Transformer

이현재

Contents

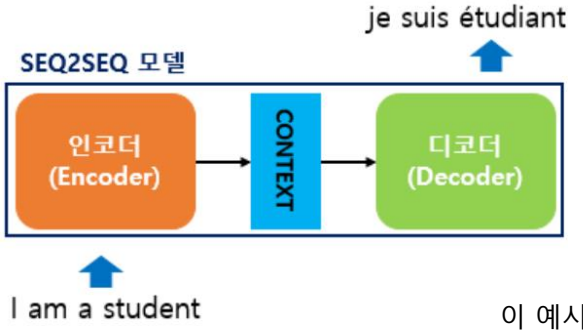
1. Seq2Seq
2. Attention
3. Transformer
4. Total Flow

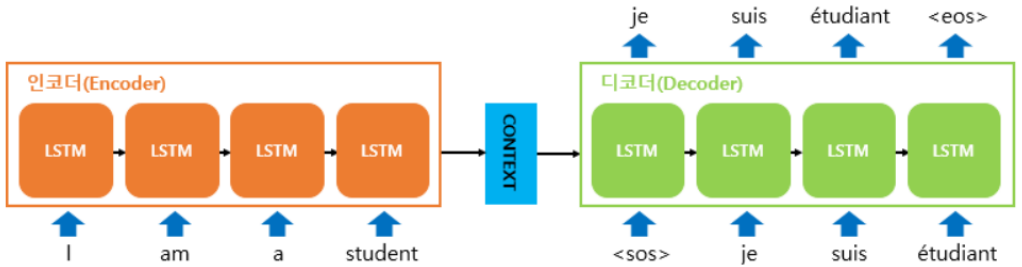
seq2seq

Hidden state를 만들어내는 영상입니다 이거 보시면 쉽게 이해되실거에요
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

- Seq2seq(sequence to sequence) 모델은 한 문장(시퀀스)을 다른 문장(시퀀스)으로 변환하는 모델
  - 예 : "I am a student" -> **[Seq2Seq model]** -> "je suis étudiant« (input -> model -> output)
- Encoder와 Decoder로 구성 (인코더는 입력 데이터를 인코딩(부호화)하고, 디코더는 인코딩된 데이터를 디코딩(복호화))
  - Context Vector : 입력에 대한 정보를 압축하고 있는 벡터
  - Encoder는 Input을 활용해 Context Vector를 만들고, Decoder는 Context Vector를 읽어 Output Sequence를 생성
  - Context Vector는 Encoder의 마지막 TimeStep이 출력한 은닉 상태와 같고, 이는 Decoder의 첫번째 은닉층에서 사용
  - 인코더로부터 전달받은 Context 벡터와 <sos>가 입력되면 그다음에 등장할 확률이 가장 높은 단어('je')를 예측합니다. 다음 스텝에서는 이전 스텝의 예측 값인 'je'가 입력되고 'je' 다음에 등장할 확률이 가장 높은 단어('suis')를 예측합니다. 이런 식으로 문장 내 모든 단어에 대해 반복합니다. 하지만 이는 **Test 단계에서의 디코더 작동 원리(학습할 때의 방법은 다음 페이지에서 정리)**

<sos> : 문장의 시작 / <eos> : 문장의 끝



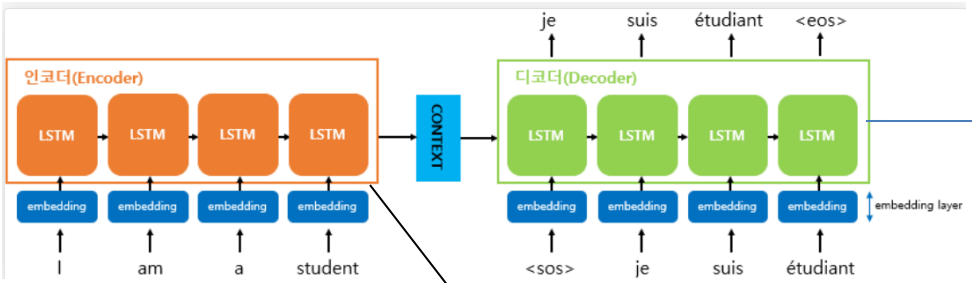


이 예시에서 Encoder와 Decoder는 LSTM을 사용해서 헷갈릴 수 있는데 그냥 RNN과 LSTM이 같다고 생각하시면 됩니다.

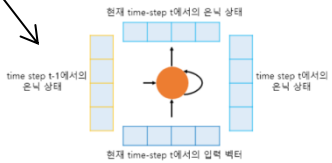
seq2seq

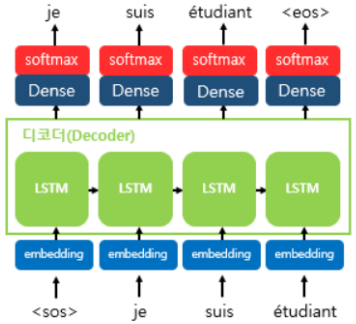
인코더와 디코더는 서로 다른 파라미터를 가진다고함

- 교사 강요(Teacher forcing)
  - Training 단계에서 Decoder 모델을 train하는 방법
  - 보통 RNN에서는 (n-1) step의 Output을 다음 단계인 n step의 Input으로 사용 (RNN 구조를 보면 쉽게 이해, RNN 모델이 예측한 값을 n step의 Input으로 사용)
  - 반면, 교사 강요는 (n-1) step의 **실제값을** n step의 Input으로 넣어주는 학습 방법
  - 이유 : (n-1) step의 예측값이 실제값과 다를 수 있기 때문. 정확한 데이터로 훈련하기 위해 예측값을 다음 step으로 넘기는 것이 아니라 실제값을 매번 Input으로 사용
  - Decoder의 훈련 단계에서는 교사 강요 방식으로 훈련하지만 테스트 단계에서는 일반적인 RNN 방식으로 예측합니다. 즉, 테스트 단계에서는 Context Vector를 입력값으로 받아 이미 훈련된 Decoder로 다음 단어를 예측하고, 그 단어를 다시 다음 스텝의 입력값으로 넣어줍니다. 이렇게 반복하여 최종 예측 문장을 생성하는 것입니다.
  - **디코더의 훈련 단계에서는 필요한 데이터가 Context 벡터와 <sos>, je, suis, étudiant입니다. 하지만 테스트 단계에서는 Context 벡터와 <sos>만 필요합니다.**
  - **훈련 단계에서는 교사 강요를 하기 위해 <sos>뿐만 아니라 je, suis, étudiant 모두가 필요한 것입니다. 하지만 테스트 단계에서는 Context 벡터와 <sos>만으로 첫 단어를 예측하고, 그 단어를 다음 스텝의 입력으로 넣습니다.**



Embedding은 저희가 알고 있는 그 문자(?)를 Vector로 만들어주는거예요 => 차원의 축소





디코더는 **인코더의 마지막 RNN 은닉층인 Context 벡터**와 **<sos>**를 입력값으로 받습니다. 디코더의 첫번째 RNN 셀은 Context 벡터와 <sos>를 통해 첫 단어를 예측합니다. 이 단어는 두 번째 스텝의 RNN 셀의 입력값이 됩니다. 두 번째 스텝의 RNN 셀은 **첫 번째 스텝의 RNN 셀이 예측한 단어**와 **두 번째 스텝에서의 입력값**을 받아 두 번째 단어를 예측합니다. 이런 식으로 최종 예측 값이 <eos> 일 때까지 반복

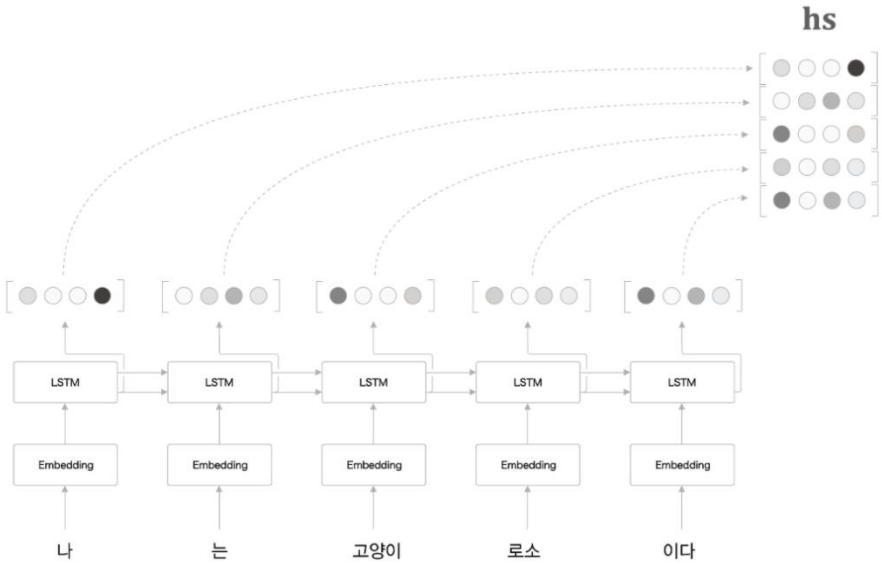
가장 확률이 높은 단어를 선택하기 위해 소프트맥스 함수를 사용 !

RNN의 재귀 구조 때문에 현재 시점 t에서의 hidden state는 과거시점의 RNN의 모든 hidden state의 영향을 받았기 때문에 Context Vector는 Input의 모든 정보를 요약해서 담을 수 있음

Attention

Hidden state를 만들어내는 영상입니다 이거 보시면 쉽게 이해되실거에요
https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

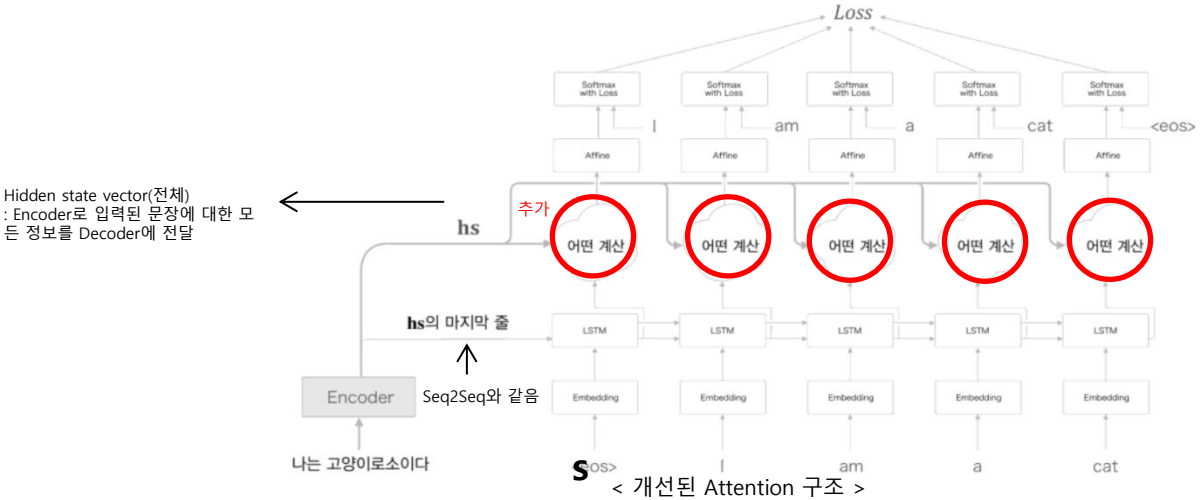
- Seq2Seq 모델의 한계
  - 하나의 **고정**된 크기의 벡터(Context Vector)에 대한 모든 정보를 압축하려고 하니 정보 손실이 발생
  - RNN(LSTM)의 고질적인 문제인 Gradient Vanishing 문제 발생 Lstm에서는 이를 해결하지 않았나...?
- Encoder 개선 (Seq2Seq 한계 1번 해결)
  - 고정된 벡터의 길이가 아닌 입력 문장의 길이에 맞추기 => **각 TimeStep마다 hidden state를 만들어 Stack으로 쌓자**



여기서 이 은닉 행렬에서 행의 개수는 단어의 개수가 되며 열의 개수는 고정적인 값을 가지게 됩니다.

Attention

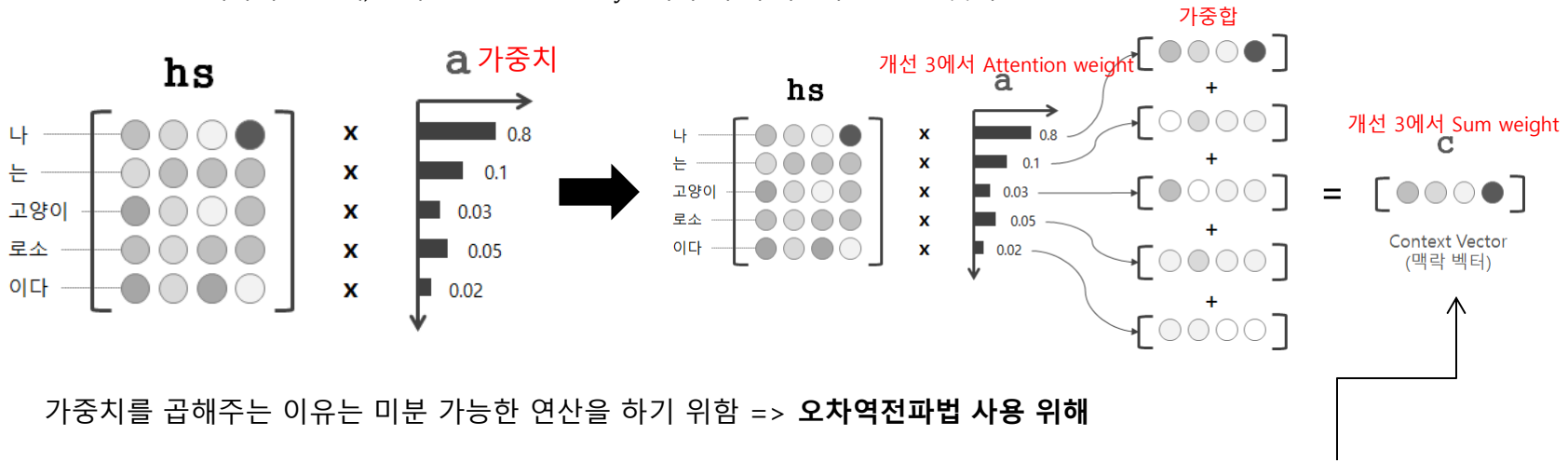
- Decoder 개선 ①
  - Encoder 개선에서 하나의 hidden state이 아닌 hidden state vector(Stack)를 만들어서 보내주었으므로, Decoder에서도 이를 모두 사용할 수 있도록 개선
  - **Alignment** : 단어의 대응 관계를 나타내는 정보
    - '특정 단어'와 대응 관계가 있는 '특정 단어'의 정보를 골라내는 것이 Attention의 핵심 -> 필요한 정보에 주목하여 그 정보로부터 문장을 변환
    - '어떤 계산' 추가 : hidden state vector와 각 TimeStep LSTM의 hidden state를 Input으로 받아 필요한 정보만을 Affine layer에 전달 **(=내적)**
    - Alignment 추출 : Decoder의 각 TimeStep에서 출력하고자 하는 단어와 대응 관계인 단어의 vector를 hidden state vector(전체)에서 선택 (이는 가중치로)
      - **예를 들어, Decoder가 'I'를 출력하려고 할 때, 전체인 Hidden state layer에서 '나'에 대응하는 vector를 가중치를 조절함으로써 찾아내겠다.**



< 개선된 Attention 구조 >

Attention

- Decoder 개선 ① (계속)
  - 'I'를 출력하려고 할 때, 전체인 Hidden state layer에서 '나'에 대응하는 vector 찾기



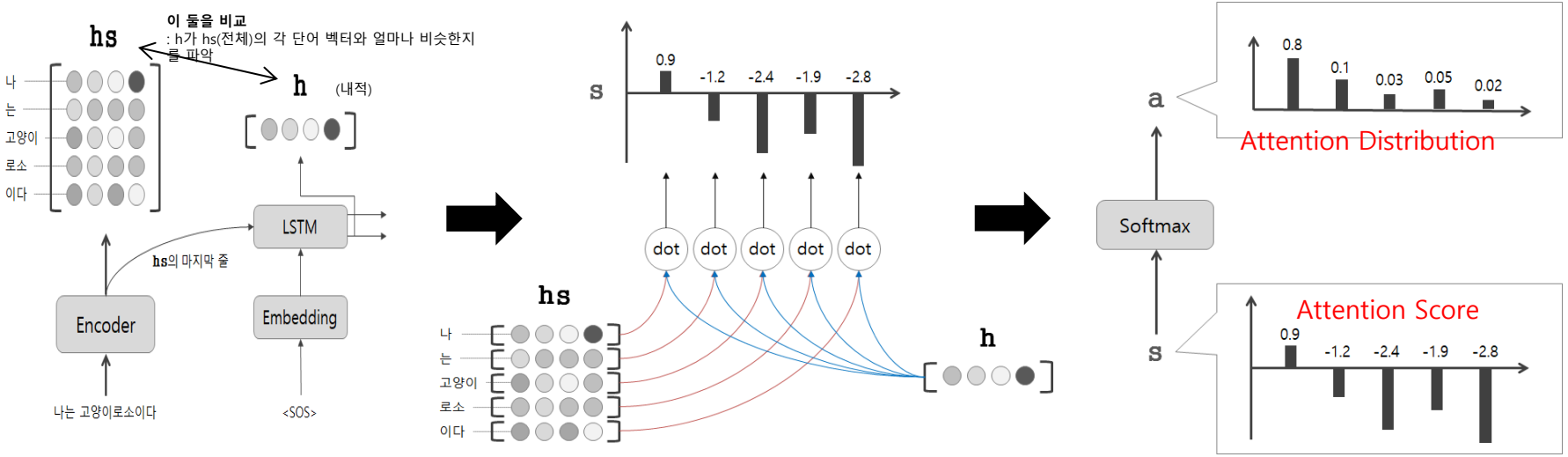
가중치를 곱해주는 이유는 미분 가능한 연산을 하기 위함 => **오차역전파법 사용 위해**

마지막 c(가중합 벡터)를 Context Vector라고 함

'나'에 대한 가중치가 0.8이므로 이 context vector는 '나'에 대한 정보가 많이 들어가 있음

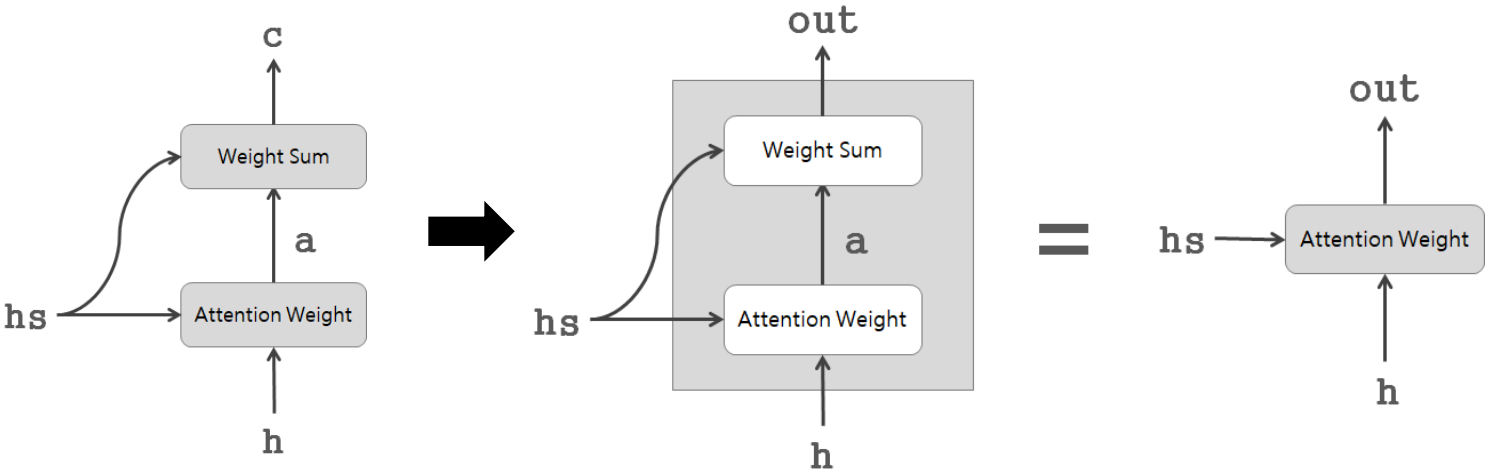
Attention

- Decoder 개선 ② (가중치 구하기)
  - **'내적'을 사용** : 내적을 통해 두 벡터가 얼마나 같은 방향을 향하고 있는지를 판단
  - Vector의 내적을 통해서 유사도 값을 구하였고, 내적은 음수가 나올 수도 있기 때문에 이를 Softmax 함수를 거쳐서 정규화
  - '나'에 대한 정보를 담고 있는 Vector와 h 벡터의 내적값(유사도)가 가장 높은 것을 확인 가능



Attention

- Decoder 개선 ③
  - Attention layer : 아래와 같은 계산을 거쳐 affine layer에 값을 보내는 layer
    => 전체 layer에서 "어떤 계산"을 하는 layer



Transformer

- Transformer의 하이퍼 파라미터

d_{model} = 512
트랜스포머의 인코더와 디코더에서의 정해진 입력과 출력의 크기를 의미합니다. 임베딩 벡터의 차원 또한 d_{model}이며, 각 인코더와 디코더가 다음 층의 인코더와 디코더로 값을 보낼 때에도 이 차원을 유지합니다. 논문에서는 512입니다.

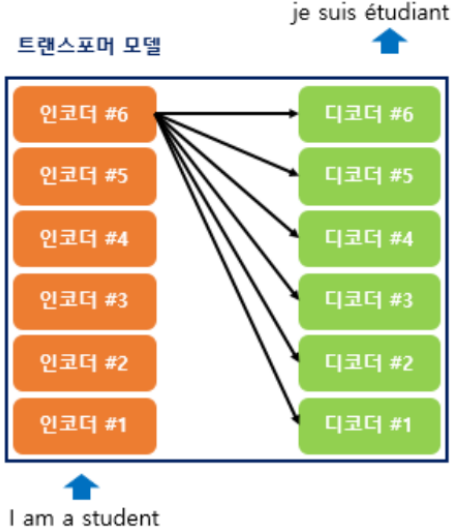
num_layers = 6
트랜스포머에서 하나의 인코더와 디코더를 층으로 생각하였을 때, 트랜스포머 모델에서 인코더와 디코더가 총 몇 층으로 구성되었는지를 의미합니다. 논문에서는 인코더와 디코더를 각각 총 6개 쌓았습니다.

num_heads = 8
트랜스포머에서는 어텐션을 사용할 때, 한 번 하는 것 보다 여러 개로 분할해서 병렬로 어텐션을 수행하고 결과값을 다시 하나로 합치는 방식을 택했습니다. 이때 이 병렬의 개수를 의미합니다.

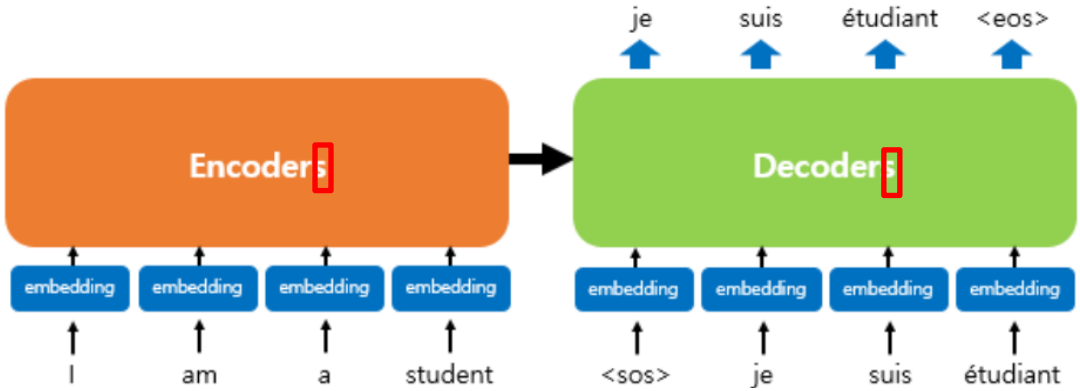
d_{ff} = 2048
트랜스포머 내부에는 피드 포워드 신경망이 존재하며 해당 신경망의 은닉층의 크기를 의미합니다. 피드 포워드 신경망의 입력층과 출력층의 크기는 d_{model}입니다.

Transformer

- 등장 배경 : Attention에서는 RNN을 보정하는 용도로만 사용. 이런 Attention만으로 Encoder와 Decoder를 구성
- seq2seq 구조에서는 Encoder와 Decoder에서 각각 하나의 RNN이 t개의 시점(time step)을 가지는 구조였다면 이번에는 Encoder와 Decoder라는 단위가 N개로 구성되는 구조
- Decoder는 마치 기존의 seq2seq 구조처럼 시작 심볼 <sos>를 입력으로 받아 종료 심볼 <eos>가 나올 때까지 연산을 진행 (RNN은 사용되지 않지만 여전히 Encoder-Decoder의 구조는 유지)



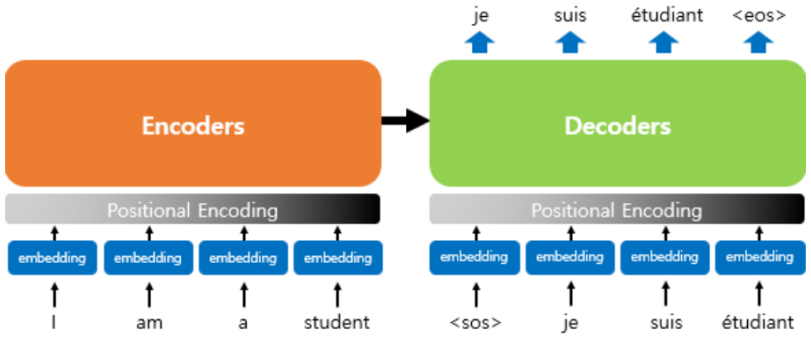
< Transformer 구조 >



< Transformer 구조 >

Transformer

- Positional Encoding
  - RNN이 자연어 처리에서 유용했던 이유는 단어의 위치에 따라 단어를 순차적으로 입력 받아서 처리하는 RNN의 특성으로 인해 각 단어의 위치 정보(position information)를 가질 수 있다는 점 (굳이 단어의 위치를 신경 쓰지 않아도 됐다.)
  - Transformer는 단어 입력을 순차적으로 받는 방식이 아니므로 단어의 위치 정보를 다른 방식으로 알려줄 필요가 있다.
  - Positional Encoding : 각 단어의 임베딩 벡터에 위치 정보들을 더해 모델의 입력으로 사용
  - Input으로 사용되는 임베딩 벡터들이 트랜스포머 입력으로 사용되기 전 Positional Encoding의 값이 더해진다(2번째 그림)



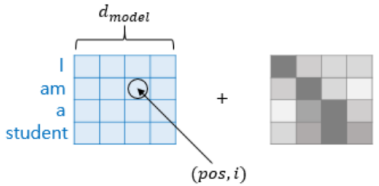
Transformer

- Positional Encoding (계속)
  - Positional Encoding 값들이 위치 정보를 담을 수 있는 이유는 위치 정보를 가진 값으로 만들어주기 위해 아래의 두 함수를 사용

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

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

사인 함수와 코사인 함수의 그래프를 상기해보면 요동치는 값의 형태를 생각해볼 수 있는데, 트랜스포머는 사인 함수와 코사인 함수의 값을 임베딩 벡터에 더해주므로서 단어의 순서 정보를 더하여 줍니다. 그런데 위의 두 함수에는 pos, i, d_{model} 등의 생소한 변수들이 있습니다. 위의 함수를 이해하기 위해서는 위에서 본 임베딩 벡터와 포지셔널 인코딩의 덧셈은 사실 임베딩 벡터가 모여 만들어진 문장 행렬과 포지셔널 인코딩 행렬의 덧셈 연산을 통해 이루어진다는 점을 이해해야 합니다.



pos는 입력 문장에서의 임베딩 벡터의 위치를 나타내며, i는 임베딩 벡터 내의 차원의 인덱스를 의미합니다. 위의 식에 따르면 임베딩 벡터 내의 각 차원의 인덱스가 짝수인 경우에는 사인 함수의 값을 사용하고 홀수인 경우에는 코사인 함수의 값을 사용합니다. 위의 수식에서 $(pos,\ 2i)$일 때는 사인 함수를 사용하고, $(pos,\ 2i+1)$일 때는 코사인 함수를 사용하고 있음을 주목합시다.

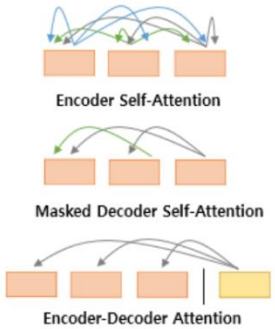
또한 위의 식에서 d_{model}은 트랜스포머의 모든 층의 출력 차원을 의미하는 트랜스포머의 하이퍼파라미터입니다. 앞으로 보게 될 트랜스포머의 각종 구조에서 d_{model}의 값이 계속해서 등장하는 이유입니다. 임베딩 벡터 또한 d_{model}의 차원을 가지는데 위의 그림에서는 마치 4로 표현되었지만 실제 논문에서는 512의 값을 가집니다.

위와 같은 포지셔널 인코딩 방법을 사용하면 순서 정보가 보존되는데, 예를 들어 각 임베딩 벡터에 포지셔널 인코딩의 값을 더하면 같은 단어라고 하더라도 문장 내의 위치에 따라서 트랜스포머의 입력으로 들어가는 임베딩 벡터의 값이 달라집니다. 이에 따라 트랜스포머의 입력은 순서 정보가 고려된 임베딩 벡터가 됩니다. 이를 코드로 구현하면 아래와 같습니다.

Transformer

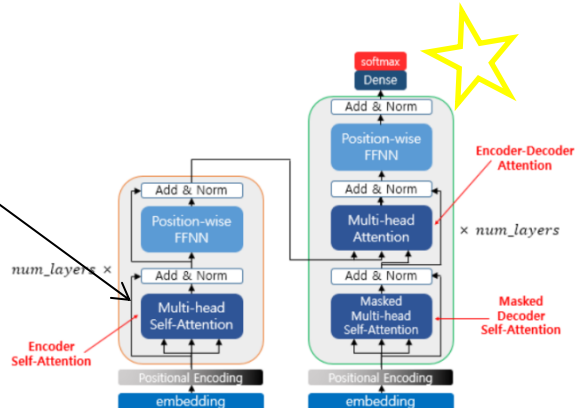
- Attention
  - 첫 번째 그림의 Self-Attention은 Encoder에서, 두 번째와 세 번째 Attention은 Decoder에서 수행
  - **Self Attention** : 본질적으로 Query, Key, Value가 동일한 경우 (Query, Key 등이 같다는 것은 벡터의 값이 같다는 것이 아니라 벡터의 출처(Encoder, Decoder)가 같다는 의미)

- Multi-head, Query, Key, Value는 다음 페이지에서 설명 드리겠습니다.



< Attention 종류 >

인코더의 셀프 어텐션 : Query = **Key** = Value
디코더의 마스크드 셀프 어텐션 : Query = **Key** = Value
디코더의 인코더-디코더 어텐션 : Query : 디코더 벡터 / **Key** = Value : 인코더 벡터

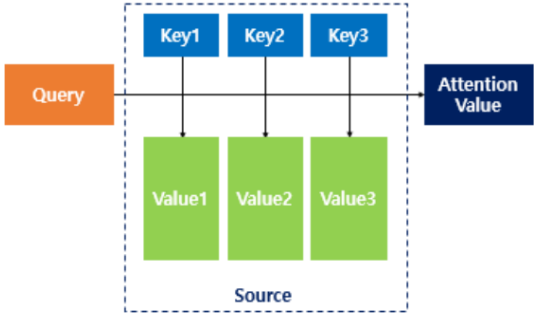


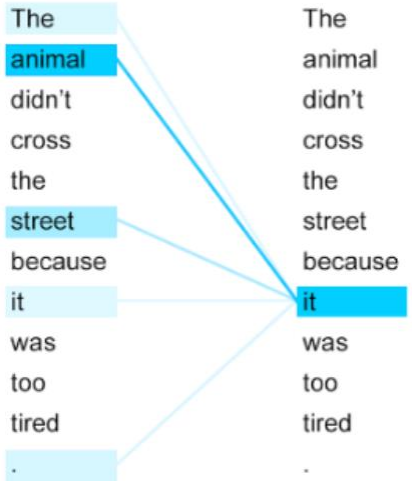
< Transformer 구조 >

Transformer

Self Attention : 입력 문장 내의 단어 유사도를 구하기 위함

- Encoder의 Self Attention
  - Q (Query) : t 시점의 Decoder의 hidden state
  - K (Keys) : 모든 시점의 Encoder의 hidden state
  - V (Values) : 모든 시점의 Encoder의 hidden state
  - Self Attention : 자기 자신에게 Attention을 수행한다는 것
  - Attention : 주어진 Query에 대해서 모든 Key와의 유사도를 각각 구하고 이 유사도를 가중치로 하여 맵핑되어있는 각각의 Value에 반영해준다. 유사도가 반영된 Value를 모두 가중합한 Attention value를 Return





이렇게 적어놓으면 당연히 이해가 안되겠네요..!

예를 들어서, 이 문장에서 'it'이 animal인지 street인지 알기 위해서 Q, K, V 개념을 도입합니다.

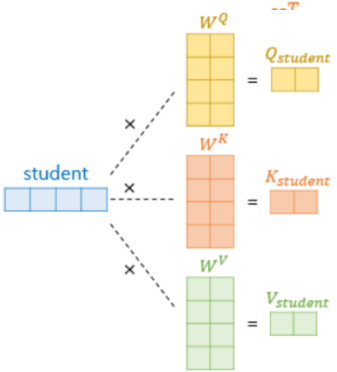
Query는 우리가 알고자 하는 객체, 이 경우에서는 it
Key는 이 문장 전체를 의미하고
Value는 각 Key를 통해서 얻어낸 유사도 값입니다!

다음 페이지에서 이 Q, K, V를 통해서 어떻게 유사도를 알아내는지 알아봅시다

Transformer

- Q, K, V 벡터 얻기
  - Self Attention 우선 각 단어 벡터들로부터 Q 벡터, K 벡터, V 벡터를 얻는 작업을 거침. 이때 이 Q 벡터, K 벡터, V 벡터들은 초기 입력인 dmodel의 차원을 가지는 단어 벡터들보다 더 작은 차원을 가지는데, 논문에서는 dmodel=512의 차원을 가졌던 각 단어 벡터들을 64의 차원을 가지는 Q벡터, K벡터, V벡터로 변환
  - 이 64라는 값은 하이퍼 파라미터인 num_heads로 결정. Transformer는 dmodel을 num_heads(나중에 알아볼 병렬 개수)로 나눈 값을 Q 벡터, K 벡터, V 벡터 차원의 크기로 결정
- 스케일드 닷-프로덕트 어텐션(Scaled dot-product Attention)
  - Q, K, V 벡터를 얻었으면 위에 저희가 공부했던 Attention 메커니즘과 같습니다.
  - 각 Q벡터와 K 벡터에 대해서 Attention Score를 구하고, Attention Distribution을 구한 뒤에 이를 사용하여 모든 V 벡터를 가중합하여 Attention value 또는 context vector를 구하게 된다.
  - 하지만 이 과정에서 단순히 내적(dot-product)를 사용하는 것이 아닌 scaled dot-product 연산을 사용

Scaled dot product Attention : $score\ function(q,k) = q \cdot k/\sqrt{n}$



Student에서 Q, K, V 벡터 얻기

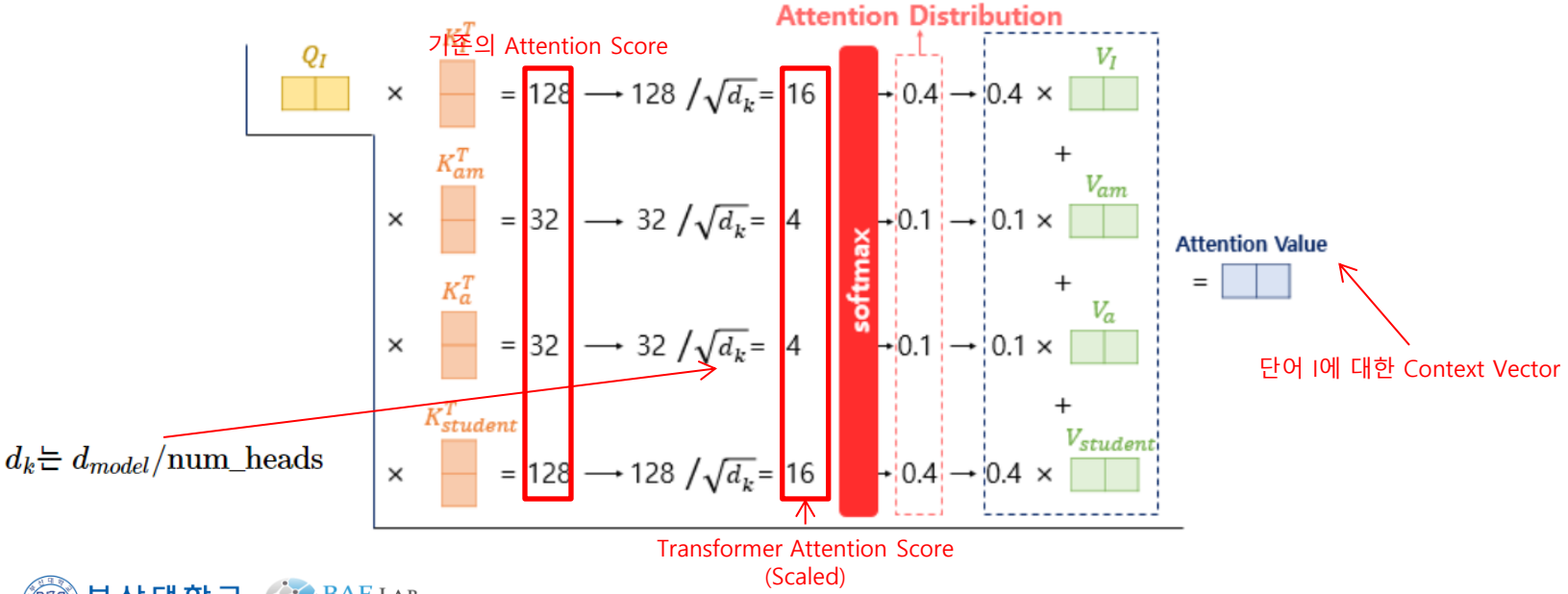
각 벡터에 3개의 서로 다른 가중치 행렬을 곱하고 64의 크기를 가진 Q, K, V 벡터를 얻는다

이를 모든 단어에 수행하여 모든 단어 각각의 Q, K, V 벡터를 얻는다.

Transformer

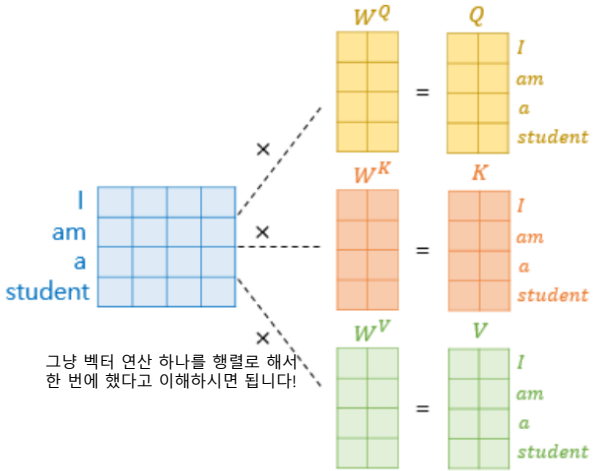
- 스케일드 닷-프로덕트 어텐션(Scaled dot-product Attention) (계속)
  - $\sqrt{d_k}$를 통하여 Scale을 해주는 이유 : 궁금해서 저도 여기에 적어놨는데 다음 새로운 연산 때 다시 언급하겠습니다...ㅎ
  - 실제 예시

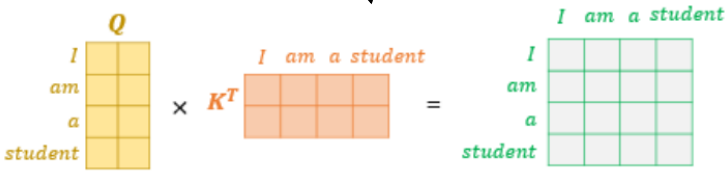
되게 복잡해보이지만 Attention이랑 똑같아요!
Scaled가 있냐 없냐 차이 그 뿐입니다.



Transformer

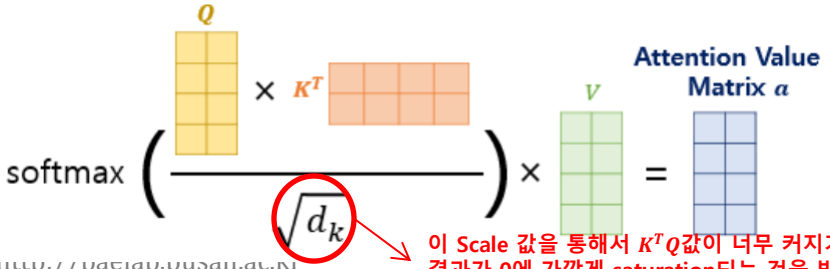
- 행렬 연산으로 일괄 처리하기
  - 벡터 연산이 아닌 행렬 연산으로 처리하면 각 단어에 대해서 일일이 할 필요가 없고 한 번에 수행 가능
  - 실제로도 행렬 연산을 사용 (지금까지 사용한 벡터 연산은 이해를 돕기 위해 첨부)
  - Q 행렬과 K 행렬을 전치한 행렬과 곱해준다면, 각각의 단어의 Q 벡터와 K 벡터의 내적이 각 행렬의 원소가 되는 행렬이 결과로 된다.





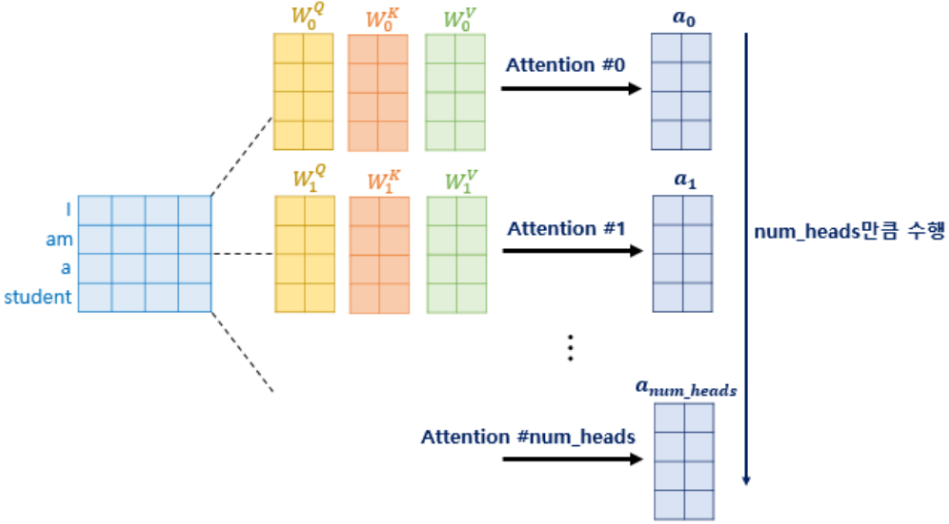
이 행렬에 $\sqrt{d_k}$로 나누어준다면 이는 각 행과 열이 Attention Score를 가지게 되는 행렬이 된다.
예를 들어 'am' 행과 'student' 열의 값은 'am'의 Q 벡터와 'student'의 K 벡터의 Attention Score 값!

이 Attention Score 행렬에 Softmax 함수를 사용하고 V 행렬을 곱하면 Attention Value 행렬을 구할 수 있다.



Transformer

- Multi Head Attention
  - $\sqrt{d_k}$로 나눠주는 심화 이유

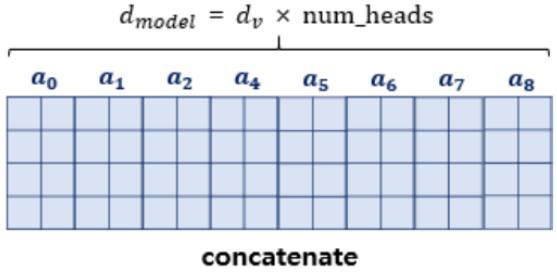


차원을 굳이 축소해서 Attention을 수행하는 이유?

dmodel의 차원을 num_heads개로 나누어 dmodel/num_heads의 차원을 가지는 Q, K, V에 대해서 num_heads개의 병렬 Attention을 수행
(Attention을 병렬로 수행하여 다른 시각으로부터 정보들을 수집)

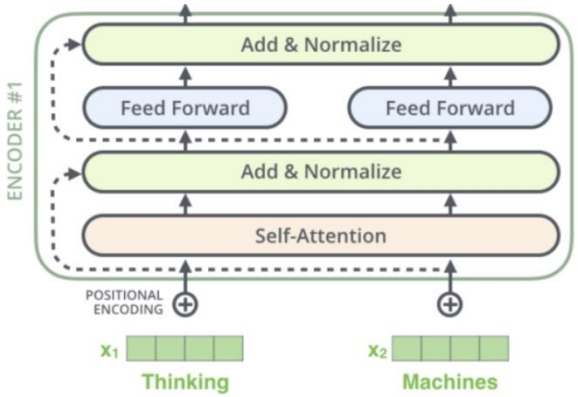
앞서 사용한 예문 '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.'를 상기해봅시다. it에 대한 Q벡터로부터 다른 단어와의 연관도를 구하였을 때 첫번째 어텐션 헤드는 '그것(it)'과 '동물(animal)'의 연관도를 높게 본다면, 두번째 어텐션 헤드는 '그것(it)'과 '피곤하였기 때문이다(tired)'의 연관도를 높게 볼 수 있습니다. 각 어텐션 헤드는 전부 다른 시각에서 보고 있기 때문입니다.

그래서 병렬 Attention을 모두 수행하였다면 모든 Attention head를 Concatenate를 해서 정보를 취합합니다.

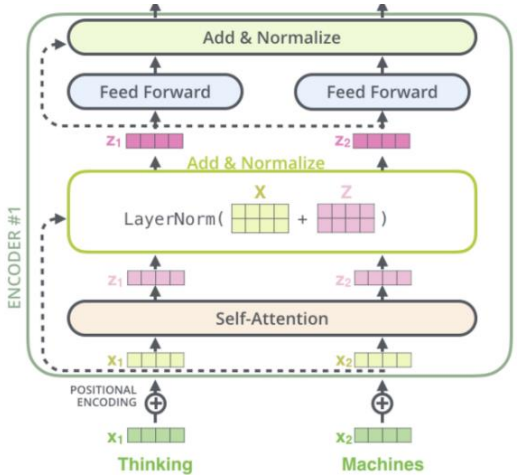


Transformer

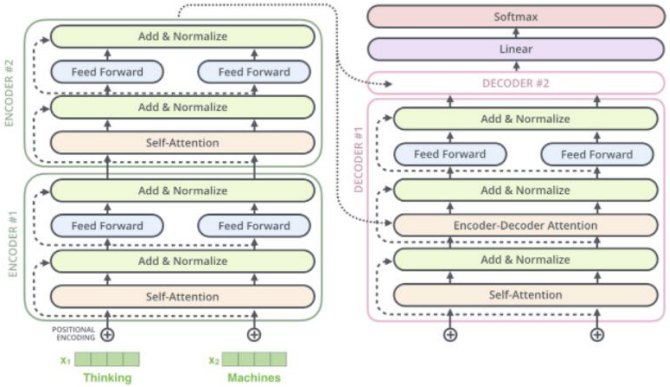
- 잔차 연결(residual connection)과 층 정규화(layer normalization)
  - 각 Encoder 내의 sub layer가 residual connection으로 연결되어 있으며, 그 이후에는 layer normalization 과정을 수행
  - Decoder에서도 똑같이 적용
  - Concatenate할 때 필요함



< residual connection >



< layer normalization >



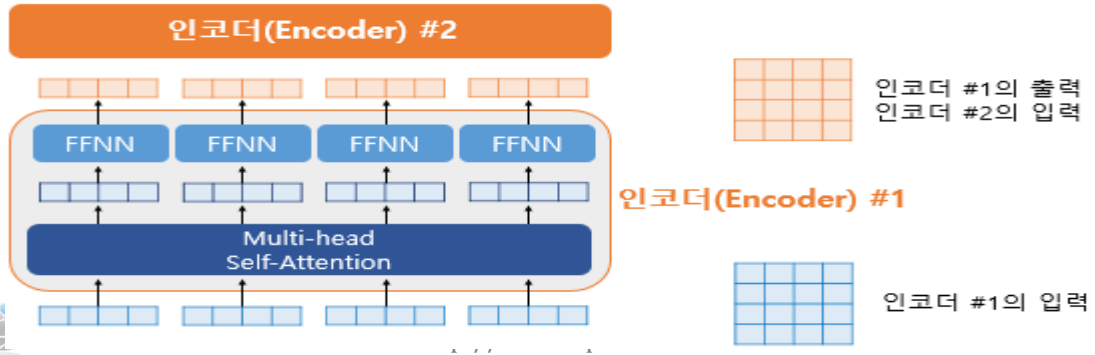
< 각각 2개의 Encoder와 Decoder를 가지는 Transformer>

그 외에 요소들 - 피드 포워드 신경망

- 각 단어들마다 행렬 연산이 이루어짐
- 인코더 개수를 여러 개로 할 경우 피드

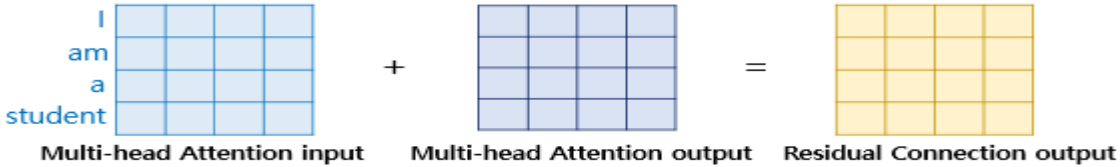
$$FFNN(x) = MAX(0, xW_1 + b_1)W_2 + b_2$$

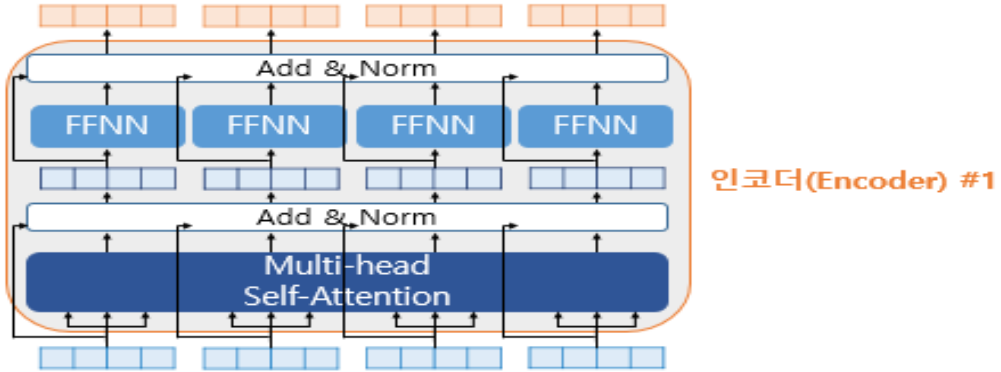
$(\text{seq\_len}, d_{model})$ x (d_{model}, d_{ff}) x (d_{ff}, d_{model}) = $(\text{seq\_len}, d_{model})$



그 외에 요소들 – 잔차 연결

- 잔차 연결 : 다음 연산을 수행하기 이전에 연산을 시행한 값과 시행하기 이전에 값을 합쳐 줌으로서 연산 이전의 정보도 함께 전달, 모델이 복잡해짐에 따라 발생할 수 있는 정보 손실을 방지

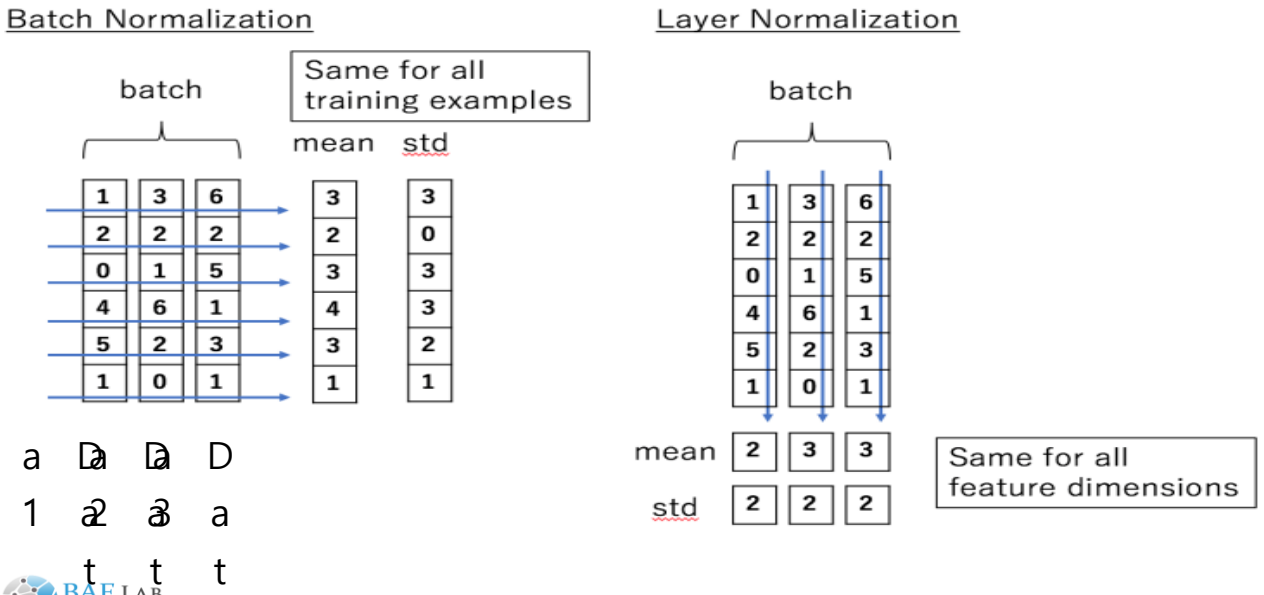




그 외에 요소들 – 정규화

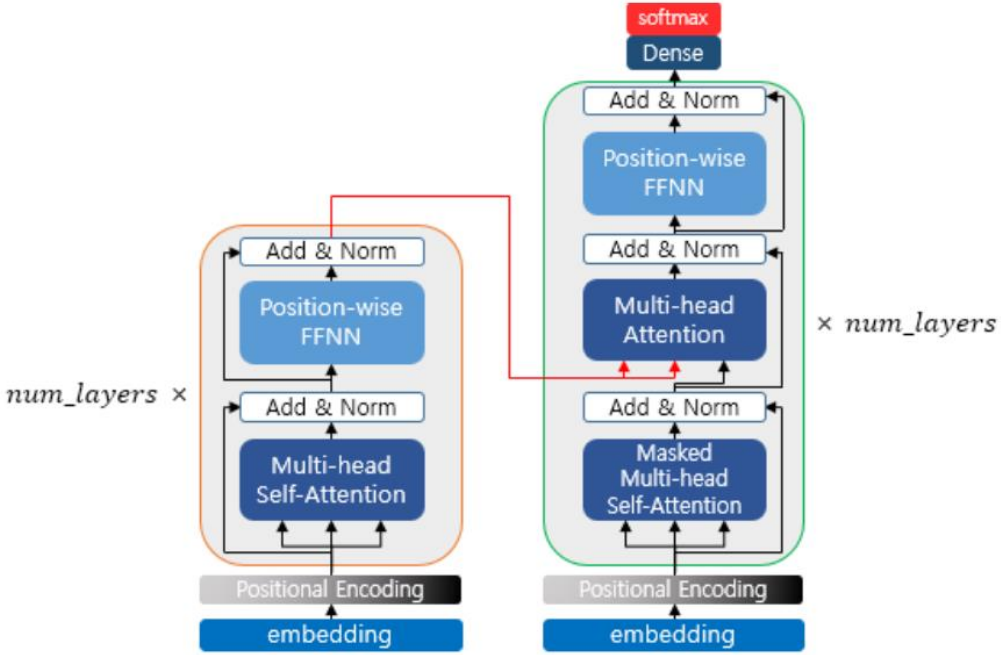
- Layer normalization: Batch Normalization 은 배치 단위로 특성들을 정규화한다면

Layer Normalization은 각 데이터들의 값을 정규화하여 사용



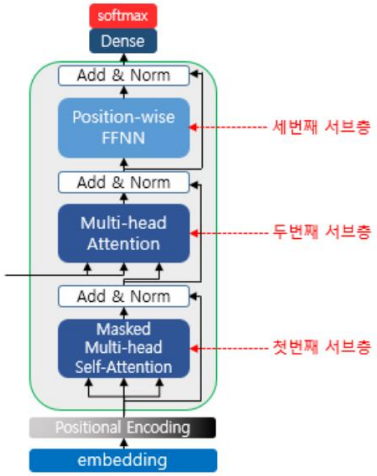
Transformer (Decoder side) 지금까지 인코더 설명이었습니다...!

- 마지막 층의 Encoder의 출력을 Decoder에게 전달합니다. Encoder 연산이 끝났으면 Decoder 연산이 시작되어 Decoder 또한 num_layers만큼의 연산을 하는데, 이때마다 Encoder가 보낸 출력을 각 Decoder 층 연산에 사용



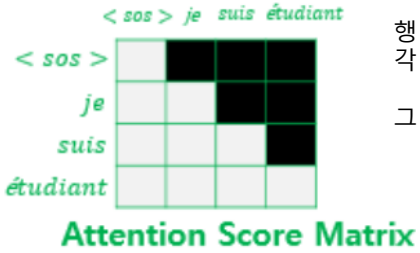
Transformer (Decoder side)

- Decoder의 첫 번째 서브층 : Self attention과 look-ahead mask
  - Decoder도 Encoder와 동일하게 임베딩 레이어와 Positional Encoding을 거친 후의 **문장 행렬**이 입력
  - 트랜스포머 또한 seq2seq와 마찬가지로 교사 강요(Teacher Forcing)을 사용하여 훈련되므로 학습 과정에서 Decoder는 번역할 문장에 해당되는 <sos> je suis étudiant의 문장 행렬을 한 번에 입력받습니다. 그리고 Decoder는 이 문장 행렬로부터 각 시점의 단어를 예측하도록 훈련됩니다.
  - **트랜스포머는 문장 행렬로 입력을 한 번에 받으므로 현재 시점의 단어를 예측하고자 할 때, 입력 문장 행렬로부터 미래 시점의 단어까지도 참고할 수 있는 현상이 발생**
  - **이를 해결하기 위해서 Transformer의 Decoder에서는 현재 시점의 예측에서 현재 시점보다 미래에 있는 단어들을 참고하지 못하도록 look-ahead mask를 도입 (padding 개념 도입)**



Look-ahead mask는 첫 번째 서브층에서 수행

첫 번째 서브층에서는 Encoder의 첫번째 서브층인 multi-head attention 층과 동일한 연산을 수행
하지만 다른 점은, 미래 시점의 값을 참고하지 못하도록 마스킹을 적용



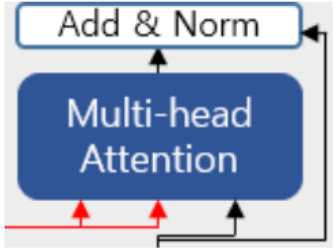
행렬을 자세히 보면,
각 행을 기준으로 자기 자신과 그 이전의 값들만 참고 가능

그 외에는, Self attention과 multi-head attention을 수행하는 것은 같다.

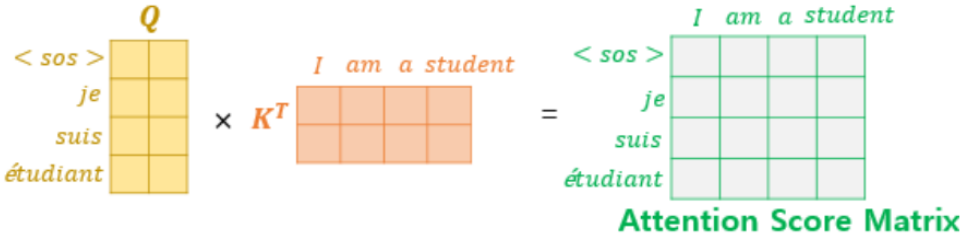
Transformer (Decoder side)

- Transformer의 attention 정리
  - Encoder의 Self attention : padding mask 전달
  - Decoder의 첫 번째 서브층인 masked Self attention : look-ahead mask 전달
  - Decoder의 두 번째 서브층인 Encoder-Decoder attention : padding mask 전달

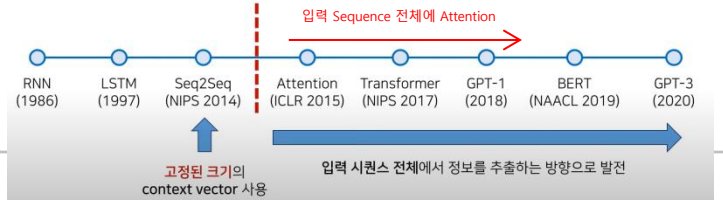
- Decoder의 두 번째 서브층 : Encoder-Decoder attention



Encoder로부터 2개의 빨간 화살표가 오는 것을 확인할 수 있는데
두 개의 화살표는 각각 Key와 Value를 의미하며 이는 Encoder의 마지막 행렬에서 얻는다고 합니다.
반면 Query는 Decoder의 첫 번째 서브층의 결과 행렬로부터 얻는다는 점이 다릅니다.



Flow



이 3가지를 또 구분하는 것은 저번 자료에서 확인 가능합니다!
자연어에서는 RNN이 처음 도입되고, 조금 더 긴 Sequence에서 적용할 수 있는 LSTM이 등장

RNN
LSTM
GRU

∴ 특징
- 순환 신경망
- Input / Output

∴ 한계
- Gradient Vanishing
- Input과 Output의 크기가 같다고 가정

Seq2Seq

고정된 Context Vector에서 정보 손실이 발생하고
각 단어의 연관성을 확인하기에는 부족해서

∴ 특징
- Encoder / Decoder
- Context Vector(입력의 정보를 담은)

∴ 한계
- 정보 손실(고정된 크기)
- Gradient Vanishing
- 병목(Bottleneck)이 발생

Attention

∴ 특징
- Decoder는 Encoder의 모든 Outputs을 참고
- 어떤 단어에 Attention할지 가중치를 통해 결정
- Attention 가중치를 사용해 각 출력이 어떤 입력 정보를 참고했는지 알 수 있음

Transformer

∴ 특징
- Encoder / Decoder를 다수 사용
- RNN과 CNN을 사용하지 않음
- Attention만 사용
- Positional Encoding
- Residual Learning(성능 향상)
- 마지막 Encoder layer의 Output이 모든 Decoder later에 입력으로 활용
- Multi-Head Attention
- Scaled Dot-Product
- Mask matrix
- Encoder Self-Attention
  : 각각의 단어가 서로에 어떤 연관성을 가지는지, 전체 문장의 Representation을 learning할 수 있게
- Masked Decoder Self-Attention
  : 앞 단어만 참고할 수 있도록
- Encoder-Decoder Attention
  : Query가 Decoder에, Key와 Value는 Encoder에

GPT-1

∴ 특징
- Semi-supervised language model
- 두가지 학습단계 **1)unsupervised pre-training ,2)supervised fine-tuning**를 통해 최소한의 구조 변화로 Target task에 transfer 가능한 언어 모델
- labeled data의 부족으로 어려움을 겪고 있는 많은 Supervised NLP task를 위한 새로운 해법을 제시

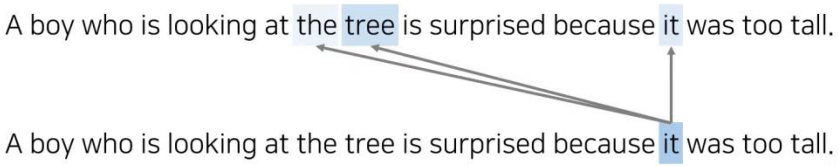
BERT

∴ 특징
- Unsupervised pre-training & supervised fine-tuning
- 양방향성을 포함하여 문맥을 더욱 자연스럽게 파악
- Token / Segment / Position Embedding
- Masked Language Model
- Next Sentence Model

Flow

트랜스포머의 동작 원리: Self-Attention

- Self-Attention은 인코더와 디코더 모두에서 사용됩니다.
  - 매번 **입력 문장에서 각 단어가 다른 어떤 단어와 연관성이 높은 지** 계산할 수 있습니다.



Q&A