ONECLICK AI

Transformer

연준모

목차

1. Transformer

Transformer?

저번주에 링크만 남겼었는데, 이거 아무도 안봤을거 아니냐 꼭 알아야 하는 논문이기에 다루려고 한다

Attention is all you need

https://arxiv.org/abs/1706.03762

진짜 개명작 논문

순서의 족쇄를 끊고, **문맥의 시대**를 열다

그저 GOAT THE TRANSFORMER

AI계의 유일신

1. Transformer

Transformer?

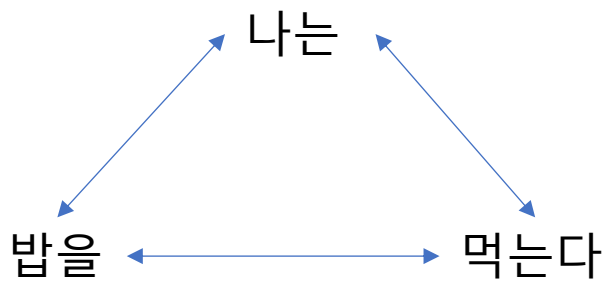
이전에는 RNN, LSTM이 최고였다.

순차적 계산 (Sequential Computation) : "나는 밥을 먹는다"라는 문장을 "나는" → "밥을" → "먹는다" 순서로, 단어를 **하나씩 순차적으로** 처리하였다.
t 시점의 계산은 t-1 시점의 계산이 끝나야만 가능했다. 이는 GPU의 강력한 병렬 처리 능력을 전혀 활용할 수 없어 학습 속도가 매우 느렸다.

치매 문제 (Long-Range Dependencies) : 문장이 길어지면, "나는"이라는 첫 단어의 정보가 "먹는다" 라는 끝 단어까지 전달될 때쯤이면 정보가 희미해지거나 소실(vanishing gradient)될 위험이 컸다.

Transformer?

만약, 문장을 순차적으로 보지 않고, **모든 단어를 한꺼번에 펼쳐놓고** 각 단어가 다른 모든 단어와 **직접 상호작용**하게 만들면 어떨까?



순환(Recurrence) 구조를 아예 없애고, 오직 '어텐션(Attention)' 메커니즘만으로 더 빠르고 더 성능 좋은 모델을 만들 수 있다.

1. Transformer 2. 트랜스포머의 전체 구조

Transformer?

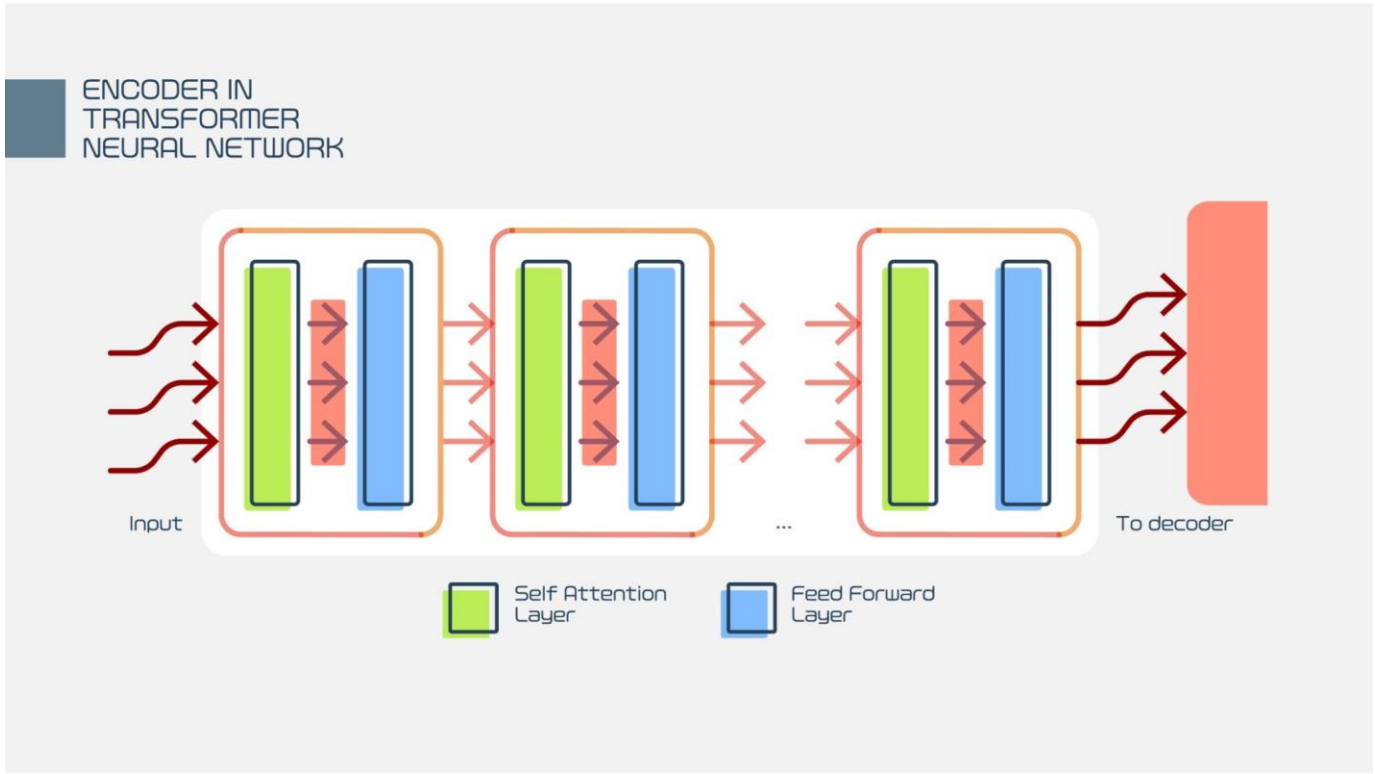
트랜스포머는 기계 번역을 위해 설계되었으며, 크게 두 부분으로 나뉜다

인코더 (Encoder):
입력 문장(예: 한국어)을 받아서, 문장의 의미와 문맥을 "이해"
하는 숫자 벡터(표현)를 만든다.
디코더 (Decoder):
인코더가 압축한 문맥 정보를 받아서, 출력 문장(예: 영어)을
한 단어씩 생성한다.

논문에서는 인코더와 디코더를 각각 6개씩 쌓아 올린 구조
(Stack)를 제안한다

Transformer?

트랜스포머는 기계 번역을 위해 설계되었으며, 크게 두 부분으로 나뉜다



1. Transformer 3. 토크나이징

Transformer?

입력될 데이터는 텍스트. 하지만, 컴퓨터는 숫자만 알아먹는다.

따라서 텍스트를 숫자로 만드는데, 이걸 토크나이징이라고 한다.
https://gihak111.github.io/ai/2025/08/29/AI_basics_1_upload.html

그 과정에서 토큰 임베딩을 하는데,

먼저 문장을 토큰(단어 또는 하위 단어)으로 쪼갠다.

각 토큰을 고유한 벡터로 변환한다. (예: "student" → `[0.1, 0.7, -0.2, ...]`)
이 벡터는 단어의 '기본 의미'를 담고 있다.

하지만, RNN을 버리면서 심각한 문제가 발생한다.
"나는 너를 사랑해"와 "사랑해 너를 나는"을 한 번에 펼쳐놓고 보면, 어텐션 메커니즘은 이 둘을 **똑같은 문장**으로 인식한다.
순서 정보가 사라졌기 때문이다.

Transformer?

이 문제를 해결하기 위해, **위치 인코딩 (Positional Encoding)**을 진행한다

단어의 '의미' 벡터에, 단어의 '위치' 정보가 담긴 벡터를 더해준다.

$$\text{InputVector} = \text{TokenEmbedding} + \text{PositionalEncoding}$$

논문에선 고정된 사인(sin)과 코사인(cos) 함수를 사용한다.

$$PE_{(pos,2i)} = \sin\left(pos/10000^{2i/d}\text{model}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d}\text{model}\right)$$

Pos : 0부터 시작하는 토큰의 위치(인덱스).

I : 512차원 임베딩 벡터 내의 차원 인덱스.

1. Transformer 3. 토크나이징

Transformer?

$$PE_{(pos,2i)} = \sin\left(pos/10000^{2i/d}\text{model}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(pos/10000^{2i/d}\text{model}\right)$$

Pos : 0부터 시작하는 토큰의 위치(인덱스).

I : 512차원 임베딩 벡터 내의 차원 인덱스.

왜 이 복잡한 식을 쓸까?

고유성 : 각 위치(`pos`)마다 고유한 PE 벡터가 생성

상대 위치 학습 : 임의의 오프셋 k에 대해 PE_{pos+k}가 PE_{pos}의 **선형 변환**으로 표현될 수 있다는 점
=> 이는 모델이 "두 단어가 3칸 떨어져 있다"와 같은 **상대적 위치 관계**를 매우 쉽게 학습할 수 있게 한다

외삽 : 학습 때 본 문장 길이(예: 50)보다 더 긴 문장(예: 100)이 들어와도 위치 값을 생성할 수 있다.

논문에서는 sin 과 cos 함수를 사용하는데, 이는 각 위치(pos)마다 고유한 벡터를 생성하고, sin 과 cos 의 주기적 특성 덕분에 모델이 **상대적인 위치 관계($pos + k$가 pos와의 관계)**를 매우 쉽게 학습할 수 있게 한다

Self Attention

이제 '의미'와 '순서' 정보가 모두 담긴 벡터가 인코더에 입력된다.

인코더의 목표는 **문맥을 깊게 이해**하는 것이다

"I went to the **bank**..."라는 문장에서 "bank"가 '은행'인지 '강둑'인지 알아내야 한다

이를 위한 **Self-Attention**

Self Attention

기존의 어텐션
Query (Q) : 내가 지금 당장 궁금한 것. (예: "머신러닝의 역사")
Key (K) : 도서관에 있는 모든 책의 '키워드' 또는 '색인'. (예: "머신러닝", "AI", "역사"...)
Value (V) : 책의 '실제 내용'.

어텐션은
(1) 내 Query로 모든 Key와 유사도를 비교하고,
(2) 유사도(가중치)가 높은 Key의 Value(실제 내용)를 가져와
(3) 가중합하여 나에게 꼭 맞는 '맞춤형 정보'를 만드는 과정이다

Self-Attention에서 Q, K, V는 모두 **자기 자신(입력 문장)**에서 나온다.
Query (Q) - "질문": "bank"가 자신의 문맥적 의미를 명확히 하기 위해 다른 단어들에게 던지는 질문이다.
(예: "얘들아, 나 'bank'인데, 나랑 '돈(money)' 관련된 단어 있니, 아니면 '강(river)' 관련된 단어 있니?")
Key (K) - "간판": 다른 모든 단어(자신 포함)가 자신의 정체성을 광고하는 '키워드 ' 이다.
(예: "I"는 '나', "river"는 '강'이라는 키워드를 내건다.)
Value (V) - "본질": 각 단어가 실제로 가지고 있는 '의미' 또는 '정보 ' 이다.

Self Attention

Q, K, V는 어디서 오는가? (WQ, WK, WV)

원래의 입력 벡터(x)에서 학습 가능한 가중치 행렬 W^Q, W^K, W^V를 곱해서 생성 된다.

$q_i = x_i \cdot W^Q$ (내 i번째 단어 x_i로 '질문'을 만드는 법을 배운다)
$k_i = x_i \cdot W^K$ (내 i번째 단어 x_i로 '간판'을 만드는 법을 배운다)
$v_i = x_i \cdot W^V$ (내 i번째 단어 x_i로 '본질'을 만드는 법을 배운다)

이 W 행렬들이야말로 트랜스포머가 **훈련 과정에서 학습하는 핵심이**다.
모델은 어떤 질문(Q)을 던져야 문맥 파악에 유리한지, 어떤 간판(K)을 내걸어야 하는지를 배운다.

Self Attention

식은 같다.

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

1. $\boldsymbol{QK^T}$ **(유사도 계산):**
   Query 행렬과 Key 행렬을 내적(Dot-Product)한다.
   `bank`의 Query(q_{bank})가 `river`의 Key(k_{river})와 내적된다.
   이는 문장 내 모든 단어(Q)가 다른 모든 단어(K)와 얼마나 유사한지를 나타내는 **유사도 점수 행렬**이다.

Self Attention

2. $\frac{\dots}{\sqrt{d_k}}$ **(스케일링 - "Scaled"):**
   d_k는 Key 벡터의 차원(논문에서는 64)이다.
   이유: d_k가 커질수록, QK^T의 내적값(유사도 점수)이 너무 커지는 경향이 있다.
   이 값이 너무 크면, 다음 단계인 Softmax 함수에서 그래디언트가 0에 가까워지는
   포화 영역(saturation region)으로 들어가 버린다.
   그래디언트가 0이 되면 모델이 학습을 멈추게 된다(Vanishing Gradient).
   따라서 d_k의 제곱근($\sqrt{64} = 8$)으로 나눠줌으로써 값을 안정화시켜 **학습이 원활하게** 되도록 한다.

3. $\mathbf{softmax}(\dots)$ **(가중치 정규화):**
   안정화된 유사도 점수에 Softmax 함수를 적용한다.
   결과: 각 행의 합이 1이 되는 **어텐션 가중치(Attention Weights)**가 나온다.
   (예: "bank" → I: 0.05, ... **river: 0.7**, ...)

Self Attention

4. **...V (가중합):**
   이 어텐션 가중치 행렬을 **Value** 행렬 V와 곱한다.
   $z_{\text{bank}} = (0.05 \times v_I) + \cdots + (0.7 \times v_{\text{river}}) + \cdots$

   최종 결과 (z_{bank}): 이제 "bank"의 새로운 벡터 z_{bank}는 "river"의 의미(Value)가 70%나 섞여 들어간, **"강둑(river-bank)"이라는 문맥적 의미**를 가진 벡터로 재탄생한다.

1. Transformer 4. Self Attention

Self Attention 종합해서 보자

입력 시퀀스의 각 요소가 **같은 시퀀스 내의 다른 모든 요소**와 얼마나 관련되어 있는지 계산

작동 방식:

1. 하나의 입력 시퀀스(예: 문장)가 들어온다.
2. 시퀀스의 각 단어(위치)는 **Q**, **K**, **V** 세 벡터를 **모두 자기 자신**으로부터 생성한다.
3. 특정 단어의 **Q**를 시퀀스 내 다른 모든 단어의 **K**와 비교하여 관계 점수를 얻는다.
4. 이 점수를 이용해 모든 **V**를 가중합하여, 문맥 정보가 풍부하게 담긴 **새로운 벡터**를 만들어 낸다.

비유 : 문장 내의 단어 **"그것(it)"**이 문장 내의 다른 단어(예: **"강아지"**) 중 누구를 가리키는지 스스로 파악하여 의미를 명확히 하는 과정과 같다.

핵심 : 하나의 시퀀스 내부의 단어들 간의 관계를 파악하여 문맥을 이해한다.

1. Transformer 4. Self Attention

Attention 까먹었을 까봐

초기 **Seq2Seq 모델(RNN 기반 인코더-디코더)**에서 성능을 향상을 위해 도입.
주로 **기계 번역**과 같은 작업에 사용

작동 방식:

1. 번역할 문장(소스 시퀀스)을 인코더가 모두 처리하여 **K**와 **V**를 만든다.
2. 번역될 단어(타겟 시퀀스)를 생성하는 디코더의 현재 상태가 **Q**가 된다.
3. 이 **Q**를 모든 **K**와 비교하여 가장 관련 있는 **K** 지점을 찾고, 해당 위치의 **V**에 높은 가중치를 부여합니다.

비유 : 번역가가 외국어 문장 전체를 읽고, 현재 번역할 단어에 해당하는 원문의 ****특정 부분****에 집중하는 것과 같다.

핵심 : 인코더 출력 (소스)과 **디코더 입력 (타겟)**이라는 **두 시퀀스**를 연결한다.

Self Attention

Attention VS Self Attention

구분	일반 어텐션 (Attention)	셀프 어텐션 (Self-Attention)
목적	**두 시퀀스** 간의 관계 (정렬, Alignment) 파악	**하나의 시퀀스 내부**의 관계 (문맥) 파악
사용 위치	주로 **인코더-디코더** 구조의 디코더 부분 (Cross-Attention)	주로 **인코더 및 디코더** 내부 (Intra-Attention)
Q, K, V **출처**	$\mathbf{Q}$ (Query): **타겟 시퀀스** (디코더 출력)	$\mathbf{Q}, \mathbf{K}, \mathbf{V}$: **동일한 시퀀스** (자신)
	$\mathbf{K}, \mathbf{V}$ (Key, Value): **소스 시퀀스** (인코더 출력)	
대표 모델	RNN 기반 Seq2Seq 모델 (기계 번역 초기)	**Transformer, BERT, GPT** 등 현대 모델
핵심 역할	소스 문장에서 타겟 단어 생성에 가장 관련 높은 부분을 찾아 **집중**	문장 내에서 단어가 다른 단어들과 얼마나 관련 있는지 파악하여 **풍부한 문맥 벡터** 생성

Multi Head Attention

한 번의 Self-Attention은 한 가지 관점(예: 'bank'와 'river'의 관계)만 볼 수 있다

한 명의 전문가가 문장을 보는 것보다, 8명의 서로 다른 전문가(Head)가 각자 다른 관점으로 동시에 보면 어떨까?

헤드 1: "bank"와 "river"의 **의미적 관계**에 주목할 수 있다.
헤드 2: "I"와 "went"의 **문법적 관계**(주어-동사)에 주목할 수 있다.
헤드 3: "to"와 "the"의 **위치 관계**에 주목할 수 있다.

Multi Head Attention

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

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

어렵죠?

1. **프로젝션 (Projection):**
   h개(예: 8개)의 헤드 각각에 대해 서로 다른 가중치 행렬(W_i^Q, W_i^K, W_i^V)을 준비한다. 원본 Q, K, V를 이 행렬들에 곱해 각 헤드를 위한 저차원(예: 64차원)의 Q_i, K_i, V_i를 만든다.
2. **병렬 어텐션:**
   8개의 헤드가 각각 독립적으로, 동시에(병렬로) Scaled Dot-Product Attention을 수행한다. (head_1 ~ head_8 계산)
3. **연결 (Concatenation):**
   8개의 헤드에서 나온 64차원 출력 벡터들을 모두 **연결**하여 다시 512차원(8×64) 행렬을 만든다.
4. **최종 프로젝션:**
   이 연결된 행렬에 또 다른 가중치 행렬 W^O를 곱하여 최종 MHA 출력을 얻는다.

Multi Head Attention 더 자세히 알아보자

1단계: 선형 변환 및 병렬 어텐션 수행

1. **입력 분할 (Projection):** 입력으로 들어온 **쿼리 (Query), 키 (Key), 값 (Value)** 벡터를 여러 개의 **헤드 (Head)** 수만큼 분할한다

   - 각 헤드 (i)는 독립적인 **선형 변환** 가중치 행렬 (W_i^Q, W_i^K, W_i^V)을 사용하여 Q, K, V를 더 작은 차원으로 투영한다

   - 예를 들어, 모델 차원 d_{model}을 사용하고 헤드 수 H를 사용한다면, 각 헤드의 차원은 $d_k = d_{\text{model}}/H$가 된다

$$Q_i = QW_i^Q, K_i = KW_i^K, V_i = VW_i^V$$

2. **독립적인 어텐션 계산:** 각 헤드 (i)는 투영된 Q_i, K_i, V_i를 사용하여 일반적인 **스케일드 닷 프로덕트 어텐션 (Scaled Dot-Product Attention)**을 독립적으로 수행한다.

$$Head_i = \text{Attention}(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

Multi Head Attention 더 자세히 알아보자

2단계: 결과 결합 (Concatenation)

- 각 헤드에서 독립적으로 계산된 H개의 출력 값 $(Head_1, Head_2, \dots, Head_H)$**을 원래의** $\boldsymbol{d}_{\mathbf{model}}$ **차원으로 복원하기 위해 하나로 이어 붙인다**

$$\text{Concat}(Head_1, \dots, Head_H)$$

3단계: 최종 선형 변환

- 이어 붙인 최종 결과에 또 하나의 **선형 변환** 가중치 행렬 (W^O)을 곱하여 최종 출력을 만들어 낸다.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(Head_1, \dots, Head_H)W^O$$

Multi Head Attention 더 자세히 알아보자

개념	설명	역할
헤드 (Head)	독립적인 셀프 어텐션 모듈 하나를 지칭.	시퀀스 내의 **하나의 특정 관계/관점**을 학습.
선형 변환	Q, K, V에 각 헤드마다 다른 가중치 행렬 (W^Q, W^K, W^V)을 곱하는 과정.	각 헤드가 서로 **다른 초기화 상태**에서 출발하도록 하여, **다양한 관점**을 학습하게 함.
병렬 처리	여러 개의 헤드가 동시에 어텐션을 계산하는 방식.	**계산 효율**을 높이고 **다양성**을 확보.
결과 결합	각 헤드의 출력을 이어 붙여 원래 차원으로 복구.	다양한 관점의 정보를 **하나의 종합적인 표현**으로 취합.

Multi Head Attention 더 자세히 알아보자

하나의 어텐션 : 단 하나의 관점으로만 문맥을 파악합니다.

멀티 헤드 어텐션 : 여러 개의 독립적인 어텐션(헤드)을 통해 다양한 관점과 관계를 동시에 파악하고, 그 결과를 결합하여 **더 풍부하고 강력한 문맥 표현 벡터**를 생성

결과: 하나의 벡터 안에 문법, 의미, 위치 등 **다양한 관점의 문맥 정보가 풍부하게** 압축된다.

1. Transformer 6. FFN

Feed Forward Network

Multi-Head Attention을 통과하며 얻은 문맥 정보를 한 번 더 **비선형적으로 변환하고 정제**하여
표현력을 높이는 역할이다.
각 토큰의 위치(Position)마다 **독립적으로** 적용되는 일반적인 신경망(MLP)이다.

? 그냥 Dense Layer 아닌가요?

Feed Forward Network

FFN은 두 개의 Dense Layer 로 구성되어 있다

1단계: 확장 (Expand)

입력 벡터 $\mathbf{x}$는 첫 번째 Dense Layer(W_1)를 통과

이때 입력 차원 d_{model}은 보통 4배 더 큰 차원 d_{ff}로 확장(Expand) (예: $512 \rightarrow 2048$)

여기에 **ReLU**와 같은 비선형 활성화 함수가 적용된다.

$$\mathbf{z} = \text{ReLU}(\mathbf{x}W_1 + \mathbf{b_1})$$

Feed Forward Network

FFN은 두 개의 Dense Layer 로 구성되어 있다

2단계: 축소 (Contract)

확장된 벡터 $\mathbf{z}$는 두 번째 Dense Layer(W_2)를 통과

이 층은 차원을 원래의 d_{model}로 **축소(Contract)**시켜 출력

$$\text{FFN}(\mathbf{x}) = \mathbf{z}W_2 + \mathbf{b_2}$$

하나의 층(Layer)이 아니라 두 개의 Dense Layer로 이루어진 작은 네트워크(Sub-network)

Add & Norm

인코더와 디코더의 모든 하위 층(MHA, FFN) 주위에는 `Add & Norm`이라는 두 가지 장치가 필수로 들어간다

Input → Sublayer(MHA or FFN) → Add&Norm → Output

Add (잔차 연결, Residual Connection):

$$\text{Output} = x + \text{Sublayer}(x)$$

하위 층의 입력 x를 하위 층의 출력 $\text{Sublayer}(x)$에 **그대로 더해준다.**
이유:
x라는 원천 정보가 그대로 다음 층으로 전달되는 고속도로 역할을 한다. 층이 6개, 12개, 100개로 깊어지더라도 정보가 소실(Vanishing Gradient)되는 것을 막아 **매우 깊은 모델의 학습을 가능하게** 하는 핵심 장치이다.

Add & Norm

Norm (계층 정규화, Layer Normalization):

$$\text{Output} = \text{LayerNorm}\big(x + \text{Sublayer}(x)\big)$$

이유:
'Add'로 더해진 값들의 스케일이 널뛰는 것을 방지하고 학습을 안정화시킨다.
+ 배치 정규화(Batch Norm)는 문장 길이가 가변적인 NLP에서 불안정할 수 있어, 배치 크기와 무관하게 각 토큰 벡터의 특성($d\_\{\text{model}\}$)을 기준으로 정규화하는 계층 정규화(Layer Norm)를 사용한다

1. Transformer 8. Encoder

Encoder Layer

인코더는 6개의 동일한 레이어로 구성되며, 각 레이어는 2개의 하위 층을 가진다

1. **Multi-Head Self-Attention** (Q, K, V가 모두 인코더의 이전 층 출력)
2. **Add & Norm**
3. **Position-wise Feed-Forward Network (FFN)**
4. **Add & Norm**

1층의 출력이 2층의 입력으로 들어간다.
6층을 거친 최종 출력은 "**입력 문장의 모든 문맥을 완벽하게 이해한**" 벡터들의 리스트(메모리)가 된다

Masked Multi-Head Self-Attention

디코더의 첫 번째 하위 층. 기본적인 **멀티 헤드 셀프 어텐션**과 작동 원리는 같지만,
한 가지 중요한 제약 사항인 **마스킹(Masking)**이 추가된다.

왜 마스킹이 필요할까?
디코더의 역할은 출력을 **순차적으로 하나씩 생성**하는 것 (예: "나는" 다음에 "밥을"을 예측).

학습 과정의 문제:
트랜스포머는 병렬 연산을 수행 ->
만약 마스킹이 없다면, 디코더는 현재 예측해야 할 단어(**현재 시점**)를 계산할 때,
이미 시퀀스에 포함된 **미래의 단어들(다음 시점 이후)**까지 보게된다

정보 유출 (Information Leakage):
미래의 정보를 미리 알고 예측하는 것은 일종의 **부정 행위**이며, 모델이 실제 서
비스 환경(순차적으로 단어를 생성해야 함)에서 제대로 작동하지 않게 만든다.

Masked Multi-Head Self-Attention

작동 방식 (마스킹 추가)

1. **Q, K, V 생성:**
   입력(이전 디코더 층 출력)으로부터 $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$를 생성하고 멀티 헤드 방식으로 분할하는 것은 일반 셀프 어텐션과 동일하다
2. **어텐션 점수 계산 및 마스킹:**
   $\mathbf{Q}$와 $\mathbf{K}$를 내적하여 어텐션 점수를 계산한다.
   이때, 점수 행렬에서 **현재 시점 이후의 모든 미래 단어 위치**에 해당하는 값들을 $-\infty$ (매우 작은 음수)로 변경한다.
3. **Softmax 적용:**
   마스킹된 어텐션 점수에 Softmax 함수를 적용하면, $-\infty$로 설정된 미래 단어의 가중치는 $\mathbf{0}$이 된다.
4. **가중합:**
   결과적으로, 현재 단어는 **자신과 자신보다 앞에 나온 단어들**의 $\mathbf{V}$만 참조하여 문맥을 파악하게 된다.

Masked Multi-Head Self-Attention

결론 : Masked Self-Attention은 디코더가 **오직 과거와 현재 시점의 정보만** 사용하여 다음 단어를 예측하도록 강제함으로써, 문장 생성이라는 순차적 작업을 모방할 수 있게 해준다.

Multi-Head Encoder-Decoder Attention

디코더의 두 번째 하위 층으로, **"진짜 '번역'"** 역할을 한다.
일반 어텐션(Cross-Attention)의 멀티 헤드 버전이다

인코더가 완벽하게 이해한 소스 문장(입력 문맥)과 **현재까지 디코더가 생성한 타겟 문장**(출력 문맥) 사이의 **가장 관련성 높은 부분**을 연결하는 것이다

Multi-Head Encoder-Decoder Attention

작동 방식: Q, K, V의 출처 분리

이 층에서는 **Q**, **K**, **V**의 출처가 완전히 분리된다.

1. **Query (Q)의 출처 (디코더):**
   Q는 **직전 층 (Masked Self-Attention의 출력)**, 즉 **디코더**의 출력에서 가져온다. (타겟 문장의 현재 문맥)

2. **Key (K)와 Value (V)의 출처 (인코더):**
   K와 **V**는 **인코더의 최종 출력 벡터**에서 가져온다. (소스 문장의 완벽히 이해된 문맥)

3. **어텐션 계산:**
   디코더의 **Q**를 인코더의 **K**와 비교하여 어텐션 가중치를 계산한다.
   이 가중치를 인코더의 **V**에 적용하여 가중합한다.

Multi-Head Encoder-Decoder Attention

결론:
디코더는 현재 생성 중인 단어를 위해, 인코더가 학습한 **전체 소스 문장** 중에서 **가장 중요한 정보를 선택적**으로 끌어와 자신의 출력에 반영한다.
예를 들어, 한국어를 영어로 번역할 때, 현재 생성할 영어 단어와 가장 관련 있는 한국어 단어들을 찾아 집중하는 과정이다.

Decoder Layer 디코더도 6개의 동일한 레이어로 구성되며, 각 레이어는 3개의 하위 층을 가진다

1. **Masked Multi-Head Self-Attention**
2. **Add & Norm**
3. **Multi-Head Encoder-Decoder Attention (진짜 "번역"):**
4. **Add & Norm**
5. **Position-wise Feed-Forward Network (FFN)**
6. **Add & Norm**

1. Transformer 12. output

Output

1. 6층의 디코더를 통과한 최종 벡터가 나온다

2. 이 벡터를 **Linear Layer(Dense Layer)**에 통과시킨다. 이 레이어는 우리가 가진 전체 어휘 (Vocabulary, 예: 3만 개)의 크기만큼의 출력을 가진다.

3. **Softmax**를 적용하여, 3만 개 단어 각각에 대한 **확률 분포**를 얻는다.
   (예: student: 95%, teacher: 2%, ...)

4. 가장 확률이 높은 **student**를 다음 단어로 선택한다.

이 "student"는 **다음 스텝(time-step)의 디코더 입력**으로 다시 들어가고, 디코더는 `<END>`(문장 끝) 토큰을 생성할 때까지 이 과정을 반복한다.

1. Transformer 13. Encoder + Decoder + Output

최종 결론

Tokenization -> Encoder -> Encoder -> Encoder -> Encoder -> Encoder -> Encoder -> Decoder -> Decoder -> Decoder -> Decoder -> Decoder -> Decoder -> Output

Encoder
1. **Multi-Head Self-Attention**
2. **Add & Norm**
3. **Position-wise Feed-Forward Network (FFN)**
4. **Add & Norm**

Output
1. **Dense Layer**
2. **Sodtmax**

Decoder
1. **Masked Multi-Head Self-Attention:**
2. **Add & Norm**
3. **Multi-Head Encoder-Decoder Attention**
4. **Add & Norm**
5. **Position-wise Feed-Forward Network**
6. **Add & Norm**

1. Transformer

결론

트랜스포머가 대단한 이유

1. **엄청난 병렬화 → 빠른 학습 속도:**
   RNN의 순차 계산($O(n)$)을 행렬 곱 기반의 병렬 계산(O(1))으로 대체했다.
   이는 GPU 연산에 최적화되어, 기존 모델보다 훨씬 빠르게 대규모 데이터를 학습할 수 있게 했다.

2. **긴 의존성 문제 해결:**
   RNN은 정보가 n단계를 거쳐야 했지만(O(n) 경로), 트랜스포머는 Self-Attention을 통해 문장의 양 끝에 있는 단어도 **한 번에(O(1) 경로) 직접 연결**하고 관계를 학습할 수 있다.

3. **확장성 (Scalability):**
   이 두 가지 장점 덕분에, 모델의 크기(파라미터 수)와 데이터의 크기를 늘릴수록 성능이 지속적으로 향상되는 **스케일링 법칙(Scaling Law)**이 가능해졌다.

1. Transformer

결론

트랜스포머가 대단한 이유

이 트랜스포머 아키텍처는 이후 기계 번역뿐만 아니라, 문장의 의미를 이해하는 **BERT** (인코더 활용)와 문장을 생성하는 **GPT** (디코더 활용)의 기반이 되었으며, 오늘날 우리가 아는 **모든 초거대 AI(LLM)의 근간**이 되었다.

https://huggingface.co/gihakkk/Transformer

감사합니다