Chapter 7

Transformer

김지환

서강대학교 컴퓨터공학과

Table of contents

7.1 Limitation of Recurrent Model

정보 손실

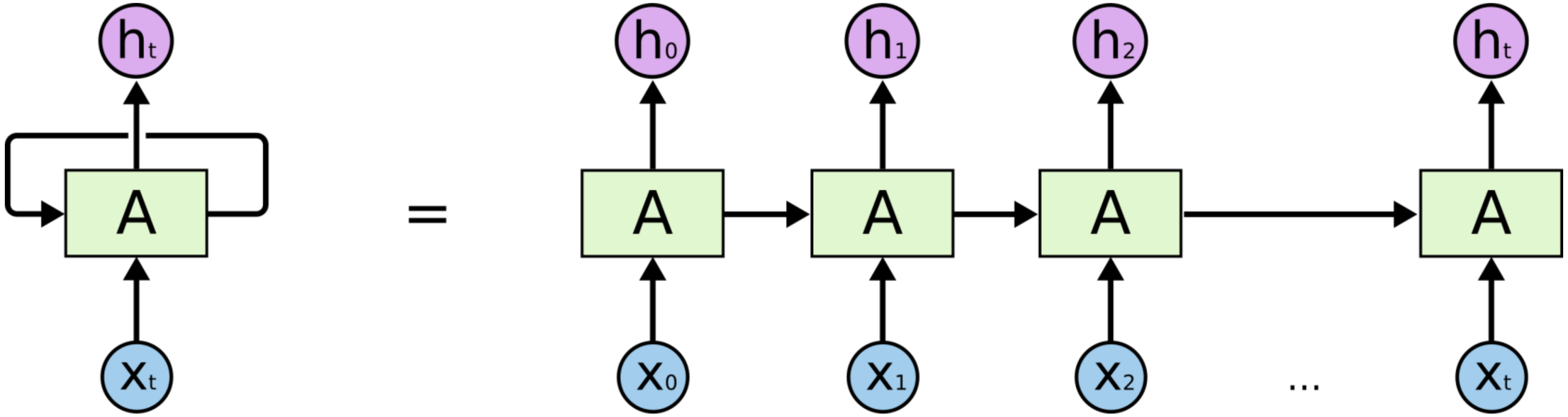
- Recurrent connection은 정보 손실을 유발

학습

- Recurrent connection은 vanishing gradients와 같은 문제로 학습이 어려움

병렬처리

- Recurrent connection은 sequential하게 처리되야 하므로 병렬 처리가 어려워 학습에 시간이 오래 걸림



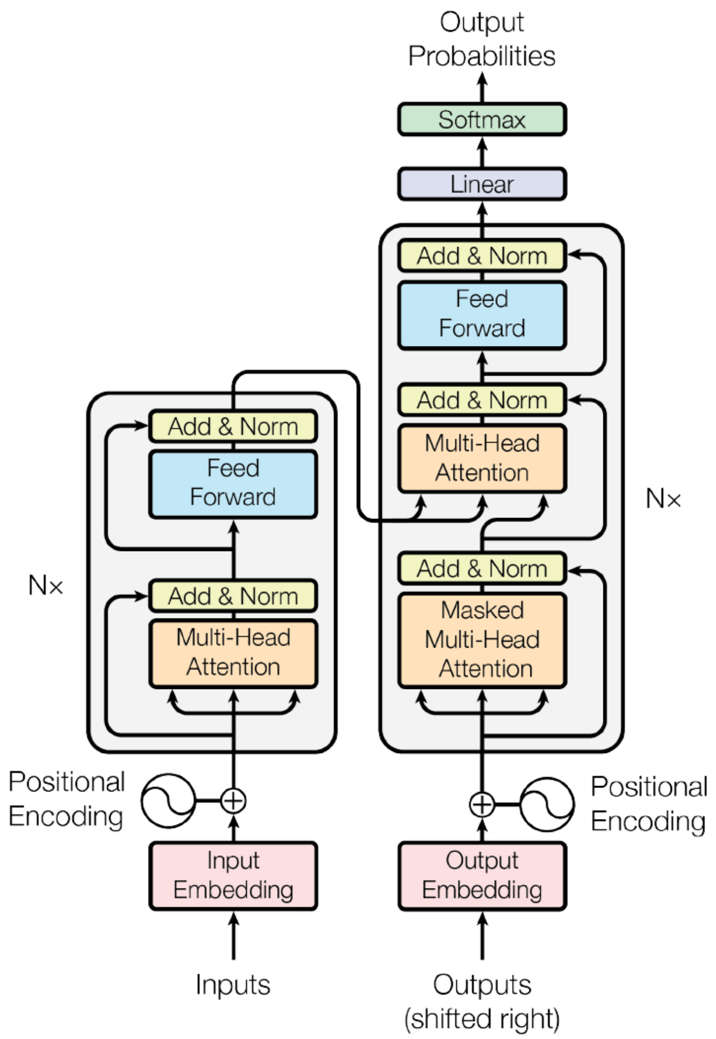
7.2 Transformers : Self-Attention Network

Transformers

- RNN의 한계점으로 인해서 recurrent connection을 제거
- input vectors $(x_1, \ldots, x_n)$ map to output vectors $(y_1, \ldots, y_n)$
- Transformers는 encoder-decoder 구조
  - the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$
  - Given $\boldsymbol{z}$, the decoder then generates an output sequence $(y_1, \ldots, y_n)$ of symbols one element at a time.

Self-attention

- Transformer의 핵심 아이디어
- RNN이 없이도 임의의 큰 컨텍스트에서 정보를 직접 추출하고 사용 가능



7.2.1 Self-Attention

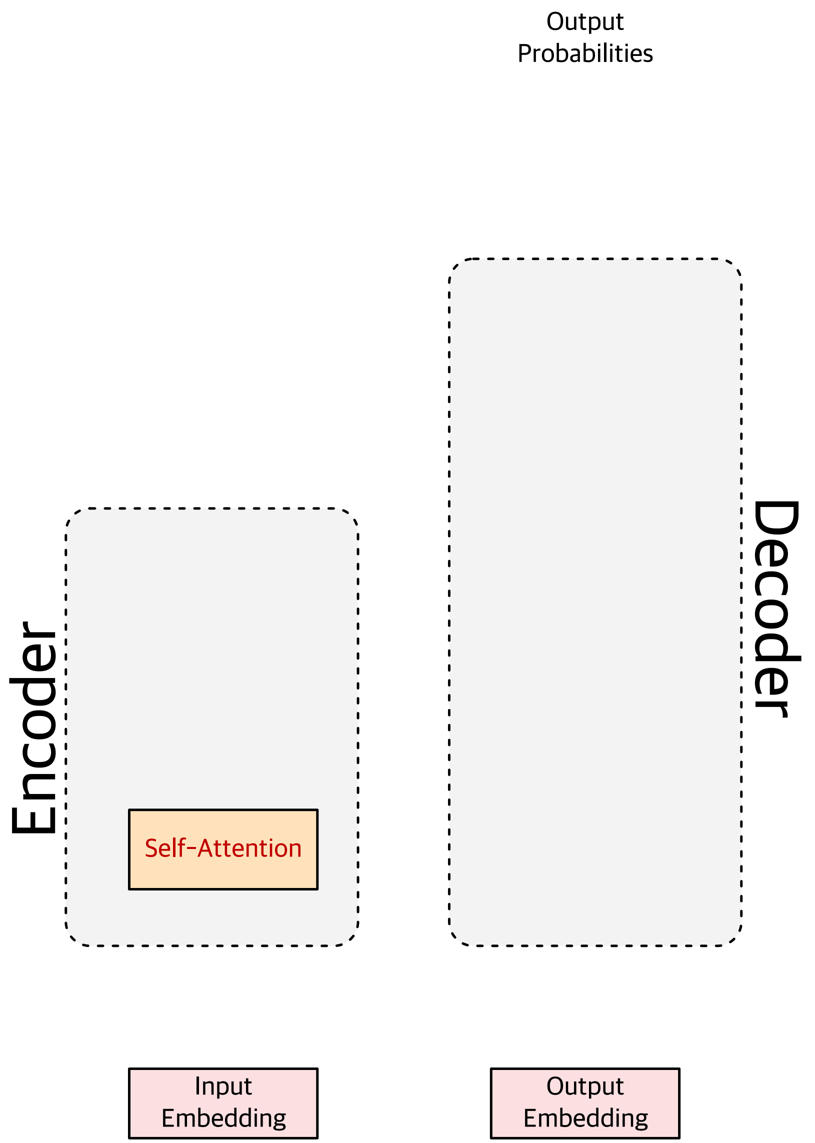
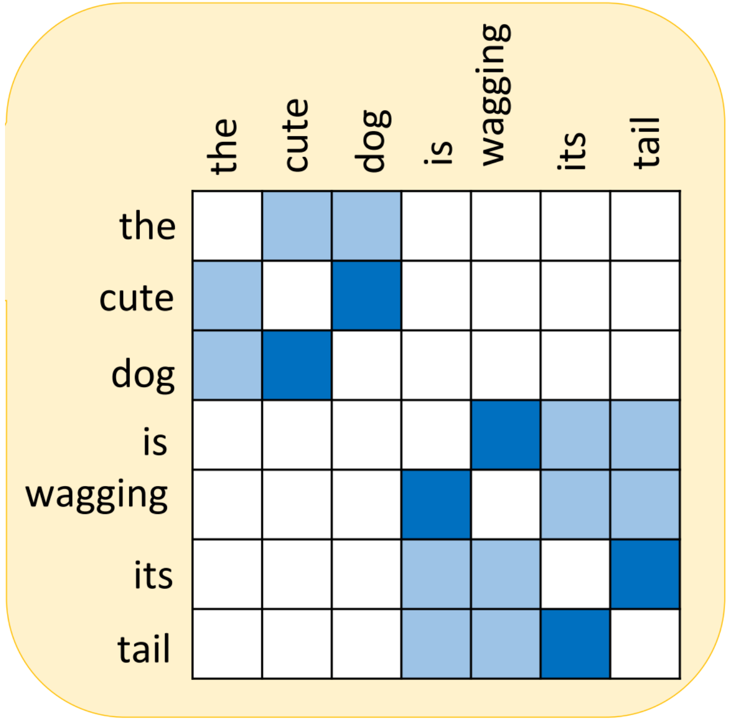


Image from : [Y. Wang *et al.*, EMNLP 2019]

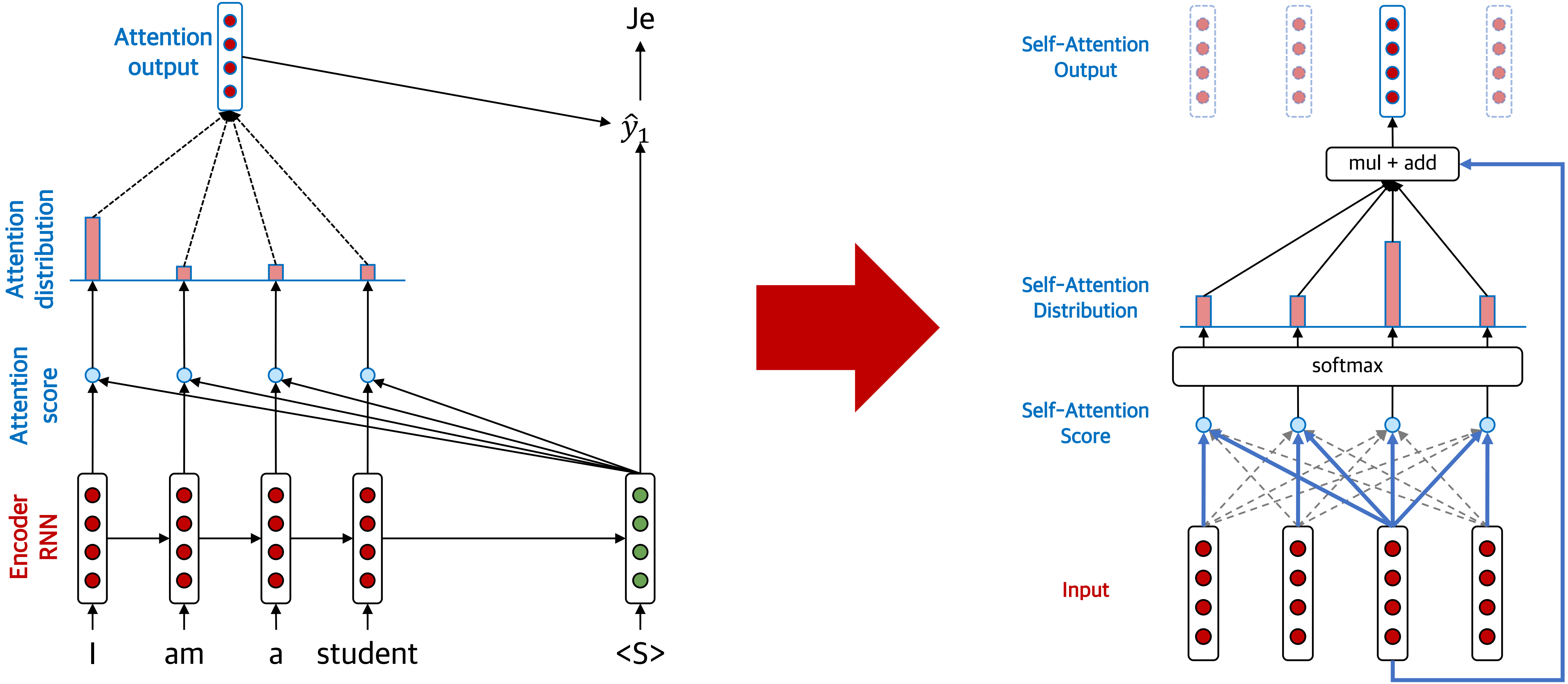
7.2.1 Self-Attention

- Self-Attention
  - Self-Attention layer는 입력 sequences $(x_1, \ldots, x_n)$를 같은 길이의 출력 sequences $(y_1, \ldots, y_n)$로 mapping
  - 입력 sequence의 element들 간의 연관성이 있다면 높은 score를 보이도록 학습
  - 입력의 각 항목을 처리할 때 모델은 고려 중인 입력까지 포함하여 모든 입력에 접근하여 정보를 사용



7.2.1.1 Attention vs Self-Attention

- 일반적인 Attention based model에서의 비교대상이 target sequence인데 반해 self-attention에서는 input간의 관계를 비교



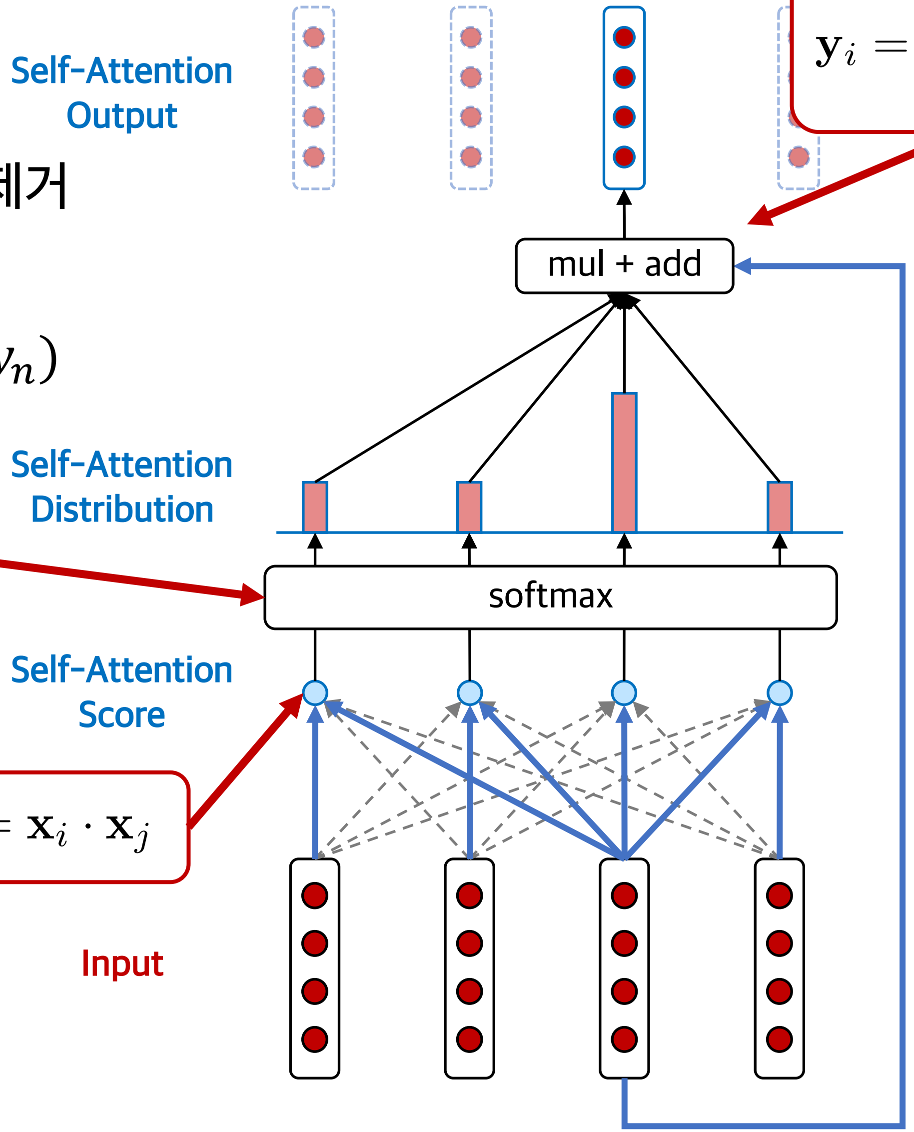
7.2.1.2 Concepts of Self-Attention

- Core idea of self-attention
  - RNN의 한계점을 제거하기 위해 recurrent connection을 제거
  - 입력 sequence의 vector간의 유사도를 계산 (dot product)
  - input vectors $(x_1, \dots, x_n)$ map to output vectors $(y_1, \dots, y_n)$

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{x}_j$$

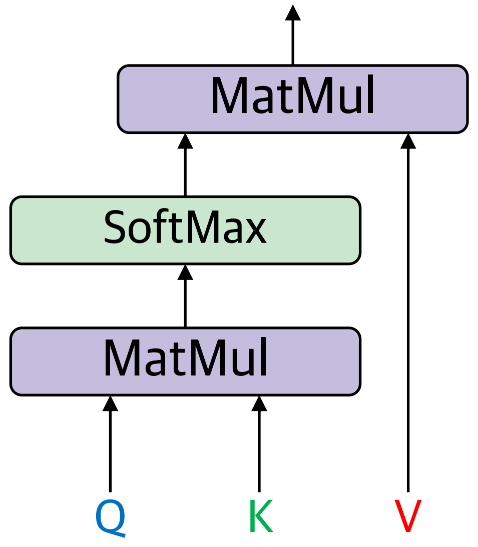
$$\alpha_{ij} = \operatorname{softmax}\left(\operatorname{score}\left(\mathbf{x}_i, \mathbf{x}_j\right)\right) = \frac{\exp\left(\operatorname{score}\left(\mathbf{x}_i, \mathbf{x}_j\right)\right)}{\sum_{k=1}^{n} \exp\left(\operatorname{score}\left(\mathbf{x}_i, \mathbf{x}_k\right)\right)}$$

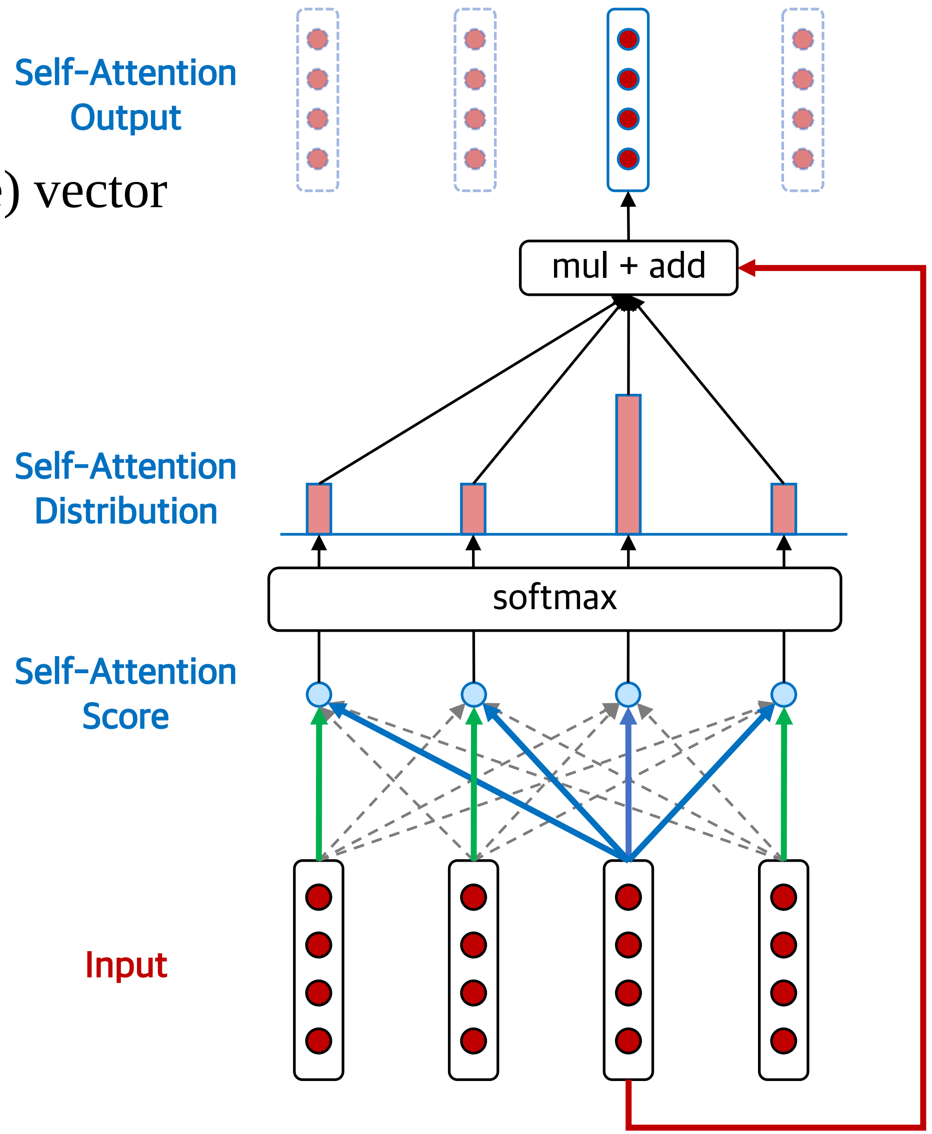
$$\operatorname{score}\left(\mathbf{x}_i, \mathbf{x}_j\right) = \mathbf{x}_i \cdot \mathbf{x}_j$$



7.2.1.3 Look Up Mechanism in Self-Attention

- Query, Key and Value
  - Query, Value : 다른 input들과 비교하려는 input(or hidden state) vector
  - Key : 다른 index의 input(or hidden state) vector

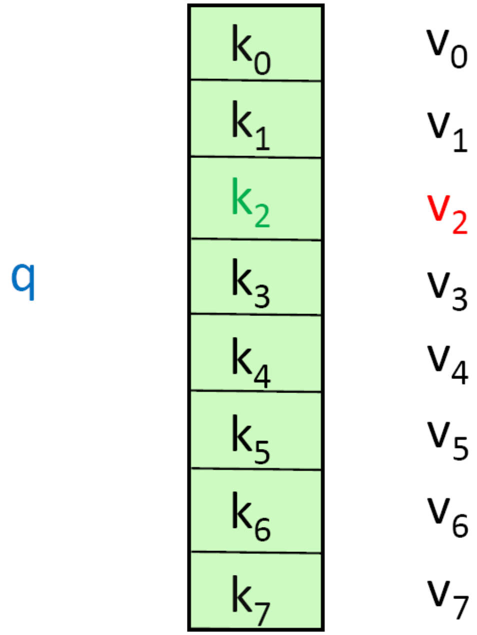




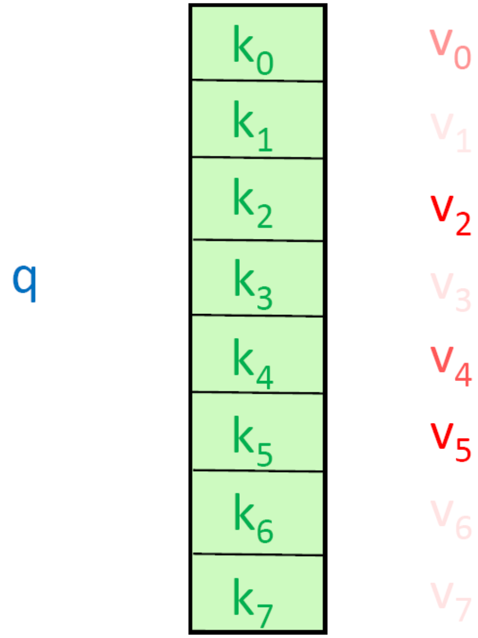
Refer to: https://web.stanford.edu/class/cs224n/slides/cs224n-2022-lecture09-transformers.pdf

7.2.1.4 Intuition For Self-Attention in Transformers

- Transformer에서의 self-attention을 hashtable과 비교
  - Value를 찾기 위해서는 hashtable의 key에 대응하는 query를 비교



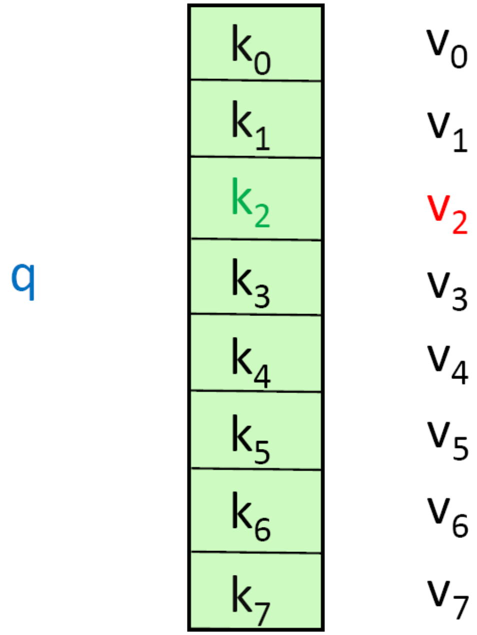
Hashtable :
query(hash) maps to exactly one key-value pair



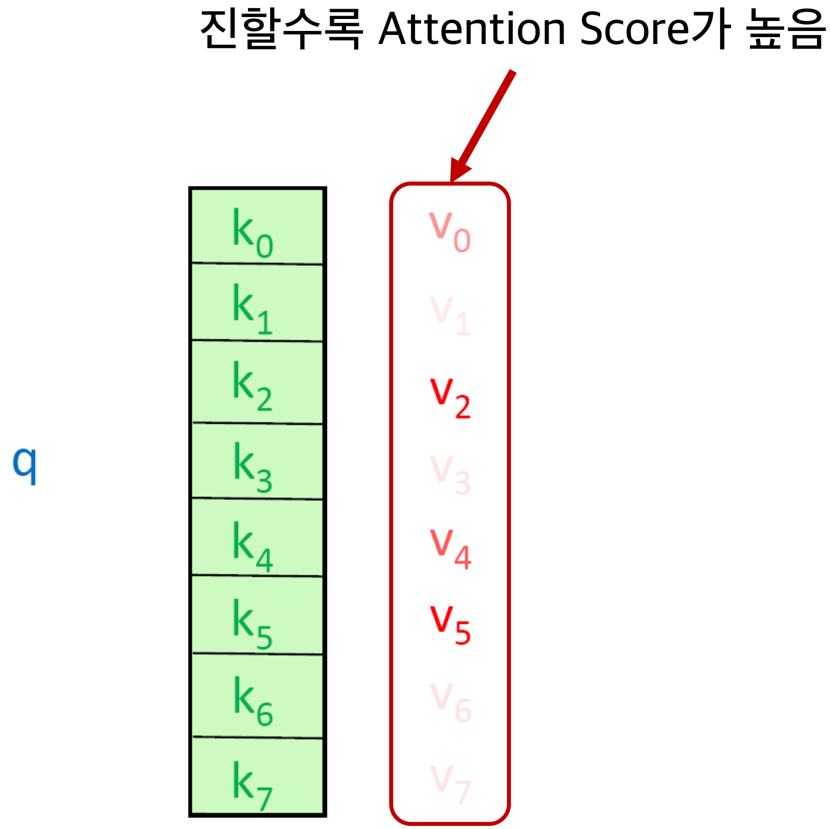
Self-Attention :
Each query matches each key to varying degrees.
We return a sum of values weighted by the query-key match

7.2.1.4 Intuition For Self-Attention in Transformers

- Transformer에서의 self-attention을 hashtable과 비교
  - Value를 찾기 위해서는 hashtable의 key에 대응하는 query를 비교



Hashtable :
query(hash) maps to exactly one **key**-**value** pair



Self-Attention :
Each **query** matches each **key** to varying degrees.
We return a sum of **values** weighted by the **query**-**key** match

Refer to: https://web.stanford.edu/class/cs224n/slides/cs224n-2022-lecture09-transformers.pdf

7.2.1.5 Recipe for Self-Attention in Transformers

- Step 1: 입력 x_i에 대한 query, key, value를 계산 ($x_i \in \mathbb{R}^{1\times d}$)

$$q_i = \mathbf{W}^Q x_i \quad k_i = \mathbf{W}^K x_i \quad v_i = \mathbf{W}^V x_i$$

$$(\mathbf{W}^Q \in \mathbb{R}^{d\times d}, \mathbf{W}^K \in \mathbb{R}^{d\times d}, \text{and } \mathbf{W}^V \in \mathbb{R}^{d\times d})$$

- Step 2 : query와 key의 attention score를 계산

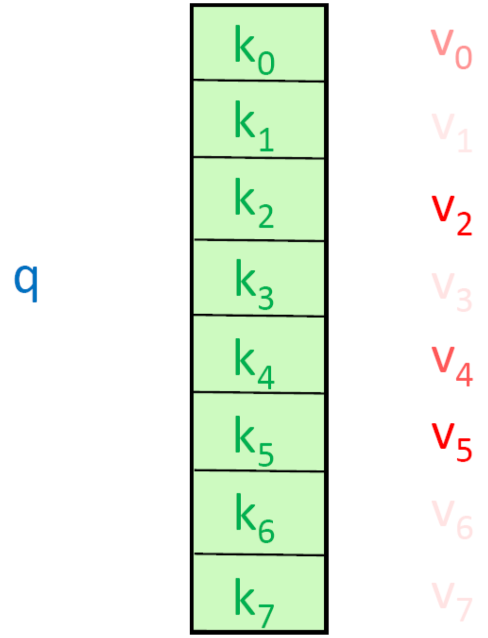
$$e_{ij} = q_i \cdot k_j$$

- Step 3: attention score에 softmax를 적용

$$\alpha_{ij} = \operatorname{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$

- Step 4: value의 weighted Sum을 계산

$$\text{Output}_i = \sum_j \alpha_{ij} v_j$$



7.2.1.5 Recipe for Vectorized Self-Attention in Transformers

- Step 1: 입력 x_i를 matrix로 X로 나타내고 이에 대한 query, key, value를 계산

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$

- Step 2 : query와 key의 attention score를 계산

$$E = QK^T$$

- Step 3: attention score에 softmax를 적용

$$A = \mathrm{softmax}(E)$$

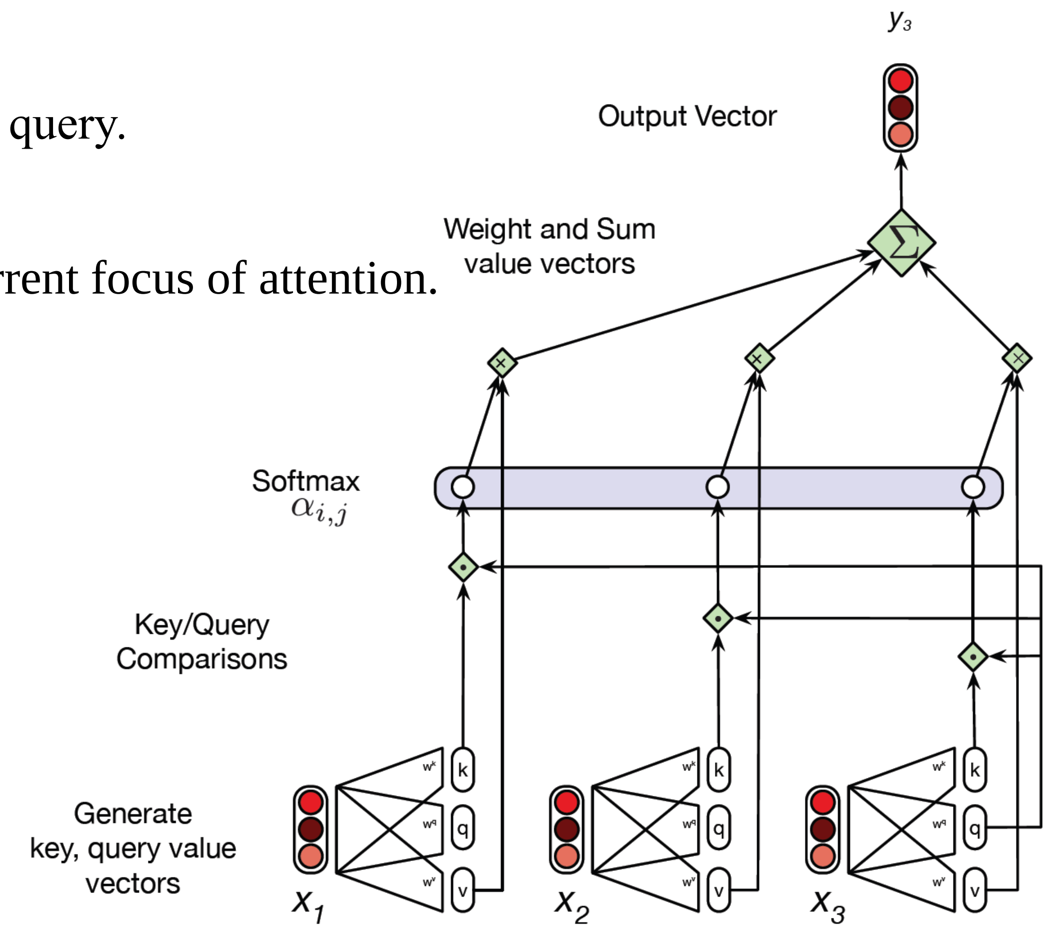
- Step 4: value의 weighted Sum을 계산

$$\mathrm{Output} \ = AV$$

$$\boxed{\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(QK^T\right) V}$$

7.2.1.6 Self-Attention in Transformers

- Attention process동안 각 input embedding이 수행하는 역할
  - Query (Q)
    - As the current focus of attention when being compared to all of the other preceding inputs. We'll refer to this role as a query.
  - Key (K)
    - In its role as all of the other input being compared to the current focus of attention. We'll refer to this role as a key.
  - Value (V)
    - And finally, as a value used to compute the output for the current focus of attention.

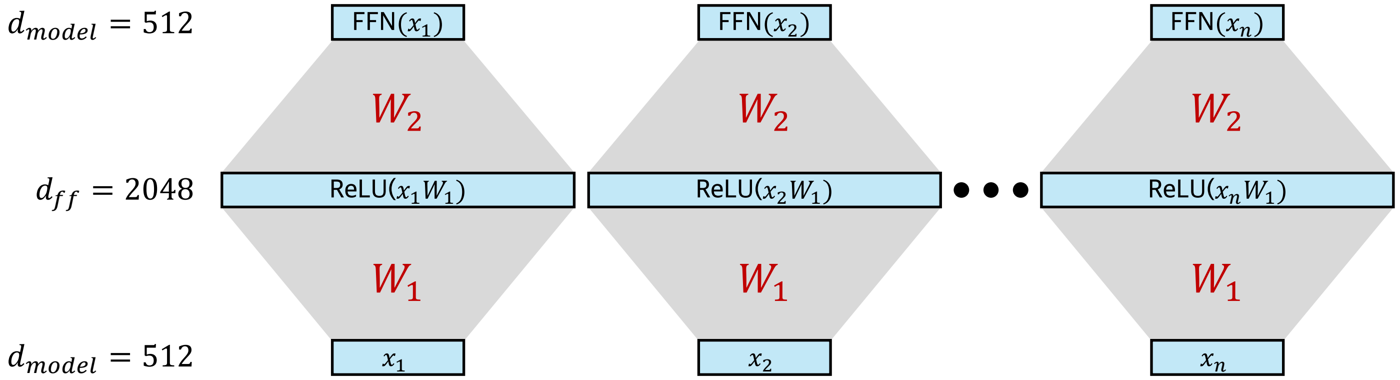


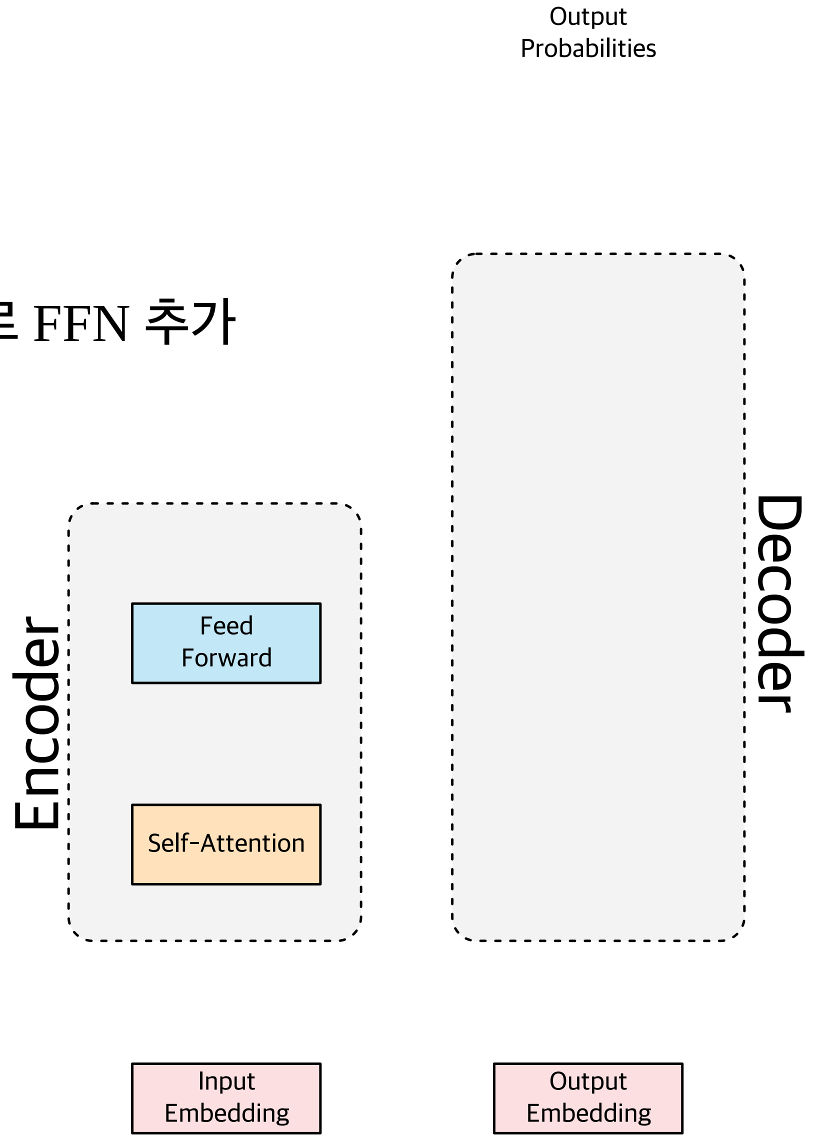
7.2.2 Encoder : Feed Forward

- Position-wise Feed-Forward Network (FFN)

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)\, W_2 + b_2$$

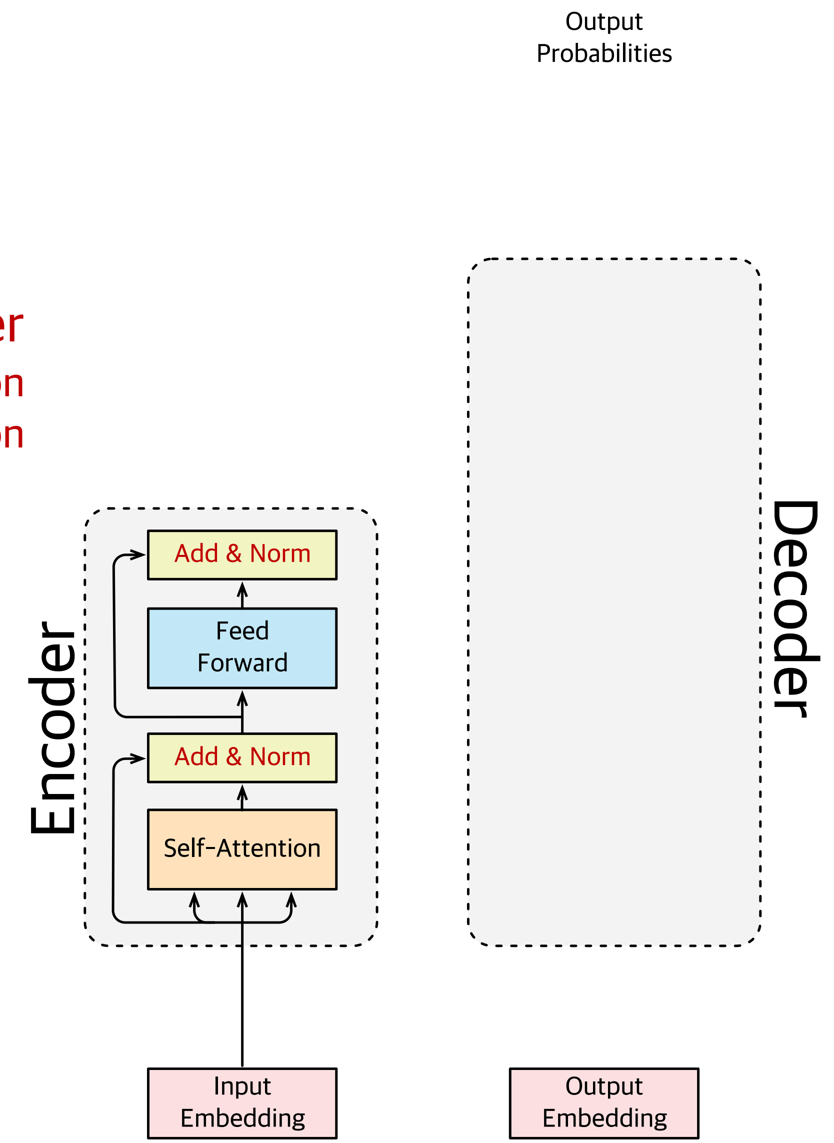
  - Self-Attention Layer의 출력에는 non-linearity가 적용되지 않았으므로 FFN 추가
  - 입력에 대해서 같은 weight를 갖는 FFN을 position별로 적용
  - 논문에서는 $d_{model} = 512, d_{ff} = 2048$





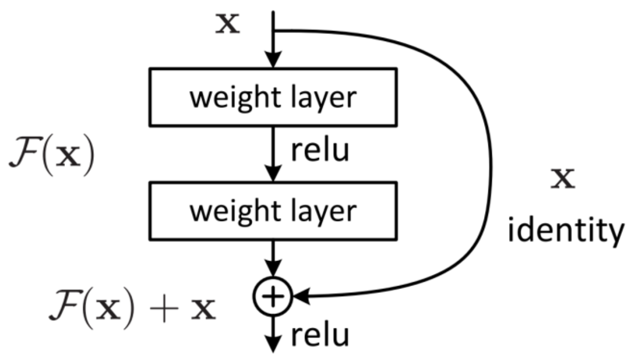
7.2.2.1 Encoder : Residual Connection & Layer Norm.

Training Tricks for Deeper Encoder
Layer normalization
Residual connection

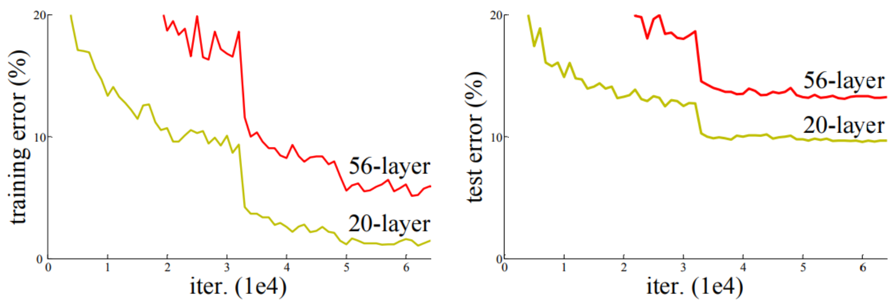


7.2.2.2 Encoder : Residual Connection

- Residual Connection
  - [He et al., 2016]에서 computer vision model을 위해 고안된 방법
  - 이전까지는 네트워크를 깊게 쌓을 수록 학습이 어려웠음
    - Overfitting이 발생하지 않음에도 deeper model의 성능이 낮음
    - Vanishing gradients, etc.
  - Residual connection을 이용하여 network가 input의 정보를 손실하지 않을 수 있음



Residual learning: a building block.



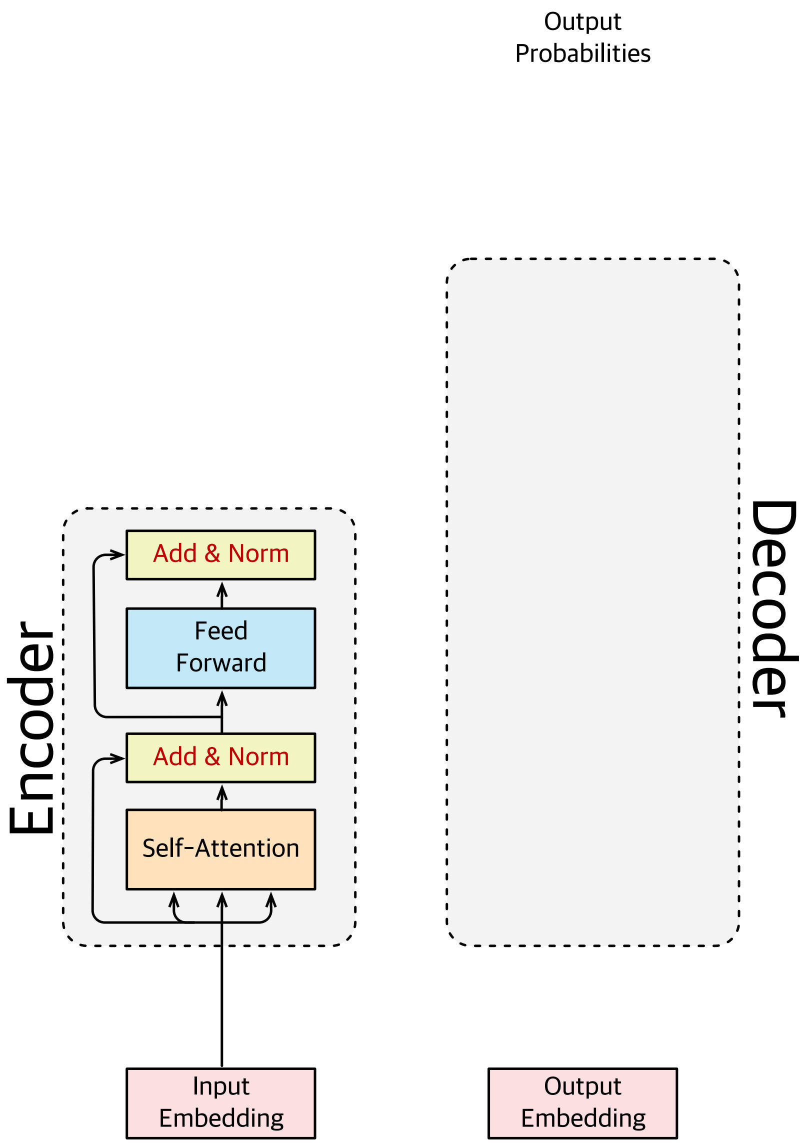
Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks.
The deeper network has higher training error, and thus test error.

7.2.2.3 Encoder : Layer Normalization

- Layer Normalization
  - [Ba et al., 2016]에서 안정적인 학습을 위해 고안된 방법
  - Layer의 output units가 N(1,0) 분포를 갖도록 normalize

$$\mu = \frac{1}{d_h}\sum_{i=1}^{d_h} x_i \quad \sigma = \sqrt{\frac{1}{d_h}\sum_{i=1}^{d_h}(x_i - \mu)^2}$$

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma}$$



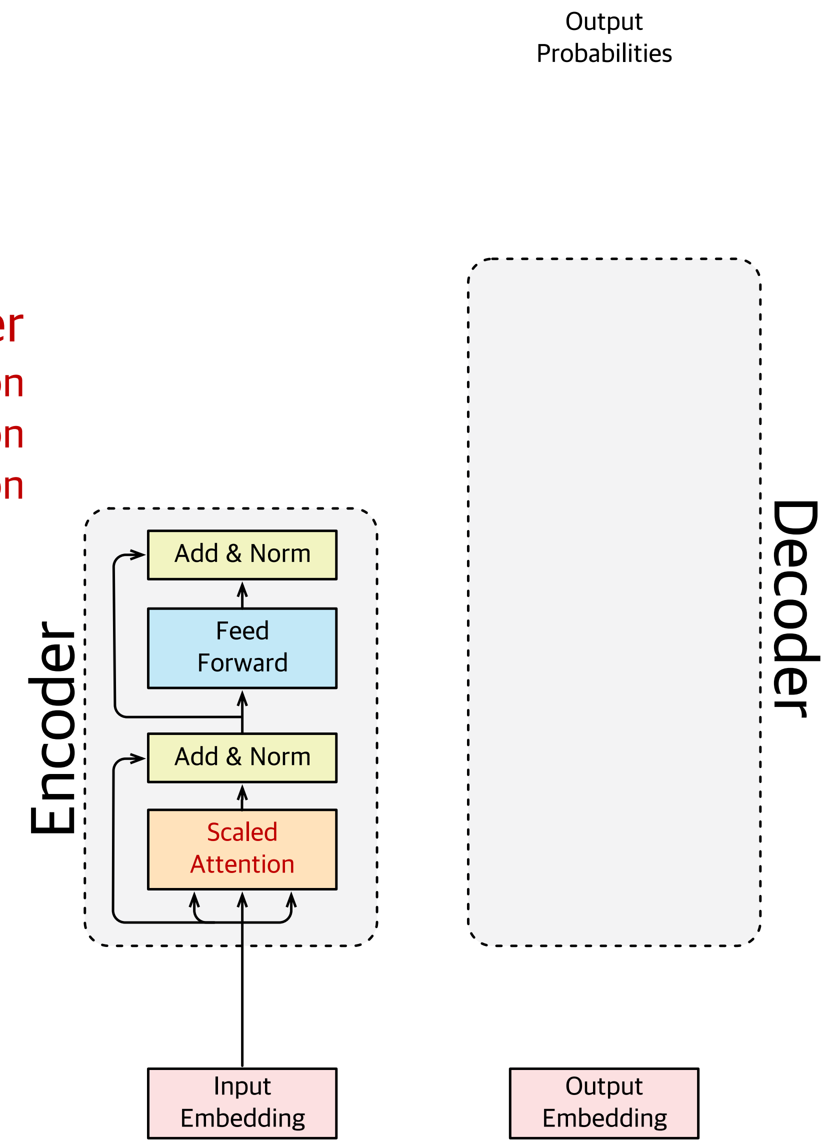
7.2.2.4 Encoder : Scaled Dot-Product Attention

Training Tricks for Deeper Encoder

Layer normalization

Residual connection

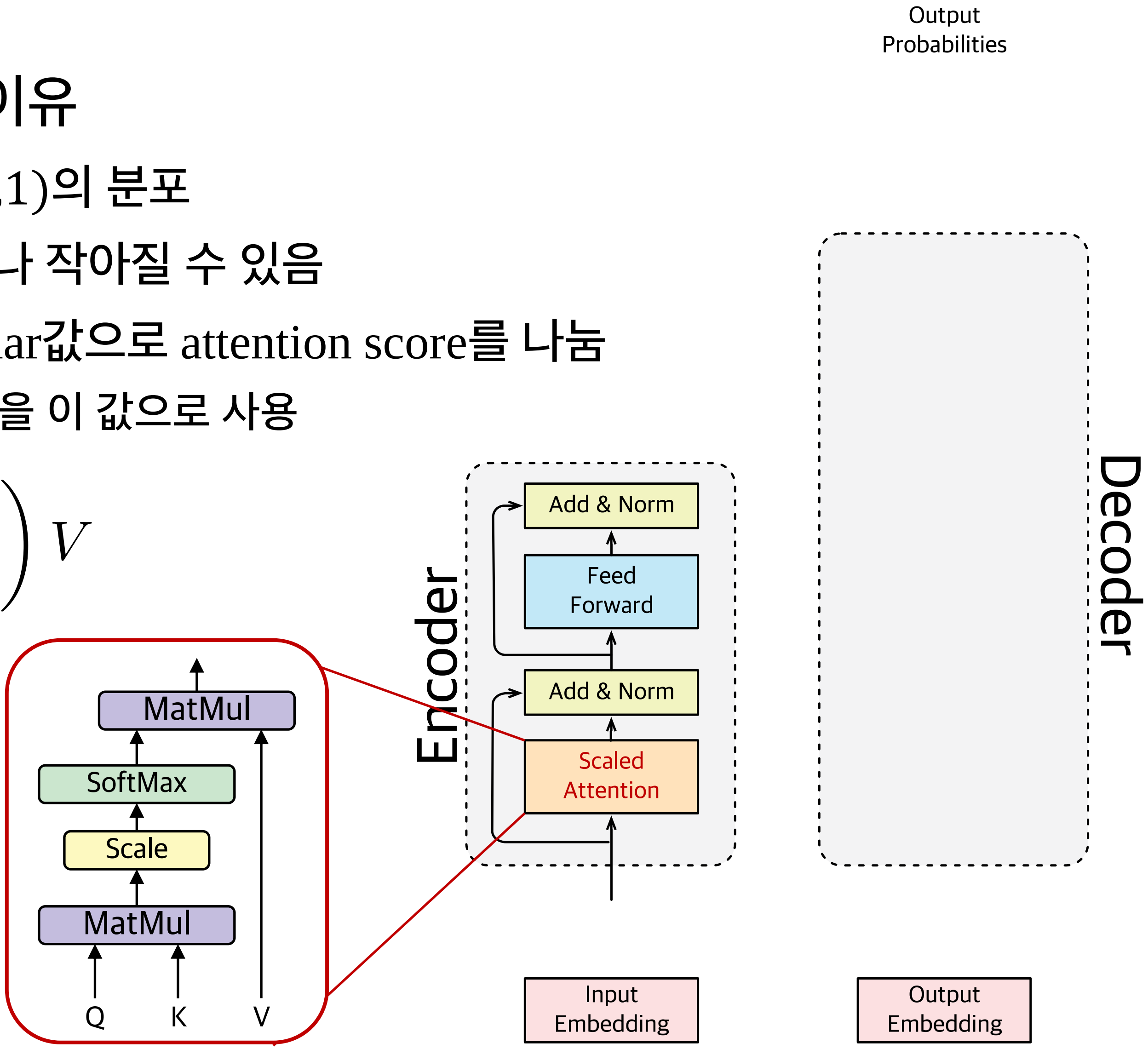
Scaled dot-product attention



7.2.2.4 Scaled Dot-Product Attention

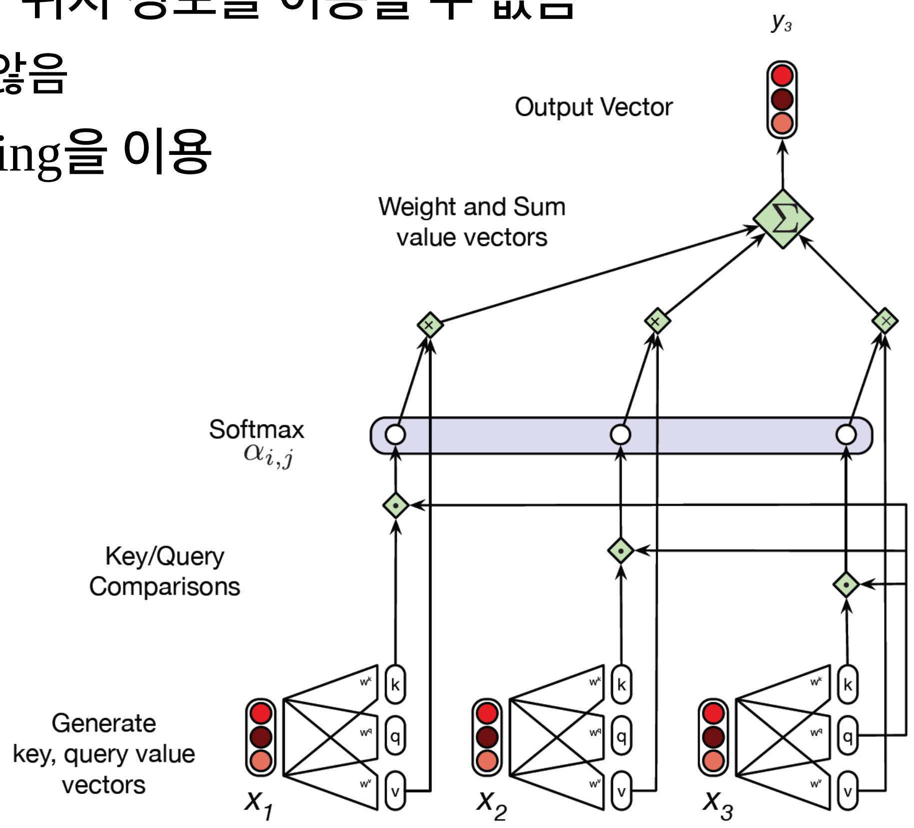
- Scaled Dot-Product Attention을 사용하는 이유
  - Layer normalization을 수행하고 후 vector는 $N(0,1)$의 분포
  - Dot product의 값은 input의 값에 따라 매우 커지거나 작아질 수 있음
  - 이를 방지하기위해 softmax를 취하기전 일정한 scalar값으로 attention score를 나눔
    - 논문에서는 queries와 keys의 dimension d_k의 제곱근을 이 값으로 사용

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



7.2.3 Positional Encoding : Modeling Input order

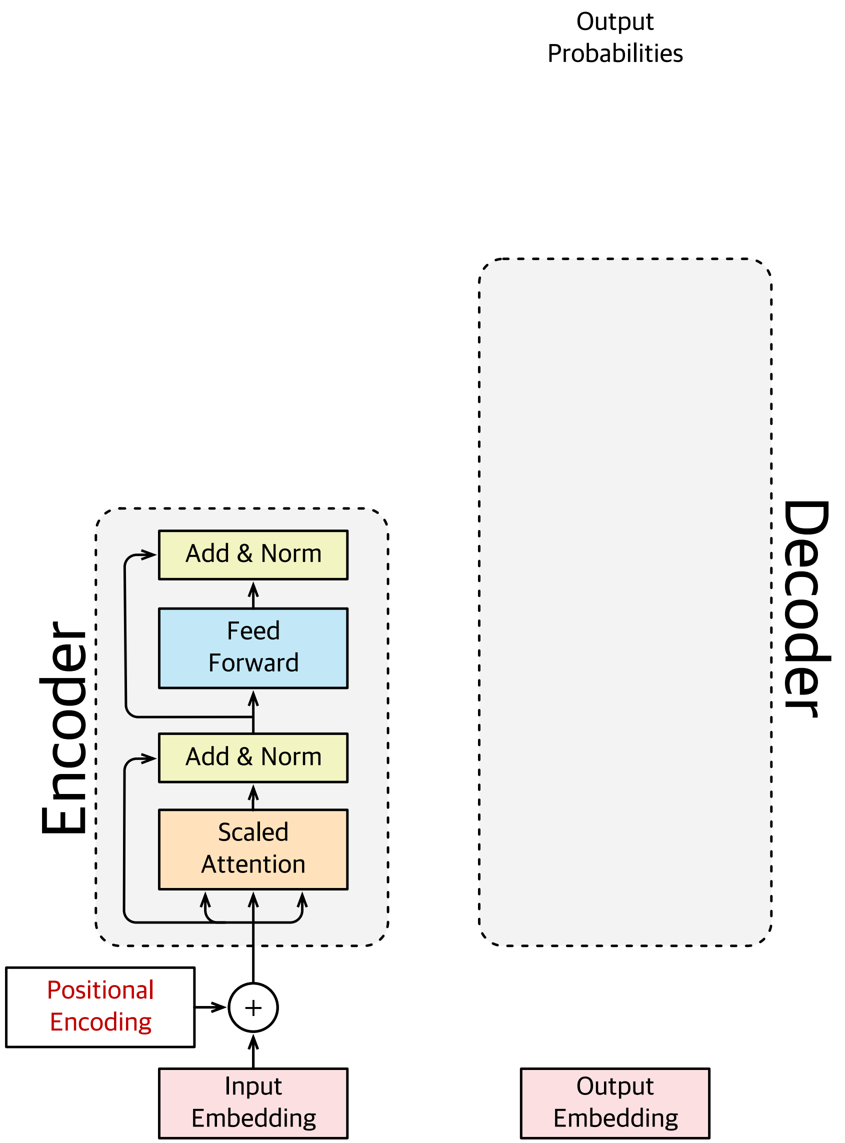
- How does a transformer model the position of each token in the input sequence?
  - RNNs에서는 모델자체적으로 input의 순서를 고려할 수 있음
  - Recurrent connection이 제거된 Self-attention에서는 모델자체가 위치 정보를 이용할 수 없음
    - 오른쪽 그림에서 입력의 순서를 바꾸더라도 attention 결과가 바뀌지 않음
  - Transformer에서는 위치정보를 제공할 수 있는 positional encoding을 이용



man eats small dinosaur
small man eats dinosaur
dinosaur eats small man
small dinosaur eats man
…

Example from : https://web.stanford.edu/class/cs224n/slides/cs224n-2022-lecture09-transformers.pdf

7.2.3 Positional Encoding : Encoder



7.2.3 Positional Encoding

- Input vector에 위치 정보를 줄 수 있는 position vector p_i를 추가

$$p_i \in \mathbb{R}^d, \text{ for } i \in \{1, 2, \ldots, T\} \text{ are position vectors}$$

- p_i를 생성하는 방법에는 여러가지가 존재
- Input vector에 positional encoding을 추가 할 수 있는 방법도 여러가지 고려 가능
  - Concatenation
  - Add
- [A. Vaswani et al., 2017]에서는 아래와 같은 방법을 사용
  - Let $\tilde{q}_i$,$\tilde{k}_i$,$\tilde{v}_i$ be our old queries, keys and values

$$v_i = \tilde{v}_i + p_i$$

$$q_i = \tilde{q}_i + p_i$$

$$k_i = \widetilde{k}_i + p_i$$

7.2.3.1 Positional Encoding : Try 1

- Positional Encoding Using Index

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}

$\boldsymbol{p_i}$	1	2	3	4	5	6	7	8	9	10

- 단순히 input vector의 index를 position encoding으로 사용

7.2.3.1 Positional Encoding : Try 1

- Positional Encoding Using Index

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}

⬇

$\boldsymbol{p_i}$	1	2	3	4	5	6	7	8	9	10

  - 단순히 input vector의 index를 position encoding으로 사용

- Problem
  - 입력의 길이에 따라서 p_i의 값이 너무 커지는 문제가 발생
    - 예를 들어 입력의 index가 512라면 [1,512] 범위의 값이 생성
  - Neural net의 학습을 위해서는 입력의 범위가 작아 져야함

7.2.3.2 Positional Encoding : Try 2

- Positional Encoding Using Index and Normalization

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}

⬇

p_i	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1

- Input vector의 index를 sequence의 길이로 나누어 [0,1] 범위의 값을 사용

7.2.3.2 Positional Encoding : Try 2

- Positional Encoding Using Index and Normalization

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
$\boldsymbol{p_i}$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1

  - Input vector의 index를 sequence의 길이로 나누어 [0,1] 범위의 값을 사용

input	x_1	x_2	x_3	x_4	x_5
$\boldsymbol{p_i}$	0.2	0.4	0.6	0.8	0.1

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
$\boldsymbol{p_i}$	0.125	0.250	0.375	0.5	0.625	0.750	0.875	1

7.2.3.2 Positional Encoding : Try 2

- Positional Encoding Using Index and Normalization

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
p_i	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1

  - Input vector의 index를 sequence의 길이로 나누어 [0,1] 범위의 값을 사용

- Problem

input	x_1	x_2	x_3	x_4	x_5
p_i	0.2	0.4	0.6	0.8	0.1

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
p_i	0.125	0.250	0.375	0.5	0.625	0.750	0.875	1

7.2.3.3 Positional Encoding : Try 3

- Positional Encoding Using Binary Representation

input	x_1	x_2	x_3	x_4	x_5	x_6	x_7

$\boldsymbol{p_{i,0}}$	1	0	1	0	1	0	1
$\boldsymbol{p_{i,1}}$	0	1	1	0	0	1	1
$\boldsymbol{p_{i,2}}$	0	0	0	1	1	1	1

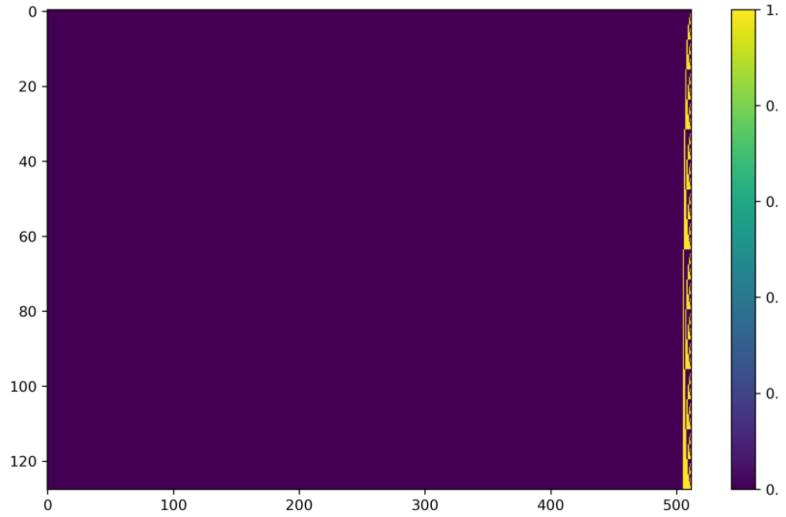
- Input vector의 index를 이진수로 나타내어 bit별로 표기

7.2.3.3 Positional Encoding : Try 3

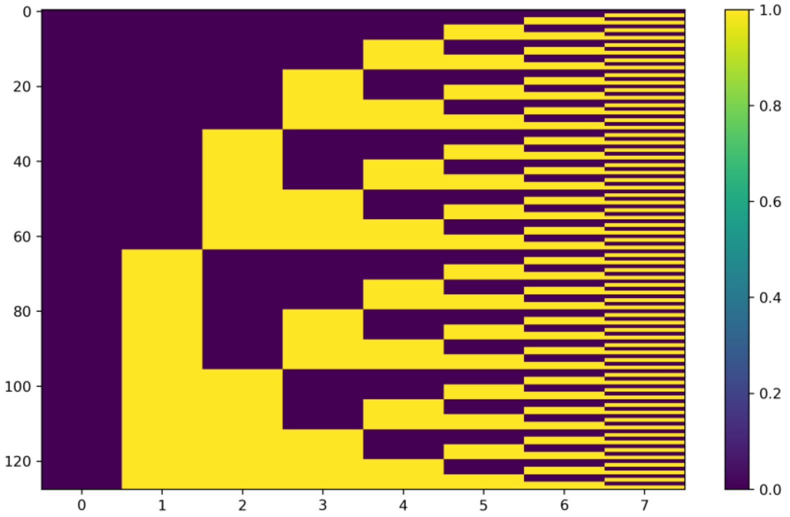
- Positional Encoding Using Binary Representation
  - Input vector의 index를 이진수로 나타내어 bit별로 표기
- Problem
  - 낭비되는 bit가 많음
  - dot-product attention에서 적합하지 않음

7.2.3.3 Positional Encoding : Try 3

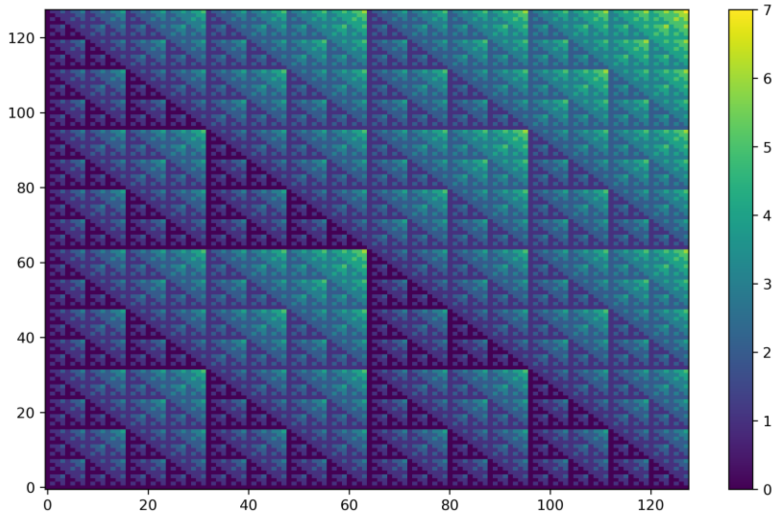
- Positional Encoding Using Binary Representation
- Problem
  - 낭비되는 bit가 많음
  - dot-product attention에서 적합하지 않음



512-dim binary representation
positional encoding (pos=[0,127])



8-dim binary representation
positional encoding (pos=[0,127])



Dot-product similarity of
512-dim positional encoding vector using
binary representation positional encoding (pos=[0,127])

7.2.3.4 Sinusoidal Positional Encoding

- Positional Encoding
  - The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed.

$$PE_{(pos,2i)} = \sin\left(pos/10000^{2i/d_{\text{model}}}\right)$$
$$PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d_{\text{model}}}\right)$$

$$PE_{pos} = \begin{pmatrix} \sin\left(pos/10000^{\frac{2*0}{d_{model}}}\right) \\ \cos\left(pos/10000^{\frac{2*0}{d_{model}}}\right) \\ \vdots \\ \sin\left(pos/10000^{\frac{2*\frac{d_{model}}{2}-1}{d_{model}}}\right) \\ \cos\left(pos/10000^{\frac{2*\frac{d_{model}}{2}-1}{d_{model}}}\right) \end{pmatrix}$$

7.2.3.4 Sinusoidal Positional Encoding

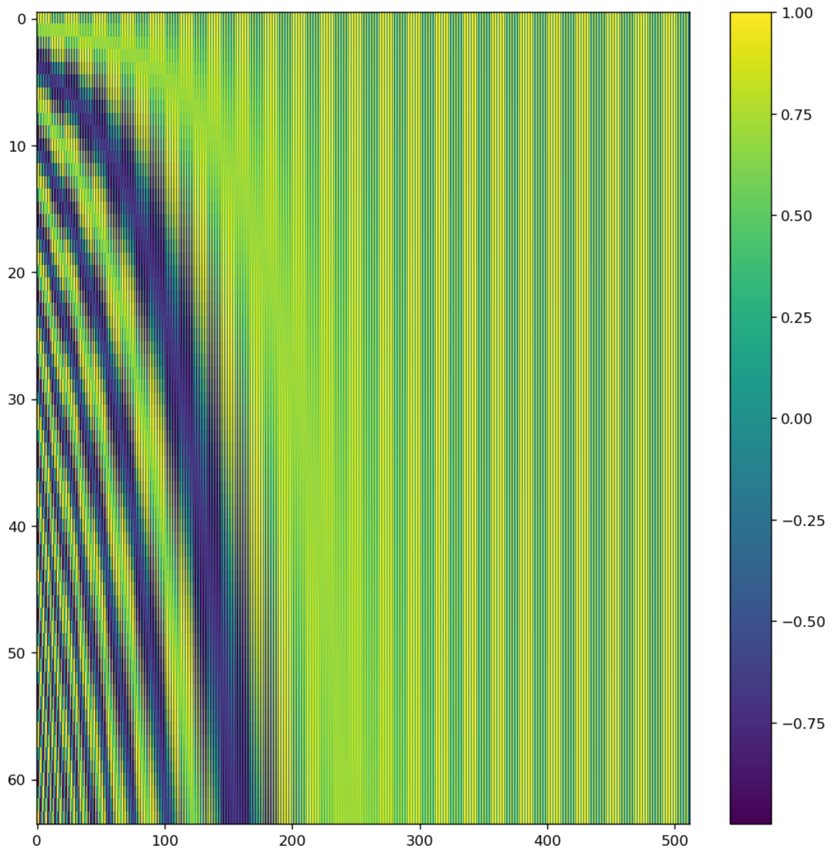
- Dot-Product of Positional Encoding
  - Δt만큼의 차이가 나는 두 positional encoding vector PE_i와 $PE_{i+\Delta t}$에 대한 dot-product로 similarity를 계산

$$
\mathrm{PE}_i^\top \mathrm{PE}_{i+\Delta t} = \begin{pmatrix} \sin\left(\frac{i}{10000^{2*0/512}}\right) \\ \cos\left(\frac{i}{10000^{2*0/512}}\right) \\ \vdots \\ \sin\left(\frac{i}{10000^{2*256-1/512}}\right) \\ \cos\left(\frac{i}{10000^{2*256-1/512}}\right) \end{pmatrix}^\top \begin{pmatrix} \sin\left(\frac{i+\Delta t}{10000^{2*0/512}}\right) \\ \cos\left(\frac{i+\Delta t}{10000^{2*0/512}}\right) \\ \vdots \\ \sin\left(\frac{i+\Delta t}{10000^{2*256-1/512}}\right) \\ \cos\left(\frac{i+\Delta t}{10000^{2*256-1/512}}\right) \end{pmatrix}
$$

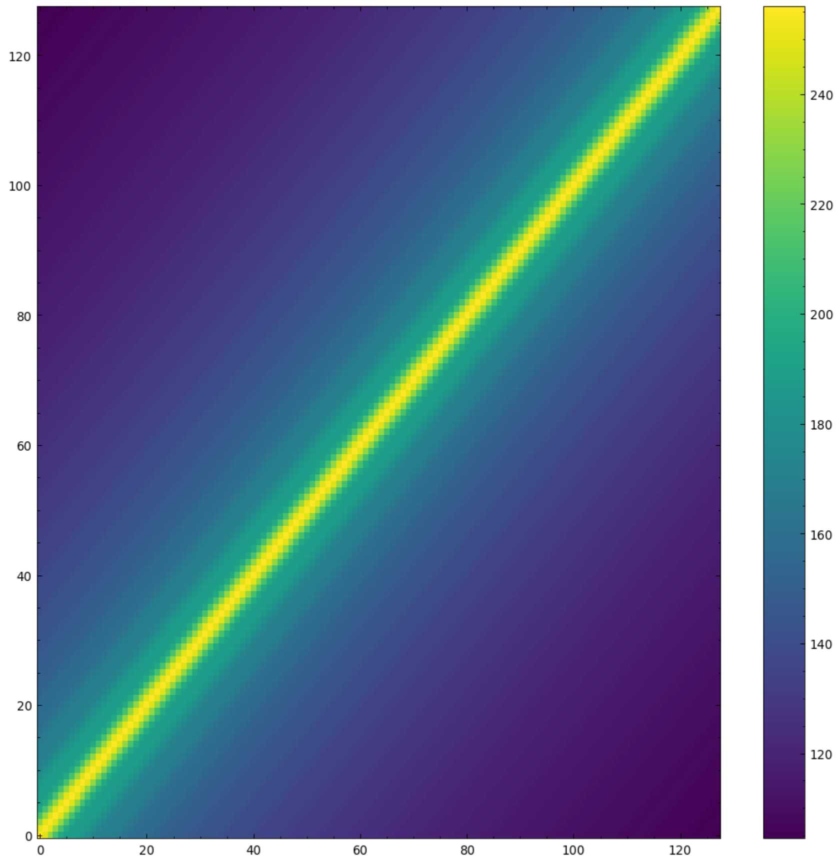
$$
\begin{aligned}
&= \sin(\omega_k t)\sin(\omega_k(t+\Delta t)) + \cos(\omega_k t)\cos(\omega_k(t+\Delta t)) + \cdots \\
&= \sin(\omega_k t)\sin(\omega_k t + \omega_k \Delta t)) + \cos(\omega_k t)\cos(\omega_k t + \omega_k \Delta t)) + \cdots \\
&= \sin(\omega_k t)(\sin(\omega_k t)\cos(\omega_k \Delta t) + \cos(\omega_k t)\sin(\omega_k \Delta t)) \\
&\quad + \cos(\omega_k t)(\cos(\omega_k t)\cos(\omega_k \Delta t) - \sin(\omega_k t)\sin(\omega_k \Delta t)) + \cdots \\
&= \sum_{k=0}^{255} \cos(\omega_k \Delta t)
\end{aligned}
$$

7.2.3.4 Sinusoidal Positional Encoding

■ Dot-Product of Positional Encoding

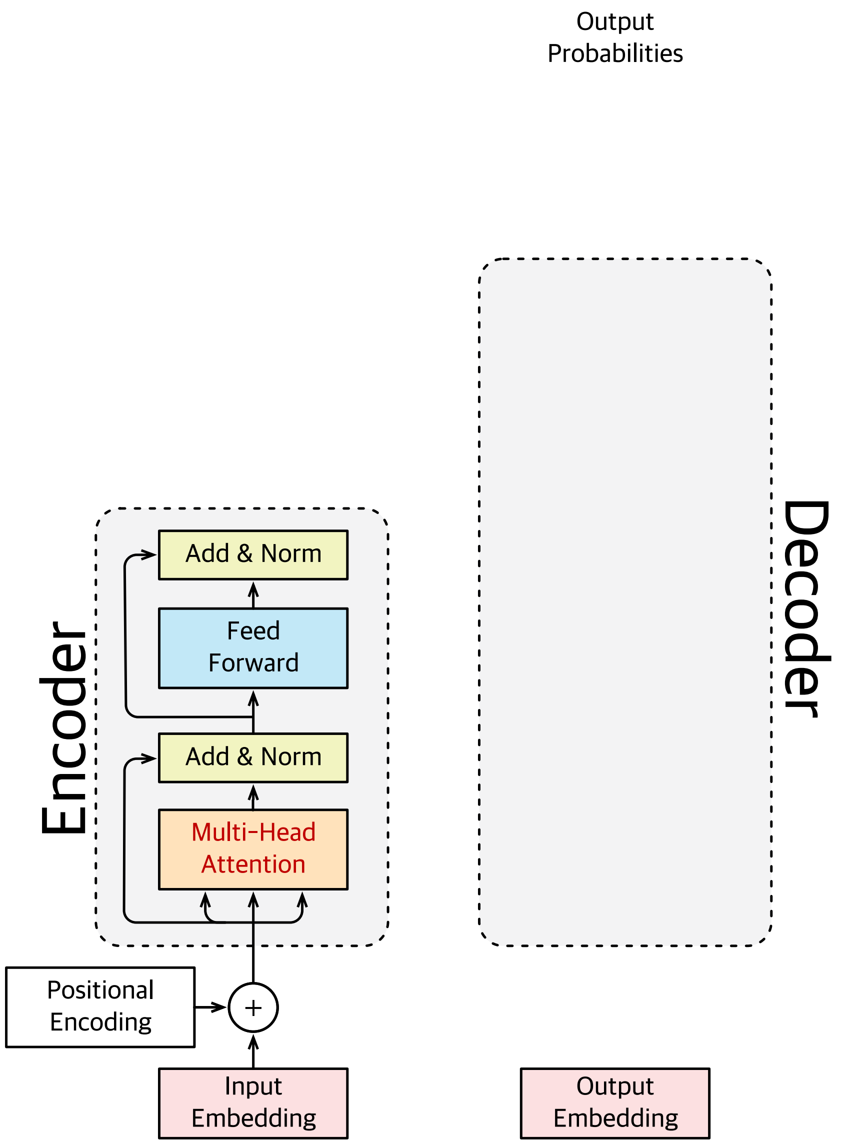


512-dim
sinusoidal positional encoding (pos=[0,127])

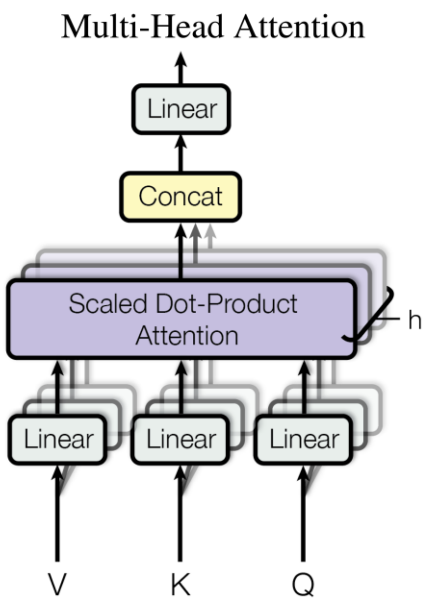


Dot-product similarity
of 512-dim positional encoding vector
using sinusoidal positional encoding (pos=[0,127])

7.2.4 Encoder : Multi-Head Attention



7.2.4 Multi-Head Attention



- input sequence의 input vector는 서로 여러 가지 방식 관련 가능
  - 문장을 예로 들면 구문론적, 의미론적, 담화적 관계
  - 음성을 예로 들면 주파수 대역별로의 관계
- Transformer는 Muti-head attention layer를 통해 이를 해결
  - heads는 Self-attention layer의 집합으로, 하나의 layer 안에서 다른 parameter를 가짐
  - 각각의 head는 입력 간에 존재하는 다른 관계의 다른 측면을 학습 가능

$$\text{Let, } W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d\times\frac{d}{h}} \text{ and } W^O \in \mathbb{R}^{h\frac{d}{h}\times d}$$

$$\text{MultiHeadAttn}\ (\mathbf{X}) = (\boldsymbol{head}_1 \oplus \boldsymbol{head}_2 \ldots \oplus \boldsymbol{head}_h)\,\mathbf{W}^O$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_i^Q; \mathbf{K} = \mathbf{X}\mathbf{W}_i^K; \mathbf{V} = \mathbf{X}\mathbf{W}_i^V$$

$$\text{head}_i = \text{SelfAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V})$$

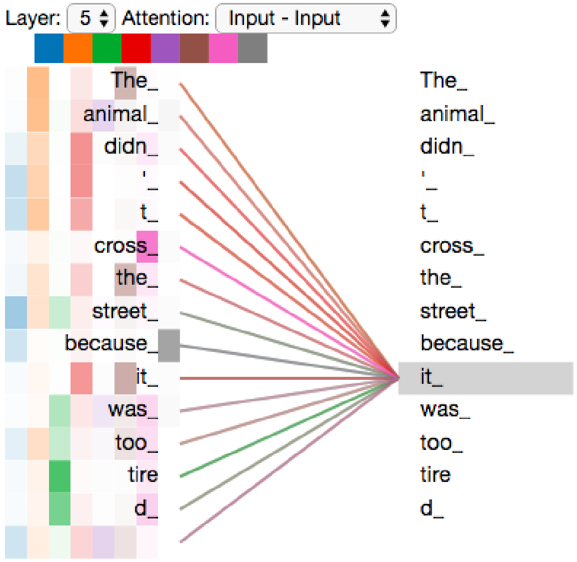


Image from https://jalammar.github.io/illustrated-transformer/

7.2.4.1 Multi-Head Attention- NLP example

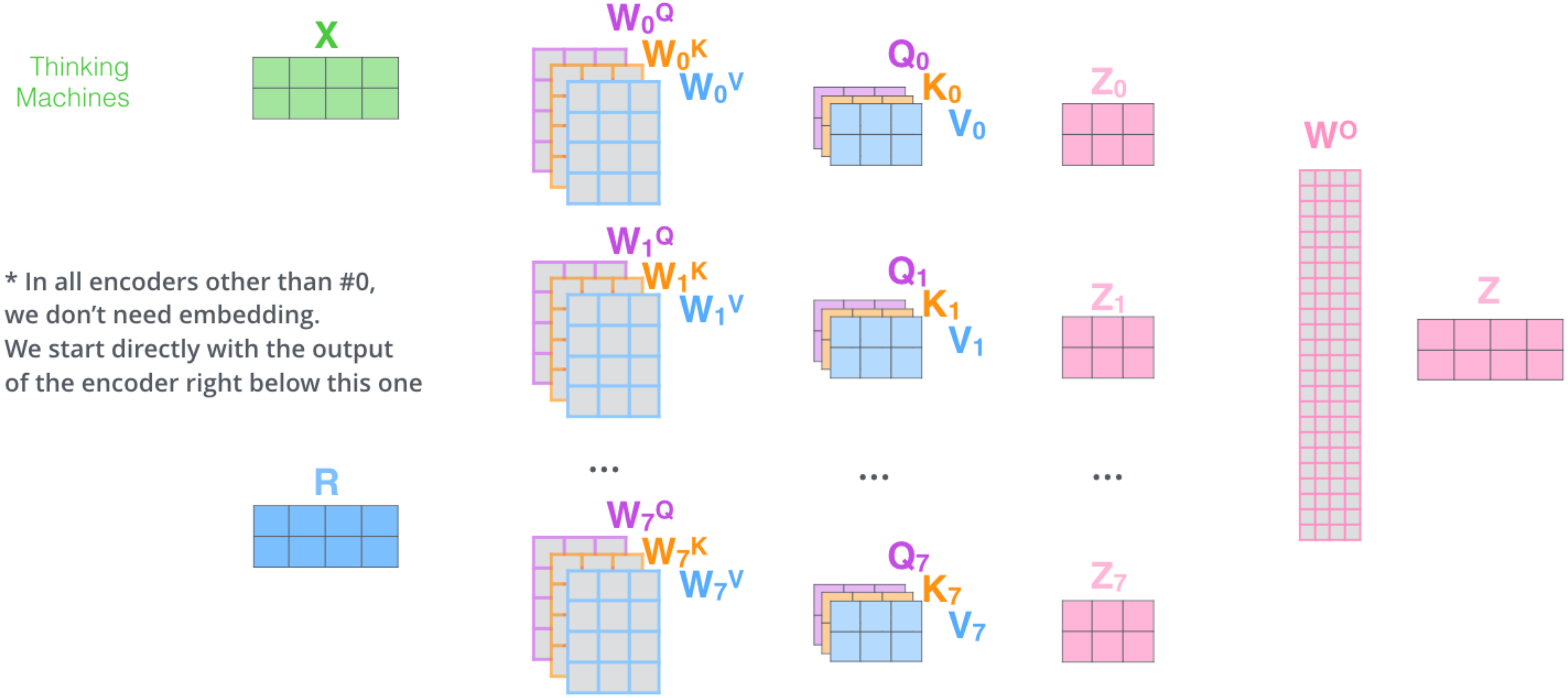
1) This is our input sentence*

2) We embed each word*

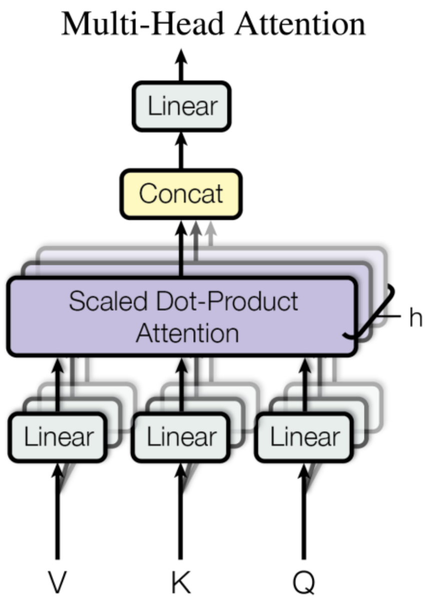
3) Split into 8 heads. We multiply X or R with weight matrices

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

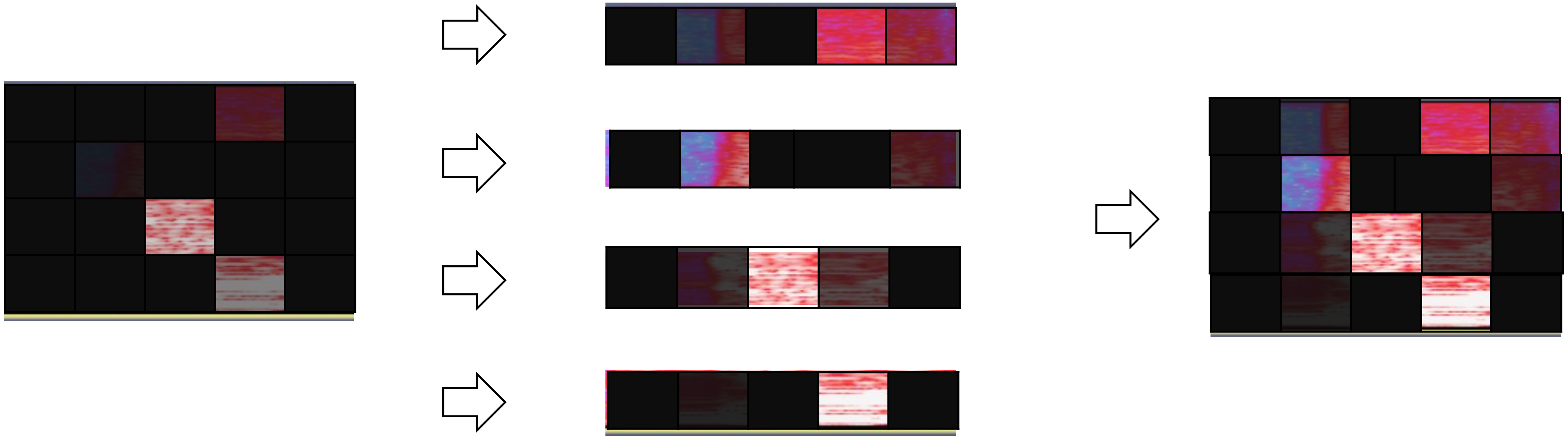


* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



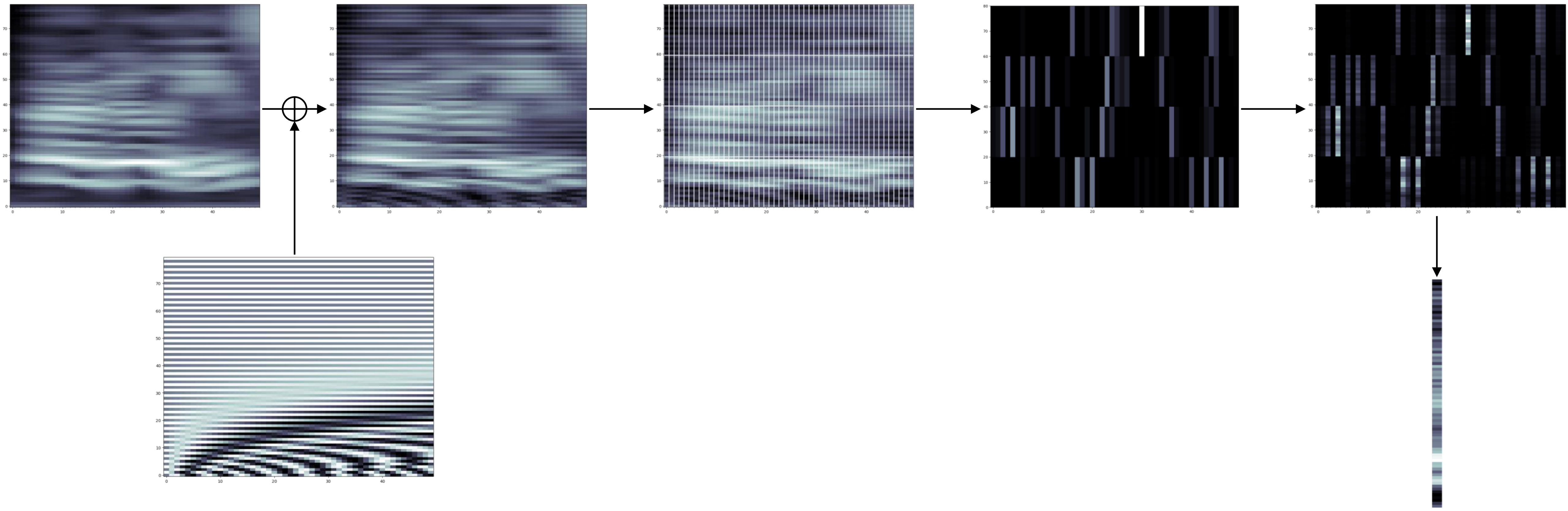
7.2.4.2 Multi-Head Attention - ASR Example

- 전체 입력 벡터의 matrix에서 가중치를 구하는 것이 아닌, 균등하게 나누어진 sub-matrix에 대해 각각의 가중치를 구하는 방법
- 음성의 경우, frequency 영역을 균등하게 나눈 뒤 각 frequency 대역에서 각 시간대에 가중치를 부여함

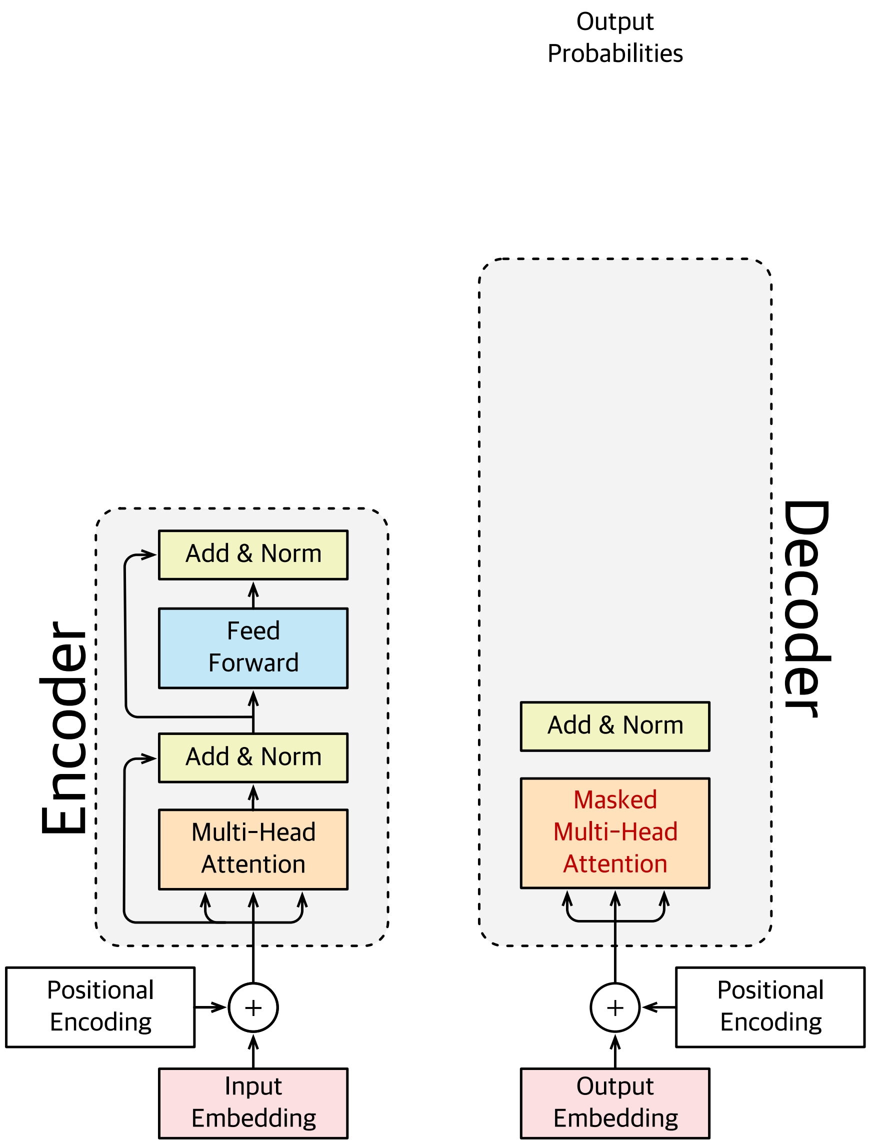


7.2.4.2 Multi-Head Attention - ASR Example

- Input : 80-dim filter banks, 4 attention heads

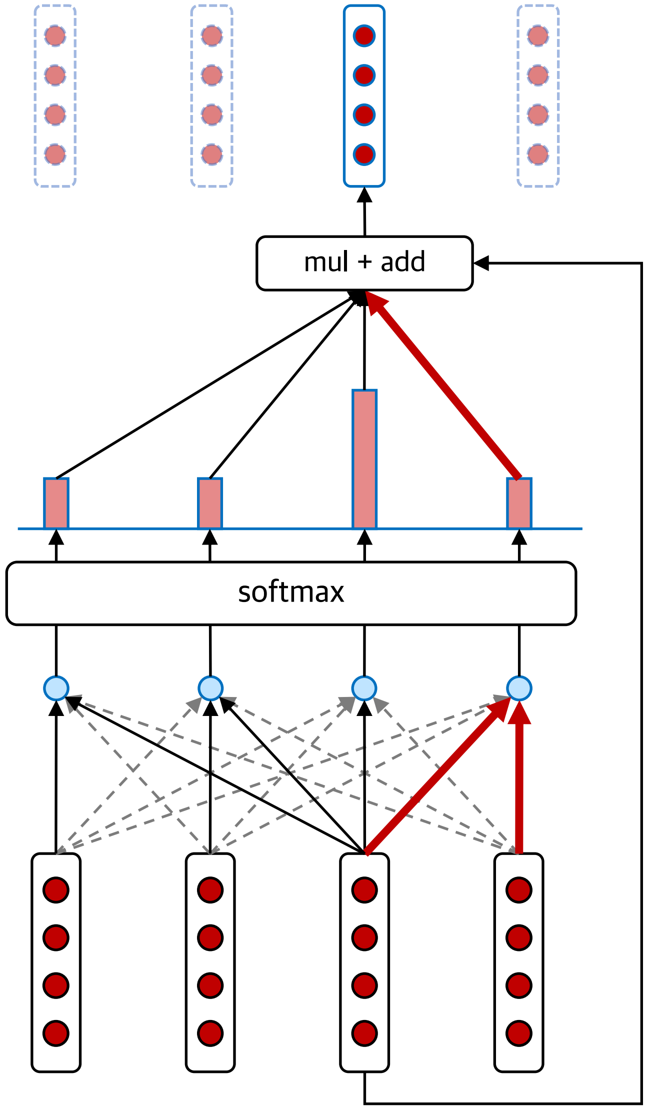


7.2.4.3 Decoder : Masked Multi-Head Attention



7.2.4.4 Masked Multi-head Attention

- Problem of Self-Attention in Decoder
  - Language modeling, ASR Task에서 Inference하는 과정의 미래의 출력은 알 수 없음
  - 일반적인 self-attention 계산방식으로는 self-attention score를 계산 하는 과정에서 미래의 출력들까지 고려하여 attention score를 계산
  - n번째 입력에 대한 출력은 n-1까지의 입력만을 고려



7.2.4.4 Masked Multi-head Attention

Problem of Self-Attention in Decoder

- Language modeling, ASR Task에서 Inference하는 과정의 미래의 출력은 알 수 없음
- 일반적인 self-attention 계산방식으로는 self-attention score를 계산 하는 과정에서 미래의 출력들까지 고려하여 attention score를 계산
- n번째 입력에 대한 출력은 n-1까지의 입력만을 고려해야함

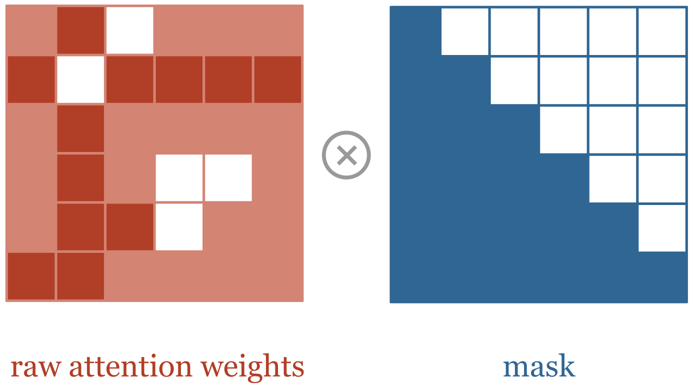
Masked Multi-head Attention

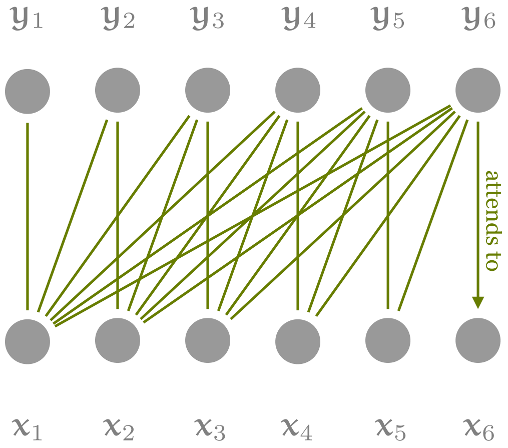
- Attention score를 계산하는 과정에서 Masking을 적용하여 n번째 입력에 대해서는 n-1까지의 입력만을 고려

N × N

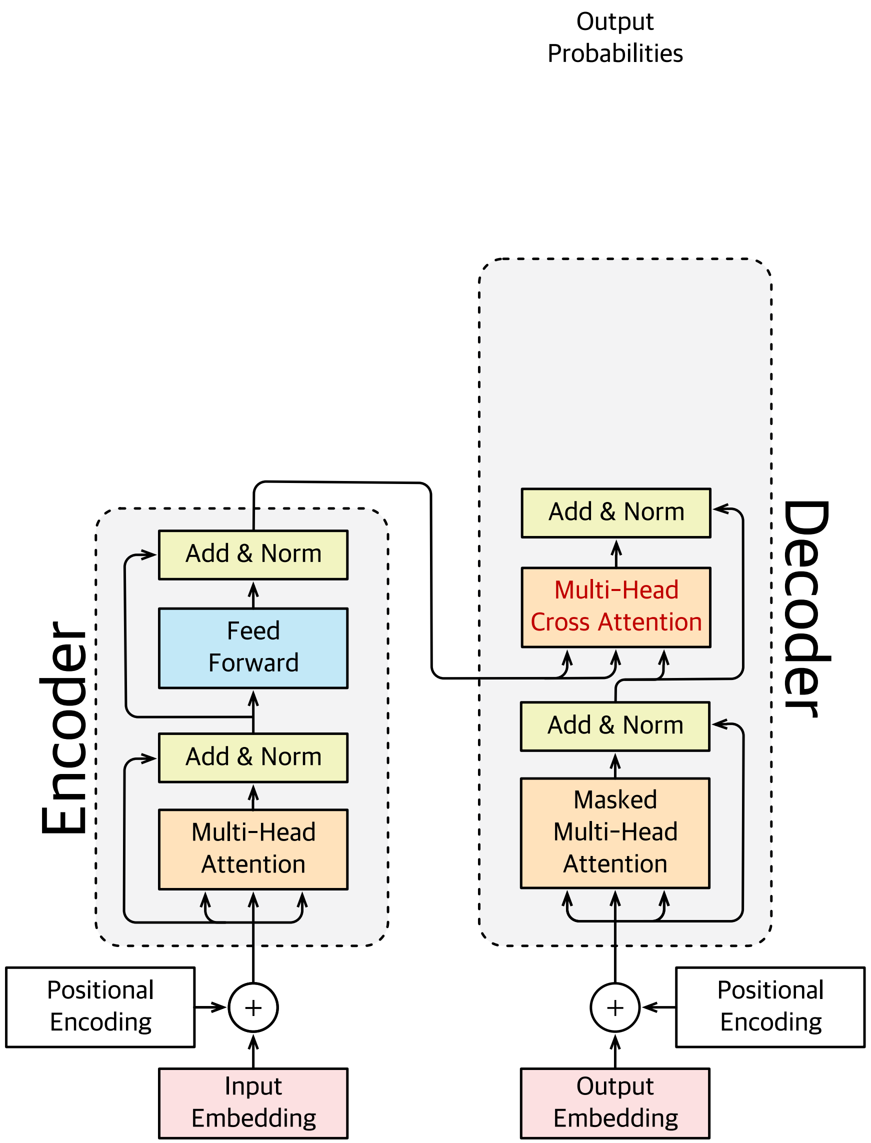
q1•k1	−∞	−∞	−∞	−∞
q2•k1	q2•k2	−∞	−∞	−∞
q3•k1	q3•k2	q3•k3	−∞	−∞
q4•k1	q4•k2	q4•k3	q4•k4	−∞
q5•k1	q5•k2	q5•k3	q5•k4	q5•k5

Figure 9.17 The $N \times N$ $\mathbf{QT}^{\intercal}$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).



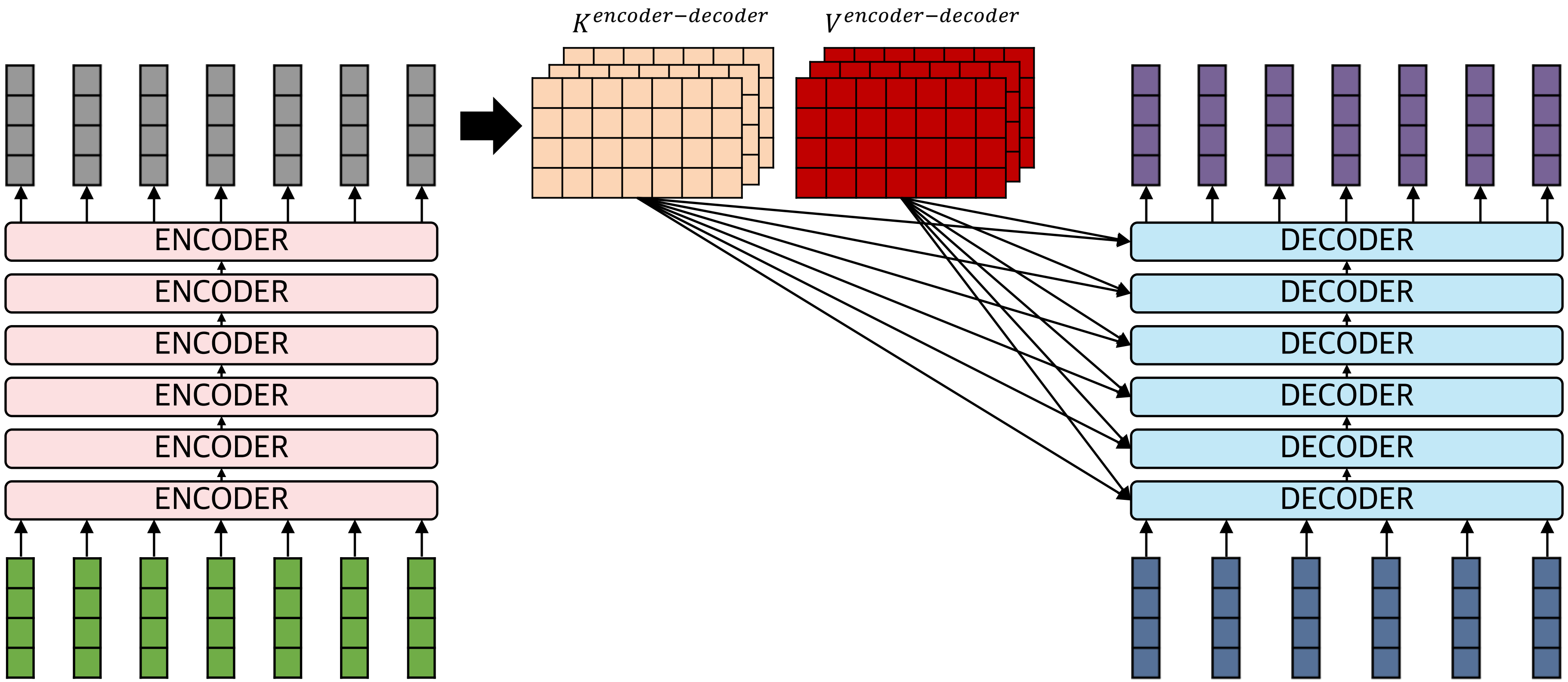


7.2.4.5 Decoder : Multi-Head Cross Attention



7.2.4.5 Decoder : Multi-Head Cross Attention

- Encoder-Decoder Attention
  - 지금까지의 self-attention은 같은 입력으로 부터의 query, key, value만을 고려
  - Encoder-Decoder 구조에서의 self-attention은 다른 입력에서의 query, key, value를 고려



7.2.5 Transformers 전체 구조