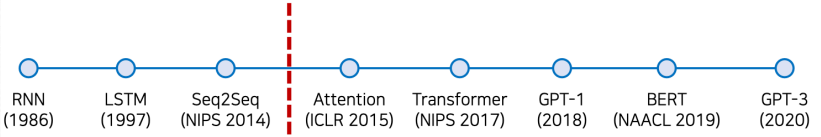
Attention Is All You Need

최민동

목차

01 Introduction

기계번역의 역사



대전환 | RNN, LSTM, GRU → TRANSFORMER

▶ 화제의 ChatGPT
→ 트랜스포머 디코더

▶ 한국어 자연어 처리 대장 KoBERT
→ 트랜스포머 인코더

▶ 이미지처리 SOTA ViT
→ 비전에서 트랜스포머의 활용

https://papers.neurips.cc › paper › 7181-attention-... PDF

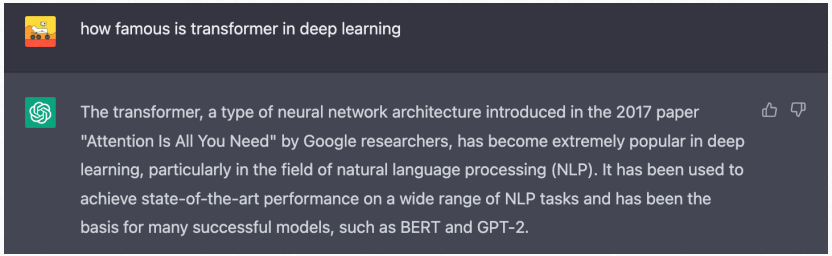
Attention is All you Need - NIPS papers

A Vaswani 저술 · 62165회 인용 — **We** propose a new simple network architecture, the Transformer, based solely on **attention** mechanisms, dispensing with recurrence and convolutions entirely.

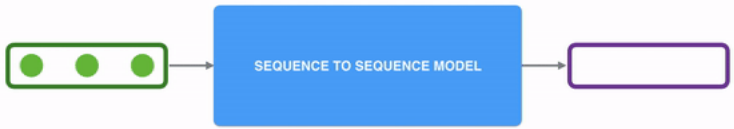
페이지 11개

Attention Is All You Need

▶ 인용수 62165회 → 연간 인용수 10360회 → 일간 인용수 334회



Traditional Seq2Seq



고전적인 Se2Seq 모델

→ 직전에 배운 RNN LSTM GRU 모델

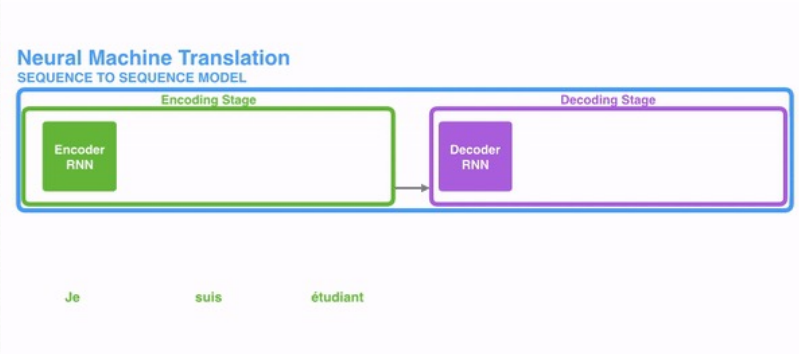
→ Input으로 원본 언어
Output으로 번역 언어
Loss 계산 ~ 업데이트

→ 직전의 Hidden State에만 의존한다면
문장 전체를 번역할 수 있을까?

→ 병렬화된 처리?
: 불가능; 직전의 단어를 알아야 다음 단어 추측

→ LSTM이 긴 문장을 이론처럼 잘 처리할까?

Traditional Seq2Seq



고전적인 Se2Seq 모델

→ 직전에 배운 RNN LSTM GRU 모델

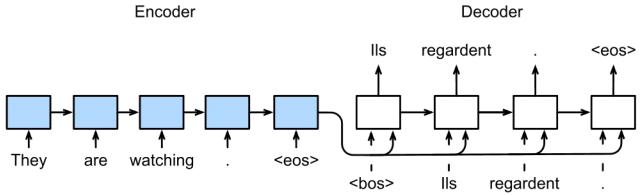
→ Input으로 원본 언어
Output으로 번역 언어
Loss 계산 ~ 업데이트

→ 직전의 Hidden State에만 의존한다면
문장 전체를 번역할 수 있을까?

→ 병렬화된 처리?
: 불가능; 직전의 단어를 알아야 다음 단어 추측

→ LSTM이 긴 문장을 이론처럼 잘 처리할까?

Seq2Seq 문맥 벡터의 사용



Context Vector

→ 압축된 문장의 정보
→ Encoder Output
→ Decoder의 모든 스텝에서 참고

1. Context 생성의 합리성 ?
2. Computation Efficiency?

문맥 벡터의 문제점

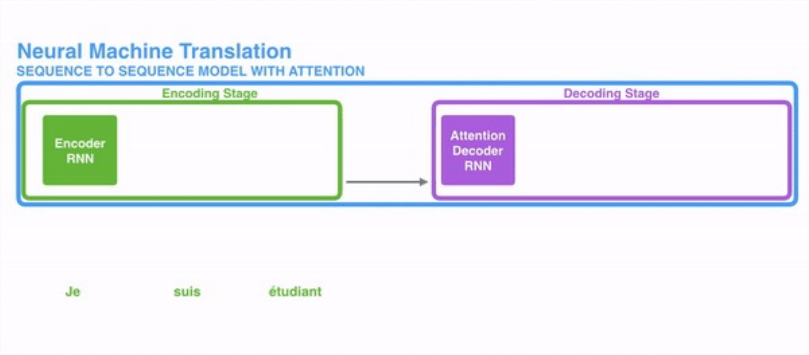
1. 고정된 벡터의 크기

→ 하나의 문맥 벡터가 소스 문장 모든 정보를 가지고 있어야함
압축과정에서 큰 손실

2. 융통성 없음

→ 모든 단어가 동일한 하나의 문맥 벡터를 참고,
번역이라는 Task
→문맥 < 단어간의 관련성

Seq2Seq + Attention



Attention

→ 모든 단어의 Hidden State
Decoder로 전달

https://towardsdatascience.com/transformers-141e32e69591

Seq2Seq + Attention

Attention

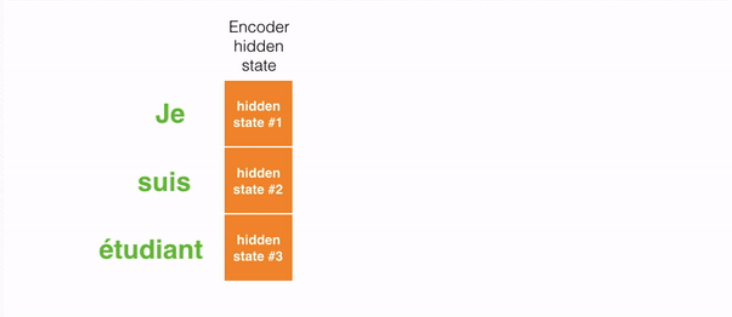
→ 모든 단어의 Hidden State
Decoder로 전달

https://towardsdatascience.com/transformers-141e32e69591

Seq2Seq + Attention

Attention

→ 모든 단어의 Hidden State
Decoder로 전달



https://towardsdatascience.com/transformers-141e32e69591

Seq2Seq + Attention

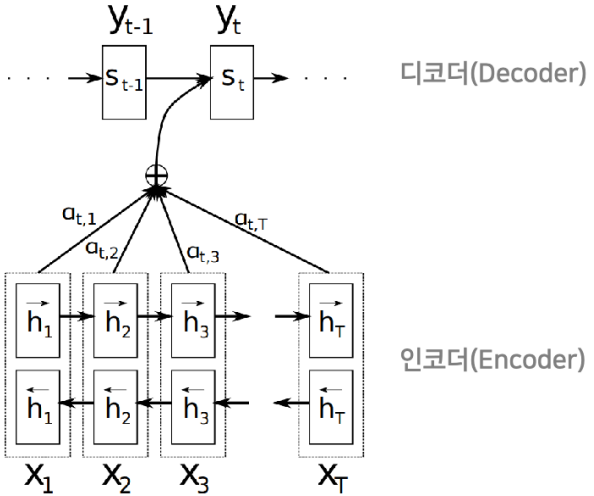
- 디코더는 **매번 인코더의 모든 출력 중에서 어떤 정보가 중요한지를 계산**합니다.
  - i = 현재의 디코더가 처리 중인 인덱스
  - j = 각각의 인코더 출력 인덱스
  - **에너지(Energy)** $e_{ij} = a(s_{i-1}, h_j)$
  - **가중치(Weight)** $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$

Seq2Seq + Attention

- 에너지(Energy) $e_{ij} = a(s_{i-1}, h_j)$
- 가중치(Weight) $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$

Weighted sum 이용

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$



Transformer

RNN 싫어!

1. Parrallelization?
불가능

2. 믿을만한 Context?
No!

3. 긴 문장에 대한 처리 가능도?
부족!

→ RNN 없이 **Linear Layer**로 처리하자

02 Attention

Attention Of Transformer

3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:

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

???

Q, K, V...?
행렬곱...?

Attention A의 B에 대한 **주의의 정도**

Q query : A 이름 벡터
K key : B 이름 벡터
V value : B의 값 벡터

Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

Attention Of Transformer

Attention A의 B에 대한 **주의의 정도**

Q query : **A 자체 벡터**
K key : **B의 이름 벡터**
V value : **B의 값 벡터**

복습 토크나이저
단어→ 벡터

EX. 사과 ~ 단어 사전 12,
~ [0.1 0.4 0.5 0.3]
배 ~ 단어 사전 2023
~ [0.2 0.3 0.4 0.2]

사과에 대한 **배**의 **어텐션**
Q [0.1 0.4 0.5 0.3]
K [0.2 0.3 0.4 0.2]
V [0.5 0.8 0.7 0.9]

를 통해 계산

Attention Of Transformer

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

사과에 대한 **배**의 **어텐션**

Q [0.1 0.4 0.5 0.3]
K [0.2 0.3 0.4 0.2]
V [0.5 0.8 0.7 0.9]

Q [0.1 0.4 0.5 0.3]
K [0.2 0.3 0.4 0.2]
V [0.5 0.8 0.7 0.9]

$$\mathbf{QK}^T = \begin{bmatrix} 0.1 \\ 0.4 \\ 0.5 \\ 0.3 \end{bmatrix} [0.2\ 0.3\ 0.4\ 0.2] = [0.4]$$

$\boldsymbol{d_k} = \mathbf{4}$ (벡터 임베딩 차원)

Why Scale?

$$\frac{\mathbf{QK^T}}{\sqrt{\mathbf{d_k}}}\mathbf{V} = \frac{0.4}{\sqrt{4}} [0.5\ 0.8\ 0.7\ 0.9] = [0.1\ 0.16\ 0.14\ 0.18]$$

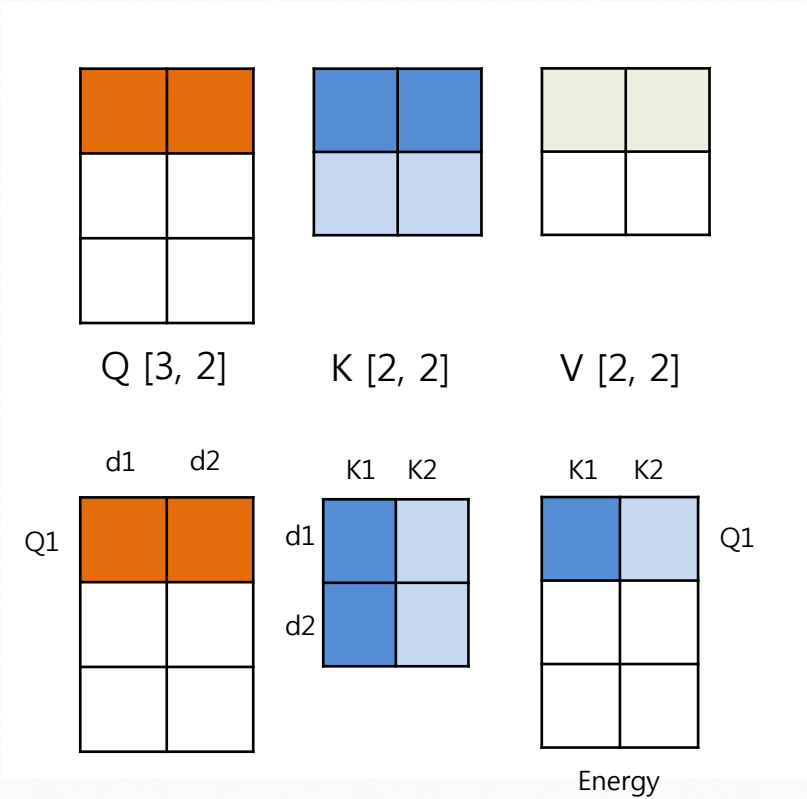
Attention Of Transformer

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

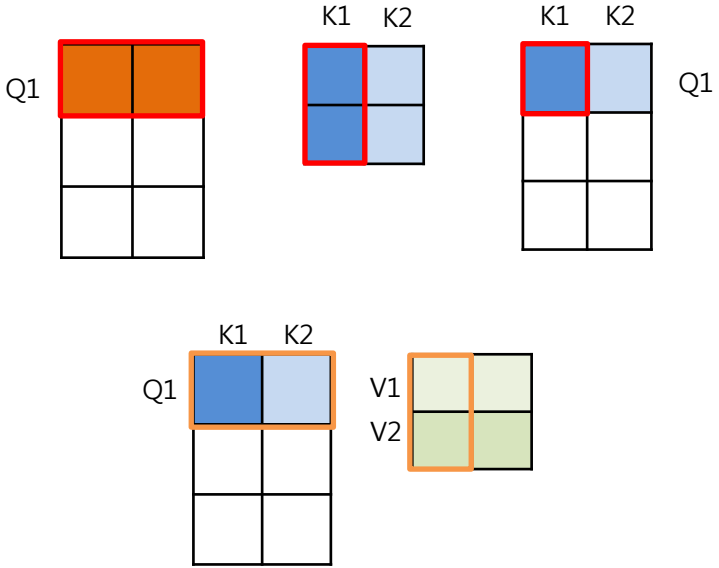
Q [N d_k]
K [M d_k]
V [M d_k]

QK^T [N M]
V [M d_k]

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \text{ [N } d_k\text{]}$$

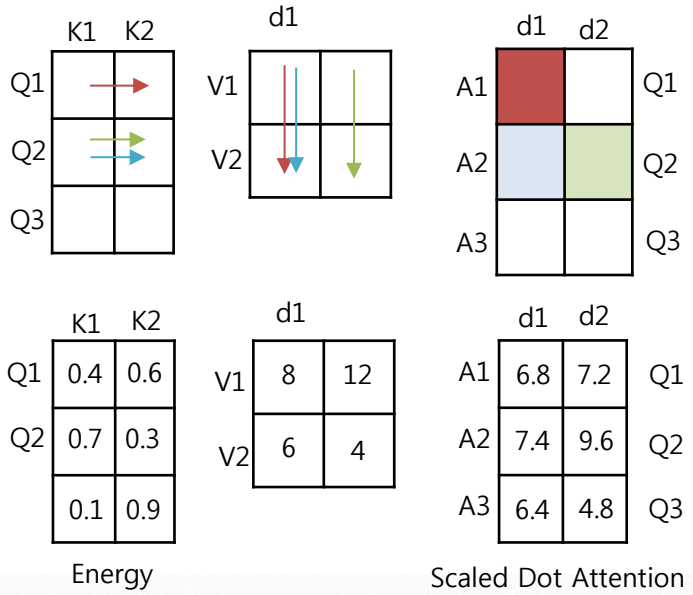


Attention Of Transformer



Q1 (사과)의 K들(배 좋아)의 벡터의 곱
V들(배 좋아)의 벡터의 첫 요소

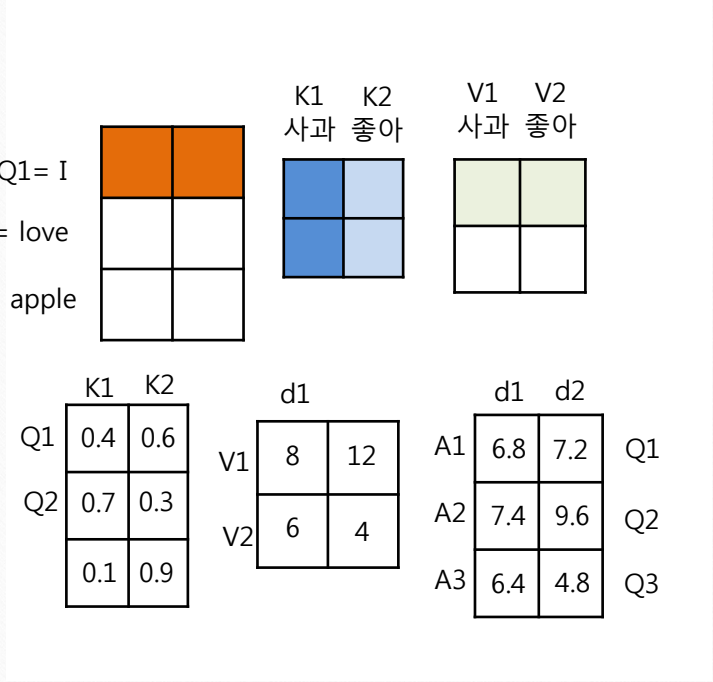
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

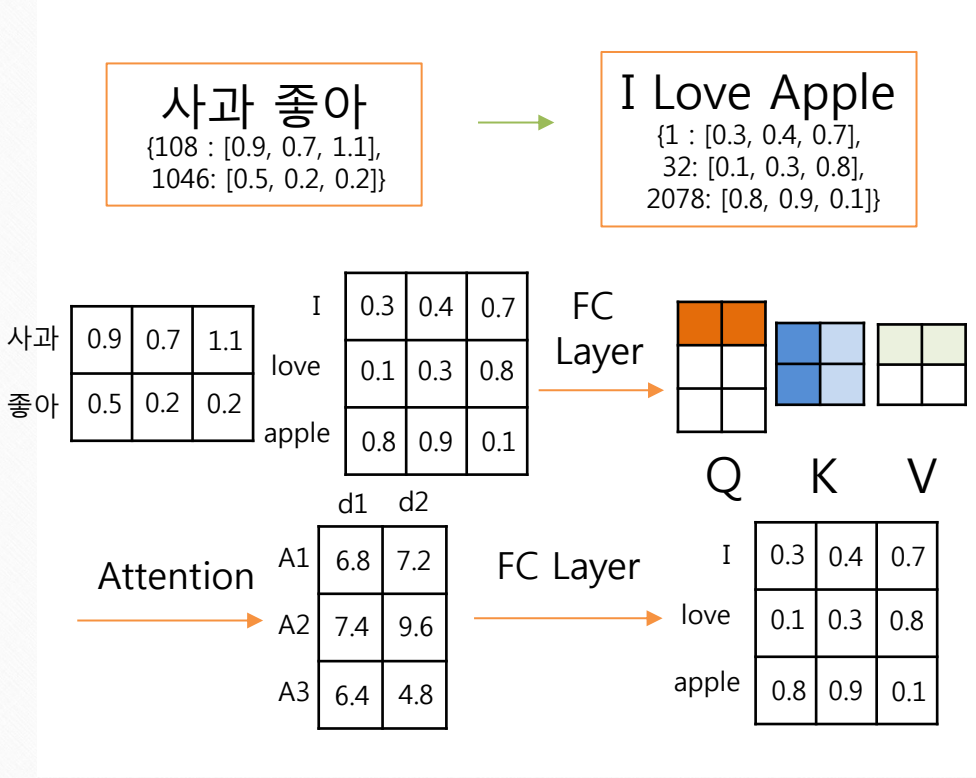


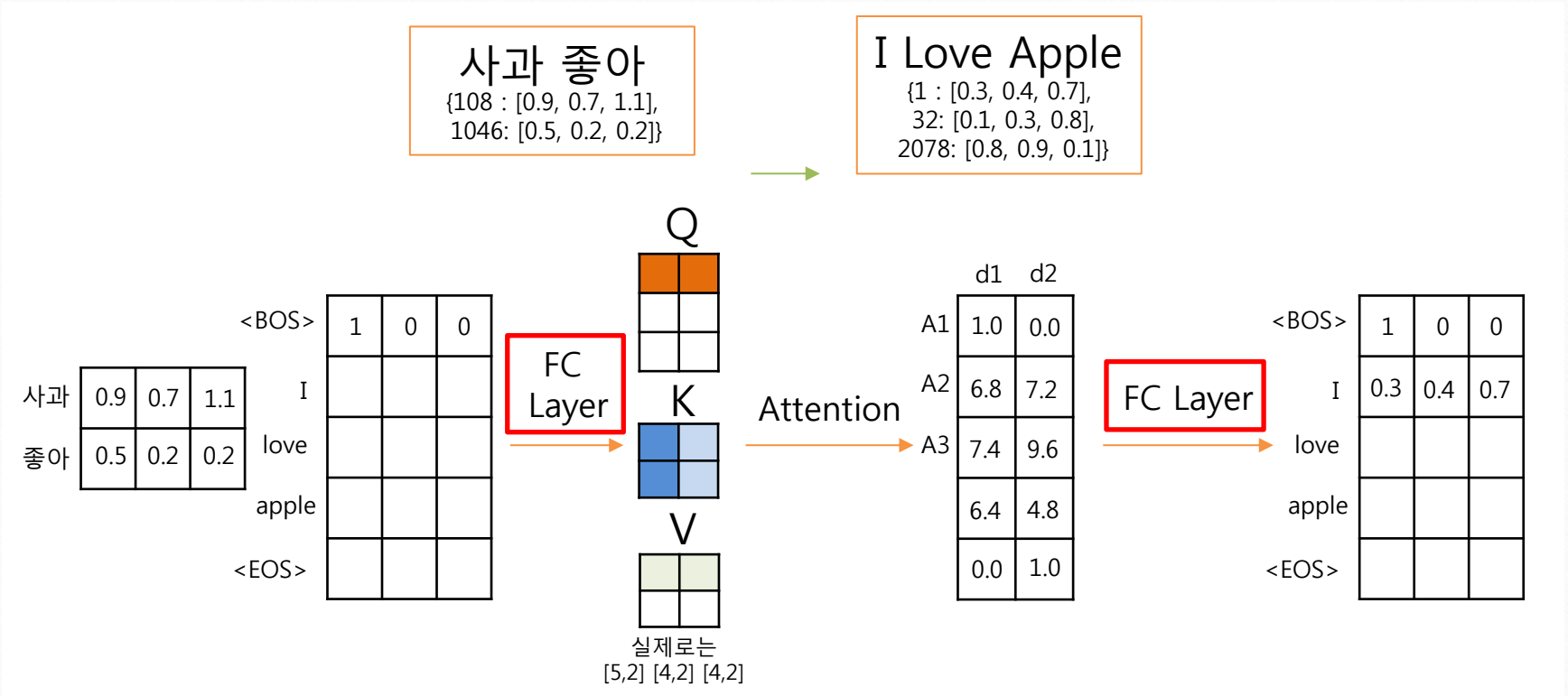
Energy

Scaled Dot Attention

Attention Of Transformer

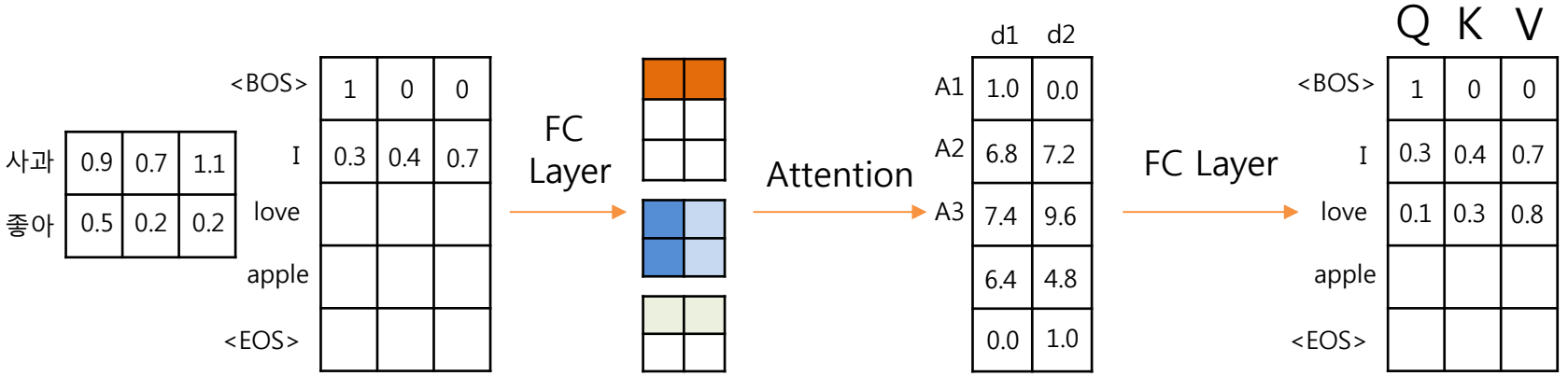





사과 좋아
{108 : [0.9, 0.7, 1.1],
1046: [0.5, 0.2, 0.2]}
I Love Apple
{1 : [0.3, 0.4, 0.7],
32: [0.1, 0.3, 0.8],
2078: [0.8, 0.9, 0.1]}
사과 0.9 0.7 1.1
좋아 0.5 0.2 0.2
<BOS> 1 0 0
I
love
apple
<EOS>
FC Layer
Q
K
V
실제로는
[5,2] [4,2] [4,2]
Attention
d1 d2
A1 1.0 0.0
A2 6.8 7.2
A3 7.4 9.6
6.4 4.8
0.0 1.0
FC Layer
<BOS> 1 0 0
I 0.3 0.4 0.7
love
apple
<EOS>

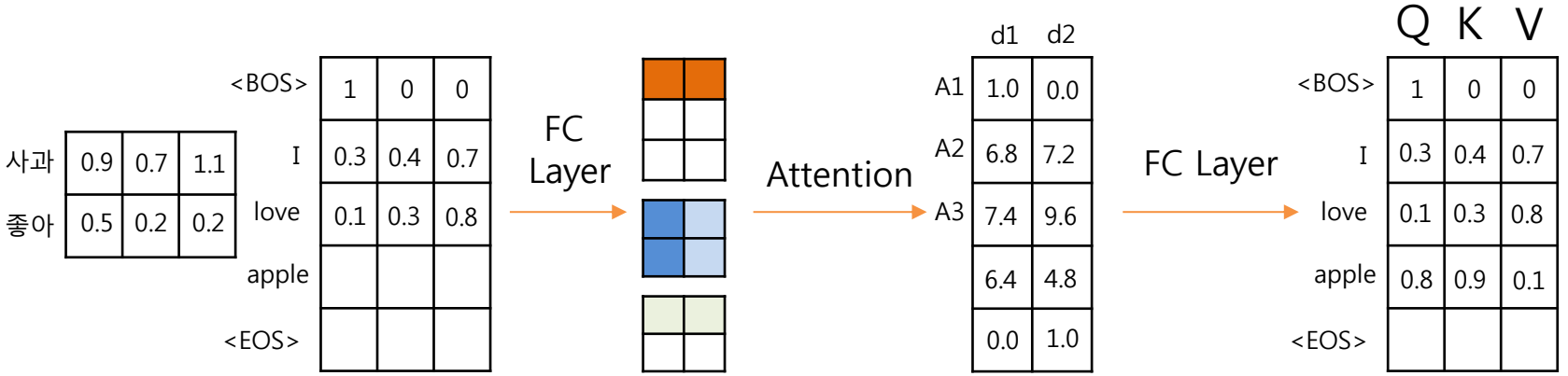
사과 좋아
{108 : [0.9, 0.7, 1.1],
1046: [0.5, 0.2, 0.2]}

I Love Apple
{1 : [0.3, 0.4, 0.7],
32: [0.1, 0.3, 0.8],
2078: [0.8, 0.9, 0.1]}



사과 좋아
{108 : [0.9, 0.7, 1.1],
1046: [0.5, 0.2, 0.2]}

I Love Apple
{1 : [0.3, 0.4, 0.7],
32: [0.1, 0.3, 0.8],
2078: [0.8, 0.9, 0.1]}



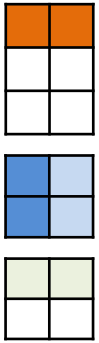
사과 좋아
{108 : [0.9, 0.7, 1.1],
1046: [0.5, 0.2, 0.2]}

I Love Apple
{1 : [0.3, 0.4, 0.7],
32: [0.1, 0.3, 0.8],
2078: [0.8, 0.9, 0.1]}

사과	0.9	0.7	1.1
좋아	0.5	0.2	0.2

<BOS>	1	0	0
I	0.3	0.4	0.7
love	0.1	0.3	0.8
apple	0.8	0.9	0.1
<EOS>			

FC Layer



Attention

	d1	d2
A1	1.0	0.0
A2	6.8	7.2
A3	7.4	9.6
	6.4	4.8
	0.0	1.0

FC Layer

<BOS>	1	0	0
I	0.3	0.4	0.7
love	0.1	0.3	0.8
apple	0.8	0.9	0.1
<EOS>	0	0	1

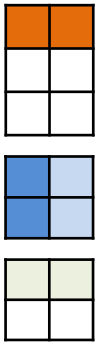
사과 좋아
{108 : [0.9, 0.7, 1.1],
1046: [0.5, 0.2, 0.2]}

I Love Apple
{1 : [0.3, 0.4, 0.7],
32: [0.1, 0.3, 0.8],
2078: [0.8, 0.9, 0.1]}

사과	0.9	0.7	1.1
좋아	0.5	0.2	0.2

<BOS>	1	0	0
I	0.3	0.4	0.7
love	0.1	0.3	0.8
apple	0.8	0.9	0.1
<EOS>	0	0	1

FC Layer



Attention

	d1	d2
A1	1.0	0.0
A2	6.8	7.2
A3	7.4	9.6
	6.4	4.8
	0.0	1.0

FC Layer

<BOS>	1	0	0
I	0.3	0.4	0.7
love	0.1	0.3	0.8
apple	0.8	0.9	0.1
<EOS>	0	0	1

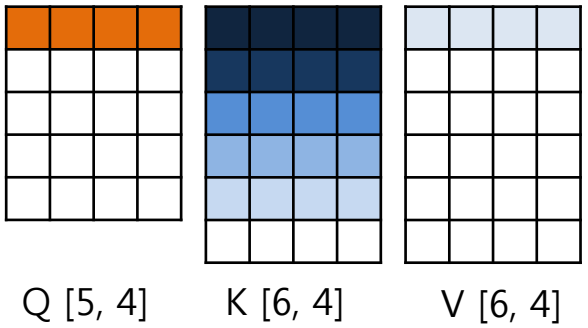
Attention Of Transformer

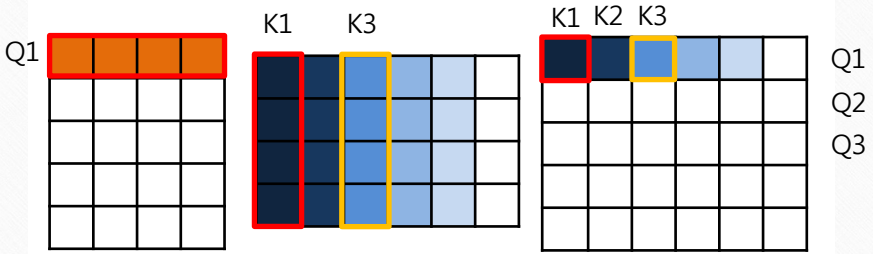
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Q [N d_k]
K [M d_k]
V [M d_k]

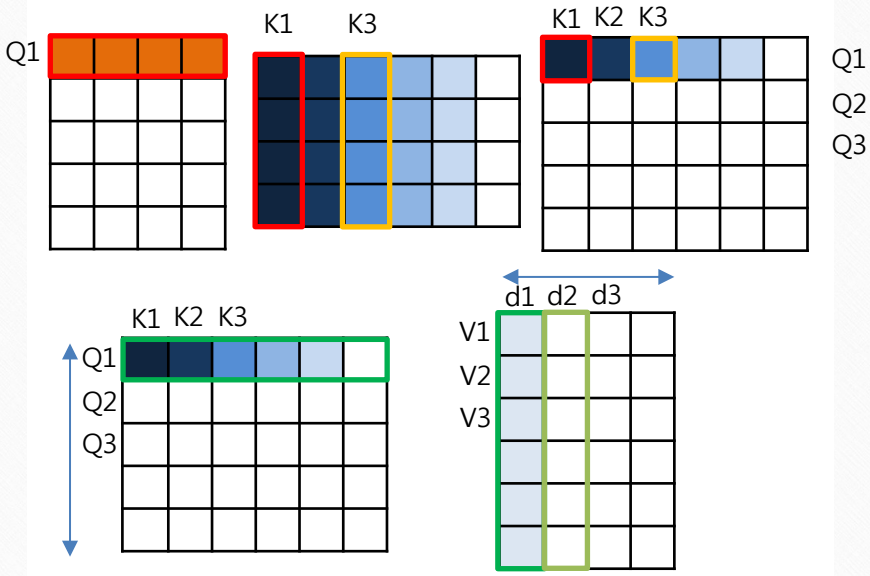
QK^T [N M]
V [M d_k]

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \text{ [N } d_k\text{]}$$





Attention Of Transformer

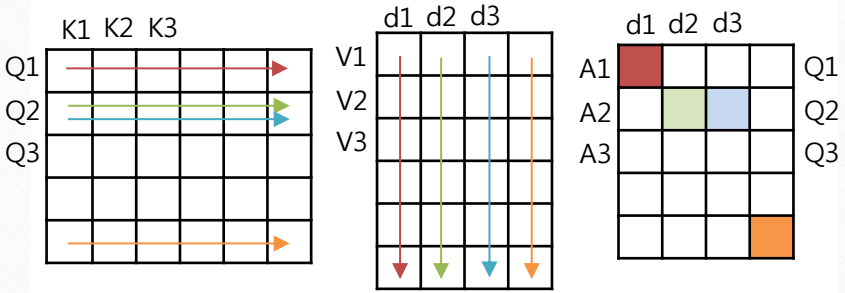


Q1 (사과)의 K들(배 너무 맛있어 좋아)의 벡터의 곱
V들(배 너무 맛있어 좋아)의 벡터의 첫 요소

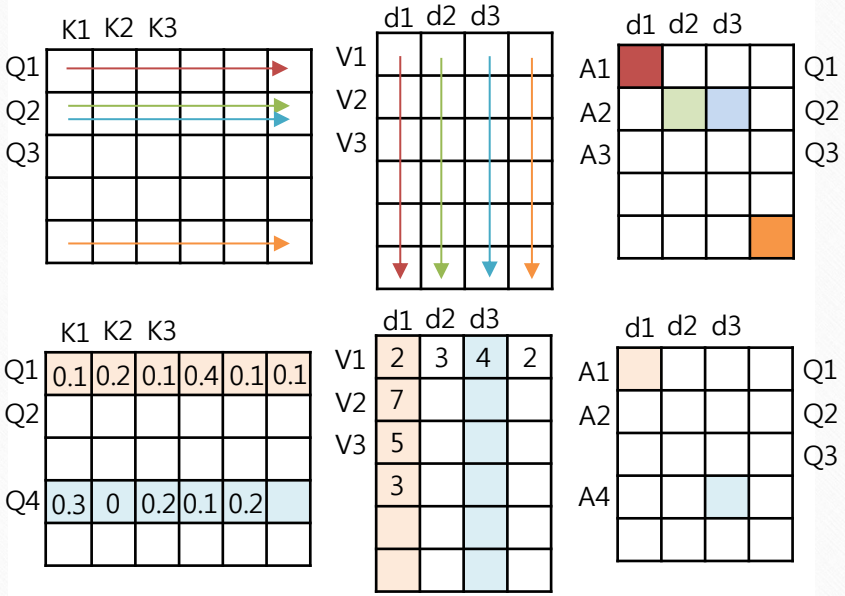
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Q의 길이 차원 (왼쪽 화살표)
V의 토큰 차원 (오른쪽 화살표)
유지

K와 **V**는 한 **Pair**임 (길이 동일



Attention Of Transformer



Q Vector

→ Key와 유사도 잘 표현할 수록 좋음

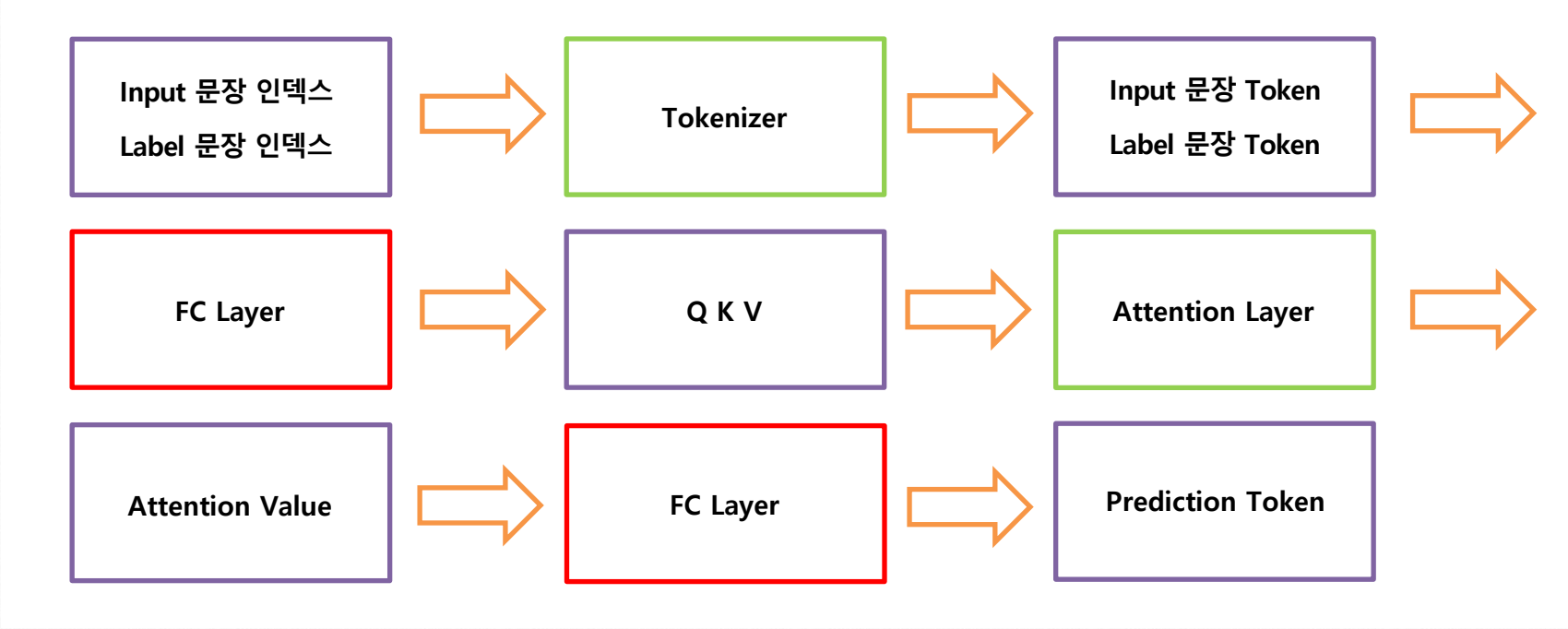
K Vector

→ Query와 유사도 잘 표현할 수록 좋음

V Vector

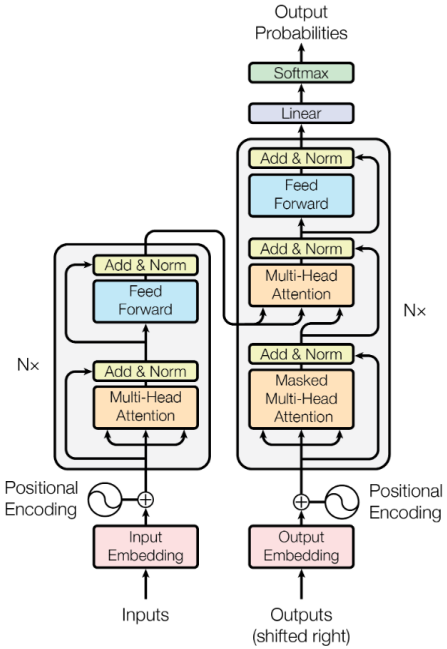
→ 번역 결과를 잘 표현할 수록 좋음

정리


Input 문장 인덱스
Label 문장 인덱스
Tokenizer
Input 문장 Token
Label 문장 Token
FC Layer
Q K V
Attention Layer
Attention Value
FC Layer
Prediction Token

03 Transformer

Transformer



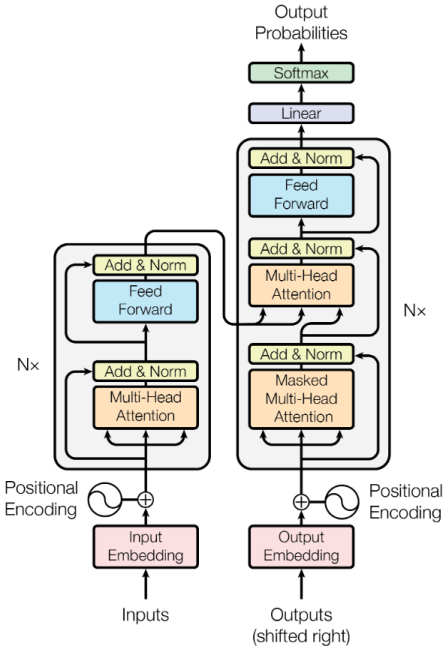
???

걱정할 필요 X

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

를 엄청 많이 쌓은 것!

Transformer



→ **Multi-Head Attention**
: 같은 QKV에 대해
여러번 다른 Attention
(다른 FC Layer)

→ **Feed Forward**
: Attention Layer Output
FC Layer로 통과시키기

→ **Positional Encoding**
: 순서 임베딩

Input Embedding

3.4 Embeddings and Softmax

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model}. We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to [30]. In the embedding layers, we multiply those weights by $\sqrt{d_{\text{model}}}$.

EX. I Love Apple → (1, 32, 2078) →

```
1    : [0.3, 0.4, 0.7]
32   : [0.1, 0.3, 0.8]
2078: [0.8, 0.9, 0.1]
```

Base d_{model}=512 | Best d_{model}=1024 | 실습의 경우 spacy 사용

Positional Encoding

3.5 Positional Encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [9].

No Convolution No Recurrence!
→ 순서를 인위적으로 표현해주어야함

Positional Encoding

In this work, we use sine and cosine functions of different frequencies:

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

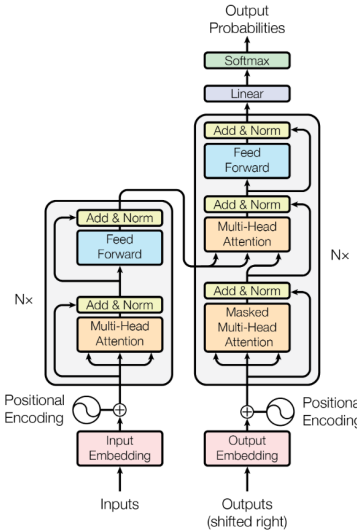
where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

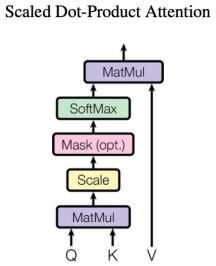
pos : 문장 내 단어의 순서 | i : Embedding Vector의 i번째 원소

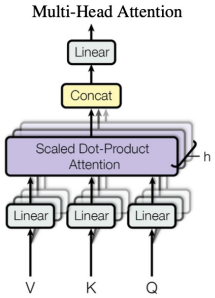
위 함수를 Embedding에 더해줌 ~ 순서 학습 가능

Detail에서 더 자세하게 설명하겠음

Multi Head Attention







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

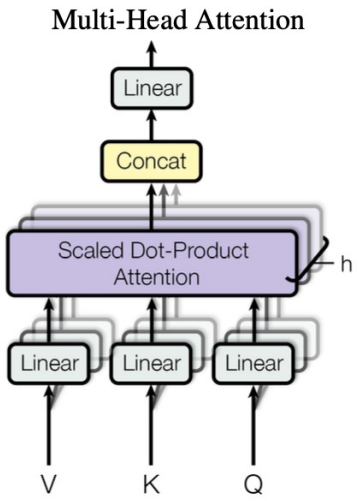
Multi Head Attention

→ 서로 다른 컨셉의 Attention 적용

→ 그림의 그림자 부분

→ Token ~ Q K V 로 가는 FC Layer를 다양하게 사용

Multi Head Attention



3.2.2 Multi-Head Attention

Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

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

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.

1. d_{model} 을 d_{model}/h로 Linear Transform
EX. d_{model} 512 $head$ 8 → $d_k = d_v = 64$
2. Linear Transform Layer를 $head$ 갯수 만큼 Train
3. Concat Everything

Multi Head Attention

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

```
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 EX. 512
        self.n_heads = n_heads                          # 헤드(head)의 개수: 서로 다른 어텐션(attention) 컨셉의 수 EX. 8
        self.head_dim = hidden_dim // n_heads           # 각 헤드(head)에서의 임베딩 차원 EX. 512 // 8 ~ 64

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

        self.fc_o = nn.Linear(hidden_dim, hidden_dim) # EX. 설명 예제, Output Token으로 바꾸기!

        self.dropout = nn.Dropout(dropout_ratio)

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

Multi Head Attention

```
def forward(self, query, key, value, mask = None):
    batch_size = query.shape[0]

    Q = self.fc_q(query) # query: [batch_size, query_len, hidden_dim]
    K = self.fc_k(key)   # key: [batch_size, key_len, hidden_dim]
    V = self.fc_v(value) # value: [batch_size, value_len, hidden_dim]

    # 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]
```

```
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
```

Multi Head Attention

```
# Attention Energy 계산
energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) / self.scale # Q @ K.T 부분
# 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
    # 아까 본 "I Love Apple --> 사과 좋아" 과정에서 <bos> ~ <eos> 과정 생각
    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]
x = x.permute(0, 2, 1, 3).contiguous()       # x: [batch_size, query_len, n_heads, head_dim]
x = x.view(batch_size, -1, self.hidden_dim)  # x: [batch_size, query_len, hidden_dim]
x = self.fc_o(x)                             # x: [batch_size, query_len, hidden_dim]

return x, attention
```

Position Wise Feed Forward

3.3 Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{ff} = 2048$.

---

1. MultiHeadAttention의 출력값 ~ Linear 두 번 통과
2. Hidden Dim 512 ~ First 2048 ~ Out 512 순서-!
3. Activation은 ReLU

→ 아하! **Attention Output 결과에 비선형성 추가**

Position Wise Feed Forward

```python
class PositionwiseFeedforwardLayer(nn.Module):
    def __init__(self, hidden_dim, pf_dim, dropout_ratio):
        super().__init__()

        self.fc_1 = nn.Linear(hidden_dim, pf_dim)
        self.fc_2 = nn.Linear(pf_dim, hidden_dim)  # EX. pf_dim : 4 * hidden_dim

        self.dropout = nn.Dropout(dropout_ratio)

    def forward(self, x):                           # x: [batch_size, seq_len, hidden_dim]
        x = self.dropout(torch.relu(self.fc_1(x)))  # x: [batch_size, seq_len, pf_dim]
        x = self.fc_2(x)                            # x: [batch_size, seq_len, hidden_dim]

        return x
```

Position Wise?

→ If Not, [batch_size, seq_len * hidden_dim] 통과 시킴

→ Position Wise, [batch_size, seq_len, hidden_dim]
hidden_dim 만 돌림!

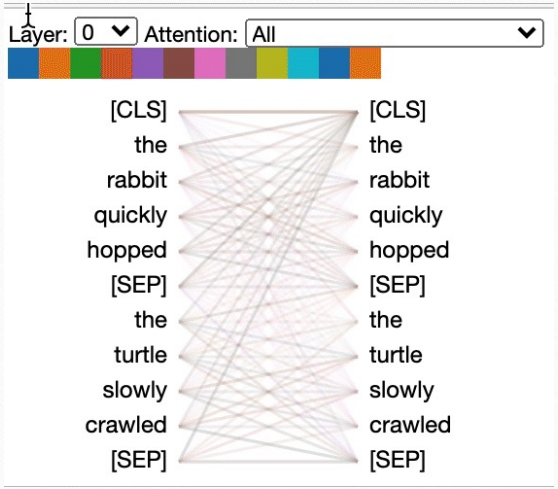
Encoder Layer

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations [4, 27, 28, 22].

MultiHead (Self Attention) + FeedForward + Residual + LayerNorm

Encoder Layer - Self Attention



Self Attention?

→ 자신을 Query Key Value로 사용

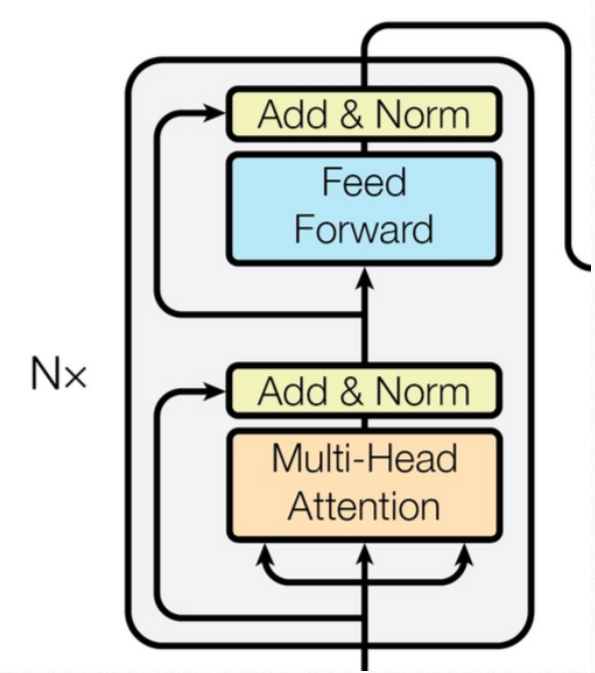
→ [1] 단어간의 관계를 학습

→ [2] 문장 자체의 Representaion을 학습
- Self Attention 이후의 Feed Forward
- Recurrency 나 Convolution에 비해 효율적

→ 이후 Code, Input이 (src, src, src) 혹은 (trg, trg, trg)

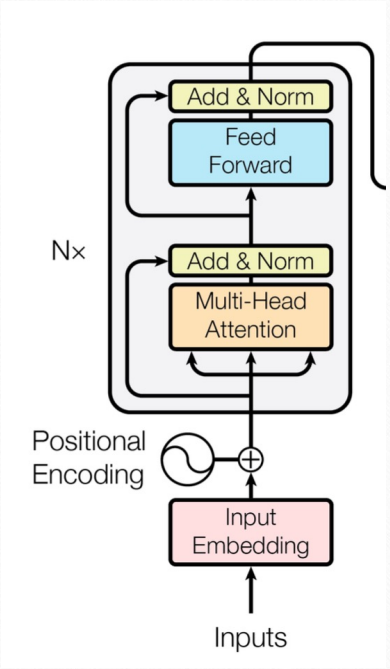
- The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.

Encoder Layer - Code




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

        self.self_attn_layer_norm = nn.LayerNorm(hidden_dim)
        self.ff_layer_norm = nn.LayerNorm(hidden_dim)
        self.self_attention = MultiHeadAttentionLayer(hidden_dim, n_heads, dropout_ratio, device)
        self.positionwise_feedforward = PositionwiseFeedforwardLayer(hidden_dim, pf_dim, dropout_ratio)
        self.dropout = nn.Dropout(dropout_ratio)

    # 하나의 임베딩이 복제되어 Query, Key, Value로 입력되는 방식
    def forward(self, src, src_mask):
        # src: [batch_size, src_len, hidden_dim]
        # src_mask: [batch_size, src_len]

        # self attention ## 필요한 경우 마스크(mask) 행렬을 이용하여 어텐션(attention)할 단어를 조절 가능
        _src, _ = self.self_attention(src, src, src, src_mask)

        src = self.self_attn_layer_norm(src + self.dropout(_src))
        # dropout, residual connection and layer norm
        # src: [batch_size, src_len, hidden_dim]

        _src = self.positionwise_feedforward(src)           # position-wise feedforward
        src = self.ff_layer_norm(src + self.dropout(_src))  # dropout, residual and layer norm
        # src: [batch_size, src_len, hidden_dim]

        return src
```

Encoder



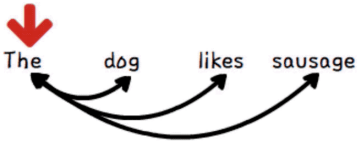

```python
class Encoder(nn.Module):
    def __init__(self, input_dim, hidden_dim, n_layers, n_heads, pf_dim, dropout_ratio, device, max_length=100):
        super().__init__()
        self.device = device

        self.tok_embedding = nn.Embedding(input_dim, hidden_dim)
        self.pos_embedding = nn.Embedding(max_length, hidden_dim)

        self.layers = nn.ModuleList([EncoderLayer(hidden_dim, n_heads, pf_dim, dropout_ratio, device) for _ in range(n_layers)])

        self.dropout = nn.Dropout(dropout_ratio)
        self.scale = torch.sqrt(torch.FloatTensor([hidden_dim])).to(device)

    def forward(self, src, src_mask): # src: [batch_size, src_len] # src_mask: [batch_size, src_len]
        batch_size, src_len = src.shape
        pos = torch.arange(0, src_len).unsqueeze(0).repeat(batch_size, 1).to(self.device) # pos: [batch_size, src_len]

        # 소스 문장의 임베딩과 위치 임베딩을 더한 것을 사용
        src = self.dropout((self.tok_embedding(src) * self.scale) + self.pos_embedding(pos)) # src: [batch_size, src_len, hidden_dim]

        # 모든 인코더 레이어를 차례대로 거치면서 순전파(forward) 수행
        for layer in self.layers:  # EX. self.layers : 6개의 Encoding Layer (MHA 6개, FF 6개)
            src = layer(src, src_mask) # src: [batch_size, src_len, hidden_dim]

        return src # 마지막 레이어의 출력을 반환
```

Decoder

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

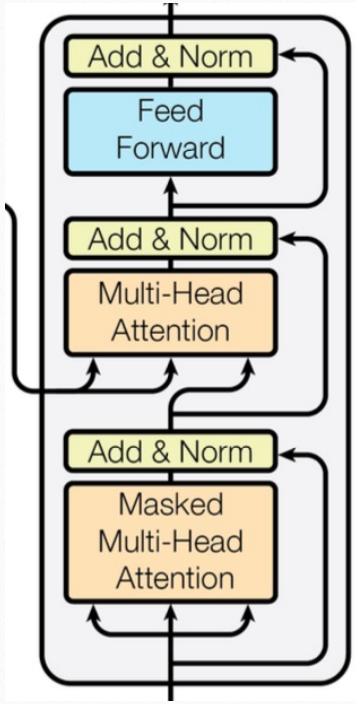
- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as [38, 2, 9].
- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections. See Figure 2.

Decoder



[Figure 3] self-attention machnism

Decoder Layer - Code

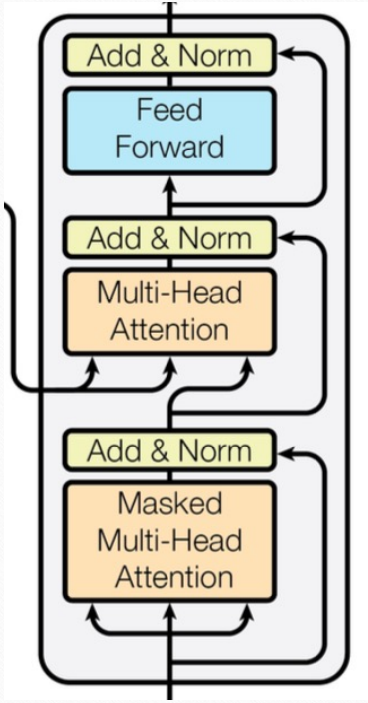



```
class DecoderLayer(nn.Module):
    def __init__(self, hidden_dim, n_heads, pf_dim, dropout_ratio, device):
        super().__init__()

        self.self_attn_layer_norm = nn.LayerNorm(hidden_dim)
        self.enc_attn_layer_norm = nn.LayerNorm(hidden_dim)
        self.ff_layer_norm = nn.LayerNorm(hidden_dim)
        self.self_attention = MultiHeadAttentionLayer(hidden_dim, n_heads, dropout_ratio, device)
        self.encoder_attention = MultiHeadAttentionLayer(hidden_dim, n_heads, dropout_ratio, device)
        self.positionwise_feedforward = PositionwiseFeedforwardLayer(hidden_dim, pf_dim, dropout_ratio)
        self.dropout = nn.Dropout(dropout_ratio)
```

Decoder Layer

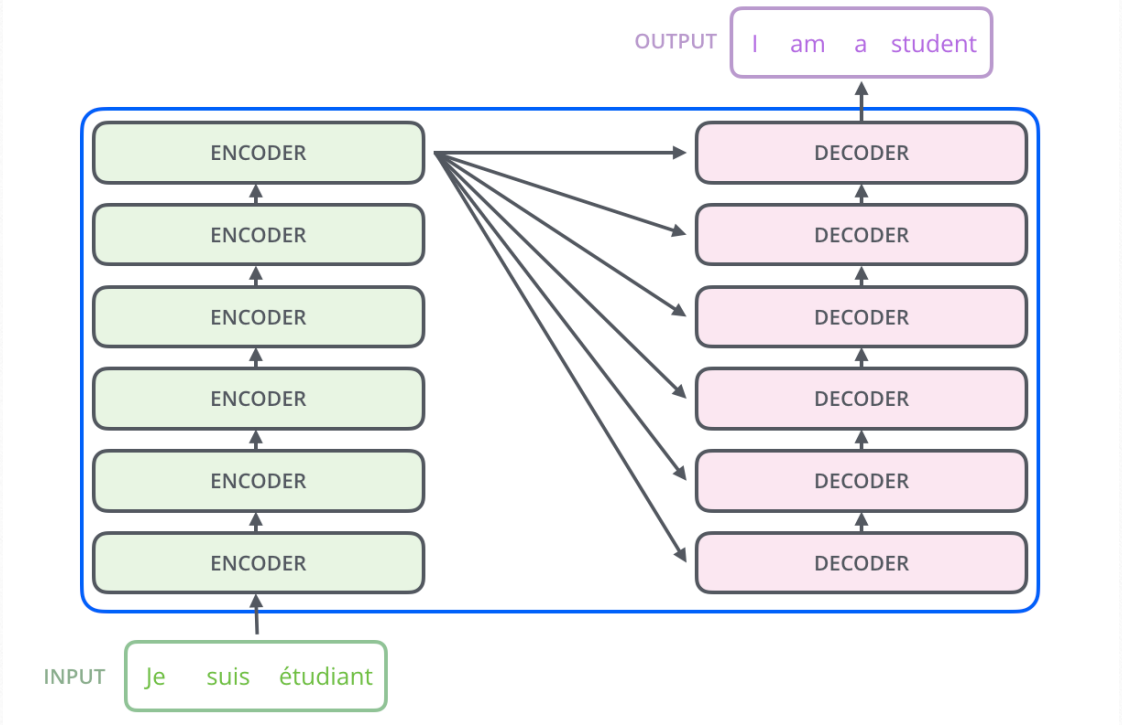
1. Self Attention
2. Decoder의 모든 층, Model Output Query 이용
Encoder를 활용해 어텐션 (KV from Encoder)
3. Encoder Layer와 비교,
Encoder와의 Attention 계산하는 층 하나 추가

Decoder Layer - Code

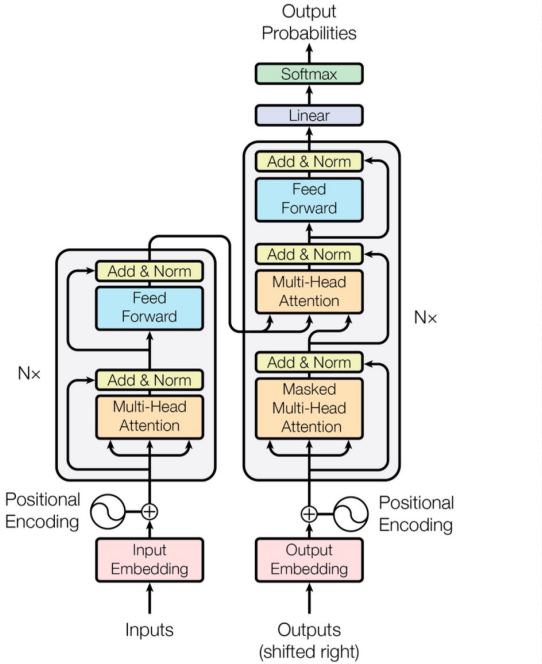



```python
# 인코더의 출력 값(enc_src)을 어텐션(attention)하는 구조
def forward(self, trg, enc_src, trg_mask, src_mask):

    # trg: [batch_size, trg_len, hidden_dim]
    # enc_src: [batch_size, src_len, hidden_dim]
    # trg_mask: [batch_size, trg_len]
    # src_mask: [batch_size, src_len]

    # self attention
    _trg, _ = self.self_attention(trg, trg, trg, trg_mask)
    trg = self.self_attn_layer_norm(trg + self.dropout(_trg)) # dropout, residual connection and layer norm
    # trg: [batch_size, trg_len, hidden_dim]

    # encoder attention # 디코더의 쿼리(Query)를 이용해 인코더를 어텐션(attention)
    _trg, attention = self.encoder_attention(trg, enc_src, enc_src, src_mask)
    trg = self.enc_attn_layer_norm(trg + self.dropout(_trg)) # dropout, residual connection and layer norm
    # trg: [batch_size, trg_len, hidden_dim]

    _trg = self.positionwise_feedforward(trg)              # positionwise feedforward
    trg = self.ff_layer_norm(trg + self.dropout(_trg))     # dropout, residual and layer norm

    # trg: [batch_size, trg_len, hidden_dim]
    # attention: [batch_size, n_heads, trg_len, src_len]
    return trg, attention
```

Encoder Decoder Attention



Decoder - Code




```python
class Decoder(nn.Module):
    def __init__(self, output_dim, hidden_dim, n_layers, n_heads, pf_dim, dropout_ratio, device, max_length=100):
        super().__init__()
        self.device = device

        self.tok_embedding = nn.Embedding(output_dim, hidden_dim)
        self.pos_embedding = nn.Embedding(max_length, hidden_dim)

        self.layers = nn.ModuleList([DecoderLayer(hidden_dim, n_heads, pf_dim, dropout_ratio, device) for _ in range(n_layers)])
        self.fc_out = nn.Linear(hidden_dim, output_dim)

        self.dropout = nn.Dropout(dropout_ratio)
        self.scale = torch.sqrt(torch.FloatTensor([hidden_dim])).to(device)

    def forward(self, trg, enc_src, trg_mask, src_mask):

        # trg: [batch_size, trg_len]
        # enc_src: [batch_size, src_len, hidden_dim]
        # trg_mask: [batch_size, trg_len]
        # src_mask: [batch_size, src_len]

        batch_size = trg.shape[0]
        trg_len = trg.shape[1]

        pos = torch.arange(0, trg_len).unsqueeze(0).repeat(batch_size, 1).to(self.device)      # pos: [batch_size, trg_len]
        trg = self.dropout((self.tok_embedding(trg) * self.scale) + self.pos_embedding(pos))   # trg: [batch_size, trg_len, hidden_dim]

        for layer in self.layers: # 소스 마스크와 타겟 마스크 모두 사용
            trg, attention = layer(trg, enc_src, trg_mask, src_mask)
        # trg: [batch_size, trg_len, hidden_dim]
        # attention: [batch_size, n_heads, trg_len, src_len]

        output = self.fc_out(trg) # output: [batch_size, trg_len, output_dim]
        return output, attention
```

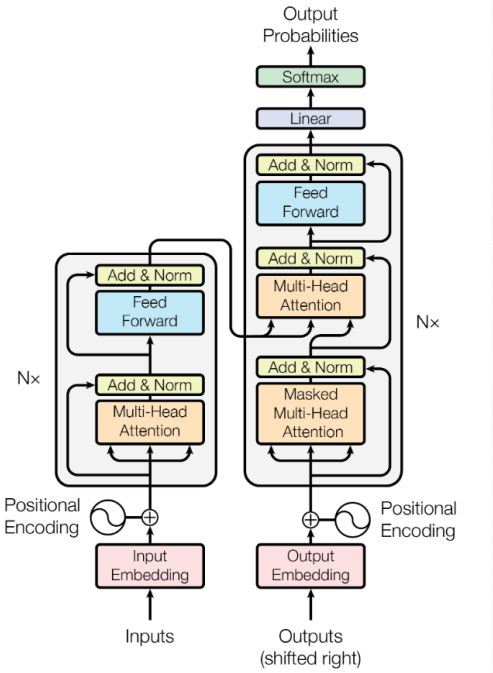
Transformer

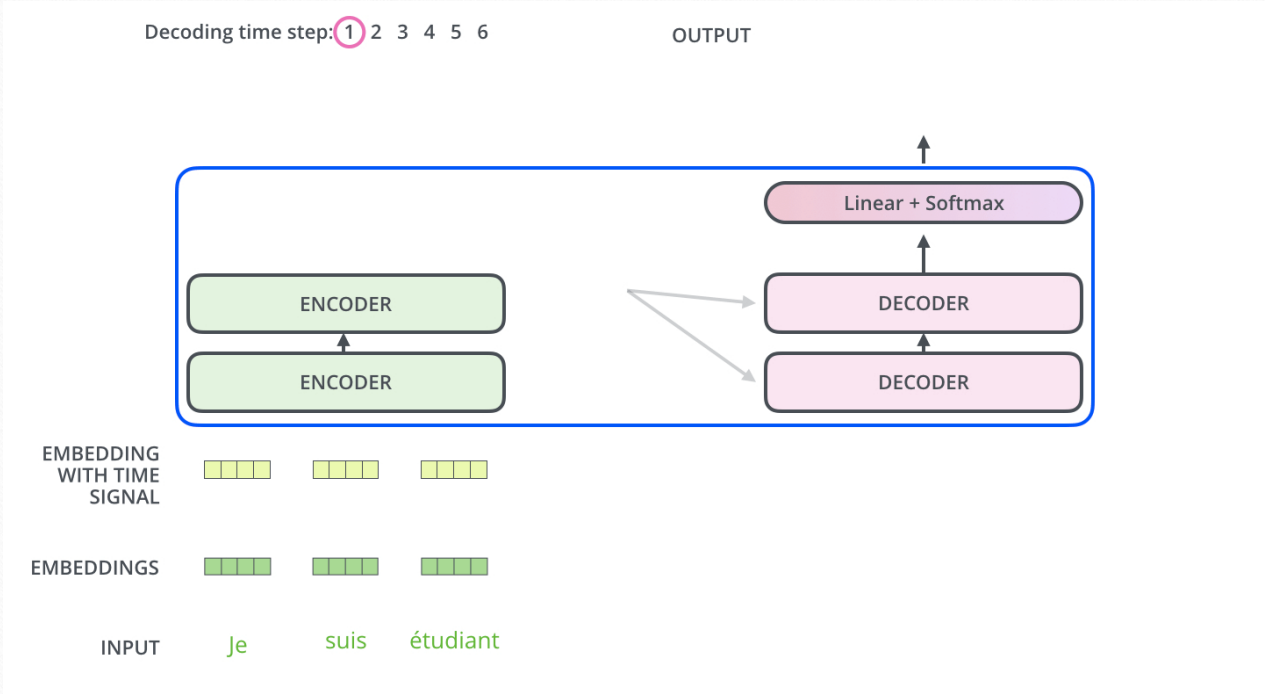
Transformer

```python
class Transformer(nn.Module):
    def __init__(self, encoder, decoder, src_pad_idx, trg_pad_idx, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.src_pad_idx = src_pad_idx
        self.trg_pad_idx = trg_pad_idx
        self.device = device

    def forward(self, src, trg):
        # src: [batch_size, src_len]
        # trg: [batch_size, trg_len]

        src_mask = self.make_src_mask(src)    # src_mask: [batch_size, 1, 1, src_len]
        trg_mask = self.make_trg_mask(trg)    # trg_mask: [batch_size, 1, trg_len, trg_len]

        enc_src = self.encoder(src, src_mask) # enc_src: [batch_size, src_len, hidden_dim]
        output, attention = self.decoder(trg, enc_src, trg_mask, src_mask)
        # output: [batch_size, trg_len, output_dim]
        # attention: [batch_size, n_heads, trg_len, src_len]

        return output, attention
```

```python
# 소스 문장의 <pad> 토큰에 대하여 마스크(mask) 값을 0으로 설정
def make_src_mask(self, src): # src: [batch_size, src_len]
    src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)  # src_mask: [batch_size, 1, 1, src_len]
    return src_mask

# 타겟 문장에서 각 단어는 다음 단어가 무엇인지 알 수 없도록(이전 단어만 보도록) 만들기 위해 마스크를 사용
def make_trg_mask(self, trg):  # trg: [batch_size, trg_len]
    """ (마스크 예시)
    1 0 0 0 0
    1 1 0 0 0
    1 1 1 0 0
    1 1 1 0 0
    1 1 1 0 0
    """
    trg_pad_mask = (trg != self.trg_pad_idx).unsqueeze(1).unsqueeze(2) # trg_pad_mask: [batch_size, 1, 1, trg_len]

    trg_len = trg.shape[1]
    """ (마스크 예시)
    1 0 0 0 0
    1 1 0 0 0
    1 1 1 0 0
    1 1 1 1 0
    1 1 1 1 1
    """
    trg_sub_mask = torch.tril(torch.ones((trg_len, trg_len), device = self.device)).bool()  # trg_sub_mask: [trg_len, trg_len]
    trg_mask = trg_pad_mask & trg_sub_mask   # trg_mask: [batch_size, 1, trg_len, trg_len]

    return trg_mask
```

Src Mask : 문장들의 길이가 다를 때 사용
Trg Mask : Decoder가 치팅하는 것 방지


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


Decoding time step: 1 2 3 4 5 6
OUTPUT
Linear + Softmax
ENCODER
ENCODER
DECODER
DECODER
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant

04 Details

Positional Encoding

In this work, we use sine and cosine functions of different frequencies:

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

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

Positional Encoding

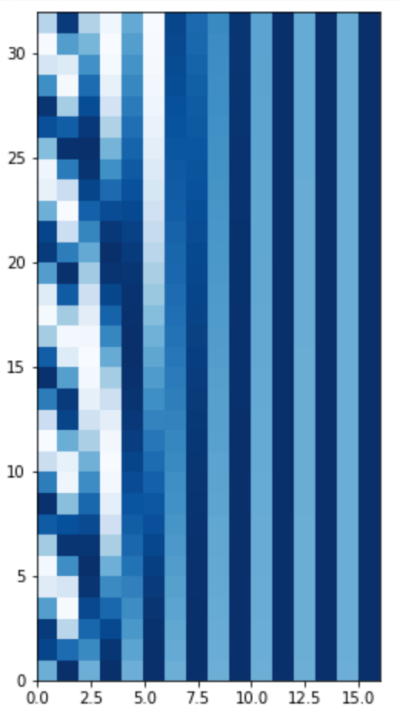
Ideal Criteria for Positional Encoding

특정 Encoding 방법론이 이상적인 Positional Encoding이기 위해서 만족해야 할 조건은 다음과 같습니다.

- 각 위치의 Positional Encoding 값은 결정론적(Deterministic)하게 결정되는 동시에 유일해야 한다.
- 데이터와 관계없이 각 위치의 Positional Encoding 값은 동일해야 한다.
- 데이터의 길이와 상관없이 적용 가능해야 한다.
- 간격이 동일하다면 위치와 관계없이 거리는 동일해야 한다. ($\mathrm{distance}(x, x+h) = \mathrm{distance}(y, y+h)$)
- 데이터와 관계없이 i번째 위치부터 j번째 위치(i < j)까지의 거리는 동일해야 한다.

https://tigris-data-science.tistory.com/entry/%EC%B0%A8%EA%B7%BC%EC%B0%A8%EA%B7%BC-%EC%9D%B4%ED%95%B4%ED%95%98%EB%8A%94-Transformer5-Positional-Encoding

Positional Encoding



Simple Indexing (Count)

Simple Indexing은 말 그대로 각 위치의 인덱스를 Positional Encoding 값으로 사용하는 것입니다.

$$\text{Positional Encoding}(a_i) = i$$

이 방법은 이상적인 Positional Encoding의 조건을 모두 만족하지만, 입력값의 길이가 길어질수록 Encoding 값이 커져 학습 시 문제(Gradient Exploding 등)를 겪을 가능성이 높아지기 때문에 실제 사용에 한계가 있습니다.

Normalize Indexing (Normalize Count)

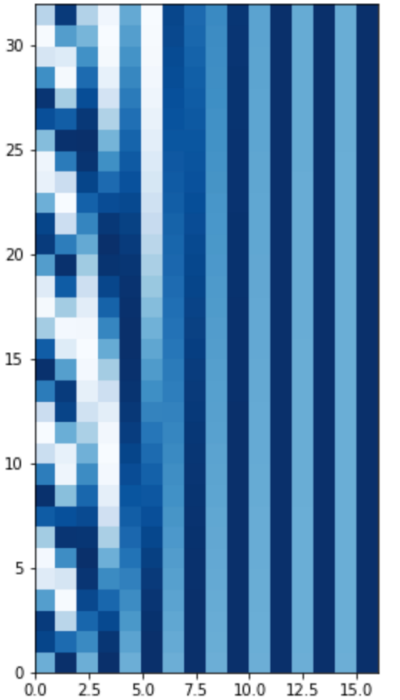
Normalize Indexing은 값이 커지지 않도록 Simple Indexing의 결과를 문장의 길이로 나누어 0부터 1까지의 값으로 표준화하는 방법입니다.

$$\text{Positional Encoding}(a_i) = \frac{i}{T}$$

이 방법은 Simple Indexing에서 발생하는 문제는 해결했지만, 이상적인 Positional Encoding의 두 번째 조건을 만족하지 못합니다. (예: 길이가 20인 데이터에서 8번째 Positional Encoding 값과 길이가 10인 데이터의 4번째 Positional Encoding 값이 동일)

https://tigris-data-science.tistory.com/entry/%EC%B0%A8%EA%B7%BC%EC%B0%A8%EA%B7%BC-%EC%9D%B4%ED%95%B4%ED%95%98%EB%8A%94-Transformer5-Positional-Encoding

Positional Encoding



Binary Indexing

Binary Indexing은 Simple Indexing의 결과를 2진수를 이용해 표현하는 방식입니다. 예를 들어 데이터의 길이가 5일 때 Binary Indexing을 이용한 Positional Encoding 값은 다음과 같습니다.

Position		0	1	2	3	4	5
Position Encoding	2^2	0	0	0	0	1	1
	2^1	0	0	1	1	0	0
	2^0	0	1	0	1	0	1

Binary Indexing 예제

Simple Indexing과 같이 숫자가 커지는 문제도 없고, 2, 3번 조건도 만족하는 것 같지만 4번 조건을 만족하지 못합니다. (예: $\text{distance}(a_1, a_2) = \sqrt{2},\ \text{distance}(a_2, a_3) = 1$)

Positional Encoding

In this work, we use sine and cosine functions of different frequencies:

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

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

$$PE_{pos+\Delta step} = PE_{pos} \cdot T(\Delta step)$$

그리고 마침 삼각함수에는 임의의 행렬을 원점을 중심으로 회전시켜주는 선형 변환인 회전변환행렬(Rotation Matrix)이 존재합니다.

$$\begin{bmatrix} \cos(\theta+\phi) \\ \sin(\theta+\phi) \end{bmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$$

https://tigris-data-science.tistory.com/entry/%EC%B0%A8%EA%B7%BC%EC%B0%A8%EA%B7%BC-%EC%9D%B4%ED%95%B4%ED%95%98%EB%8A%94-Transformer5-Positional-Encoding

Training

5.1 Training Data and Batching

We trained on the standard WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs. Sentences were encoded using byte-pair encoding [3], which has a shared source-target vocabulary of about 37000 tokens. For English-French, we used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary [38]. Sentence pairs were batched together by approximate sequence length. Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

5.2 Hardware and Schedule

We trained our models on one machine with 8 NVIDIA P100 GPUs. For our base models using the hyperparameters described throughout the paper, each training step took about 0.4 seconds. We trained the base models for a total of 100,000 steps or 12 hours. For our big models,(described on the bottom line of table 3), step time was 1.0 seconds. The big models were trained for 300,000 steps (3.5 days).

→ 독일어-영어 번역 데이터셋
→ 비슷한 길이끼리 Bucket Sampling
→ 1억 5천만 원 정도 GPU로 3.5일...

5.3 Optimizer

We used the Adam optimizer [20] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. We varied the learning rate over the course of training, according to the formula:

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \quad (3)$$

This corresponds to increasing the learning rate linearly for the first $warmup\_steps$ training steps, and decreasing it thereafter proportionally to the inverse square root of the step number. We used $warmup\_steps = 4000$.

5.4 Regularization

We employ three types of regularization during training:

Residual Dropout We apply dropout [33] to the output of each sub-layer, before it is added to the sub-layer input and normalized. In addition, we apply dropout to the sums of the embeddings and the positional encodings in both the encoder and decoder stacks. For the base model, we use a rate of $P_{drop} = 0.1$.

→ ADAM + Customized Scheduler
→ Residual DropOut

Result

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

요약

짱좋음
다 이김

BLUE Score

BLEU

BLEU(Bilingual Evaluation Understudy)score란 성과지표로 데이터의 X가 순서정보를 가진 단어들(문장)로 이루어져 있고, y 또한 단어들의 시리즈(문장)로 이루어진 경우에 사용되며, 번역을 하는 모델에 주로 사용된다. 3가지 요소를 살펴보자.

- n-gram을 통한 순서쌍들이 얼마나 겹치는지 측정(precision)
- 문장길이에 대한 과적합 보정 (Brevity Penalty)
- 같은 단어가 연속적으로 나올때 과적합 되는 것을 보정(Clipping)

$$BLEU = min(1, \frac{output\ length(\text{예측 문장})}{reference\ length(\text{실제 문장})})(\prod_{i=1}^{4} precision_i)^{\frac{1}{4}}$$

감사합니다