

Computer Vision

Week 10

2025-2

Mobile Systems Engineering
Dankook University

Motivation & Bridge from Last Lecture

■ Seq2Seq Model and Its Limitation

- **Seq2Seq (Sequence-to-Sequence)**

- Converts one sequence into another sequence (ex: Korean → English translation).
 - Uses **encoder** (to read the source sentence) and **decoder** (to generate the target sentence).

- **Context Vector v**

- The encoder compresses the entire source sentence into a **single fixed-size vector**.
 - This vector is then passed to the decoder to generate the target sentence.

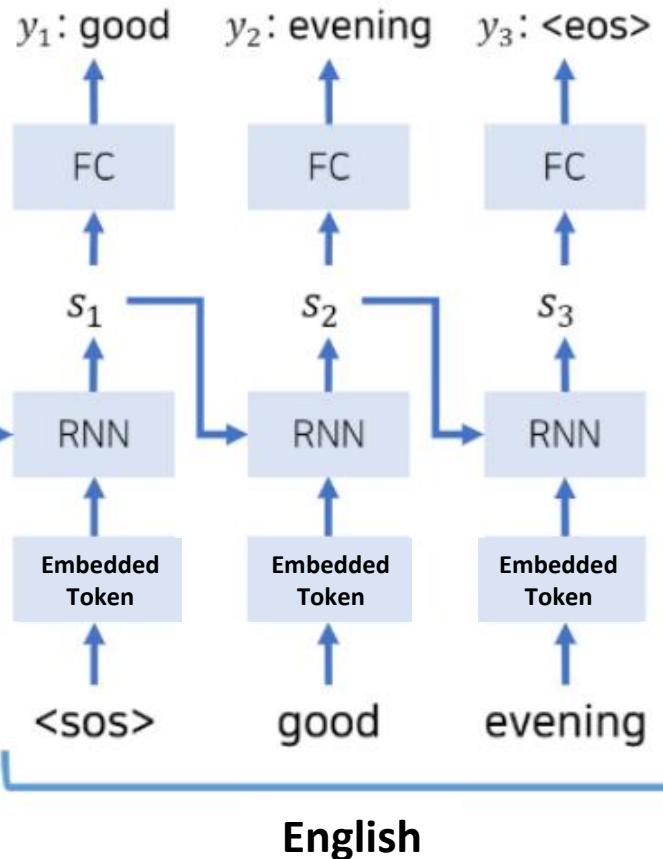
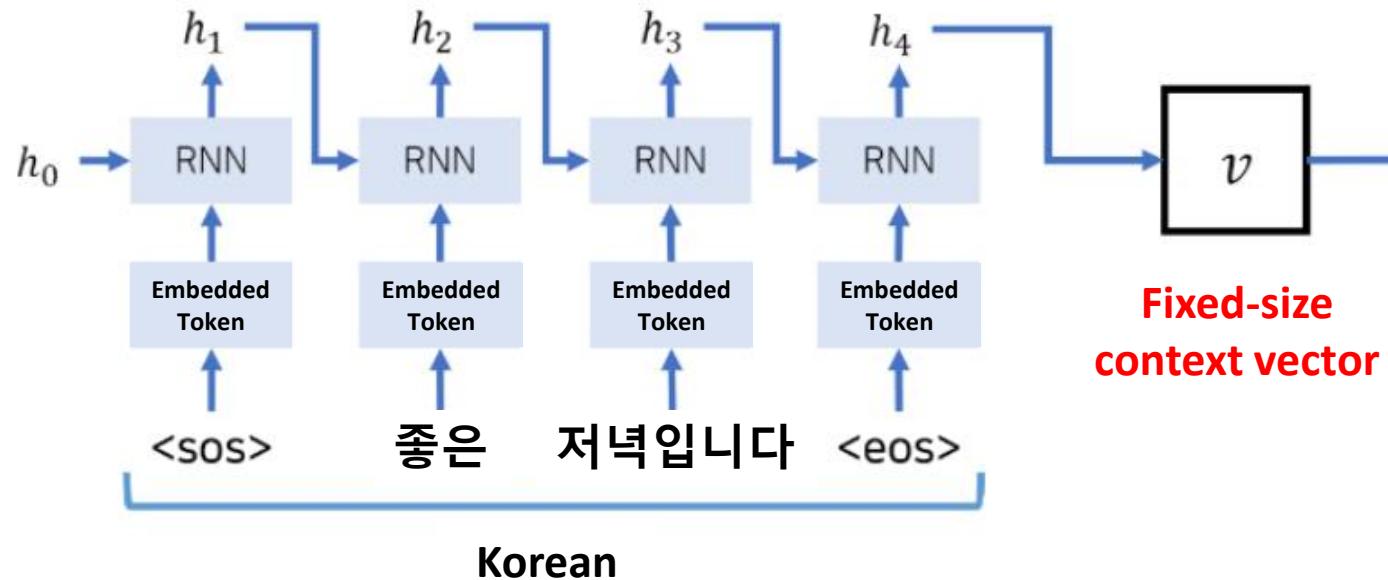
- **Problem – Bottleneck**

- All information from a long source sentence must fit into one vector.
 - This **bottleneck** causes **performance degradation** as sentence length increases.

Motivation & Bridge from Last Lecture

■ Seq2Seq Example

- Example: Korean → English translation



- Source sentence (Korean)

✓ “좋은 저녁입니다.”

- Encoder

✓ Processes each word step by step → produces **hidden states** $h_1, h_2, h_3\dots$ → compresses into one **context vector** v .

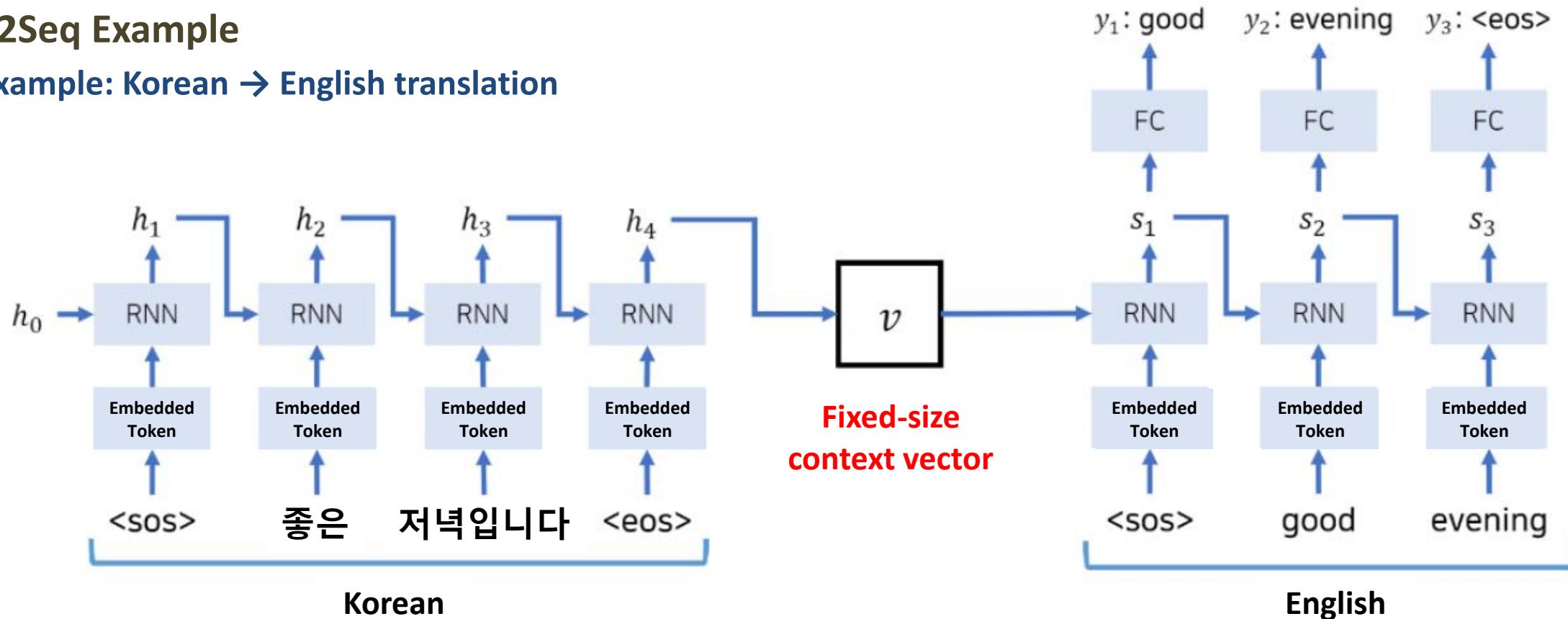
- Decoder

✓ Uses v to generate English sentence: “Good evening.”

Motivation & Bridge from Last Lecture

■ Seq2Seq Example

- Example: Korean → English translation



○ Process

- ✓ **Step 1.** Each Korean word updates the hidden state.
- ✓ **Step 2.** Final hidden state becomes the **context vector**.
- ✓ **Step 3.** Decoder uses \mathcal{V} to produce hidden states s_1, s_2, s_3, \dots
- ✓ **Step 4.** Each step outputs one English word until <eos> token is reached.

Motivation & Bridge from Last Lecture

■ Why Attention Is Needed

• 1. Limitation of Recurrent Models (RNN, LSTM, GRU)

- They generate the hidden state h_t as a function of the previous hidden state h_{t-1} and the current input.
- This sequential dependency prevents parallelization across time steps, making training inefficient for long sequences.
- Memory usage grows and batch processing becomes difficult as the sequence length increases.

• 2. Seq2Seq with Fixed Context Vector

- Early neural machine translation used Seq2Seq with LSTMs.
- The encoder compresses the entire input sentence into a single fixed-size context vector.
- The decoder then generates the output sequence word by word from this vector.
- Problem: For long sentences, this bottleneck CANNOT capture all information → performance drops.
- This also leads to a locality issue: the decoder only sees the compressed context, losing global sentence structure.

Motivation for Attention

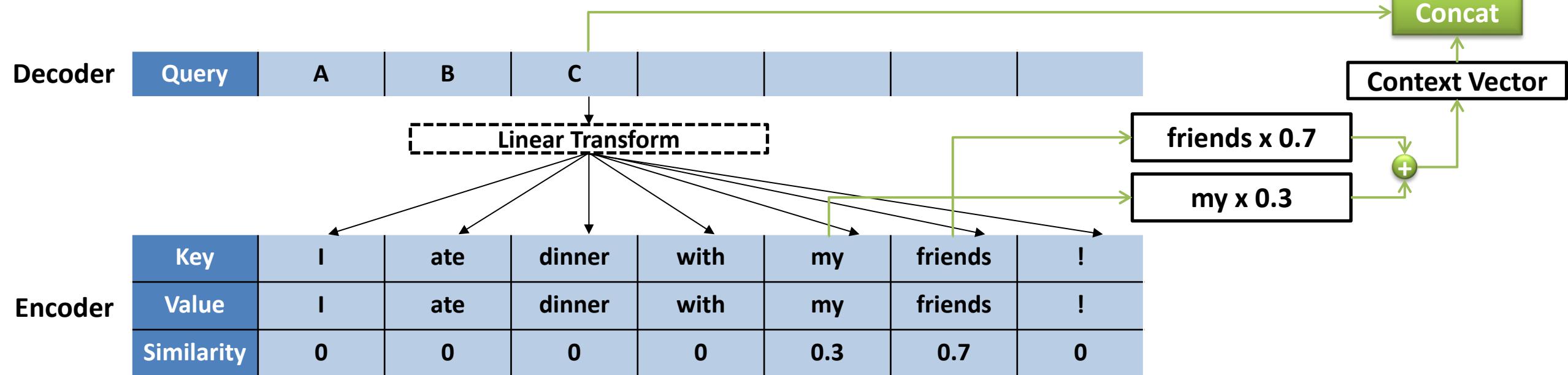
- Instead of relying on **one fixed vector**, what if the decoder could access **all encoder hidden states** dynamically?
- This allows the decoder to look back at the **entire input sequence**, not just a compressed summary.
→ This leads to Seq2Seq with Attention (2015), where the decoder attends to all encoder outputs.
→ Later, the Transformer (2017) took this further by removing recurrence entirely and using only Attention, enabling parallelization and capturing global dependencies effectively.

Motivation & Bridge from Last Lecture

■ Recap – Basic Attention Mechanism

- Overview – Attention as Key-Value Lookup

- Attention is a **differentiable key-value function**.
- Unlike a dictionary, the query does not need to exactly match a key.
- Instead, the query compares with all keys to compute **similarity scores**.
- The output is a **weighted sum of values**, where weights are based on similarity.



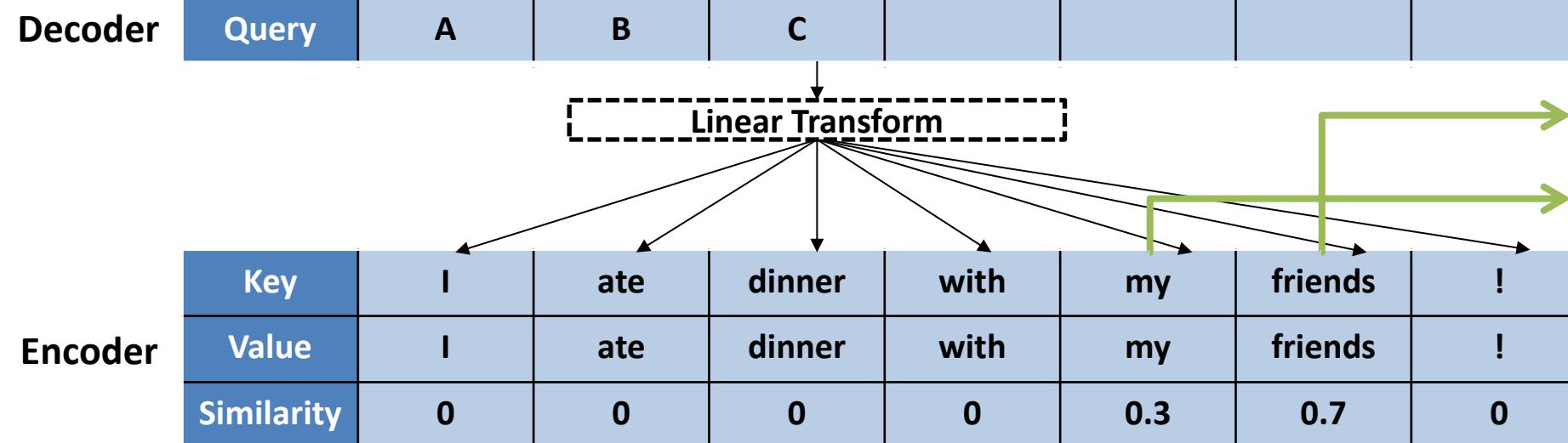
Motivation & Bridge from Last Lecture

Recap – Basic Attention Mechanism

- Key Components – Key, Value, and Linear Transformation in Attention

- 1. Key and Value

- ✓ Each **Key** and **Value** can be seen as the **hidden state vectors** produced by the encoder (RNN/LSTM/GRU).
- ✓ They are **not just the original word embeddings**, but representations that contain **contextual information up to each token**.
- ✓ In most attention mechanisms, **Key** and **Value** come from the same encoder hidden states:
 - **Key**: used to measure similarity with the Query
 - **Value**: used to form the **context vector** through a weighted sum



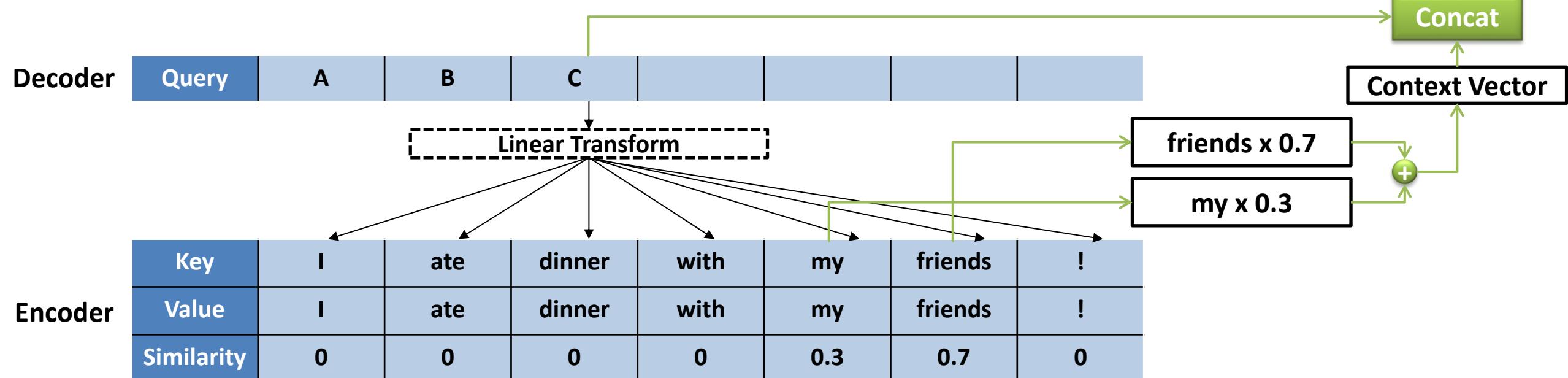
Motivation & Bridge from Last Lecture

■ Recap – Basic Attention Mechanism

- Key Components – Key, Value, and Linear Transformation in Attention

- 2. Linear Transformation

- ✓ The “Linear Transform” in the diagram is applied to the **decoder hidden state** (the Query).
 - ✓ This transforms the decoder’s hidden state into a **Query vector** that can be compared with the Keys.



Motivation & Bridge from Last Lecture

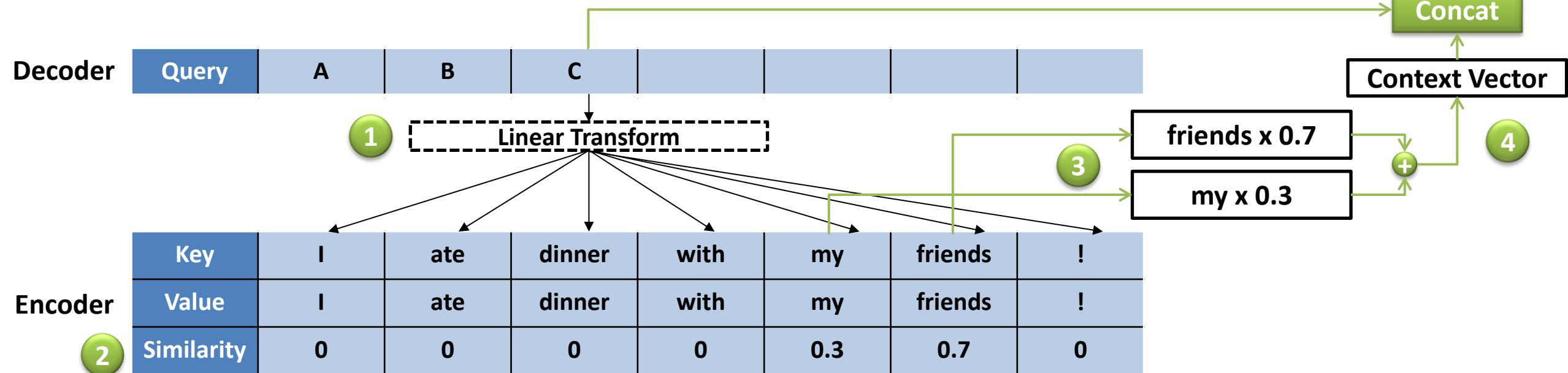
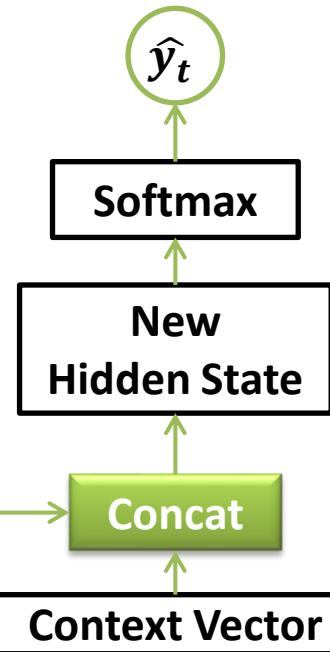
■ Recap – Basic Attention Mechanism

- Process – Attention Mechanism

- Steps

- ✓ (1) Transform decoder hidden state with linear map W_a .
- ✓ (2) Compare query with all keys → compute similarity.
- ✓ (3) Apply softmax to normalize scores into weights w .
- ✓ (4) Compute context vector cas weighted sum of values.

- Query (Q): current decoder hidden state h_t^{dec}
- Keys (K): encoder hidden states $\{h_1^{enc}, \dots, h_m^{enc}\}$
- Values (V): usually the same as encoder hidden states



Motivation & Bridge from Last Lecture

- Recap – Basic Attention Mechanism

- Step 1. Linear Transformation (Decoder side)



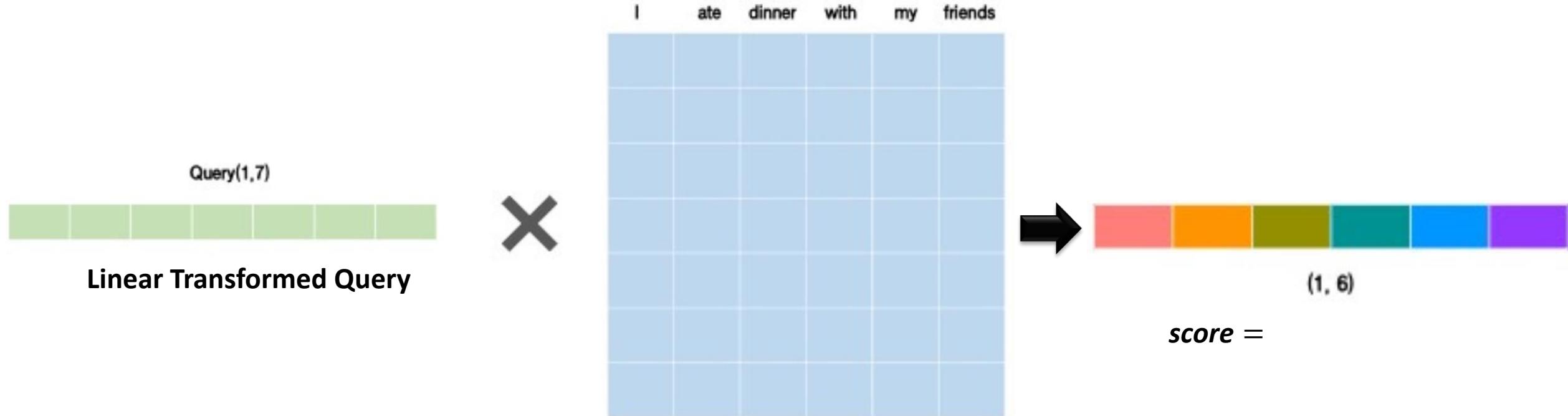
- The decoder hidden state h_t^{dec} has dimension **hidden size = 7**.
- Apply a linear transformation using matrix $W_a \in \mathbb{R}^{7 \times 7}$.
- This produces the **Query vector**

$$Q =$$

- Shape: (1, 7)

Motivation & Bridge from Last Lecture

- Recap – Basic Attention Mechanism
 - Step 2. Query–Key Similarity



- Encoder hidden states $h_{1:m}^{enc}$ act as Keys.
 - Compute similarity between Query and each Key
- score** =
- Shape: (1, m), where m = number of tokens in the source sentence.

Motivation & Bridge from Last Lecture

- Recap – Basic Attention Mechanism

- Step 3. Attention Weights (Softmax)

- Apply softmax to the similarity scores



$score =$

$w =$

- These weights represent the **relevance of each source token** to the current decoder step.

- Step 4. Context Vector Construction

- Multiply weights with encoder **hidden states (Values)** and sum them up

$c =$

- This **context vector** encodes the most relevant information from the source sequence for predicting the next word.

Motivation & Bridge from Last Lecture

- Recap – Basic Attention Mechanism

- Summary – Attention Mechanism with Equations

- Attention Weights, Context Vector, and Decoder Output

$w =$

$c =$

✓ $c \in \mathbb{R}^{batch \times 1 \times hidden_size}$: context vector

✓ $W_a \in \mathbb{R}^{hidden_size \times hidden_size}$: linear transform

$\tilde{h}_t^{dec} =$

$\hat{y}_t =$

✓ Combine decoder hidden state and context vector.

✓ Transform back to hidden size with W_{concat} .

✓ Generate prediction with softmax over vocabulary.

Motivation & Bridge from Last Lecture

■ Recap – Basic Attention Mechanism

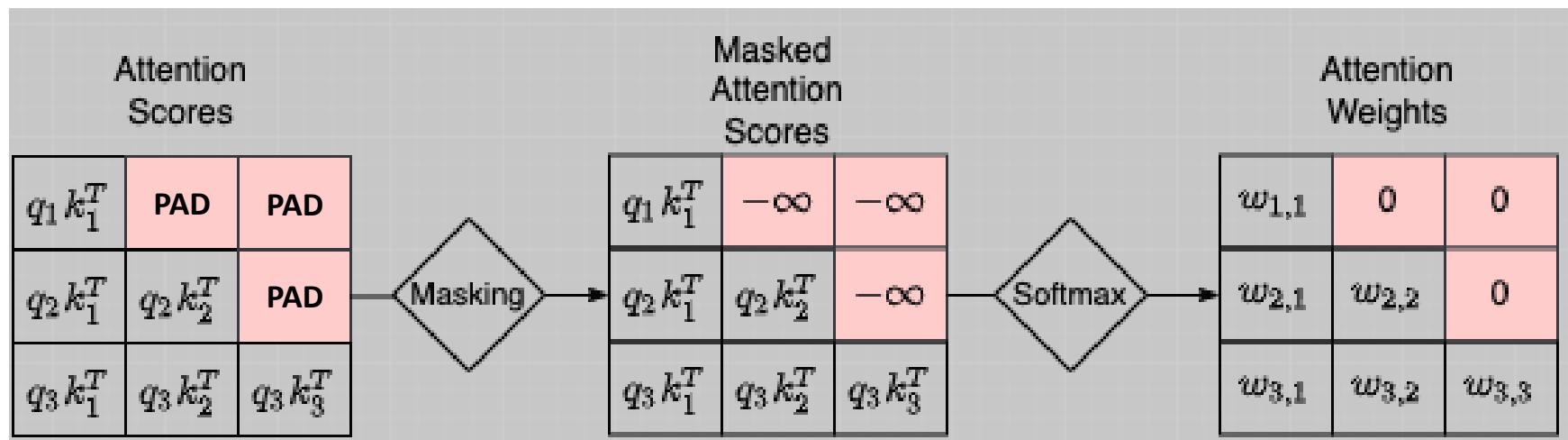
- Masking in Attention

- Problem

- ✓ Sentences in a batch have different lengths. → Shorter sentences padded with <PAD> tokens.
→ If <PAD> contributes to attention, it introduces noise.

- Solution

- ✓ Apply mask → replace <PAD> positions with $-\infty$ before softmax.
 - ✓ After softmax, weights for <PAD> become exactly 0.



Transformer Architecture

■ Architecture Overview

- The Transformer has an **Encoder–Decoder structure**, each built as a **stack of N identical layers** (originally $N = 6$).

- Encoder layer

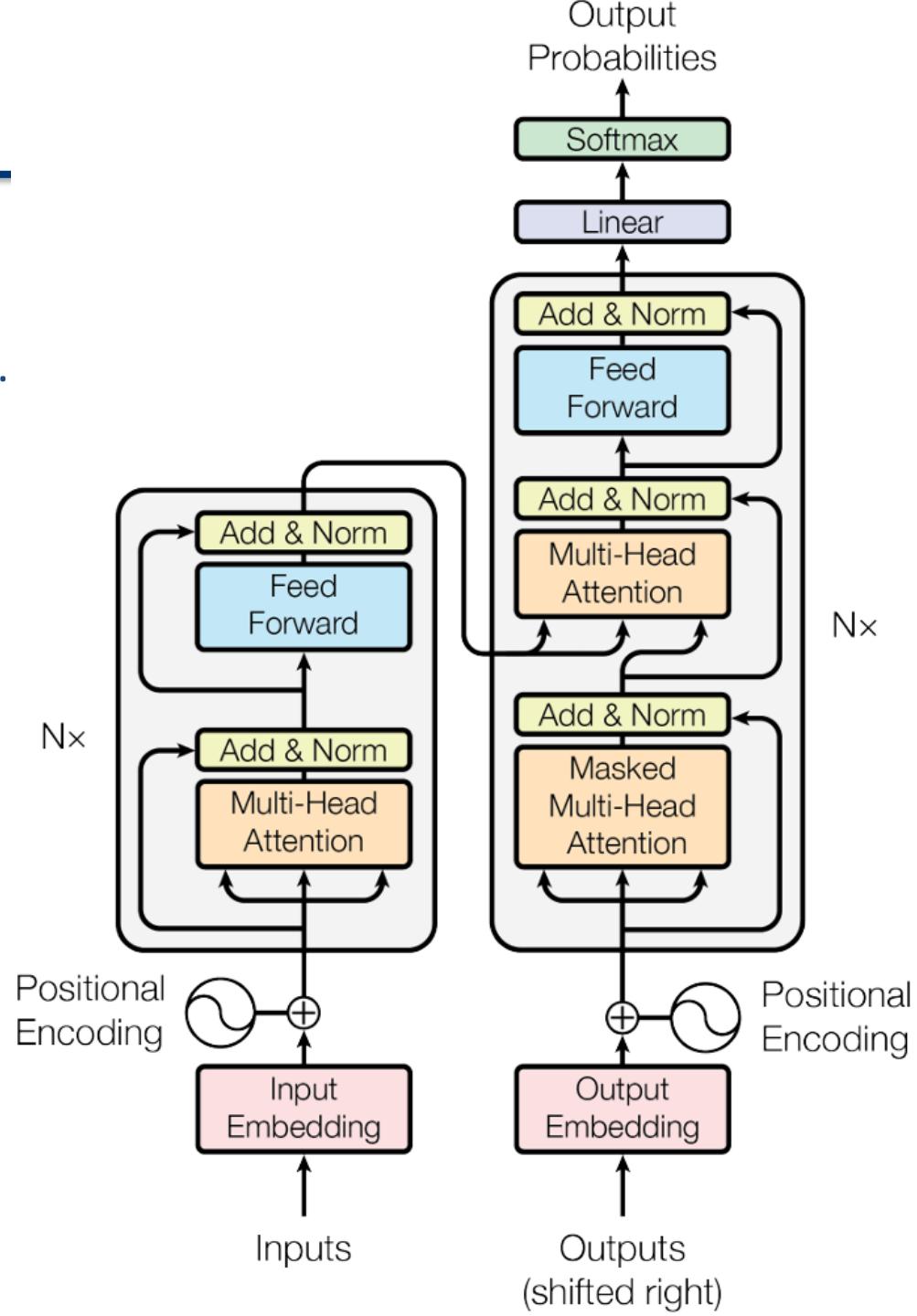
- Attention Type 1 – Multi-Head Self-Attention (MHSA)
 - Feed-Forward Network (FFN)
 - Residual connection & LayerNorm around each sublayer

- Decoder layer

- Attention Type 2 – Masked Multi-Head Self-Attention
 - Attention Type 3 – Encoder–Decoder Attention
 - Feed-Forward Network
 - Residual connection & LayerNorm around each sublayer

- Base model sizes

- $d_{model} = 512$, $d_{ff} = 2048$, $h = 8$



Transformer – Key Components

- **Attention Type 1** – Multi-Head Self-Attention (MHSA)

- **Self-Attention**

- Before we dive into Multi-Head Attention, let's carefully understand how **Self-Attention** works.

- **Problem**

- ✓ Traditional models like RNNs and CNNs struggle when sequences get long

- RNNs may forget earlier words due to the fixed hidden state.

- CNNs capture local patterns well, but long-range dependencies are harder to learn.

- **Goal**

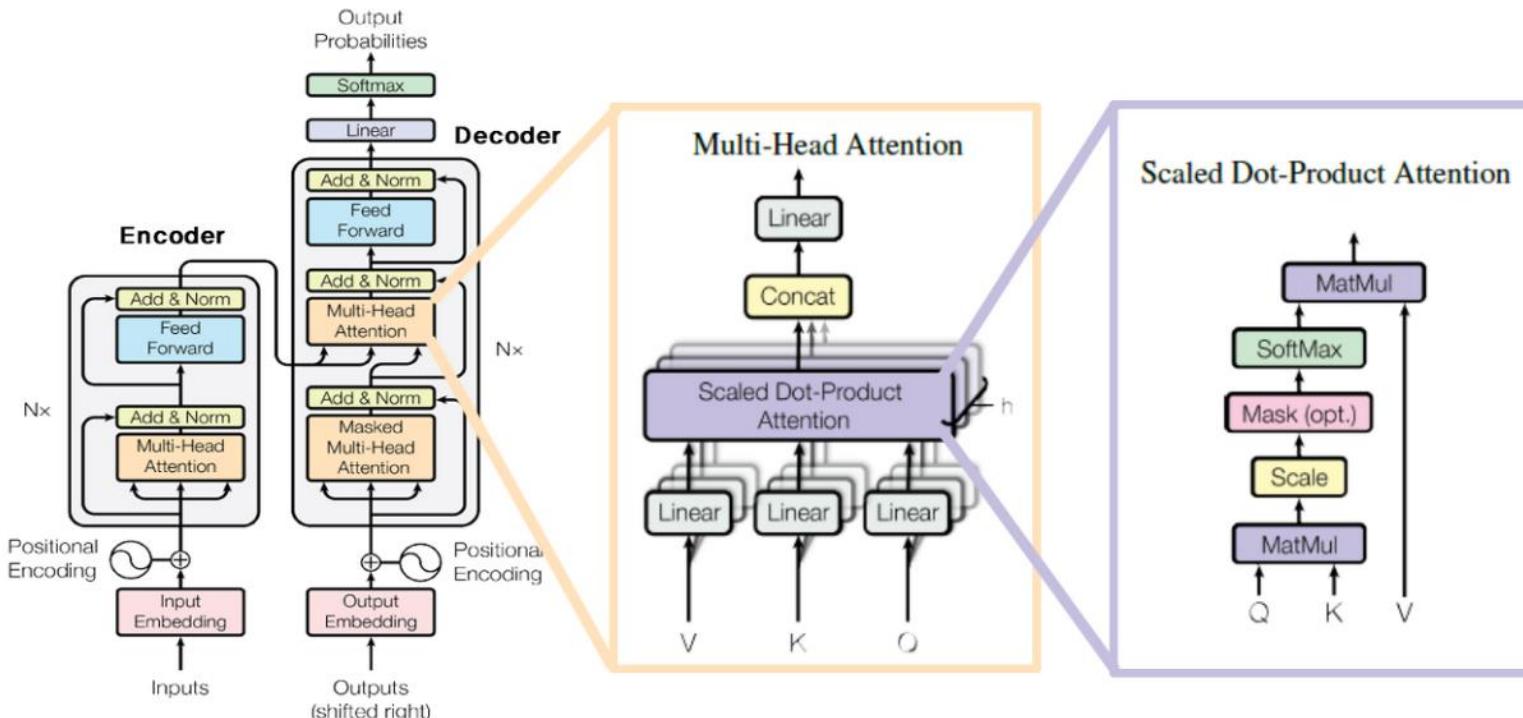
- ✓ Determine **how much each word in a sequence should attend to the other words.**

Transformer – Key Components

▪ Attention Type 1 – Multi-Head Self-Attention (MHSA)

- **Self-Attention**

- Self-Attention solves this by allowing **every word to directly attend to every other word** in the sequence.
 - ✓ It answers the question: “*Which words should this word pay attention to, and by how much?*”



- **Example**

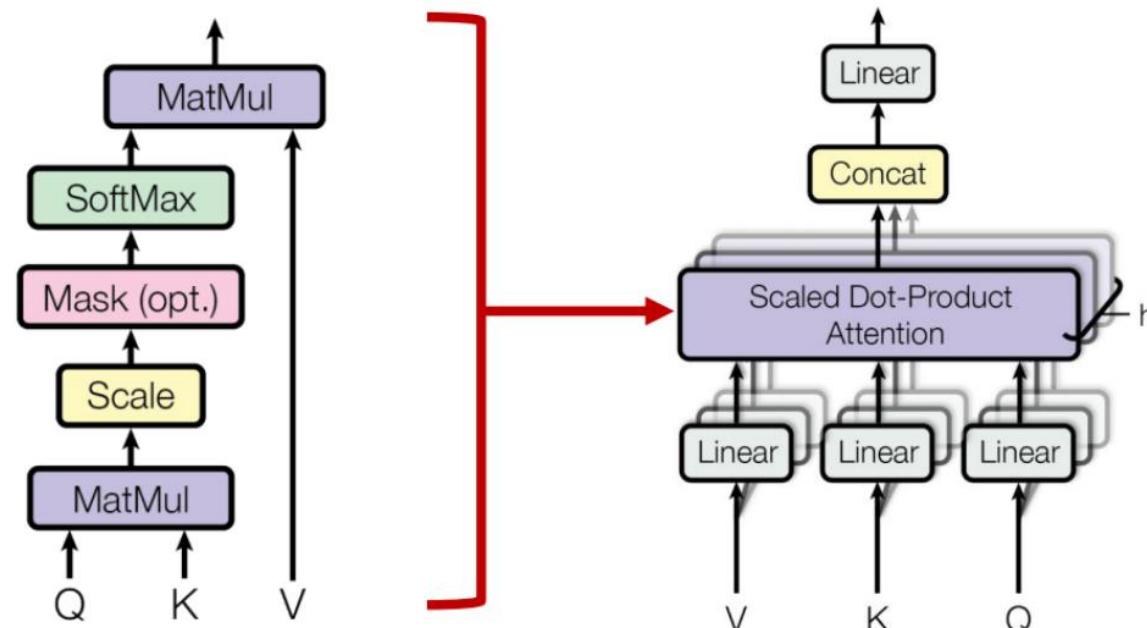
- ✓ In “*I love you*”, the representation of “I” is updated by looking at both “love” and “you,” weighted by their importance.

Transformer – Key Components

▪ Attention Type 1 – Multi-Head Self-Attention (MHSA)

- **Queries, Keys, and Values**

- To compute attention, we first generate three sets of vectors from our input embeddings



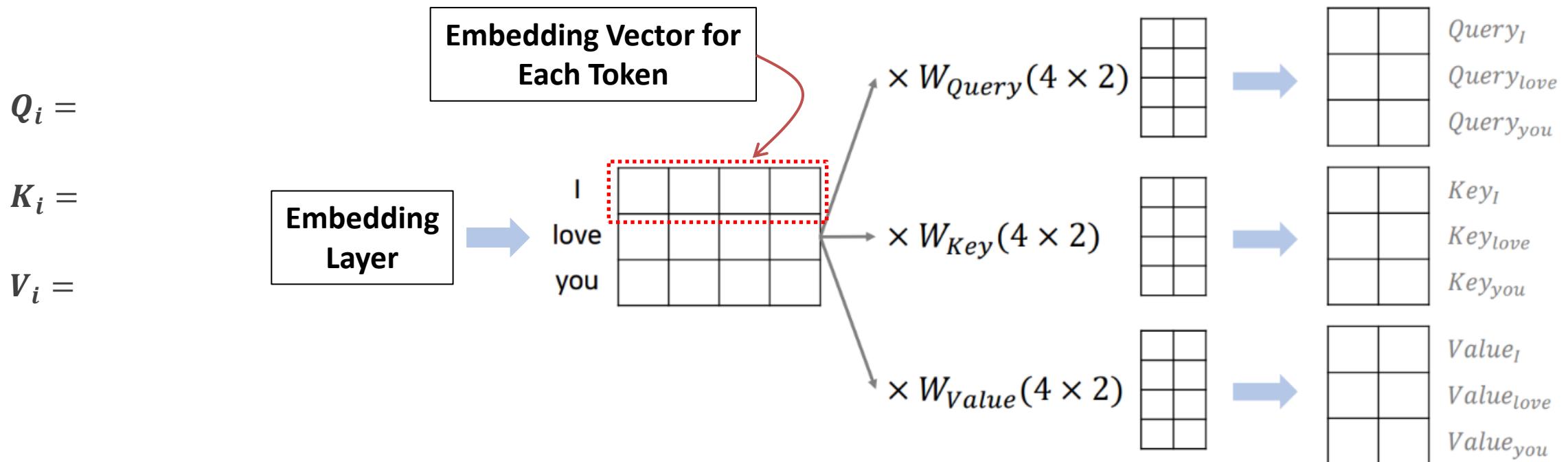
- ✓ **Query (Q):** Represents the word we are focusing on – What am I looking for?
- ✓ **Key (K):** Represents how relevant each word is when queried – What do I contain that might be useful to others?
- ✓ **Value (V):** Contains the actual information that will be combined to produce the attention output – The actual information to be passed along.

Transformer – Key Components

▪ Attention Type 1 – Multi-Head Self-Attention (MHSA)

- **Queries, Keys, and Values**

- Each word embedding x_i (e.g., 4-dimensional vector) is projected into three different spaces



✓ The weight matrices W_{Query} , W_{Key} , W_{Value} are **learnable parameters**.

✓ They are **linear transformations**, not embeddings.

➤ They map the same input into different roles for computing attention.

Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

• Scaled Dot-Product Attention

- The formula for attention is

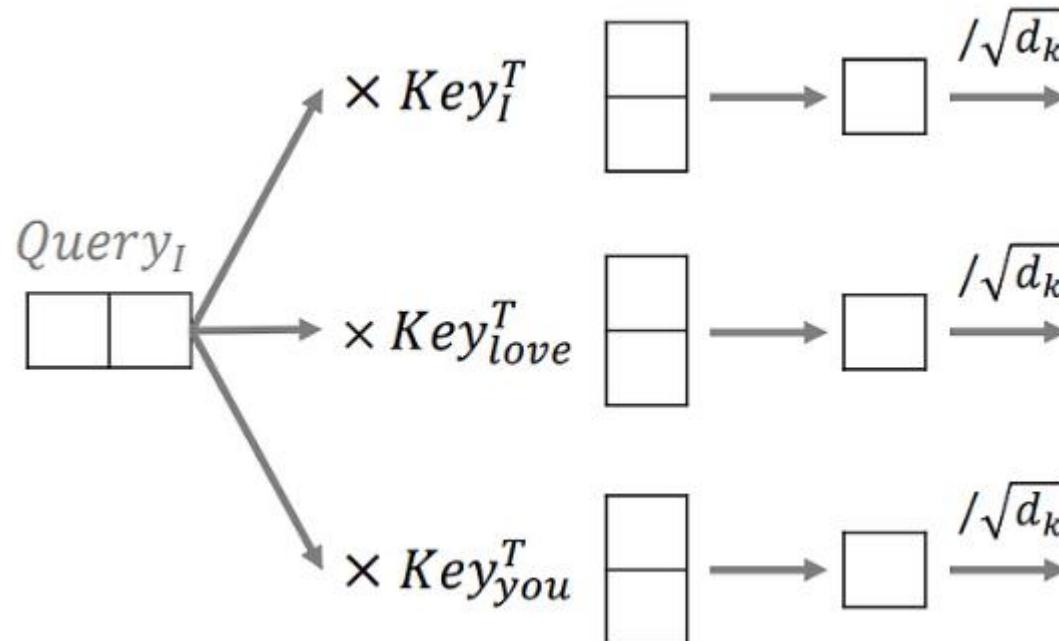
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Step 1: Similarity

- ✓ Compute the dot product between a Query and all Keys.
 - ✓ This measures how relevant each word is to the Query word.

- Step 2: Scaling

- ✓ Divide by $\sqrt{d_k}$ to keep values small.
 - ✓ Prevents softmax from becoming too sharp when d_k is large.

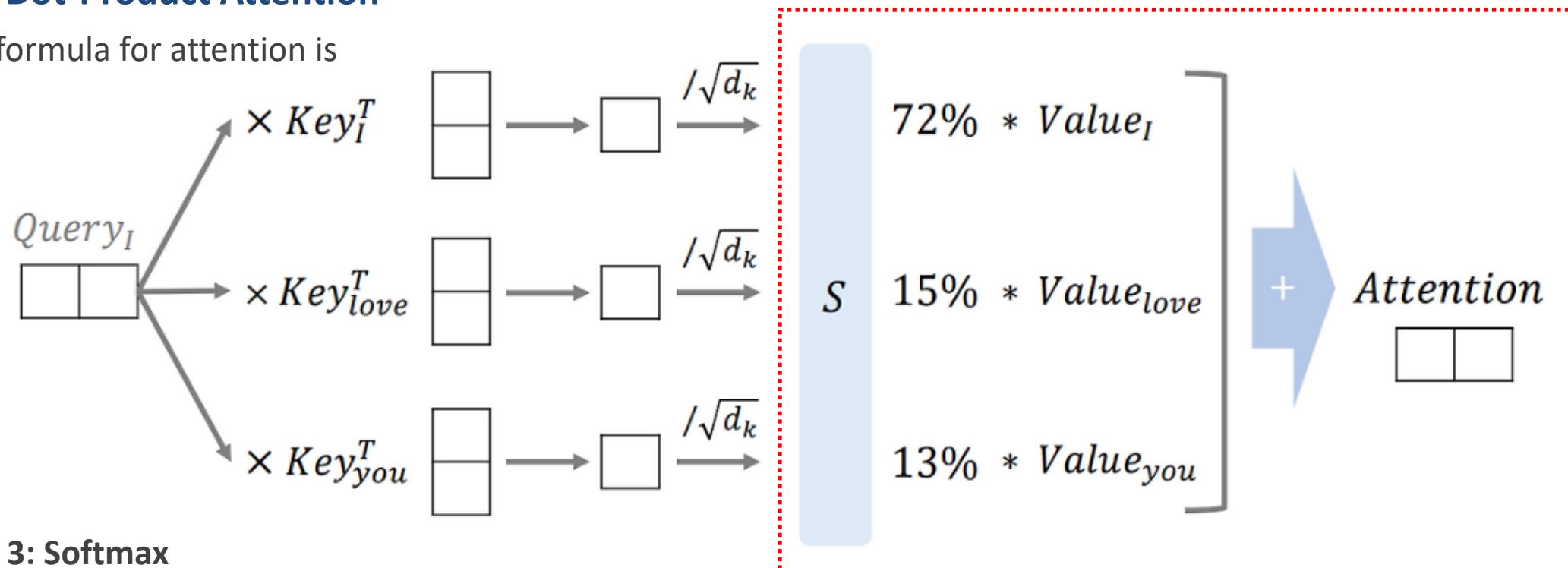


Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

• Scaled Dot-Product Attention

- The formula for attention is



- Step 3: Softmax

- ✓ Convert similarity scores into probabilities (attention weights).

- Step 4: Weighted sum

- ✓ Multiply each Value vector by its attention weight.
 - ✓ The result is the new representation for the word.

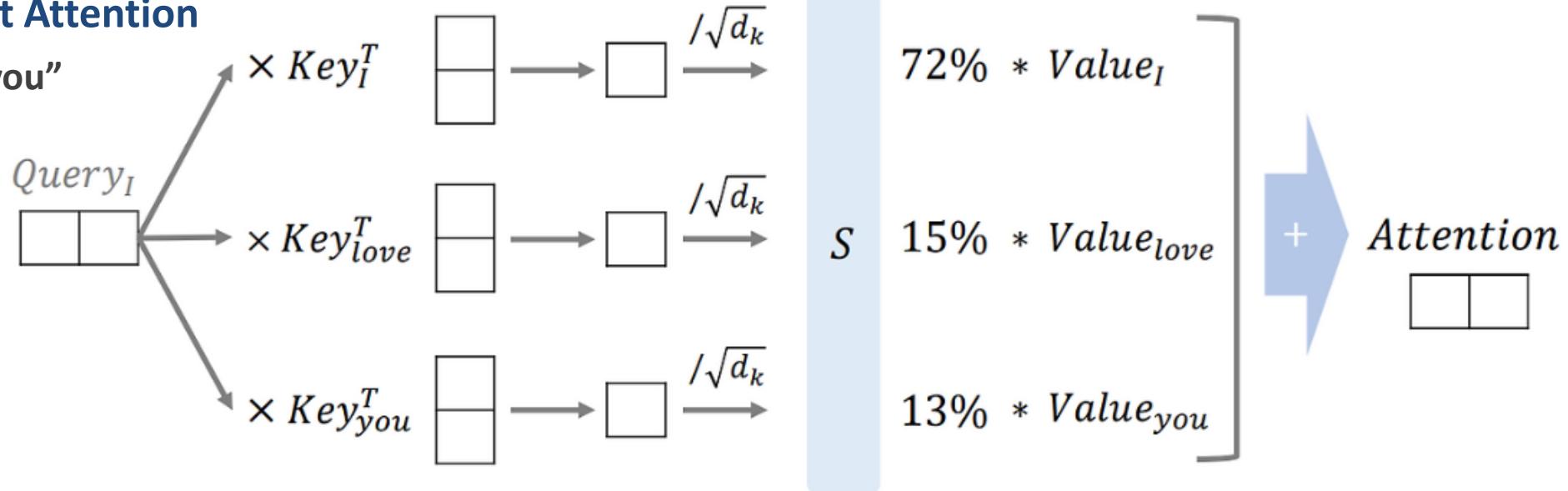
$$\text{Attention}(Q, K, V) =$$

Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

• Scaled Dot-Product Attention

- Example: “I love you”



✓ Suppose we focus on the Query for “I”

- Dot product with Key_I → strong similarity → weight = 72%
- Dot product with Key_{love} → moderate → weight = 15%
- Dot product with Key_{you} → weaker → weight = 13%

✓ Final output for “I”: $0.72 \cdot V_I + 0.15 \cdot V_{love} + 0.13 \cdot V_{you}$

✓ The word “I” is no longer just its original embedding.

It is now a **contextual embedding** that captures relationships with “love” and “you.”

Transformer – Key Components

- **Attention Type 1** – Multi-Head Self-Attention (MHSA)

- Scaled Dot-Product Attention

- Why Scaling is Necessary

- ✓ Without scaling, the dot products can be very large if d_k is large.

- When we compute self-attention, we use the dot product between a Query and a Key

$$s =$$

- If the dimension d_k is large, this dot product tends to grow in magnitude.

- This can cause the softmax function to produce extremely peaked distributions, leading to very small gradients and unstable training.

- ✓ Large values → softmax outputs are close to 0 or 1 → gradients vanish.

- ✓ Scaling by $\sqrt{d_k}$ stabilizes training and ensures smoother gradients.

Transformer – Key Components

- **Attention Type 1** – Multi-Head Self-Attention (MHSA)

- Scaled Dot-Product Attention
 - Why Scaling is Necessary
 - ✓ Variance Analysis of the Dot Product

➤ Let us assume

- Each element q_i and k_i is drawn from an independent distribution with mean 0 and variance 1.
 - That is, $= \mathbf{0}$ and $= \mathbf{1}$.

➤ Then the variance of the dot product can be computed as

$$\text{Var}(q \cdot k) =$$

➤ Since q_i and k_i are independent

$$\text{Var}(q \cdot k) =$$

=

=

Transformer – Key Components

- **Attention Type 1** – Multi-Head Self-Attention (MHSA)

- Scaled Dot-Product Attention

- Why Scaling is Necessary

- ✓ Variance Analysis of the Dot Product

- The variance of the dot product grows **linearly with d_k** .
 - When d_k is large, dot products become very large in magnitude.
 - Feeding these large values directly into the softmax makes the output distribution very sharp (close to one-hot).
 - **As a result, gradients become extremely small, which slows down or destabilizes training.**

- The Scaling Solution

- ✓ To address this, the Transformer introduces a **scaling factor**

$$s =$$

- ✓ By dividing by $\sqrt{d_k}$, we normalize the variance

$$\text{Var}\left(\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}\right) = \quad = \quad =$$

- This ensures that regardless of the dimensionality d_k , the variance of the scaled dot product remains stable (≈ 1).

Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

- **From Self-Attention to Multi-Head Attention**

- So far, we learned that **self-attention** allows each word to attend to all the other words in a sequence.
 - But the original Transformer goes a step further by introducing **Multi-Head Attention**.

- **Why Do We Need Multiple Heads?**

- A **single attention head** projects Queries, Keys, and Values into one subspace.
 - This limits the variety of relationships it can capture.
 - By using **multiple attention heads**, the model can learn to focus on different aspects of relationships **in parallel**:
 - ✓ One head may capture **syntactic structure** (e.g., subject–verb relationships).
 - ✓ Another may capture **semantic meaning** (e.g., word similarity).
 - ✓ Another may capture **positional dependencies**.
 - In short, multiple heads allow the model to **look at the sequence from different perspectives** at the same time.

Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

• How Multi-Head Attention Works

- For each head i , we apply separate linear projections

$$\text{head}_i =$$

- where each projection matrix has size

$$W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$$

✓ d_{model} : input embedding dimension (e.g., 512)

✓ h : number of heads (e.g., 8)

✓ $d_k = d_v = \frac{d_{\text{model}}}{h}$ (e.g., 64)

- Each head computes its own attention output independently.

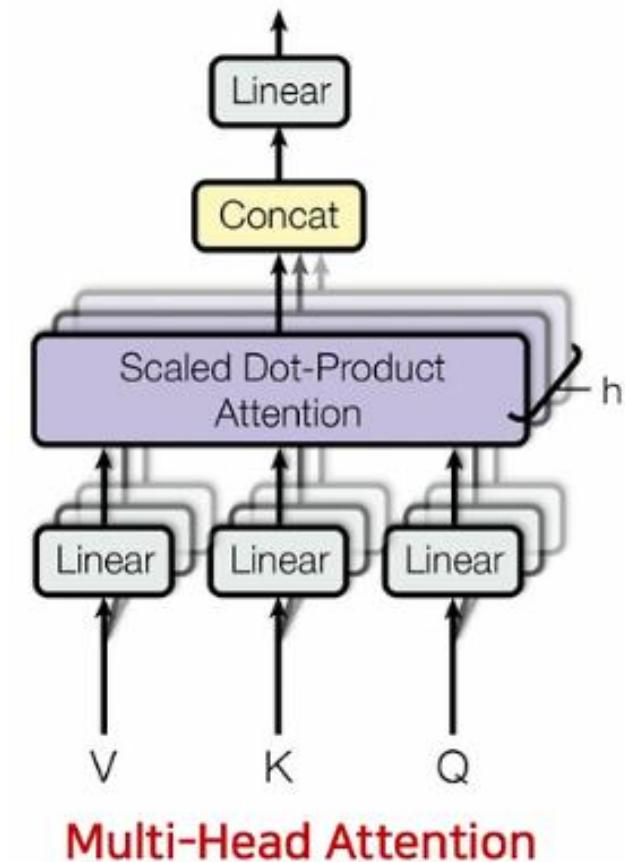
• Combining the Heads

- Once all heads are computed

$$\text{MultiHead}(Q, K, V) =$$

- The outputs of all heads are concatenated along the feature dimension.

- Then, a final projection $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ brings the dimension back to d_{model} .



Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

• The Role of Multi-Head Attention – Different Heads, Different Focus

- The following examples illustrate that different attention heads specialize in different roles

- **1. Sentence Type & Structure Attention**

- ✓ Some heads focus on functional words and punctuation, helping the model understand the **overall sentence form**.

- **2. Noun-Focused Attention**

- ✓ Other heads attend strongly to nouns, helping the model capture **entities** in the sentence.

- **3. Relation Attention**

- ✓ Certain heads capture **relations** between words, such as verb-object or subject-predicate dependencies.

- **4. Sentiment or Emphasis Attention**

- ✓ Some heads highlight emotionally charged words or intensifiers, focusing on the **tone of the sentence**.

Which do you like better, coffee or tea?

Transformer – Key Components

- **Attention Type 1** – Multi-Head Self-Attention (MHSA)

- The Role of Multi-Head Attention – Different Heads, Different Focus

- **Interpretability of Attention**

- ✓ Another advantage of attention is **interpretability**.
 - ✓ By visualizing attention weights, we can see **which words influence the prediction the most**.
 - ✓ Different colors in the visualization represent different heads.
 - ✓ This reveals that each head is not redundant but instead complements others.

Transformer – Key Components

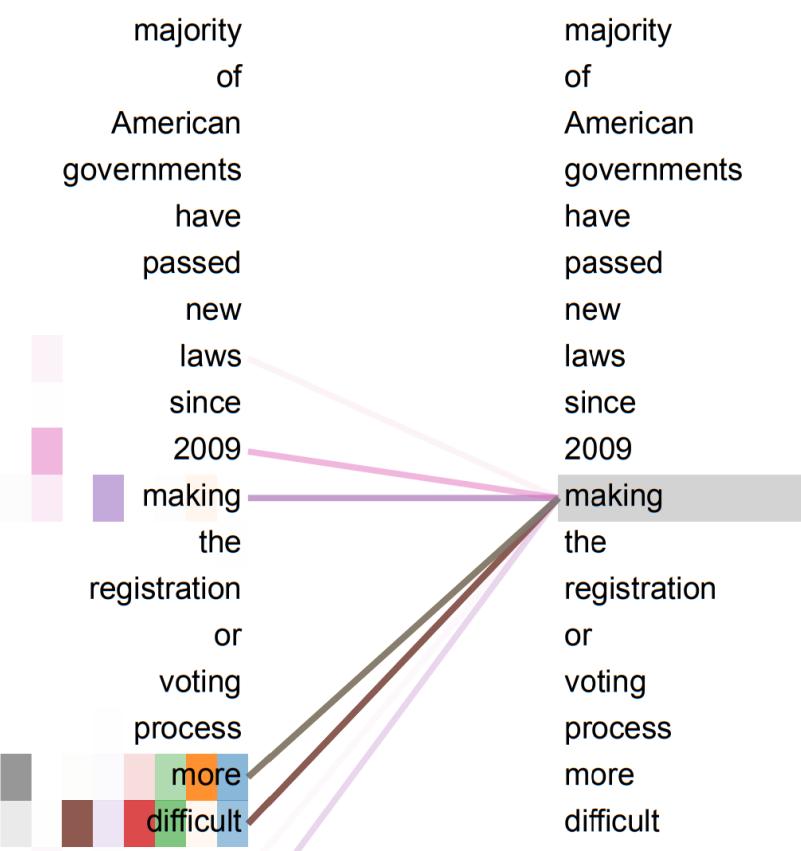
■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

- The Role of Multi-Head Attention – Different Heads, Different Focus

- Case Studies - Interpretability of Attention

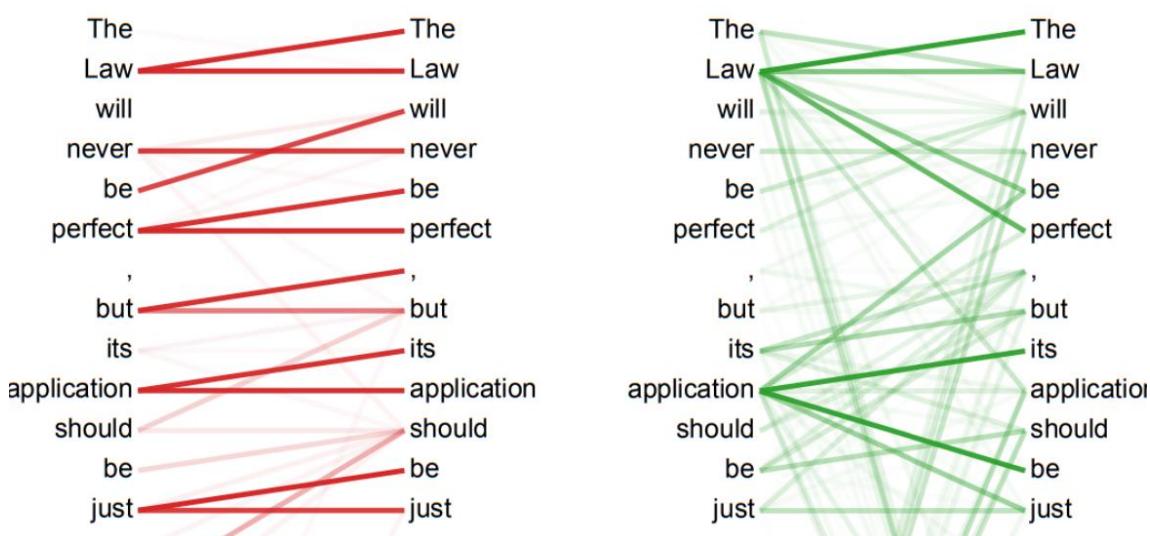
- ✓ 1. Long-Distance Dependency (Making example)**

- In one head, the verb “making” attends to words far earlier in the sentence.
 - This shows how the model captures **long-range dependencies** that RNNs struggle with.



- ✓ 2. Sentence Structure (Parallel Examples)**

- Some heads align multiple words with their repeated counterparts in translation or generation.
 - This reflects the model’s ability to learn **syntactic alignment**.



Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

- Multi-Head Attention in the Encoder

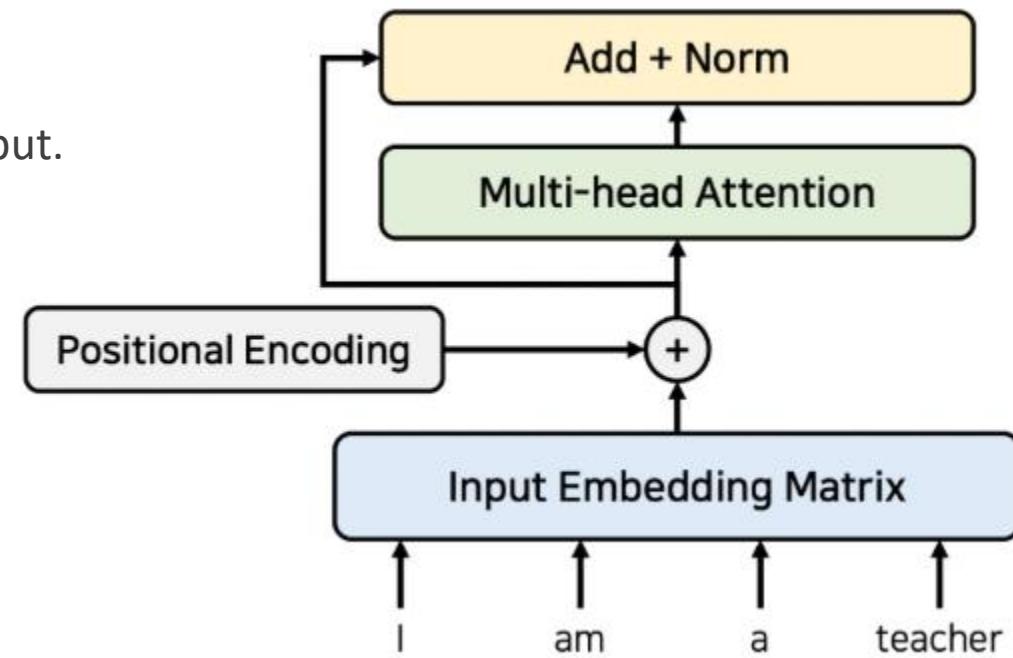
- Residual Connections in Transformers

- ✓ In the Transformer encoder, each sub-layer (such as **Multi-Head Attention** or the **Feedforward Network**) is wrapped with a residual connection

- 1. Input goes into the sub-layer.
 - 2. Output of the sub-layer is added back to the original input.
 - 3. This sum is then normalized (LayerNorm).

- Why Do We Use Residual Learning Here?

- ✓ It makes training more stable for deep architectures (Transformers often use 6–12 layers or more).
 - ✓ It allows the original input information to **flow through the network unaltered**, even if the sub-layer changes only a little.
 - ✓ It improves gradient flow during backpropagation, making optimization easier.



Transformer – Key Components

■ Attention Type 1 – Multi-Head Self-Attention (MHSA)

• Multi-Head Attention in the Encoder

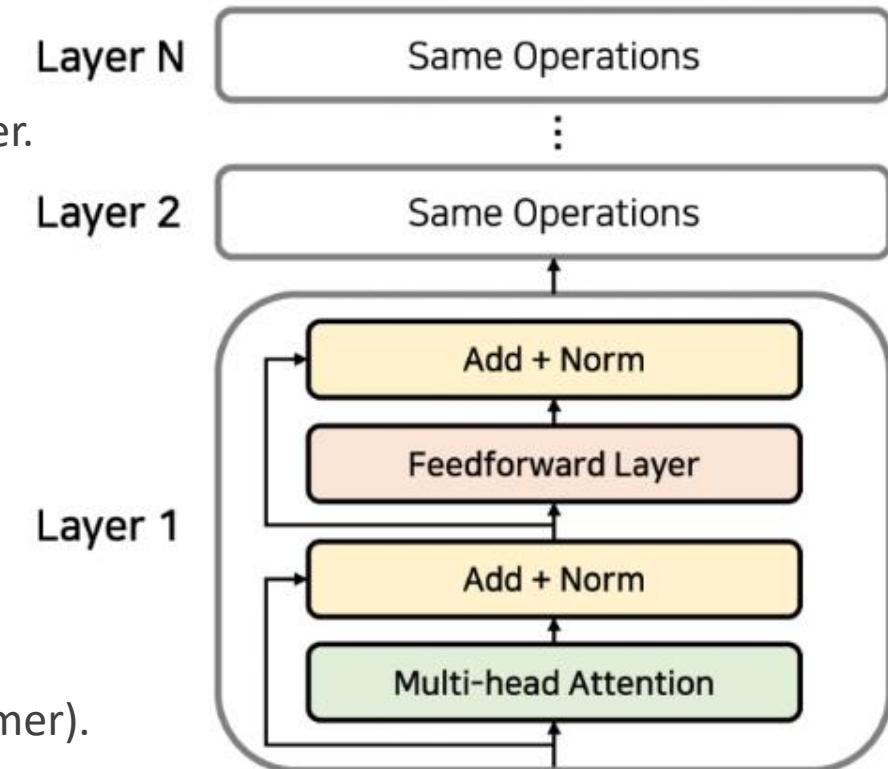
- In the Transformer Encoder, Multi-Head Attention is used in every layer.

- Each Encoder layer consists of
 - ✓ 1. Multi-Head Self-Attention
 - ✓ 2. Add & Norm (Residual Connection + Layer Normalization)
 - ✓ 3. Feedforward Network
 - ✓ 4. Another Add & Norm

- These layers are stacked **N times** (e.g., 6 layers in the original Transformer).

- The input embeddings are enriched step by step as they pass through each stacked layer.

- Importantly, **self-attention in the encoder** lets each word look at all the other words in the same sentence, helping the encoder build contextualized word representations.



Transformer – Key Components

▪ Attention Type 2 – Masked Multi-Head Self-Attention

- Motivation

- The **goal** of Decoder Self-Attention is the same as in the Encoder
→ To capture relationships between tokens in the sequence.

- Difference

- Encoder operates on the **input sentence** (all tokens available).
 - Decoder operates on the **output sentence** (generated tokens so far).

→ This means the decoder must ***not look at future words*** — otherwise it would be cheating.

- Auto-Regressive Property

- In sequence generation, the model predicts words **one by one**.
 - At time step t , the prediction must depend **only on words before t** .

- Example

- ✓ Query = “I”, Key = [“I”]
 - ✓ Query = “love”, Key = [“I, love”]
 - ✓ Query = “you”, Key = [“I, love, you”]

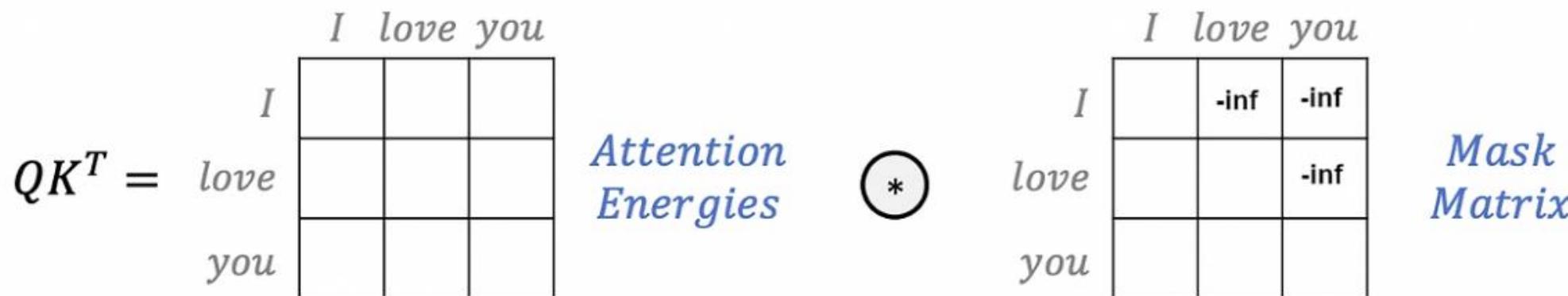
→ The decoder must not use information from future tokens when predicting the next word.

Transformer – Key Components

■ Attention Type 2 – Masked Multi-Head Self-Attention

- **Masking Mechanism**

- To prevent access to future tokens, we use a **mask matrix**.
- Mask values are set to $-\infty$ for forbidden positions.
- When passed through the **softmax function**, these positions become 0, meaning they are ignored.



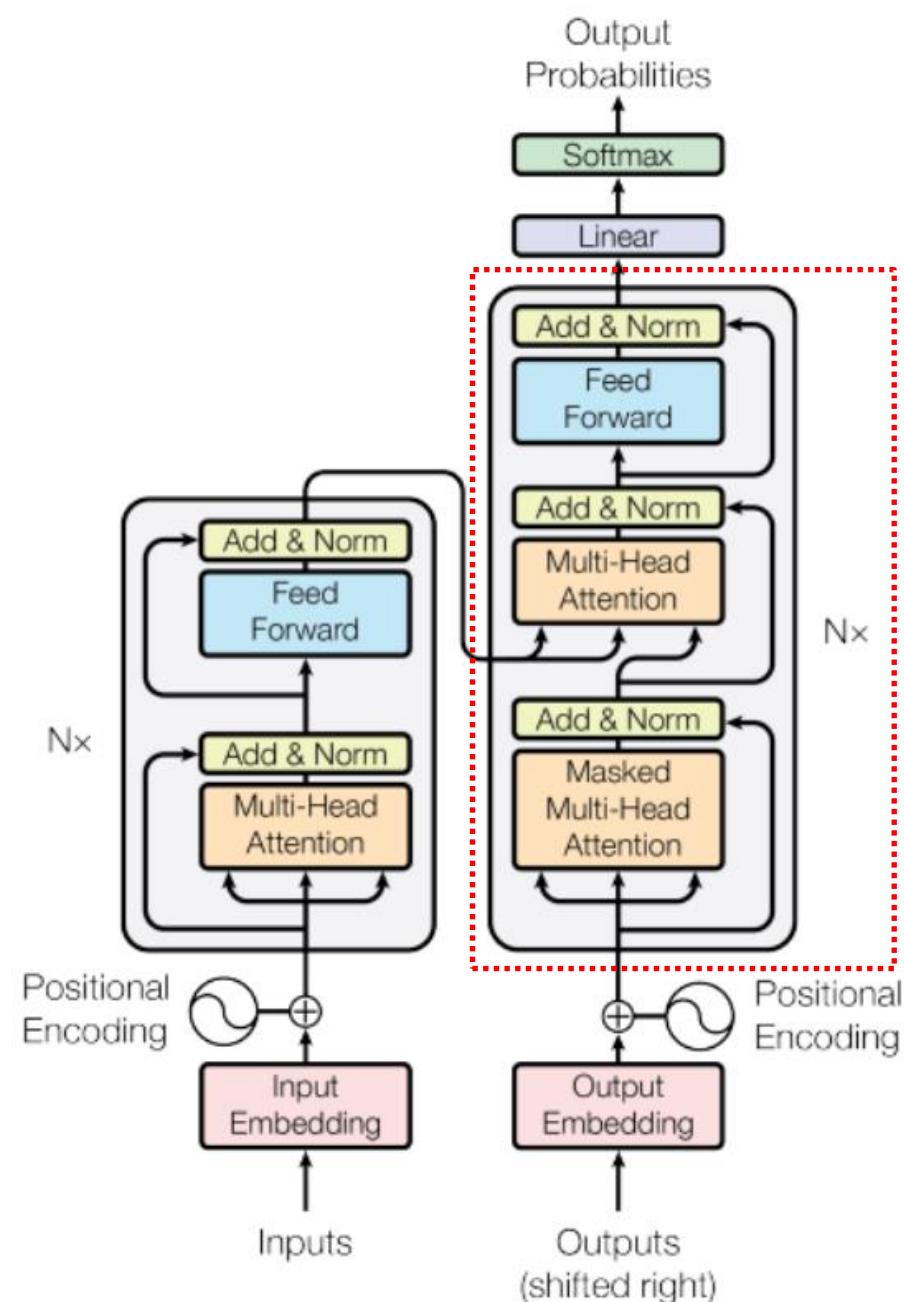
→ This ensures that each token only attends to itself and earlier tokens.

Transformer – Key Components

▪ Attention Type 3 – Encoder–Decoder Attention

• What is Encoder–Decoder Attention?

- In the decoder, there are **two types of attention**
 - ✓ **Masked Self-Attention** (prevents looking at future words).
 - ✓ **Encoder–Decoder Attention** (focuses on the input sequence).
- In **Encoder–Decoder Attention**
 - ✓ **Queries (Q)** come from the decoder.
 - ✓ **Keys (K)** and **Values (V)** come from the encoder.
- This allows the decoder to decide **which parts of the source sentence are most relevant** when generating each target word.



Transformer – Key Components

■ Attention Type 3 – Encoder–Decoder Attention

- Why Do We Need It?

- Self-attention in the encoder learns contextual representations of the input sentence.
- The decoder needs to know: “*When generating this word, which input words should I focus on?*”
- Encoder–Decoder Attention provides this mechanism by linking each decoder word to the encoder’s representations.

- Example

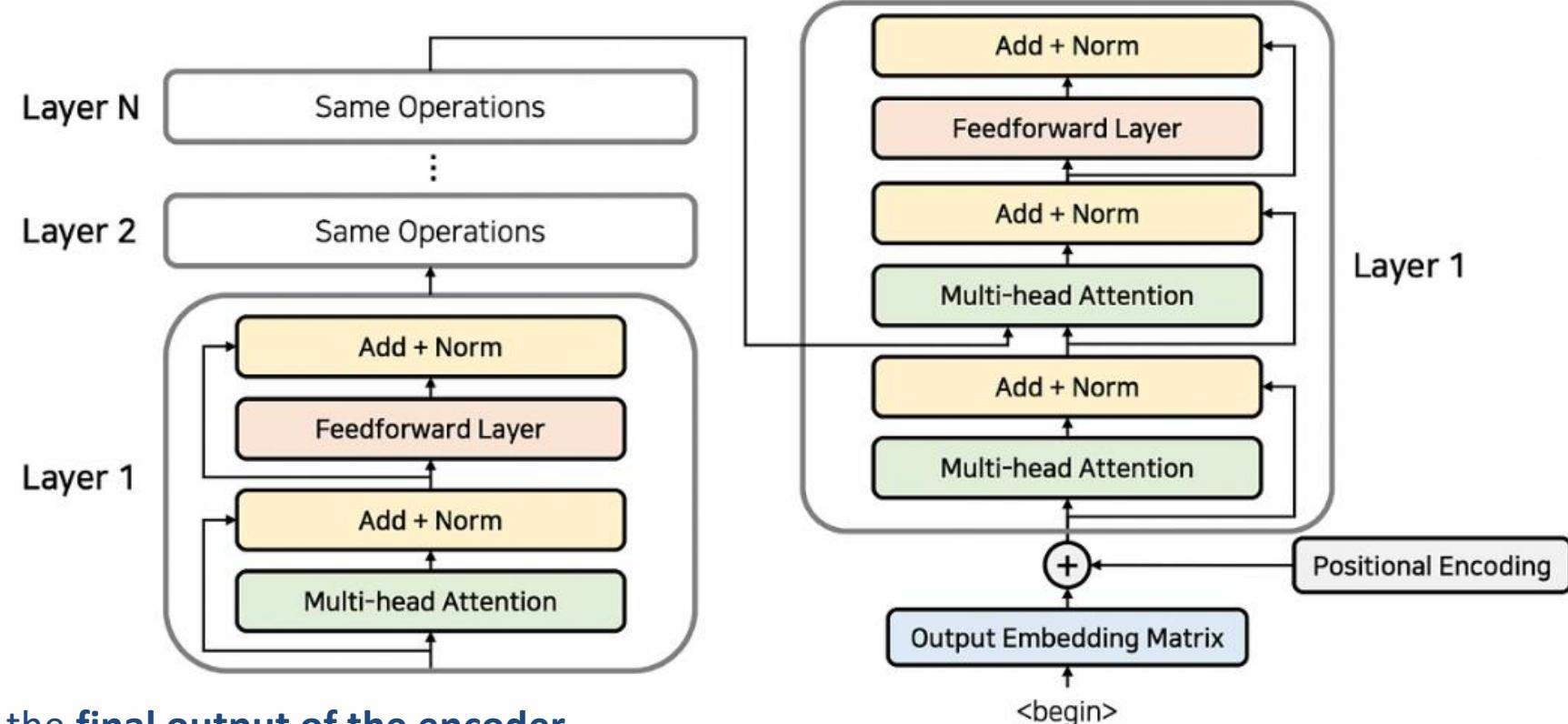
- When translating “*guten abend*” → “*good evening*”
 - ✓ The decoder word “*good*” should attend strongly to “*guten*.“
 - ✓ The decoder word “*evening*” should attend strongly to “*abend*.“

Transformer – Key Components

■ Attention Type 3 – Encoder–Decoder Attention

• How It Works

- At each decoding step



- ✓ 1. The decoder takes the **final output of the encoder**.
- ✓ 2. It computes attention scores between the decoder's Query and the encoder's Keys.
- ✓ 3. These scores weight the encoder's Values, producing a context vector.
- ✓ 4. This context vector guides the decoder in generating the next word.

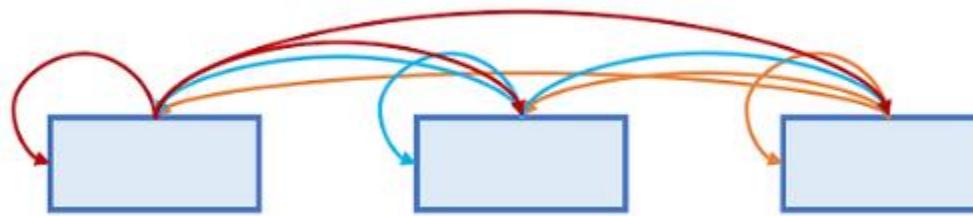
$$\text{Attention}(Q_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}})$$

Transformer – Key Components

- Summary – Three Types of Attention in the Transformer

- Attention Type 1 – Encoder Multi-Head Self-Attention (MHSA)

- Each word in the input sequence attends to **all other words** in the same sentence.
 - Purpose: Build **contextual embeddings** that capture global relationships.



→ *Global context inside the encoder.*

■ : Encoder

- Attention Type 2 – Decoder Masked Multi-Head Self-Attention (Masked MHSA)

- Used in the **decoder** when generating output.
 - Each word can only attend to **itself and previous words**.
 - Future words are **masked** to preserve the **auto-regressive property**.

■ : Decoder



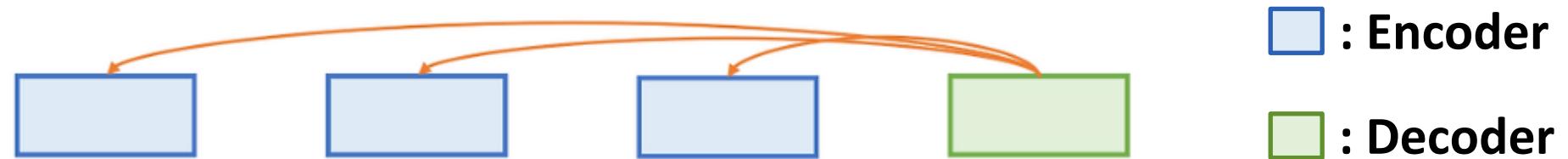
→ *Prevents cheating by blocking future tokens.*

Transformer – Key Components

- Summary – Three Types of Attention in the Transformer

- **Attention Type 3 – Encoder–Decoder Attention**

- Queries come from the **decoder**, while Keys and Values come from the **encoder**.
 - Allows the decoder to focus on the **most relevant parts** of the input sentence.



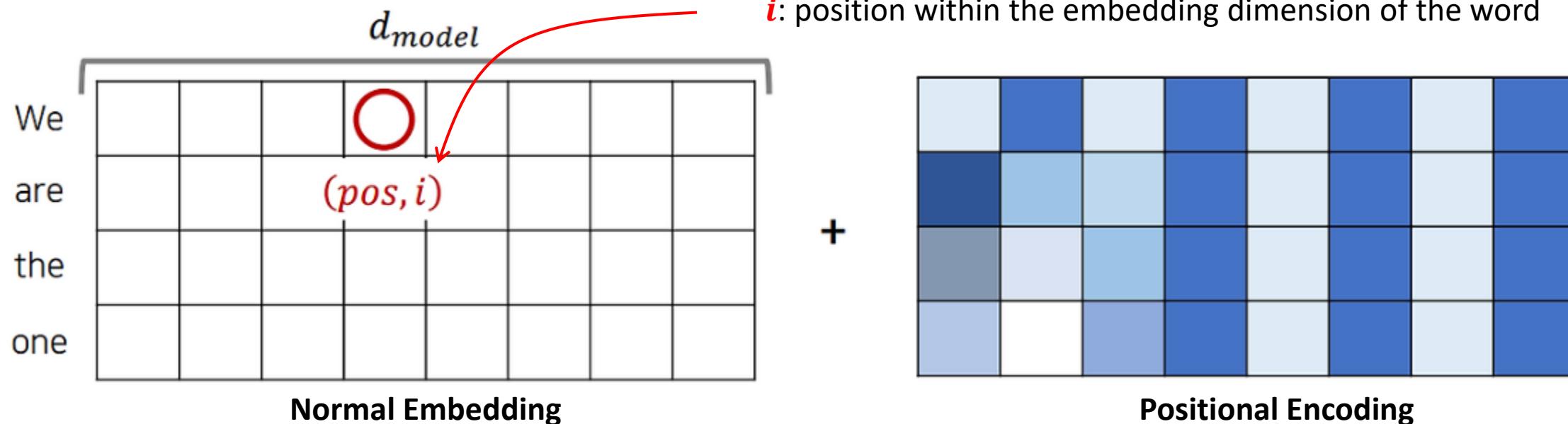
→ *Links the input sentence to the output generation.*

- *"Together, these three types of attention form the core of the Transformer architecture"*

Transformer – Other Components

▪ Positional Encoding

- Why Do We Need Positional Information?



- Unlike RNNs or LSTMs, **Transformers process the entire input sequence in parallel**.
- This parallelization is powerful but **loses the natural order of tokens** that RNNs inherently have.
- To give the model a sense of sequence order, we add **Positional Encoding** to input embeddings.

Transformer – Other Components

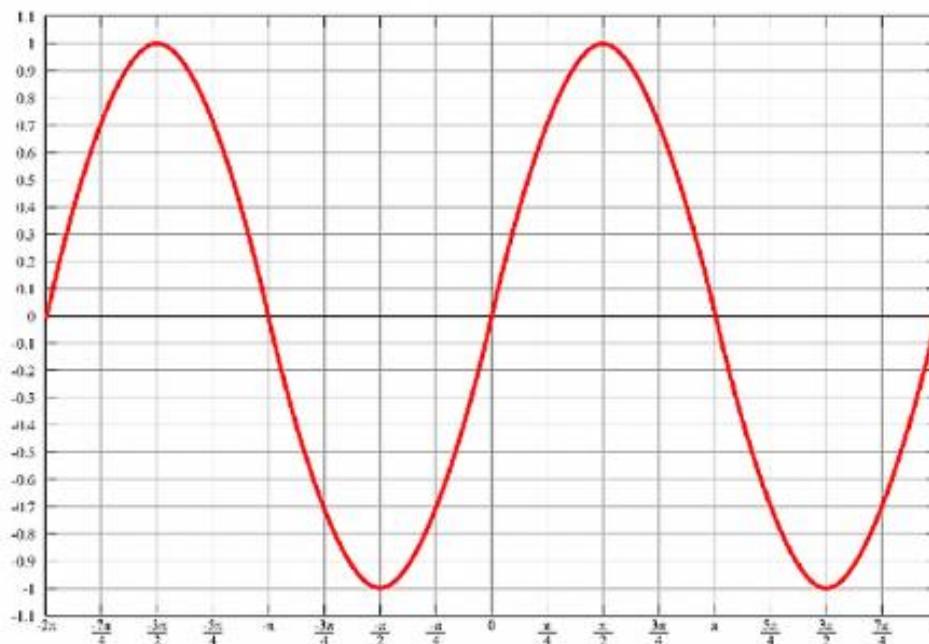
▪ Positional Encoding

- **The Idea of Positional Encoding**

- Each token embedding **does not contain position by itself**.
- We generate a **positional vector** that encodes the position of each token in the sequence.
- Then, we simply **add this positional vector** to the word embedding before feeding it into the Transformer.

$$PE_{(pos, 2i)} =$$

$$, \quad PE_{(pos, 2i+1)} =$$



$$PE_{(pos, 2i)} =$$

$$PE_{(pos, 2i+1)} =$$

Transformer – Other Components

■ Positional Encoding

• Why Use Sine and Cosine?

- Using integers (1, 2, 3, ...) as positional indices is unstable for training.

→ *As the sequence length grows, indices become very large, which makes learning difficult.*

- Using ratios in [0, 1] also fails.

✓ For example, the value 0.9 represents the 9th element in a sequence of length 10, but the 90th element in a sequence of length 100.

→ *Thus, the same value has different meanings depending on sequence length.*

- Using binary vectors (e.g., [0, 1, 0, 1, 0, 0, 0, 1]) can also cause problems.

→ *In high-dimensional spaces, distance metrics may treat them inconsistently, leading to incorrect similarity judgments.*

- Instead, sine and cosine functions provide **continuous, smooth, and periodic signals** that are stable across different sequence lengths.

Transformer – Other Components

■ Positional Encoding

- Why Use Sine and Cosine?

- Using both of sine and cosine.

→ If we only use sine (or only cosine), ***positions can overlap*** because the function ***repeats values periodically***.

- Properties of Positional Encoding

- Sine and cosine positional encodings are used because they satisfy four essential conditions:
 - **Uniqueness** – Each token position is assigned a unique value.
 - **Consistency of distance** – The relative distance between tokens is preserved.
 - ✓ e.g., the difference between token 1 and token 2 is the same as between token 2 and token 3.
 - **Scalability** – The encoding can generalize to sequences longer than those seen during training, without causing errors.
 - **Predictability** – Since sine and cosine are deterministic functions, the position values can always be reconstructed.

Transformer

■ Why Self-Attention? – From Quantitative Analysis

- Traditional RNNs, LSTMs, and Seq2Seq models have been powerful, but **Transformers replaced them with Self-Attention**. Why?
- There are three main reasons.
 - 1. Computational Complexity per Layer

n : Sequence length, d : Representation dimension, k : kernel size

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

- ✓ **RNNs:** Complexity = $O(n \cdot d^2)$, because computation is sequential across all tokens.
- ✓ **Self-Attention:** Complexity = $O(n^2 \cdot d)$.
- ✓ When the sequence length n is not too large compared to the representation dimension d , Self-Attention is more efficient.
- ✓ For natural language tasks with vocab sizes in the thousands, representation dimensions like 256 or 512 are typical. In such cases, Self-Attention is often computationally better.

Transformer

■ Why Self-Attention? – *From Quantitative Analysis*

- Traditional RNNs, LSTMs, and Seq2Seq models have been powerful, but **Transformers replaced them with Self-Attention**. Why?
- **There are three main reasons.**
 - 2. Amount of Computation

n : Sequence length, d : Representation dimension, k : kernel size

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

- ✓ **RNNs:** Must process tokens sequentially — computation takes $O(n)$.
- ✓ **Self-Attention:** Can process all positions **in parallel**, because all Attention Scores can be computed at once.
- ✓ This parallelism makes Self-Attention significantly faster in modern GPU/TPU systems.

Transformer

■ Why Self-Attention? – *From Quantitative Analysis*

- Traditional RNNs, LSTMs, and Seq2Seq models have been powerful, but **Transformers replaced them with Self-Attention**. Why?
- There are three main reasons.
 - 3. Path Length for Long-Range Dependencies

n : Sequence length, d : Representation dimension, k : kernel size

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

- ✓ RNNs struggle with **long-range dependencies** because information must flow through many recurrent steps.
 - Maximum Path Length = $O(n)$.
- ✓ Self-Attention directly connects all tokens, so the path length = $O(1)$.
- ✓ This allows Transformers to **capture long-distance relationships** much more effectively than RNNs or CNNs.