Transformer

트랜스포머는 2017년 구글이 발표한 논문인 'Attention is all you need'에서 나온 모델로 기존의 seq2seq의 구조인 인코더-디코더를 따르면서도, 논문의 이름처럼 attention만으로 구현한 모델이다.

이 모델은 RNN을 사용하지 않고, 인코더-디코더 구조를 설계하였음에도 성능도 RNN보다 우수하다는 특징을 갖고 있다.

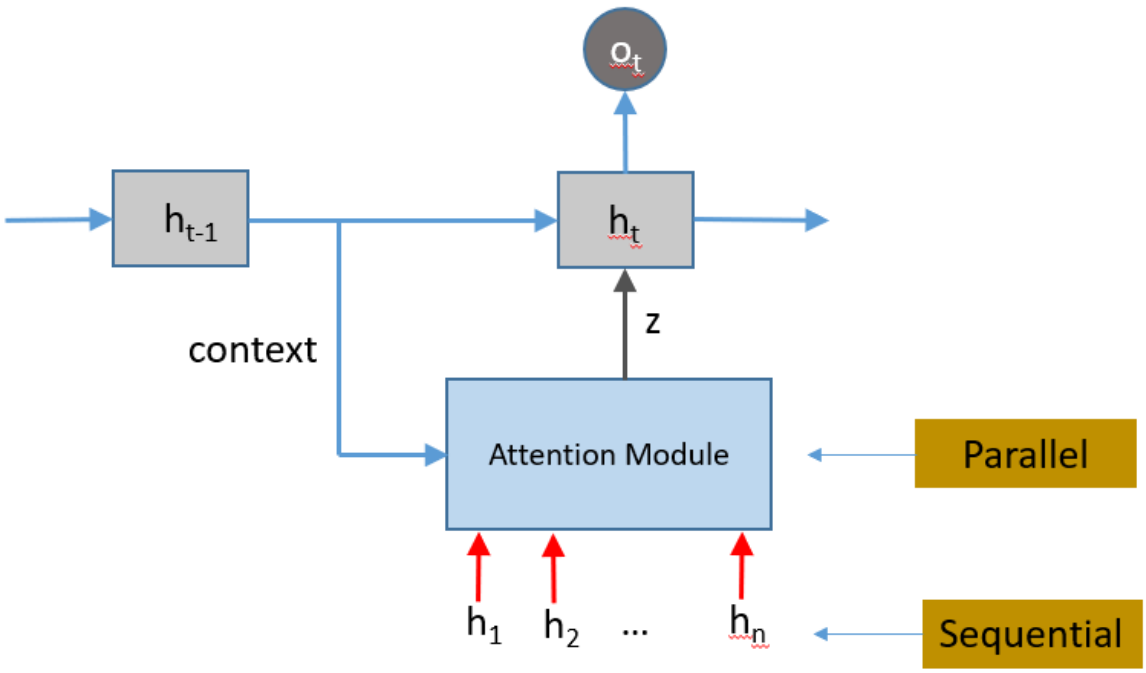
1. 기존의 seq2seq 모델의 한계

기존의 seq2seq 모델은 인코더-디코더 구조로 구성되어져 있다. 여기서 인코더는 입력 시퀀스를 하나의 벡터 표현으로 압축하고, 디코더는 이 벡터 표현을 통해서 출력 시퀀스를 만들어냈다. 하지만 이러한 구조는 인코더가 입력 시퀀스를 하나의 벡터로 압축하는 과정에서 입력 시퀀스의 정보가 일부 손실된다는 단점이 있었고, 이를 보정하기 위해 어텐션이 사용되었다. 그런데 어텐션을 RNN의 보정을 위한 용도가 아니라 아예 어텐션으로 인코더와 디코더를 만들어보면 어떨까?

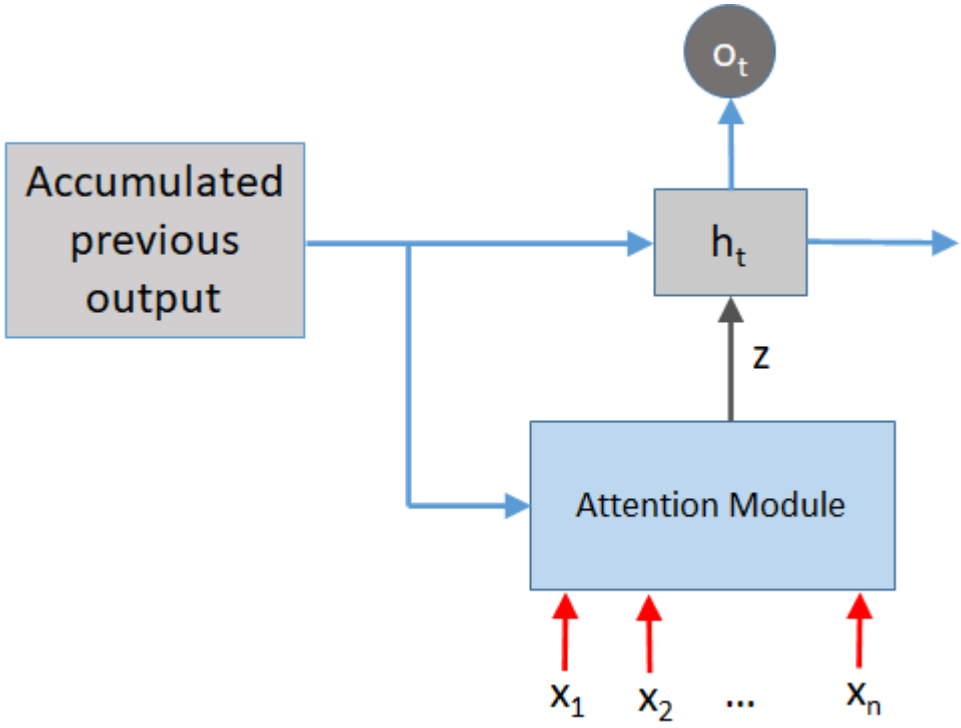
또한, Sequential computation prevents parallelization

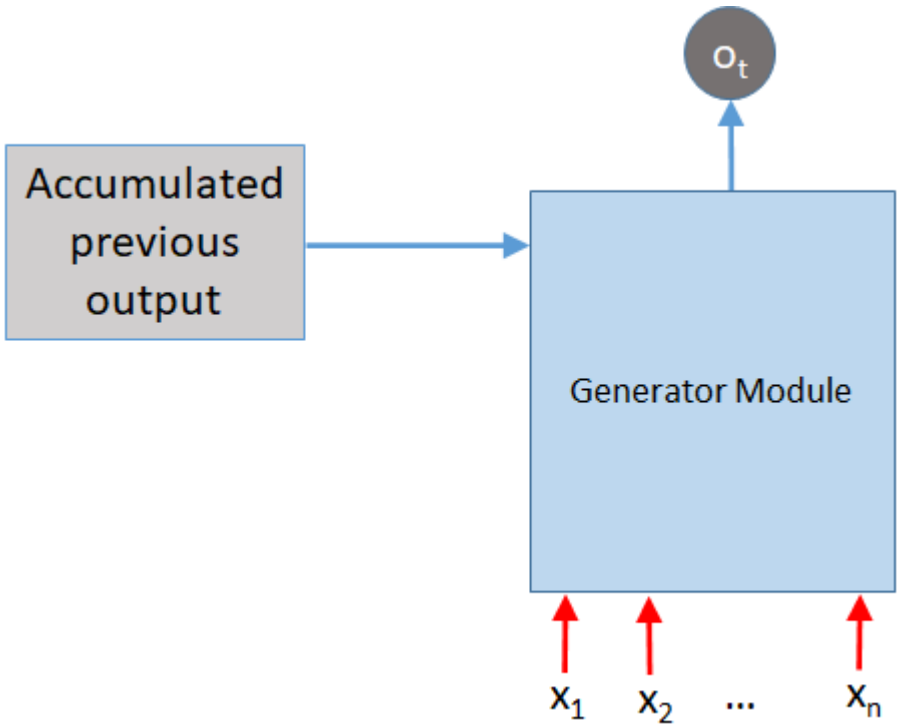
>>속도가 느려진다. Can we parallelize the encoding precess??

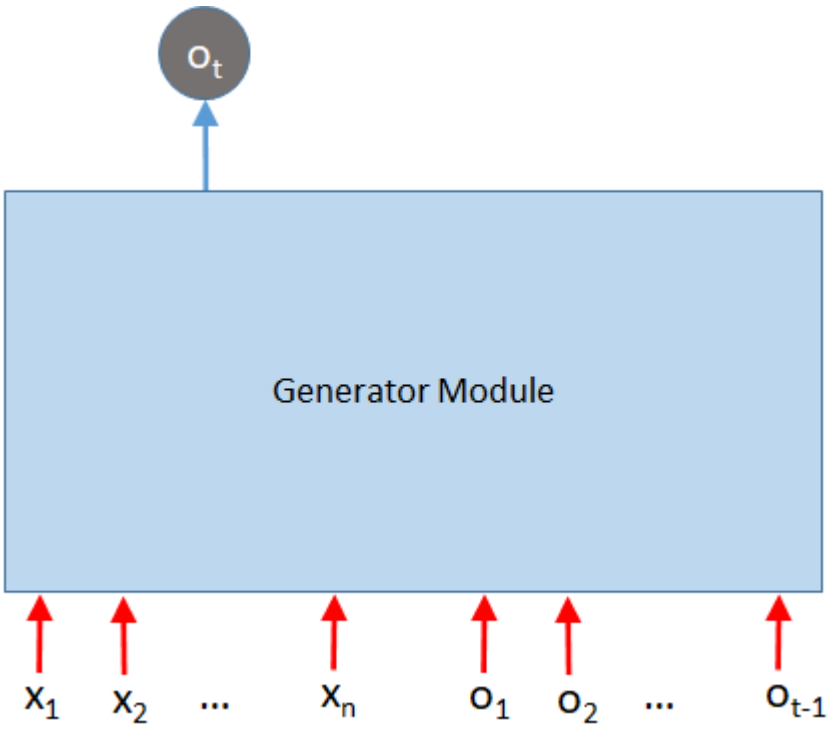
,



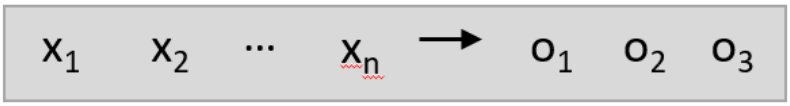
Another Viewpoint of Naive Attention Module

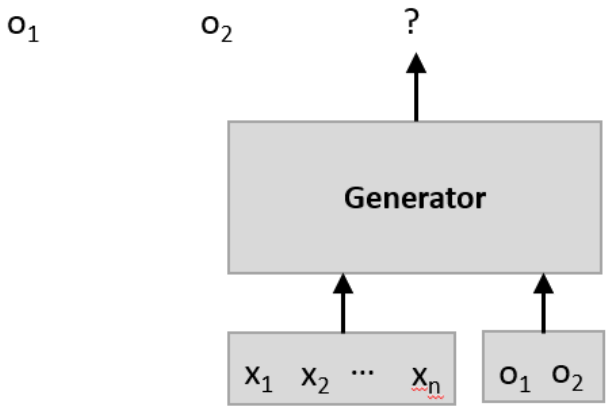




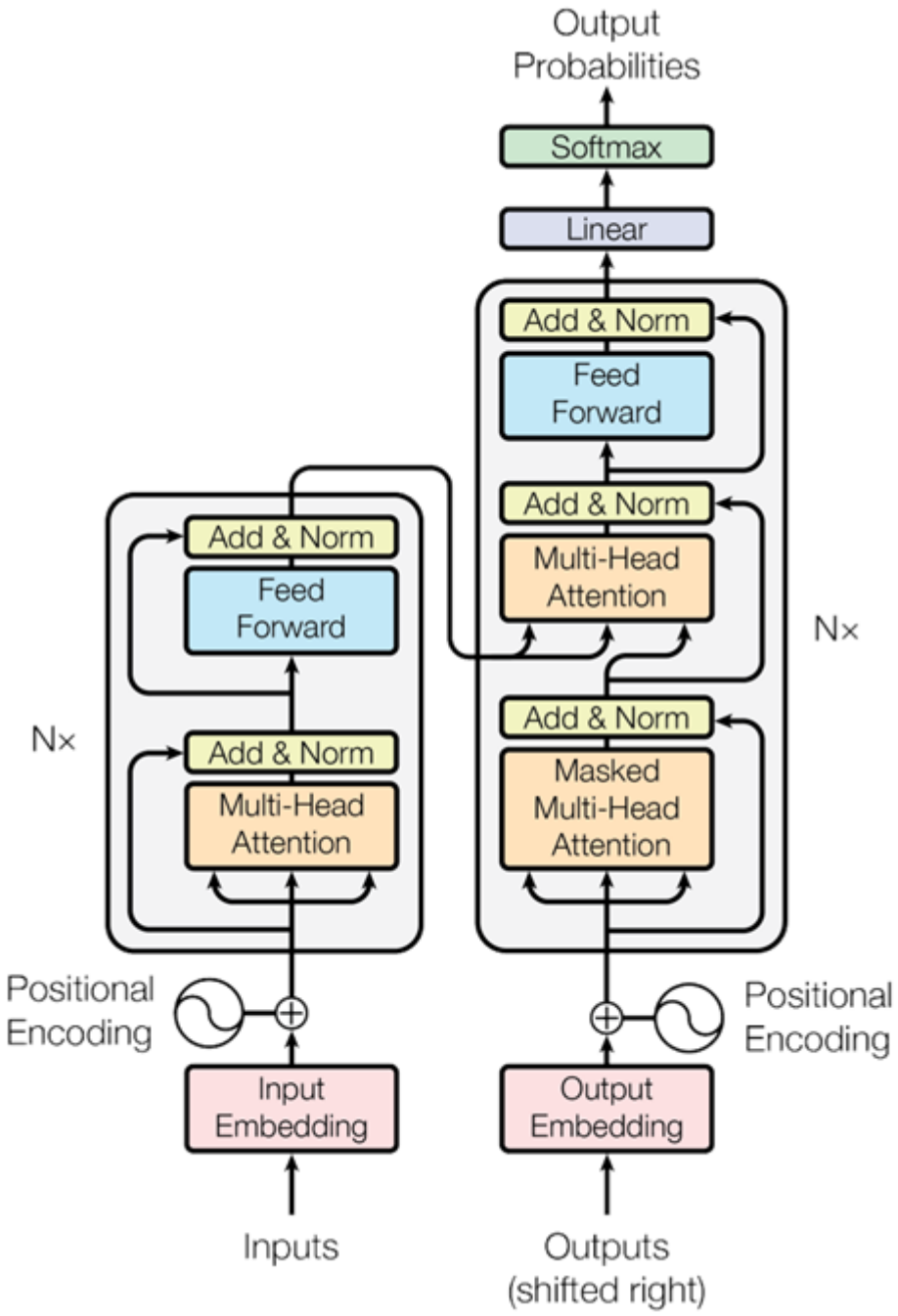


without Sequential Encoder & Decoder



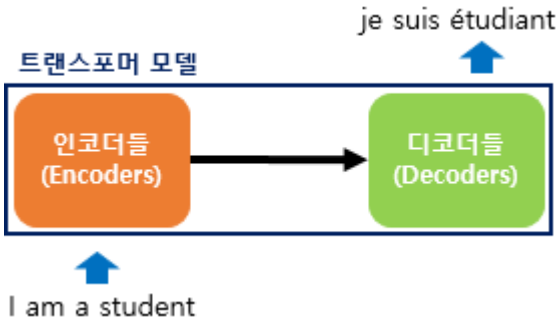


Transformer Overview



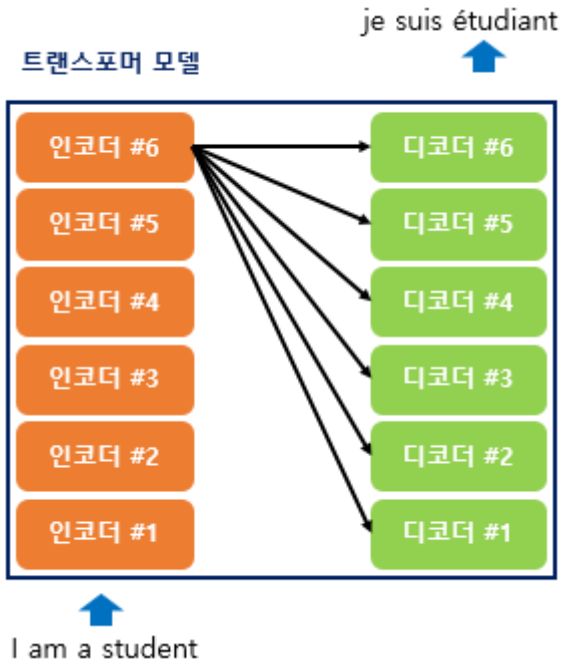
이그림의 왼쪽 오른쪽이 둘다 Generator이다.

- Encoder-Decoder approach
- Task: machine translation with parallel corpus
- Predict each translated word.
- Final cost/error function is standard cross-entropy error on top of a softmax classifier.

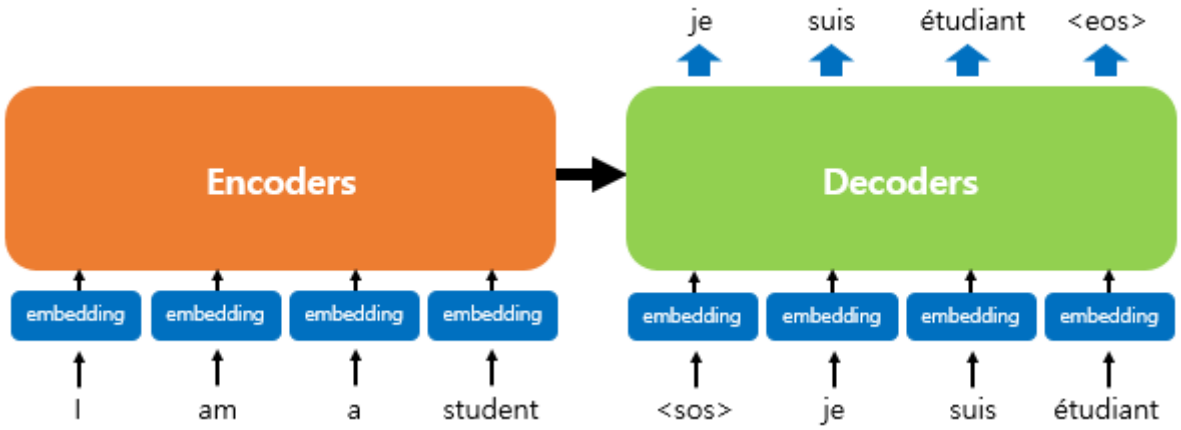


트랜스포머는 RNN을 사용하지 않지만 기존의 seq2seq처럼 인코더에서 입력 시퀀스를 입력받고, 디코더에서 출력 시퀀스를 출력하는 인코더-디코더 구조를 유지하고 있다. 다만 다른 점은 인코더와 디코더라는 단위가 N개가 존재할 수 있다는 점이다.

이전 seq2seq구조에서는 인코더와 디코더에서 각각 하나의 RNN이 t개의 시점(time-step)을 가지는 구조였다면 이번에는 인코더와 디코더라는 단위가 N개로 구성되는 구조이다. 트랜스포머를 제안한 논문에서는 인코더와 디코더의 개수를 각각 6개를 사용하였다.



위의 그림은 인코더와 디코더가 6개씩 존재하는 트랜스포머의 구조를 보여준다. 인코더와 디코더가 각각 여러 개 쌓여있다는 의미를 사용할 때는 알파벳 s를 뒤에 붙여 encoders, decoders라고 표현하겠다.

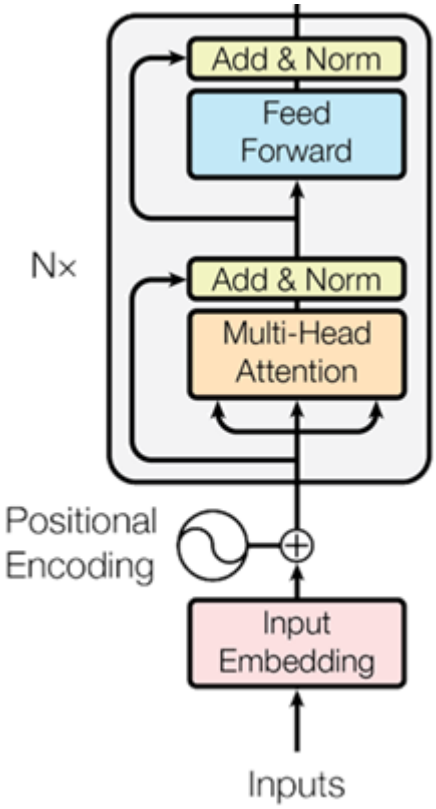


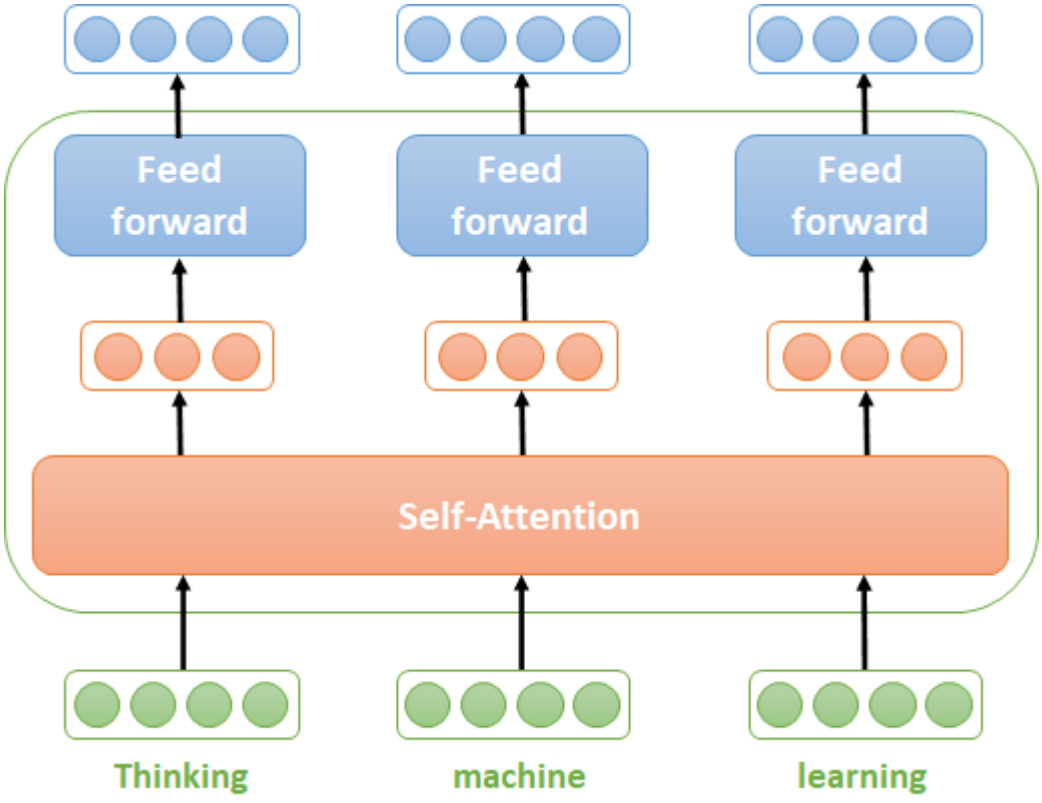
위의 그림은 인코더로부터 정보를 전달받아 디코더가 출력 결과를 만들어내는 트랜스포머 구조를 보여준다. 디코더는 마치 기존의 seq2seq 구조처럼 시작 심볼 <sos>를 입력으로 받아 종료 심볼 <eos>가 나올 때까지 연산을 진행한다. 이는 RNN은 사용되지 않지만 여전히 인코더-디코더의 구조는 유지되고 있음을 보여준다.

우선 인코더의 내부를 살펴보자!

Encoder Internals

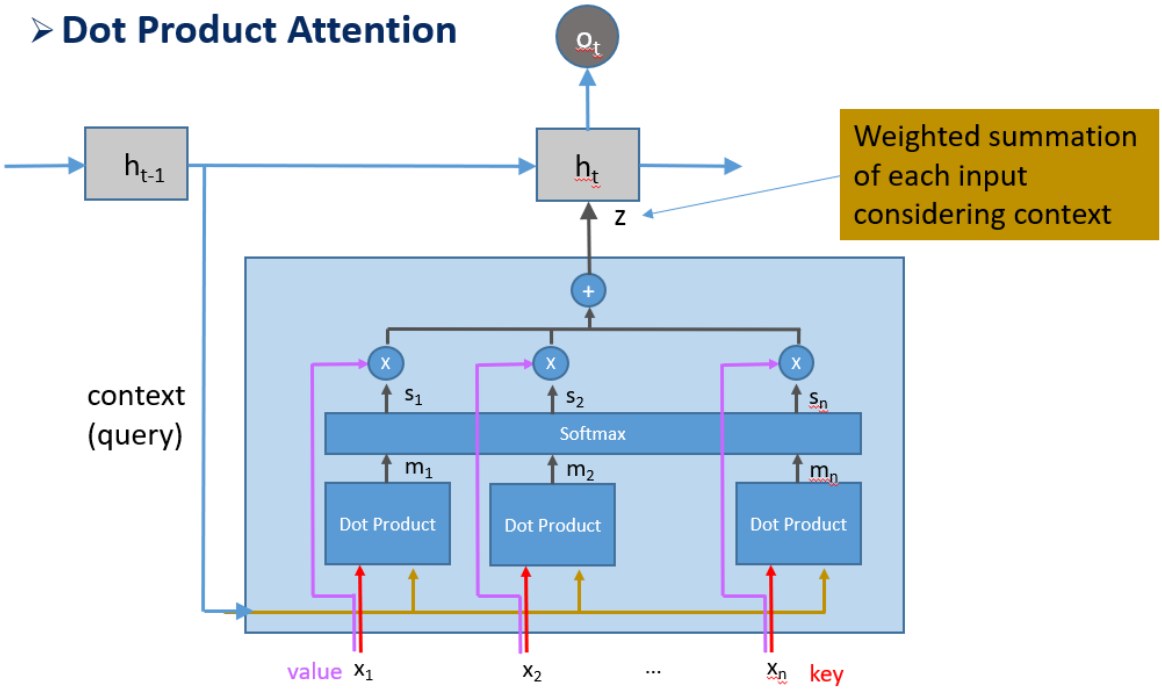
- After embedding the words in the input sentence, each of them flows through the two layers of the encoder.


Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Input Embedding
Inputs


Feed forward
Feed forward
Feed forward
Self-Attention
Thinking
machine
learning

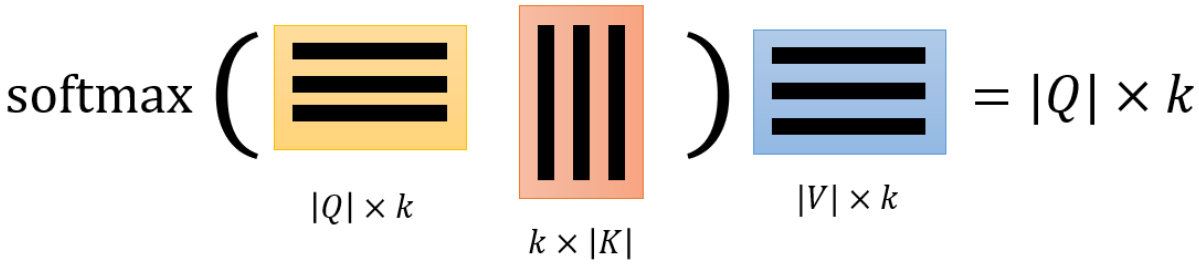
트랜스포머는 하이퍼파라미터인 num_layers 개수의 인코더 층을 쌓는다. 논문에서는 총6개의 인코더 층을 사용했다. 인코더를 하나의 층이라는 개념으로 생각한다면, 하난의 인코더 층은 크게 총 2개의 서브층으로 나뉘어진다. 바로 **셀프 어텐션**과 **피드 포워드 신경망**이다. 멀티 헤드 셀프 어텐션과 포지션 와이즈 피드 포워드 신경망이라고 적혀있지만, 멀티 헤드 셀프 어텐션은 셀프 어텐션을 병렬적으로 사용하였다는 의미고, 포지션 와이즈 피드 포워드 신경망은 우리가 알고있는 일반적인 피드 포워드 신경망이다.

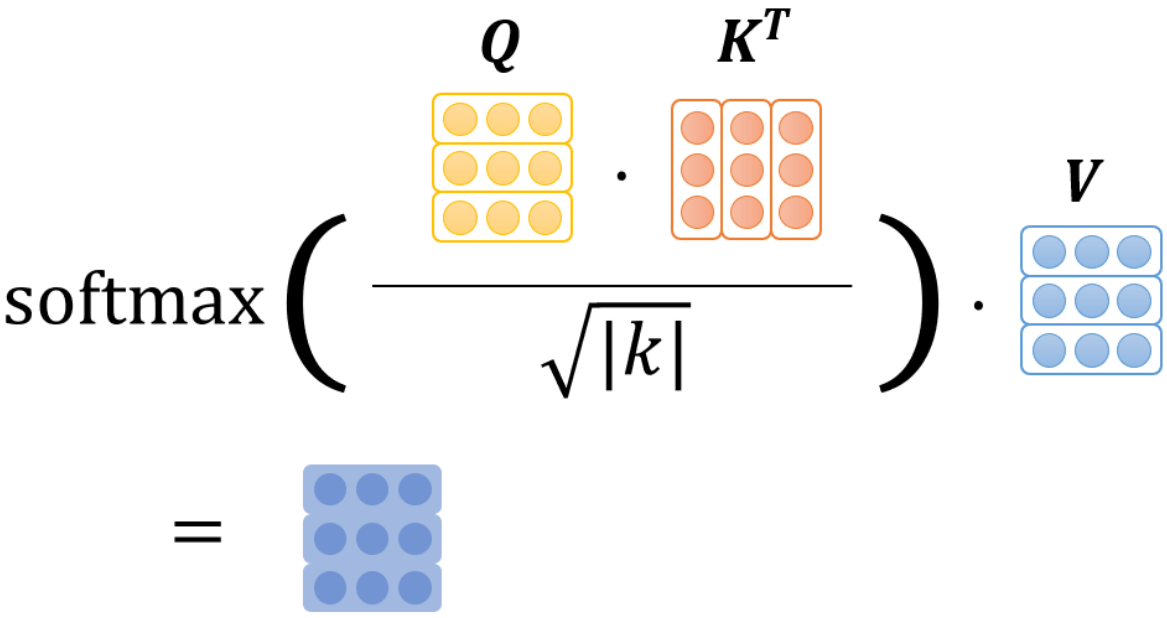
Attention Mechanism



- Using multiple queries, we stack them into a matrix Q.

$$A(Q,K,V) = softmax(QK^T)V$$





|K|는 임베딩 벡터의 차원이다. 그런데 만약 k절대값이 커지면 softmax결과 값이 크게 달라진다. 한쪽으로 치우침 . 그래서 루트 k로나눠서 스케일링해주는것이다.

As |K| gets large, the variance of QK^T increases.

→ Some values inside the softmax get large.

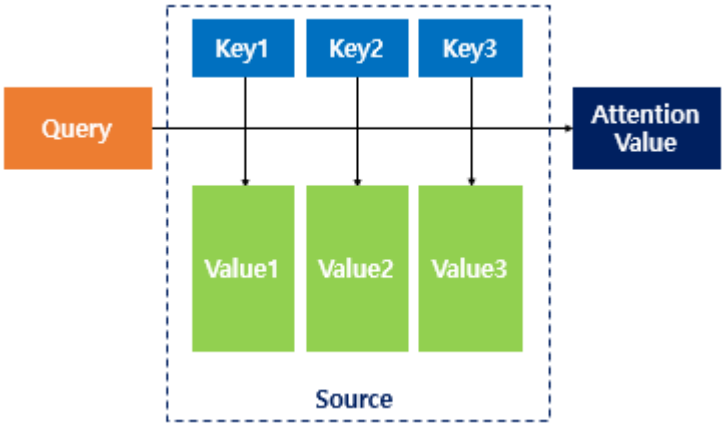
→ The softmax get very peaked.

→ Its gradient gets smaller.

Self-Attention

1. self -attention의 의미와 이점

어텐션 함수는 주어진 'query'에 대해서 모든 'key'와의 유사도를 각각 구한다. 그리고 구해낸 이 유사도를 가중치로 하여 키와 매핑되어있는 각각의 value에 반영해준다. 그리고 유사도가 반영된 value를 모두 가중합하여 리턴한다.



여기까지는 앞서 배운 어텐션의 개념이다. 그런데 어텐션 중에서는 셀프 어텐션(self-attention)이라는 것이 있다. 단지 어텐션을 자기 자신에게 수행한다는 의미이다. 앞서 배운 seq2seq에서 어텐션을 사용할 경우의 Q, K, V의 정의를 다시 생각해보자

Q = Query : t 시점의 디코더 셀에서의 은닉 상태
K = Keys : 모든 시점의 인코더 셀의 은닉 상태들
V = Values : 모든 시점의 인코더 셀의 은닉 상태들

그런데 사실 t 시점이라는 것은 계속 변화하면서 반복적으로 쿼리를 수행하므로 결국 전체 시점에 대해서 일반화를 할 수도 있다.

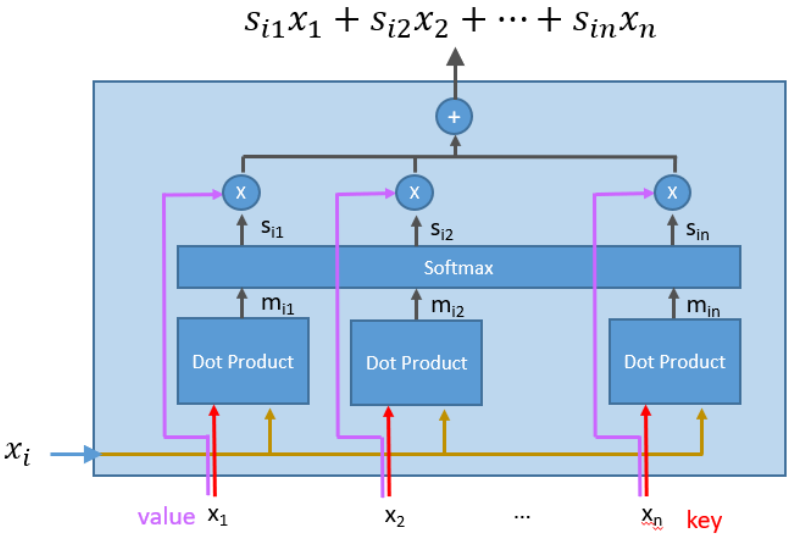
Q = Querys : 모든 시점의 디코더 셀에서의 은닉 상태들
K = Keys : 모든 시점의 인코더 셀의 은닉 상태들
V = Values : 모든 시점의 인코더 셀의 은닉 상태들

그런데 셀프 어텐션에서는 Q, K, V가 전부 동일하다. 트랜스포머의 셀프 어텐션에서의 Q, K, V는 아래와 같다. bn

Q : 입력 문장의 모든 단어 벡터들
K : 입력 문장의 모든 단어 벡터들

V : 입력 문장의 모든 단어 벡터들

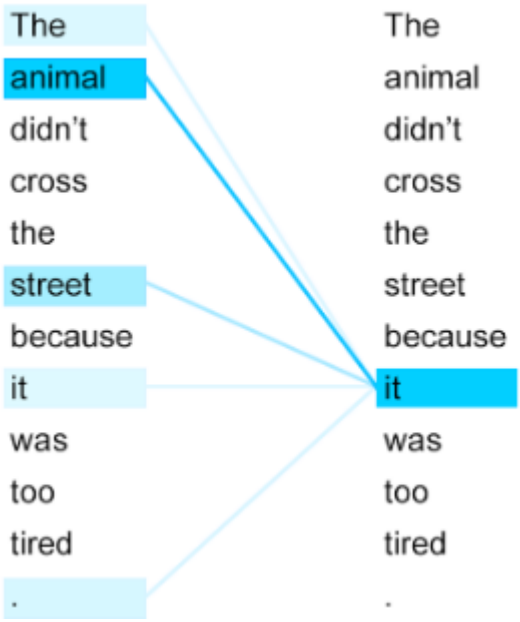
➢ **Definition:** $A(x_i, X, X)$



$$A(x_i, X, X) = s_{i1}x_1 + s_{i2}x_2 + \cdots + s_{in}x_n = x_i'$$

$$A(X, X, X) = \begin{pmatrix} s_{11}x_1 + s_{12}x_2 + \cdots + s_{1n}x_n \\ s_{21}x_1 + s_{22}x_2 + \cdots + s_{2n}x_n \\ \cdots \\ s_{n1}x_1 + s_{n2}x_2 + \cdots + s_{nn}x_n \end{pmatrix} = X'$$

그니까 이게 뭔뜻이면 query : xi 를 저렇게 재해석한건데 모든 input과의 유사도를 비교하여서 표현했다 생각할 수 있다.결국 다합치면 xi가 되기 때문



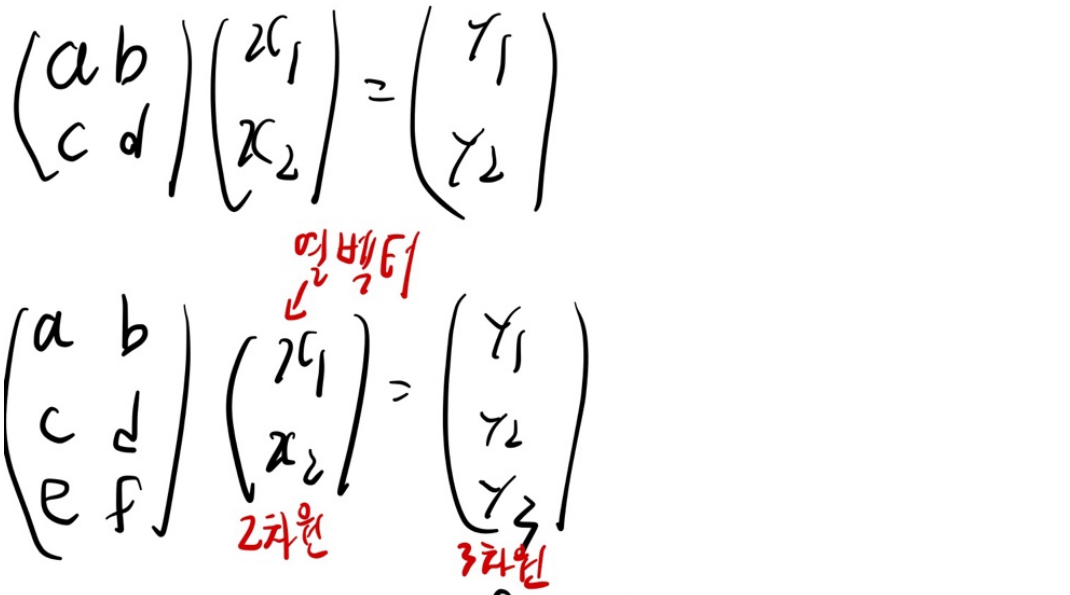
위의 예시 문장을 번역하면 '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.' 라는 의미가 된다. 그런데 여기서 그것(it)에 해당하는 것은 과연 길(street)일까? 동물(animal)일까? 우리는 피곤한 주체가 동물이라는 것을 아주 쉽게 알 수 있지만 기계는 그렇지 않다. 하지만 셀프 어텐션은 입력 문장 내의 단어들끼리 유사도를 구하므로서 그것(it)이 동물(animal)과 연관되었을 확률이 높다는 것을 찾아낸다.

- Learning long-range dependencies in the network
- Effective for parallelization for efficient computation n

Multi-headed Attention

Refined the self-attention layer by adding a mechanism called '**multi-headed**' attention

- It expands the model's ability to focus on different positions.
- It gives the attention layer multiple representation subspaces.


열벡터
2차원
3차원
Linear Transformation
행벡터
행벡터 Linear Transform 인경우 변환해주는
행렬의 위에은?

- Attention

$$A(Q,K,V) = softmax(QK^T)V$$

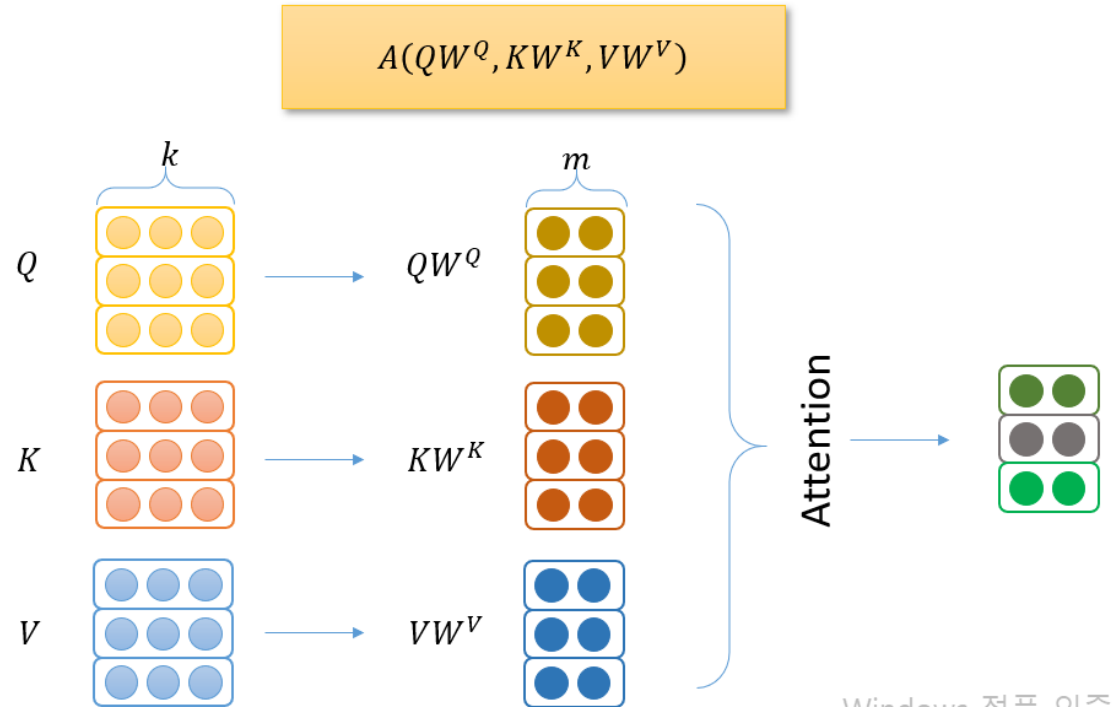
- What is **headed?**

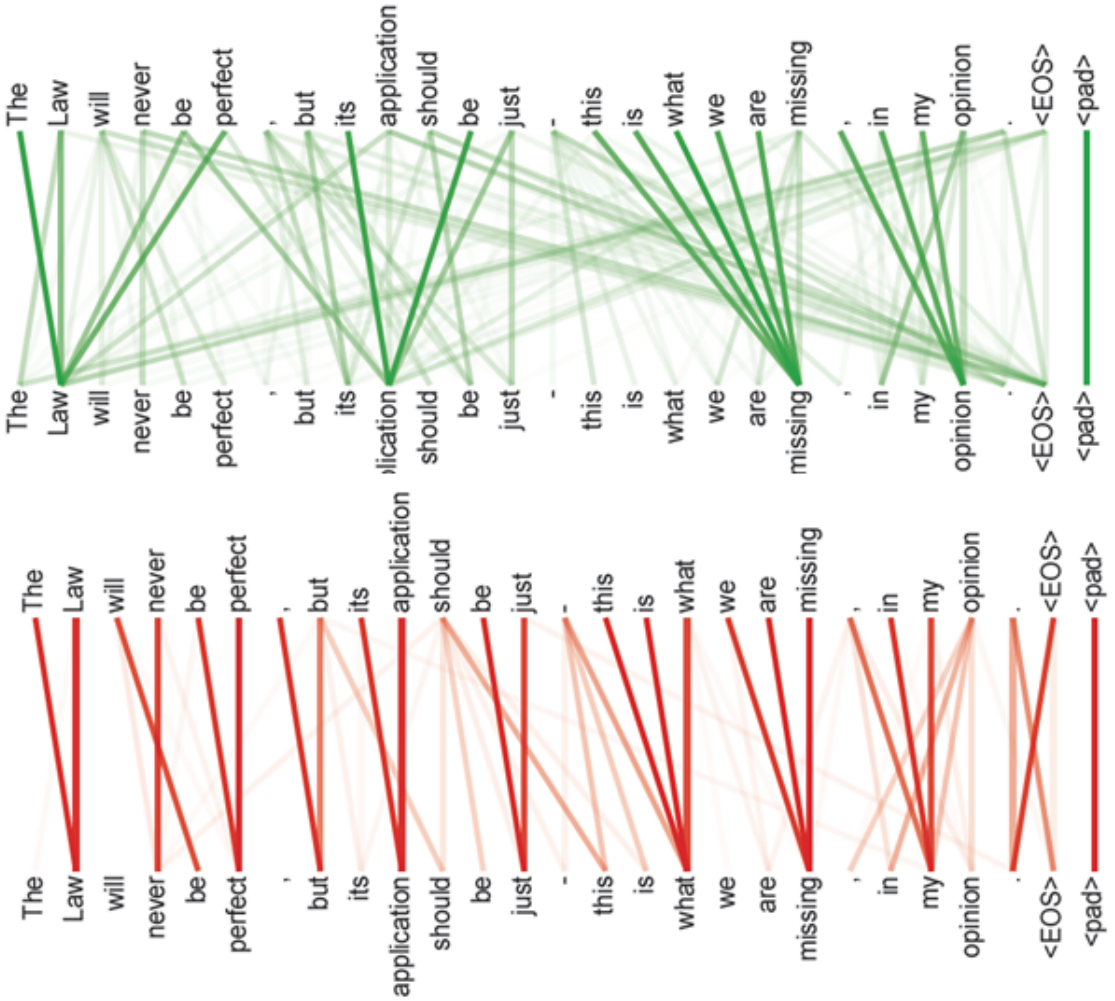
  - Linear Transformed: $W^Q, W^K, W^V \ (= k \times m)$

$$A(QW^Q, KW^K, VW^V)$$

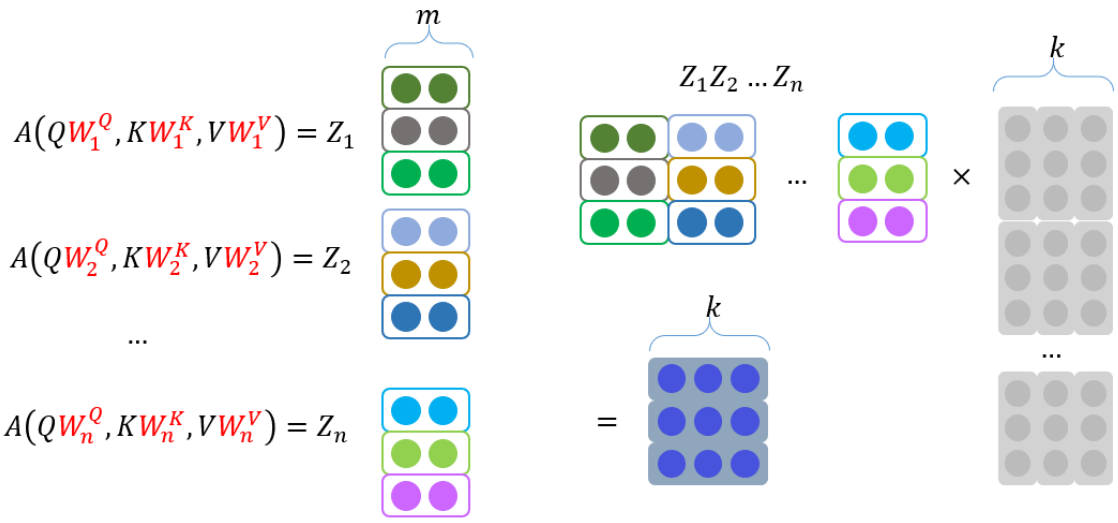
Head : A viewpoint of similarity

$$A(QW^Q, KW^K, VW^V)$$


$A(QW^Q, KW^K, VW^V)$
k
m
Q
K
V
QW^Q
KW^K
VW^V
Attention


The Law will never be perfect , but its application should be just - this is what we are missing , in my opinion . <EOS> <pad>

Let's use multiple heads to capture various similiarities.



그렇다면 이 w matrix들은 어떻게 구하나?—>GDM을통해 학습(학습 파라미터)

이런 병렬 어텐션을 모두 수행하였다면 모든 어텐션 헤드를 연결(concatenate)한다.

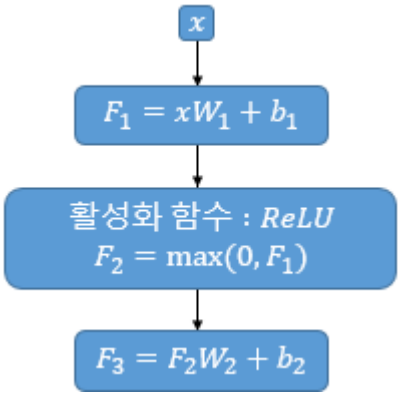
어텐션 헤드를 모두 연결한 행렬은 또 다른 가중치 행렬을 곱하게 되는데 ,이렇게 나온 결과 행렬이 멀티 헤드 어텐션의 최종 결과물이다.

#그 가중치 행렬은 코드에서 찾아니 단순히 Dense layer에 넣는다.

다시 말해 인코더의 첫번째 서브층인 멀티-헤드 어텐션 단계를 끝마쳤을 때, 인코더의 입력으로 들어왔던 행렬의 크기가 아직 유지되고 있음을 기억해두자. 첫번째 서브층인 멀티-헤드 어텐션과 두번째 서브층인 포지션 와이즈 피드 포워드 신경망을 지나면서 인코더의 입력으로 들어올 때의 행렬의 크기는 계속 유지되어야 한다. 트랜스포머는 다수의 인코더를 쌓은 형태인데(논문에서는 인코더가 6개), 인코더에서의 입력의 크기가 출력에서도 동일 크기로 계속 유지되어야만 다음 인코더에서도 다시 입력이 될 수 있기 때문이다.

Feed Forward

- Point - wise Feed Forward



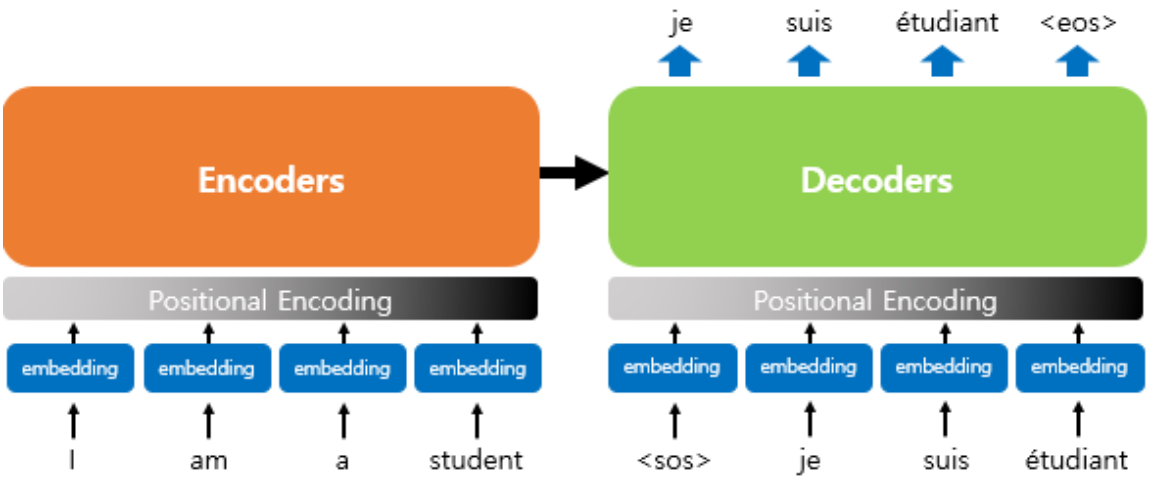
여기서 x는 앞서 멀티 헤드 어텐션의 결과로 나온 행렬이다.

Positional Encoding

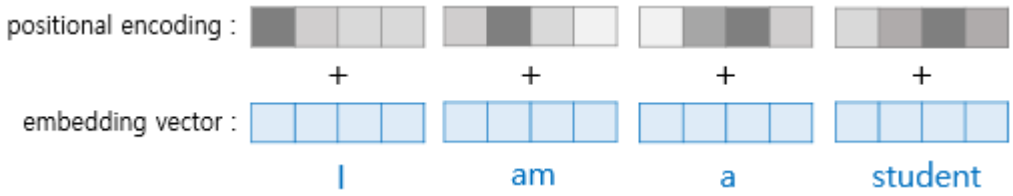
- Need to consider the order of the words in the input sentence.

RNN이 자연어 처리에서 유용했던 이유는 단어의 위치에 따라 단어를 순차적으로 입력받아서 처리하는 RNN의 특성으로 인해 각 단어의 위치 정보(position information)를 가질 수 있다는 점에 있었다.

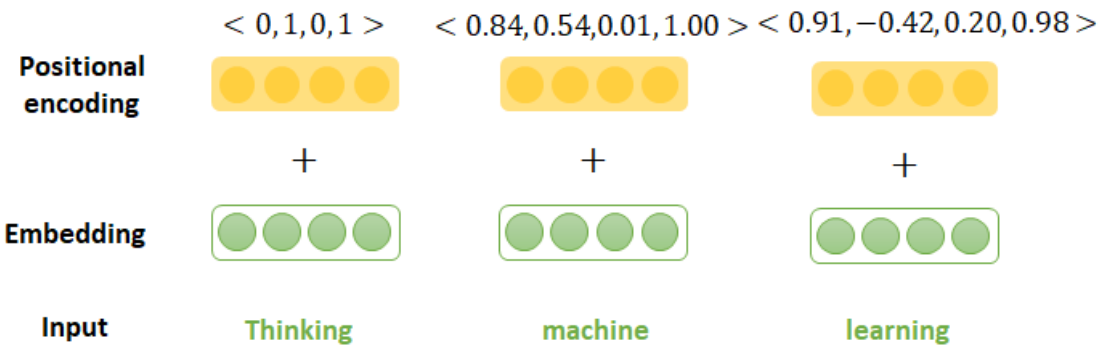
하지만 트랜스포머는 단어 입력을 순차적으로 받는 방식이 아니므로 단어의 위치 정보를 다른 방식으로 알려줄 필요가 있다. 트랜스포머는 단어의 위치 정보를 얻기 위해서 각 단어의 임베딩 벡터의 위치 정보들을 더하여 모델의 입력으로 사용하는데, 이를 **Positional Encoding**이라고 한다.



위의 그림은 입력으로 사용되는 임베딩 벡터들이 트랜스포머의 입력으로 사용되기 전에 포지셔널 인코딩값이 더해지는 것을 보여준다. 임베딩 벡터가 인코더의 입력으로 사용되기 전에 포지셔널 인코딩값이 더해지는 과정을 시각화하면 아래와 같다.



- Assume that embedding has a dimensionality of 4.
- The positional embedding would look like this:
  - Use cosine and sine curves.



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

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

pos는 입력 문장에서의 임베딩 벡터의 위치를 나타내며, i는 임베딩 벡터내의 차원의 인덱스를 의미한다.

d_model은 트랜스포머의 모든 층의 출력차원을 의미하는 트랜스포머의 하이퍼파라미터이다.

논문에서는 512

포지셔널 인코딩 방법을 사용하면 순서 정보가 보존되는데, 예를 들어 각 임베딩 벡터에 포지셔널 인코딩값을 더하면 같은 단어라고 하더라도 문장 내의 위치에 따라서 트랜스포머의 입력으로 들어가는 임베딩 벡터의 값이 달라집니다.

Complete Transformer Block

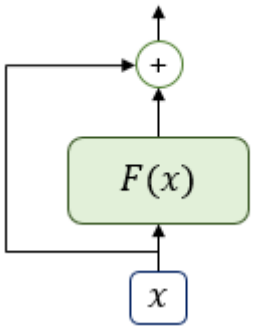
- Each block has two sublayers.
  - Multi-headed attention
  - 2 layer feed-forward net(with relu)>>linear transform 2개

- Each of these two steps also has:
  - Residual connection and Layer Norm

$$LayerNorm(X + Sublayer(X))$$

Residual Connection

$$H(x) = x + F(x)$$

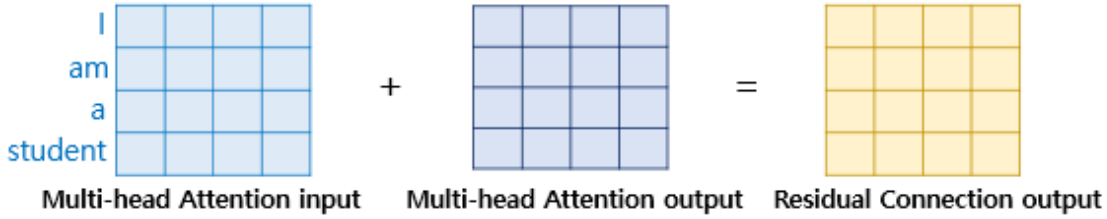


위 그림은 입력 x와 x에 대한 어떤 함수 F(x)의 값을 더한 함수 H(x)의 구조를 보여준다. 어떤 함수 F(x)가 트랜스포머에서는 서브층에 해당됩니다. 다시 말해 잔차 연결은 서브

층의 입력과 출력을 더하는 것을 말한다. 앞서 언급했듯이 트랜스포머에서 서브층의 입력과 출력은 동일한 차원을 갖고 있으므로, 서브층의 입력과 서브층의 출력은 덧셈 연산을 할 수 있다. 이것이 바로 위의 인코더 그림에서 각 화살표가 서브층의 입력에서 출력으로 향하도록 그려졌던 이유다. 잔차 연결은 컴퓨터 비전 분야에서 주로 사용되는 모델의 학습을 돕는 기법이다.

이를 식으로 표현하면 x+Sublayer(x)라고 할 수 있다.

가령, 서브층이 멀티 헤드 어텐션이었다면 잔차 연결 연산은 다음과 같습니다.



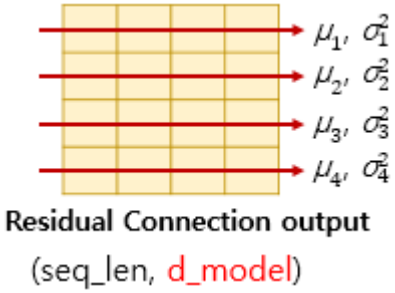
위 그림은 멀티 헤드 어텐션의 입력과 멀티 헤드 어텐션의 결과가 더해지는 과정을 보여준다.

Layer Normalization

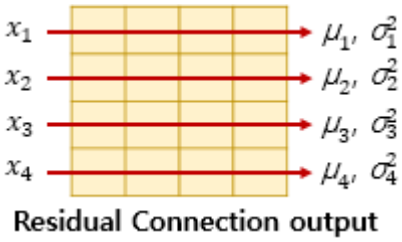
잔차 연결을 거친 결과는 이어서 층 정규화 과정을 거치게 된다. 잔차 연결의 입력을 x, 잔차 연결과 층 정규화 두가지 연산을 모두 수행한 후 의 결과 행렬을 LN이라고 하였을 때, 잔차 연결 후 층 정규화 연산을 수식으로 표현하자면 다음과 같다.

LN = LayerNorm(x+Sublayer(x))

층 정규화는 텐서의 마지막 차원에 대해서 평균과 분산을 구하고, 이를 가지고 어떤 수식을 통해 값을 정규화하여 학습을 돕는다. 여기서 텐서의 마지막 차원이란 것은 트랜스포머에서는 d_model 차원을 의미한다. 아래 그림은 d_model 차원의 방향을 화살표로 표현하였다.



층 정규화를 위해서 우선, 화살표 방향으로 각각 평균 μ과 분산 σ2을 구한다. 각 화살표 방향의 벡터를 xi라고 하자.



층 정규화를 수행한 후에는 벡터 xi는 lni라는 벡터로 정규화가 된다.

lni=LayerNorm(xi)

Layer Normalization을 두가지 과정으로 나누어서 말하면 첫번째는 평균과 분산을 통한 정규화, 두번째는 감마와 베타를 도입하는것이다. 우선, 평균과 분산을 통해 벡터 xi를 정규화 해준다. xi는 벡터인 반면, 평균 μi과 분산 σ2i은 스칼라다. 벡터 xi의 각 차원을 k라고 하였을 때, xi,k는 다음의 수식과 같이 정규화 할 수 있다. 다시 말해 벡터 xi의 각 k차원의 값이 다음과 같이 정규화 되는 것이다.

$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

이제 **감마와 베타**라는 벡터를 준비한다. 단, 이들의 초기값은 각각 1과 0이다.

γ | 1 | 1 | 1 | 1 |

β | 0 | 0 | 0 | 0 |

감마와 베타를 도입한 Layer Normalization의 최종 수식은 다음과 같으며, 감마와 베타는 학습 가능한 파라미터이다.

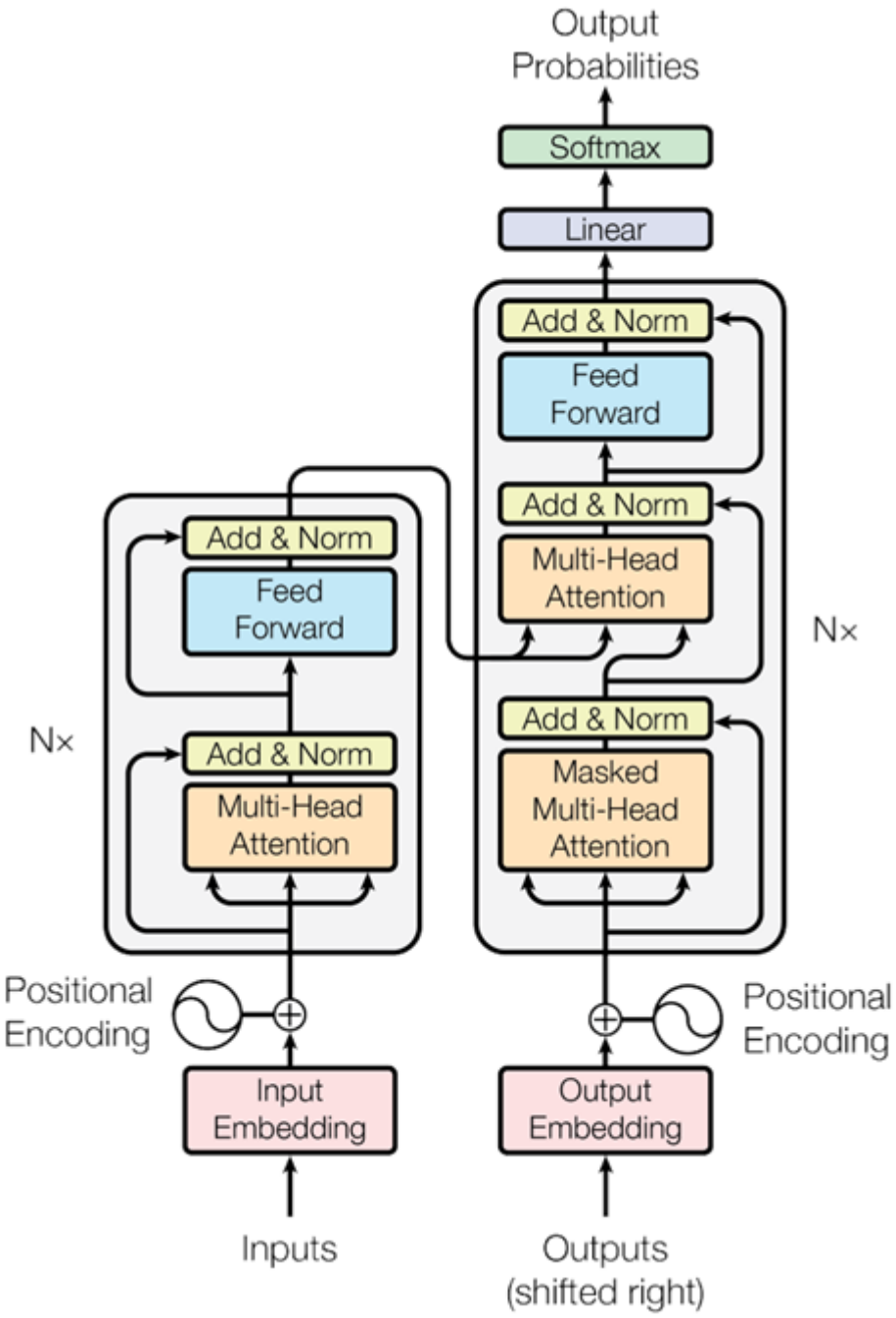
$$ln_i = \gamma \hat{x}_i + \beta = LayerNorm(x_i)$$

BatchNorm과 layerNorm

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

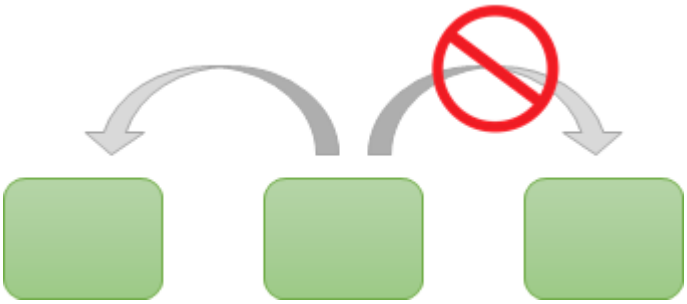
Decoder

Masked attention

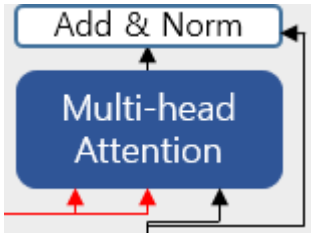


여기서 문제가 있다.seq2seq의 디코더에 사용되는 RNN계열의 신경망은 입력 단어를 매 시점마다 순차적으로 받으므로 다음 단어 예측에 현재 시점 이전에 입력된 단어들만 참고할 수 있다. 반면 트랜스포머는 문장 행렬로 입력을 반 번에 받으므로 현재 시점의 단어를 예측하고자 할 때, 입력 문장 행렬로부터 미래 시점의 단어까지도 참고할 수 있는 현상이 발생한다.

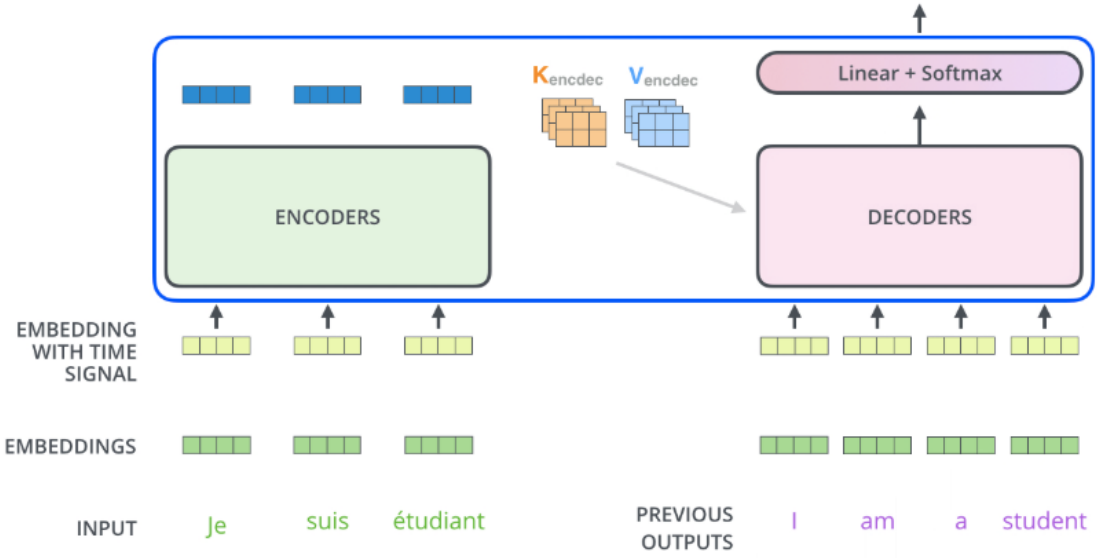
이를위해 **Masked**도입



Encoder-Decoder Attention

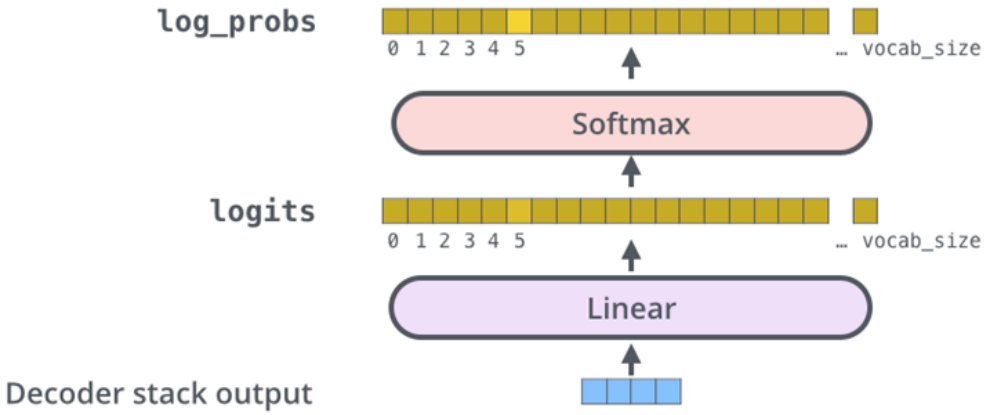


빨간색 두 화살표는 각각 key와 value를 의미하며, 이는 인코더의 마지막 층에서 온행렬로부터 얻는다. 반면 query는 디코더의 첫번째 서브층의 결과 행렬로부터 얻는다.



Final Linear and Softmax Layer

- The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.
- The softmax layer then turns those scores into probabilitles(all positive, all add up to 1.0)



Reference

https://wikidocs.net/31379

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

http://jalammar.github.io/illustrated-transformer/