

Transformer Architecture

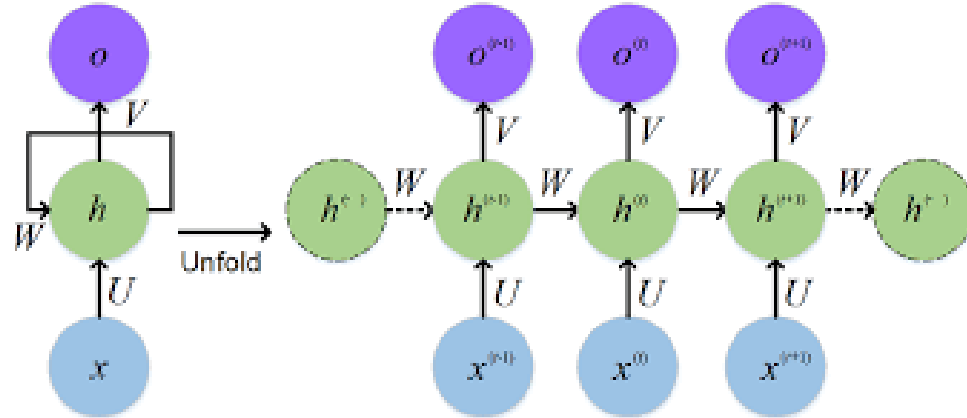
정승민

목차

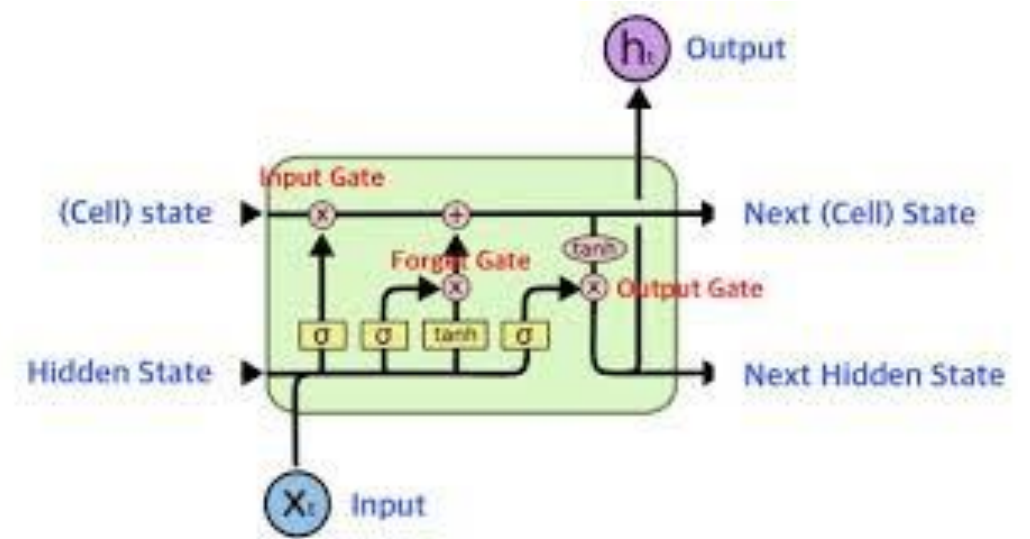
- 기존 NLP 모델의 문제점
- Attention mechanism 이란?
- Transformer 구조 알아보기
 - Encoder
 - Decoder
- ViT : Vision Transformer

RNN, LSTM의 문제점

RNN



LSTM



Input Gate : 새로운 정보를 셀 상태에 얼마나 잘 반영할지를 결정한다

Forget Gate : 이전 셀 상태의 정보를 얼마나 유지할지 결정한다.

Output Gate : 셀 상태의 정보를 얼마나 출력할지 결정한다.

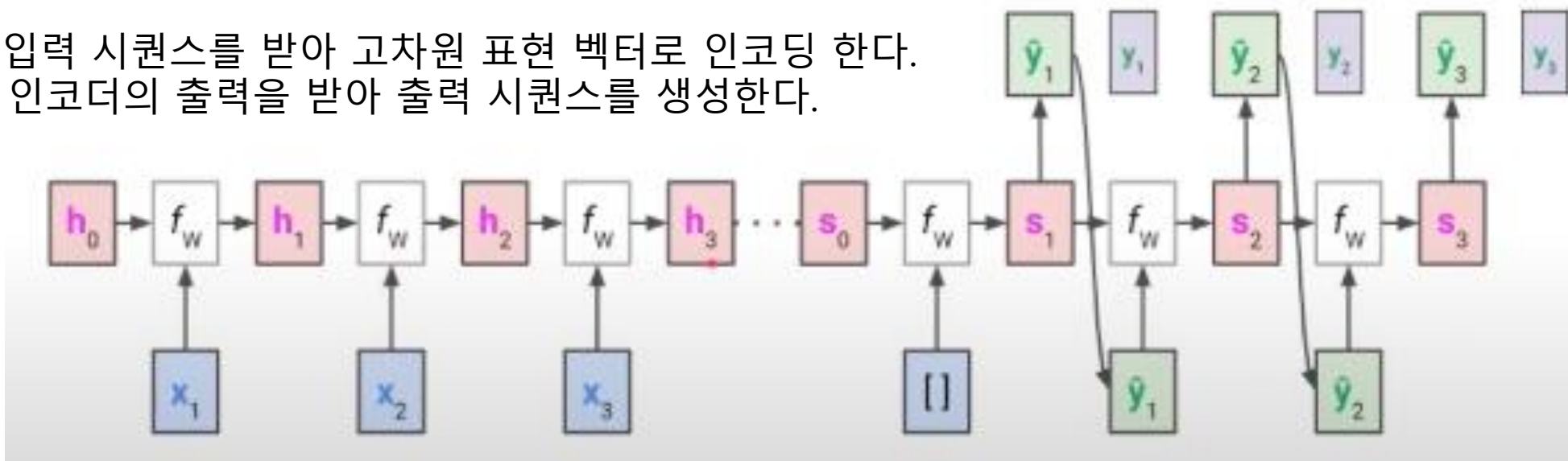
단점 :

1. 고정된 길이의 벡터로 압축
2. 출력 시퀀스의 유연성 부족. (입력과 출력의 길이가 달라야 하는 경우 처리하기 어렵다)
3. 1대1 매칭이 아닌 경우 처리하기 어렵다

Seq2Seq (encoder-decoder)

Encoder : 입력 시퀀스를 받아 고차원 표현 벡터로 인코딩 한다.

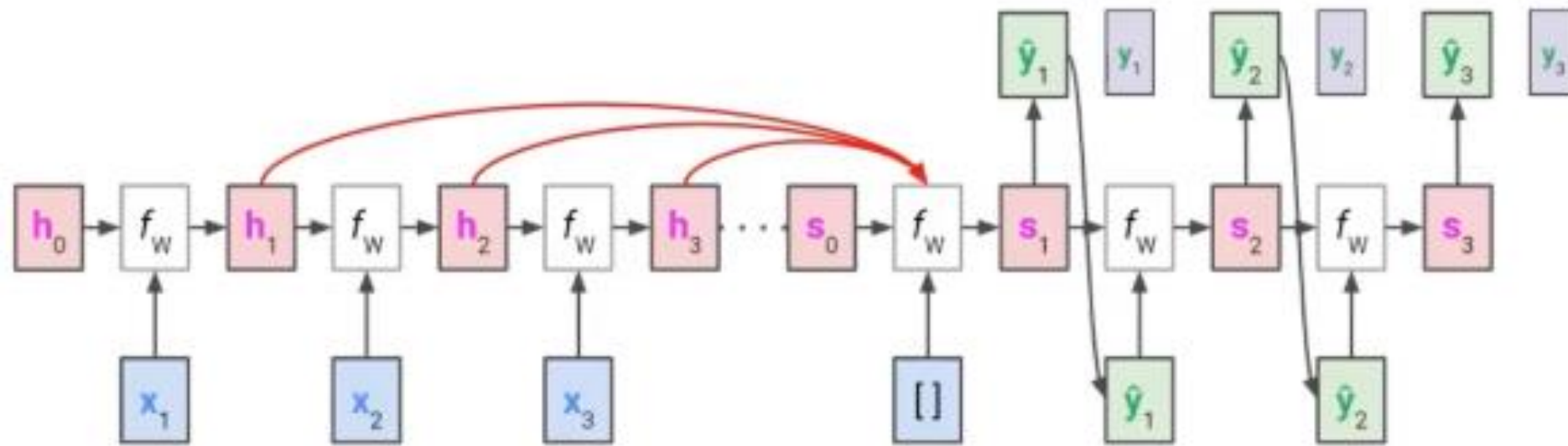
Decoder : 인코더의 출력을 받아 출력 시퀀스를 생성한다.



극복한점 : 입력길이와 출력길이를 유연하게 결정 할 수 있다.

하지만 여전히 RNN 기반의 연산을 진행하기에 긴 데이터가 들어오게 되면 이를 모두 하나의 벡터로 표현하기에 어렵다는 단점이 있습니다.

Attention mechanism



장거리 의존성을 증가시키기 위해서 앞에서 계산했던 과거의 hidden state 들을 모두 사용하는 아이디어

번역시 관련 있는 'hidden state'를 선택적으로 강조하여 Decoder에서 해당 부분을 더 집중 하여 보다 정확하고 문맥에 맞는 번역을 생성 할 수 있습니다.

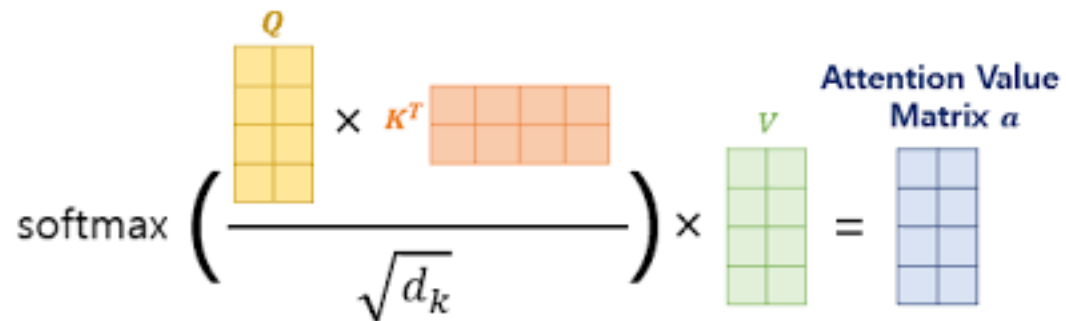
Attention mechanism

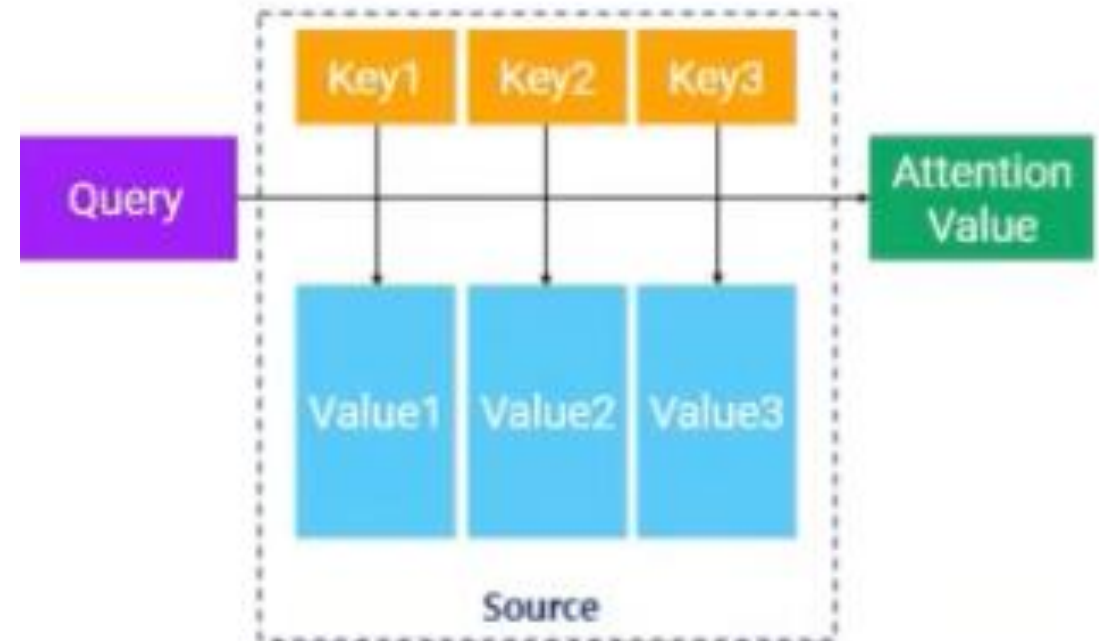
- 1. **Query (Q)**: 현재 상태에서 “내가 궁금한 정보”. : 1개
- 2. **Key (K)**: 각 데이터의 특성(고유한 특징). : N개
- 3. **Value (V)**: 각 데이터의 실제 정보. : N개

Query 주인공

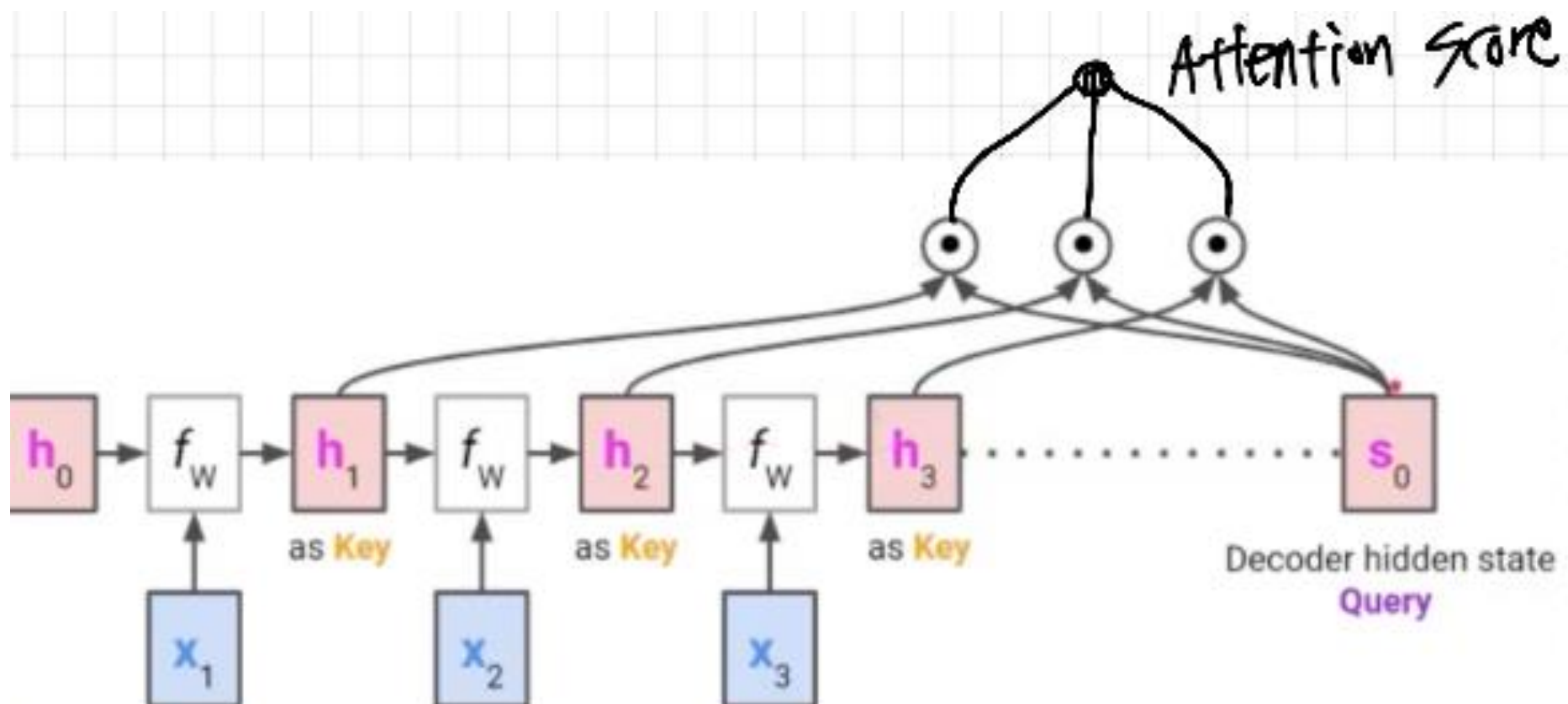
Key는 value의 대변인

Value는 실제 정보

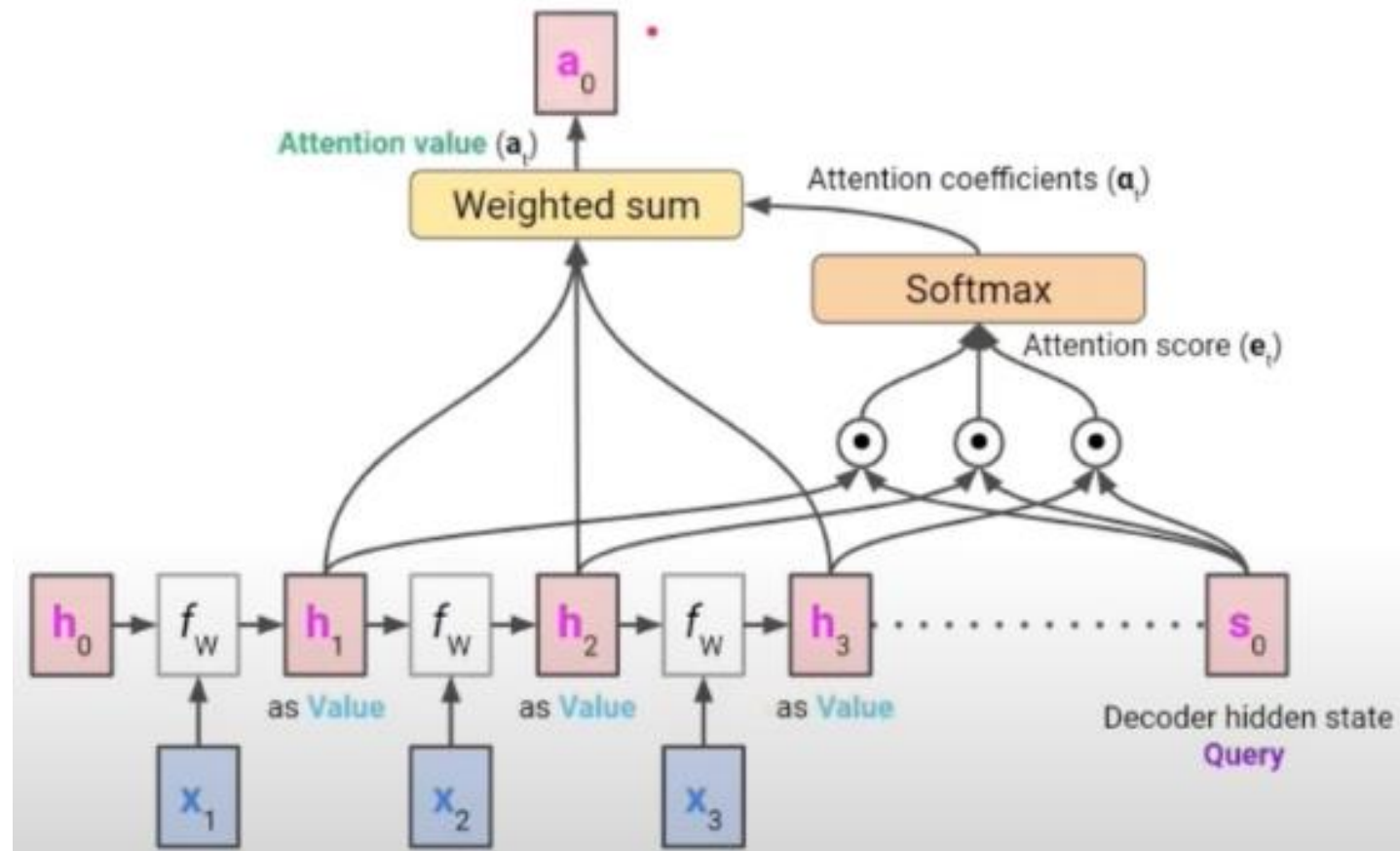
$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) \times V = \text{Attention Value Matrix } a$$




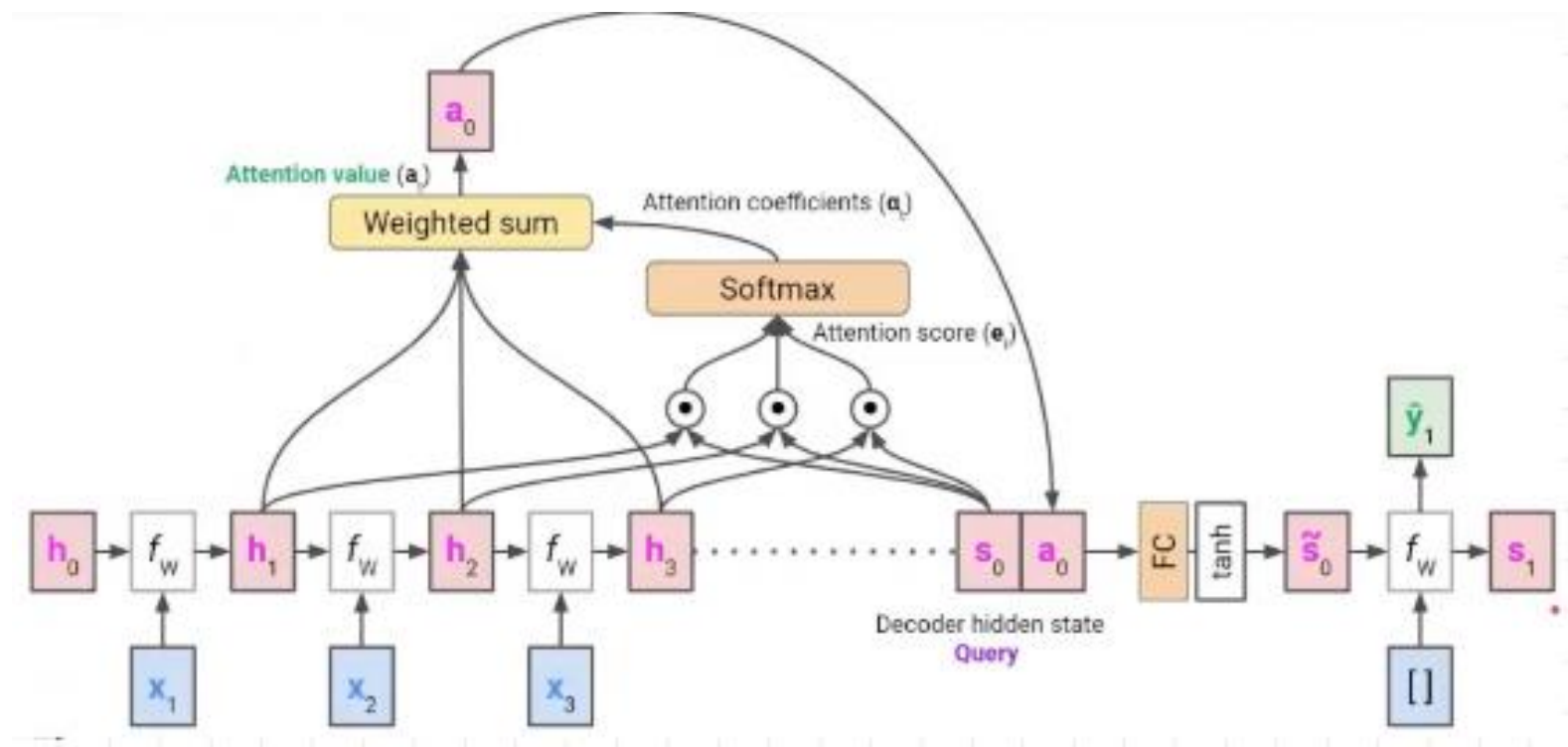
Attention mechanism



Attention mechanism



Attention mechanism



$$Q = \begin{array}{|c|} \hline 1 \\ \hline \end{array} \quad \begin{array}{|c|} \hline 5|2 \\ \hline \end{array} \quad (1, 5|2)$$

$$k = 3 \quad \begin{array}{|c|} \hline 5|2 \\ \hline \end{array} \quad (3, 5|2)$$

$$\checkmark Q = 3 \quad \begin{array}{|c|} \hline 5|2 \\ \hline \end{array} \quad (3, 5|2)$$

$$Qk^T = (1, 5|2) @ (5|2, 3) = (1, 3)$$

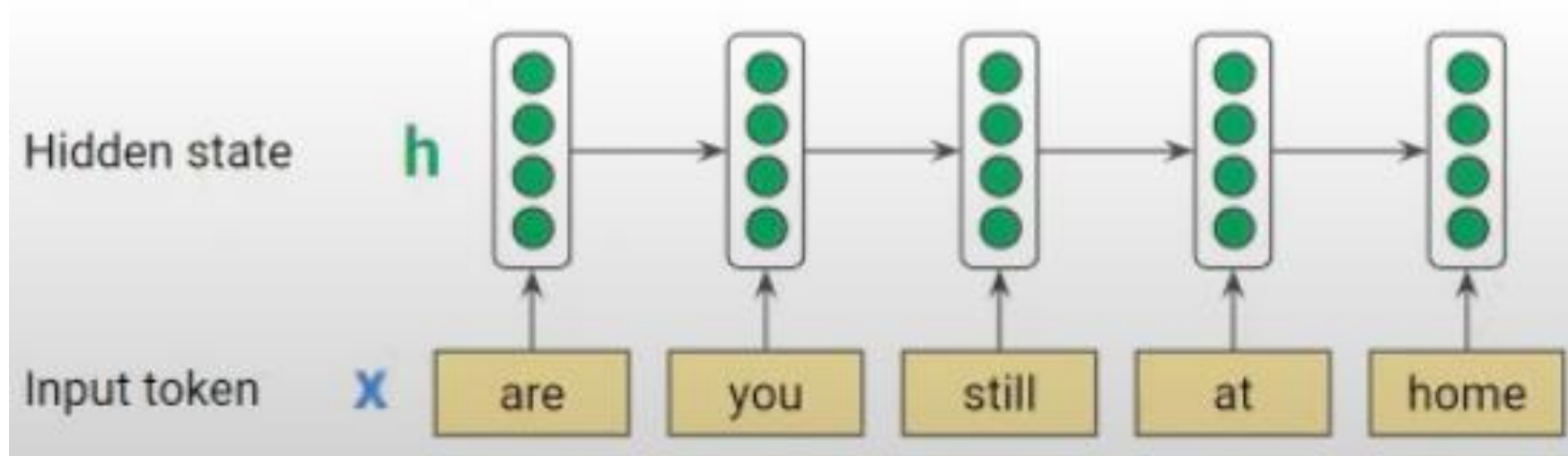
$$\text{softmax}(Qk^T) = (1, 3)$$

$$\text{softmax}(Qk^T) v = (1, 3) @ (3, 5|2) = (1, 5|2)$$

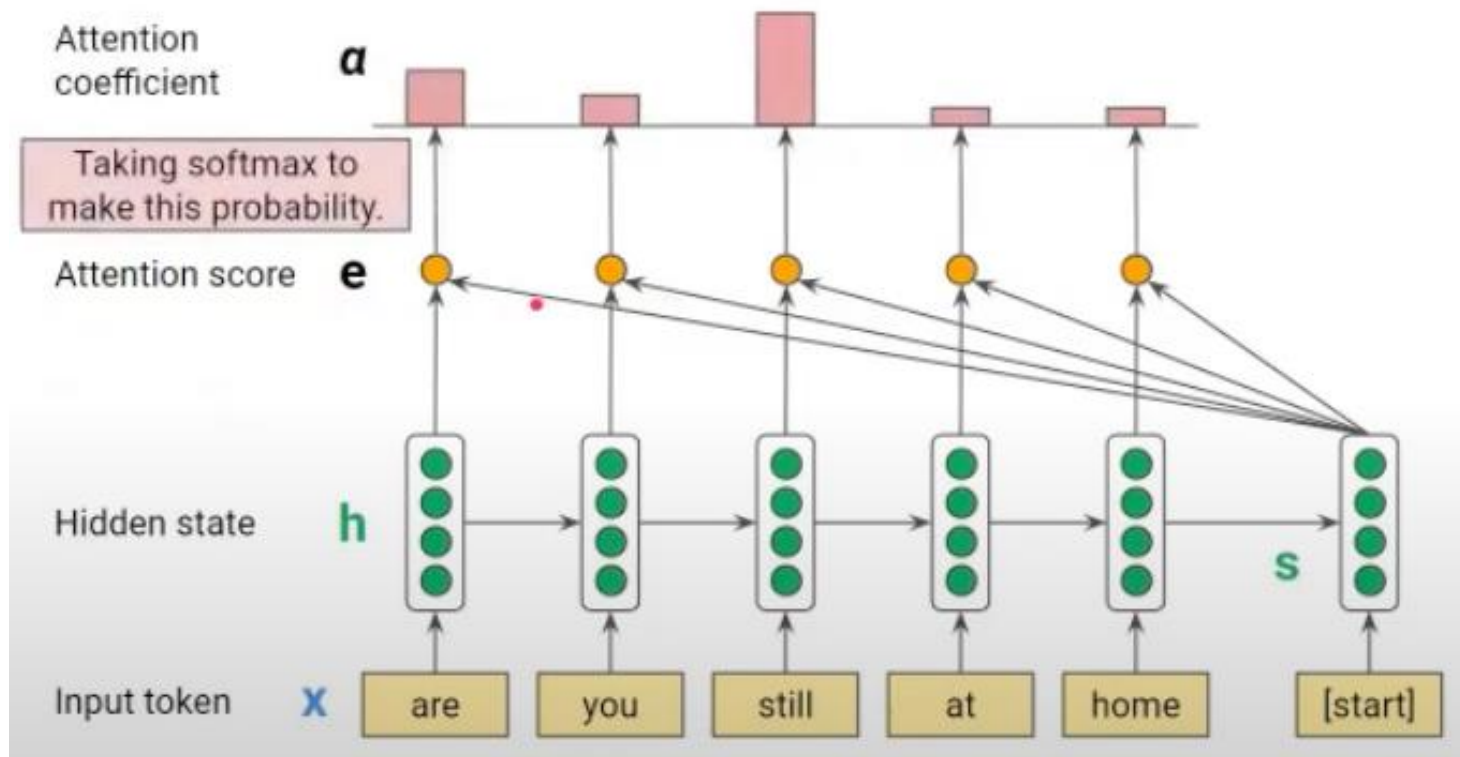
Attention 정리

- Query : decoder hidden state
- Key, Value : encoder hidden state { h_1, h_2, h_3 }
- Attention Value : weighted average of hidden states
 - Weight : similarity to S_0

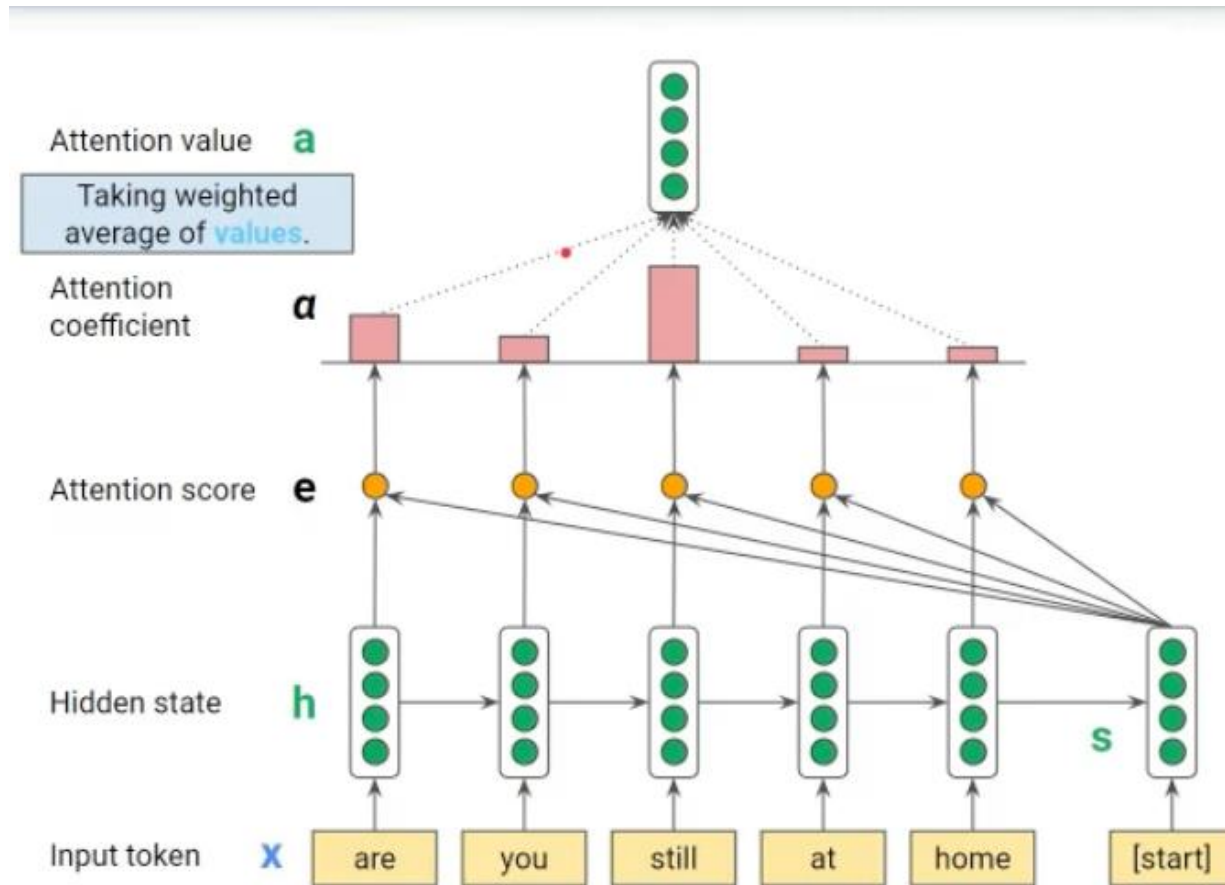
Encoding



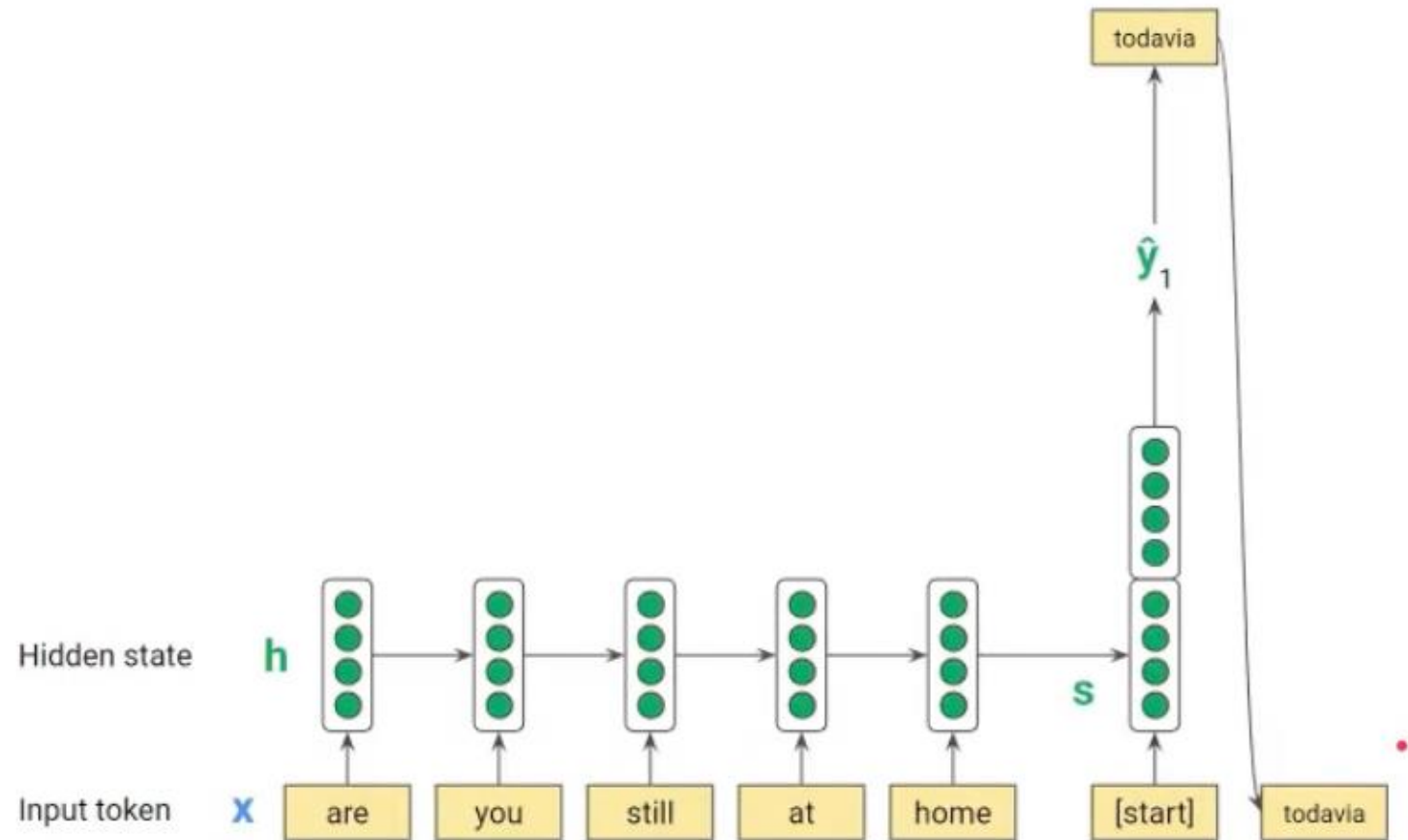
Decoding



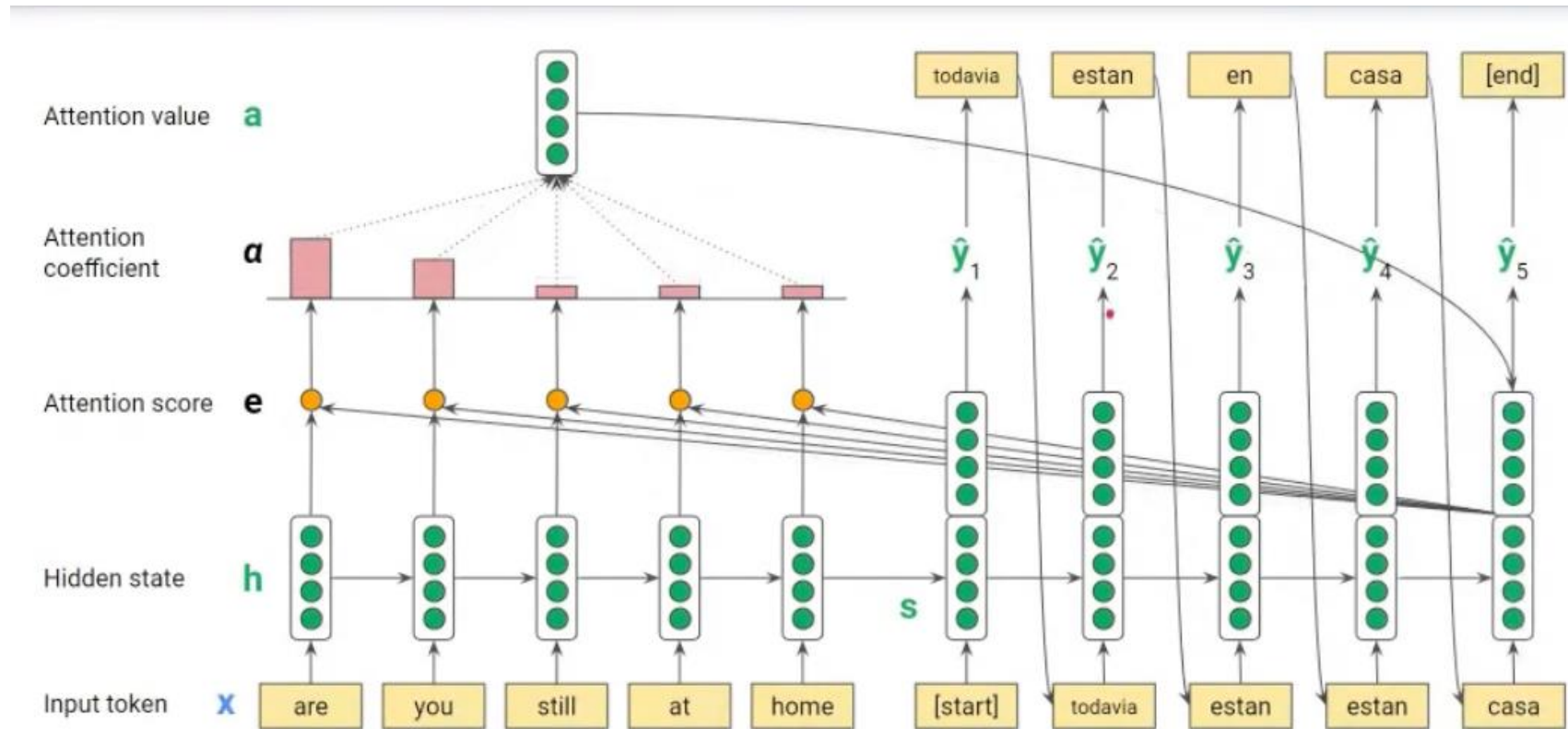
Decoding



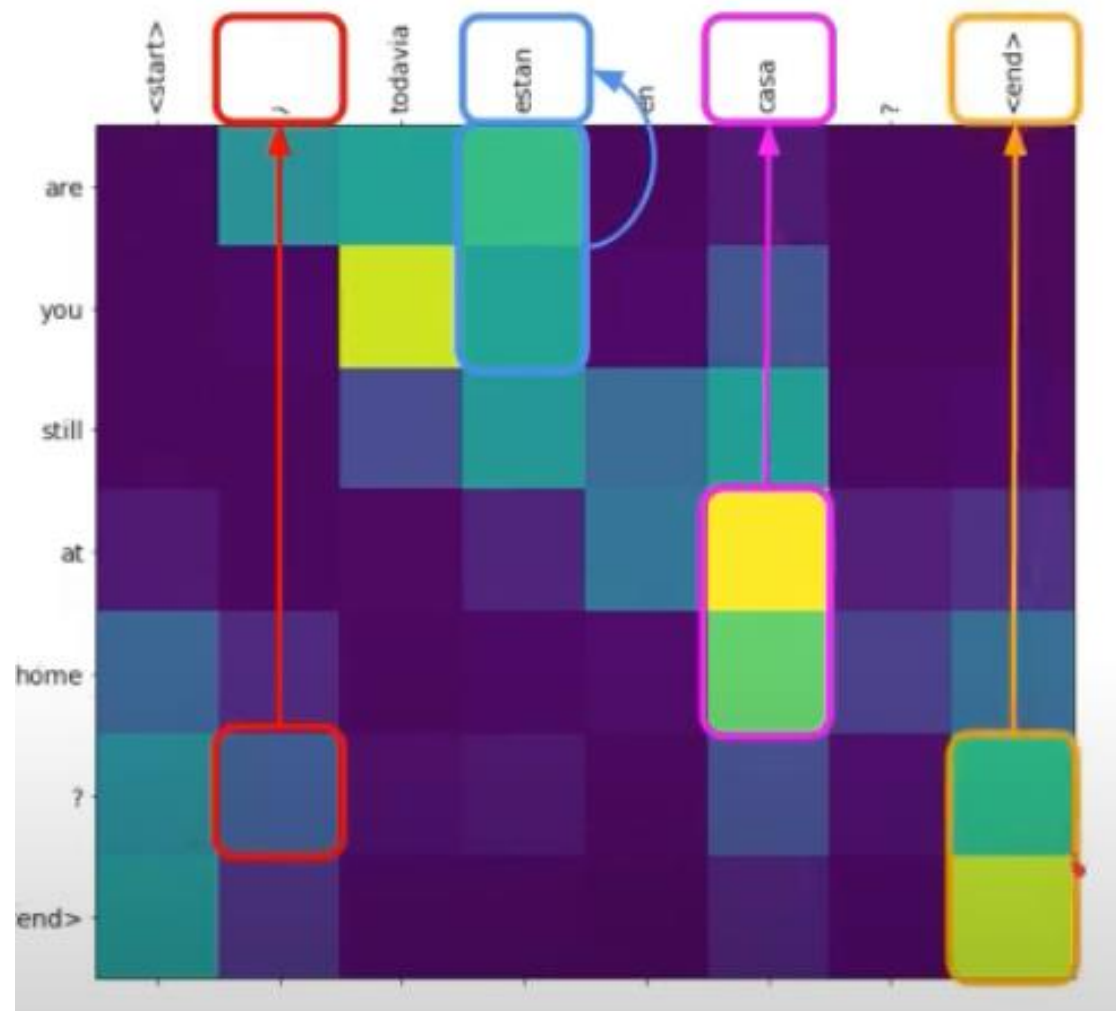
Decoding



Decoding



Attention Map



Transformer의 새로운 관점



사회 = 특정 모임
특정 모임 = 사람

다음과 같이 '사회' 는 더 작은 그룹으로 나눌수 있고,
이렇게 나뉘어진 것들은 서로 연관성이 가지고 있다.

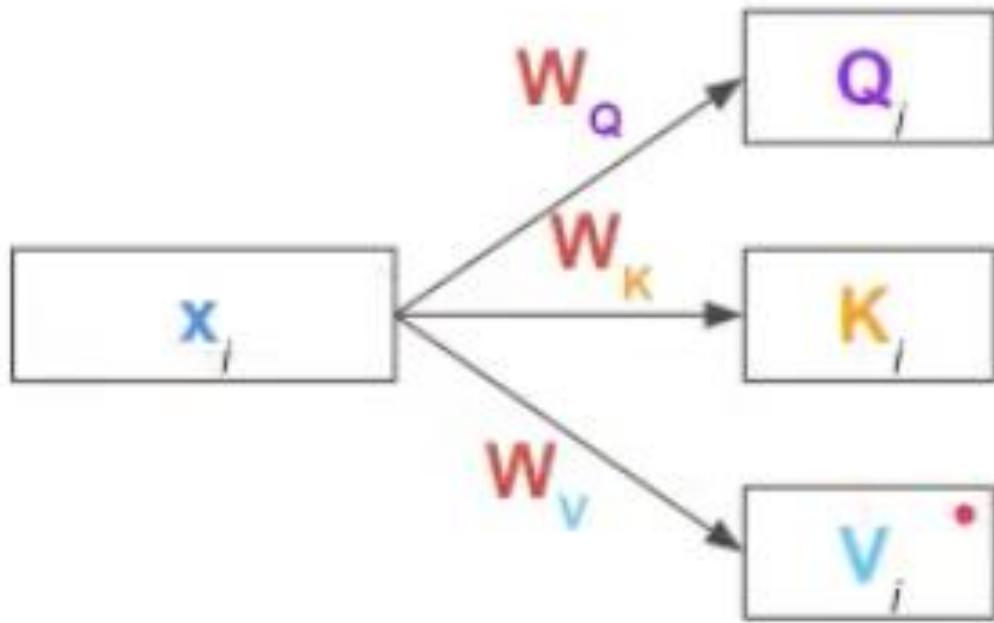
즉, 문장은 단어, 토큰 등으로 나뉘수 있고, 각 단어,토큰
들은 서로 연관성을 가지고 있다.

Transformer

- 내가 키우는 귀여운 고양이
- 참새를 잡아먹는 고양이
- 게임 속 고양이 캐릭터

" 고양이 " 라는 단어가 문장 내 다른 구성요소들과의 관계속에서 해당 의미가 유의미하게 바뀌게 됩니다.
이러한 관계를 Attention을 통해서 자기 스스로 표현하는 방법이 바로 Self Attention 입니다.

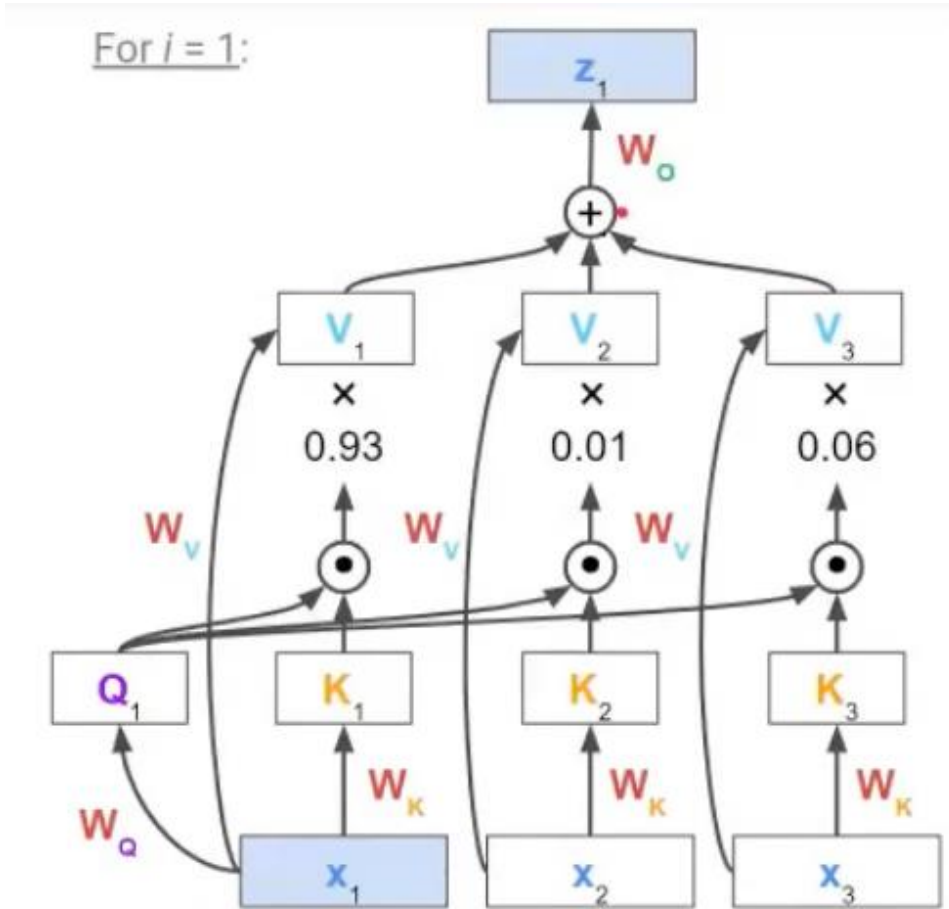
Transformer



다른 x 와 관계속에서 x_i (나)를 다시 표현한다

Transformer

즉, 다른 요소들과의 관계속에서 나를 표현하자! contextualize



Transformer Architecture

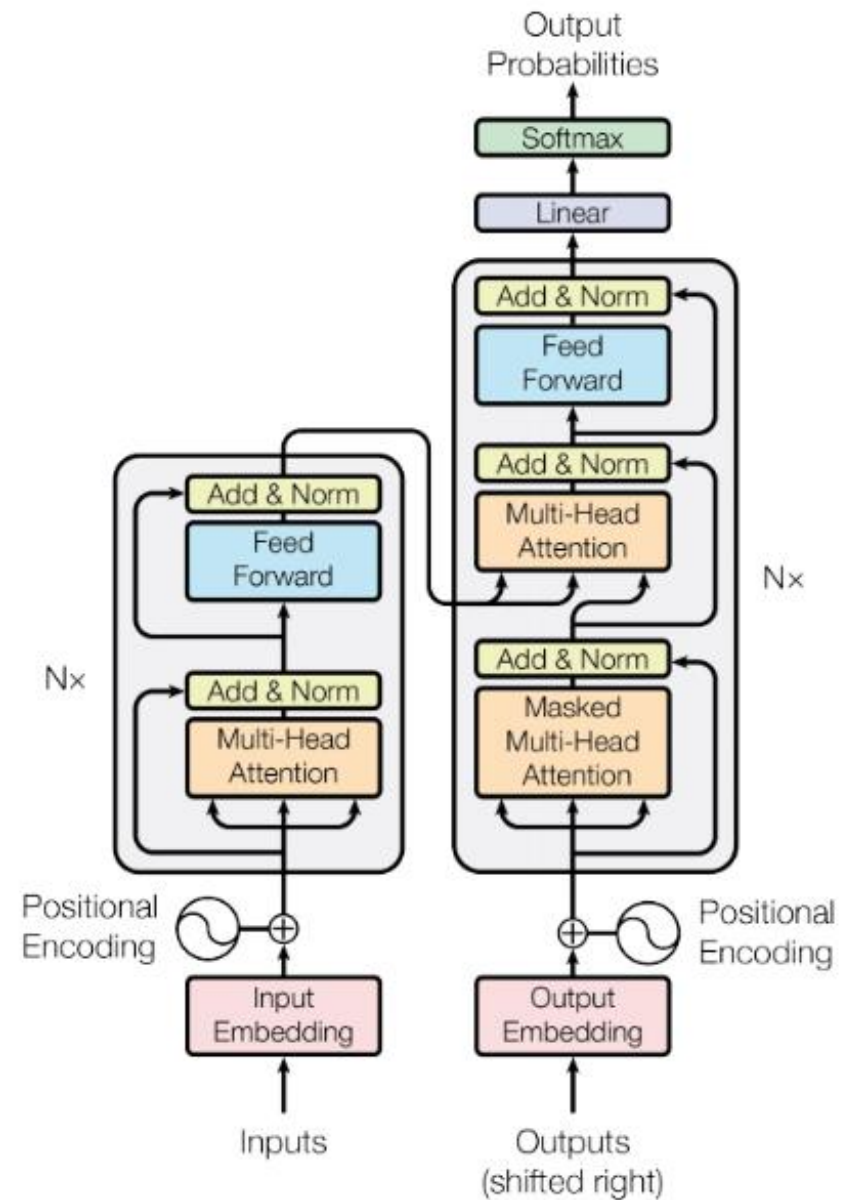
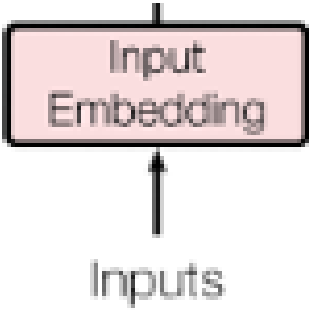


Figure 1: The Transformer - model architecture.

Transformer Architecture : Input



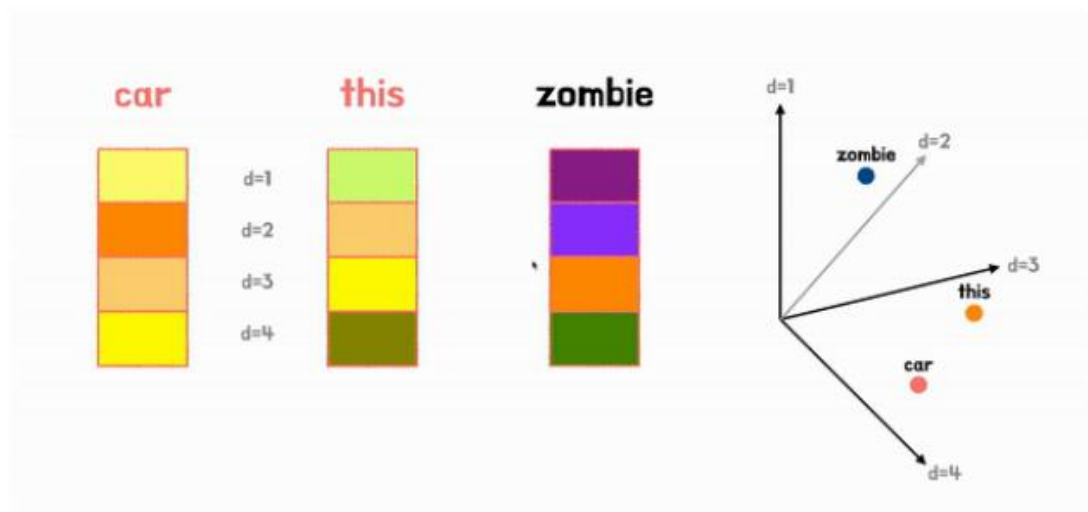
Vocab에서 해당 단어와 매칭되는 index를 input으로 변경



각 index들을 특정 vector로 변경



시각화



Transformer : positional Encoding

① Although I did **not** get 95 in last TOEFL, I could get in the Ph.D program.

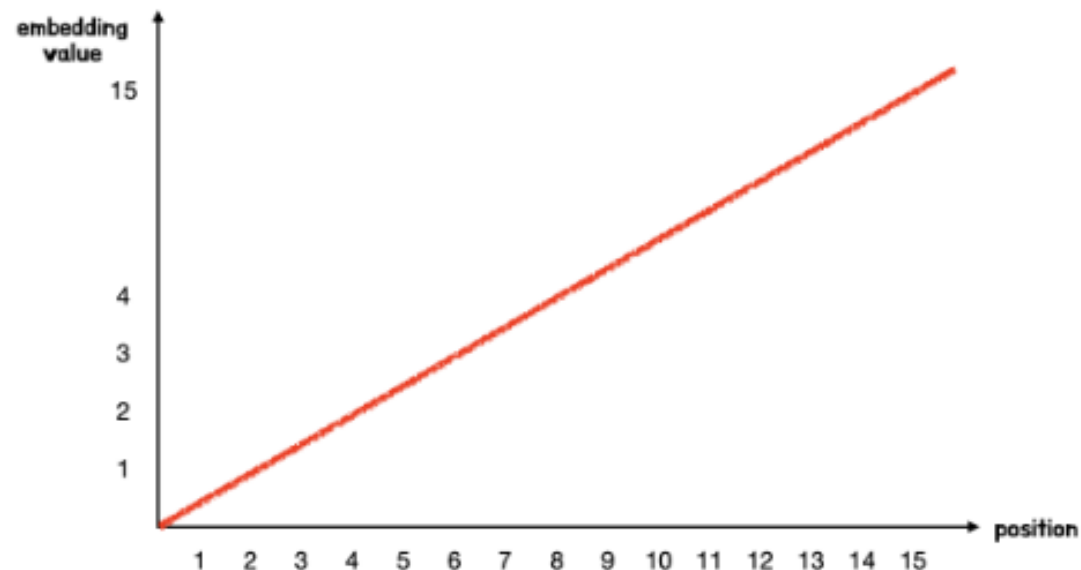
② Although I did get 95 in last TOEFL, I could **not** get in the Ph.D program.

규칙

1. 모든 위치 값은 시퀀스의 길이나 input값에 상관없이 동일한 식별자를 갖는다.
2. 모든 위치 값은 너무 크면 안된다.

Transformer : positional Encoding

1. 정수로 그냥 나타내면 안되나?



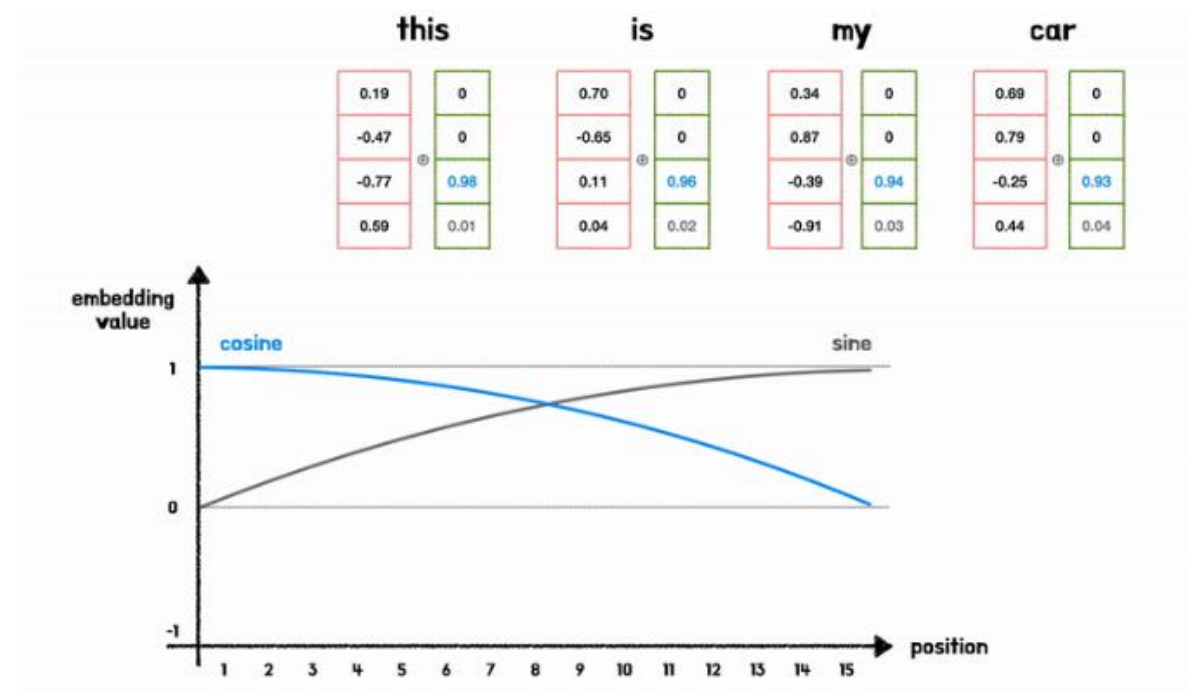
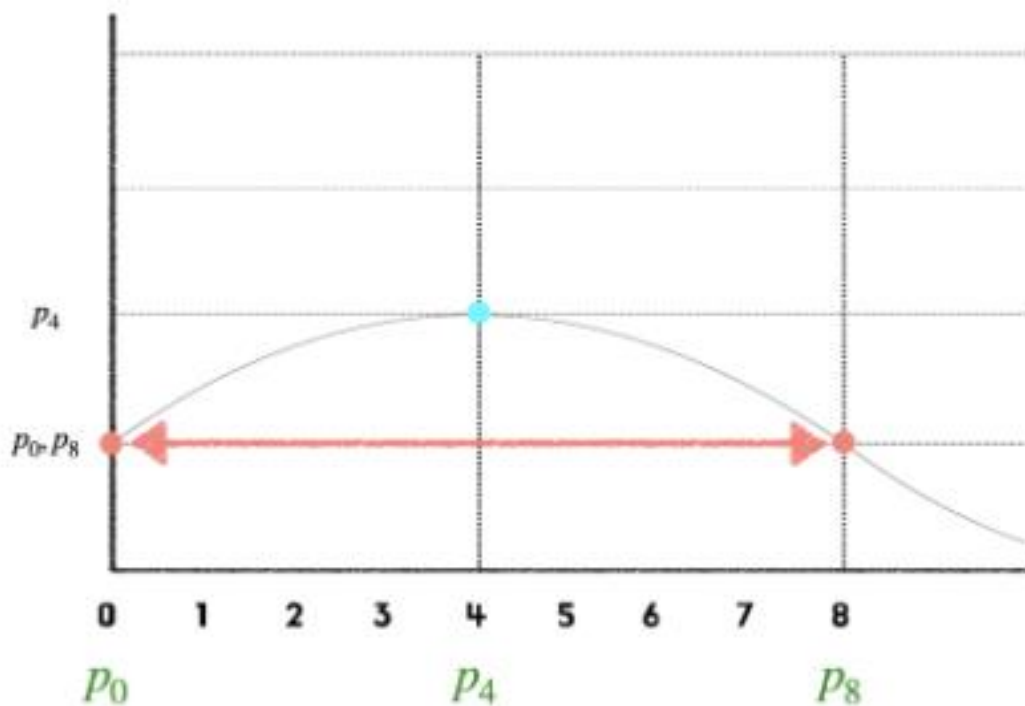
2. 토큰 수로 정규화를 한다면?

-> 토큰의 길이가 달라지면 각 위치의 고유성이 사라진다.

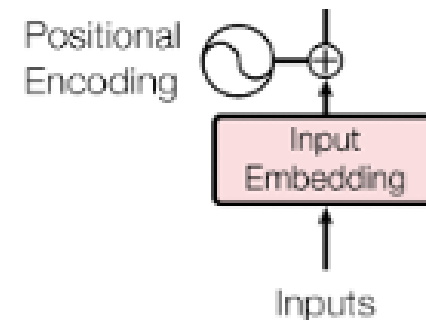
Transformer : positional Encoding

Sin, Cos \rightarrow -1과 1 사이의 값을 갖는다.

주기함수이기에 입력 길이가 길어도 상관없이 무한히 반복가능하다.



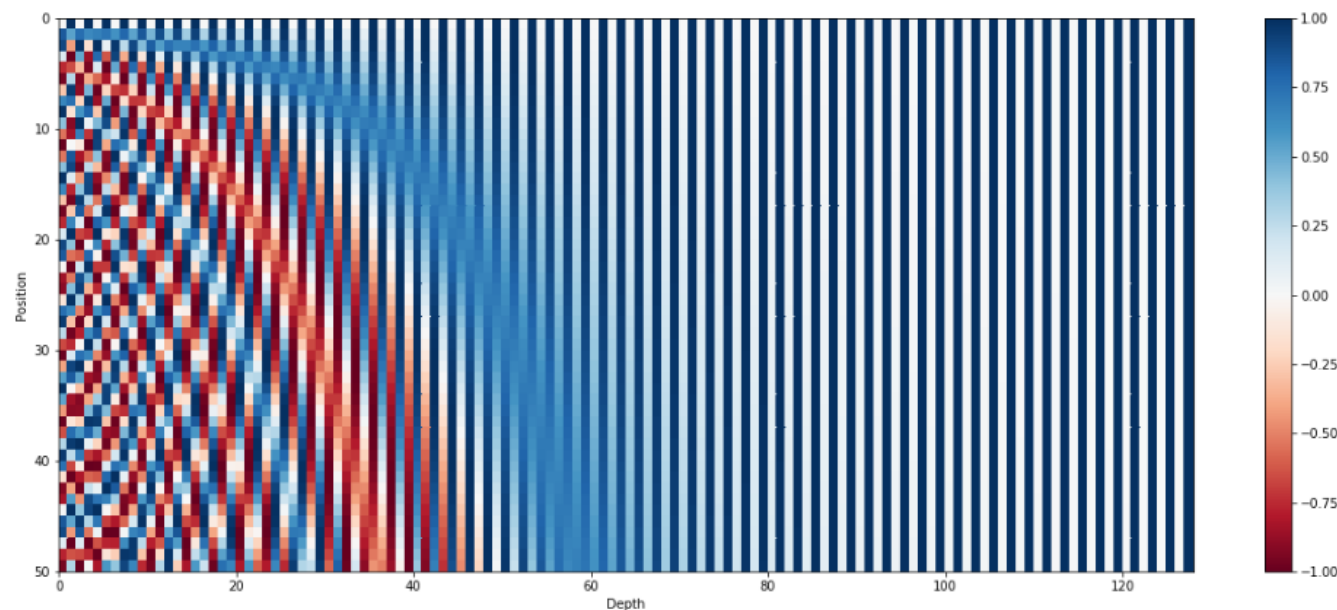
Transformer : positional Encoding



$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

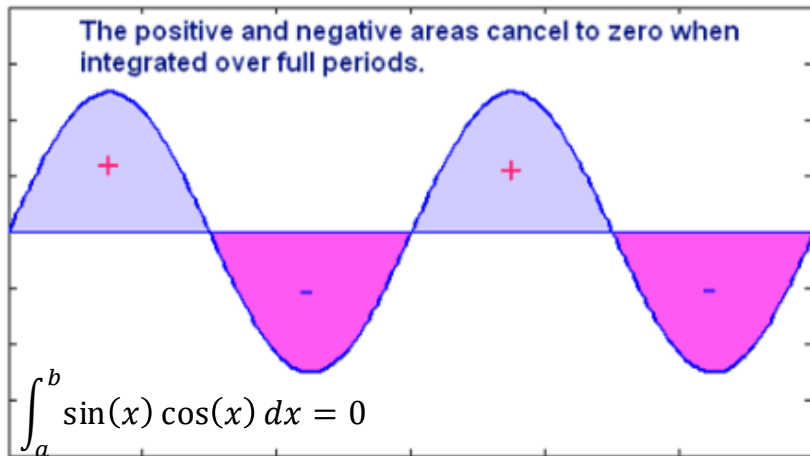
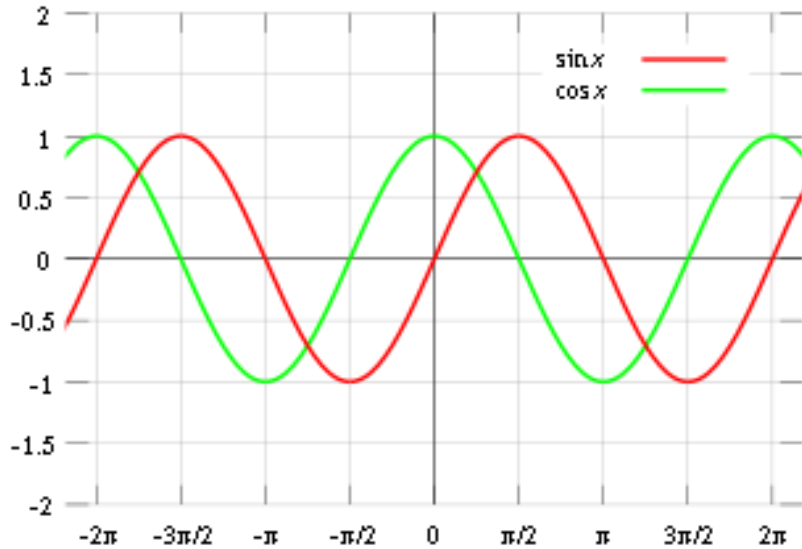
$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

10,000이라는 수는 논문의 저자들이 경험적으로 선택한 값, 임베딩 차원보다 크면 된다 = 임베딩 차원내에서 주파수가 중복되지만 않으면 된다.



낮은 차원(i 가 작을 때)은 분모가 비교적 작아 주파수가 낮으며, 위치 변화에 느리게 반응한다. (긴 주기)
 높은 차원(i 가 클 때)은 분모가 커져서 주파수가 높으며, 위치 변화에 민감하게 반응한다. (짧은 주기)

Transformer : 삼각함수를 사용하는 이유



Fourier series

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{2\pi n}{T}x\right) + b_n \sin\left(\frac{2\pi n}{T}x\right) \right]$$

다음과 같이 내적값이 0이 되기 때문에 $\sin(x)$, $\cos(x)$ 가 주기마다 서로 직교성을 갖게 됩니다. = 주기성을 갖는 모든 주기함수를 span(설명)할 여기에 수식을 입력하십시오. 수 있다.

특정 주파수 대역의 위치 정보가 서로 구별 가능하고 섞이지 않는다는 것을 뜻한다.

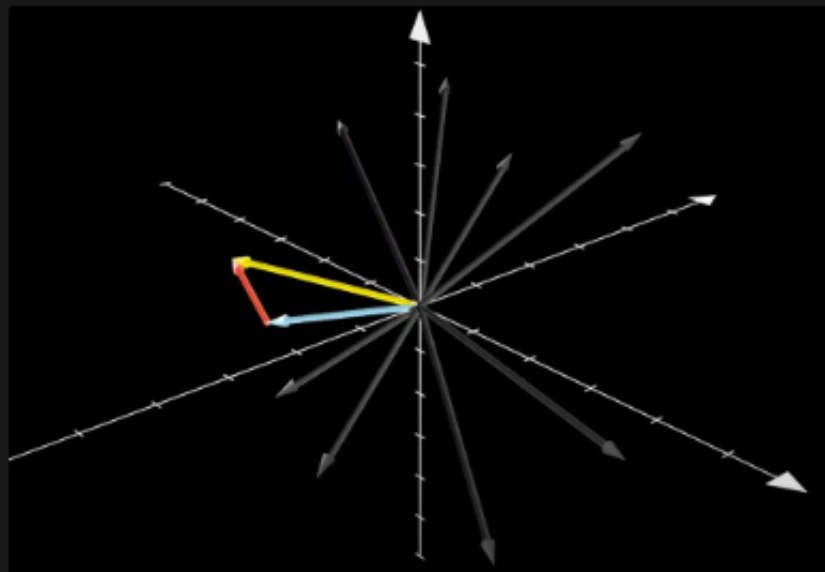
이는 모델이 어떤 특정 위치 패턴을 학습할 때, 다른 차원에서 오는 잡음이나 상관성을 최소화 할 수 있다.

Span : vector들의 **Linear combination**으로 나타낼수 있는 모든 vector의 집합. (내가 가진 vector로 표현 가능한 영역(공간))

Linear Combination이란 [스칼라 X 벡터] 의 합으로 표현하는것

$$x_1V_1 + x_2V_2 + \dots + x_pV_p$$

이때 V_p 를 얼마나 쓸까? 를 의미하는게 x_p 가 됩니다.



다음과 같이 3차원에서 2개의 서로 독립인 벡터를 선형조합으로 나타낼 수 있는 모든 벡터들을 그려보면 다음과 같이 2차원을 **Span** 한다는 것을 알수 있습니다.

그리고 Column Vector들이 **span**하는 영역을 Column space = $\text{range}(A) = C(A)$ 라고 표현합니다.

Linearly Independent & Basis

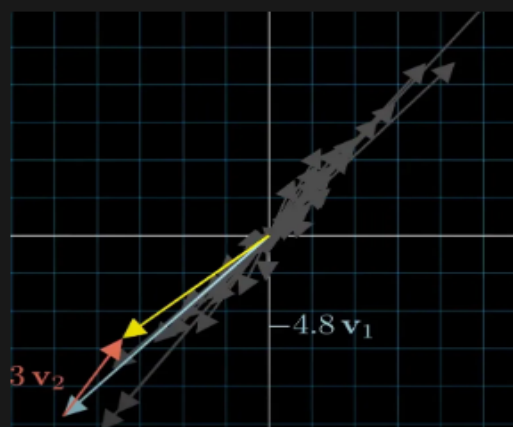
Linearly Independent란

$$a_1V_1 + a_2V_2 + \dots + a_pV_p = 0$$

위 수식을 만족하는 조합이 오직 아래와 같이 모든 계수가 0인 경우

$$a_1 = a_2 = a_3 = \dots = a_p = 0$$

이때 우리는 모든 Vector들이 linearly independent 하다고 합니다.
즉 각각의 벡터가 서로다른 차원으로 영향력을 가지고 있음을 의미합니다.



다음과 같이 **Linearly Independent** 한 두 벡터를 선형조합하게 되면 결국에는 2차원 평면 전체를 **span**할수 있게 됩니다.

그리고 그 중 특별한 Case가 바로 **Orthogonal** (직교) 하는 경우 입니다.

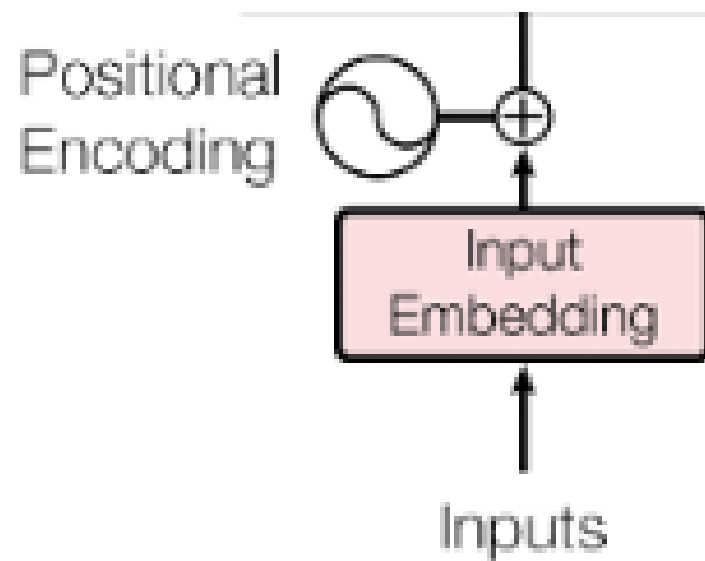
Basis : 특정 space를 구성하는 필수적인 벡터

예를들어 2차원 공간의 **Basis**는 아래와 같이 나타낼수 있습니다.

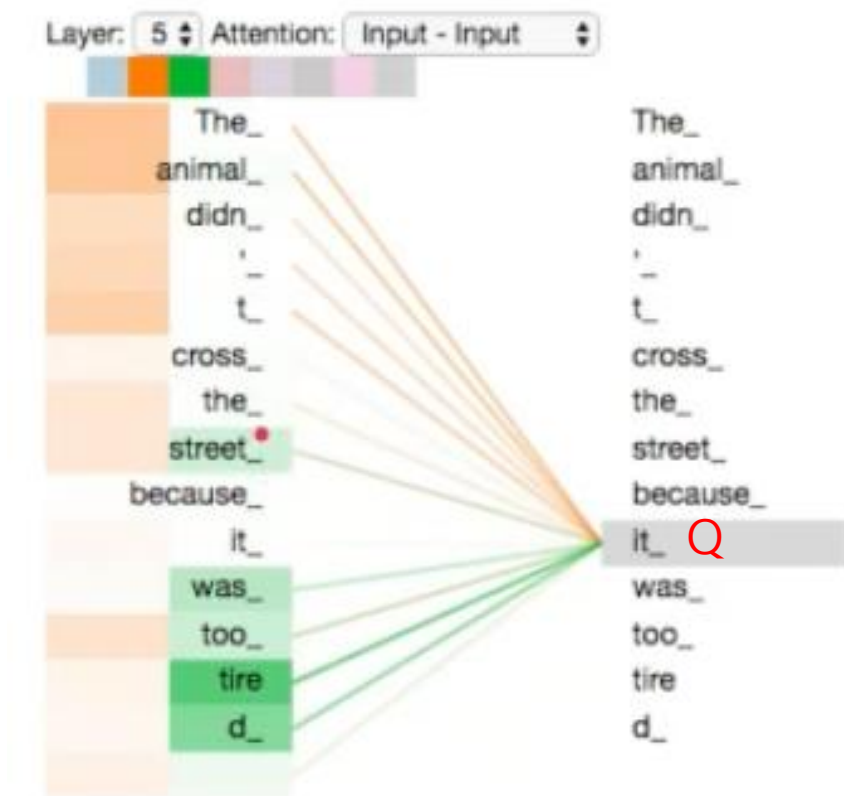
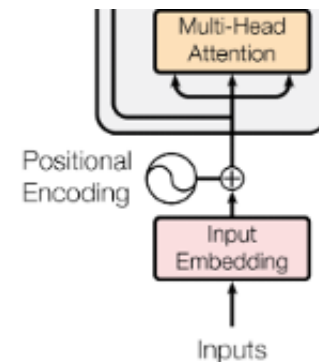
$$\alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \alpha \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

이게 아니더라도 서로 독립인 두개의 벡터라면 뭐든 **Basis**가 될수 있습니다.

Input 준비 완료



Transformer : Multi-Head Attention

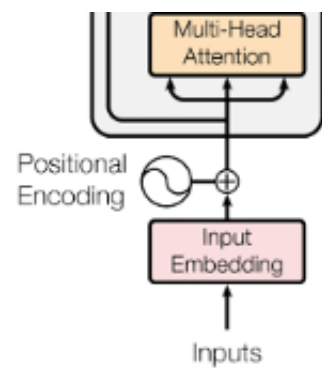
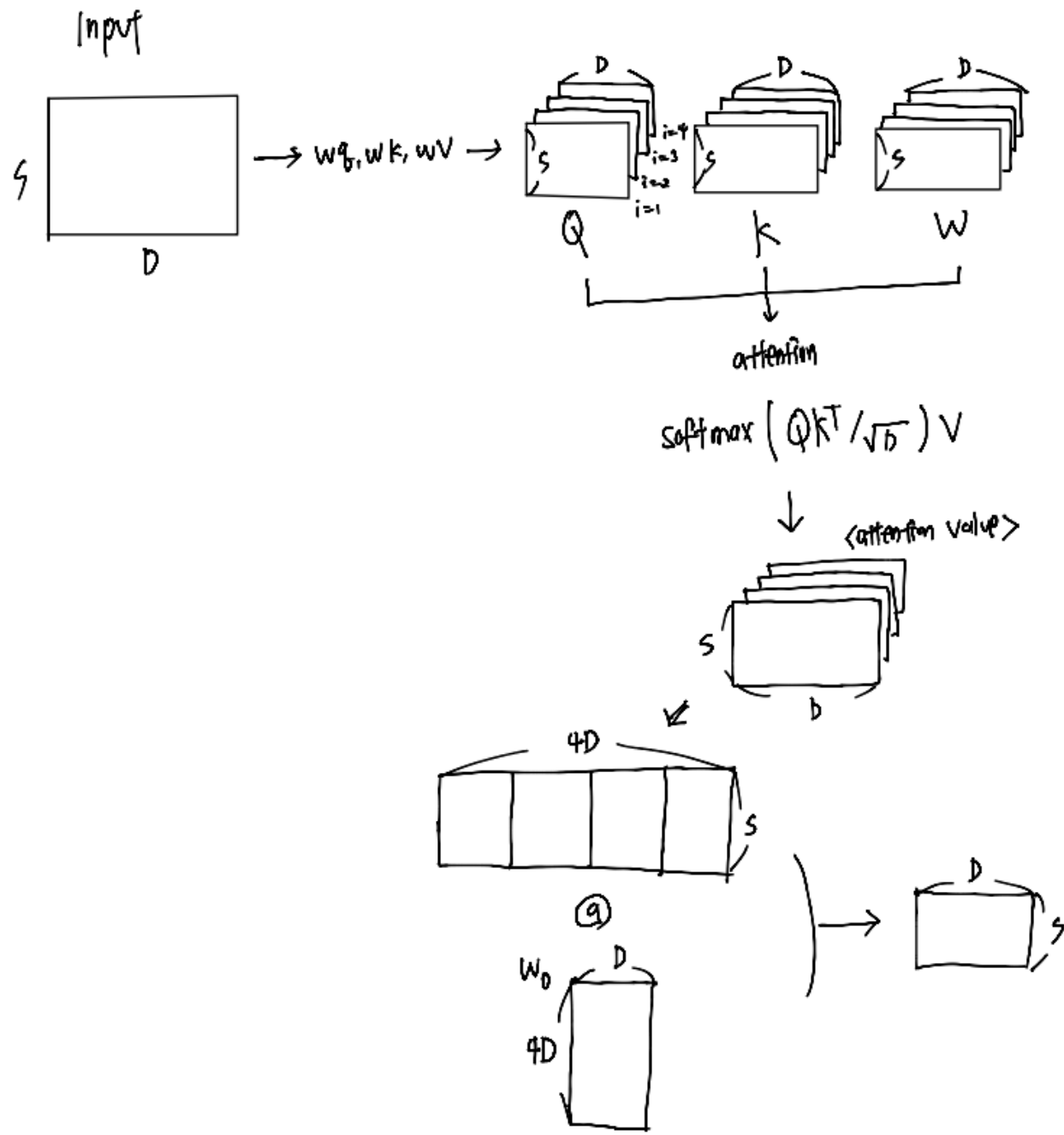


Input shape : (S, D)
D = 512

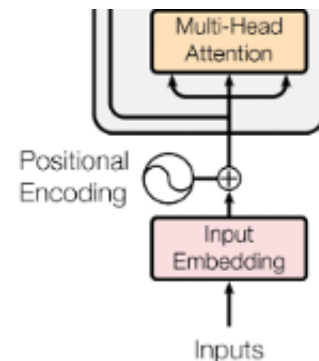
$$X \rightarrow \begin{pmatrix} w_1^q, w_1^k, w_1^v \xrightarrow{A} (S, 512) \\ \vdots \\ w_8^q, w_8^k, w_8^v \xrightarrow{A} (S, 512) \end{pmatrix} \rightarrow (S, 512 \times 8) \xrightarrow{w_o} (S, 512)$$

Encoder

num_heads = 4



Transformer : Multi-Head Attention



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

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

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

In this work we employ $h = 8$ parallel attention layers, or heads. For each of these we use $d_k = d_v = d_{\text{model}}/h = 64$. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

$$d_{\text{model}} = D // \text{num_head}$$

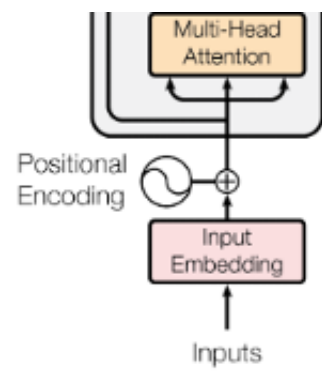
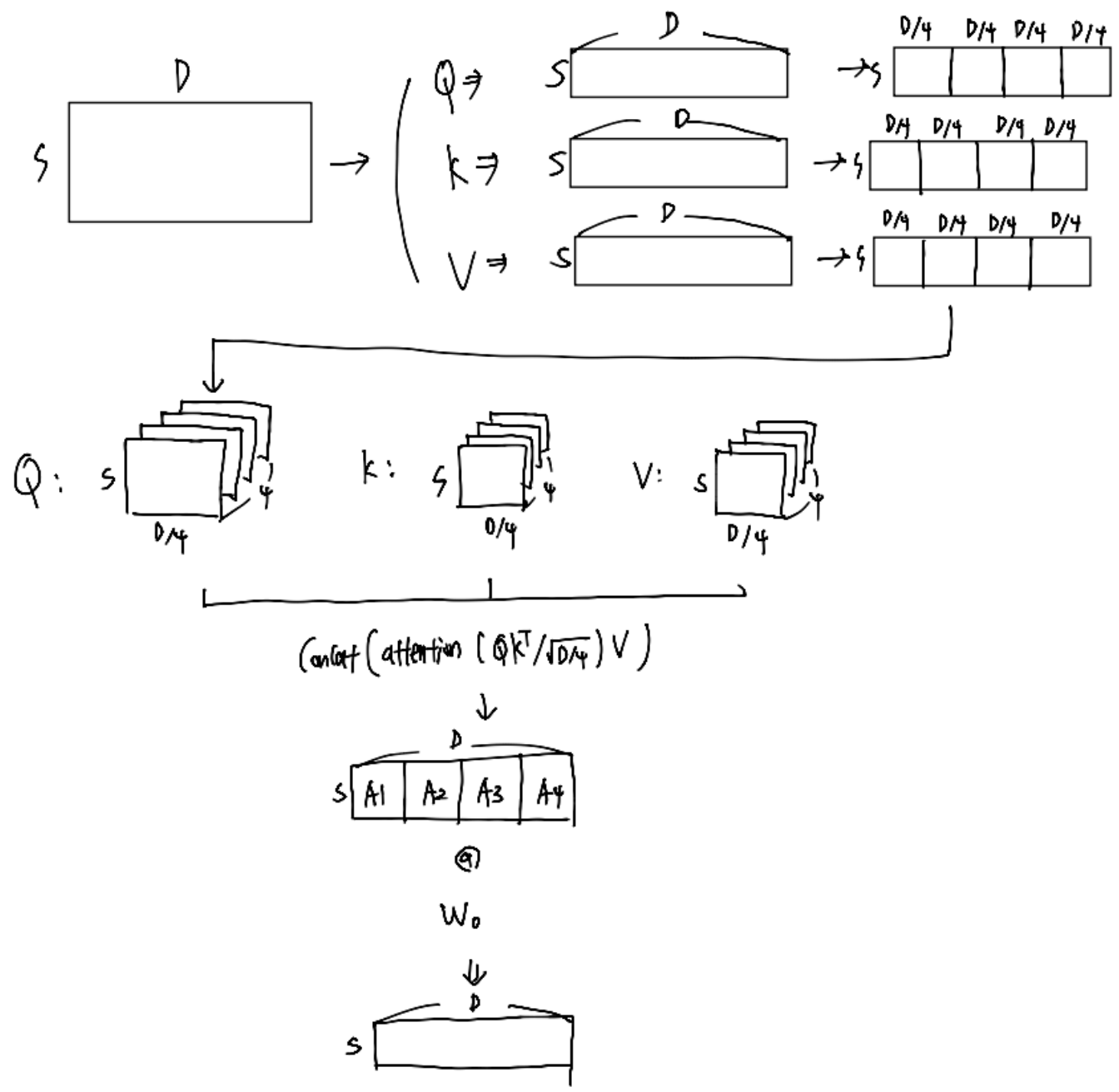
$$\text{If } D=512, \text{ num_head}=8, W_0 = \text{nn.Linear}(d_{\text{model}}, D)$$

$$x \rightarrow \begin{pmatrix} w_{1q}, w_{1k}, w_{1v} \rightarrow (5, 64) \\ \vdots \\ w_{8q}, w_{8k}, w_{8v} \rightarrow (5, 64) \end{pmatrix} \rightarrow (5, 8 \times 64) \xrightarrow{W_0} (5, 512)$$

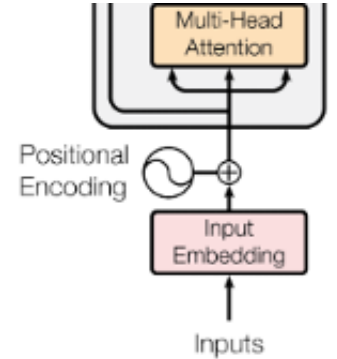
Q,K,V를 num_head만큼 진행하는 것이 아니라 하나의 Q,K,V를 num_head만큼 분할하여 구현하여 Single Attention 과 비슷한 연산량을 보여주면서 더 다양한 패턴을 학습할 수 있도록 한다.

Encoder

num_heads = 4



Transformer : Multi-Head Attention



- 개념적 접근 :

num_head 개의 Q, K, V -> 각각 512차원 생성

각 헤드에서 독립적으로 512차원 Attention

num_head 개의 512차원 결과를 결합 및 선형 변환하여 결과를 얻는다

- 구현적 접근 :

하나의 Q,K,V -> 각각 512차원 생성

각 Q,K,V를 num_head 개의 64차원으로 분할

각 헤드에서 독립적으로 64차원 Attention 계산

num_head 개의 D/ num_head 차원을 결합 및 선형 변환하여 결과를 얻는다.

=> single Head Attention과 비슷한 연산량을 갖는다.

=> 저의 생각에는 Multihead를 capacity관점 보다는 병렬/효율 처리에 초점

Transformer : Multi-Head Attention

```
class MultiheadAttention(nn.Module):
    """Multihead Attention 모듈.

    Multi-Head Attention을 구현하며, Q, K, V를 여러 헤드로 분리하여
    각각 독립적으로 어텐션을 수행한 후 결과를 연결(concat)하고 선형 변환을 적용합니다.
    """
    def __init__(self, in_channels, num_head):
        """MultiheadAttention 모듈을 초기화합니다.

        Args:
            in_channels (int): 입력의 임베딩 차원.
            num_head (int): 사용할 헤드의 개수.
        """
        super().__init__()
        self.in_channels = in_channels
        self.num_head = num_head

        assert in_channels % self.num_head == 0, 'in_channels는 num_head로 나누어 떨어져야 합니다.'
        self.d_model = in_channels // self.num_head

        self.WQ = nn.Linear(self.in_channels, self.in_channels)
        self.WK = nn.Linear(self.in_channels, self.in_channels)
        self.WV = nn.Linear(self.in_channels, self.in_channels)

        self.WO = nn.Linear(self.in_channels, self.in_channels)

        self.layer_norm = nn.LayerNorm(in_channels)
```

```
def forward(self, x):
    """Multi-Head Attention을 적용합니다.

    Args:
        x (Tensor): 입력 텐서 (B, S, D).

    Returns:
        Tensor: Multi-Head Attention과 잔차 연결을 거친 출력 (B, S, D).
    """
    print(f'\t\t[ MultiHead Attention ]')

    residual = x
    B, S, D = x.size()

    Q = self.WQ(x)
    K = self.WK(x)
    V = self.WV(x)

    Q = Q.view(B, S, self.num_head, self.d_model).transpose(1, 2)
    K = K.view(B, S, self.num_head, self.d_model).transpose(1, 2)
    V = V.view(B, S, self.num_head, self.d_model).transpose(1, 2)

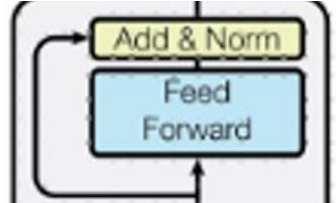
    attention_score = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(self.d_model)
    attention_value = torch.matmul(F.softmax(attention_score, dim=-1), V)

    out = attention_value.transpose(1, 2).contiguous().view(B, S, D)

    out = self.WO(out)

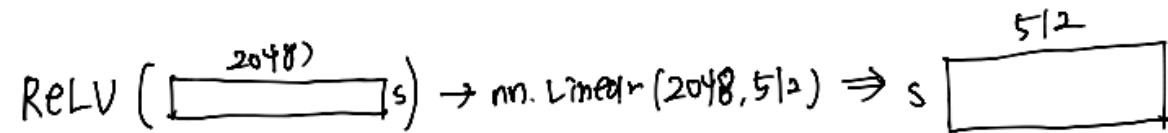
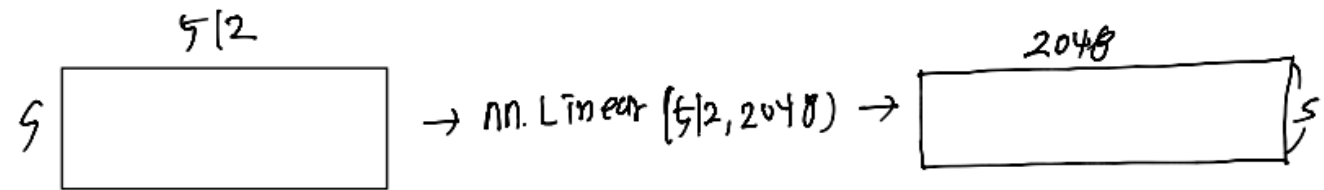
    out = out + residual
    out = self.layer_norm(out)
    return out
```

Transformer : Position-wise Feed-Forward Networks



The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{\text{ff}} = 2048$.

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



Transformer : Position-wise Feed-Forward Networks

```
class FeedForward(nn.Module):
    """FeedForward 모듈.

    Encoder에서 사용되는 Feed-Forward Network (FFN)를 구현합니다.
    """
    def __init__(self, in_channels, ffn_expand):
        """FeedForward 모듈을 초기화합니다.

        Args:
            in_channels (int): 입력값의 임베딩 차원.
            ffn_expand (int): 첫 번째 선형 계층에서 확장할 차원.
        """
        super().__init__()
        self.in_channels = in_channels
        self.ffn_expand = ffn_expand

        self.fc_out1 = nn.Linear(self.in_channels, self.ffn_expand)
        self.relu = nn.ReLU()
        self.fc_out2 = nn.Linear(self.ffn_expand, self.in_channels)
        self.layer_norm = nn.LayerNorm(in_channels)
```

```
def forward(self, x):
    """Feed-Forward Network를 적용합니다.

    Args:
        x (Tensor): Multihead Attention의 결과 (B, S, D).

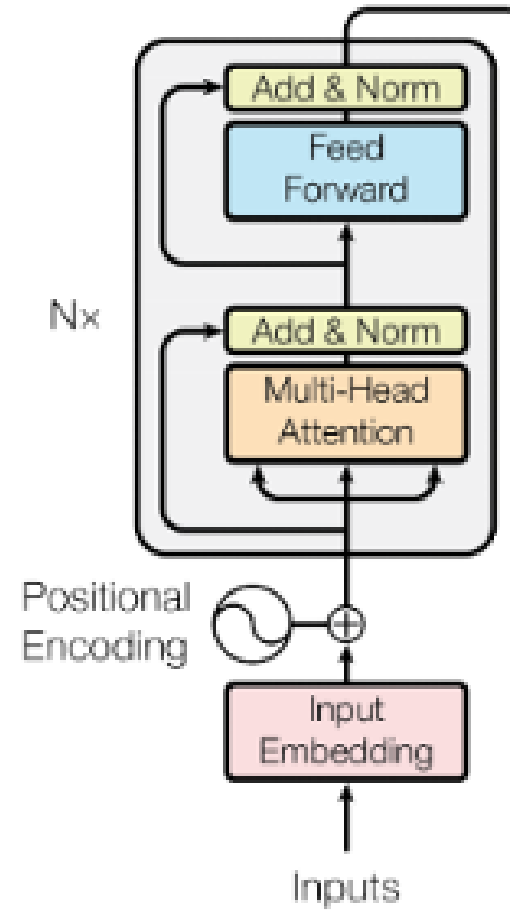
    Returns:
        Tensor: FFN과 잔차 연결을 거친 후의 출력 (B, S, D).
    """
    print(f'\t\t[ FFN ]')

    residual = x
    out = self.fc_out1(x)
    out = self.relu(out)
    out = self.fc_out2(out)

    out = out + residual
    out = self.layer_norm(out)

    return out
```

Transformer : Encoder Summary



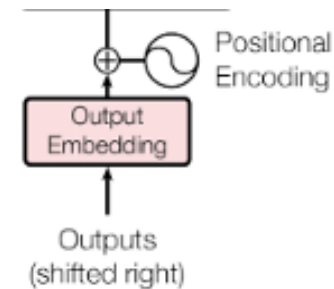
Transformer : decoder

훈련 시

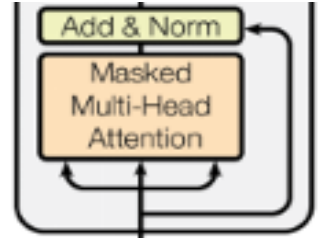
- Encoder Input : ['I', 'want', 'buy', 'a', 'car']
- Decoder Input : [[sos], '나는', '차를', '사고싶다']
Shifted right : SOS token

추론 시

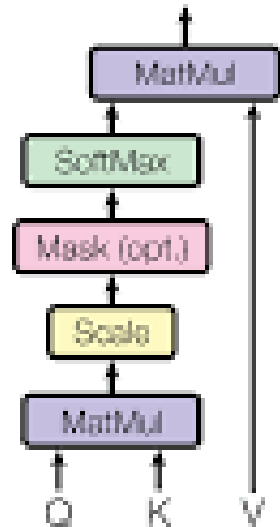
Decoder Input : [[sos], '이전의 정답']



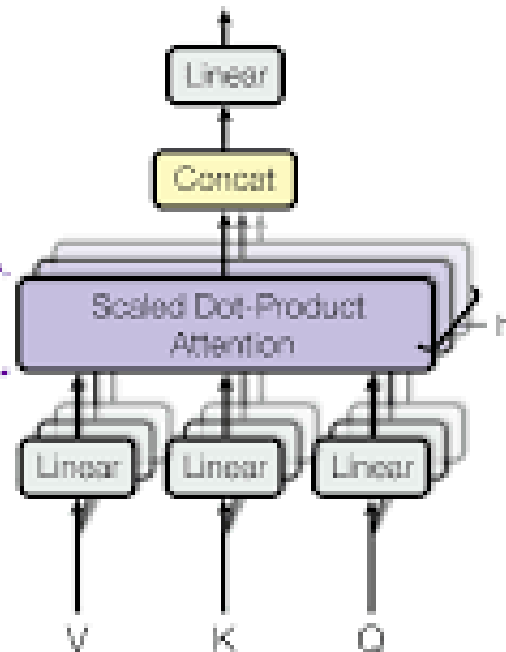
Transformer : Masked Multi-Head Attention



Scaled Dot-Product Attention



Multi-Head Attention



Masked Attention Scores		
$q_1 k_1^T$	$-\infty$	$-\infty$
$q_2 k_1^T$	$q_2 k_2^T$	$-\infty$
$q_3 k_1^T$	$q_3 k_2^T$	$q_3 k_3^T$

Auto Regressive는 이전 값들을 기반으로 다음 값을 예측하며, 순차적인 데이터 모델링에 적합한 접근법입니다.

Transformer : Masked Multi-Head Attention

```
class MaskedMultiHeadAttention(nn.Module):
    """Masked MultiHead Attention 모듈.

    Self-Attention 에서 미래의 토큰을 마스킹하여 Auto Regressive를 유지합니다.
    """
    def __init__(self, in_channels, num_head):
        """MaskedMultiHeadAttention 모듈을 초기화합니다.

        Args:
            in_channels (int): 입력의 임베딩 차원.
            num_head (int): 사용할 헤드의 개수.
        """
        super().__init__()
        self.in_channels = in_channels
        self.num_head = num_head

        assert self.in_channels % self.num_head == 0, 'in_channels는 num_head로 나누어 떨어져야 합니다.'
        self.d_model = self.in_channels // self.num_head

        self.WQ = nn.Linear(self.in_channels, self.in_channels)
        self.WK = nn.Linear(self.in_channels, self.in_channels)
        self.WV = nn.Linear(self.in_channels, self.in_channels)

        self.WO = nn.Linear(self.in_channels, self.in_channels)

        self.layer_norm = nn.LayerNorm(self.in_channels)
```

```
def forward(self, x):
    """Masked Multi-Head Attention을 적용합니다.

    Args:
        x (Tensor): 입력 텐서 (B, S, D).

    Returns:
        Tensor: Masked Multi-Head Attention과 잔차 연결을 거친 출력 (B, S, D).
    """
    print(f'\t\t[ Masked MultiHead Attention ]')

    B, S, D = x.size()
    residual = x

    Q = self.WQ(x)
    K = self.WK(x)
    V = self.WV(x)

    Q = Q.view(B, S, self.num_head, self.d_model).transpose(1,2)
    K = K.view(B, S, self.num_head, self.d_model).transpose(1,2)
    V = V.view(B, S, self.num_head, self.d_model).transpose(1,2)

    attention_score = torch.matmul(Q, K.transpose(-1,-2)) / np.sqrt(self.d_model)

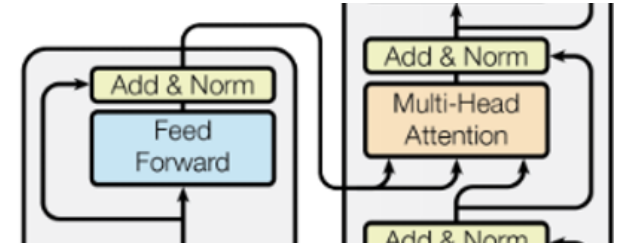
    # 마스킹을 적용하여 미래의 토큰을 무시
    masked = torch.tril(torch.ones(S, S)).unsqueeze(0).unsqueeze(0).to(x.device) # (1, 1, S, S)
    attention_score = attention_score.masked_fill(masked == 0, float('-inf'))
    attention_score = F.softmax(attention_score, dim=-1)

    attention_value = torch.matmul(attention_score, V)

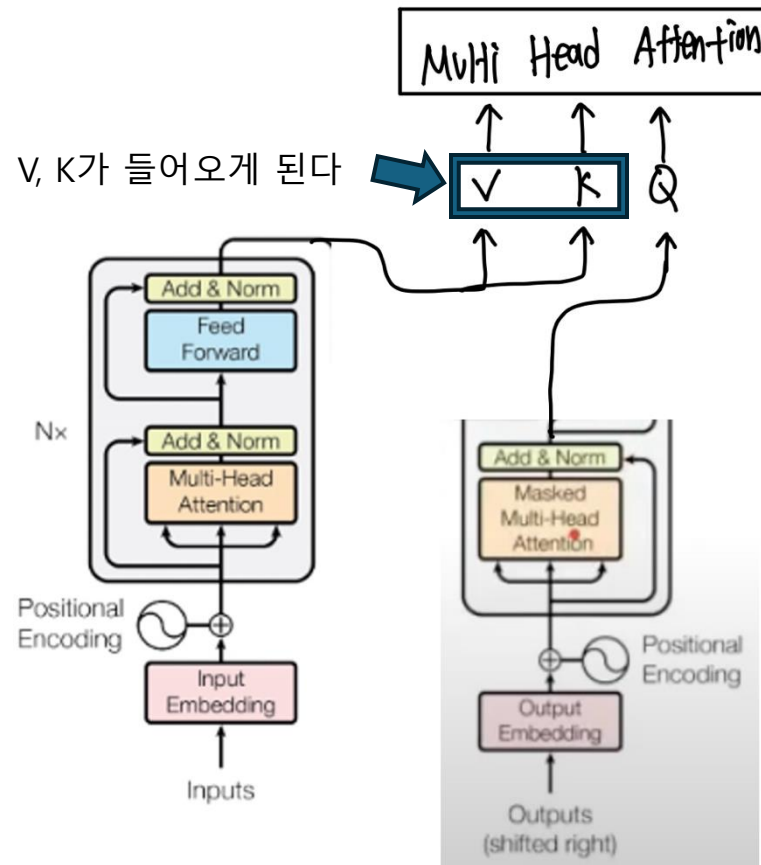
    attention_result = attention_value.transpose(1,2).contiguous().view(B, S, D)
    attention_result = self.WO(attention_result)

    result = attention_result + residual
    result = self.layer_norm(result)
    return result
```

Transformer : Encoder Decoder MultiHead Attention



원문 문장의 문맥 정보가 담긴 V, K가 들어오게 된다



Transformer : Encoder Decoder MultiHead Attention

```
class EncoderDecoderAttention(nn.Module):
    """Encoder-Decoder Attention 모듈.

    Encoder의 출력과 Decoder의 입력을 기반으로 Multi-Head Attention을 수행합니다.
    """
    def __init__(self, in_channels, num_head):
        """EncoderDecoderAttention 모듈을 초기화합니다.

        Args:
            in_channels (int): 입력의 임베딩 차원.
            num_head (int): 사용할 헤드의 개수.
        """
        super().__init__()
        self.in_channels = in_channels
        self.num_head = num_head

        assert self.in_channels % self.num_head == 0, 'in_channels는 num_head로 나누어 떨어져야 합니다.'
        self.d_model = self.in_channels // self.num_head

        self.WQ = nn.Linear(self.in_channels, self.in_channels)
        self.WK = nn.Linear(self.in_channels, self.in_channels)
        self.WV = nn.Linear(self.in_channels, self.in_channels)

        self.WO = nn.Linear(self.in_channels, self.in_channels)

        self.layer_norm = nn.LayerNorm(self.in_channels)
```

```
def forward(self, encoder_output, decoder_input):
    """Encoder와 Decoder의 출력을 기반으로 Multi-Head Attention을 적용합니다.

    Args:
        encoder_output (Tensor): Encoder의 출력 (B, S, D).
        decoder_input (Tensor): Decoder의 입력 (B, S, D).

    Returns:
        Tensor: Encoder-Decoder Attention과 잔차 연결을 거친 출력 (B, S, D).
    """
    print(f'\t\t[ Encoder-Decoder MultiHead Attention ]')

    B, S, D = decoder_input.size()
    residual = decoder_input

    Q = self.WQ(decoder_input)
    K = self.WK(encoder_output)
    V = self.WV(encoder_output)

    Q = Q.view(B, S, self.num_head, self.d_model).transpose(1, 2)
    K = K.view(B, S, self.num_head, self.d_model).transpose(1, 2)
    V = V.view(B, S, self.num_head, self.d_model).transpose(1, 2)

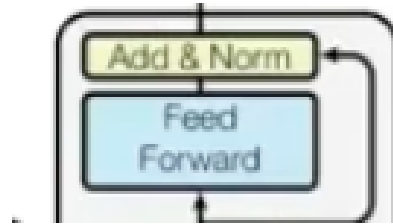
    attention_score = torch.matmul(Q, K.transpose(-1,-2)) / np.sqrt(self.d_model)
    attention_score = F.softmax(attention_score, dim=-1)
    attention_value = torch.matmul(attention_score, V)

    attention_result = attention_value.transpose(1,2).contiguous().view(B, S, D)
    attention_result = self.WO(attention_result)

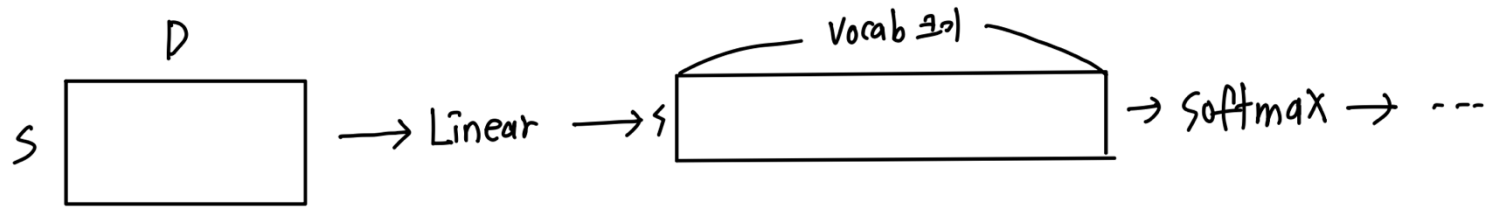
    result = attention_result + residual
    result = self.layer_norm(result)

    return result
```

Transformer : Position-wise Feed-Forward Networks



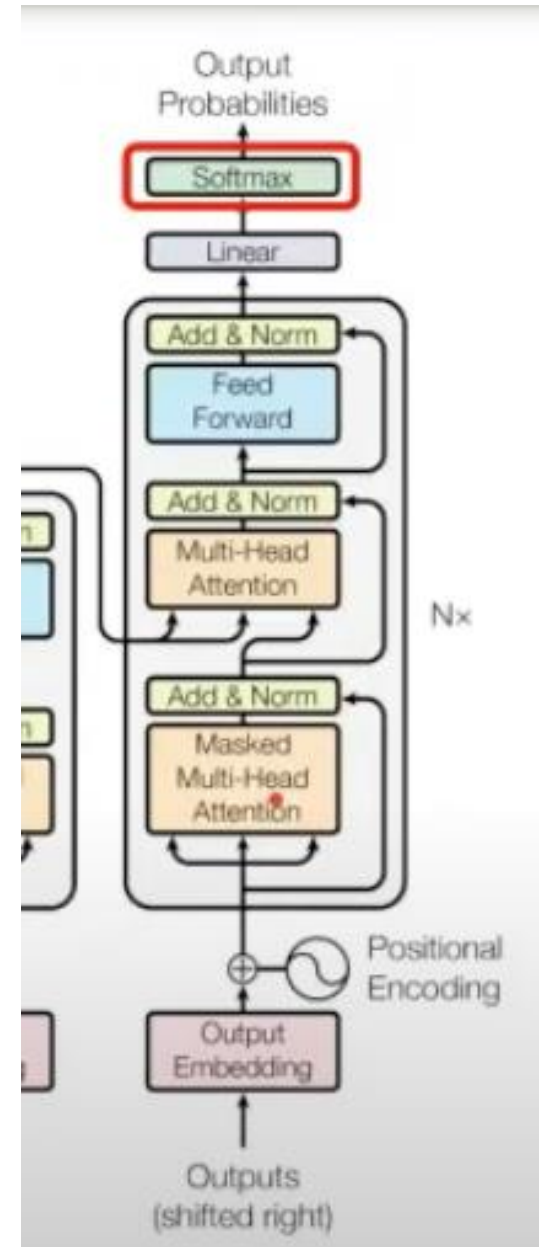
Transformer : output



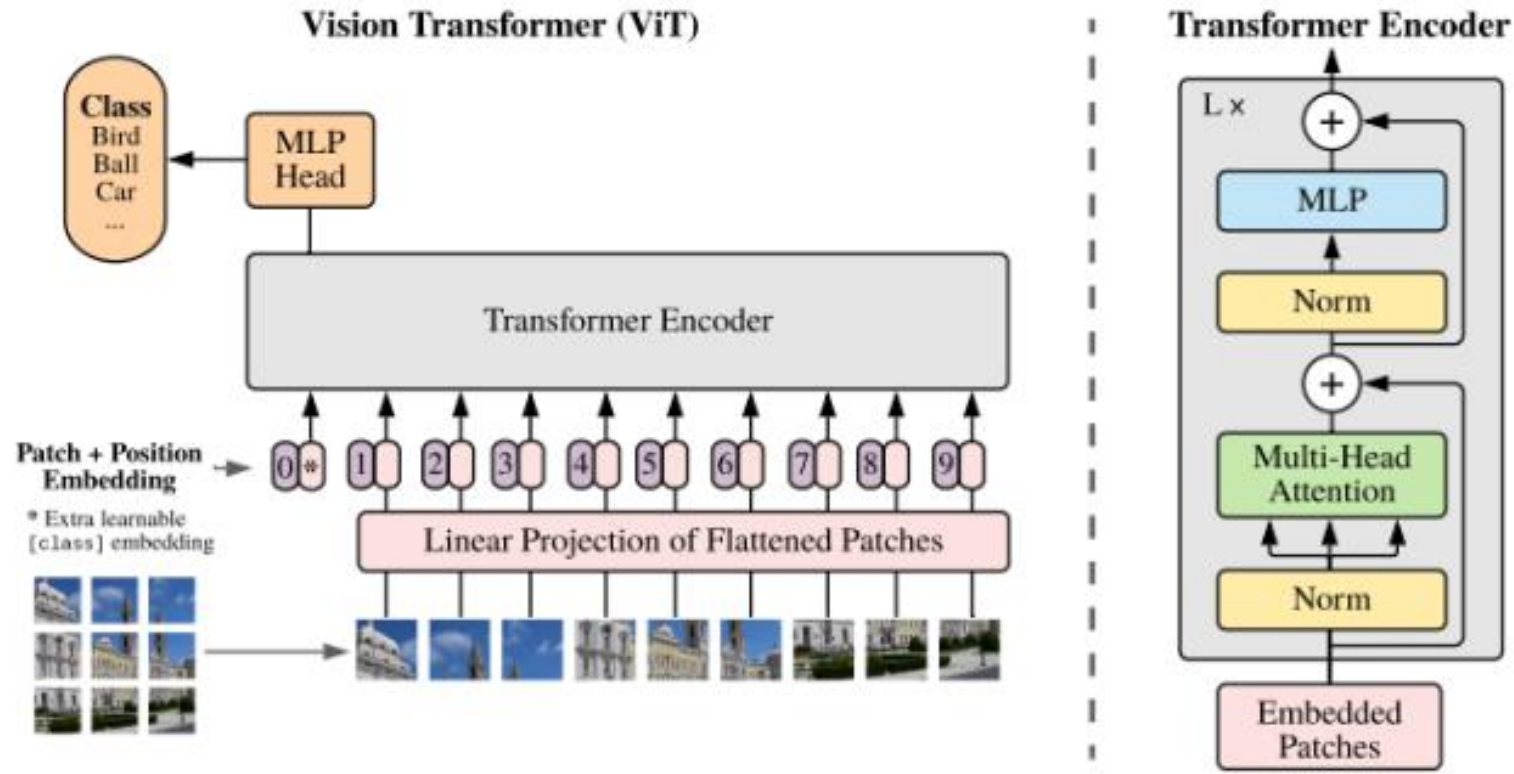
Result example.

[[0.2, 0.1, 0.7],
 [0.4, 0.9, 0.1],
 [0.9, 0.05, 0.05]]

1	"Je"	0.8	$-\log(0.8) \approx 0.223$
2	"suis"	0.6	$-\log(0.6) \approx 0.511$
3	"un"	0.7	$-\log(0.7) \approx 0.357$
4	"étudiant"	0.5	$-\log(0.5) \approx 0.693$
5	" "	0.9	$-\log(0.9) \approx 0.105$
Total Loss			$(0.223 + 0.511 + 0.357 + 0.693 + 0.105) / 5 \approx 0.378$



ViT : Vision Transformer

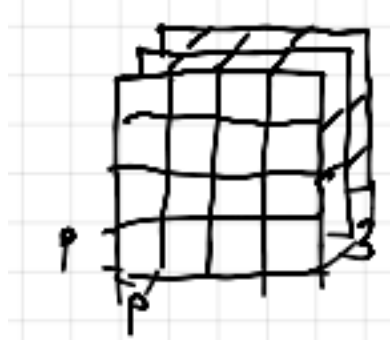


ViT : Input Image

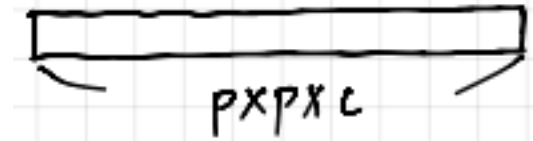
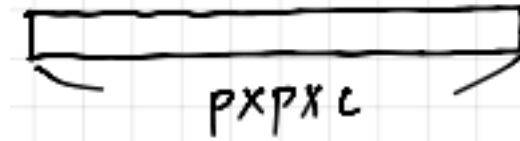
입력 이미지



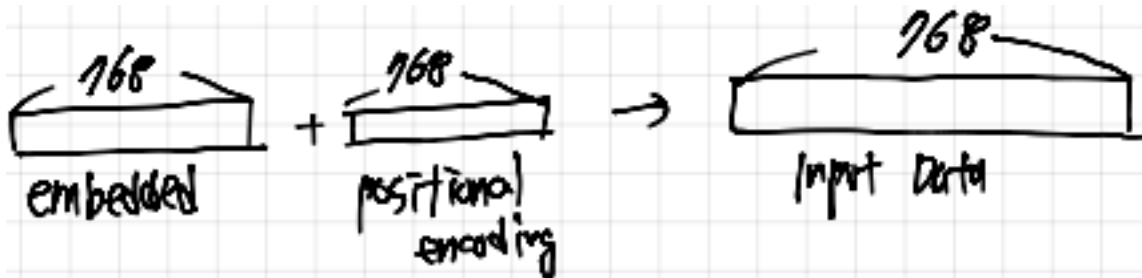
패치 단위로 분리



1D 로 임베딩 + positional encoding



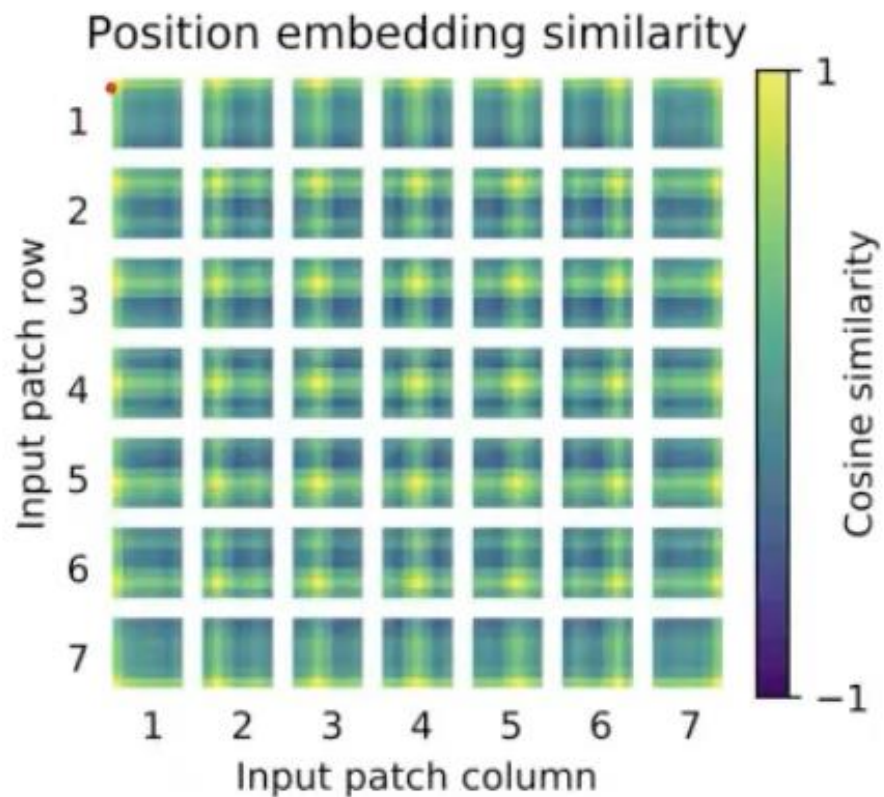
If) image_size = 224, P = 16, C = 3 인 경우



CLS token을 추가하여 1D 임베딩



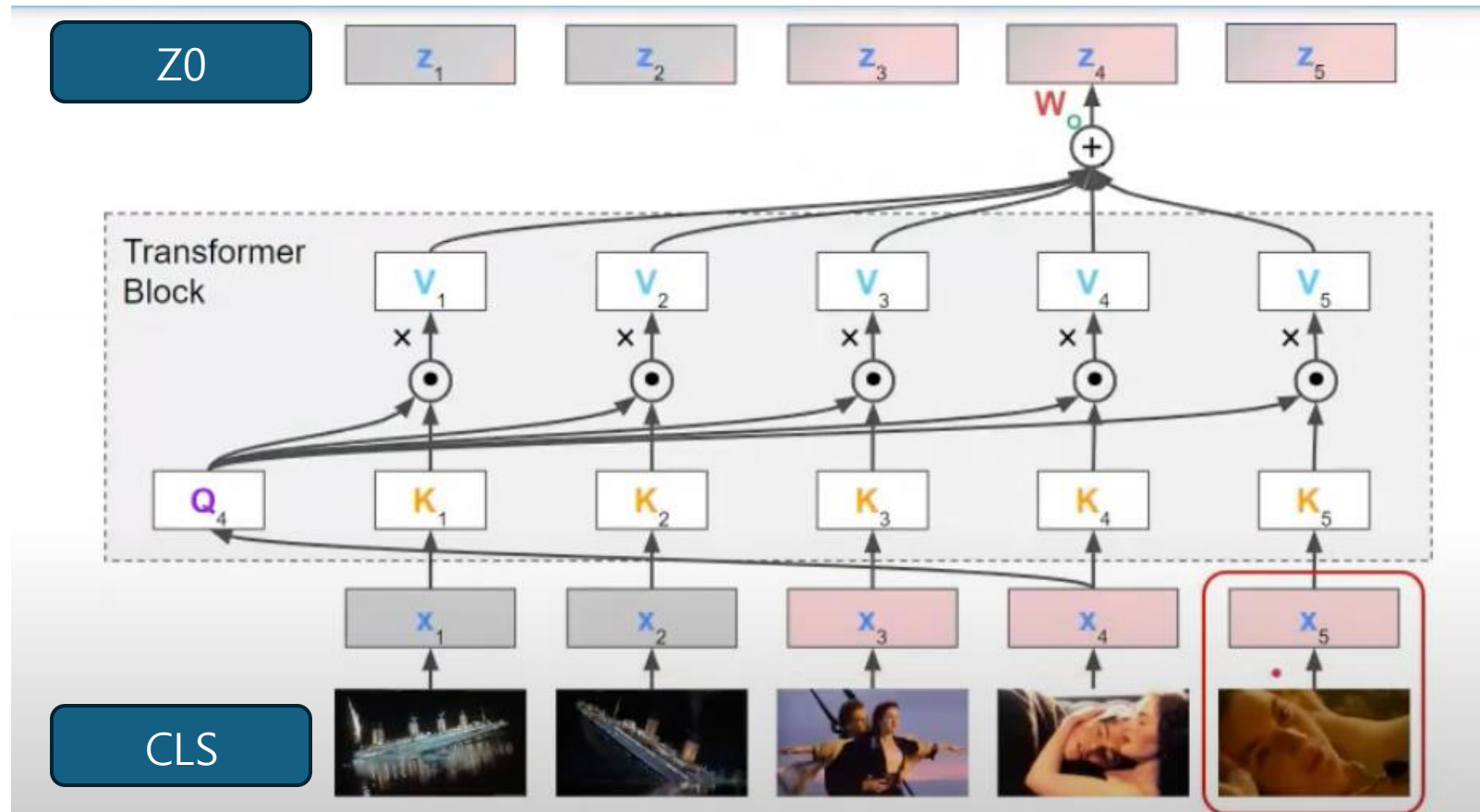
ViT : Positional Encoding



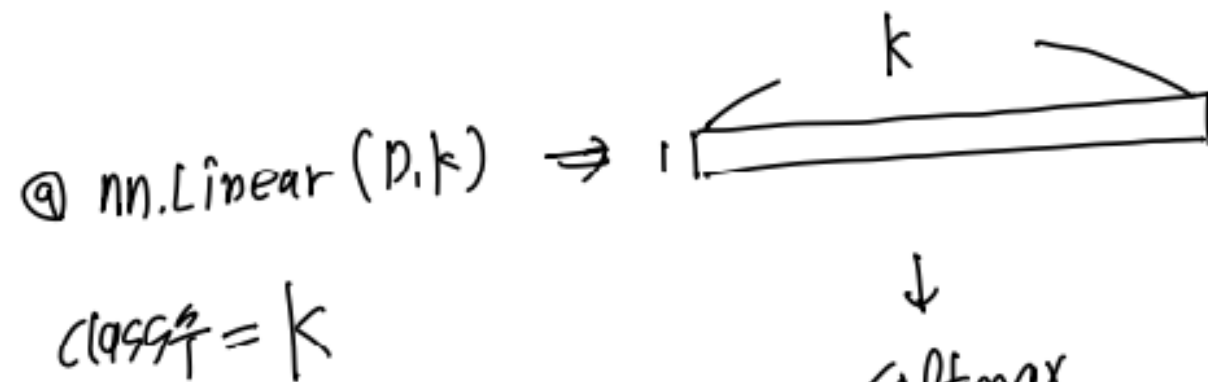
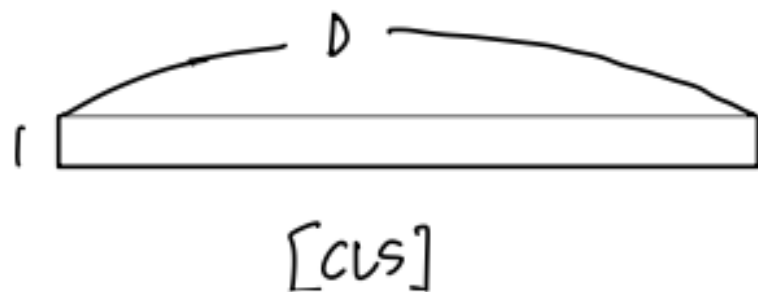
Transformer와 다르게 학습 가능한 Vector를 사용합니다.
단순히 위치 정보는 위, 아래, 가운데와 같은 정보가 일정하다.
문장처럼 감탄사가 붙으면 Left Shifted되고 이런 개념이 없다.

그래서 positional encoding끼리의 similarity를 구하게 되면 row, col이 동일한 경우 가장 강하게 나타나고 row, col이 같은 곳에서 다음으로 강하게 나타나는것을 통해서, image data에서는 positional Encoding을 그렇게 큰 고려사항에 포함되지 않는다고 합니다.

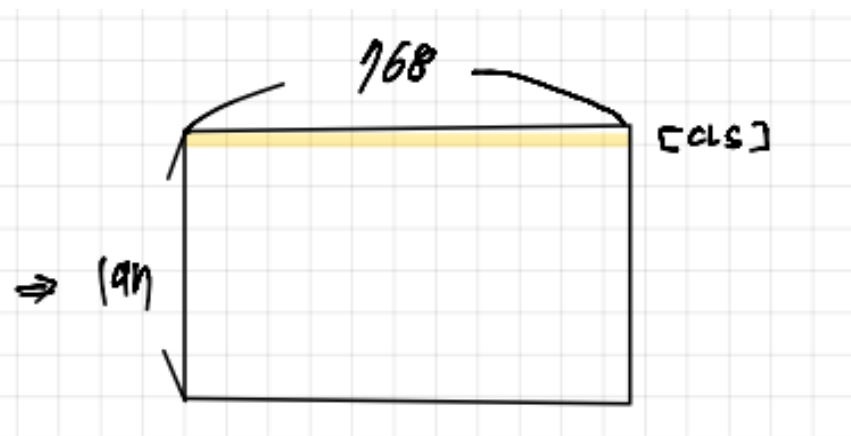
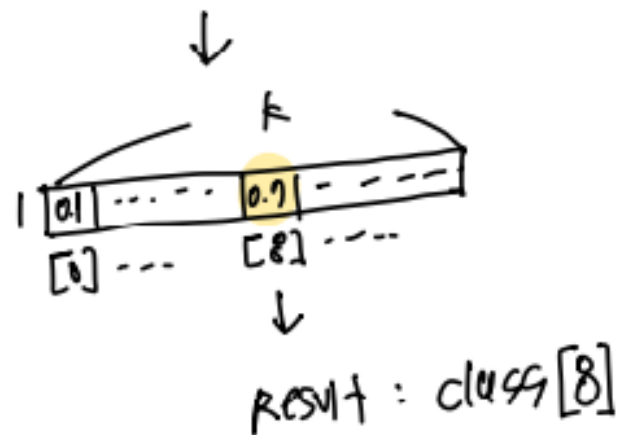
ViT : CLS Token



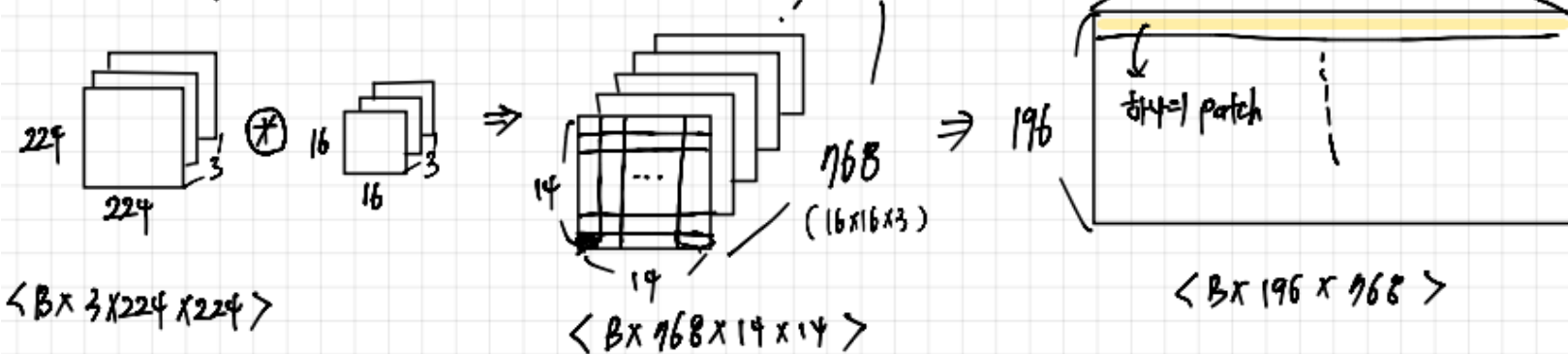
ViT : Output



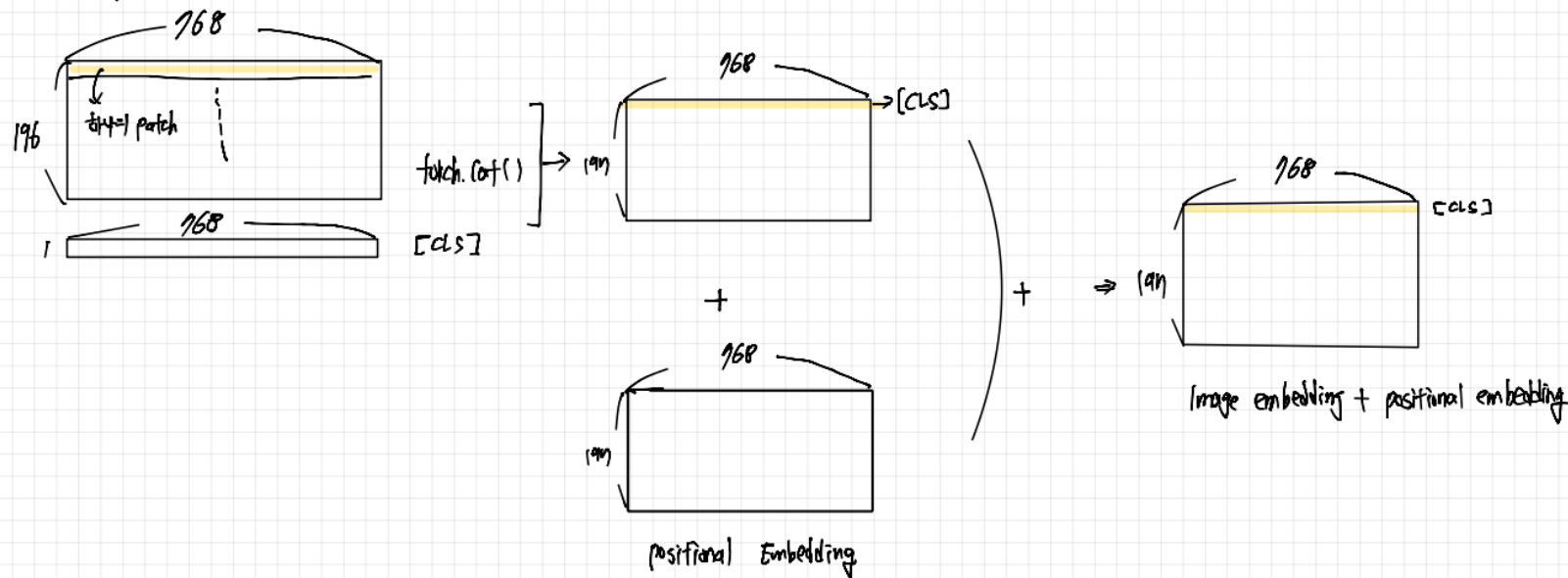
↓
softmax

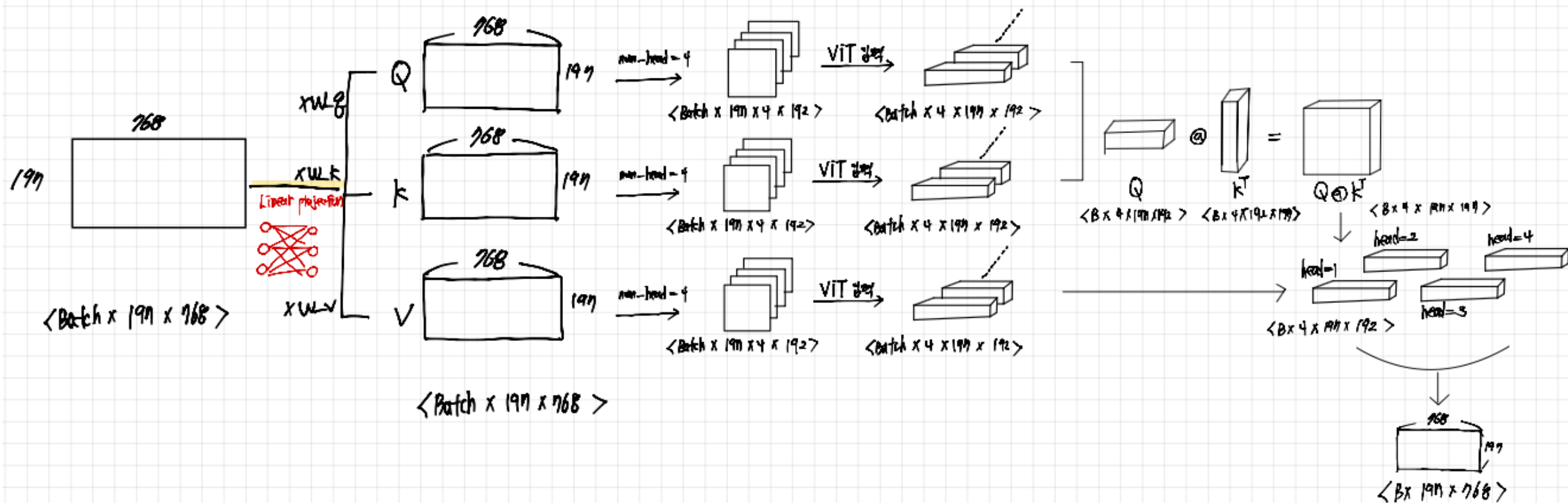


Step 1) Image를 patch로 쪼갬. Conv를 사용해서 변환함.



Step 2) patched된 Image의 [CLS] token 은 추가 생성 & positional embedding을 더함.





왜 ViT 성능이 잘 나올까?

Inductive bias. We note that Vision Transformer has much less image-specific inductive bias than CNNs. In CNNs, locality, two-dimensional neighborhood structure, and translation equivariance are baked into each layer throughout the whole model. In ViT, only MLP layers are local and translationally equivariant, while the self-attention layers are global. The two-dimensional neighborhood structure is used very sparingly: in the beginning of the model by cutting the image into patches and at fine-tuning time for adjusting the position embeddings for images of different resolution (as described below). Other than that, the position embeddings at initialization time carry no information about the 2D positions of the patches and all spatial relations between the patches have to be learned from scratch.

Inductive Bias : 모델이 학습할 때 내재적으로 가정하는 규칙이나 가정을 의미합니다.

CNN : conv filter를 통해 이미지의 지역적, 2D 공간적 관계를 강하게 반영
ViT : 지역적 정보 보다는 관계를 데이터 학습에 의존.

THE END