Attention & Transformers

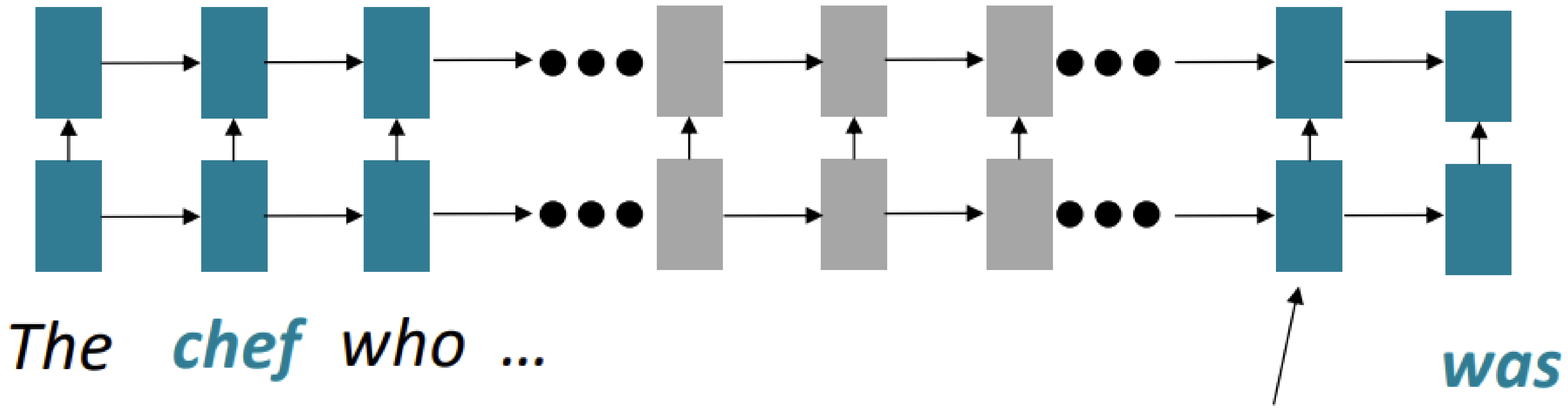
Mausam
IIT Delhi

(some figures taken from Jay Alammar's blog,
some slides taken from Tatsunori Hashimoti)

Issues with RNNs – Linear Interaction Distance

O(sequence length) steps for distant word pairs to interact means:

- Hard to learn long-distance dependencies (because gradient problems!)
- Linear order of words is "baked in"; we already know linear order isn't the right way to think about sentences…

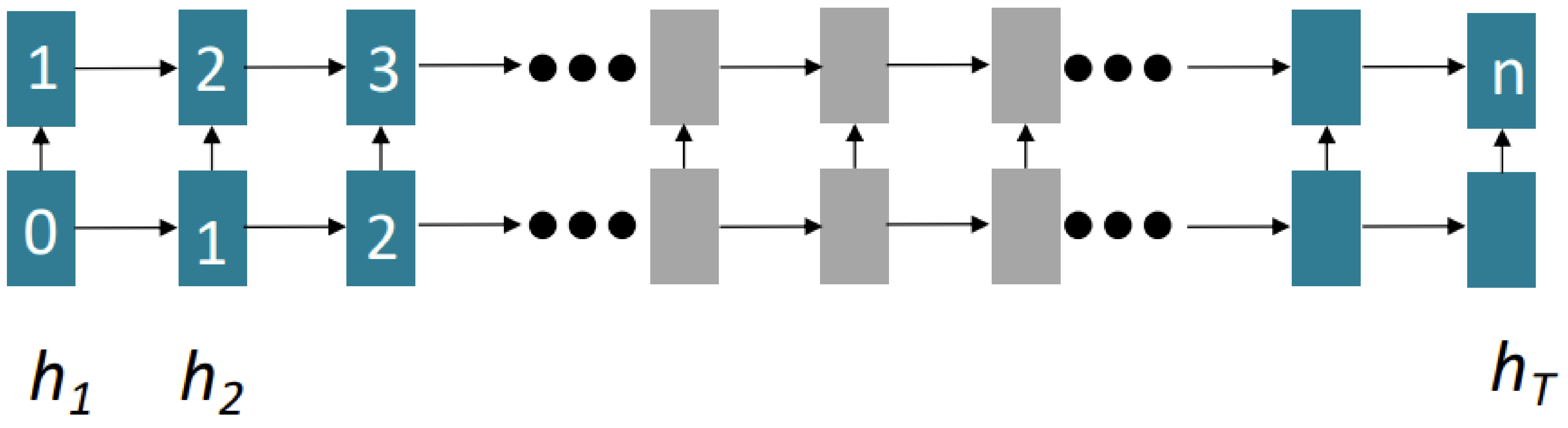


Info of ***chef*** has gone through
O(sequence length) many layers!

Issues with RNNs – Lack of Parallelizability

Forward and backward passes have **O(sequence length)** unparallelizable operations

- GPUs can perform a bunch of independent computations at once!
- But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
- Inhibits training on very large datasets!



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

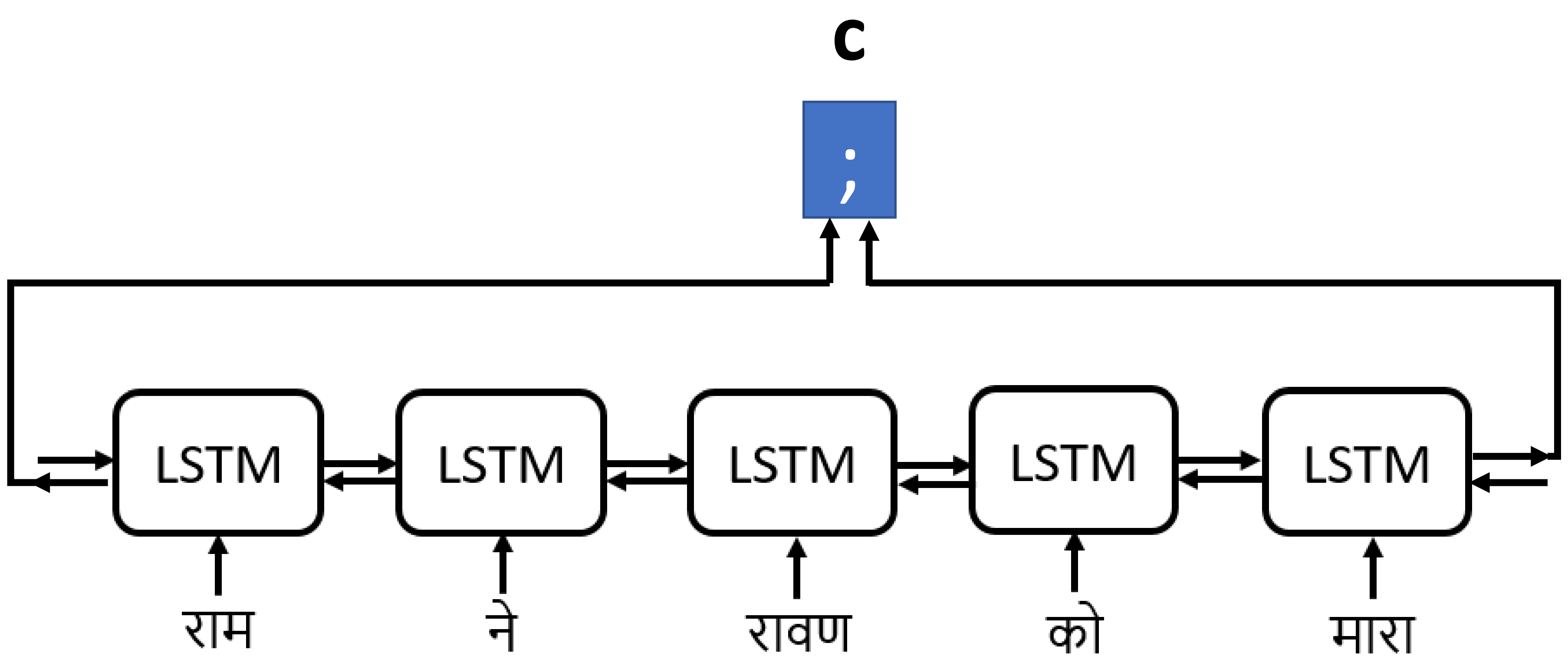
Attention

Sentence Representation

*You can't cram the meaning of the whole *%#@ing sentence in a single *%#@ing vector.*

- Encoding a single vector is too restrictive.
  Instead of producing a single vector for the sentence, produce one vector for each word.
- But, eventually need 1 vector.
  Multiple vectors → Single vector
  Sum/Avg operators give equal importance to each input
- We dynamically decide which input is more/less important for a task.
- Create a weighted sum to reflect this variation: Attention
- query (q): decides importance of each input
  attention weights (α_i): normalized importance of input
  unnormalized attention weights ($\bar{\alpha}_i$): intermediate step to compute α_i
  attended summary: weighted avg of input with α weights

LSTM Encoder



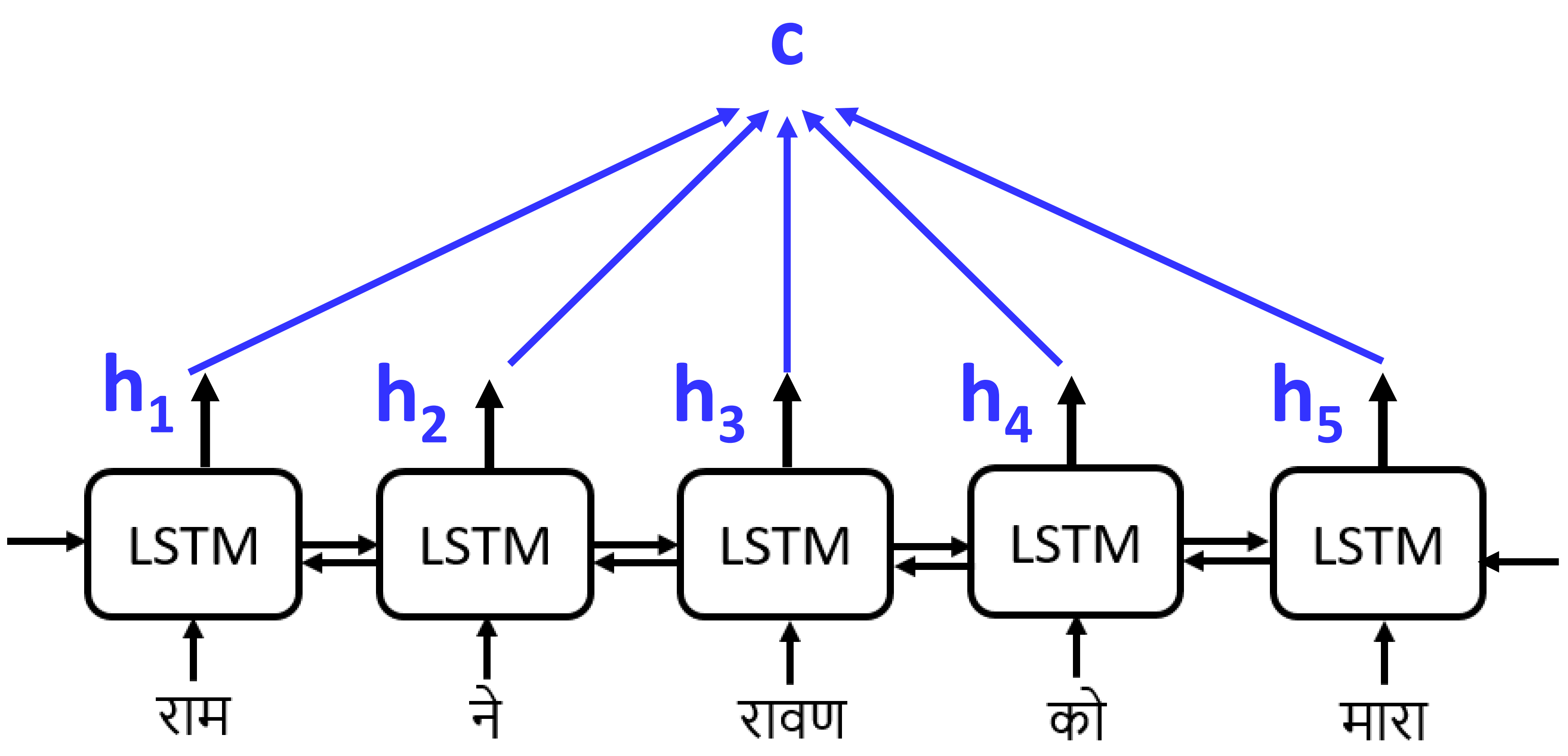
Encoder
(Bi-LSTM)

Multiple Encoded Vectors → Single Summary

$$h_{1:T} = \text{biLSTM}(x_{1:T})$$

$$c = \sum_{i=1}^{T} \alpha_i . h_i$$

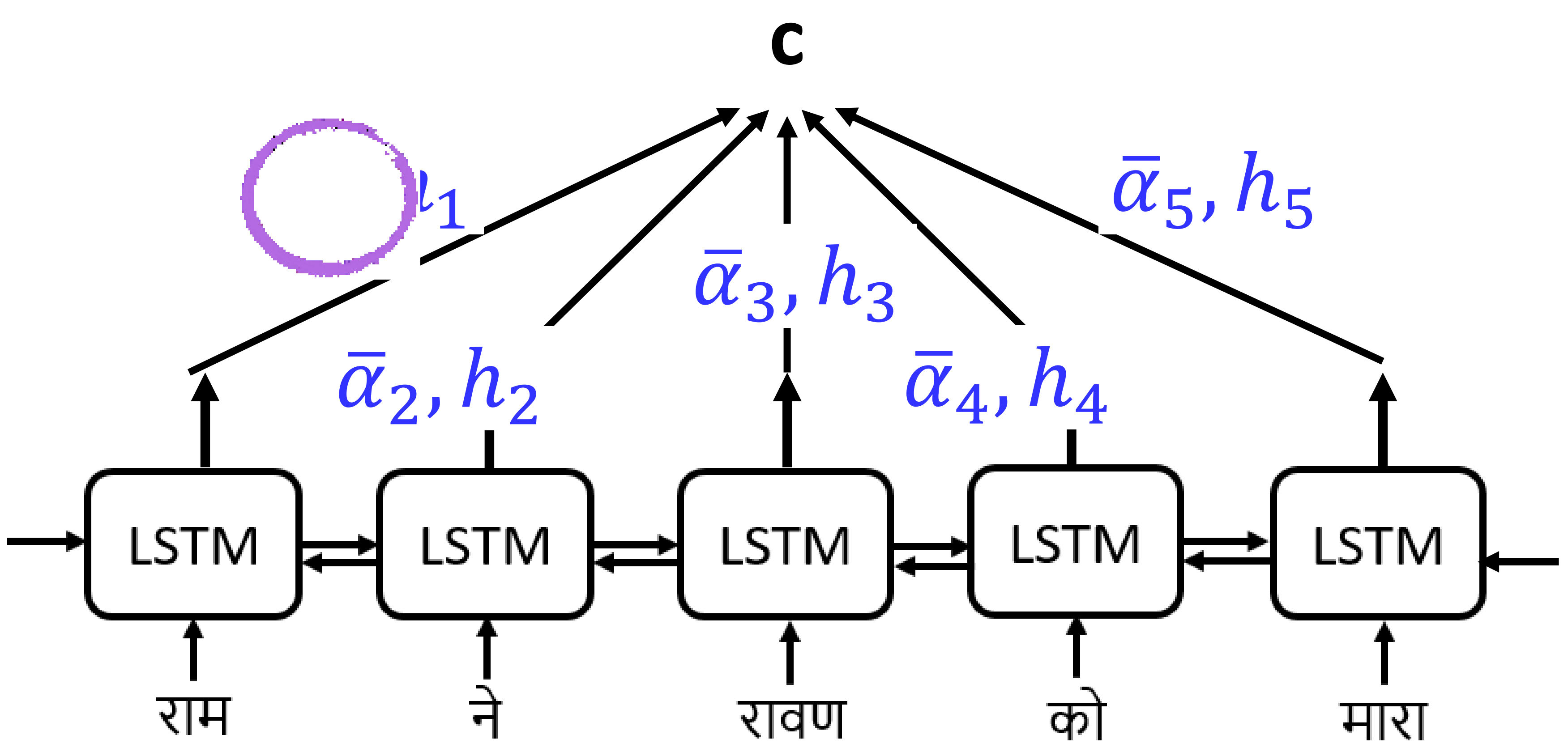
Need to convert h_is to c



Multiple Encoded Vectors → Single Summary

$$c = \sum_{i=1}^{T} \alpha_i h_i$$

$$\alpha_{1:T} = \text{softmax}(\bar{\alpha}_1, \bar{\alpha}_2, \ldots, \bar{\alpha}_T)$$

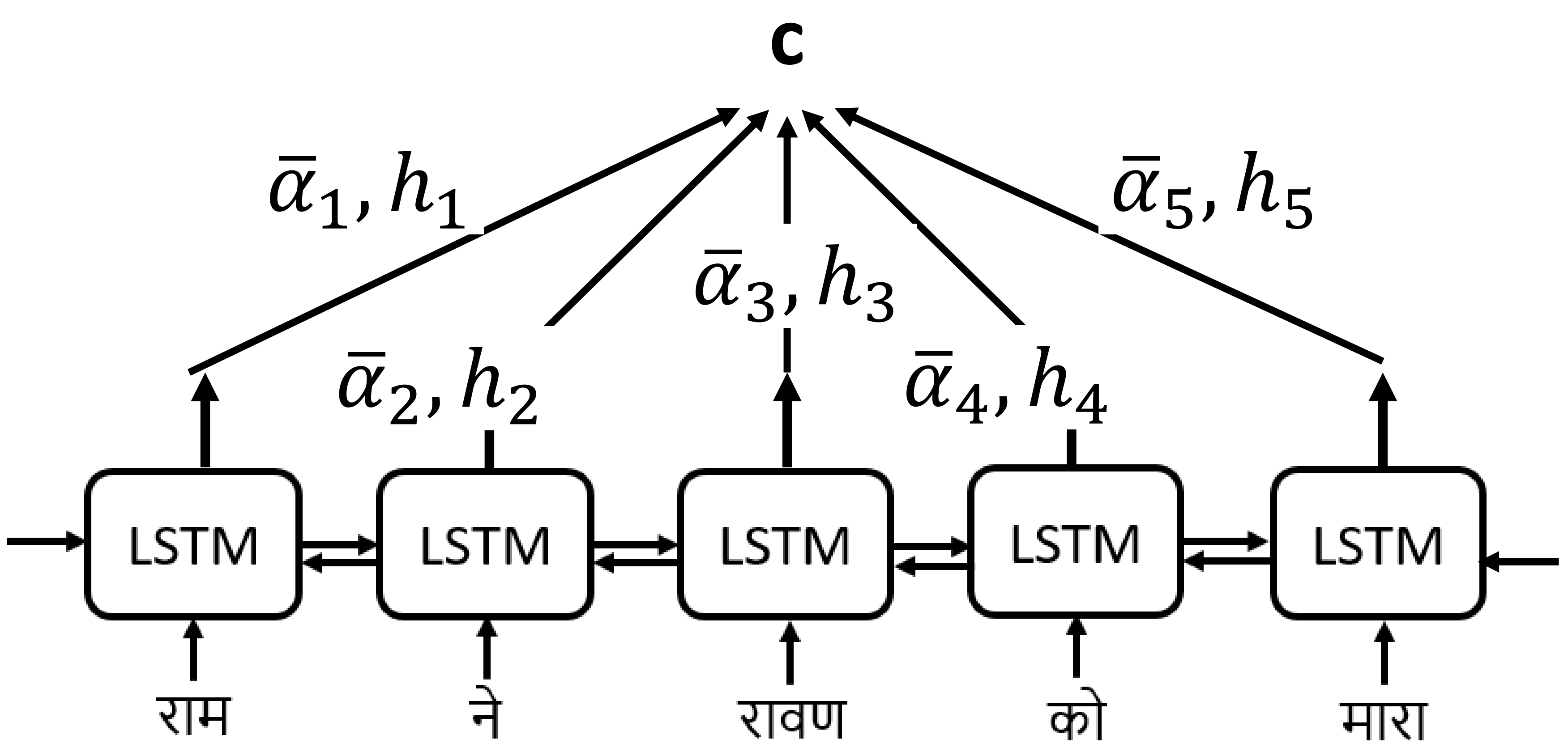


Multiple Encoded Vectors → Single Summary

$$c = \sum_{i=1}^{T} \alpha_i . h_i$$

$$\alpha_{1:T} = \text{softmax}(\bar{\alpha}_1, \bar{\alpha}_2, \ldots, \bar{\alpha}_T)$$

$$\bar{\alpha}_i = \phi^{\text{att}}(q, h_i)$$



Attention: Encoding

$$c = \sum_{i=1}^{T} \alpha_i . h_i$$

$$h_{1:T} = \text{biLSTM}_{enc}(x_{1:T})$$

$$\alpha = \text{softmax}(\bar{\alpha}_1, \dots, \bar{\alpha}_T)$$

$$\bar{\alpha}_i = \phi^{\text{att}}(q, h_i)$$

what is ϕ^{att}?

what is q?

Attention Functions ϕ^{att}

- Bahadanau Attention: $\phi^{\mathrm{att}}(q, h) = \mathrm{u.\,g(Wq + W'h + b)}$
- Luong Attention: $\phi^{\mathrm{att}}(q, h) = \mathrm{q.\,h}$
- Scaled Dot Product Attention: $\phi^{\mathrm{att}}(q, h) = \frac{\mathrm{q.h}}{\sqrt{d}}$
- Bilinear Attention: $\phi^{\mathrm{att}}(q, h) = \mathrm{h}\mathbf{W}\mathrm{q}$
- Reduced Rank Multiplicative Attention (h**U**)(**V**q)
  - **U** has dim $d_1$xk, **V** has dim kd_2, k << d_1, d_2

Additive vs Multiplicative

While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients [4]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

d is the dimensionality of q and h $\frac{q.h}{\sqrt{d}}$

Scaled dot product attention

Paper's Justification:

To illustrate why the dot products get large, assume that the components of q and h are independent random variables with mean 0 and variance 1 → Then their dot product, q · h has mean 0 and variance d

Attention: Encoding

$$c = \sum_{i=1}^{T} \alpha_i . h_i$$

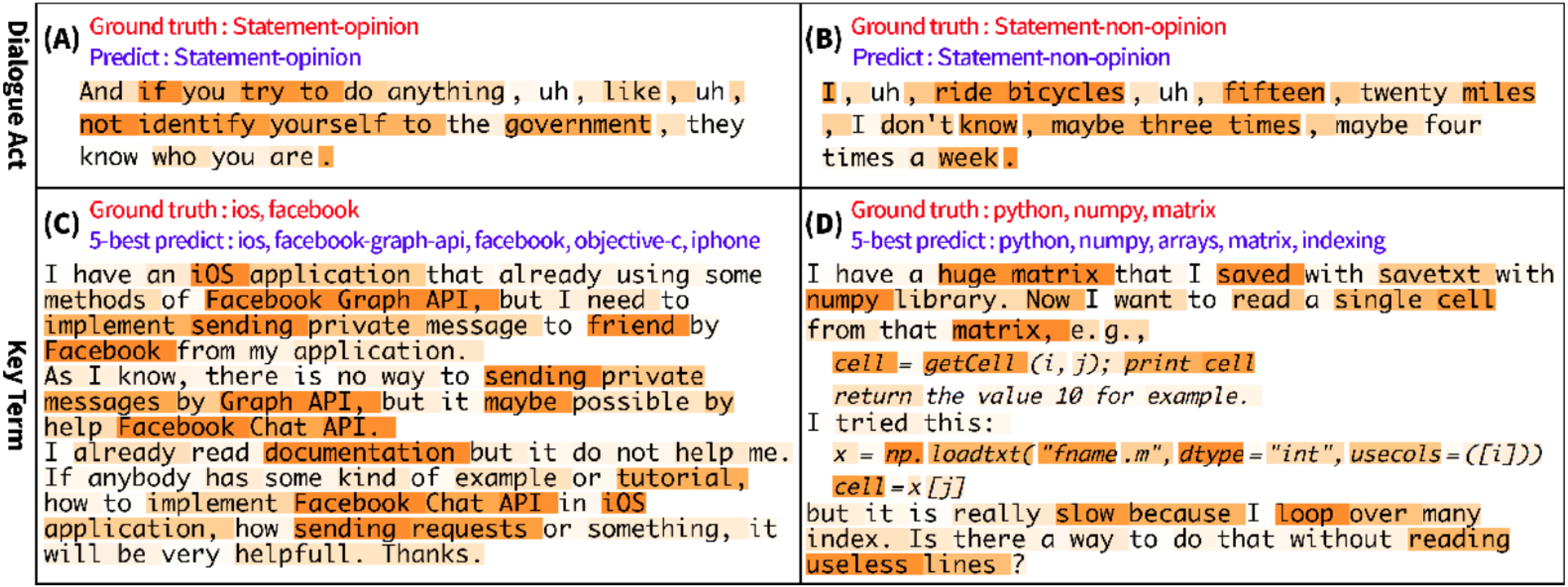
$$h_{1:T} = \text{biLSTM}_{enc}(x_{1:T})$$

$$\alpha = \text{softmax}(\bar{\alpha}_1, \dots, \bar{\alpha}_T)$$

$$\bar{\alpha}_i = \phi^{\text{att}}(q, h_i)$$

what is q?

Attention and/vs Interpretation



Published in INTERSPEECH 2016

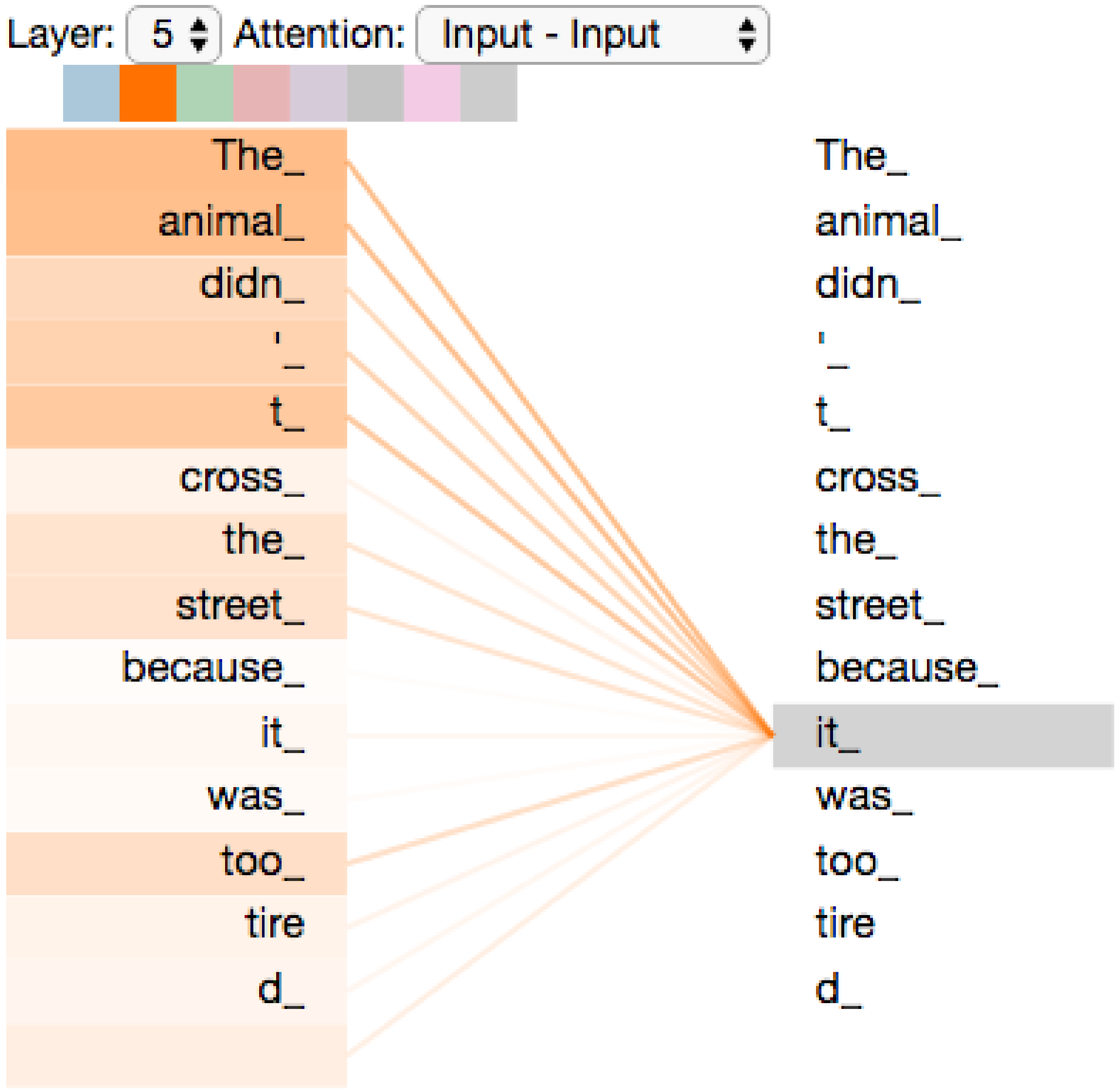
Neural Attention Models for Sequence Classification: Analysis and Application to Key Term Extraction and Dialogue Act Detection

S. Shen, Hung-yi Lee

Multi-head Key-Value Self Attention

Self-attention (single-head, high-level)

"`The animal didn't cross the street because it was too tired`"



There is no external query q.
The input is also the query.
Many approaches:
https://ruder.io/deep-learning-nlp-best-practices/

Transformers: query q is another $x_{j:}$ $\phi^{att}(x_j,x_i)$

Attention: Encoding (h → x)

$$c = \sum_{i=1}^{T} \alpha_i . x_i$$

$$\alpha = \text{softmax}(\bar{\alpha}_1, \dots, \bar{\alpha}_T)$$

$$\bar{\alpha}_i = \phi^{\text{att}}(q, x_i)$$

Attention: Encoding

$$c = \sum_{i=1}^{T} \alpha_i . x_i$$

$$\alpha = \text{softmax}(\bar{\alpha}_1, \ldots, \bar{\alpha}_T)$$

$$\bar{\alpha}_i = \phi^{\text{att}}(q, x_i)$$

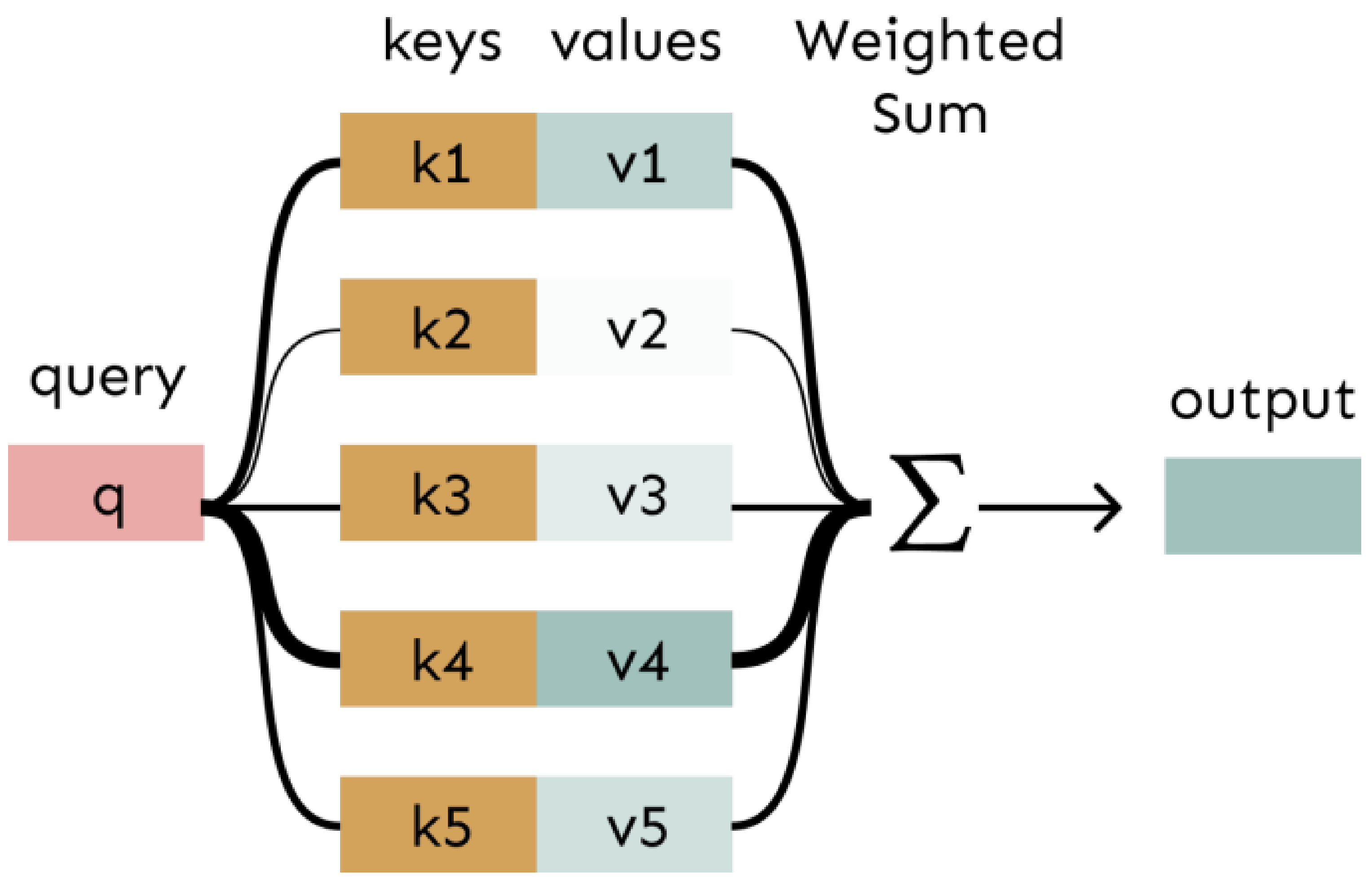
Each vector (x) playing two roles

(1) computing importance
(2) weighted sum

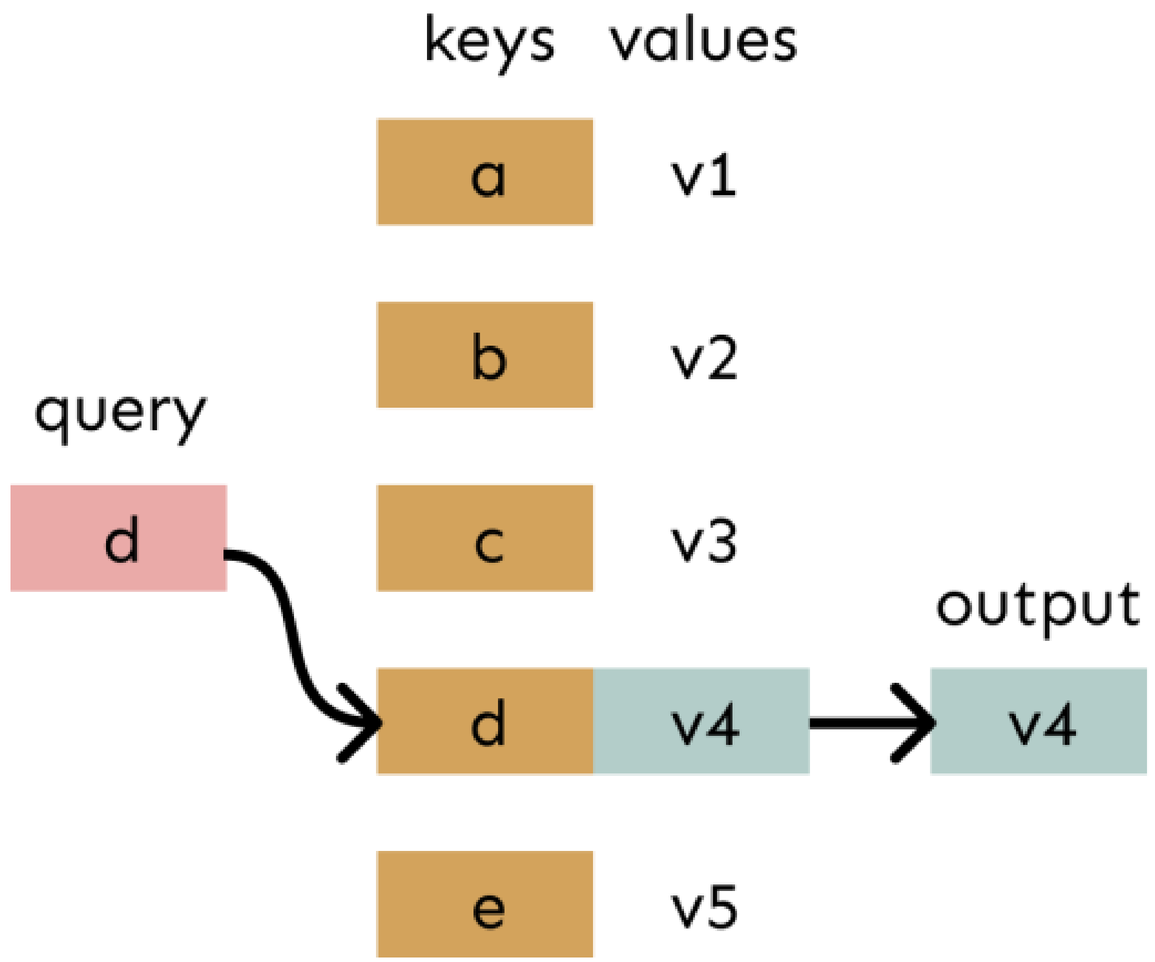
Attention is a continuous way to do lookups

Attention is just a **weighted** average – this is very powerful if the weights are learned!

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



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



Key-Value Attention

- Project an input vector x_i into two vectors
  k: key vector $k_i = W^K x_i$
  v: value vector $v_i = W^V x_i$

- Use key vector for computing attention

  $\phi^{att}(q, x_i) = \phi^{att}(q, k_i) = \frac{k_i . q}{\sqrt{d}}$ //scaled multiplicative

- Use value vector for computing attended summary

$$c = \sum_{i=1}^{T} \alpha_i . v_i$$

Key-Value Single-Head Self Attention

- Project an input vector x_i into vectors
  k: key vector: $k_i = W^K x_i$
  v: value vector: $v_i = W^v x_i$
  q: vector: $q_i = W^Q x_i$

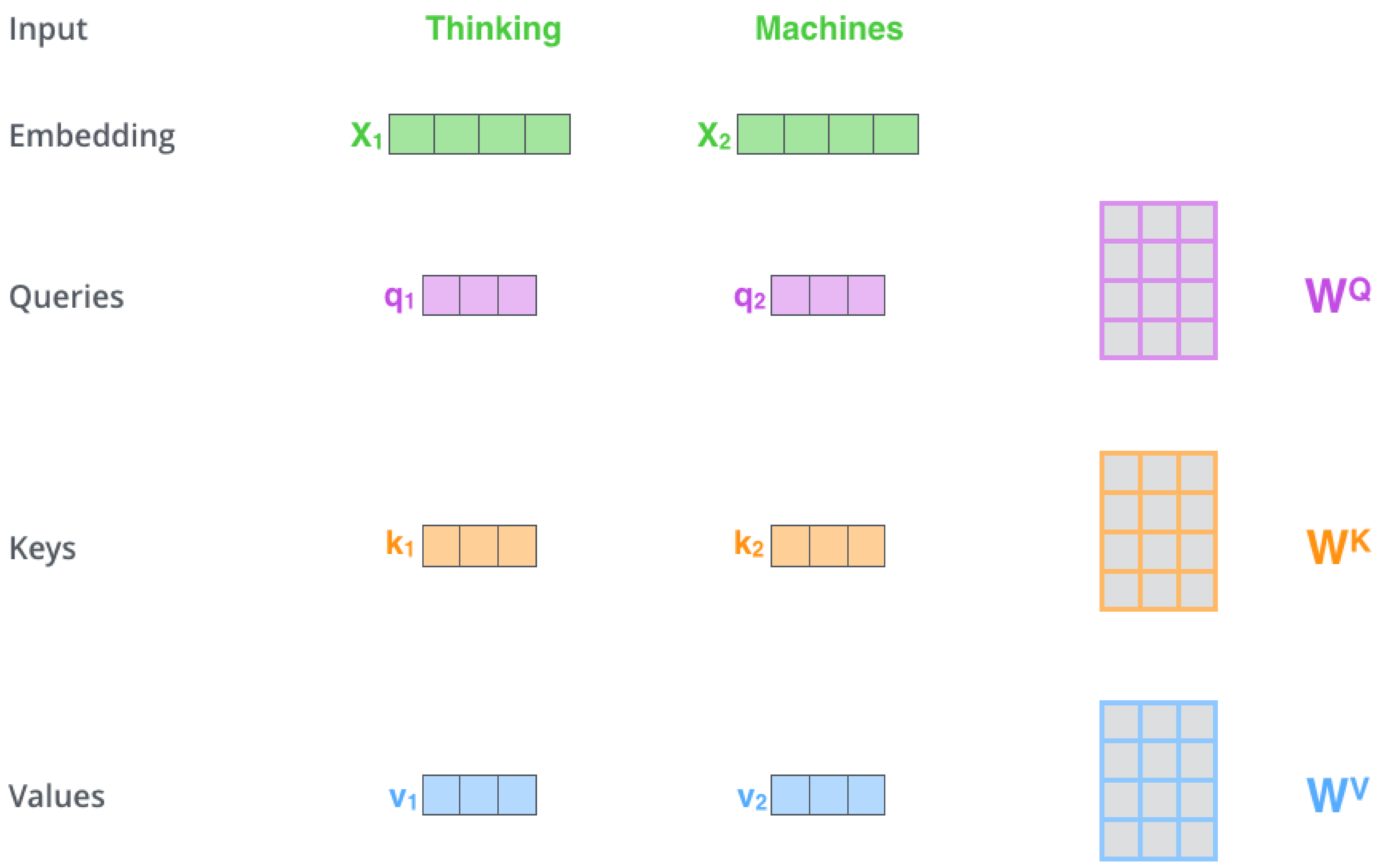
- Use key and query vectors for computing attention of i^{th} word at word j

  $\phi^{att}(x_j; x_i) = \frac{k_i . q_j}{\sqrt{d}}$ //scaled multiplicative

- Use value vector for computing attended summary

$$= \sum_{i=1}^{T} \alpha_i . v_i$$

Key-Value Single-Head Self Attention



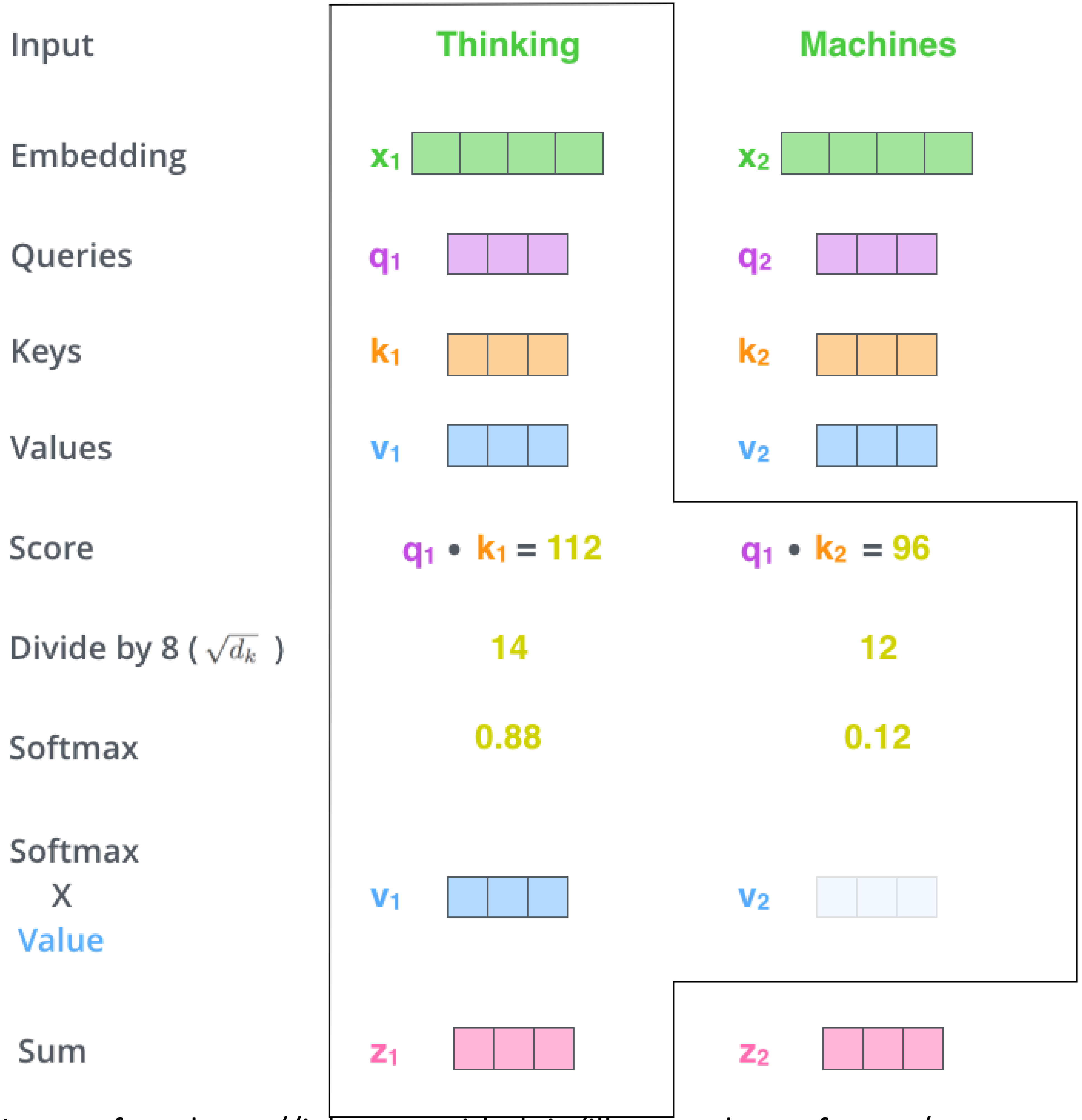
Creation of query, key and value vectors by multiplying by trained weight matrices

Separation of Value and Key and Query

Matrix multiplications are quite efficient and can be done in aggregated manner

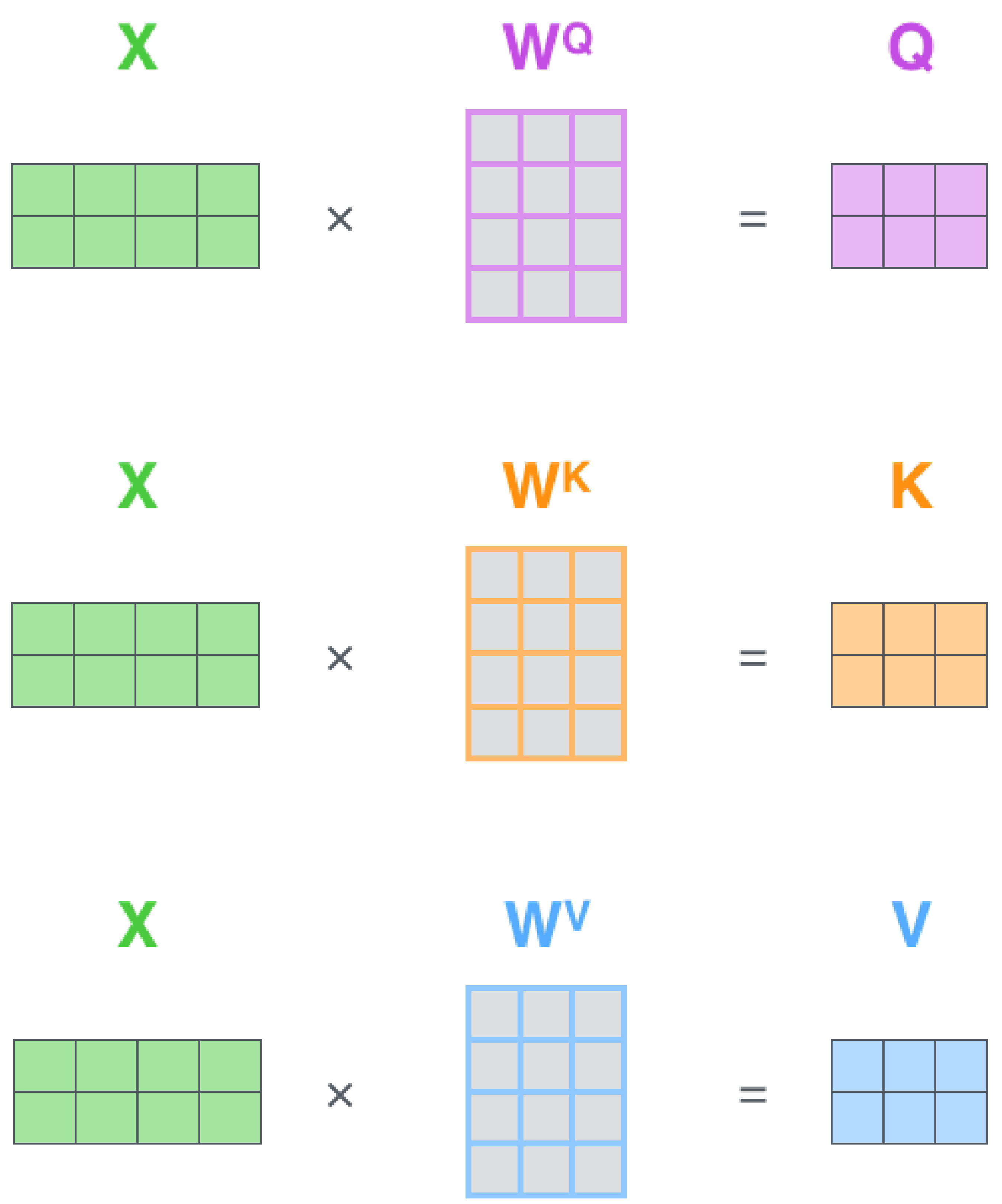
Images from https://jalammar.github.io/illustrated-transformer/

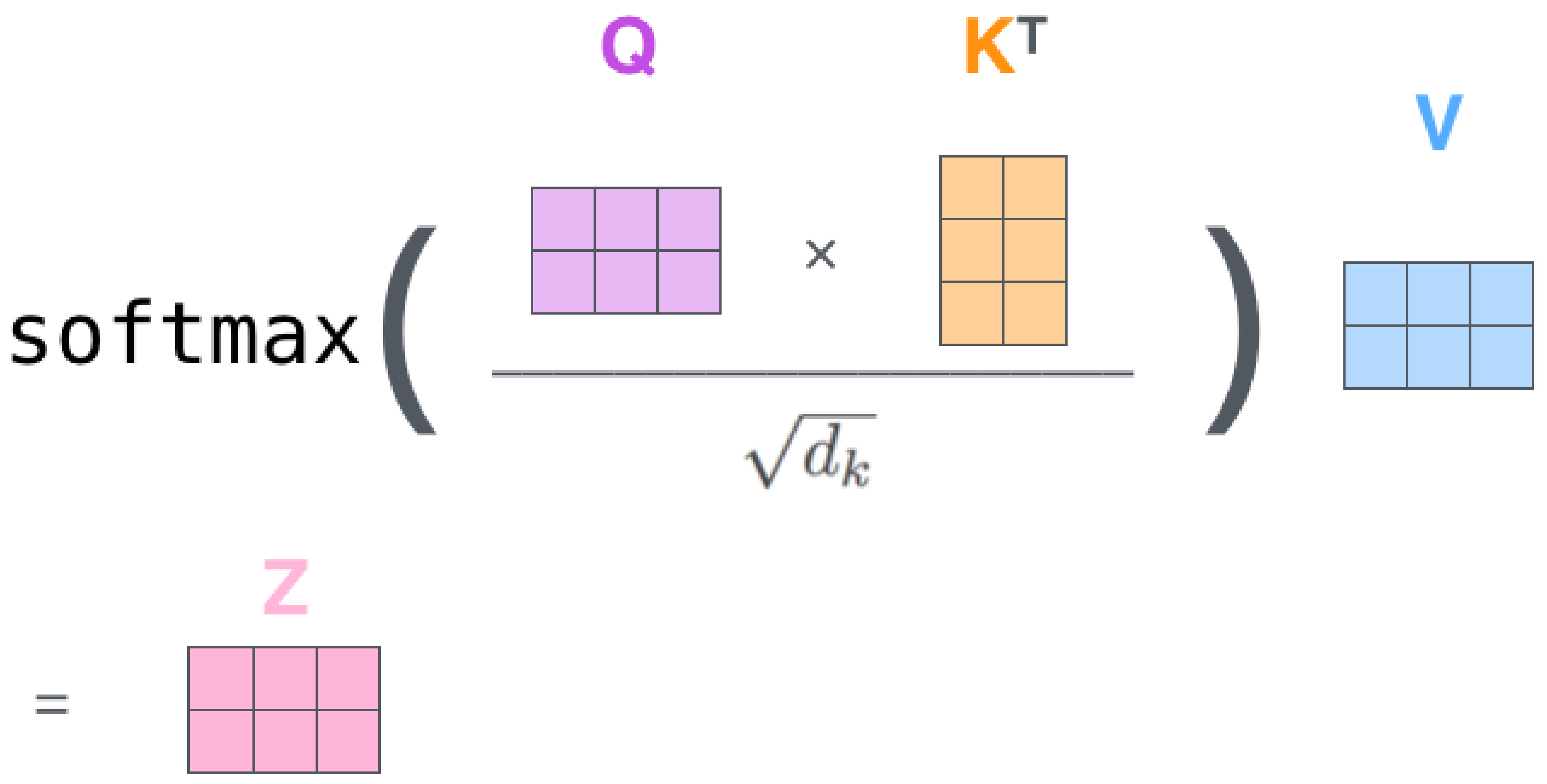
Key-Value Single-Head Self Attention



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

Key-Value Single-Head Self Attention





$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V = Z$$

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

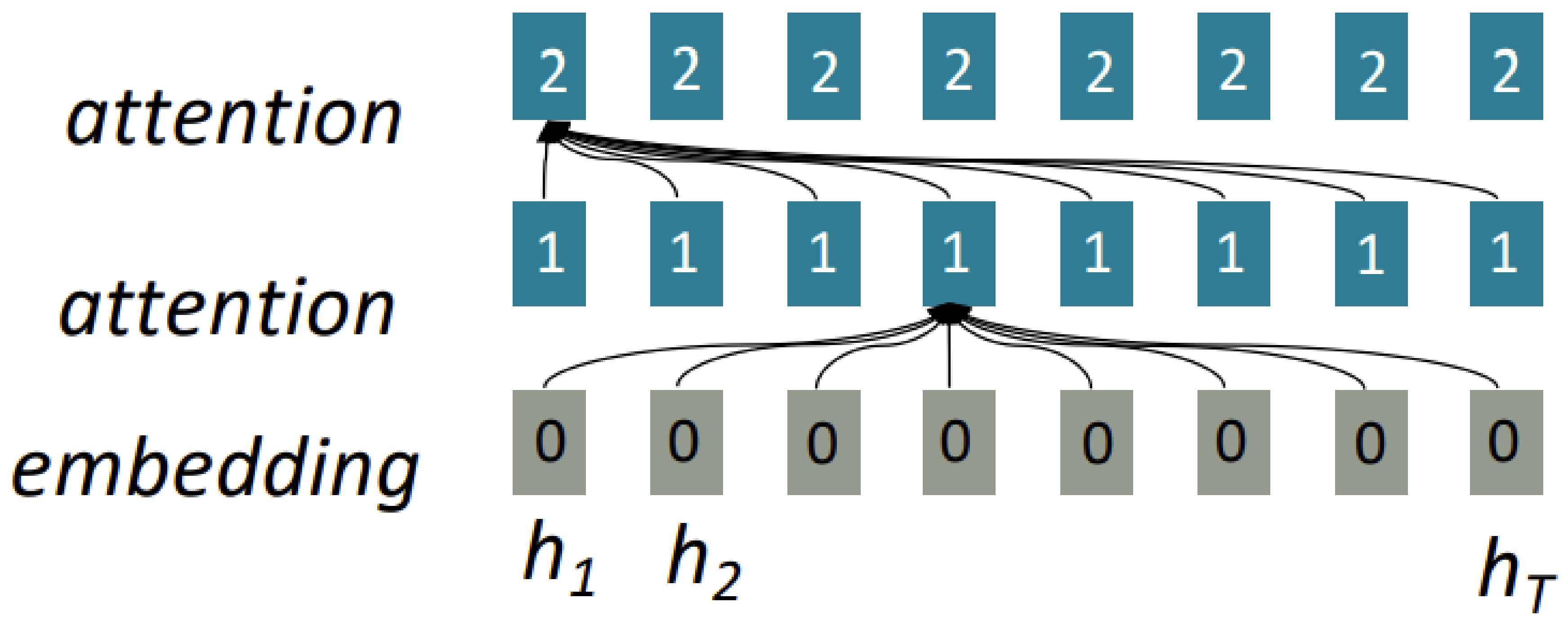
Self Attention is Parallelizable & solves linear distance issues

Attention treats each word's representation as a **query** to access and incorporate information from **a set of values.**

- We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.

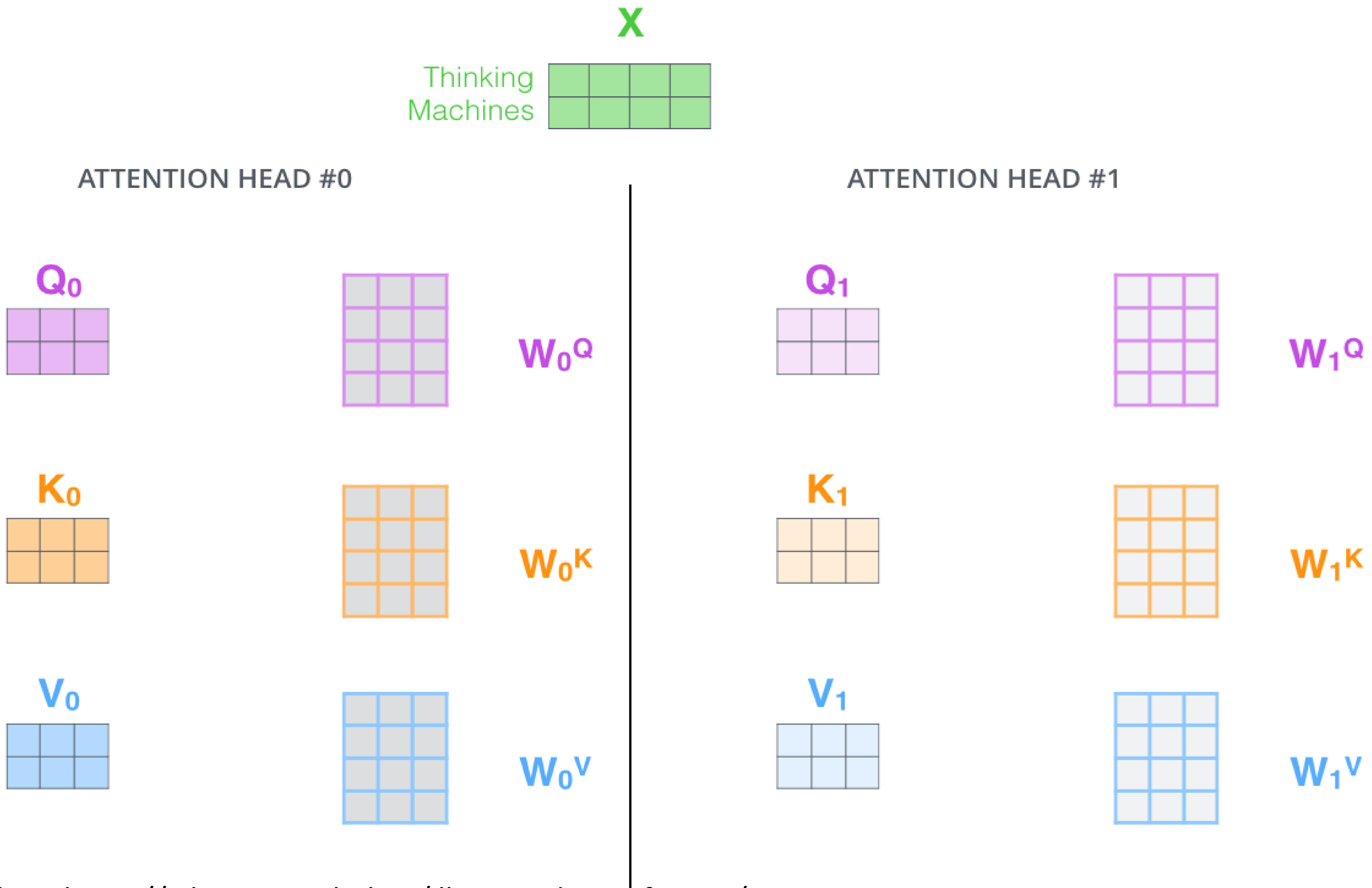
Number of unparallelizable operations does not increase with sequence length.

Maximum interaction distance: O(1), since all words interact at every layer!



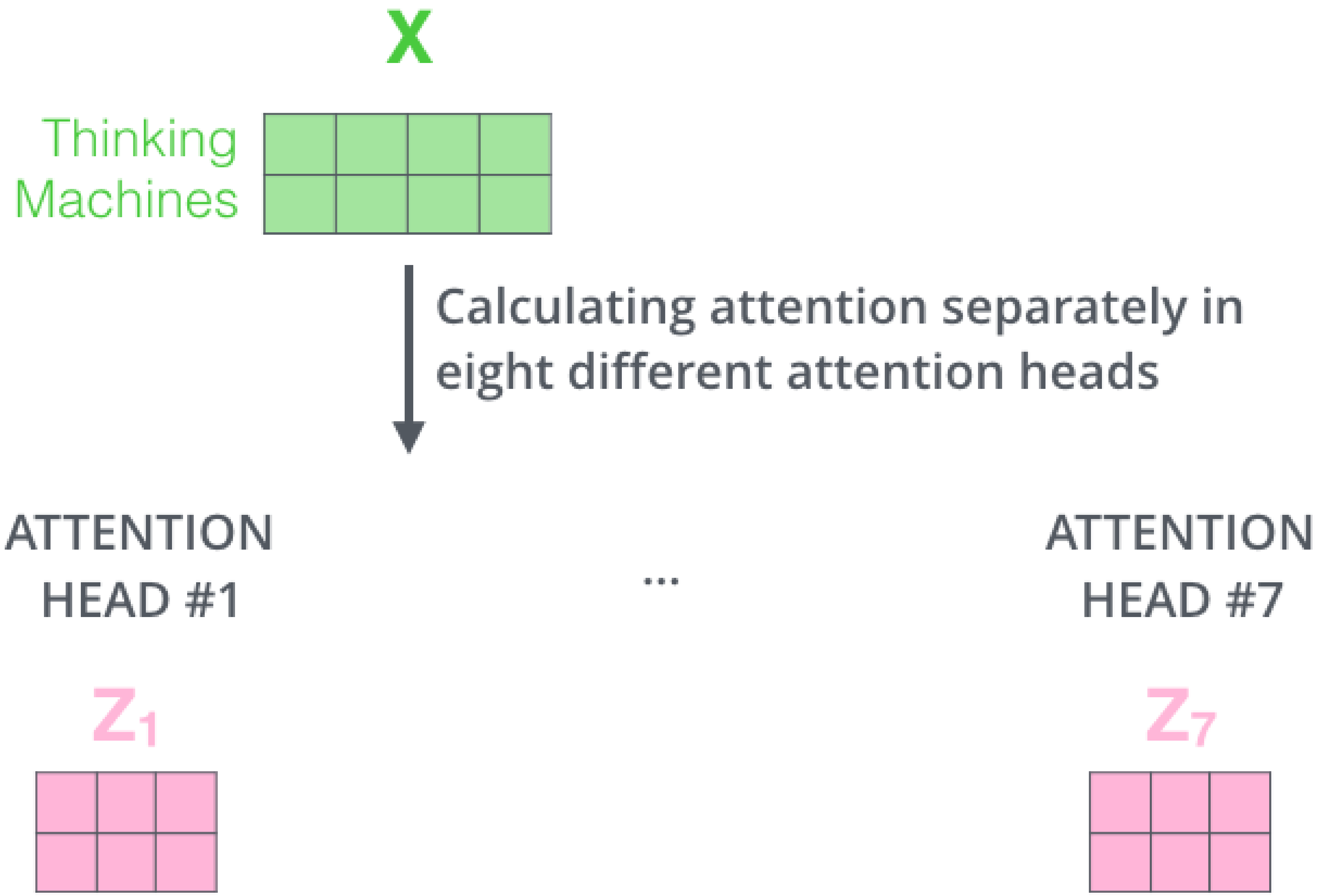
All words attend to all words in previous layer; most arrows here are omitted

Key-Value Multi-Head Self Attention



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

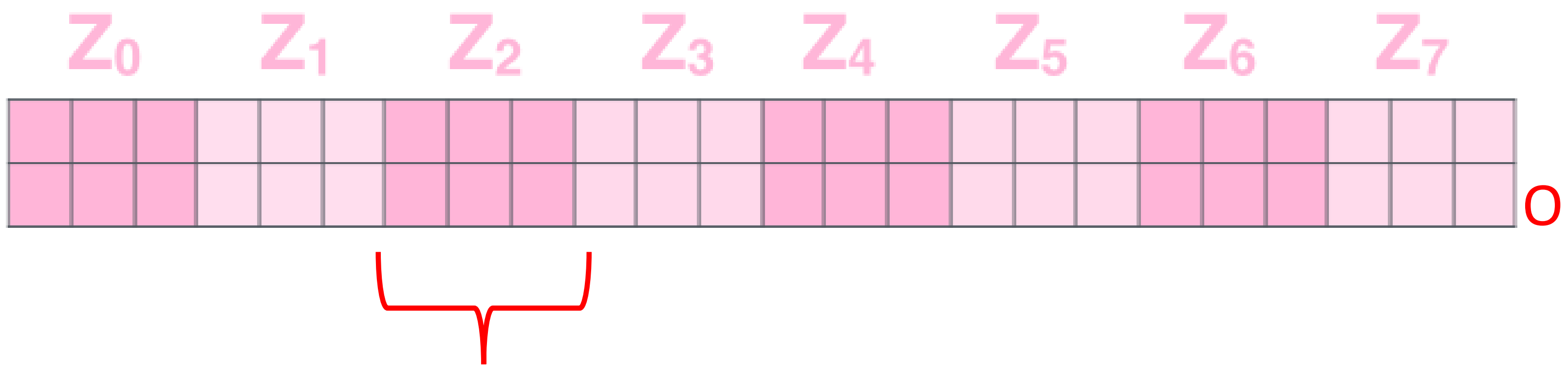
Multi-Head Attention



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

Multi-Head Attended Vector → Output

1) Concatenate all the attention heads

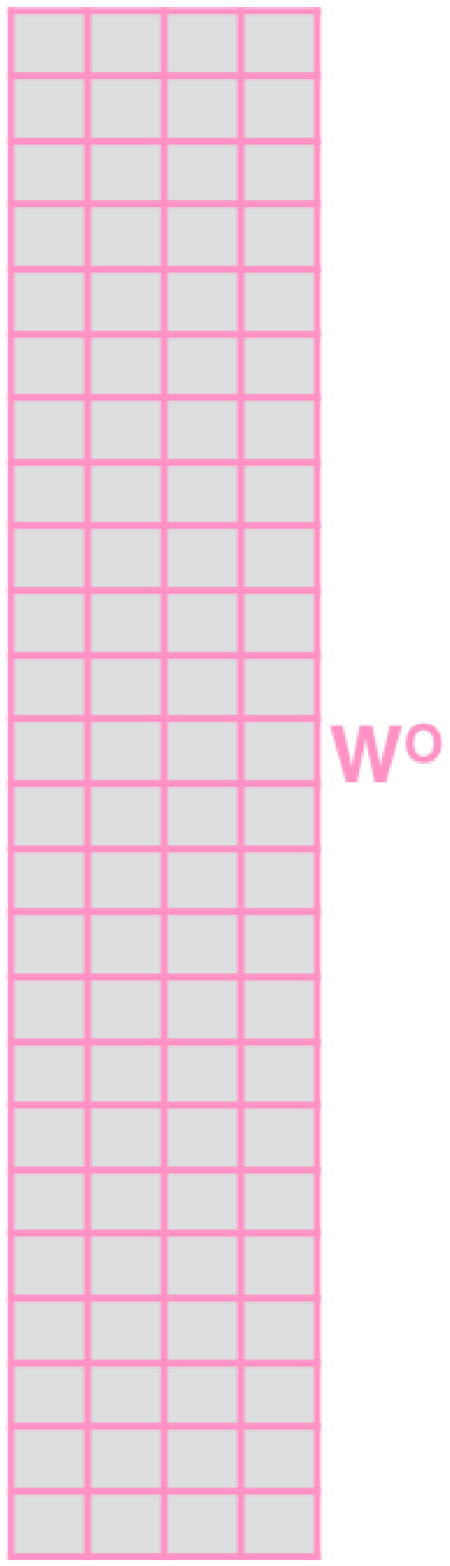


One for each word

One for each attention head

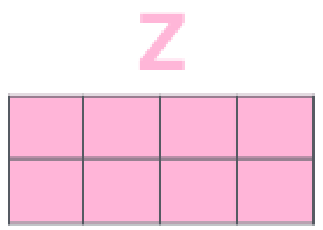
2) Multiply with a weight matrix W^O that was trained jointly with the model

X



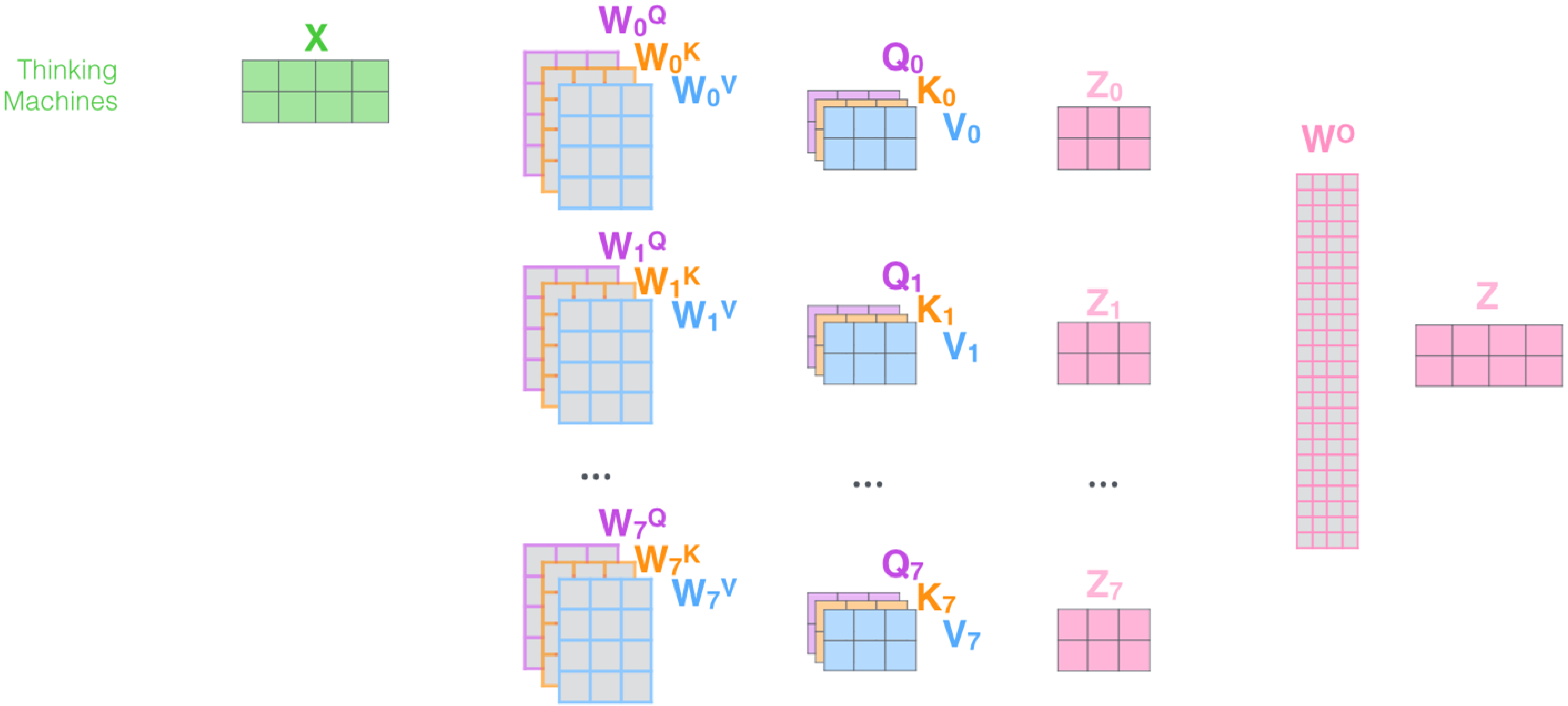
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

=



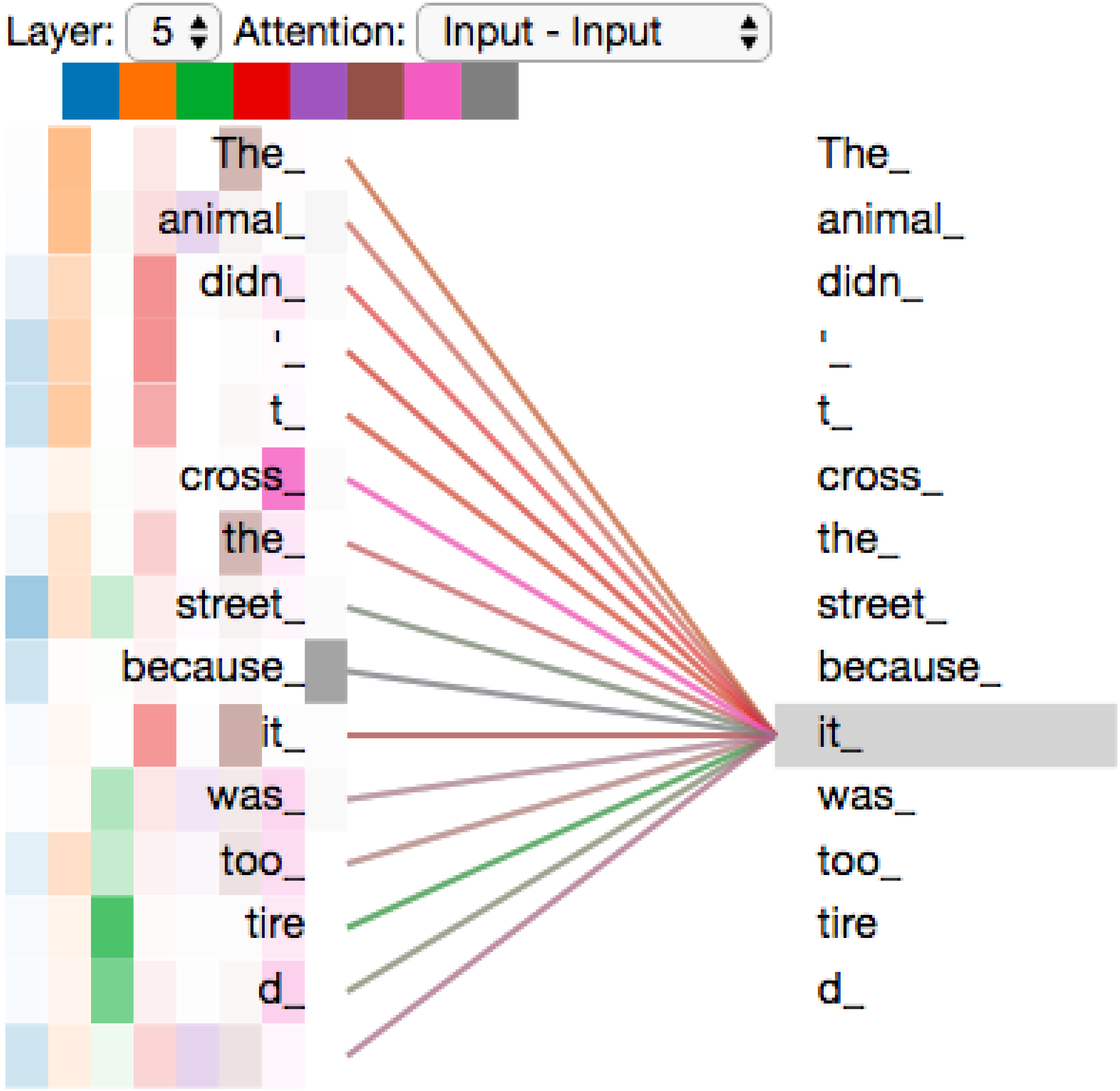
Images from https://jalammar.github.io/illustrated-transformer/

Key-Value Multi-Head Self Attention (summary)



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

Multi-head Self attention visualisation (Interpretable?!)



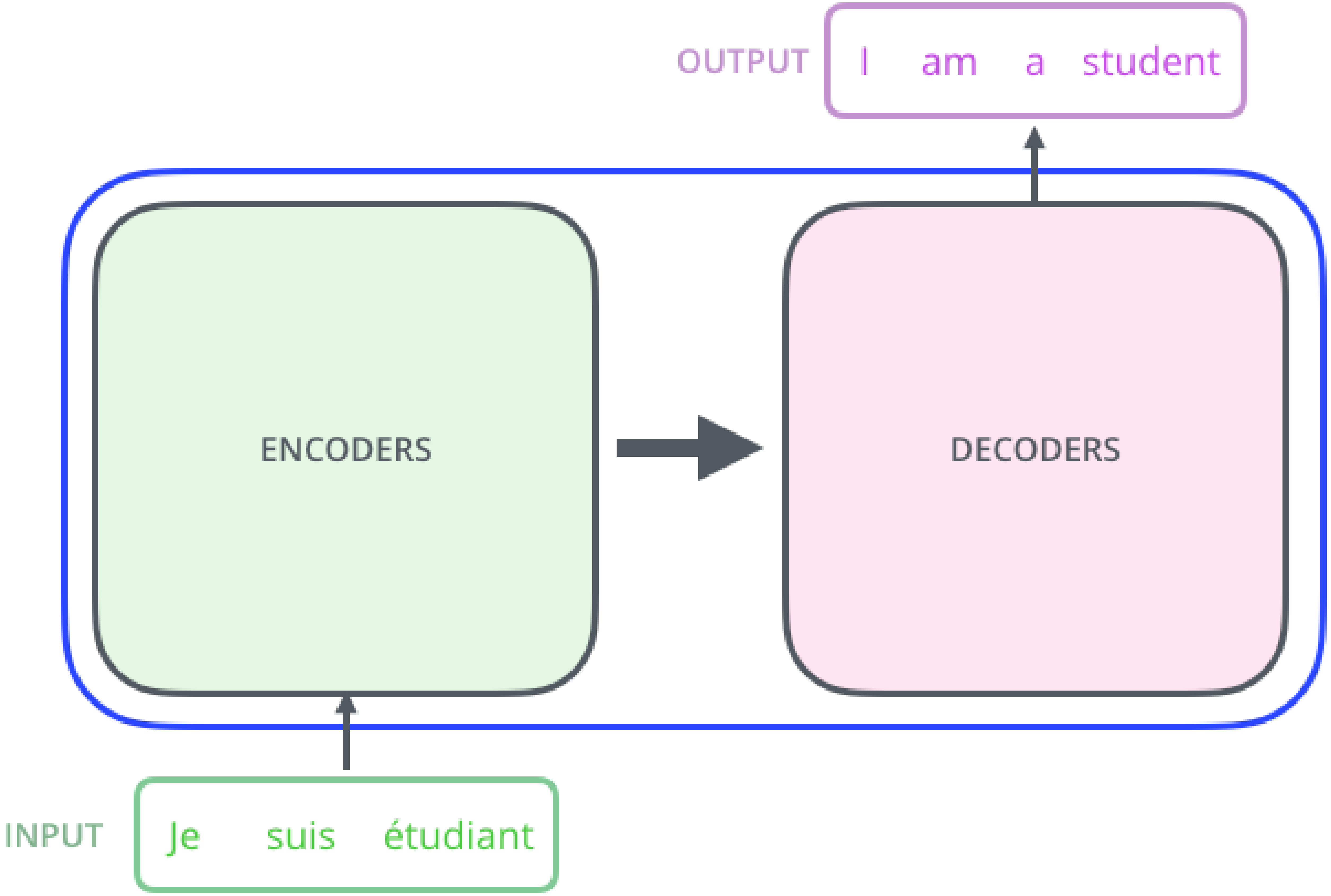
Images from https://jalammar.github.io/illustrated-transformer/

Transformer Encoders

Do we need recurrence at all?

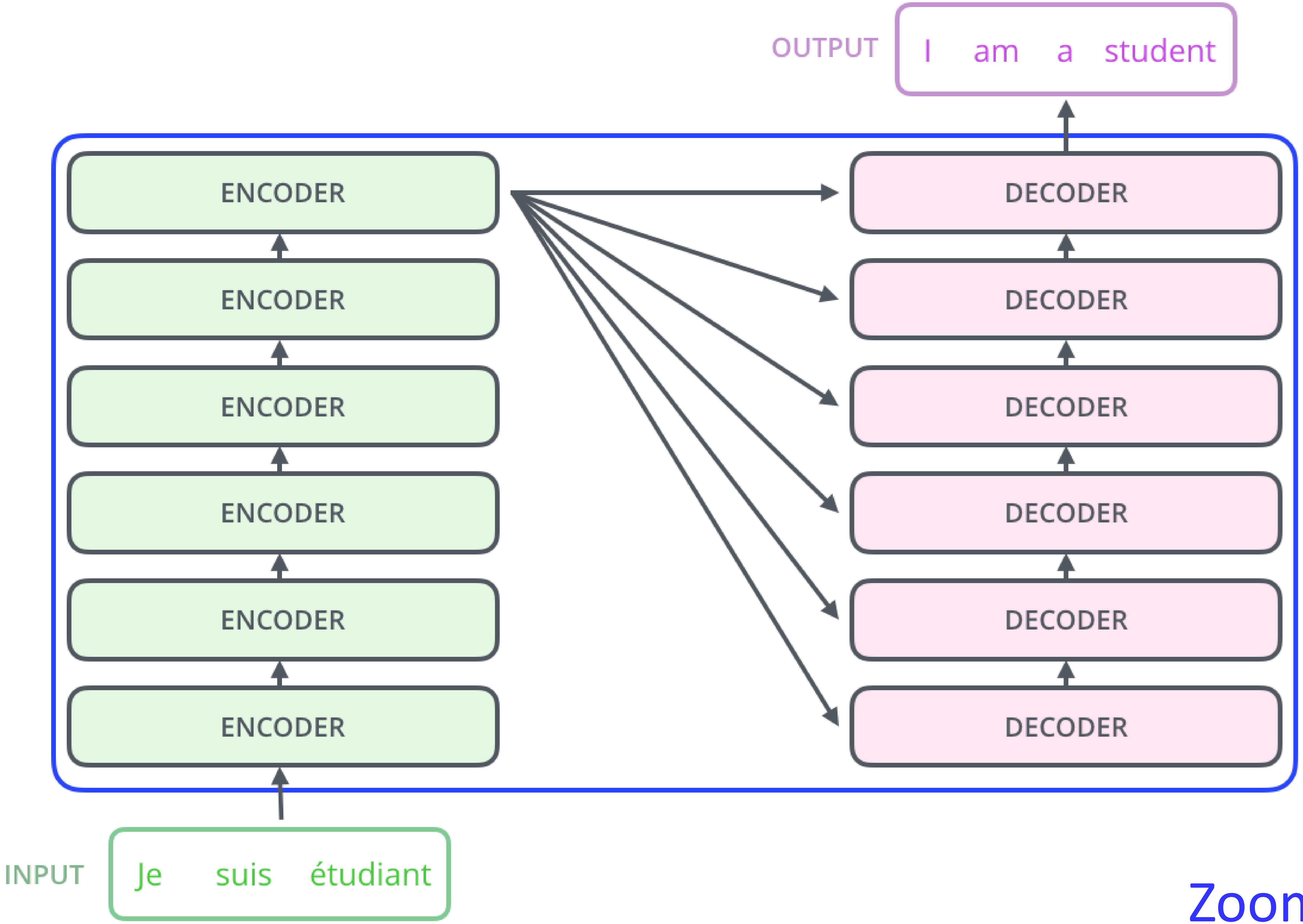
- Recurrence is powerful but
  - Issues with learnability: vanishing gradients
  - Issues with remembering long sentences
  - Issues with scalability:
    - backpropagation time high due to sequentiality in sentence length
  - Issues with scalability:
    - can’t be parallelized even at test time – O(sentence length)

- Abstractly: Attention is a way to pass information from a sequence input to a neural network input
  - That is also exactly what RNNs are used for – to pass information
  - Get rid of RNN? – maybe attention is a better way to pass information



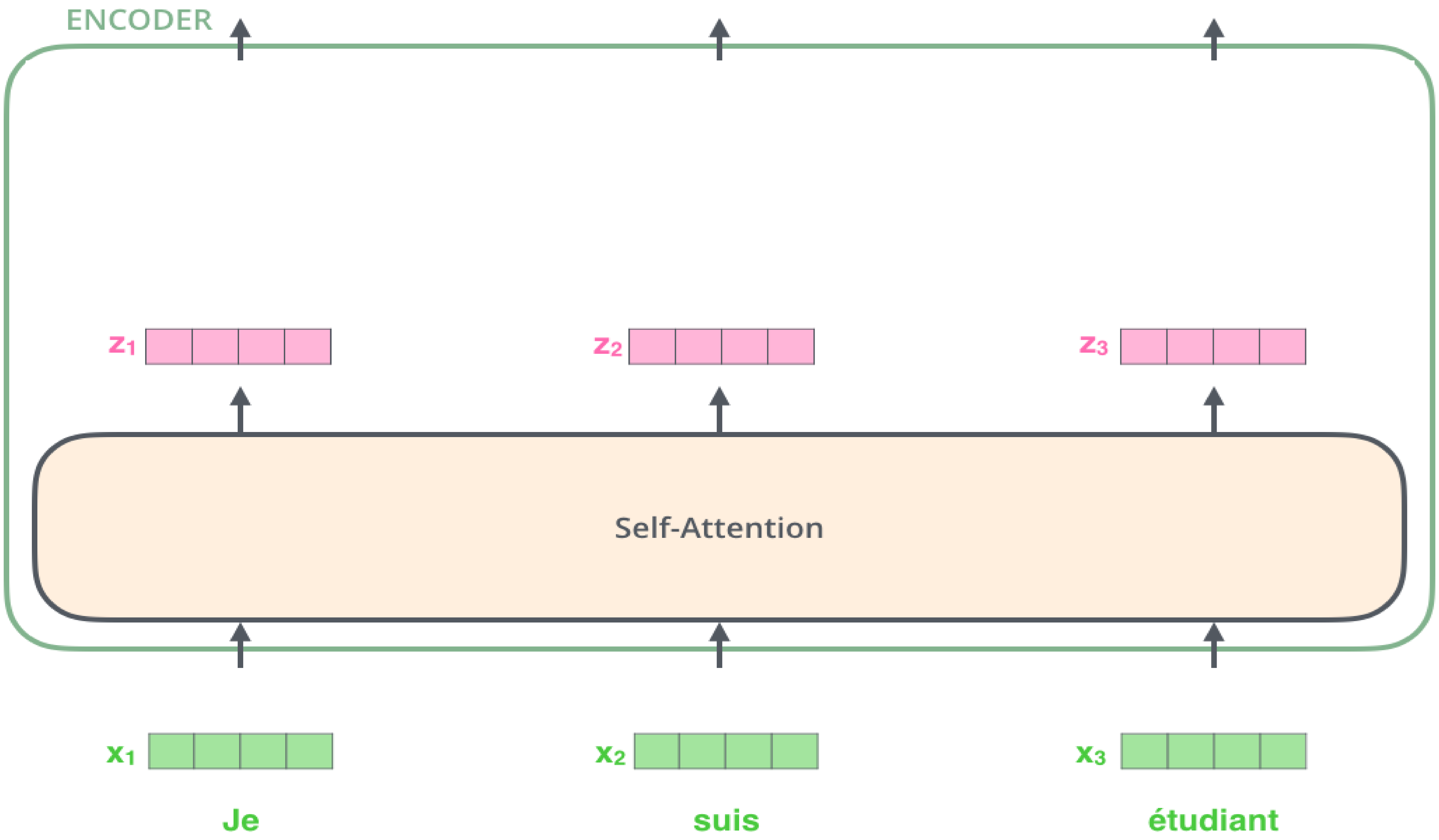
We focus only on encoder for now… (decoder is an extension of sequence decoders)

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



Zooming in...

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



Can you see a fundamental limitation?

Encoders have same architecture but different weights…

Zooming in further...

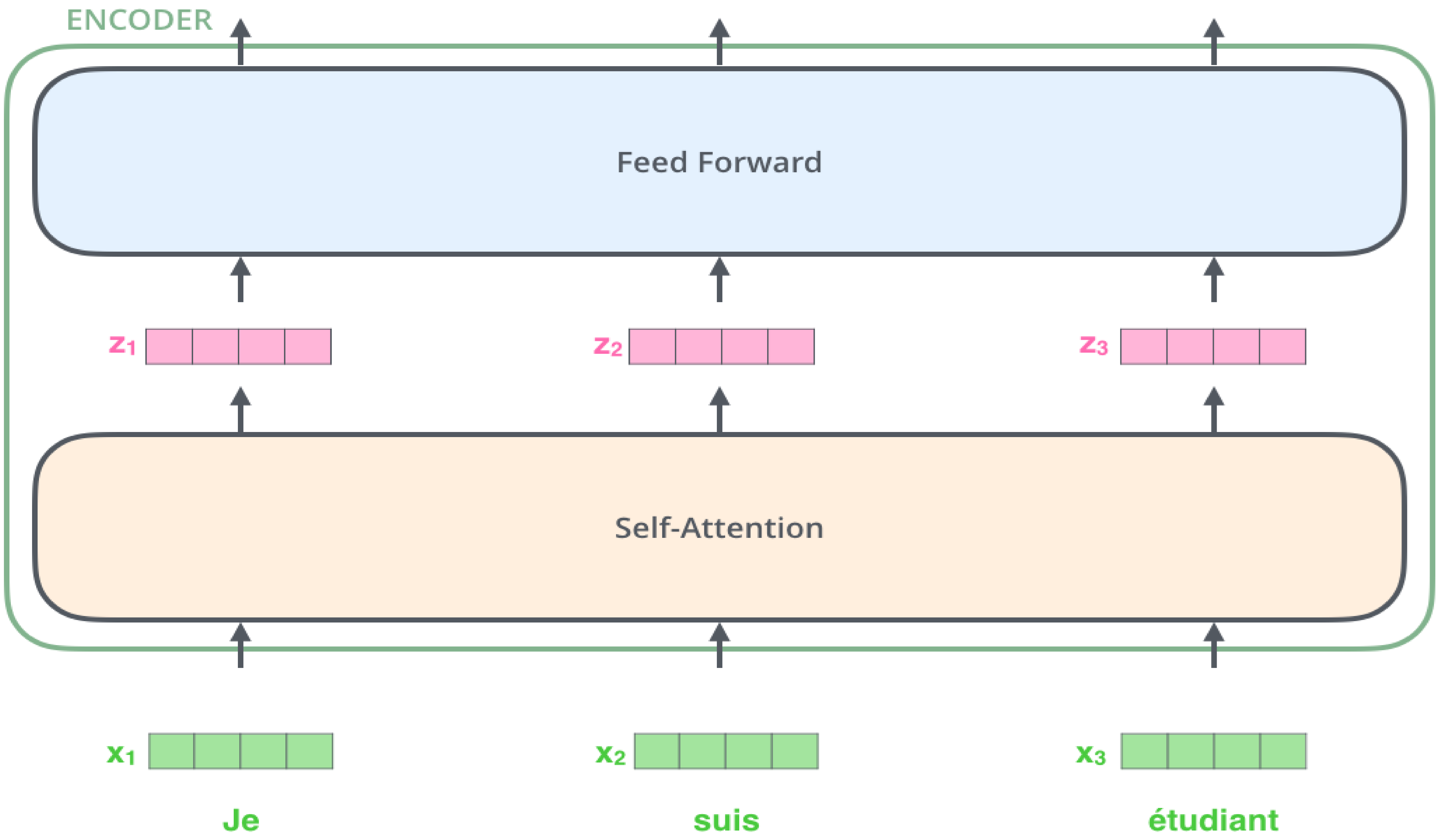
Images from https://jalammar.github.io/illustrated-transformer/

Issue

- No non-linearities for deep learning
  - Its all just weighted averages

- Solution: add a feed forward layer

$$W_2 * \mathrm{ReLU}(W_1 \, \mathrm{output}_i + b_1) + b_2$$



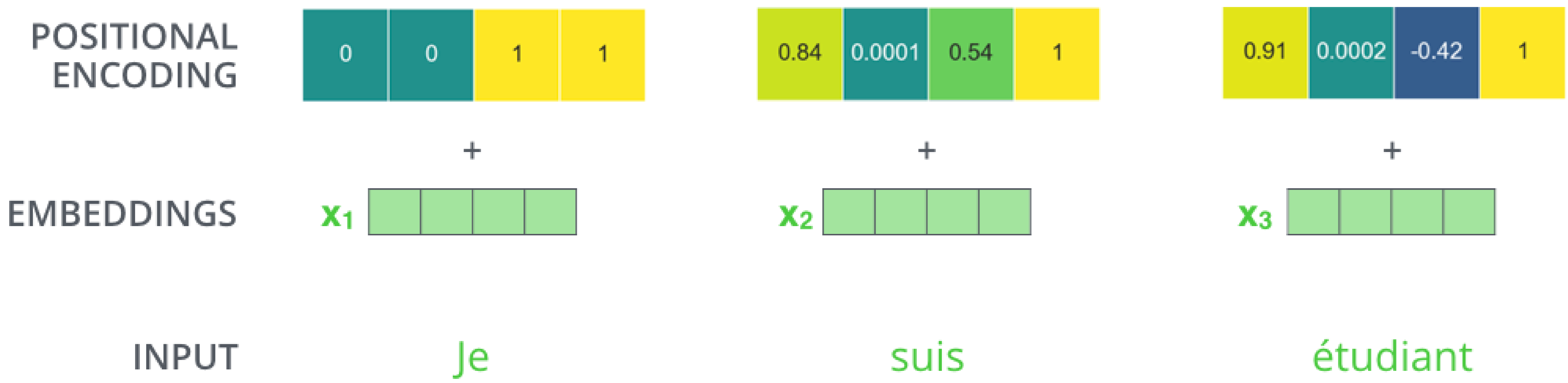
Can you see a fundamental limitation?

Encoders have same architecture but different weights…

Zooming in further…

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

A note on Positional embeddings



Positional embeddings can be extended to any sentence length but if any test input is longer than all training inputs then we will face issues.

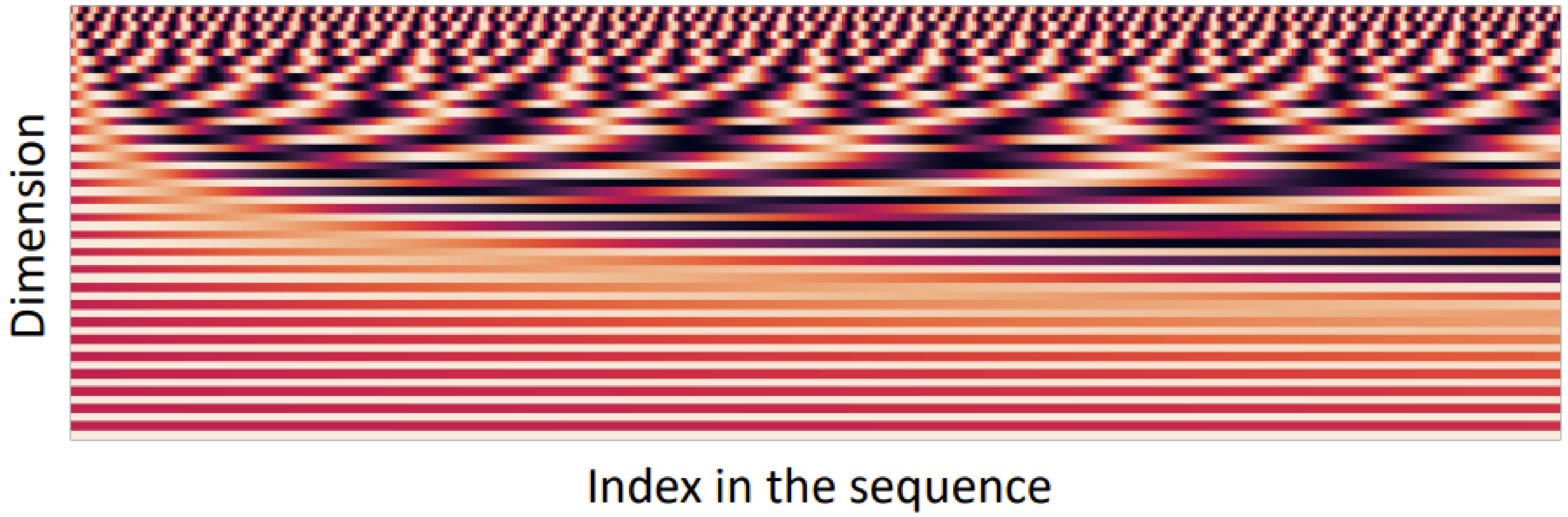
Solution: use a functional form (as in Transformer paper – sinuisoidal encoding)

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

Sinusoidal Embeddings

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

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



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

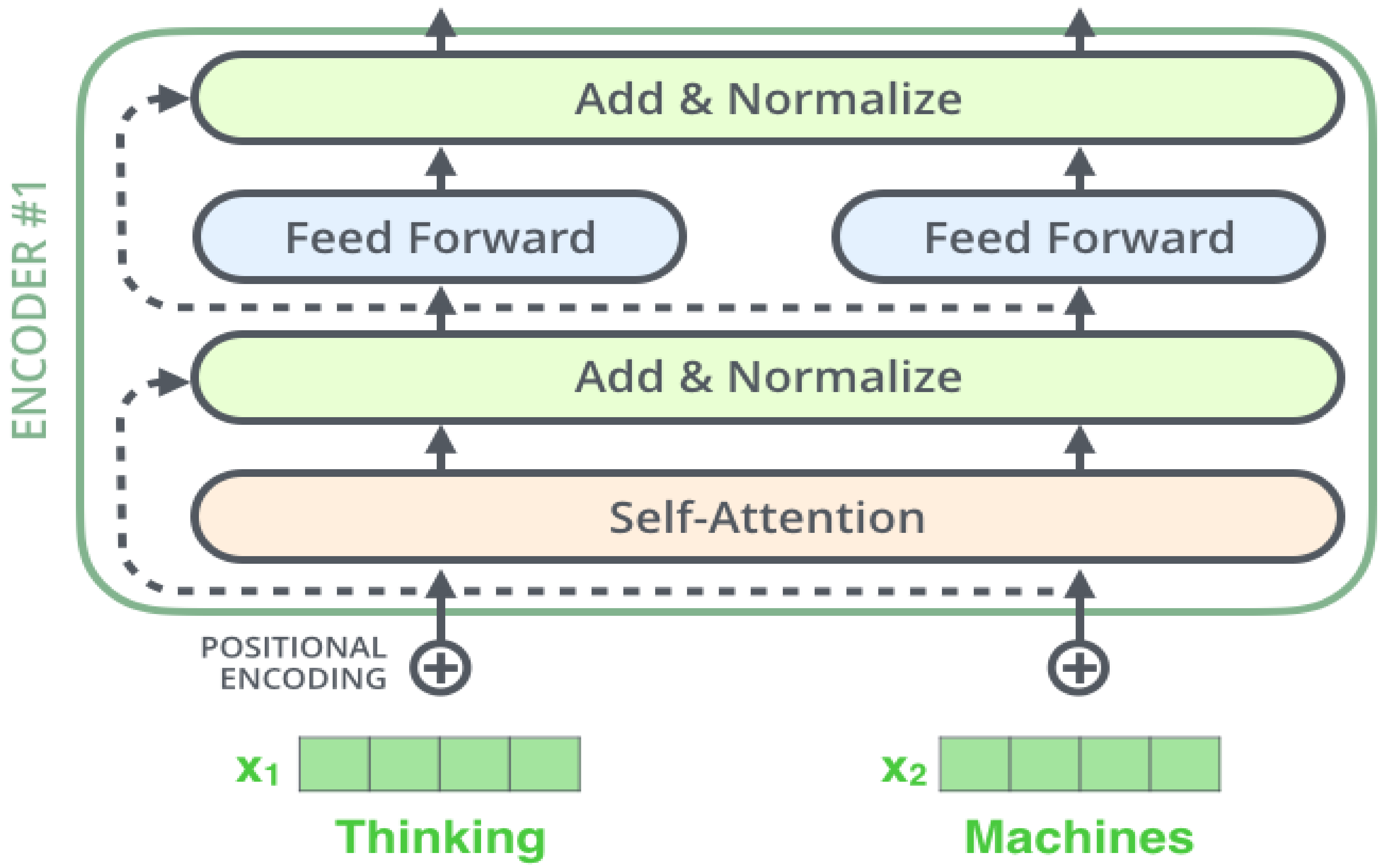
Activate Window

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

Position Encodings: Learned from Scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters! Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$, *and let each* $\boldsymbol{p}_i$ *be a column of that matrix!*
- Pros:
  - Flexibility: each position gets to be learned to fit the data
- Cons:
  - Definitely can't extrapolate to indices outside $1, \ldots, n$.
- Most systems use this!
- Sometimes people try more flexible representations of position:
  - Relative linear position attention [Shaw et al., 2018]
  - Dependency syntax-based position [Wang et al., 2019]

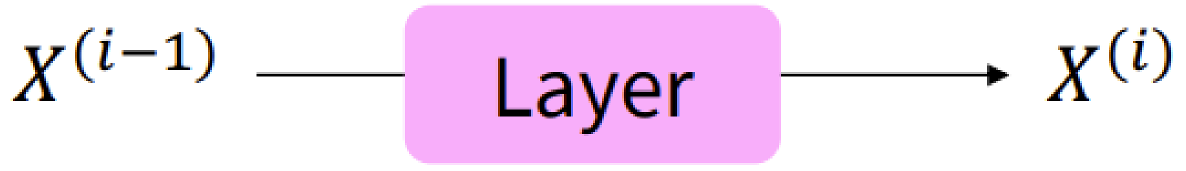
Two more Components



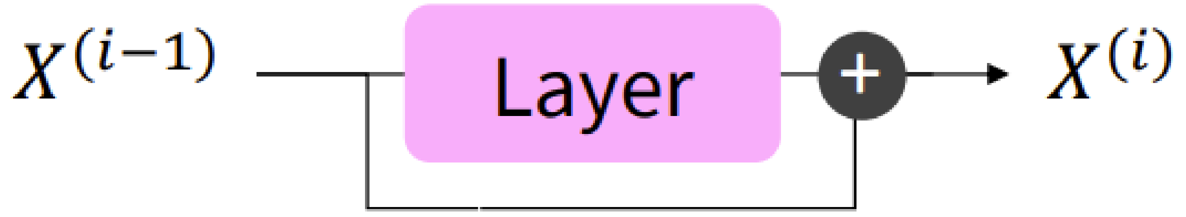
- Now that we've replaced self-attention with multi-head self-attention, we'll go through two **optimization tricks** that end up being :
  - **Residual Connections**
  - **Layer Normalization**
- In most Transformer diagrams, these are often written together as "Add & Norm"

Residual Connections

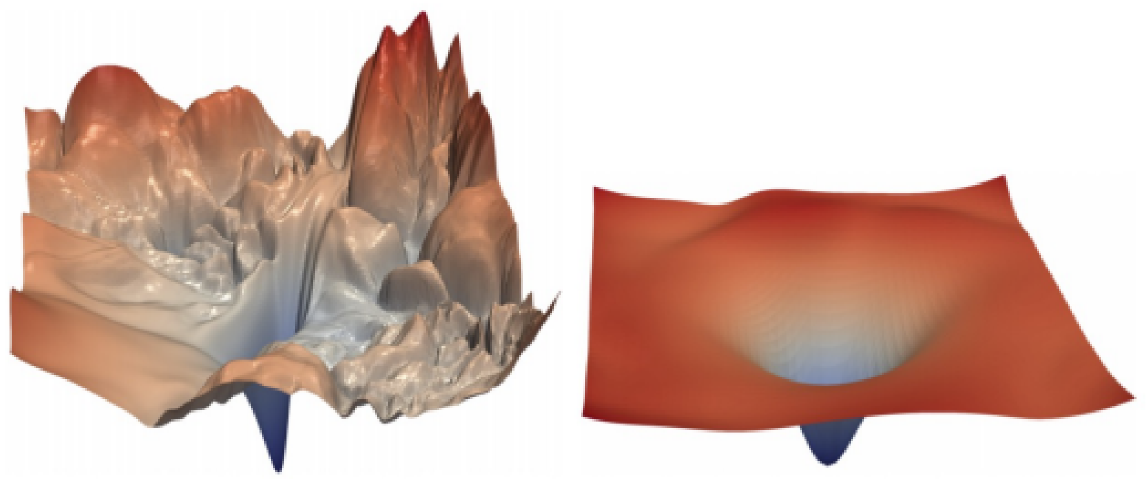
- **Residual connections** are a trick to help models train better.
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)



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



  - Gradient is **great** through the residual connection; it's 1!
  - Bias towards the identity function!



[no residuals] [residuals]

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

Activate Win

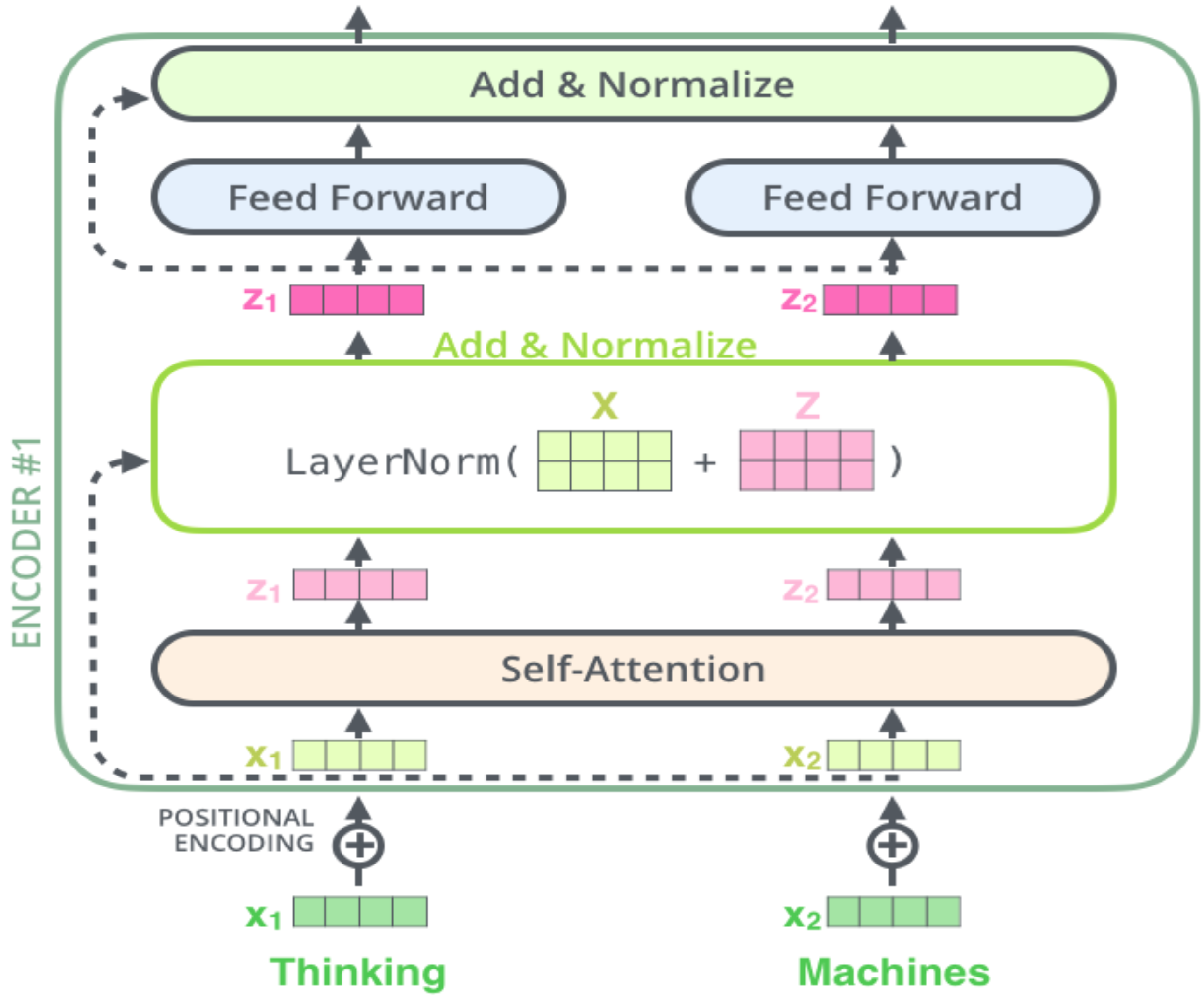
Layer Normalization

- **Layer normalization** is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
  - LayerNorm's success may be due to its normalizing gradients [Xu et al., 2019]
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.
- Let $\mu = \sum_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.
- Let $\sigma = \sqrt{\frac{1}{d} \sum_{j=1}^{d} (x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.
- Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned "gain" and "bias" parameters. (Can omit!)
- Then layer normalization computes:

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

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias



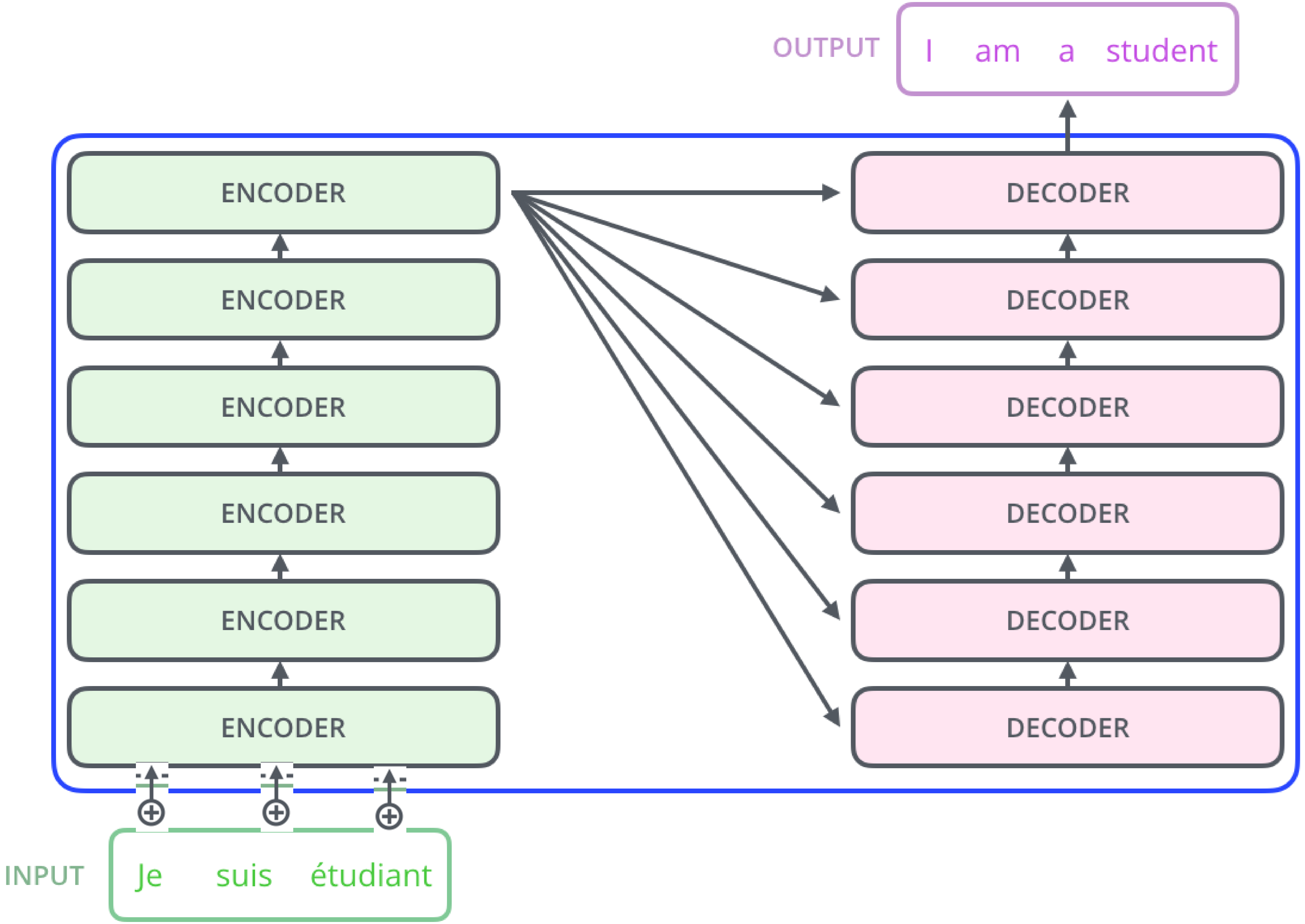
Images from https://jalammar.github.io/illustrated-transformer/

Regularization

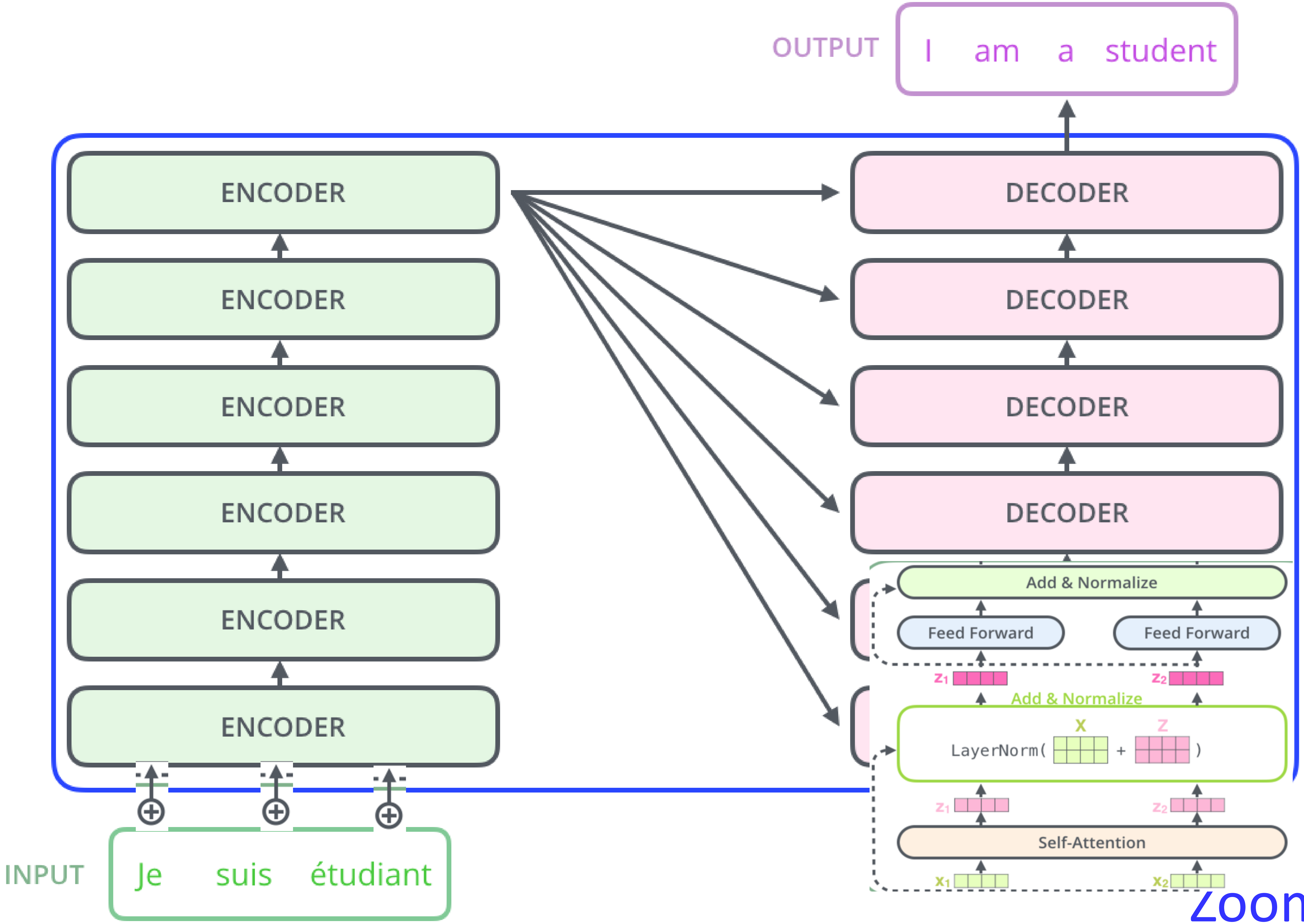
Residual dropout: Dropout added to the the output of each sublayer, before it is added to the input of the sublayer and normalized

Label Smoothing: During training label smoothing was employed. This hurts perplexity, as the model learns to be more unsure, but improves accuracy and BLEU score. (skip for now. Additional reading: https://towardsdatascience.com/what-is-label-smoothing-108debd7ef06)

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

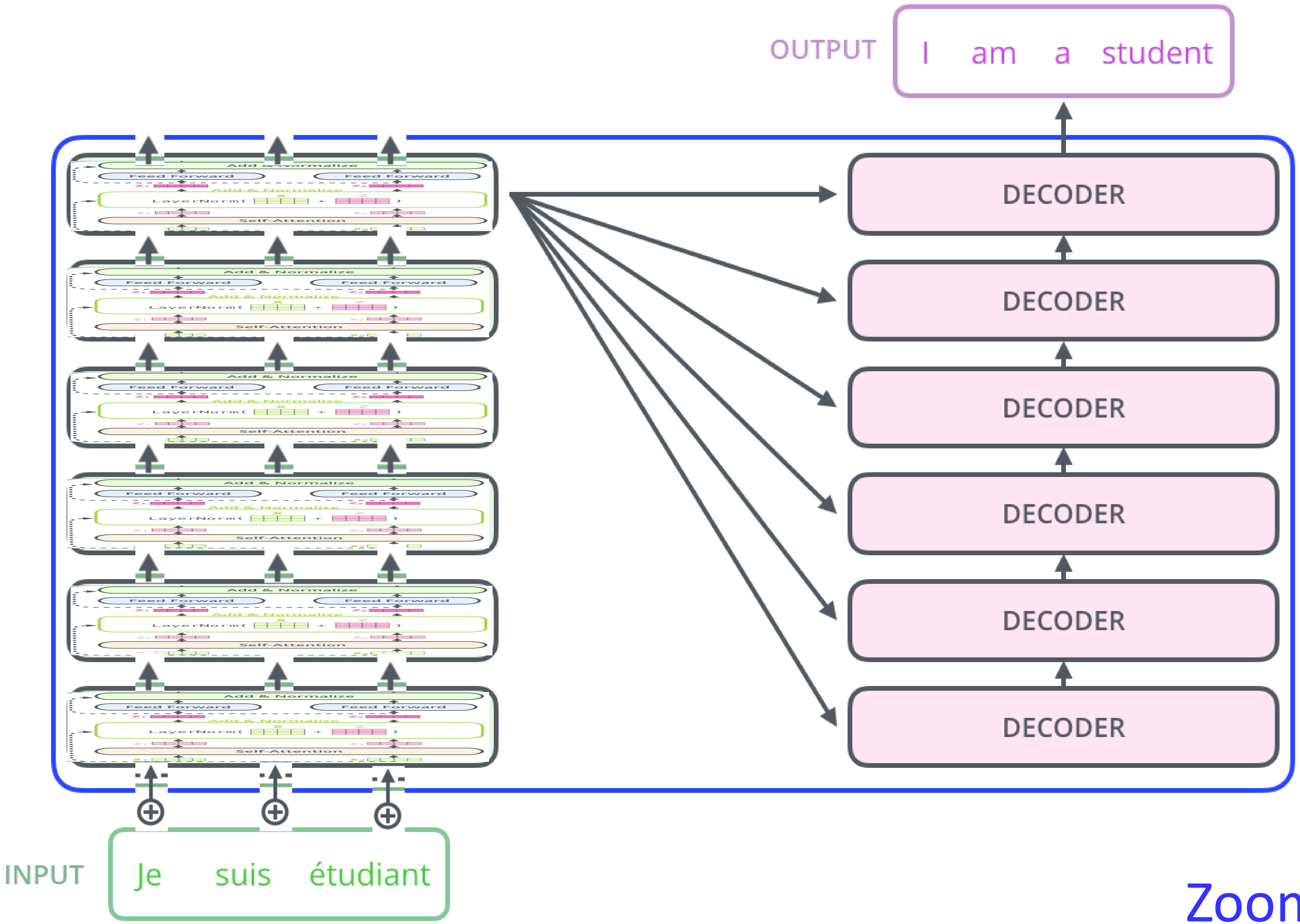


Images from https://jalammar.github.io/illustrated-transformer/



Zooming in...

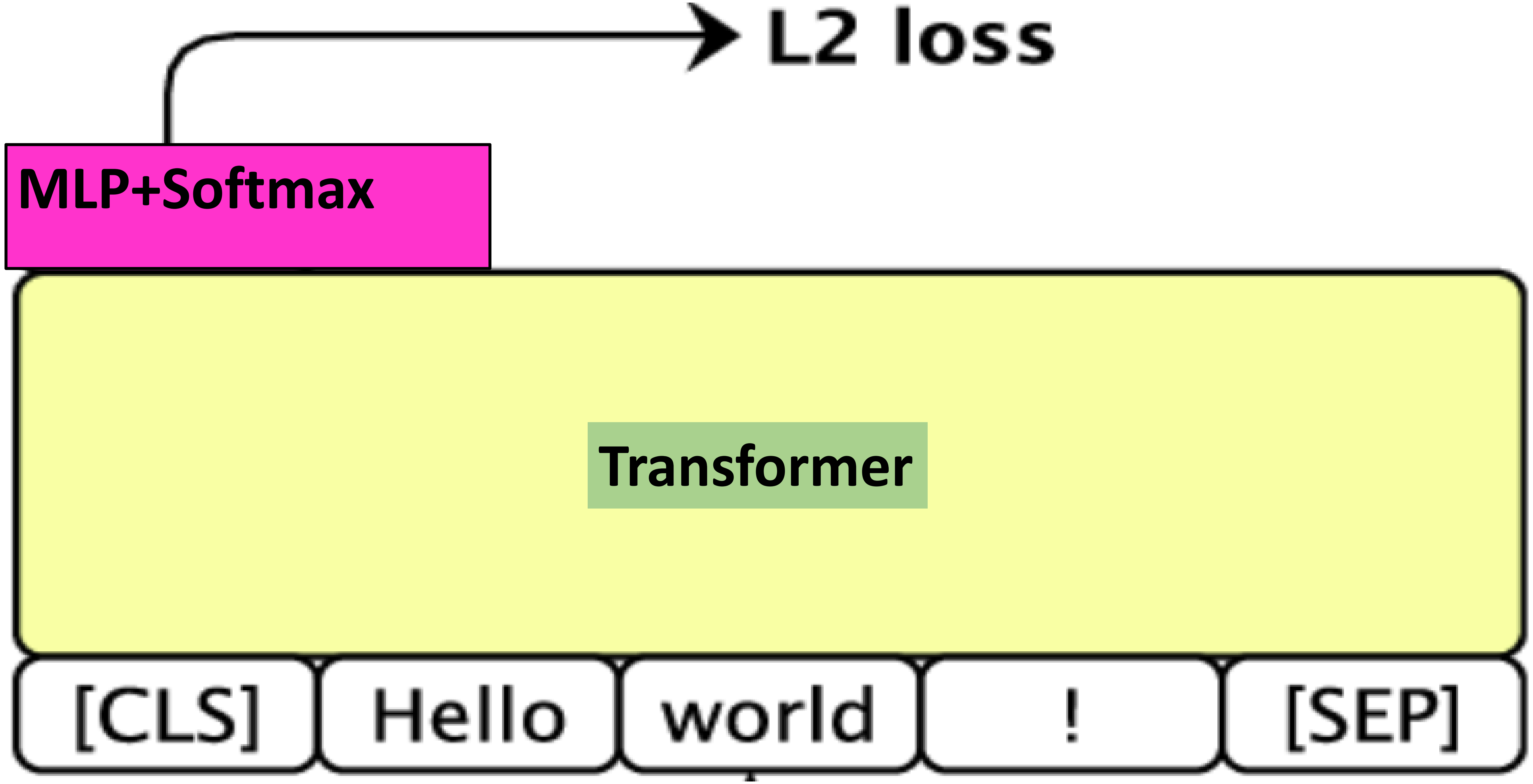
Images from https://jalammar.github.io/illustrated-transformer/



Zooming in...

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

Use of [CLS] for Text Classification



Pros

- Current state-of-the-art.
- Enables deep architectures
- Easier learning of long-range dependencies
- Can be efficiently parallelized
- Gradients don’t suffer from vanishing gradients

Cons

Huge number of parameters so

- Very data hungry
- Takes a long time to train (Why?)
  - Quadratic compute in self attention!
- Memory inefficient

Other issues

- Keeping sentence length limited
- How to ensure multi-head attention has diverse perspectives.