

Attention is all you need

a.k.a. Transformer

19.03.30

김현우(<https://hwkim94.github.io>)

Introduction

Transformer 이전의 SOTA : RNN + Encoder-Decoder + Attention

Introduction

Transformer 이전의 SOTA : **RNN** + Encoder-Decoder + Attention

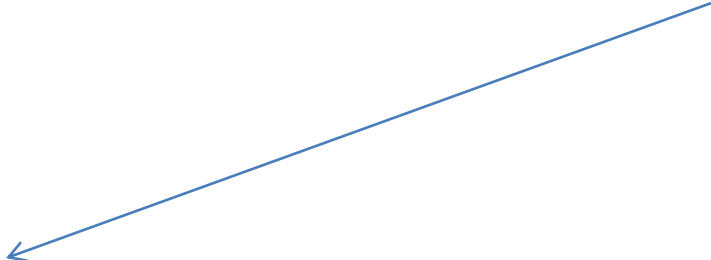


"This inherently sequential nature precludes parallelization"

즉, previous hidden state를 사용하기 때문에 병렬처리 할 수 없어서 학습에 오랜 소요

Introduction

Transformer 이전의 SOTA : RNN + Encoder-Decoder + **Attention**



"allowing modeling of dependencies without regard to their distance in the input or output sequences"
즉, 문장의 길이(단어들 사이의 거리)에 상관없이 고려해야 하는 단어의 중요도를 파악할 수 있게 도와줌

Introduction

Transformer 이전의 SOTA :  Encoder-Decoder + Attention

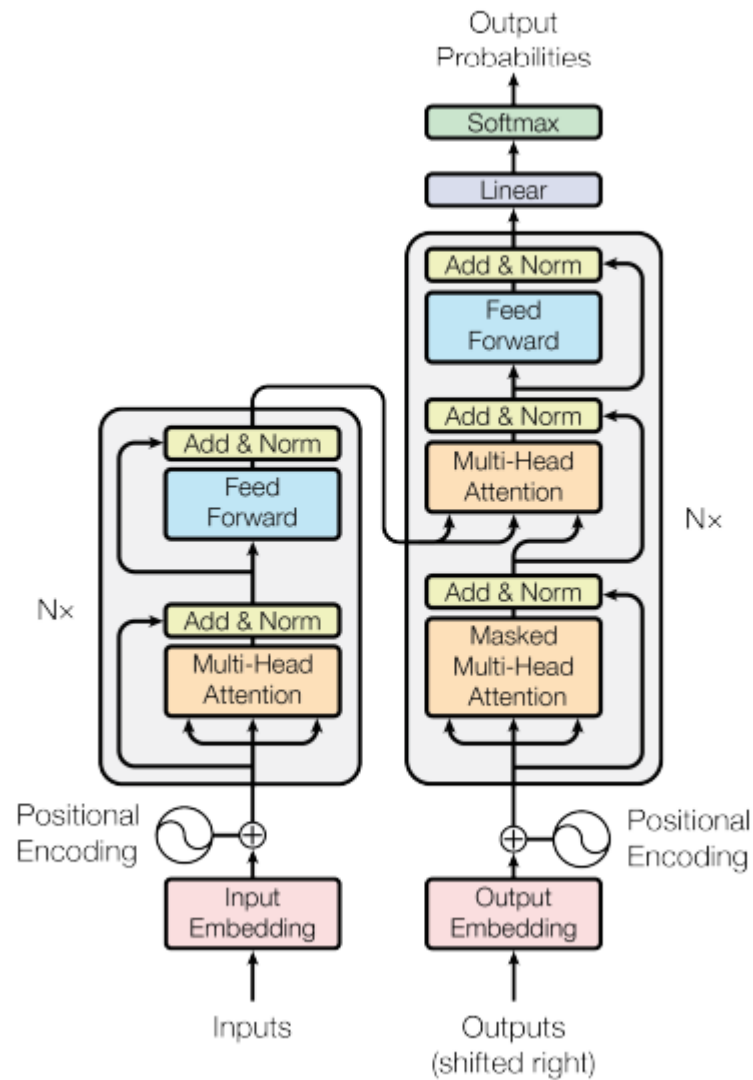
Introduction

Transformer 이전의 SOTA : ~~Encoder-Decoder~~ + **Encoder-Decoder** + **Attention**

“more parallelization and can reach a new state of the art”

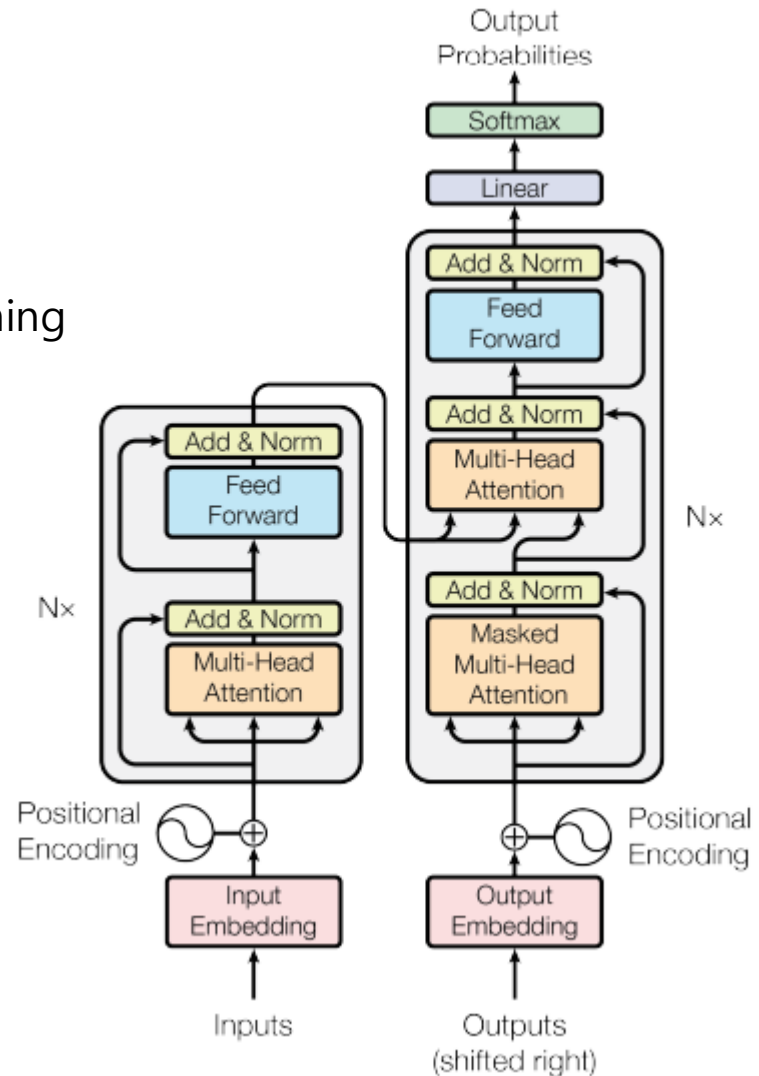
Overall Architecture

Architecture



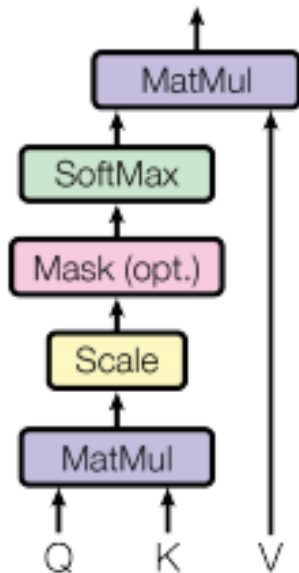
Architecture

1. Encoder(x6) - Decoder(x6)
2. Sublayer + Residual Connection
3. Multi-head Attention + FFN
4. Layer Norm + Dropout + Label Smoothing
5. Positional Encoding
6. Embedding



Attention in Transformer

Scaled Dot-Product Attention



query key value

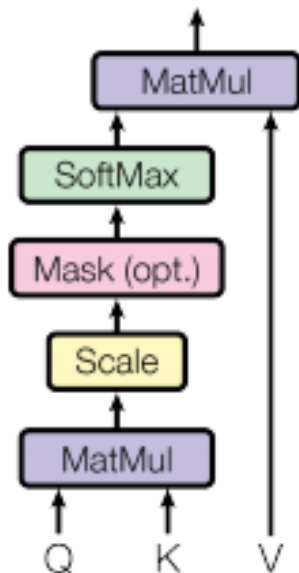
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

d_k = key / value vector의 dimension

특정 단어(query)가 어떤 단어(key)와 관련되어 있는지 찾은 후, 그 중요도를 다시 그 단어(value)에 곱함
즉, query가 어떤 key와 얼마나 높은 확률로 연관성이 있는지 계산하여 다시 value에 곱해주는 것

이때, d_k 가 커질수록 내적 값이 커지므로 softmax의 값이 편중될 위험이 있으므로 scaling을 진행

Scaled Dot-Product Attention



query key value

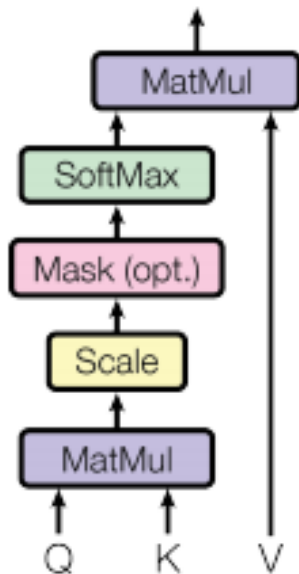
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

d_k = key / value vector의 dimension

특정 단어(query)가 어떤 단어(key)와 관련되어 있는지 찾는 후, 그 중요도를 다시 그 단어(value)에 곱함
즉, query가 어떤 key와 얼마나 높은 확률로 연관성이 있는지 계산하여 다시 value에 곱해주는 것

이때, d_k 가 커질수록 내적 값이 커지므로 softmax의 값이 편중될 위험이 있으므로 scaling을 진행

Scaled Dot-Product Attention



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

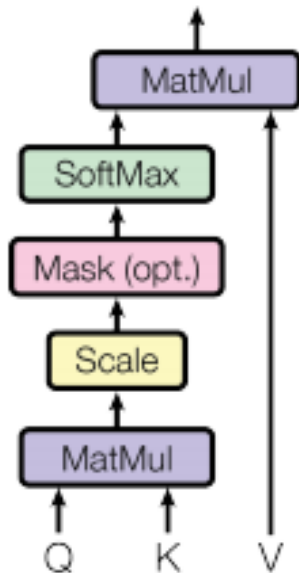
query key value

d_k = key / value vector의 dimension

특정 단어(query)가 어떤 단어(key)와 관련되어 있는지 찾은 후, 그 중요도를 다시 그 단어(value)에 곱함
즉, query가 어떤 key와 얼마나 높은 확률로 연관성이 있는지 계산하여 다시 value에 곱해주는 것

이때, d_k 가 커질수록 내적 값이 커지므로 softmax의 값이 편중될 위험이 있으므로 scaling을 진행

Scaled Dot-Product Attention



query key value

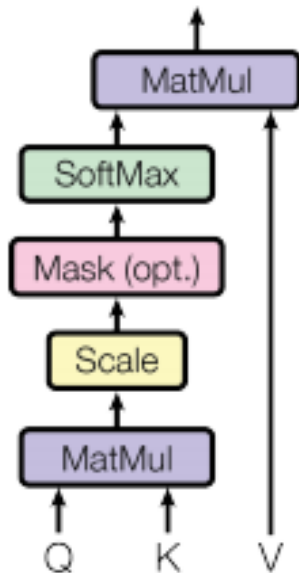
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

d_k = key / value vector의 dimension

특정 단어(query)가 어떤 단어(key)와 관련되어 있는지 찾은 후, 그 중요도를 다시 그 단어(value)에 곱함
즉, query가 어떤 key와 얼마나 높은 확률로 연관성이 있는지 계산하여 다시 value에 곱해주는 것

이때, d_k 가 커질수록 내적 값이 커지므로 softmax의 값이 편중될 위험이 있으므로 scaling을 진행

Scaled Dot-Product Attention



query key value

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

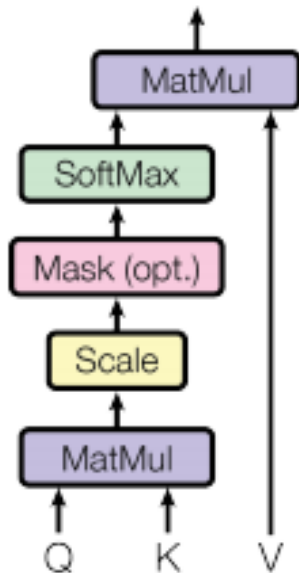
d_k = key / value vector의 dimension

같은 벡터?

특정 단어(query)가 어떤 단어(key)와 관련되어 있는지 찾은 후, 그 중요도를 다시 그 단어(value)에 곱함
즉, query가 어떤 key와 얼마나 높은 확률로 연관성이 있는지 계산하여 다시 value에 곱해주는 것

이때, d_k 가 커질수록 내적 값이 커지므로 softmax의 값이 편중될 위험이 있으므로 scaling을 진행

Scaled Dot-Product Attention



query key value

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

d_k = key / value vector dimension

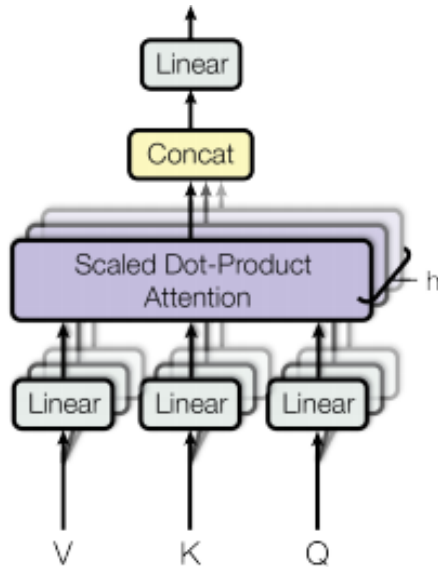
같은 벡터? \longrightarrow No!

Weight가 곱해지기 때문

특정 단어(query)가 어떤 단어(key)와 관련되어 있는지 찾은 후, 그 중요도를 다시 그 단어(value)에 곱함
즉, query가 어떤 key와 얼마나 높은 확률로 연관성이 있는지 계산하여 다시 value에 곱해주는 것

이때, d_k 가 커질수록 내적 값이 커지므로 softmax의 값이 편중될 위험이 있으므로 scaling을 진행

Multi-head Attention



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

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

$$d_k = d_v = d_{\text{model}}/h = 64. \quad h = 8$$

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

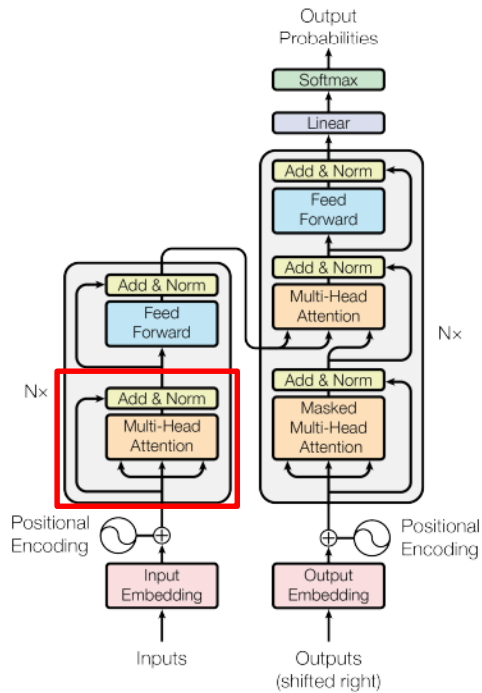
$$W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$$

query, key, value를 그냥 사용하는 것이 아니라, 각각 h=8번 linear projection 따라서, 서로 다른 representation으로부터 attention을 계산

linear projection을 h=8번 해주기 때문에 연산비용이 더 요구될 것 같지만,
linear projection을 통해 차원을 줄여주므로 비슷

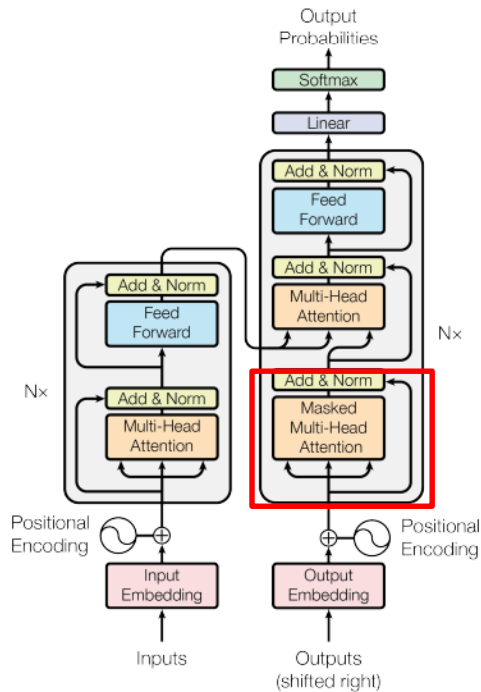
Detailed Architecture

Self-Attention



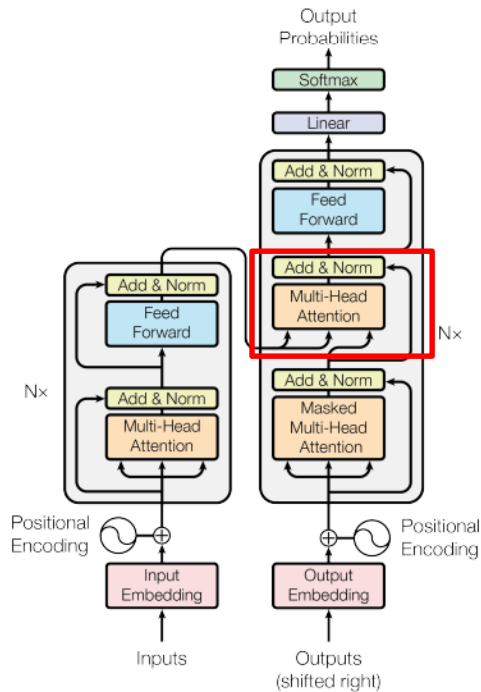
query, key, value를 처음에는 input에서, 이후에는 previous layer에서 가져옴
즉, $Q=K=V$

Masked Self-Attention



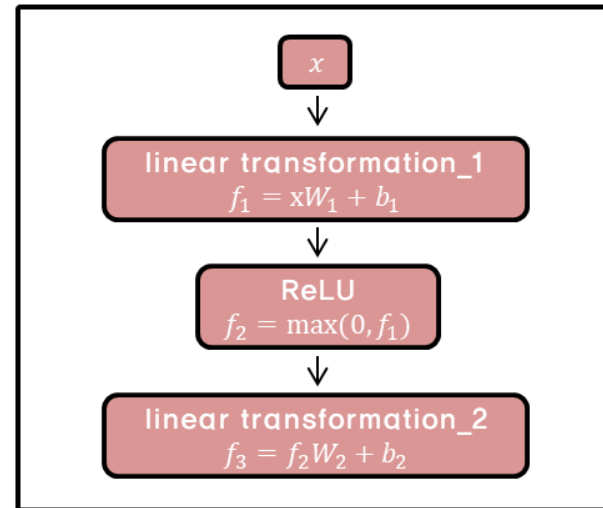
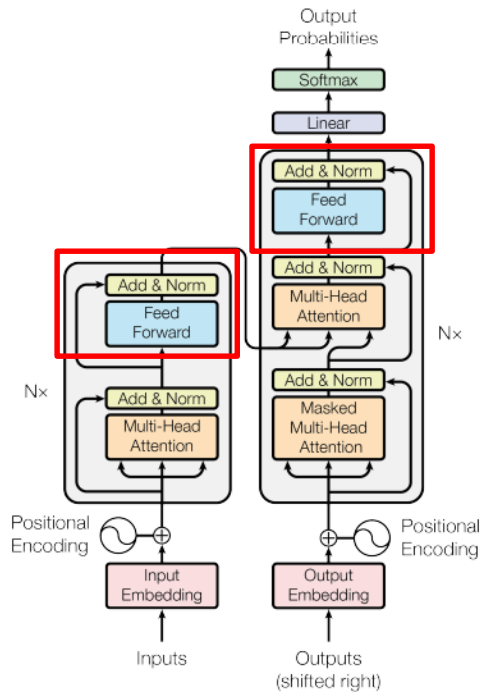
auto-regressive한 모델이기 때문에 현재 decoding하는 position 이전의 생성된 단어들만 사용 (softmax에 들어가는 matrix에서 masking해야 하는 position들을 모두 $-\infty$ 로 설정)

Encoder-Decoder Attention



query Q가 decoder에서 오고, key K와 value V는 encoder에서 오기 때문에
decoder의 모든 단어가 encoder의 모든 단어에 attention을 계산

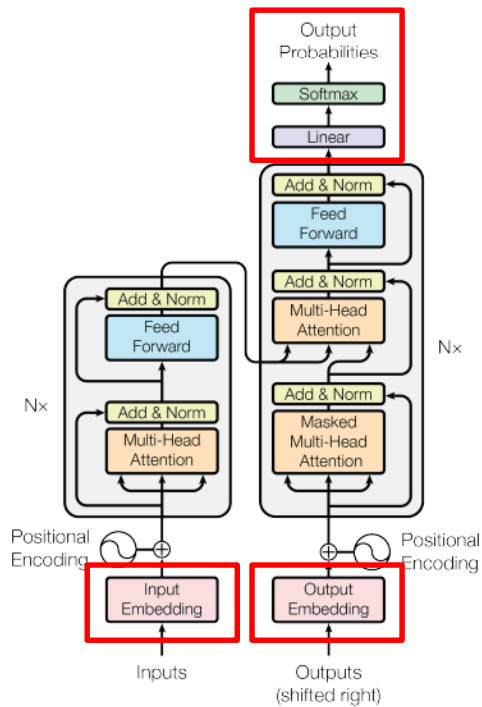
Position-wise Feed Forward Network



position마다 다른 network 사용
1x1 convolution이랑 비슷한 느낌

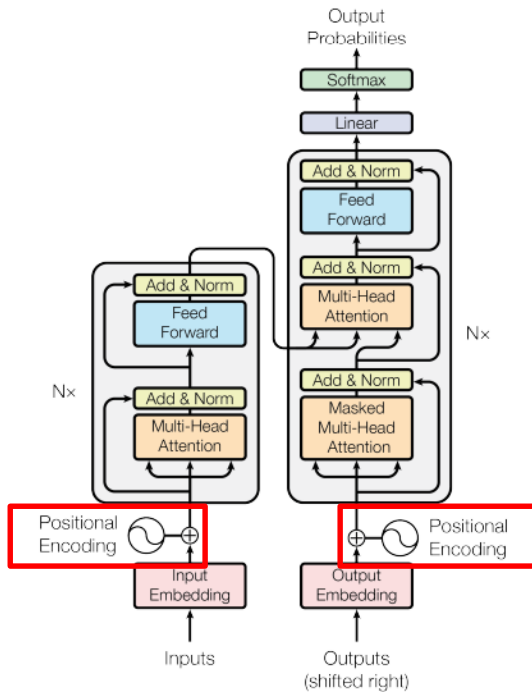
즉, "나는 개발자가 될래요" 라는 문장에서 ["나는", "개발자가", "될래요"] 가 각각 다른 network에 의해 계산
같은 position의 weight는 공유

Learned Embedding



고정된 embedding을 사용하는 것이 아니라, embedding weight도 계속 학습
(모두 공유됨)

Positional Encoding



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

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

$$PE_{pos} = [\cos(pos/1), \sin(pos/10000^{2/d_{model}}), \cos(pos/10000^{2/d_{model}}), \dots, \sin(pos/10000)]$$

Transformer는 attention만 사용하기 때문에 위치정보가 결여되어 있다는 문제가 발생
따라서, embedding vector에 positional vector를 더해주는 방식으로 위치정보를 반영

d_{model} 차원의 vector에서 i-th elt의 pos-vec의 값이 position (단어의 위치)마다 달라짐

Training

Training

- Adam Optimizer

$$lr_{rate} = d_{model}^{-0.5} \cdot \min(step_num^{-0.5}, step_num \cdot warmup_steps^{-1.5})$$

- Layer Normalization
- Dropout
- Label Smoothing

Result

SOTA 다.

SOTA 였다.

SOTA 였었다.

SOTA 였었었었었다.

Reference

- <https://arxiv.org/abs/1706.03762>
- <https://pozialabs.github.io/transformer/>
- <https://reniew.github.io/43/>
- <https://hwkim94.github.io/>