Natural Language Processing (NLP)

Transformers Part 01

Self Attentions, Multi Headed Attentions and Positional Encoding

By:

Dr. Zohair Ahmed

www.youtube.com/@ZohairAI Subscribe

www.begindiscovery.com

Transformer

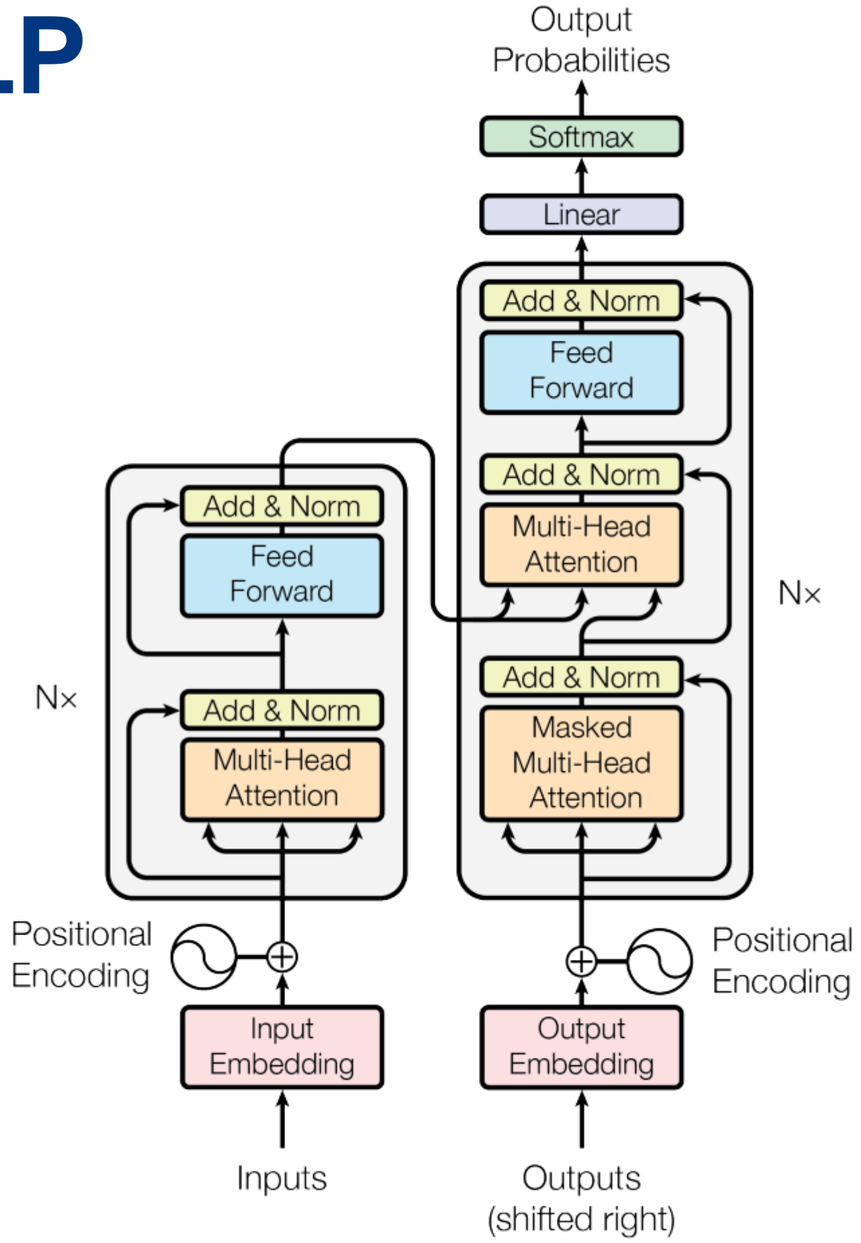
- In **Natural Language Processing (NLP)**, a **Transformer** is a deep learning architecture introduced in the paper *"Attention Is All You Need"* (Vaswani et al., 2017).
- https://papers.neurips.cc/paper/7181-attention-is-all-you-need.pdf
- It revolutionized NLP by replacing recurrent and convolutional models with a mechanism called **self-attention**, enabling models to process sequences in parallel rather than sequentially.

Key Components of a Transformer

- **Input Embeddings**
  Converts tokens into dense vectors, often combined with **positional encoding** to retain word order.
- **Self-Attention Mechanism**
  - Computes relationships between all tokens in a sequence.
  - Uses **Query (Q)**, **Key (K)**, and **Value (V)** matrices to calculate attention scores.
  - Allows the model to focus on relevant words regardless of their position.
- **Multi-Head Attention**
  Multiple attention layers run in parallel to capture different types of relationships.
- **Feed-Forward Network**
  A fully connected layer applied to each token independently after attention.
- **Residual Connections & Layer Normalization**
  Helps stabilize training and improve gradient flow.
- **Encoder-Decoder Structure**
  - **Encoder**: Processes input sequence into contextual representations.
  - **Decoder**: Generates output sequence using encoder outputs and previous predictions.

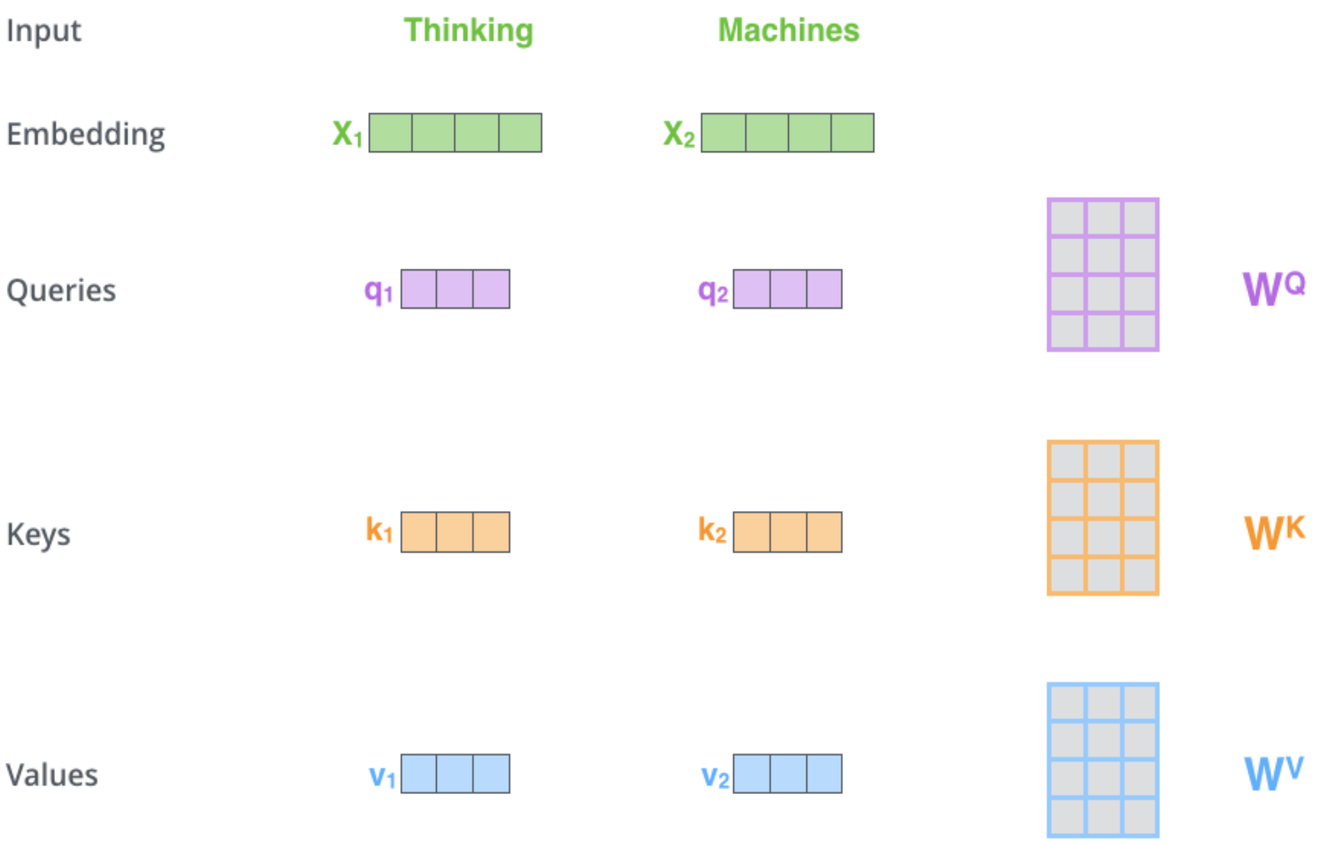
Why Transformers Changed NLP

- **Parallelization**: Unlike RNNs, Transformers process all tokens simultaneously.
- **Long-Range Dependencies**: Self-attention captures relationships across distant words effectively.
- **Scalability**: Enabled large models like **BERT**, **GPT**, **T5**, and **LLaMA**.



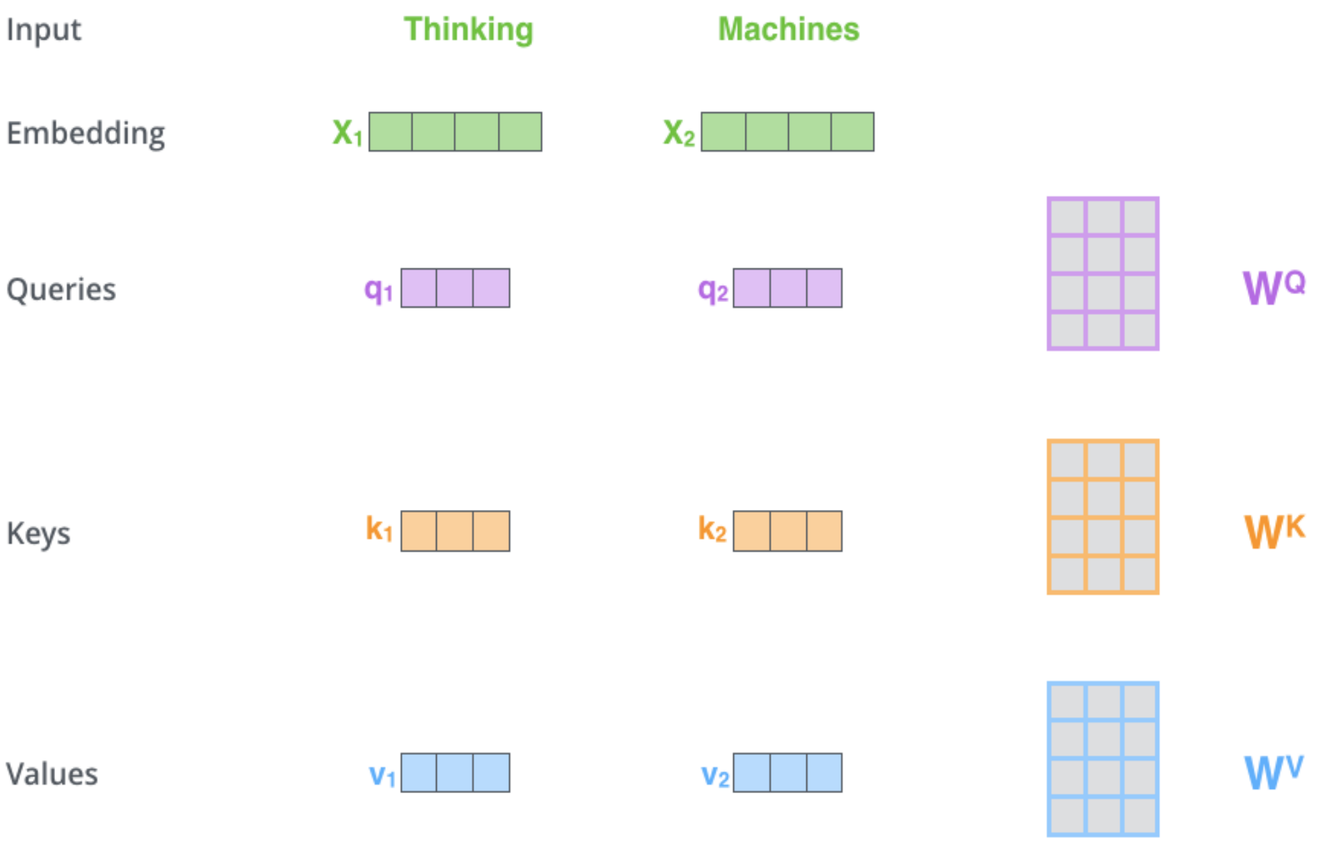
Self-Attention in Detail

- 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.



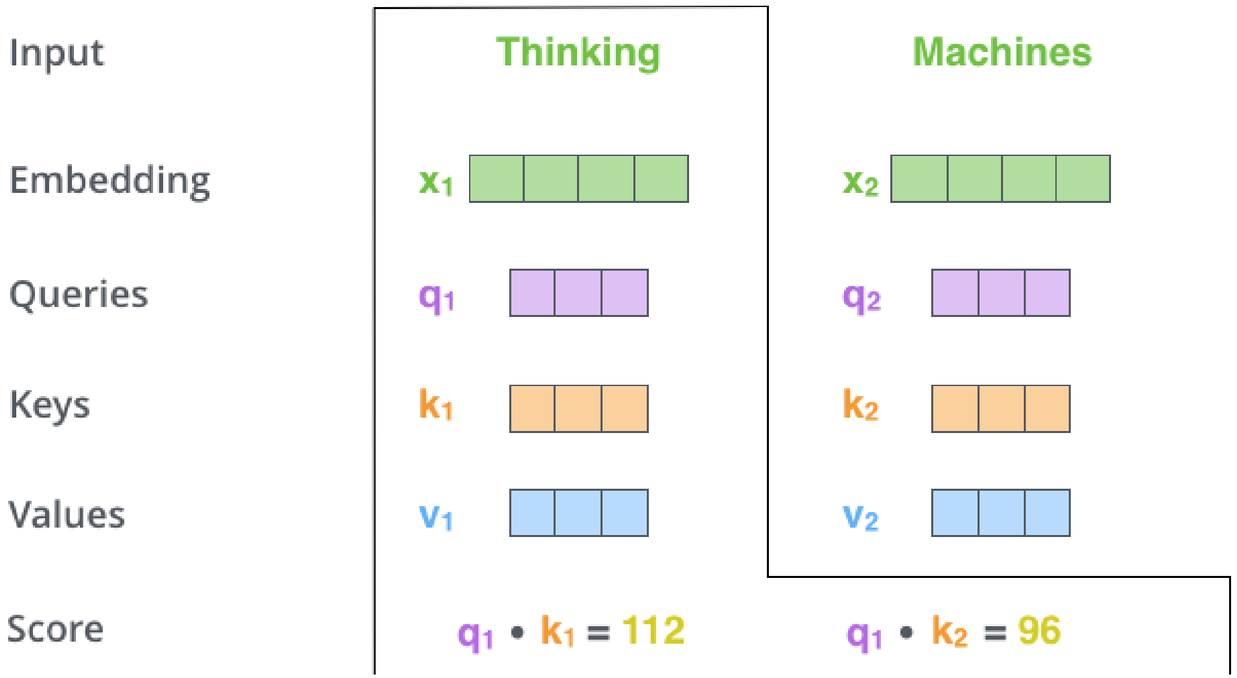
Step 1 - Self-Attention in Detail

- 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.



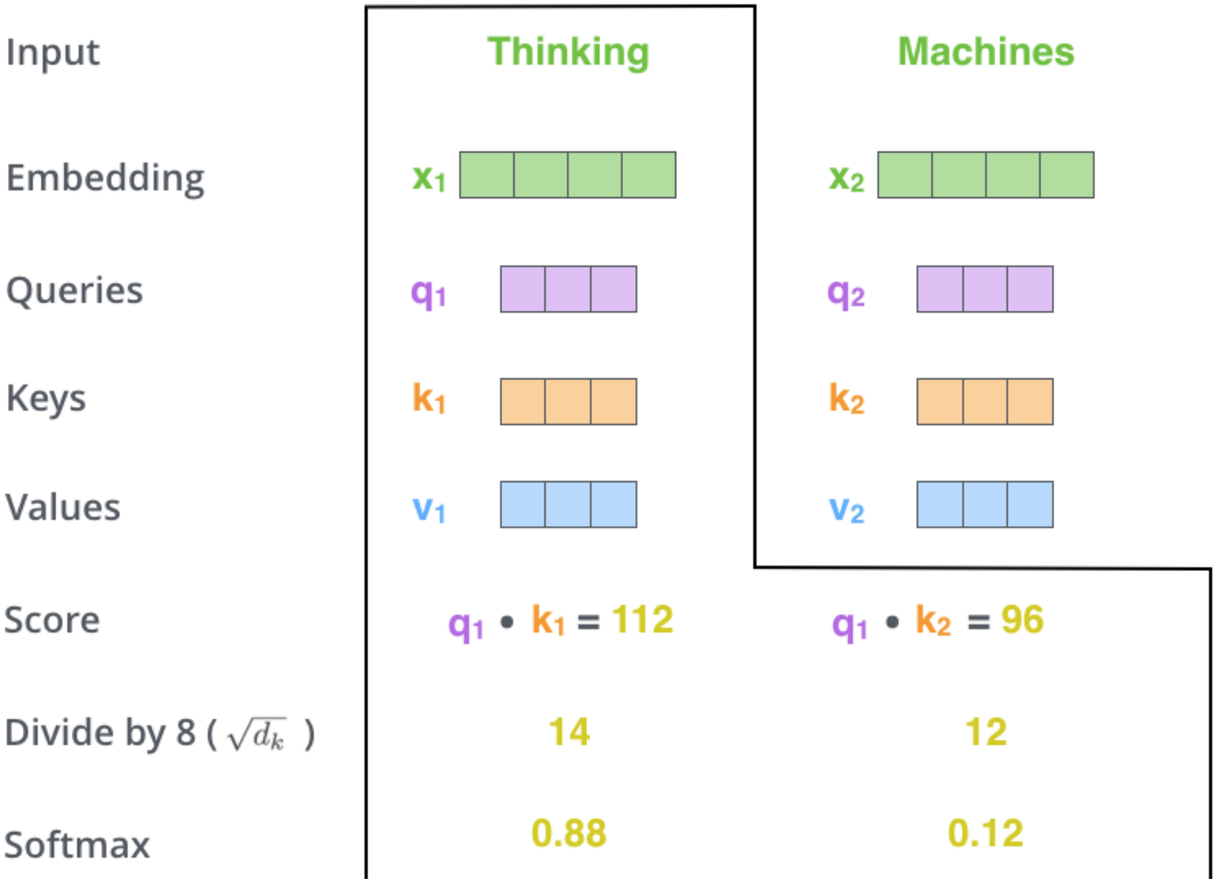
Step 2 - Self-Attention in Detail

- The **second step** in calculating self-attention is to calculat a score.
- Say we're calculating the self-attention for the first word in this example, "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 worc 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.



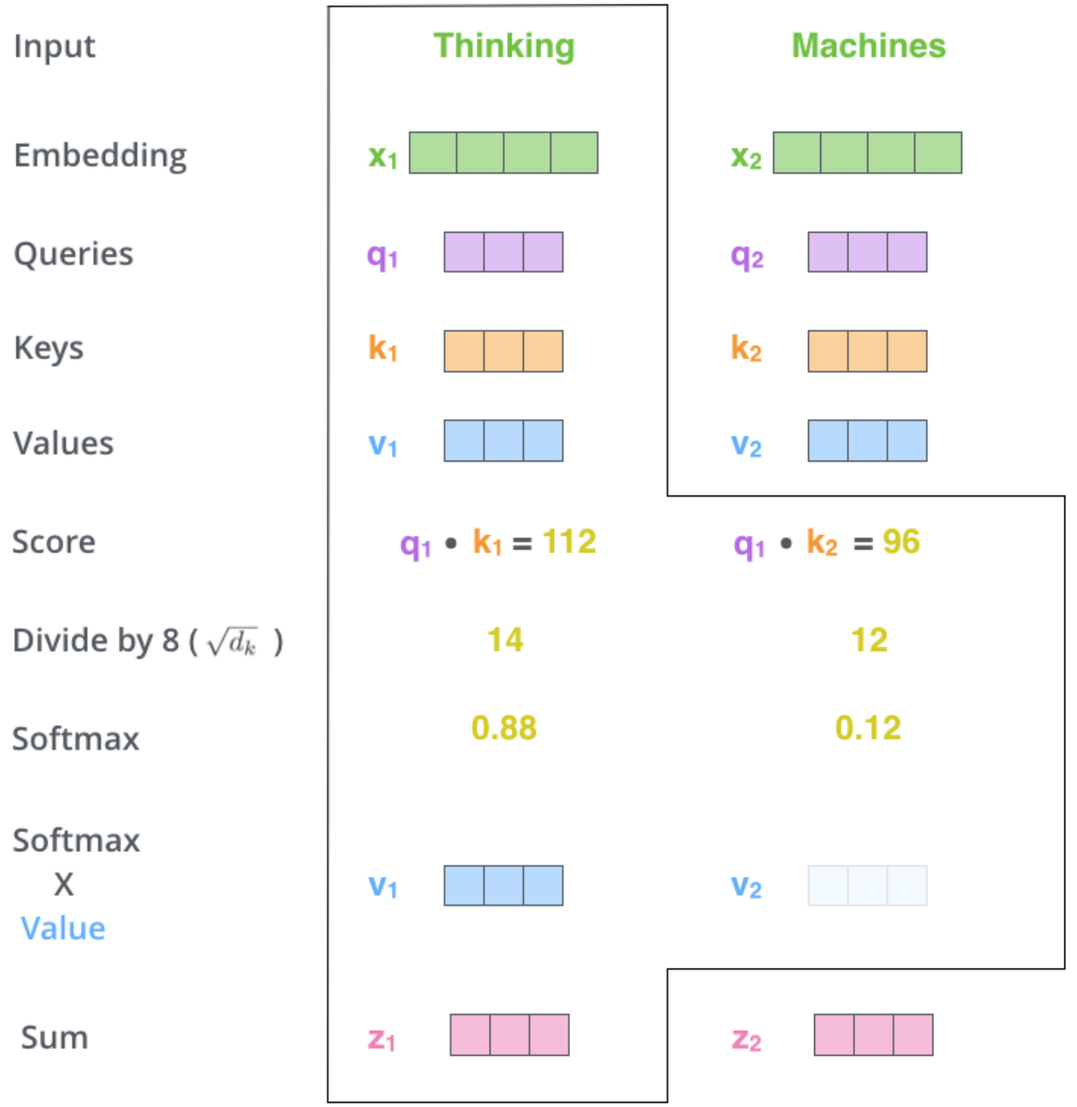
Step 3,4 - Self-Attention in Detail

- The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used 64.
- This leads to having more stable gradients.
- 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.
- 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.



Step 5,6 - Self-Attention in Detail

- 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).



Step 5,6 - Self-Attention in Detail

- 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.
- So let’s look at that now that we’ve seen the intuition of the calculation on the word level.

What Achieves- Self-Attention in Detail

- **Attention Score Matrix (before softmax):**
- Scores=QK^T
- Each row = one query word comparing itself to all keys.
- Example for 3 words:
- **Row 1 =** how much “The” attends to The, Cat, Sat.
- **Row 2 =** how much “Cat” attends to The, Cat, Sat.
- **Row 3 =** how much “Sat” attends to The, Cat, Sat.

$$\begin{bmatrix} \text{The} \rightarrow [\text{The, Cat, Sat}] \\ \text{Cat} \rightarrow [\text{The, Cat, Sat}] \\ \text{Sat} \rightarrow [\text{The, Cat, Sat}] \end{bmatrix}$$

What Achieves- Self-Attention in Detail

- **After softmax → Attention Weights**
- **Multiply by V → Output=α×V**
- Each row of Output = weighted sum of all V rows.
- So, Row 1 (The) = $0.3 \times V_1 + 0.4 \times V_2 + 0.3 \times V_3$.
- That's why "The" now has values from Cat and Sat.
- So when you see 1.2797 in Row 1, second dimension, it's (Sat) have strong values in that dimension.
- Attention weights gave them importance.
- So the second dimension in Row 1 is context from other words.

$$\begin{bmatrix} \text{The} \rightarrow [\text{The, Cat, Sat}] \\ \text{Cat} \rightarrow [\text{The, Cat, Sat}] \\ \text{Sat} \rightarrow [\text{The, Cat, Sat}] \end{bmatrix}$$

$$\alpha = \begin{bmatrix} 0.3 & 0.4 & 0.3 \\ 0.2 & 0.5 & 0.3 \\ 0.4 & 0.3 & 0.3 \end{bmatrix}$$

What Achieves- Self-Attention in Detail

- **Original V (Value) Matrix:** Each row is the original representation of a word.
- **After Attention:** Each row is **a weighted sum of all three V rows**, where weights come from attention scores.

$$V = \begin{bmatrix} 1.0 & 0.0 & 0.5 \\ (\text{The}) & & \\ 0.0 & 1.0 & 0.3 \\ (\text{Cat}) & & \\ 1.0 & 1.0 & 0.2(\text{Sat}) \end{bmatrix}$$

$$\text{Output} = \begin{bmatrix} 1.3869 & 1.2797 & 0.3914 \\ (\text{new The}) & & \\ 1.4006 & 1.1692 & 0.3911 \\ (\text{new Cat}) & & \\ 1.4296 & 1.2905 & 0.4074(\text{new Sat}) \end{bmatrix}$$

What Achieves- Self-Attention in Detail

- **Look at row 1 (The):** Original was **[1.0, 0.0, 0.5].**
- Now it has non-zero values in all positions, especially the second dimension (1.2797), which originally belonged to “Cat”.
- This means **“The”** borrowed context from **“Cat”** and **“Sat”**.
- **Row 2 (Cat):**Original was **[0.0, 1.0, 0.3].**
- Now it has ~1.4 in the first dimension (from “The”) and ~1.29 in the second (from “Sat”).
- “Cat” now knows about “The” and “Sat”.
- **Row 3 (Sat):** Original was **[1.0, 1.0, 0.2].**
- Now it’s even more blended, showing influence from both “The” and “Cat”.

What Achieves- Self-Attention in Detail

- **Intuition:**
- Before attention: each word = isolated meaning.
- After attention: each word = **context-aware meaning**, because it has integrated information from other words based on relevance.

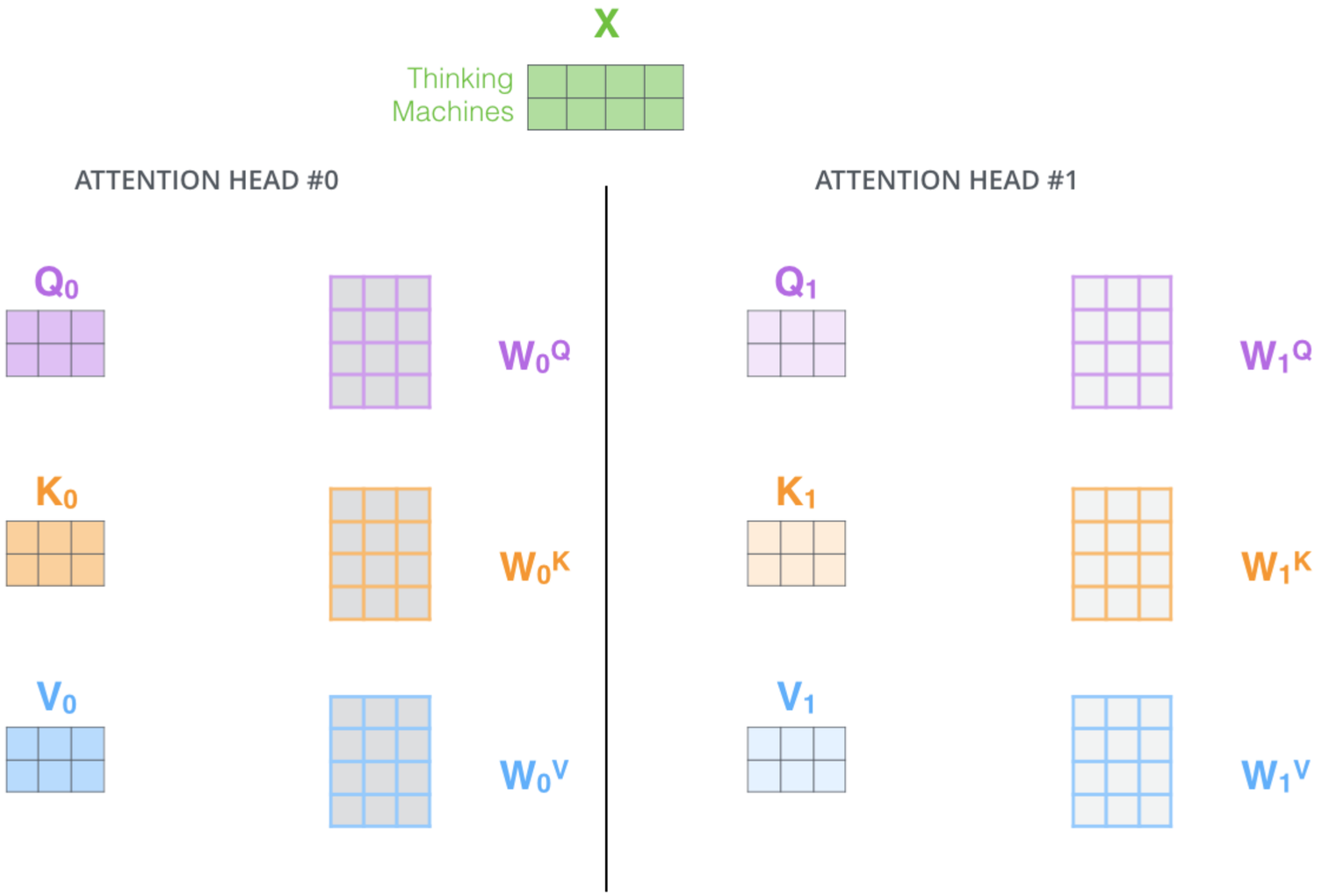
	The	Cat	Sat	On	The	Mat
The	high	medium	medium	low	high	low
Cat	medium	high	high	medium	low	high
Sat	low	high	high	medium	low	high
On	high	low	high	high	low	high
The	high	high	medium	low	low	low
Mat	low	medium	high	medium	low	high

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, z1 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”.
- Each head looks at the same words but in a different subspace because the weight matrices are different.
- **Example: Head 1** might focus on syntactic relationships (e.g., subject → verb).
- **Head 2** might focus on semantic similarity (e.g., synonyms).
- **Head 3** might focus on position or context.

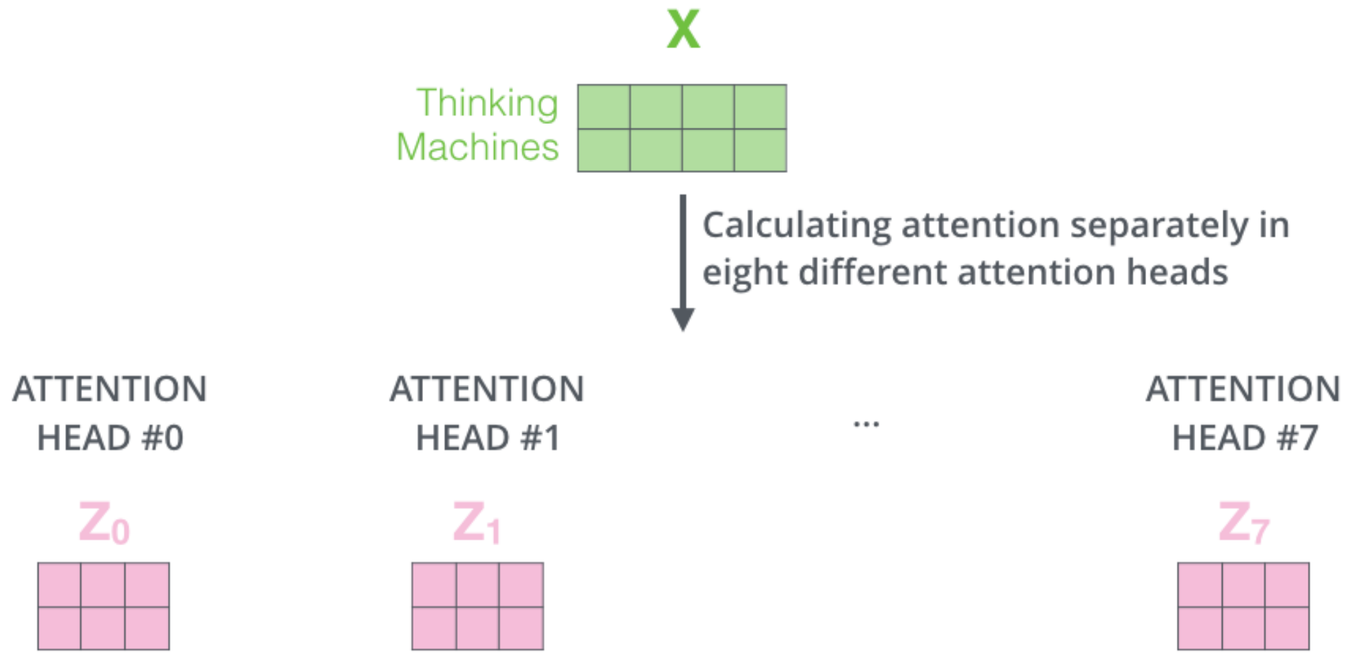
Multi-Headed Attention

- 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.



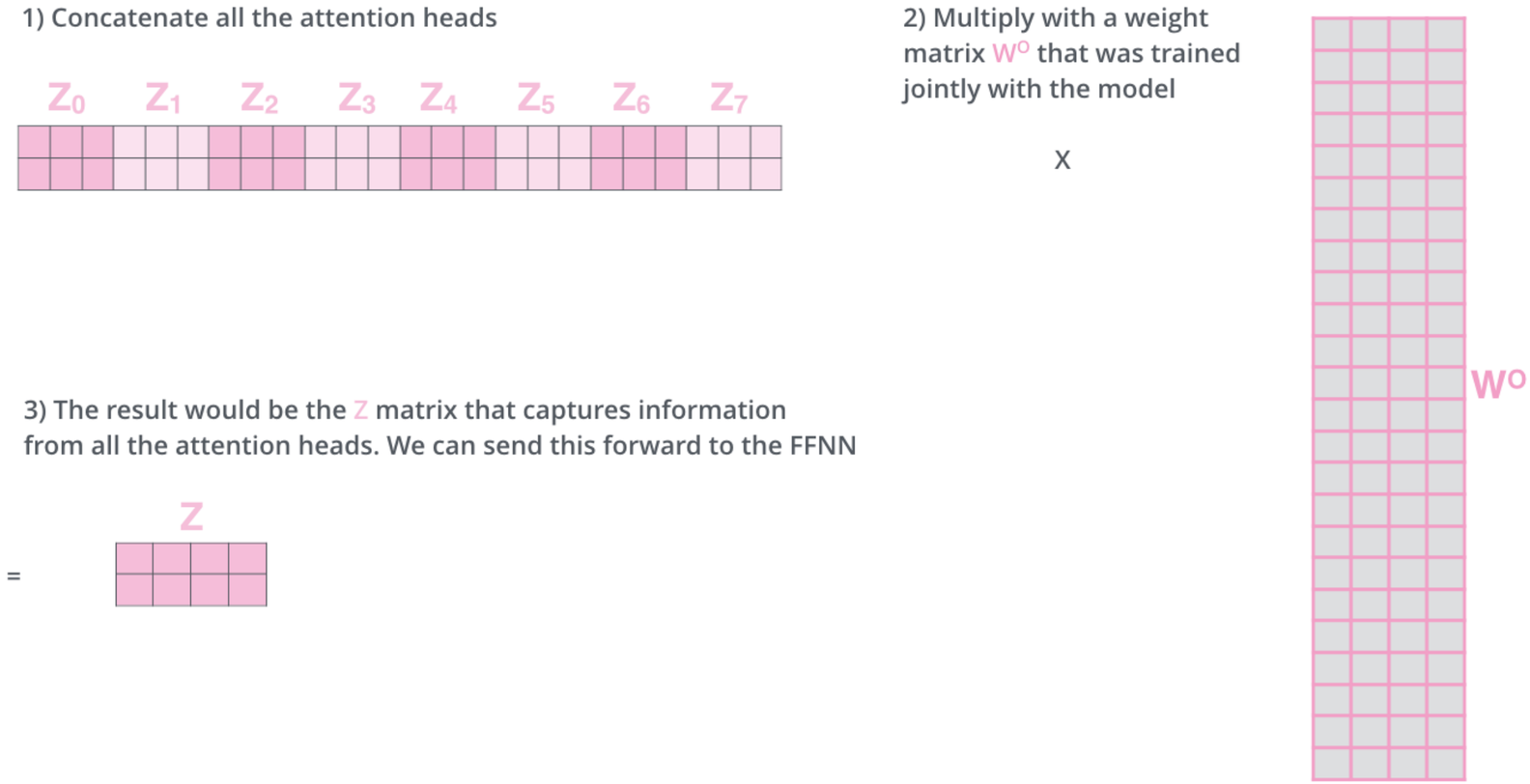
Multi-Headed Attention

- 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



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 concat the matrices then multiply them by an additional weights matrix WO.



Multi-Headed Attention

1) This is our input sentence*

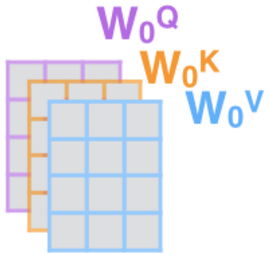
2) We embed each word*

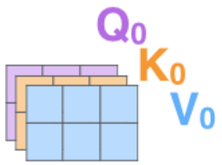
3) Split into 8 heads. We multiply X or R with weight matrices

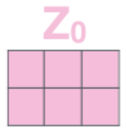
4) Calculate attention using the resulting Q/K/V matrices

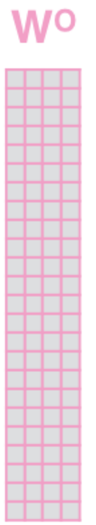
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

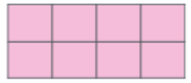
Thinking Machines



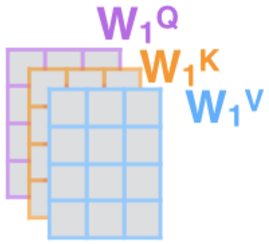


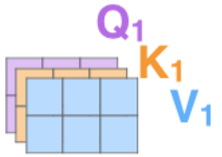


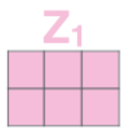




* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



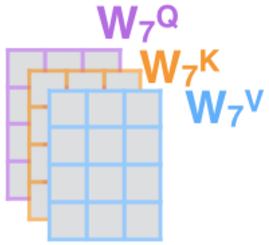


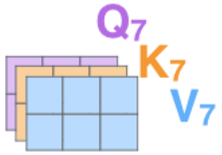


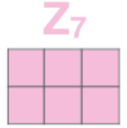
...

...

...

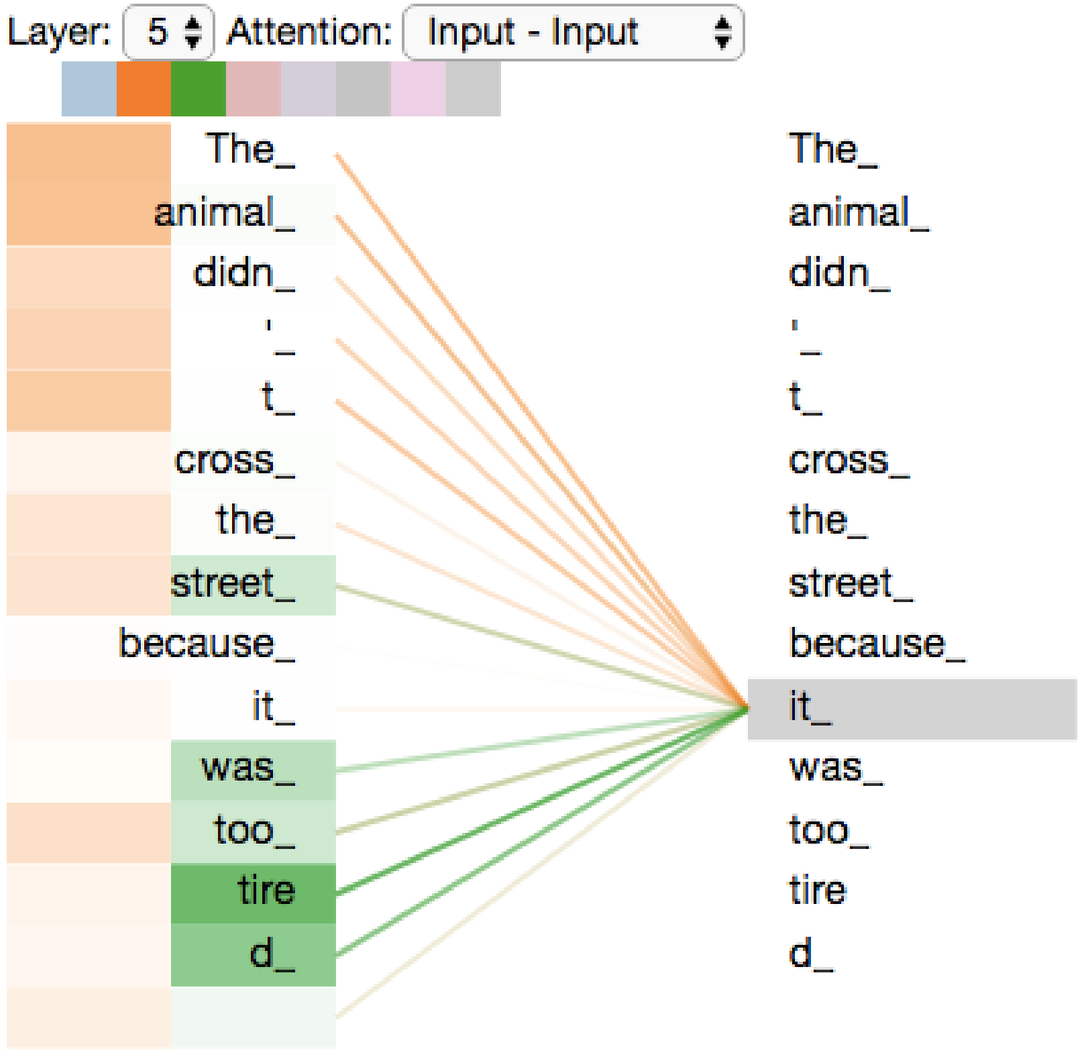






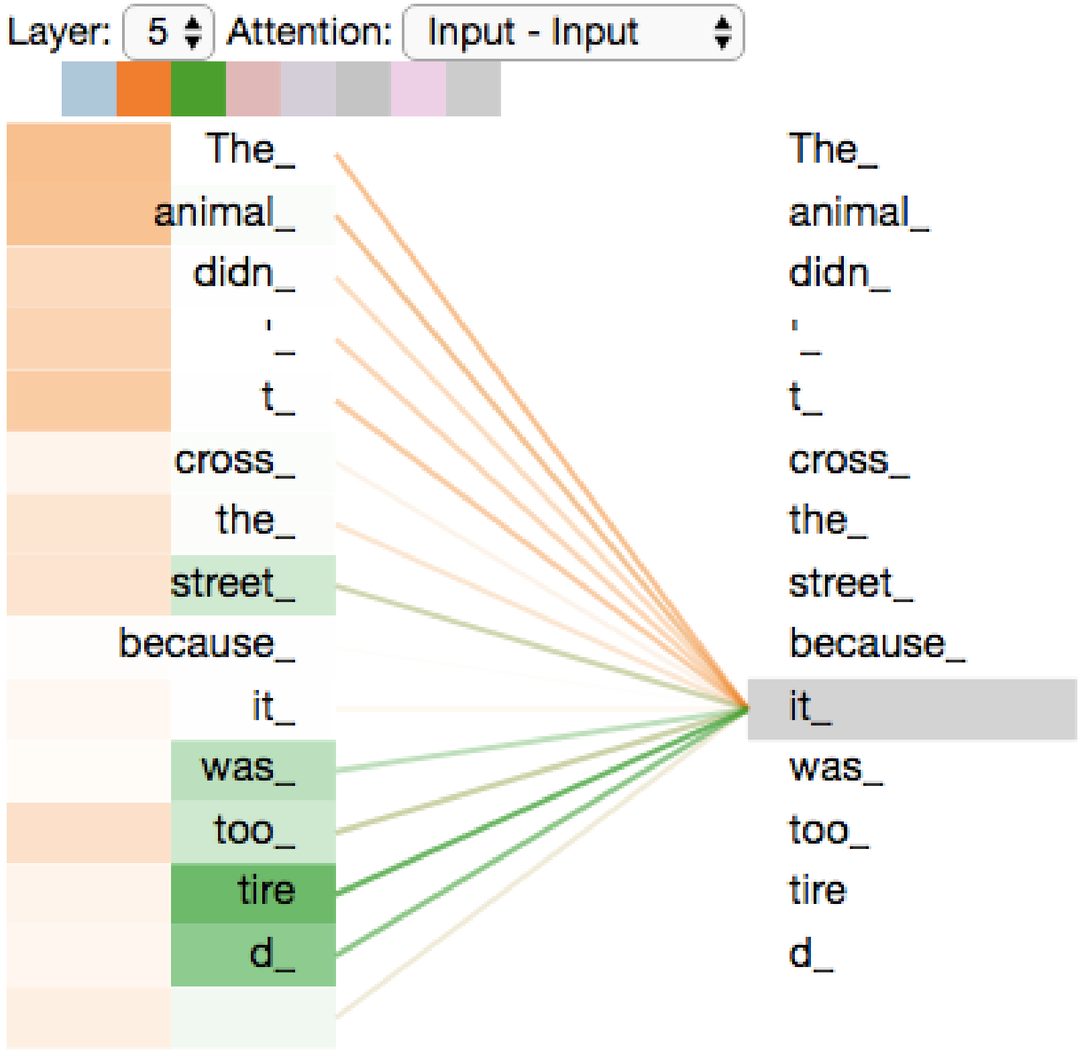
Multi-Headed Attention

- 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:



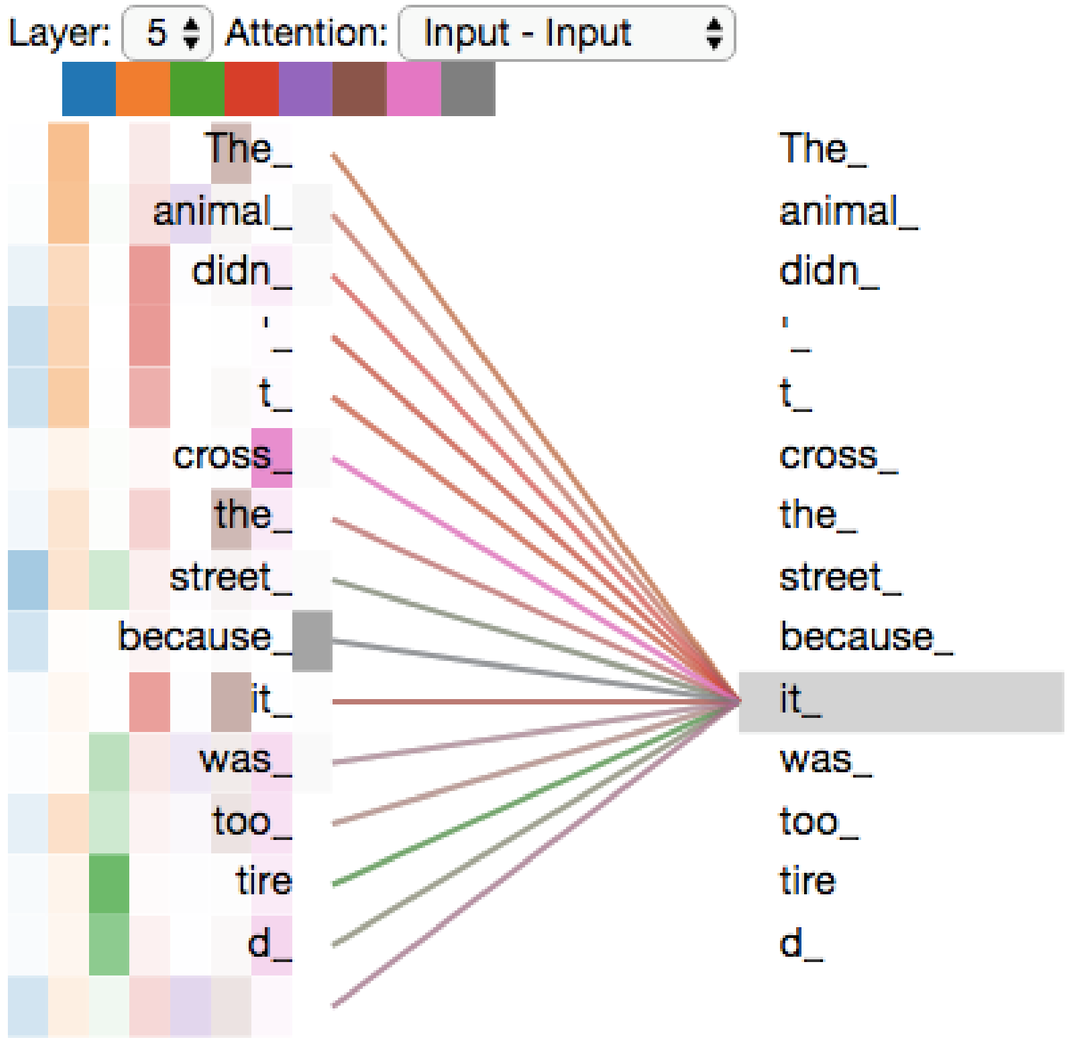
Multi-Headed Attention

- 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".



Multi-Headed Attention

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



Why Do We Need Positional Encoding?

- Positional encoding is a technique that adds information about the position of each token in the sequence to the input embeddings.
- This helps transformers to understand the relative or absolute position of tokens which is important for differentiating between words in different positions and capturing the structure of a sentence.
- Without positional encoding, transformers would struggle to process sequential data effectively.

What is Masked Self-Attention?

- In a **transformer decoder** (like for language generation), *masked self-attention* ensures that the model **cannot "peek ahead" at future words**.
- At each step, when predicting the next word, the model is only allowed to pay attention to the current and previous words, **not words that come after it**.
- **Why?**

  If the model could see all positions, it would "cheat" by seeing the answer it is supposed to predict.

How Does Masked Self-Attention Work?

- Let’s break it down with an example.
- **Example Sentence:**
- Suppose the model is generating The cat sat
- Position 1: “The”
- Position 2: “cat”
- Position 3: “sat”
- At position 2 (“cat”), the model should only see “The” and “cat”, **not “sat”**.

Step 1: Attention Score Matrix (before masking)

- Without masking, every word can attend to every other (including the future):

  For three words, an attention score matrix would look like:

	The	**cat**	**sat**
The	s11	s12	s13
cat	s21	s22	s23
sat	s31	s32	s33

Step 2: Create Mask

- The **mask** makes sure each word can only attend to itself and words before it (not after).
- For position 1 (“The”) → only itself
- For position 2 (“cat”) → “The” and “cat”, not “sat”
- For position 3 (“sat”) → all three
- **The mask matrix (for 3 words):**
- 0 means allow attending
- -∞ (a very large negative number) means block (softmax will turn it into 0 probability)

	The	cat	sat
The	0	-∞	-∞
cat	0	0	-∞
sat	0	0	0

Step 2: Create Mask

- The **mask** makes sure each word can only attend to itself and words before it (not after).
- For position 1 (“The”) → only itself
- For position 2 (“cat”) → “The” and “cat”, not “sat”
- For position 3 (“sat”) → all three
- **The mask matrix (for 3 words):**
- 0 means allow attending
- -∞ (a very large negative number) means block (softmax will turn it into 0 probability)

	The	**cat**	**sat**
The	0	-∞	-∞
cat	0	0	-∞
sat	0	0	0

Step 3: Apply Mask to Attention Scores

- Before softmax, **add the mask**: all positions after the current become -∞
- After softmax, those entries become zero probability (cannot focus there).
- **Step 4: Calculate Output as Usual**
- The model produces output for each position, but attention at each position can only incorporate **past and current words**.
- **Why?**
- During *training*, it forces the decoder to act like it is generating one word at a time, so it can't "cheat".
- During *inference* (actual generation), the situation is naturally masked — you don't know the future anyway.

Why Masking Need

- **1. Unmasked Training (Cheating Scenario)**
- **Training Phase:**
- **Input sequence:** The sun sets and the sky turns red
- The self-attention has **no mask**. When the model is predicting the blank ("red"), it **can see “red” already**.
- It attends to *all* tokens in the sequence, including the token "red".
- **What happens?**
- The model learns a shortcut:
  "When I need to predict the next word, just look ahead
- I don’t need to understand, I just copy!"
- It achieves **100% accuracy** in training, because it can always "see" the right answer!
- No real pattern is learned.

- **2. Generation (Real Use):**
- **Prompt: The sun sets and the sky turns**
- Now you ask the model to **generate what comes next**.
- *This time*, there is **no answer word present** (no "red" in the input).
- The self-attention tries to look ahead, but... **there’s nothing to look at**!
- **What happens?**
- The model is LOST—it never learned how to guess from context alone.
- It might generate a blank, a meaningless word, or even repeat "The sun...".

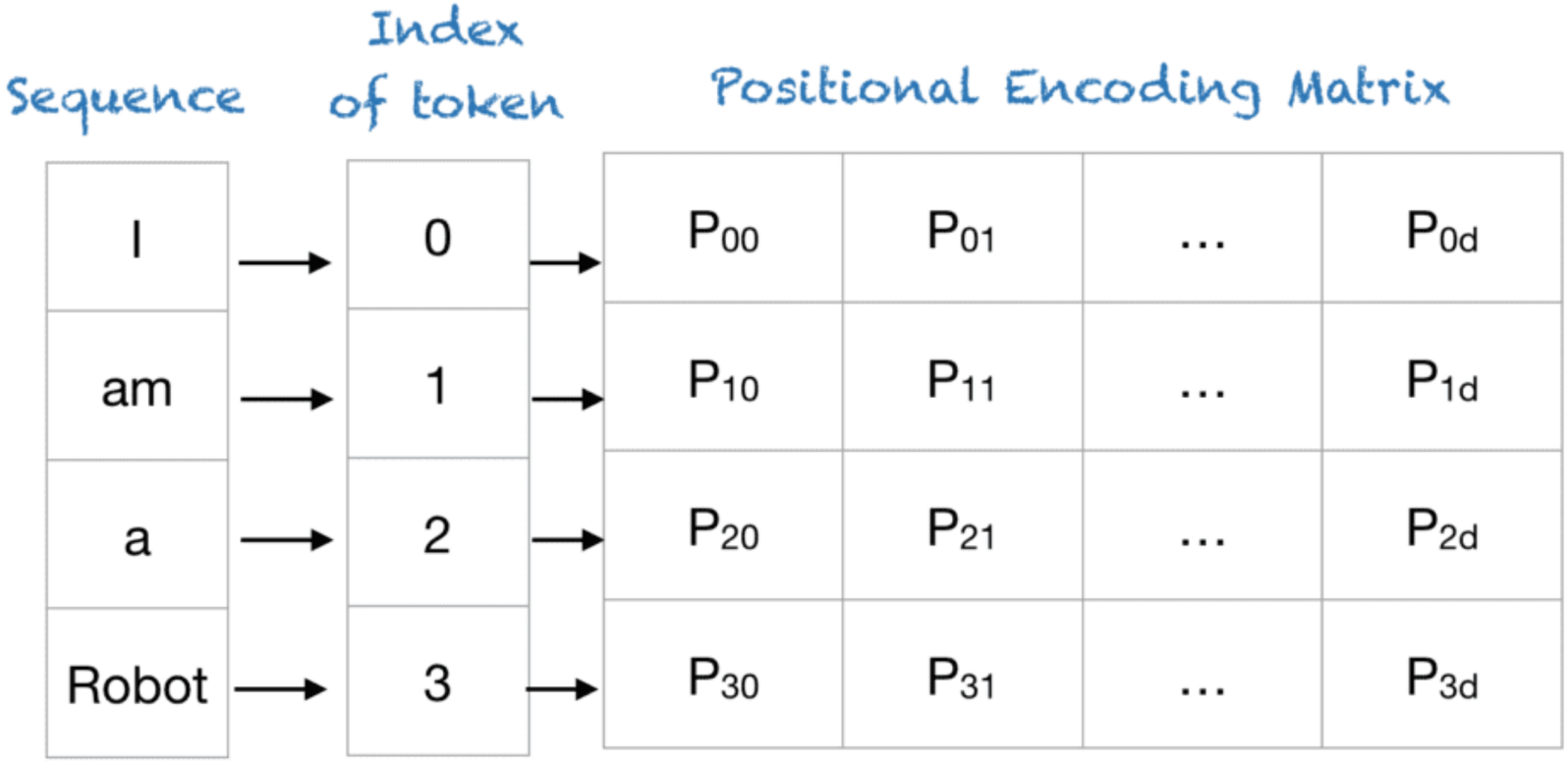
Why Do We Need Positional Encoding?

- **Positional encoding helps preserve word sequence information**, but it does so indirectly.
- The Transformer's self-attention mechanism is **order-agnostic** by default, it treats tokens as a set.
- To make it aware of order, we add **positional encodings** to token embeddings.
- These encodings don't store the words themselves they store **position patterns** (like sine/cosine signals or learned vectors).
- When combined with word embeddings, the model can distinguish:
  - "I like pizza" (positions 0,1,2)
  - from "pizza like I" (positions 2,1,0)
- **The sequence is preserved because each token carries its position info throughout the network**.
- **Important Clarification:** Positional encoding **does not literally save the sentence**.
- It **enables the model to reconstruct the order internally** because every token embedding is unique to its position.

Why Do We Need Positional Encoding?

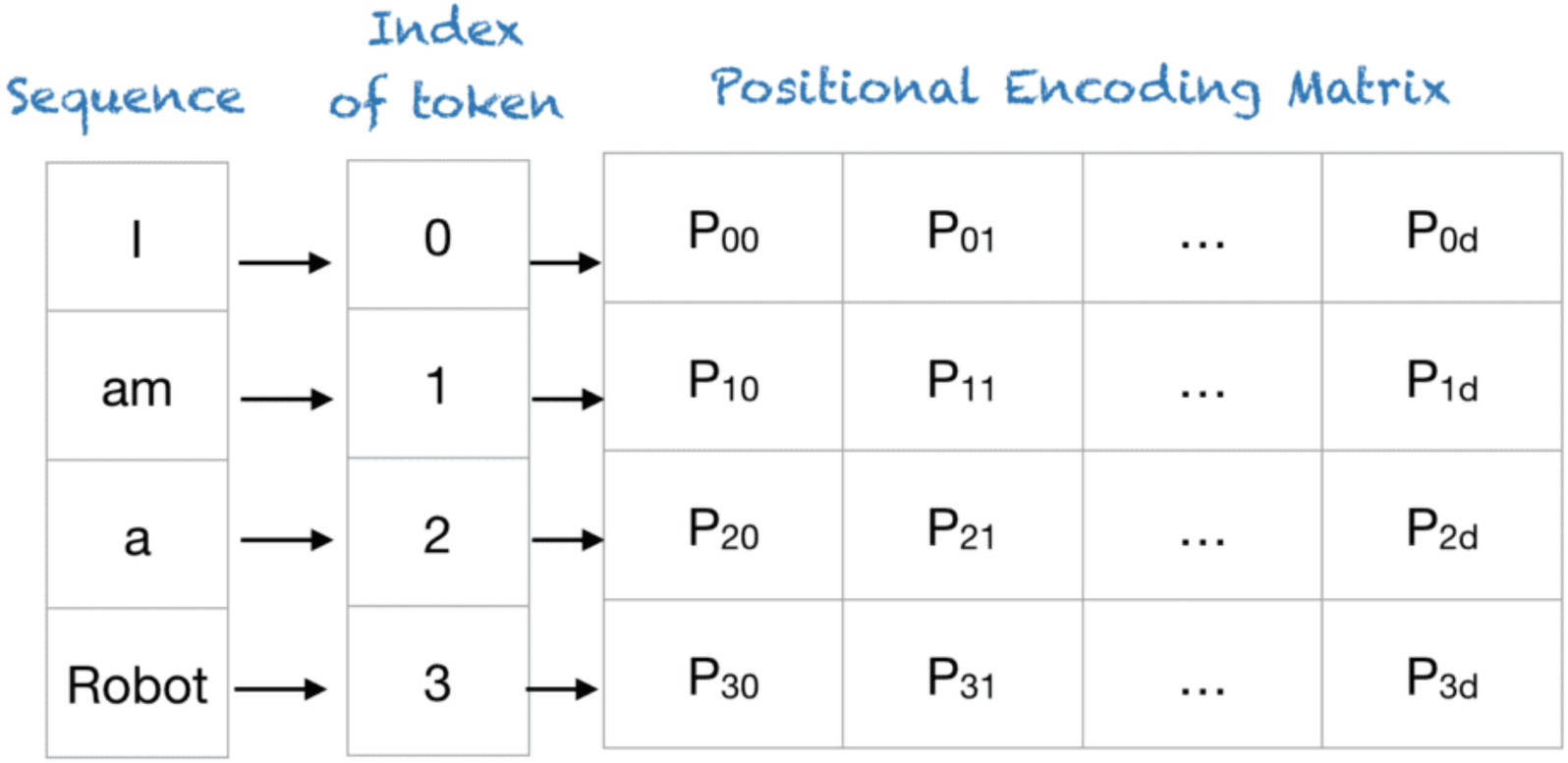
- **Positional encoding helps preserve word sequence information**, but it does so indirectly.
- The Transformer's self-attention mechanism is **order-agnostic** by default, it treats tokens as a set.
- To make it aware of order, we add **positional encodings** to token embeddings.
- These encodings don't store the words themselves they store **position patterns** (like sine/cosine signals or learned vectors).
- When combined with word embeddings, the model can distinguish:
  - "I like pizza" (positions 0,1,2)
  - from "pizza like I" (positions 2,1,0)
- **The sequence is preserved because each token carries its position info throughout the network**.
- **Important Clarification:** Positional encoding **does not literally save the sentence**.
- It **enables the model to reconstruct the order internally** because every token embedding is unique to its position.

Positional Encoding?



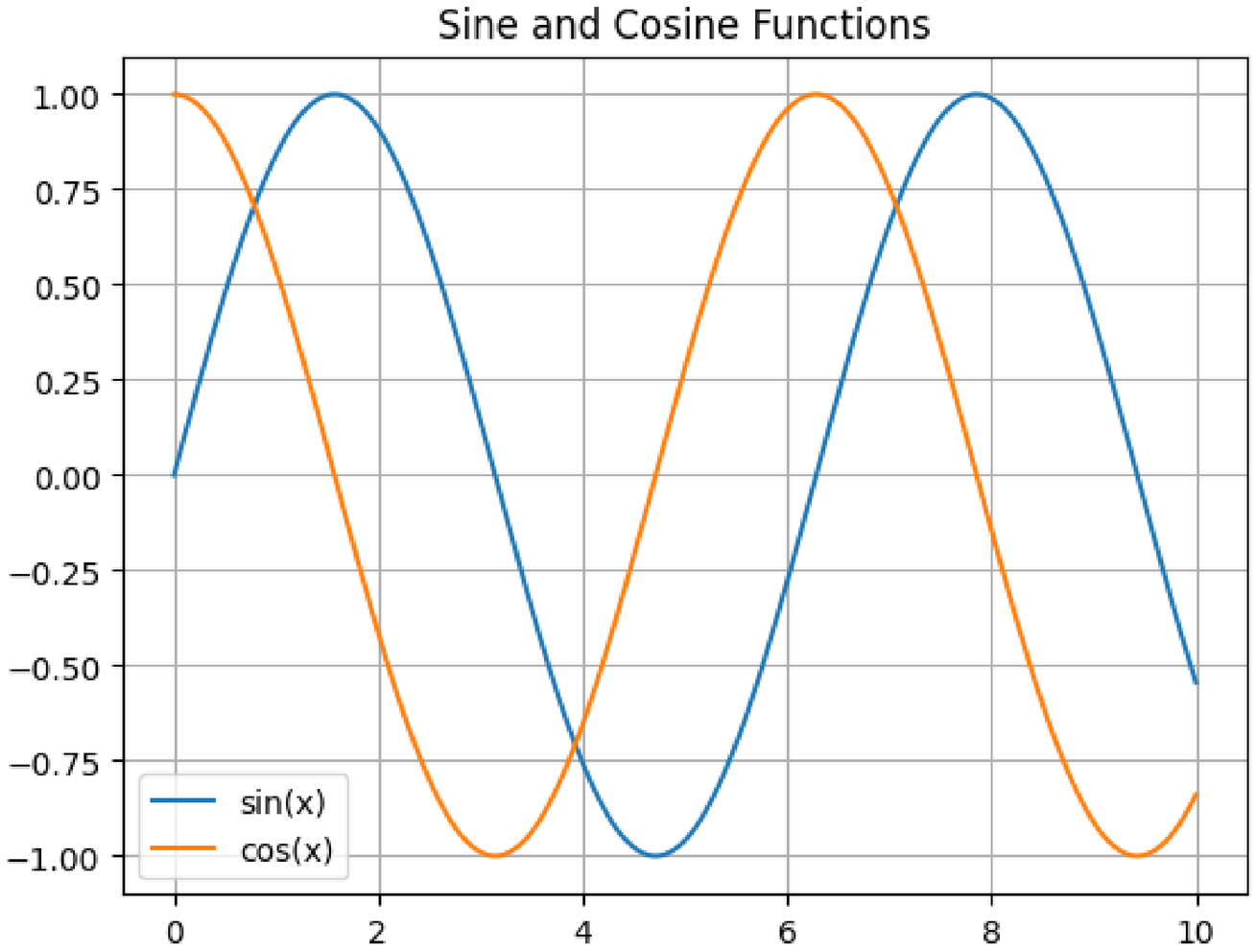
Positional Encoding Matrix for the sequence 'I am a robot'

Positional Encoding?

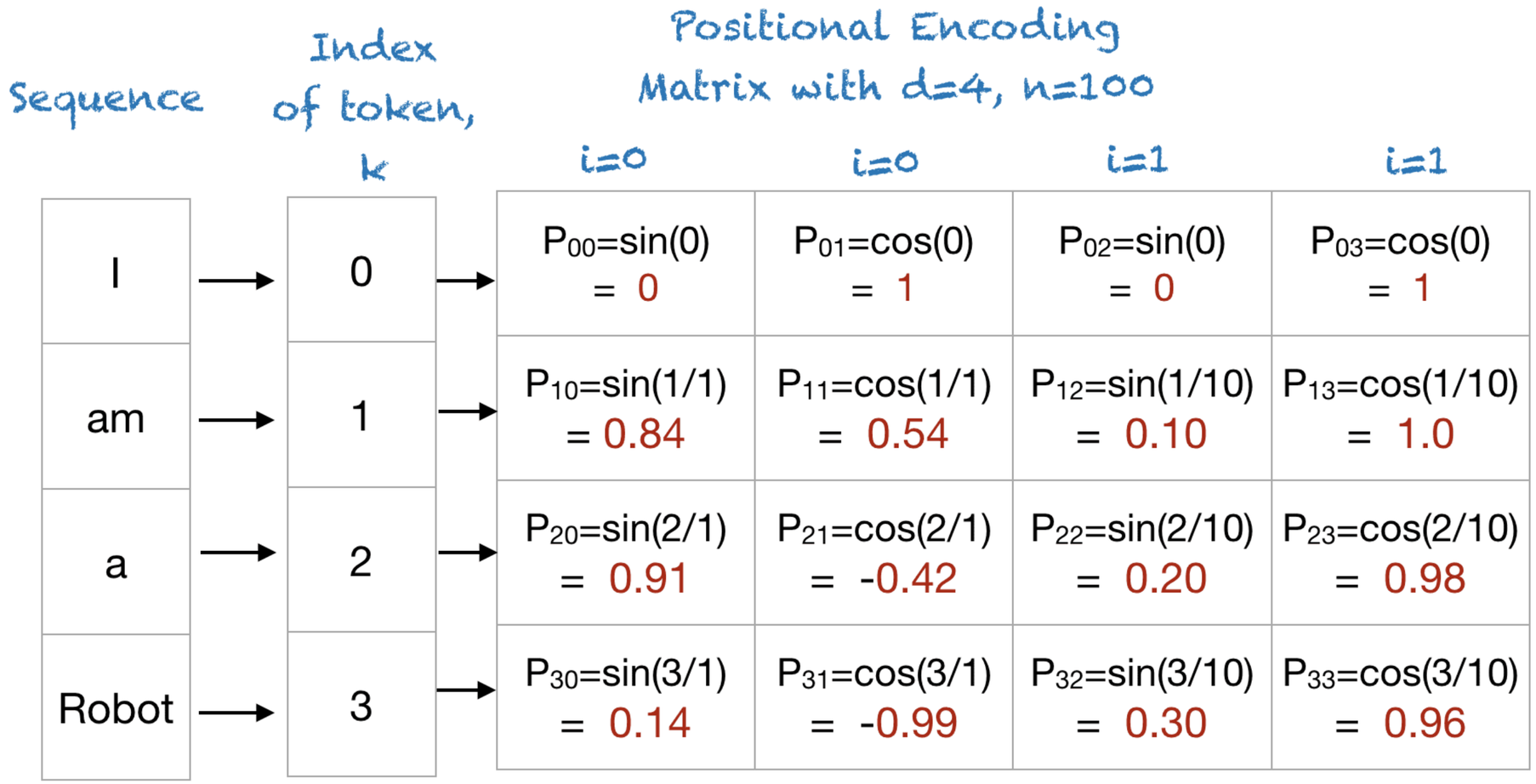


Positional Encoding Matrix for the sequence 'I am a robot'

Positional Encoding?



Positional Encoding?



Positional Encoding Matrix for the sequence 'I am a robot'