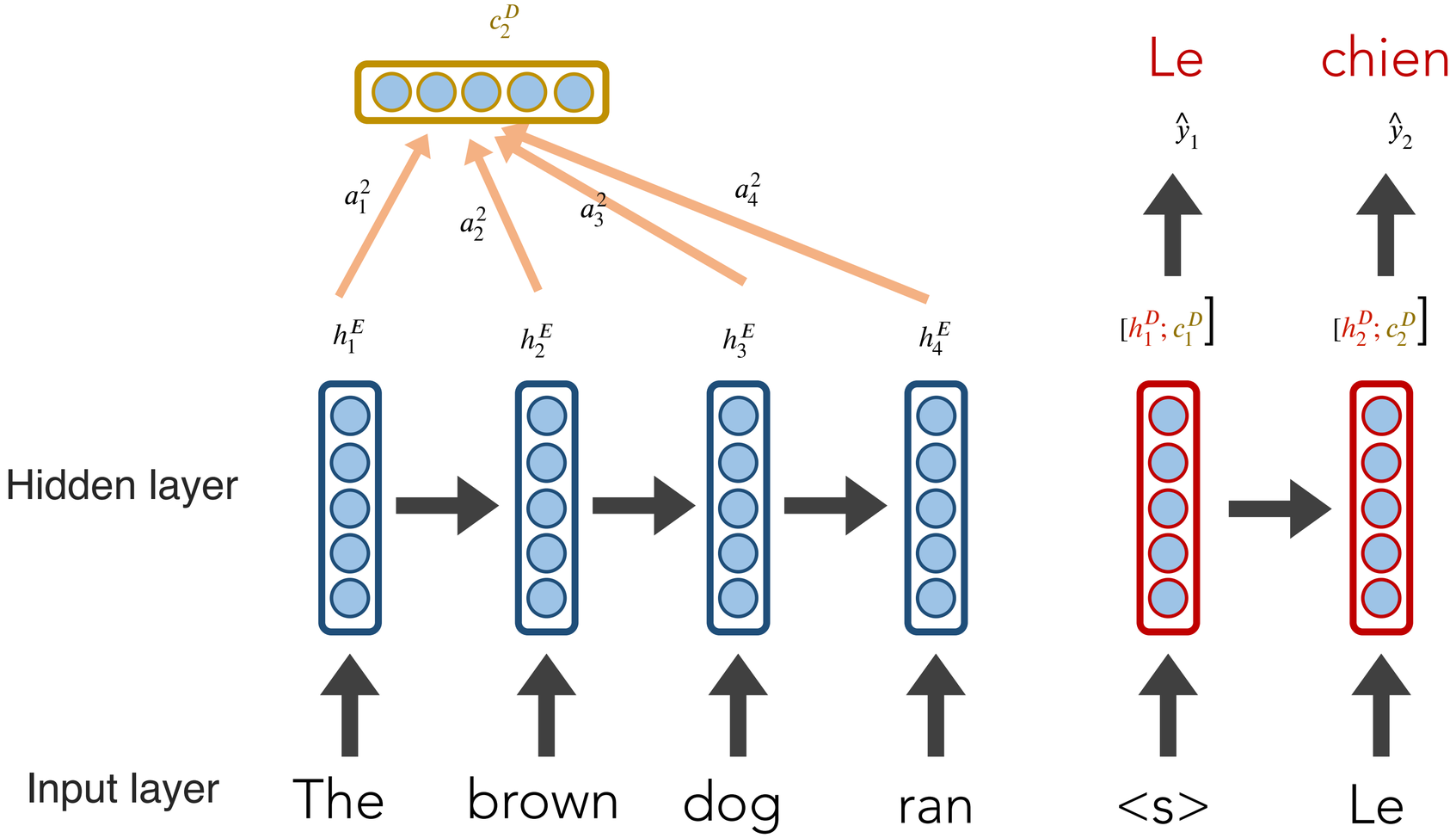
Attention Architecture

REMEMBER: each attention weight a_i^j is based on the decoder's current hidden state, too.



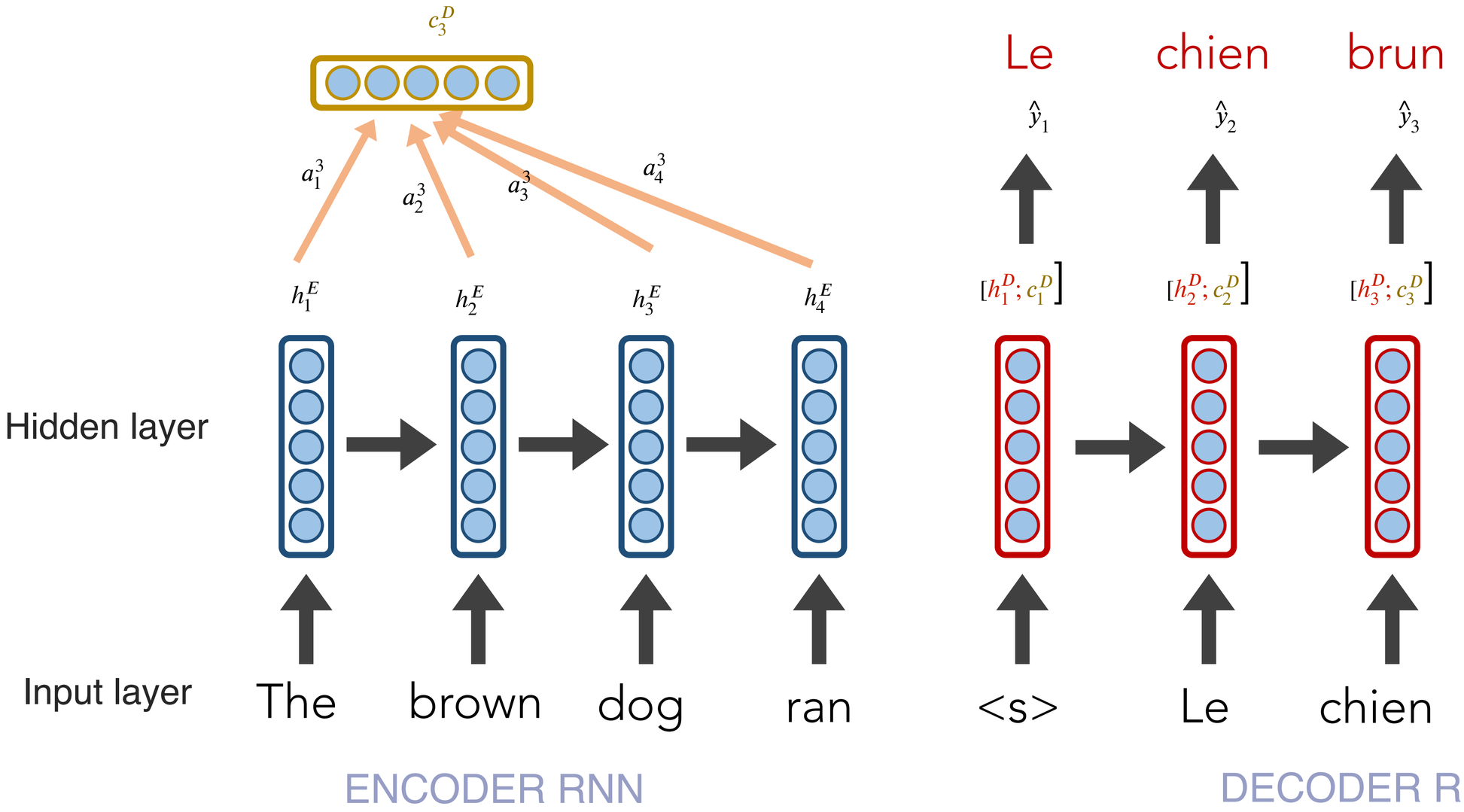
Attention Architecture

REMEMBER: each attention weight a_i^j is based on the decoder's current hidden state, too.



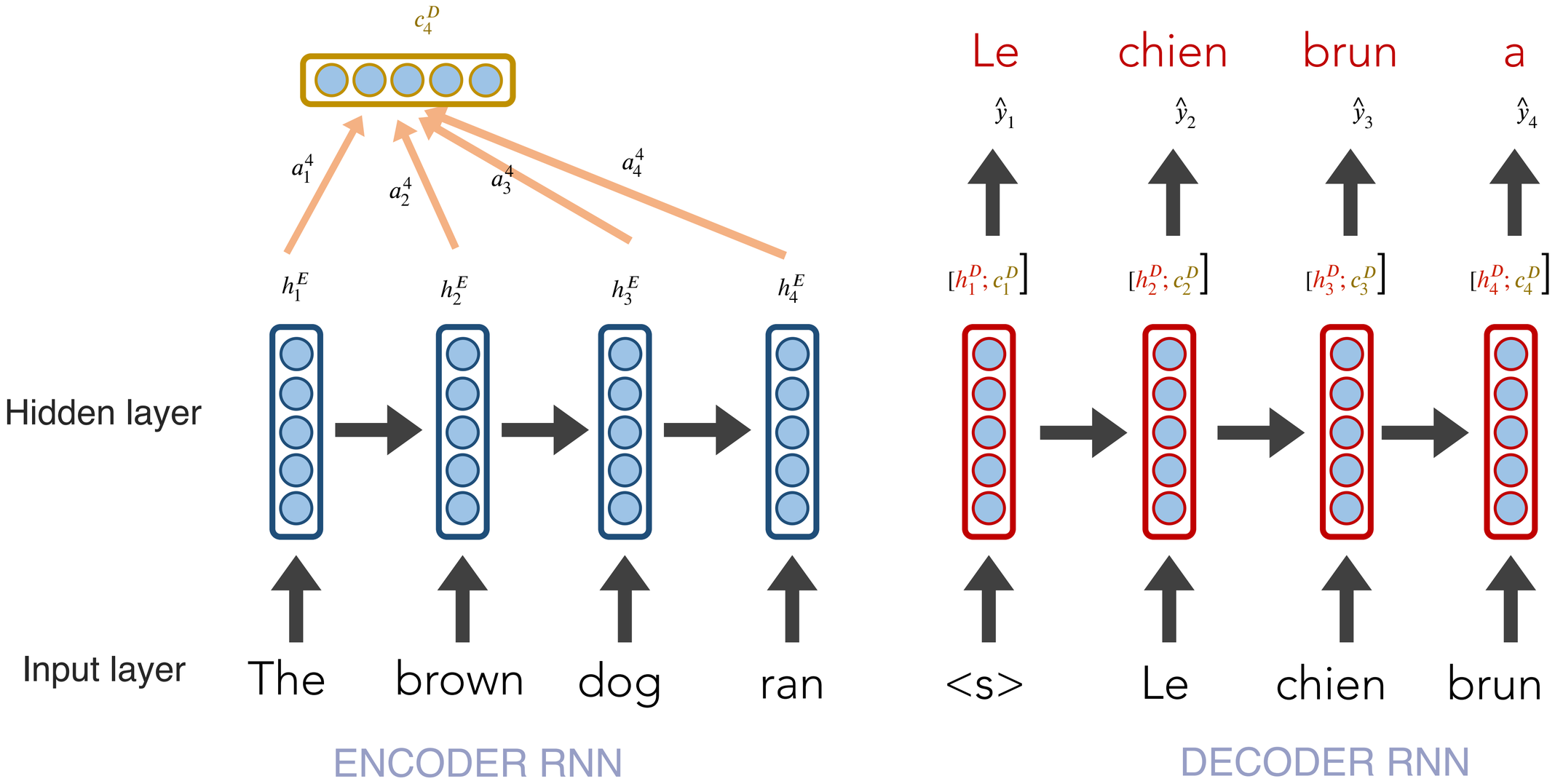
Attention Architecture

REMEMBER: each attention weight a_i^j is based on the decoder's current hidden state, too.



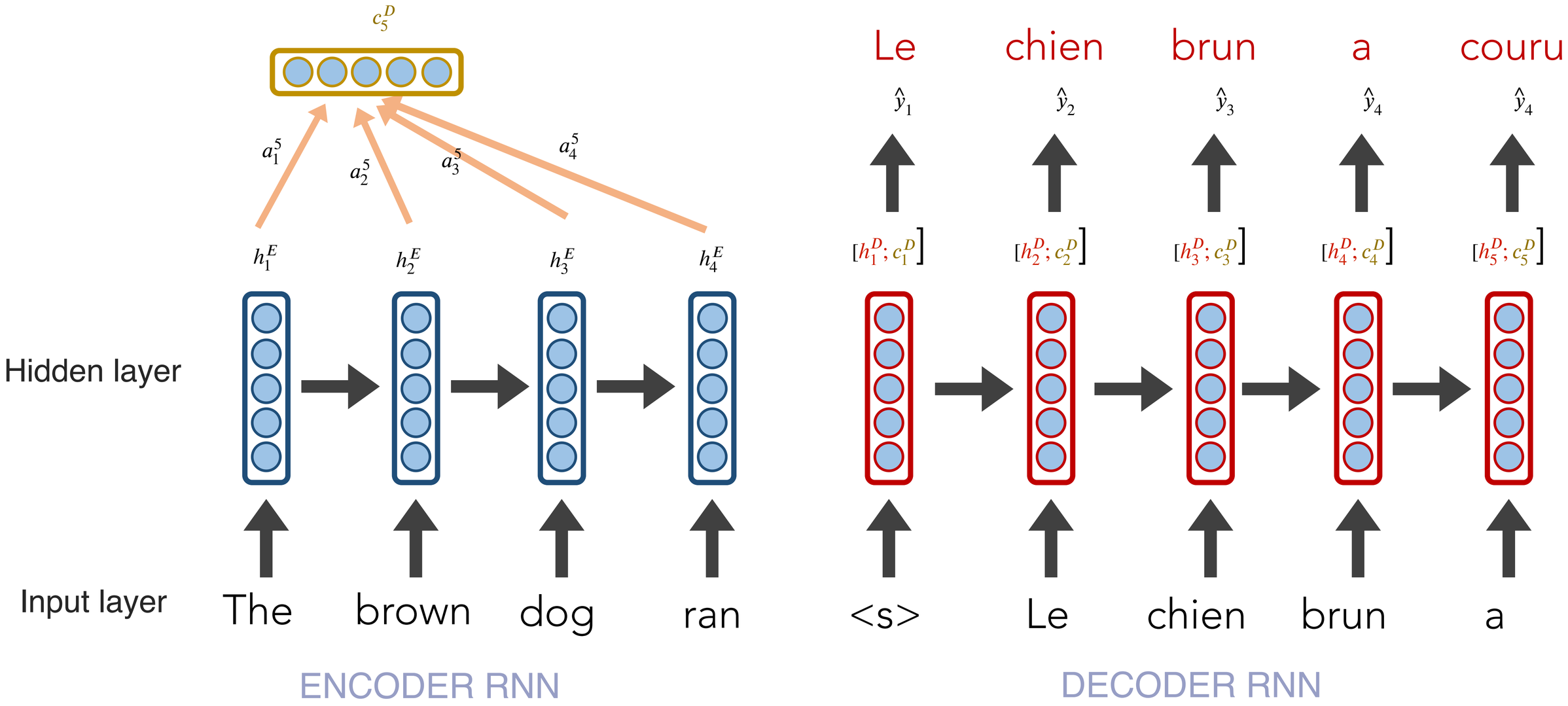
Attention Architecture

REMEMBER: each attention weight a_i^j is based on the decoder's current hidden state, too.



Attention Architecture

REMEMBER: each attention weight a_i^j is based on the decoder's current hidden state, too.



Attention Summary

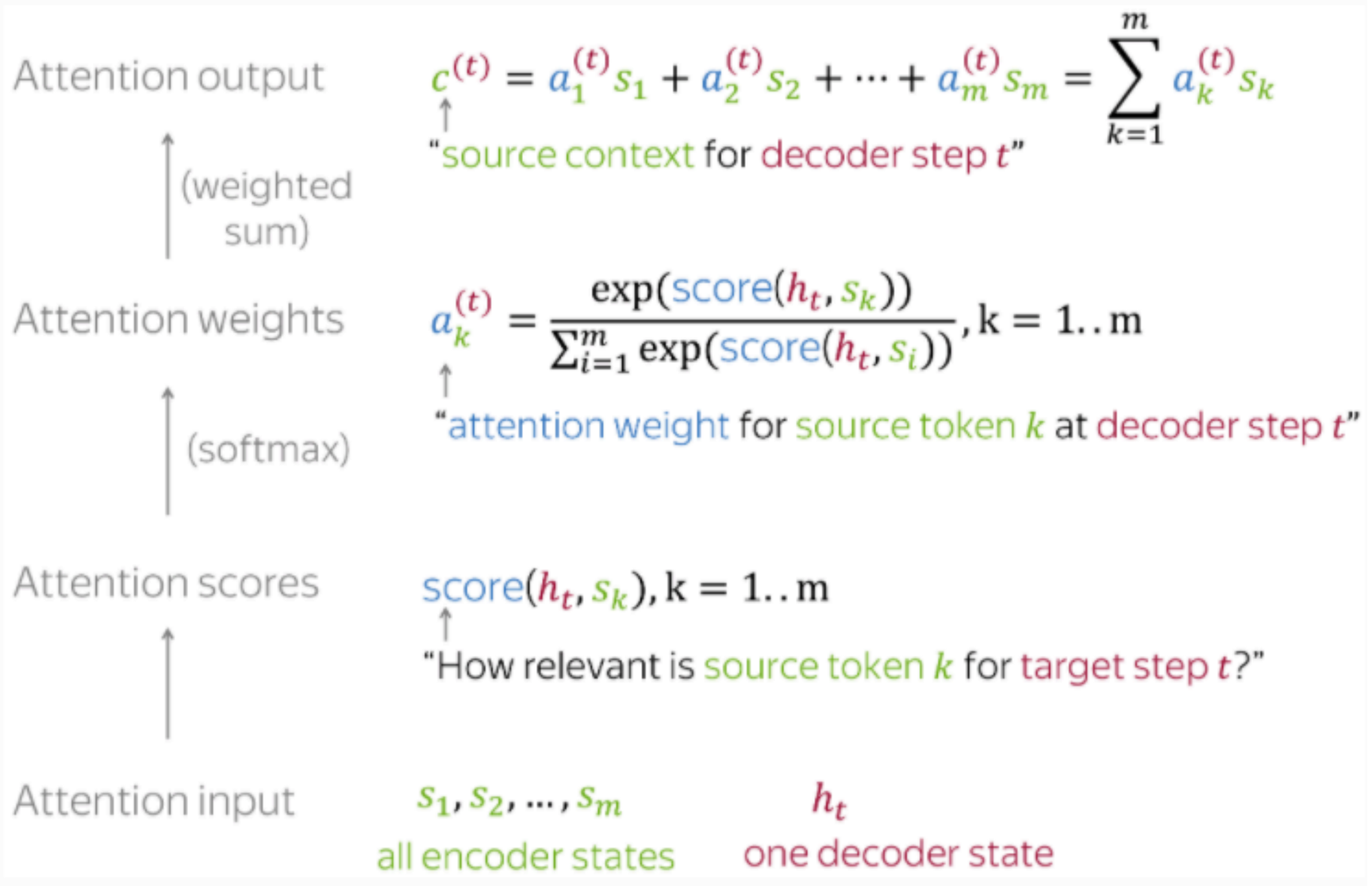


Photo credit: https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Attention Summary

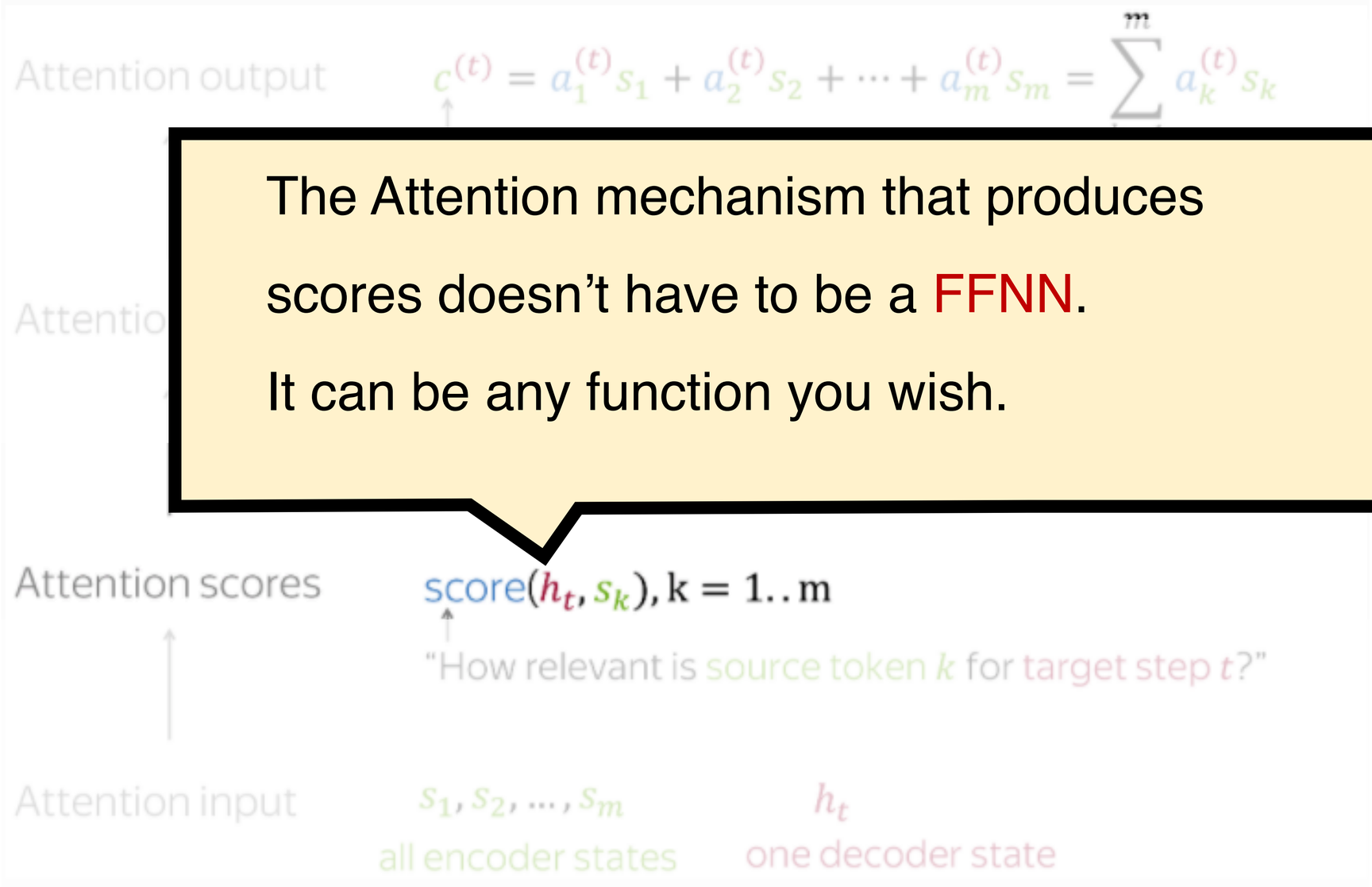
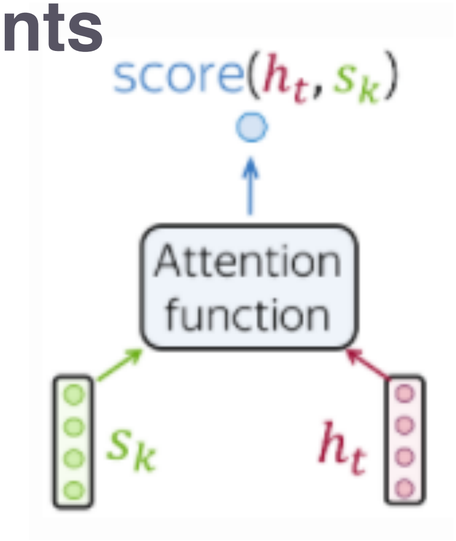


Photo credit: https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Attention Variants



Dot-product

$$h_t^T \times s_k$$

$$\text{score}(h_t, s_k) = h_t^T \, s_k$$

Bilinear

$$h_t^T \times [W] \times s_k$$

$$\text{score}(h_t, s_k) = h_t^T \, W s_k$$

Multi-Layer Perceptron

$$w_2^T \times \tanh\left[W_1 \times \begin{matrix} h_t \\ s_k \end{matrix}\right]$$

$$\text{score}(h_t, s_k) = w_2^T \cdot \tanh(W_1 [h_t, s_k])$$

Photo credit: https://lena-voita.github.io/nlp_course/seq2seq_and_attention.html

Attention Summary

Attention:

- greatly improves seq2seq results
- allows us to visualize the contribution each encoding word gave for each decoder's word

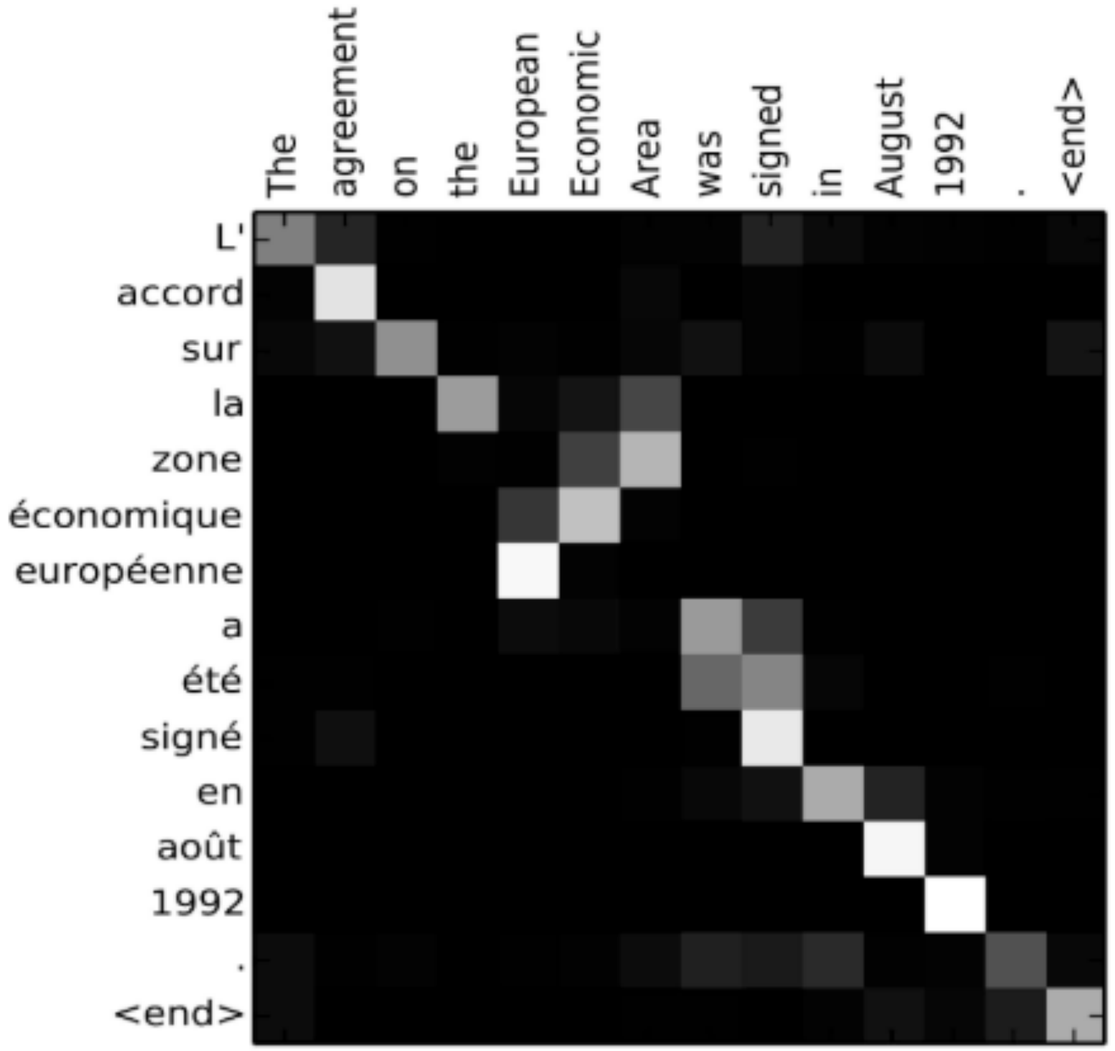
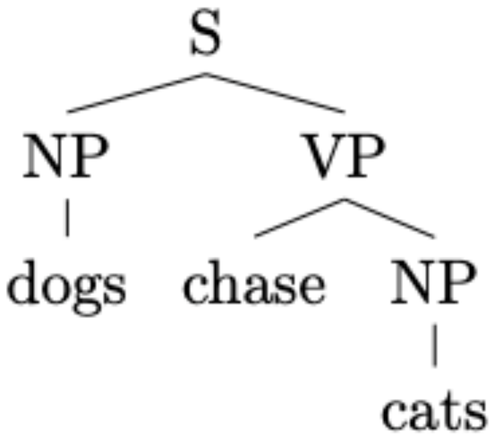


Image source: Fig 3 in [Bahdanau et al., 2015]

Attention Applications

Input: dogs chase cats

Output:



or a flattened representation

$$(\mathrm{S}\ (\mathrm{NP}\ \mathrm{dogs}\ )_{\mathrm{NP}}\ (\mathrm{VP}\ \mathrm{chase}\ (\mathrm{NP}\ \mathrm{cats}\ )_{\mathrm{NP}}\ )_{\mathrm{VP}}\ )_{\mathrm{S}}$$

Attention Applications

Input: I shot an elephant in my pajamas

Output:

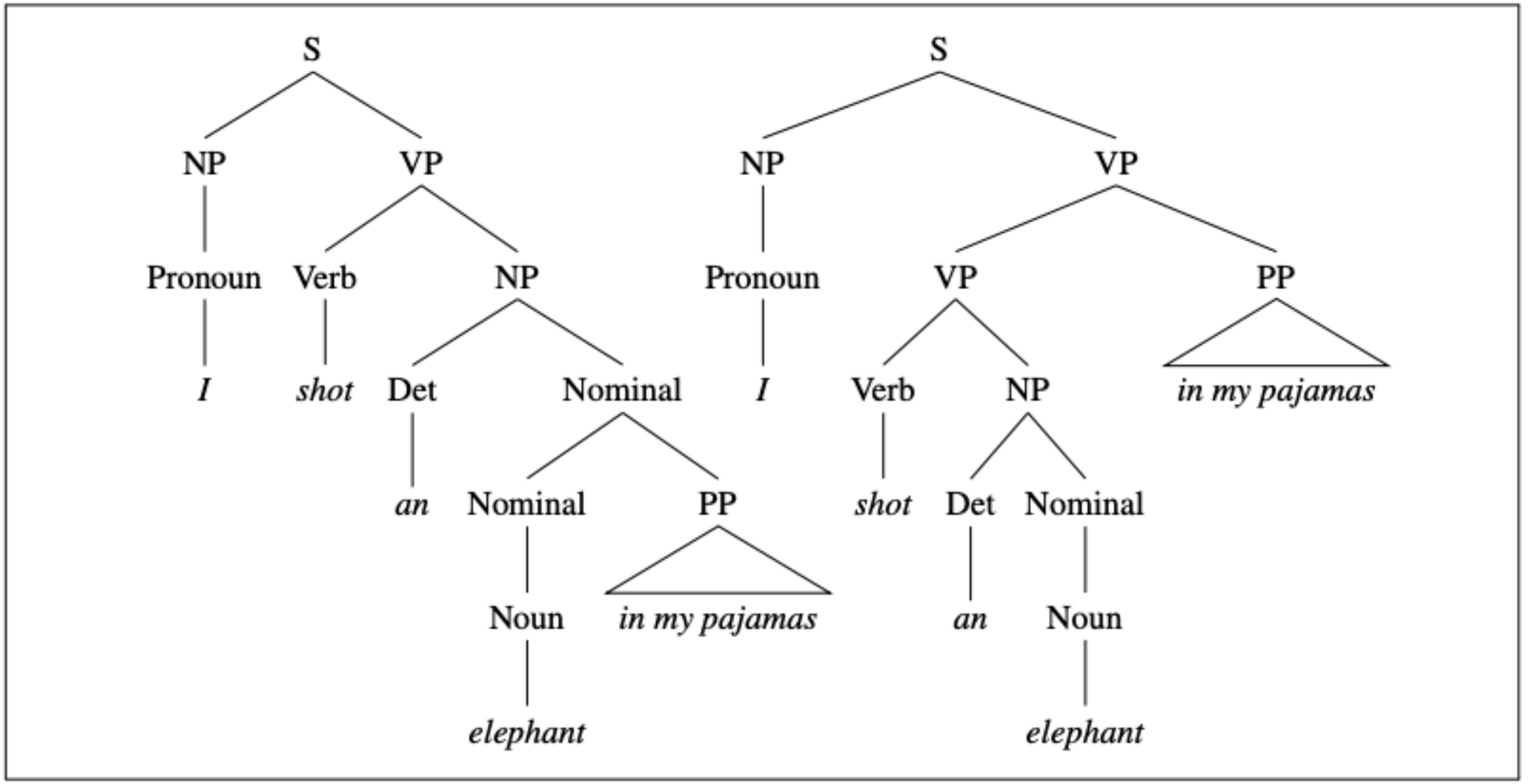


Figure 13.2 Two parse trees for an ambiguous sentence. The parse on the left corresponds to the humorous reading in which the elephant is in the pajamas, the parse on the right corresponds to the reading in which Captain Spaulding did the shooting in his pajamas.

https://web.stanford.edu/~jurafsky/slp3/13.pdf

Attention Applications

Model	English			Chinese		
	LR	LP	F1	LR	LP	F1
Shen et al. (2018)	92.0	91.7	91.8	86.6	86.4	86.5
Fried and Klein (2018)	-	-	92.2	-	-	87.0
Teng and Zhang (2018)	92.2	92.5	92.4	86.6	88.0	87.3
Vaswani et al. (2017)	-	-	92.7	-	-	-
Dyer et al. (2016)	-	-	93.3	-	-	84.6
Kuncoro et al. (2017)	-	-	93.6	-	-	-
Charniak et al. (2016)	-	-	93.8	-	-	-
Liu and Zhang (2017b)	91.3	92.1	91.7	85.9	85.2	85.5
Liu and Zhang (2017a)	-	-	94.2	-	-	86.1
Suzuki et al. (2018)	-	-	94.32	-	-	-
Takase et al. (2018)	-	-	94.47	-	-	-
Fried et al. (2017)	-	-	94.66	-	-	-
Kitaev and Klein (2018)	94.85	95.40	95.13	-	-	-
Kitaev et al. (2018)	95.51	96.03	95.77	91.55	91.96	91.75
Zhou and Zhao (2019) (BERT)	95.70	95.98	95.84	**92.03**	92.33	92.18
Zhou and Zhao (2019) (XLNet)	96.21	96.46	96.33	-	-	-
Our work	**96.24**	**96.53**	**96.38**	91.85	**93.45**	**92.64**

Table 3: Constituency Parsing on PTB & CTB test sets.

https://aclanthology.org/2020.findings-emnlp.65.pdf

Visualizing Attention

Input: image

Output: generated text

A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. Xu et al. CVPR (2016)

Visualizing Attention

Input: image

Output: generated text

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

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

Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. Xu et al. CVPR (2016)

Visualizing Attention

A large white bird standing in a forest.

A woman holding a clock in her hand.

Figure 5. Examples of mistakes where we can use attention to gain intuition into what the model saw.

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. Xu et al. CVPR (2016)

Visualizing Attention

A woman is sitting at a table with a large pizza.

A person is standing on a beach with a surfboard.

Figure 5. Examples of mistakes where we can use attention to gain intuition into what the model saw.

Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. Xu et al. CVPR (2016)

Self Attention

Self Attention Motivation

- Word meaning is contextual!
  - *The shirt is green*
  - *The recruits are green*
  - *Green policy has wrecked the economy*
- But word (type) embeddings conflate all of these meanings
- Every occurrence (token) of “green” carries the same semantic mix
- Instead: embed individual term occurrences

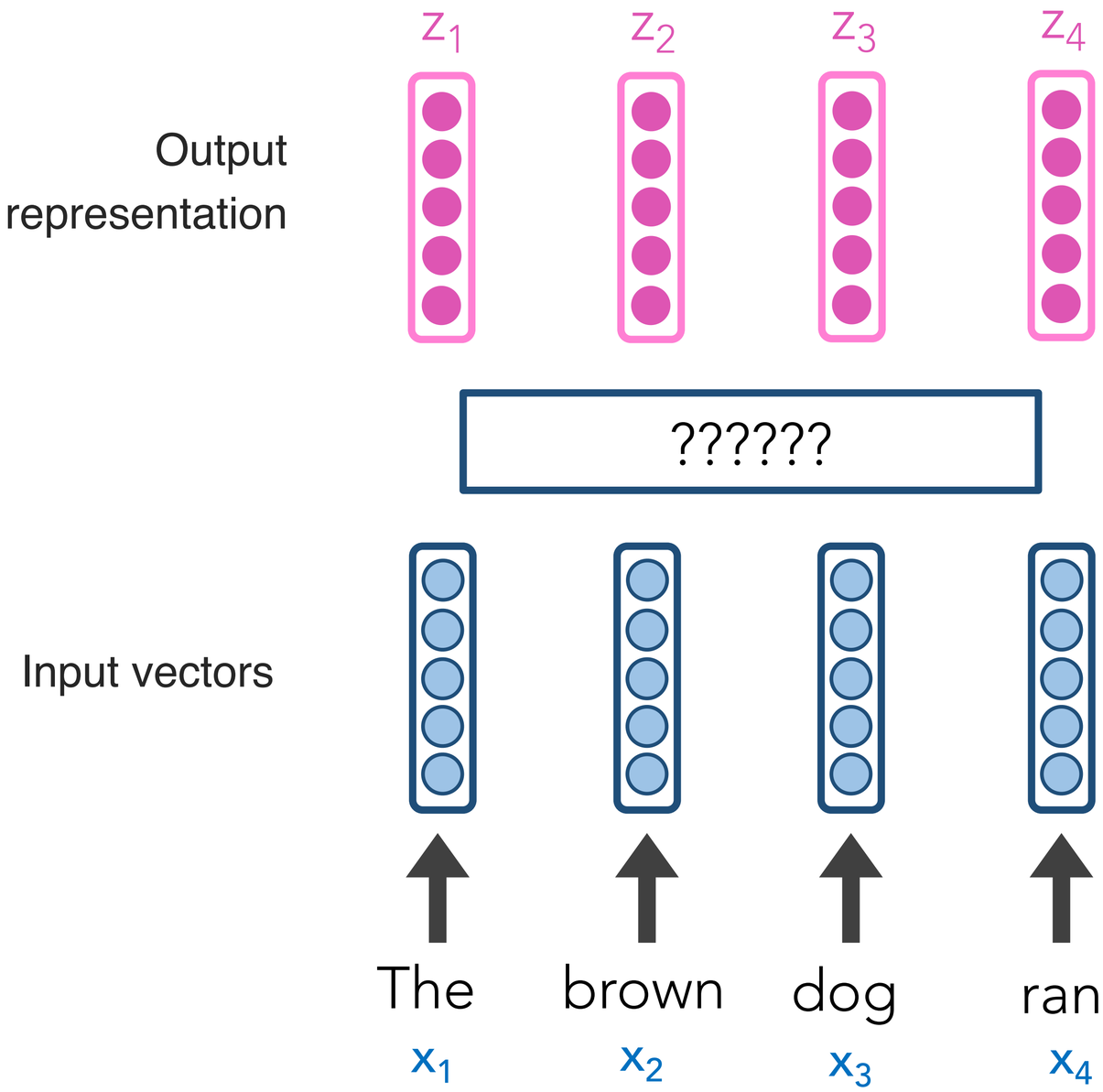
Self Attention Motivation

- Each word in a sequence is transformed into a rich, abstract representation (context embedding) based on the weighted sums of the other words in the same sequence (akin to deep CNN layers)

- Inspired by Attention, we want each word to determine, “how much should I be influenced by each of my neighbors”

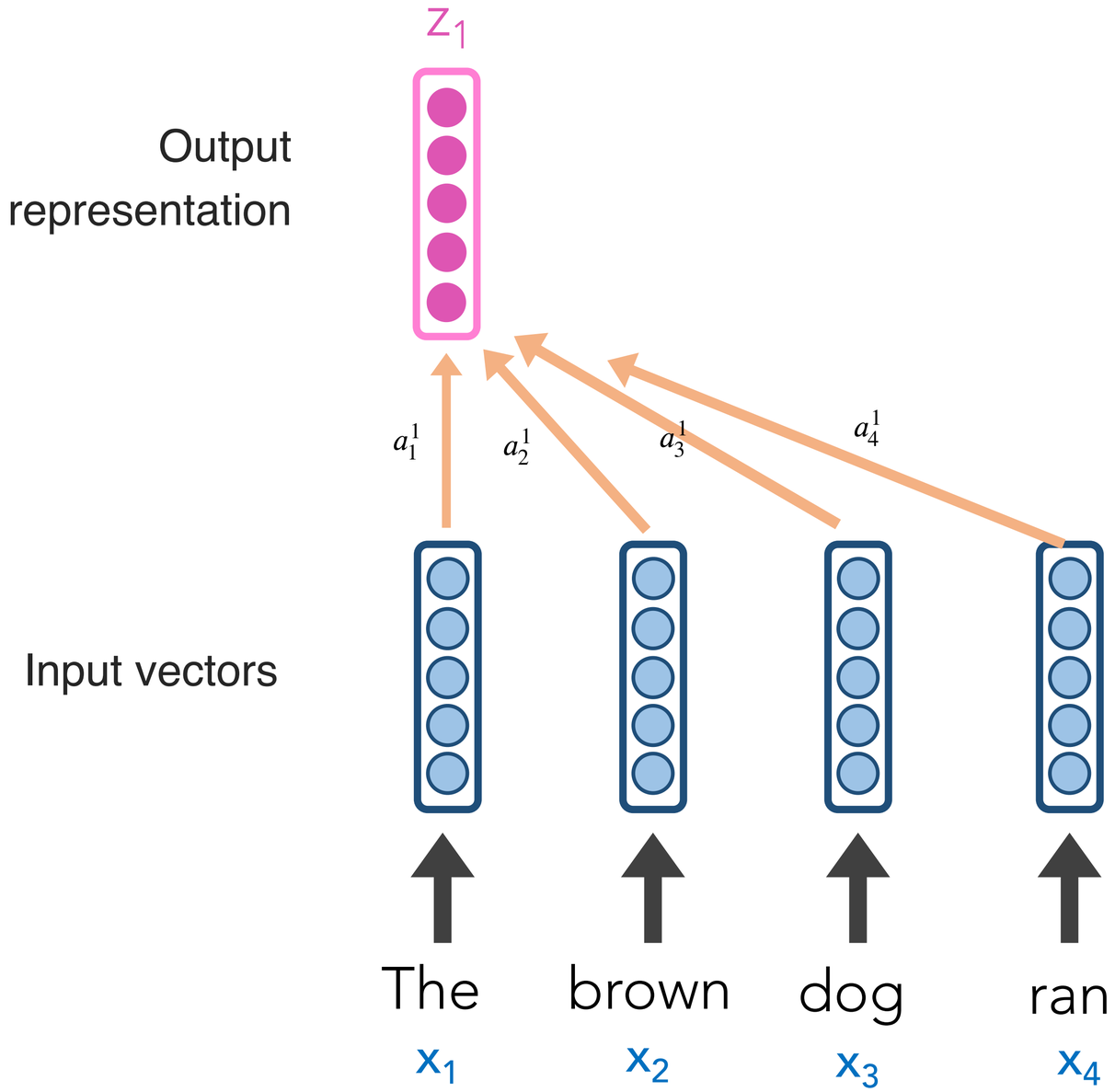
- We want (explicit) positionality

Self Attention Motivation



Self-Attention's goal is to create great representations, z_i, of the input

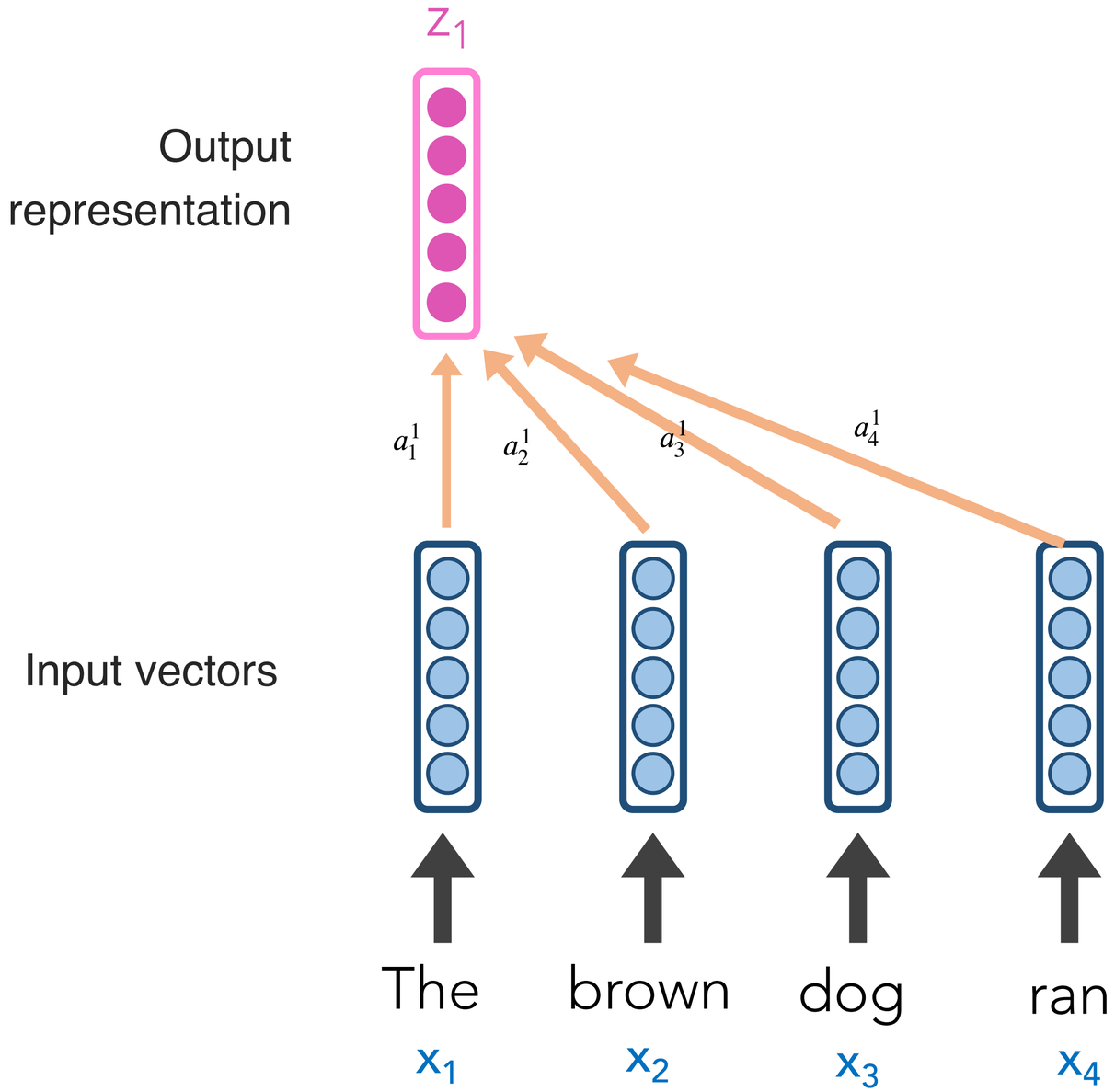
Self Attention Motivation



Self-Attention's goal is to create great representations, z_i, of the input

z_1 will be based on a weighted contribution of x_1, x_2, x_3, x_4

Self Attention Motivation

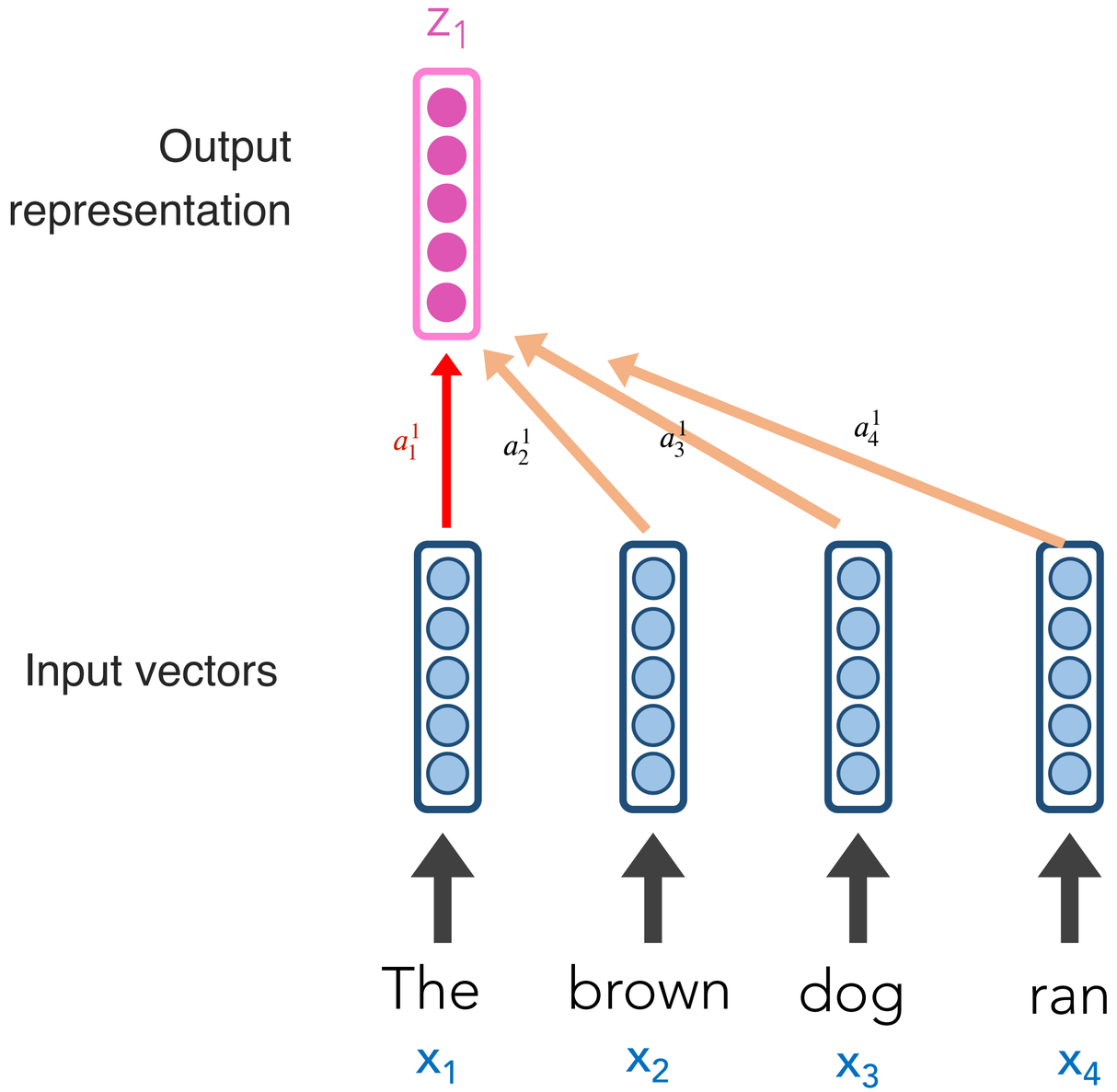


Self-Attention's goal is to create great representations, z_i, of the input

z_1 will be based on a weighted contribution of x_1, x_2, x_3, x_4

a_i^1 is "just" a weight. More is happening under the hood, but it's effectively weighting *versions* of x_1, x_2, x_3, x_4

Self Attention Motivation



Under the hood, each x_i has 3 small, associated vectors.

For example, x_1 has:

- Query q_i
- Key k_i
- Value v_i

Self-Attention Architecture

Step 1: Our Self-Attention Head has just 3 weight matrices W_q, W_k, W_v in total. These same 3 weight matrices are multiplied by each x_i to create all vectors:

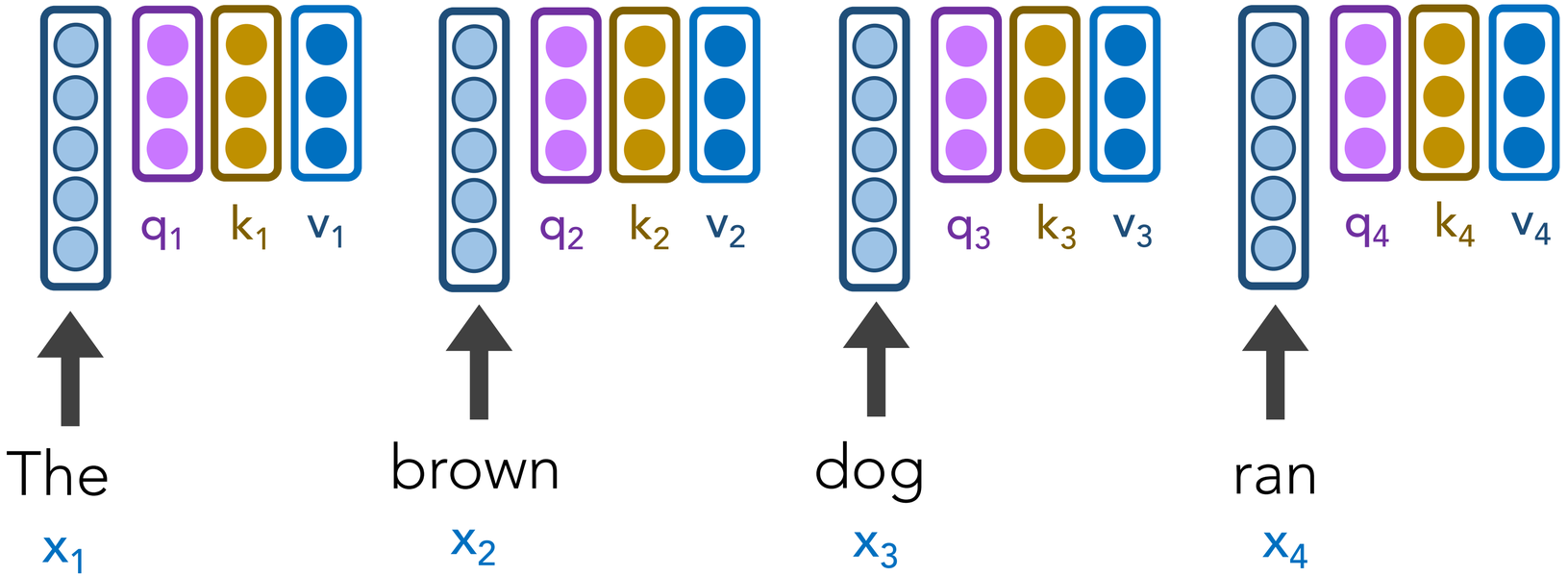
$$q_i = w_q x_i$$

$$k_i = w_k x_i$$

$$v_i = w_v x_i$$

Under the hood, each x_i has 3 small, associated vectors. For example, x_1 has:

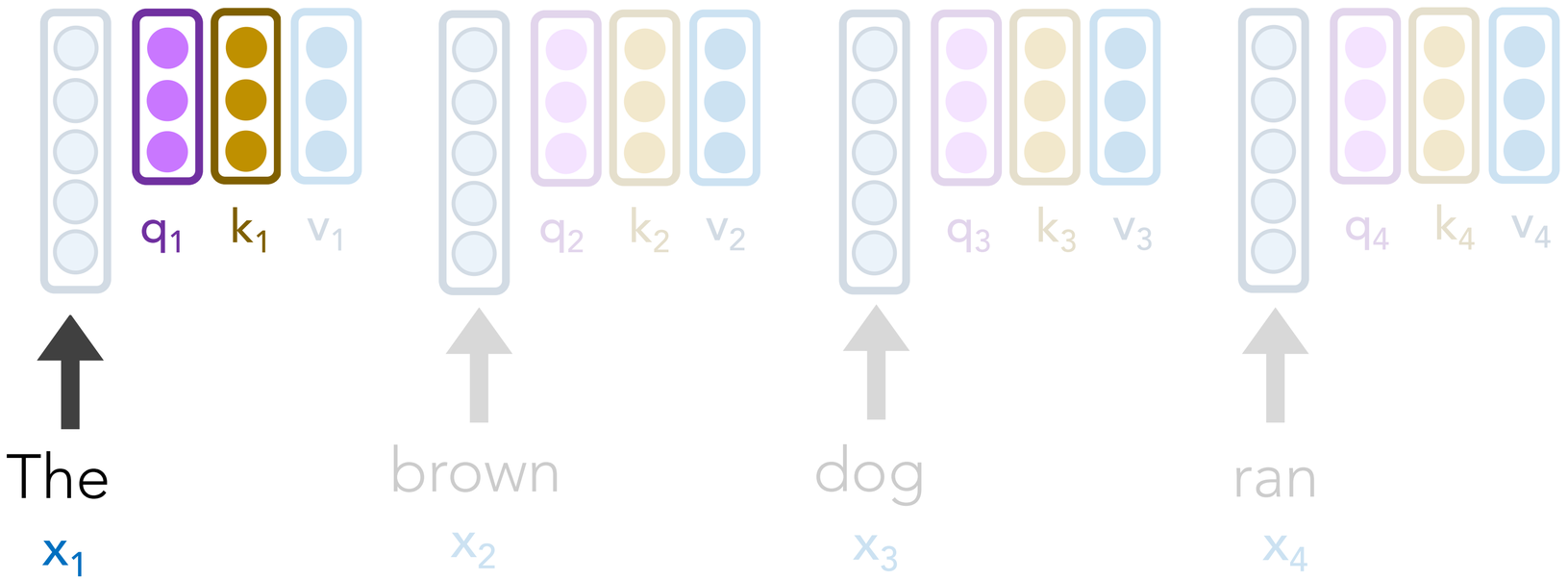
- Query q_1
- Key k_1
- Value v_1



Self-Attention Architecture

Step 2: For word x_1, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_1 = q_1 . k_1 = 112$

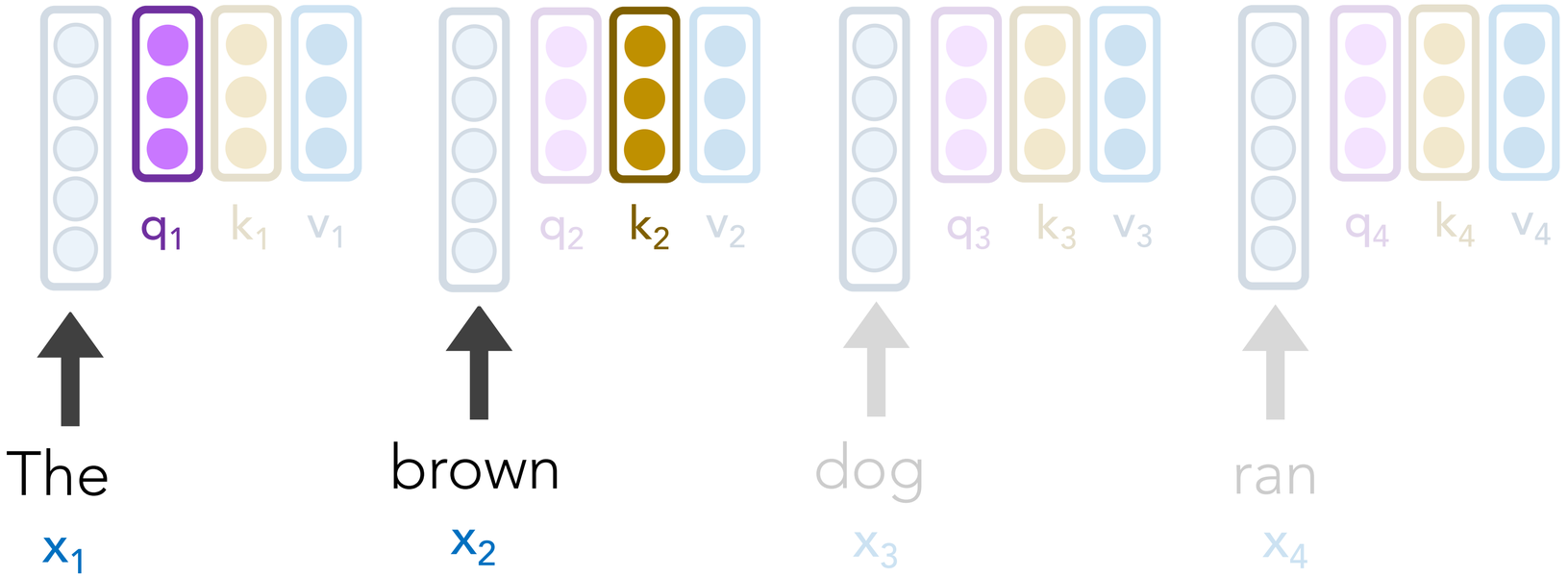


Self-Attention Architecture

Step 2: For word x_1, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_2 = q_1 . k_2 = 96$

$s_1 = q_1 . k_1 = 112$



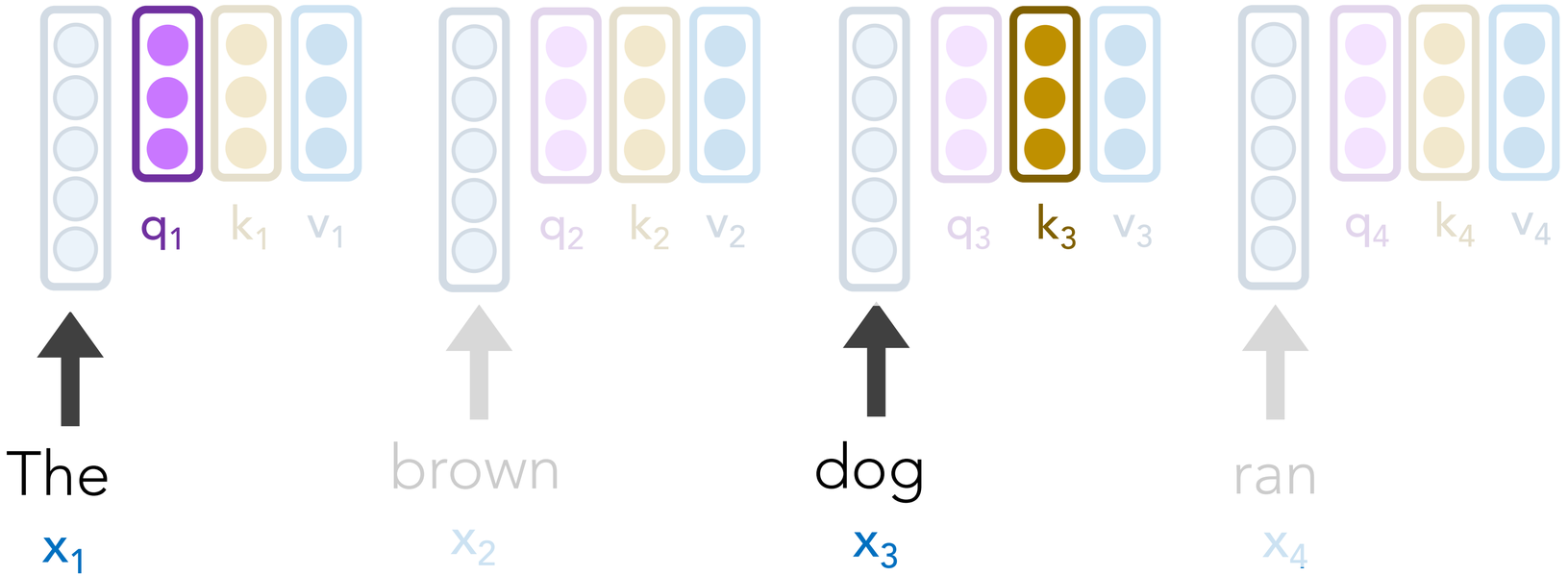
Self-Attention Architecture

Step 2: For word x_1, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_3 = q_1 . k_3 = 16$

$s_2 = q_1 . k_2 = 96$

$s_1 = q_1 . k_1 = 112$



Self-Attention Architecture

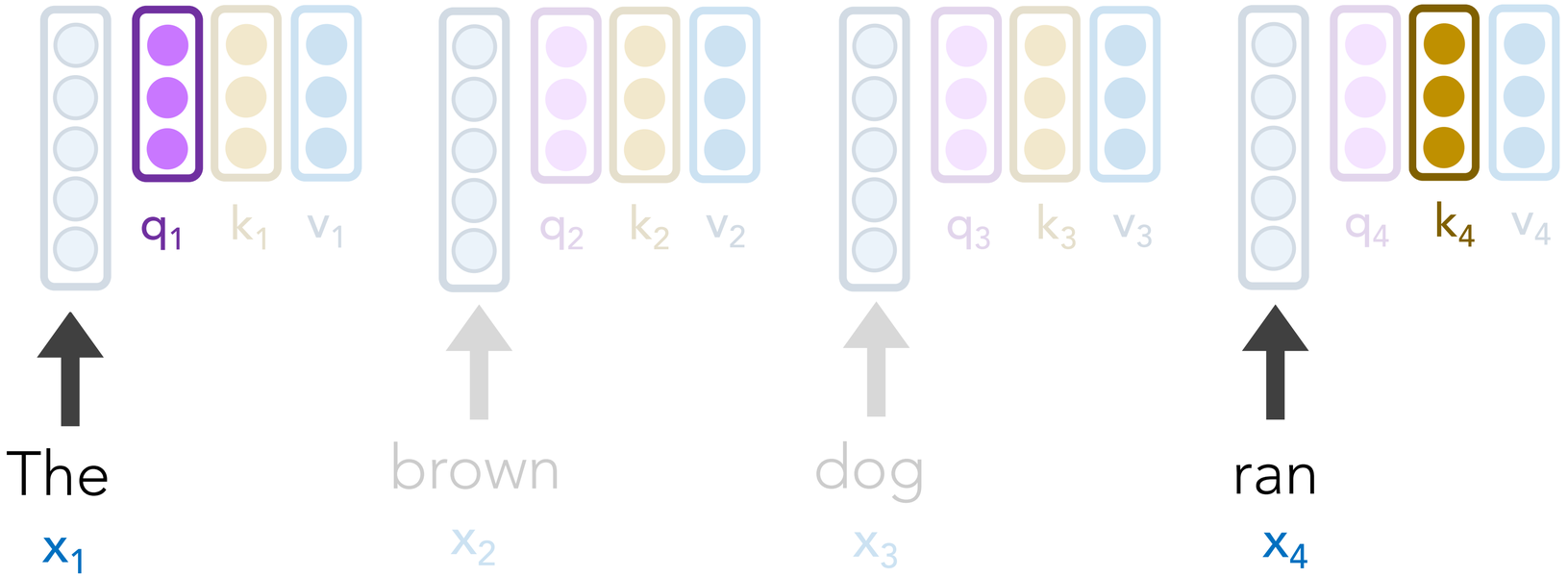
Step 2: For word x_1, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_4 = q_1 . k_4 = 8$

$s_3 = q_1 . k_3 = 16$

$s_2 = q_1 . k_2 = 96$

$s_1 = q_1 . k_1 = 112$



Self-Attention Architecture

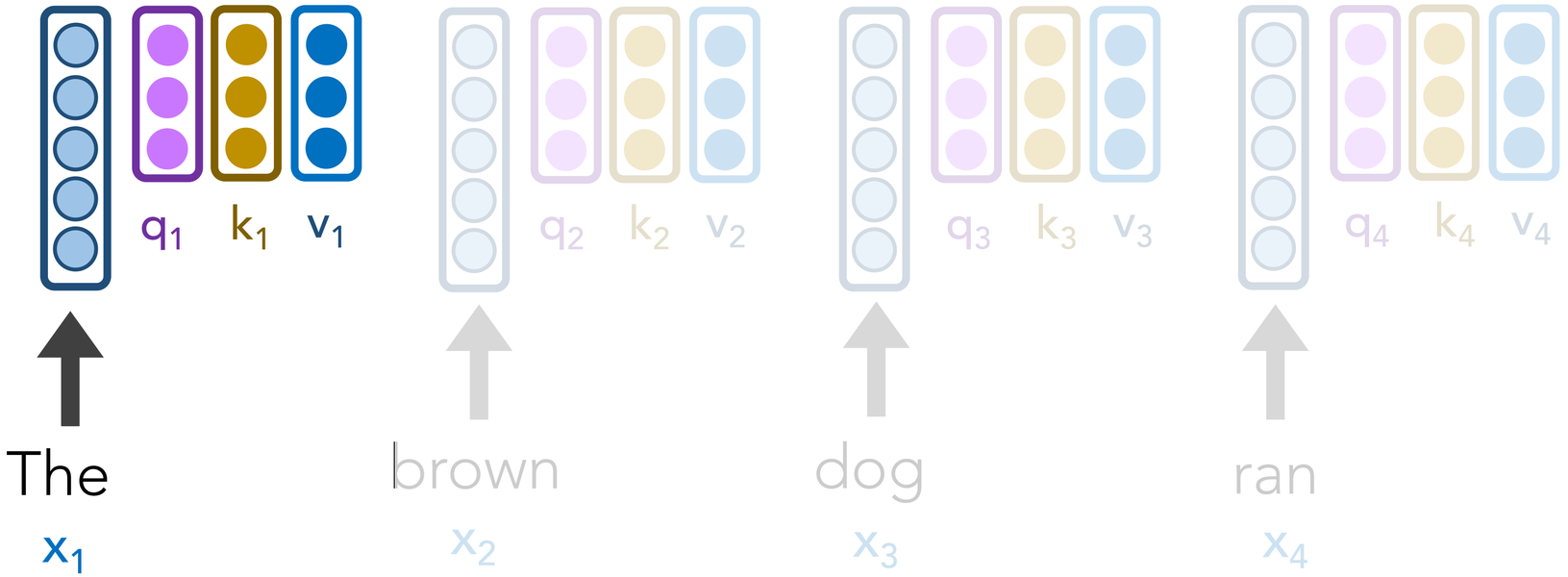
Step 3: Our scores s_1, s_2, s_3, s_4 don't sum to 1. Let's divide by $\sqrt{len(k_i)}$ and softmax it

$s_4 = q_1 . k_4 = 8$ $\quad a_4 = \sigma(s_4/8) = 0$

$s_3 = q_1 . k_3 = 16$ $\quad a3 = \sigma(s_3/8) = .01$

$s_2 = q_1 . k_2 = 96$ $\quad a_2 = \sigma(s_2/8) = .12$

$s_1 = q_1 . k_1 = 112$ $\quad a_1 = \sigma(s_1/8) = .87$



Self-Attention Architecture

Step 3: Our scores s_1, s_2, s_3, s_4 don't sum to 1. Let's divide by $\sqrt{len(k_i)}$ and softmax it

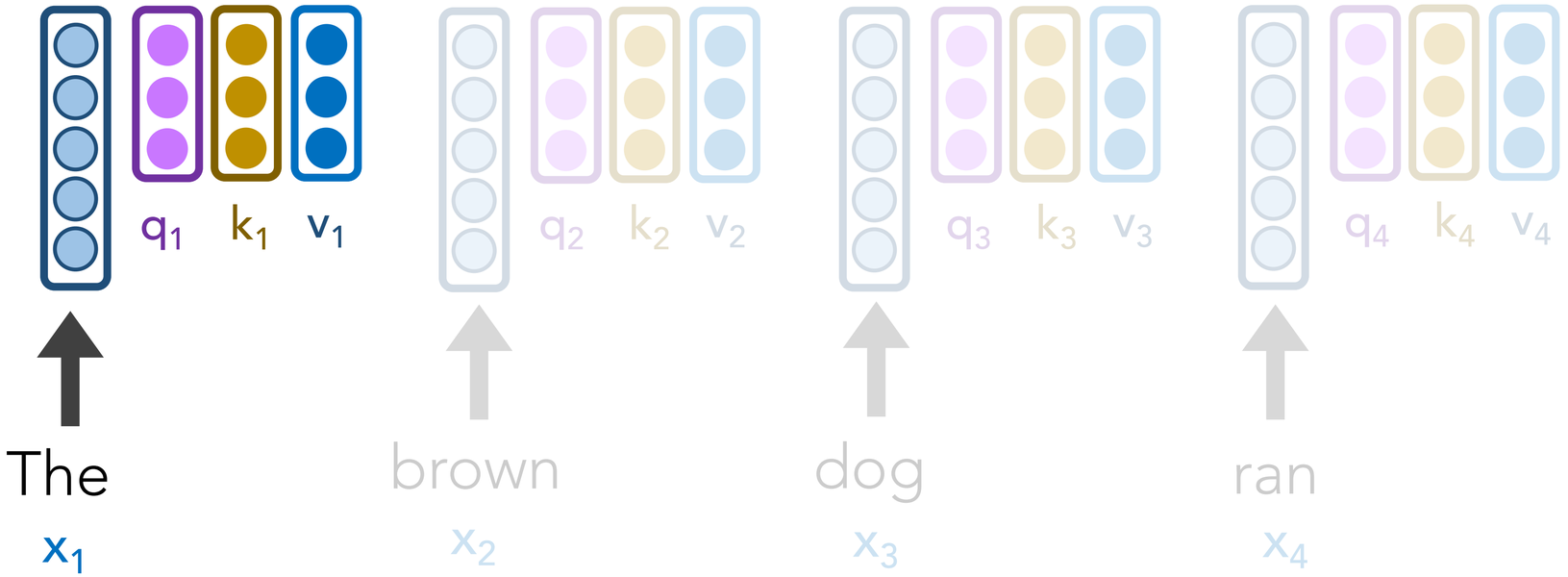
$s_4 = q_1 \cdot k_4 = 8$ $\quad a_4 = \sigma(s_4/8) = 0$

$s_3 = q_1 \cdot k_3 = 16$ $\quad a3 = \sigma(s_3/8) = .01$

$s_2 = q_1 \cdot k_2 = 96$ $\quad a_2 = \sigma(s_2/8) = .12$

$s_1 = q_1 \cdot k_1 = 112$ $\quad a_1 = \sigma(s_1/8) = .87$

Instead of these a_i values directly weighting our original x_i word vectors, they weight our v_i vectors.

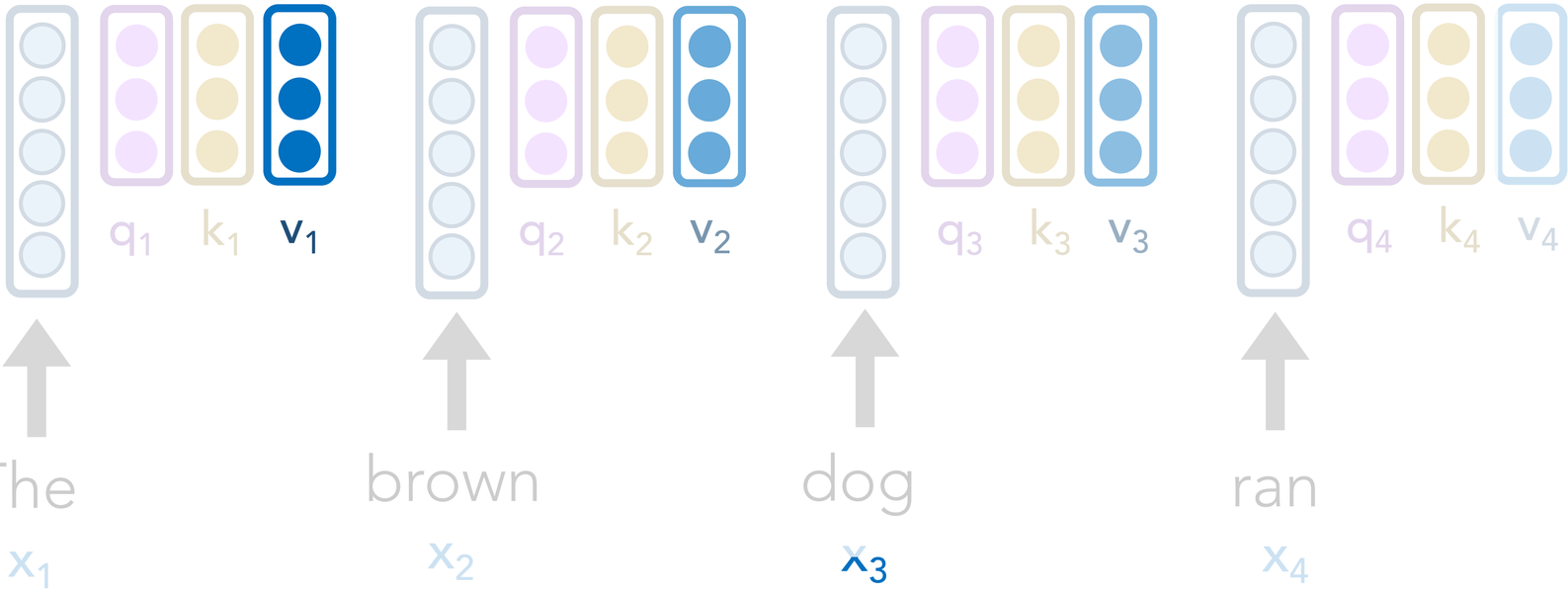


Self-Attention Architecture

Step 4: Let's weight our v_i vectors and simply sum them up!

$$z_1 = \mathbf{a_1} \cdot v_1 + \mathbf{a_2} \cdot v_2 + \mathbf{a_3} \cdot v_3 + \mathbf{a_4} \cdot v_4$$

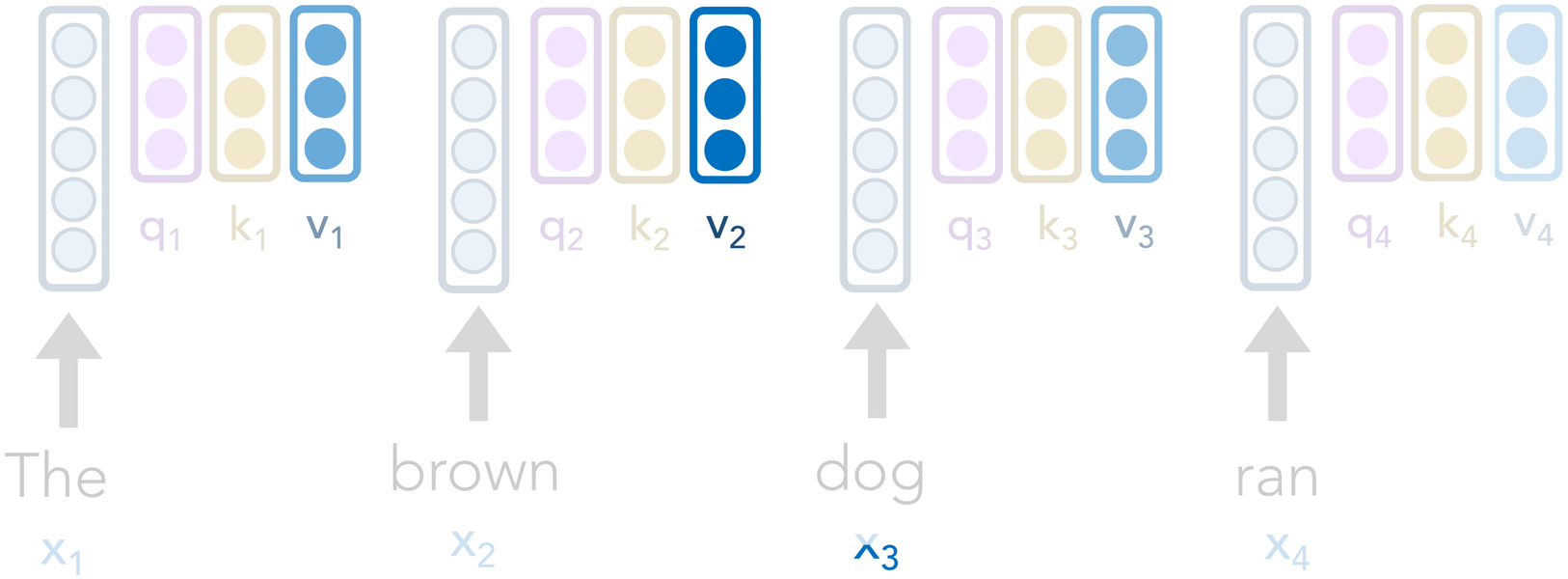
$$= 0.87 \cdot v_1 + \mathbf{0.12} \cdot v_2 + \mathbf{0.01} \cdot v_3 + \mathbf{0} \cdot v_4$$



Self-Attention Architecture

Step 5: We repeat this for all other words, yielding us with great, new z_i representations!

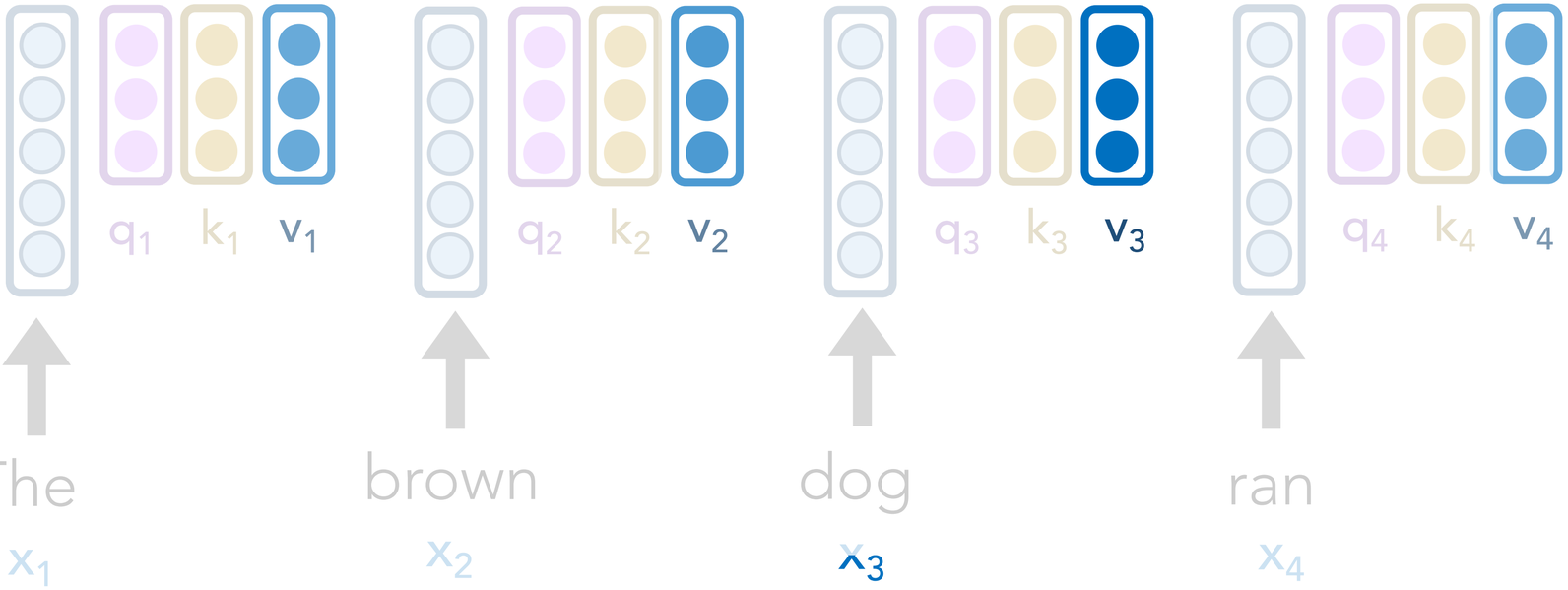
$$z_2 = a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + a_4 \cdot v_4$$



Self-Attention Architecture

Step 5: We repeat this for all other words, yielding us with great, new z_i representations!

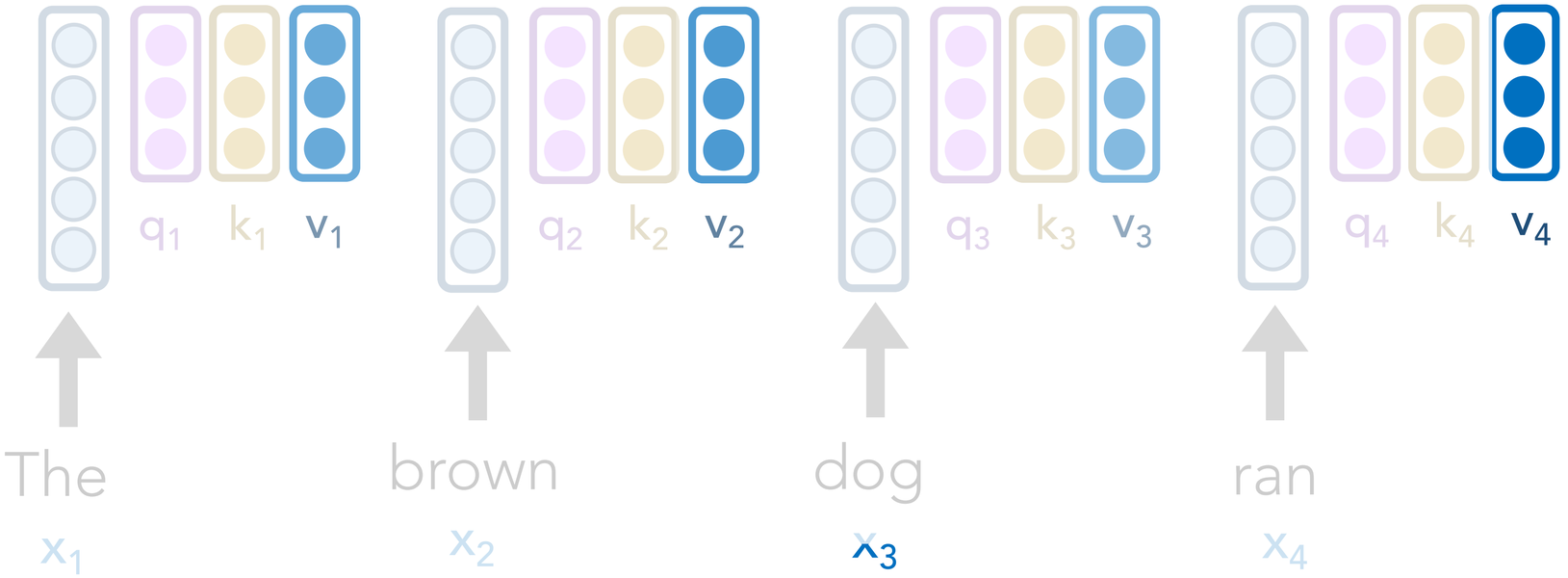
$$z_3 = a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + a_4 \cdot v_4$$



Self-Attention Architecture

Step 5: We repeat this for all other words, yielding great, new z_i representations!

$$z_4 = a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + a_4 \cdot v_4$$



Self-Attention Architecture

Let’s illustrate another example:

$$z_2 = a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + a_4 \cdot v_4$$

Remember, we use the same 3 weight matrices

W_q, W_k, W_v as we did for computing z_1.

This gives us q_2, k_2, v_2

Self-Attention Architecture

Step 1: Our Self-Attention Head I has just 3 weight matrices W_q, W_k, W_v in total. These same 3 weight matrices are multiplied by each x_i to create all vectors:

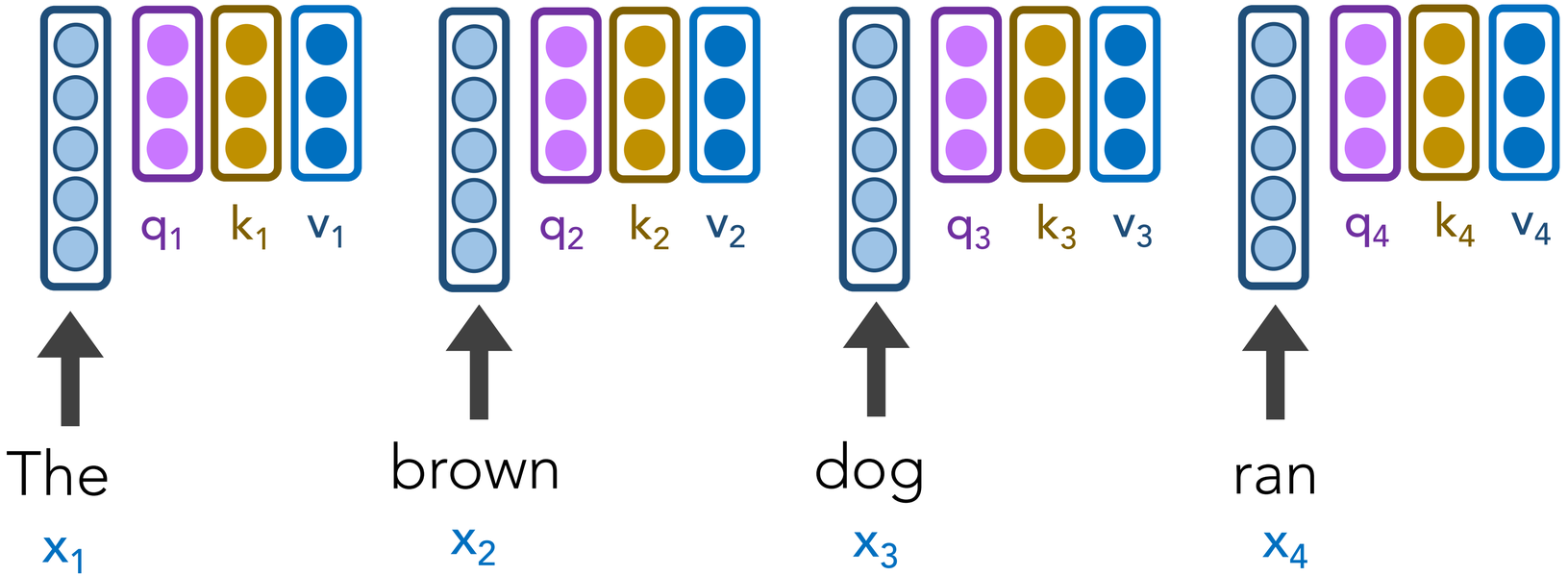
$$q_i = w_q x_i$$

$$k_i = w_k x_i$$

$$v_i = w_v x_i$$

Under the hood, each x_i has 3 small, associated vectors. For example, x_1 has:

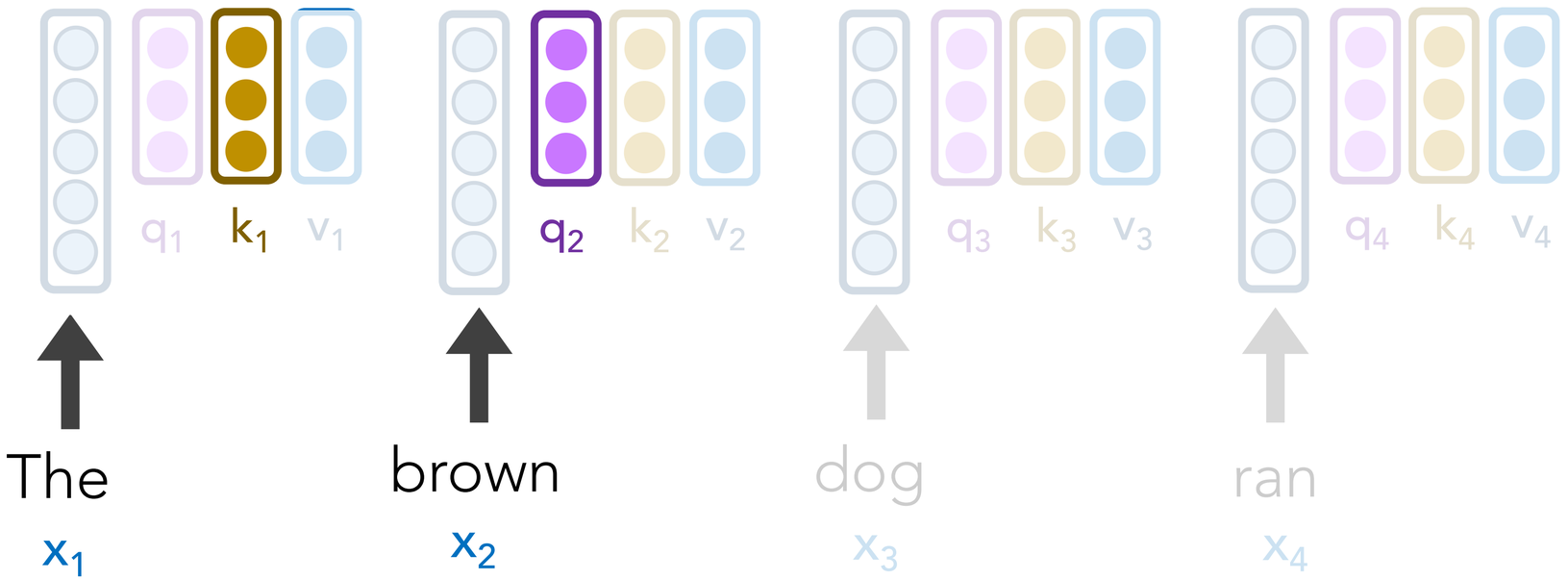
- Query q_1
- Key k_1
- Value v_1



Self-Attention Architecture

Step 2: For word x_2, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_1 = q_2 \cdot k_1 = 92$

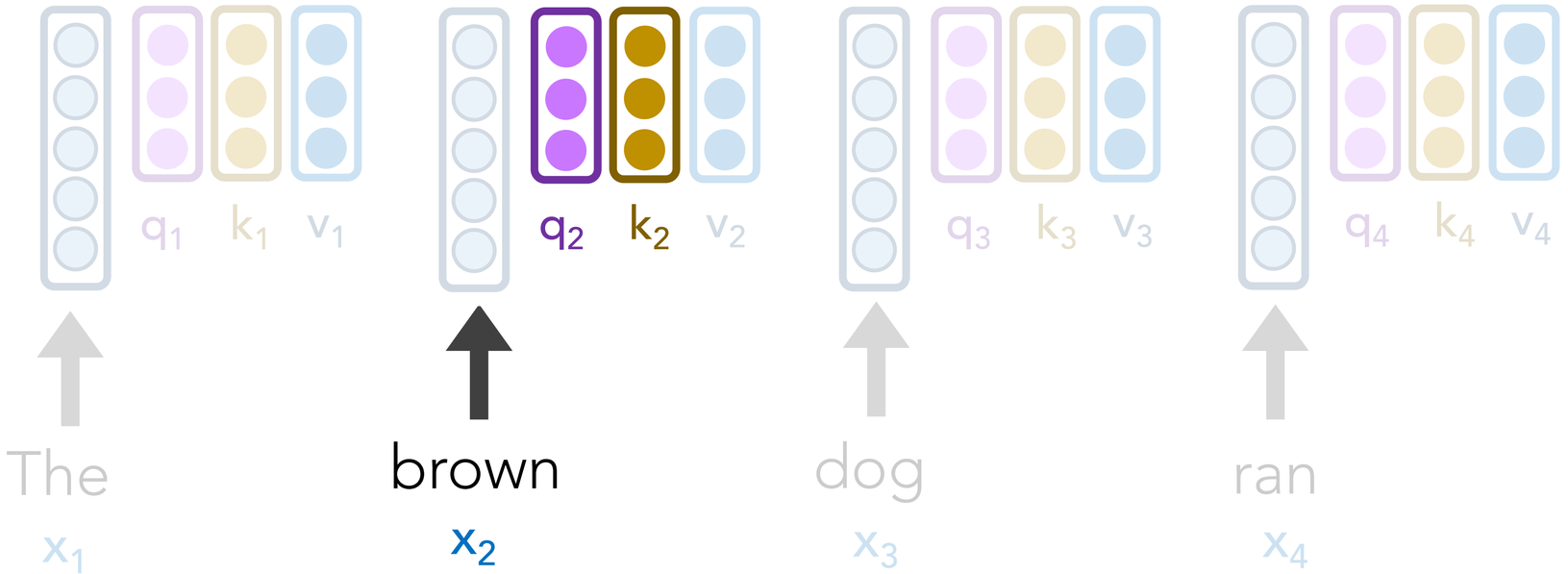


Self-Attention Architecture

Step 2: For word x_2, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_2 = q_2 \cdot k_2 = 124$

$s_1 = q_2 \cdot k_1 = 92$



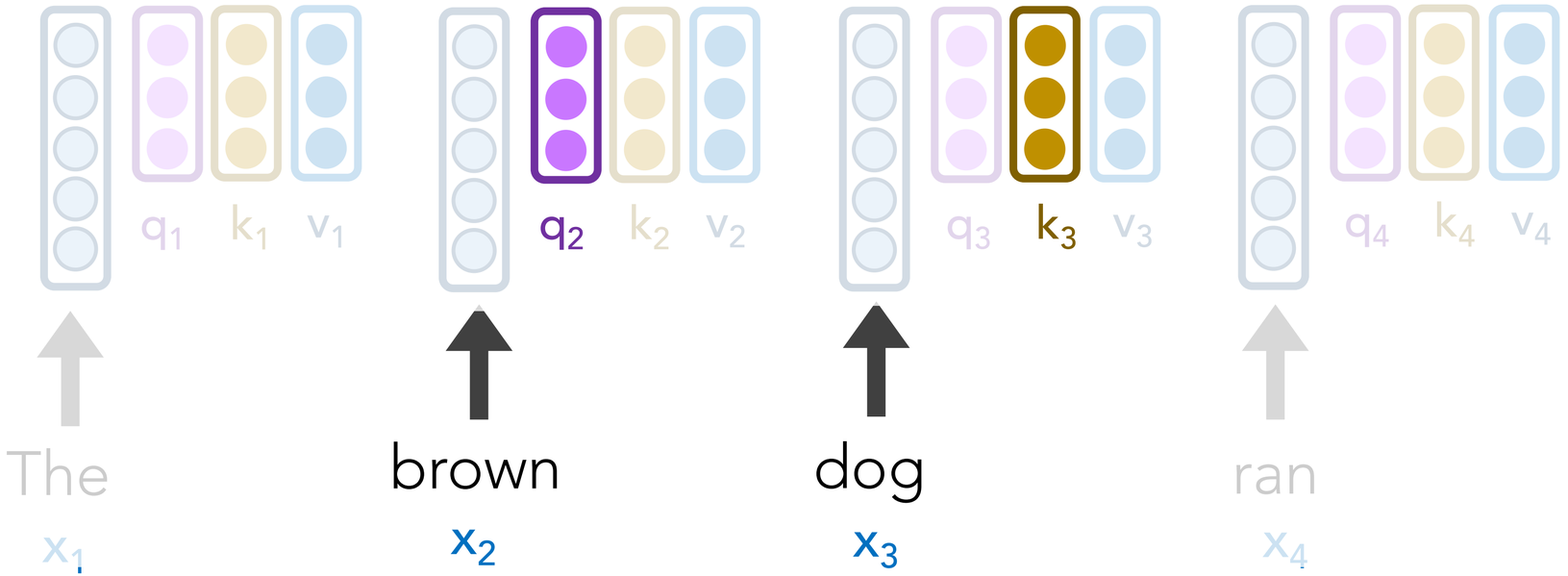
Self-Attention Architecture

Step 2: For word x_2, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_3 = q_2 \cdot k_3 = 22$

$s_2 = q_2 \cdot k_2 = 124$

$s_1 = q_2 \cdot k_1 = 92$



Self-Attention Architecture

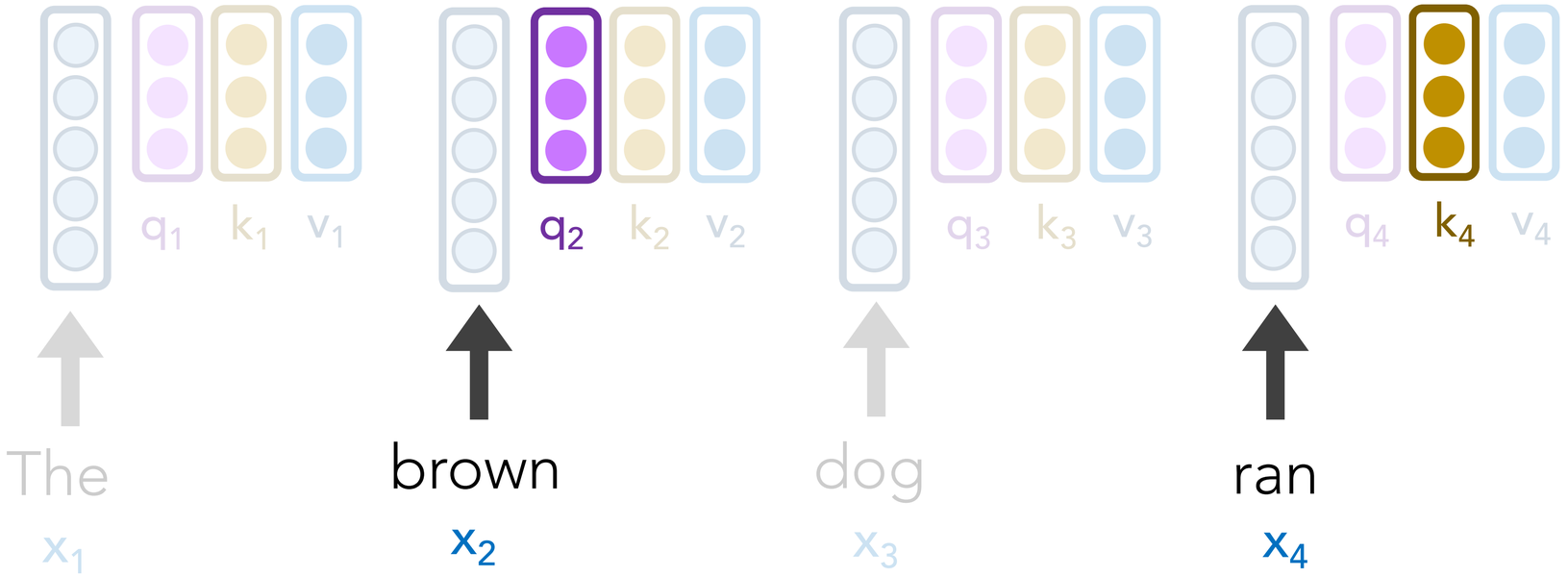
Step 2: For word x_2, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_4 = q_2 . k_4 = 8$

$s_3 = q_2 . k_3 = 22$

$s_2 = q_2 . k_2 = 124$

$s_1 = q_2 . k_1 = 92$



Self-Attention Architecture

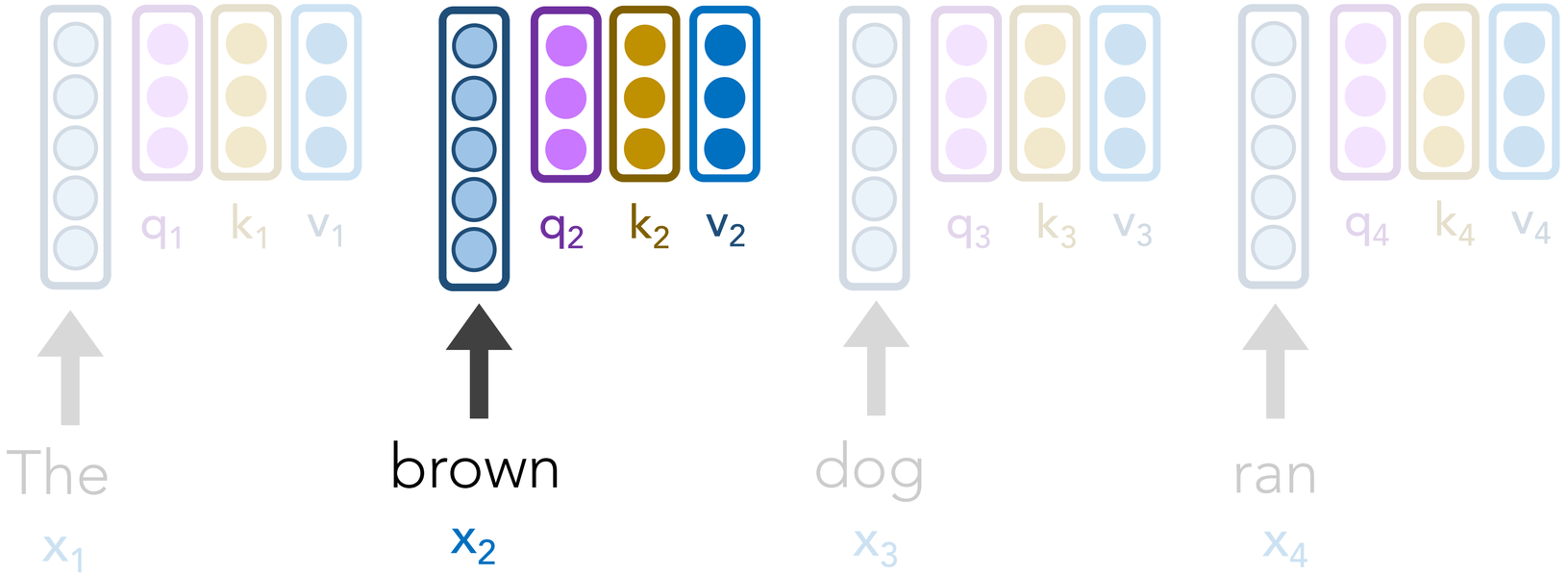
Step 3: Our scores s_1, s_2, s_3, s_4 don't sum to 1. Let's divide by $\sqrt{len(k_i)}$ and softmax it

$s_4 = q_2 \cdot k_4 = 8$ $\quad a_4 = \sigma(s_4/8) = 0$

$s_3 = q_2 \cdot k_3 = 22$ $\quad a3 = \sigma(s_3/8) = .01$

$s_2 = q_2 \cdot k_2 = 124$ $\quad a_2 = \sigma(s_2/8) = .91$

$s_1 = q_2 \cdot k_1 = 92$ $\quad a_1 = \sigma(s_1/8) = .08$



Self-Attention Architecture

Step 3: Our scores s_1, s_2, s_3, s_4 don't sum to 1. Let's divide by $\sqrt{len(k_i)}$ and softmax it

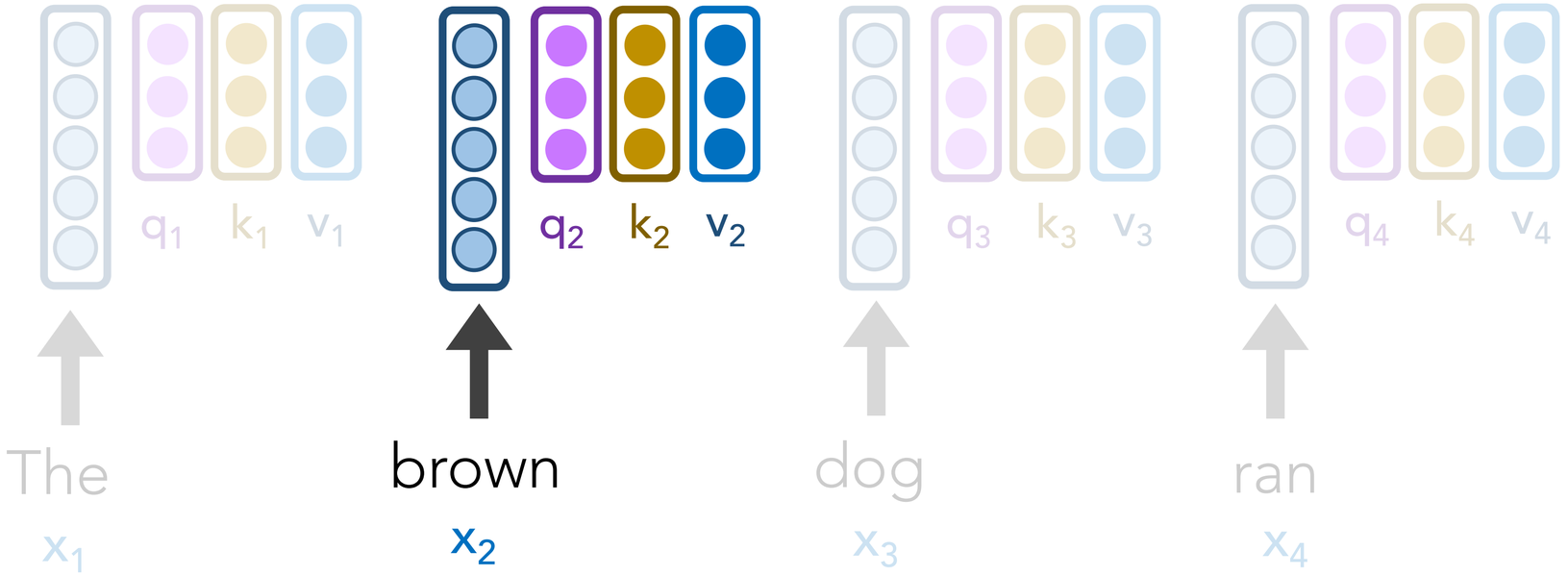
$s_4 = q_2 \cdot k_4 = 8$ $\quad$ $a_4 = \sigma(s_4/8) = 0$

$s_3 = q_2 \cdot k_3 = 22$ $\quad$ $a3 = \sigma(s_3/8) = .01$

$s_2 = q_2 \cdot k_2 = 124$ $\quad$ $a_2 = \sigma(s_2/8) = .91$

$s_1 = q_2 \cdot k_1 = 92$ $\quad$ $a_1 = \sigma(s_1/8) = .08$

Instead of these a_i values directly weighting our original x_i word vectors, they directly weight our v_i vectors.

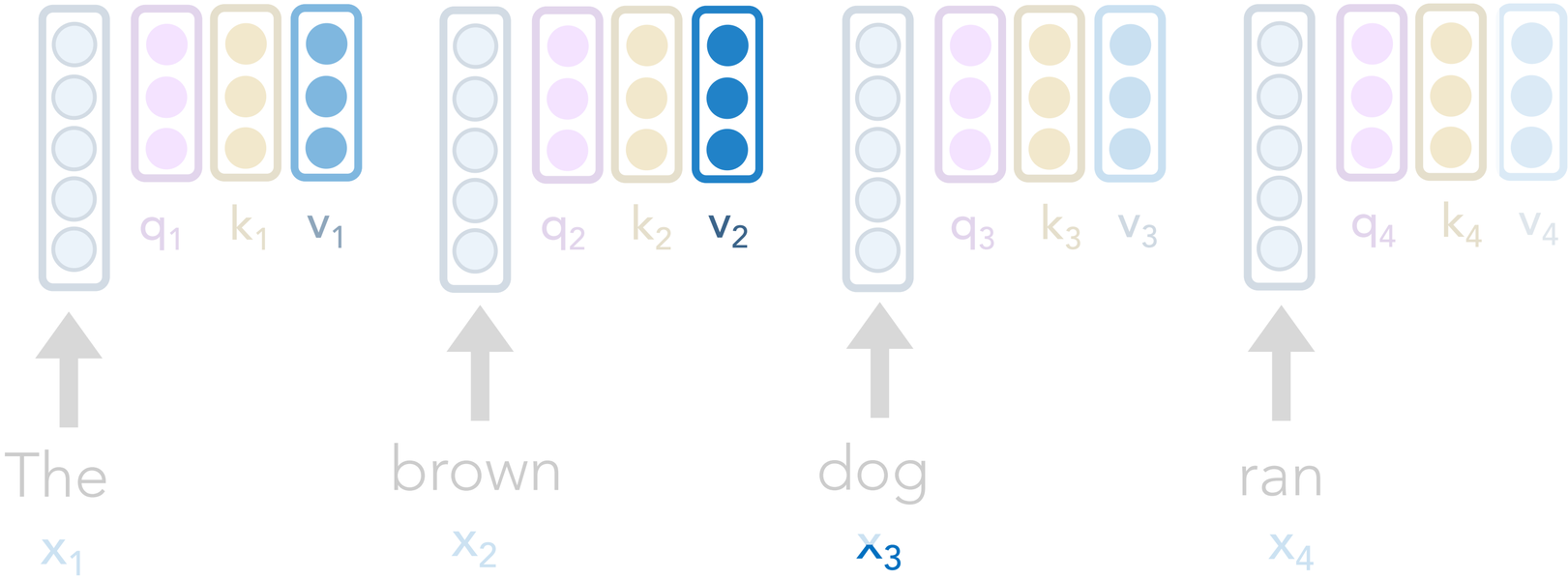


Self-Attention Architecture

Step 4: Let's weight our v_i vectors and simply sum them up!

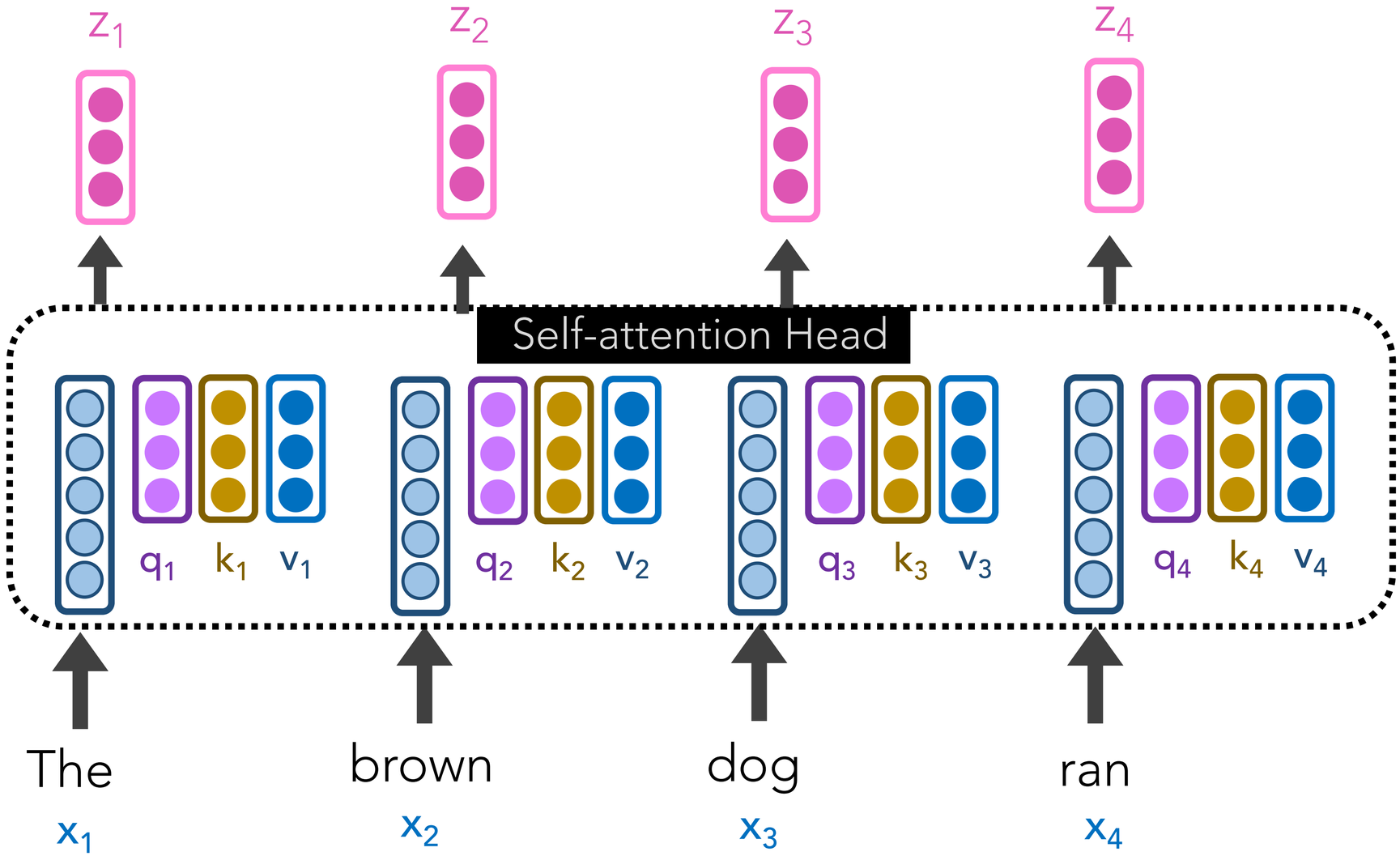
$$z_2 = a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + a_4 \cdot v_4$$

$$= 0.08 \cdot v_1 + 0.91 \cdot v_2 + 0.01 \cdot v_3 + 0 \cdot v_4$$



Self-Attention Architecture

Tada! Now we have great, new representations z_i via a self-attention head



Self-Attention Summary

Strengths

- Context-aware
- Much more powerful than static embeddings

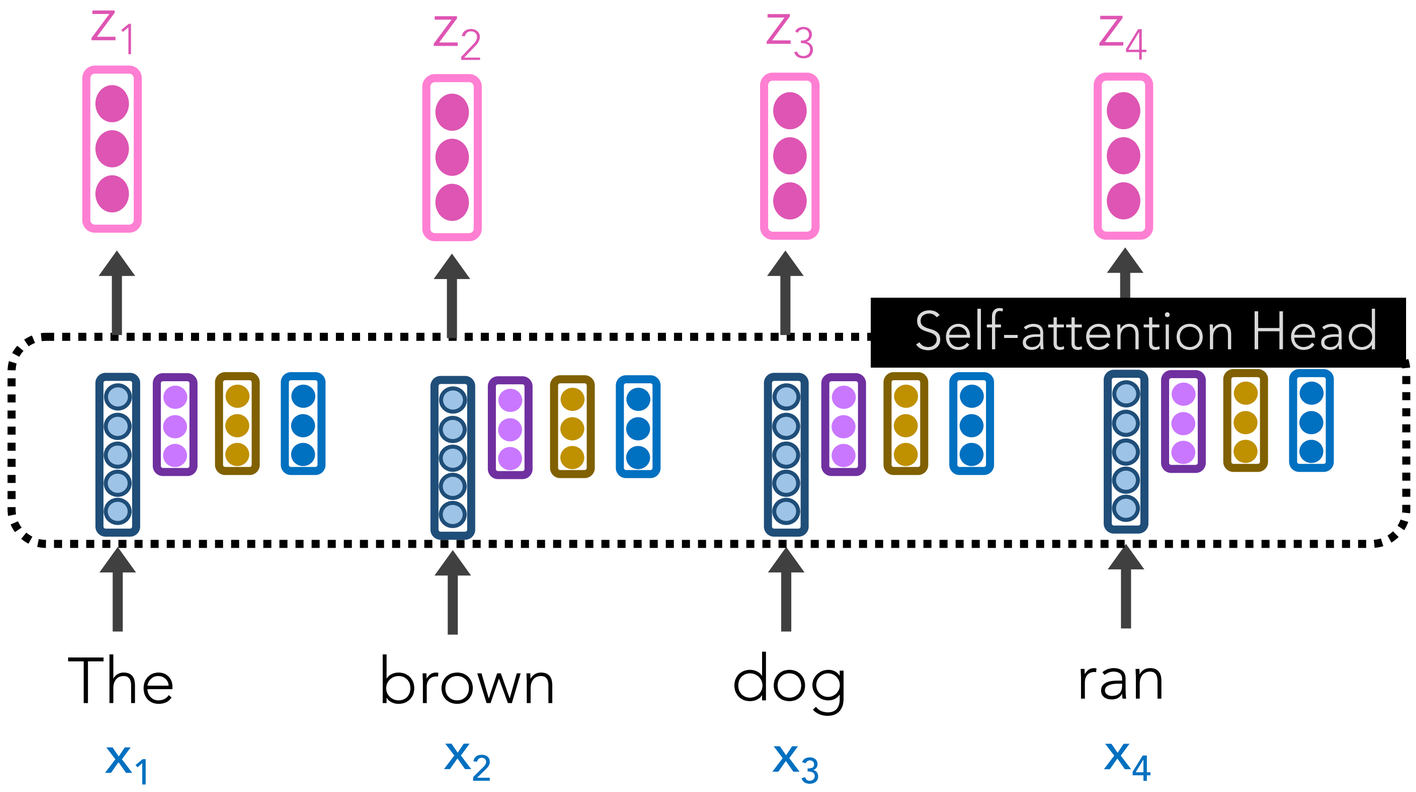
Issues?

- Slower to train
- Cannot easily be pre-computed

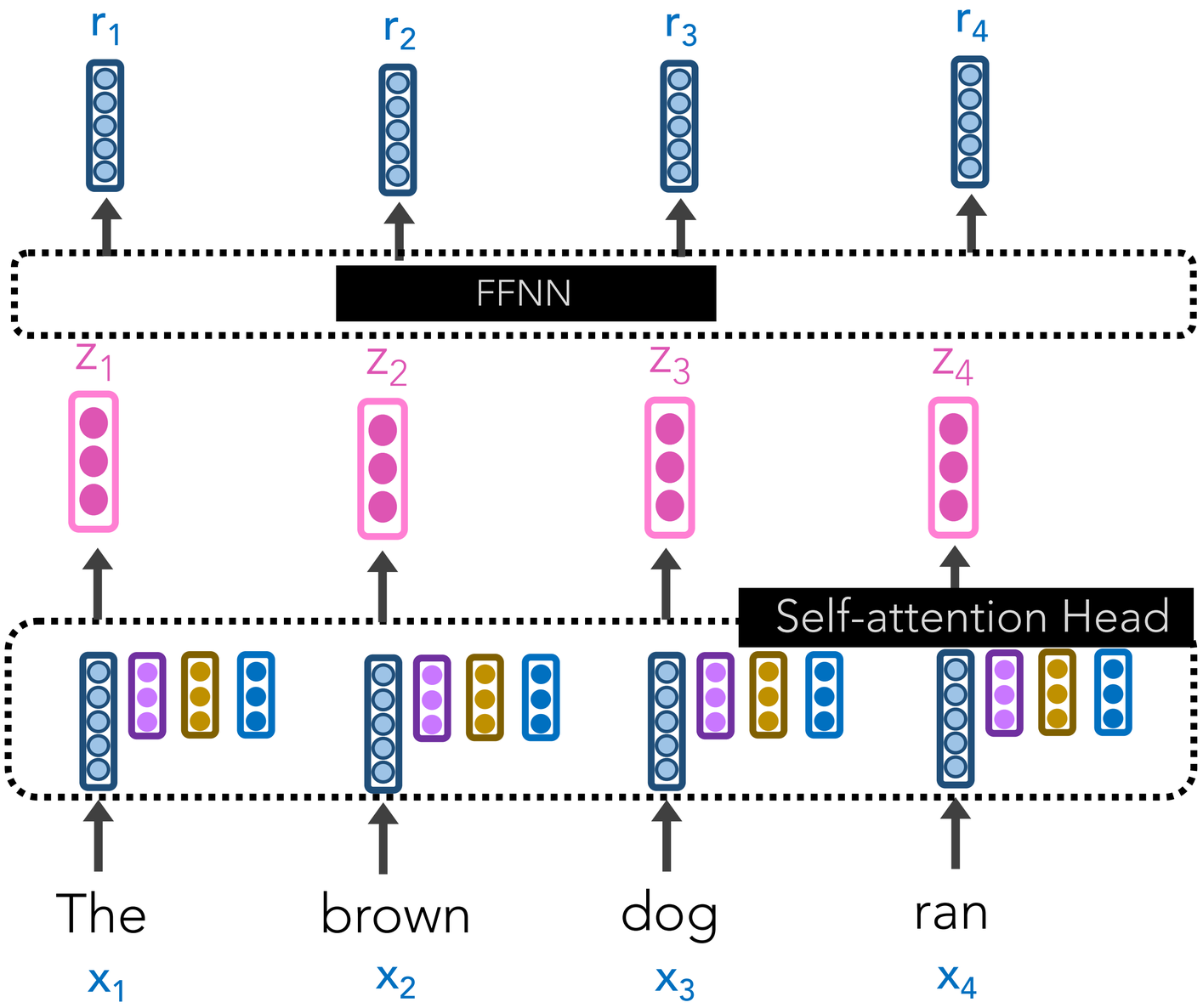
Transformers

Transformer Architecture

Let's further pass each z_i through a FFNN

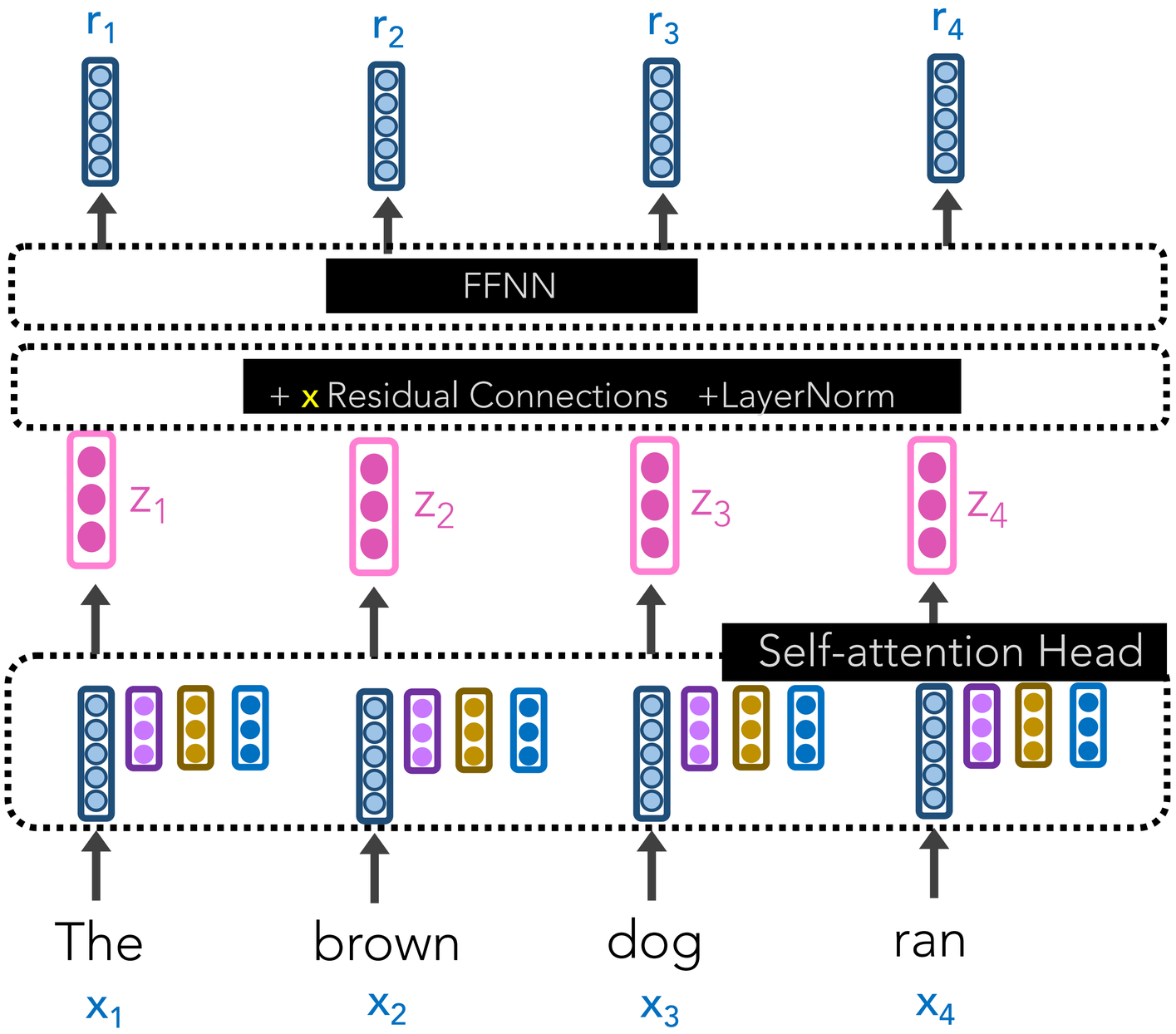


Transformer Architecture



Let's further pass each z_i through a FFNN

Transformer Architecture

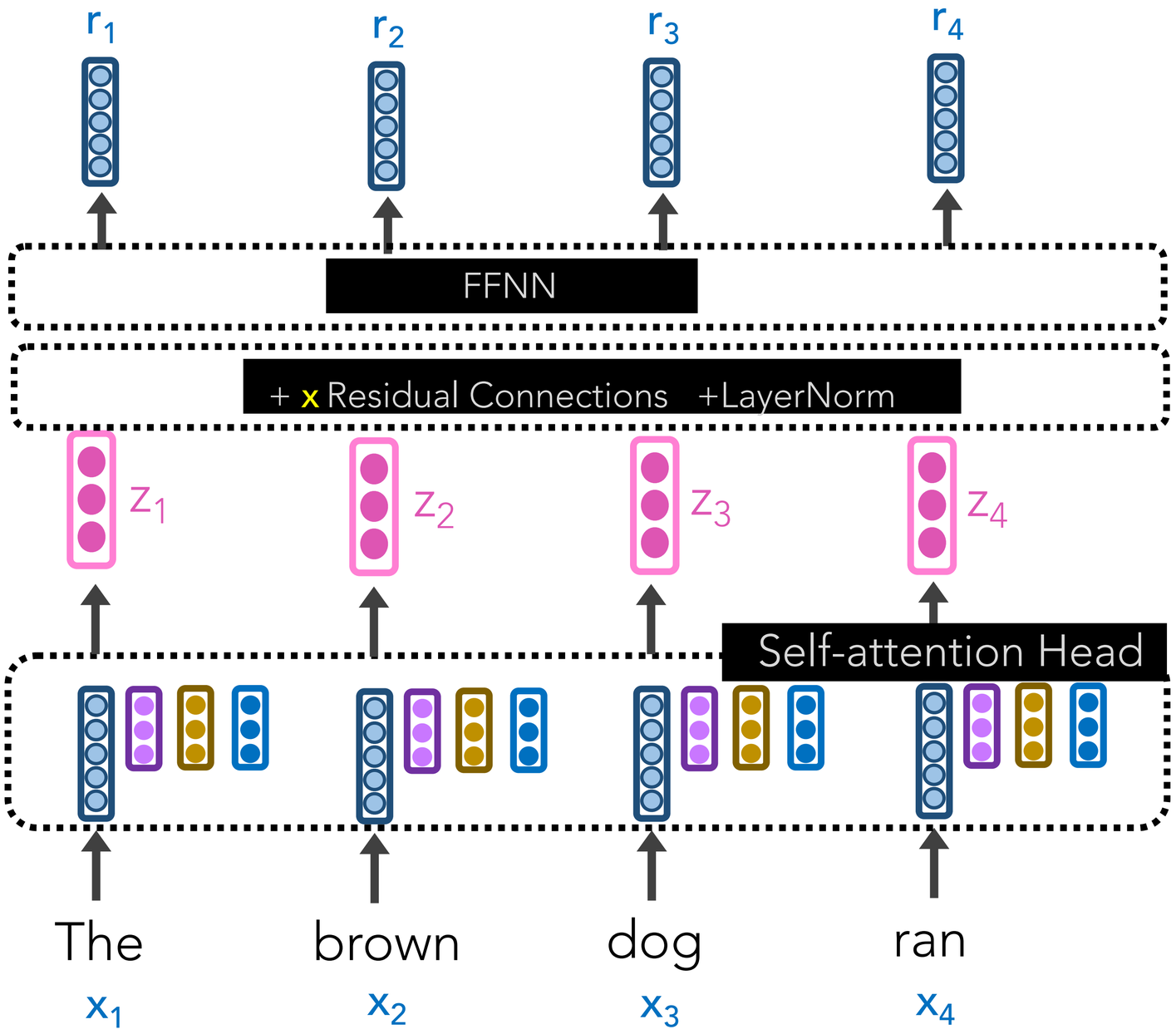


Let's further pass each z_i through a FFNN

We concat w/ a residual connection to help ensure relevant info is getting forward passed.

We perform LayerNorm to stabilize the network and allow for proper gradient flow. You should do this after the FFNN, too.

Transformer Architecture



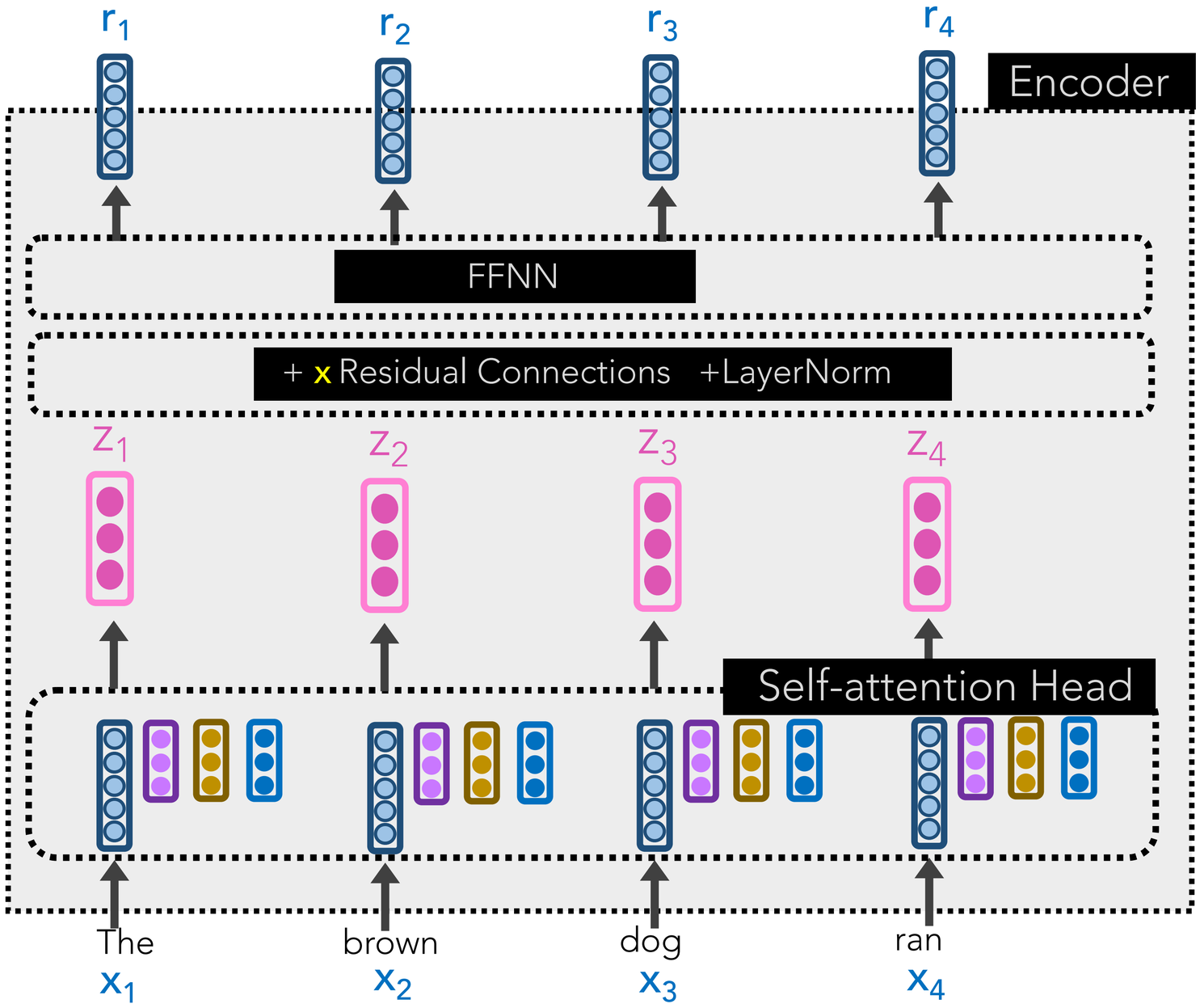
Let's further pass each z_i through a FFNN

We concat w/ a residual connection to help ensure relevant info is getting forward passed.

We perform LayerNorm to stabilize the network and allow for proper gradient flow. You should do this after the FFNN, too.

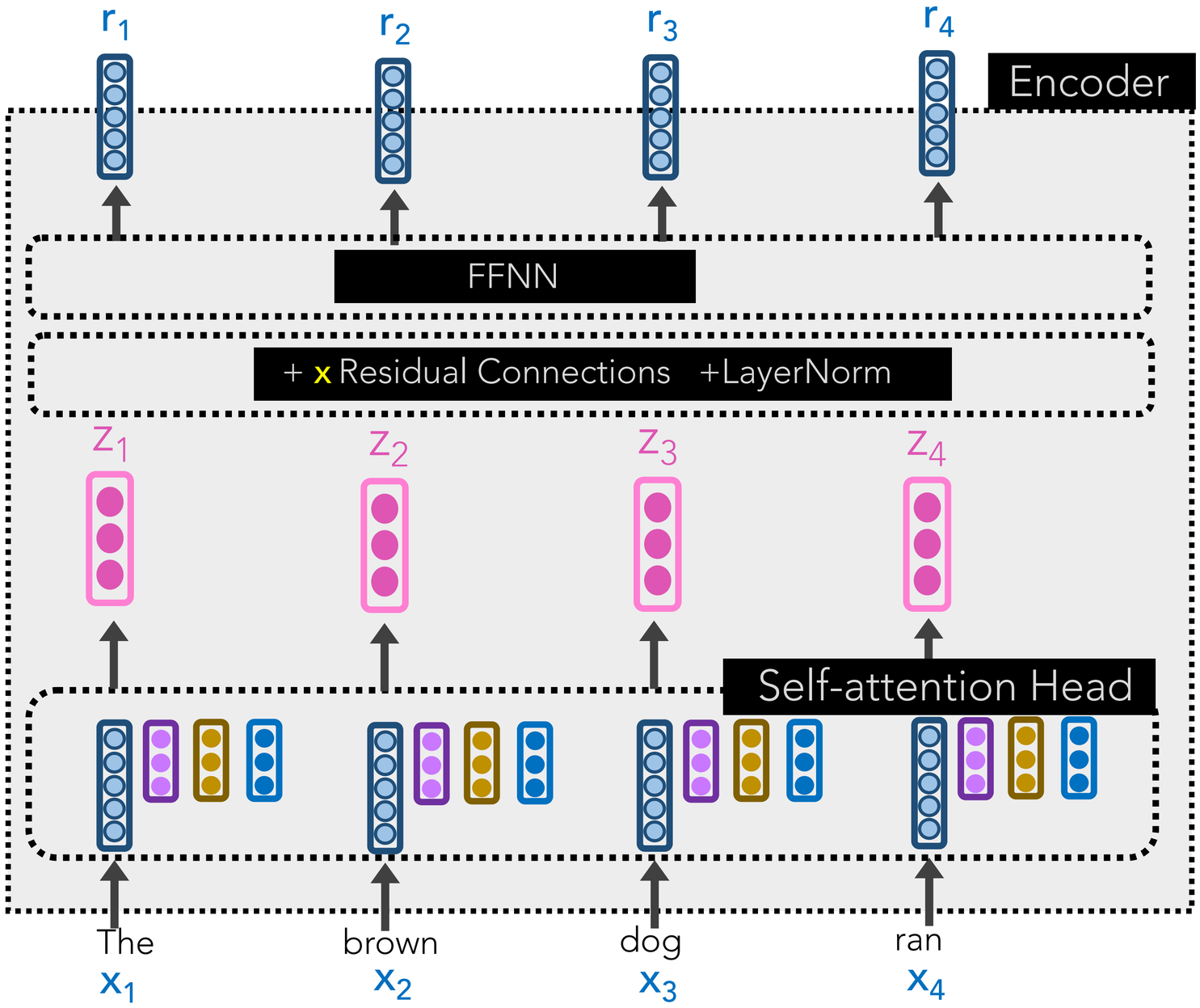
Each z_i can be computed in parallel, unlike LSTMs!

Transformer Architecture



Yay! Our r_i vectors are our new representations, and this entire process is called a Transformer Encoder

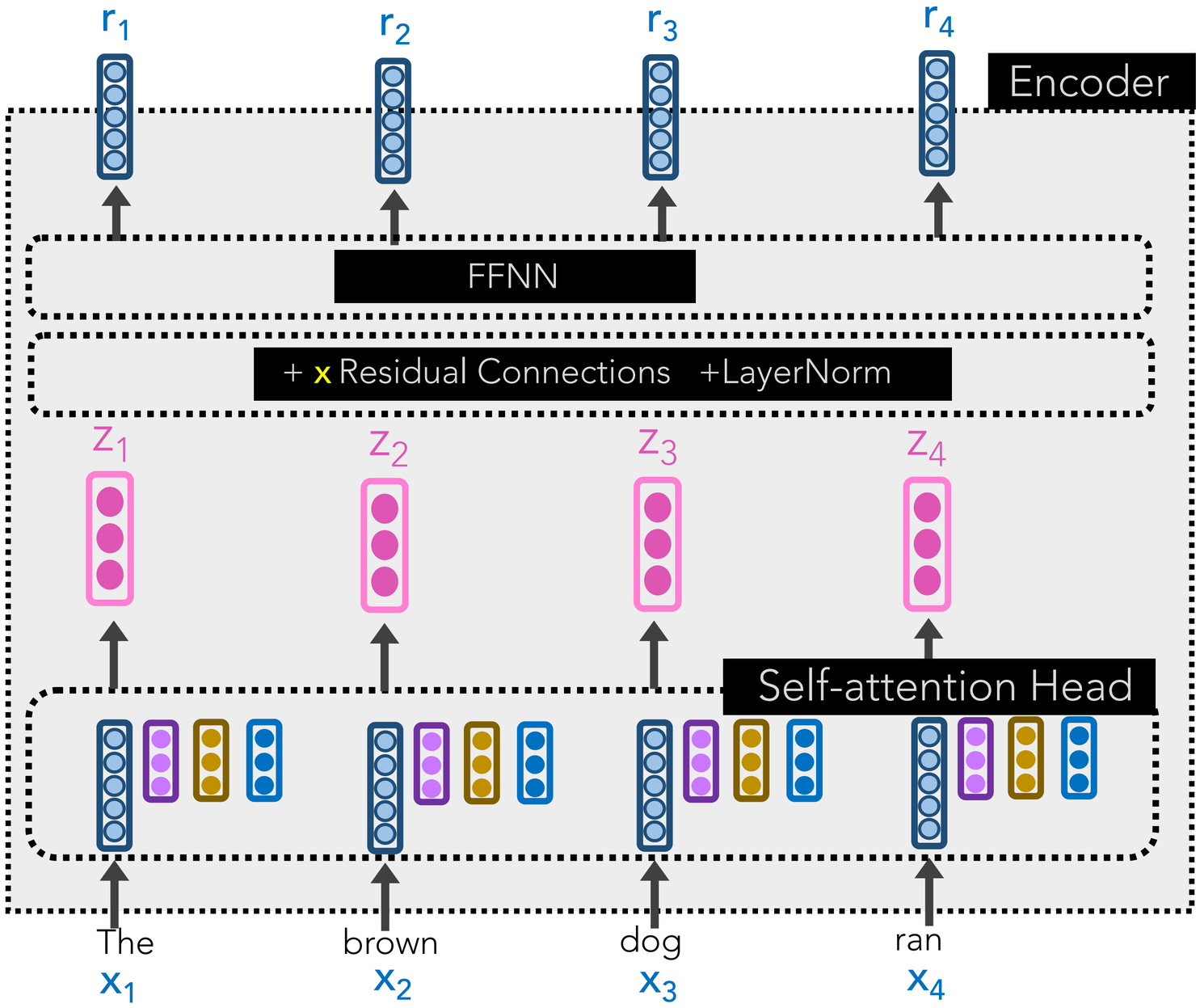
Transformer Architecture



Yay! Our r_i vectors are our new representations, and this entire process is called a Transformer Encoder

Problem: there is no concept of positionality. Words are weighted as if a “bag of words”

Transformer Architecture

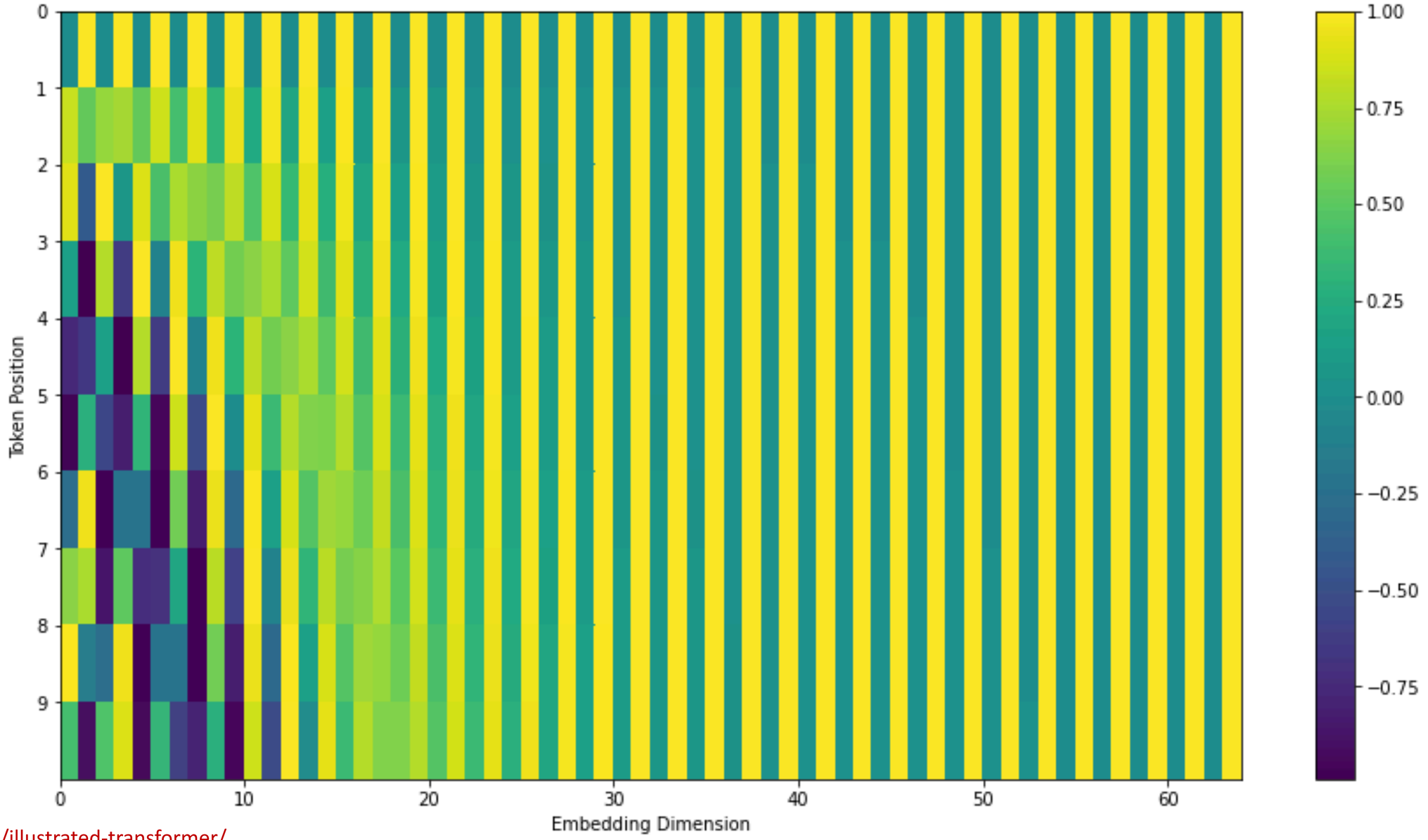


Yay! Our r_i vectors are our new representations, and this entire process is called a Transformer Encoder

Problem: there is no concept of positionality. Words are weighted as if a "bag of words"

Solution: add to each input word x_i a positional encoding $\sim \sin(i)\cos(i)$

Positional Encodings



https://jalammar.github.io/illustrated-transformer/

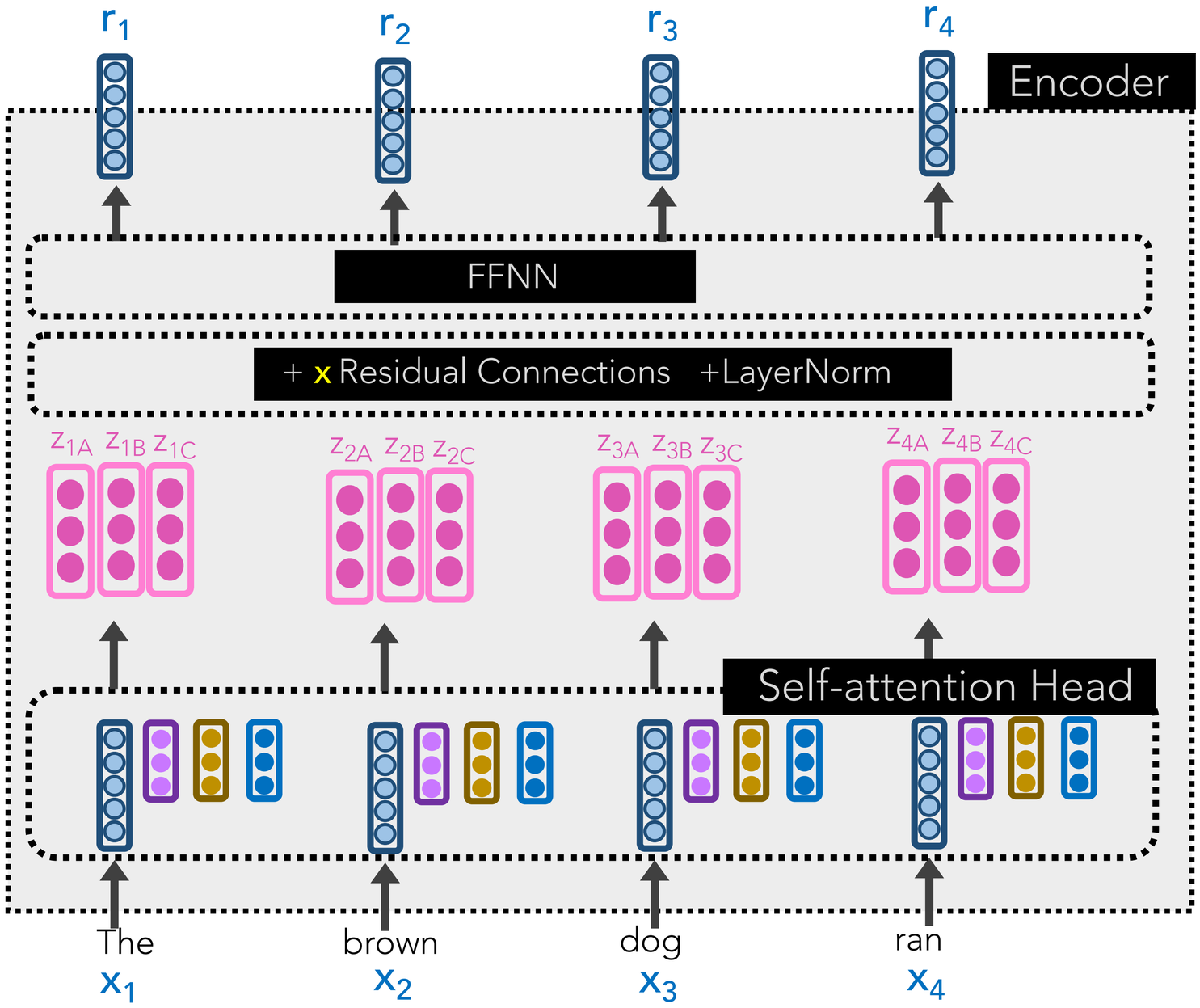
Transformer Architecture

A Self-Attention Head has just one set of query/key/value weight matrices W_q, W_k, W_v

Words can relate in many ways, so it's restrictive to rely on just one Self-Attention Head in the system.

Let's create Multi-headed Self-Attention

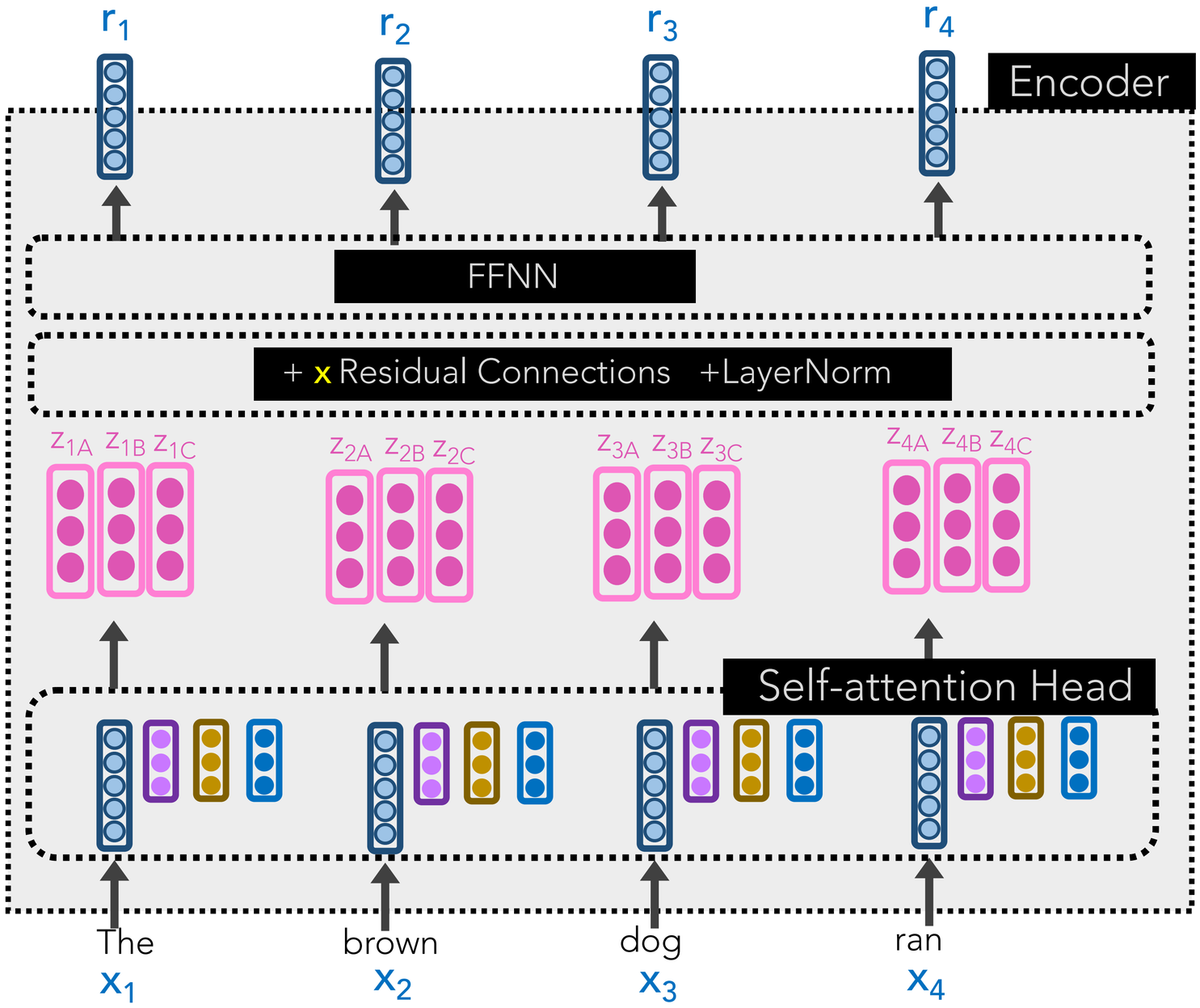
Transformer Architecture



Each Self-Attention Head produces a z_i vector.

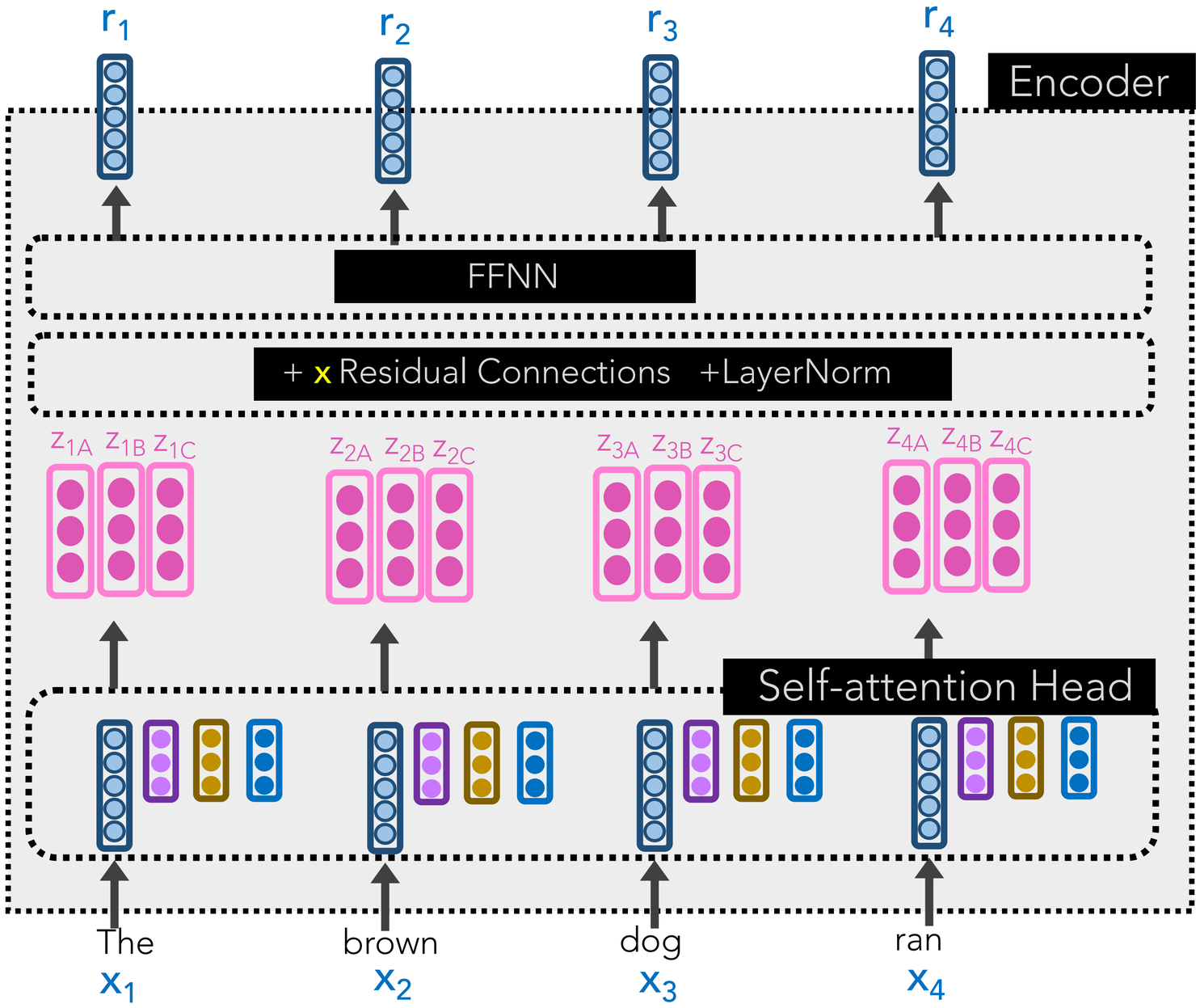
We can, in parallel, use multiple heads and concat the z_i's.

Transformer Architecture

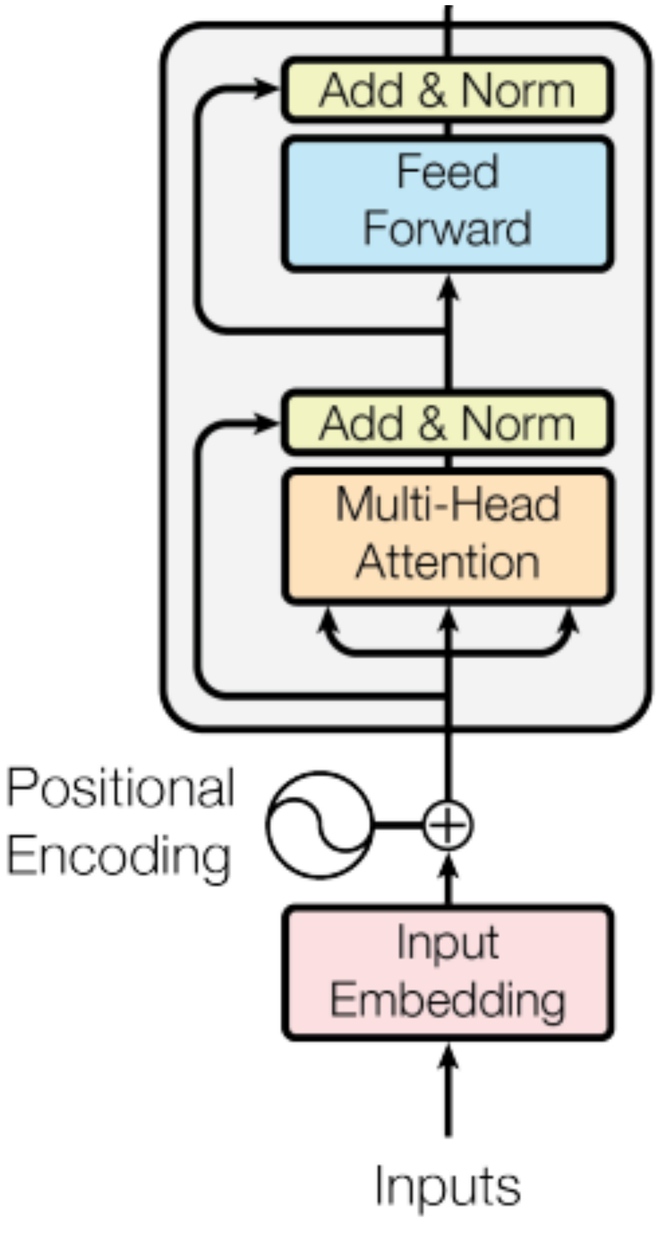


To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding** r_i of each word x_i

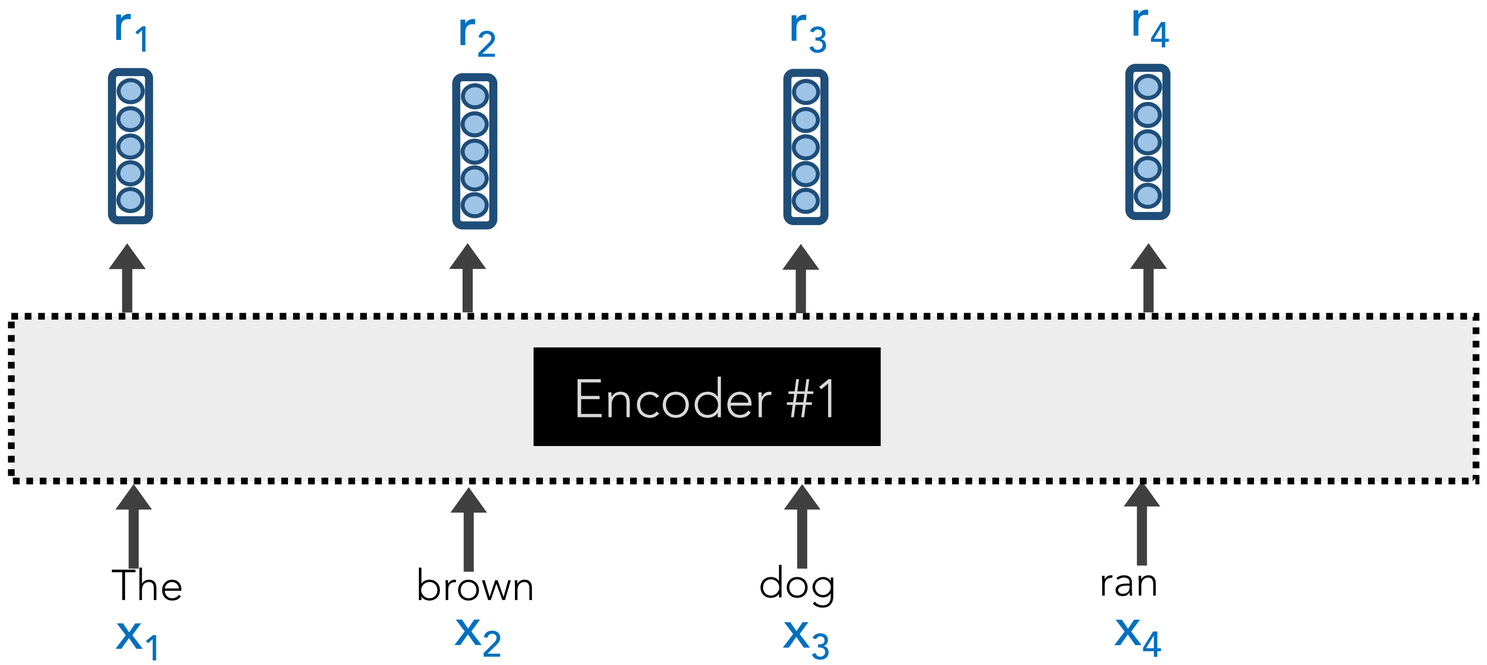
Transformer Architecture



=

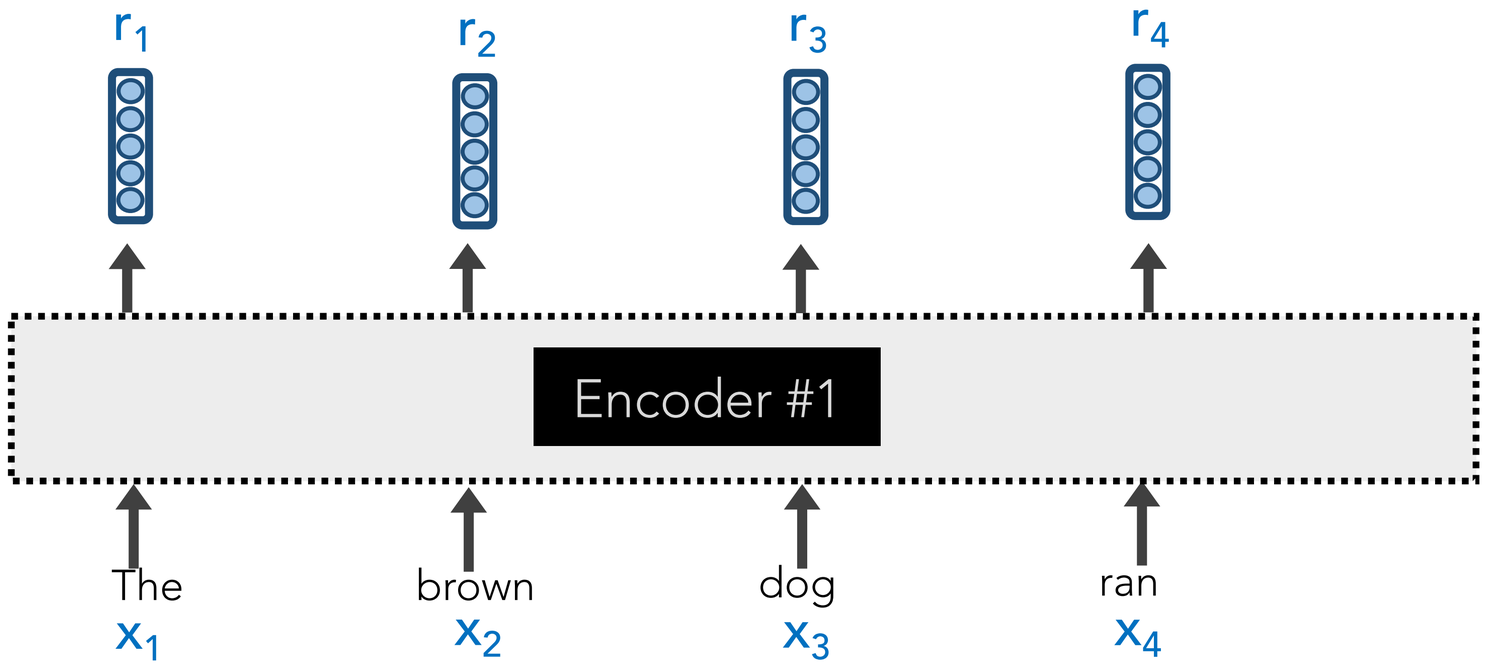


Transformer Architecture



To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding** r_i of each word x_i

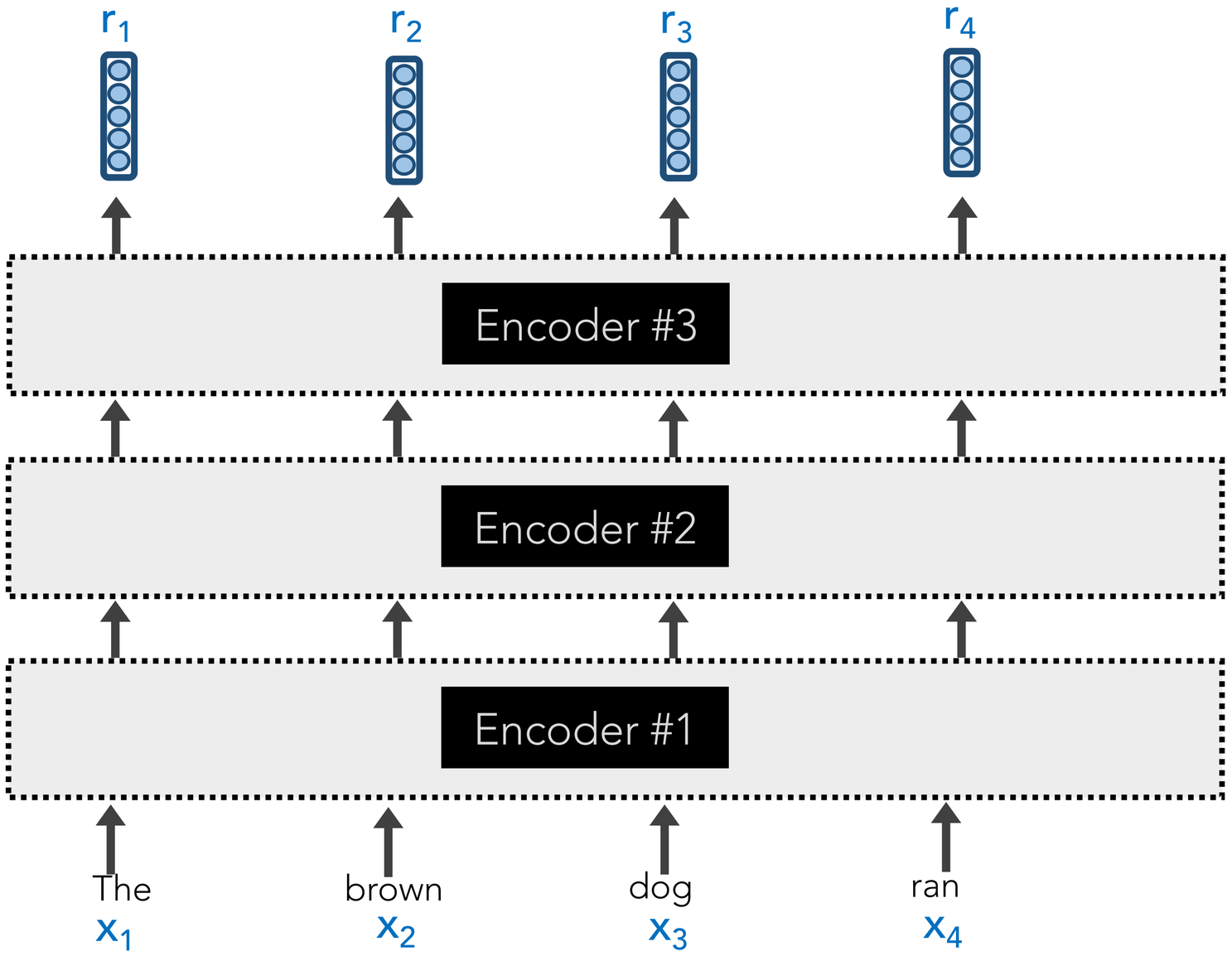
Transformer Architecture



To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding** r_i of each word x_i

Why stop with just 1 Transformer Encoder? We could stack several!

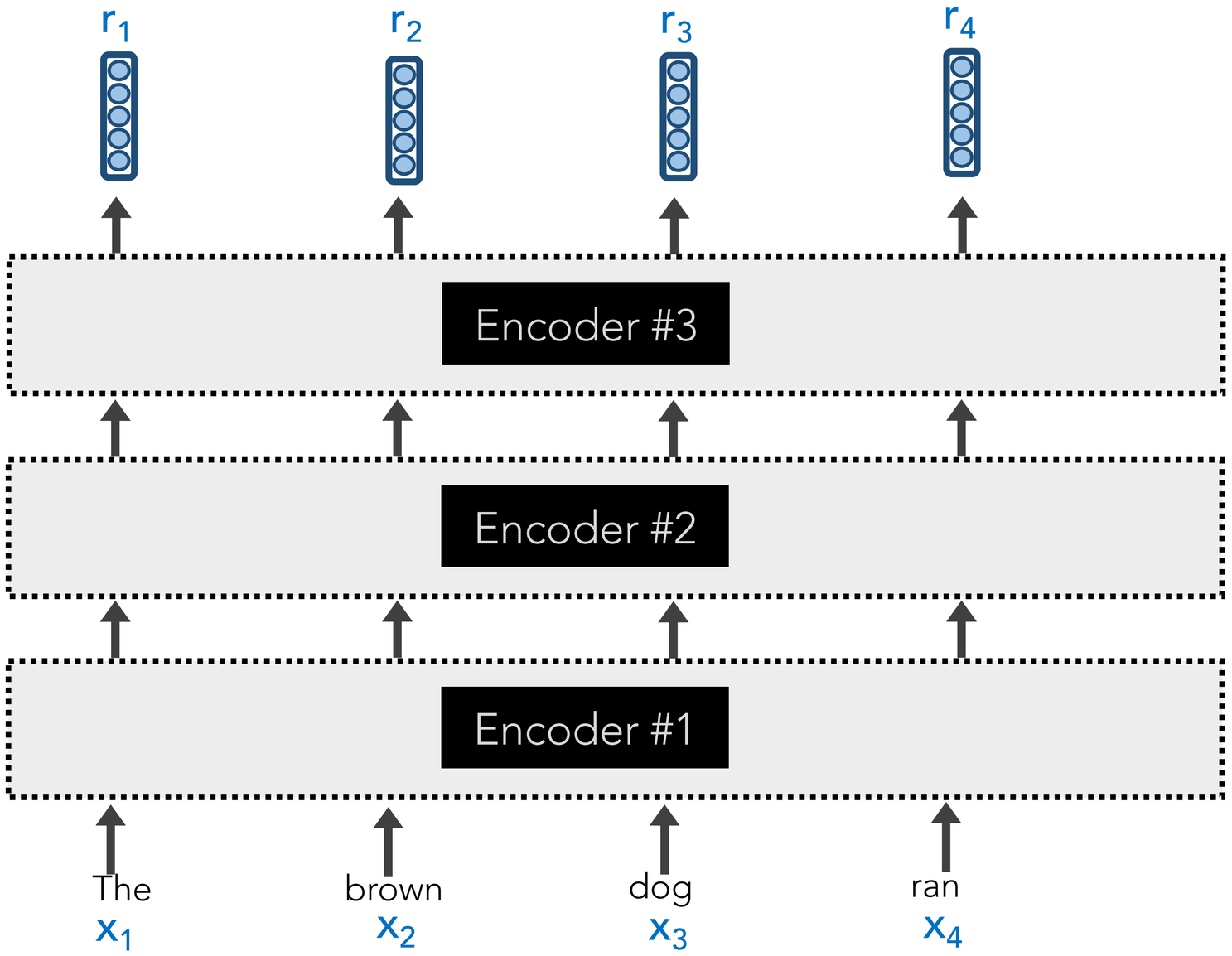
Transformer Architecture



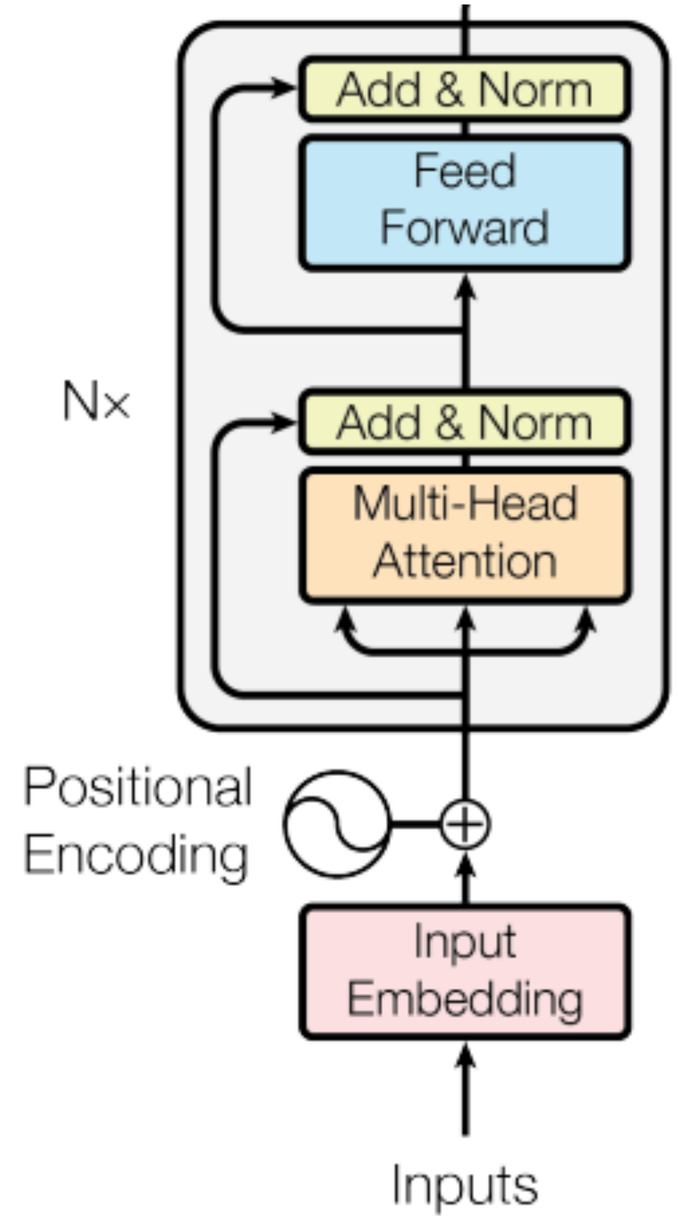
To recap: all of this looks fancy, but ultimately it's just producing a very good **contextualized embedding** r_i of each word x_i

Why stop with just 1 Transformer Encoder? We could stack several!

Transformer Architecture



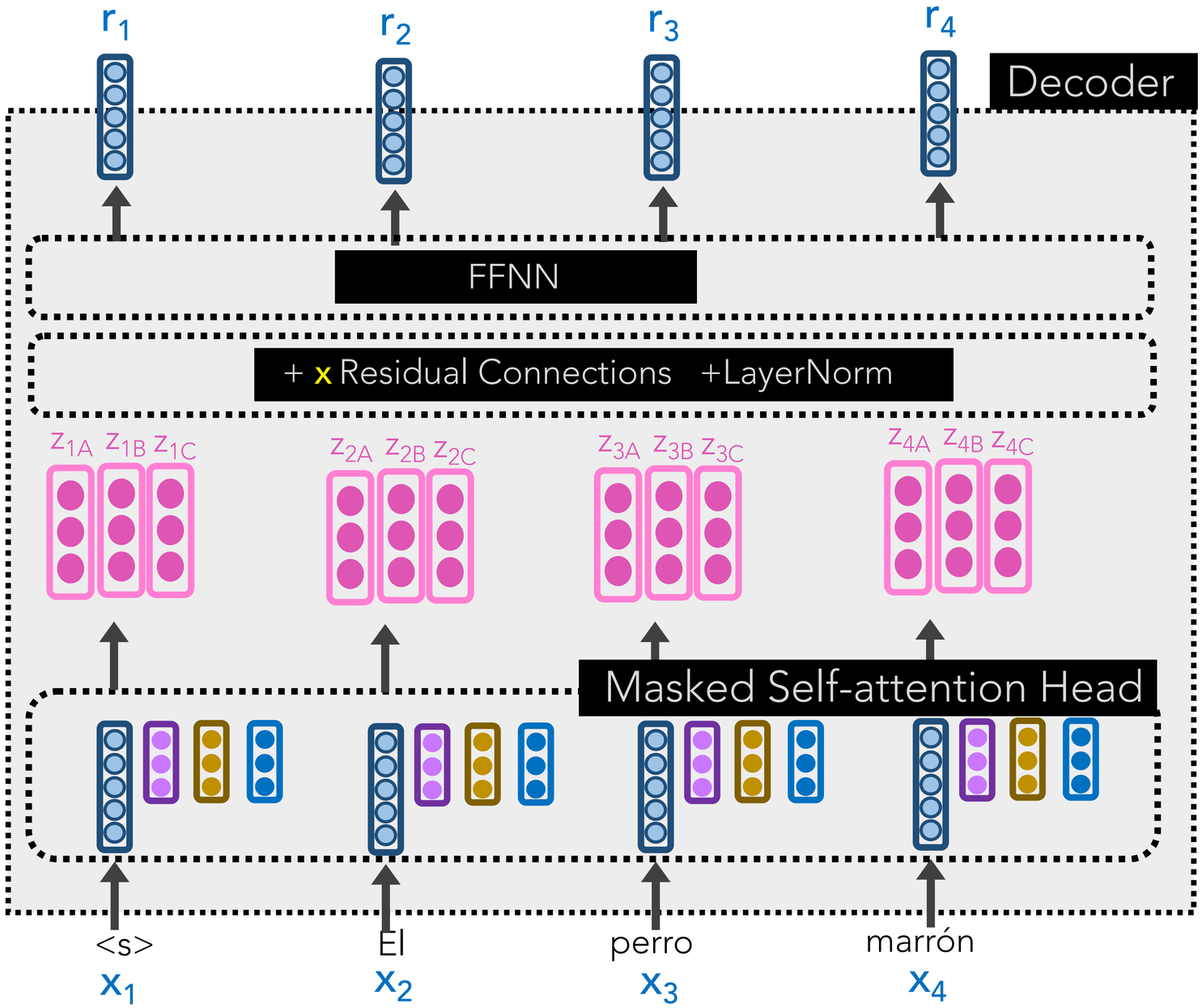
=



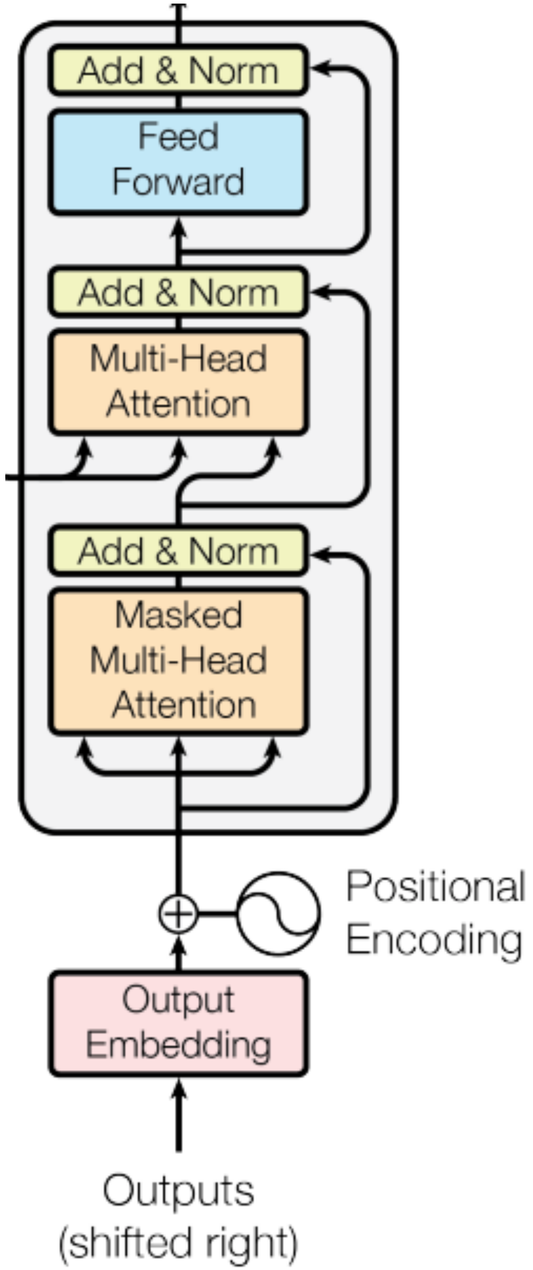
Transformer Decoders

The original Transformer model was intended for Machine Translation, so it had Decoders, too

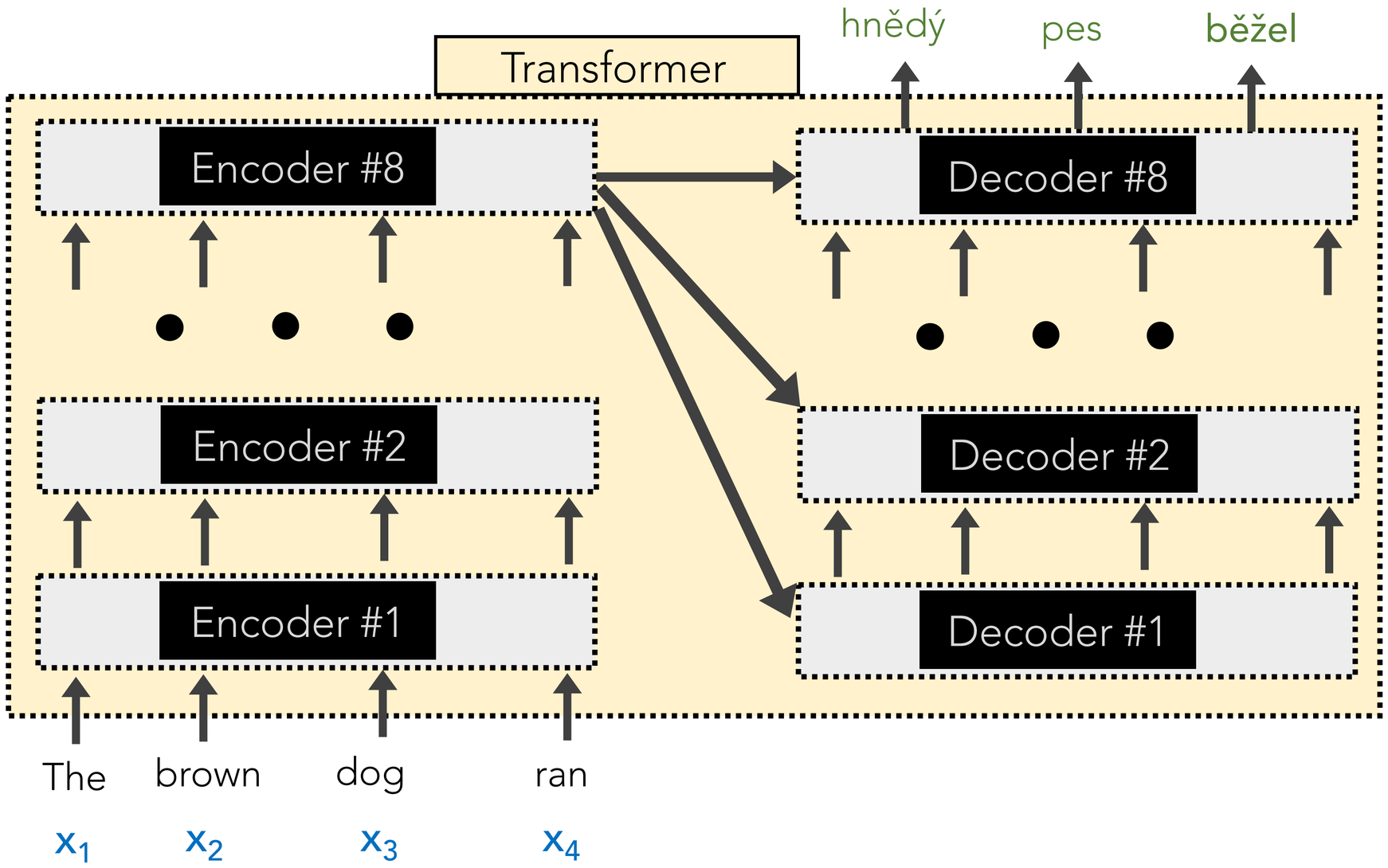
Transformer Decoders



=



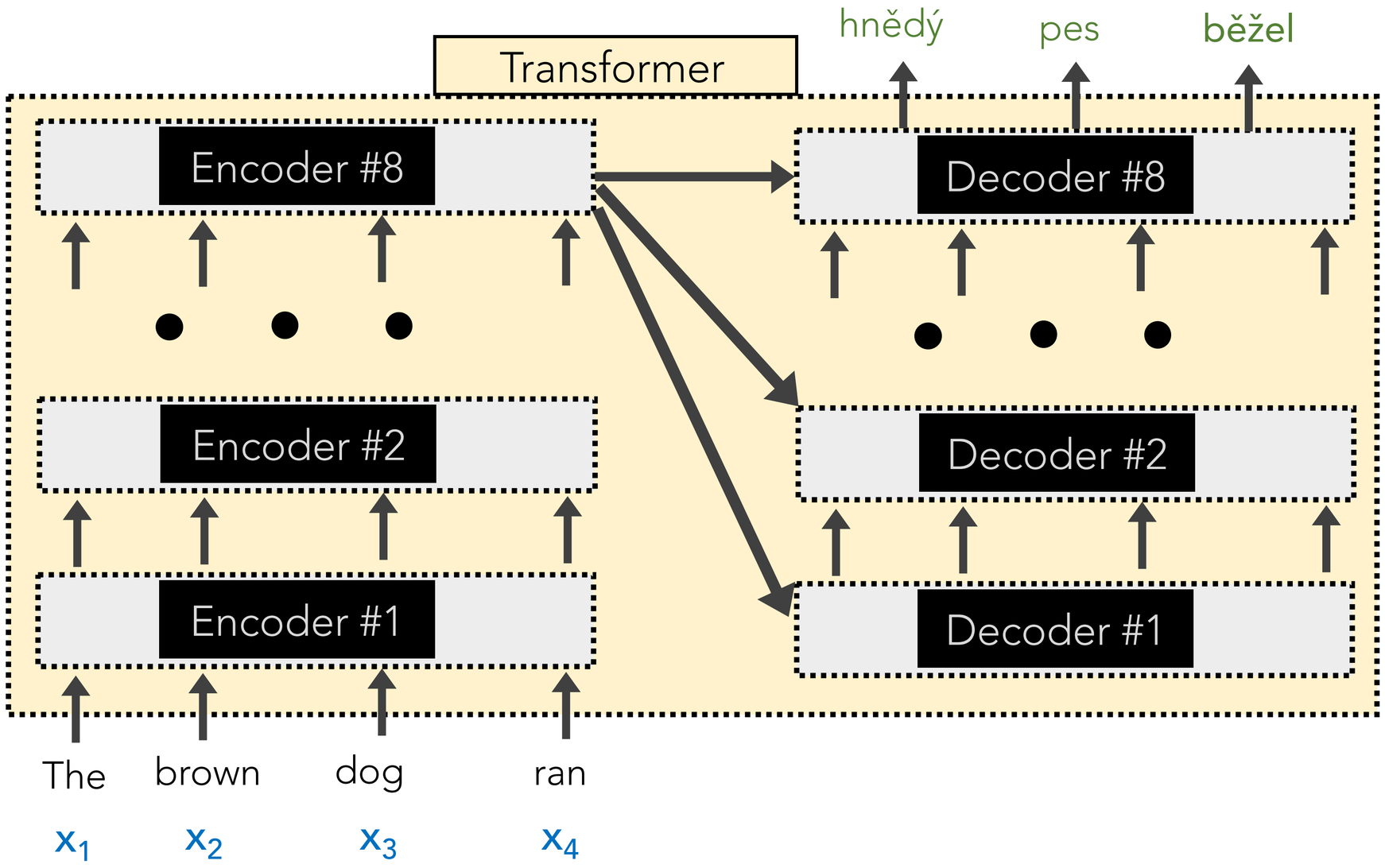
Transformer Decoders



Transformer Encoders produce **contextualized embeddings** of each word

Transformer Decoders generate new sequences of text

Transformer Decoders

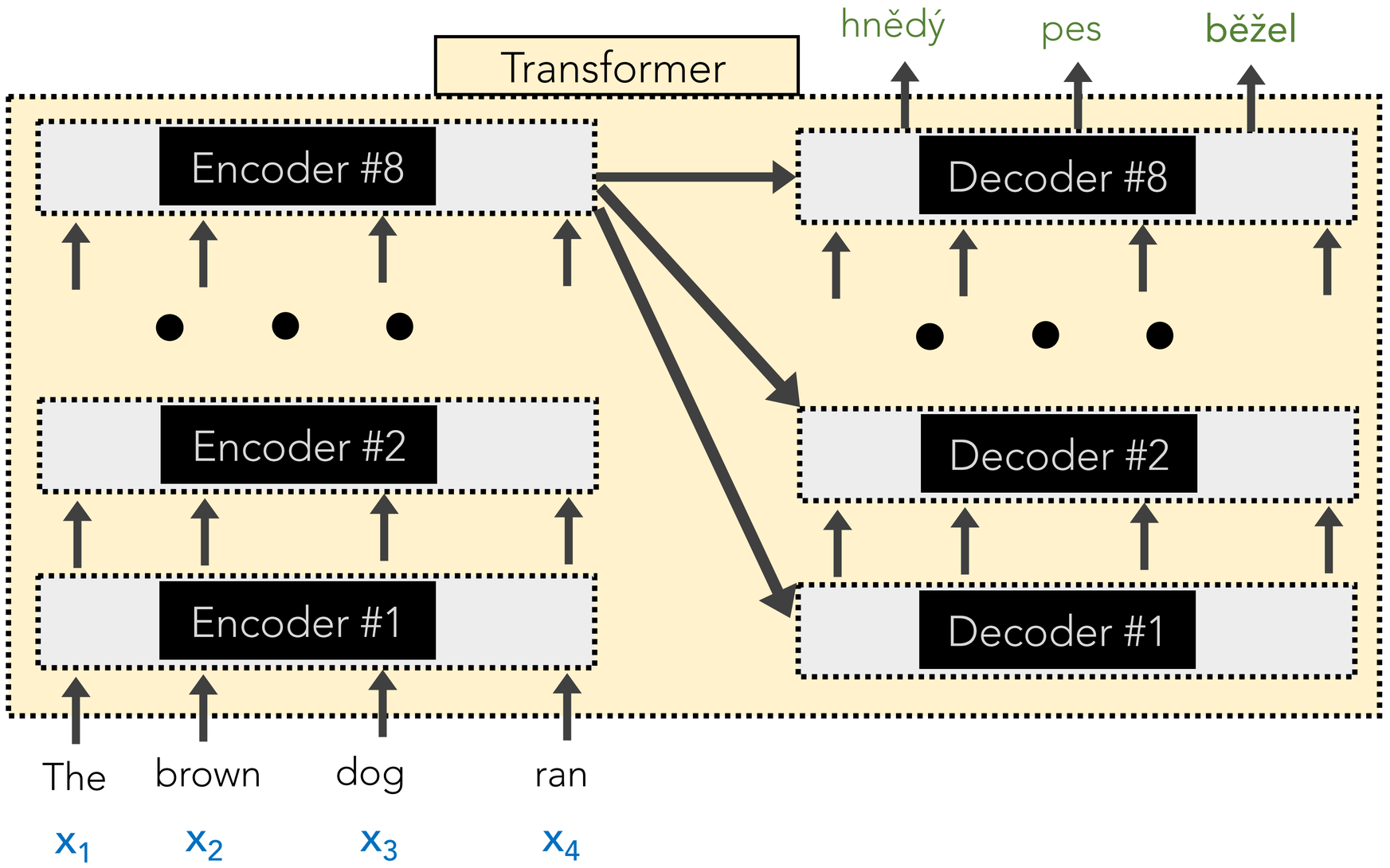


NOTE

Transformer Decoders are identical to the Encoders, except they have an additional Attention Head in between the Self-Attention and FFNN layers.

This additional Attention Head **focuses on parts of the encoder's representations.**

Transformer Decoders

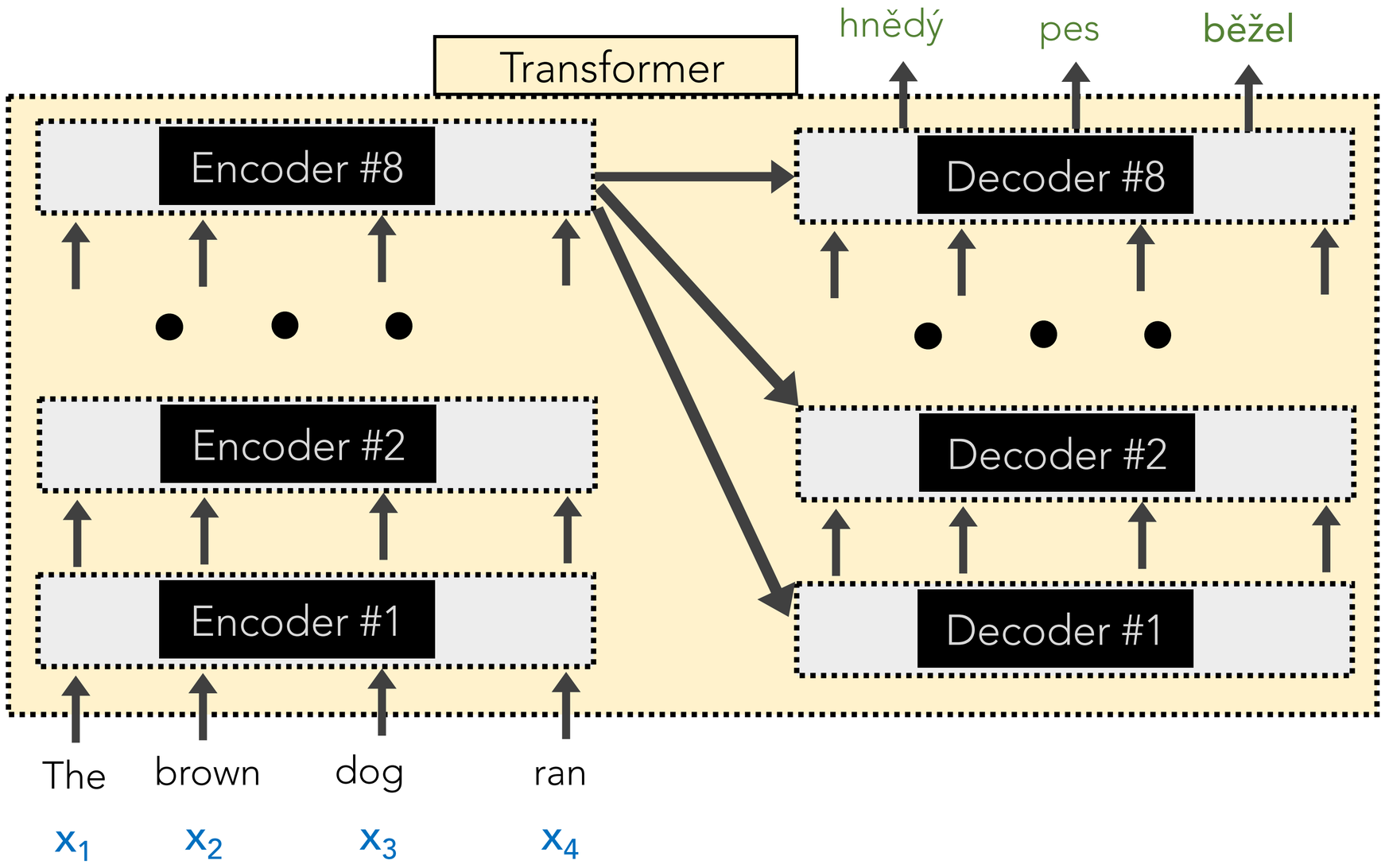


NOTE

The **query** vector for a Transformer Decoder's **Attention Head** (not Self-Attention Head) is from the output of the previous decoder layer.

However, the **key** and **value** vectors are from the Transformer Encoders' outputs.

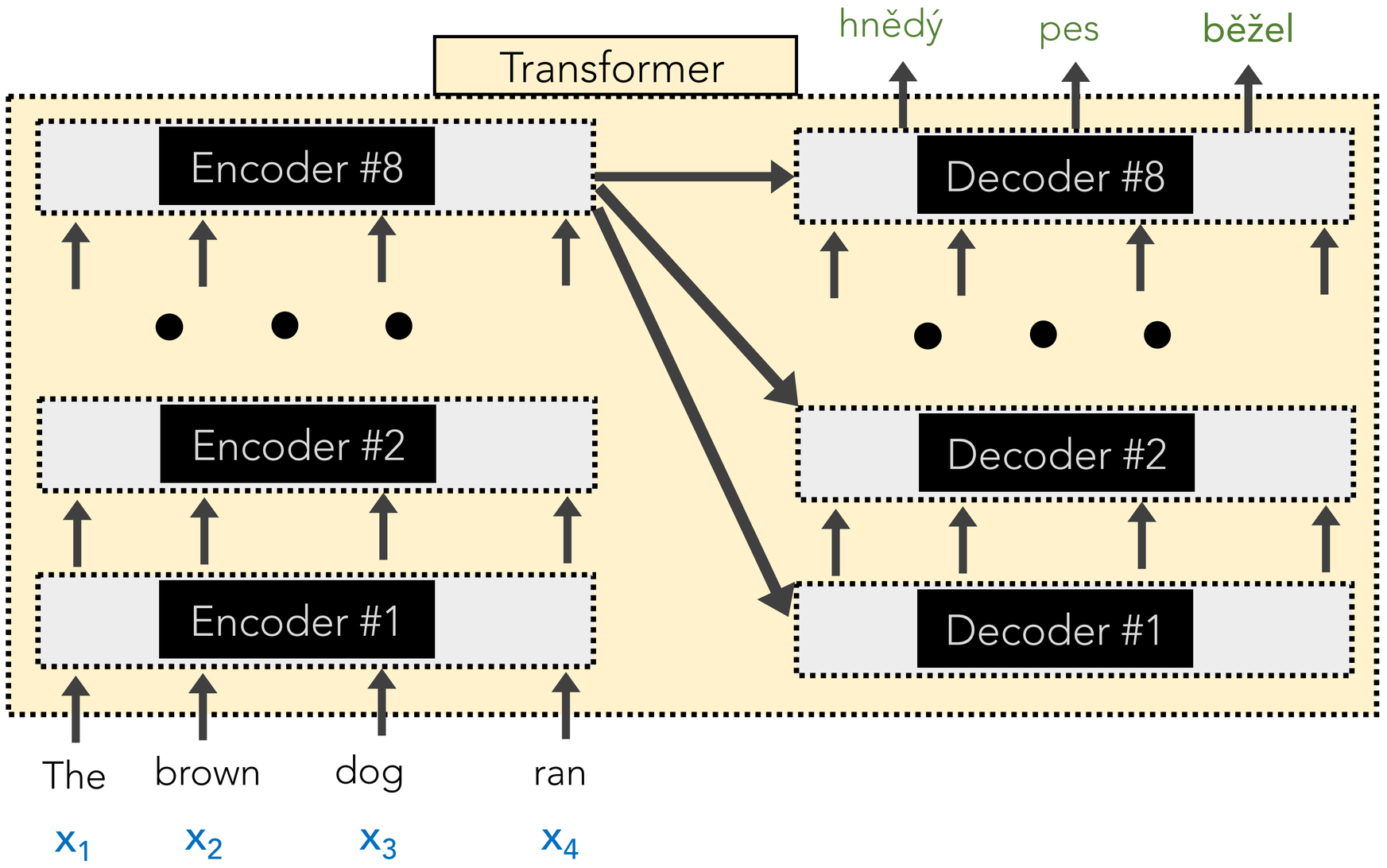
Transformer Decoders



NOTE

The **query, key,** and **value** vectors for a Transformer Decoder's **Self-Attention Head** (not Attention Head) are all from the output of the previous decoder layer.

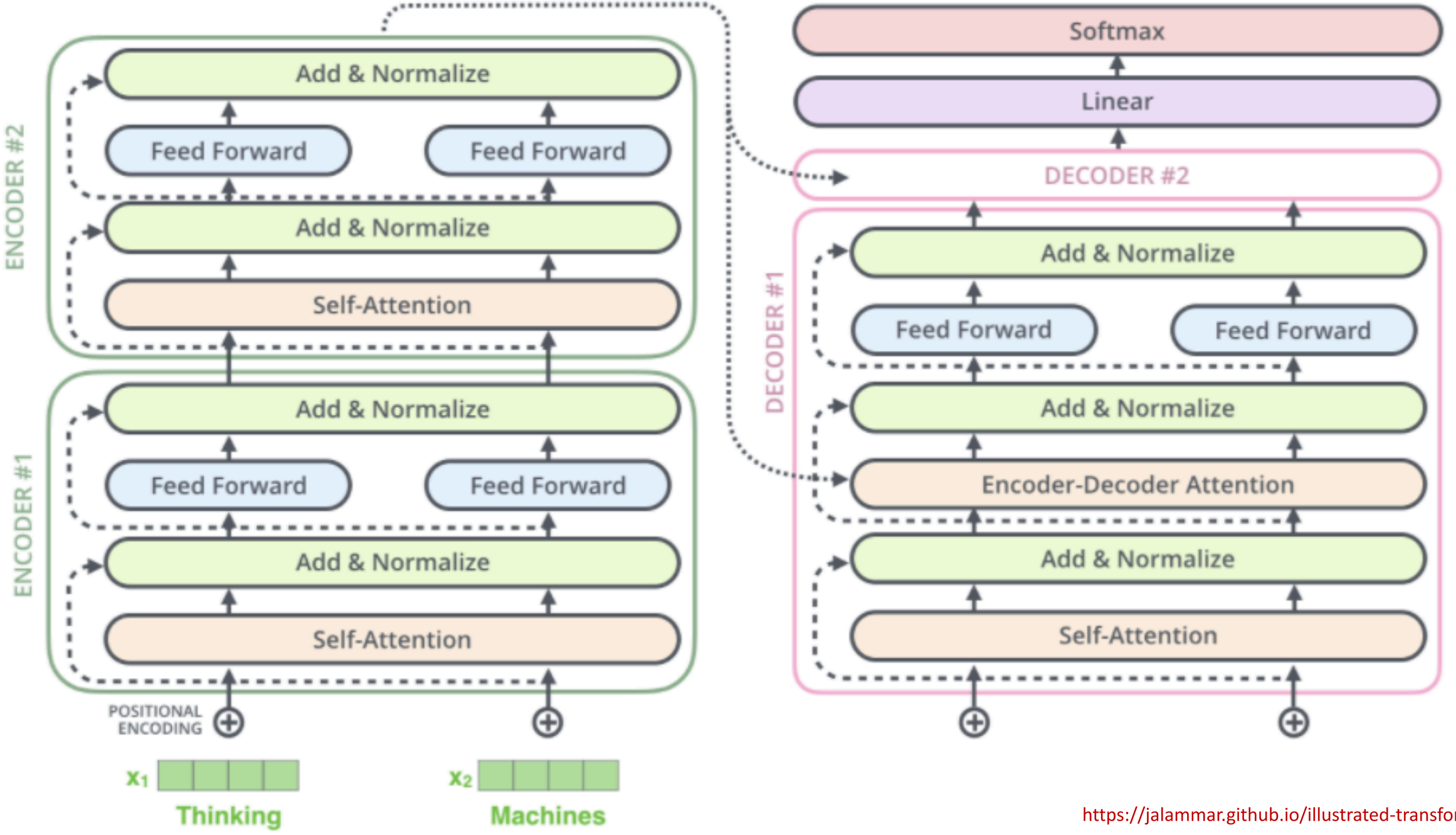
Transformer Decoders



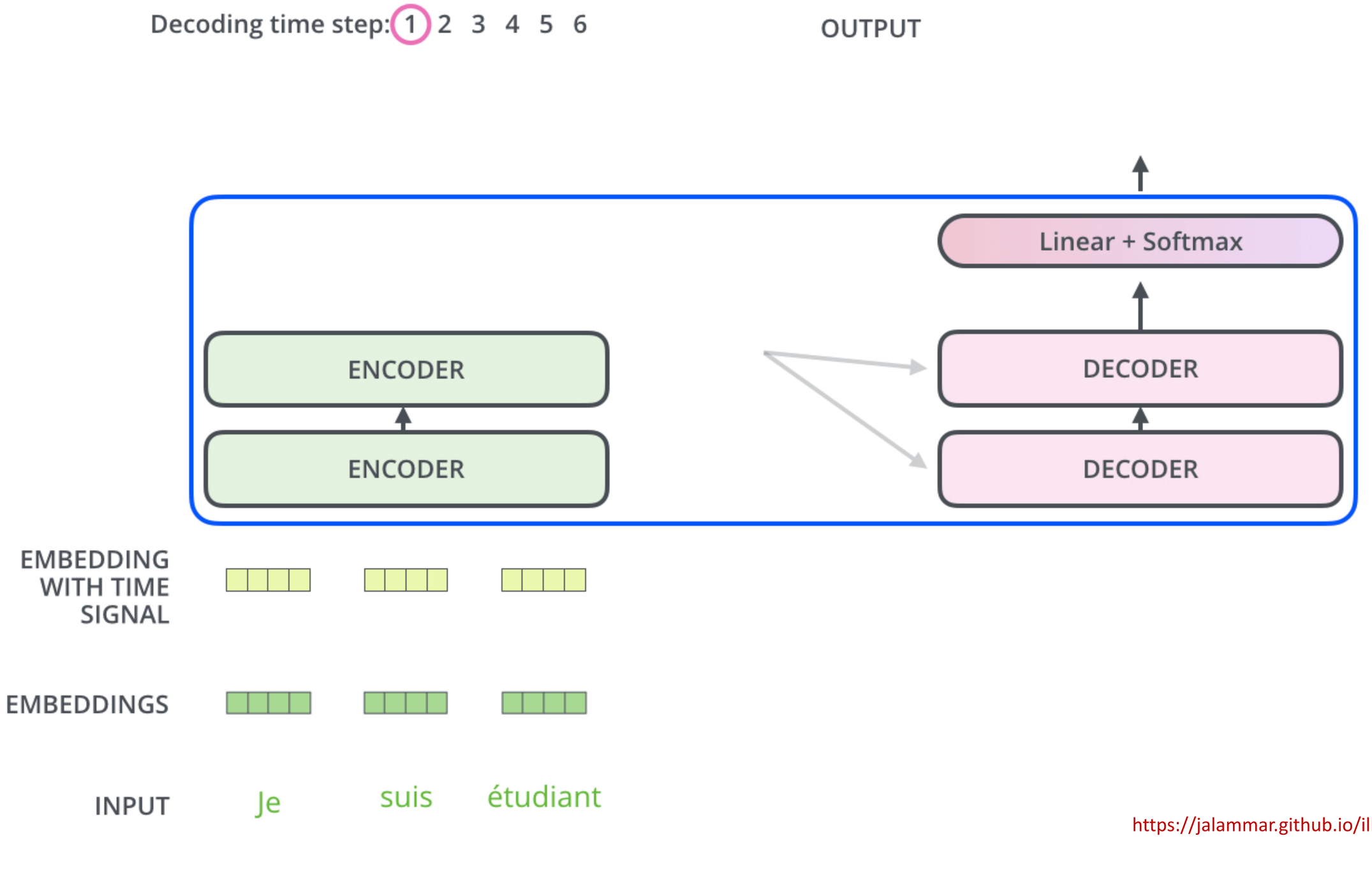
IMPORTANT

The Transformer Decoders have positional embeddings, too, just like the Encoders.

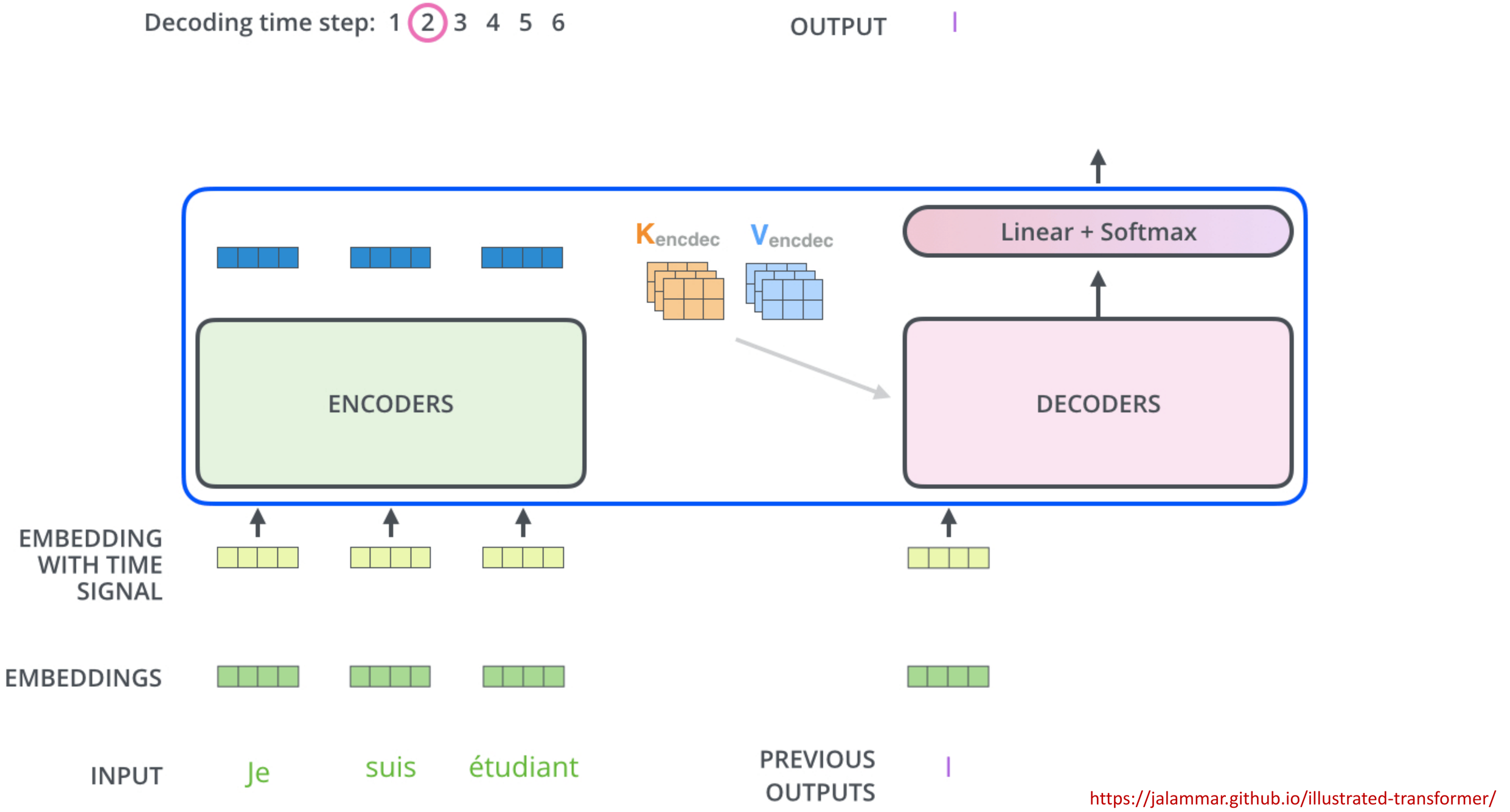
Critically, each position is only allowed to attend to the previous indices. This masked Attention preserves it as being an auto-regressive LM.



https://jalammar.github.io/illustrated-transformer/



https://jalammar.github.io/illustrated-transformer/



https://jalammar.github.io/illustrated-transformer/

Transformer Architecture

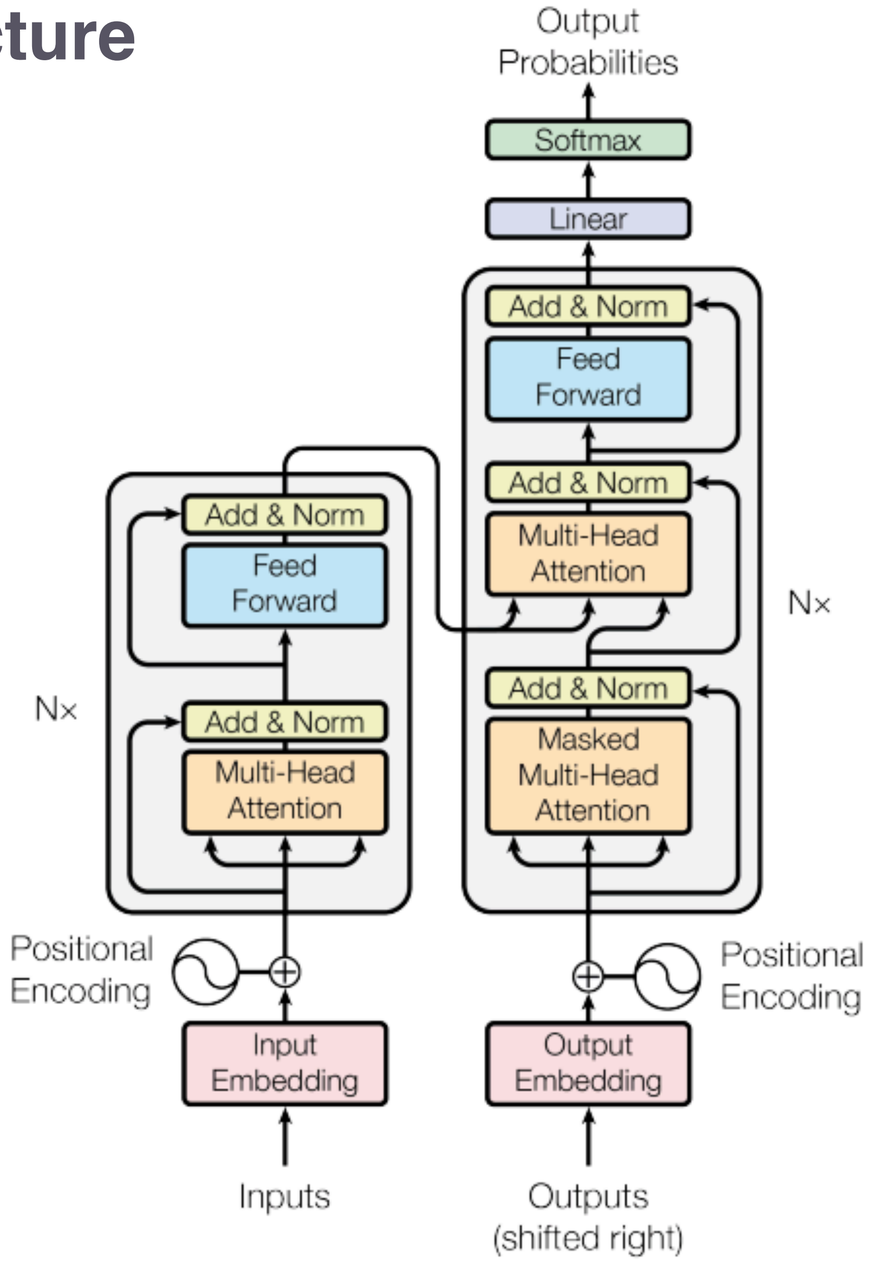


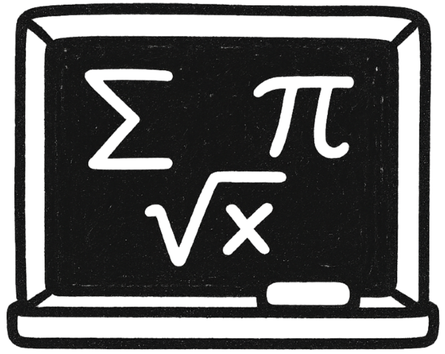
Figure 1: The Transformer - model architecture.

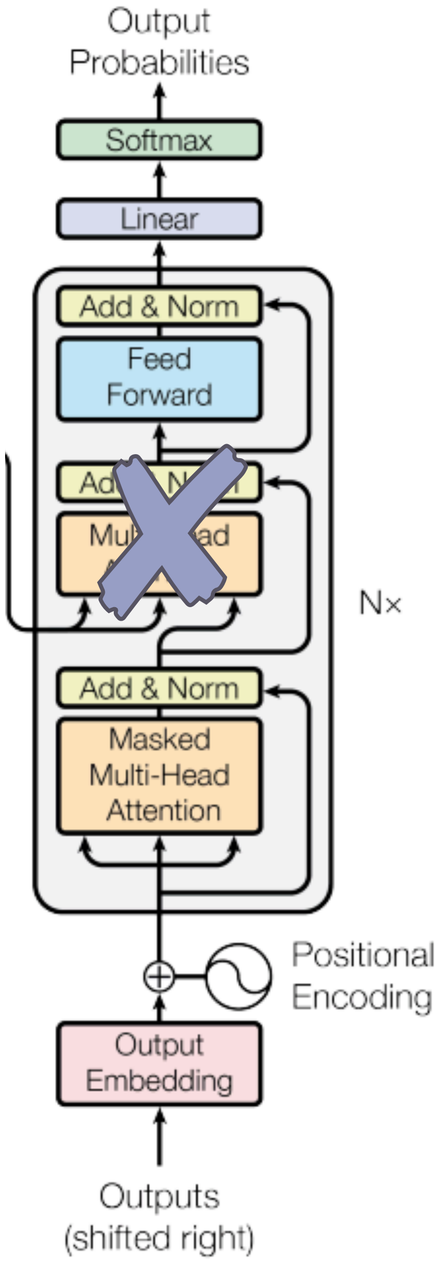
Attention is All you Need (2017) https://arxiv.org/pdf/1706.03762.pdf

Transformer Architecture: Decoder-only

Transformer Decoders
generate new sequences of text

When used as a standalone model, they only use self-attention (no cross-attention)





Attention is All you Need (2017) https://arxiv.org/pdf/1706.03762.pdf

What just happened!?

- Attention
- Self-Attention
- Transformers

What is to come…

- NO practical session on **May 1st**
- Next lecture: more transformers! **May 6th**
- **HW1 released! Due on May 13th, via upload on Moodle**