ATTENTION IS ALL YOU NEED

Transformer Neural Networks

(Vaswani et al., NeurIPS 2017)

Presented by:

Mahir Al Shahriar (RegNo: 2019831077)

Sequence Transduction – The Core Problem

Sequence transduction refers to any task that maps an input sequence to an output sequence:

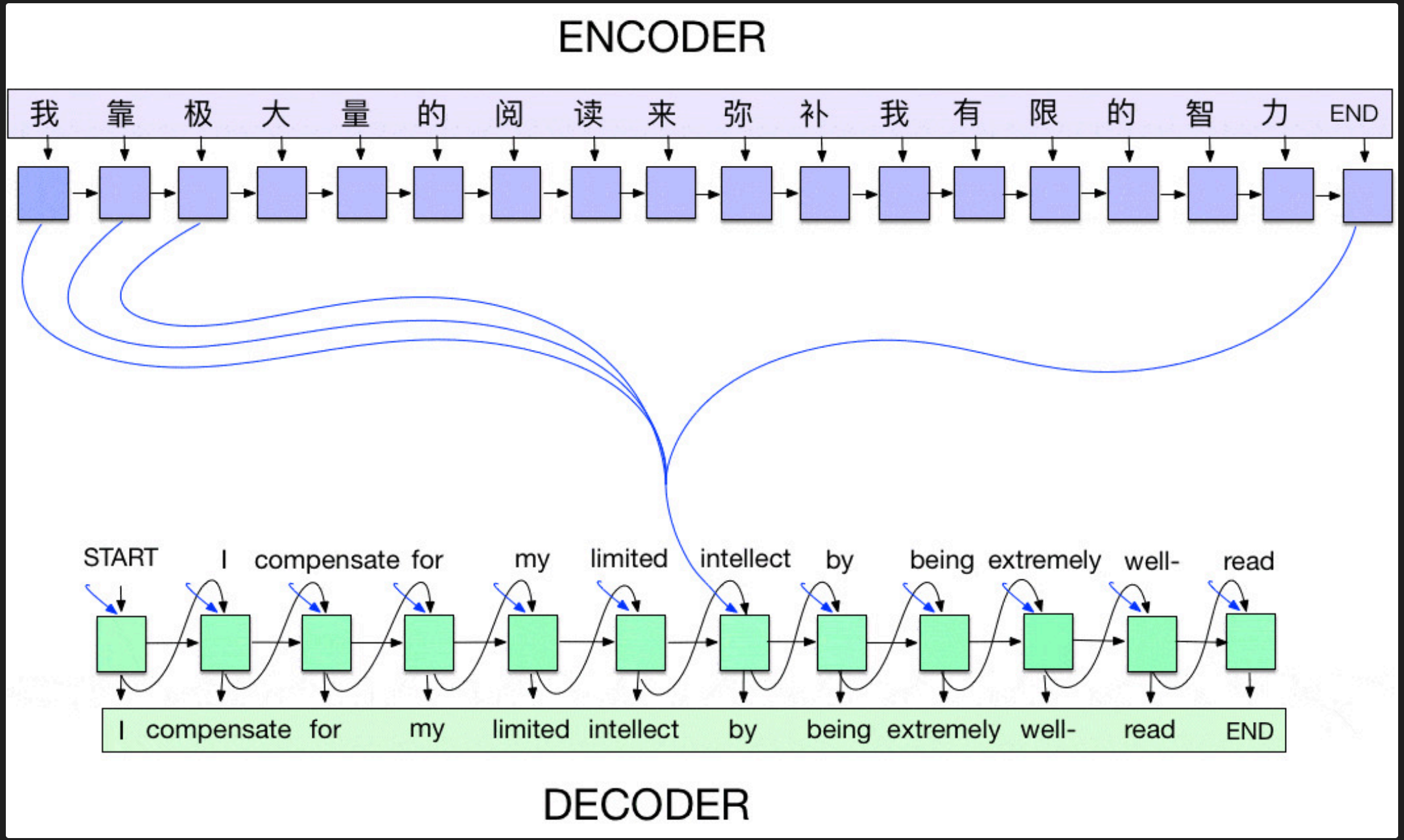
Input: $x = (x_1, x_2, ..., x_n)$

Output: $y = (y_1, y_2, ..., y_m)$

(Note: input and output sequence lengths, n and m, may differ)

Key Examples

Machine Translation	"Hello"	"Bonjour"
Speech Recognition	Audio waveform	"Play music"
Text Summarisation	Long article	Short summary



Foundations and Early Approaches

Before 2014

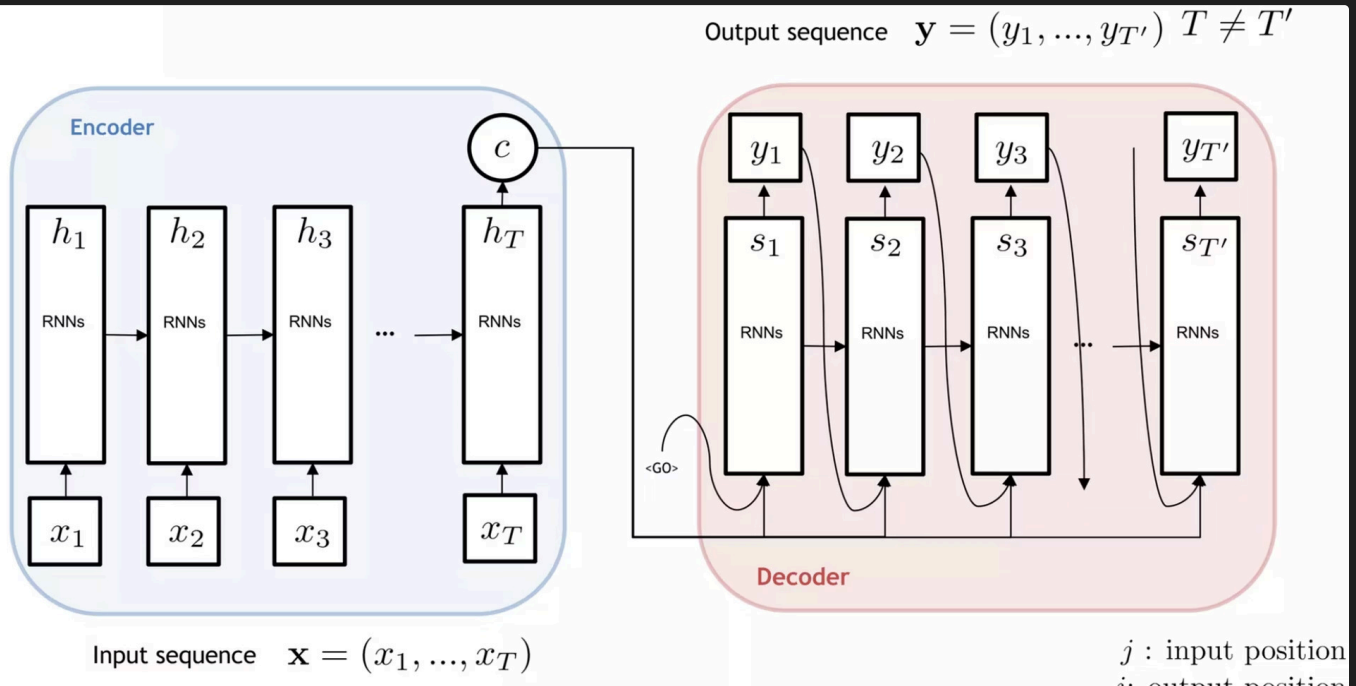
Dominant sequence models were RNNs, LSTMs, and GRUs

Challenges: Recurrent models struggled with long-range dependencies and often forgot earlier inputs

Breakthrough: Bahdanau et al. (2014)

"Neural Machine Translation by Jointly Learning to Align and Translate"

- Introduced the concept of *attention* within a bidirectional RNN encoder and RNN decoder setup
- Enabled models to selectively focus on relevant hidden states while generating each output word
- Marked the first practical use of an attention mechanism — but still depended on recurrent structures

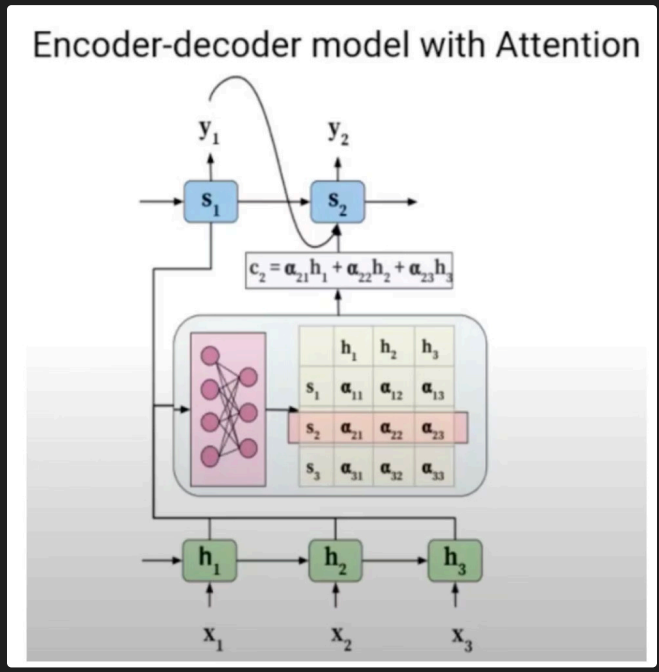


j : input position
i: output position
h_j : hidden state of Encoder RNN $EncRNN$
s_i : hidden state of Decoder RNN $DecRNN$

$$h_j = EncRNN(h_{j-1}, x_j)$$
$$s_i = DecRNN(s_{i-1}, y_{i-1}, c)$$

c : context vector

➔ Static context vector(c) leads to loss of information, especially in long sentences.



Problem Context: Why Move Beyond RNNs?

Traditional Models

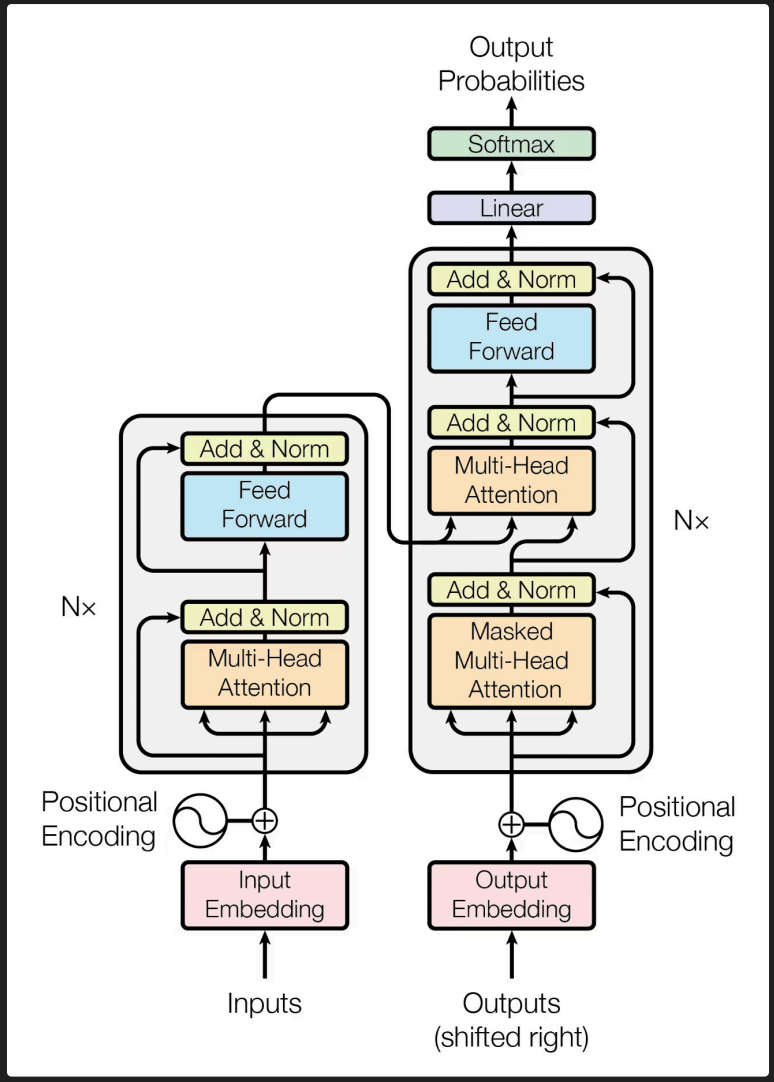
RNNs, LSTMs, GRUs

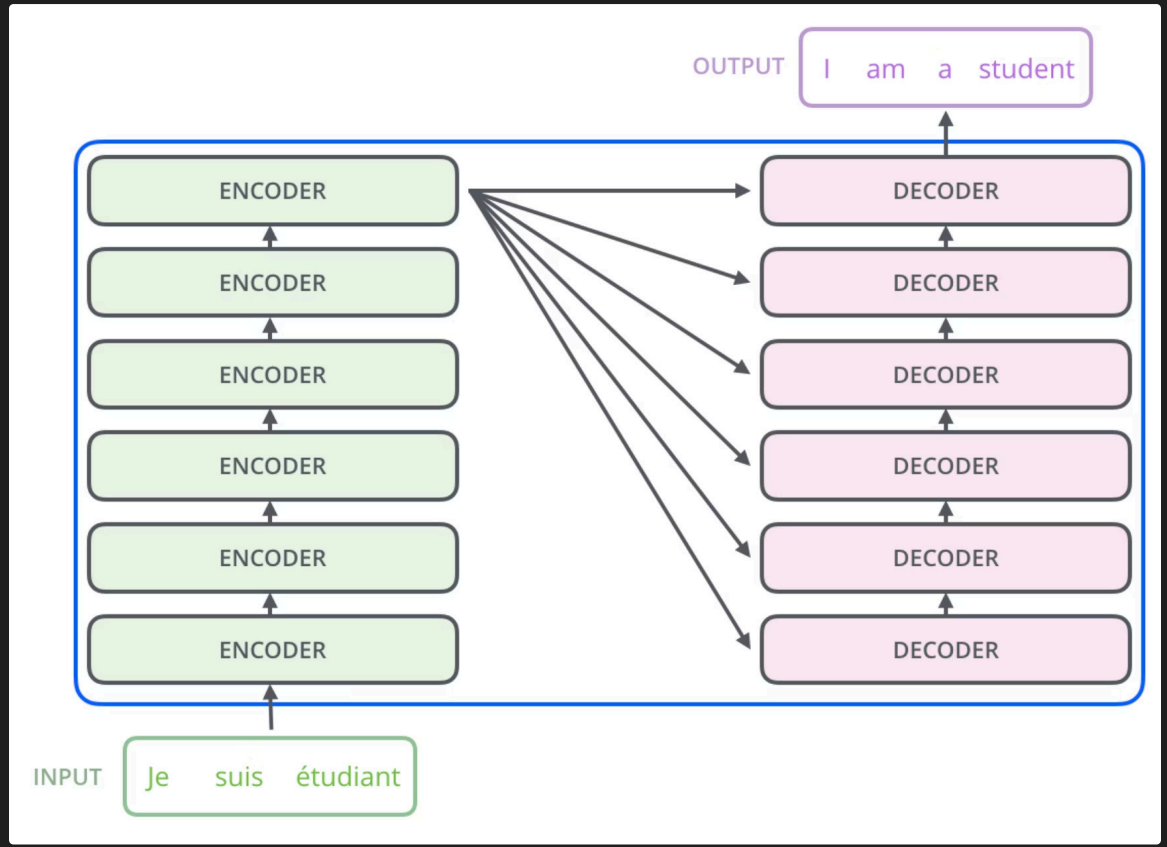
Major Drawbacks

- Training is inherently sequential — hard to parallelize, slow to scale
- Prone to exploding or vanishing gradients during training
- Struggle with long-term dependencies — earlier parts of the input can get lost as only the final hidden state feeds the decoder

Example: A typical RNN must store a masked word's information in its hidden state, pass it step by step, and recover it later — a fragile and inefficient process!

The Transformer solves this with a fresh design: pure attention, no recurrence, no convolutions — enabling faster, more scalable learning.



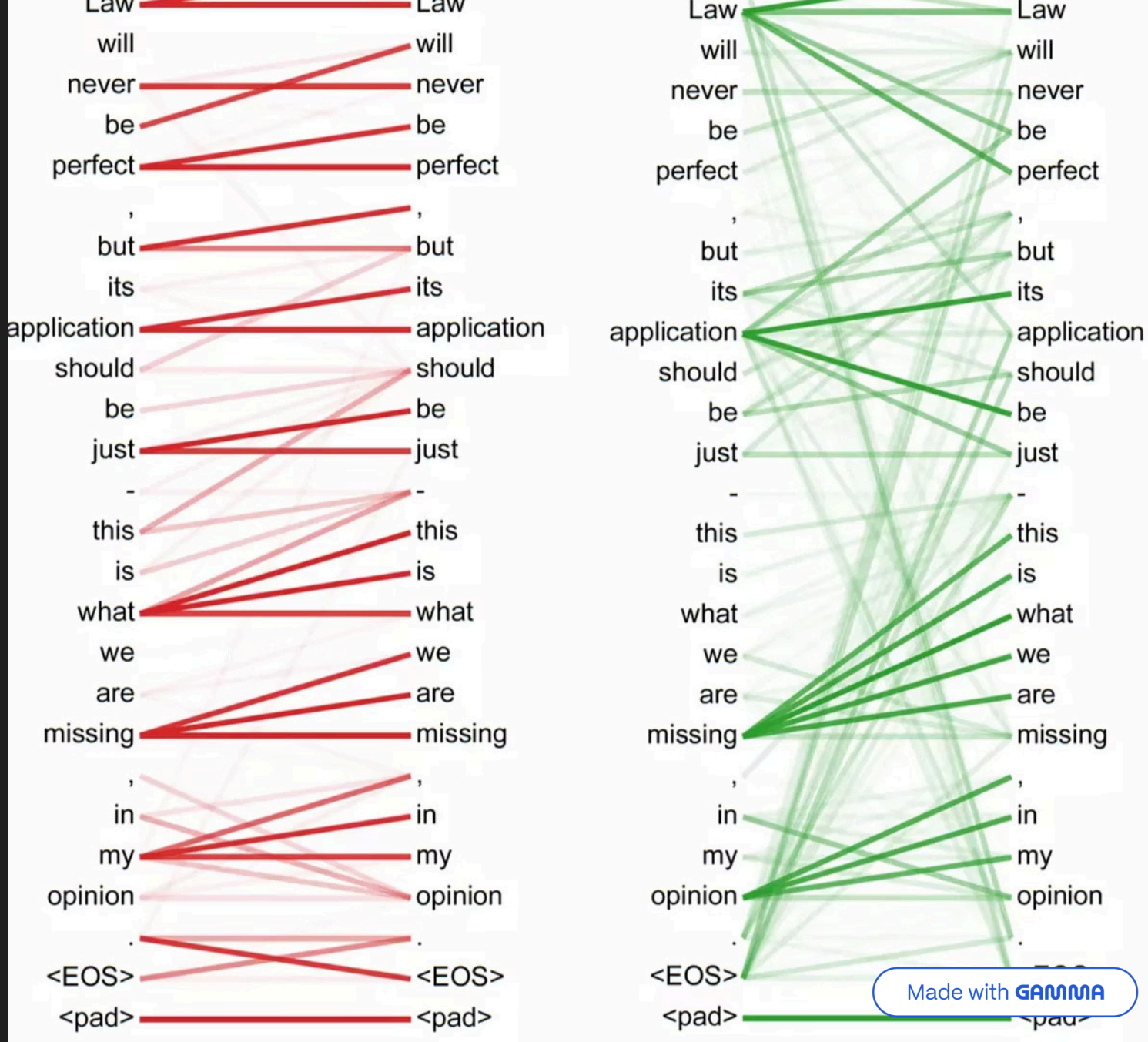


Self-Attention Explained

Core idea: Every token can directly consider all other tokens in the sequence.

Purpose: Capture dependencies between different positions to build richer representations.

Mechanism: Each token is transformed into a *query*, *key*, and *value* by applying learned weight matrices.



The Concept of Attention

> From Vaswani et al. (2017): "Attention maps a query and a set of key-value pairs to an output vector. The output is a weighted sum of the values, where each weight reflects how well the query matches the corresponding key."

Goal: Learn, in a differentiable way, which parts of the input are most relevant.

1️⃣ Generate Vectors

For each input, generate three vectors: query, key, and value.

2️⃣ Compute Scores

Compute a score indicating how much focus each input position should receive during encoding.

3️⃣ Select and Weight

Select and weight relevant values based on how compatible they are with the query.

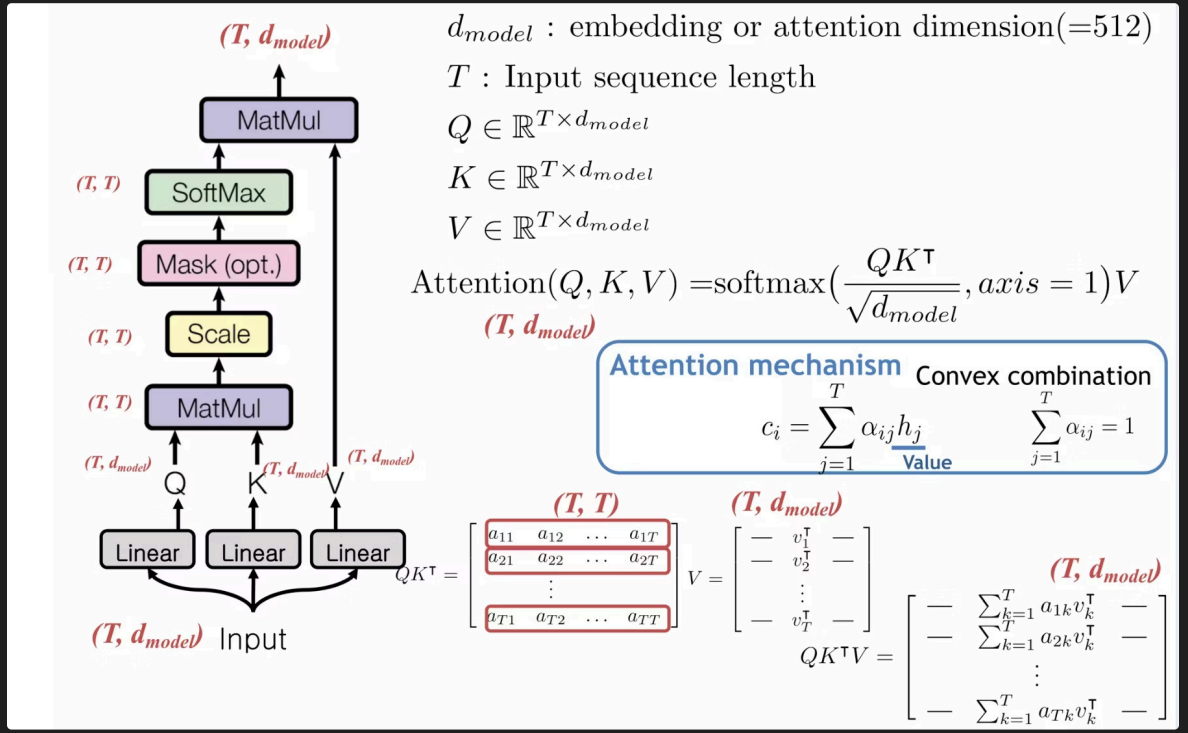
Scaled Dot-Product Attention

Inputs:

- Queries (dimension: dk) → packed into matrix Q
- Keys (dimension: dk) and Values (dimension: d) → packed into matrices K and V

Core mechanism:

- Compute similarity scores between queries and keys.
- Apply softmax to get attention weights.
- Use these weights for a weighted sum of values.



Multi-Head Attention Mechanism

1

How it works:

1. Multiple independent linear projections (weight matrices) generate separate sets of queries, keys, and values.
2. Each "head" performs attention in parallel.
3. Outputs from all heads are concatenated and passed through another linear layer with non-linearity.

2

Why multiple heads?

1. Enables the model to focus on various parts of the input simultaneously.
2. Creates diverse "representation subspaces" for richer learned features.
3. Prevents a single attention head from becoming a bottleneck.

Parallel Computation

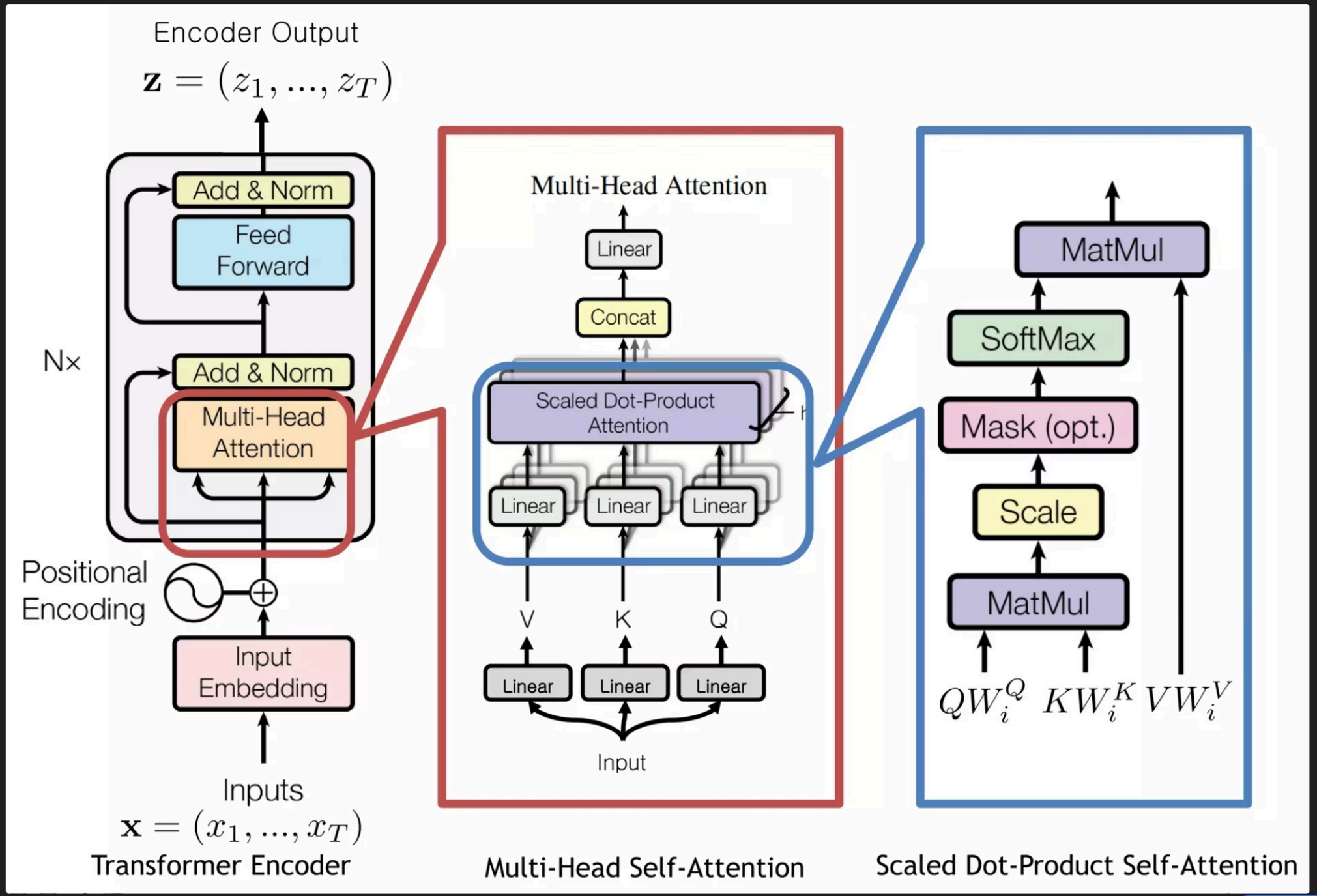
The transformer typically computes multiple (e.g., 8) independent attention heads in parallel.

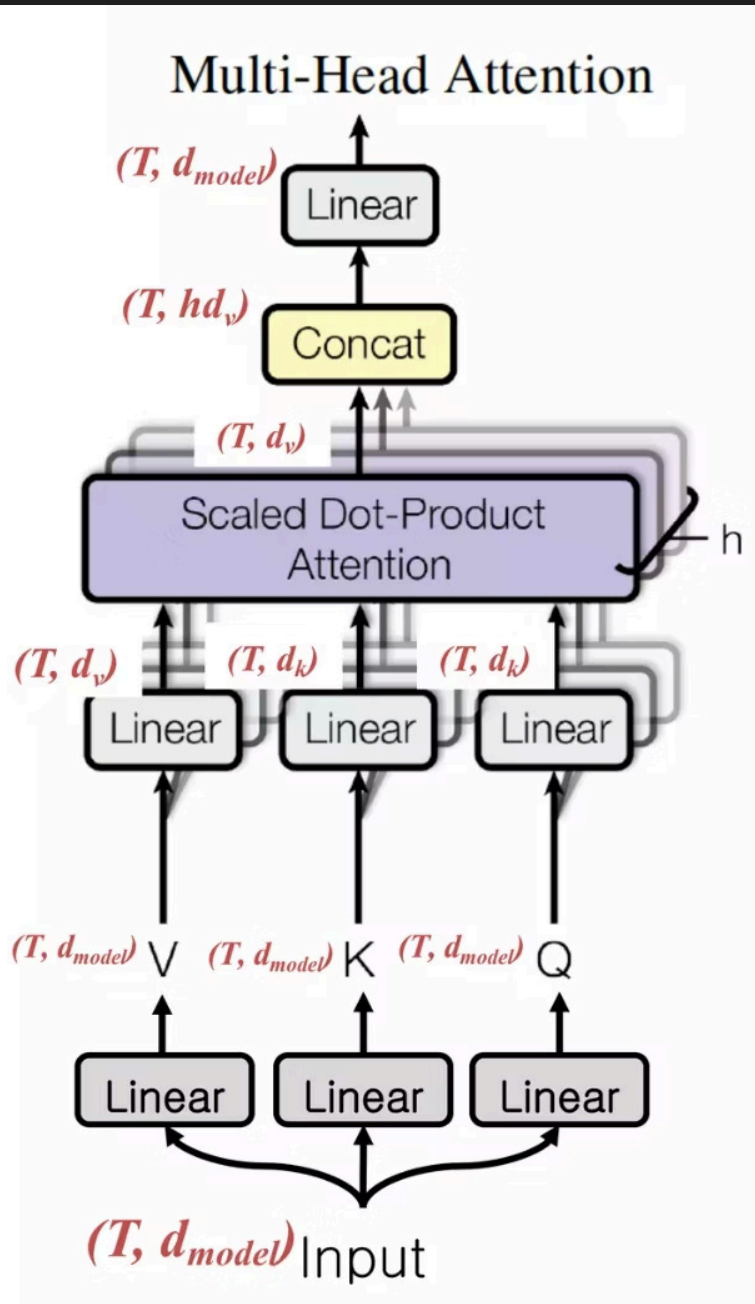
Diverse Perspectives

Each head attends to different aspects or positions in the sequence, capturing diverse relationships.

Unified Representation

The outputs from these heads are merged (concatenated) and transformed to form a unified, richer representation.





$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k} \qquad h : \#\text{ of parralel attention}$$

$$W_i^K \in \mathbb{R}^{d_{model} \times d_k}$$

$$W_i^V \in \mathbb{R}^{d_{model} \times d_v}$$

$$W_i^O \in \mathbb{R}^{hd_v \times d_{model}}$$

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

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- It allows the model to jointly attend to information from different representation subspaces at different positions
  ex) i=1 : word class, i=2 : pronoun
  i=3 : singular/plural

1) This is our input sentence*

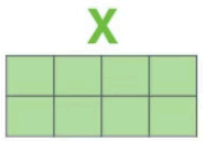
2) We embed each word*

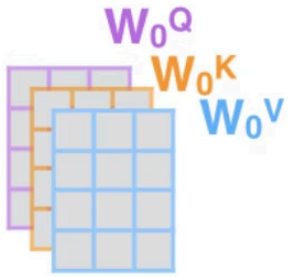
3) Split into 8 heads. We multiply X or R with weight matrices

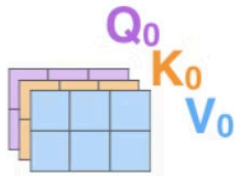
4) Calculate attention using the resulting Q/K/V matrices

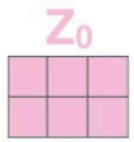
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

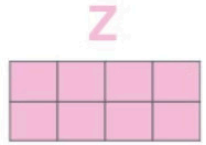
Thinking
Machines

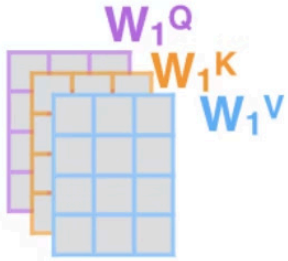


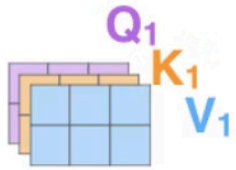


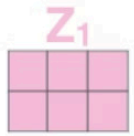




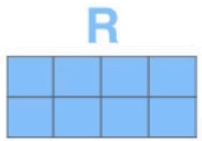


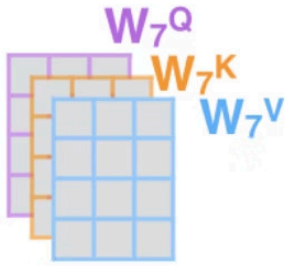


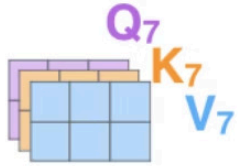


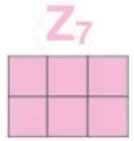


* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one









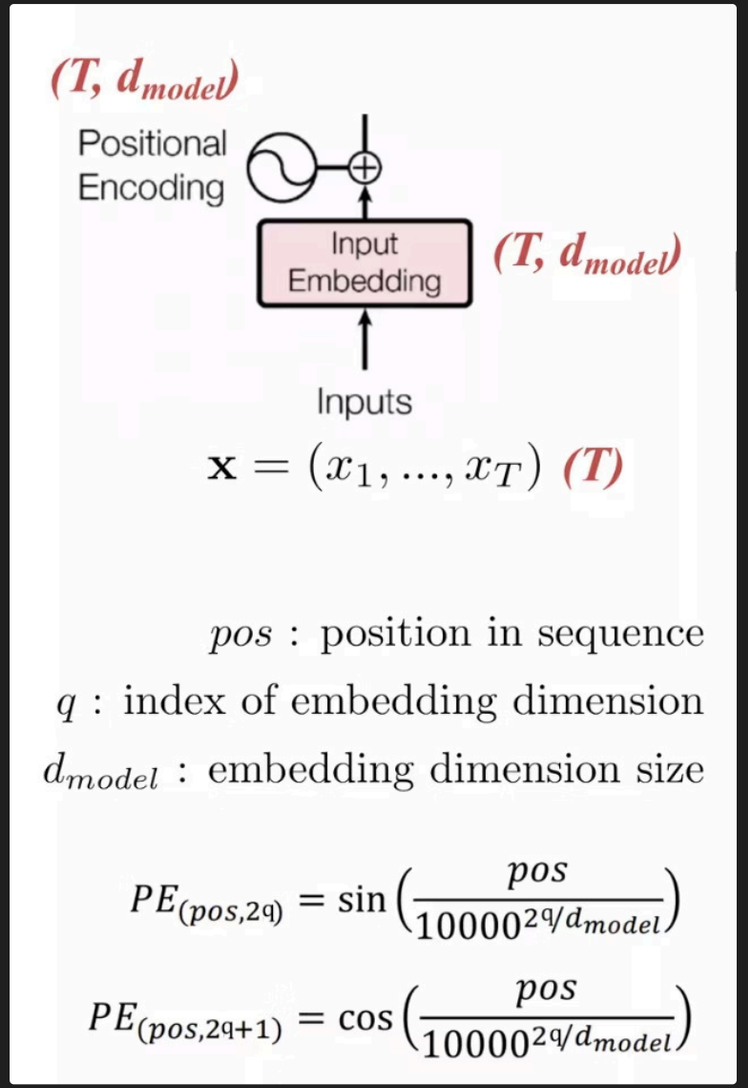
Positional Encoding: Capturing Word Order

Why it's needed:

- Unlike RNNs, transformers have no inherent sense of sequence order.
- So, we add positional information to token embeddings.

How it works:

- A positional embedding is a vector that encodes each token's position.
- Added element-wise to the word embedding to inject order.
- These embeddings can be predefined (as in the original paper) or learned during training.
- The sinusoidal function design allows the model to learn relative positions easily — any fixed offset can be expressed as a simple function of position.



Decoder & Masked Self-Attention

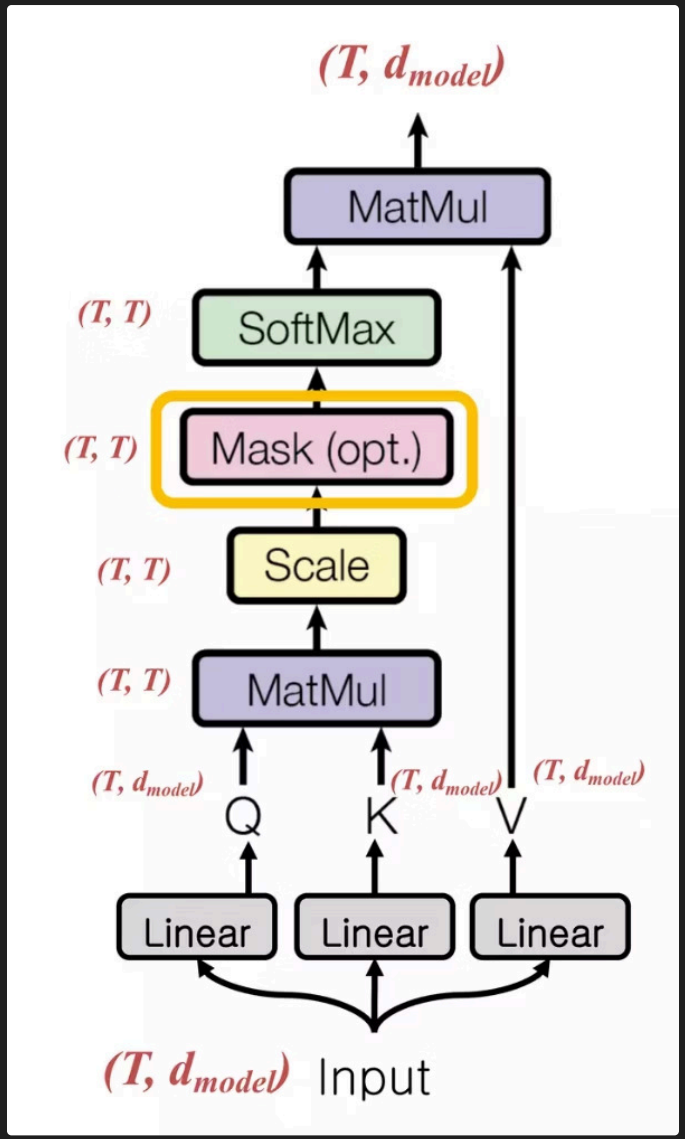
Key Difference from Encoder: Autoregressive generation → must **prevent future token access** (no "cheating").

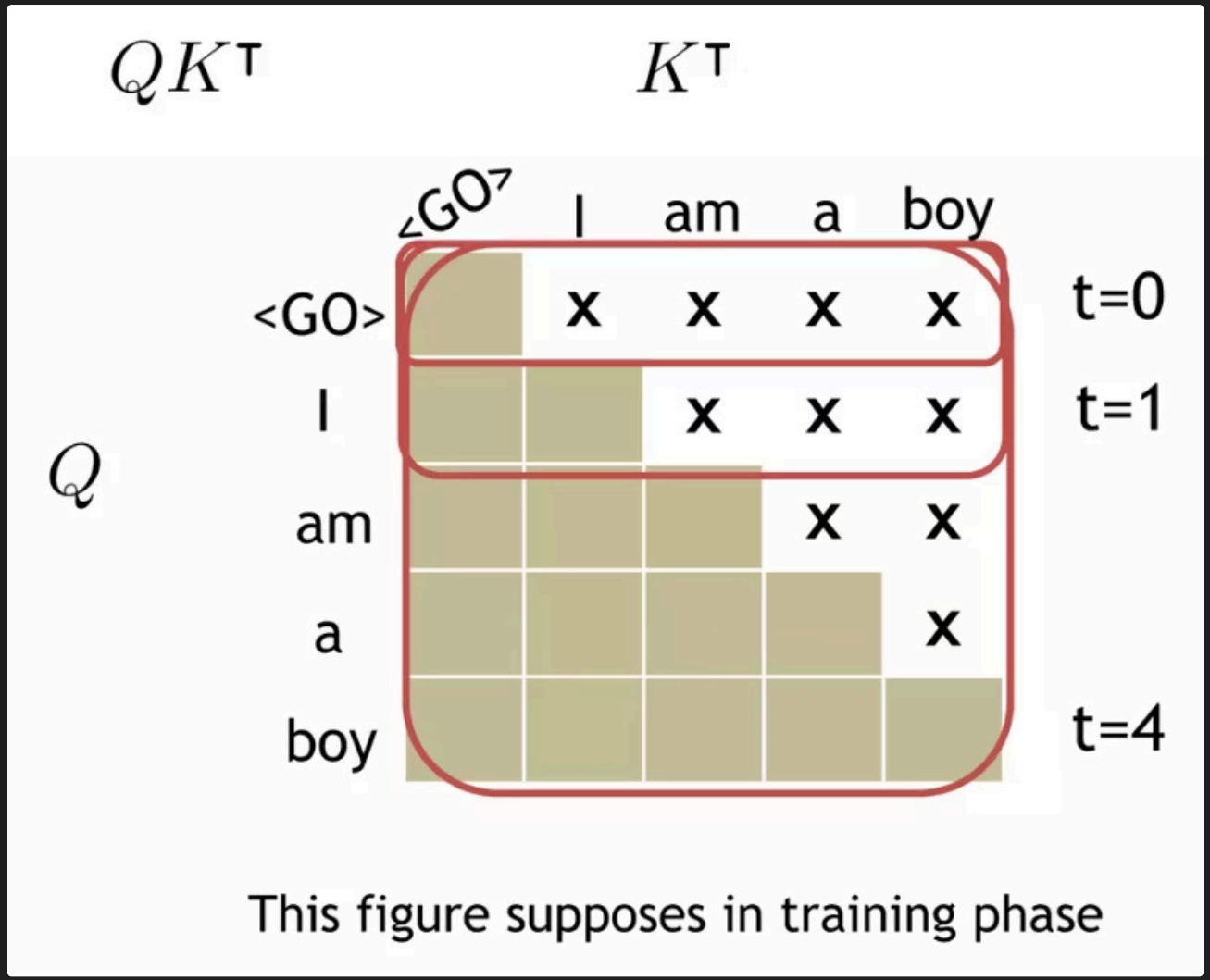
1. Masked Self-Attention (Lower-Triangular Mask)

Purpose: Enforce _causal_ attention: position _i_ can only attend to positions _≤i_.

Mechanism:

1. **Compute Scores**: Query-key dot products (like encoder).
2. **Apply Mask**:
   - **Lower-triangular matrix** (1s allowed, upper set to **-∞**):
   - After softmax, **0** positions → zero attention weight.
3. **Weighted Sum**: Only past tokens contribute.





Encoder-Decoder Attention (Cross-Attention)

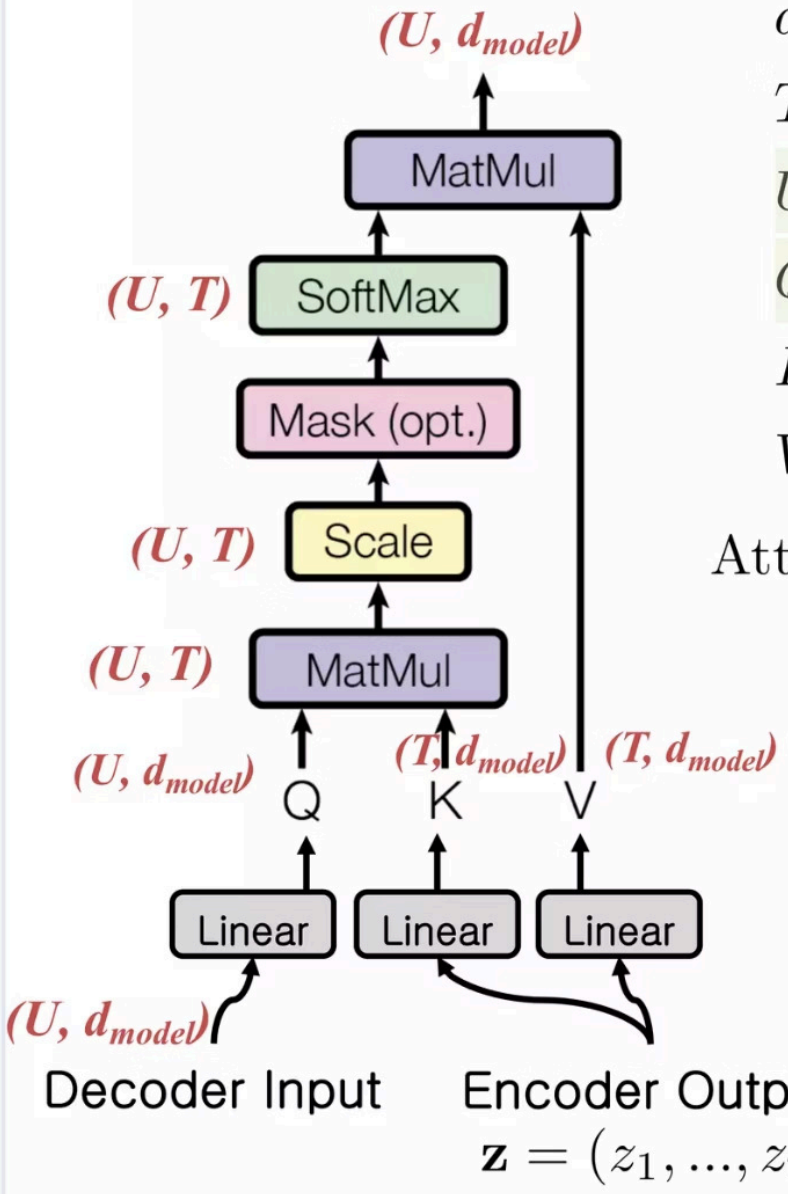
Purpose: Bridges encoder and decoder by allowing the decoder to **dynamically retrieve** relevant information from the input sequence.

Key Mechanism:

1. **Inputs**:
   - **Queries (Q)**: Current decoder layer's output (autoregressive context).
   - **Keys/Values (K, V)**: Final output of the encoder (_memory_ of the input).
2. **Process**:
   - Computes attention scores between decoder queries and encoder keys.
   - Uses scores to weight encoder values (standard scaled dot-product attention).
   - _No masking_ (all encoder positions are accessible).

Why It's Unique:

- **Asymmetric**: Decoder "asks" questions (Q), encoder "answers" (K, V).
- **Contextual Retrieval**: Decoder learns to "look back" at input when needed.
  - Example: In translation, decoder attends to source-language words while generating target-language words.



d_{model} : embedding or attention dimension(=512)

T : Input sequence length

U : Generated output sequence length

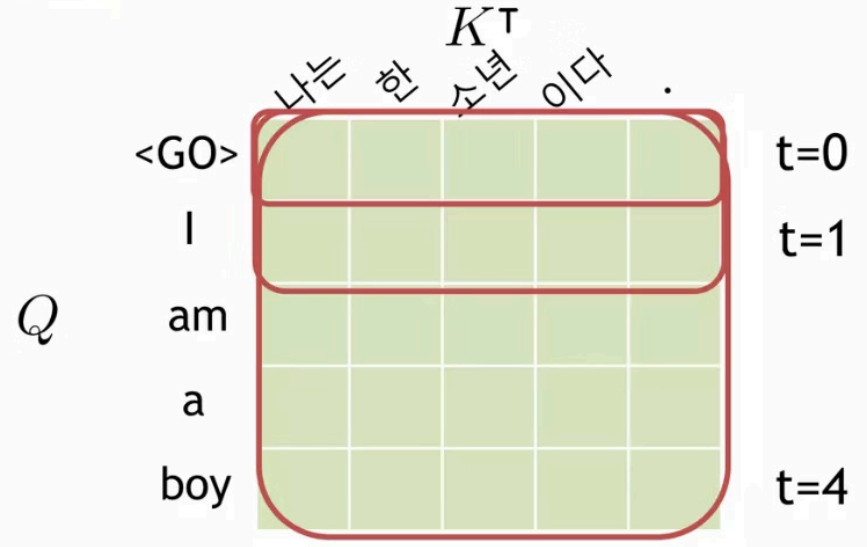
$Q \in \mathbb{R}^{U \times d_{model}}$

$K \in \mathbb{R}^{T \times d_{model}}$

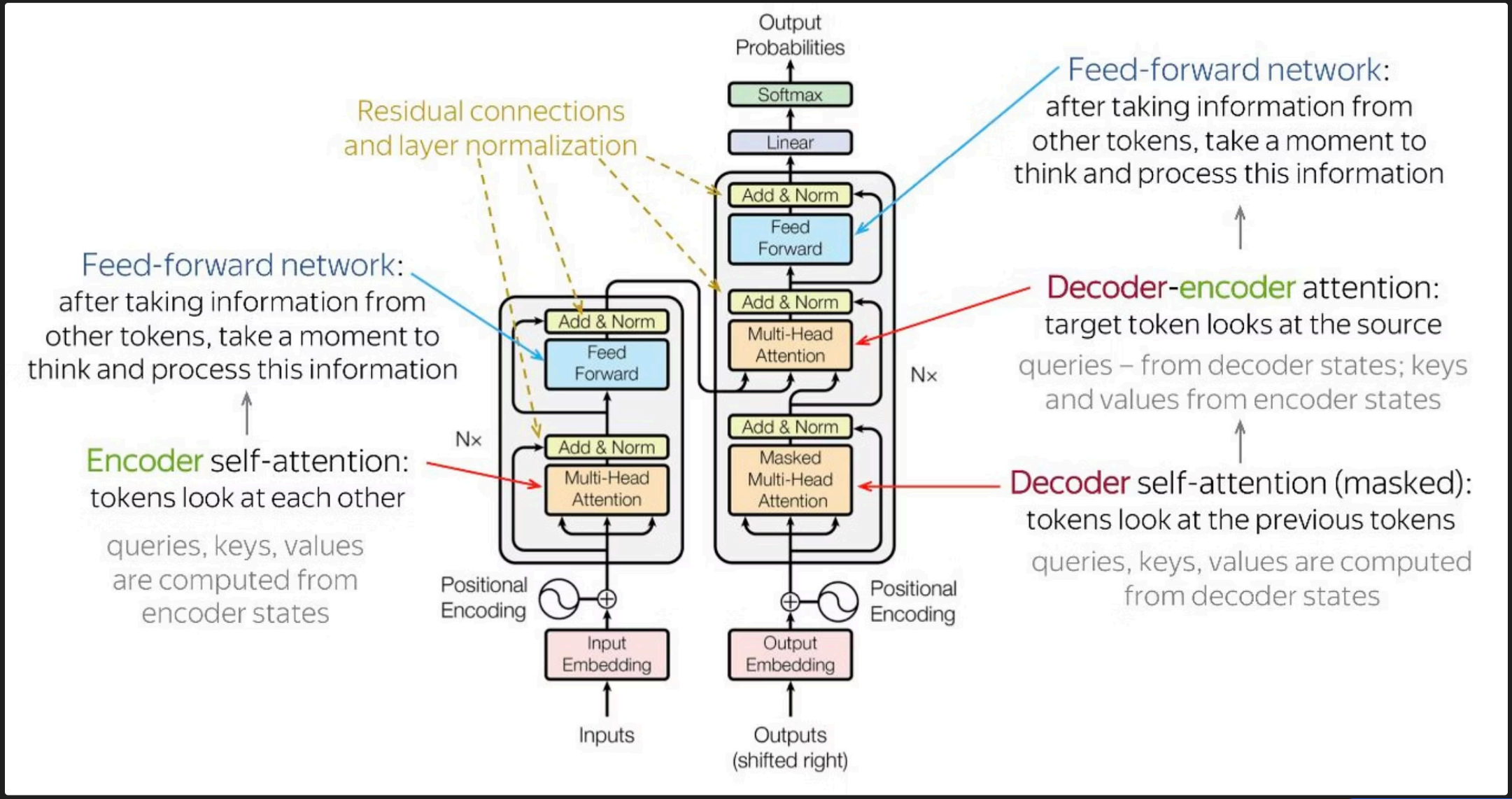
$V \in \mathbb{R}^{T \times d_{model}}$

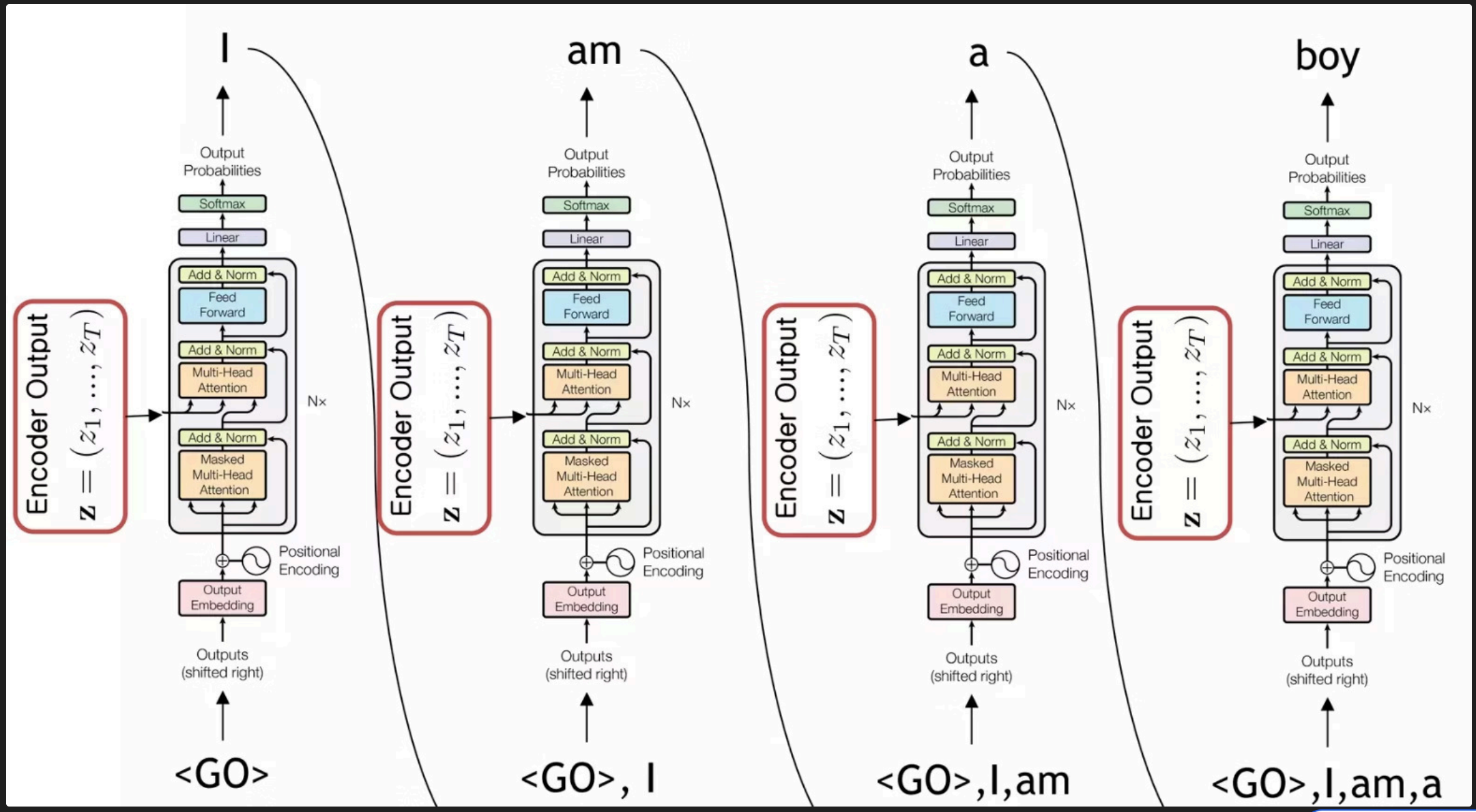
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\intercal}{\sqrt{d_{model}}}, axis = 1\right)V$$

(U, d_{model})



Full Architecture





Training Details and Experimental Setup

Hardware Details:

- Trained on 8 NVIDIA P100 GPU
- The big models were trained for 300,000 steps (3.5 days)

Datasets:

- Trained on WMT 2014 English-to-German translation task
- 4.5 million sentence pairs
- Trained on WMT 2014 English-to-French translation task
- 36 million sentences

Results

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	