CS60075
Natural Language Processing
Autumn 2020

Module 8:

Transformers – Part 1

29 October 2020

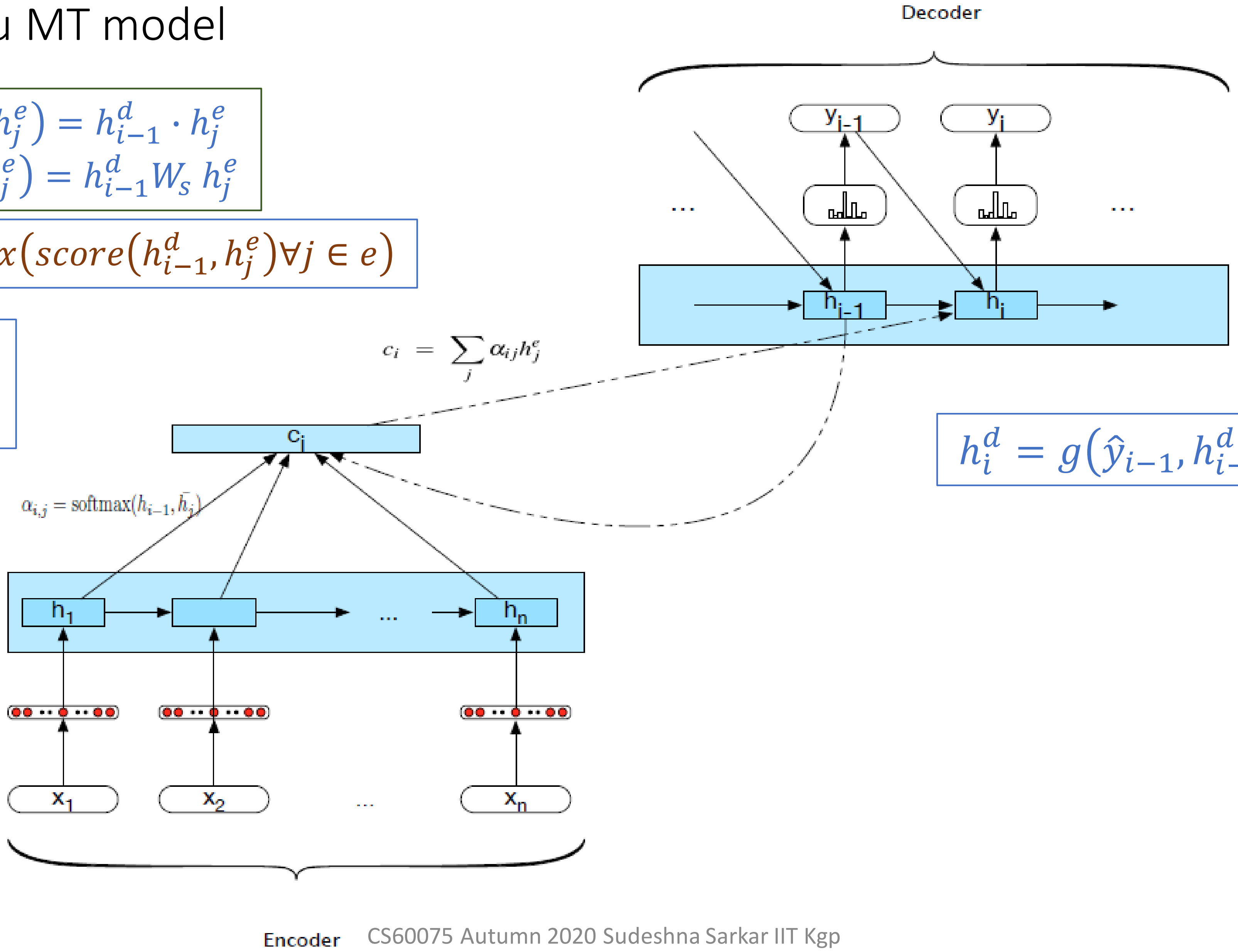
Encoder-Decoder Attention

in Bahdanau MT model

$$score1(h^d_{i-1}, h^e_j) = h^d_{i-1} \cdot h^e_j$$
$$score2(h^d_{i-1}, h^e_j) = h^d_{i-1} W_s\, h^e_j$$

$$\alpha_{i,j} = softmax(score(h^d_{i-1}, h^e_j) \forall j \in e)$$

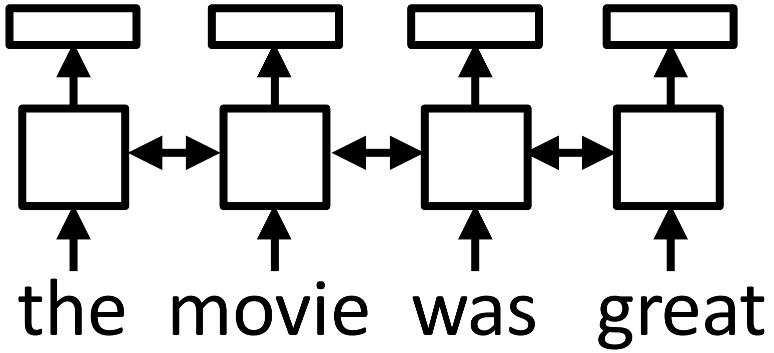
$$c_i = \sum_j \alpha_{i,j} h^e_j$$



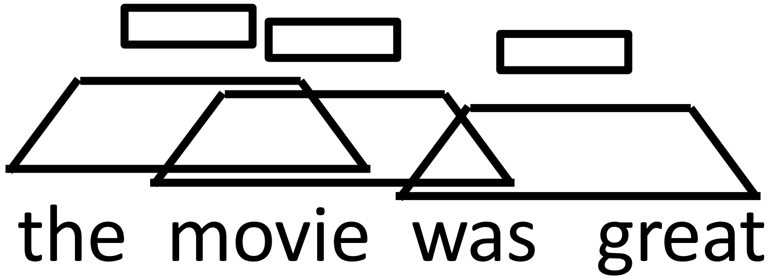
$$h^d_i = g(\hat{y}_{i-1}, h^d_{i-1}, c_i)$$

Encoders

- RNN: map each token vector to a new context-aware token using a autoregressive sequential process

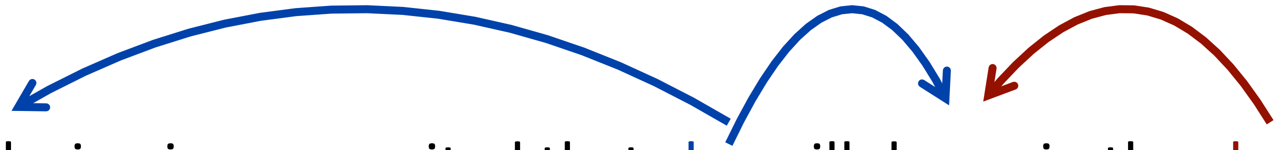


- Attention can be an alternative method to generate context-dependent embeddings



LSTM/CNN Context

- What context do we want token embeddings to take into account?

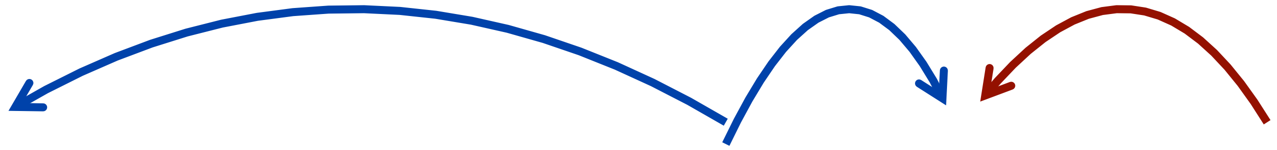


The ballerina is very excited that she will dance in the show.

- What words need to be used as context here?
  - Pronouns context should be the antecedents (i.e., what they refer to)
  - Ambiguous words should consider local context
  - Words should look at syntactic parents/children
- Problem: RNNs (i.e., LSTMs) and CNNs fail to do this

LSTM/CNN Context

Want:

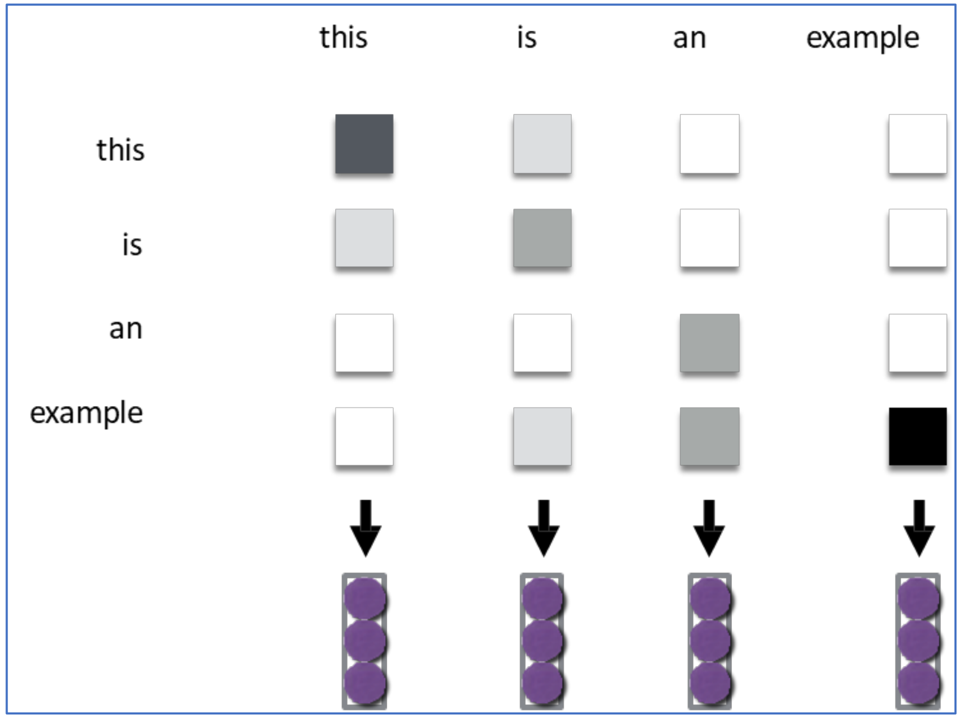


The ballerina is very excited that she will dance in the show.

- LSTMs/CNNs: tend to be local
- To appropriately contextualize, need to pass information over long distances for each word

Self-attention

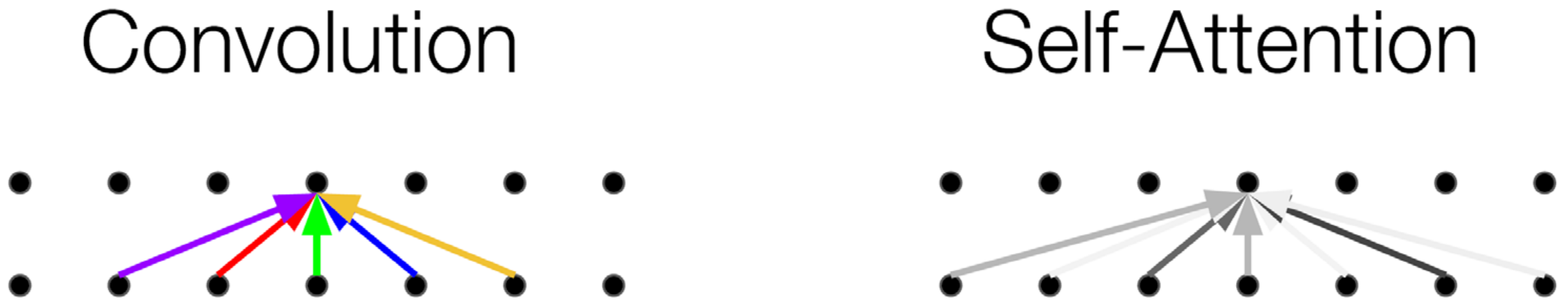
- Each word is a query to form attention over all tokens
- This generates a context-dependent representation of each token: a weighted sum of all tokens
- The attention weights dynamically mix how much is taken from each token
- Can run this process iteratively, at each step computing self-attention on the output of the previous level



Self-Attention

Information flows from within the same subnetwork (either encoder or decoder).

- Convolution applies fixed transform weights.
- Self-attention applies variable weights.



Self-attention

k : level number

X : input vectors

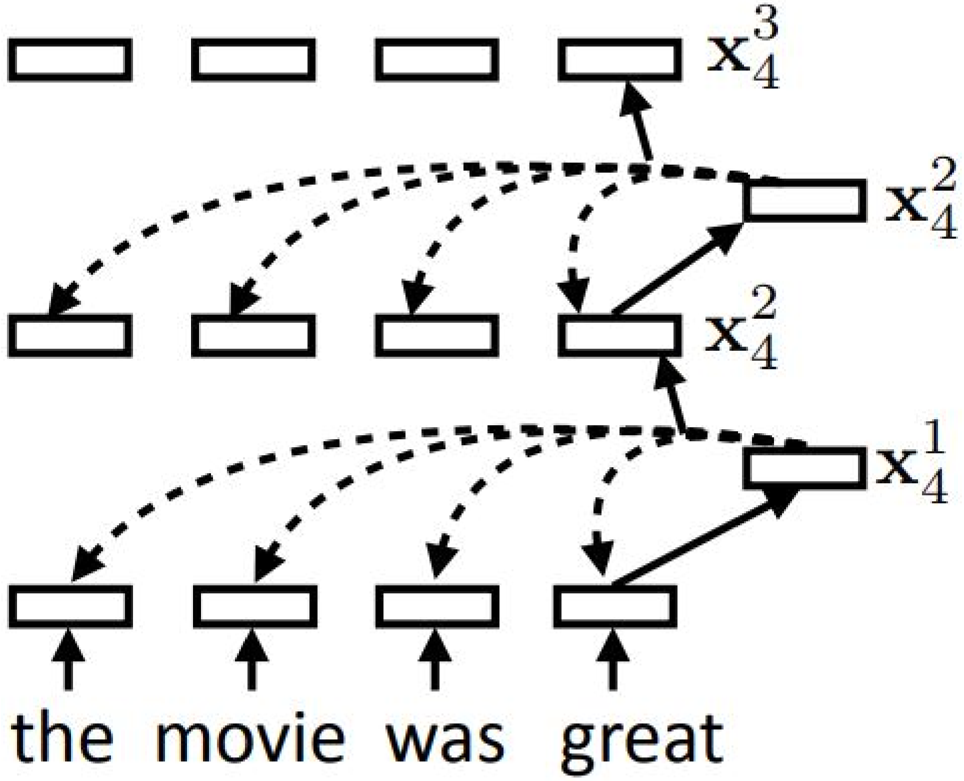
$X = \mathbf{x}_1, \dots, \mathbf{x}_n$

$\mathbf{x}_i^1 = \mathbf{x}_i$

$\bar{\alpha}_{i,j}^k = \mathbf{x}_i^{k-1} \cdot \mathbf{x}_j^{k-1}$

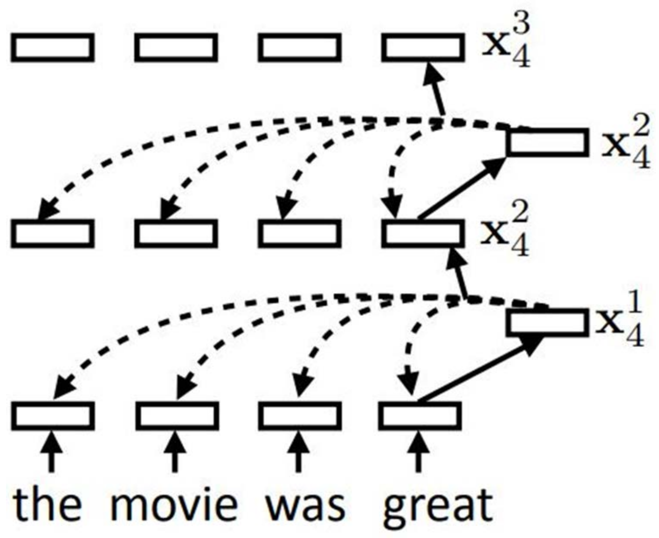
$\alpha_i^k = softmax(\bar{\alpha}_{i,1}^k, \dots, \bar{\alpha}_{i,n}^k)$

$$x_i^k = \sum_{i=1}^{n} \alpha_{i,j}^k \mathbf{x}_j^{k-1}$$



Multiple Attention Heads

- Multiple attention heads can learn to attend in different ways
- Requires additional parameters to compute different attention values and transform vectors



k : level number

L: number of heads

$X = \mathbf{x}_1, \dots, \mathbf{x}_n$

$\mathbf{x}_i^1 = \mathbf{x}_i$

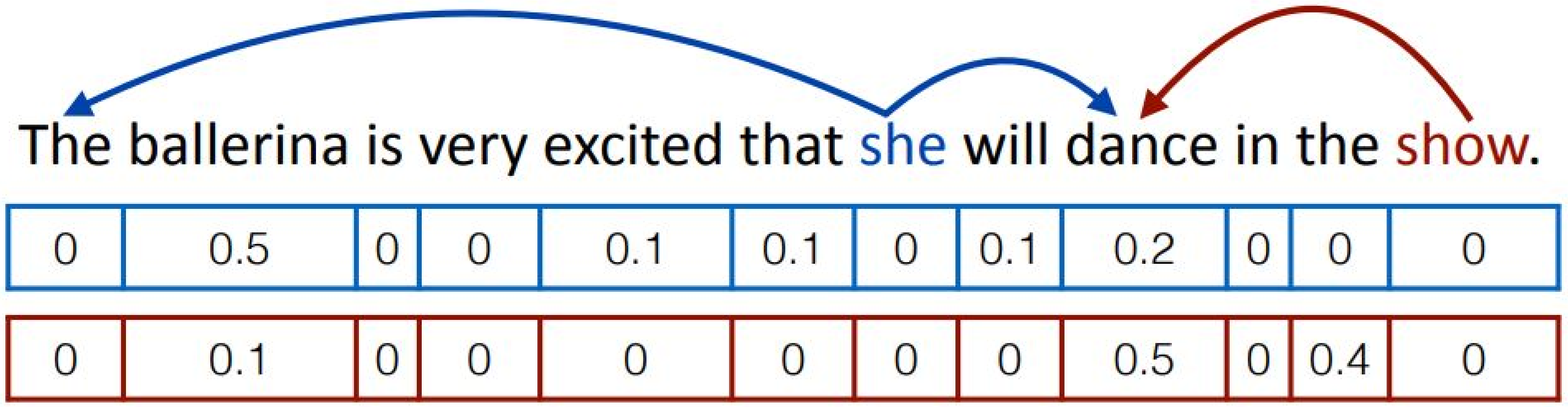
$$\bar{\alpha}_{i,j}^{k,l} = \mathbf{x}_i^{k-1} \mathbf{W}^{k,l}\, \mathbf{x}_j^{k-1}$$

$$\alpha_i^{k,l} = \text{softmax}(\bar{\alpha}_{i,1}^{k,l}, \dots, \bar{\alpha}_{i,n}^{k,l})$$

$$x_i^{k,l} = \sum_{i=1}^{n} \alpha_{i,j}^{k,l} \mathbf{x}_j^{k-1}$$

$$\mathbf{x}_i^k = V^k [\mathbf{x}_i^{k,1}; \dots; \mathbf{x}_i^{k,L}]$$

What Can Self-attention do?



- Attend to nearby related terms
- But just the same to far semantically related terms

Details

- This is the basic building block of an architecture called Transformers
- There are many details to get it to work, see Vaswani et al. 2017, later work, and available implementations
- Significant improvements for many tasks, including machine translation (Vaswani et al. 2017) and context-dependent pre-trained embeddings (BERT; Devlin et al. 2018)

Links

- Annotated Transformer

http://nlp.seas.harvard.edu/2018/04/03/attention.html

- Illustrated Transformer

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

Attention-only Translation Models

Problems with recurrent networks:

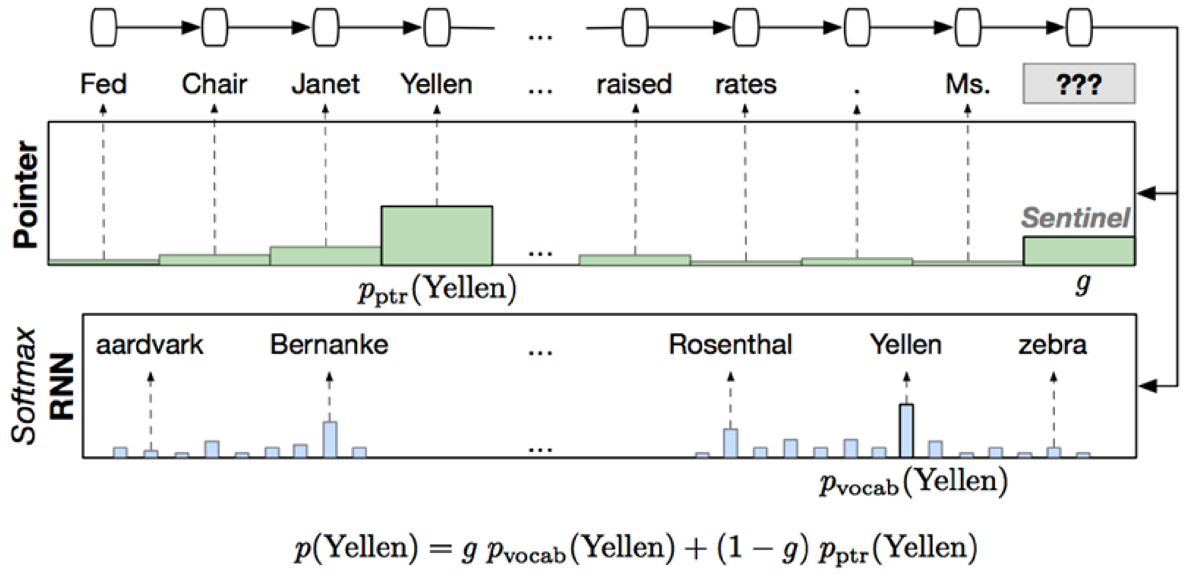
- Sequential training and inference: time grows in proportion to sentence length. Hard to parallelize.
- Long-range dependencies have to be remembered across many single time steps.
- Tricky to learn hierarchical structures (“car”, “blue car”, “into the blue car”…)

Alternative:

- Convolution – but has other limitations.

Attending to Previously Generated Things

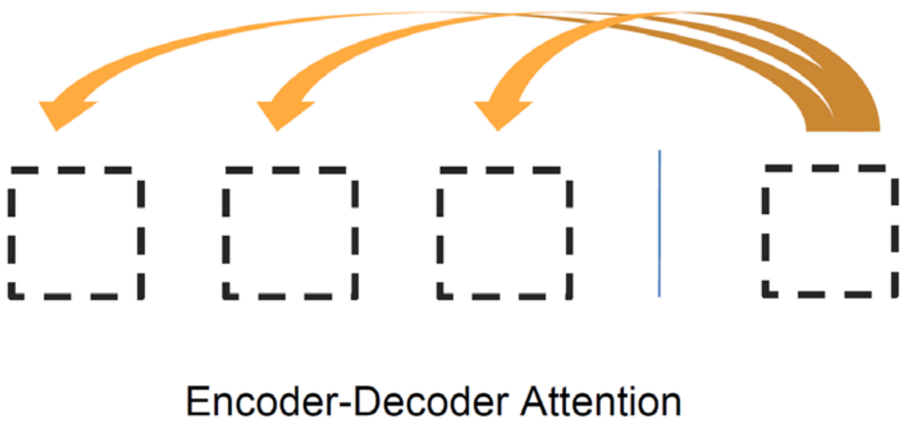
- In language modeling, attend to the previous words (Merity et al. 2016)

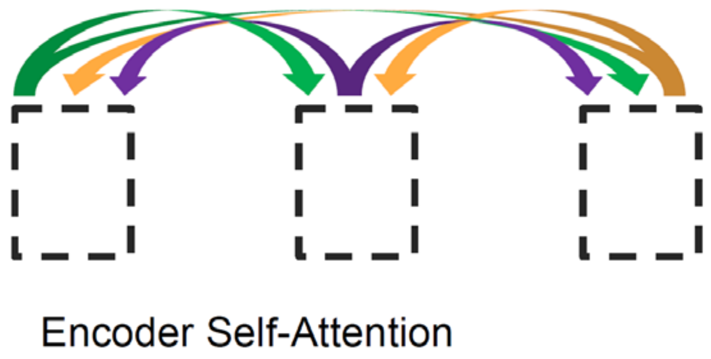


$$p(\text{Yellen}) = g\, p_{\text{vocab}}(\text{Yellen}) + (1-g)\, p_{\text{ptr}}(\text{Yellen})$$

- In translation, attend to either input or previous output (Vaswani et al. 2017)

Attention in Transformer Networks





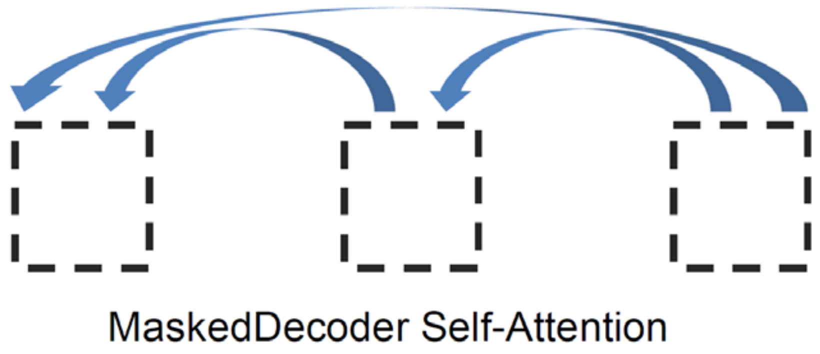
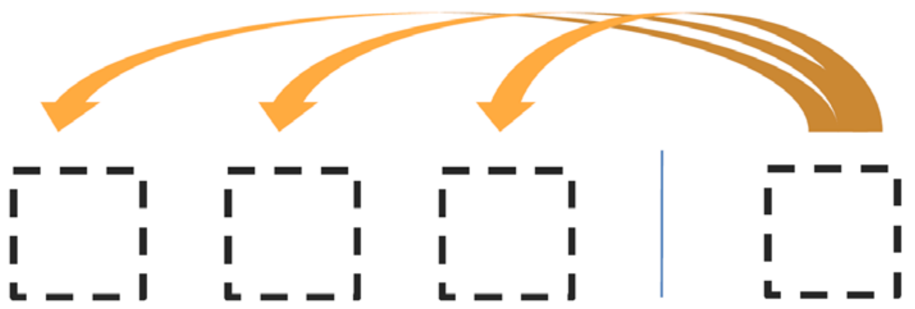


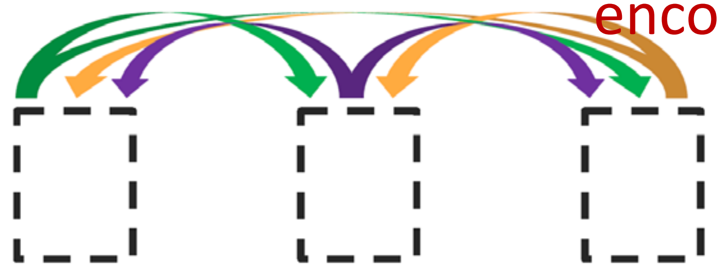
image from Lukas Kaiser, Stanford NLP seminar

Attention in Transformer Networks

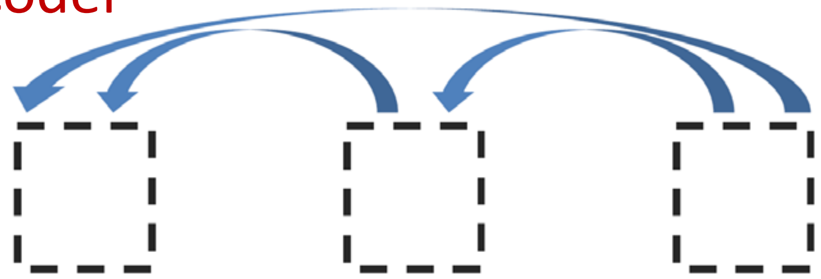


Encoder-Decoder Attention

Replaces word recurrence in encoder and decoder



Encoder Self-Attention



MaskedDecoder Self-Attention

Masking limits attention to earlier units: y_i depends only on y_j for $j < i$.

image from Lukas Kaiser, Stanford NLP seminar

CS60075
Natural Language Processing
Autumn 2020

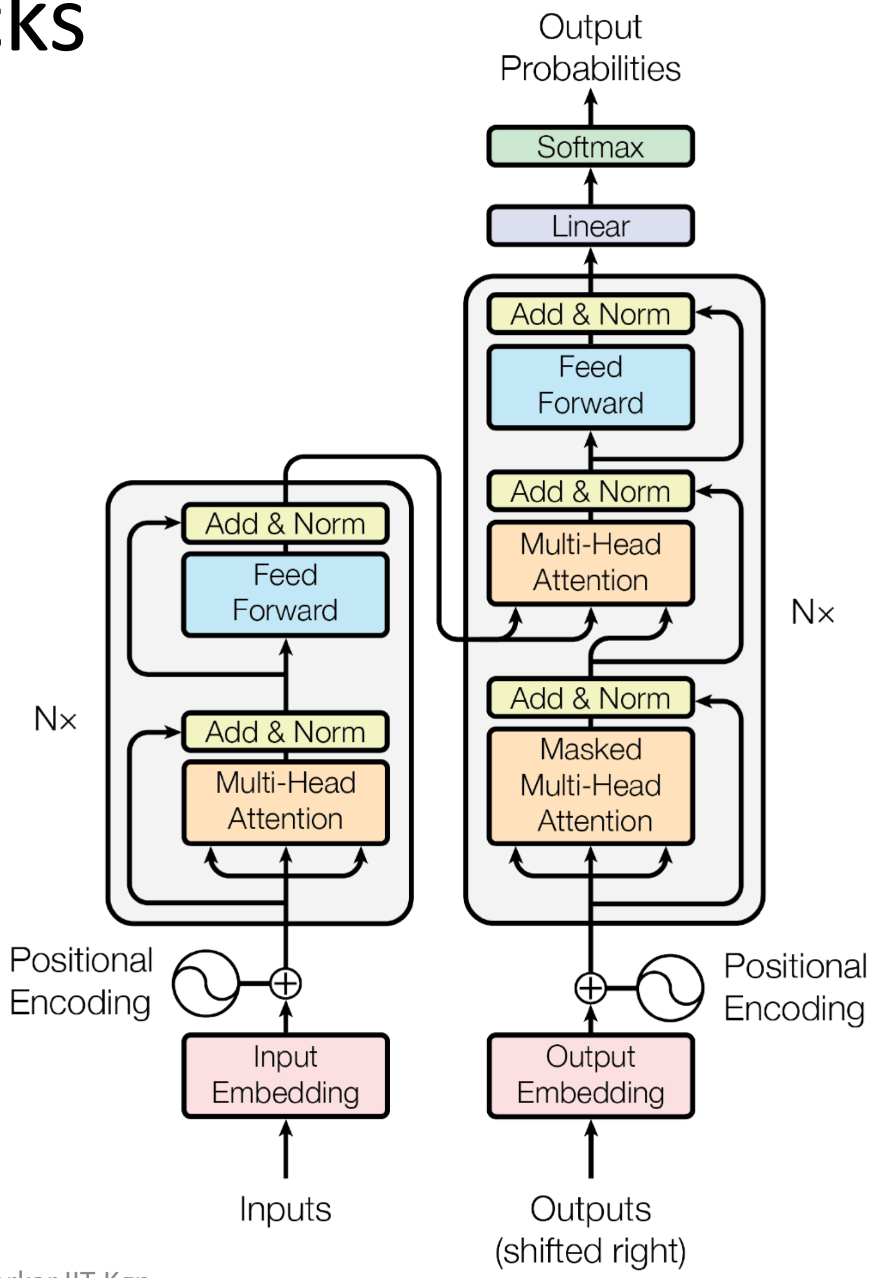
Module 8:

Transformers – Part 2

3 November 2020

The Transformer Attention Tricks

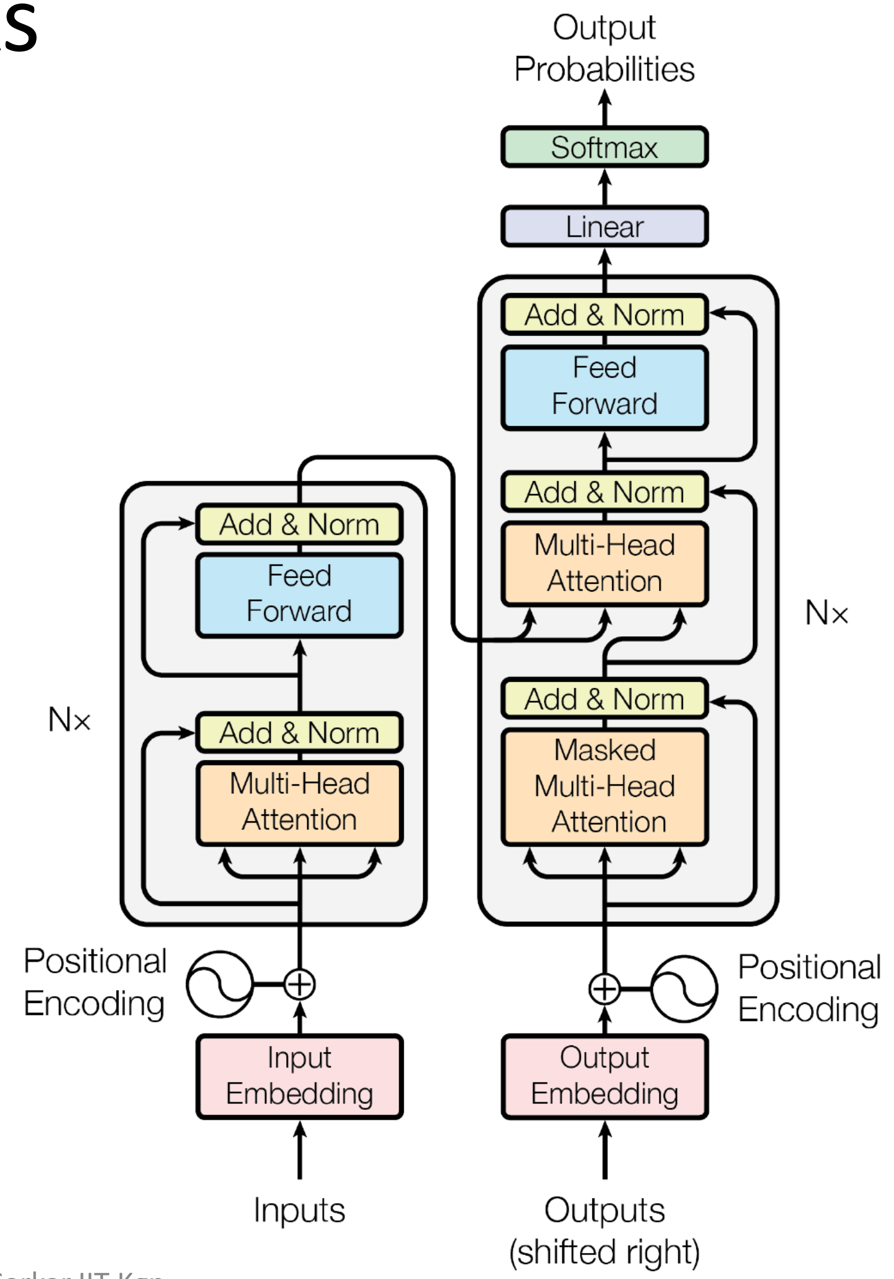
- **Self Attention:** Each layer combines words with others
- **Multi-headed Attention:** 8 attention heads function independently
- **Normalized Dot-product Attention:** Remove bias in dot product when using large networks
- **Positional Encodings:** Make sure that even if we don't have RNN, can still distinguish positions



The Transformer Training Tricks

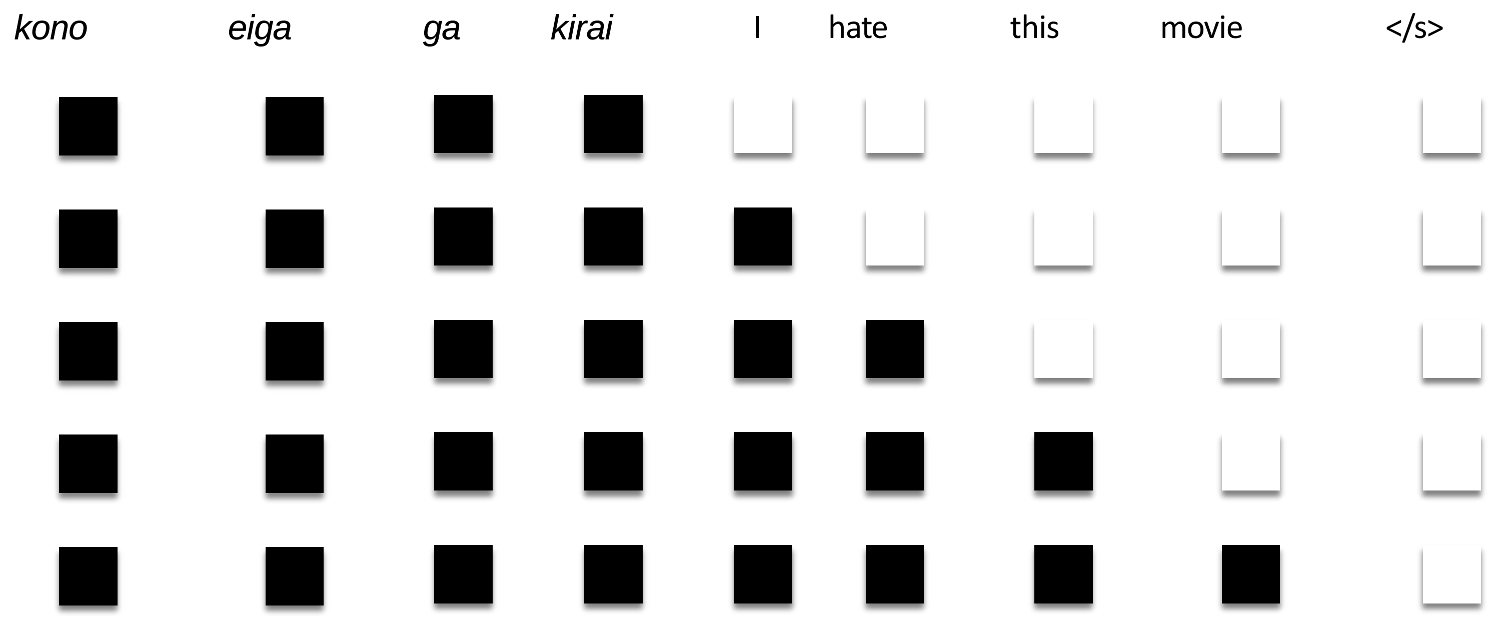
- **Layer Normalization:** Help ensure that layers remain in reasonable range
- **Specialized Training Schedule:** Adjust default learning rate of the Adam optimizer
- **Label Smoothing:** Insert some uncertainty in the training process

  Masking for Efficient Training



Masking for Training

- We want to perform training in as few operations as possible using big matrix multiplies
- We can do so by “masking” the results for the output



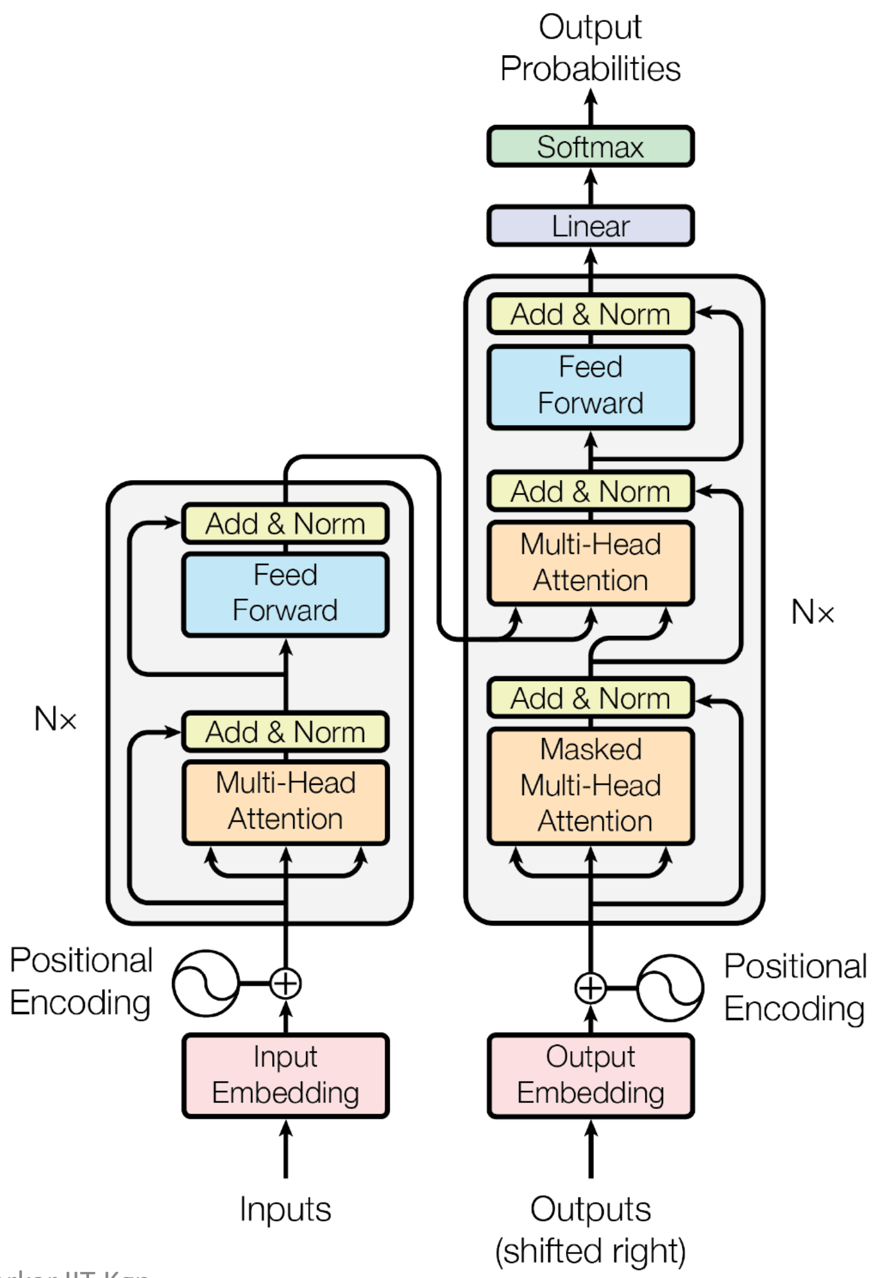
The Transformer

- In experiments, stacked with N=6.

- Output words fed back as input, shifted right. Can use beam search as before.

- Inputs and outputs are embedded in vector spaces of fixed dimension.

- Positional encoding: when words are combined through attention, their location is lost. Positional encoding adds it back.



Attention Implementation

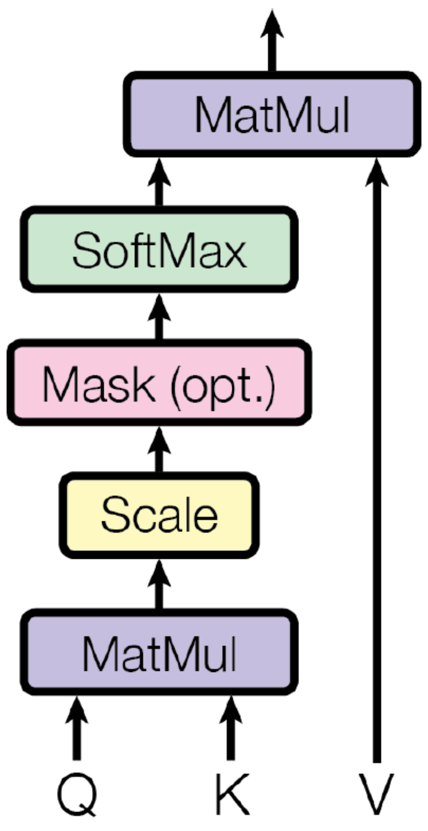
- Attention is modeled as a key-value store:
  Q = query vector
  K = key
  V = value

Encoder-decoder layer: the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. (Similar to Bahdanau).

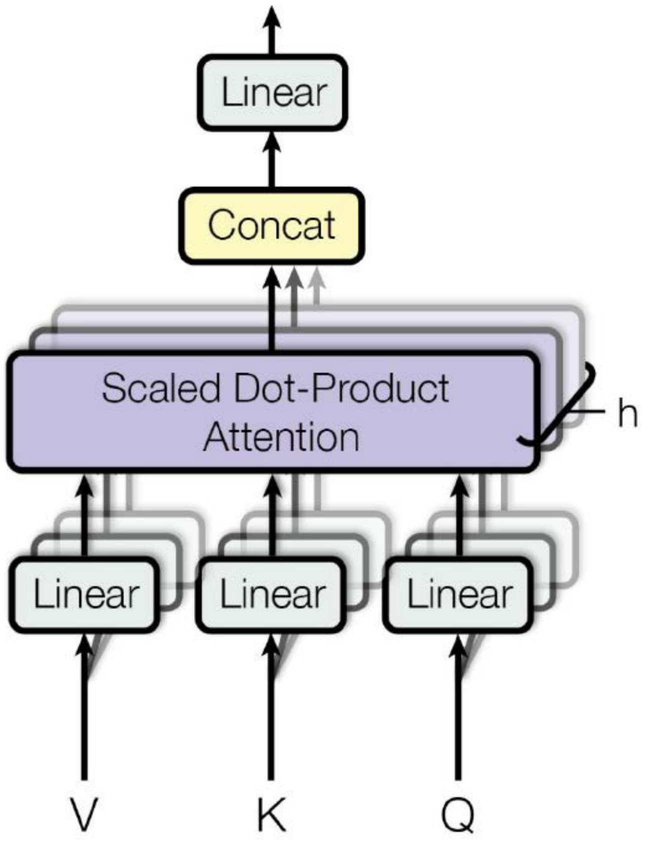
Self-attention layer: all of the keys, values and queries come from the output of the previous layer in the encoder.

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

Scaled Dot-Product Attention



Multi-Headed Attention



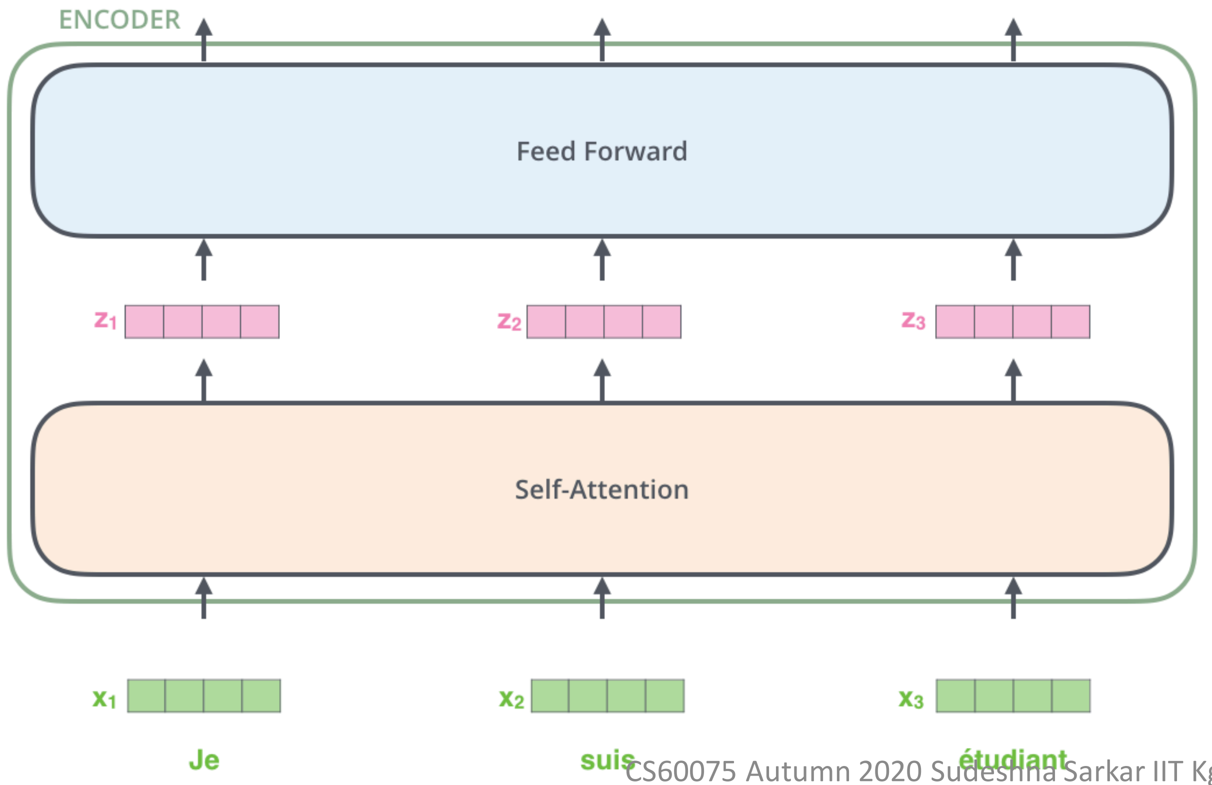
The Illustrated Transformer

- The following slides are based on material from

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

Self-attention

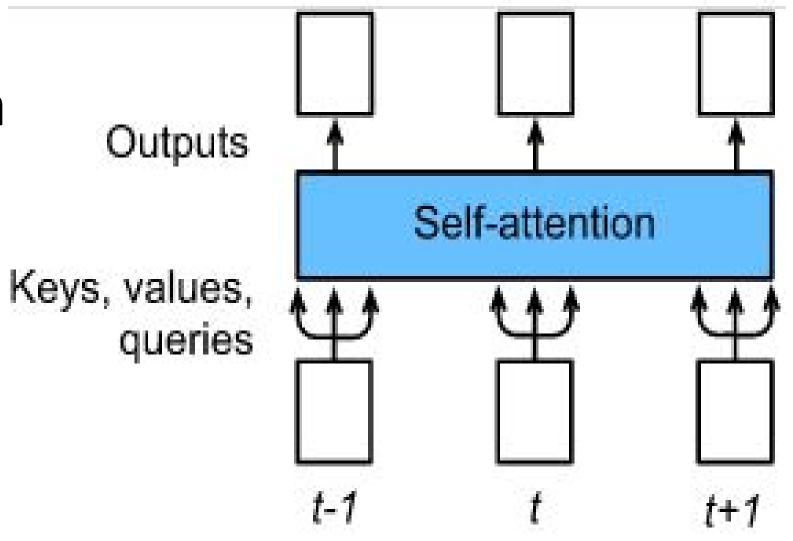
- An attention mechanism relating different positions of a single sequence in order to compute a representation of the same sequence
- Information flows from within the same subnetwork (either encoder or decoder).



Self-Attention Transformers

"I arrived at the bank after crossing the river":

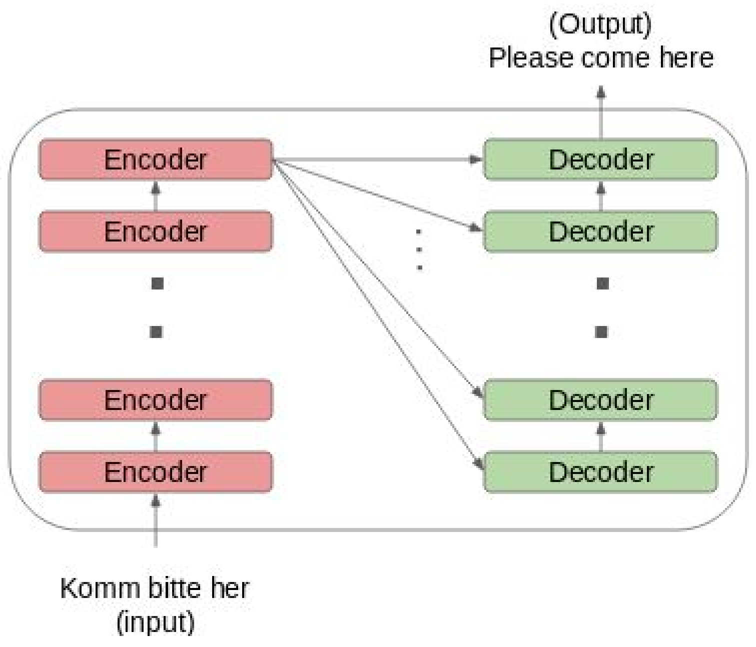
- **T**o determine that the word "bank" refers to the shore of a river, the Transformer can learn to immediately attend to the word "river".
- To compute the next representation for a given word - "bank"-
  1. the Transformer compares it to every other word in the sentence.
  2. Compute an attention score for every other word in the sentence.
  3. The attention scores determine how much each of the other words should contribute to the next representation of "bank"

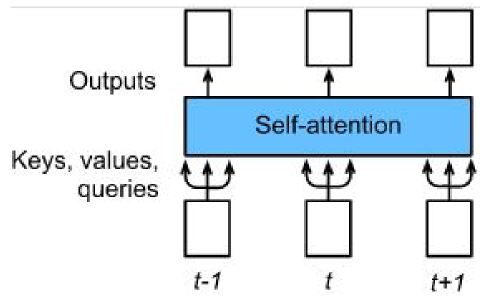


Vaswani et al. "Attention is all you need", arXiv 2017

Transformers

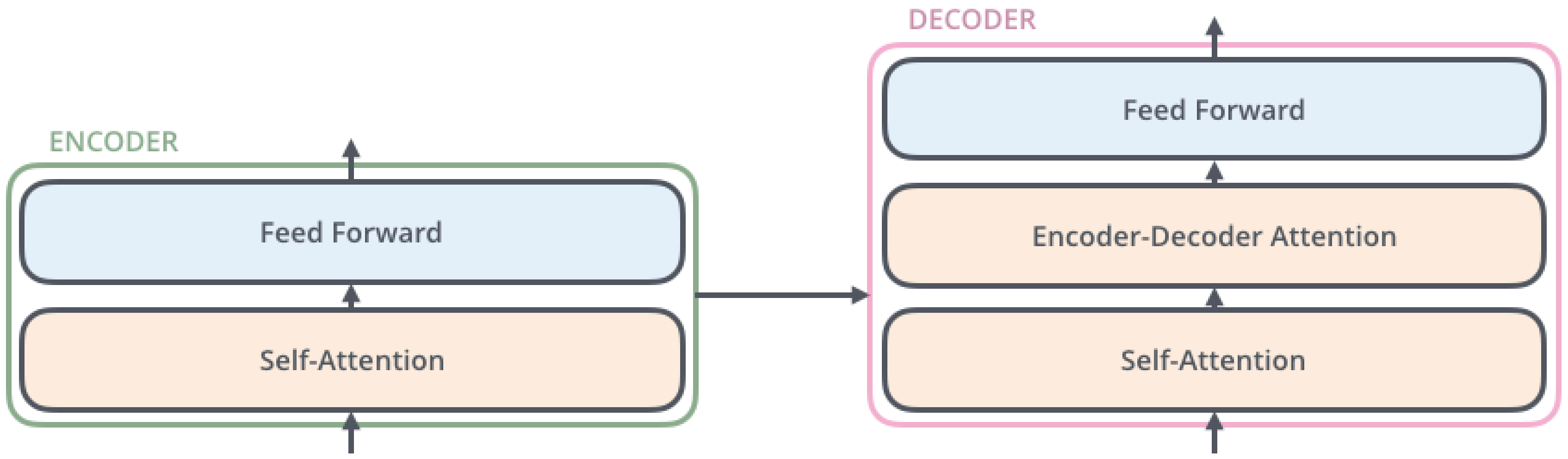
- The Transformer starts by generating initial representations for each word.
- using self-attention, it aggregates information from all of the other words, generating a new representation per word
- This step is repeated multiple times in parallel for all words, successively generating new representations.





Attention in Transformer

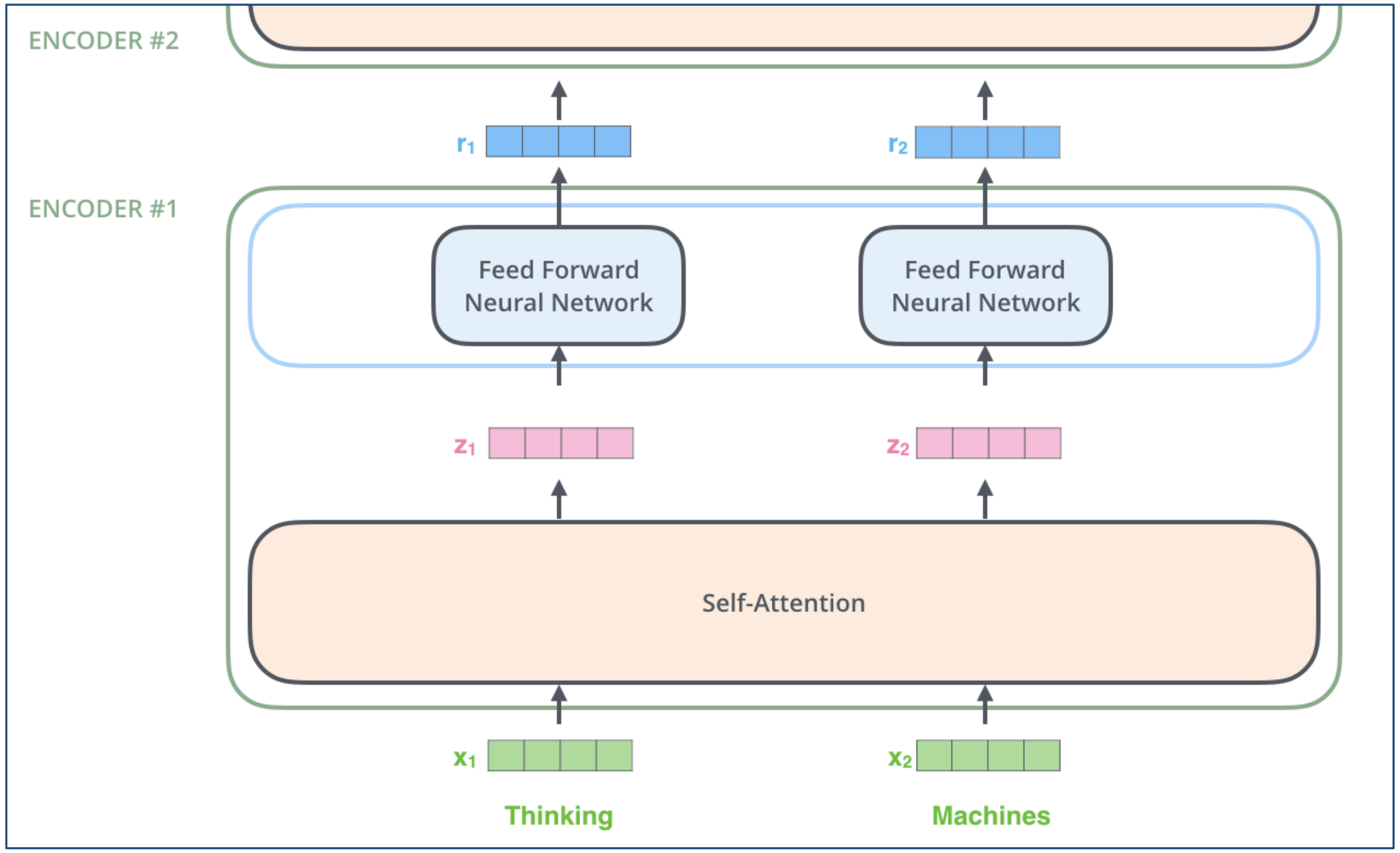
- The encoder's inputs first flow through a self-attention layer
- The outputs are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.
- The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence



Encoder

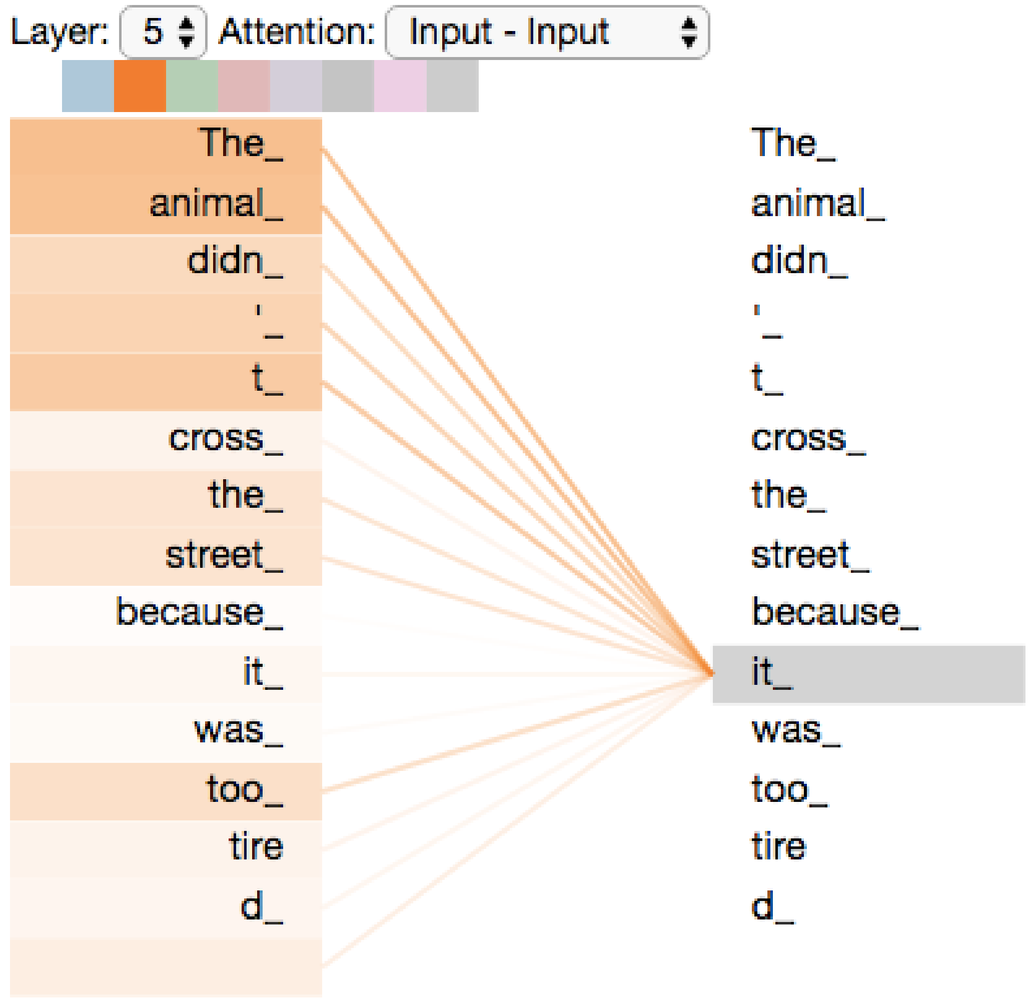
The word at each position passes through a self-attention process.

Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.



Self-attention

Self-attention is the method the Transformer uses to take the "understanding" of other relevant words into the one we're currently processing.



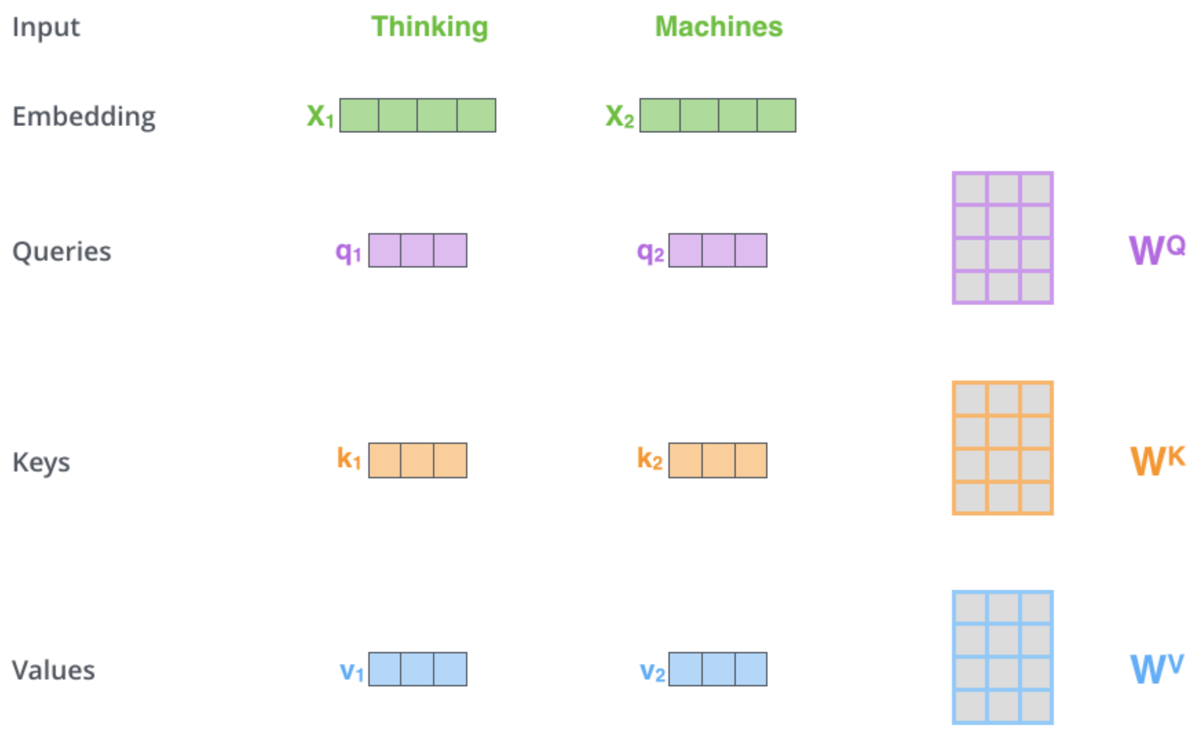
Self-Attention in Detail:

Step 1 : Create Vectors

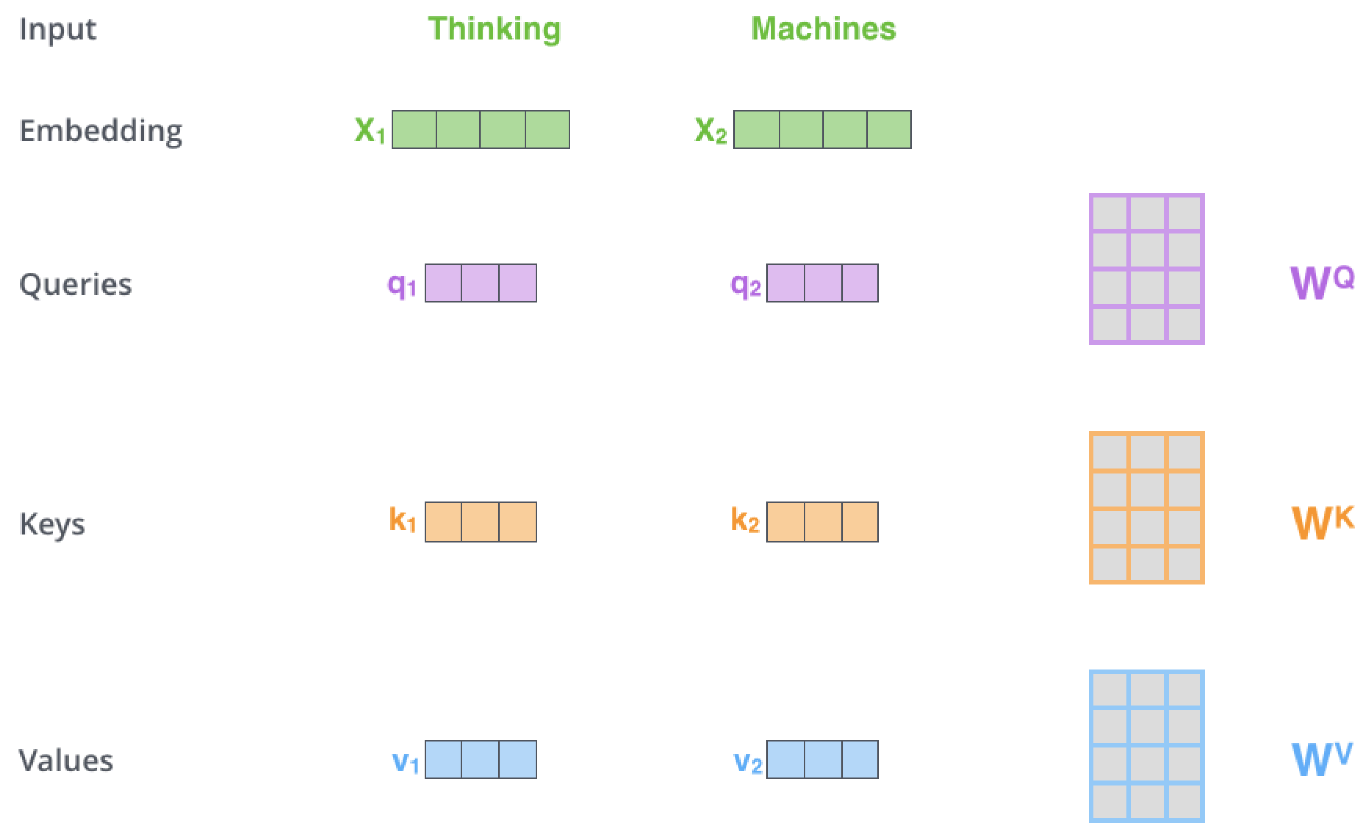
Step 1: create three vectors from each of the encoder's input vectors

1. Query vector
2. Key vector
3. Value vector.

- These vectors are created by multiplying the embedding by three matrices that we trained during the training process



Step 1 : Create three vectors vectors from each of the encoder's input vectors.

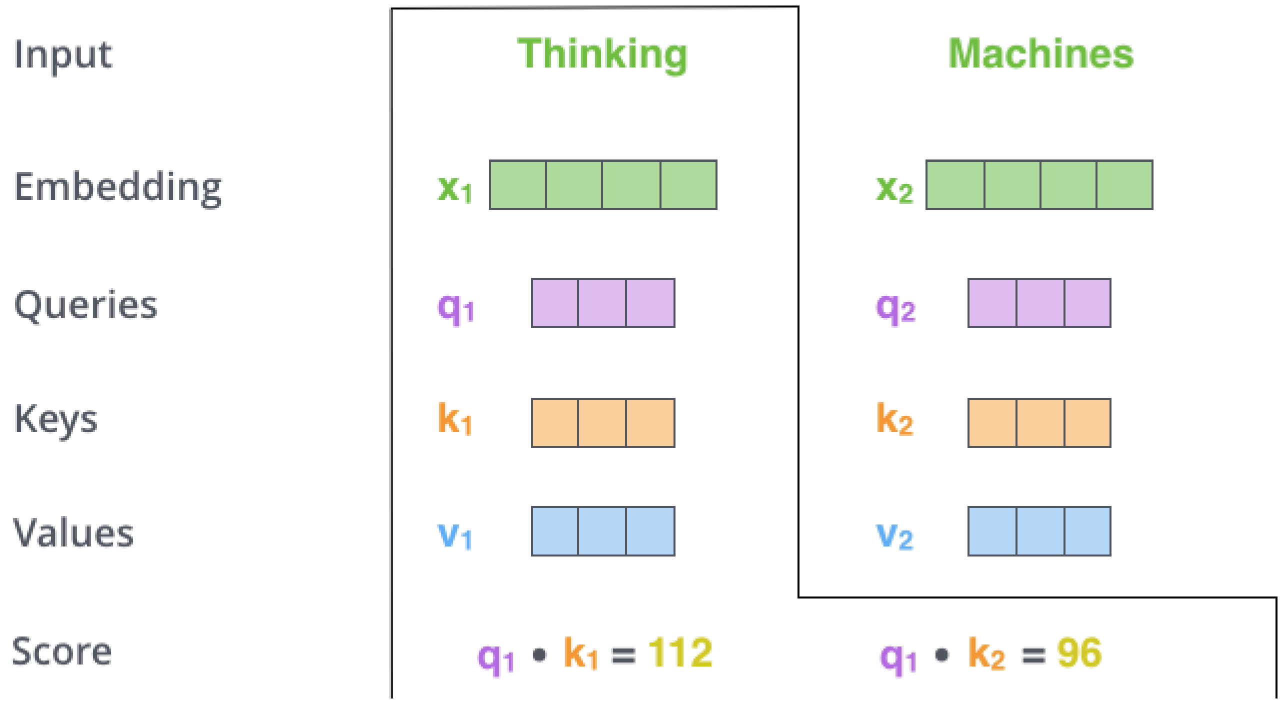


They're abstractions that are useful for calculating and thinking about attention.

Step 2: Calculate score

Say we're calculating the self-attention for the first word "Thinking".

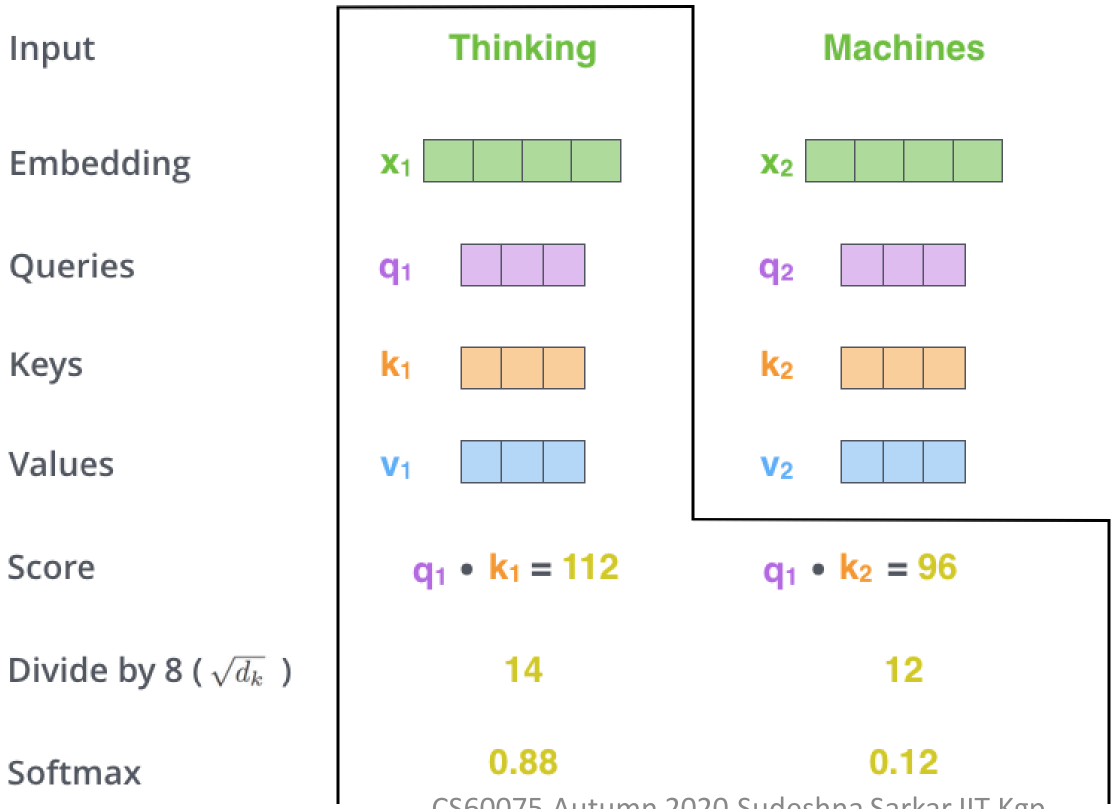
The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.



Steps 3-4

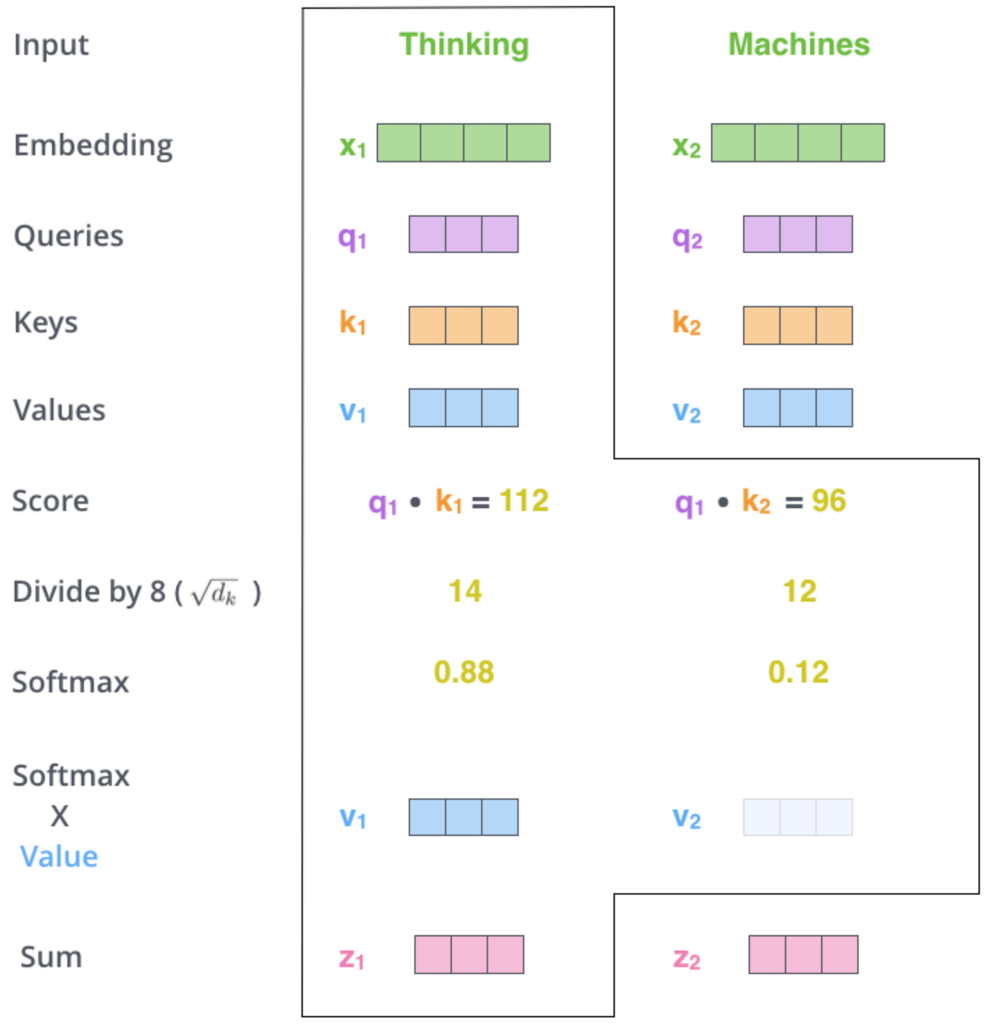
3. Divide the scores by 8 (the square root of the dimension of the key vectors– 64. This leads to having more stable gradients

4. Softmax : This softmax score determines how much how much each word will be expressed at this position.



Step 5

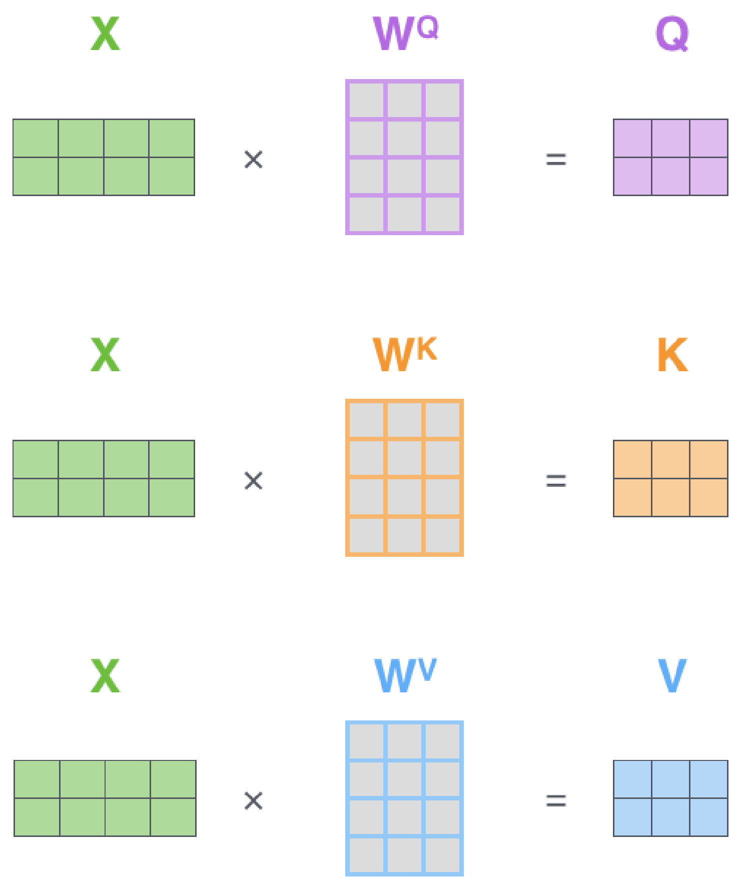
Multiply each value vector by the softmax score



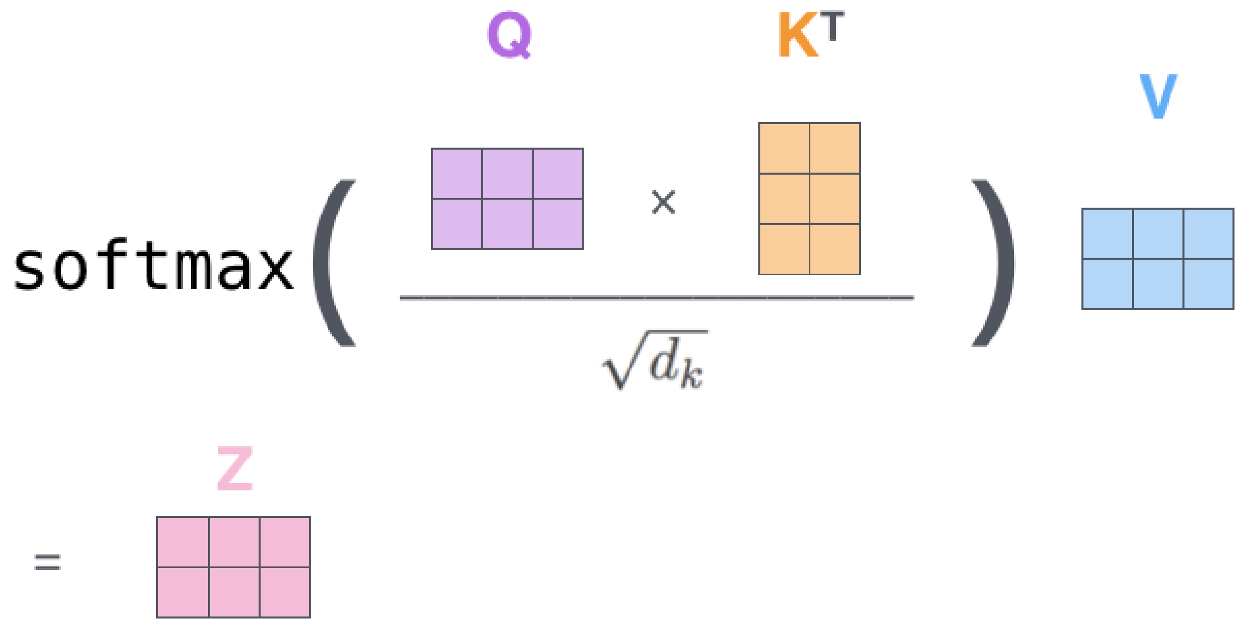
Matrix Calculation of Self-Attention

Calculate the Query, Key, and Value matrices.

Multiply input X by the weight matrices (WQ, WK, WV).



the outputs of the self-attention layer.



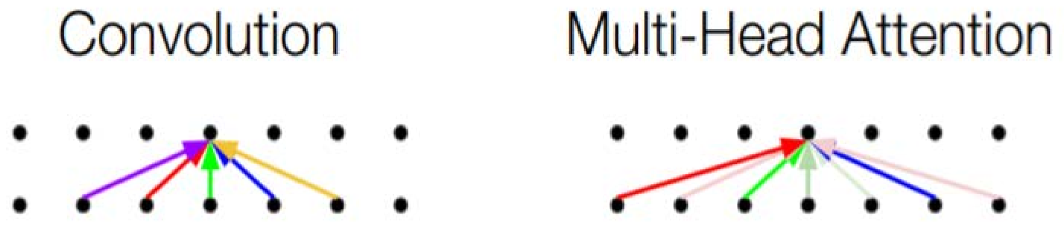
Scaling factor

Cause: Dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

Solution: To counteract this effect, scale the dot products by $\frac{1}{\sqrt{d_k}}$

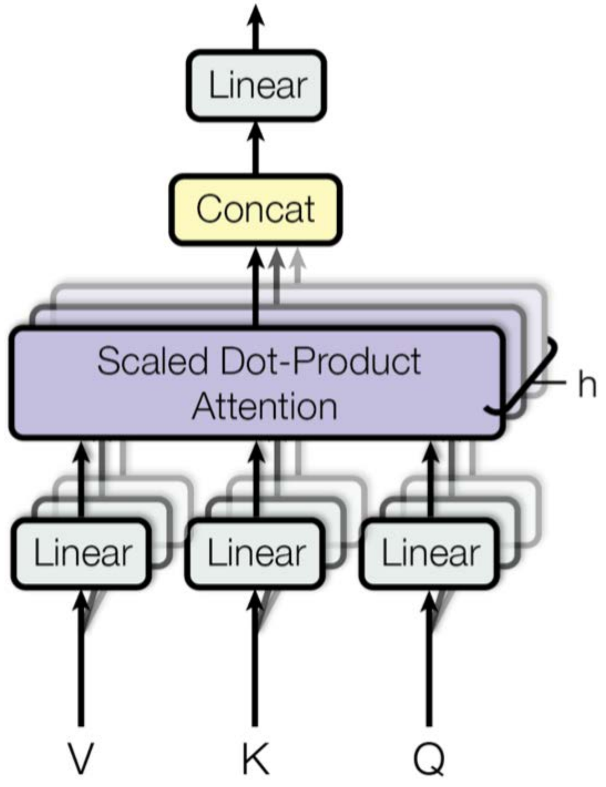
Multi-head Attention

- Multiple attention layers (heads) in parallel (shown by different colors)
- Each head uses different linear transformations.
- Different heads can learn different relationships.



Multi-headed attention

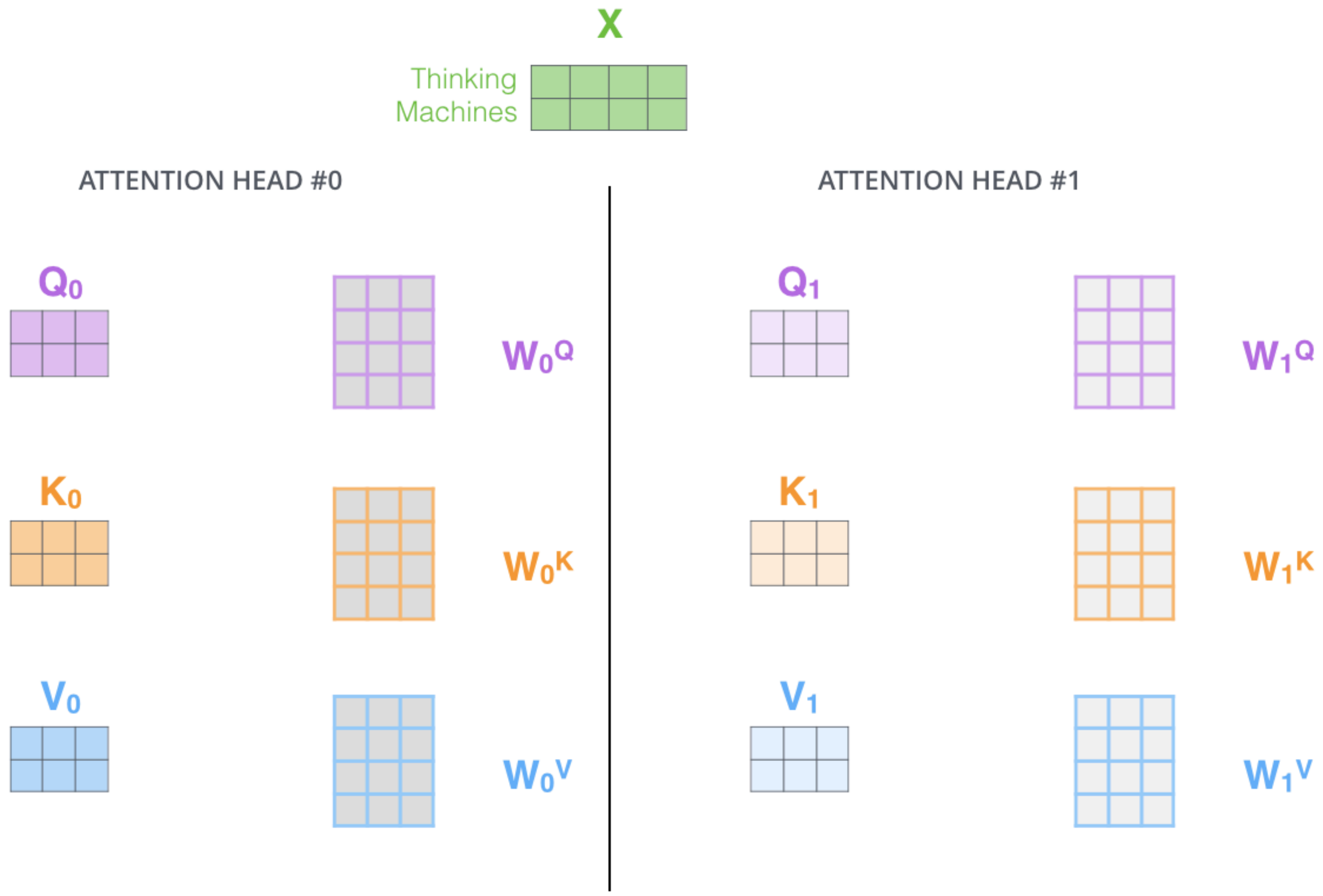
- Use multiple sets of Query/Key/Value weight matrices
- The Transformer uses eight attention heads
  - so there are eight sets for each encoder/decoder
  - Each of these sets is randomly initialized.
- After training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

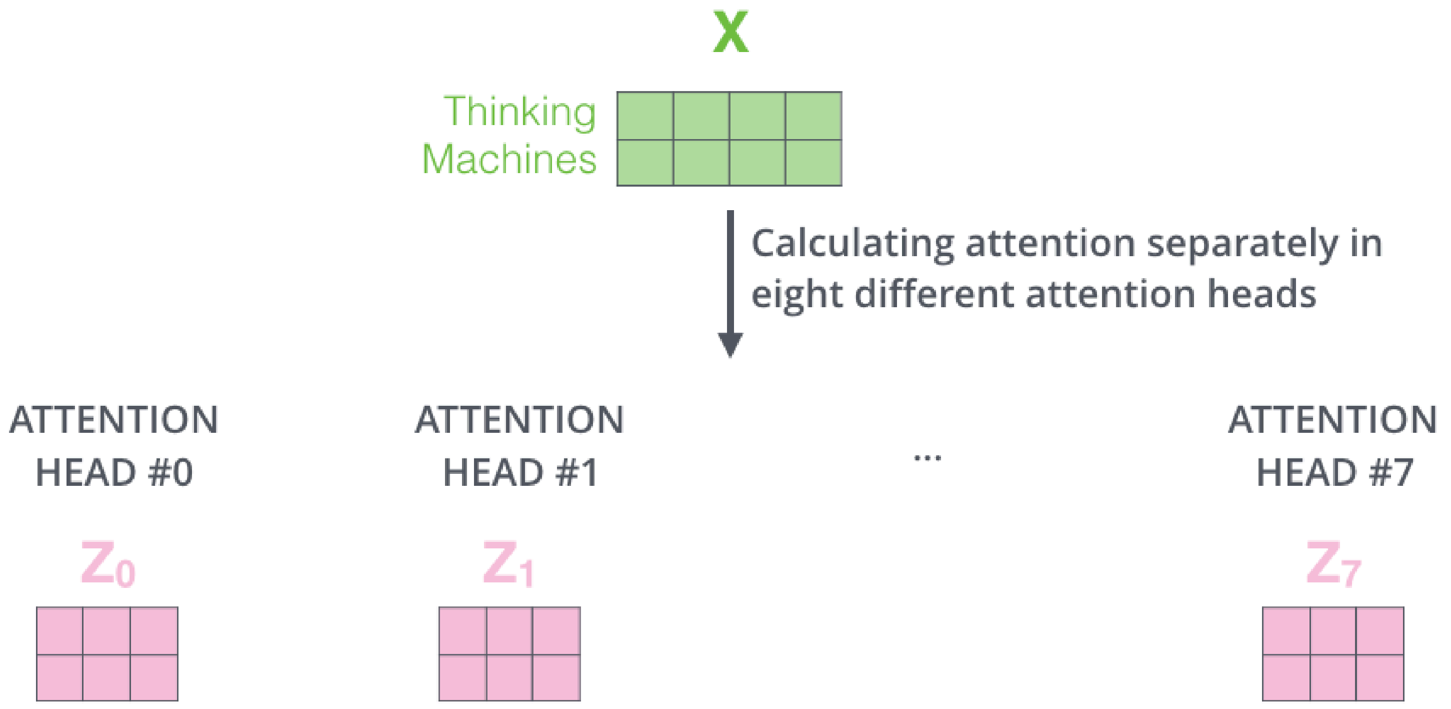


$$\mathrm{MultiHead}\,(Q,K,V) = [head_1;\ldots;head_h]W^O$$

where $\mathrm{head}_i = \mathrm{Attention}(QW^Q{}_i, KW^K{}_i, VW^V{}_i)$

With multi-headed attention, maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices.



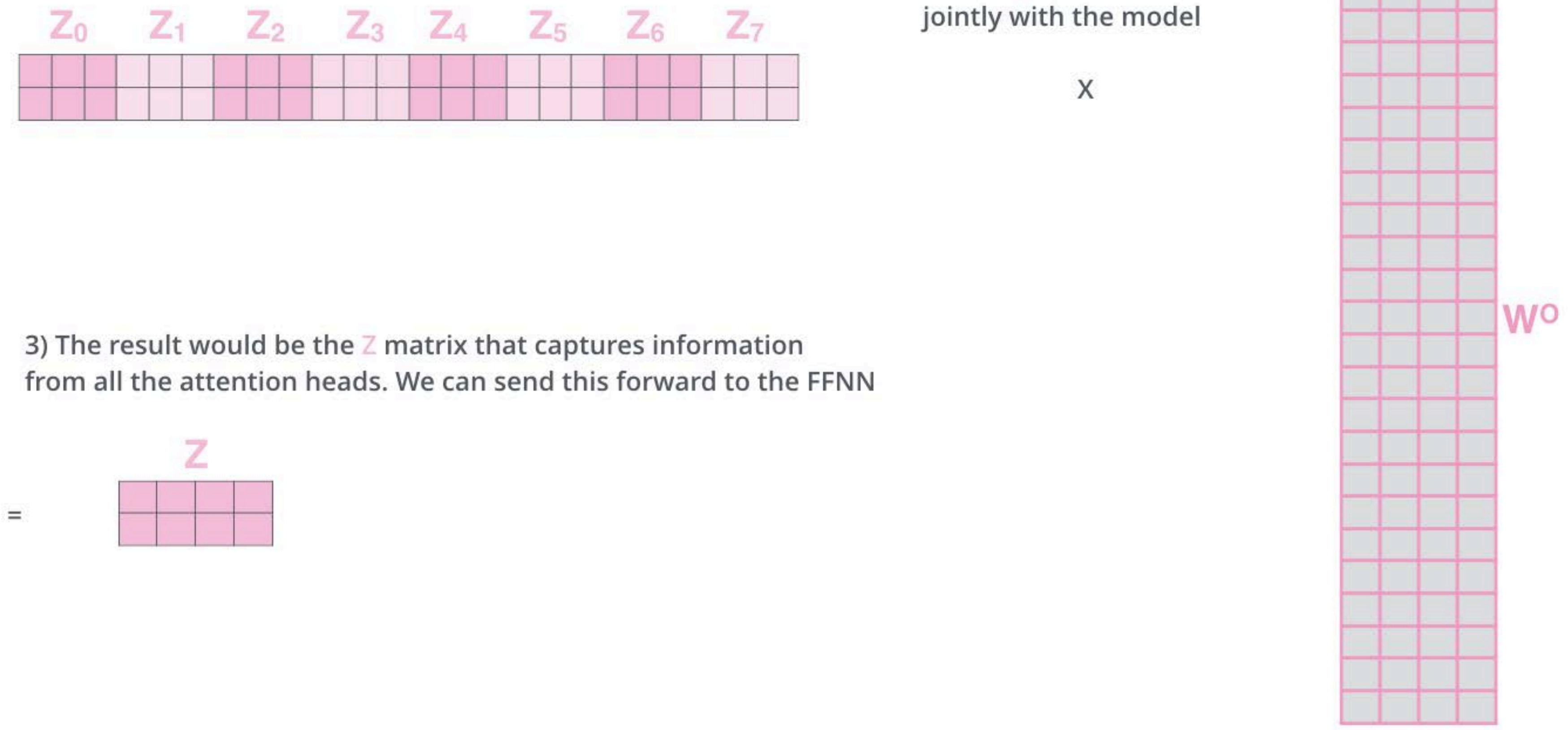


we need a way to condense these eight down into a single matrix as input to the feedforward network.

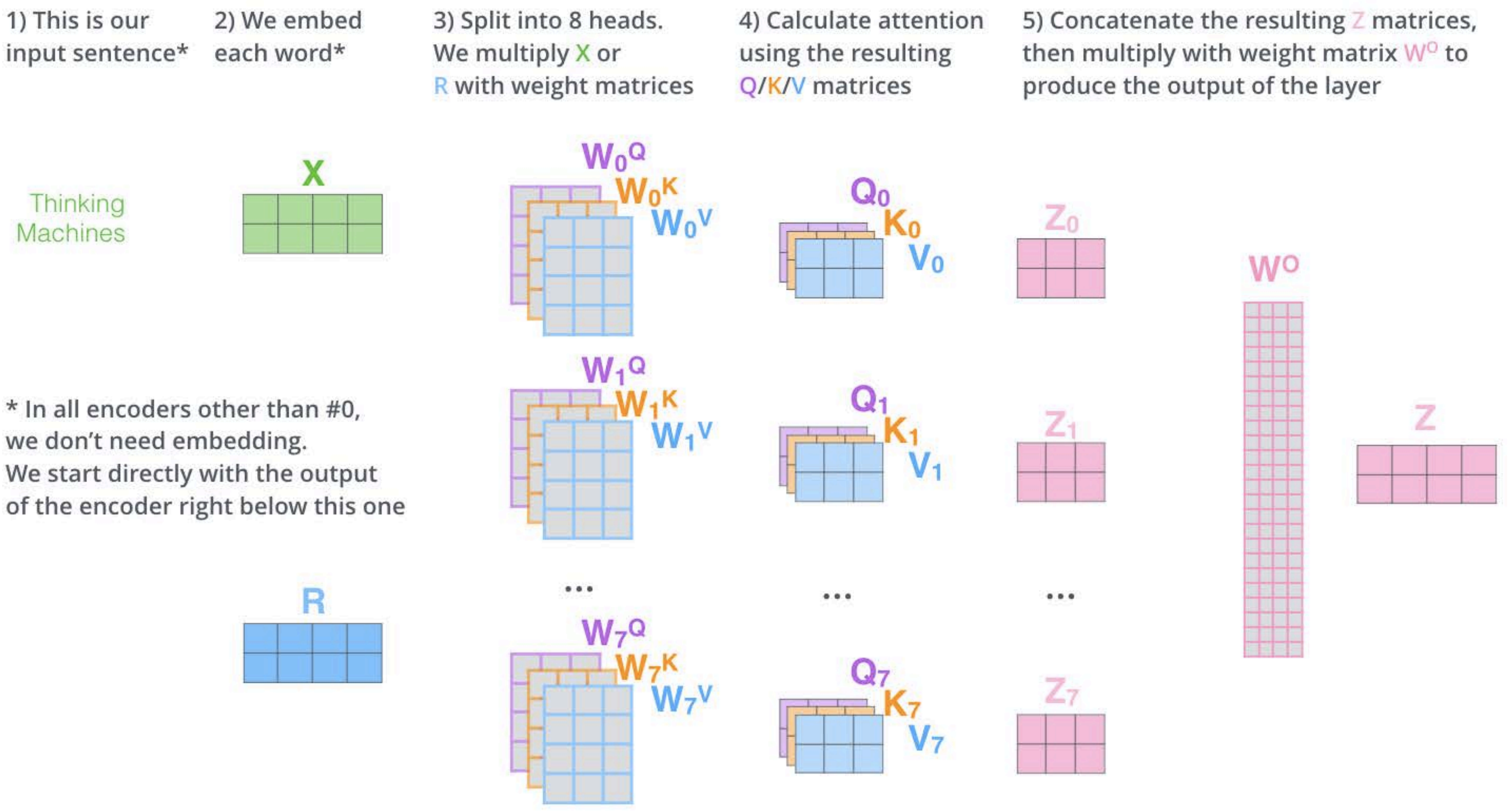
1) Concatenate all the attention heads

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

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



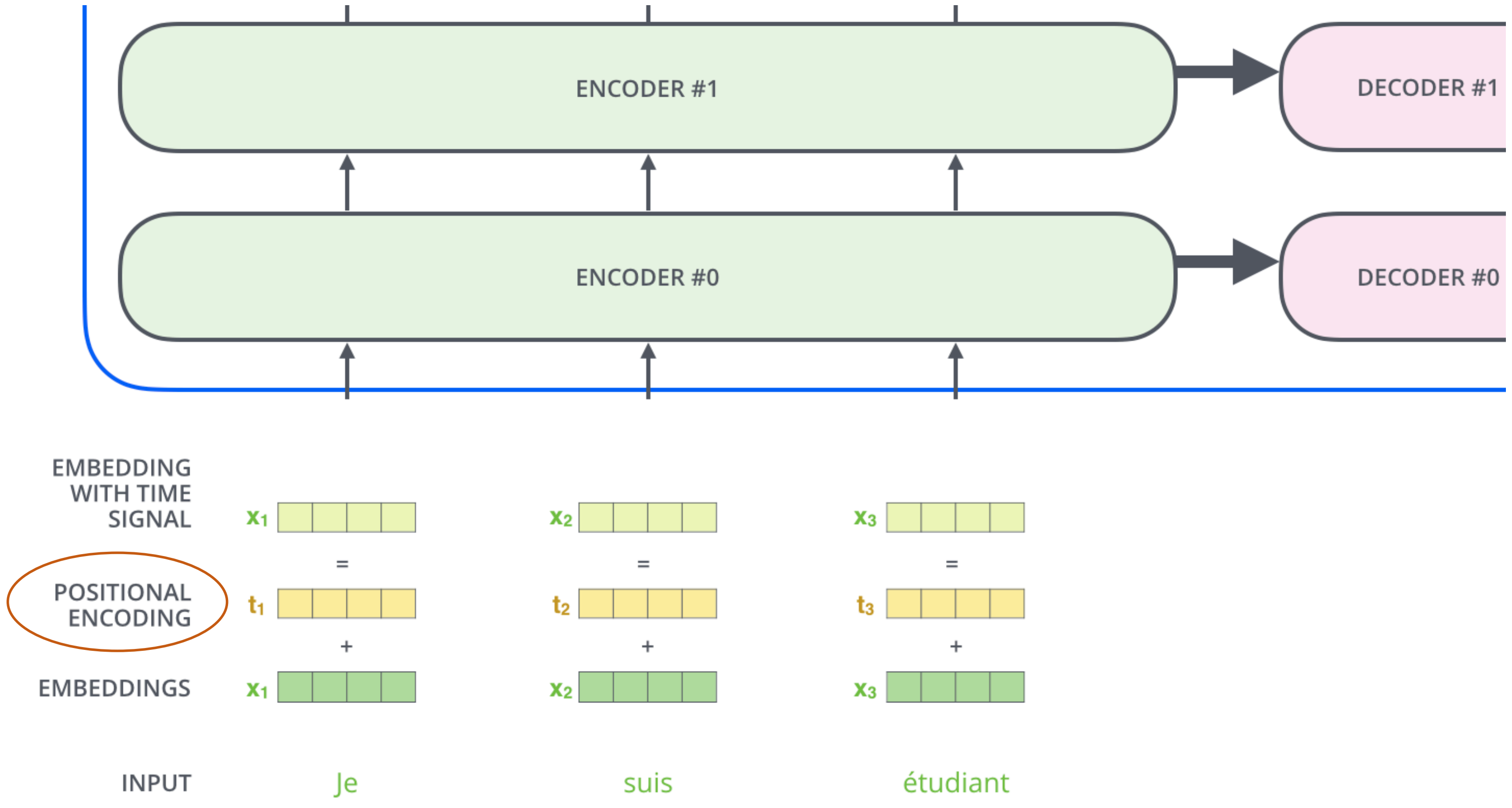
Putting it all together



1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
R
W_0^Q
W_0^K
W_0^V
W_1^Q
W_1^K
W_1^V
W_7^Q
W_7^K
W_7^V
Q_0
K_0
V_0
Q_1
K_1
V_1
Q_7
K_7
V_7
Z_0
Z_1
Z_7
W^O
Z
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

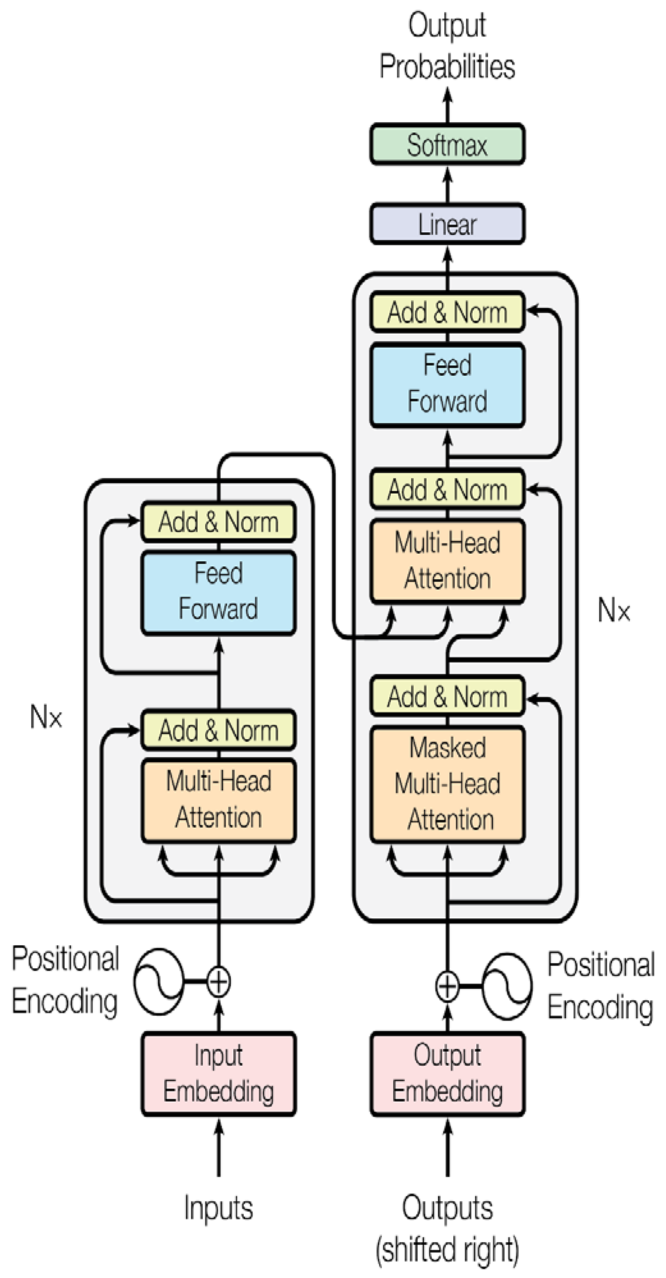
Representing The Order of The Sequence Using Positional Encoding

- Add a vector to each input embedding.
- These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence.
- The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors



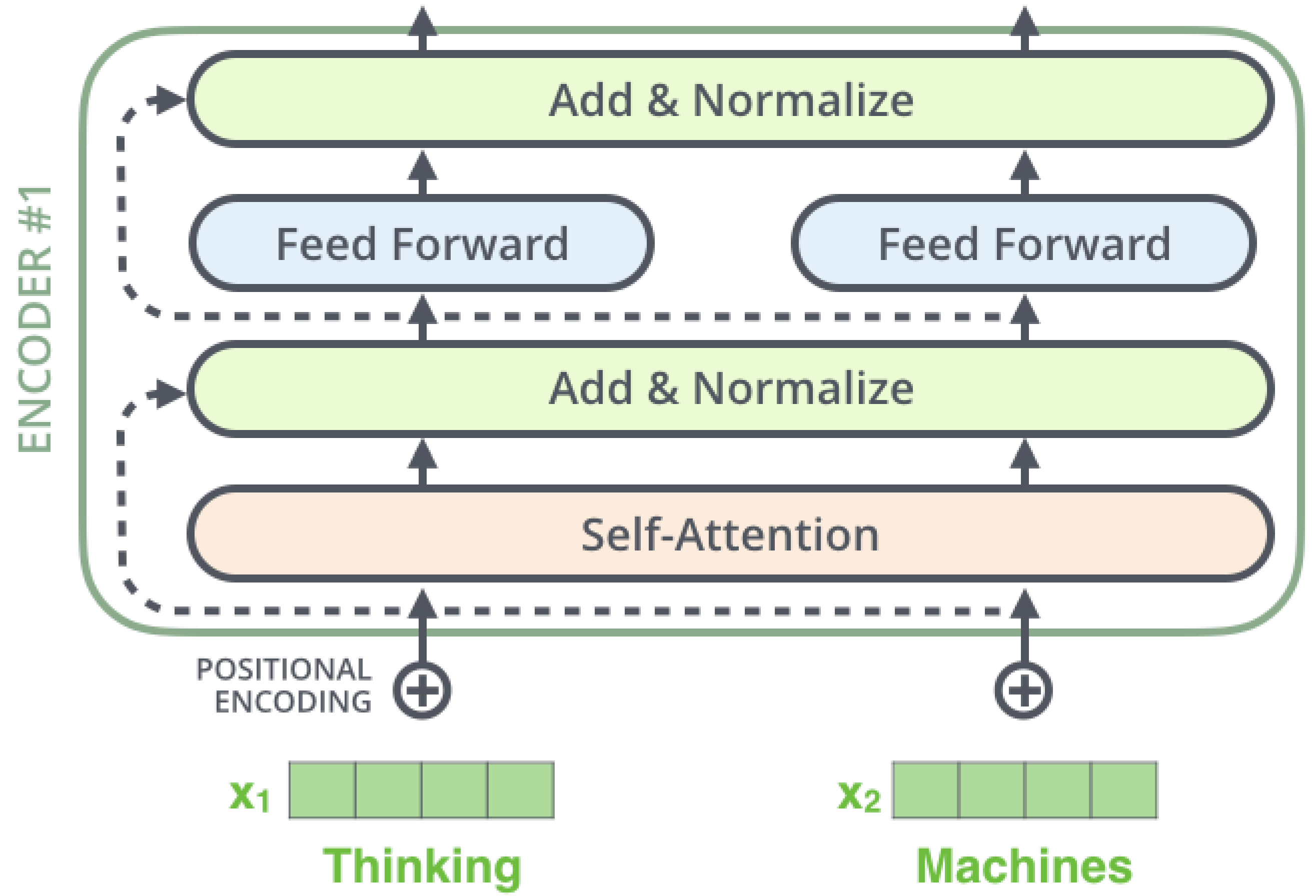
The Transformer

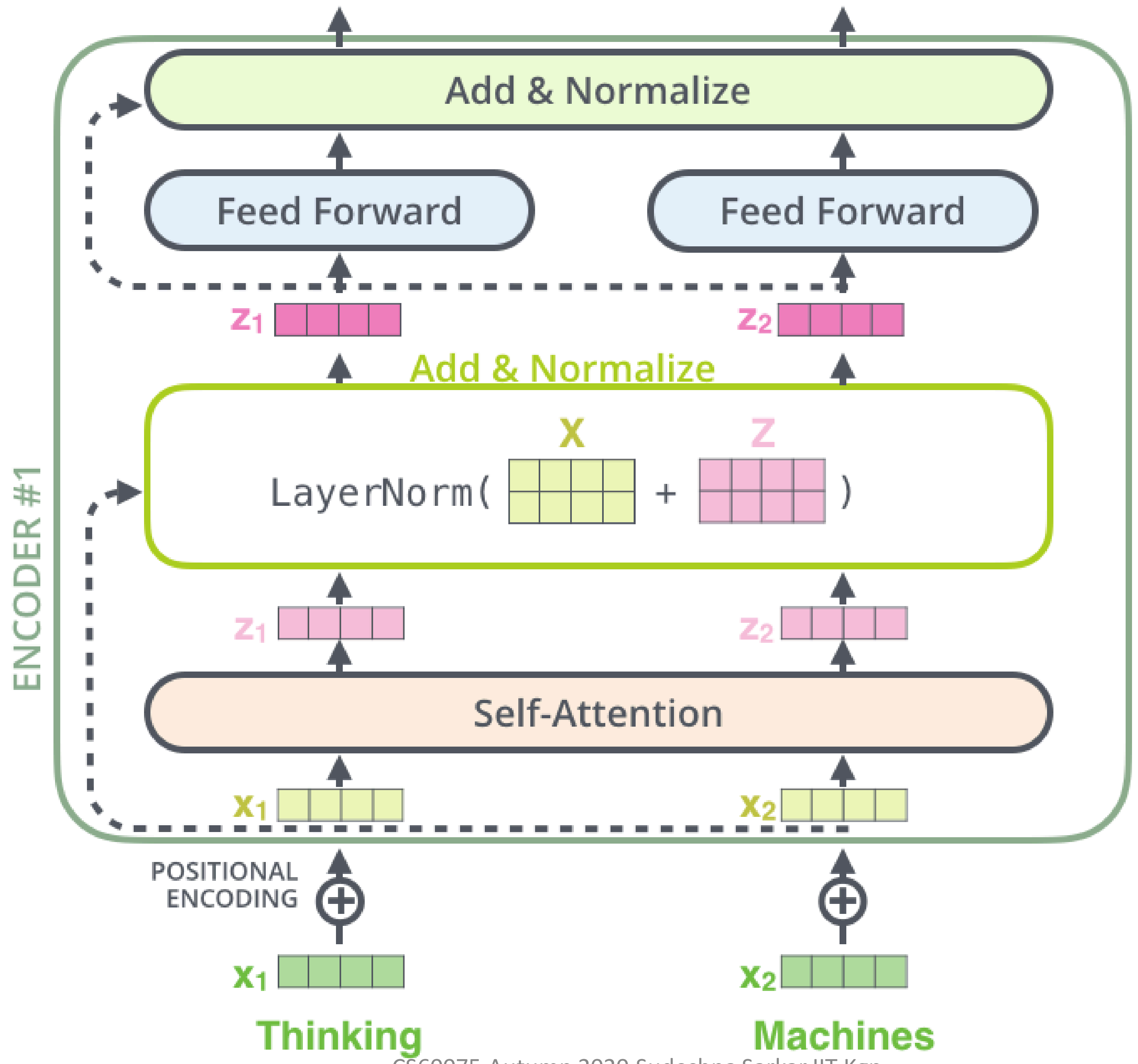
- Encoder and Decoder Stacks
    - Attention
    - Position-wise FF Networks
    - Positional Encoding
    - Add & Norm



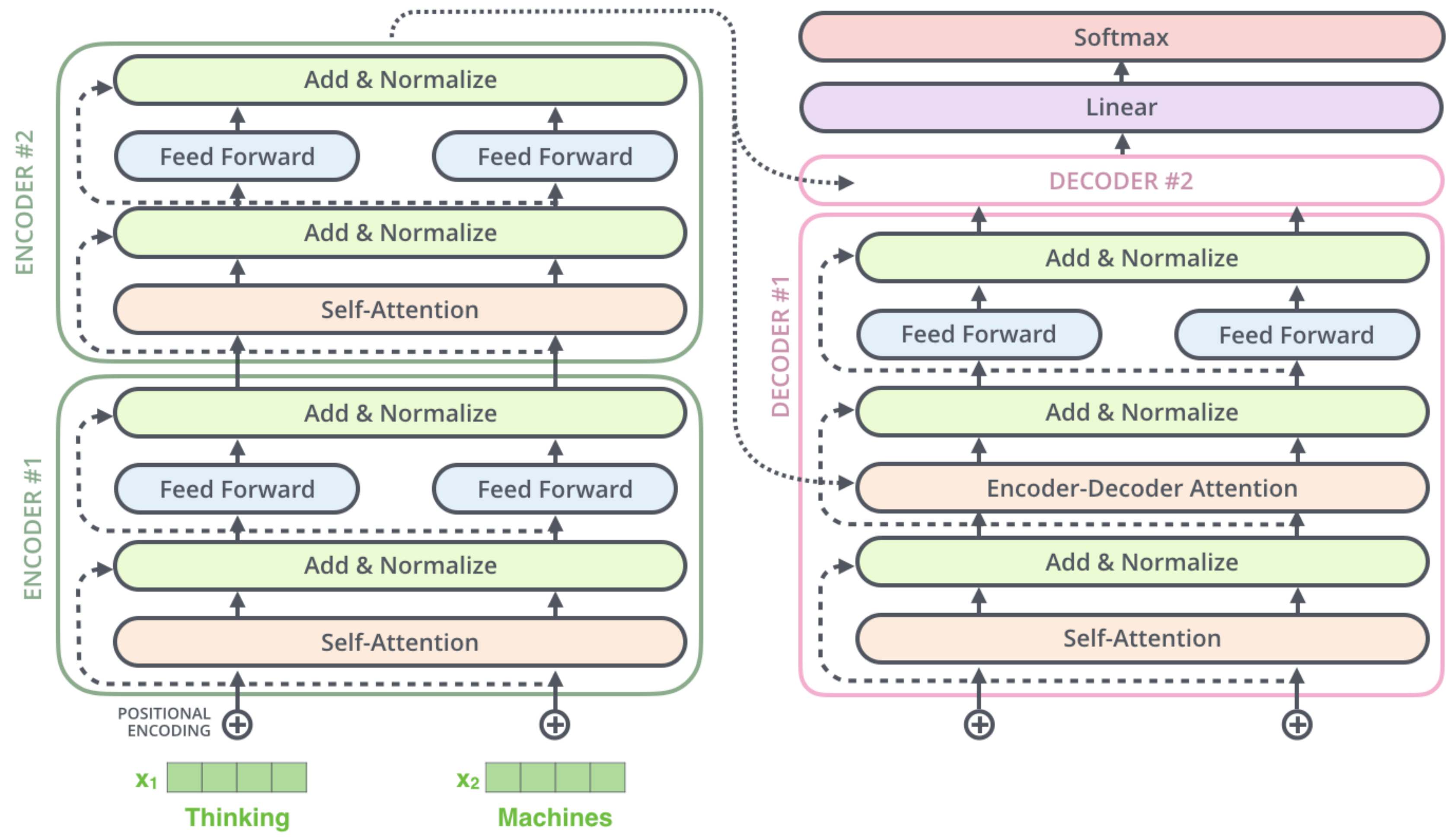
The Residuals

each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.




Add & Normalize
Feed Forward
Feed Forward
z_1
z_2
Add & Normalize
X
Z
LayerNorm(+)
z_1
z_2
Self-Attention
x_1
x_2
ENCODER #1
POSITIONAL ENCODING
x_1
x_2
Thinking
Machines

Stacked Encoder

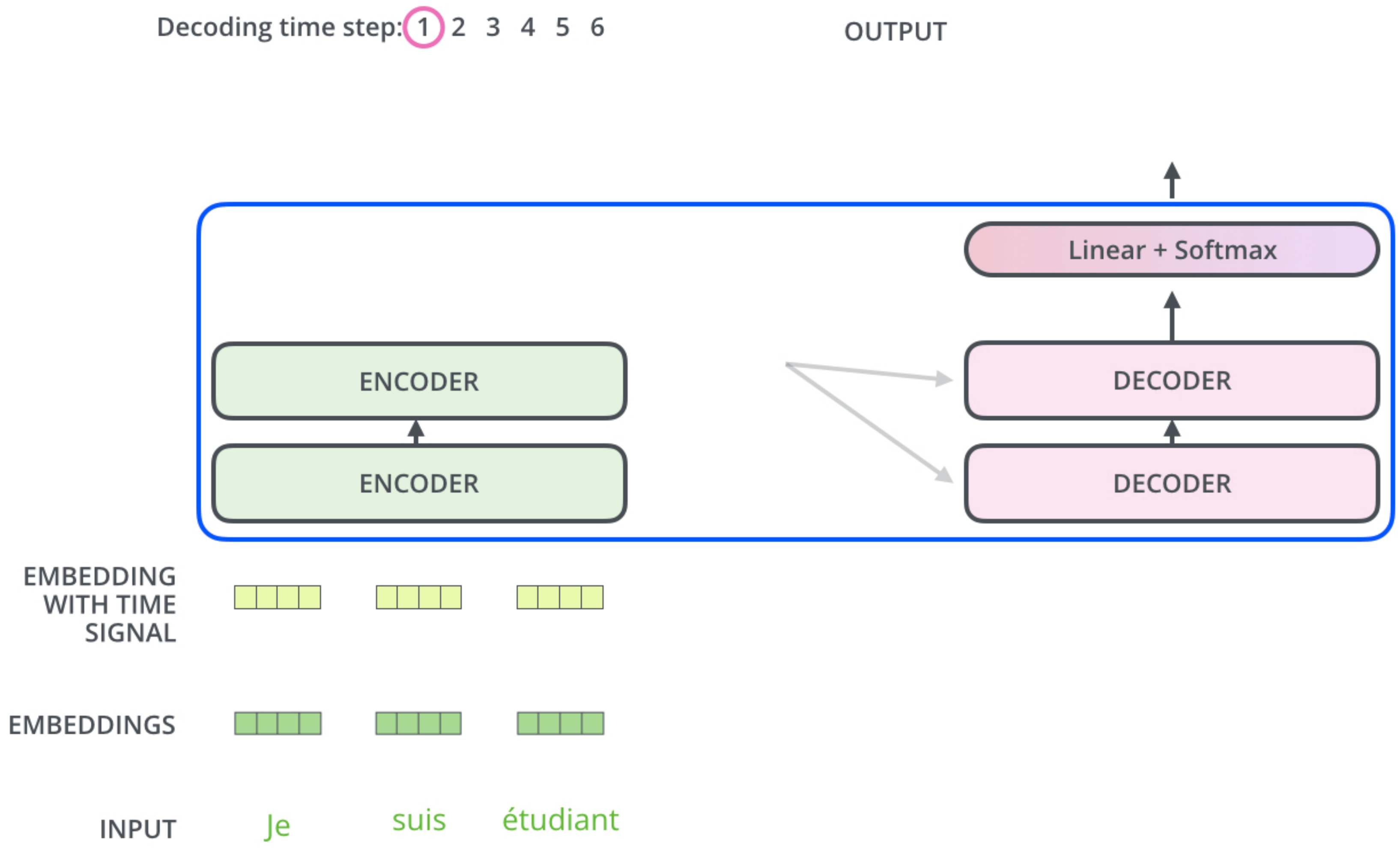


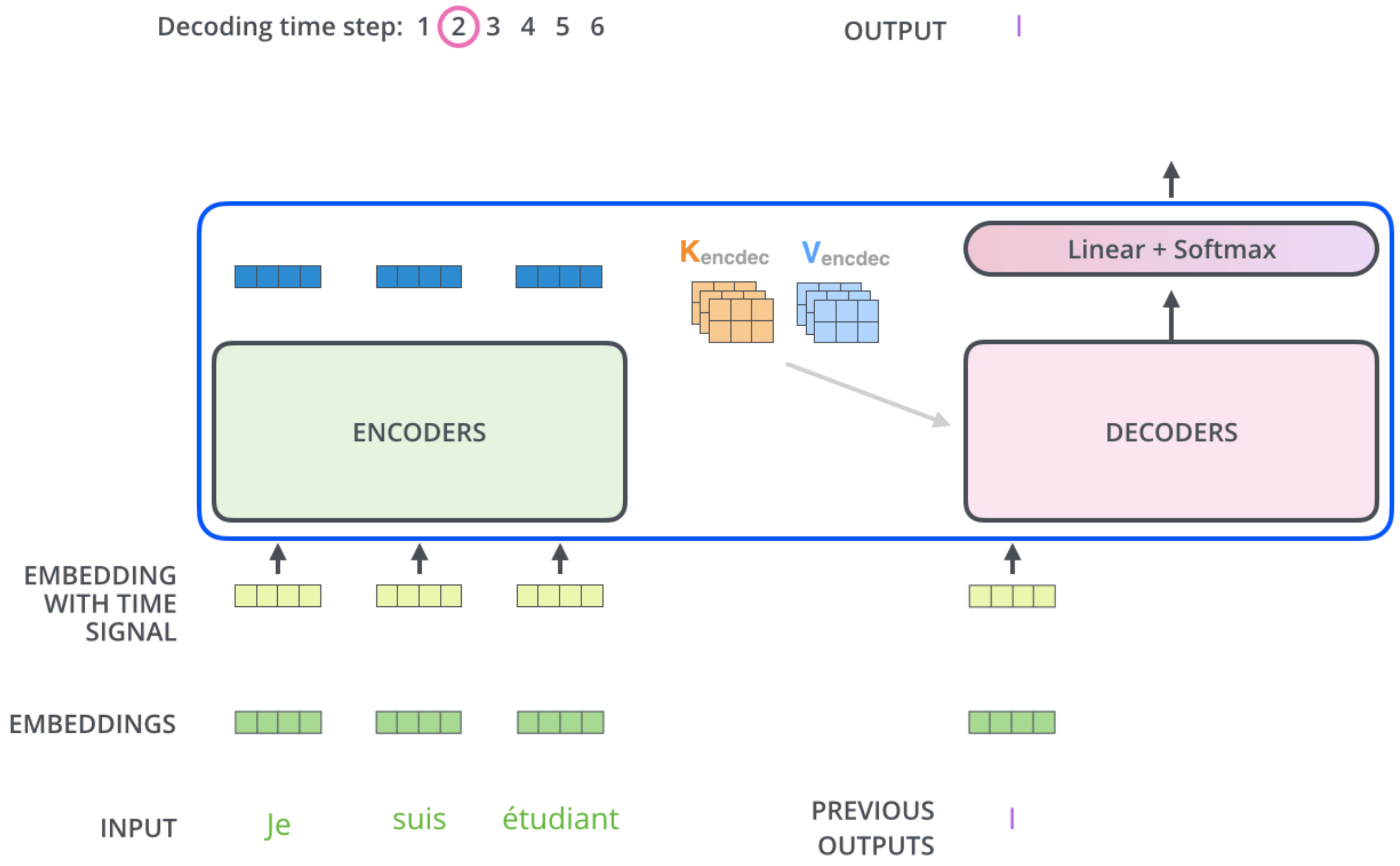
Decoder Side

The encoder start by processing the input sequence.

The output of the top encoder is then transformed into a set of attention vectors K and V.

These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence:




Decoding time step: 1 2 3 4 5 6
OUTPUT
I
Kencdec
Vencdec
Linear + Softmax
ENCODERS
DECODERS
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant
PREVIOUS OUTPUTS
I

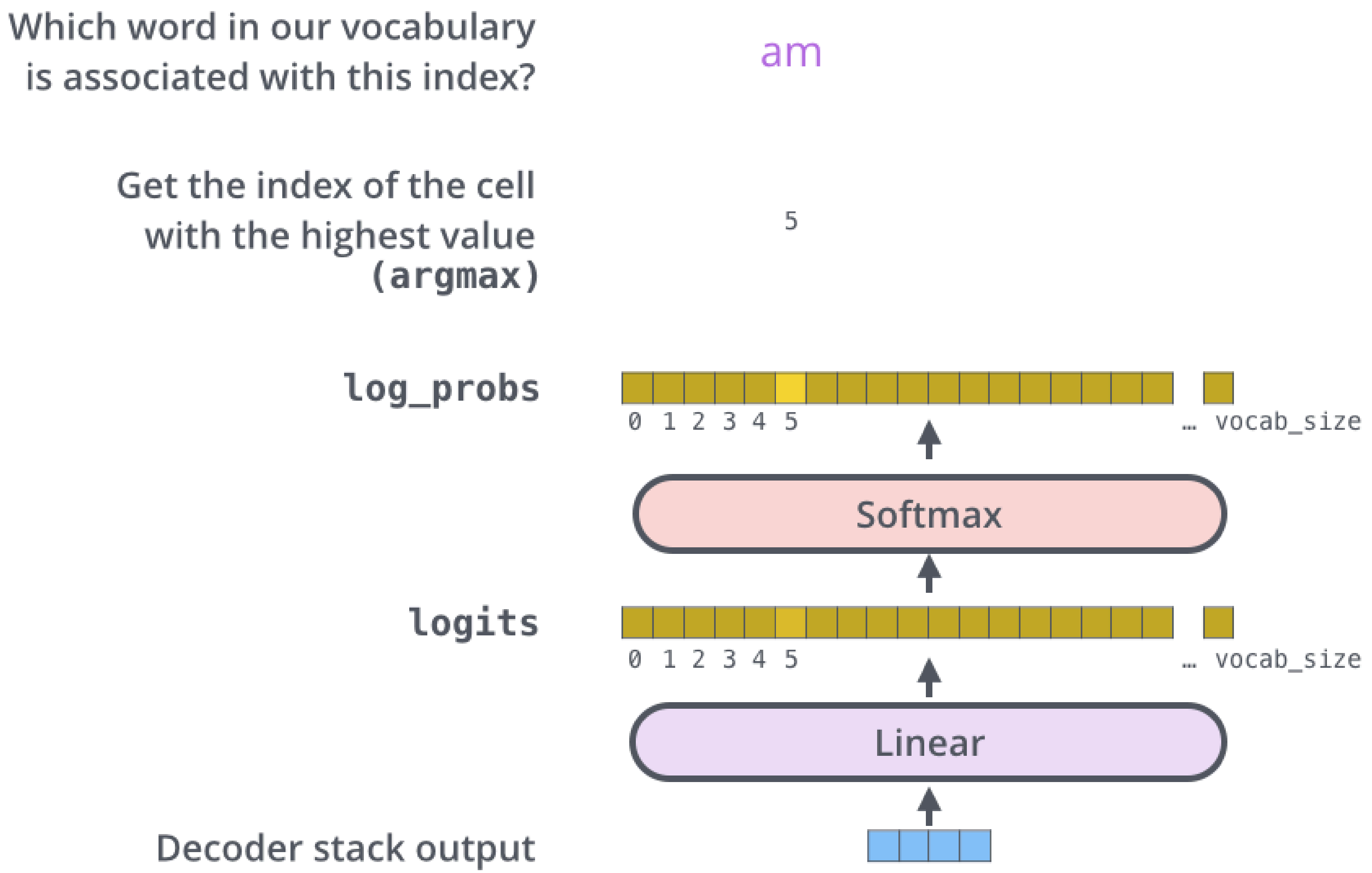
Decoder Attention

- In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

- The “Encoder-Decoder Attention” layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

The Final Linear and Softmax Layer

- a Softmax Layer to output word
- The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders into the logits vector.
- Let’s assume that our model knows 10,000 unique English words (our model’s “output vocabulary”) that it’s learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.
- The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

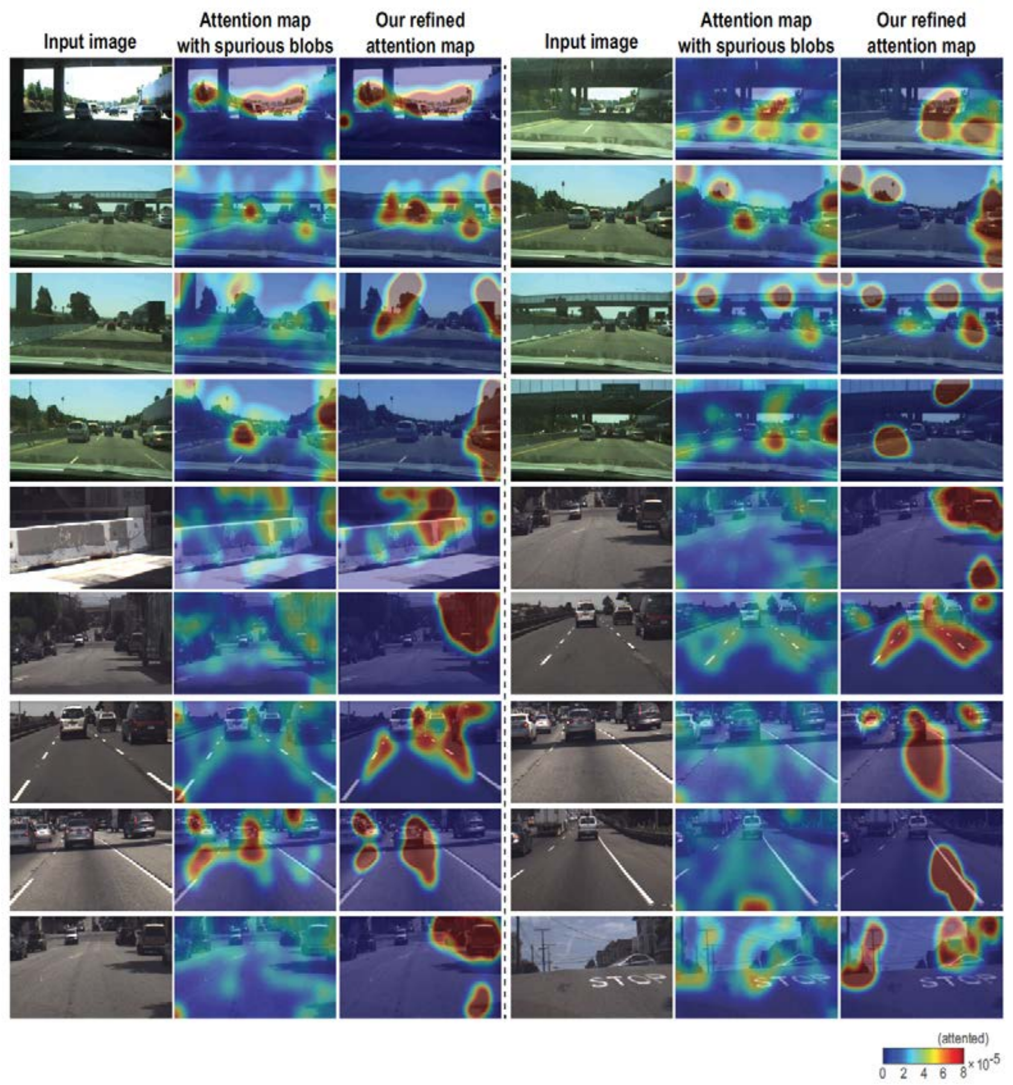

Which word in our vocabulary is associated with this index?
am
Get the index of the cell with the highest value (argmax)
5
log_probs
0 1 2 3 4 5
… vocab_size
Softmax
logits
0 1 2 3 4 5
… vocab_size
Linear
Decoder stack output

- https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html

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

Attention and Interpretability

- Attention models learn to predict salient (important) inputs.
- Attention visualizations help users understand the causes of the network's behavior.
- Not every attended region is actually important, but post-processing can remove regions that aren't.


Input image
Attention map with spurious blobs
Our refined attention map
Input image
Attention map with spurious blobs
Our refined attention map
(attented)
0 2 4 6 8 ×10⁻⁵