CSCI 1470

Eric Ewing

Friday,
3/14/25

Deep Learning

Day 22: Transformers

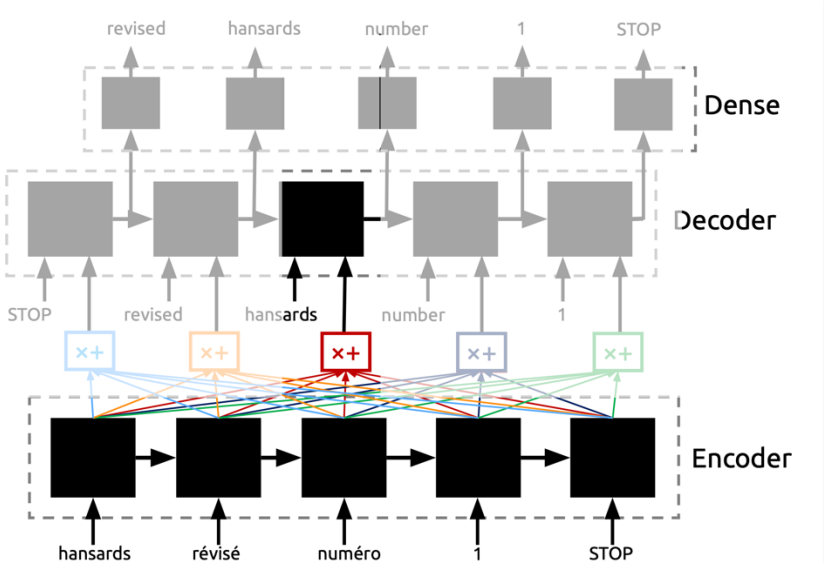
Transformers' home: Cybertron

The Plan

- Finish Supervised Learning portion of class this week
  - Today: Transformers
  - Friday: LLMs and GPT
- After Break:
  - Unsupervised Learning
  - Reinforcement Learning
- Weekly Quiz is out

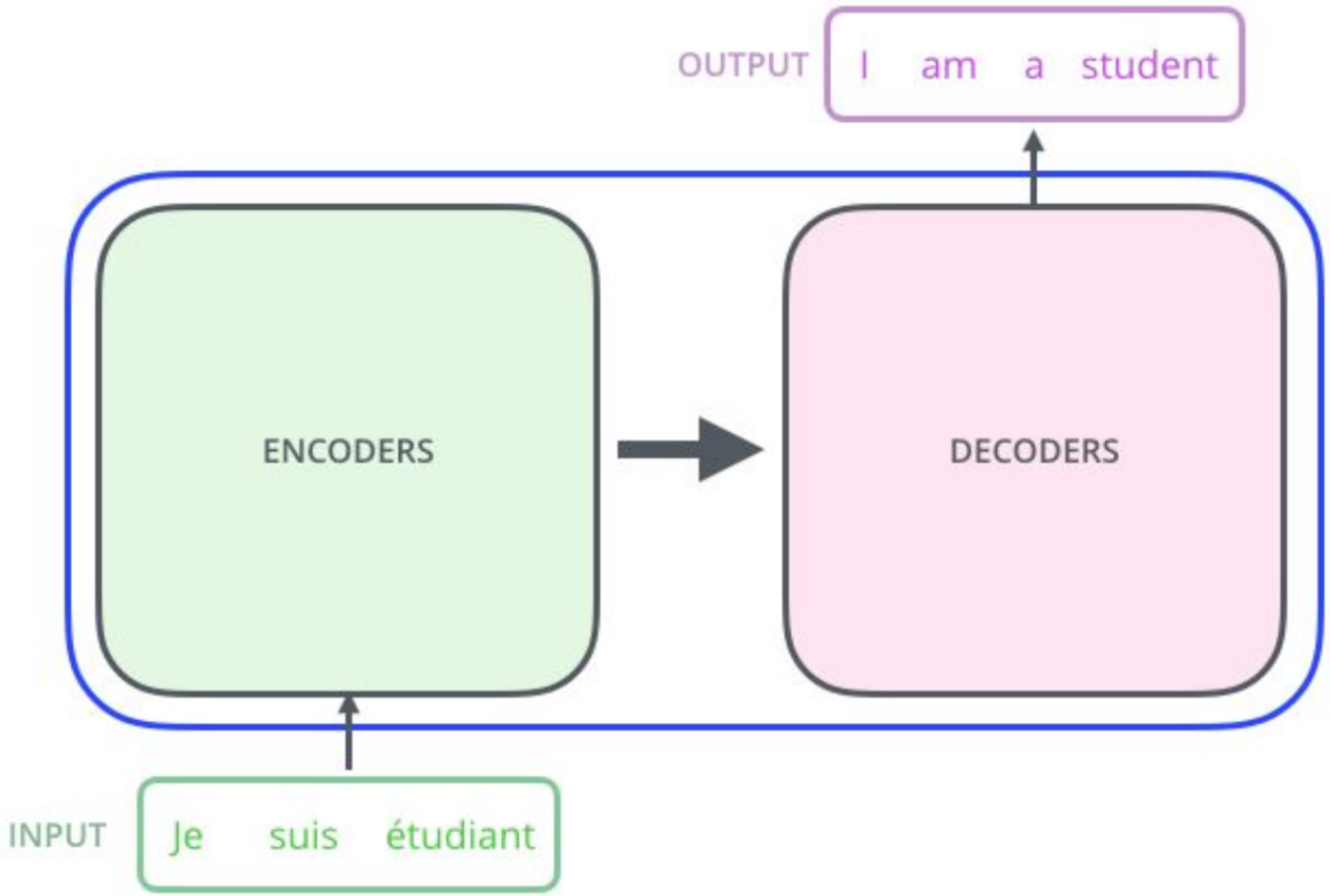
Review

- Seq2Seq modeling task using encoder-decoder architecture
- Attention measures similarity between embeddings, calculates scores, and ends with a weighted sum of vectors.



Transformer Model Overview

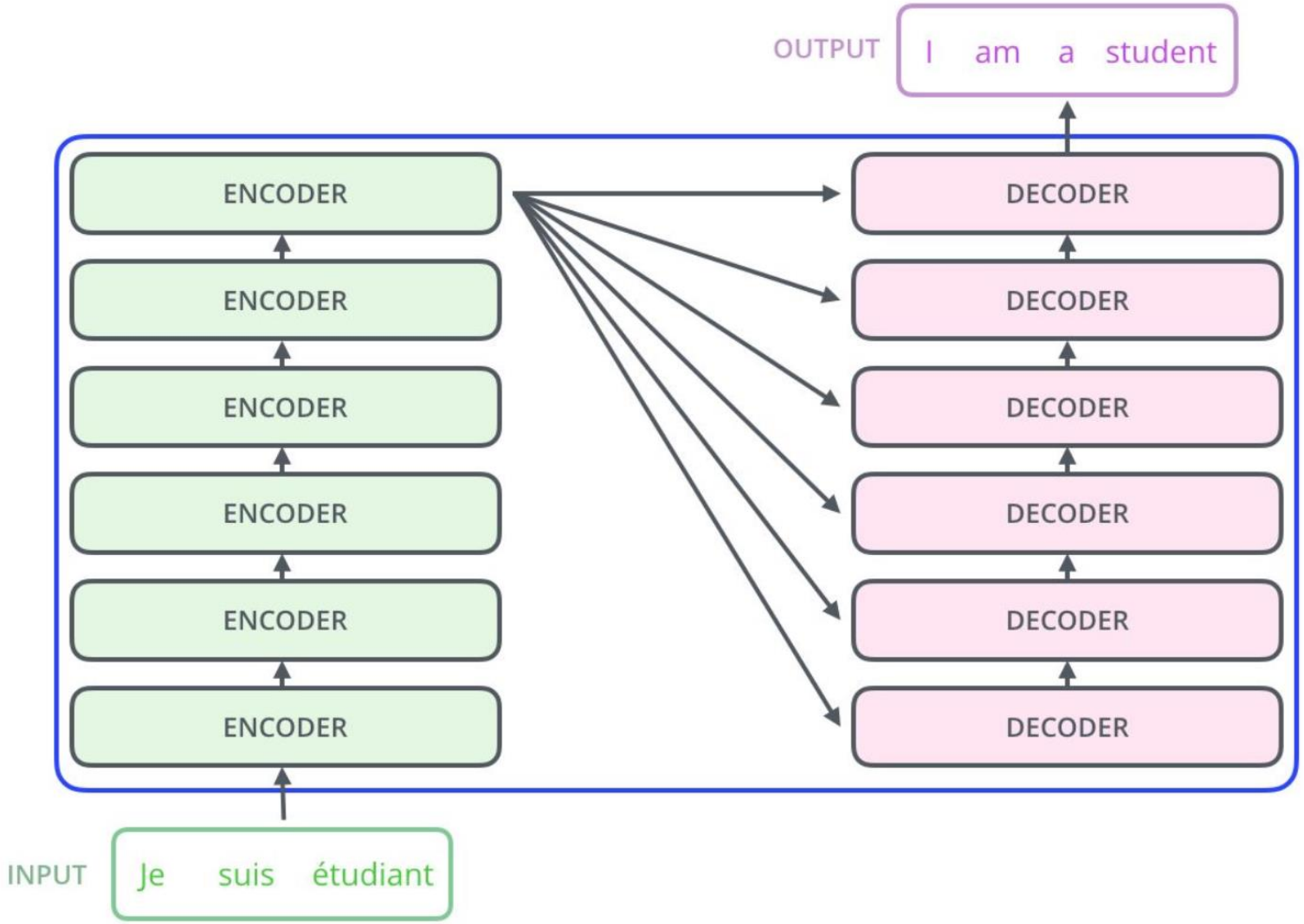
- The Transformer model breaks down into Encoder and Decoder blocks.
- At a high level, similar to the seq2seq architecture we've seen already...
- ...but there are no recurrent nets inside the Encoder and Decoder blocks!



Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Transformer Model Overview

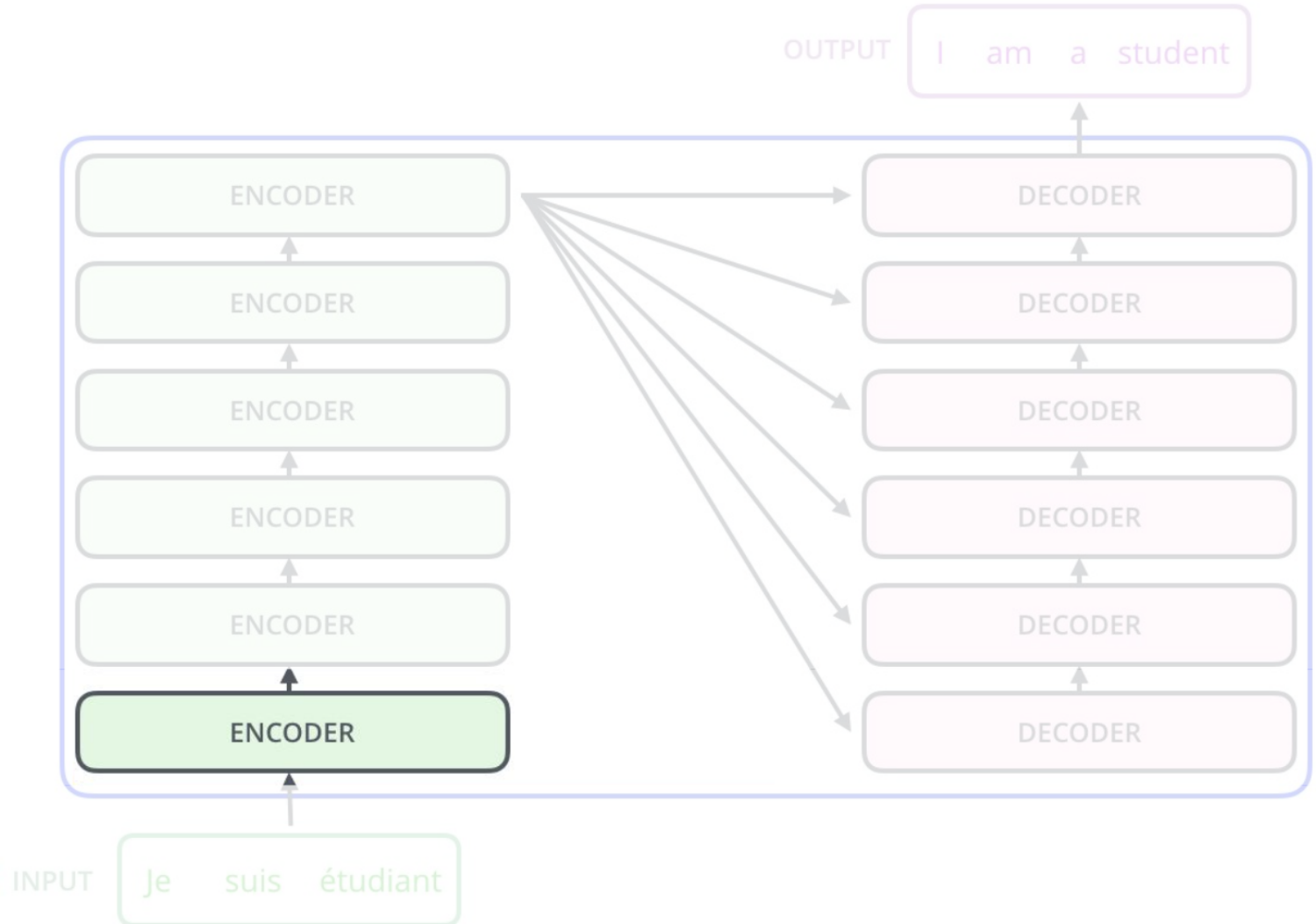
- The Transformer model breaks down into Encoder and Decoder blocks.
- At a high level, similar to the seq2seq architecture we've seen already...
- ...but there are no recurrent nets inside the Encoder and Decoder blocks!
- For better performance, often stack multiple Encoder and Decoder blocks (deeper network)



Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Transformer Model Overview

- Let's look at what goes on inside one of these Encoder blocks

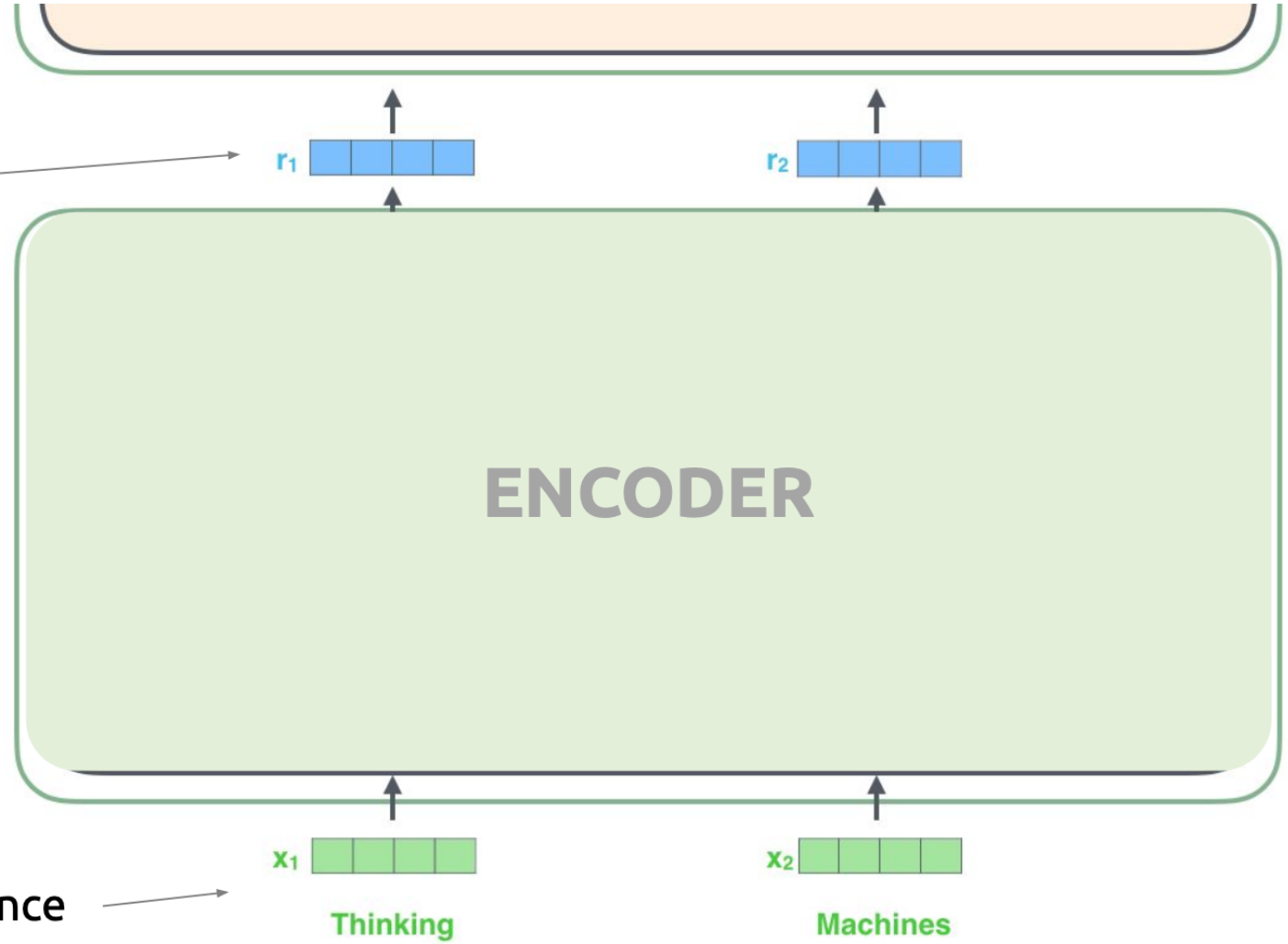


Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Encoder Block Map

These per-word output vectors are analogous to the LSTM hidden states from the seq2seq2 model

- They should capture *"what information about the input sentence is relevant to translating this word?"*

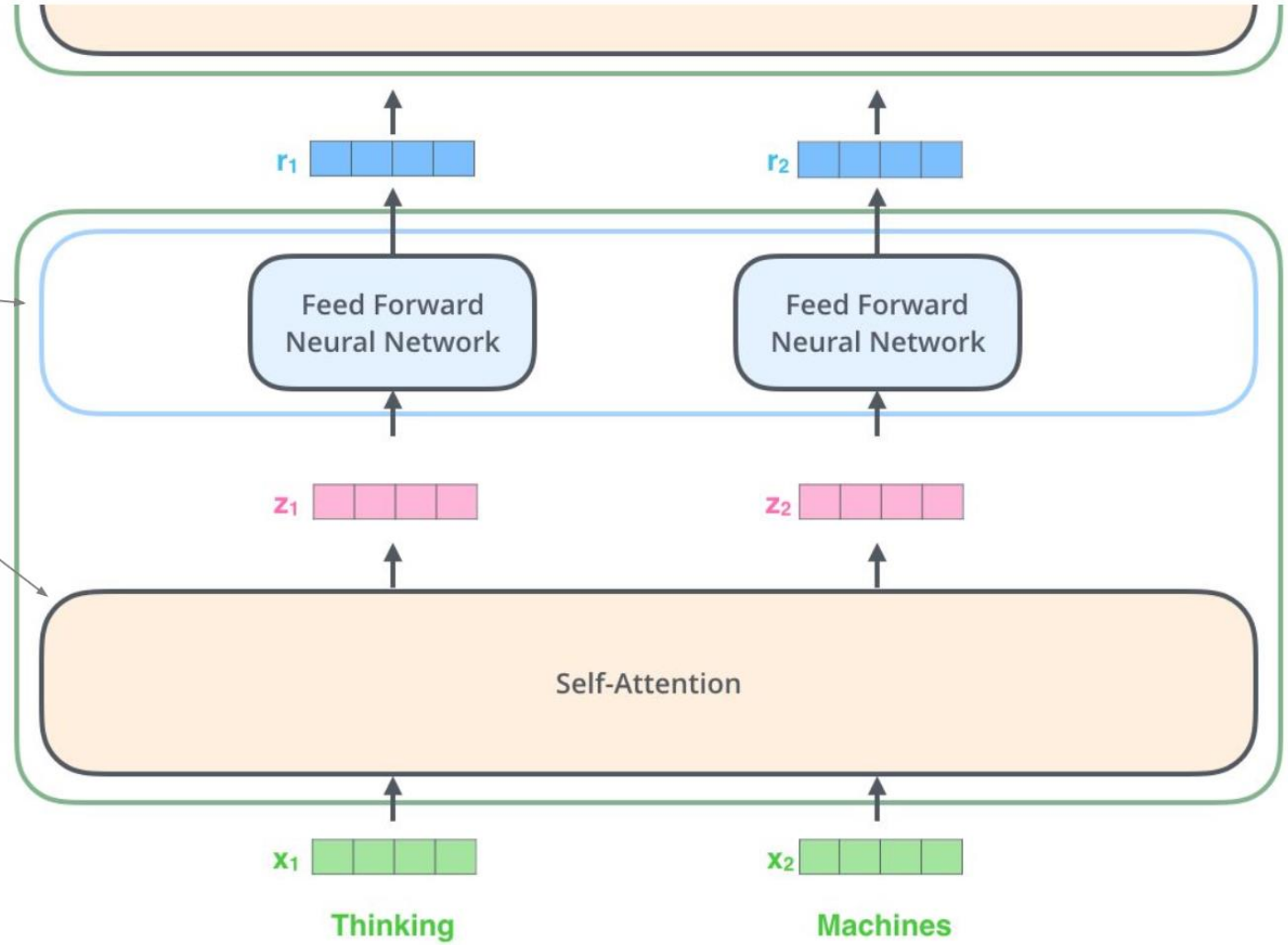


Words in input sentence

Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Encoder Block Map

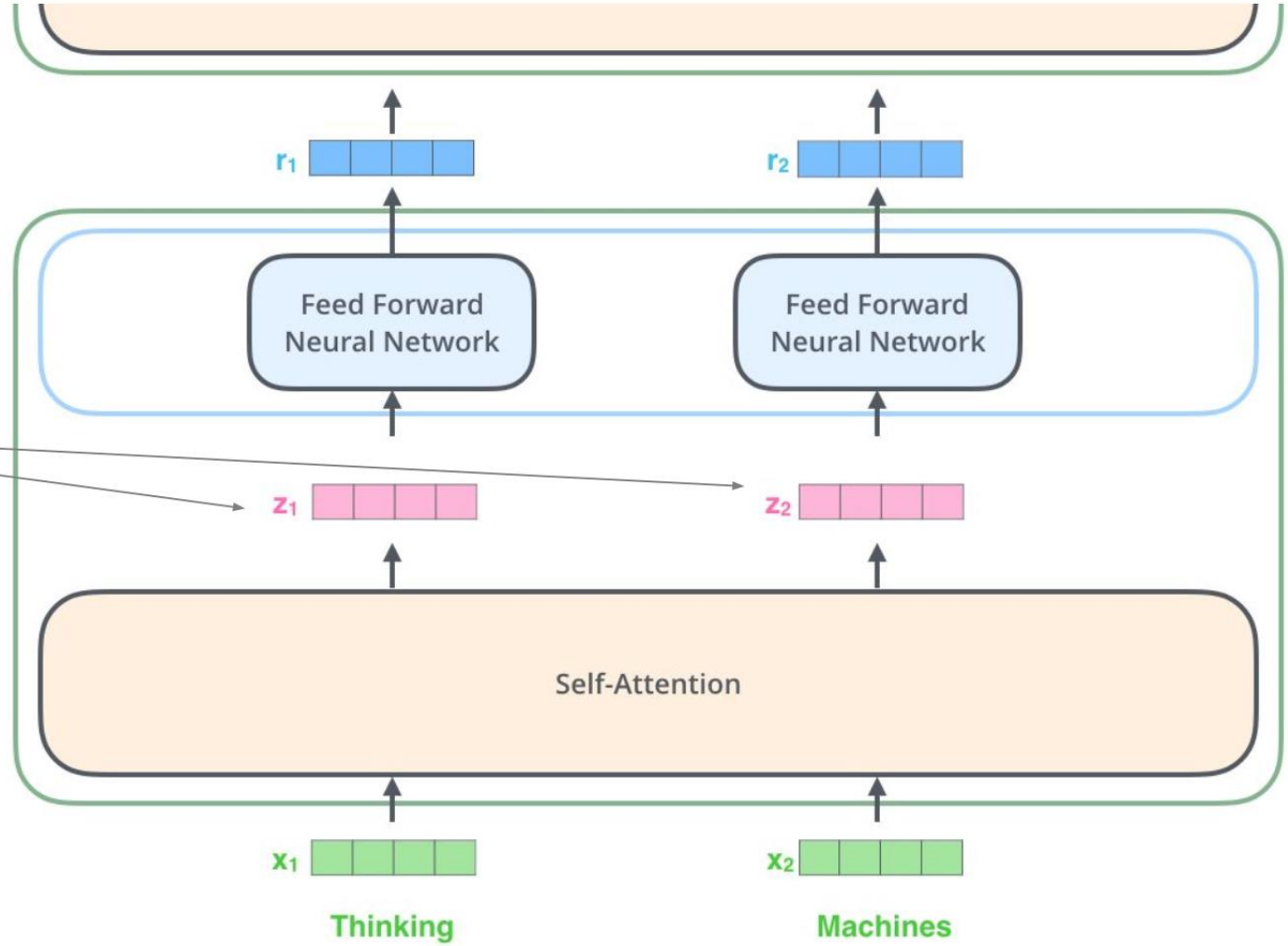
- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.



Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

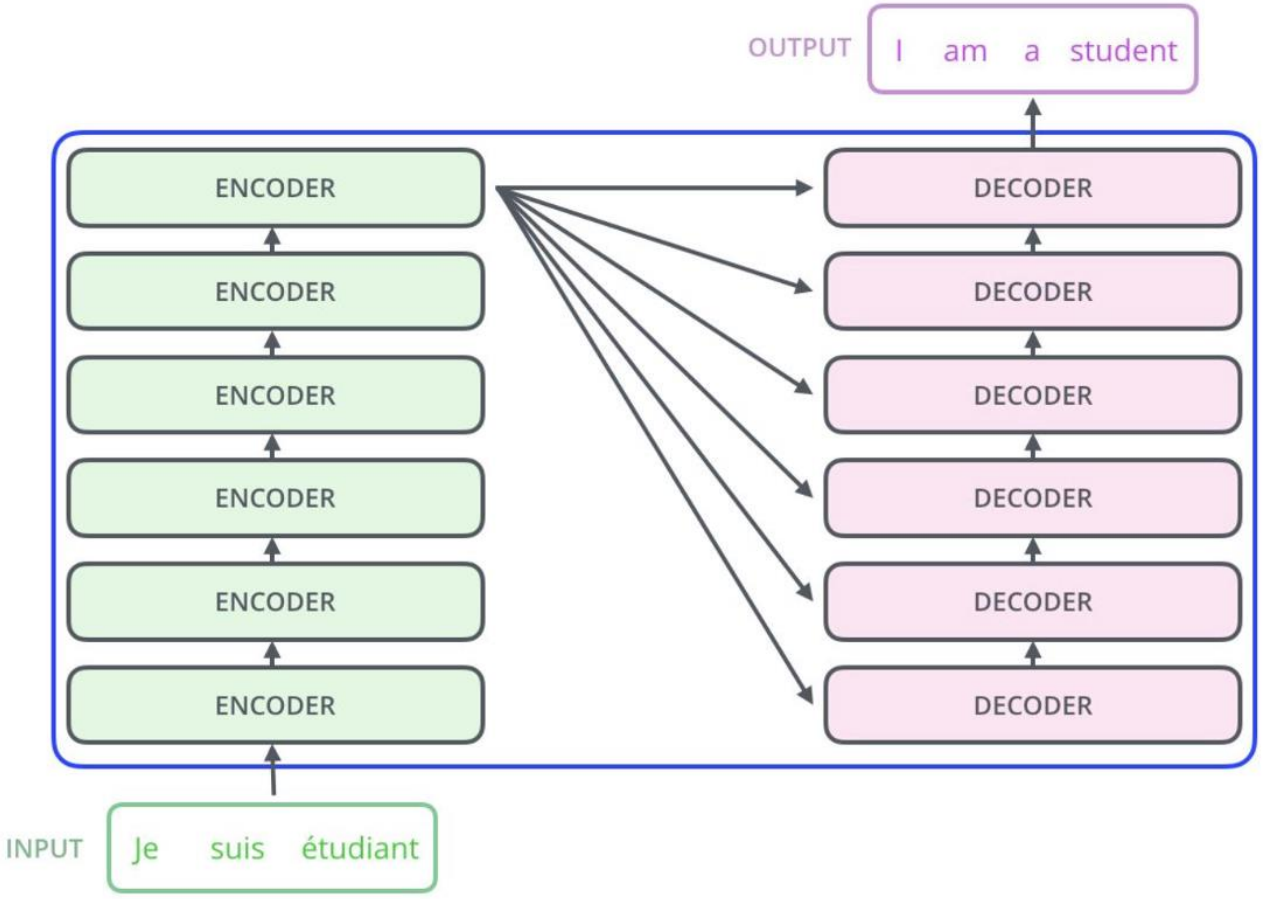
Encoder Block Map

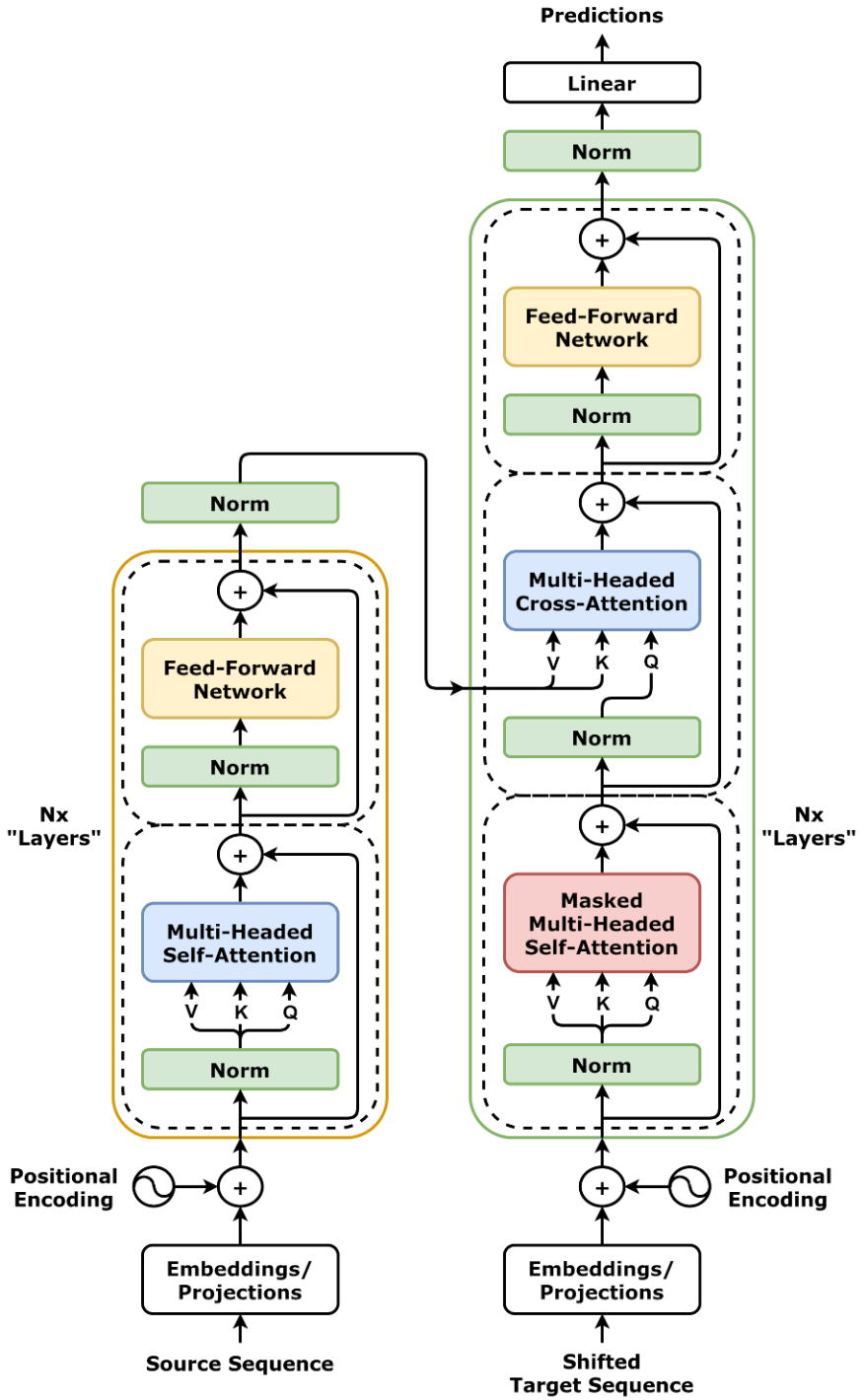
- Encoder block breaks down into two main parts: Self-Attention, and Feed Forward layers.
- Self-Attention layer is applied to each word individually.



Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

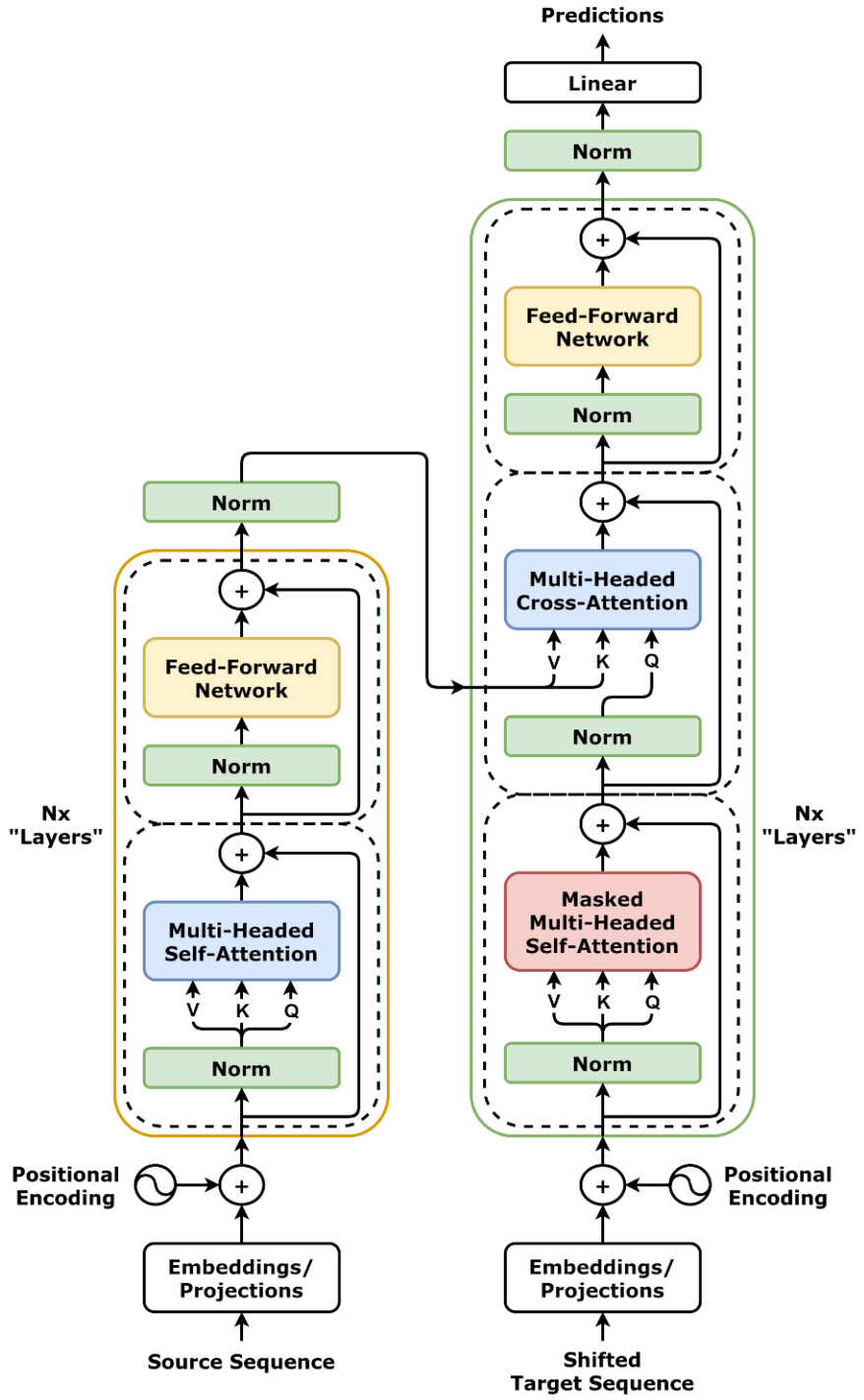
The Transformer



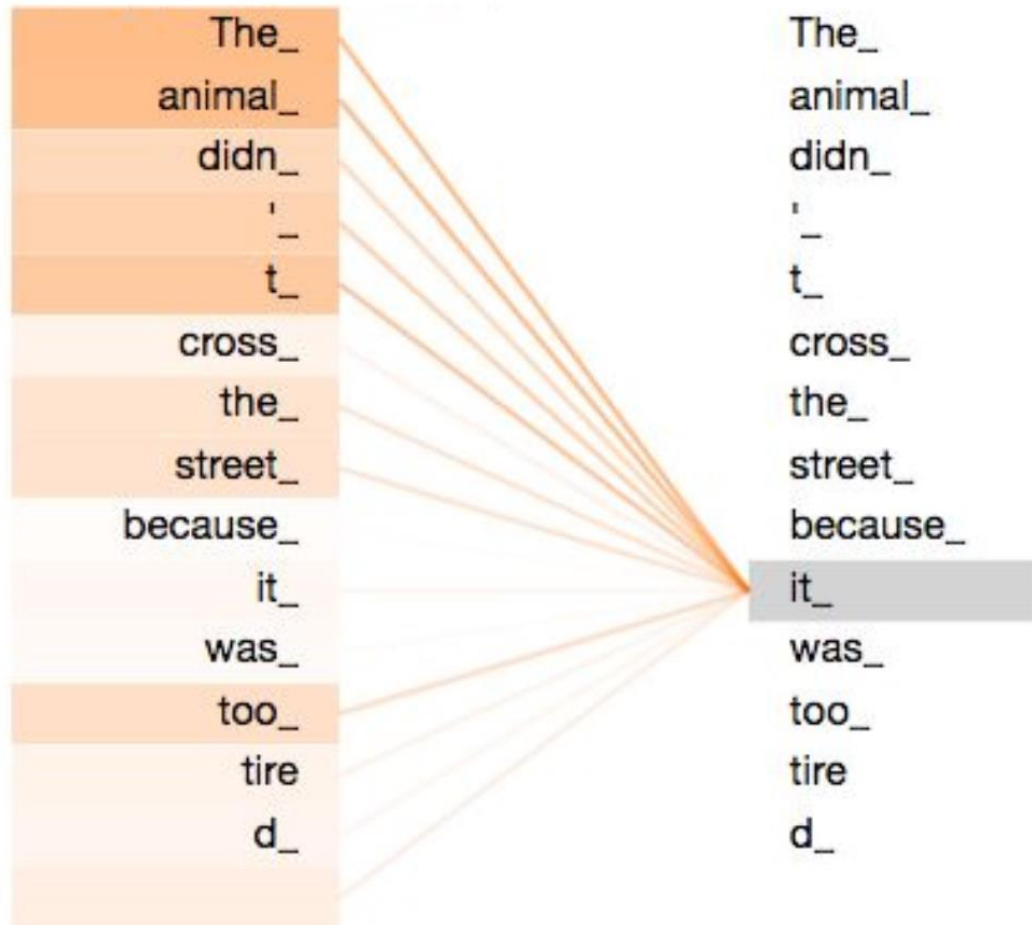


Components

- Self-Attention
- Cross-Attention
- Position Encoding
- Norm
- Feed-Forward Networks



Self-Attention: Input's attention on itself



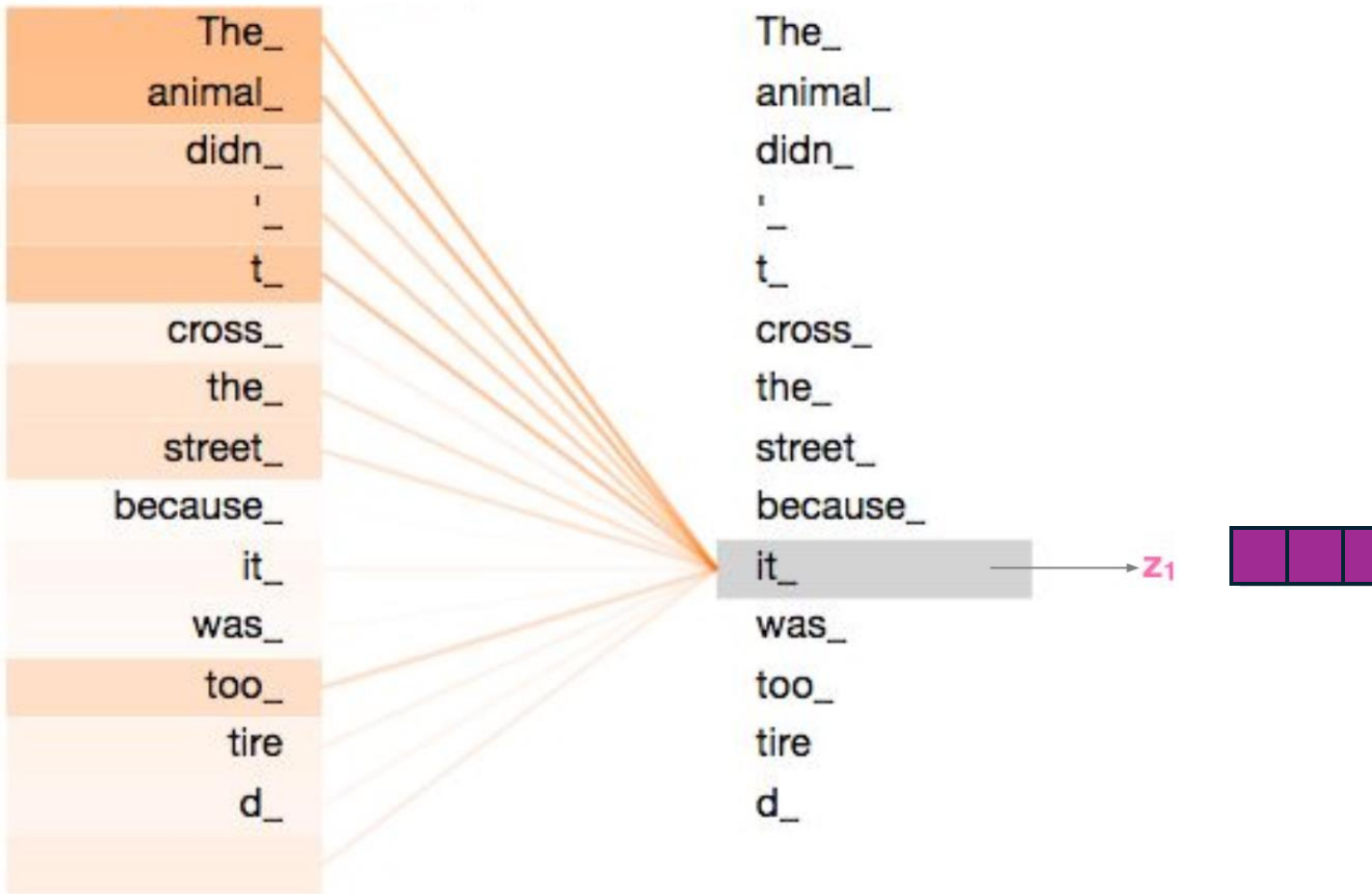
Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Self-Attention: Overview

- **The big idea:**
  Self-attention computes the output vector z_i for each word via a weighted sum of vectors extracted from each word in the input sentence

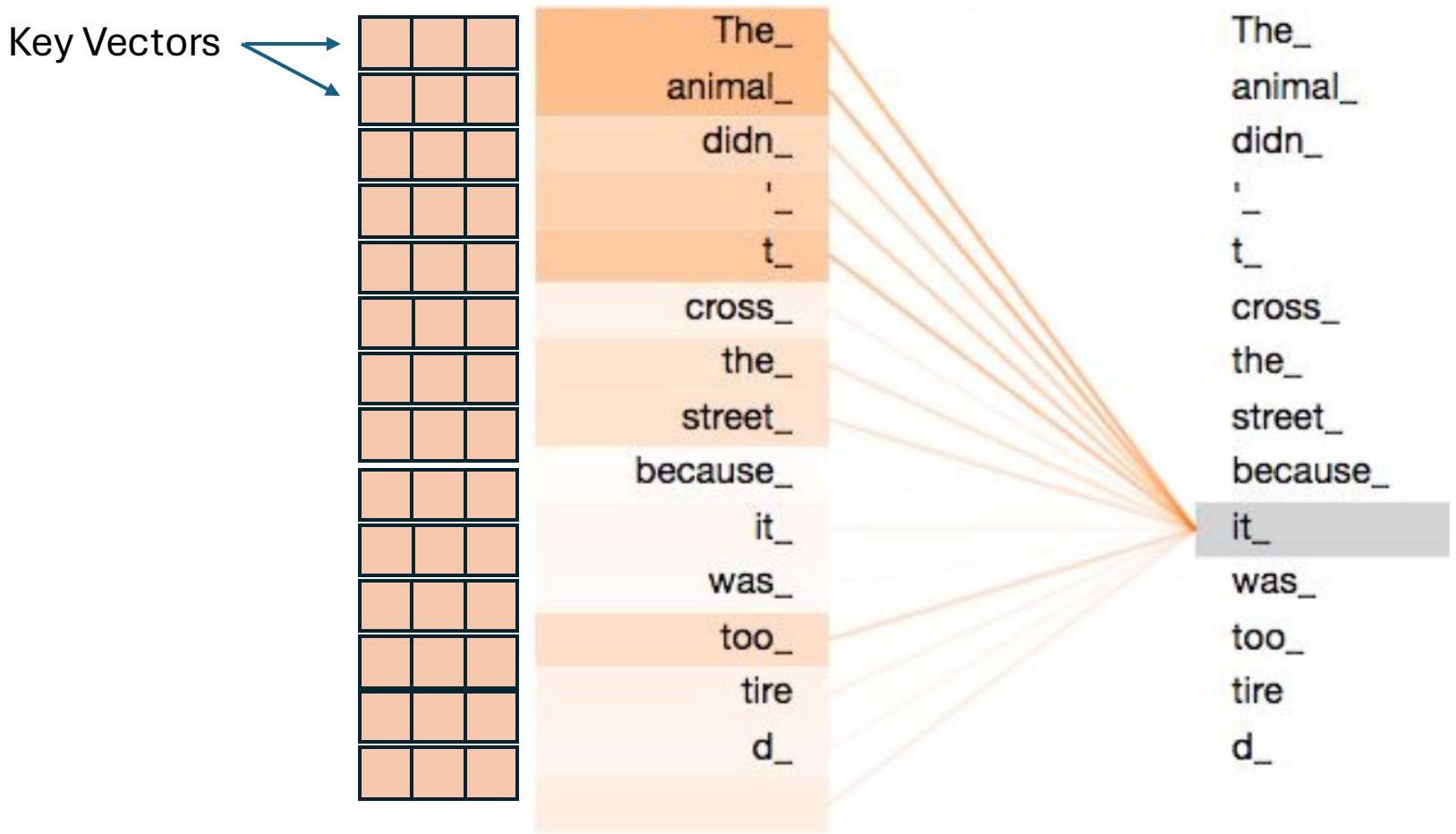
- Here, self-attention learns that “it” should pay attention to “the animal” (i.e. the entity that “it” refers to)

- Why the name ***self***-attention?
  This describes attention that the input sentence pays to itself



Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

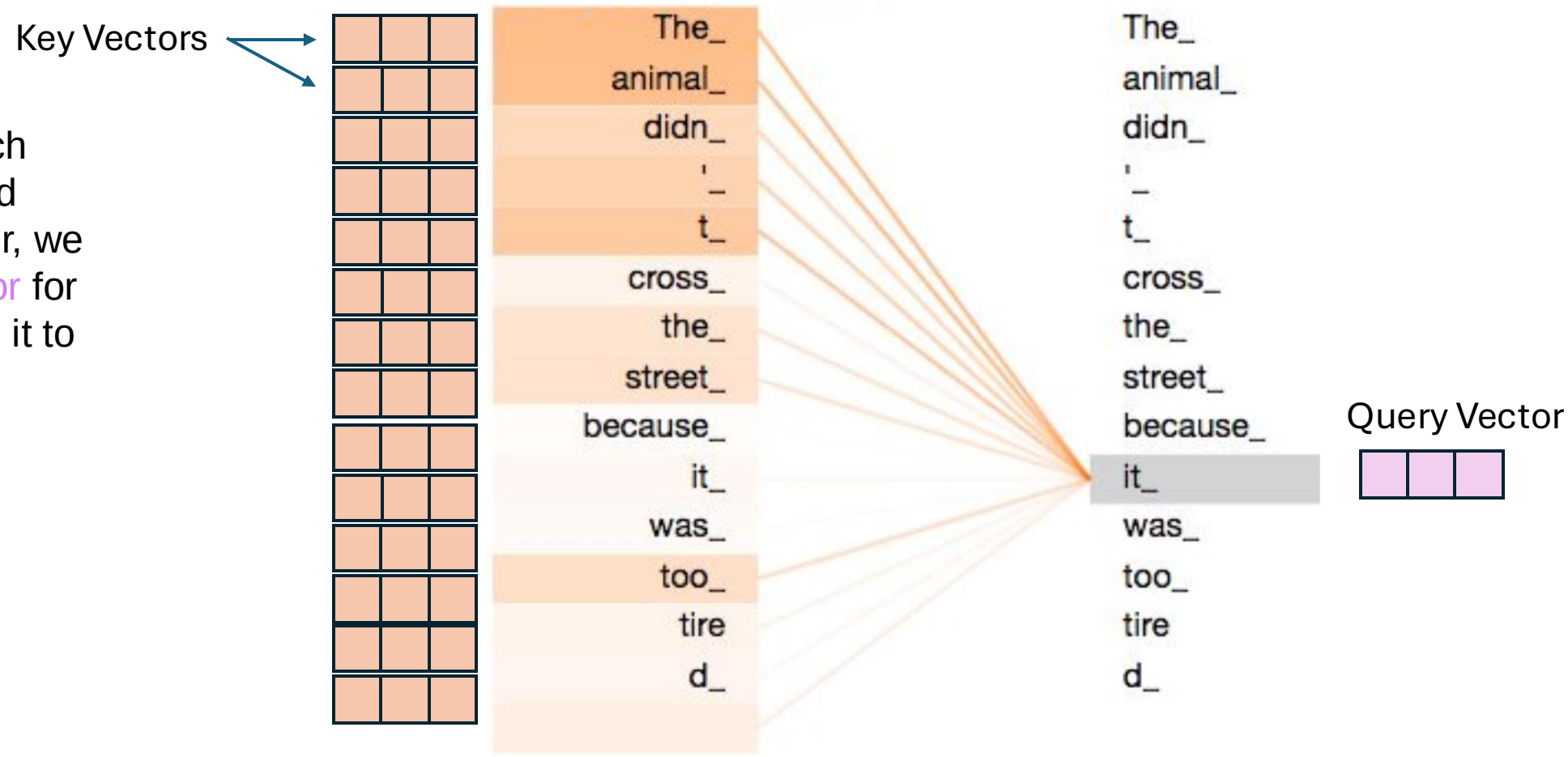
Self-Attention: Input's attention on itself



Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Self-Attention: Input's attention on itself

To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...



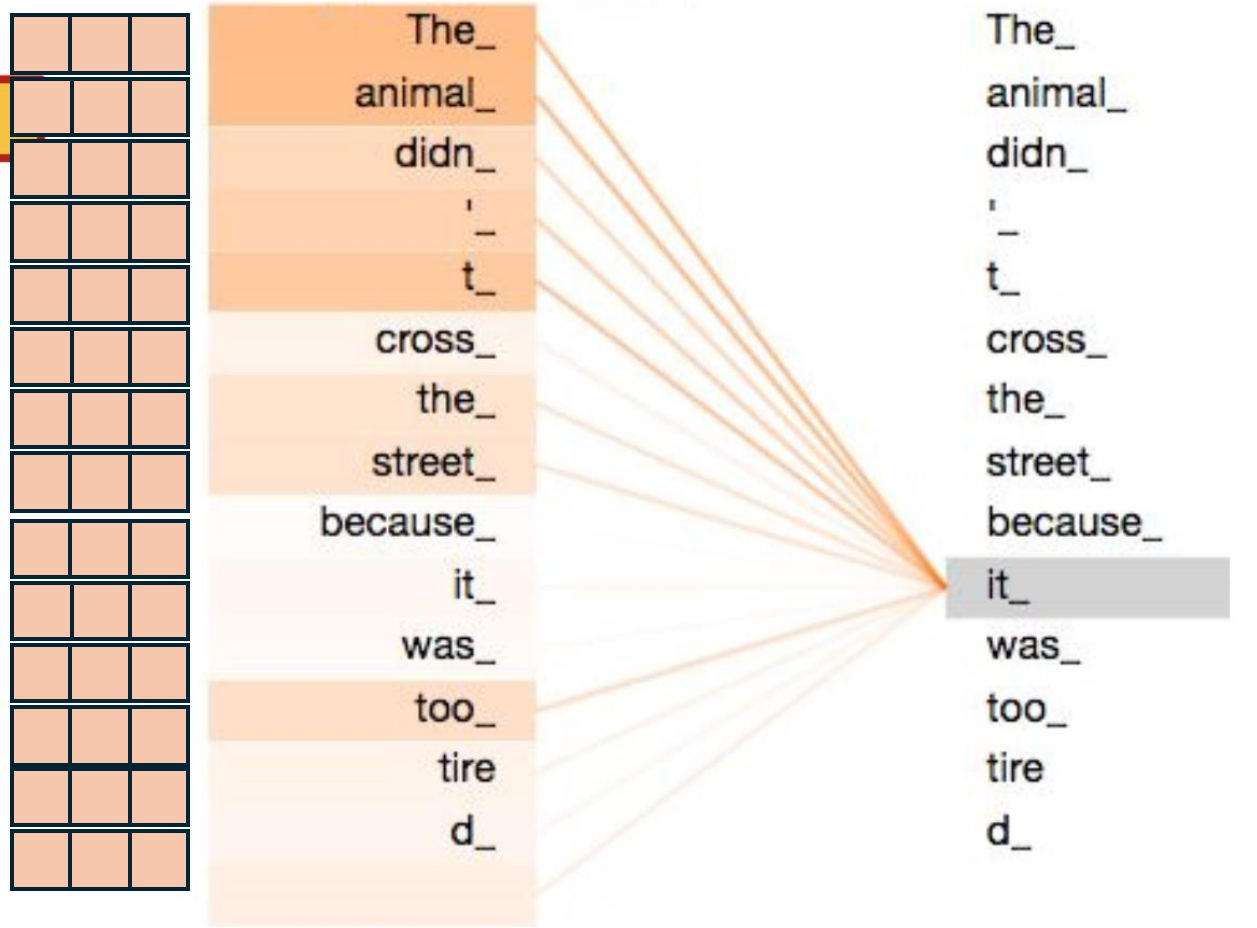
Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Self-Attention: Input's attention on itself

Key Vectors

What do we do next?

To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...



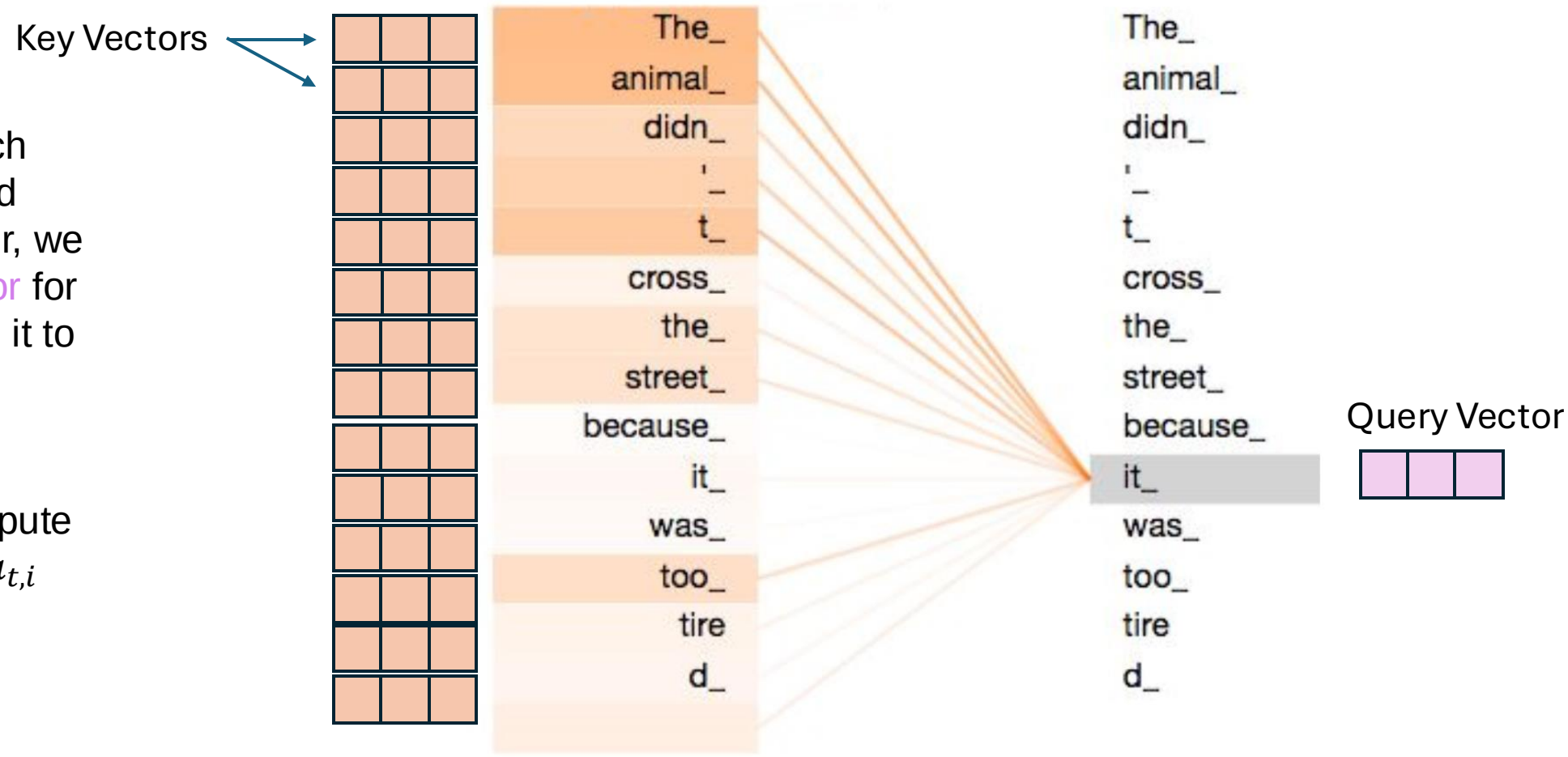
Query Vector

Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Self-Attention: Input's attention on itself

To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...

Which use use to compute the alignment scores $a_{t,i}$

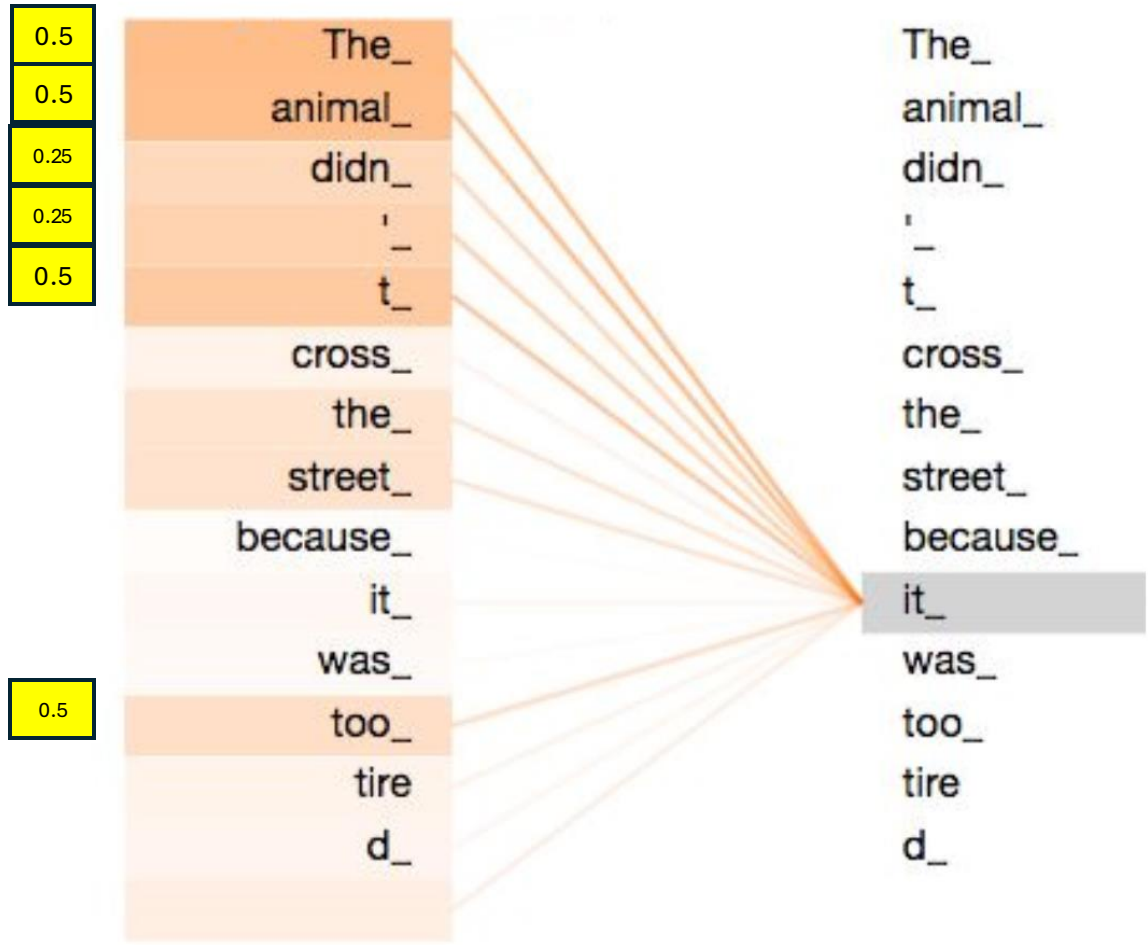


Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Self-Attention: Input's attention on itself

To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...

Which use use to compute the alignment scores $a_{t,i}$

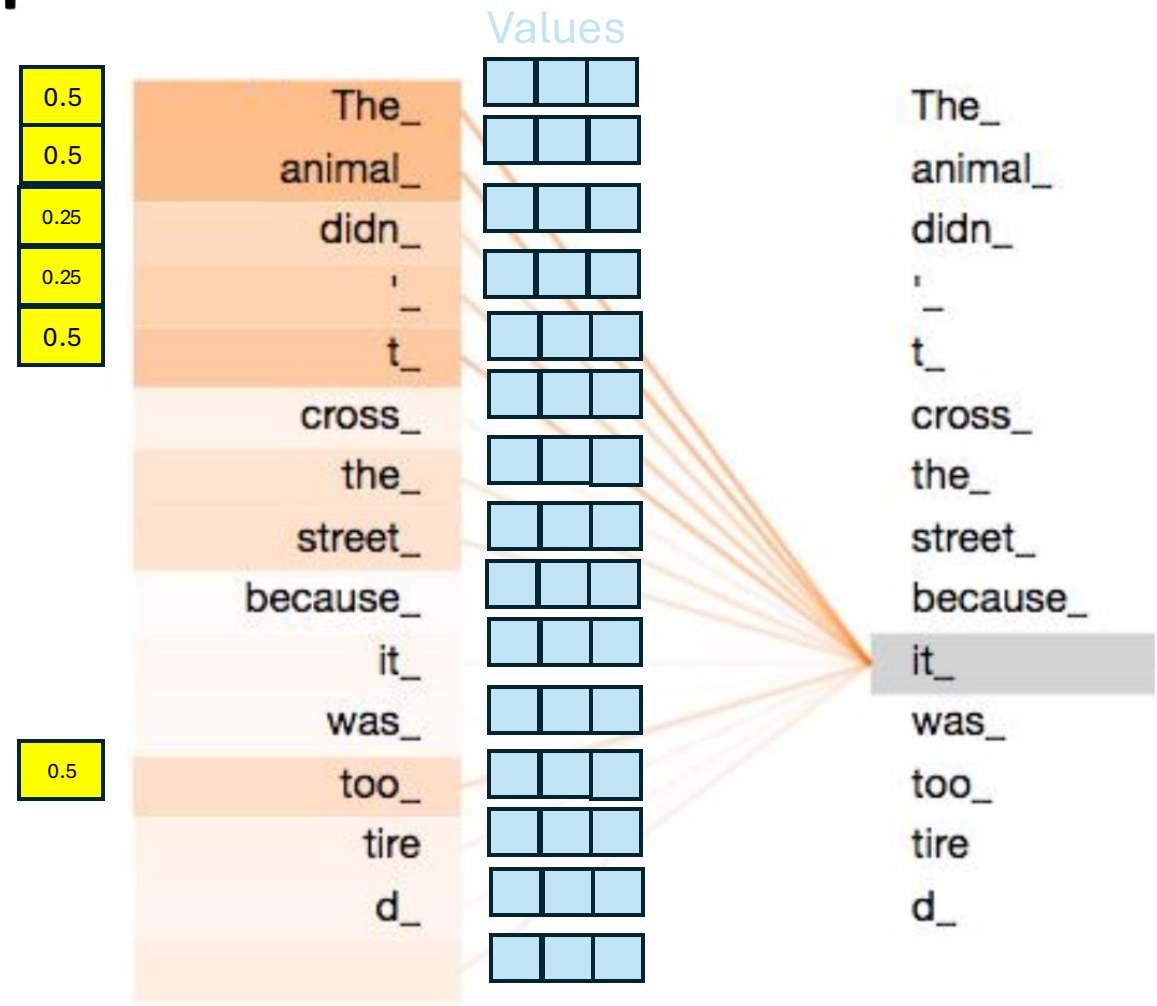


Self-Attention: Input's attention on itself

What do we do next?

To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...

Which use use to compute the alignment scores $a_{t,i}$

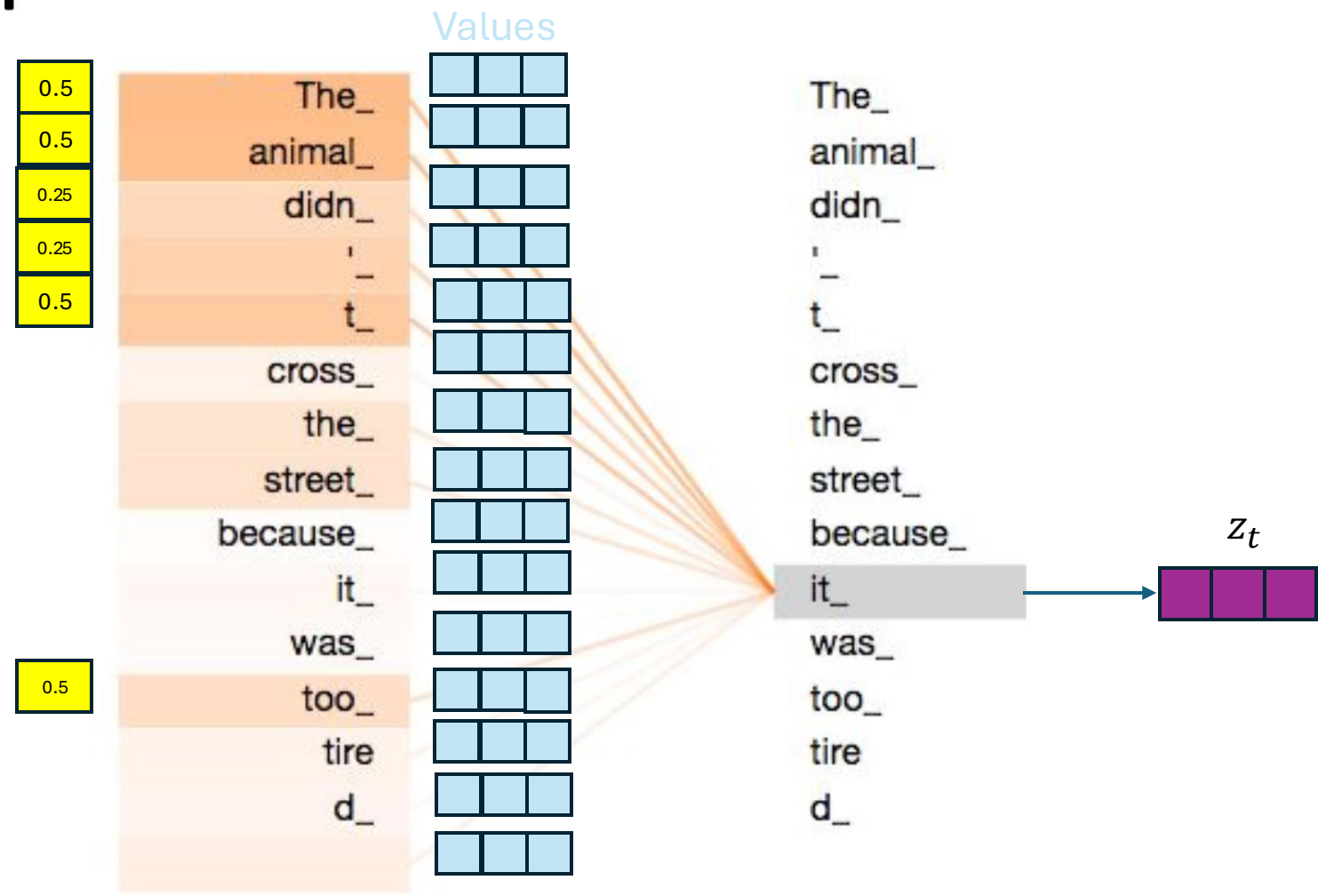


Self-Attention: Input's attention on itself

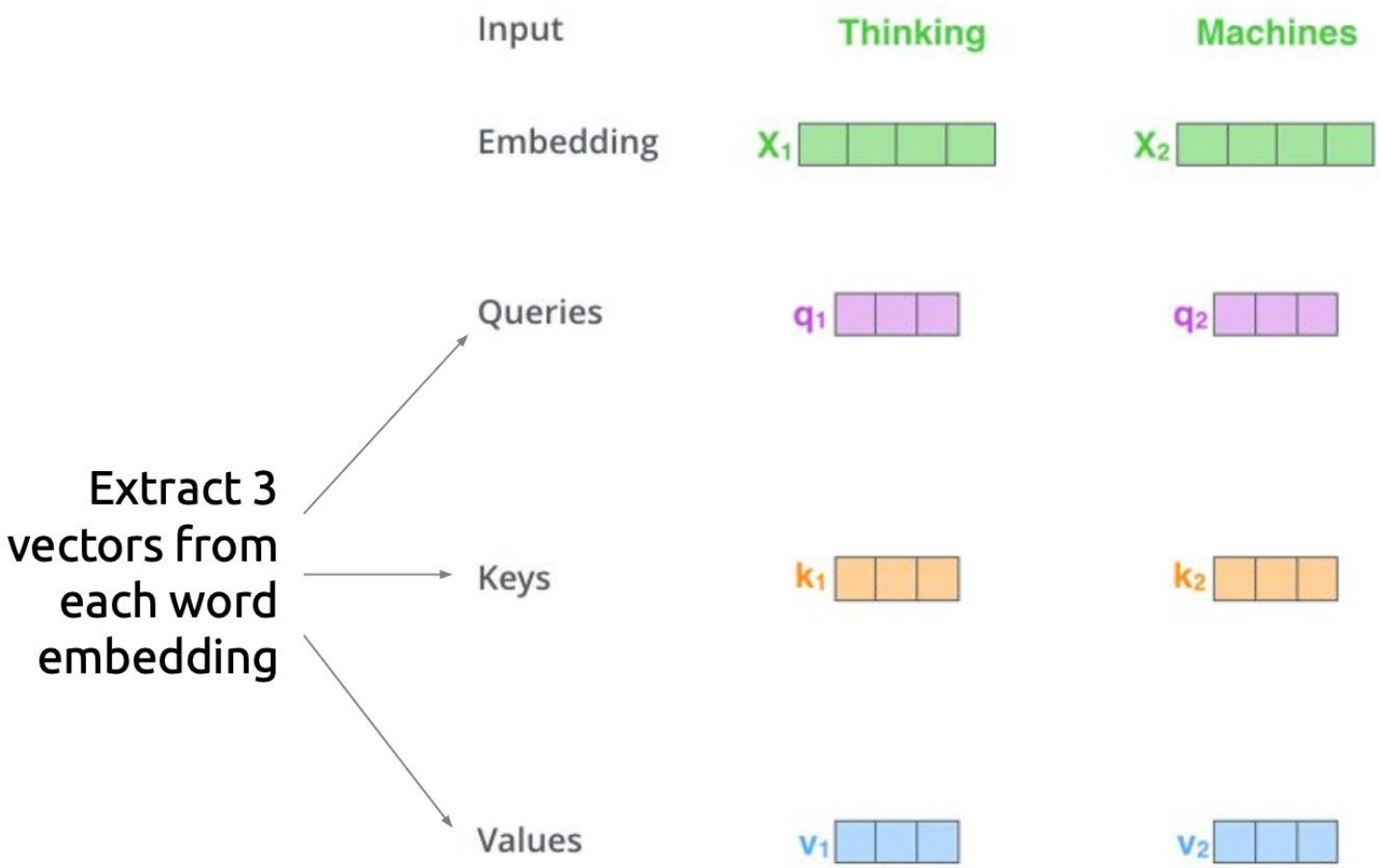
To determine how much attention a word should pay to each other other, we compute a query vector for the word and compare it to a key vector for every other word...

Which use use to compute the alignment scores $a_{t,i}$

To produce the output vector, we sum up the value vectors for each word, weighted by the score we computed in step 1

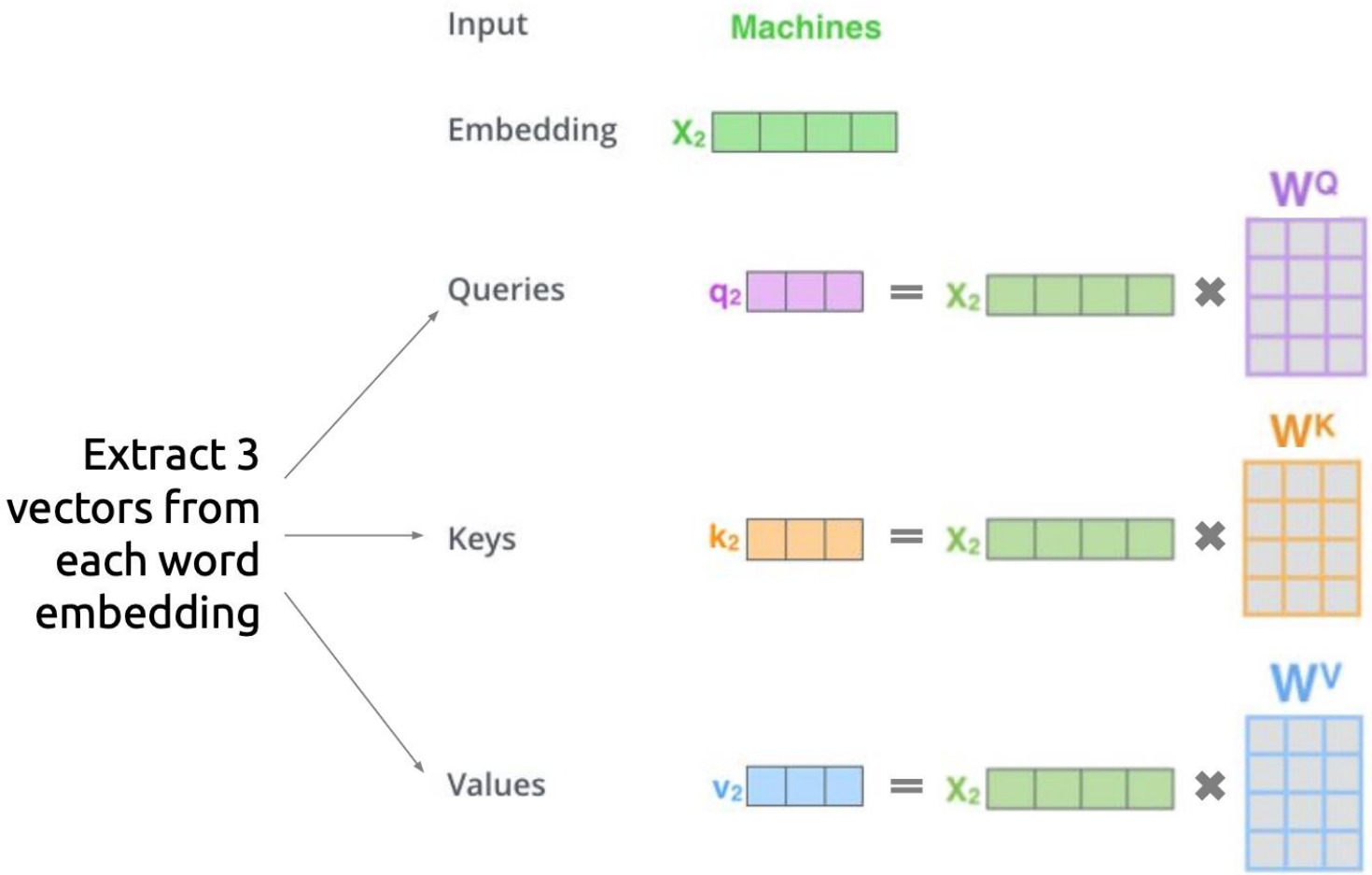


Self-Attention: Details



Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Self-Attention: Details



Each vector is obtained by multiplying the embedding with the respective weight matrix.

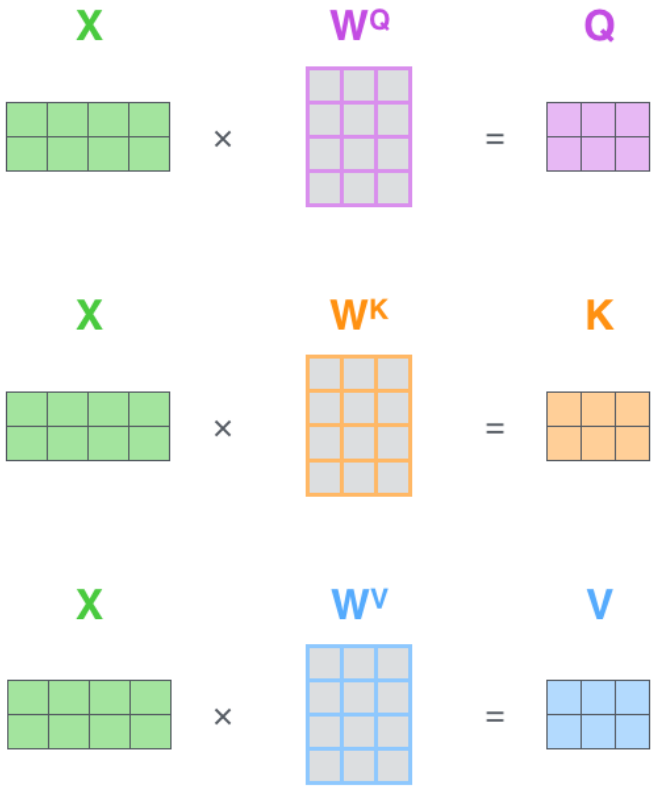
How do we get these weight matrices?

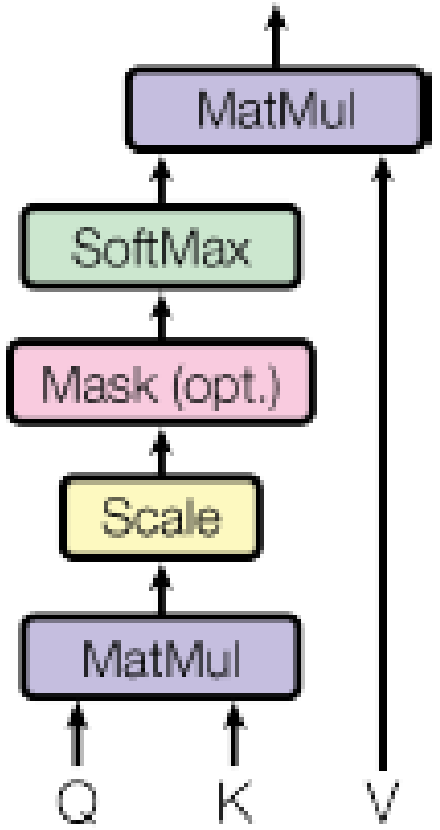
These matrices are the ***trainable parameters*** of the network

Alammar, Jay (2018). The Illustrated Transformer [Blog post]. Retrieved from https://jalammar.github.io/illustrated-transformer/

Scaled Dot Product Attention

Generate Q, K, V, by multiplying word embedding X by weight matrix (i.e., pass through a fully connected layer)





$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK}{\sqrt{d_k}}\right)V$$

Multi-Headed Attention

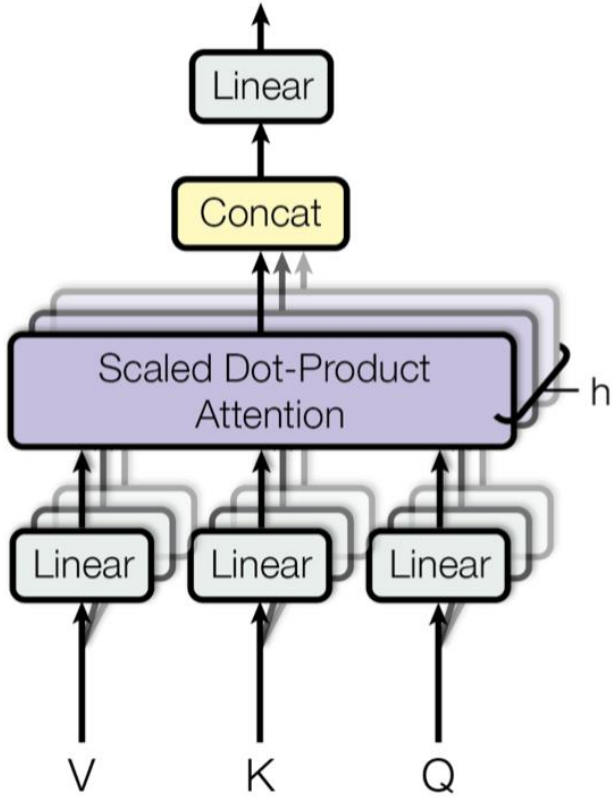
Similar to convolutional layers with multiple filters, we can have "multi-headed attention"

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_\text{h})W^0$$

Where: $head_i = \text{Attention}(\text{QW}_\text{i}^\text{q}, \text{KW}_\text{i}^\text{k}, \text{VW}_\text{i}^\text{v})$

Projected Attention: Project (Q, K, V) with learned parameters W^q, W^k, W^v

Separate learned fully-connected layer for each head *i* and for each of (Q, K, V)



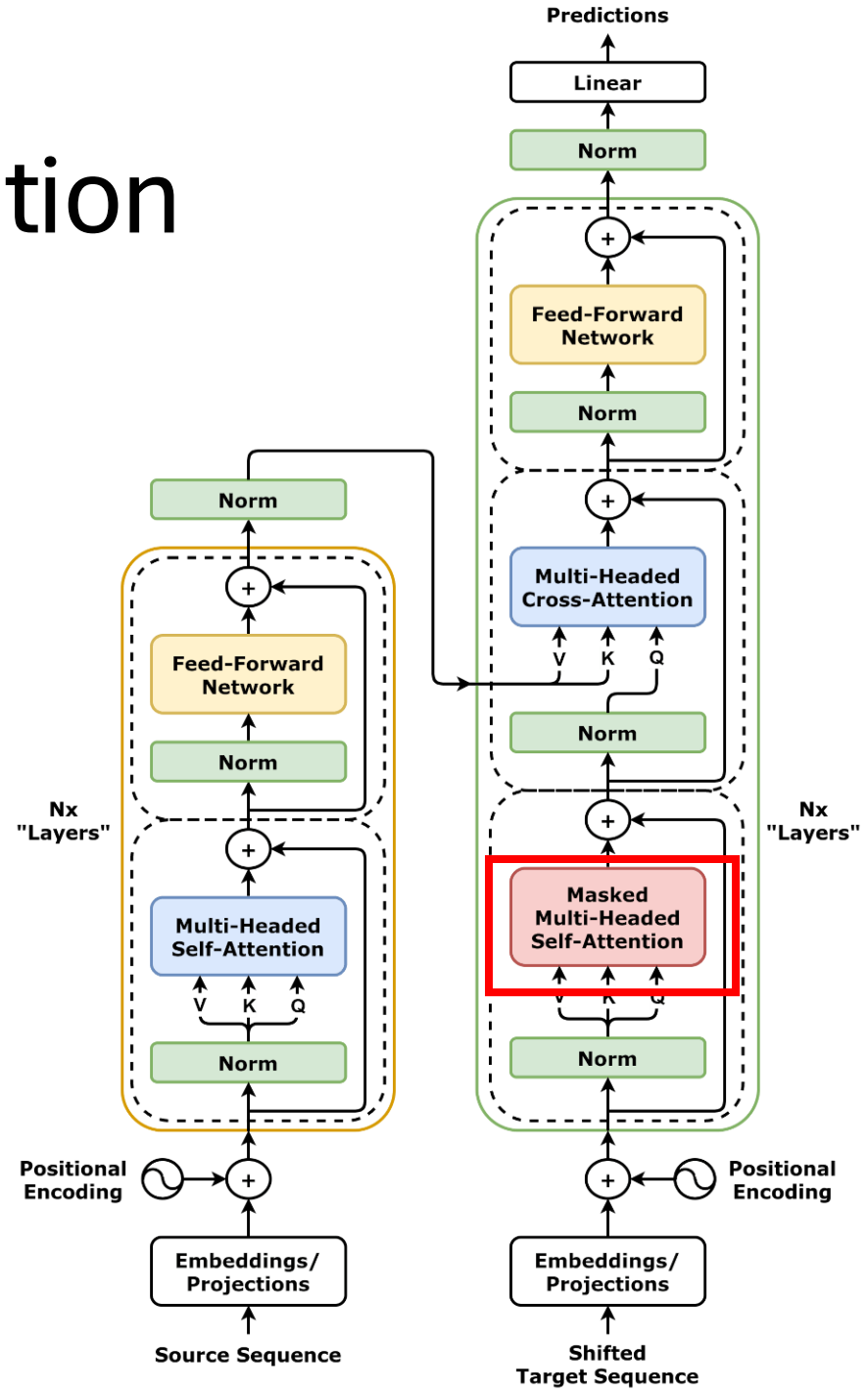
Masked Multi-Headed Attention

In the decoder self attention, there is a “Mask”…

Remember, the target sequence is something we are producing sequentially.

Words early in the output should not be able to attend to words later in the output.

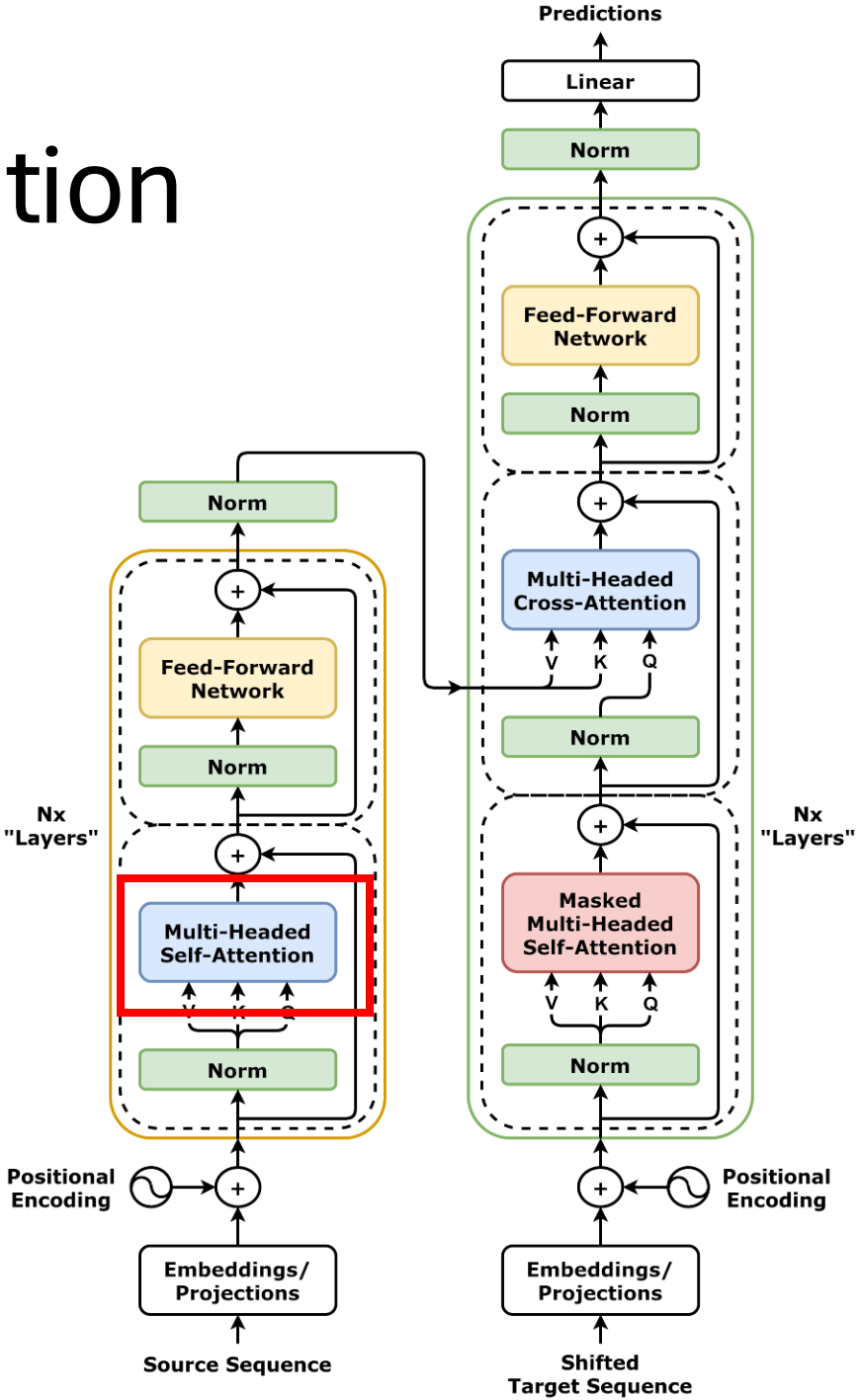
The mask restricts attention only to previous words/targets.



Masked Multi-Headed Attention

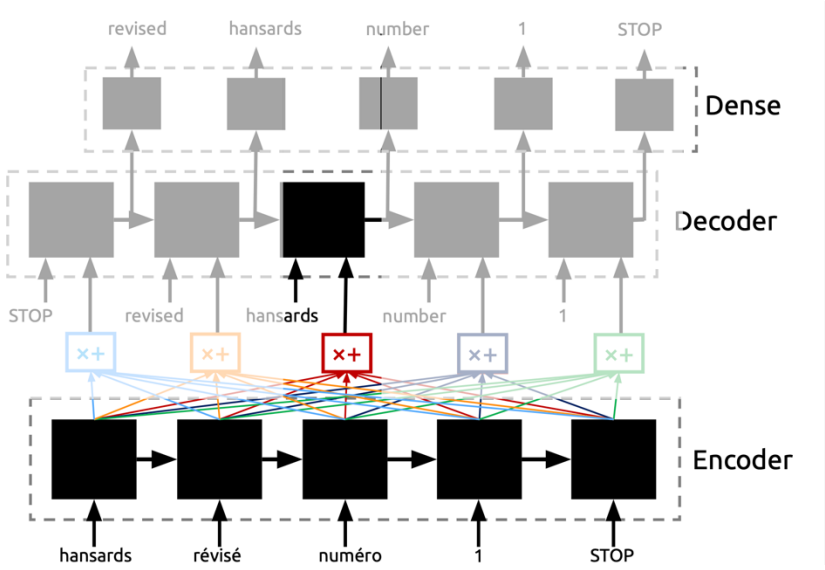
Alternatively, for practical reasons we may feed in "padding" with our inputs (i.e., to run batches we need inputs to be the same size)

We don't want our real words to attend to our padding (empty) words



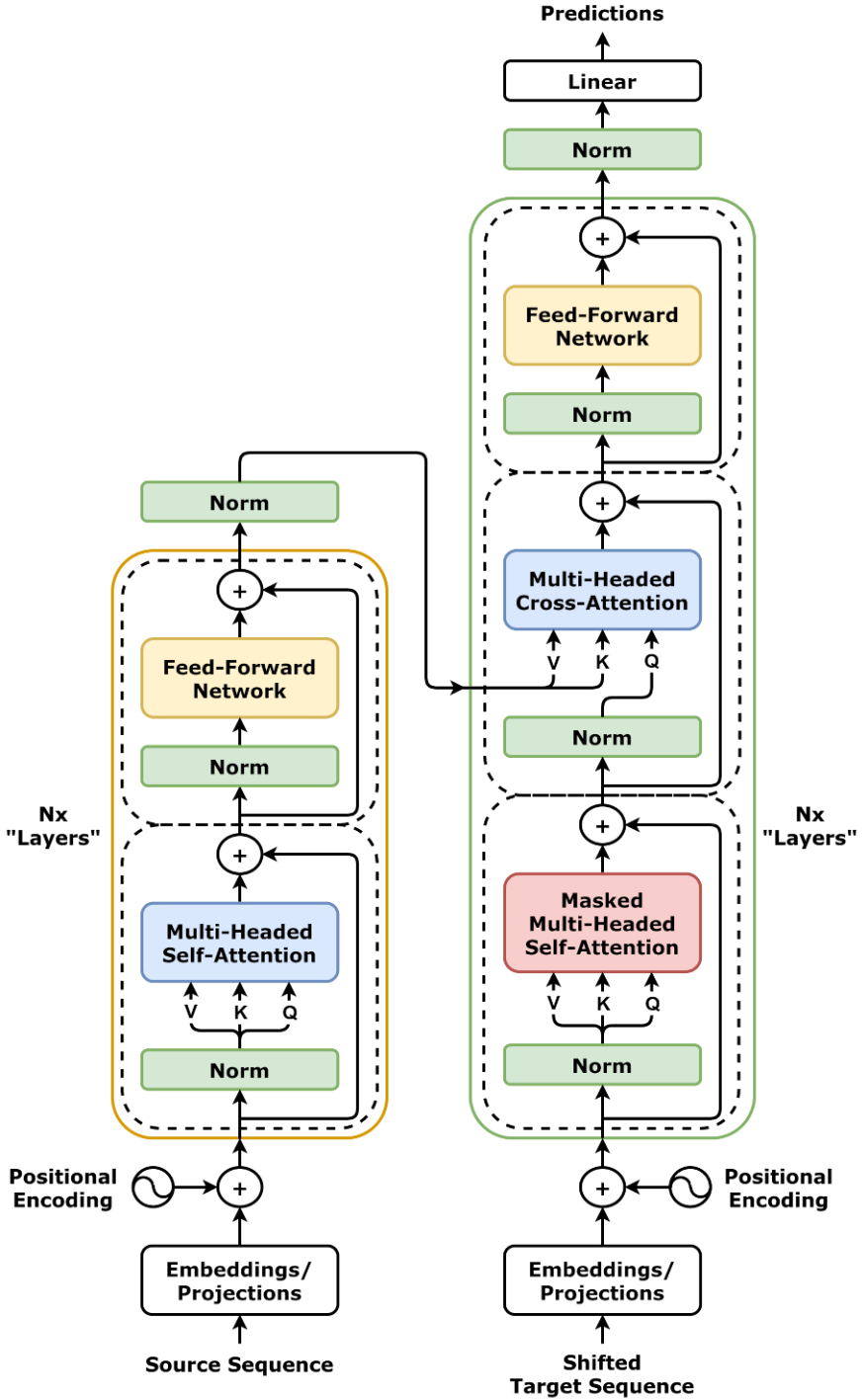
Cross Attention

- Self-Attention is how much each input “attends” to every other input
- Cross-Attention is how much every output “attends” to every input (i.e., our original motivation for attention)



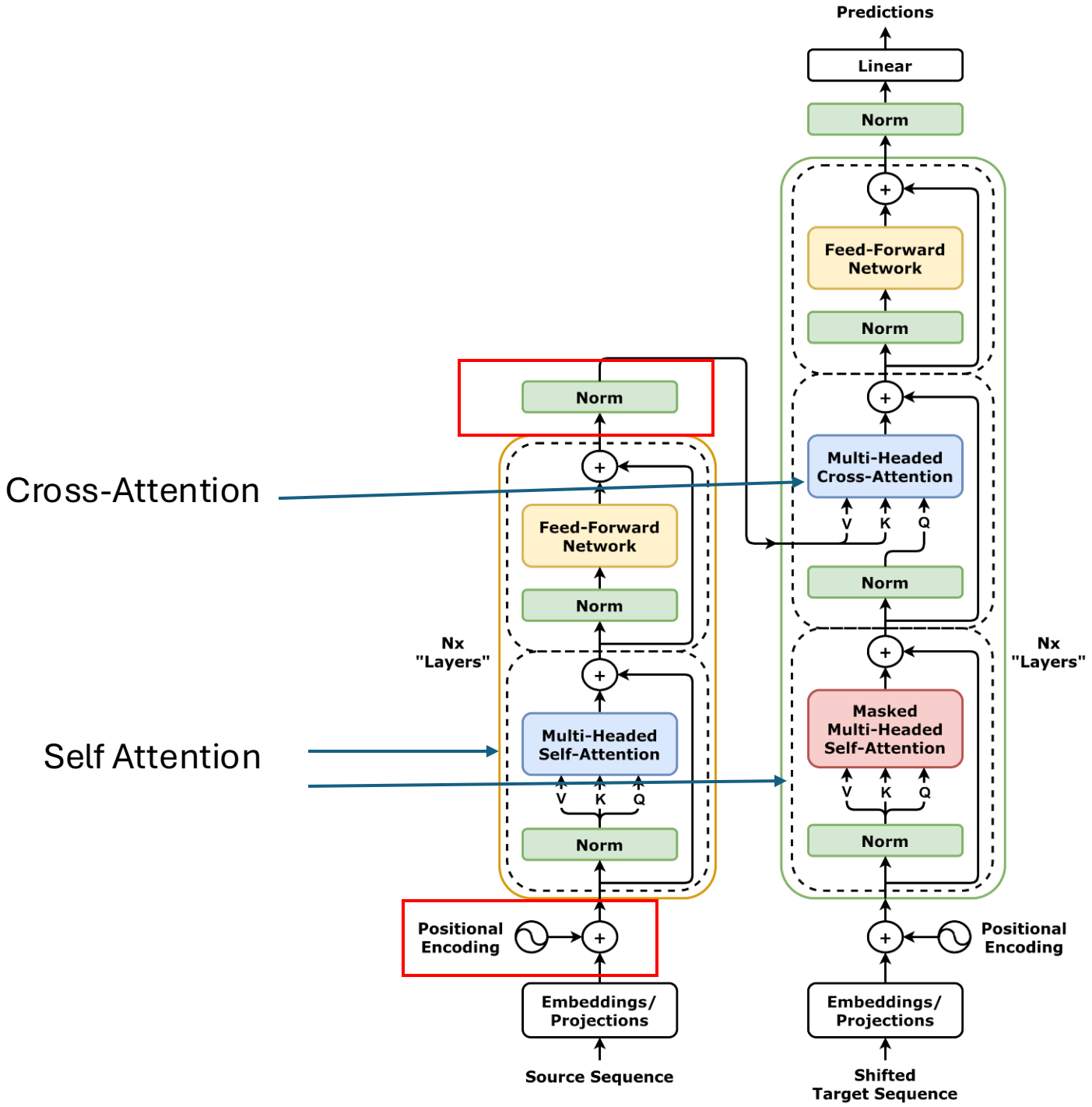
Cross-Attention

Uses V, K from encoder, Q from Decoder



Transformer

What's left?
1. Position Encoding
2. Norm



Position Encoding

- Part of the original motivation behind using RNNs for sequence data was to incorporate the *structure* of the problem (i.e., that order matters in the sequence)
- Attention (so far) does not care about the order that inputs arrive, all the operations are symmetric
- How can we get our networks to realize that there is an ordering to our inputs without using RNNs?

What's the difference between:

1. The cow jumped over the moon
2. Over the jumped cow moon the

Word order matters!

Positional Encoding

Want: a unique encoding (vector) for every value of position

Option 1:

Make this a learnable parameter.

Learn an embedding for every position a word can be in (i.e., 1, 2, 3,… max_length)

Option 2:

Do what “Attention is All you Need” did

$$PE(\text{pos}, 2i) = \sin(\text{pos}/10000^{\frac{2i}{d}})$$

$$PE(\text{pos}, 2i+1) = \cos(\text{pos}/10000^{\frac{2i}{d}})$$

Positional Encoding

Option 1:

Make this a learnable parameter.

Learn an embedding for every position a word can be in (i.e., 1, 2, 3,… max_length)

1. Fix a size for the output of your Position embedding d (has to match size of embeddings/projections)

Option 2:

Do what “Attention is All you Need” did

$$PE(\text{pos}, 2i) = \sin(\text{pos}/10000^{\frac{2i}{d}})$$

$$PE(\text{pos}, 2i+1) = \cos(\text{pos}/10000^{\frac{2i}{d}})$$

Positional Encoding

Option 1:

Make this a learnable parameter.

Learn an embedding for every position a word can be in (i.e., 1, 2, 3,… max_length)

Option 2:

Do what “Attention is All you Need” did

$$PE(\text{pos}, 2i) = \sin(\text{pos}/10000^{\frac{2i}{d}})$$

$$PE(\text{pos}, 2i+1) = \cos(\text{pos}/10000^{\frac{2i}{d}})$$

1. Fix a size for the output of your Position embedding d (has to match size of embeddings/projections)

2. At each index of the encoding, evaluate the proper formula (i.e., even positions use sin, odd positions use cos)

Positional Encoding

"We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE(pos+k) can be represented as a linear function of PE(pos)."

-- Vaswani et al. 2017, Attention is All You Need

$$PE(pos + k) = A \cdot PE(pos)$$

Any Linear function
can be represented as
a matrix multiplication

Positional Encoding

For every pair of adjacent values in the position encoding

$$A \cdot \begin{pmatrix} \sin(c \cdot pos) \\ \cos(c \cdot pos) \end{pmatrix} = \begin{pmatrix} \sin(c \cdot (pos + k)) \\ \cos(c \cdot (pos + k)) \end{pmatrix}$$

$$A = \begin{pmatrix} \cos(\mathrm{c} \cdot k), \ \sin(c \cdot k) \\ -\sin(\mathrm{c} \cdot k), \cos(c \cdot k) \end{pmatrix}$$

Normalization

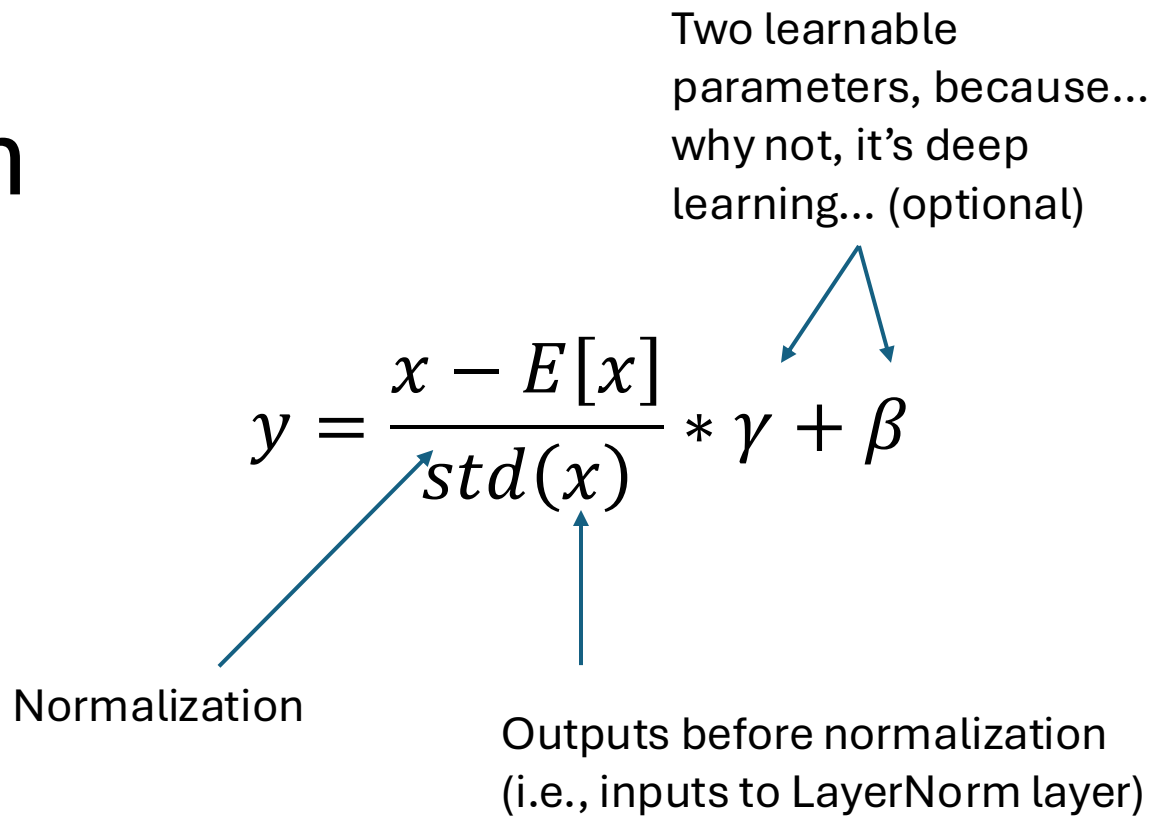
BatchNorm: Normalize outputs of neurons based on mean and standard deviation of the values for a batch of inputs

Issues:

1. RNNs don’t batch well (LayerNorm came before Transformers)
2. When batches are small, mean and standard deviation can vary highly

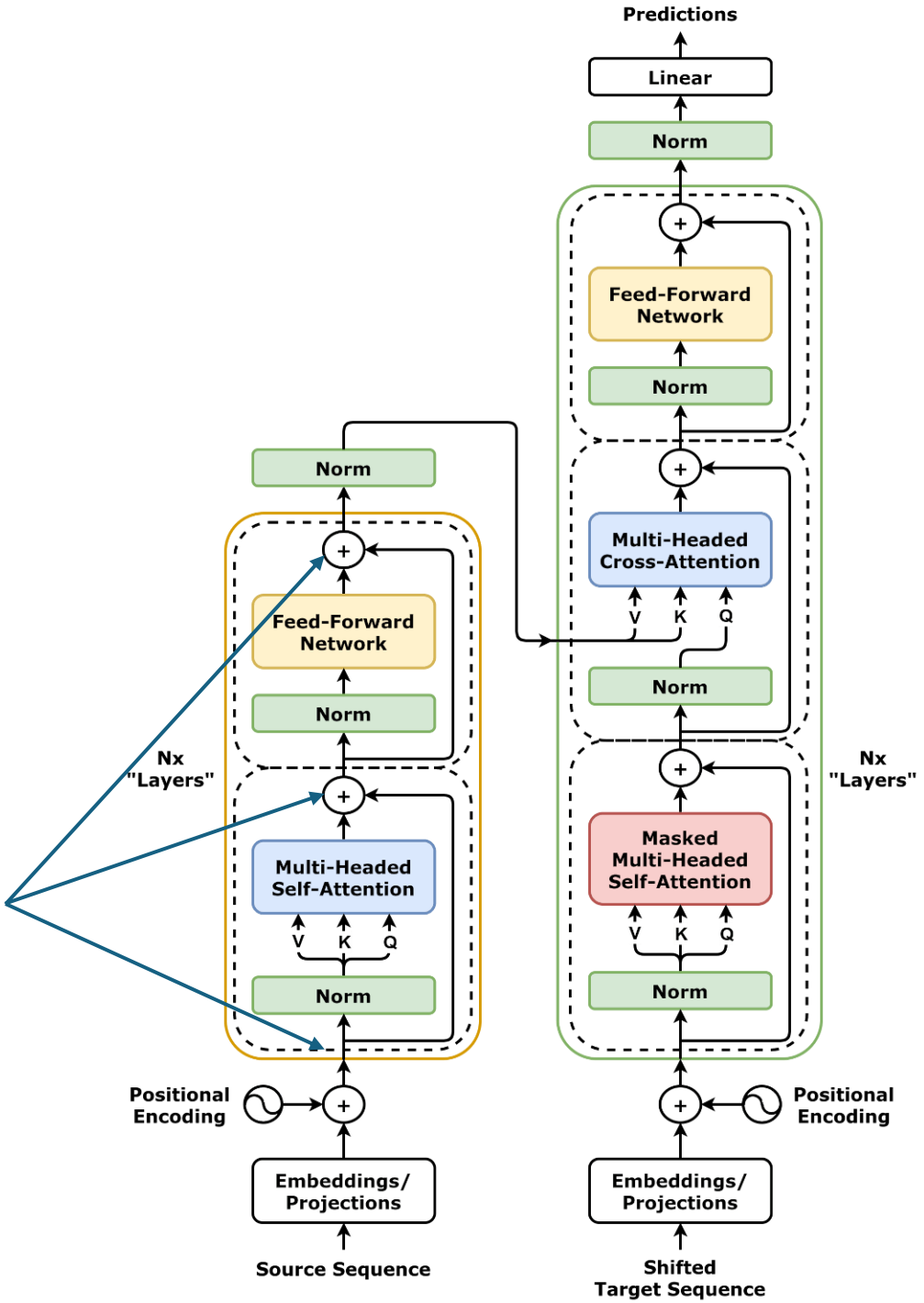
LayerNorm: Instead of normalizing based on the batch dimension, normalize the outputs of a layer based on the mean and standard deviation of the outputs of that layer.

LayerNorm


Two learnable parameters, because… why not, it's deep learning… (optional)
$y = \frac{x - E[x]}{std(x)} * \gamma + \beta$
Normalization
Outputs before normalization (i.e., inputs to LayerNorm layer)

Transformer

All intermediate outputs have same dimension, only one hyperparameter for dimension (many more for number of heads, number of encoder/decoders Nx)



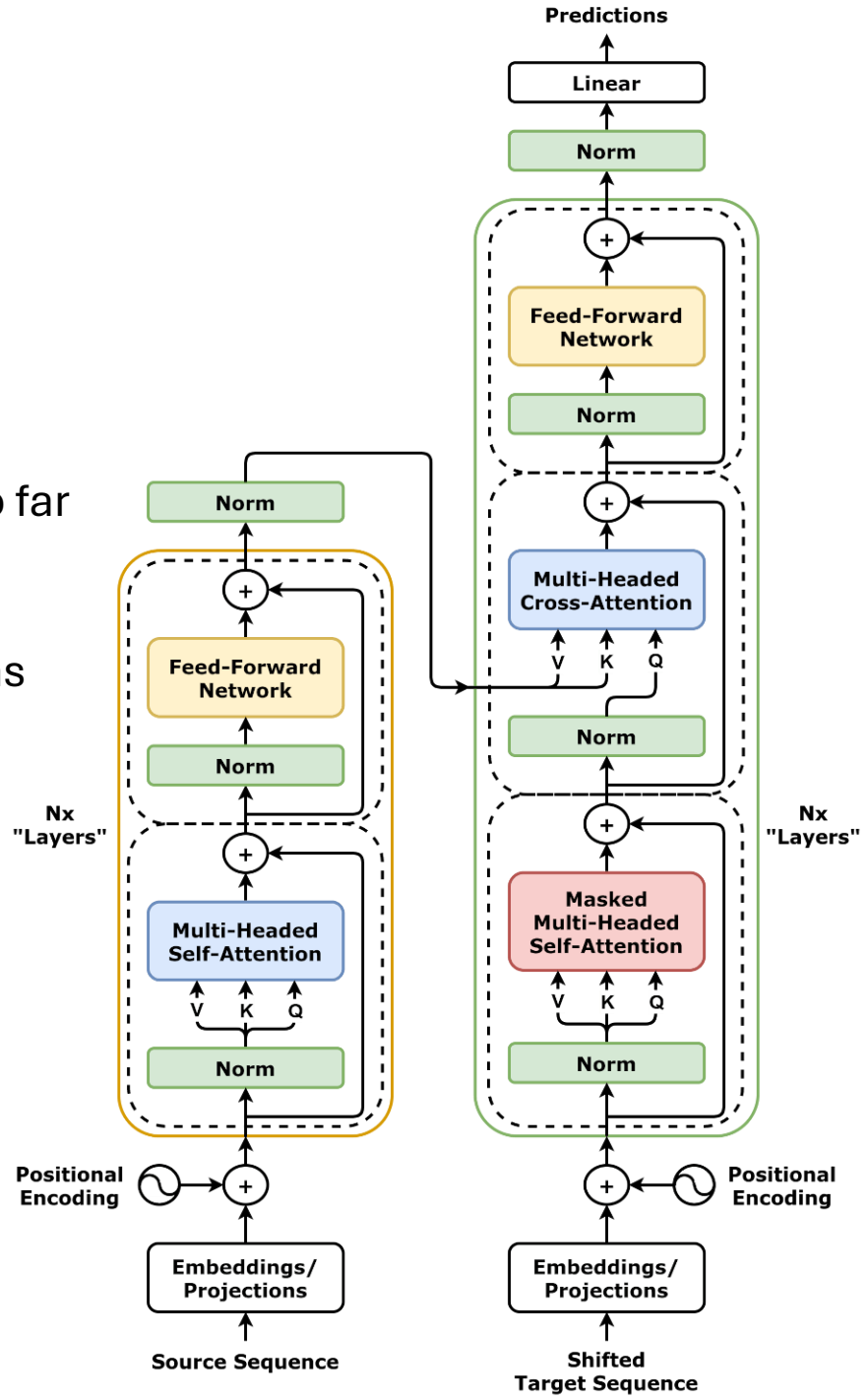
Transformer

Transformers are… complicated

They have many unique components, unlike networks we've covered so far

- CNNs can be large, but they only really have 2 components: Convolutions and linear layers
- The internals of an RNN can be complicated, but it's 3 or 4 operations

Why do they work so well?



Transformer Strengths

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

1. Attention is faster than RNNs ($n \ll d$)
2. Don't require sequential operations, like RNNs
3. Have a lower path length (how many operations does it take for information about words *n* distance apart to spread to each other)

Transformer Strengths

params	dimension	n heads	n layers	learning rate	batch size	n tokens
6.7B	4096	32	32	$3.0e^{-4}$	4M	1.0T
13.0B	5120	40	40	$3.0e^{-4}$	4M	1.0T
32.5B	6656	52	60	$1.5e^{-4}$	4M	1.4T
65.2B	8192	64	80	$1.5e^{-4}$	4M	1.4T

Table 2: **Model sizes, architectures, and optimization hyper-parameters.**

Deep Networks do better. More parameters are better. Transformers have many learnable parameters.

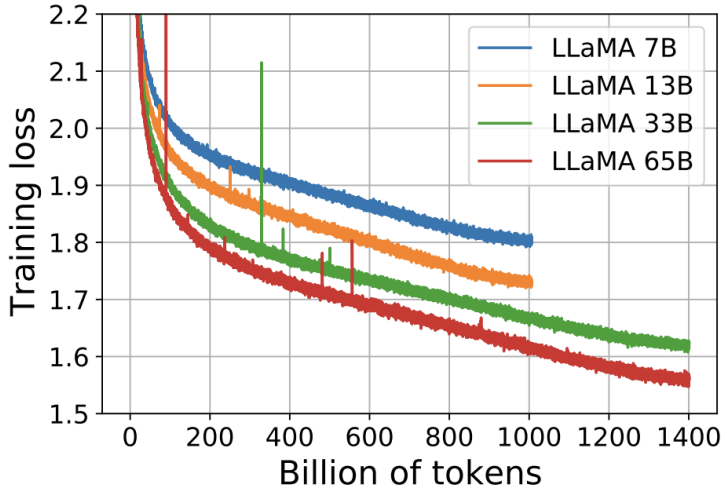


Figure 1: **Training loss over train tokens for the 7B, 13B, 33B, and 65 models.** LLaMA-33B and LLaMA-65B were trained on 1.4T tokens. The smaller models were trained on 1.0T tokens. All models are trained with a batch size of 4M tokens.

LLaMA: Open and Efficient Foundation Language Models

Transformer Weaknesses

- Transformers are not good at small scale tasks, they have many parameters and tend to overfit easily.
- There are really not that many hyperparameters in transformers, just the number of attention heads, number of layers, and embedding size.
  - Hard to get them to not overfit

Why is this weakness not actually a problem?

Large Language Models: Friday

The bigger the better

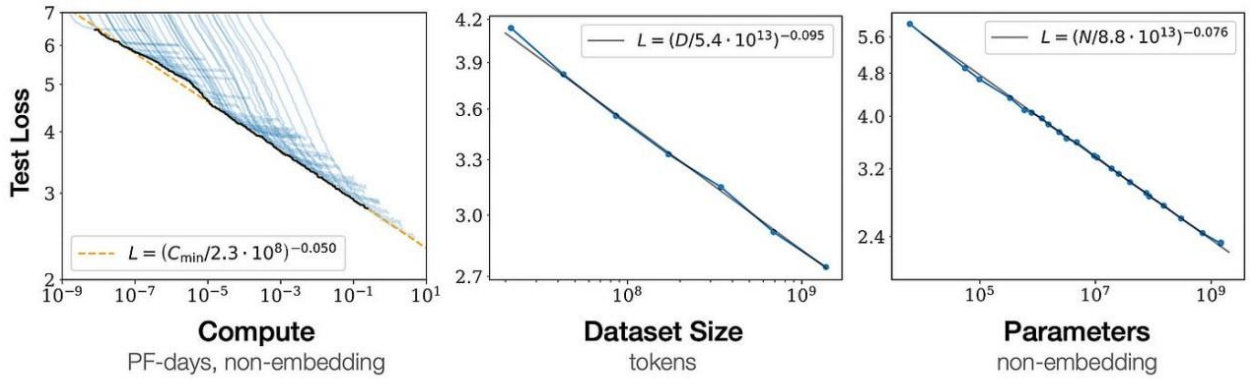
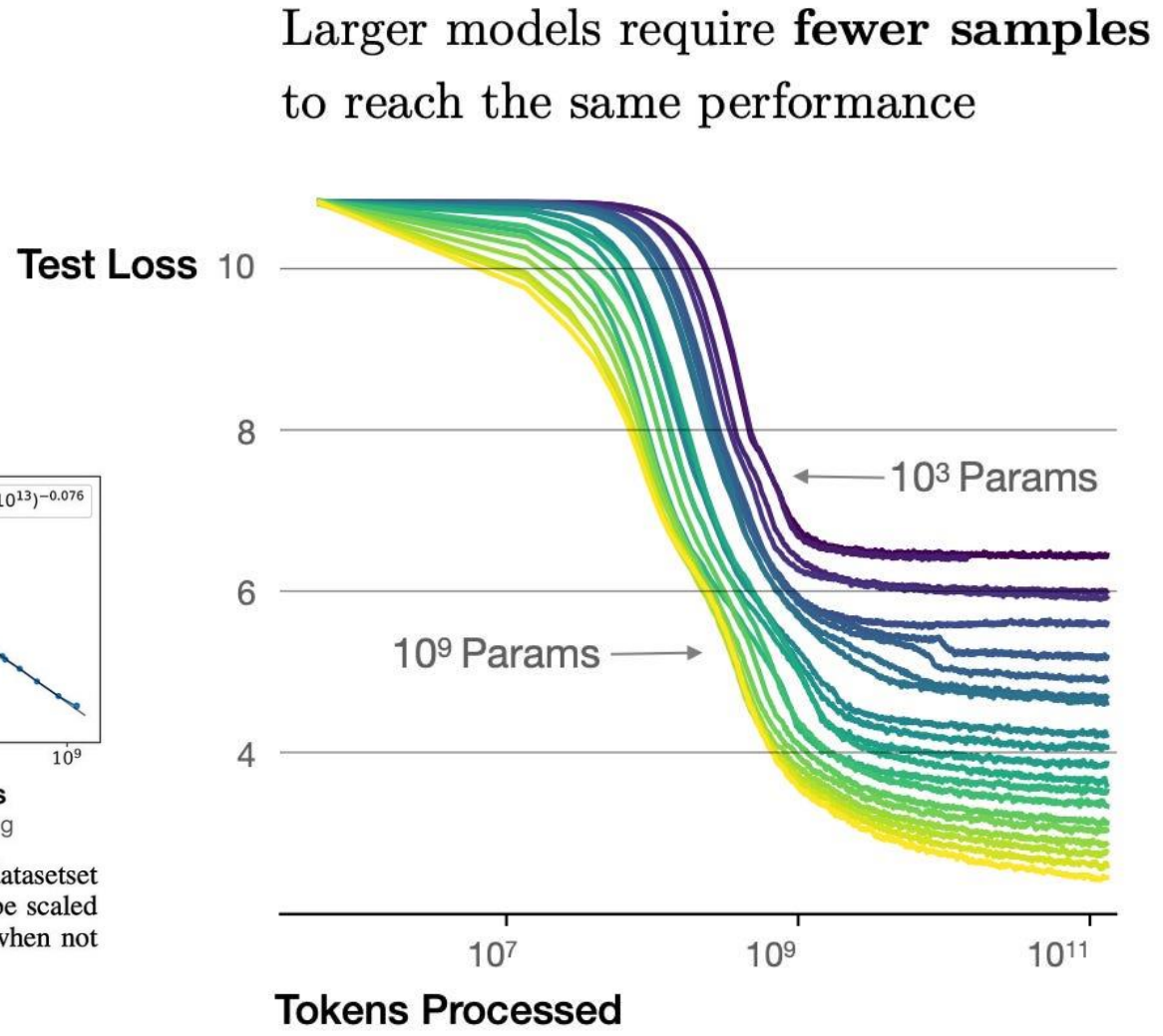


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.



Out-of-scope use cases

Because large-scale language models like GPT-2 do not distinguish fact from fiction, we don't support use-cases that require the generated text to be true.

Additionally, language models like GPT-2 reflect the biases inherent to the systems they were trained on, so we do not recommend that they be deployed into systems that interact with humans unless the deployers first carry out a study of biases relevant to the intended use-case. We found no statistically significant difference in gender, race, and religious bias probes between 774M and 1.5B, implying all versions of GPT-2 should be approached with similar levels of caution around use cases that are sensitive to biases around human attributes.

GPT-2 Official Open Source Repository:
https://github.com/openai/gpt-2/blob/master/model_card.md

Recap

Transformers consist of a set of encoders and decoders that use attention to make predictions for sequence tasks.

They have *a lot* of learned parameters