Transformer

박지훈, 김성환, 은하영

목차

1. Transformer 소개
2. Input
3. Encoder
    a. Encoder Block
        i. Attention
            1. Self Dot-Product Attention Layer
            2. Multi-Head Attention Layer
        ii. Position Wise Feed Forward Layer
        iii. Residual Connection Layer
4. Decoder
    a. Decoder Block
        i. input
        ii. self Multi-Head Attention Layer
        iii. Cross Multi-Head Attention Layer
        iv. Position Wise Feed Forward Layer
5. After Decoder(Generator)
    a. log_softmax()

Transformer

- Transformer는 구글이 2017년 발표한 “Attention is all you need”에서 나온 모델로 기존의 seq2seq의 구조인 인코더-디코더를 따르면서, **Attention만을 이용**한 모델.
- RNN을 사용하지 않고, 인코더-디코더를 설계했지만, 기존 RNN보다 성능이 우수하다.
  - 기존 seq2seq의 한계
    - 1. 병렬처리 불가능
    - RNN은 GPU의 발달에 따른 혜택을 받지 못함. (병렬처리 불가능)
    - 앞에서 부터 순차적으로 context벡터를 만들어내기 때문에, 이전시점에서 완료되지 않으면, 뒤의 연산을 수행할 수 없음. (병렬처리 불가능 이유)
- Transformer
  - Attention을 이용해서, 특정 시점에서 정보를 집중했고,
  - **Positional Encoding**을 사용해 sequential한 위치정보를 보존,
  - **이전 시점에 masking을 사용**해서 이전시점의 값만 이후 시점 결과에 영향을 제한함.
  - 또한 **모든 과정을 병렬처리**로 구현

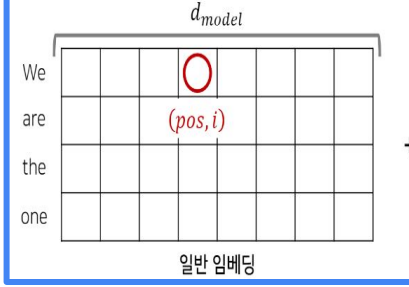
Transformer 전체적인 구조 및 input

RNN에서의 전시점이 끝나고 그다음 시점이 진행되어야 하는게 아니기 때문에, input에서 sentence뿐만이 아니라, 그 순서를 담을 수 있게끔 Positional Encoding을 진행해야한다.

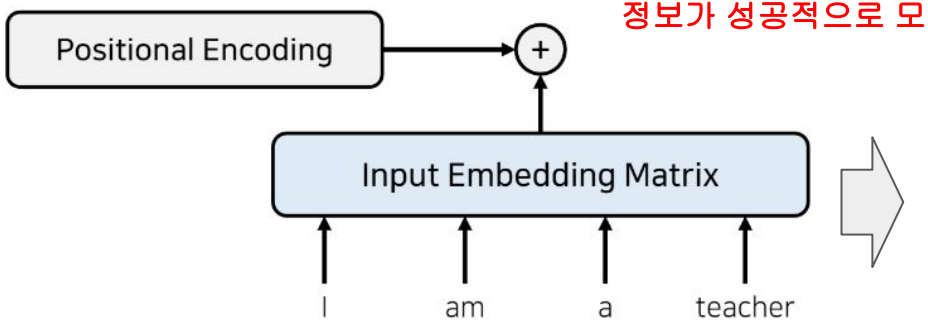
각 단어의 상대적인 위치 정보를 네트워크에 입력.

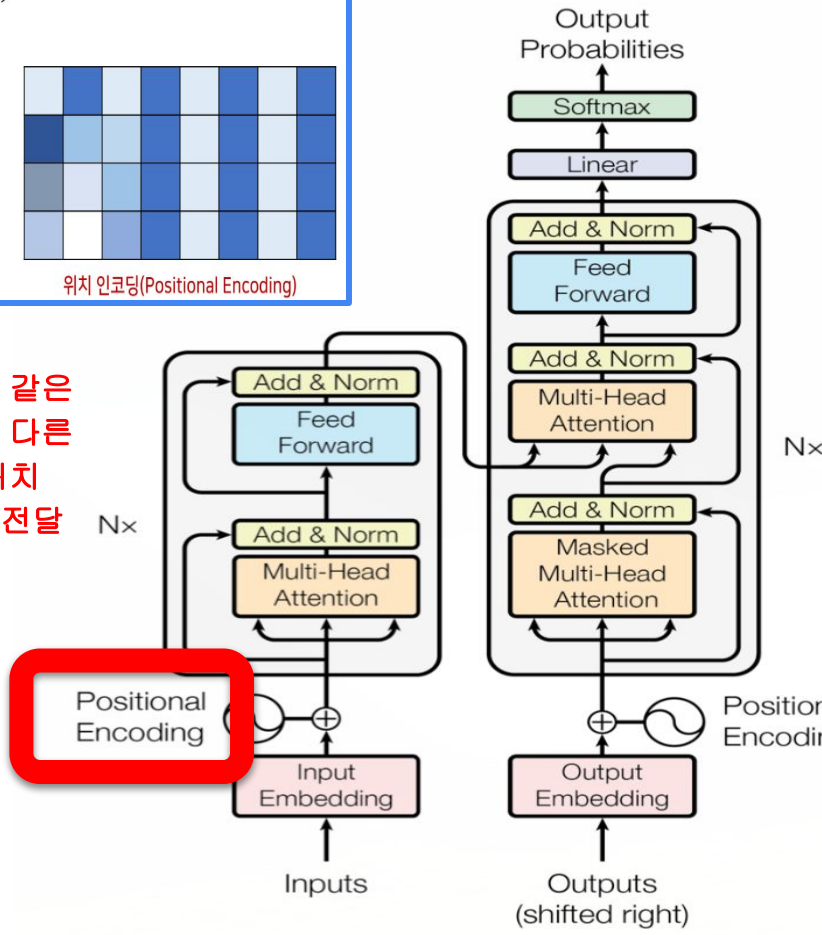
$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

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



positional encoding 덕분에 **같은 단어일 지라도 위치에 따라 다른 임베딩 벡터를 갖게 되어 위치 정보가 성공적으로 모델에 전달**



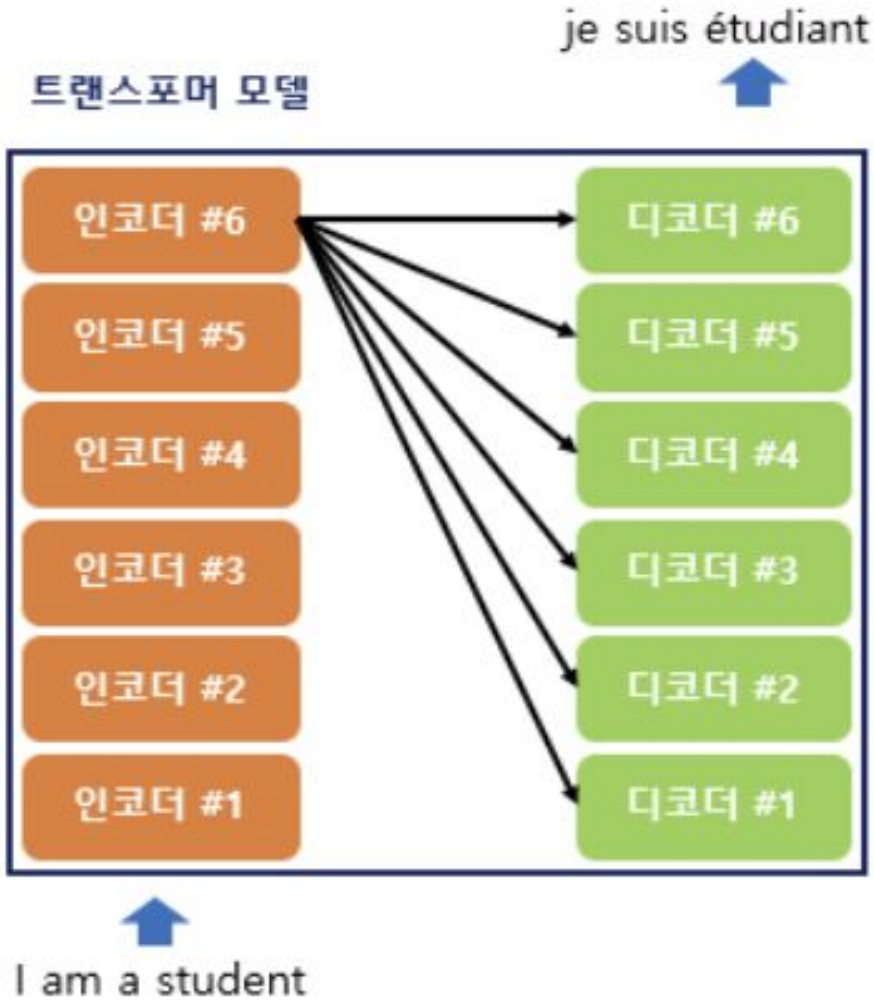


Transformer

마지막 인코더의 output이 각각의 디코더의 input으로 들어간다.

각각의 Encoder와 Decoder를 뜯어보고,

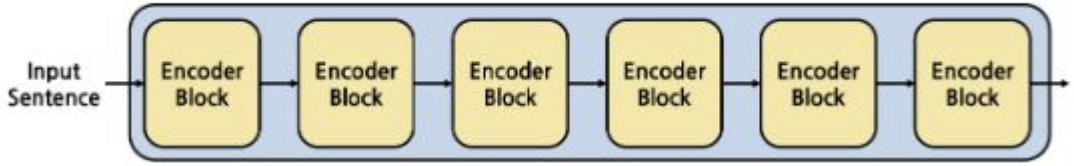
그 안에서 사용되는 Block들의 Layer들을 해석하자.



Encoder

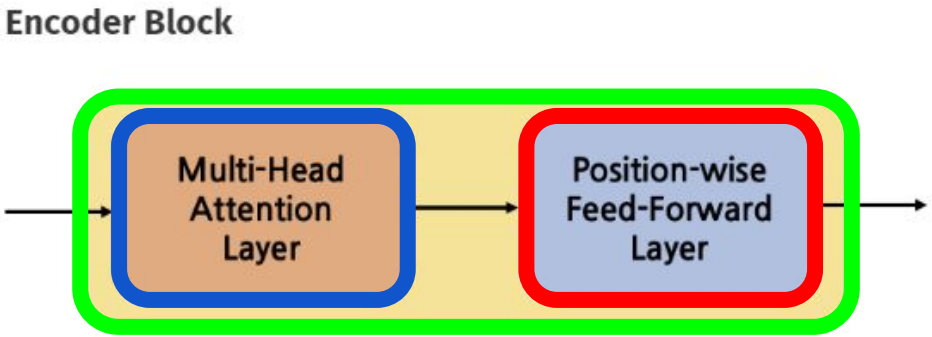
각 Encoder를 뜯어보면 여러개의 Encoder Block으로 이루어져 있다.

각 Encoder를 지날때마다, 더 높은 차원의 의미를 담는 context vector를 생성한다.



각각의 Encoder Block들은 2개(+1)의 층으로 이루어져 있다.

- **Multi-Head Attention Layer**
- **Position - wise Feed-Forward Layer**
- **Residual Connection Layer**



What is Attention?

Multi-Head Attention은 Scaled Dot-Product Attention을 병렬적으로 여러개 수행한 층이다.

Scaled Dot-Product Attention이 무엇인지 알아보자.

예제 문장

“The animal didn’t cross the street, because it was too tired”

RNN의 경우, 시간이 진행될수록 오랜된 시점의 token에 대한 정보가 점차 희미해져간다.

예제 문장에서 ‘didn’t’의 시점에서 hidden state를 구한다고 하면, 바로 직전의 token인 ‘animal’에 대한 정보는 뚜렷하게 남아있지만, 점차 앞으로 나아갈수록 ‘becuase’나 ‘it’의 시점에서는 ‘didn’t’시점보다는 ‘animal’에 대한 정보가 희미하게 남게 된다. **결국 서로 거리가 먼 token사이의 관계에 대한 정보가 제대로 반영되지 못한다.**

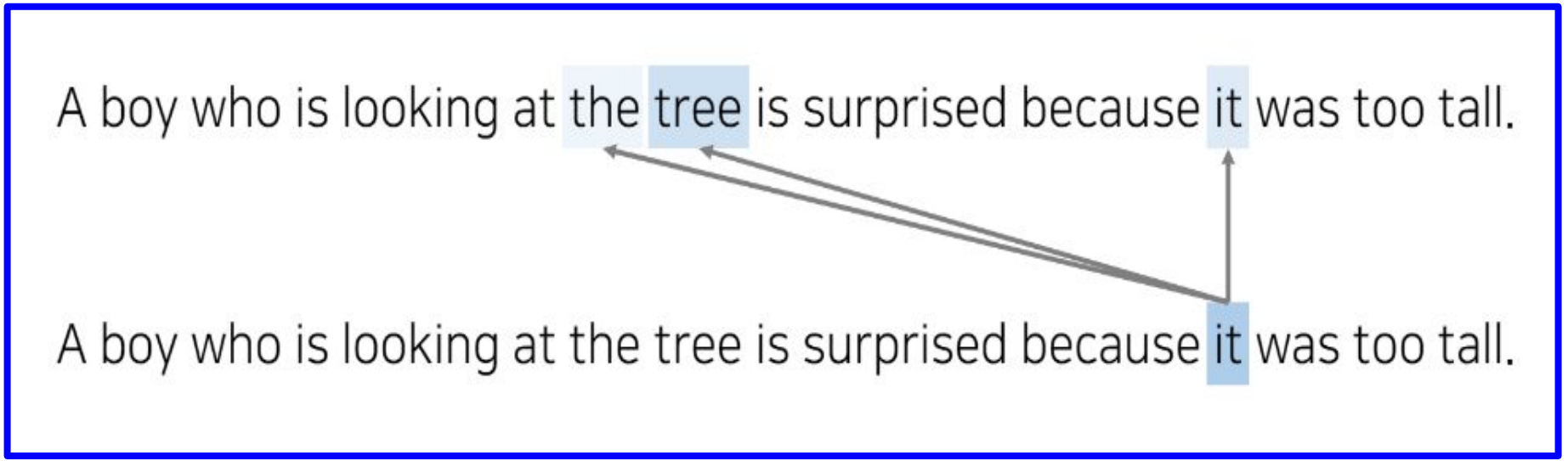
반면 Attention은 문장에 token이 n개 있다고 가정할 경우, nxn번 연산을 수행해 모든 token들 사이의 관계를 직접 구해낸다.

중간의 다른 token들을 거치지 않고 바로 direct한 관계를 구하는 것이기 때문에 RNN에 비해 더 명확한 관계를 잡아낼수 있다.

Self-Attention 추가설명

Self -Attention은 인코더, 디코더에서 모두 사용되는 Attention이다.

- 매번 입력문장에서 **각 단어가 다른 어떤 단어와 연관성이 높은지 계산 할 수 있습니다.**



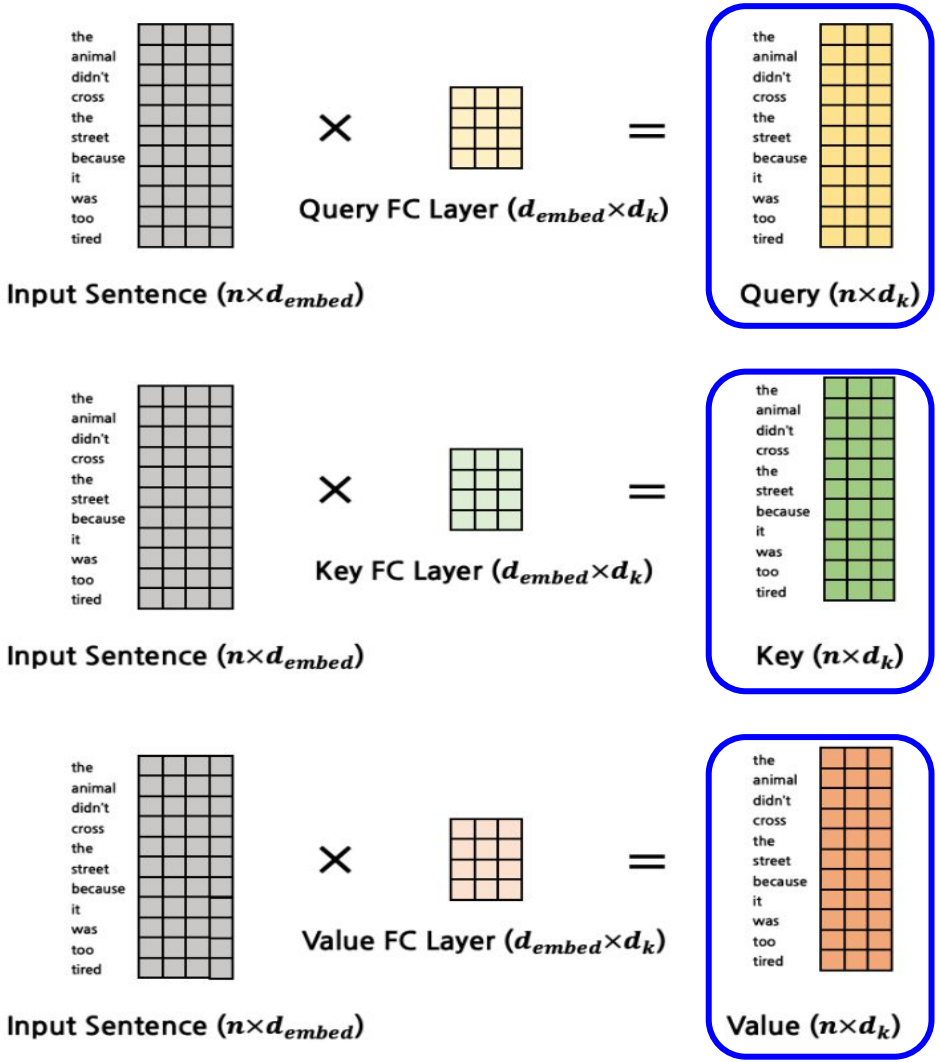
Attention(1-Q,K,V)

어텐션을 위한 3가지 입력요소

1. 쿼리(Query)
   - 구하고자 하는 목적token
2. 키(Key)
   - Attention을 구하고자 하는 대상

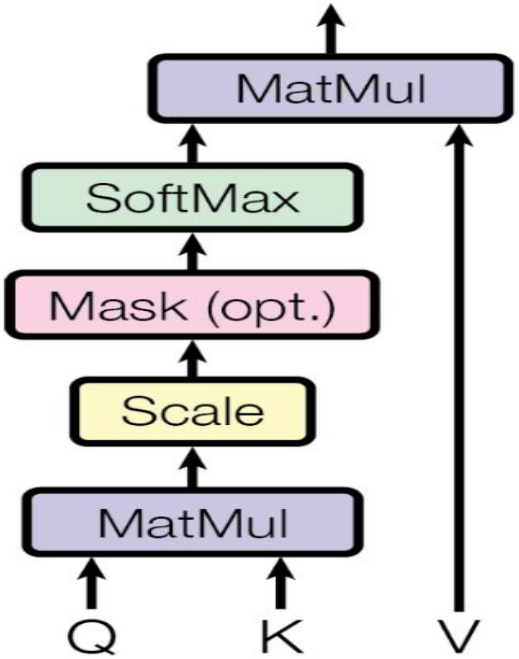
     token을 의미
3. 값(Value)
   - 키와 동일(다른 값)

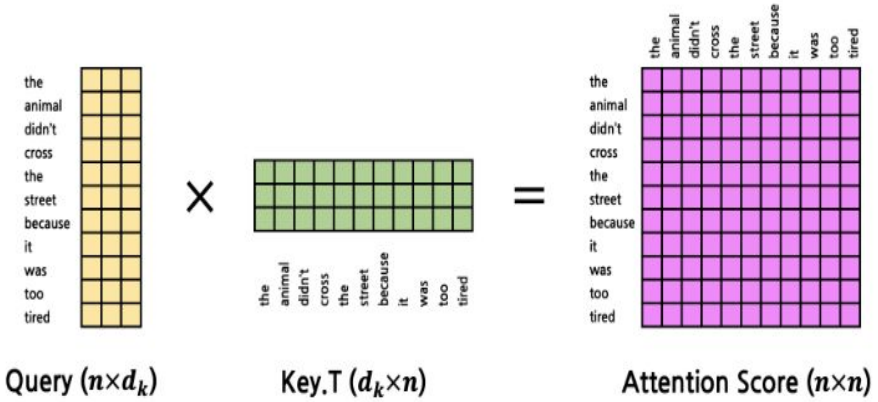
각각 FClayer이 1개씩 필요함.

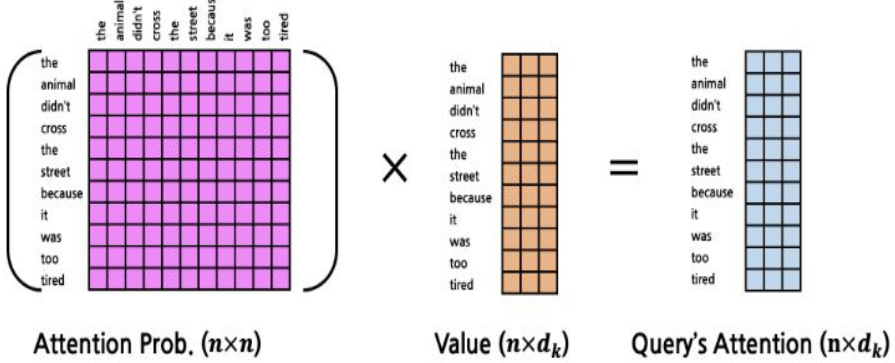


Attention(2-Self Dot-Product Attention)

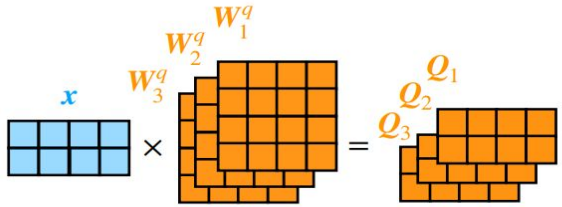
$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

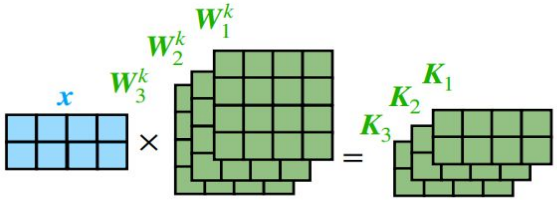


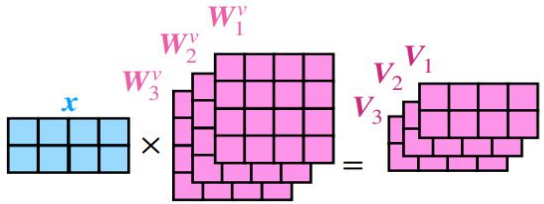


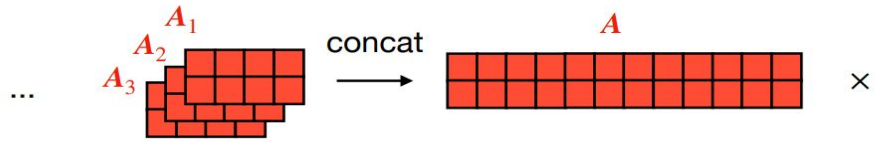


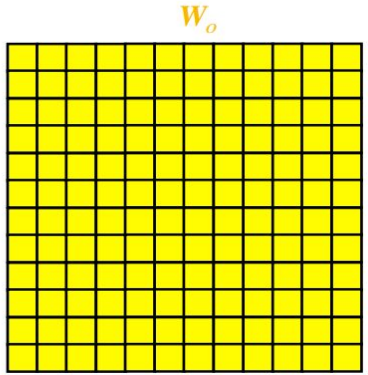
Attention(3-Multi-Head Attention)

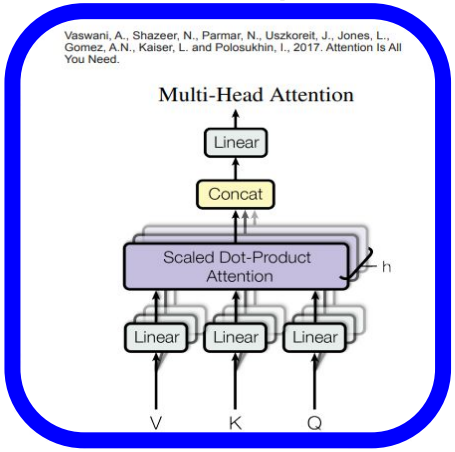












$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

$$MultiHead(Q,K,V) = Concat(head_1, \ldots, head_h)W^O$$

다양한 Attention을 반영하기 위해서 우리는 Encoder에서 1회씩 Attention을 계산하는 것이 아니라, h회씩 묶어서 한번의 Scaled Dot-Product Attention을 진행한다.

논문에선 h를 8회라고 정했지만, 우리는 이해를 돕기 위해 간단하게 3회로 지정해서 살펴보자.

Position wise Feed-forward Layer

Encoder Block안에서 Multi -Head Attention의 output을 Position Wise Feed-Forward Layer의 input으로 받습니다.

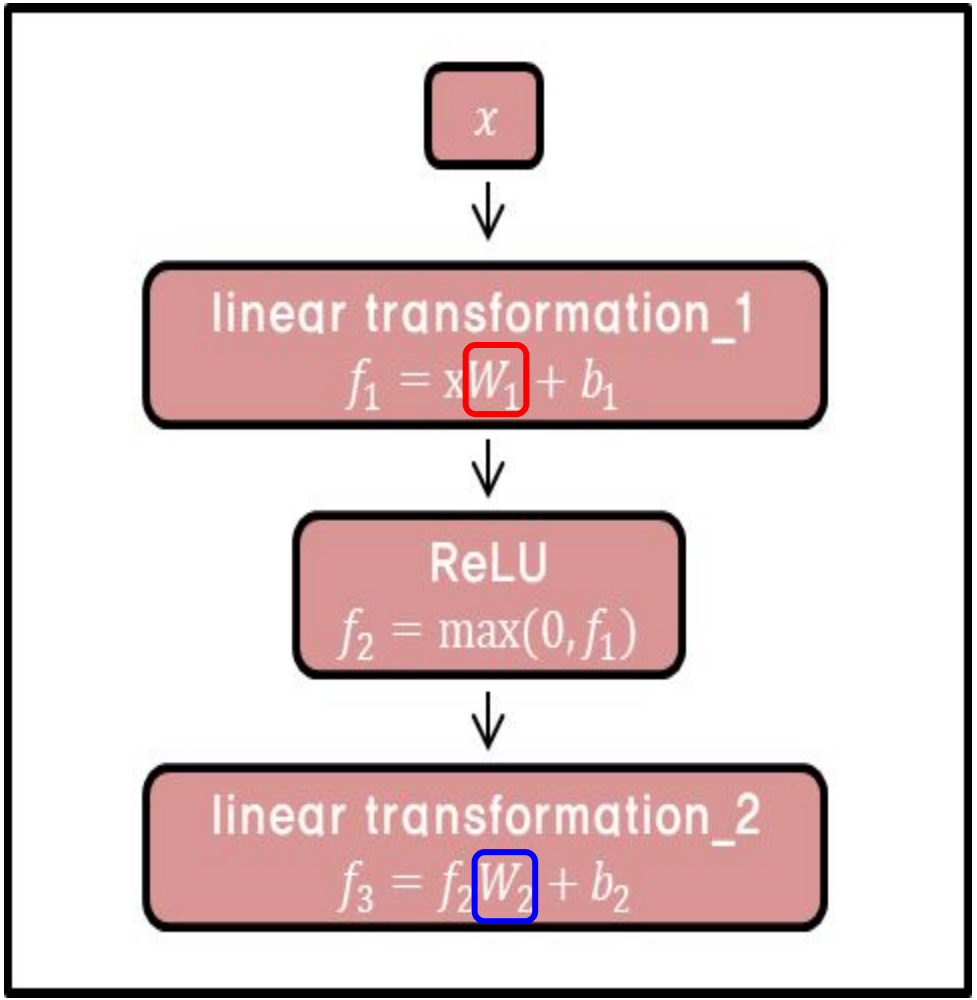
이 층의 output은 다음 Encoder Block의 Multi-Head Attention의 input으로 들어갑니다.

단순하게 2개의 FC Layer를 갖는 Layer이다. 각 FC Layer는 **(d_embed x d_ff)**, **(d_ff x d_embed)**의 wieght matrix를 갖는다.

멱등한 shape을 유지한채로 넘겨줘야하기 때문에 wieght matrix의 크기는 정해져 있다.

논문에 따르면, 첫번째 FCLayer의 output에 **ReLU를** 받습니다.

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



Residual Connection Layer

위에서 Muliti-Head Attention Layer과 Position Wise Feed-forward Layer 다뤘다.

이 2개의 층은 Residual Connection Layer로 둘러싸여 있다.

이 층은 정말 단순하다.

y = f(x)를 y = f(x) + x 로 변경하는 것이다.

즉, output을 그대로 사용하지 않고, output에 추가로 input을 추가적으로 더한 값이다.

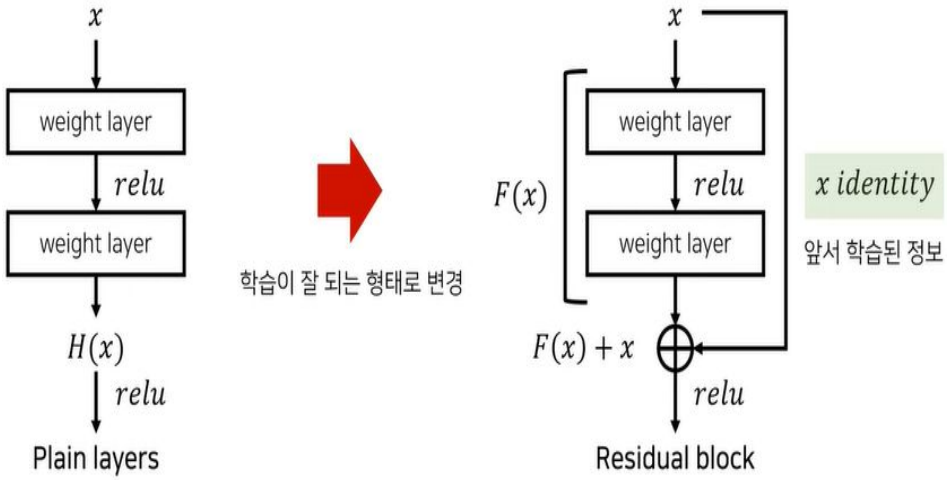
개념적으로 논문에 따르면, 역전파 중 발생할 수 있는 기울기 소실을 방지할 수 있다.

여기에 보통 Normalization과 DropOut까지 추가하지만, 본 세미나에선 생략한다.

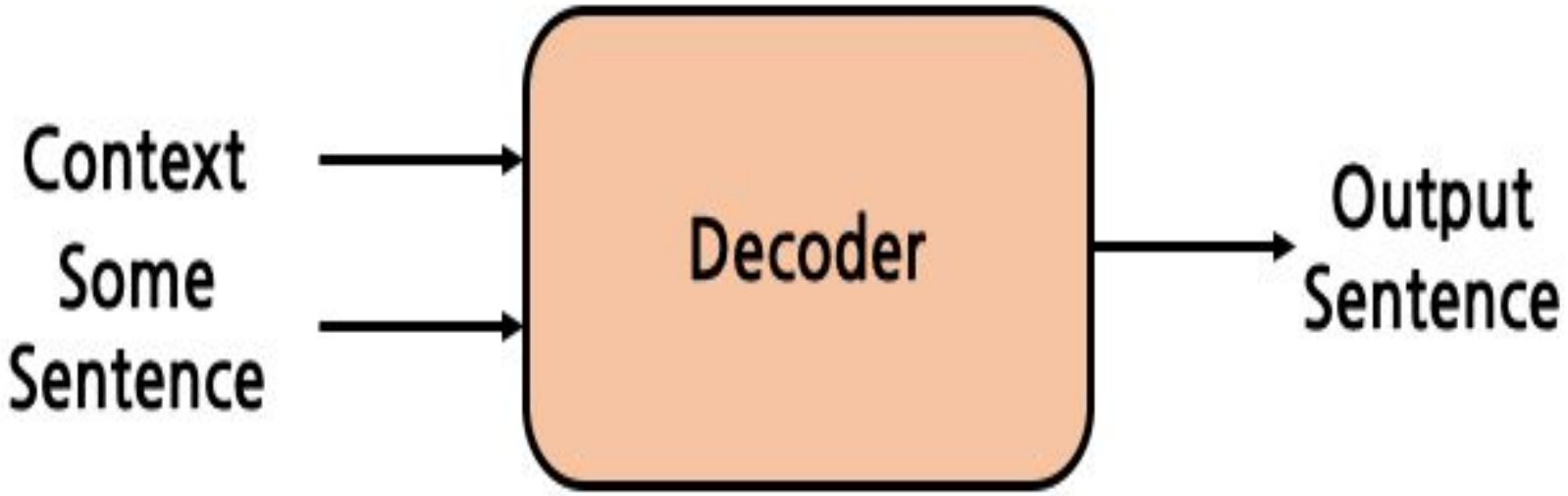
RC층은 CV분야에서 많이 사용되는 기법이다.

https://daeun-computer-uneasy.tistory.com/28

포스팅에서 CV분야에 대한 자세한 설명과 함께 추가적인 내용을 다룬다.



Decoder



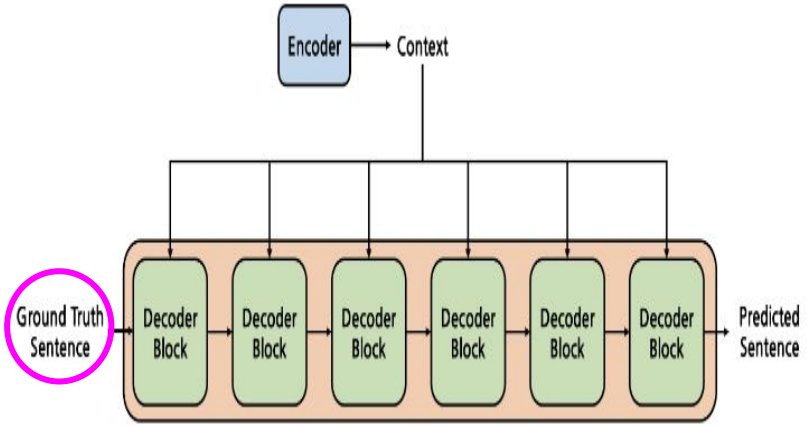
$$y = \mathrm{Decoder}(c, z)$$

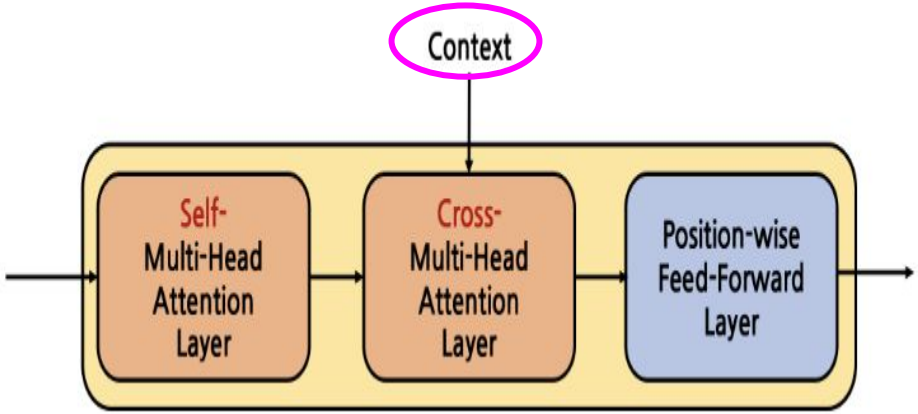
$y,\ z$: sentence

c : context

Decoder의 구성과 Input

Decoder도 Encoder과 마찬가지로,
Decoder Block으로 이루어져 있다.





각각의 Decoder Block은

1. Self Multi-Head Attention Layer
2. Cross-Multi-Head Attention Layer
3. Position-wise Feed-Forward Layer

로 구성되어 있다.

Decoder Block Input : Ground Truth Sentence, Context

Self(Masked)-Multi-Head Attention Layer

한→영번역기라고 가정할때, encoder에서 input값은 한국어에 대한 임베딩 정보였다면, decoder에선 이 **정보가 영어에 대한 임베딩** 정보로 바뀐다.

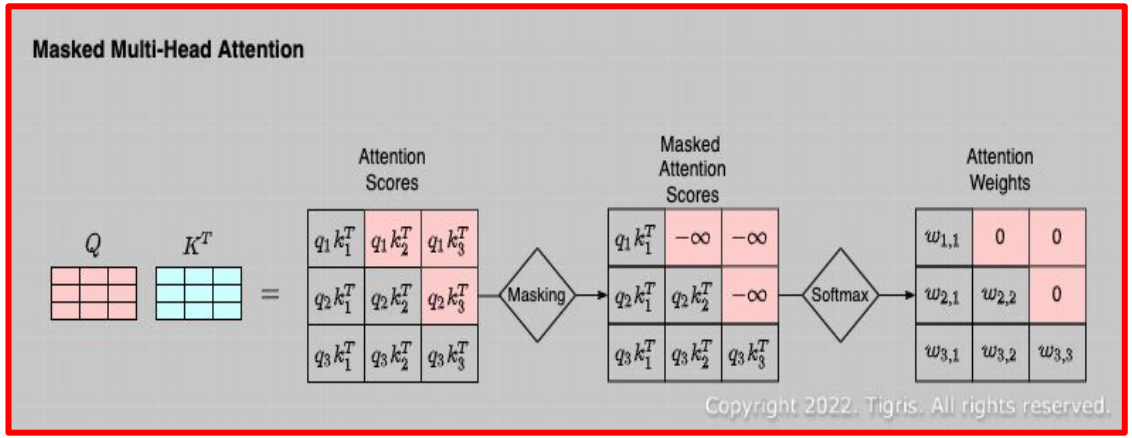
Encoder에서 사용했던, Multi-Head Attention과 input값이 제외하곤 크게 다르지 않다.

이 층의 이름을 생각하면서, 어떤 층인지 생각하면 쉽다.

Decoder의 input으로 주어지는 **Sentence내부**에서의 Attention을 계산한다.

또한 우리가 여태까지 다루지 않았던 **Masked**란 개념을 사용한다.

기존의 Encoder-Decoder구조의 모델들은 순차적으로 입력값을 전달받기 때문에 t+1시점의 예측을 위해 사용할 수 있는 데이터가 t시점까지로 한정됩니다.



하지만 Transformer의 경우 전체 입력값을 전달받기 때문에 과거 시점의 **입력값을 예측할 때 미래 시점의 입력값까지 참고할 수 있다는 문제**가 있습니다.
이러한 문제를 방지하기 위해 나온 기법을 **Look-Ahead(Subsequent Masking) Mask**라고 하고, 이것을 이용한 Attention을 **Masked-Attention**이라고 합니다.

Teacher Forcing in Transformer

Cross-Multi-Head Attention Layer

Decoder에서 가장 핵심적인 부분이다.

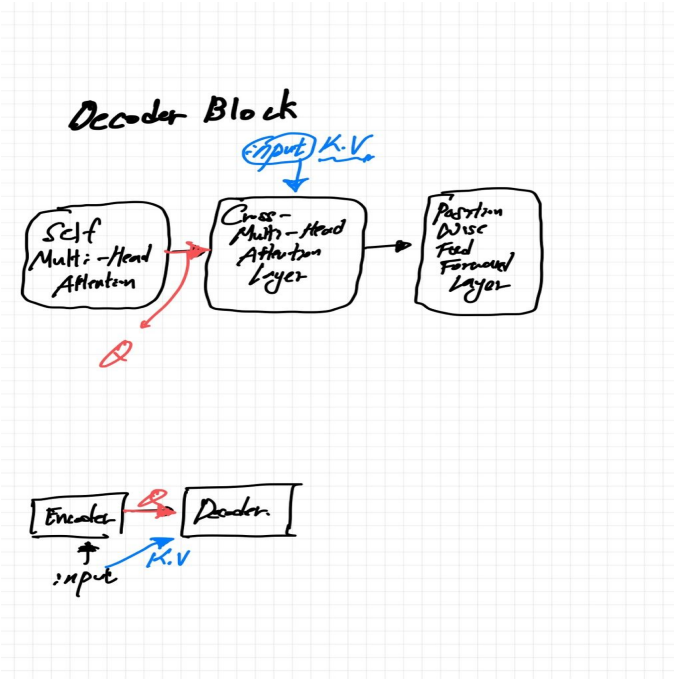
이 층에선 input값이 2개로 들어가는데

1. Self-Muti-Head Attention Layer에서 넘어온 output
2. Encoder에서 도출된 Context도 input으로 받는다.

이에 1번을 Query로, 2번을 Key, Value로 받아서 Multi-Head Attention을 계산한다.

★Cross Attention★

해석 : **Decoder에서 넘어온 input**의 **Encoder에서 넘어온 input에 대한** Attention계산



이후 Position Wise Feed-Forward Layer은 Encoder의 것과 동일하므로 생략.

After Decoder(Generator)

Decoder의 output이 그대로 Transformer의 최종 Output이 되는 것은 아니다.

Decoder의 output shape은 n_batch * seq_len * d_embed인데, 우리가 원하는 output은 target sentence인 n_batch * seq_len이다.

즉, Embedding이 아니라 실제 target vocab에서의 target sequence를 원하는 것이다.

이를 위해 추가적인 FC Layer를 거쳐간다.

이것은 대게 **Generator**이라고 부른다.

Generator

- **Decoder output의 마지막 dimension을 d_embed에서 len(vocab)으로 변경한다.**

- 이를 통해 실제 vocabulary 내 token에 대응시킬수 있는 shape이 된다.

- 이후 softmax()를 사용해 각 vocabulary에 대한 확률값으로 변환하게 되는데, 이 때 **log_softmax()**를 사용해 성능을 향상시킨다.

log_softmax()

log_softmax()는 소프트맥스에 log함수를 취한것으로 log(softmax()), softmax함수를 보완하는 역할을 한다.

딥러닝 학습시, softmax()를 사용하면 Vanishing **Gradients문제가 발생하기 쉬운데, 이 문제를 해결**하기 위해 사용하는 것이 LogSoftmax이다.

LogSoftmax를 사용하면, (*,/)를 (+,-)형식으로 변형할 수 있어 log-sum-exp trick이라고도 불린다.

이것은 **수식에 있어서 더욱 안정적**이다.

(numerically more stable)

$$\text{logsoftmax} = \log\left(\frac{e^{x_i}}{\sum_{j=1} e^{x_j}}\right)$$

$$= x_i - \log\left(\sum_{j=1} e^{x_j}\right)$$

감사합니다