논문 리뷰

Attention is all you need

- 2017.12 Google이 발표한 논문
- Attention을 이용한 **Transformer** 모델을 제안

*RNN*의 한계점

- RNN의 병렬 처리 불가
  - RNN은 이전 시각에 계산한 결과를 이용하여 순서대로 계산. 따라서 RNN의 계산을 시간 방향으로 병렬 계산하기란 불가능. 보통 GRU를 사용한 병렬 계산 환경에서 학습이 이루어지는 딥러닝 학습에서 이는 큰 단점
- 따라서, RNN을 없애는 연구 / 병렬 계산 가능 RNN 연구가 활발히 진행
  - transformer
  - CNN을 이용한 seq2seq

트랜스포머(Transformer)의 장점

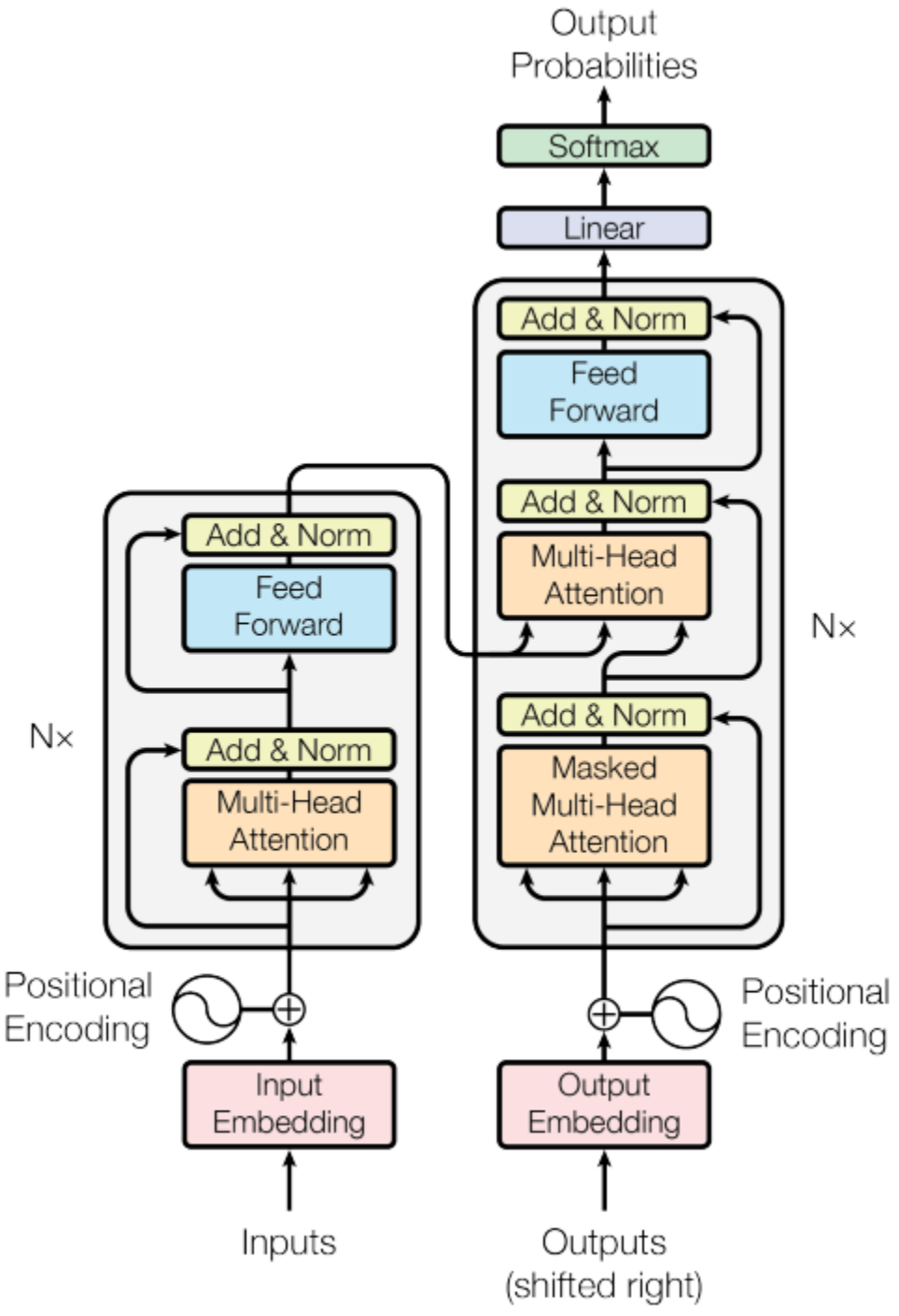
- 성능을 개선
- 훈련 속도가 빠르고, 병렬화하기 쉬움

트랜스포머의 기본 구조

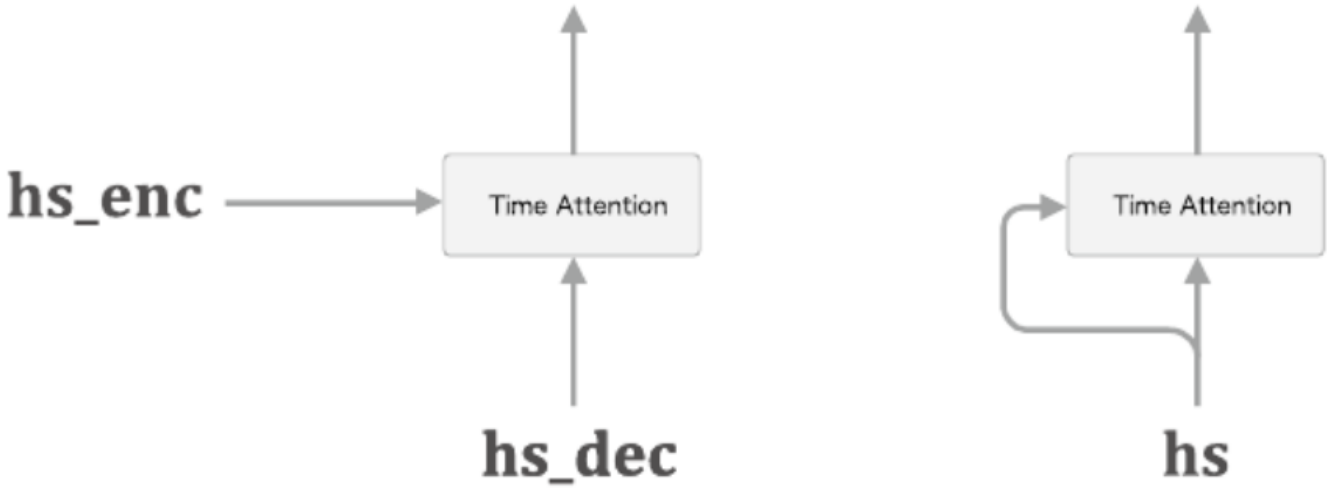
- 큰 틀 : Encoder, Decoder
- **셀프어텐션(self-Attention)** 기술을 이용
  - 하나의 시계열 데이터를 대상으로 한 어텐션. '하나의 시계열 데이터 내에서' 각 원소가 다른 원소들과 어떻게 관련되는지를 살핌
  - 문장 자기 자신에 주의를 기울임
  - 멀티 헤드 어텐션을 통해 수행
- **위치 인코딩을 제외한 모든 층은 타임 스텝에 독립적**
  - 오직 위치 인코딩을 통해서만 상대적/절대적 위치가 전해짐
- 전체적인 구조
  - **Encoder**

    1. Input Embedding
    2. Positional Encoding결과를 더함
    3. Multi-Head Attention
    4. 2의 output과의 residual connection & Normalize
    5. Feed Forward
    6. 4의 output과의 residual connection & Normalize
    7. 3~6의 과정을 N번(여기선 6번) 반복
  - **Decoder**

1. Output Embedding
2. Positional Encoding결과를 더함
3. Masked Multi-Head Attention
4. 2의 output과의 redidual connection & Normalize
5. Multi-Head Attention(Encoding output을 사용)
6. 4의 output과의 residual connection & Normalize
7. Feed Forward
8. 6의 output과의 residual connection & Normalize
9. 3~7의 과정을 N번(여기선 6번)반복
10. Linear
11. Softmax



셀프 어텐션

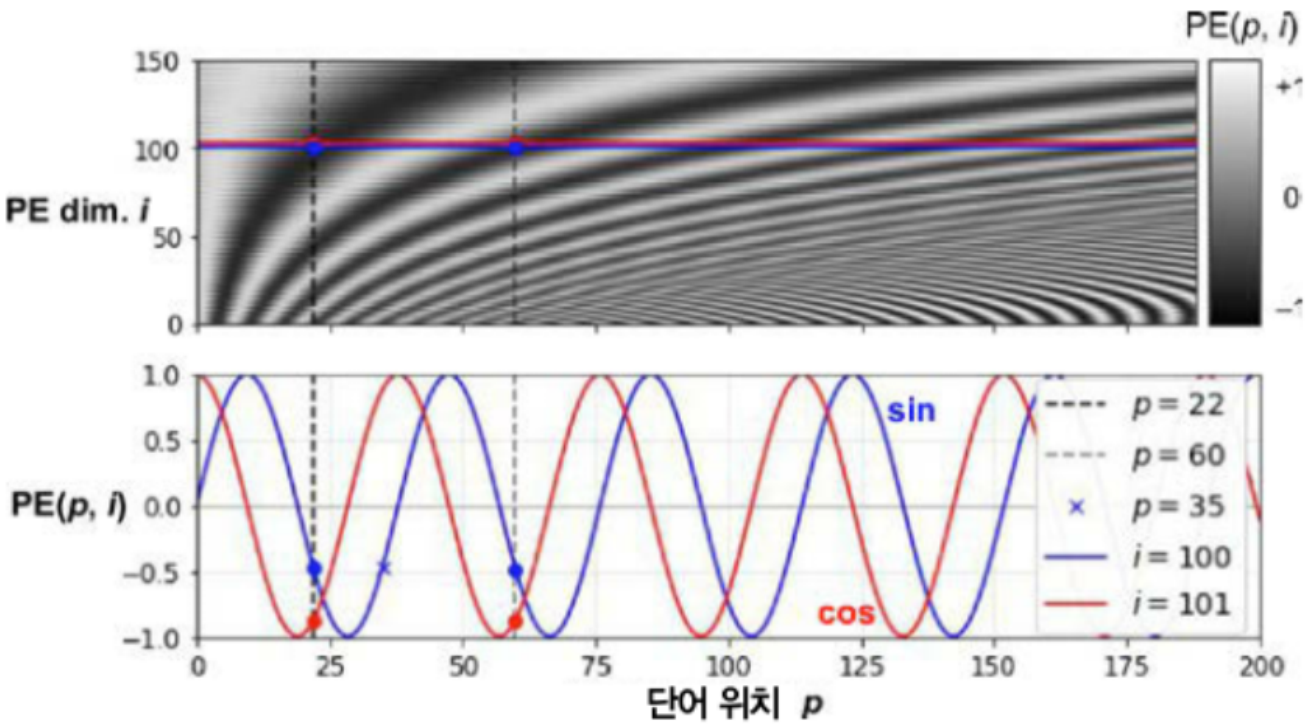


1. 위치 인코딩(positional encoding)

- 문장 안에 있는 단어의 위치를 인코딩한 밀집벡터
- n번째 위치 인코딩이 각 문장에 있는 n번째 단어의 단어 임베딩에 더해짐
- i번째의 위치 인코딩이 i번째 단어의 단어 임베딩에 더해짐
- 모델을 통해 학습할 수도, sin, cos함수로 정의한 고정된 위치 인코딩을 할 수도 있음

1.1 고정 위치 인코딩

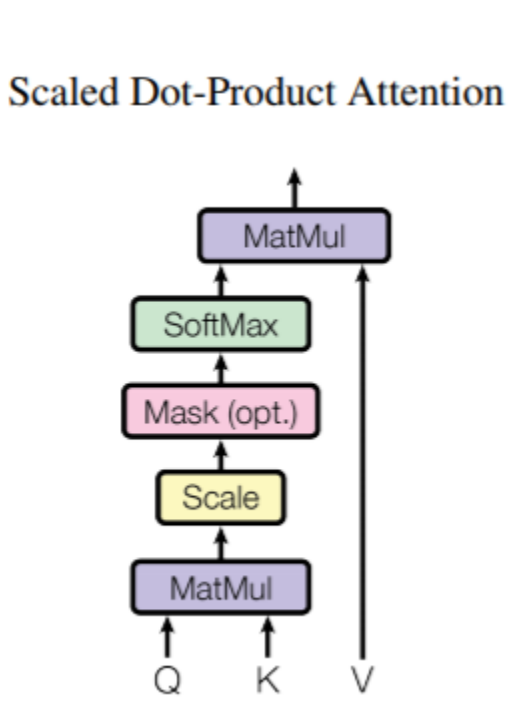
- 학습된 위치 인코딩과 동일한 성능을 내면서 임의의 긴 문장으로 확장 가능
- **절대적 위치를 알 수 있음**
  - i에 따라 sin, cos의 주기가 달라지기 때문에(i가 커질수록 파장이 커짐) 각 위치마다 고유한 vector가 만들어짐
  - 따라서 위치 인코딩을 단어 임베딩에 더하면 모델이 문장에 있는 단어의 절대 위치를 알 수 있음.
- **상대적 위치를 알 수 있음**
  - **같은 주기의 sin, cos함수를 사용함으로써** : 모델이 문장에 있는 단어의 상대 위치 또한 알 수 있음. 예를 들어 그래프에서 볼 수 있듯이 38개 단어만큼 떨어진 두 단어(ex p=22, p=60)는 위치 인코딩 차원 i=100, i=101에서 항상 같은 위치 인코딩 값을 가짐. 즉, 아, i=100, i=101에서 값이 같다면, '아! 이 단어들은 38개만큼 떨어진 단어이구나'를 알 수 있음. 따라서 같은 주기의 사인과 코사인 함수를 사용해야 함.(만약 주기가 다르다면 동일한 주기 마다 i=100, i=101의 원소에서 값이 같지 않을 것임.)
  - **sin, cos 함수를 모두 이용함으로써** : 둘 중 하나의 함수만 사용하면 모델이 p=22과 p=35의 위치를 구별할 수 없음.
- 파라미터
  - p : 해당 word의 순서
  - d : encoding vector의 크기(column)
  - $P_{p,2i}$: 각 위치 p에서의 2xi번째 원소 값
- 위치 인코딩 하는 방법(같은 주기의 sin, cos 함수를 사용해야 함)
  - $P_{p,2i} = sin(p/10000^{2i/d})$
  - $P_{p,2i+1} = cos(p/10000^{2i/d})$



2. Attention

2.1 스케일드 점-곱 어텐션(scaled dot-product attention)

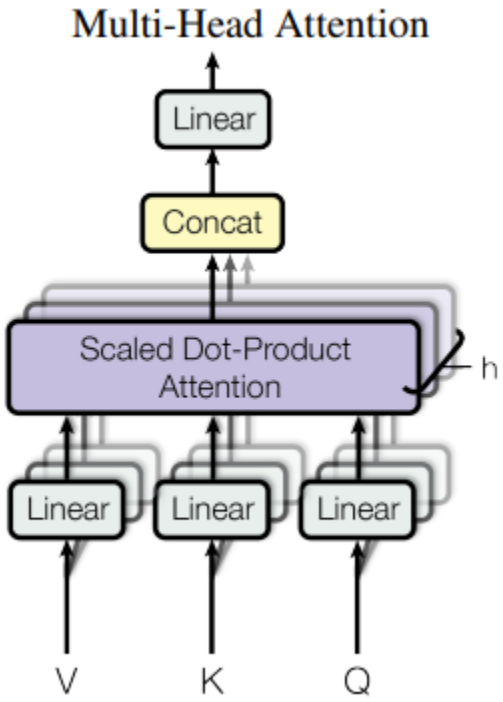
- Attention에 스케일링 인자를 추가한 것
  - d_{keys}를 나눔으로써 softmax function이 포화되지 않도록 함
- 파라미터
  - Q : 영향을 받는 인자(query)
    - $[n_{queries}, d_{keys}]$
  - K : 영향을 주는 인자(key)
    - $[n_{keys}, d_{keys}]$
  - V : 영향을 주는 인자의 값
    - $[n_{keys}, d_{values}]$



- Attention 계산
  - $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_{keys}}})V$
    - QK^T : 영향을 받는 word와 영향을 주는 word 간의 가중치
      - $[n_{queries}, n_{keys}]$
    - 최종 출력 : $[n_{queries}, d_{values}]$
- 마스크 계층
  - decoder에서만 사용
  - 정보가 왼쪽으로 가는 흐름을 막기 위해서 사용. i번째 output을 i+1번째 input으로 사용하는 auto-regressive한 특성을 유지하기 위함
  - 즉, **각 단어는 이전에 등장한 단어에만 주의를 기울일 수 있음**
  - softmax이전에 흐름에 $-\infty$ 으로 마스킹

2.2 멀티-헤드 어텐션(multi-head attention)

- 관련이 많은 단어에 더 많은 주의를 기울이면서 각 단어와 동일한 문장에 있는 다른 단어의 관계를 인코딩.(decoder의 Multi-Head Attention의 경우 encoder의 output인 입력문장과의 관계를 인코딩)
  - ex) 'They welcomed the Queen of the United Kingdom'과 같은 문장이 있다면 단어 Queen에 대해 이 층의 출력은 모든 단어에 의존하겠지만, 특히 United와 Kingdom에 더 주의를 기울일 것.
- scaled dot-product attention을 여러 층(6층)을 병렬적으로 쌓은 것
  - 이를 통해 모델이 단어 표현을 여러 부분 공간(subspace)로 다양하게 투영할 수 있음. 각 부분 공간은 단어의 일부 특징에 주목. 정보를 다양하게 표현할 수 있게 되면서 성능이 향상
  - 따라서 각 6층의 weight가 다르게 학습됨
- multi-head attention 계산
  - $MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^o)$
    - where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^v)$
    - $W_i^Q \in R^{d_{model} \times d_{keys}}, W_i^K \in R^{d_{model} \times d_{keys}}, W_i^V \in R^{d_{model} \times d_v}, W_i^O \in R^{hd_v \times d_{model}}$
    - d_{model}은 embedding vector와 feature 개수가 같음(여기선 512)
    - $d_v = d_h = d_{model}/6 = 64$



3. Position-wise Feed-Forward Networks

- 중간에 ReLU 활성화 함수가 있는 두 개의 선형변환으로 구성
- $FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$
- 모든 position에 같은 weight를 사용하지만, layer마다 다른 weight를 사용
  - W_1, b_1, W_2, b_2는 position에 무관하게 동일하게 적용되지만, W_1, b_1이 각각 W_2, b_2와 같지는 않음
  - 즉, kernel size가 1인 convolutions layer을 2번 사용하는 것과 같음
    - d_{ff} : 2048, (W_1 의 차원)
    - W_2의 차원 : 512
    - ff layer에 입력되는 data의 차원 : 512

4. 가중치 공유 기법을 사용

- Embedding layer Weight와 output의 linear transformation layer Weight 간에 같은 가중치 matrix를 공유 (학습된 embedding matrix를 사용)
- Embedding layer에는 $\sqrt{d_{model}}$을 곱해줌
- 효과
  - 파라미터의 개수가 줄어들면서 학습할 매개변수의 수를 줄일 수 있음
    - 학습하기가 더 쉬워짐
    - overfitting 방지 효과

5. 최적화

- Adam을 사용
  - $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$
- learning rate를 학습동안 변화시킴

- $lrate = d_{model}^{-0.5} \cdot min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$
  - warmup_step까지는 linear하게 ($warmup\_steps^{-1.5}$ 씩만큼 learning rate를 증가시키다가 warmup_step 이후에는 step_num의 inverse square root에 비례하도록 감소시킴(사용 warmup_step = 4000)

6. Regularization

6-1. Residual Dropout

- 각 sub-layer의 output에 dropout을 적용하고, sub-layer input에 추가하고 normalized한다.
- 각 stack의 embedding 및 positional encoding의 합에 dropout을 적용. (dropout 비율은 0.1로 설정)

6-2. Label Smooting

- 훈련하는 동안, 라벨 스무딩을 적용
- 이것을 통해 모델이 불확실함을 학습함.
- 정확성과 BLEU 점수를 향상시킴
- $\epsilon_{ls} = 0.1$

7. 결론

- recurrence를 이용하지 않고 encoder와 decoder에서 multi-headed self-attention을 이용하여 sequential data를 처리할 수 있는 model
- recurrent or convolutional layers를 이용한 구조보다 훨씬 **빠르고**, **정확**