Deep Learning
CSC-Elective

Instructor : Dr Muhammad Ismail Mangrio

Slides prepared by Dr. M Asif Khan

ismail@iba-suk.edu.pk

Unit 02 NLP Week 2

Contents

- Embedding layer
- Encoder & Decoder
- Attention model
- Transformers

Embedding layer

- It is a component of neural networks used to **map words or tokens to vectors** of real numbers, allowing them to be processed by DL models.
- This transformation converts discrete, symbolic representations (words) into **continuous vectors** in a lower-dimensional space.
- The goal is to **capture the semantic** meaning of words, where **similar words** are represented by **similar vectors**.
- **Why** Use an **Embedding** Layer?
  - **Reduces dimensionality** compared to using one-hot encoding.
  - **Encodes semantic** information, allowing models to **understand relationships** between words.

Embedding layer working

1. **Input Representation**:
   - Each word is represented by an integer index corresponding to its position in a vocabulary.
2. **Embedding Matrix**:
   - The embedding layer contains a **matrix of weights**, where each **row corresponds to a word index** and each **column corresponds** to a **dimension** in the **embedding space**.
   - Typically, these embeddings are **initialized randomly** and are **fine-tuned during training**.
3. **Process**:
   - The **input sequence** (list of word indices) is **passed through the embedding layer**, which **converts** each index into its corresponding **word vector**.
   - These vectors are used **as input to subsequent layers** (e.g., **LSTMs, CNNs**) for further processing.

Embedding layer working

- **Example of Word Embedding:**
  - Word: "King“, e.g., its is Embedding vector with dimension=3 could be [0.3, 0.7, 0.4]
- **Advantages of Using Embeddings:**
  - **Semantic Relationships**: Embeddings help capture relationships like synonyms, antonyms, and word analogies (e.g., "King - Man + Woman = Queen").
  - **Efficient Computation**: Embedding layers reduce the dimensionality compared to methods like one-hot encoding.

Every actor (word) has a **role** (vector) that defines:
- who they are
- how they interact with others

*Code is available at elearnnig with file NLP14_Word_embedding_Layer.ipynb

Embedding layer working

```python
import tensorflow as tf

# Number of unique words in vocabulary
vocab_size = 10

# Each word will be represented by 4 numbers
embedding_dim = 4

# Create the embedding layer
embedding_layer = tf.keras.layers.Embedding(
    input_dim=vocab_size,    # total words
    output_dim=embedding_dim # vector size for each word
)

# Example sentence encoded as word IDs
# Sentence: [2, 5, 7]
word_ids = tf.constant([[2, 5, 7]])

# Convert word IDs to embeddings
embedded_output = embedding_layer(word_ids)

print("Embedding Output:\n", embedded_output)
print("\nOutput Shape:", embedded_output.shape)
```

```
Embedding Output:
 tf.Tensor(
[[[ 0.01640708 -0.00819281  0.04352799 -0.04505451]
  [ 0.00727729  0.02796285 -0.02823366  0.04813262]
  [-0.00543053 -0.04293538 -0.00775633 -0.04536946]]], shape=(1, 3, 4), dtype=float32)

Output Shape: (1, 3, 4)
```

seq2seq models

- Seq2Seq is a DL model architecture used for transforming **one sequence of data into another sequence.** It can learn grammar, meaning, context, translation patterns.
- It is commonly used in NLP tasks where both the **input and output are sequences**, such as:
  - Machine Translation (e.g., English → French)
  - Text Summarization (Long Text --> Short Summary)
  - Speech Recognition (Audio → Text)
  - Image Captioning (Image features → Caption)
  - Medical Reports (Symptoms -> Diagnosis notes)

Encoder RNN (LSTM/GRU)
↓
Context (thought) Vector
↓
Decoder RNN (LSTM/GRU)

seq2seq models: Encoders & Decoders

- **Encoder**:
  - It processes the input sequence (e.g., a sentence in a language) **and transforms it into a fixed-size context vector** that summarizes the information.
  - It is responsible for **capturing the relevant features of the input** to pass on to the decoder for generating output.
- **Components of the Encoder**:
  - **Input Sequence**:
    - A sequence of tokens (words or characters) from the input text.
    - Example: "Hello, how are you?“
  - **Embedding Layer**:
    - Converts words into continuous vector representations (embeddings) to make them understandable for the model.
  - **Recurrent Neural Networks (RNNs), GRUs, LSTMs**:
    - Common architectures used in the encoder to process input sequences.
    - RNNs, GRUs, and LSTMs are capable of handling sequential data and learning long-range dependencies.
  - **Context Vector**:
    - The output of the encoder is a **fixed-length vector** (often referred to as the context vector), which contains all the encoded information from the input sequence.

seq2seq models: Encoders & Decoders

- **Decoder**:

1. Takes the context vector.
2. Generates output sequence **one word at a time**.
3. Uses previous output as input for next step.

During training:

- The correct previous word (ground truth) is given to the decoder.

During testing:

- The decoder uses its **own predicted word** for the next step.
- This introduces exposure bias, but works well in practice.

seq2seq models: Encoders & Decoders

- **Decoder**:
  - The Decoder takes the **context vector** from the encoder and **generates the output sequence** (e.g., a translated sentence, summary, or response)..
- **Components of the Decoder**:
  - **Decoding process:** Initial input: Typically starts with a special token like <START> or <SOS> to initiate the generation.
  - **Subsequent tokens:** The decoder **generates one word/token** at a time **based on previous predictions**.
  - **Hidden states:** The hidden state is updated after each word generation, which is fed as input to predict the next word.
- **Input to the Decoder:**
  - **Context Vector**: Encodes the source sentence or input sequence and provides the initial hidden state for the decoder.
  - **Decoder's own predictions**: After the first word is predicted, the predicted word is used as input for generating the next word.
- **Training the Decoder:**
  - The decoder is trained alongside the encoder in an end-to-end fashion using supervised learning.
  - The model is optimized to minimize the difference between the predicted output and the actual target sequence.

seq2seq models: Encoders & Decoders

- **Use Cases of Decoders:**
  - **Machine Translation**: Translating text from one language to another (e.g., English to Urdu).
  - **Text Summarization**: Generating concise summaries from longer texts.
  - **Speech-to-Text**: Converting spoken language into written text.
  - **Image Captioning**: Describing the contents of an image with sentences.

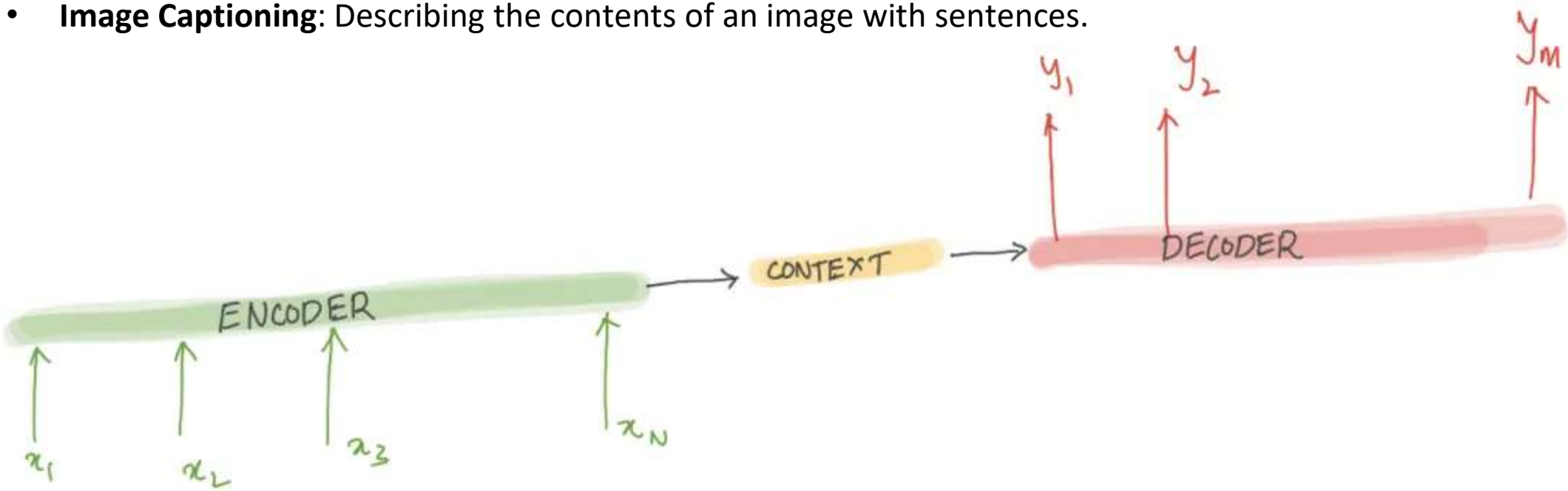


Diagram from: https://medium.com/nerd-for-tech/nlp-theory-and-code-encoder-decoder-models-part-11-30-e686bcb61dc7

seq2seq models: Encoders & Decoders

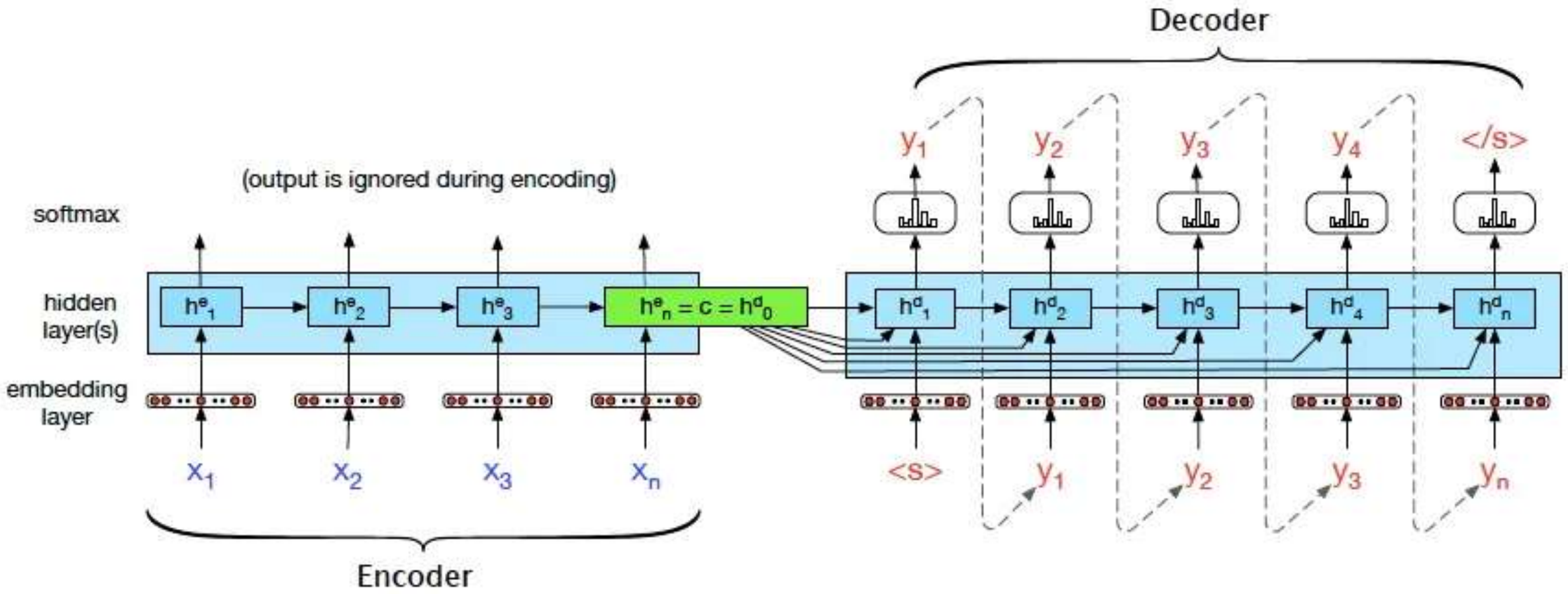


Diagram from: https://medium.com/nerd-for-tech/nlp-theory-and-code-encoder-decoder-models-part-11-30-e686bcb61dc7

seq2seq models: Encoders & Decoders challenges

- The primary issue that arises in long sequences is that traditional encoder-decoder models rely on a **fixed-length context vector** to represent the entire input sequence.
- The encoder compresses all the information in the sequence into a single vector (often a final hidden state or the last layer of the RNN/LSTM/GRU), and the decoder uses this context vector to generate the output.
- For **shorter sequences,** the context vector is usually sufficient to capture the essential information. There is **less risk of losing information**, and the model can generate accurate outputs with less difficulty.
- **Long Sequences**: When the input sequence is long, the fixed-length context vector cannot represent all the information effectively. The model has to compress potentially thousands of tokens into a single vector, which is difficult to do accurately, leading to:
  - Vanishing/exploding gradient problem
  - Loss of information
  - Difficulty in capturing long term dependencies
- Encoder & Decoder code available on elearning.

Diagram from: https://medium.com/nerd-for-tech/nlp-theory-and-code-encoder-decoder-models-part-11-30-e686bcb61dc7

Attention model

- This paper proposed the use of attention mechanisms in neural machine translation to overcome the limitations of traditional sequence-to-sequence (seq2seq) models.

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

Attention model

- **Problem with Traditional Encoder-Decoder Models**:
  - **Fixed Context Vector**: Traditional models (e.g., seq2seq with LSTM/GRU) compress the entire input sequence into a single context vector, which limits the model's ability to handle long sequences.
  - **Information Loss**: Long-range dependencies or important words might not be effectively represented in a single context vector.
- **The Attention Solution**:
  - **Focus on Relevant Information**: Attention allows the model to dynamically focus on specific parts of the input sequence **at each decoding step**.
  - **Improved Contextual Understanding**: Instead of relying on a single fixed context vector, the decoder uses a weighted sum of encoder states (**context vectors**) tailored to the current decoding step.

Attention model working

- **Encoder**:
  - **Bi-directional LSTM/GRU**: The encoder processes the input sequence (e.g., a sentence) and outputs a sequence of hidden states that capture contextual information from both directions (left-to-right and right-to-left).
- **Attention Calculation**:
  - **Attention Scores**: At each decoding step, the decoder calculates a set of attention scores to determine which part of the encoder's hidden states to focus on.
  - **Dot Product/Other Scoring Mechanisms**: The attention score is typically **computed using the decoder's current hidden state and the encoder's hidden states**.
- **Context Vector**:
  - The **attention scores** are used to compute a **weighted sum** of the encoder's hidden states (ht), creating a context vector. This vector informs the decoder at each step.

Attention model working

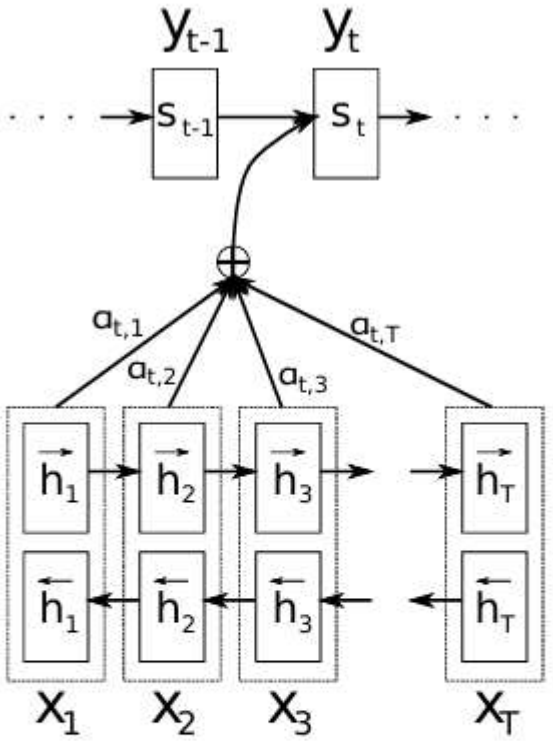


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})},$$

$$e_{ij} = a(s_{i-1}, h_j)$$

$\boldsymbol{e_{ij}}$ is the un-normalized attention score between the i-th decoder hidden state s_{i-1} (from the previous decoding step) and the j-th encoder hidden state h_j (from the input sequence). It is typically computed as the **dot product of the encoder state and the decoder state** (or, more precisely, the decoder's hidden state).
Si is decoder state, Ci is context vector, eij is energy score

Attention model working

- The attention layer dynamically selects which parts of the input (encoder outputs) are most relevant to the current decoding step.
  1. Compute alignment scores:
     - For each encoder hidden state h_i, a score $e_{t,i}$ is computed with the current decoder state s_{t-1}:

$$e_{t,i} = a(s_{t-1}, h_i)$$

     - (where $a(\cdot)$ is a small neural network, e.g., a feed-forward layer):
  2. Normalize to get attention weights:

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{k=1}^{T} \exp(e_{t,k})}$$

     - These $\alpha_{t,i}$ tell how much the decoder should "attend" to each input word when generating y_t.
  3. Compute context vector:

$$c_t = \sum_{i=1}^{T} \alpha_{t,i} h_i$$

     - **The $\oplus$ (circle with cross)** in the diagram represents this weighted sum of encoder outputs.
- Output Generation:

$$P(y_t | y_{<t}, X) = \text{softmax}(W_o[s_t; c_t])$$

  - Once the context vector c_t is computed, it is combined with the decoder's current hidden state s_t to predict the next word y_t:

Attention model working

- This equation (S_i) describes how the decoder's hidden state at the current time step i is computed

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

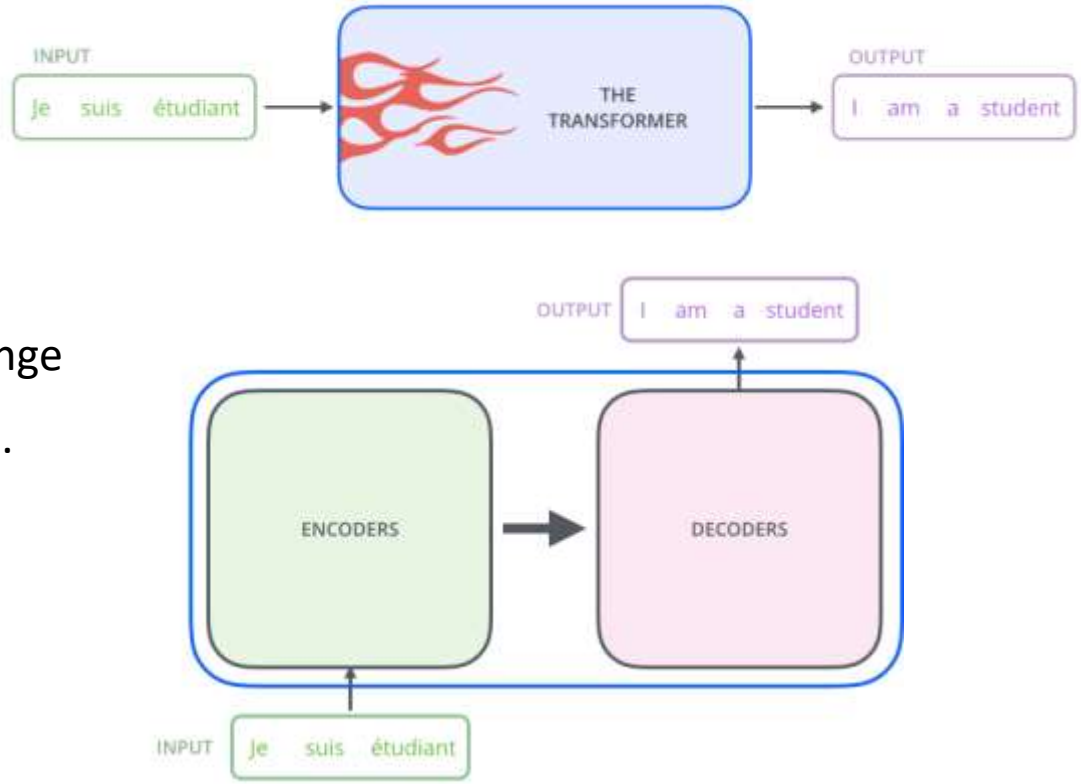
- s_i is the hidden state of the decoder at time step i.
- s_{i-1} is the previous hidden state of the decoder (i.e., the hidden state from the previous time step, i−1).
- y_{i-1} is the previous output of the decoder, which is typically the token generated at time step i−1 in the target language.
- C_i is the context vector at time step i, which is computed by the attention mechanism. It is a weighted sum of the encoder's hidden states based on the attention scores.
- f is a function (often a neural network layer such as an LSTM, GRU, or simple feed-forward network) that combines the previous decoder state s_{i-1} , the previous output y_{i-1} , and the context vector c_i to produce the new decoder state s_i.

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j.$$

- α_{ij} (weight) is the attention score between the i-th decoder state and the j-th encoder state.
- The attention score α_{ij} indicates how much importance or attention the decoder should give to the j-th encoder hidden state h_j when generating the i-th output word in the target sequence.
- This represents the j-th encoder hidden state. Each h_j is a vector representing the context of the input sentence at the j-th position.
- T is upper limit of the summation indicates the total number of encoder hidden states (or the length of the input sequence). T is the total number of time steps in the input sequence.

Transformers

- **Transformers** are a type of neural network architecture introduced in the paper "Attention is All You Need" (Vaswani et al., 2017)
- Revolutionized NLP by leveraging the **self-attention mechanism** to capture relationships between words in a sequence.
- Key benefits:
  - **Parallel processing**: Unlike RNNs, transformers process all tokens simultaneously.
  - **Long-range dependencies**: Efficiently captures long-range dependencies without the vanishing gradient problem.



Credit to link: https://jalammar.github.io/illustrated-transformer/

Transformers

- **Key Components of Transformer:**
  - **Encoder-Decoder Structure:**
    - **Encoder**: Processes the input sequence to extract features.
    - **Decoder**: Generates the output sequence from the features.
  - **Self-Attention:** Weighs the importance of each word in relation to others within a sentence.
  - **Positional Encoding:** Adds information about the position of words in the sequence (since transformers don't have a built-in sense of word order).
  - **Feed-forward Neural Networks:** Applied after the attention mechanism to process the sequence further.

Transformers

- We need to understand each component of the following model, which is **Transformer**:
- Each component will be explained step b step to clarify understand of this marvelous architecture.
- This figure is taken from: “Attention is All You Need” paper.

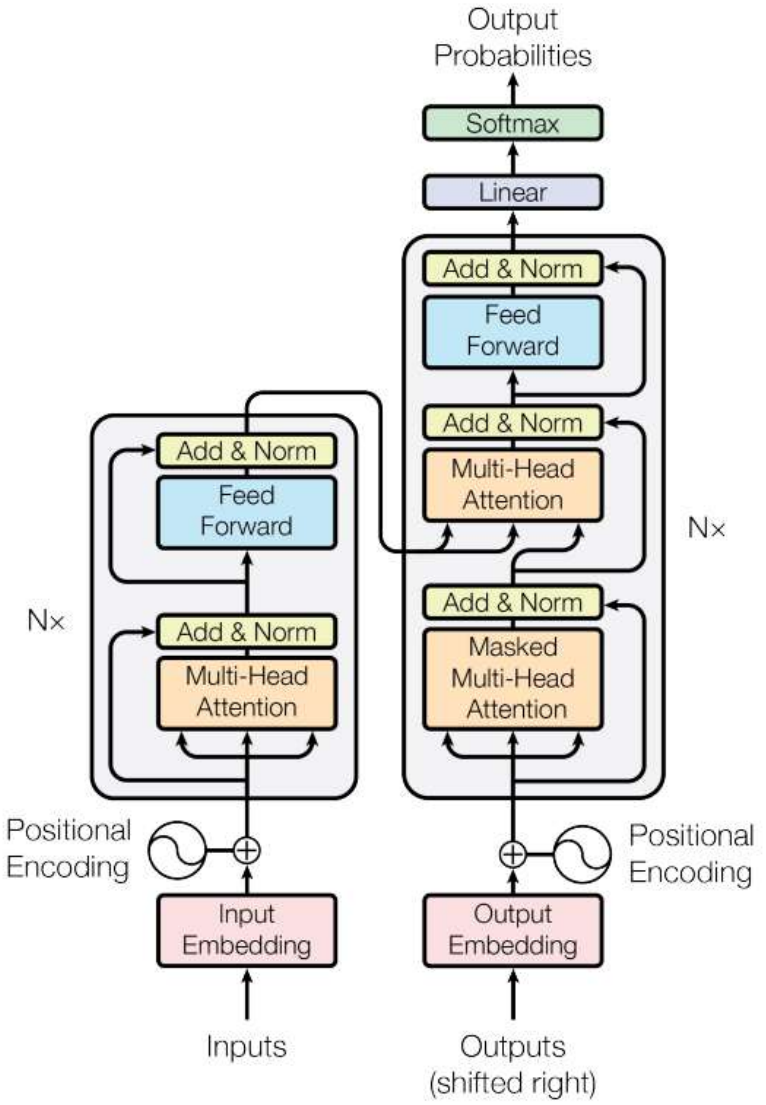
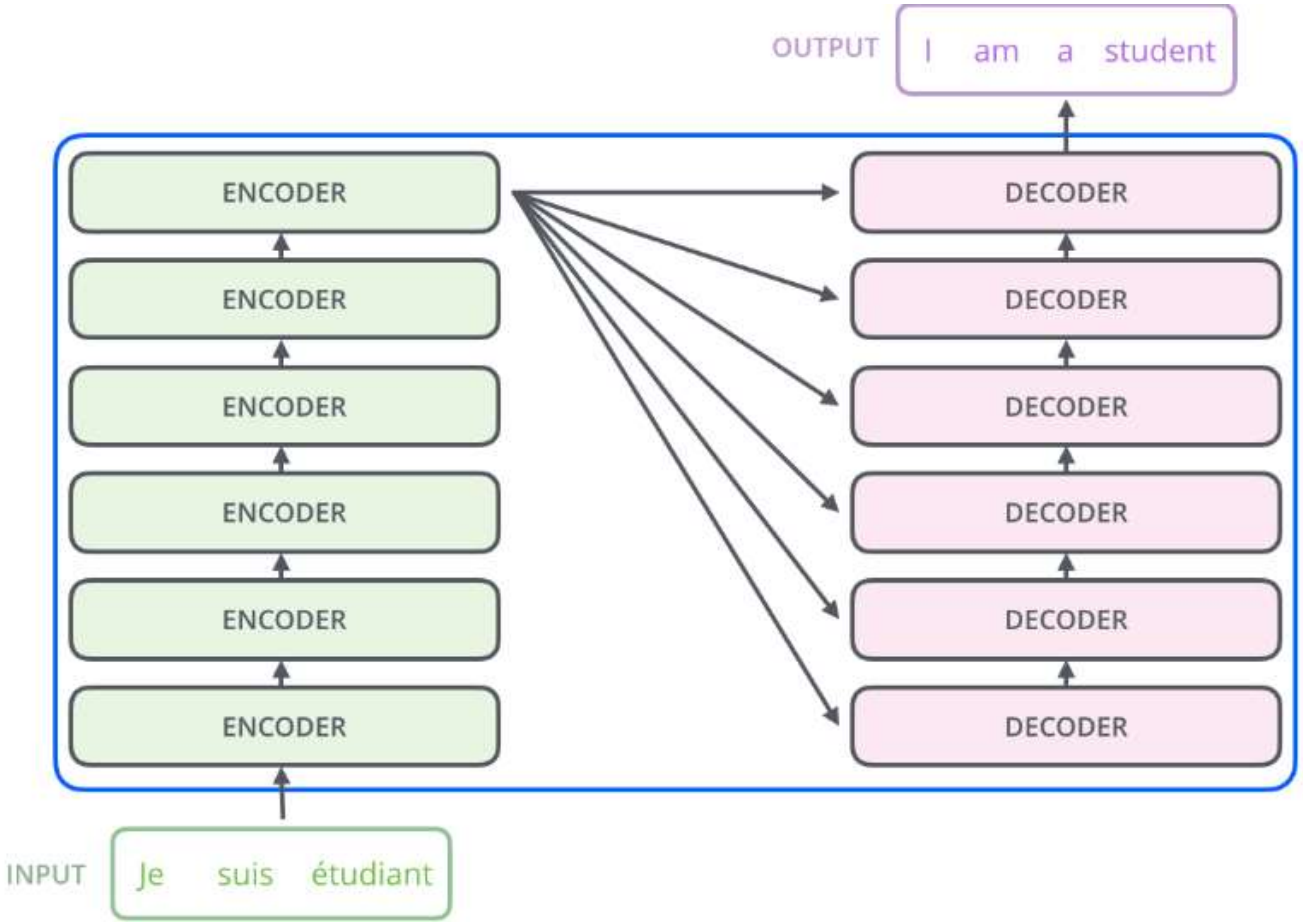


Figure 1: The Transformer - model architecture.

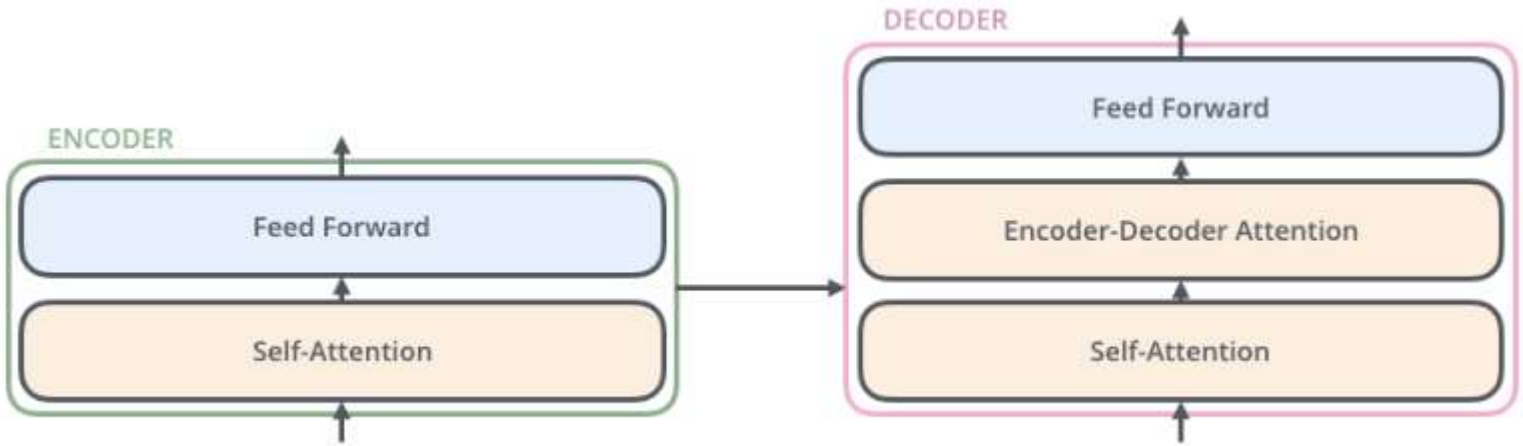
Transformers

- The encoding component is a **stack of encoders** (the **paper stacks six** of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements).
- The decoding component is a **stack of decoders** of the **same number**.
- The encoders are all identical in structure (yet they do not share weights).
- Each encoder one is broken down into two **sub-layers**.



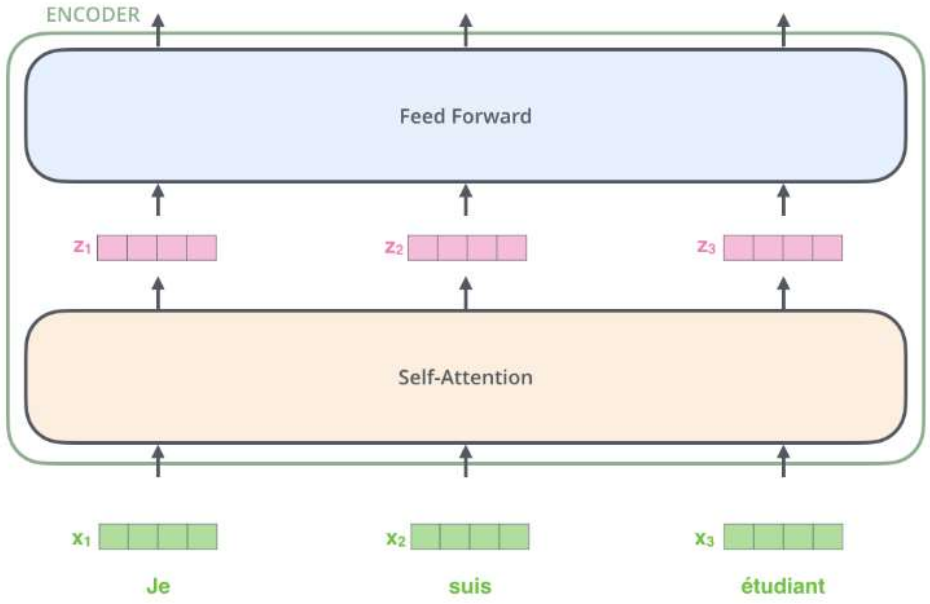
Transformers

- The encoder's inputs first flow through a **self-attention layer** – a layer that helps the encoder **look at other words** in the input sentence as it encodes a specific word.
- The outputs of the self-attention layer 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 (similar what attention does in seq2seq models).



Transformers

- The **embedding** only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the **size 512**.
- In the bottom encoder that would be the word embeddings, but in other (after first encoder in the stack of encoders) encoders, it would be the output of the encoder that's directly below.
- The **number of encoder** stack's size is **hyper parameter**.
- After embedding, words in our input sequence, flows through each of the two layers of the encoder.

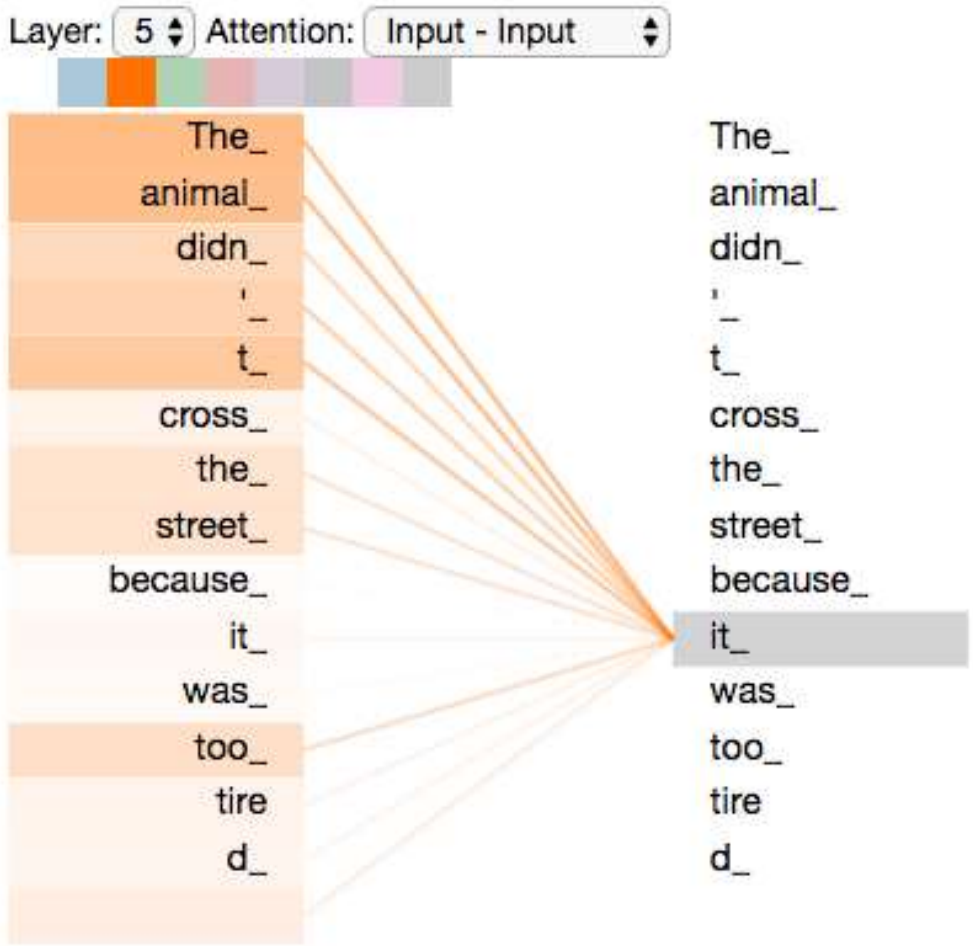


Transformers

- Say the following sentence is an input sentence we want to translate:
- ”The animal didn't cross the street because it was too tired”
- What does “it” in this sentence refer to? Is it referring to the street or to the animal? It’s a simple question to a human, but not as simple to an algorithm.
- When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.
- As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

Transformers (Self attention)

- Self-attention is the method the Transformer uses to bake the “understanding” of other relevant words into the one we’re currently processing.



Transformers (Self attention)

- The **first step** in calculating self-attention is to create **three vectors** from each of the encoder's input vectors (in this case, the embedding of each word).
- So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**.
- These vectors are created by multiplying the embedding by three matrices that we trained during the training process.
- Each of the three vectors dimension is 64, whereas input word from embeddings is 512.
- These three vectors (Q, k & V) are abstractions that are useful for calculating and thinking about attention.

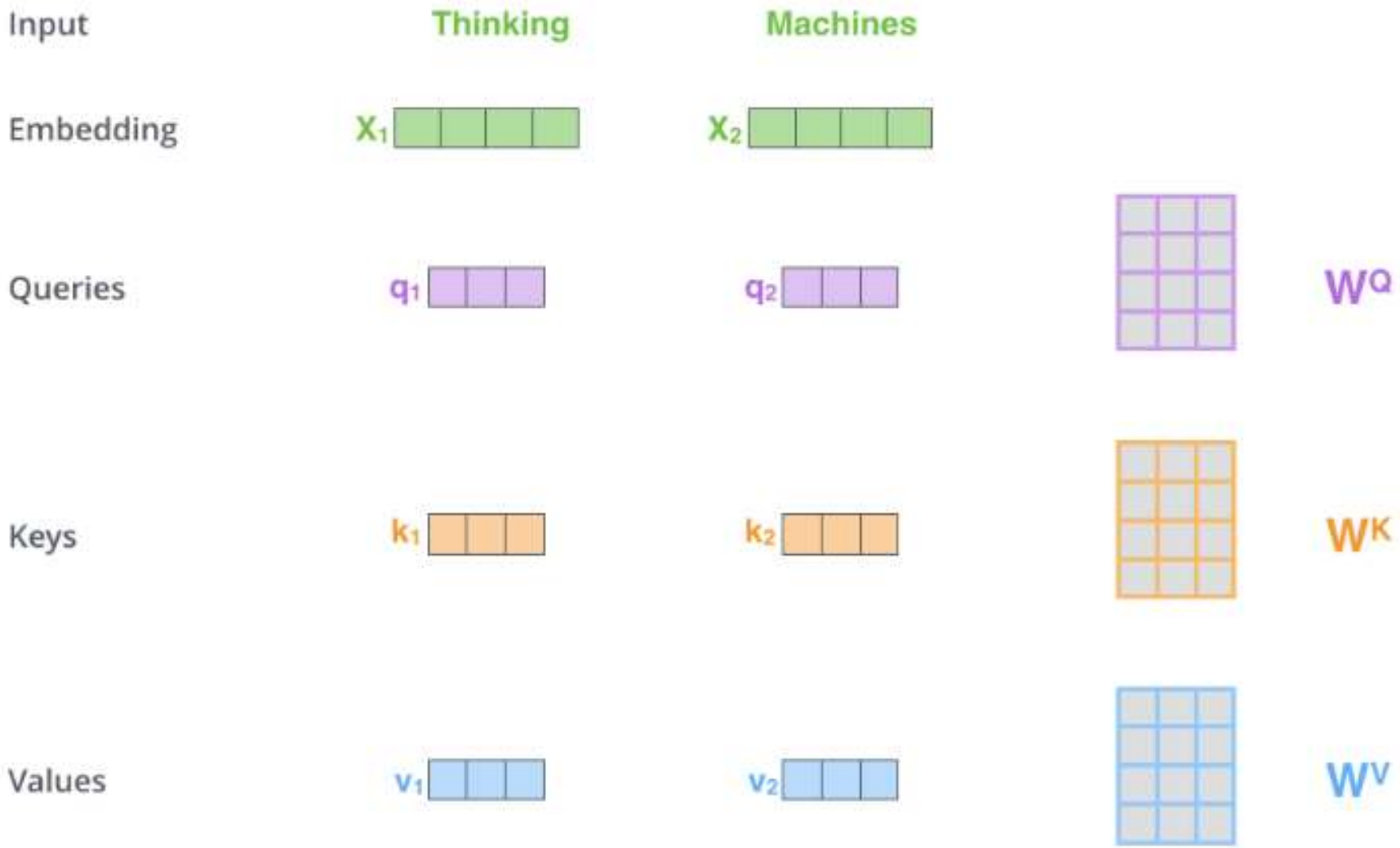
Vector	Purpose	Intuition
Query (Q)	What this token is looking for in other tokens	Like asking, "What am I trying to find?"
Key (K)	What this token offers to others	Like saying, "Here's what I contain."
Value (V)	The actual information to be passed along	The content that will be combined, if attended to

Transformers (Self attention)

- Q (Query) → the question being asked: “Whom should I pay attention to?”
- K (Key) → the tag that identifies what kind of information each note contains.
- V (Value) → the actual content of the note — the useful information to read once you decide to pay attention.
- Step-by-Step Example: “The cat sat”

Word	Query (Q)	Key (K)	Value (V)
The	What noun should I describe?	I am a determiner	My meaning is “The”
Cat	Which verb tells my action?	I am a noun	My meaning is “cat”
Sat	Who performed me?	I am a verb	My meaning is “sat”

Transformers (Self attention)



Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

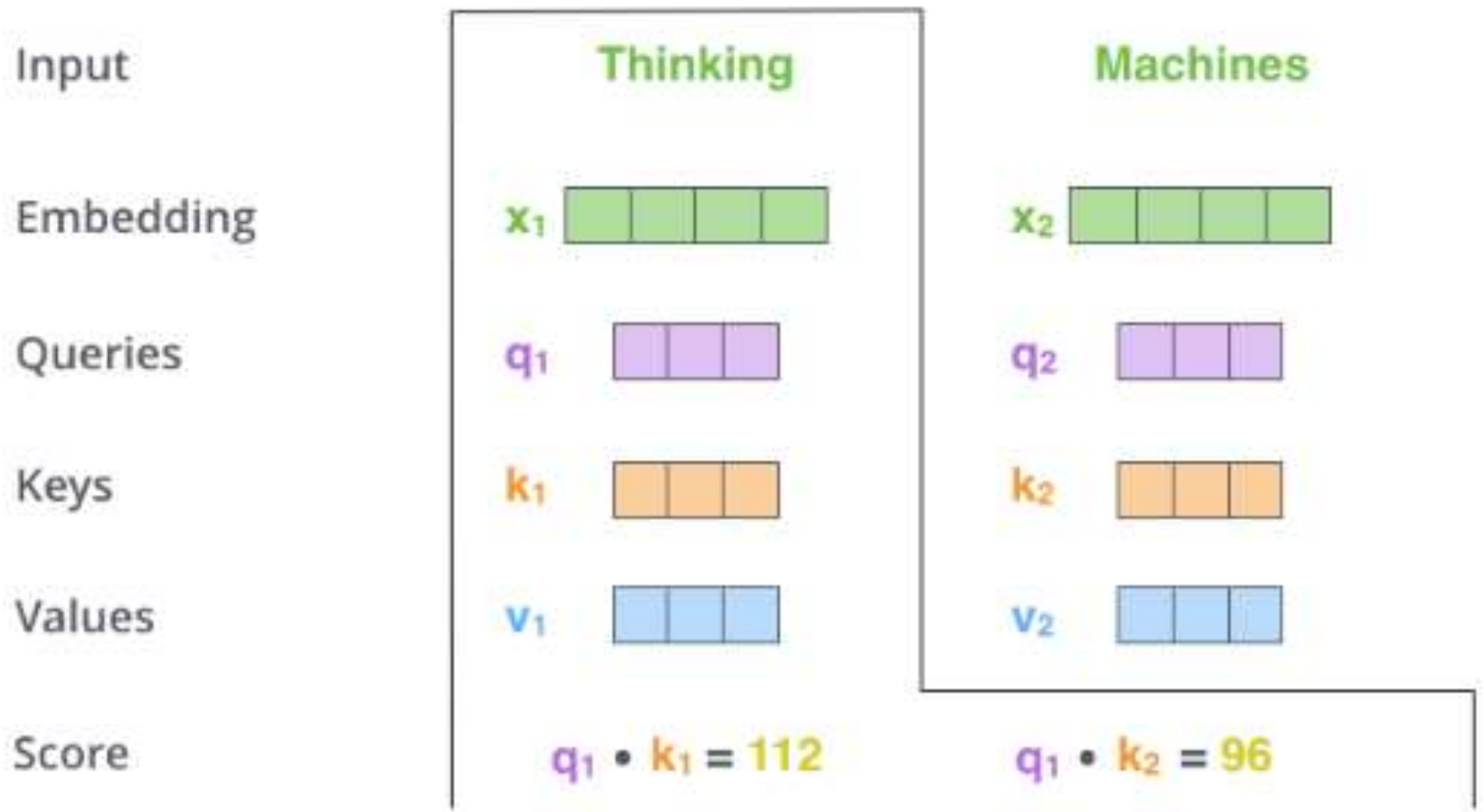
Transformers (Self attention)

- The **second step** in calculating self-attention is to calculate a score.
- For calculating the self-attention of “Thinking”. We need to **score each word** of the input sentence against this word.
- The **score determines how much focus to place** on other parts of the input sentence as we encode a word at a certain position.
- The score is calculated by taking the **dot product of the query vector with the key vector** of the respective word we’re scoring.
- So if we’re processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.

Transformers (Self attention)

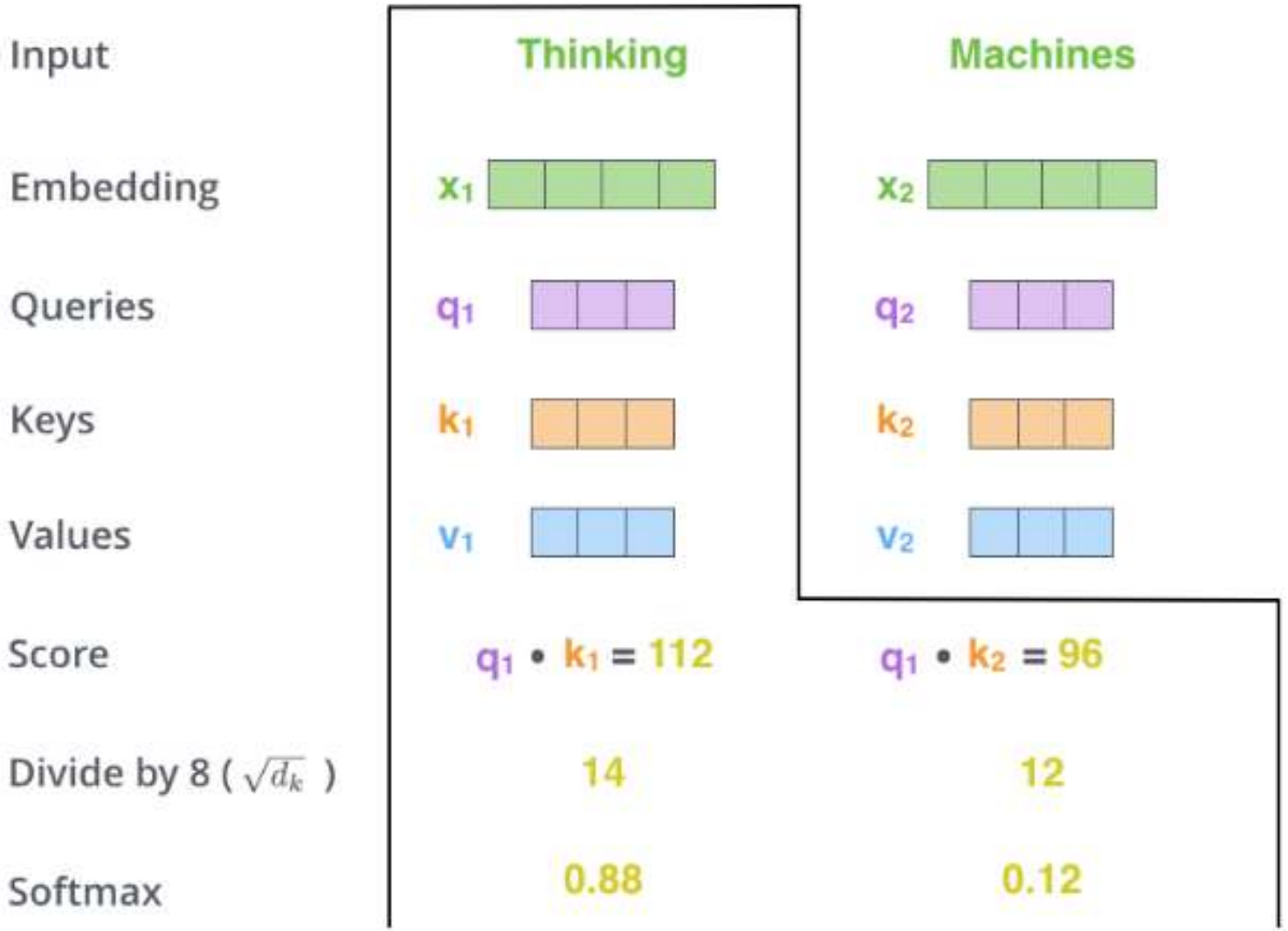
- To find the attention score for the first word (let's call it "The"), you do the following:
  - Compute the dot product of the Query (Q) vector of the first word with the Key (K) vectors of all the words in the sequence (including itself).
- Mathematically, for the first word (index 1):
- Score1 = $Q1 \cdot K1$, score2 = $Q1 \cdot K2$, score3 = $Q1 \cdot K3$, score4 = $Q1 \cdot K4$, score5 = $Q1 \cdot K5$
- So, you calculate 5 dot products:
  - score1 is the dot product of Q of "The" with K of "The“
  - score2 is the dot product of Q of "The" with K of "cat“
  - score3 is the dot product of Q of "The" with K of "sat“
  - score4 is the dot product of Q of "The" with K of "on“
  - score5 is the dot product of Q of "The" with K of "mat“
- These dot products give the attention scores that measure how much focus the first word should give to each of the other words in the sequence.

Transformers (Self attention)



Transformers (Self attention)

- **3rd & 4th steps** divides the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64.
- In general we divide the attention score by the square root of the dimension of the Key vector ($\sqrt{d_k}$)
- This leads to having more stable gradients (**to prevent values becoming too large**).
- There could be other possible values here, but this is the default), then pass the result through a softmax operation.
- Softmax **normalizes the scores** so they're all positive and add up to 1.

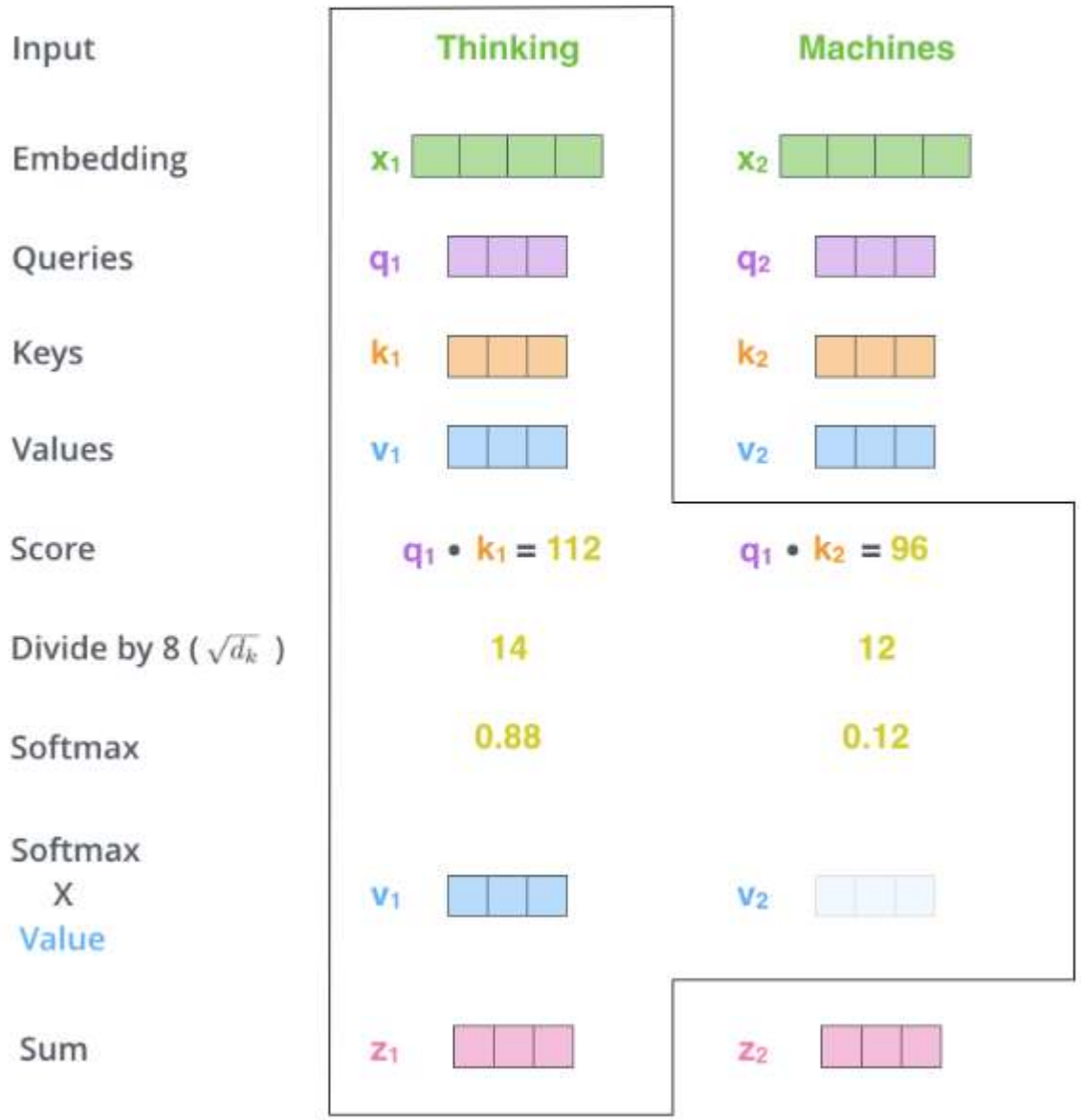


Transformers (Self attention)

- **3rd & 4th steps**
- After computing the attention scores, a softmax function is applied to them.
- The **softmax** function **converts the raw scores into probabilities that sum to 1**, ensuring that each word's attention weight is normalized.
- This tells you how much each word should "attend to" (i.e., the relative importance of each word with respect to the first word).
- This softmax score determines how much each word will be expressed at this position.
- Clearly the word at this position will have the highest softmax score, but sometimes it’s useful to attend to another word that is relevant to the current word.

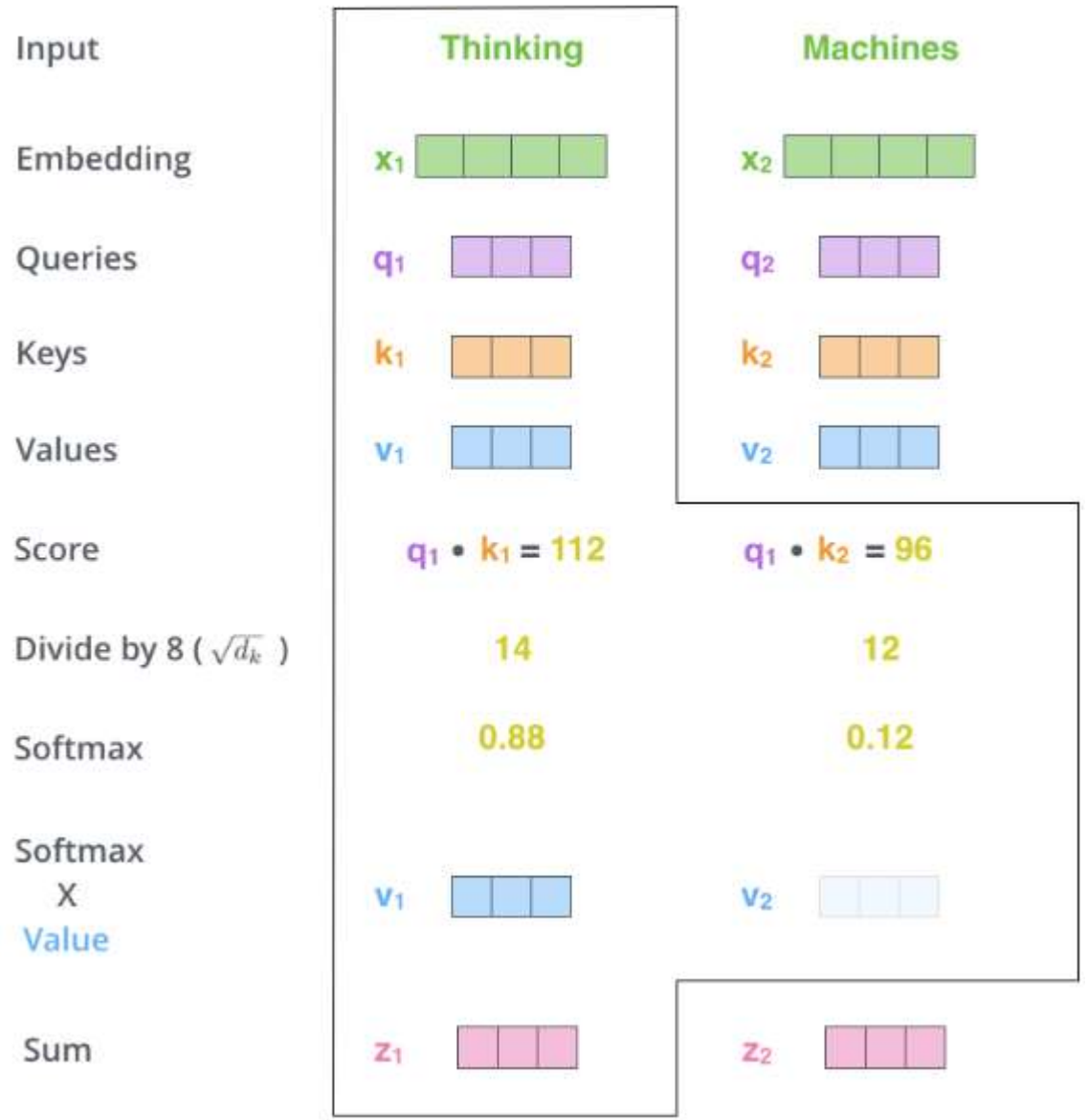
Transformers (Self attention)

- The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up).
- The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).
- The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).



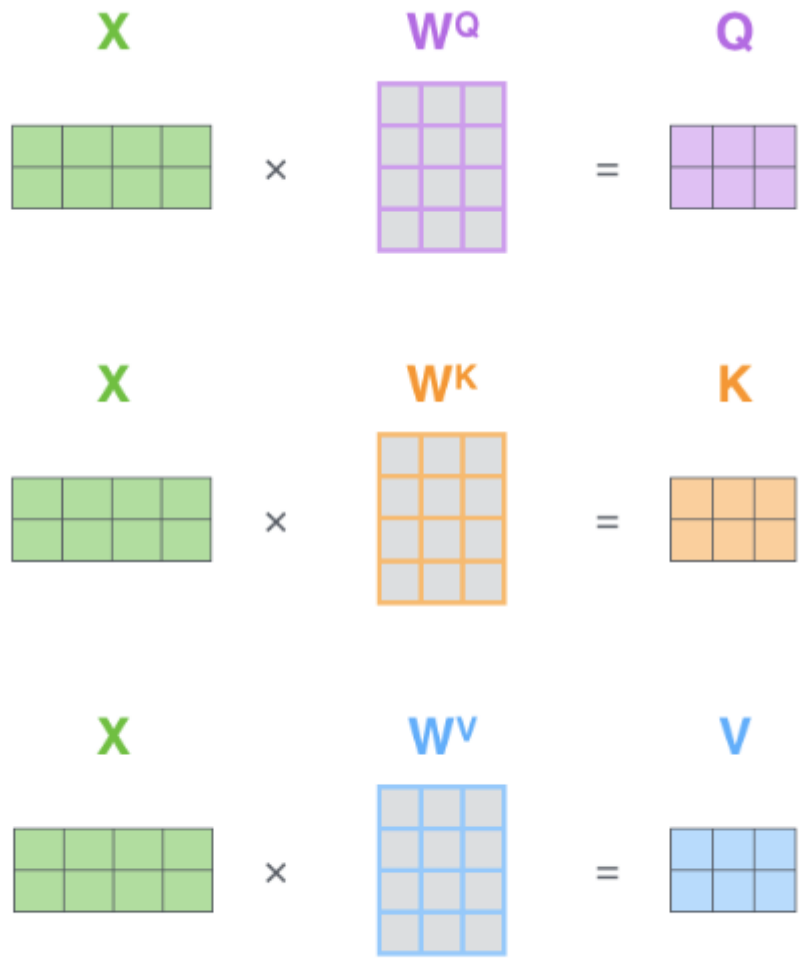
Transformers (Self attention)

- So, the output for the first word would be: $output_1 = softmax(score_1) \times V_1 + softmax(score_2) \times V_2 + softmax(score_3) \times V_3 + softmax(score_4) \times V_4 + softmax(score_5) \times V_5$
- That concludes the self-attention calculation.
- The resulting vector is one we can send along to the feed-forward neural network.
- In the actual implementation, however, this calculation is done in matrix form for faster processing.



Transformers (Matrix Calculation of Self-Attention)

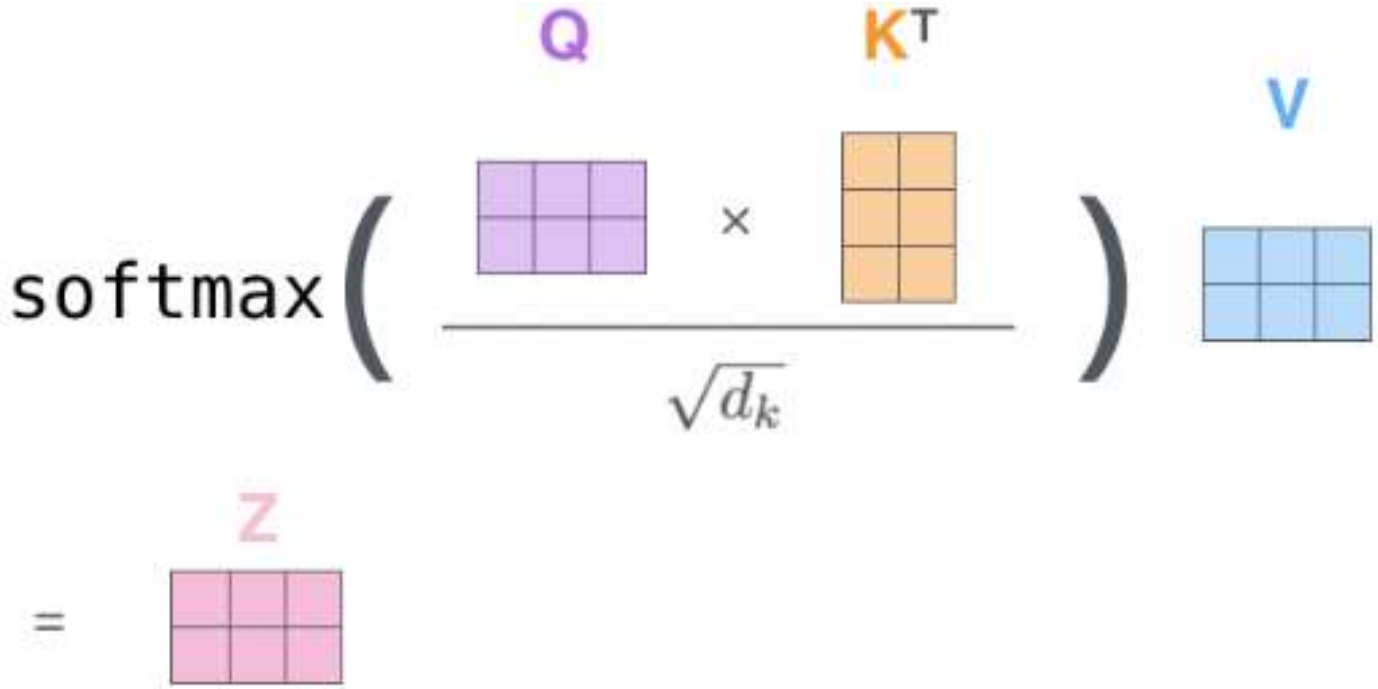
- The first step is to calculate the Query, Key, and Value matrices.
- We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V).



Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

Transformers (Matrix Calculation of Self-Attention)

- Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.



The self-attention calculation in matrix form

Transformers (Multi-headed attention)

- The paper further refined the self-attention layer by adding a mechanism called "**multi-headed**" attention. This improves the performance of the attention layer in two ways:

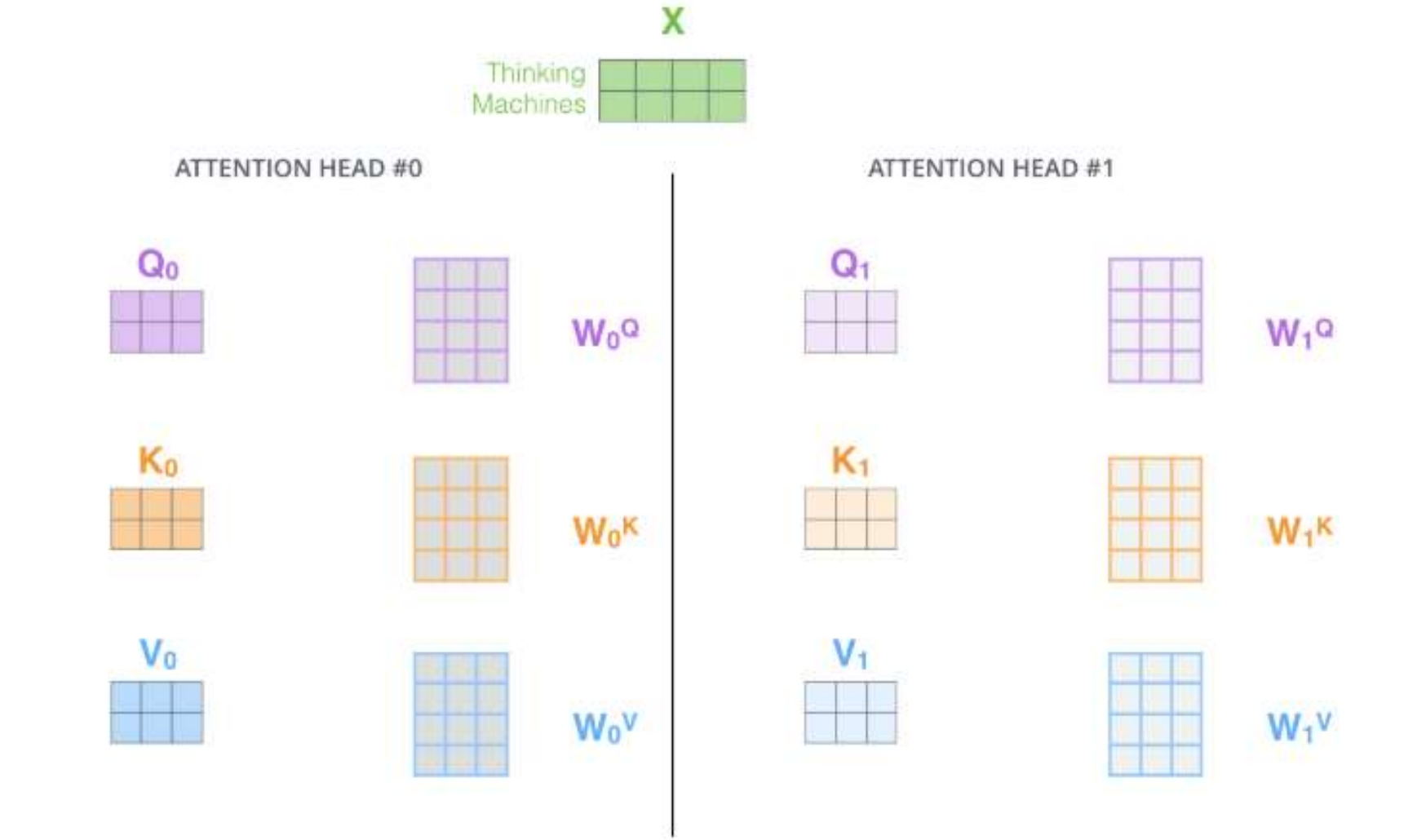
- It expands the model's ability to **focus on different positions**. Yes, in the example above, z_1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would be useful to know which word "it" refers to.

- It gives the attention layer **multiple "representation subspaces"**. As we'll see next, with multi-headed attention we have not only one, but **multiple sets of Query/Key/Value** weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

Transformers (Multi-headed attention)

- In the context of the Transformer model, multi-headed attention refers to the technique of **running the self-attention mechanism multiple times in parallel**, each with different parameterized queries (Q), keys (K), and values (V).
- These parallel attention mechanisms are called "**heads**," and **each one learns different aspects of the relationships** between words in the input sequence.
- Why multi-head attention:
  - The main reason for using **multiple attention heads** is to allow the model to focus on different parts of the input sequence simultaneously and capture a variety of relationships between the words. Each attention head can learn to focus on different "types" of dependencies:
    - **One head might focus on syntax**, capturing dependencies like subject-verb relationships.
    - **Another head might focus on semantics**, capturing more abstract or long-range dependencies between words.
  - Using multiple heads allows the model to capture these different aspects in parallel.

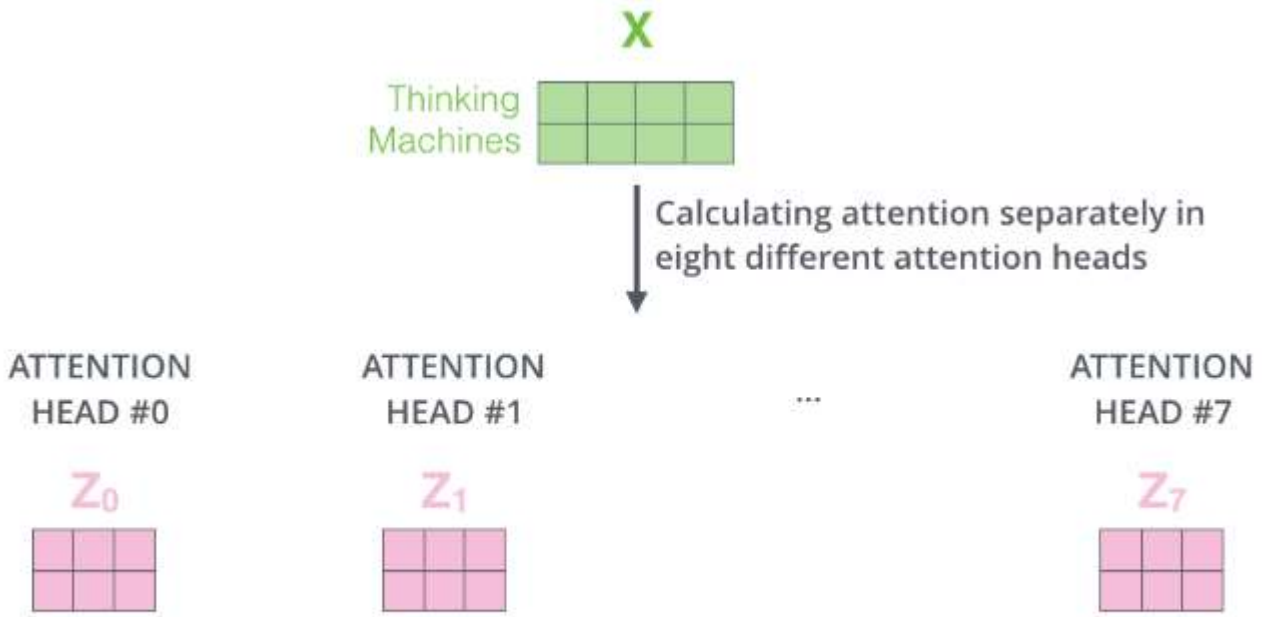
Transformers (Multi-headed attention)



With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

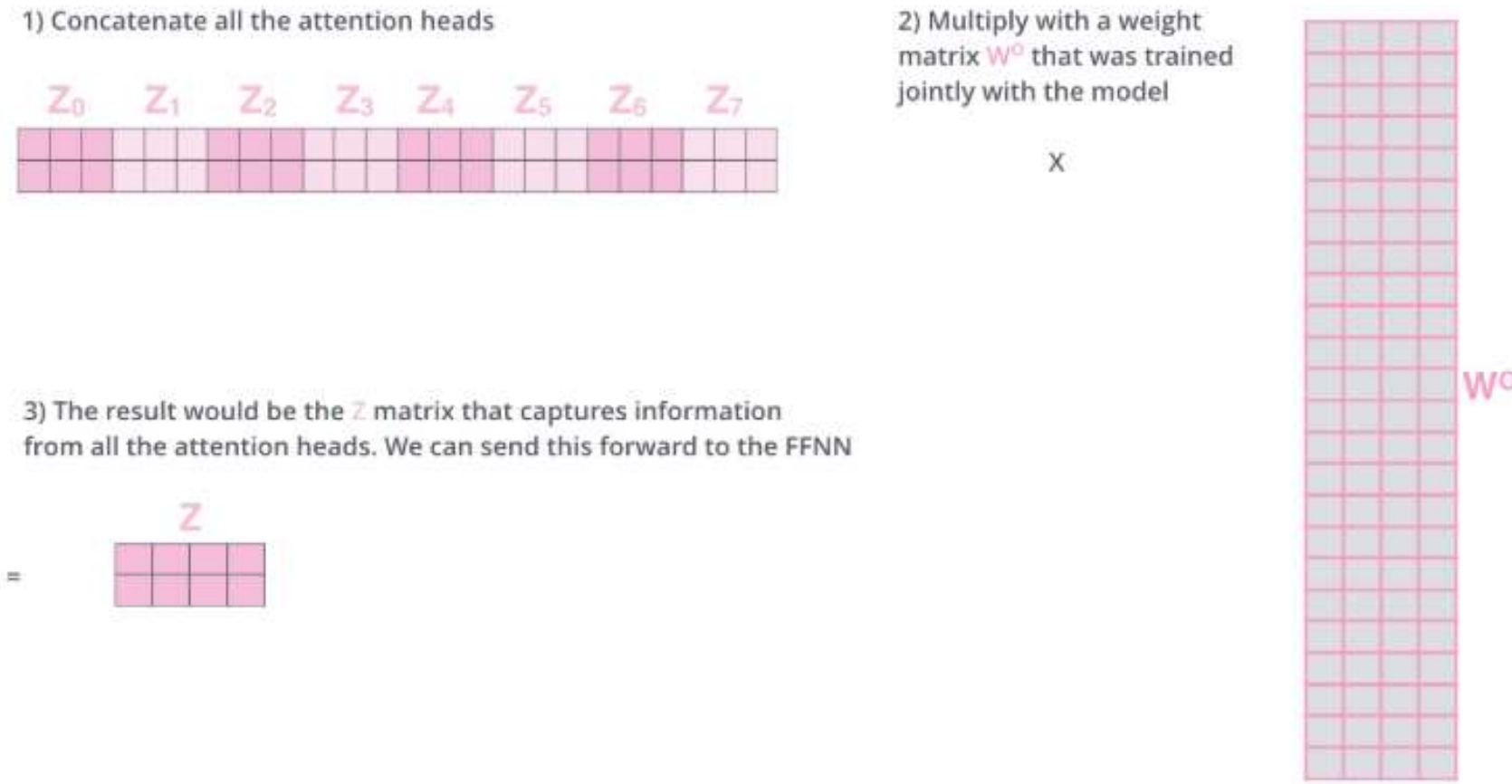
Transformers (Multi-headed attention)

- If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices



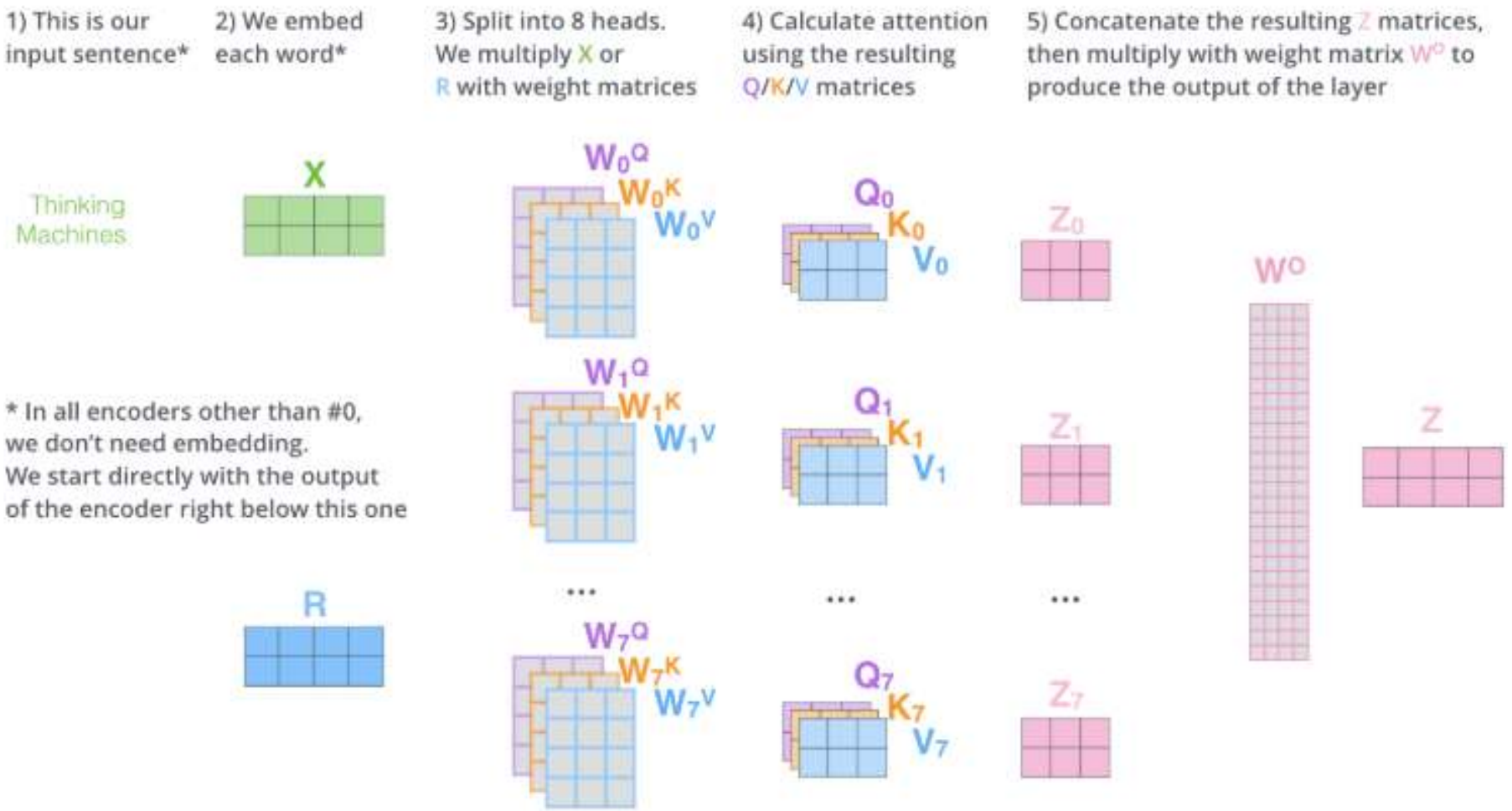
Transformers (Multi-headed attention)

- This leaves us with a bit of a challenge. The **feed-forward layer is not expecting eight matrices** – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.
- How do we do that? We concatenate the matrices then multiply them by an additional weights matrix W^O.
- The W^O matrix is used to project the concatenated vector back to the correct size, i.e., from the concatenated output (with size $8 \times d_{\text{head}}$) to the model's dimensionality d_{model}.
- This allows the output to be compatible with the next layer, which might be a feedforward network or another attention layer.



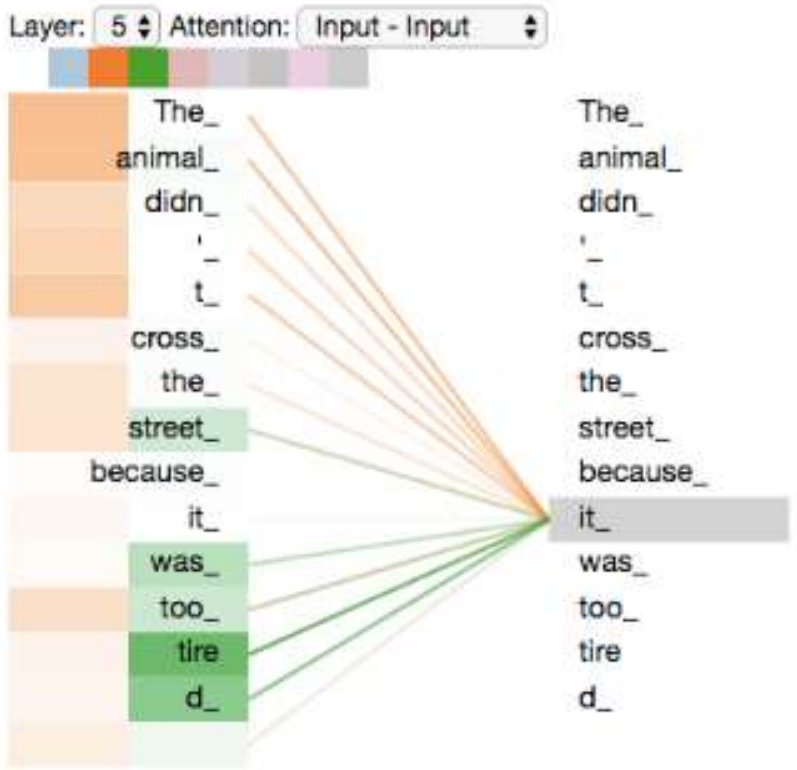
Transformers (Multi-headed attention)

- That's pretty much all there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place.



Transformers (Multi-headed attention)

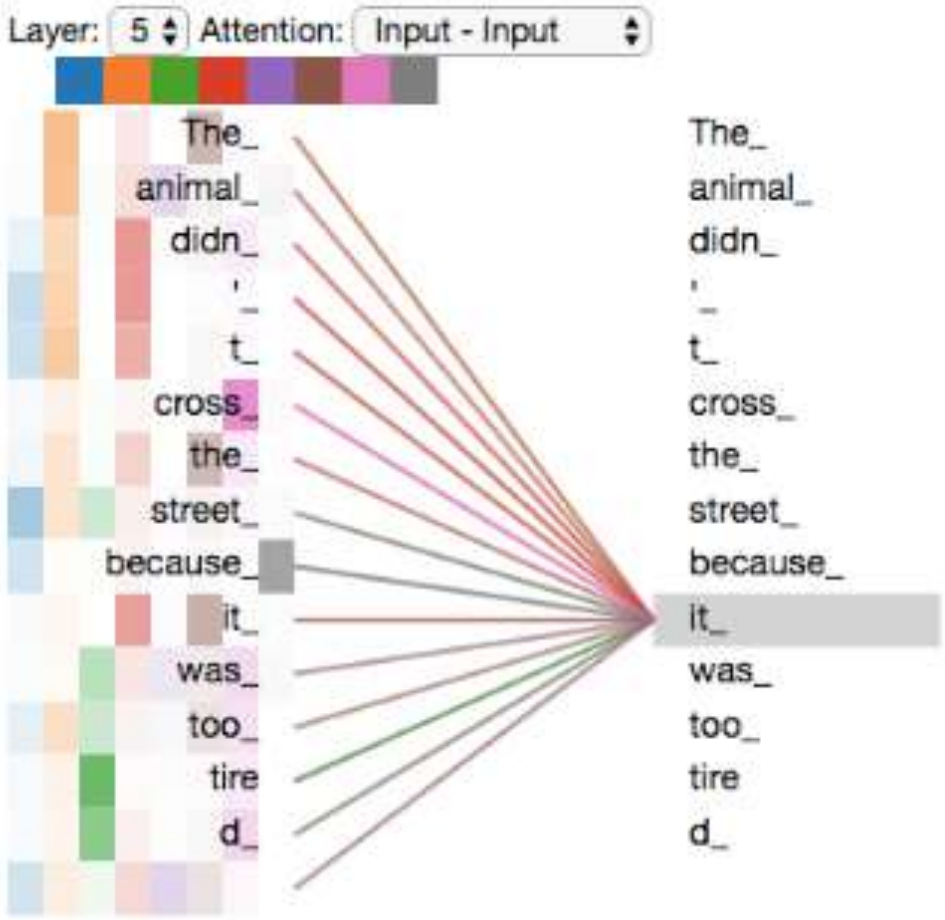
- Now that we have touched upon attention heads, let's revisit our example from before to see where the different attention heads are focusing as we encode the word "it" in our example sentence:



As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

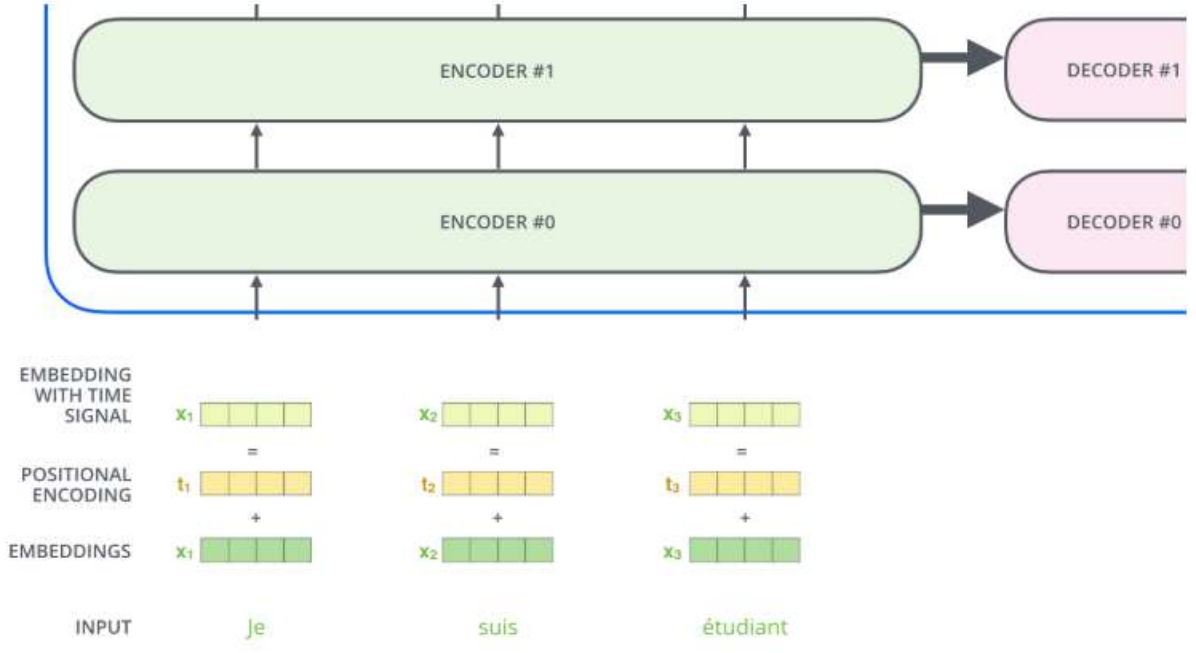
Transformers (Multi-headed attention)

- If we add all the attention heads to the picture, however, things can be harder to interpret:



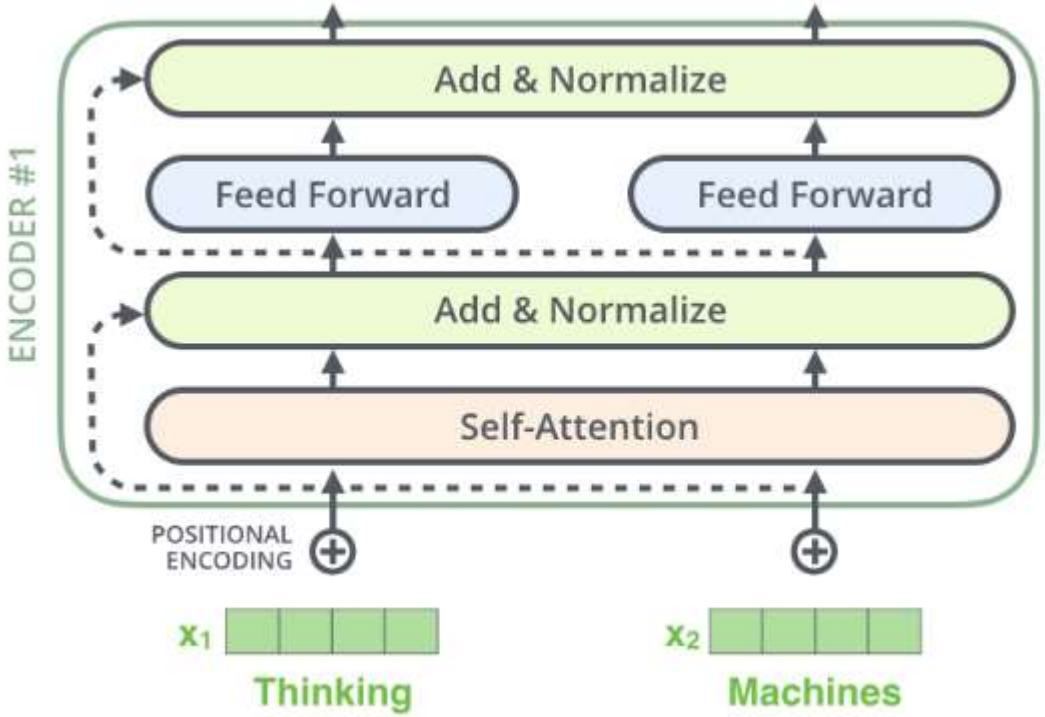
Transformers (Positional encoding)

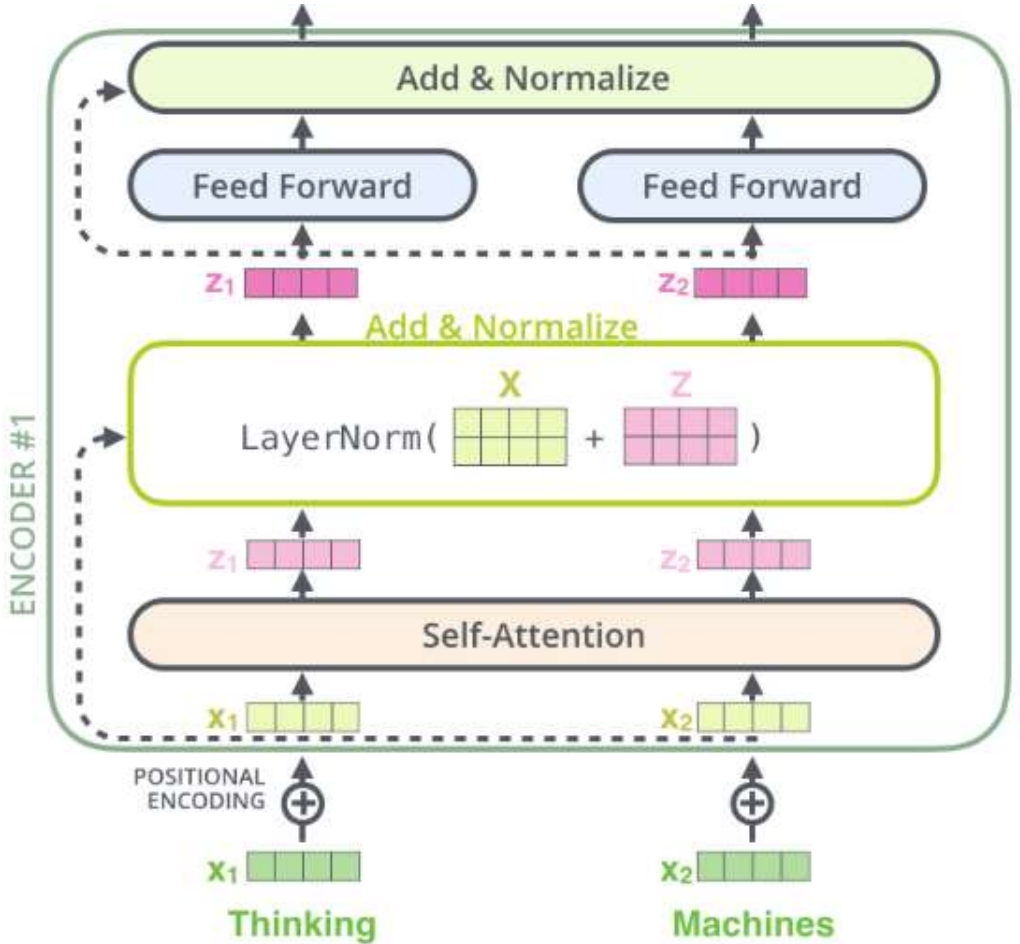
- One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.
- To address this, the transformer adds a vector to each input embedding.
- These vectors follow a specific pattern that the model learns, which helps it determine 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 once they're projected into Q/K/V vectors and during dot-product attention.



Transformers (The Residuals)

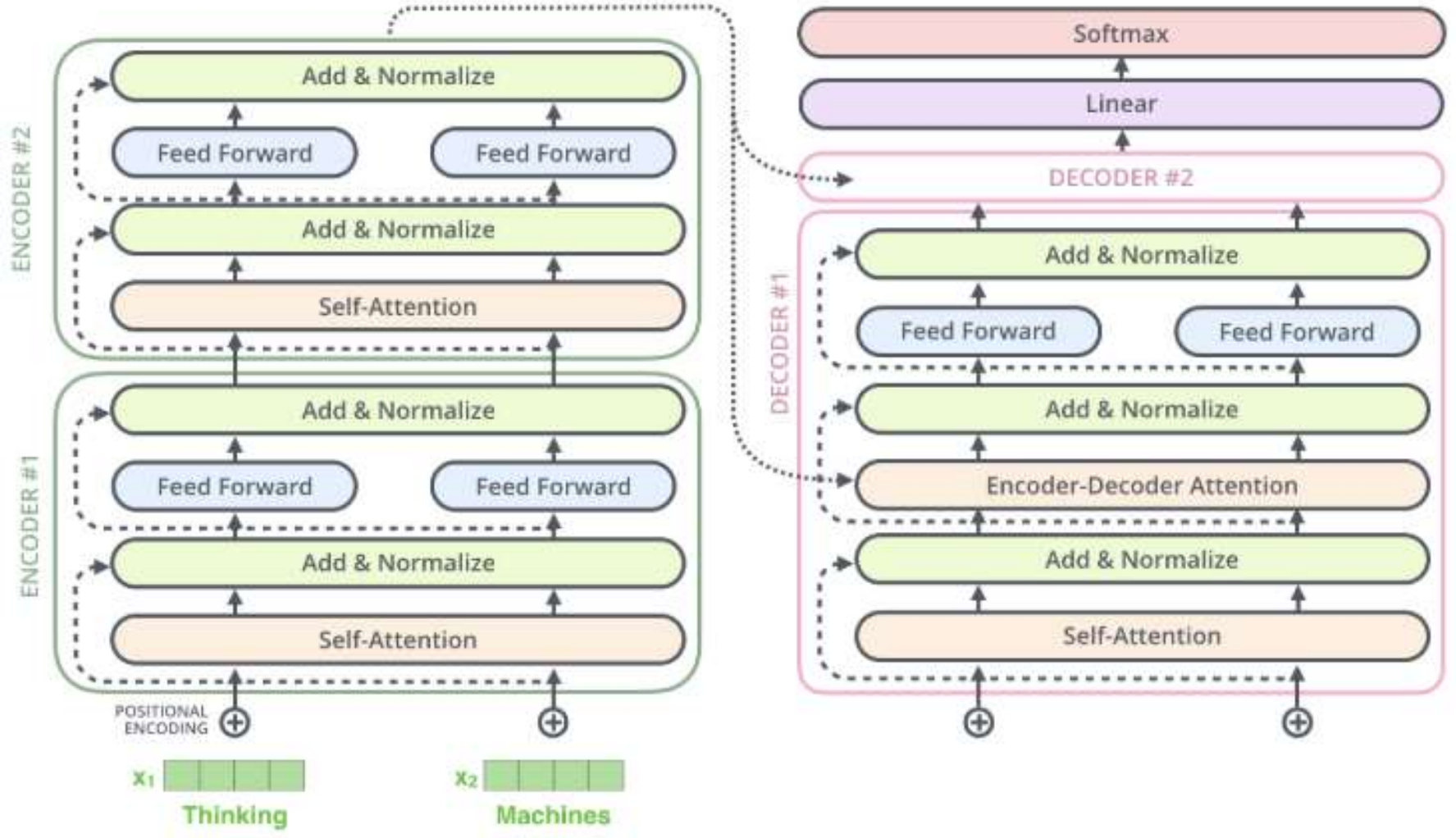
- One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.
- If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this (right side):





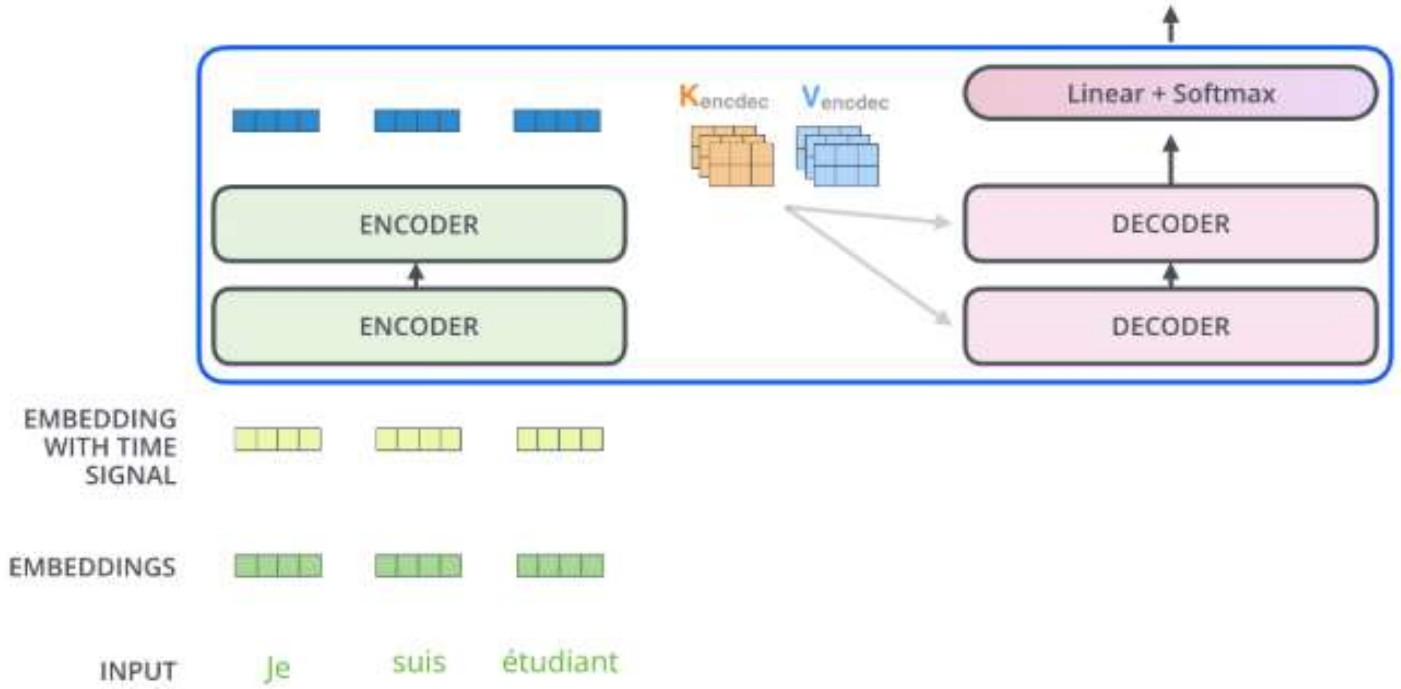
Transformers (The Residuals)

- This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:



Transformers (The 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:



- The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

Transformers (The decoder side)

- 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 decoder stack outputs a vector of floats. How do we turn that into a word? That’s the job of the final Linear layer which is followed by a Softmax Layer.
- The **Linear layer** is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a 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.

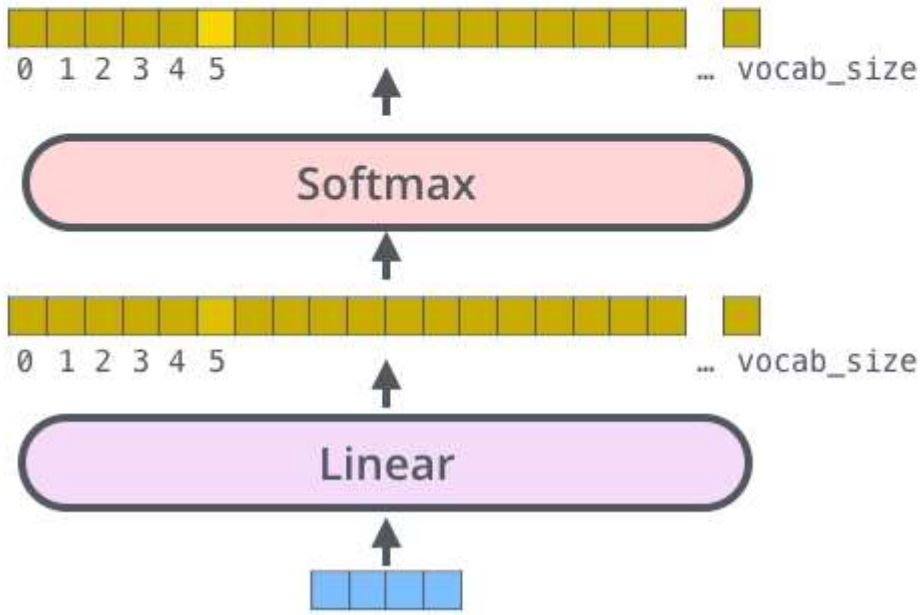
Transformers (The decoder side)

Which word in our vocabulary is associated with this index?

am

Get the index of the cell with the highest value (**argmax**)

5



Summary

- Introduced embedding layer for producing meaningful vector.
- Encoder, Decoder an important concept of seq2seq model
- Limitations of Encoder, Decoder are addressed in Attention model
- Transformer is the state-of-the-art architecture produced to overcome the limitations of Attention model and considered as latest NLP technique.