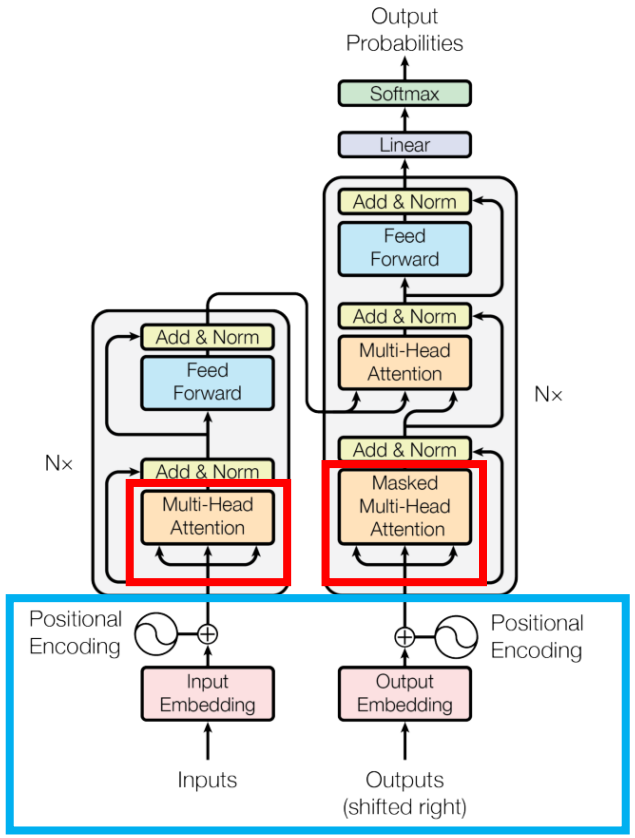
Transformer

Transformer Intro

Transformer

Transformer Architecture



입력 방법 : Positional Encoding

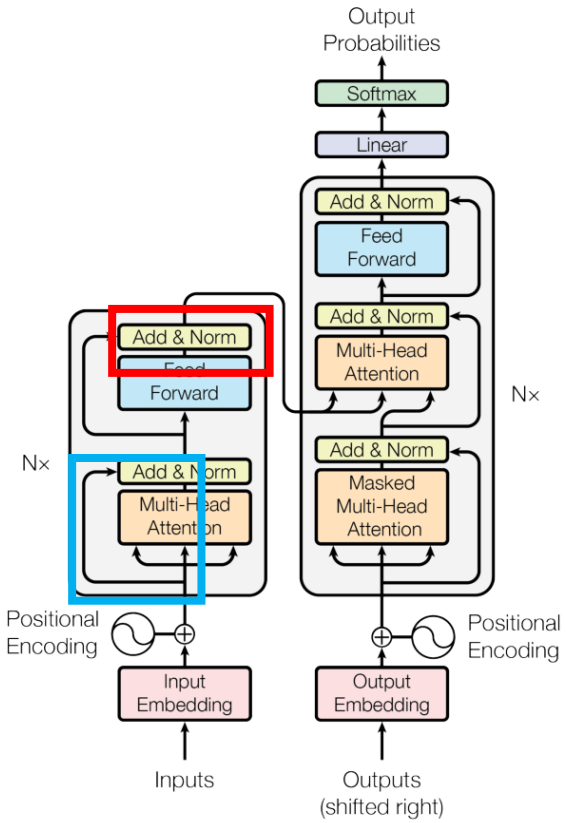
기존의 RNN에서는 가능 했던 순서처리가 안되는 Transformer

핵심 모듈 : (Masked)Multi head attention

Self-attention 메커니즘을 이용한 자연어 처리 향상 모듈

Transformer

Transformer Architecture



성능 향상 1 : Skip Connection

성능 향상 2 : Layer Normalization

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**	$2.3 \cdot 10^{19}$	

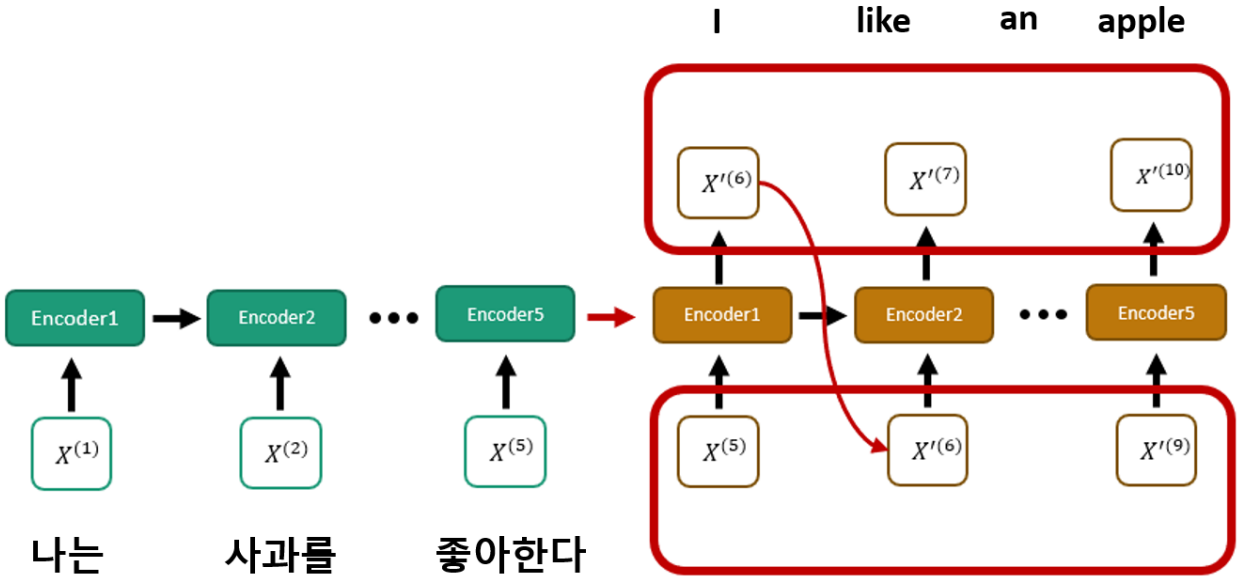
Transformer

- 기계번역 task에서 기존의 연구들 보다 성능적으로 우수
- 병렬적으로 처리가 가능한 모델 -> time complexity 감소
- 이후에 사용되는 Bert, GPT 모델에서 일반화에 강점이 있다는 것이 확인
- 이번 클립에서는 Transformer의 모듈별로 자세한 메커니즘을 공부

Positional Encoding

Positional Encoding

어째서 필요한 것일까?

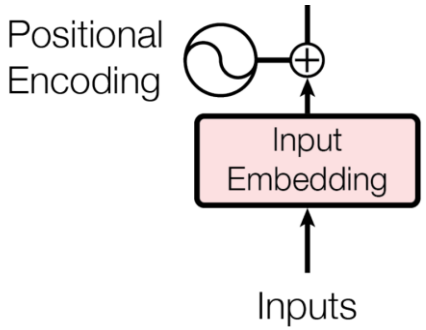


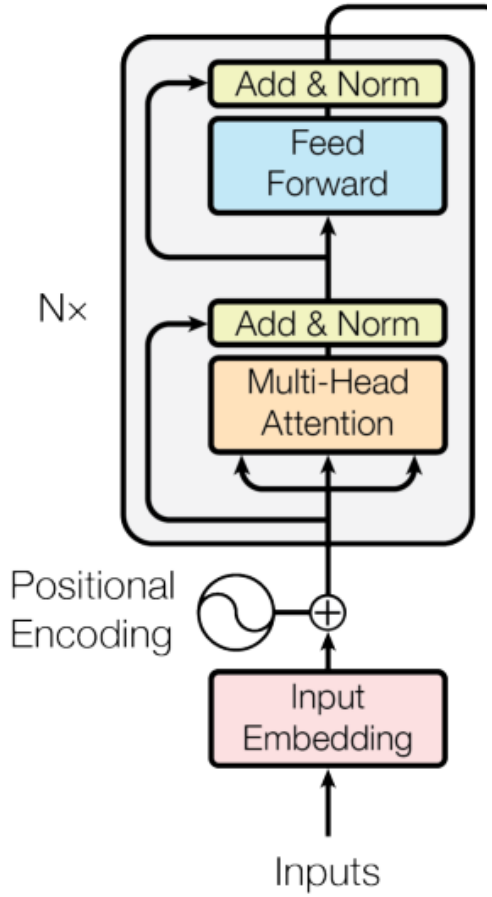
- 기존의 RNN 기반의 방법의 경우, " → " 와 같은 context 벡터를 추출을 해야함
- 그러나 이러한 추출을 하기 위해서 문장의 단어들을 순차적으로 처리해야만 했음
- 디코더를 위해서도 마찬가지로 순차적으로 처리됨
- 이러한 순차적인 방법은 문장의 순서를 고려하게 됨

Positional Encoding

어째서 필요한 것일까?

$$QW_i^Q,\ W_i^Q \in \mathbb{R}^{d_{model} \times d_k},$$



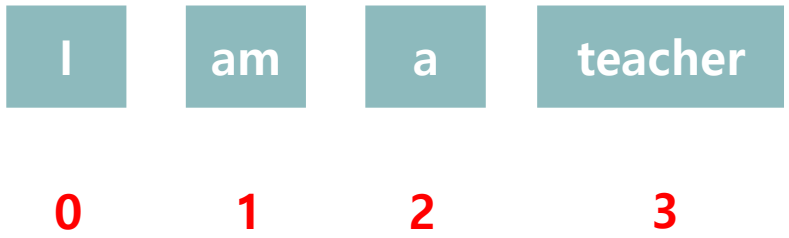


- **반면 Transformer 의 입력 Q 의 경우 행렬 연산을 통해 입력 벡터로 변환되어 Multi-head attention 모듈에 들어감**
- **문장의 순서에 대한 정보를 넣어줄 필요성이 생김**

Positional Encoding

어떻게 줄까?

- **단어 순서대로 숫자를 카운팅?**



- **숫자가 너무 빨리 커져서 Weight 학습 시 어려움이 있음 (CNN에서의 initialization method 를 생각)**

Positional Encoding

어떻게 줄까?

- **단어 순서대로 숫자를 카운팅 후 정규화?**

- **0 부터 1사이 이므로 학습 weight 는 안정적**
- **그러나 두번째 단어는 같은 값을 할당해야 하는데 단어의 길이가 다를 경우 값이 변함**

Positional Encoding

어떻게 줄까?

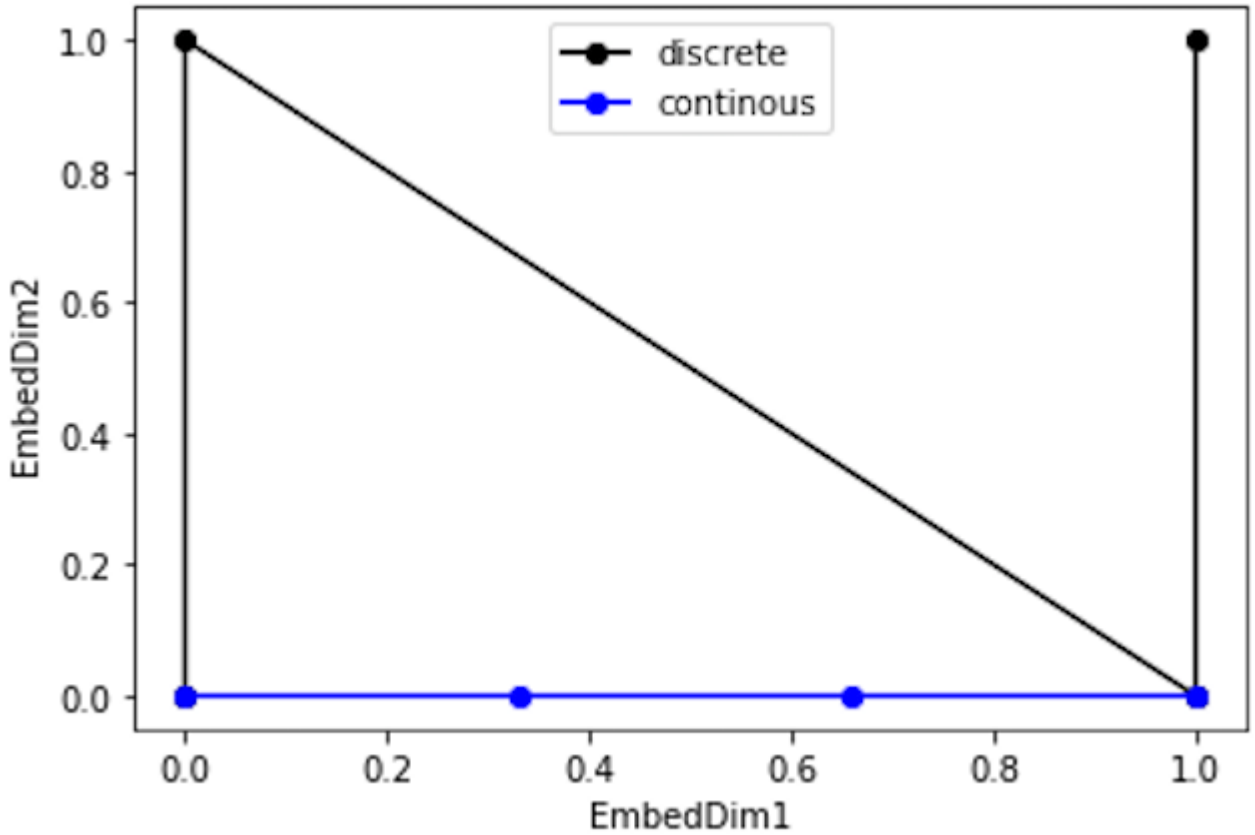
- **단어 순서대로 벡터로 표현 할 경우?**

I	am	a	teacher
0	0	0	0
0	0	1	1
0	1	0	1

- **가변적인 길이에 상관 없이 같은 벡터를 할당 가능함**
- **그런데 단어 순서끼리의 거리가 다르게 된다.**

Positional Encoding

어떻게 줄까?



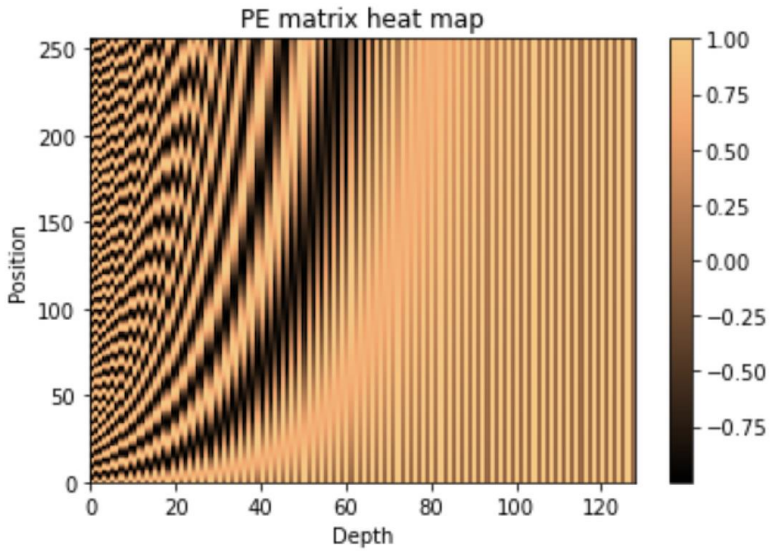
$$(0,0) \rightarrow (0,1) : 1$$
$$(0,1) \rightarrow (1,0) : \sqrt{2}$$
$$(1,0) \rightarrow (1,1) : 1$$

Positional Encoding

어떻게 줄까?

- 연속함수이며 주기함수인 sin 과 soc를 이용하자

I | am | a | teacher



Sinusoidal Encoding

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

$$\omega_k = \frac{1}{10000^{2k/d}}$$

- **벡터의 차원에 따라서 진동수가 줄어 든다.**

Positional Encoding

방법

Sinusoidal Encoding

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

$$\omega_k = \frac{1}{10000^{2k/d}}$$

$$\overrightarrow{p_t} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_{d \times 1}$$

Positional Encoding

어떻게 줄까?

- **이러한 복잡한 방법이 상대적인 거리와 무슨 관계가 있을까?**

- 상대적 positioning

Sinusoidal positional encoding의 또다른 특징은 모델이 별다른 노력 없이 상대적 position을 갖게 된다는 것이다.

$$M.\begin{bmatrix}\sin(\omega_k.t)\\ \cos(\omega_k.t)\end{bmatrix}=\begin{bmatrix}\sin(\omega_k.(t+\phi))\\ \cos(\omega_k.(t+\phi))\end{bmatrix}$$

[1]

문장에서 위치 t와 t+ϕ 사이의 관계는 어떨까. 2 x 2 행렬인 M을 구하면 서로 다른 위치 사이의 관계를 구할 수 있을 것이다.

$$\begin{bmatrix}a_1 & a_2\\ a_3 & a_4\end{bmatrix}\cdot\begin{bmatrix}\sin(w_k\cdot t)\\ \cos(w_k\cdot t)\end{bmatrix}=\begin{bmatrix}a_1\sin(w_k\cdot t)+a_2\cos(w_k\cdot t)\\ a_3\sin(w_k\cdot t)+a_4\cos(w_k\cdot t)\end{bmatrix}=\begin{bmatrix}\sin(w_k\cdot(t+\phi))\\ \cos(w_k\cdot(t+\phi))\end{bmatrix}$$

삼각함수 덧셈 정리에 의해

$$\sin(a+b)=(\sin a)(\cos b)+(\sin b)(\cos a)$$
$$\cos(a+b)=(\cos a)(\cos b)-(\sin a)(\sin b)$$

$$\therefore M=\begin{bmatrix}a_1 & a_2\\ a_3 & a_4\end{bmatrix}=\begin{bmatrix}\cos(w_k\cdot\phi) & \sin(w_k\cdot\phi)\\ -\sin(w_k\cdot\phi) & \cos(w_k\cdot\phi)\end{bmatrix}$$

그런데 여기서 M이 rotation matrix이다. 이러한 선형 변환 행렬이 상대적인 위치 값을 표시하는 데 매우 적합하다. 기준 위치에서 증가하거나 감소함에 따라 대칭적으로 식이 적용되고, 거리가 멀어짐에 따라 그 값이 감소하는 점 역시 적절했다.

https://skyjwoo.tistory.com/entry/positional-encoding%EC%9D%B4%EB%9E%80-%EB%AC%B4%EC%97%87%EC%9D%B8%EA%B0%80

Positional Encoding

어떻게 줄까?

- **변환행렬**

Positional Encoding

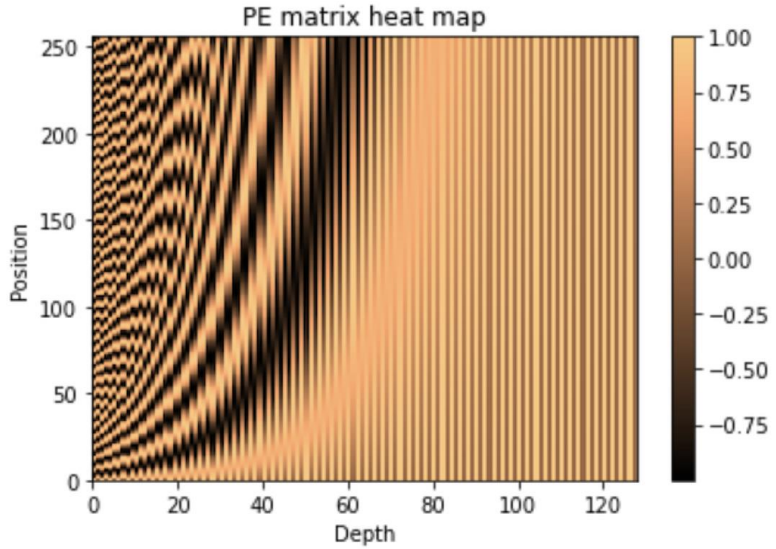
Code

```
#### Numpy version ####
def positional_encoding(max_position, d_model, min_freq=1e-4):
    position = np.arange(max_position)
    freqs = min_freq**(2*(np.arange(d_model)//2)/d_model)
    pos_enc = position.reshape(-1,1)*freqs.reshape(1,-1)
    pos_enc[:, ::2] = np.cos(pos_enc[:, ::2])
    pos_enc[:, 1::2] = np.sin(pos_enc[:, 1::2])
    return pos_enc
```

Positional Encoding

Summary

- 기존의 RNN 방법과는 달리 위치 정보를 얻을 수 없는 Transformer의 input embedding
- 이를 해결하기 위해서 Positional Encoding 을 사용한다.

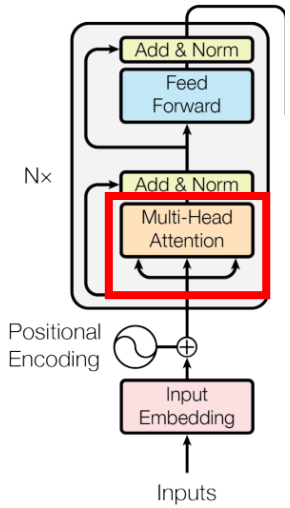


$$\omega_k = \frac{1}{10000^{2k/d}}$$

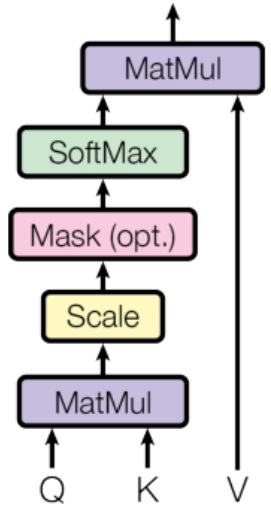
Multi-head Self-attention

Multi-head Self-attention

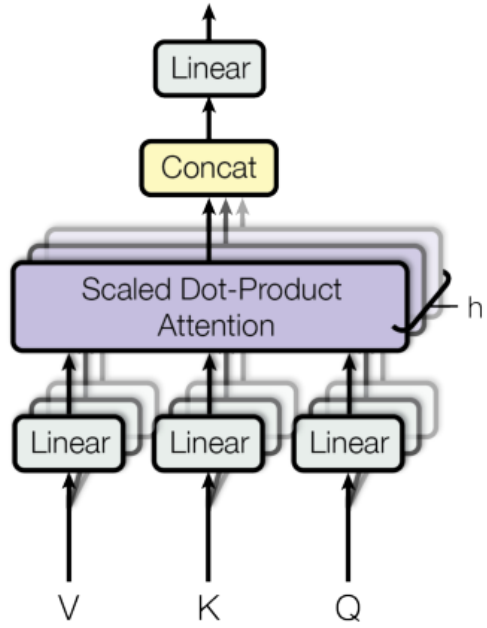
intro



Scaled Dot-Product Attention



Multi-Head Attention



- **Transformer 의 핵심 아이디어인 Multi-head Self attention 모듈**
  - **Key, Query, Value attention 철학을 기반으로**
  - **Scaled Dot-product attention 을 사용함**

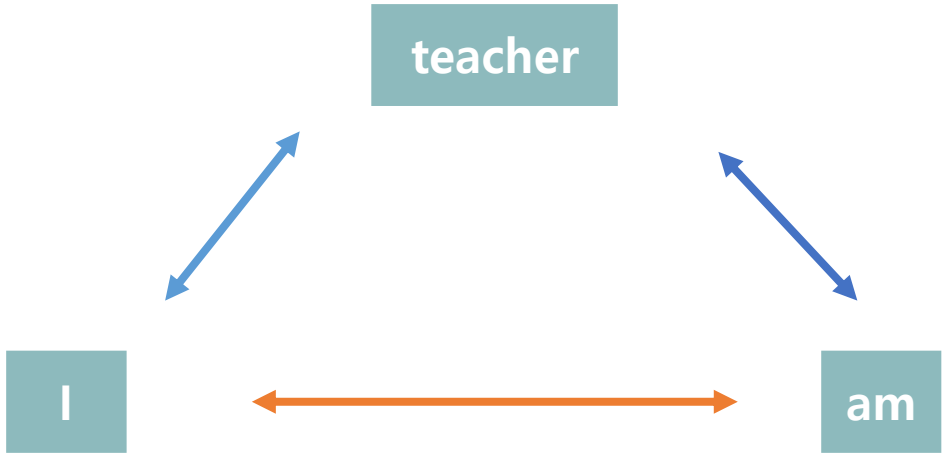
Multi-head Self-attention

intro

1. Teacher 는 am 과 I 와 연관이 있을 것이다.
2. Am과 I 가 연관이 되는 근거는 서로 다르다.

Multi-head Self-attention

intro



a

- 문장을 이해 할 때 각각의 단어는 서로 영향을 끼친다.
- 그 강도는 다르다.
- 연관의 근거는 다 다르다

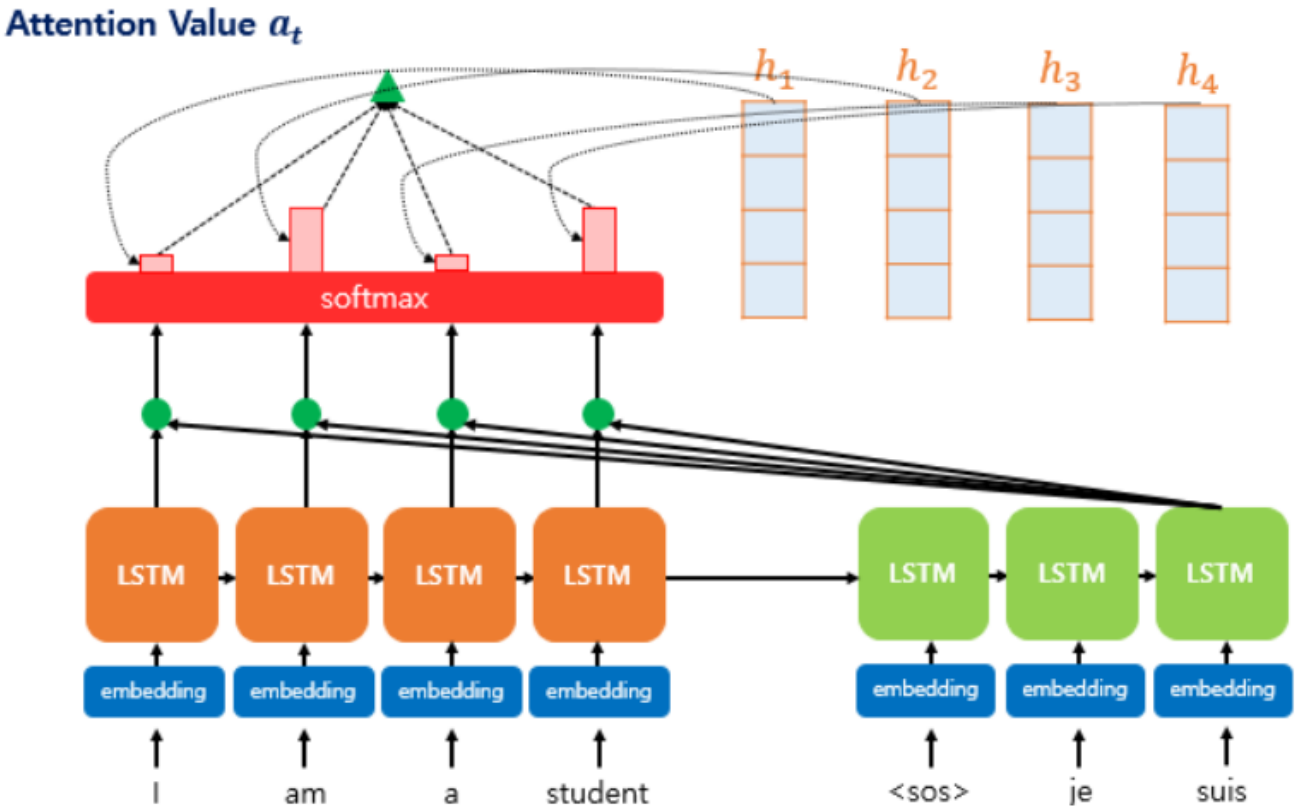
attention

Key value query

multi head

Multi-head Self-attention

attention



$$a_t = \sum_{i=1}^{N} \alpha_i^t h_i$$

Multi-head Self-attention

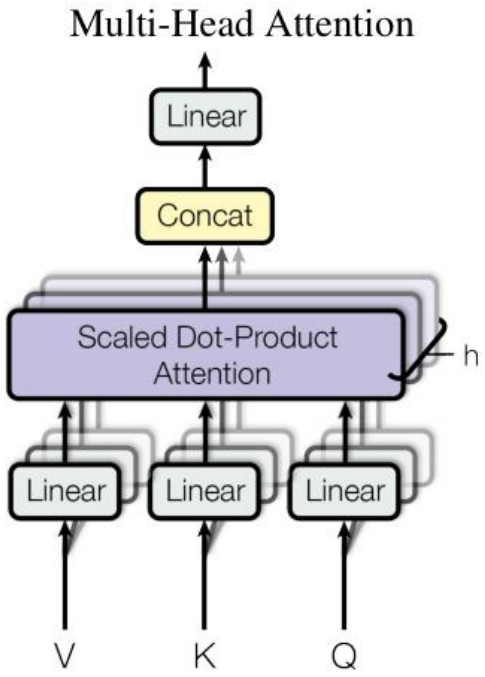
attention

- 기존의 attention 기법의 경우 Key 와 Query 만 존재
- 그러나 Transformer의 attention 의 경우 Value 까지 존재
- 문장을 이해할 때 단어들은 서로 영향을 끼치며 그 강도는 다르다
- 이 단어에 대한 벡터는 (Q) 주어진 단어들에 대해서 유사한 정도 만큼 (K) 고려하고

  각 주어진 단어들은 V 만큼의 중요도를 가진다.

Multi-head Self-attention

attention



$$x'_k \leftarrow \sum_{i=1} similarity(query_k, key_i) value_i$$

$$query_k \leftarrow x_k Q$$
$$value_k \leftarrow x_k V$$
$$key_k \leftarrow x_k K$$

$$similarity(x, y) = x \cdot y$$

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

Multi-head Self-attention

attention

$$x'_k \leftarrow \sum_{i=1} similarity(query_k, key_i) value_i$$

$$query_k \leftarrow x_k Q$$
$$value_k \leftarrow x_k V$$
$$key_k \leftarrow x_k K$$

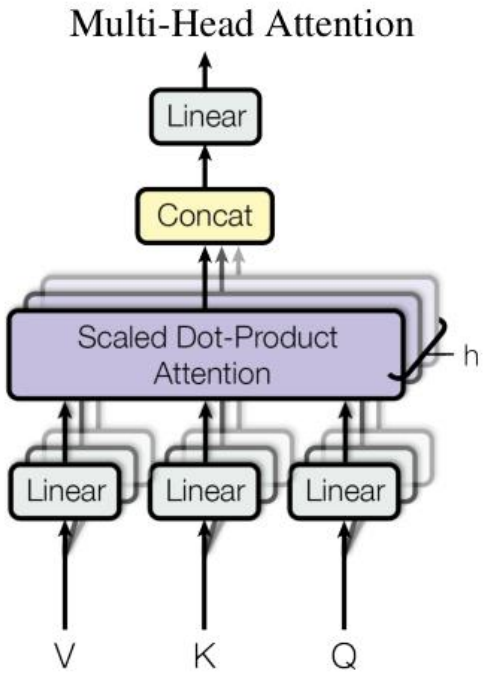
$$similarity(x, y) = x \cdot y$$

I				
love				
you				

https://www.youtube.com/watch?v=AA621UofTUA&ab_channel=%EB%8F%99%EB%B9%88%EB%82%98

Multi-head Self-attention

attention



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

$$x'_k \leftarrow \sum_{i=1} similarity(query_k, key_i) value_i$$

$$query_k \leftarrow x_k Q$$
$$value_k \leftarrow x_k V$$
$$key_k \leftarrow x_k K$$

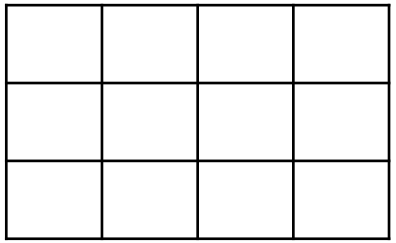
$$similarity(x, y) = x \cdot y$$

https://www.youtube.com/watch?v=AA621UofTUA&ab_channel=%EB%8F%99%EB%B9%88%EB%82%98

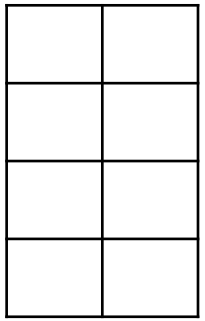
I

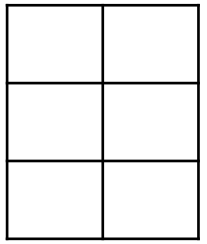
love

you



W_Q(4x2)



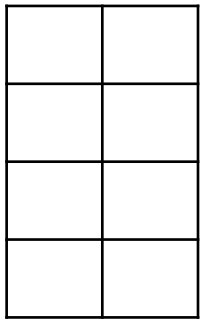


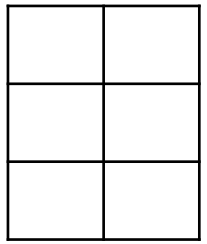
Q_I

Q_{love}

Q_{you}

W_K



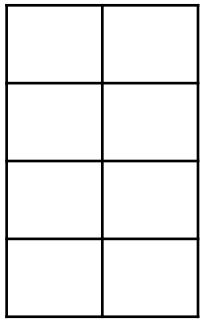


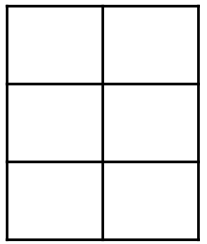
K_I

K_{love}

K_{you}

W_V



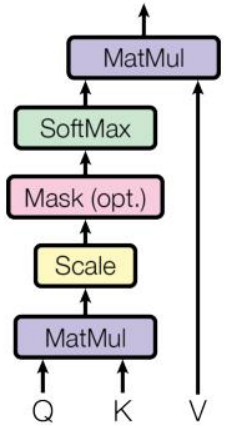


V_I

V_{love}

V_{you}

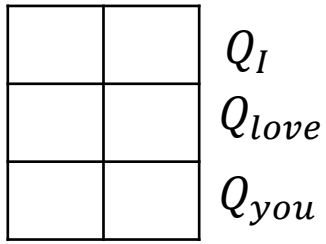
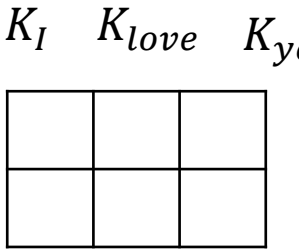
Scaled Dot-Product Attention



$$x'_k \leftarrow \sum_{i=1} similarity(query_k, key_i) value_i$$

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

softmax(Q_I / Q_{love} / Q_{you}) × (K_I K_{love} K_{you}) / $\sqrt{d_k}$)

softmax(  $/\sqrt{d_k}$)

$\boldsymbol{Q_1 \circ K_1}$		
$\boldsymbol{Q_2 \circ K_1}$		

$Q_1 \circ K_1$		
$Q_2 \circ K_1$		

V_I	

- Query 로 단어를 주었을 때
- 이 단어와 유사한 Key 값을 더욱더 attention 을 주고
- 이 key 값의 중요도에 따라서 Value 값을 준다.

X_I^1	

- 최종 결과 값이 Query 의 차원수와 동일하게 된다.
- 계속해서 같은 차원으로 Self-attention 수행이 가능하게 된다.

$$\text{softmax}\left(\begin{bmatrix} x_1Q \\ x_2Q \\ \vdots \\ x_NQ \end{bmatrix} \begin{bmatrix} (x_1K)^\top & (x_2K)^\top & \dots & (x_NK)^\top \end{bmatrix}\right) \begin{bmatrix} x_1V \\ x_2V \\ \vdots \\ x_NV \end{bmatrix}$$

$$= \text{softmax}\left(\begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_N \end{bmatrix} \begin{bmatrix} k_1^\top & k_2^\top & \cdots & k_N^\top \end{bmatrix}\right) \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} q_1k_1^\top & q_1k_2^\top & \cdots & q_1k_N^\top \\ q_2k_1^\top & q_2k_2^\top & \cdots & q_2k_N^\top \\ \vdots & & & \vdots \\ q_Nk_1^\top & q_Nk_2^\top & \cdots & q_Nk_N^\top \end{bmatrix}\right) \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}$$

Multi-head Self-attention

Masked

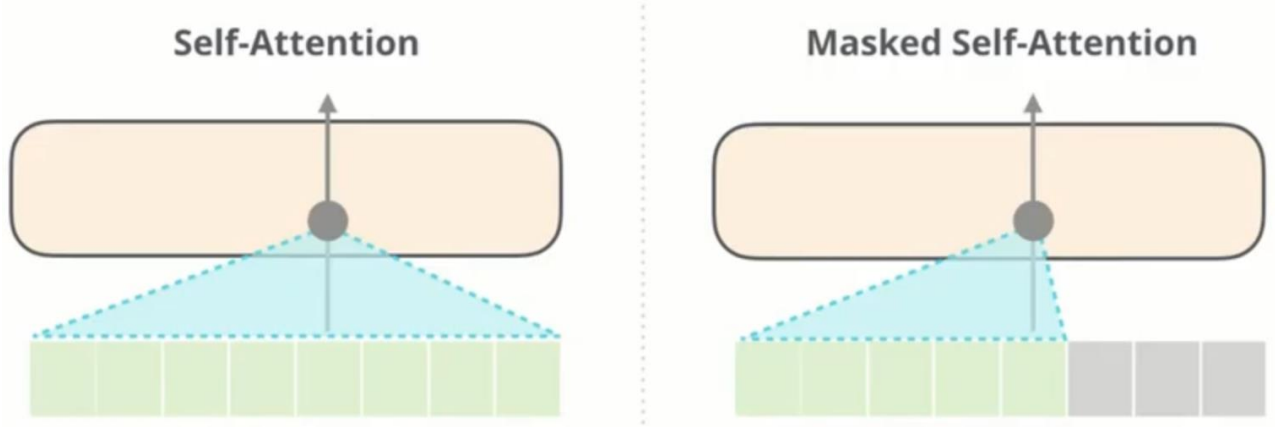
$Q_1 \circ K_1$		
$Q_2 \circ K_1$		

$Q_1 \circ K_1$	0	0
$Q_2 \circ K_1$		0
		0

- **마스크 값으로 0으로 만들어 주게 할 경우?**

Multi-head Self-attention

Masked – 왜 디코더에 masked self attention 일까?



Multi-head Self-attention

Multi-head 연관의 근거는 다 다르다

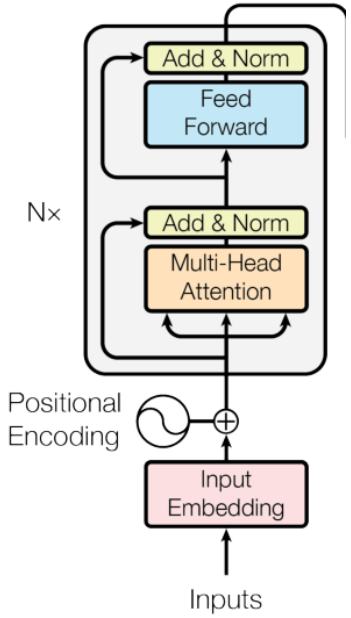
$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_\mathrm{h})W^O$$
$$\text{where } \mathrm{head}_\mathrm{i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- **처음 쿼리의 차원수가 head의 개수 만큼 나눠주기 때문에 최종적으로 입력의 차원수와 같게 된다.**

Multi-head Self-attention

Multi-head 연관의 근거는 다 다르다

- **Head와 차원수 그리고 skip connection**



Multi-head Self-attention

Multi-head 연관의 근거는 다 다르다

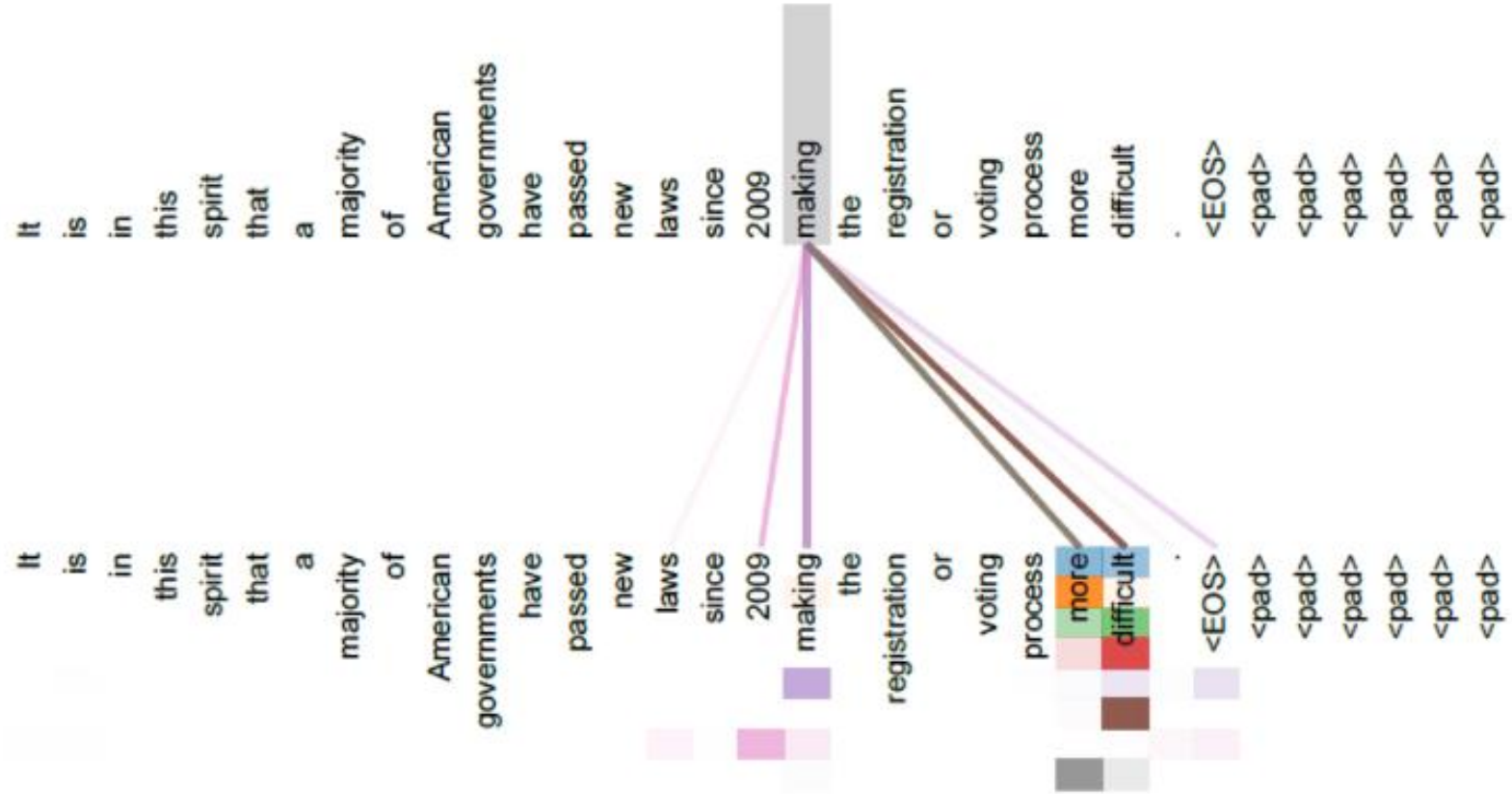


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word 'making'. Different colors represent different heads. Best viewed in color.

https://github.com/ndb796/Deep-Learning-Paper-Review-and-Practice/blob/master/code_practices/Attention_is_All_You_Need_Tutorial_(German_English).ipynb

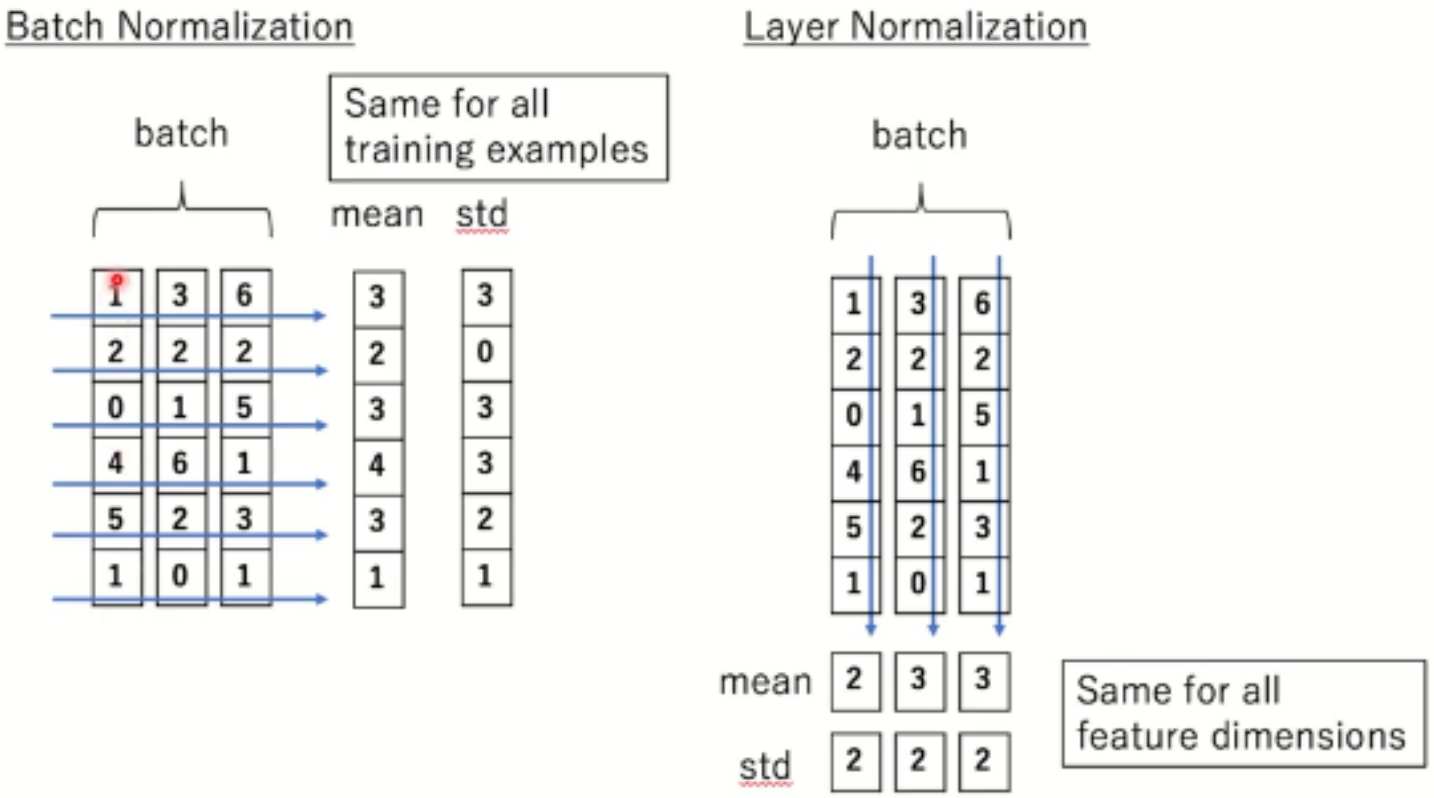
```
class MultiHeadAttentionLayer(nn.Module):
    def __init__(self, hidden_dim, n_heads, dropout_ratio, device):
        super().__init__()

        assert hidden_dim % n_heads == 0

        self.hidden_dim = hidden_dim # 임베딩 차원
        self.n_heads = n_heads # 헤드(head)의 개수: 서로 다른 어텐션(attention) 컨셉의 수
        self.head_dim = hidden_dim // n_heads # 각 헤드(head)에서의 임베딩 차원

        self.fc_q = nn.Linear(hidden_dim, hidden_dim) # Query 값에 적용될 FC 레이어
        self.fc_k = nn.Linear(hidden_dim, hidden_dim) # Key 값에 적용될 FC 레이어
        self.fc_v = nn.Linear(hidden_dim, hidden_dim) # Value 값에 적용될 FC 레이어

        self.fc_o = nn.Linear(hidden_dim, hidden_dim)

        self.dropout = nn.Dropout(dropout_ratio)

        self.scale = torch.sqrt(torch.FloatTensor([self.head_dim])).to(device)
```

```python
def forward(self, query, key, value, mask = None):

    batch_size = query.shape[0]

    # query: [batch_size, query_len, hidden_dim]
    # key: [batch_size, key_len, hidden_dim]
    # value: [batch_size, value_len, hidden_dim]

    Q = self.fc_q(query)
    K = self.fc_k(key)
    V = self.fc_v(value)

    # Q: [batch_size, query_len, hidden_dim]
    # K: [batch_size, key_len, hidden_dim]
    # V: [batch_size, value_len, hidden_dim]

    # hidden_dim → n_heads X head_dim 형태로 변형
    # n_heads(h)개의 서로 다른 어텐션(attention) 컨셉을 학습하도록 유도
    Q = Q.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1, 3)
    K = K.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1, 3)
    V = V.view(batch_size, -1, self.n_heads, self.head_dim).permute(0, 2, 1, 3)

    # Q: [batch_size, n_heads, query_len, head_dim]
    # K: [batch_size, n_heads, key_len, head_dim]
    # V: [batch_size, n_heads, value_len, head_dim]
```

```python
# Attention Energy 계산
energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) / self.scale

# energy: [batch_size, n_heads, query_len, key_len]

# 마스크(mask)를 사용하는 경우
if mask is not None:
    # 마스크(mask) 값이 0인 부분을 -1e10으로 채우기
    energy = energy.masked_fill(mask==0, -1e10)

# 어텐션(attention) 스코어 계산: 각 단어에 대한 확률 값
attention = torch.softmax(energy, dim=-1)

# attention: [batch_size, n_heads, query_len, key_len]

# 여기에서 Scaled Dot-Product Attention을 계산
x = torch.matmul(self.dropout(attention), V)

# x: [batch_size, n_heads, query_len, head_dim]
```

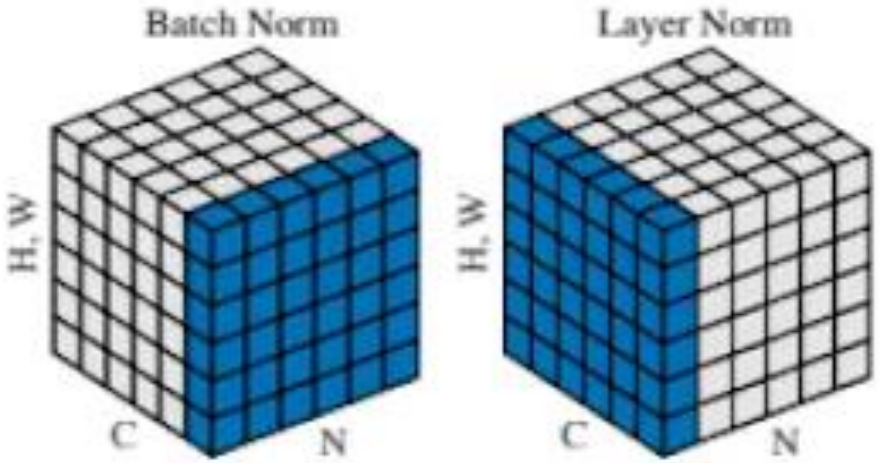
Layer normalization & Final

Layer normalization



https://yonghyuc.wordpress.com/2020/03/04/batch-norm-vs-layer-norm/

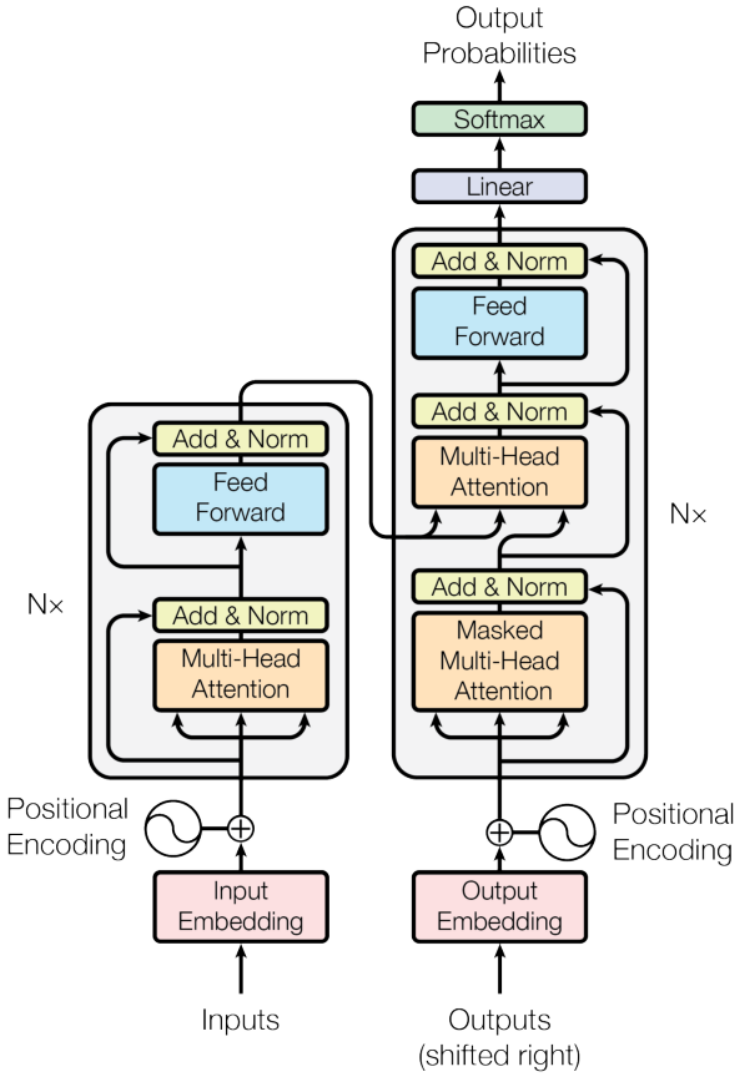
Layer normalization



- Batch norm : sample 들의 feature 별 평균과 분산 -> batch size에 따라서 성능 변화가 심함
-
- Layer norm : 각 batch 에 대해서 feature들의 평균과 분산

```
class LayerNorm(nn.Module):
    def __init__(self, d_model, eps=1e-8):
        super(LayerNorm, self).__init__()
        self.gamma = nn.Parameter(torch.ones(d_model))
        self.beta = nn.Parameter(torch.zeros(d_model))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.gamma * (x - mean) / (std + self.eps) + self.beta
```

Final


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Thank you

나눔스퀘어