Lecture 10: Transformers

From Self-Attention to Transformers

Harvard

AC295/CS287r/CSCI E-115B

Chris Tanner


WHERE ARE THE BANANAS BERT?
I'M ON THE PHONE ERN.

ANNOUNCEMENTS

- HW1 is graded (few remaining). Solutions are posted on Canvas -> Files
- HW2 is due tonight @ 11:59pm!
- HW3 will be released tonight @ 11:59pm! The shortest assignment yet.
- Candidate Research Projects have been announced.
  - Read them on `Research Brainstorming` spreadsheet.
  - Indicate your preferences on the Google Form (see Ed post) by Wed 11:59pm
  - Tonight @ 8pm, Zoom will be open for anyone who wishes to discuss projects

RESEARCH PROJECTS

- **Phase II** is due Oct 14 @ 11:59pm. See website for full expectations.
- Abstract + Related Works + Introduction (this will improve over time).

Outline

- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

Outline

- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

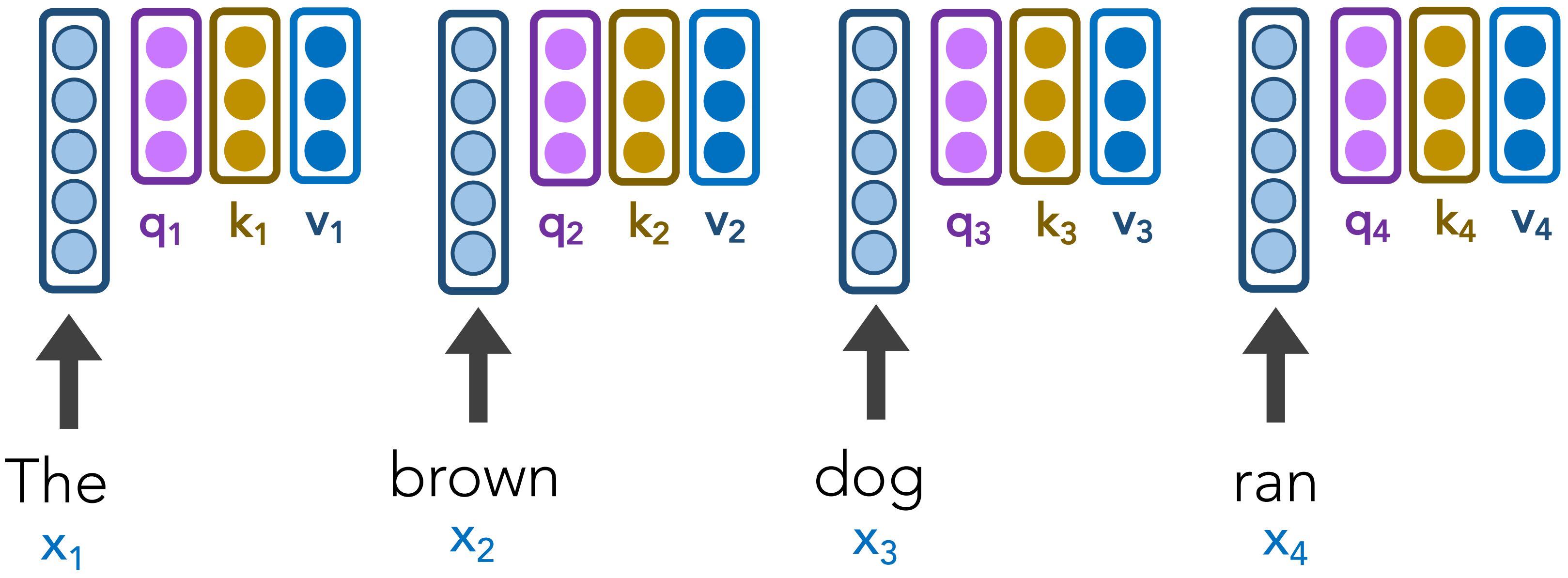
Self-Attention

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:

$$q_i = w_q x_i$$

$$k_i = w_k x_i$$

$$v_i = w_v x_i$$



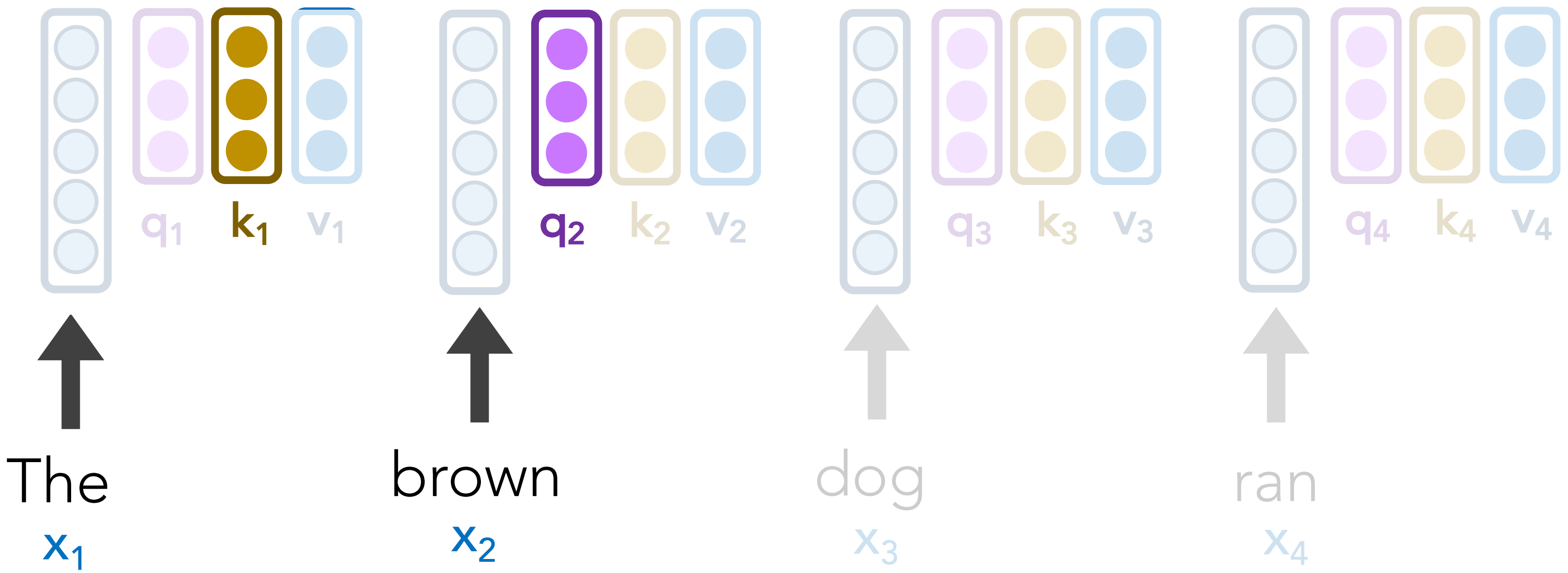
Under the hood, each x_i has 3 small, associated vectors. For example, x_1 has:

- Query $\mathbf{q_1}$
- Key $\mathbf{k_1}$
- Value $\mathbf{v_1}$

Self-Attention

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

$s_1 = q_2 \cdot k_1 = 92$

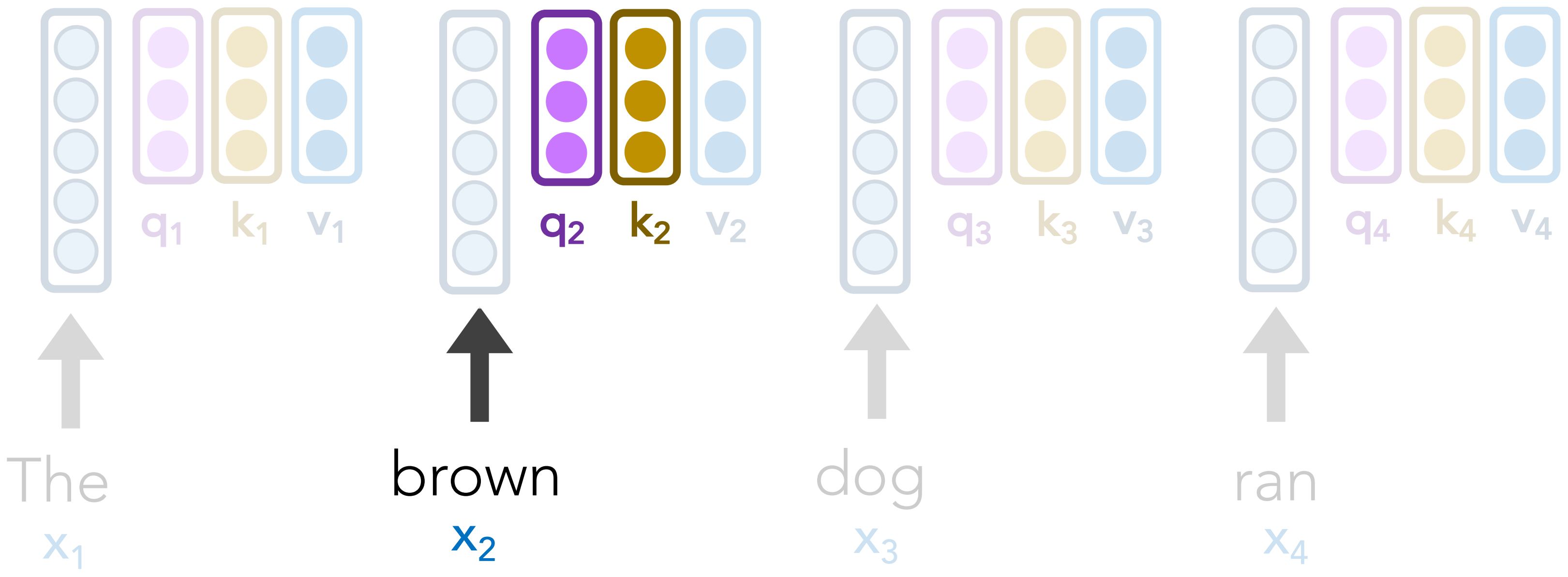


Self-Attention

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

$s_2 = q_2 \cdot k_2 = 124$

$s_1 = q_2 \cdot k_1 = 92$



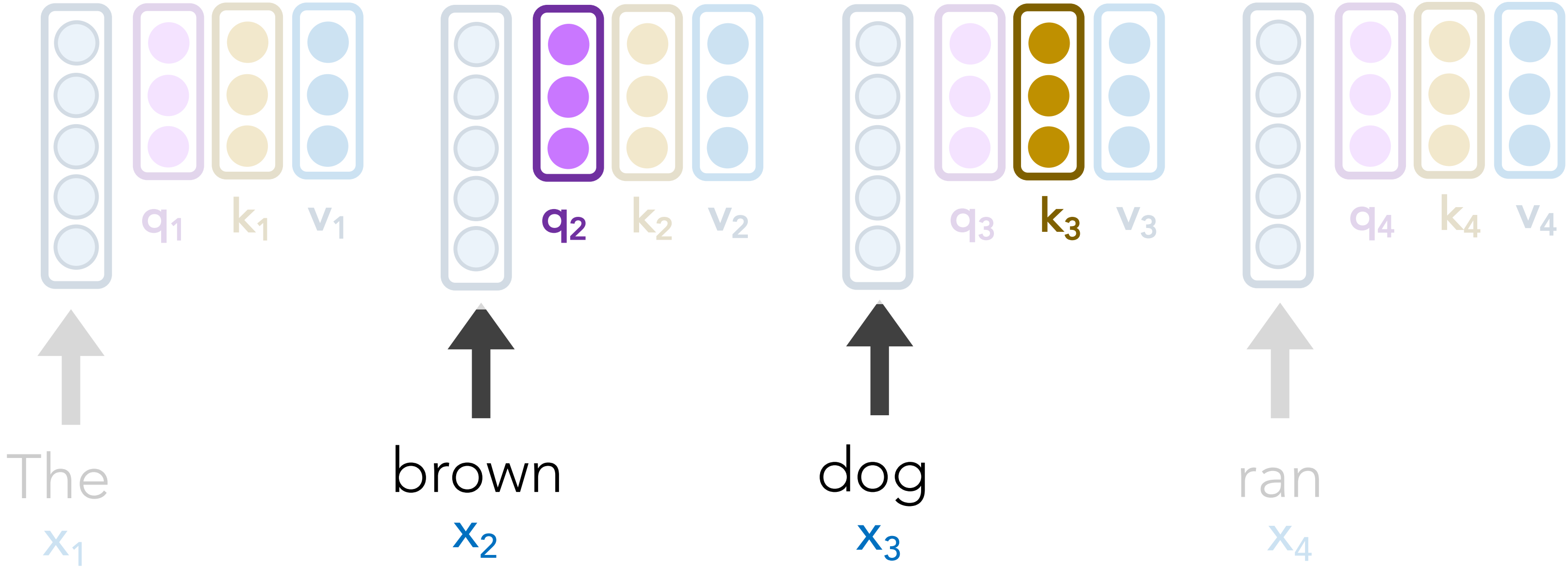
Self-Attention

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

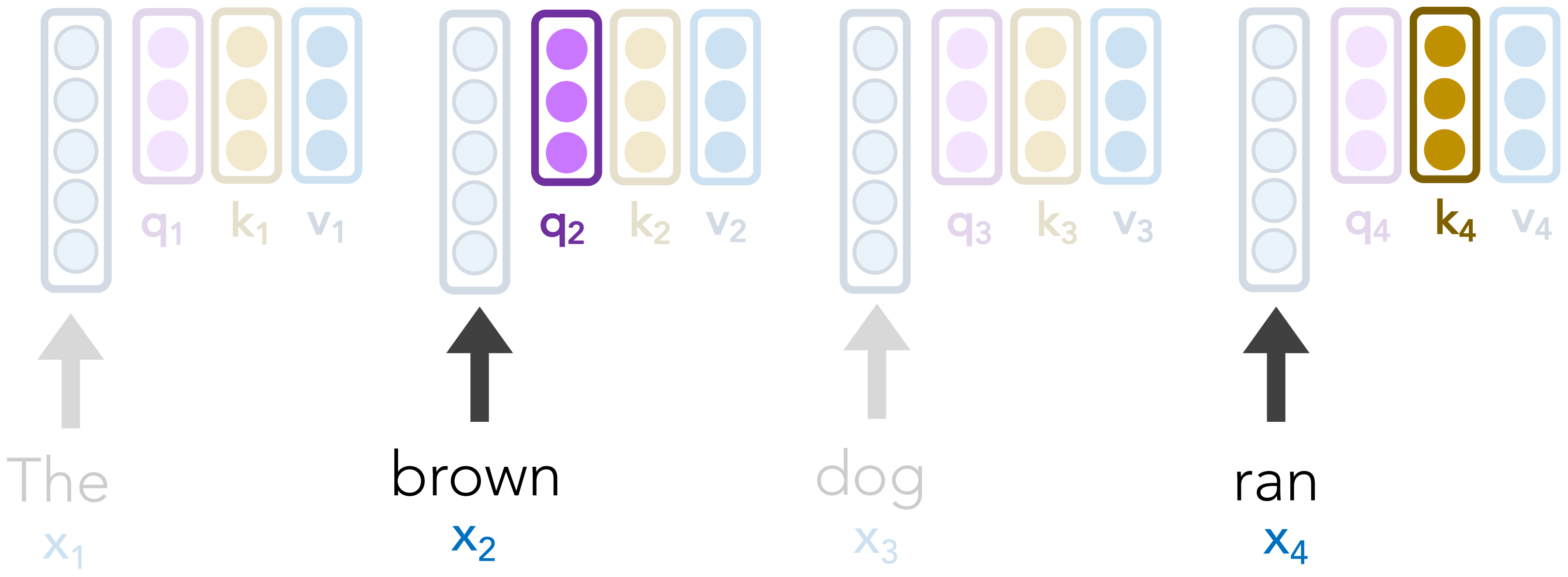
Step 2: For word x_2, let's calculate the scores s_1, s_2, s_3, s_4, which represent how much attention to pay to each respective "word" v_i

$s_4 = q_2 \cdot k_4 = 8$

$s_3 = q_2 \cdot k_3 = 22$

$s_2 = q_2 \cdot k_2 = 124$

$s_1 = q_2 \cdot k_1 = 92$



Self-Attention

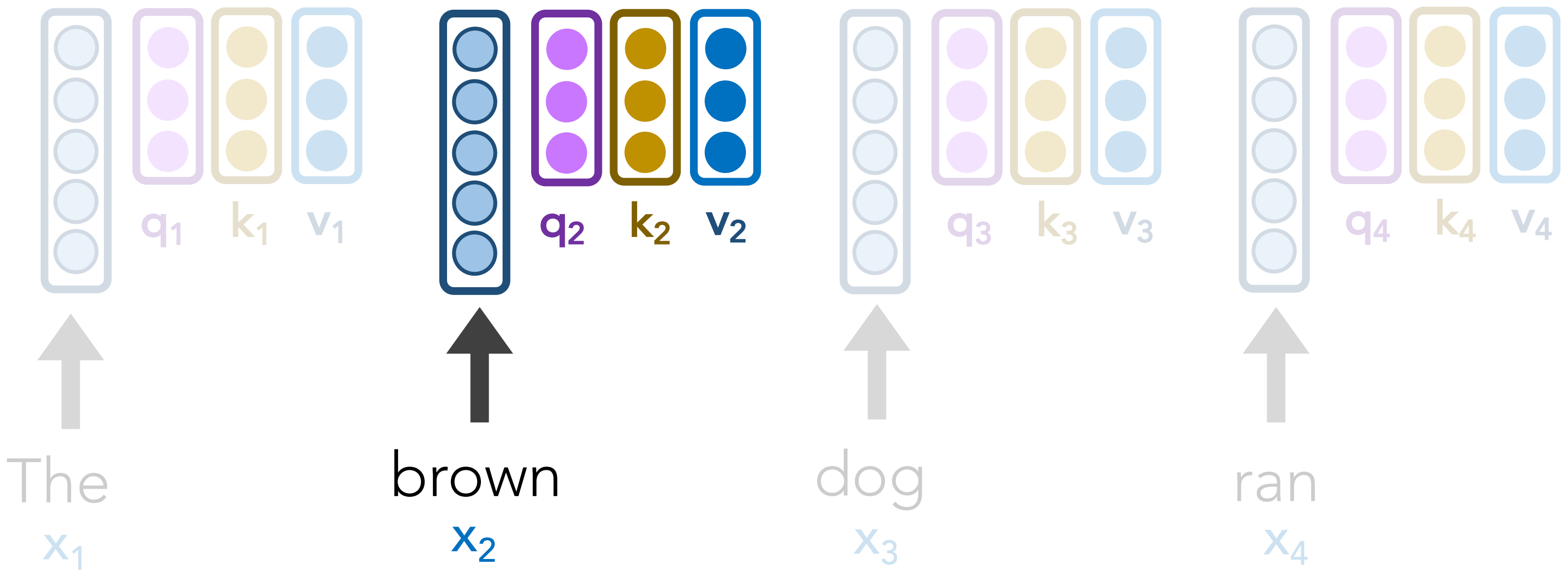
Step 3: Our scores s_1, s_2, s_3, s_4 don't sum to 1. Let's divide by $\sqrt{len(k_i)}$ and softmax it

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

$s_3 = q_2 \cdot k_3 = 22$ $\quad$ $a_3 = \sigma(s_3/8) = .01$

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

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



Self-Attention

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

$s_4 = q_2 \cdot k_4 = 8$

$s_3 = q_2 \cdot k_3 = 22$

$s_2 = q_2 \cdot k_2 = 124$

$s_1 = q_2 \cdot k_1 = 92$

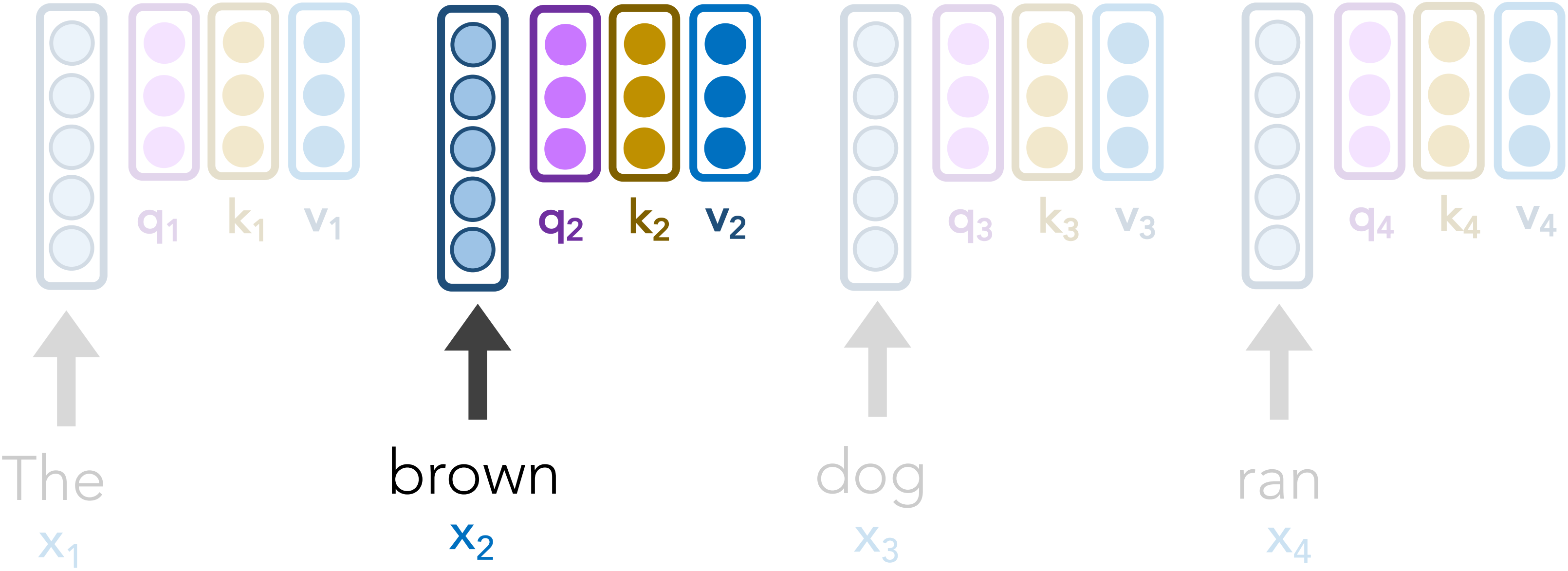
$a_4 = \sigma(s_4/8) = 0$

$a_3 = \sigma(s_3/8) = .01$

$a_2 = \sigma(s_2/8) = .91$

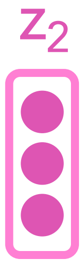
$a_1 = \sigma(s_1/8) = .08$

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



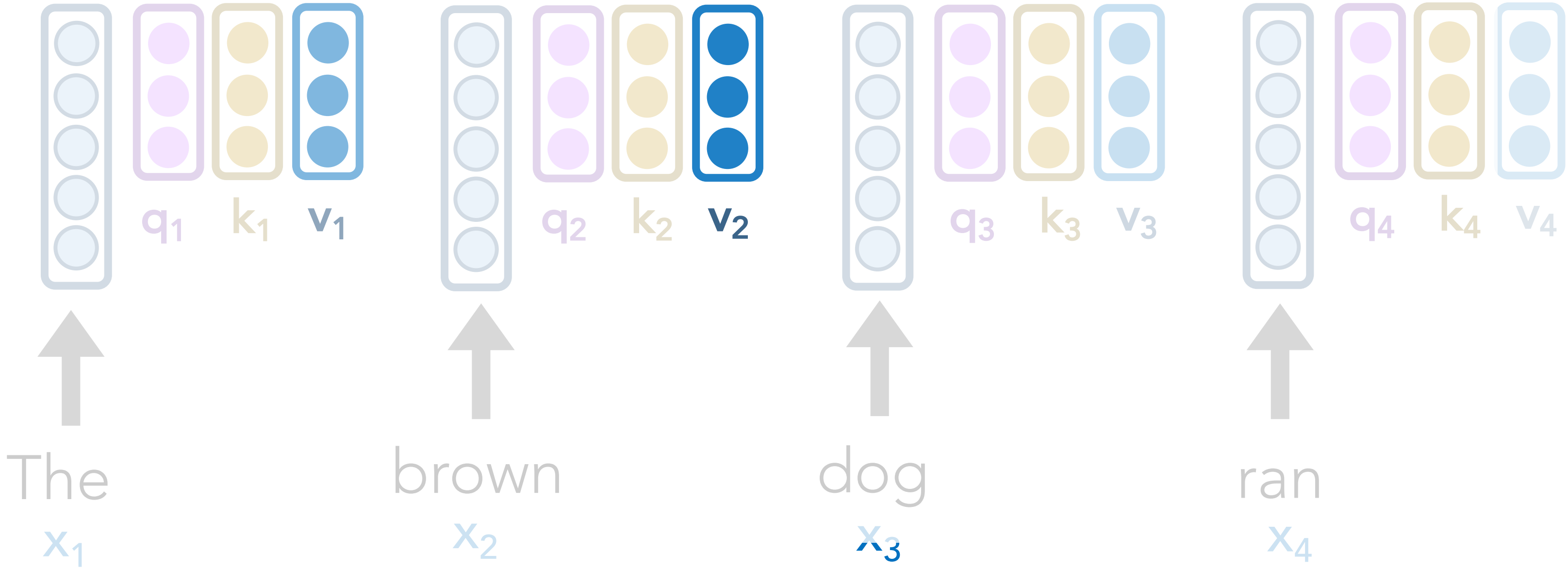
Self-Attention

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



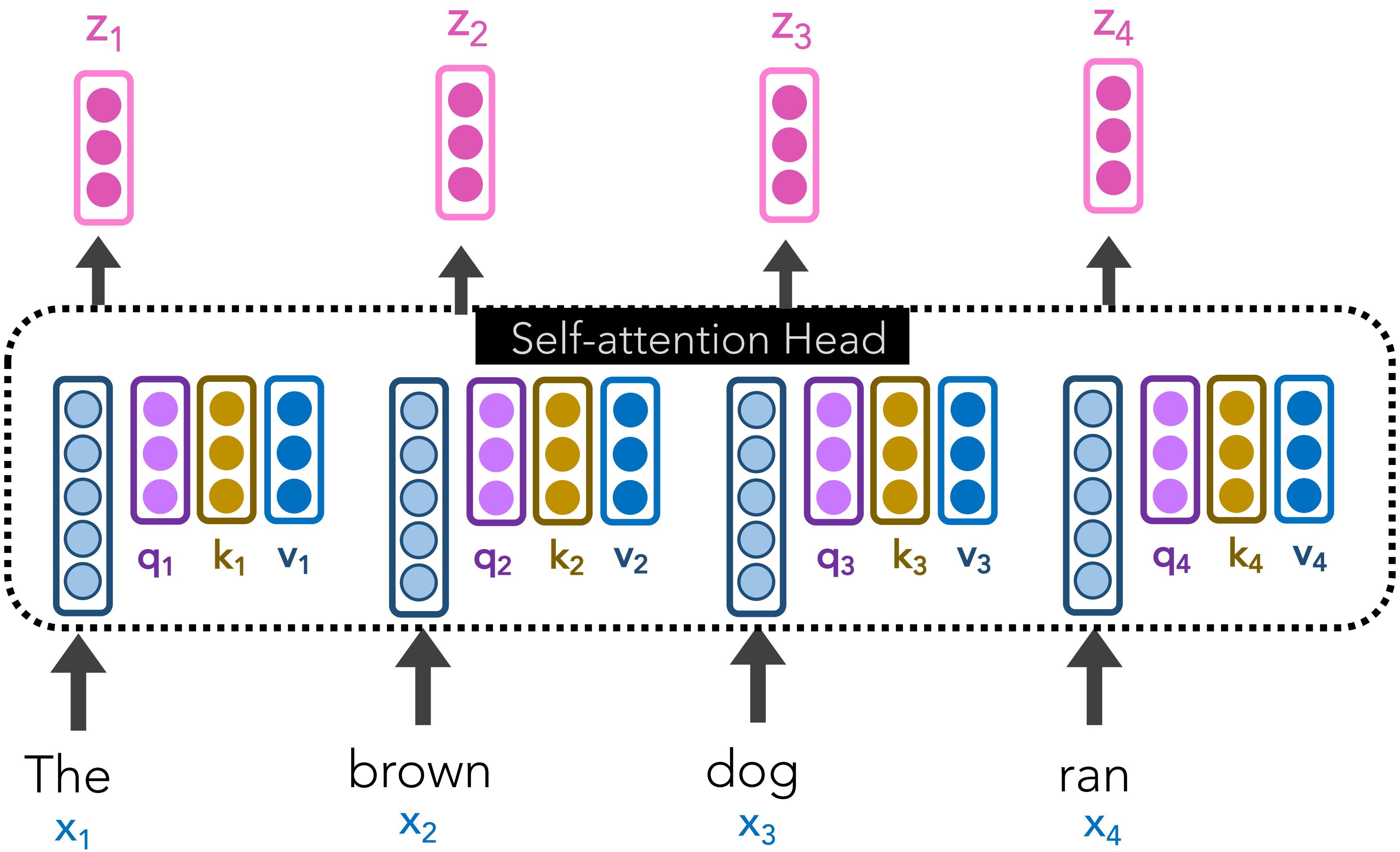
$$z_2 = a_1 \cdot v_1 + a_2 \cdot v_2 + a_3 \cdot v_3 + a_4 \cdot v_4$$

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



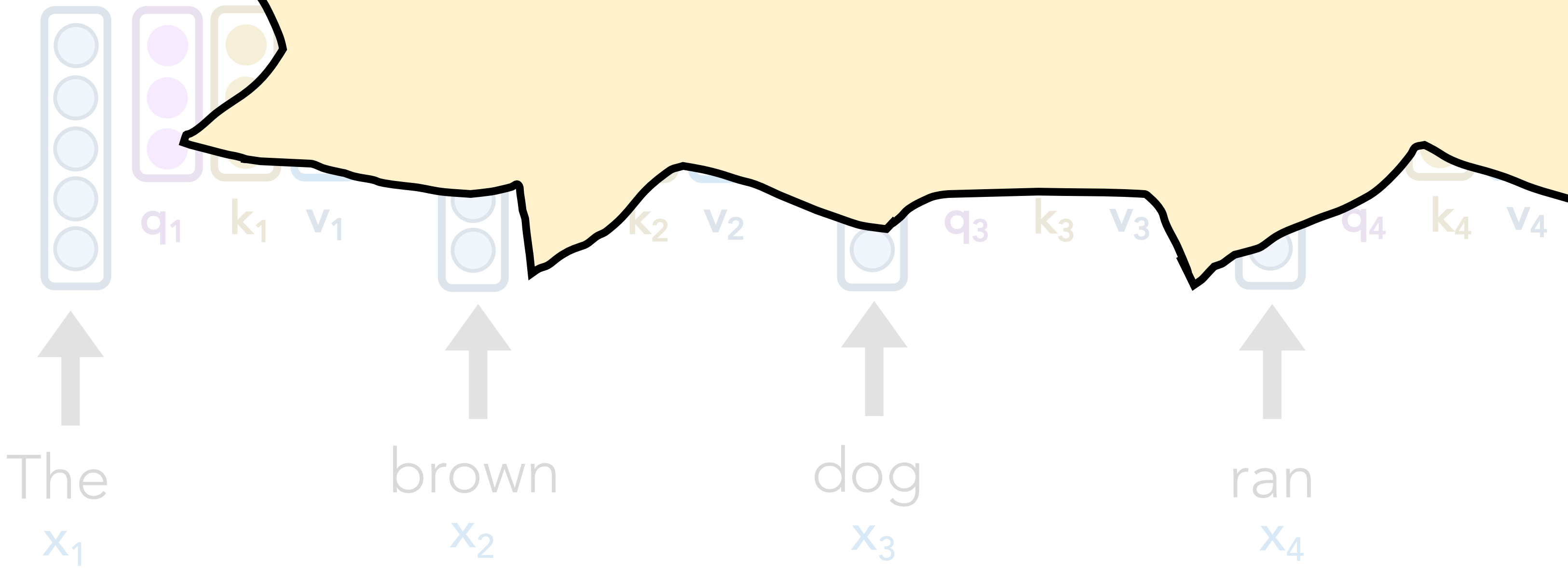
Self-Attention

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



Takeaway:

Self-Attention is powerful; allows us to create great, context-aware representations



Self-Attention may seem strikingly like Attention in seq2seq models

Q: What are the key, query, value vectors in the Attention setup?

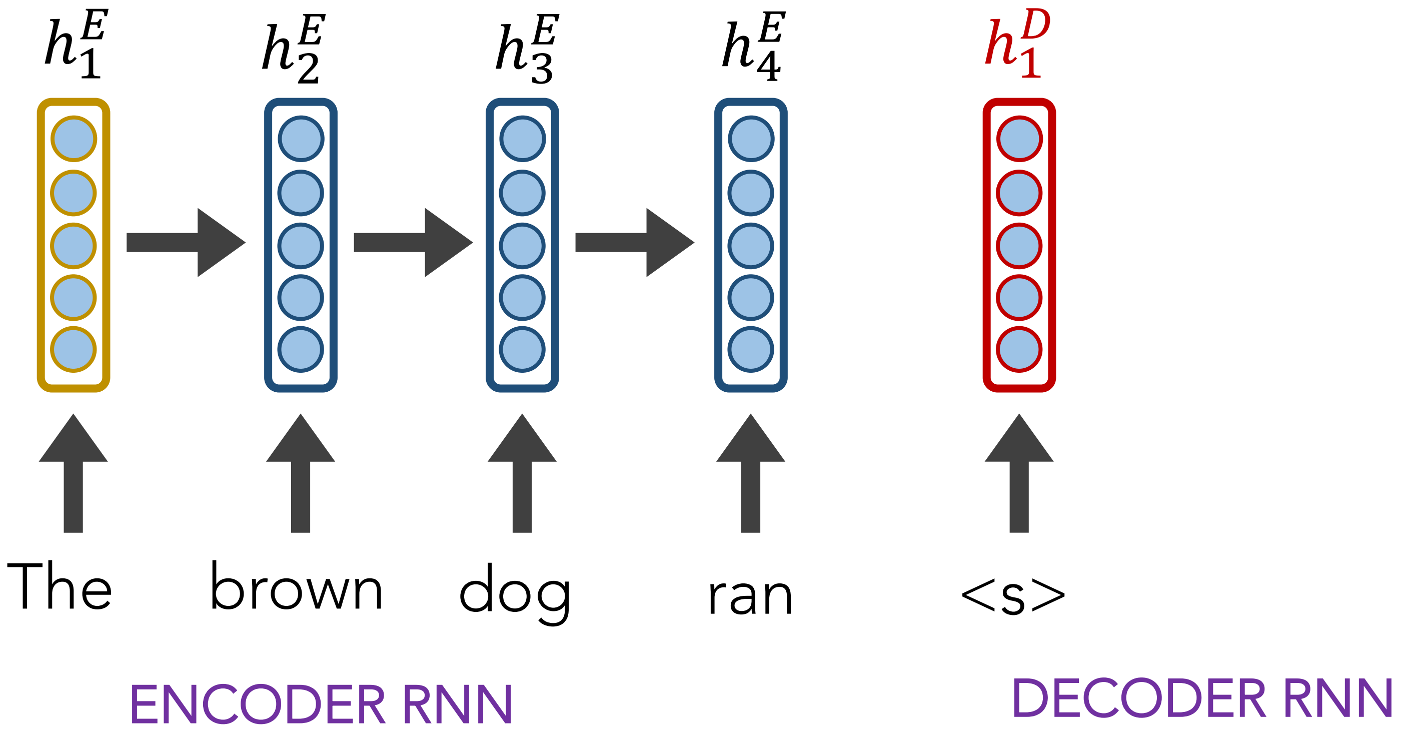
$s_4 = h_1^D * h_4^E$ $\qquad a_4 = \sigma(s_4)$

$s_3 = h_1^D * h_3^E$ $\qquad a_3 = \sigma(s_3)$

$s_2 = h_1^D * h_2^E$ $\qquad a_2 = \sigma(s_2)$

$s_1 = h_1^D * h_1^E$ $\qquad a_1 = \sigma(s_1)$

Attention



$s_4 = h_1^D * h_4^E$ $\quad$ $a_4 = \sigma(s_4)$

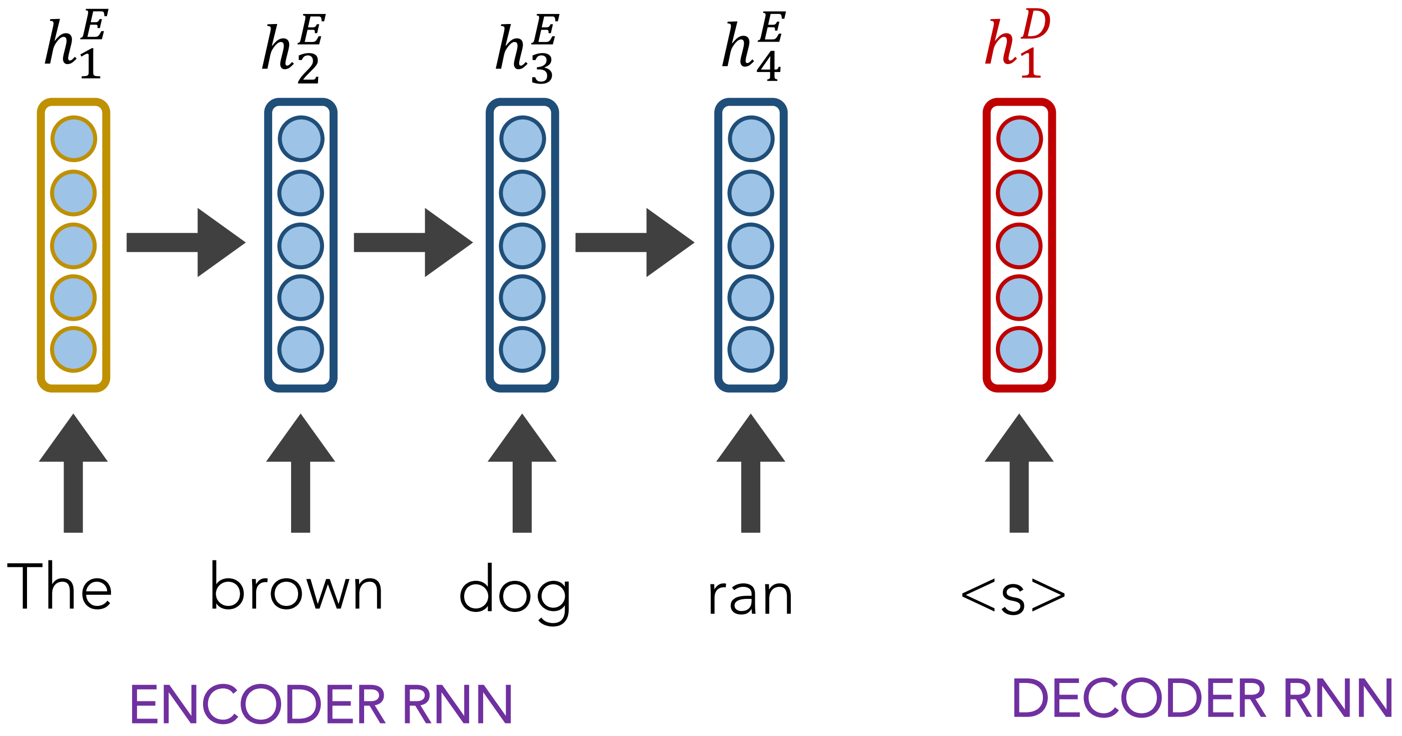
$s_3 = h_1^D * h_3^E$ $\quad$ $a_3 = \sigma(s_3)$

$s_2 = h_1^D * h_2^E$ $\quad$ $a_2 = \sigma(s_2)$

$s_1 = h_1^D * h_1^E$ $\quad$ $a_1 = \sigma(s_1)$

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

Attention



Attention

$s_4 = h_1^D * h_4^E$ $\quad a_4 = \sigma(s_4)$

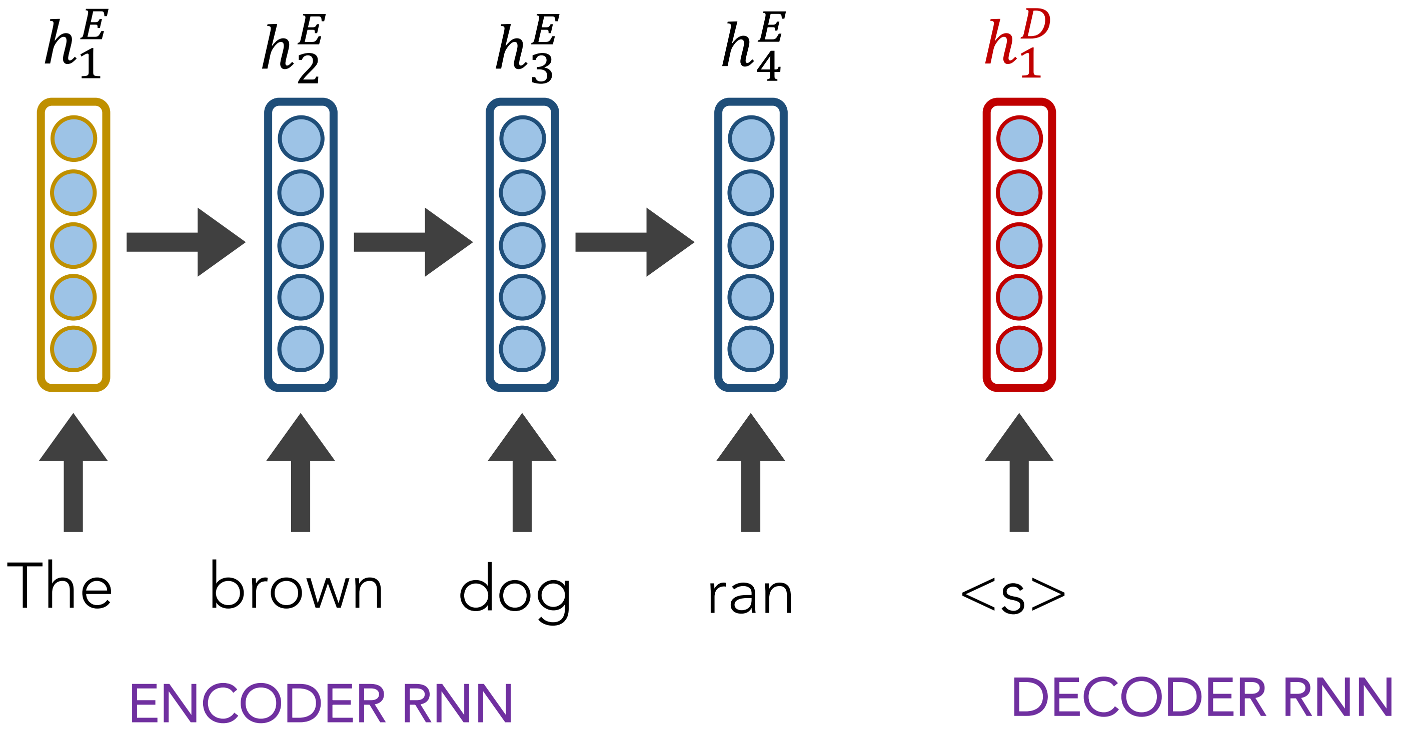
$s_3 = h_1^D * h_3^E$ $\quad a_3 = \sigma(s_3)$

$s_2 = h_1^D * h_2^E$ $\quad a_2 = \sigma(s_2)$

$s_1 = h_1^D * h_1^E$ $\quad a_1 = \sigma(s_1)$

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

$$c_1^D = a_1 \cdot h_1^E + a_2 \cdot h_2^E + a_3 \cdot h_3^E + a_4 \cdot h_4^E$$



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

$s_3 = q_2 \cdot k_3$ $\quad a_3 = \sigma(s_3/8)$

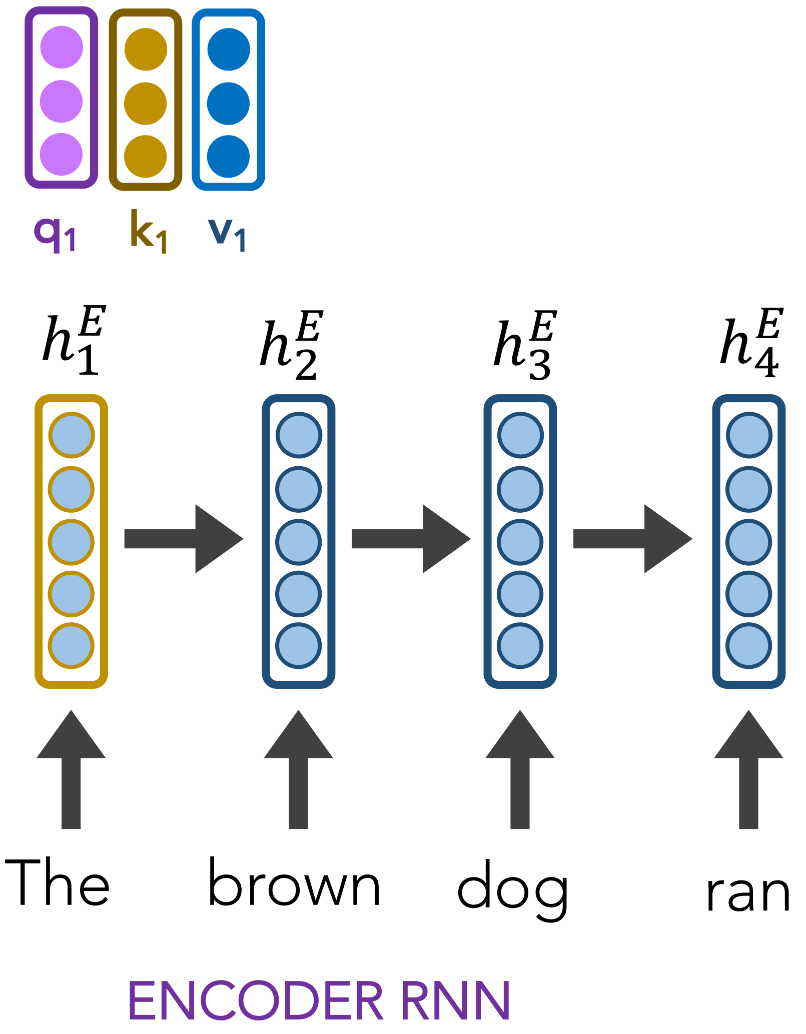
$s_2 = q_2 \cdot k_2$ $\quad a_2 = \sigma(s_2/8)$

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

We multiply each word's value vector by its a_i^1 attention weights to create a better vector z_1

$$z_1 = a_1 \cdot v_1^E + a_2 \cdot v_2^E + a_3 \cdot v_3^E + a_4 \cdot v_4^E$$

Self-Attention

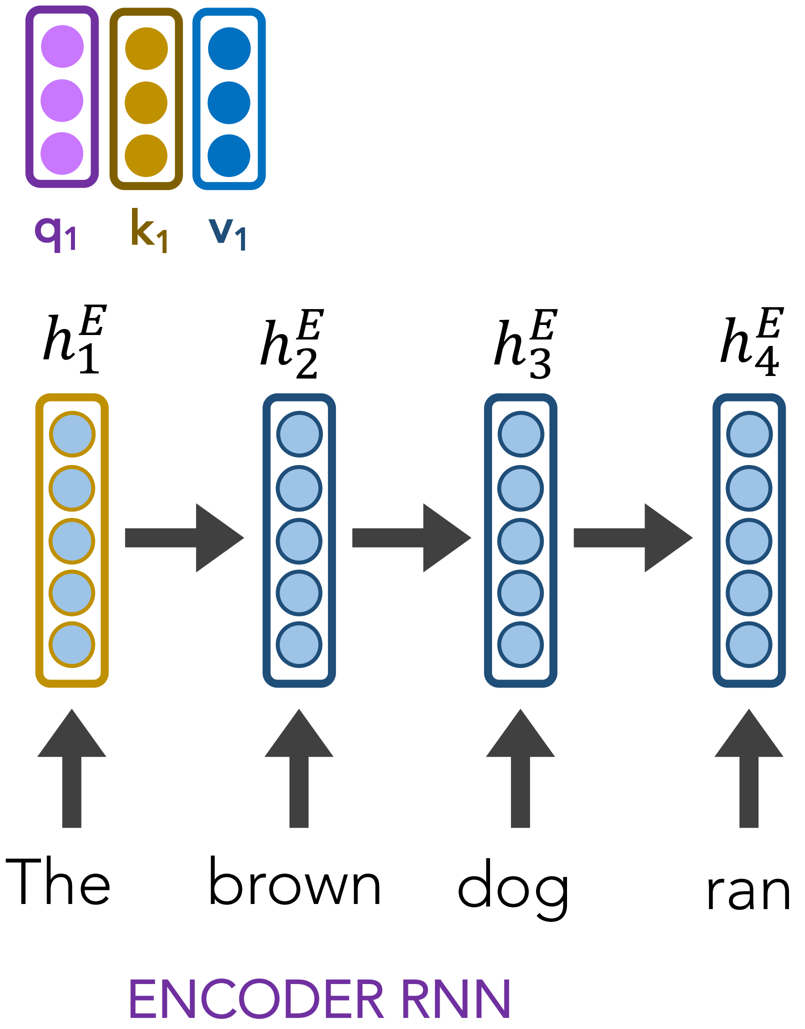


Self-Attention

Self-Attention	Attention	Description
q_i	h_i^D	the probe
k_i	h_i^E	item being compared
v_i	h_i^E	item being weighted

vector by its a_i^1 attention weights to create a better vector z_1

$z_1 = a_1 \cdot v_1^E + a_2 \cdot v_2^E + a_3 \cdot v_3^E + a_4 \cdot v_4^E$



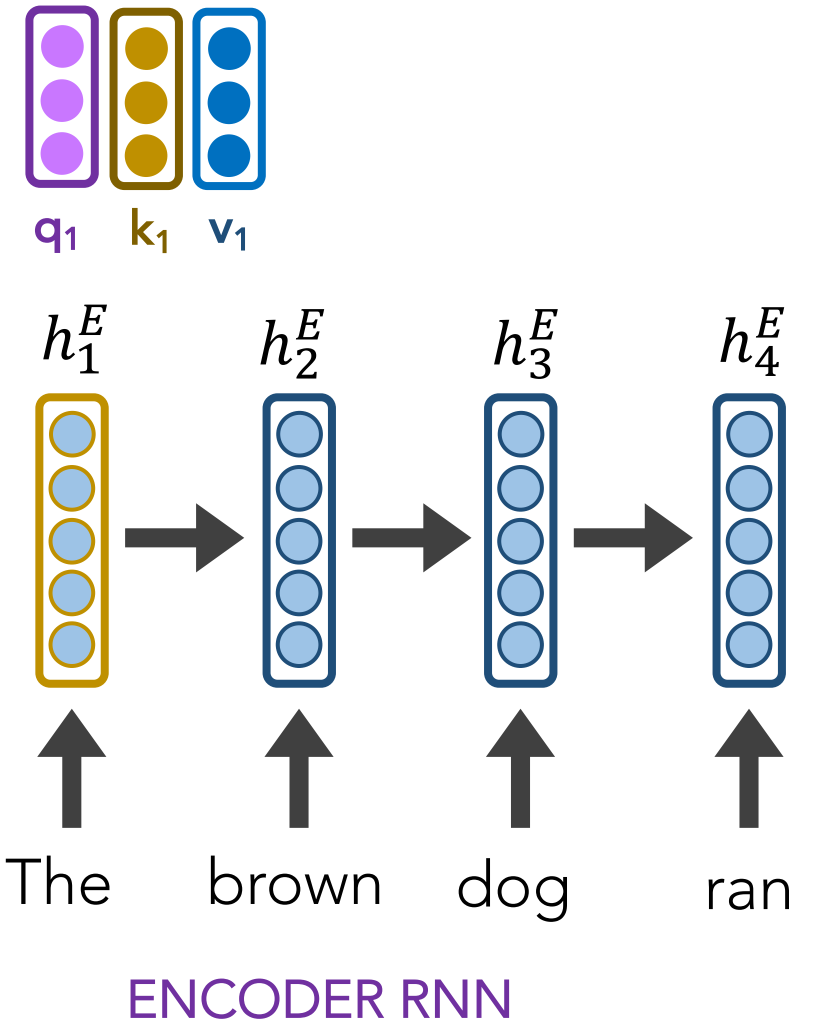
Self-Attention

Self-Attention	Attention	Description
q_i	h_i^D	the probe
k_i	h_i^E	item being compared
v_i	h_i^E	item being weighted

All of these are like surrogates/proxies/abstractions.

This provides flexibility and fewer constraints.

More room for rich abstractions.



Outline

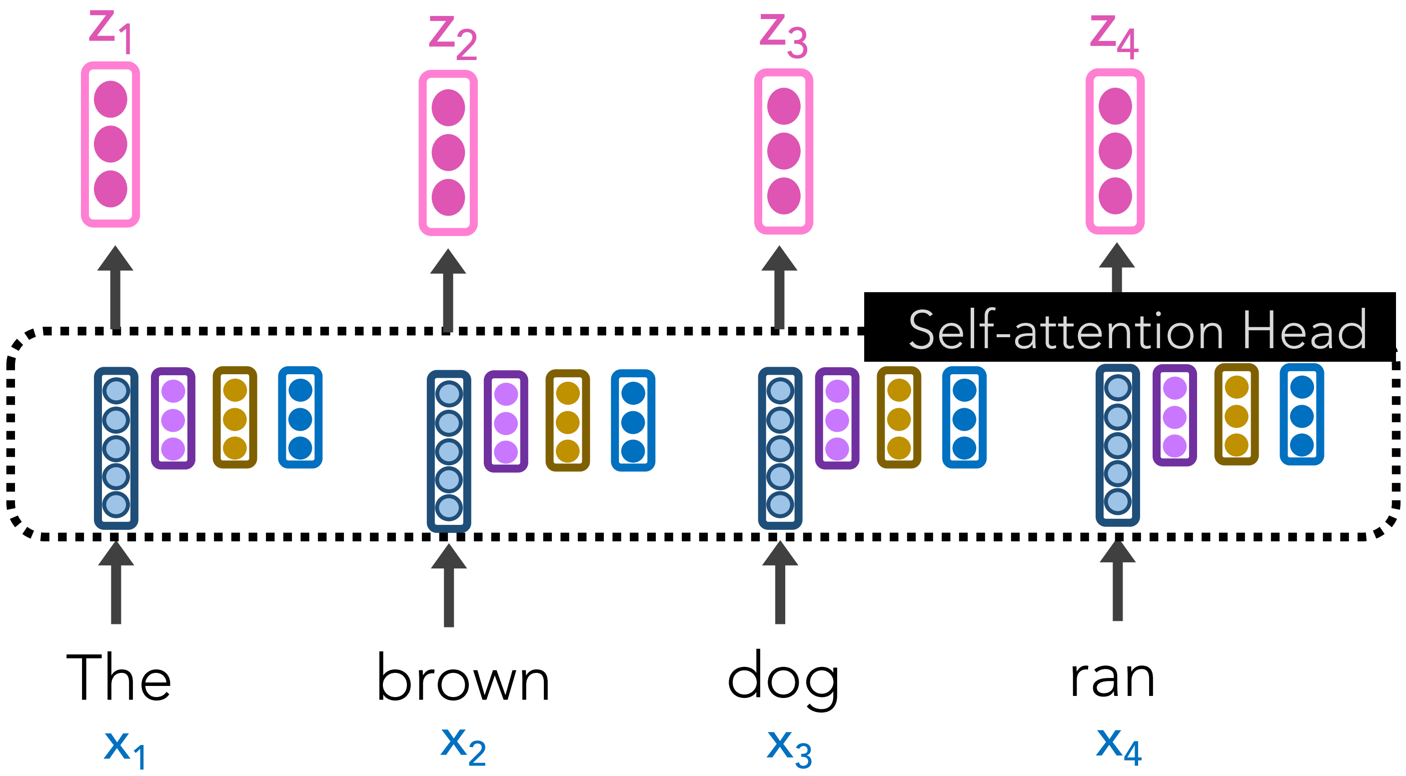
- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

Outline

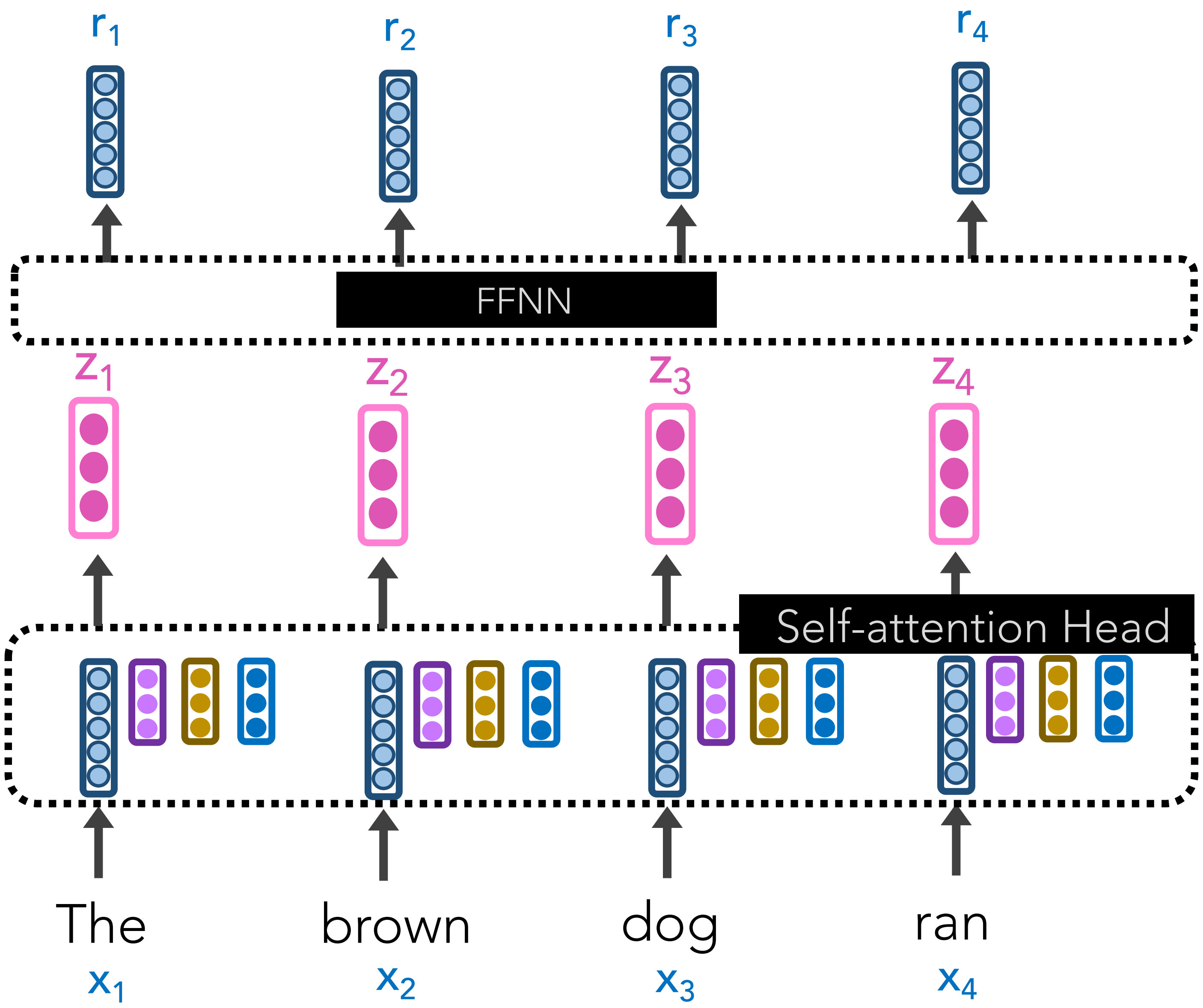
- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

Self-Attention

Let's further pass each z_i through a FFNN

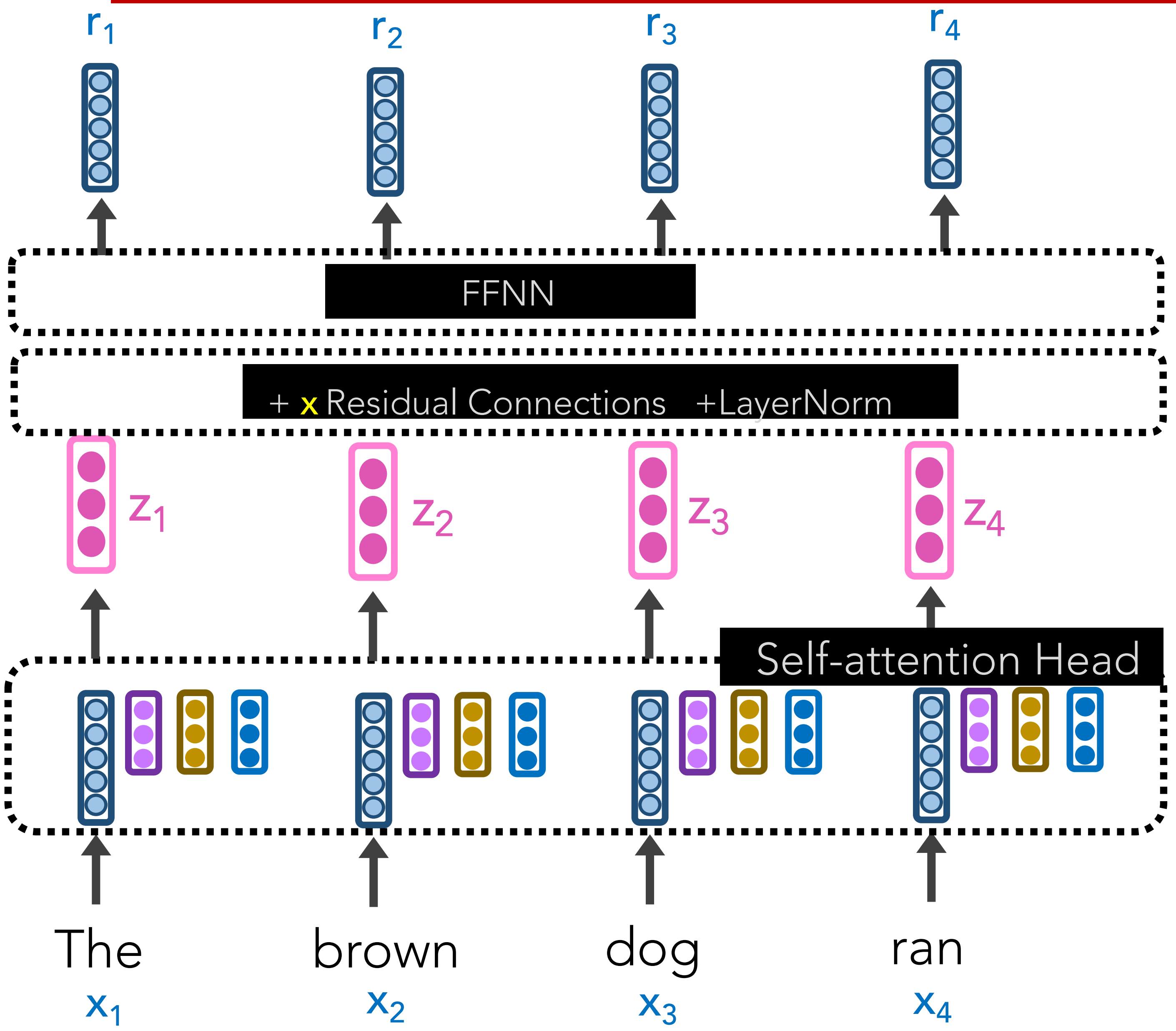


Self-Attention + FFNN



Let's further pass each z_i through a FFNN

Self-Attention + FFNN

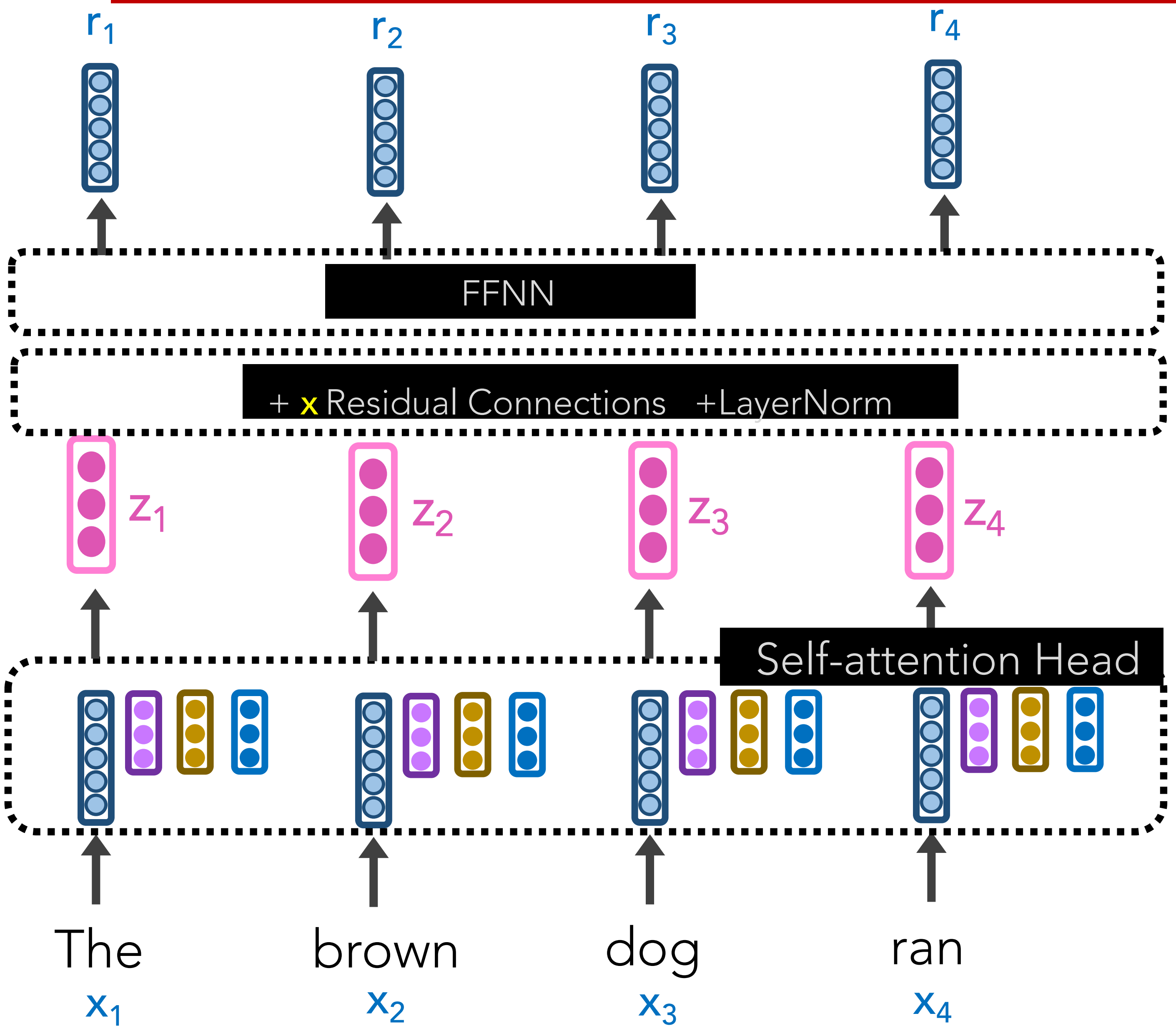


Let's further pass each z_i through a FFNN

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.

Self-Attention + FFNN



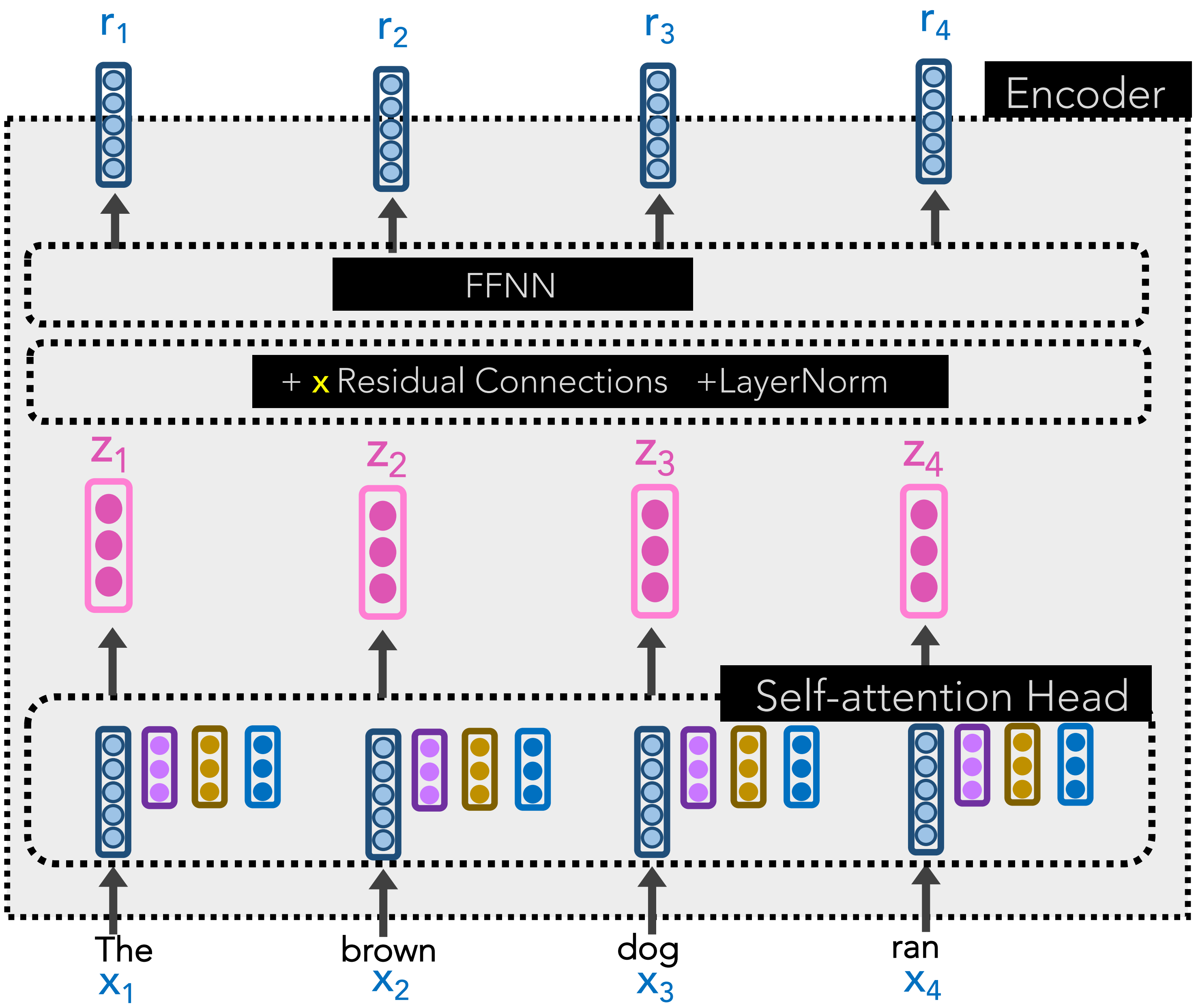
Let's further pass each z_i through a FFNN

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.

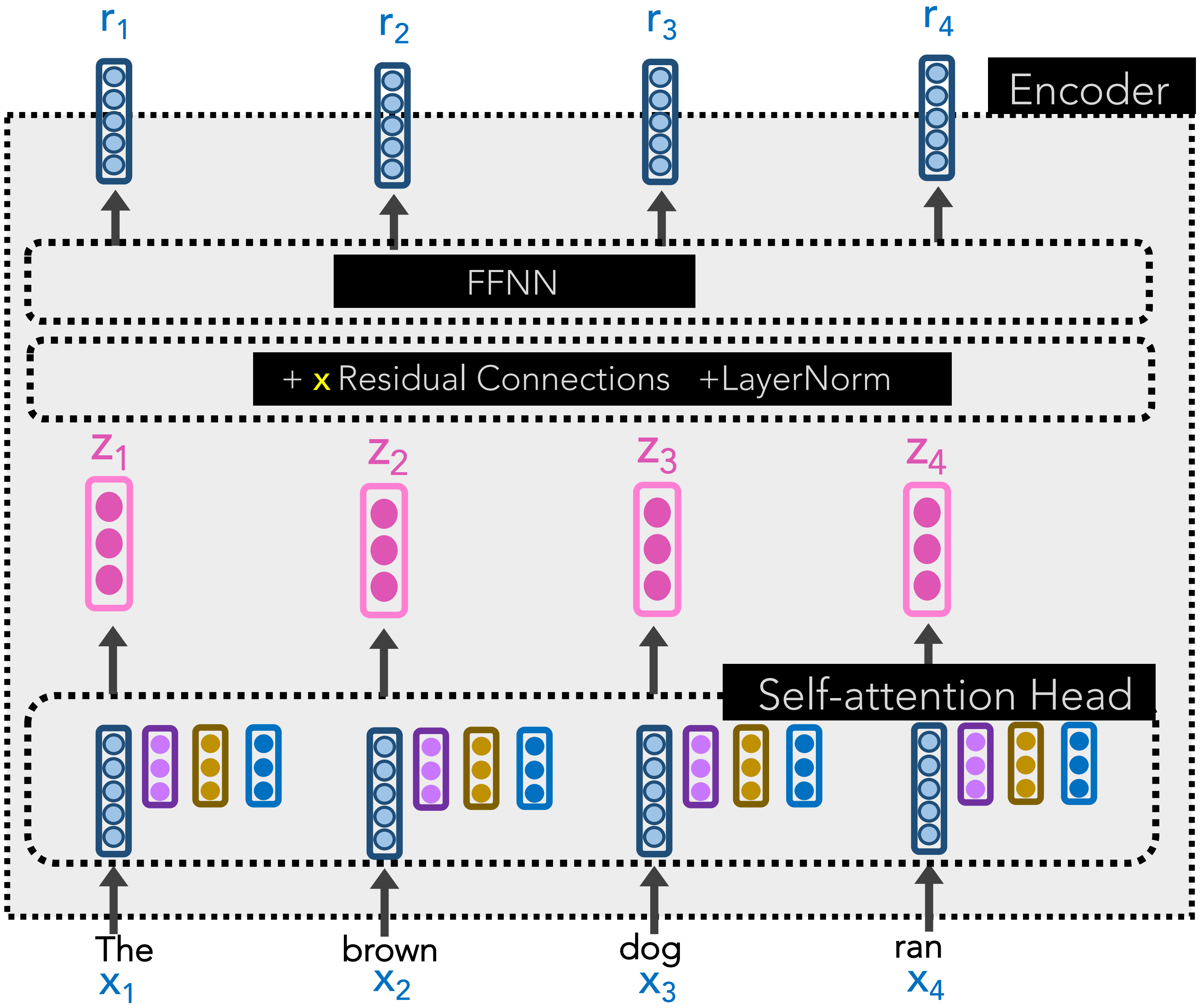
Each z_i can be computed in parallel, unlike LSTMs!

Transformer Encoder



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

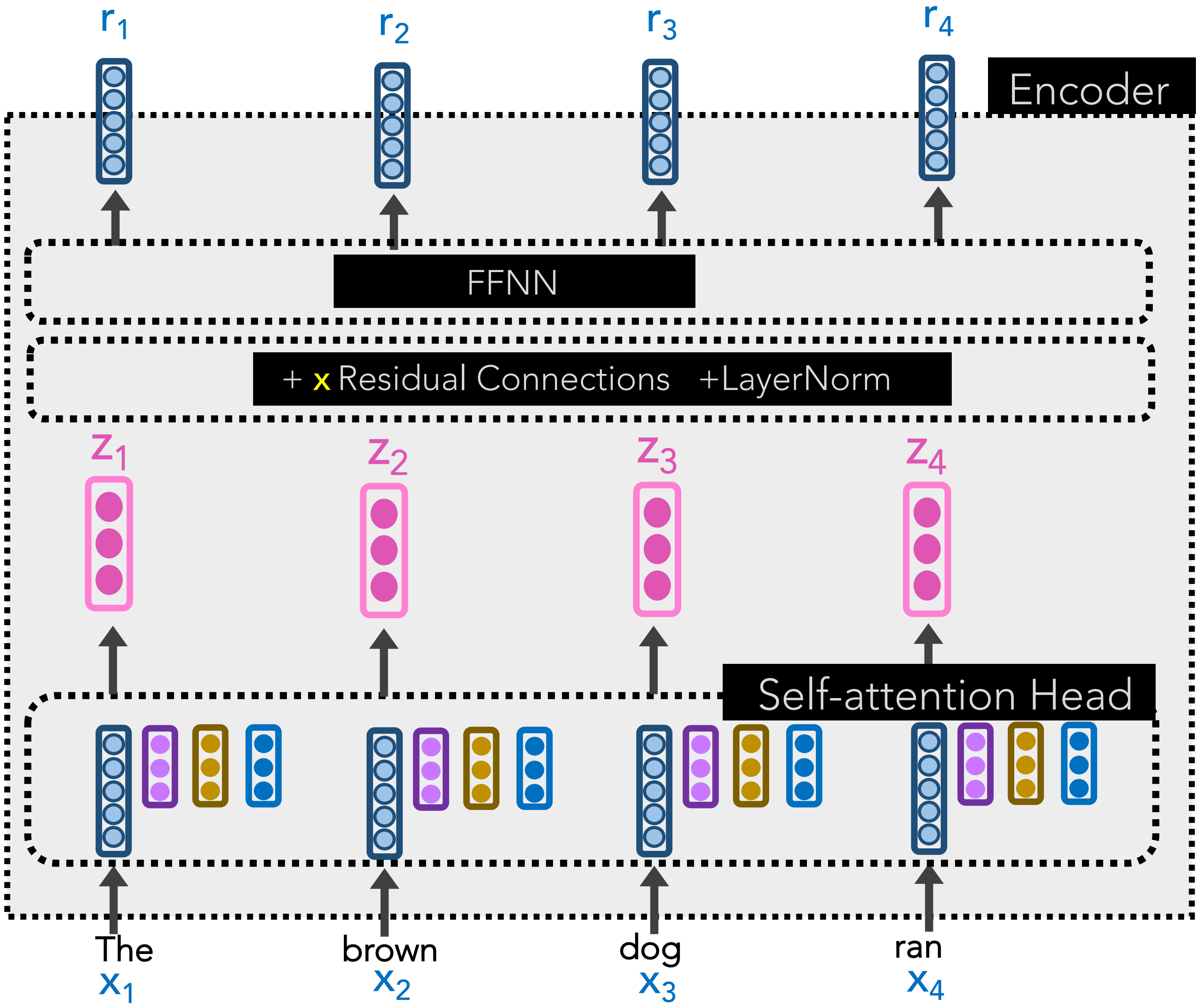
Transformer Encoder



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 Encoder

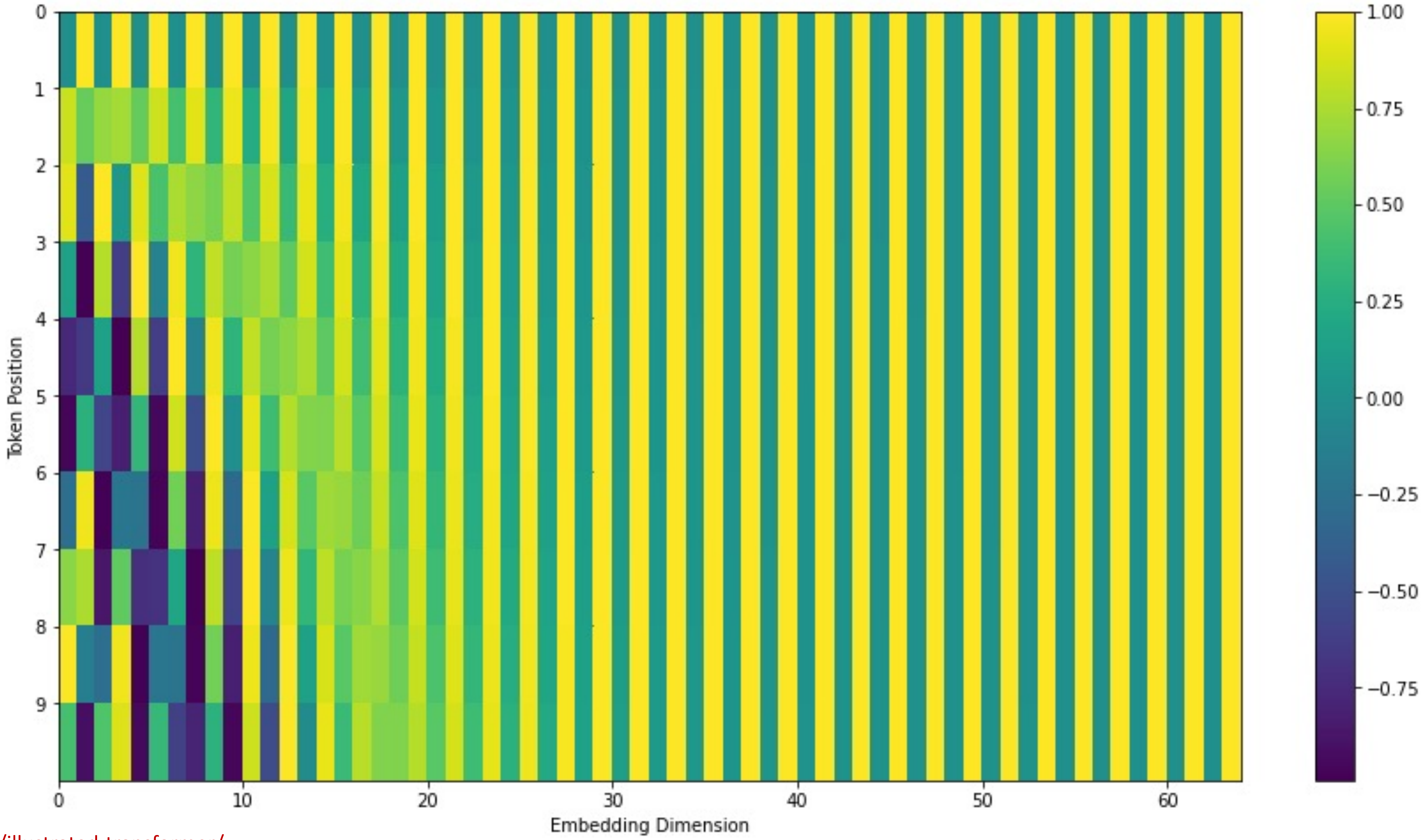


Yay! Our r_i vectors are our new representations, and this entire process is called a **Transformer Encoder**

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 $\sim \sin(i) \cos(i)$

Position Encodings



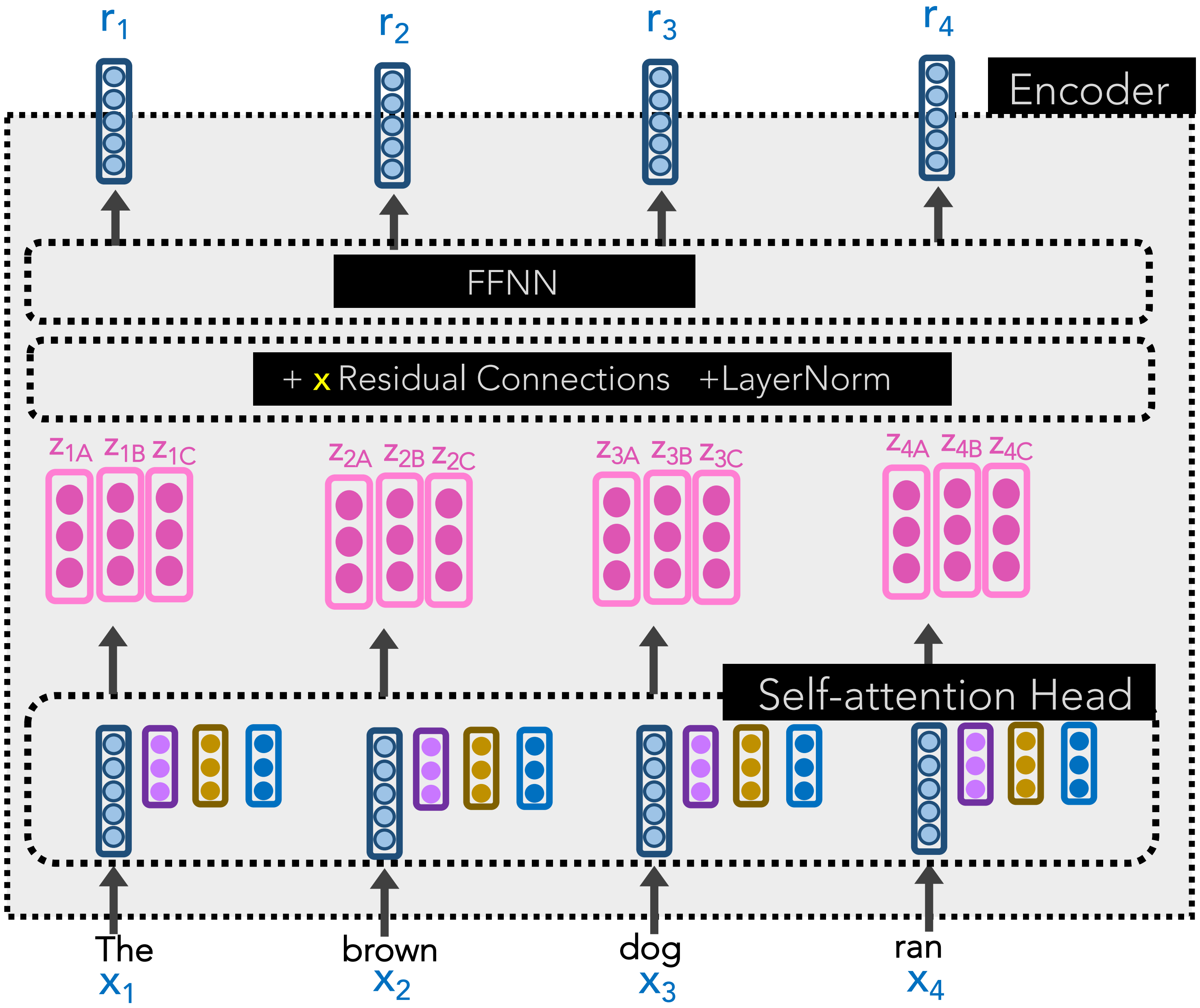
https://jalammar.github.io/illustrated-transformer/

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

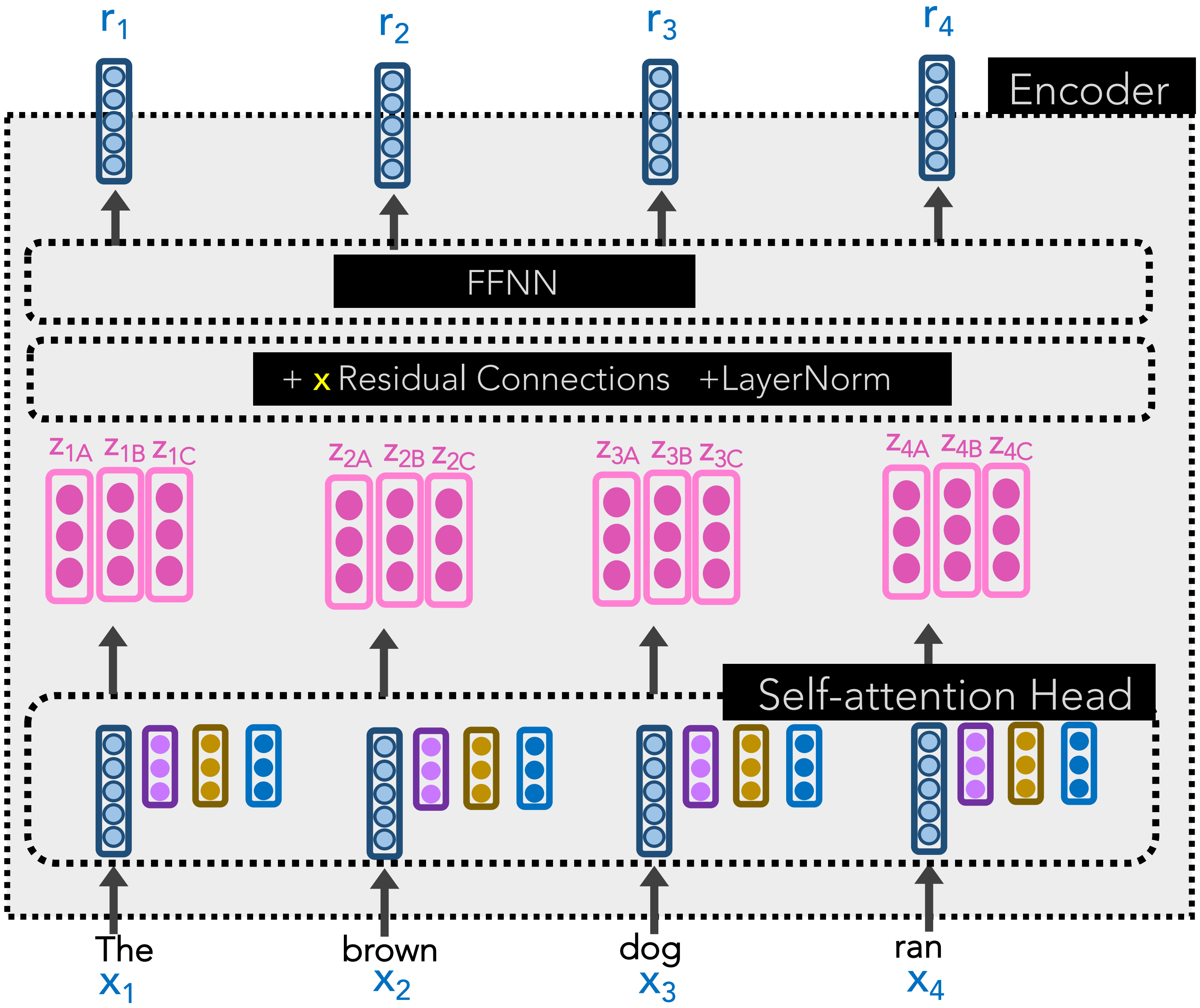
Transformer Encoder



Each Self-Attention Head produces a z_i vector.

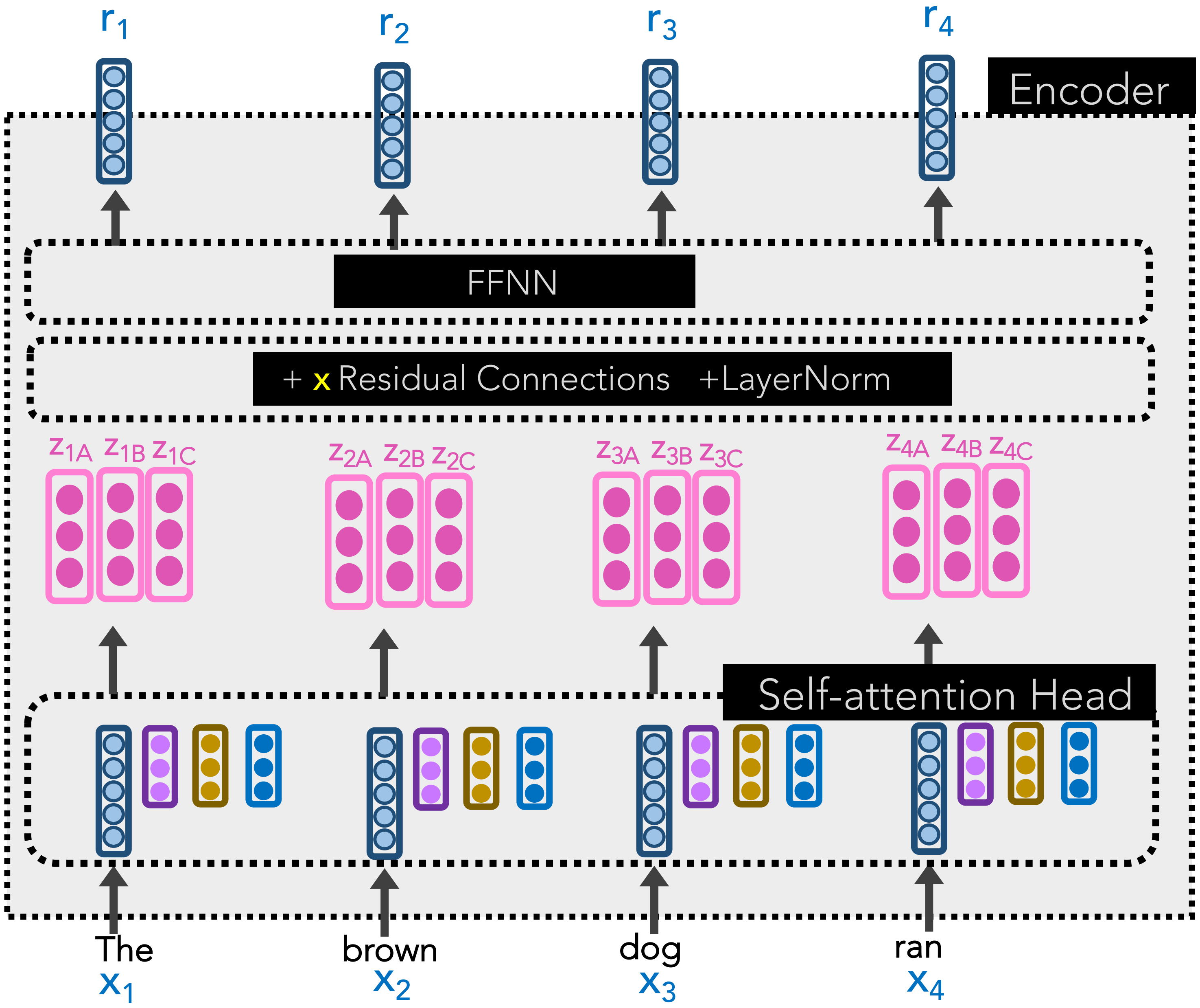
We can, in parallel, use multiple heads and concat the z_i's.

Transformer Encoder

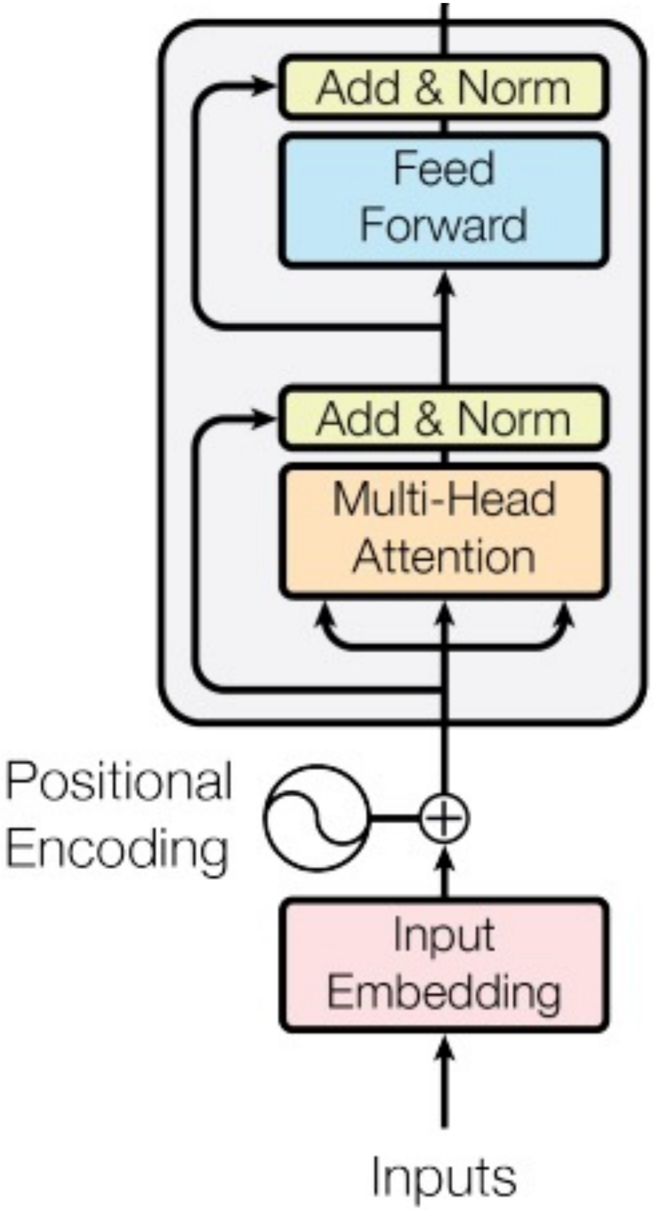


To recap: all of this looks fancy, but ultimately it's just producing a very good contextualized embedding r_i of each word x_i

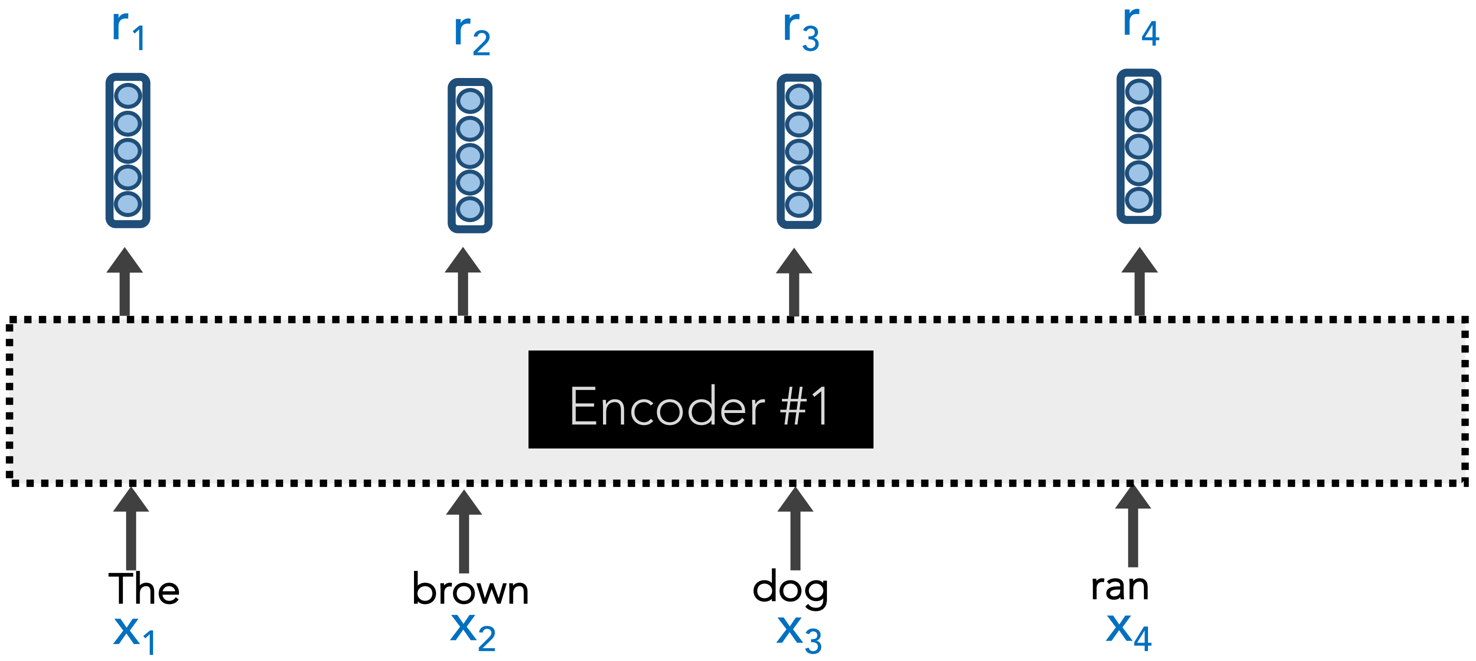
Transformer Encoder



=

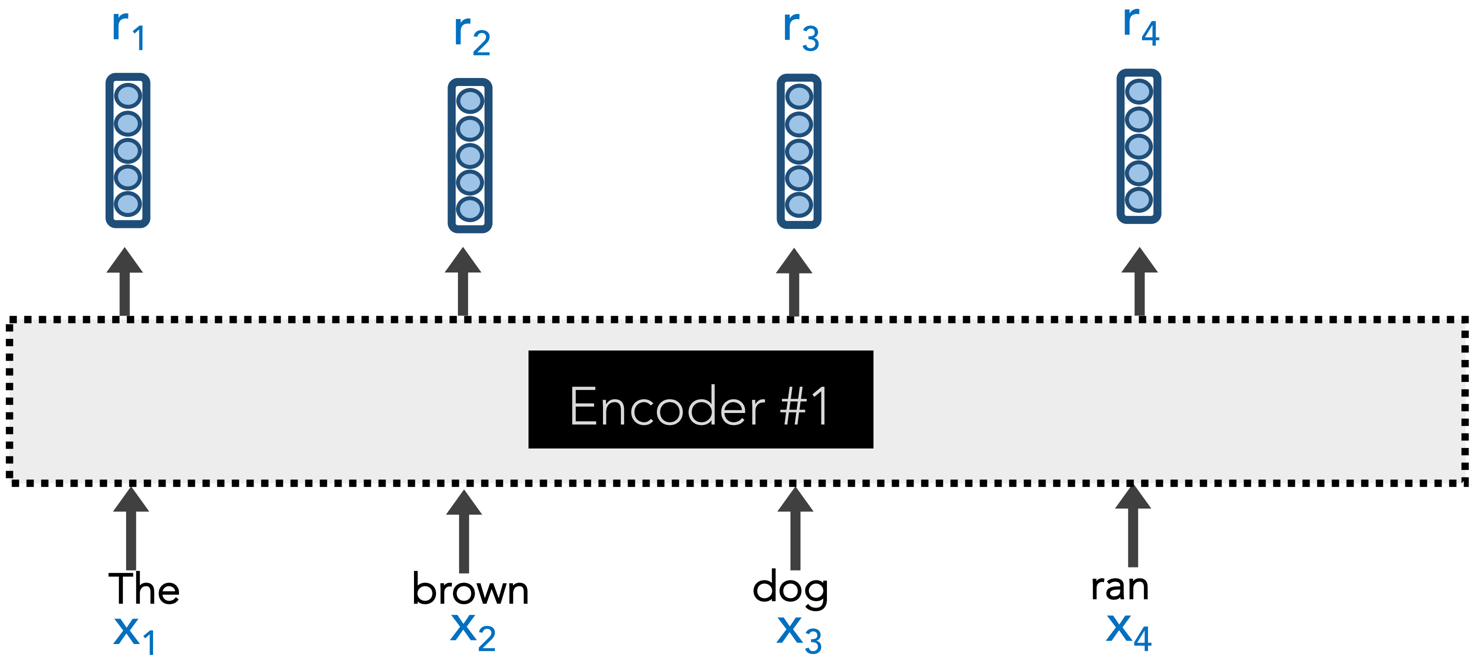


Transformer Encoder



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

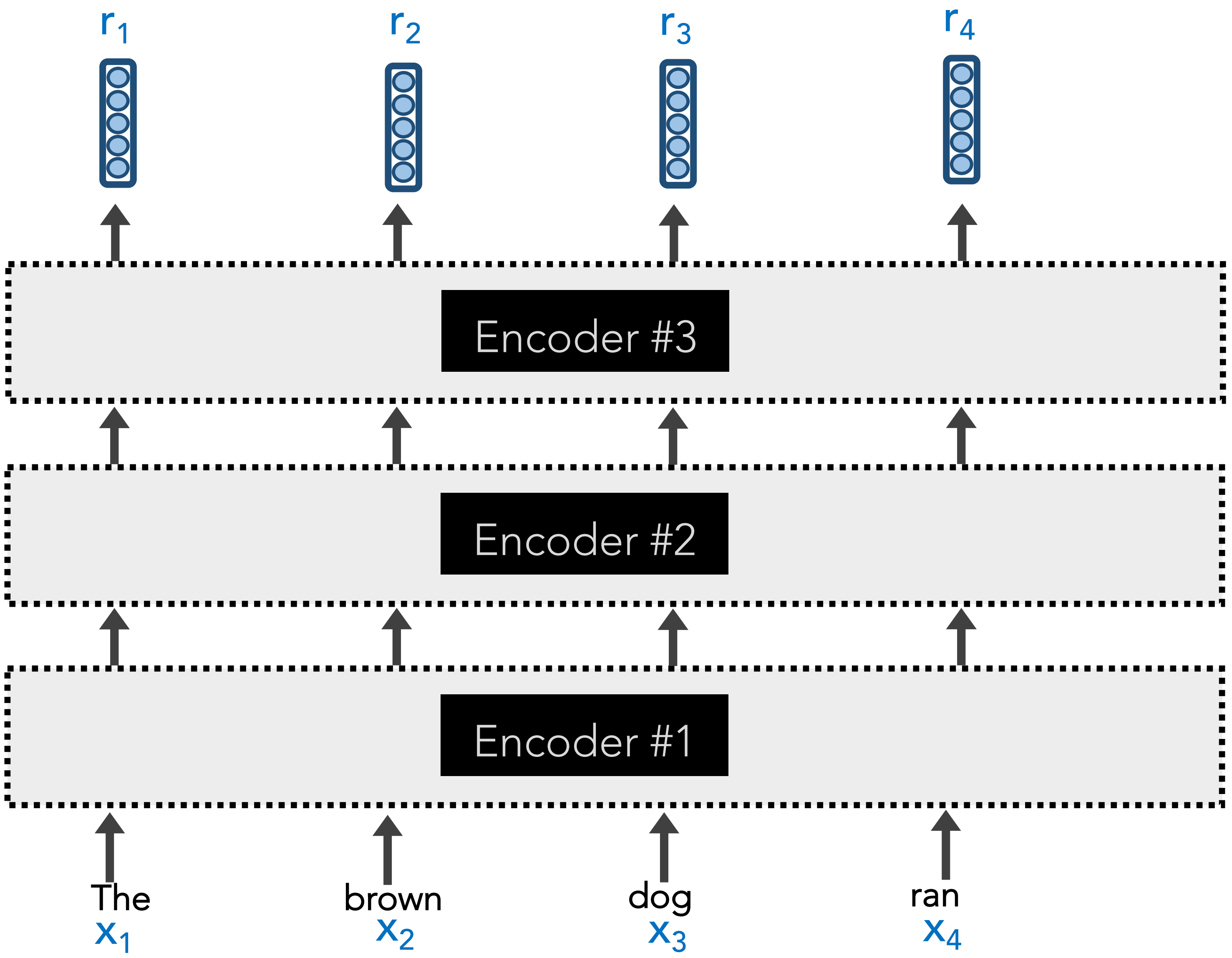
Transformer Encoder



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!

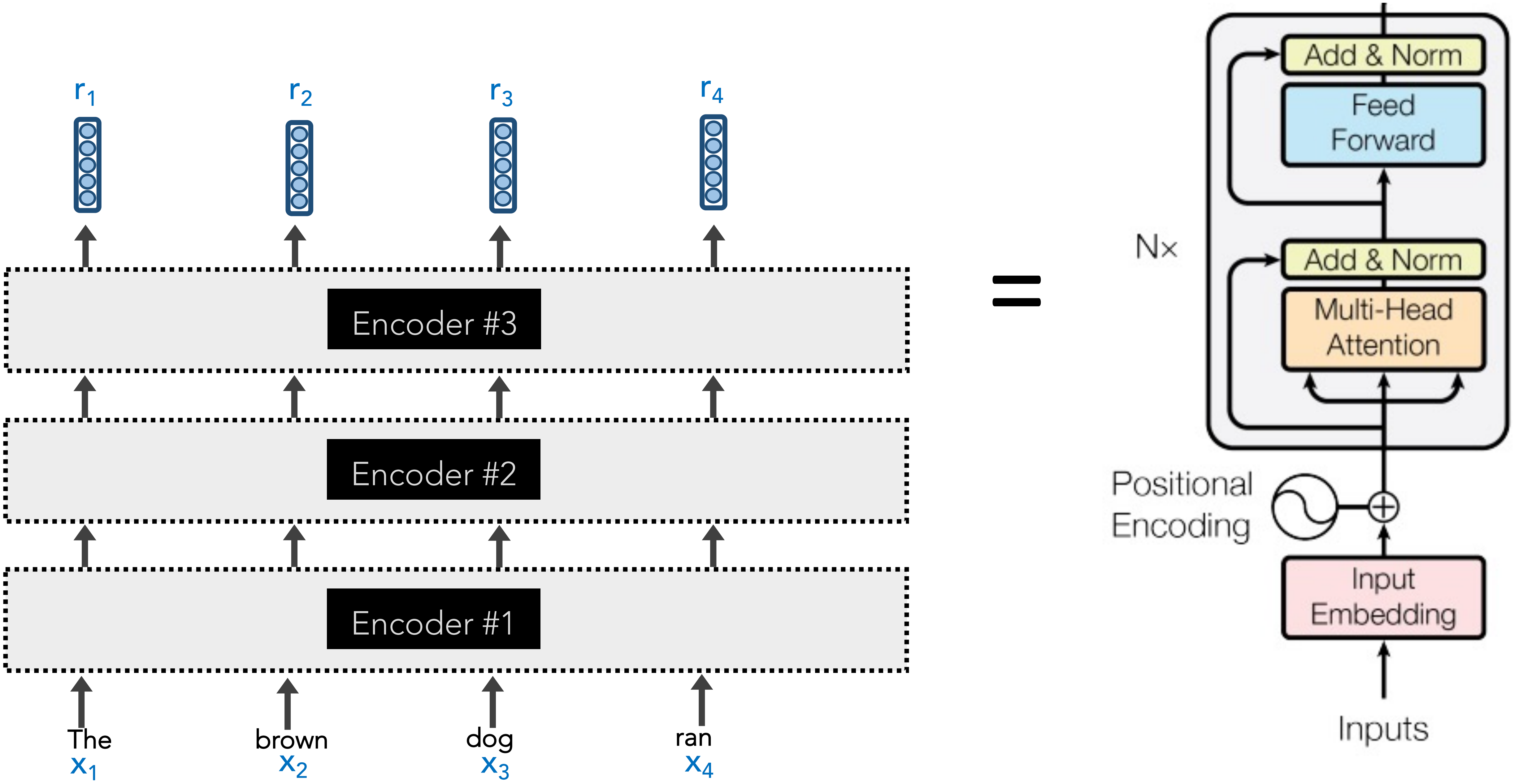
Transformer Encoder



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 Encoder



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

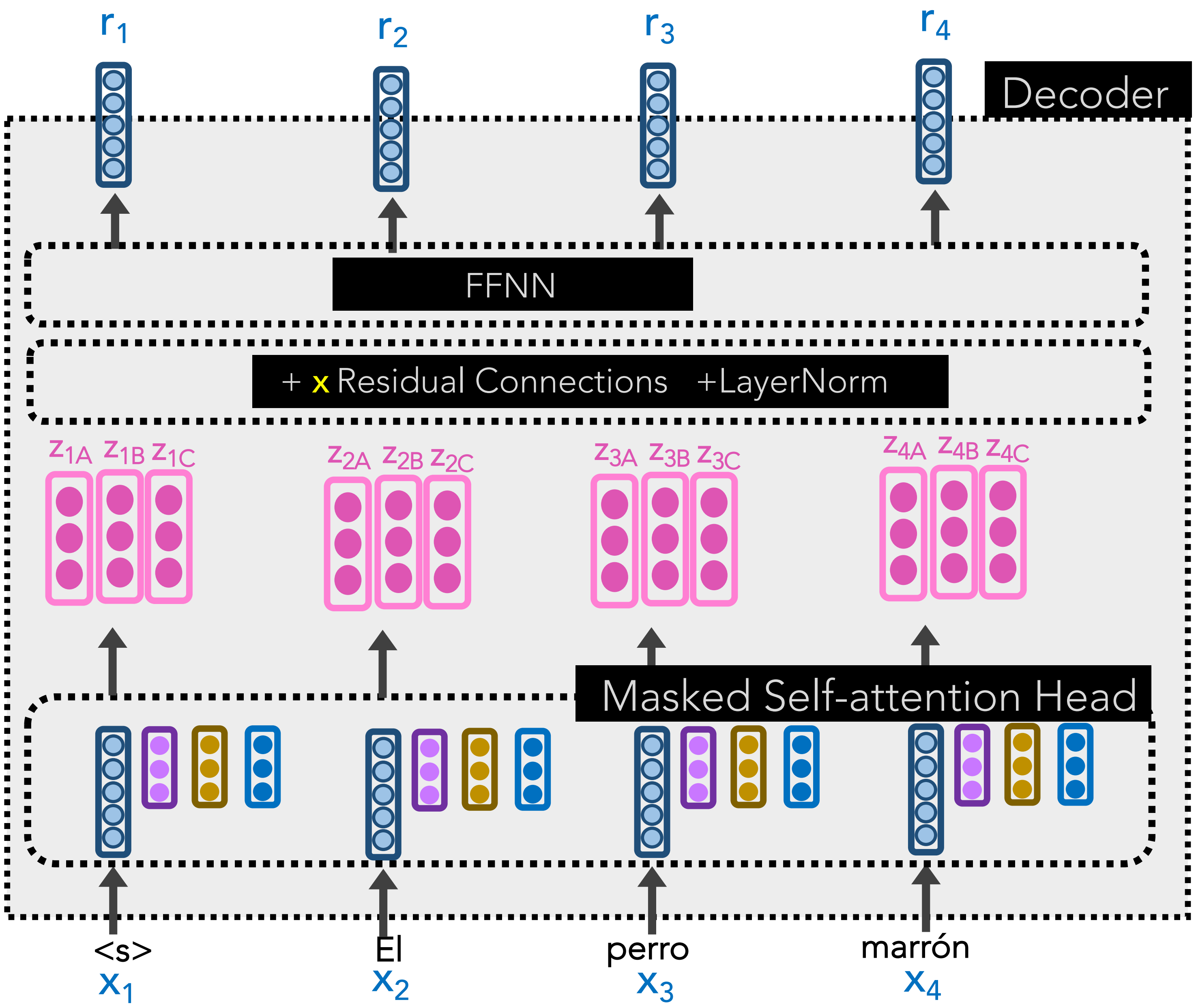
Outline

- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

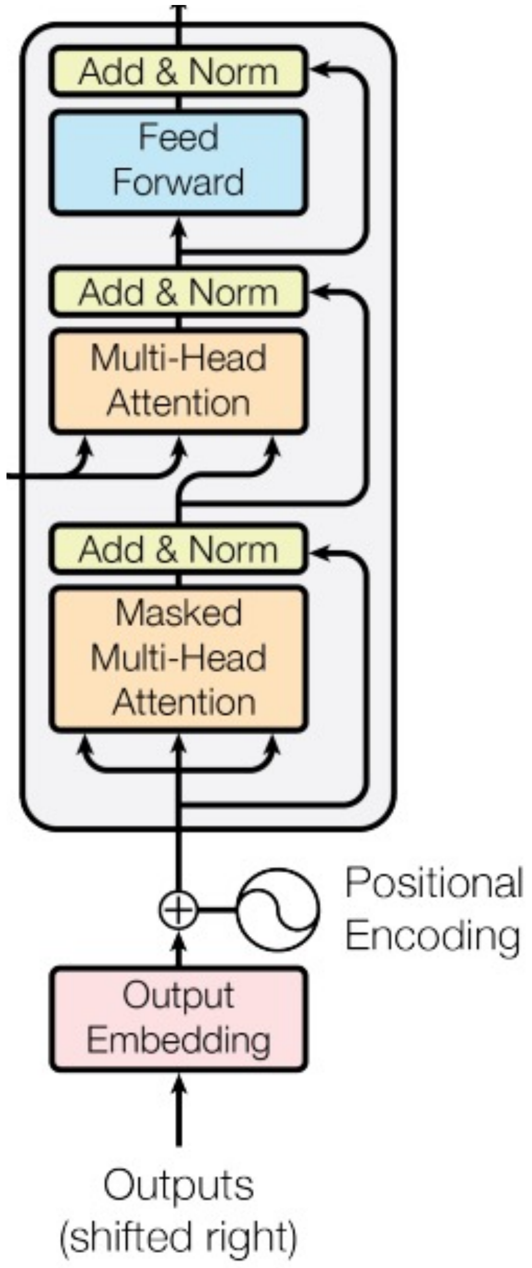
Outline

- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

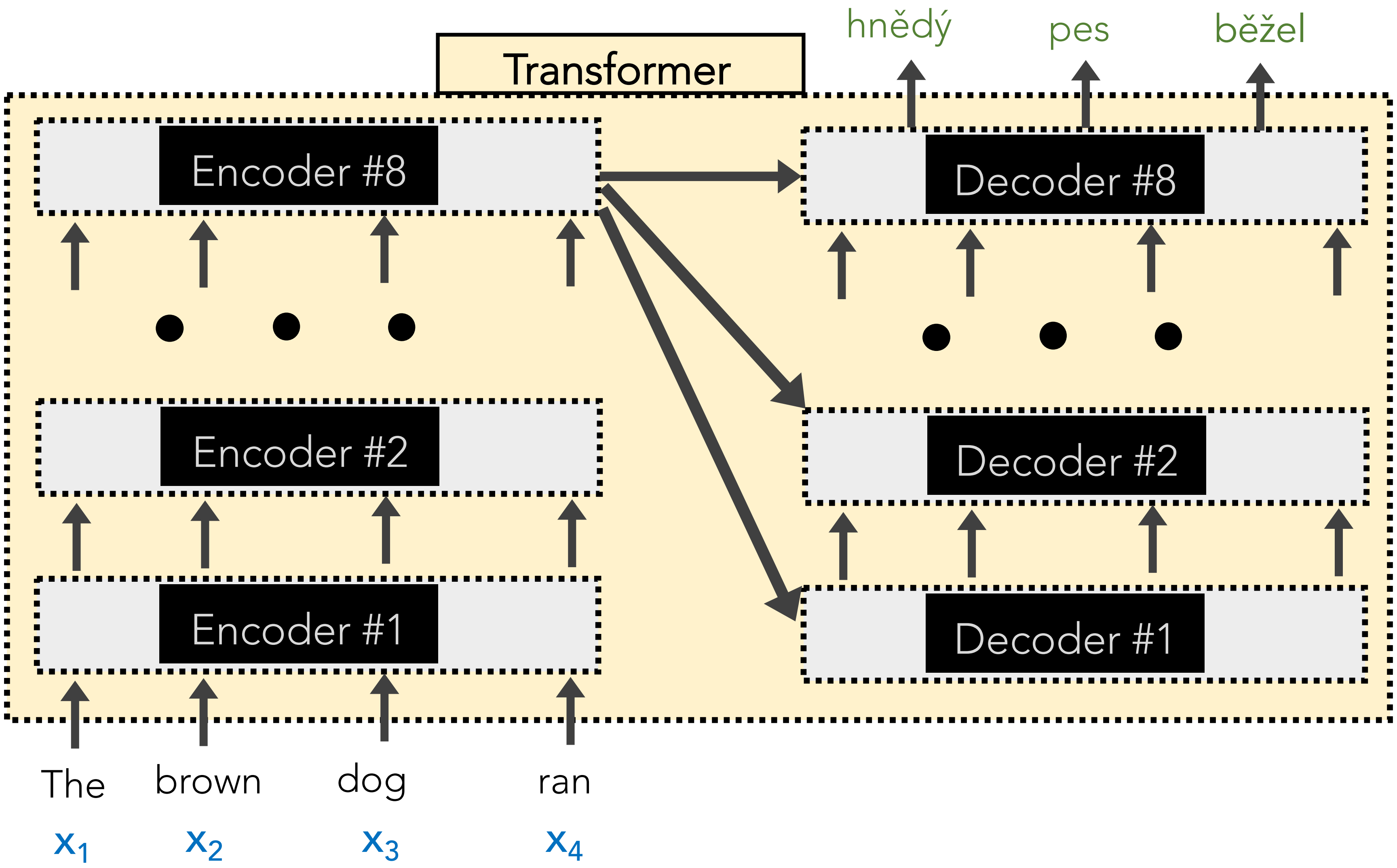
Transformer Decoder



=



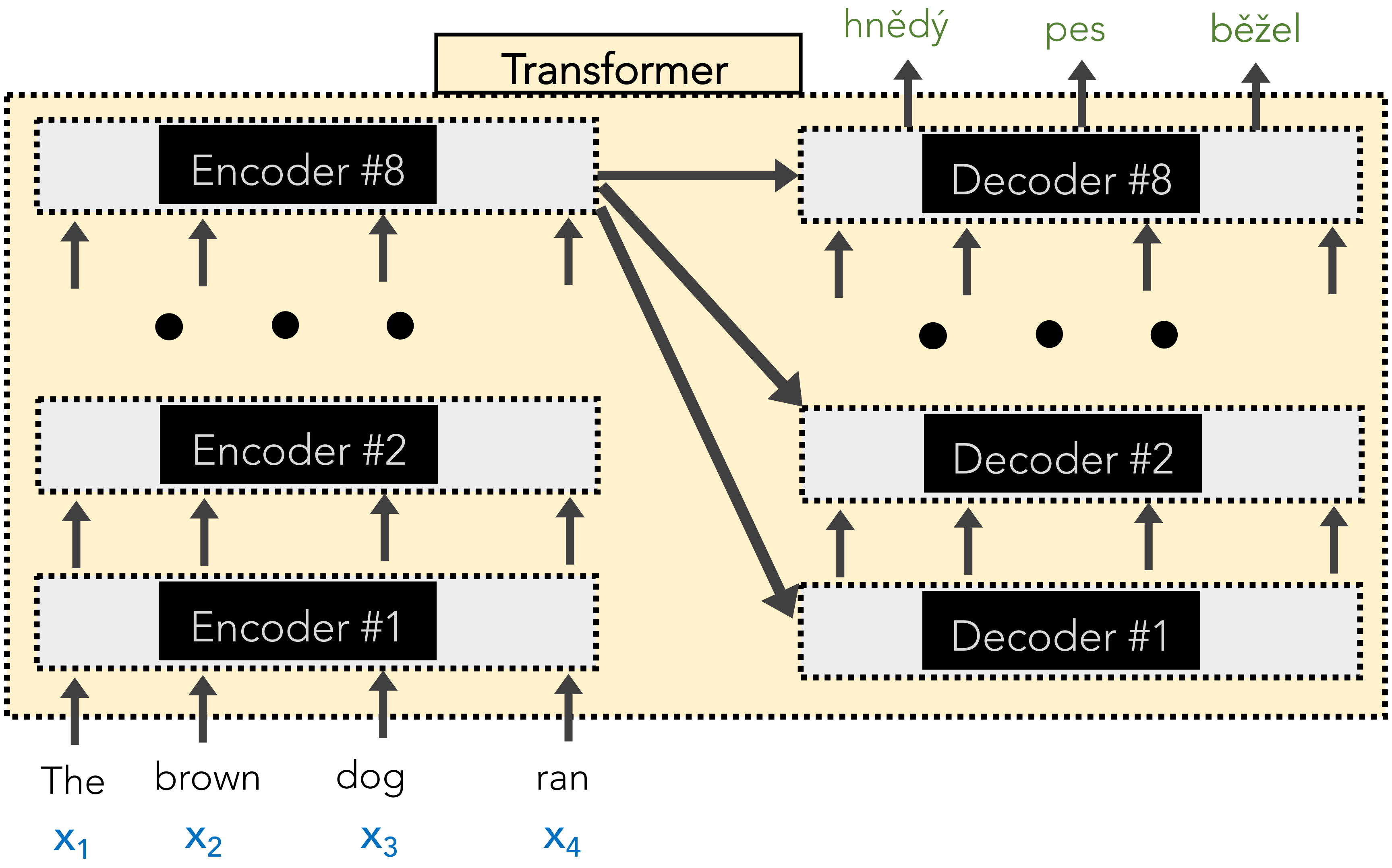
Transformer Encoders and Decoders



Transformer Encoders produce contextualized embeddings of each word

Transformer Decoders generate new sequences of text

Transformer Encoders and Decoders

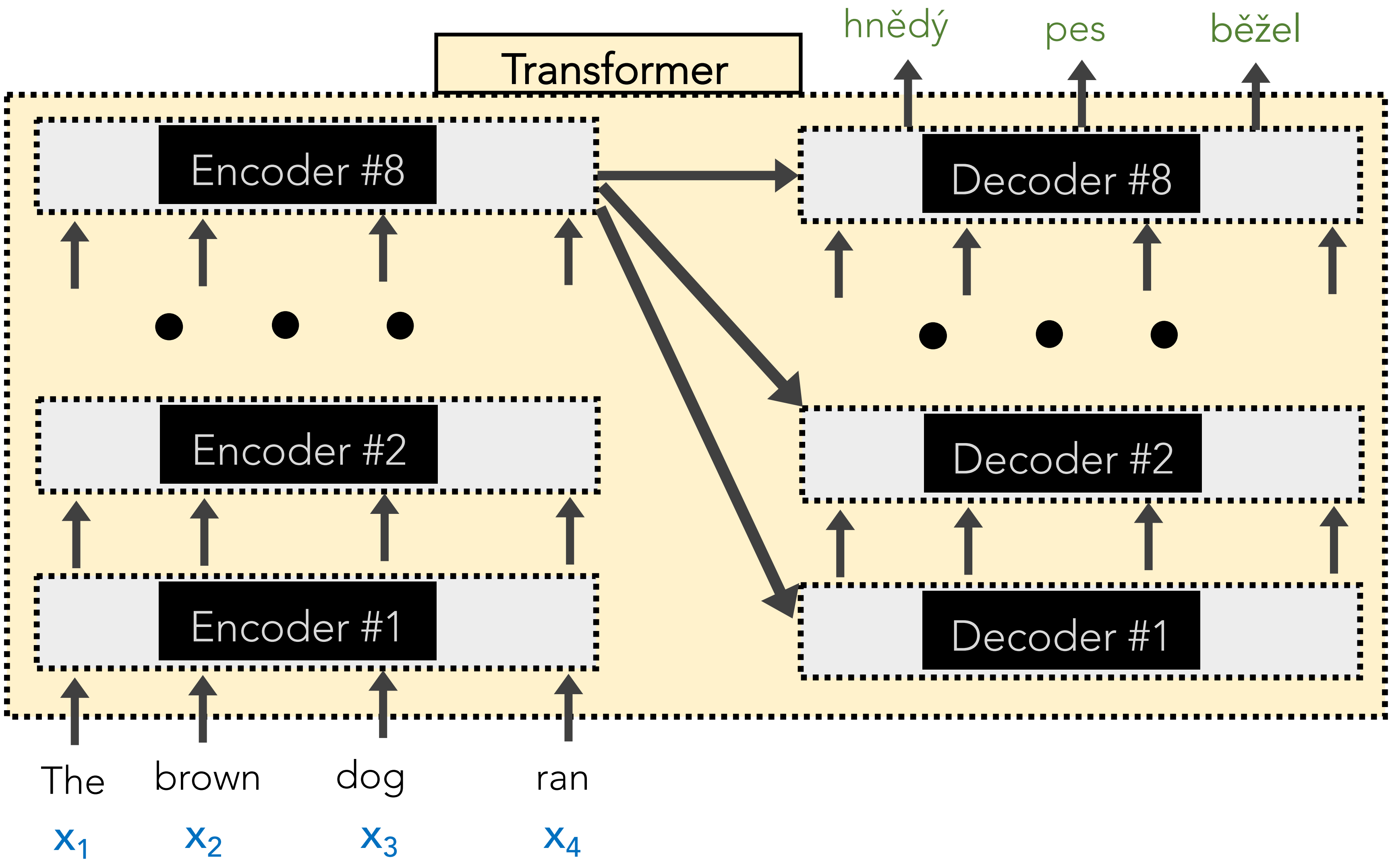


NOTE

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 Encoders and Decoders

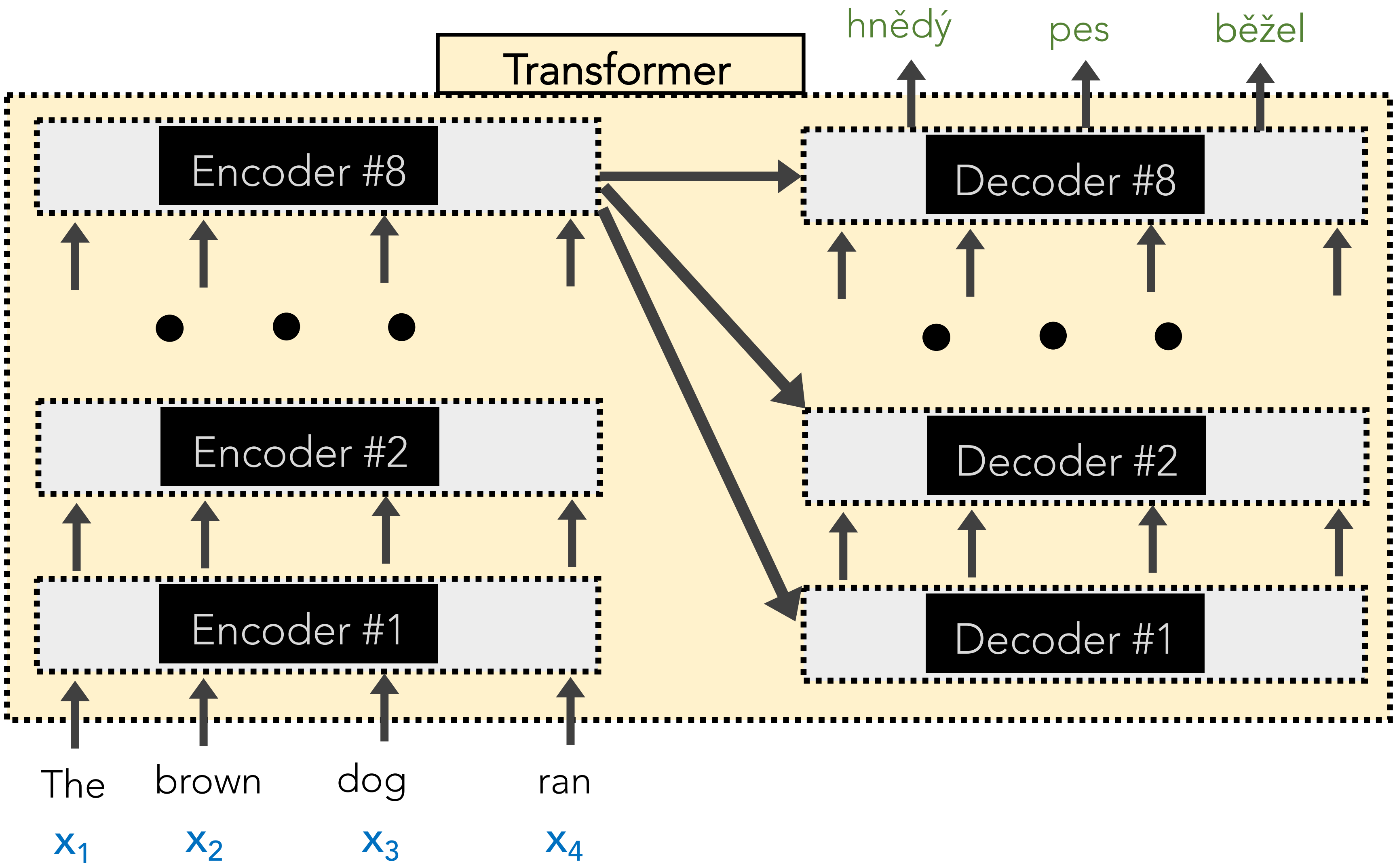


NOTE

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.

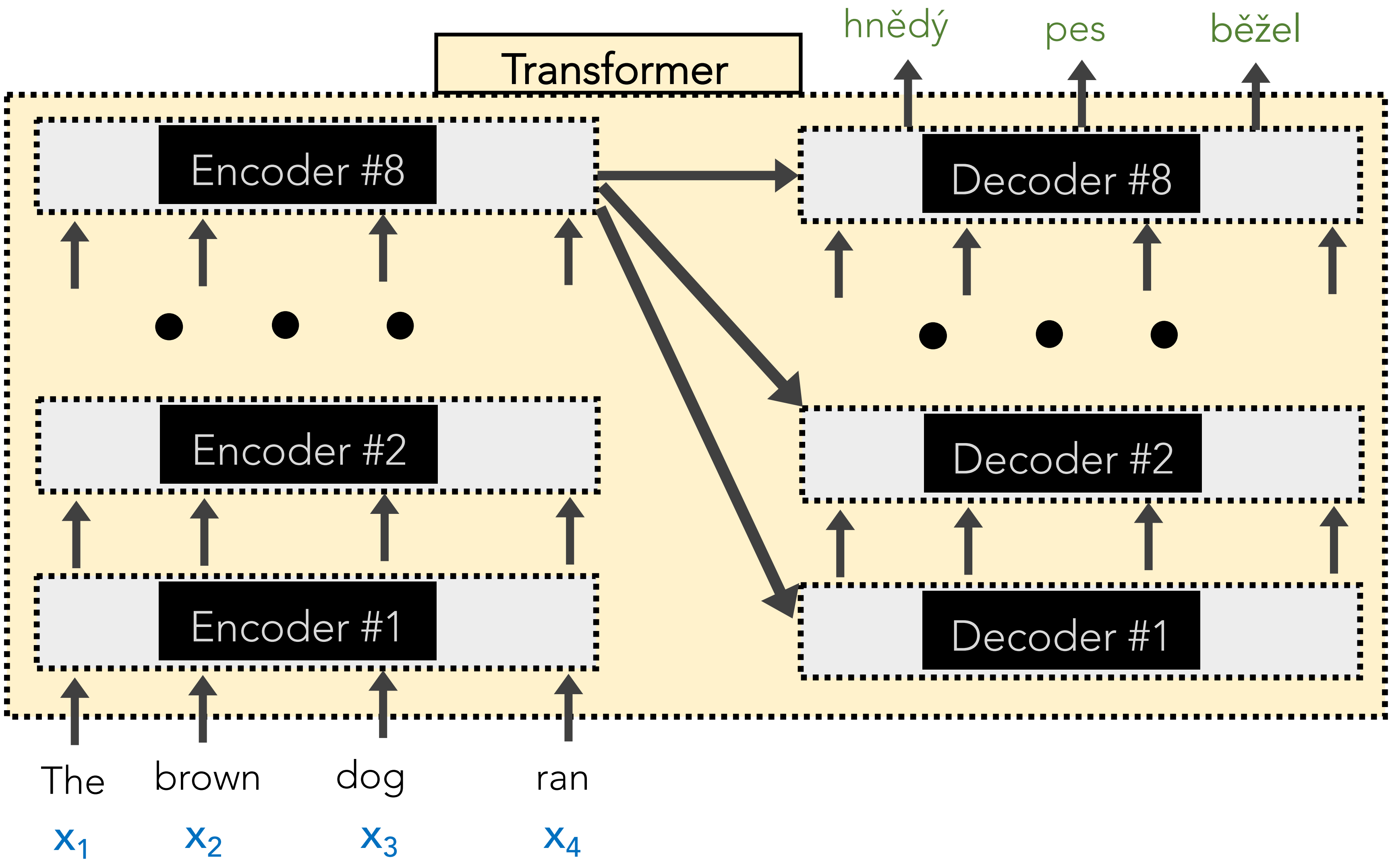
Transformer Encoders and Decoders



NOTE

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

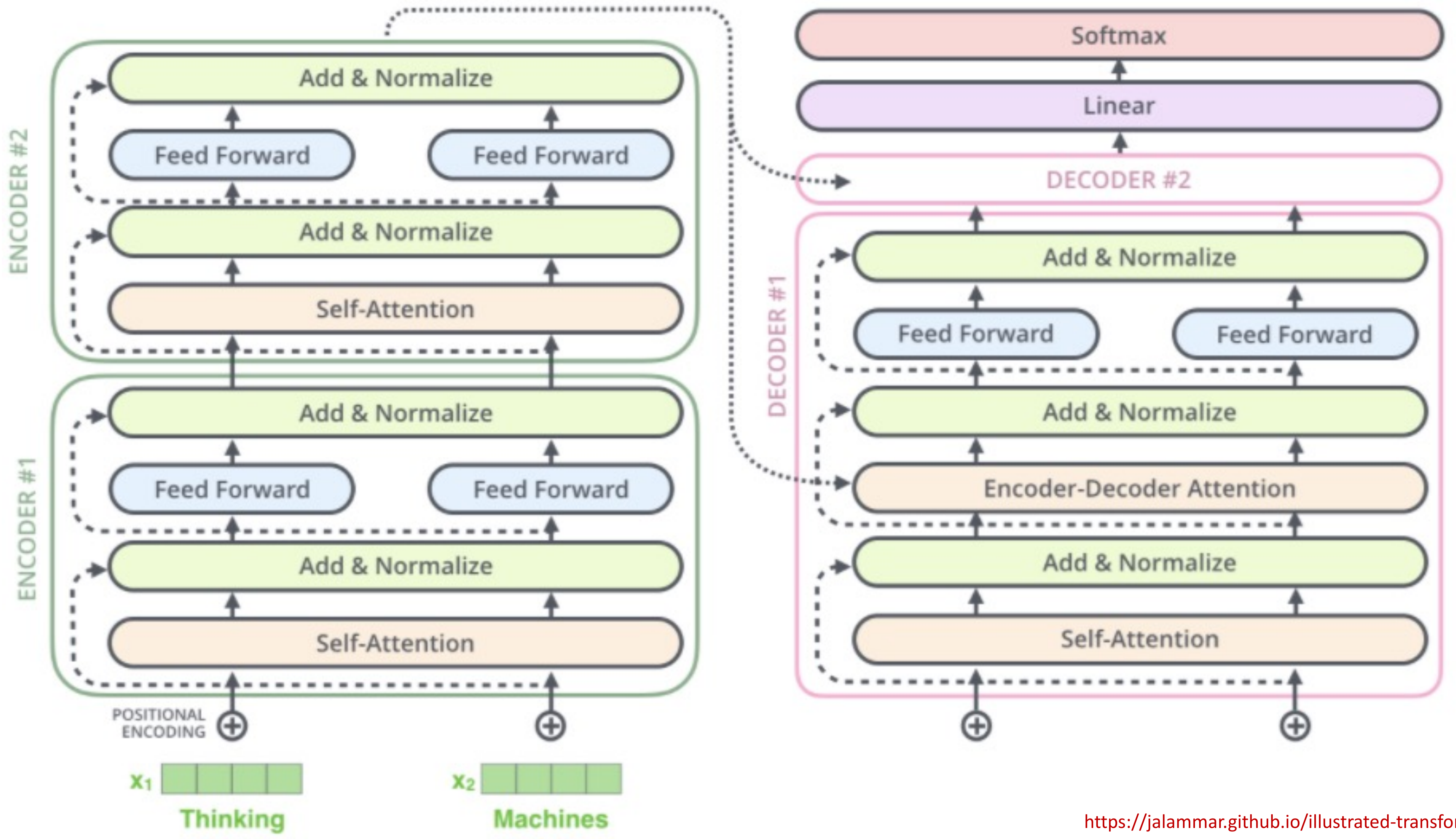
Transformer Encoders and Decoders



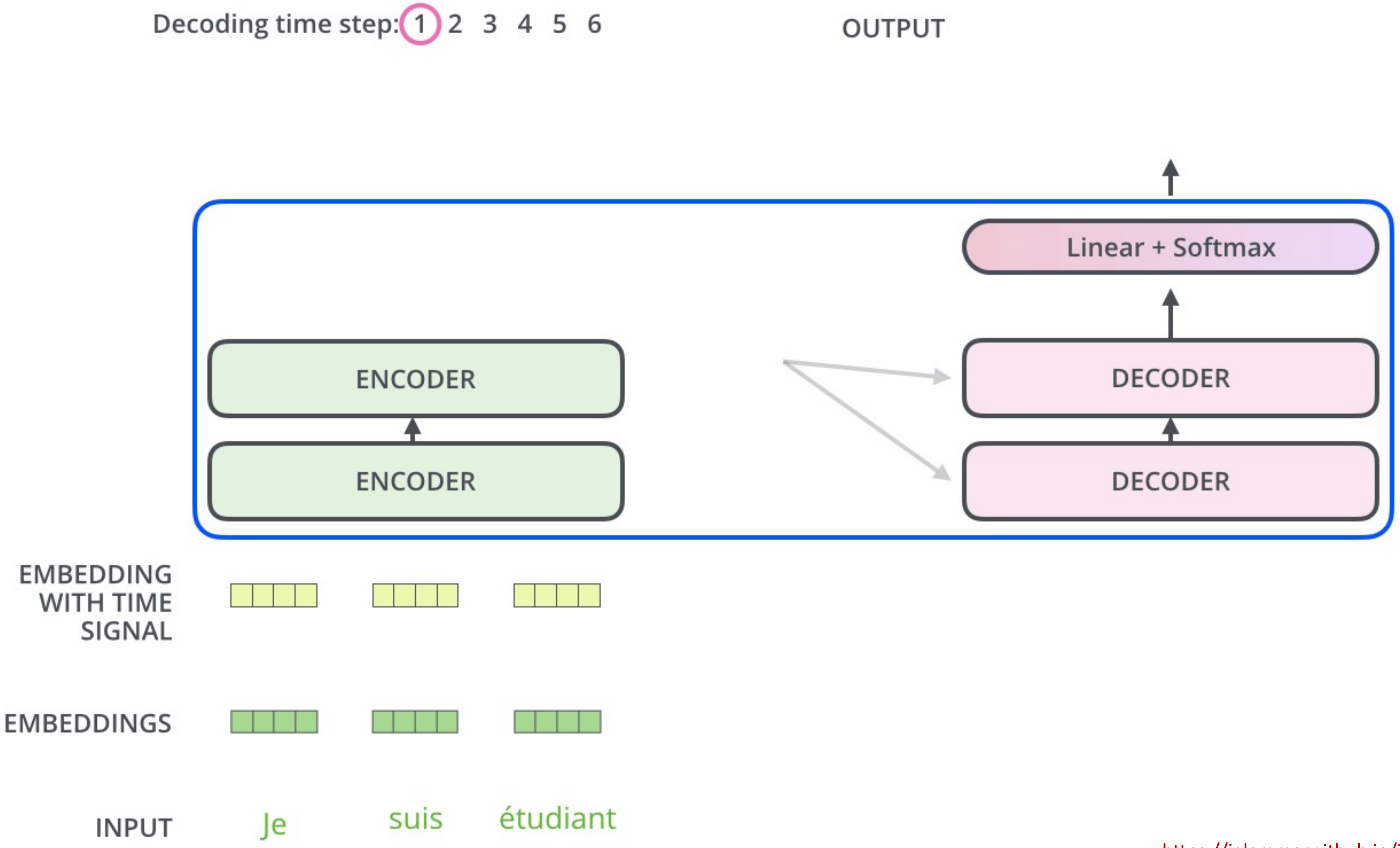
IMPORTANT

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

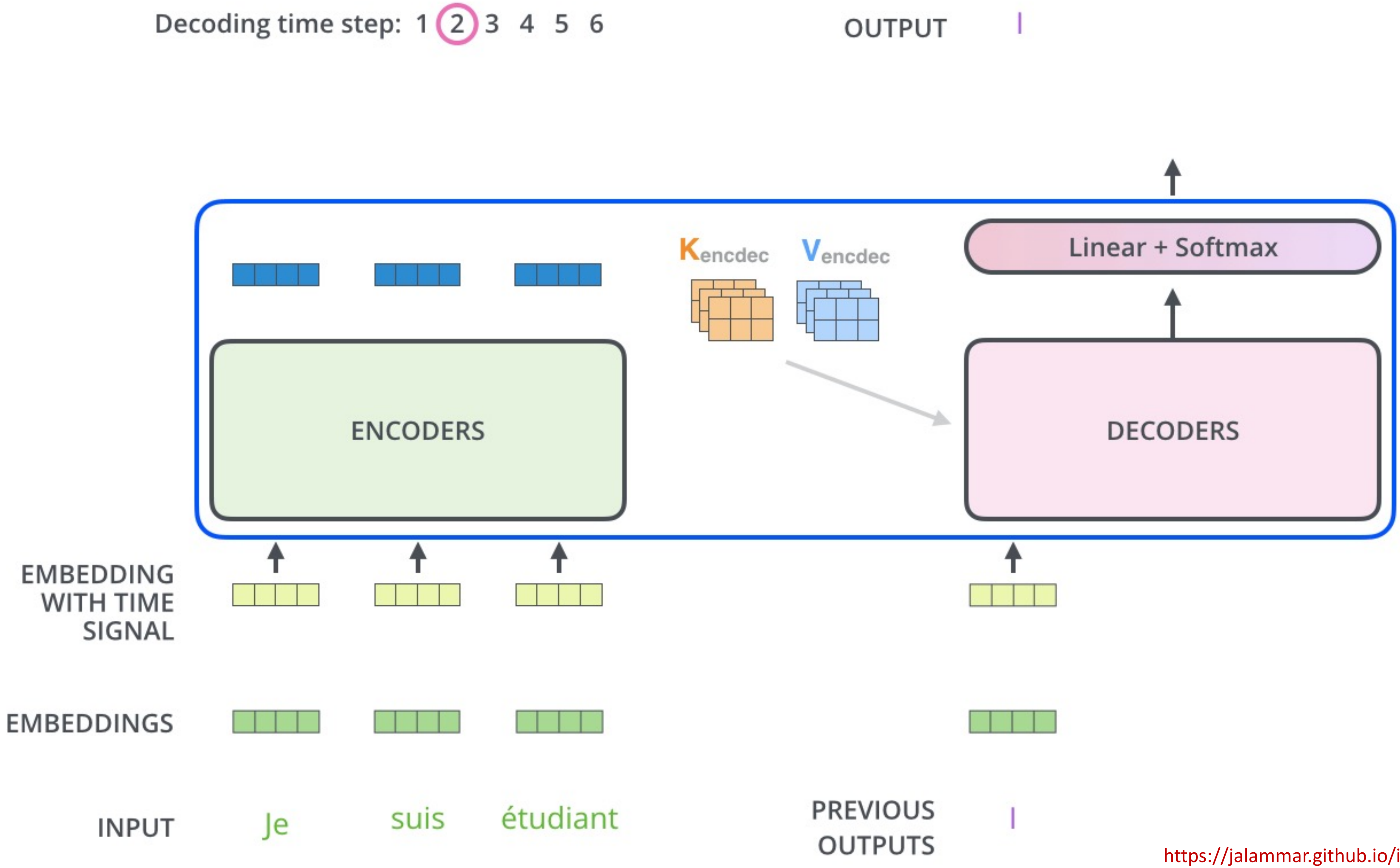
Critically, each position is only allowed to attend to the previous indices. This *masked* Attention preserves it as being an auto-regressive LM.



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



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



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

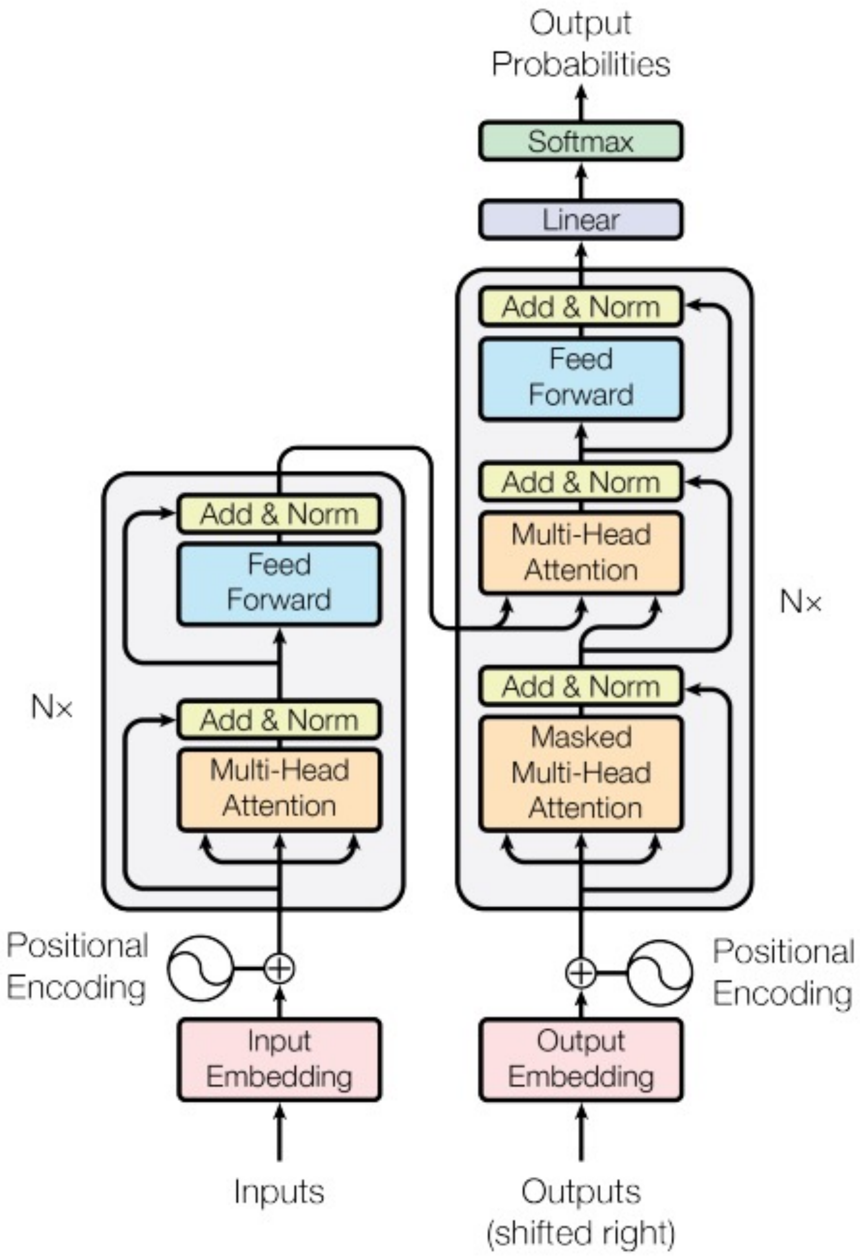


Figure 1: The Transformer - model architecture.

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

Loss Function: cross-entropy (predicting translated word)

Training Time: ~4 days on (8) GPUs

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

n = sequence length

d = length of representation (vector)

Q: Is the complexity of self-attention good?

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Important: when learning dependencies b/w words, you don't want long paths. Shorter is better.

Self-attention connects all positions with a constant # of sequentially executed operations, whereas RNNs require $O(n)$.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

https://arxiv.org/pdf/1706.03762.pdf

Machine Translation results: state-of-the-art (at the time)

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Machine Translation results: state-of-the-art (at the time)

You can train to translate from Language A to Language B.

Then train it to translate from Language B. to Language C.

Then, without training, it can translate from Language A to Language C

- What if we don’t want to decode/translate?
- Just want to perform a particular task (e.g., classification)
- Want even more robust, flexible, rich representation!
- Want positionality to play a more explicit role, while not being restricted to a particular form (e.g., CNNs)

Outline

- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

Outline

- Self-Attention
- Transformer Encoder
- Transformer Decoder
- BERT

BERT

Bidirectional Encoder Representations from Transformers

BERT

Bidirectional Encoder Representations from Transformers

Like *Bidirectional LSTMs*, let's look in **both** directions

BERT

Bidirectional Encoder Representations from Transformers

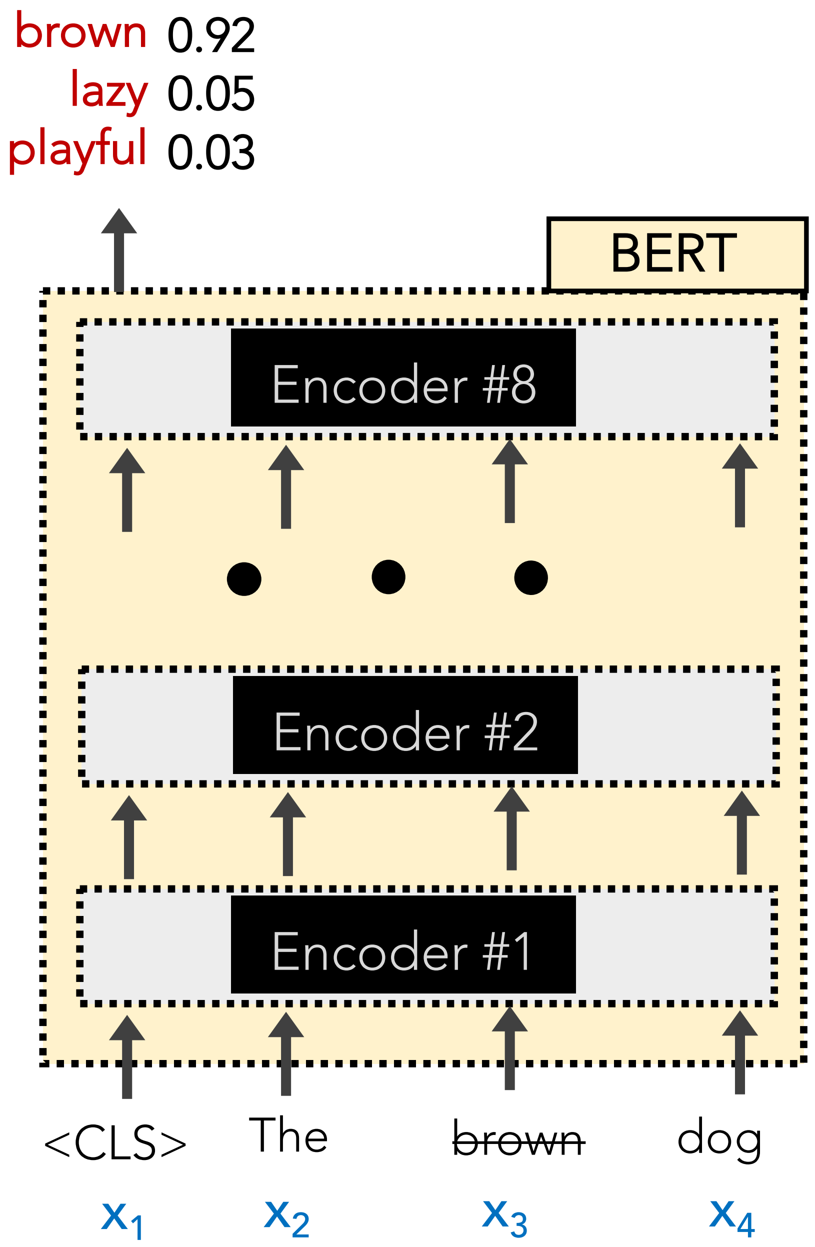
Let's only use Transformer *Encoders*, no Decoders

BERT

Bidirectional Encoder Representations from Transformers

It's a language model that builds rich representations

BERT



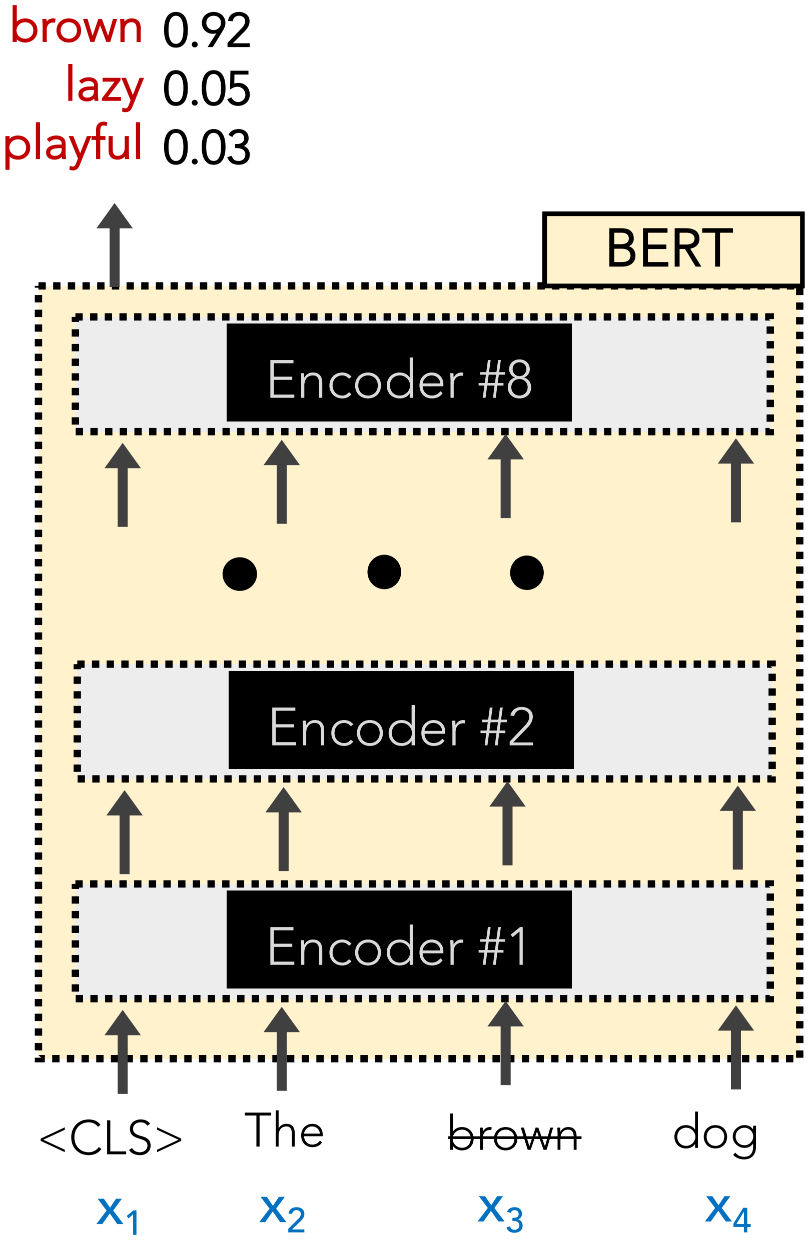
BERT has 2 training objectives:

1. Predict the **Masked word** (a la CBOW)

15% of all input words are randomly masked.

- 80% become [MASK]
- 10% become revert back
- 10% become are deliberately corrupted as wrong words

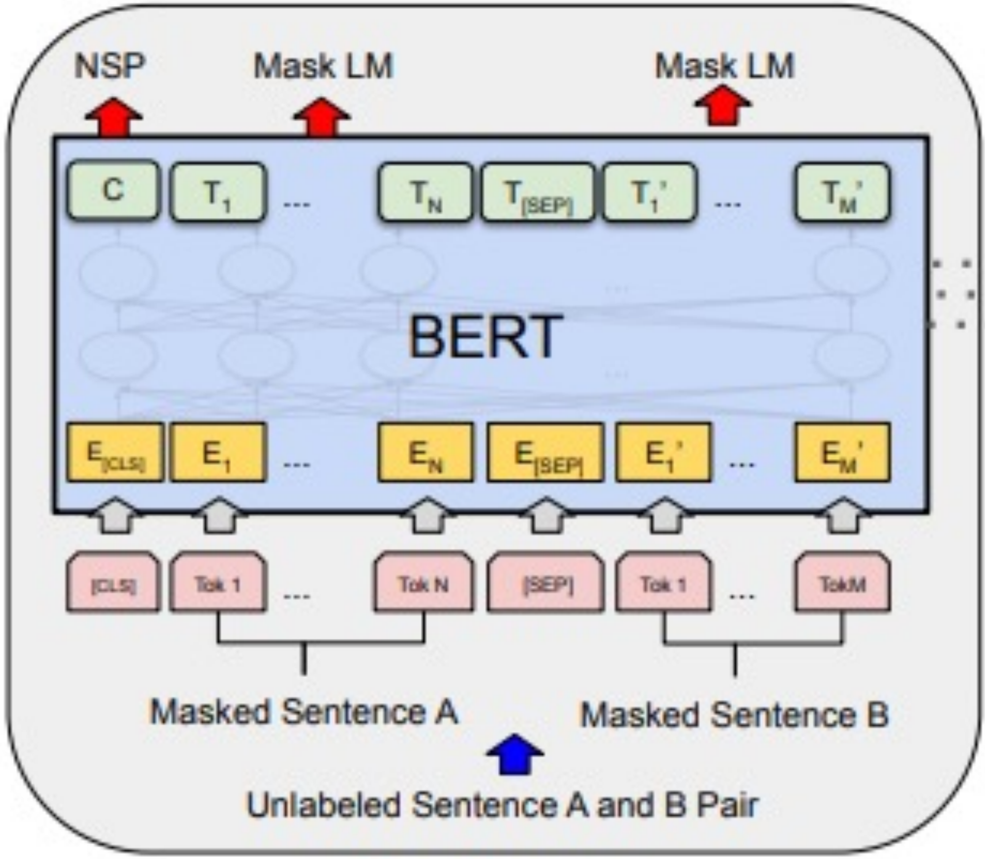
BERT



BERT has 2 training objectives:

2. Two sentences are fed in at a time. Predict the if the second sentence of input truly follows the first one or not.

BERT



Pre-training

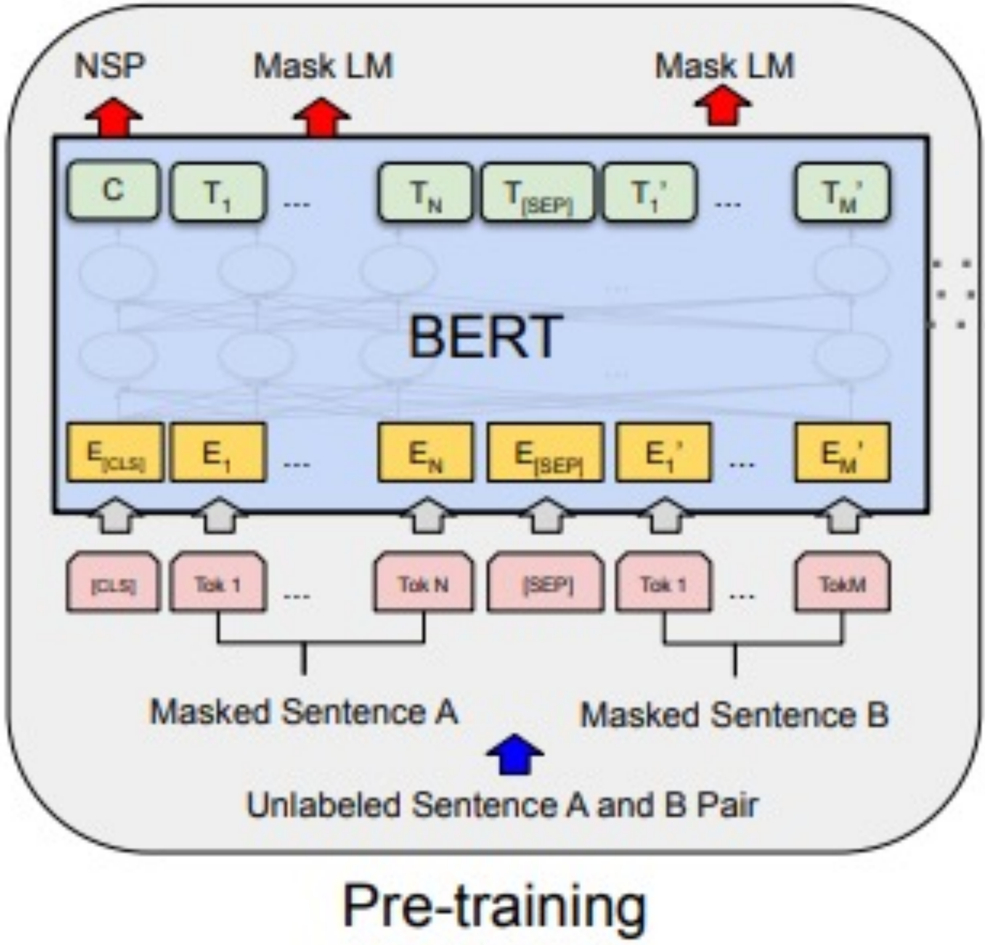
Every two sentences are separated by a <SEP> token.

50% of the time, the 2nd sentence is a randomly selected sentence from the corpus.

50% of the time, it truly follows the first sentence in the corpus.

Source: original BERT paper: https://arxiv.org/pdf/1810.04805.pdf

BERT



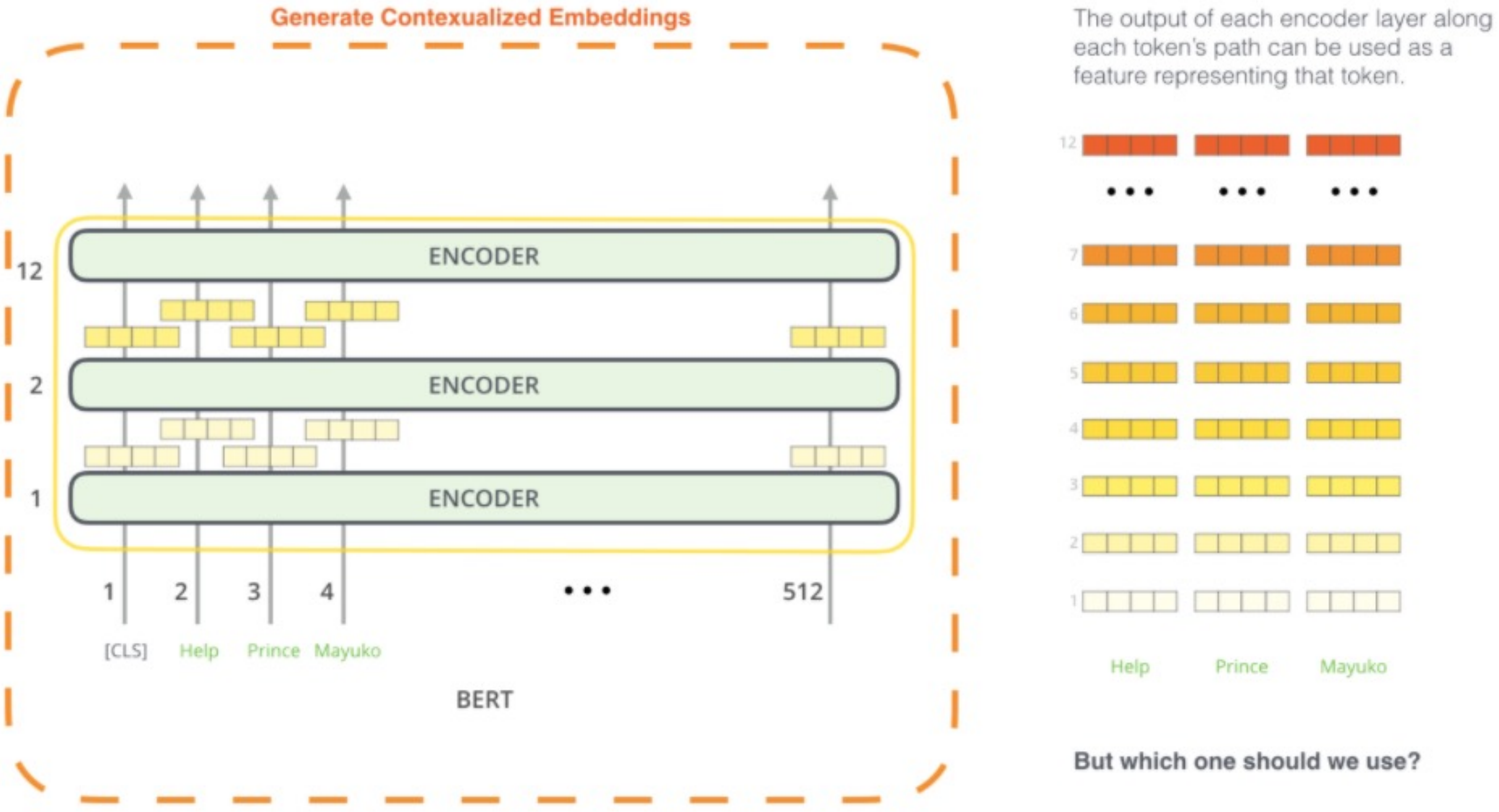
NOTE: BERT also embeds the inputs by their WordPiece embeddings.

WordPiece is a sub-word tokenization learns to merge and use characters based on which pairs maximize the likelihood of the training data if added to the vocab.

Source: original BERT paper: https://arxiv.org/pdf/1810.04805.pdf

BERT

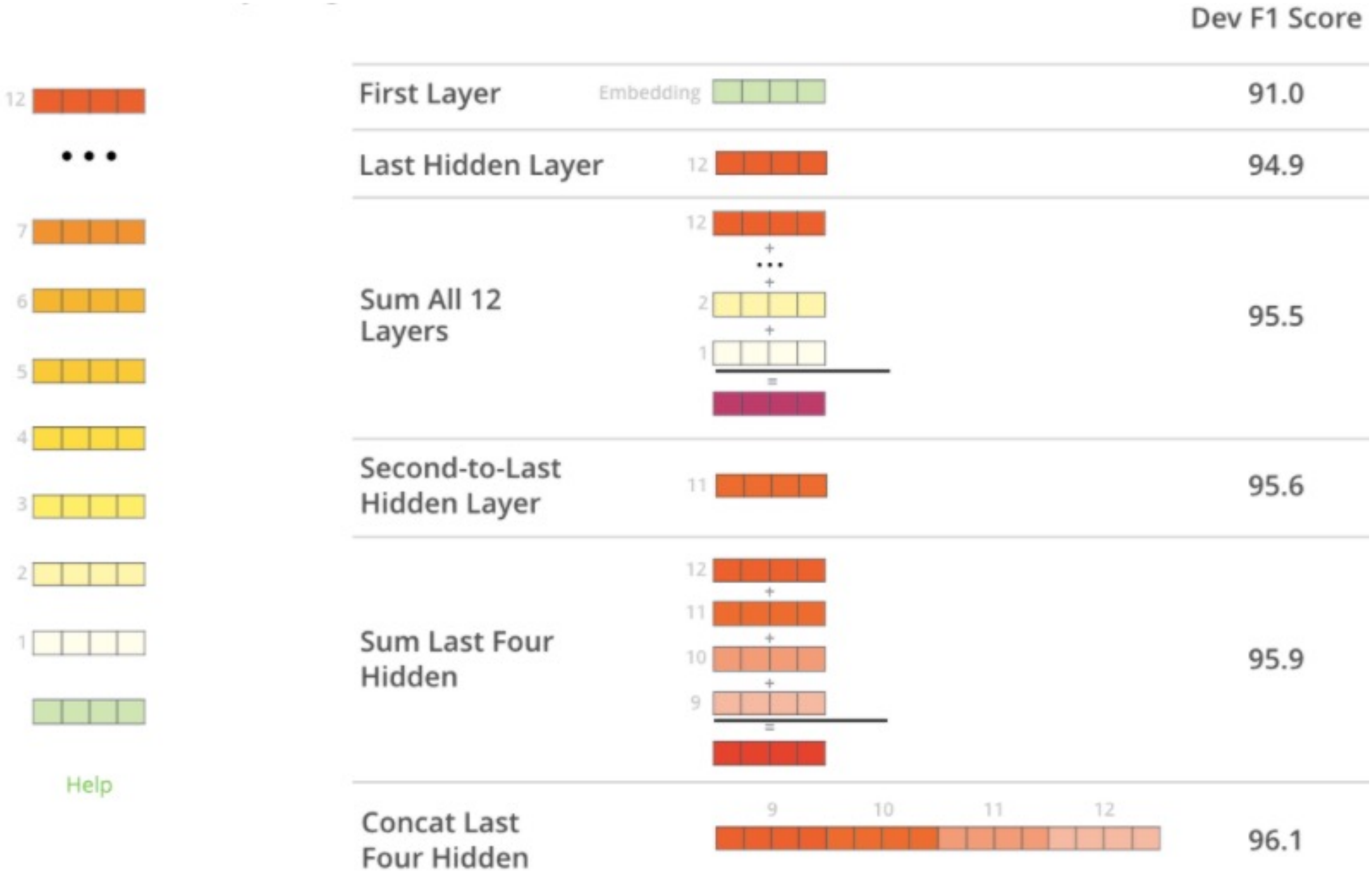
One could extract the contextualized embeddings



Picture: https://jalammar.github.io/illustrated-bert/

BERT

Later layers have the best contextualized embeddings



Picture: https://jalammar.github.io/illustrated-bert/

BERT yields state-of-the-art (SOTA) results on many tasks

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	**Average** -
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**92.7**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**82.1**

Table 1: GLUE Test results, scored by the evaluation server (`https://gluebenchmark.com/leaderboard`).

Source: original BERT paper: https://arxiv.org/pdf/1810.04805.pdf

Takeaway

BERT is incredible for learning **contextualized embeddings** of words and using transfer learning for other tasks (e.g., classification).

Can't generate new sentences though, due to no decoders.