ICT공학부 201904192 김성중

산업데이터사이언스학부 201904217 우사랑

== REPORT ==

< Transformer 구조 조사 및 분석 >

과 목	딥러닝프레임워크
담당 교수	최입엽 교수님
제출 일자	2024-06-03
전공	ICT 공학부, 산업데이터사이언스학부
학번	201904192, 201904217
이름	김성중, 우사랑

ICT공학부 201904192 김성중

산업데이터사이언스학부 201904217 우사랑

- 목 차 -

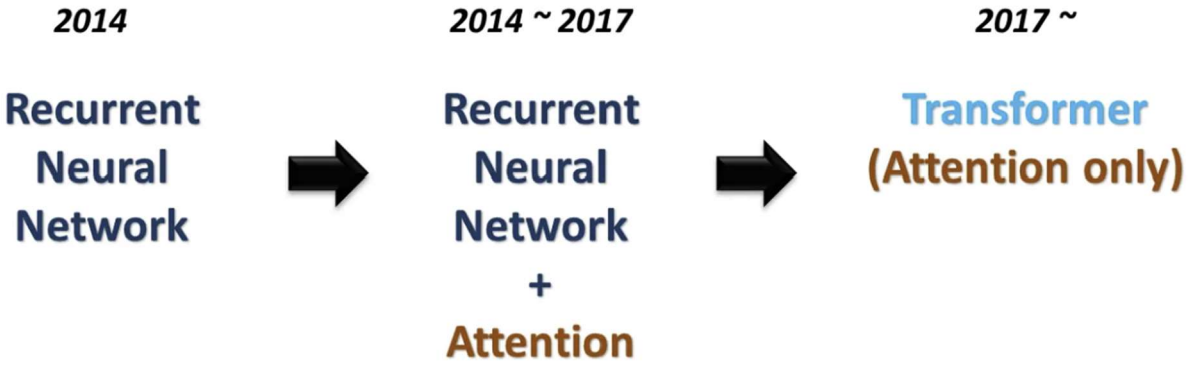
Ⅰ. 서론

1.1 Transformer 모델의 소개

AI 기술의 발전에 따라, Seq2Seq 모델을 구현하는 방법도 크게 진화해왔다. 초기 Seq2Seq모델은 주로 RNN을 사용하여 구현되었는데, RNN은 시퀀스 데이터를 처리하는데 강력한 능력을 보였으며, 많은 연구와 응용에서 중요한 역할을 했다. 그러나 RNN을 사용한 Seq2Seq 모델은 몇 가지 문제점에 직면한다. 가장 대표적인 문제는 Vanishing Gradient, Exploding Gradient이다. 이 문제는 신경망이 깊어질수록, 시퀀스가 길어질수록 Gradient가 점점 소실되거나 폭발적으로 증가하는 현상을 의미한다. 이로 인핸 모델의 학습이 어려워지거나 불안정해지는 문제가 발생한다. 또한, 장기 의존성문제로 인해 시퀀스의 앞부분과 뒷부분 사이의 연관성을 효과적으로 학습하기 어렵다는 문제점도 지적되었다.

이러한 문제점을 해결하기 위해 Attention기술이 제안되었다. 어텐션은 모델이 입력 시퀀스의 중요 부분에 집중하여 필요한 정보를 선택적으로 추출하는 방법을 제공한다. 이를 통해 모델은 전체 시퀀스를 일괄적으로 처리하는 대신 관련성이 높은 정보에 집중하여 효율적으로 처리할 수 있게 된다.

이후 Seq2Seq 구현의 패러다임을 완전히 바꾼 것이 'Transformer' 아키텍처의 등장이다. Transformer는 오로지 Attention만을 사용하여 Seq2Seq문제를 해결하는 새로운 방식을 제시했다. 복잡한 RNN구조를 제거하고, 전체 시퀀스를 한 번에 처리할 수 있는 구조로 설계되었다. 이로 인해 학습 속도가 크게 향상되었으며, 긴 시퀀스와 깊은 네트워크에 대한 처리가 가능 해졌다.



1.2 Transformer모델의 중요성 / NLP 및 기타 분야에서의 역할

Transformer는 자연어 처리 분야에서 혁신적인 발전을 이끈 딥러닝 모델 중 하나로, 2017년에 발표된 'Attention Is All You Need'논문에서 처음 제안되었다. 이 모델은 주로 번역 및 자연어 이해 작업에서 사용되며, 이전의 RNN보다 뛰어난 성능과 병렬 처리 기능을 제공합니다. Transformer는 주로 자연어 처리 분야에서 강력한 성능을 발휘하며, 문장의 의미를 파악하고, 새로운 문장을 생성하는 등의 작업에 뛰어난 능력을 보인다. 그외에도 이미치처리, 음성인식 등 다양한 AI응용 분야에서도 그 가능성을 탐색하고 있다.

Transformer모델이 소개되고 나서 GPT -> Bert -> GPT-2 -> DistillBert -> GPT-3 -> GPT-

4 다음과 같이 강력하고 우수한 모델이 추가적으로 도입되었다.

이는 Transformer가 RNN이나 CNN의 기존 틀을 부수고, 한계점을 이겨낸 것이 큰 이유라고 생각한다.

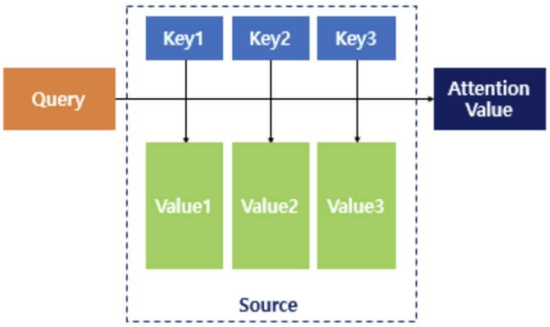
Ⅱ. 본론

1. 이론적 배경

1-1. Self-Attention

(어텐션 기본구조)

어텐션 함수는 주어진 '쿼리(Query)'에 대해서 모든 '키(Key)'와의 유사도를 각각 구한다. 그리고 구해낸 이 유사도를 가중치로 하여 키와 맵핑되어있는 각각의 '값(Value)'에 반영해준다. 그리고 유사도가 반영된 '값(Value)'을 모두 가중합하여 리턴한다.



어텐션 중에서는 셀프 어텐션이라는 것이 있다. 어텐션을 자기 자신에게 수행한다는 의미이다. 앞서 배운 seq2seq에서 어텐션을 사용할 경우의 Q,K,V의 정의는 달라진다.

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

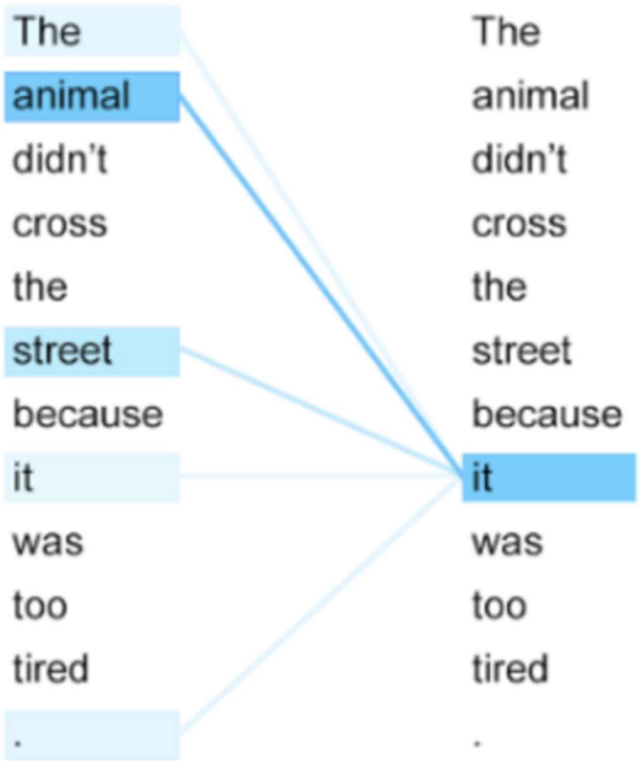
사실 t 시점이라는 것은 계속 변화하면서 반복적으로 쿼리를 수행하므로 결국 전체 시점에 대해서 일반화를 할 수도 있다.

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

이처럼 기존에는 디코더 셀의 은닉 상태가 Q이고 인코더 셀의 은닉 상태가 K라는 점에서 Q와 K가 서로 다른 값을 가지고 있다. 그런데 셀프 어텐션에서는 Q, K, V가 전부 동일하다.
트랜스포머의 셀프 어텐션에서의 Q, K, V는 아래와 같다.

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

(self-attention의 작동원리)



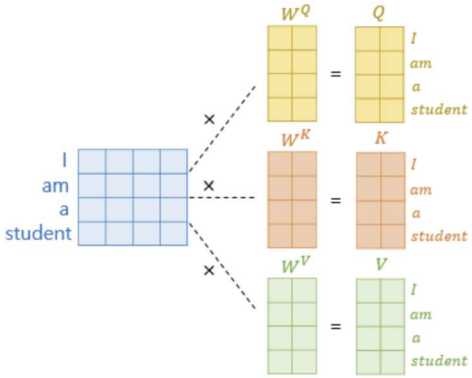
위의 예시 문장을 번역하면 '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.'라는 의미가 된다. 여기서 그것(it)에 해당하는 것은 동물이라는 것을 쉽게 알 수 있지만 기계는 그렇지 않다.

셀프어텐션에서는 입력 문장 내의 단어들끼리 유사도를 구하므로 그것(it)이 동물과 연관되었을 확률이 높다는 것을 찾아낸다.

(self-attention 의 구조)

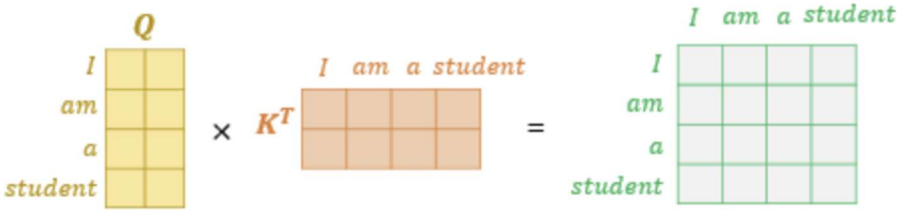
(1) Q, K, V 의 벡터 구하기

입력 시퀀스로부터 Q, K, V 벡터를 구한다. 예를 들어, 'I am a student'라는 문장이 있을 때, I, am, a, student 모든 단어의 Q, K, V 벡터를 구하게 된다.



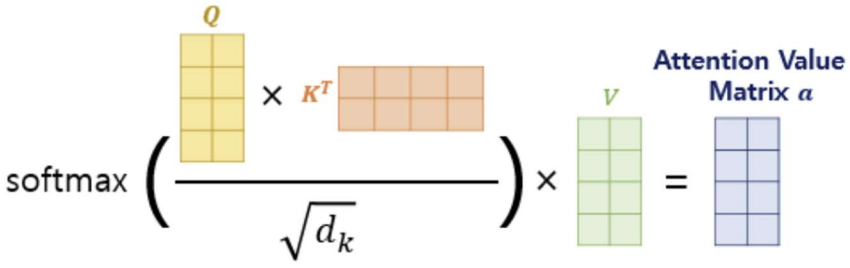
(2) Scaled dot-product Attention

Q, K, V 벡터를 얻었다면, 이를 이용하여 Attention score 를 구할 것이다. 기존에 Attention 구조에서는 score(q,k) = q*k 식으로 score 를 구한다. Self-attention 은 score(q,k) = q*k/√n 를 사용한다.



(3) Softmax / 가중합 곱하기

score 에 소프트맥스 함수를 사용하여 어텐션 분포(Attention Distribution)을 구하고, 각 V 벡터와 가중합하여 어텐션 값(Attention Value)을 구한다.

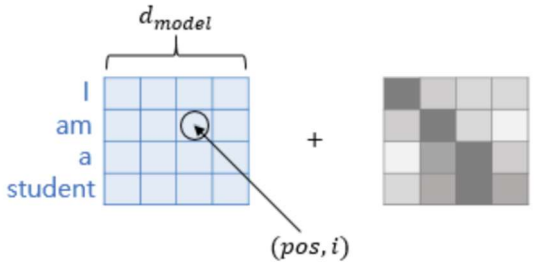


$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

이 구조를 통함으로써 입력 시퀀스의 모든 부분 간의 상호작용을 고려한다.
단어 행렬을 통해서 각 단어끼리 상호작용을 할 수 있다.

1-2. Positional Encoding

RNN 이 자연어 처리에서 유용했던 이유는 단어의 위치에 따라 단어를 순차적으로 입력받아서 처리하는 RNN 의 특성으로 인해 각 단어의 위치 정보를 가질 수 있다는 점에 있다. 하지만 트랜스포머는 단어 입력을 순차적으로 받는 벡터방식이 아닌 전체적인 문장을 행렬구조로 받기 때문에 위치정보를 다른 방식으로 알려줄 필요가 있다. 트랜스포머는 단어 위치정보를 얻기 위해서 각 단어의 임베딩 벡터에 위치 정보를 더하여 모델의 입력으로 사용하는데, 이를 Positional encoding 이라고 한다.



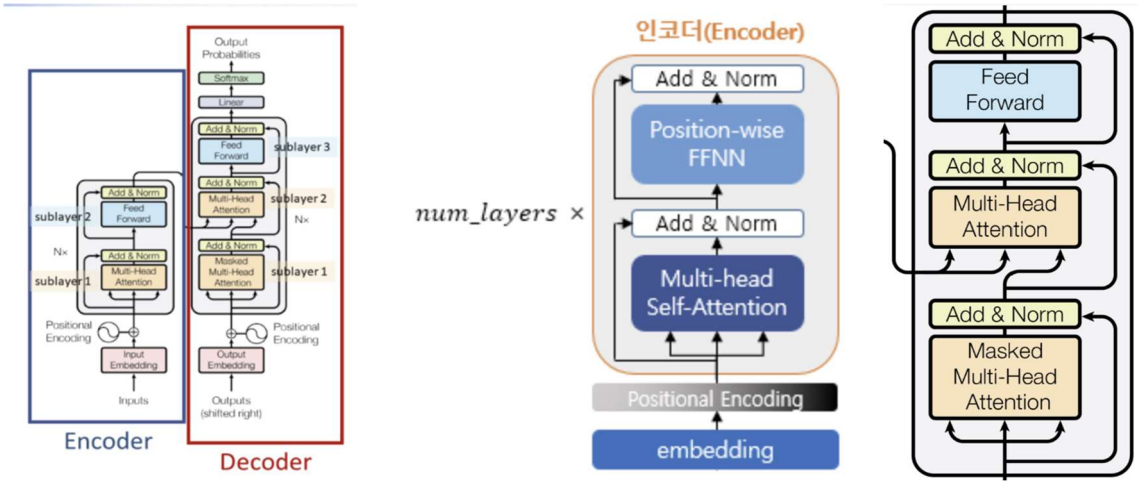
$$PE_{(pos,\,2i)} = sin(pos/10000^{2i/d_{model}})$$

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

사인 함수와 코사인 함수의 그래프를 보면 요동치는 값의 형태를 생각해볼 수 있는데, 트랜스포머는 사인함수와 코사인 함수의 값을 임베딩 벡터에 더해주므로서 단어의 순서 정보를 더한다. 짝수인 경우에는 사인 함수의 값을 사용하고 홀수인 경우에는 코사인 함수의 값을 사용한다.
pos 는 입력문장에서의 임베딩 벡터의 위치를 나타내며, i 는 임베딩 벡터 내의 차원의 인덱스를 의미한다.

2. Transformer 구조 분석

2-1. Encoder 와 Decoder



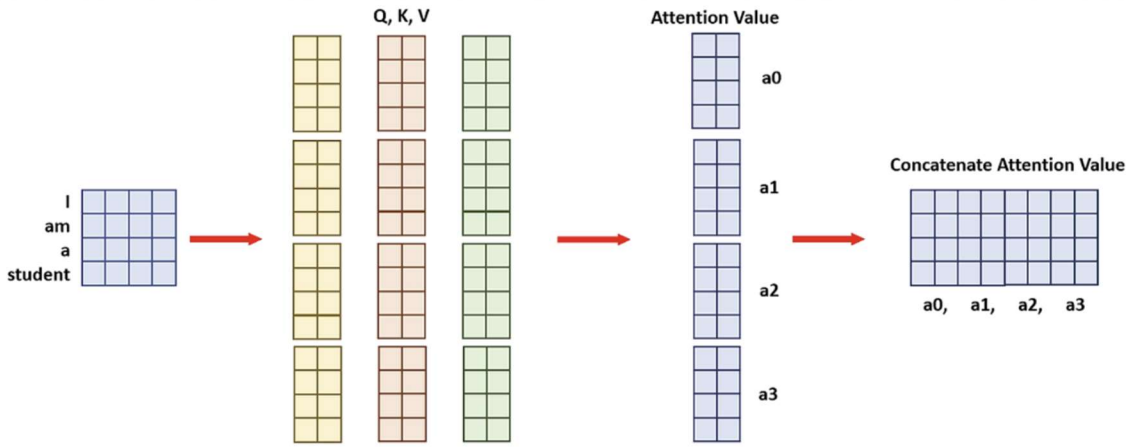
(1) Encoder
하나의 인코더 층이 두개의 서브 층으로 이루어진다.

1) self-Attention : multi-head 블록으로 self-attention 을 병렬적으로 사용
2) FFNN : position-wise FFNN 블록으로 일반적인 신경망
중간에 Add/Norm(잔여 학습/정규화) : 입력값과 출력값을 합함으로써 한 계층에서 과도하게 데이터가 변경되는 부분을 방지

(2) Decoder
1) Masked Multihead Attention : 이전에 설명한 self-Attention 방식에서 Masking 작업만 추가한것. Decoder 의 입력값으로 Attention 과정을 수행하는데, 이때 정답이 입력이 되기 때문에 입력을 masking 함으로써 과적합으로 인한 성능저하를 막기 위함이다.
2) Multi-Head : Encoder 와 동일하게 self-attention 을 병렬적으로 사용
3) FFNN : position-wise FFNN 블록으로 일반적인 신경망
중간에 Add/Norm(잔여 학습/정규화) 진행

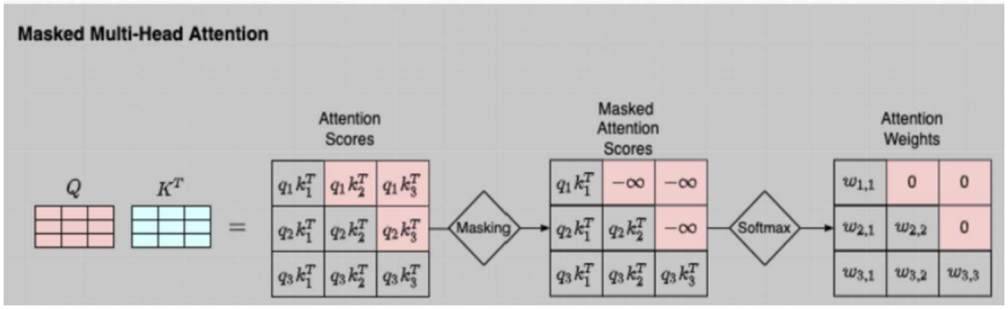
2-2. Multi-head Attention

(1) Multi-head Attention



트랜스포머 연구진은 한 번의 어텐션을 하는 것보다 여러 번의 어텐션을 병렬로 사용하는 것이 더 효과적이라고 판단했다. 그래서 num_heads 개로 나누어 Q,K,V 에 대해서 num_heads 개의 병렬 어텐션을 수행한다. 이때 각각의 어텐션 값을 행렬을 어텐션 헤드라고 부른다. 가중치 행렬 W(Q), W(K), W(V)의 값은 num_heads 의 어텐션 헤드마다 전부 다르다. 어텐션 해드 값을 통해, 각 Attention value 값을 도출한다. 도출된 Attention value 값을 concatenate 하여 최종 Attention value 를 도출한다.
예를 들면, '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.'라는 문장에서 그것(it)이 쿼리였다고 하자. 즉, it 에 대한 Q 벡터로부터 다른 단어와의 연관도를 구하였을 때 첫번째 어텐션 헤드는 '그것(it)'과 '동물'의 연관도를 높게 본다면, 두번째 어텐션 헤드는 '그것(it)'과 '피곤하였기 때문이다.'의 연관도를 높게 볼 수 있다. 각 어텐션 헤드는 전부 다른 시각에서 보고있기 때문이다.

(2) Masked Multihead-Attention



Masked Multihead-Attention 은 Decoder 부분에서 들어간다.
Self-Attention 방식에서 Masking 작업만 추가한 것으로 생각하면 된다. Decoder 의 입력값으로 Attention 과정을 수행하는데 이때 문제점이 문장 전체 입력값을 받기 때문에 과거 시점의 입력값을 예측할 때 미래 시점의 입력 값 까지 참고한다는 점이다. 쉽게 말하면 정답이 입력된 값이기 때문에 과적합이 일어날 수 있다. 이 문제점을 해결하기 위해서 마스킹을 한다. 마스킹 작업이란 말그대로 가린다는 뜻이다. Look-ahead Mask 사용 후 Softmax 를 취하여 미래 시점의 입력 값까지 참고한다. Attention 기법으로 단어가 유사도를 계산한 결과에서 뒤쪽 부분을 0(Zero Weight)으로 만들어주는 방법이다. 이렇게 출력된 가중치를 Encoder 에서 나온 출력값과 비교하면서 학습하는 과정을 거지게 된다.

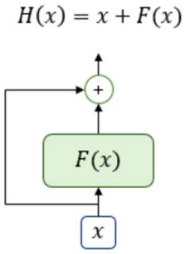
2-3. Feed-Forward Networks

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

FFNN 은 인코더와 디코더에서 공통적으로 가지고 있는 서브층이다.
FFNN(Fully-connected FFNN)이라고 해석할 수 있다.
위의 수식에서 x 는 앞서 멀티 헤드 어텐션의 결과로 나온 (seq_len, d(model))의 크기를 가지는 행렬을 말한다. 가중치 행렬 W1 은 (d(model), d(ff)의 크기를 가지고, 가중치 행렬 W2 은(d(ff),d(model))의 크기를 가집니다. 여기서 W1, b1, W2, b2 는 하나의 인코더 층 내에서는 다른 문장, 다른 단어들마다 정확하게 동일하게 사용된다. 하지만 인코더 층마다는 다른 값을 가진다. 위의 그림에서 좌측은 인코더의 입력을 벡터 단위로 봤을 때, 각 벡터들이 멀티 헤드 어텐션 층이라는 인코더 내 첫번째 서브 층을 지나 FFNN 을 통과하는 것을 보여줍니다. 이는 두번째 서브층인 Position-wise FFNN 을 의미한다. 물론, 실제로는 그림의 우측과 같이 행렬로 연산되는데, 두번째 서브층을 지난 인코더의 최종출력은 인코더의 입력의 크기였던 (seq_len, d(model))의 크기가 보존된다. 하나의 인코더 층을 지난 이 행렬은 다음 인코더 층으로 전달되고, 다음 층에서도 동일한 인코더 연산이 반복된다.

2-4 Residual Connections & Layer Normalization

(1) Residual Connections (잔차 연결)

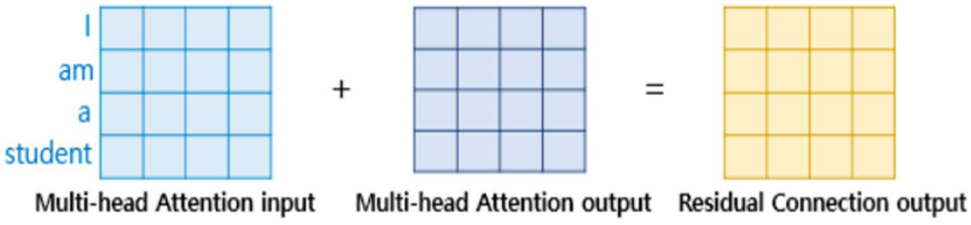


(잔차 연결 그림 1 번)

위 그림은 입력 x 와 함수 F(x)의 값을 더한 함수 H(x) 구조를 보입니다.
여기서 함수 F(x)는 Transformer 서브층에 해당됩니다.
그림에 보이듯 잔차 연결은 서브 층의 입력과 출력을 더하는 것입니다.

Transformer 에서 행렬의 연산으로 Multi-Head Attention, FFNN 등을 구하고 마지막에 가중치 값인 W 값을 구해서 입력 값과 출력 값을 동일한 차원을 가지게 만들어줍니다.
그로 인해, 잔차 연결에서는 입력 값과 출력 값이 동일한 차원을 가지고 있어 더하기(+) 연산이 가능하게 됩니다.

식으로 표현한다면 x + sublayer(x) 입니다.
서브 층이 Multi-Head Attention 이라면, 아래의 수식과 같습니다.
H(x) = x + Multi-Head Attention(x)



(잔차 연결 그림 2 번)

서브 층이 Multi-Head Attention 이라면, 잔차 연결은 위 그림과 동일합니다.

(2) Layer Nomalization (층 정규화)

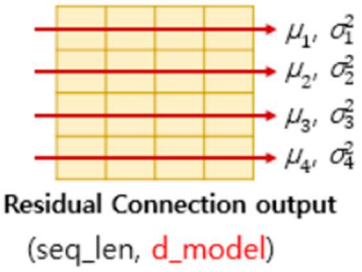
잔차 연결의 입력 값을 x, 잔차 연결과 층 정규화를 모두 거친 결과 값을 LN 이라고 할 경우, 잔차 연결 후 층 정규화 연산을 수식으로 표현하면 아래와 같습니다.

$$LN = LayerNorm(x + Sublayer(x))$$

(층 정규화 공식 1 번)

층 정규화는 Tensor 의 마지막 차원에 대해서 평균과 분산을 구하고, 이를 가지고 어떤 수식을 통해 값을 정규화 하여 학습에 도움을 주게 됩니다.

Tensor 의 마지막 차원이라는 것은 Transformer 의 dmodel 차원을 의미합니다.
아래 그림은 dmodel 차원의 방향을 화살표로 표현하였습니다.



(층 정규화 그림 1 번)

층 정규화를 위해 먼저 화살표 방향으로 각 평균 u 와 분산 $\sigma 2$ 을 구합니다.
여기서 각 화살표의 방향을 벡터를 xi 라고 칭했을 때

(층 정규화 그림 2 번)

층 정규화를 수행한 후에 벡터 xi 는 ln_i 벡터로 정규화 됩니다.
수식으로 표현하자면 $ln_i = LayerNorm(x_i)$ 과 같습니다.

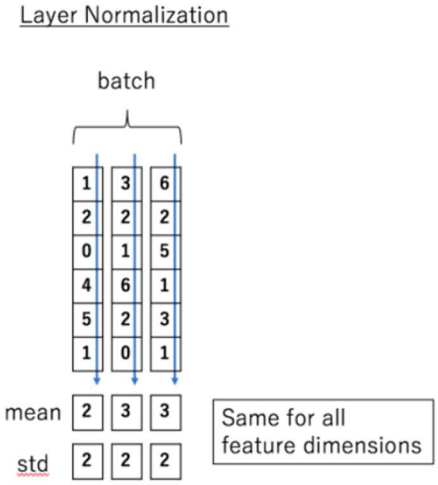
층 정규화를 2 가지 과정으로 설명할 수 있습니다.
1. 평균과 분산을 통한 정규화
2. 감마와 베타를 도입

이를 통해 도출된 최종 수식은 아래 그림과 같습니다.

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

(층 정규화 그림 3 번)

Layer Normalization 의 경우 각 input 의 Feature 에 대한 평균과 분산을 구해 batch 에 있는 각 input 을 정규화합니다.



(층 정규화 그림 4 번)

LN 은 각 input(=sample)에 대해서만 처리되므로 Batch Size 와는 전혀 상관 없게 됩니다.

장점

1. Mini-batch 의 크기에 영향을 받지 않는다.
2. 서로 다른 길이를 갖는 Sequence 가 batch 단위로 들어오는 경우에도 각각 다른 Normalization Term 을 가지기 때문에 적용이 가능하다.
3. 데이터마다 각각 다른 Normalization Term(감마, 베타)를 갖는다.

Ⅲ. 참고문헌

Wikidoc, 'transformer 구조', https://wikidocs.net/31379 ,2024-06-04

medium, 'transformer 구조' , https://medium.com/@hugmanskj/transformer%EC%9D%98-%ED%81%B0-%EA%B7%B8%EB%A6%BC-%EC%9D%B4%ED%95%B4-%EA%B8%B0%EC%88%A0%EC%A0%81-%EB%B3%B5%EC%9E%A1%ED%95%A8-%EC%97%86%EC%9D%B4-%ED%95%B5%EC%8B%AC-%EC%95%84%EC%9D%B4%EB%94%94%EC%96%B4-%ED%8C%8C%EC%95%85%ED%95%98%EA%B8%B0-5e182a40459d , 2024-06-04

tistory, 'transformer 구조', https://for-data-science.tistory.com/71 ,2024-06-04

tistory, 'transformer 구조', https://skyil.tistory.com/256 , 2024-06-04

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, Attention Is All You Need, arXiv, 2017.6.17,p1~p15