💚

Attention Is All You Need

기간	@03/12/2025 → 03/18/2025
주차	2주차
논문	NIPS-2017-attention-is-all-you-need-Paper.pdf
상태	완료
예습/복습	예습과제
참고 자료	참고 블로그1 참고 블로그2 참고 블로그3

▼ [관련 개념 공부]

순환 신경망(Recurrent Neural Network, RNN)

- 순서가 있는 **연속적인 데이터(Sequence data)** 처리하는 데 적합한 구조.
- 각 시점(Time step)의 데이터가 이전 시점 데이터와 독립적이지 않다는 특성.

  ex) 일일 주가 데이터 - 3/7 주가는 3/6 주가의 영향을 받았을 가능성 높음. 3/6 주가는 3/5 주가의 영향 받았을 가능성 높음. ⇒ 이전 시점 영향을 받는 데이터 = 연속형 데이터.
- 자연어 데이터 ⇒ 연속적인 데이터의 일종.
    - 문장 안에서 단어들이 순서대로 등장, 각 단어는 이전에 등장한 단어의 영향을 받아 해당 문장 의미 형성함.
    - 주어진 문장들에서 이전 단어들의 패턴과 의미 고려해 다음에 올 단어 유추 가능.
    - 긴 문장일수록 앞선 단어들과 뒤따르는 단어들 사이에 강한 **상관관계(Correlation)** 존재.

      ⇒ 자연어 처리에서는 이러한 문맥과 상호작용을 모델링하여 정확한 의미 파악이 필요함.
- **순환 신경망**
    - **연속적인 데이터** 처리를 위해 개발된 인공 신경망.
    - 이전에 처리한 데이터를 다음 단계에 활용하고 현재 입력 데이터와 함께 모델 내부에서 과거 상태 기억해 현재 상태 예측하는 데 사용됨.
    - 시계열 데이터, 자연어 처리, 음성 인식 및 기타 시퀀스 데이터와 같은 도메인에서 널리 사용됨.
        - 일반적으로 길이가 가변적, 순서에 따라 의미가 있는 데이터.
    - **셀(cell)** : 각 시점의 데이터를 입력으로 받아 **은닉 상태(Hidden state)**와 출력 값을 계산하는 노드.
        - 이전 시점의 은닉 상태 h_{t-1} 입력으로 받아 현재 시점의 은닉 상태 h_t를 계산함.

- 각 시점 t에서 현재 입력값 x_t와 이전 시점 $t-1$의 은닉 상태 h_{t-1} 이용해 현재 시점의 은닉 상태 h_t와 출력값 y_t 계산함.

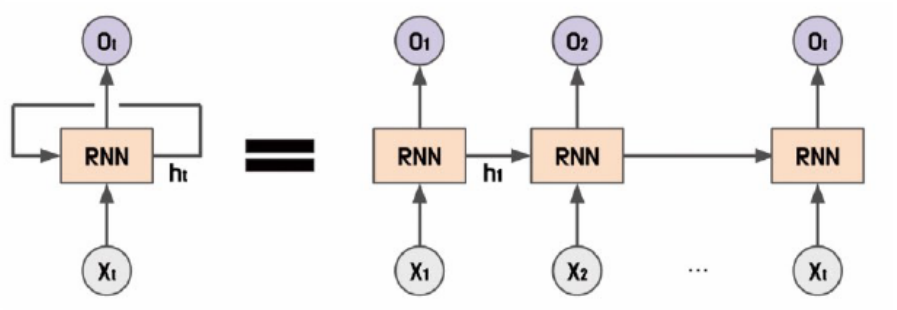


그림 6.12 순환 신경망의 셀

- **은닉 상태(Hidden state)**

$$h_t = \sigma_h(h_{t-1}, x_t)$$
$$h_t = \sigma_h(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

  - σ_h : 순환 신경망의 은닉 상태 계산하기 위한 활성화 함수
    - 이전 시점 $t-1$의 은닉 상태 h_{t-1}과 입력값 x_t 입력받아 현재 시점 은닉 상태 h_t 계산.
    - 가중치(W)와 편향(b) 이용하여 계산함.
      - W_{hh} : 이전 시점 은닉 상태 h_{t-1}에 대한 가중치
      - W_{xh} : 입력값 x_t에 대한 가중치
      - b_h : 은닉 상태 h_t의 편향
- 출력값

$$y_t = \sigma_y(h_t)$$
$$y_t = \sigma_y(W_{hy}h_t + b_y)$$

  - σ_y : 순환 신경망 출력값 계산하기 위한 활성화 함수.
  - 동일하게 가중치(W)와 편향(b) 이용하여 계산함.
    - W_{hy} : 현재 시점 은닉 상태 h_t에 대한 가중치

        - b_y : 출력값 y_t의 편향
- 구조 → 다양한 구조로 모델 설계 가능함.
    - 일대다 구조(One-to-Many)
        - 하나의 입력 시퀀스에 대해 여러 개의 출력값 생성.
        - 이미지 캡셔닝(Image Captioning)
    - 다대일 구조(Many-to-One)
        - 여러 개의 입력 시퀀스에 대해 하나의 출력값 생성.
        - 감성 분류(Sentimental Analysis) : 특정 문장의 감정(긍정, 부정) 예측
        - 자연어 추론(Natural Language Inference) : 두 문장 간 관계 추론.
    - 다대다 구조(Many-to-Many)
        - 입력 시퀀스와 출력 시퀀스의 길이가 여러 개인 경우.
        - 번역기, 음성 인식 시스템 등
        - 입력 시퀀스와 출력 시퀀스의 길이가 서로 다른 경우?
            - 입력 시퀀스, 출력 시퀀스 길이 맞추기 위해 패딩 추가하거나 잘라내는 등의 전처리 과정 수행.
        - **시퀀스-시퀀스(Seq2Seq)** 구조
            - 입력 시퀀스 처리하는 인코더(Encoder) : 고정 크기 벡터 출력.
            - 출력 시퀀스 처리하는 디코더(Decoder) : 벡터를 입력으로 받아 출력 시퀀스 생성.
    - 양방향 순환 신경망(Bidirectional Recurrent Neural Network, BiRNN)
        - 이전 시점($t-1$)의 은닉 상태뿐만 아니라 이후 시점($t+1$)의 은닉 상태로 함께 이용함.
        - t 시점 이후의 데이터도 t 시점 데이터를 예측하는 데 사용 가능.
        - 입력 데이터를 순방향 + 역방향 처리 방식 모두 이루어짐.
    - 다중 순환 신경망(Stacked Recurrent Neural Network)
        - 여러 개의 순환 신경망 연결하여 구성한 모델, 각 순환 신경망이 서로 다른 정보 처리하도록 설계.
        - 각 층에서의 출력값은 다음 층으로 전달되어 처리.

        - 여러 개의 층으로 구성된 RNN은 데이터 다양한 특징 추출할 수 있어 성능 향상될 수 있음.
        - 층이 깊어질수록 복잡한 패턴 학습할 수 있다는 장점. / 학습 시간 오래 걸리고, 기울기 소실 문제 발생할 가능성 높아진다는 단점.
- **LSTM(장단기 메모리, Long Short-Term Memory)**
    - 기존 순환 신경망 갖고 있던 기억력 부족과 기울기 소실 문제 해결한 모델.
    - 일반적인 순환 신경망 - 시간적으로 먼 과거의 정보는 잘 기억 x.

      ⇒ 학습 데이터 크기 커질수록 앞서 학습한 정보가 충분히 전달되지 않는다는 단점.

      ⇒ **장기 의존성 문제(Long-term dependencies)** 발생, 하이퍼볼릭 탄젠트 &ReLU 함수 특성으로 인한 역전파 과정에서 기울기 소실 및 폭주 발생 가능성.
    - 이러한 문제 해결을 위해 장단기 메모리 사용.
        - **메모리 셀(Memory cell)**과 **게이트(Gate)**라는 구조 도입.
- 장단기 메모리
    - **셀 상태(Cell state)**
        - 정보 저장하고 유지하는 메모리 역할.
        - 출력&망각 게이트에 의해 제어됨.
    - **망각 게이트(Forget gate)**
        - 이전 셀 상태에서 어떠한 정보 삭제할지 결정하는 역할.
        - 현재 입력과 이전 셀 상태 입력으로 받아 어떤 정보 삭제할지 결정.
    - **기억 게이트(Input gate)**
        - 새로운 정보를 어떤 부분에 추가할지 결정하는 역할.
        - 현재 입력과 이전 셀 상태 입력으로 받아 어떤 정보 추가할지 결정.
    - **출력 게이트(Output gate)**
        - 셀 상태 정보 중 어떤 부분 출력할지 결정하는 역할.
        - 현재 입력과 이전 셀 상태, 새로 추가된 정보 입력으로 받아 출력 정보 결정.
    - 세 가지 게이트 모두 활성화 함수로 **시그모이드** 사용.
        - 시그모이드 함수는 입력값을 0과 1 사이의 값으로 변환하므로 게이트 출력 값은 각각 0~1 사이의 값으로 결정.

Transformer

- 딥러닝 모델 중 하나로, 기계 번역, 챗봇, 음성 인식 등 다양한 자연어 처리 분야에서 많은 성과 내는 모델.
- **어텐션 메커니즘(Attention Mechanism)** → 시퀀스 임베딩 표현함.
    - **인코더와 디코더** 간의 상호작용으로 입력 시퀀스의 중요한 부분에 초점 맞추어 문맥 이해하고 적절한 출력 생성.
        - 인코더 : 입력 시퀀스 임베딩하여 고차원 벡터로 변환.
        - 디코더 : 인코더 출력을 입력으로 받아 출력 시퀀스 생성함.
    - 인코더와 디코더 단어 사이의 상관관계 계산하여 중요 정보에 집중.

      ⇒ 입력 시퀀스의 각 단어가 출력 시퀀스의 어떤 단어와 관련 있는지 파악.
- 기존 순환 신경망 기반 모델보다 학습 속도 빠르고, 병렬 처리 가능하여 대규모 데이터셋에 높은 성능 보임.
- 임베딩 과정에서 문장 전체 정보 고려하기 때문에 문장 길이 길어지더라도 성능 유지됨.
- 구조

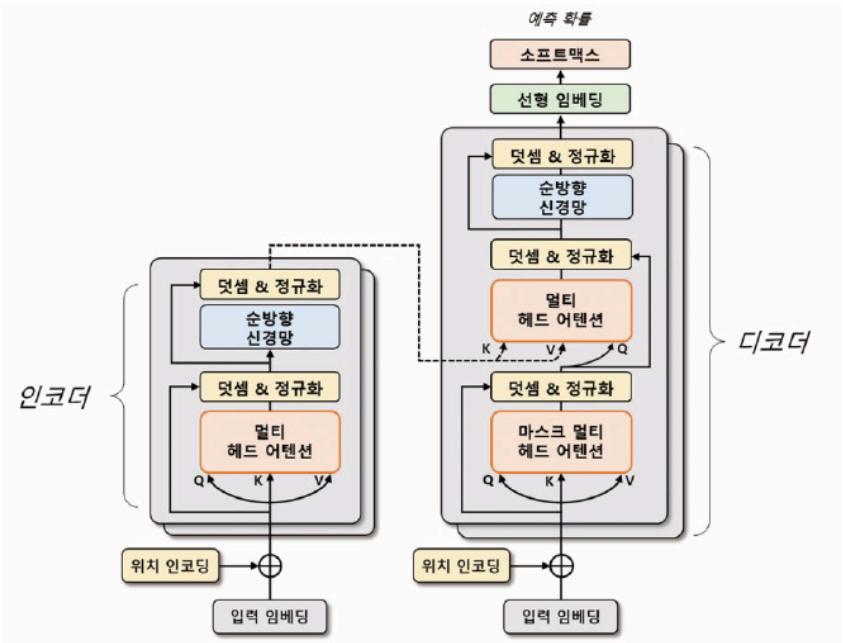


그림 7.2 트랜스포머 모델 구조(Q: 쿼리, K: 키, V: 값)

    - 인코더, 디코더는 두 부분으로 구성되어 있으며, 각각 N개의 **트랜스포머 블록(Transformer Block)**으로 구성됨.

- 이 블록은 **멀티 헤드 어텐션(Multi-Head Attention)**과 순방향 신경망으로 이뤄져 있음.
  - **쿼리(Query), 키(Key), 값(Value)** 벡터 정의하여 입력 시퀀스들의 관계를 셀프 어텐션(Self-Attention)하는 벡터 표현 방법.
  - 이 과정에서 쿼리와 각 키의 유사도를 계산, 해당 유사도를 가중치로 사용하여 값 벡터를 합산함.
  - 이렇게 계산된 어텐션 행렬은 입력 시퀀스 각 단어의 임베딩 벡터를 대체함. 결국 입력 시퀀스의 단어 사이의 상호작용 고려하여 임베딩 벡터 갱신함.
- 순방향 신경망 : 산출된 임베딩 벡터 더욱 고도화하기 위해 사용됨.
  - 여러 개의 선형 계층으로 구성, 앞선 순방향 신경망 구조와 동일하게 입력 벡터와 가중치 곱하고, 편향 더하며, 활성화 함수 적용함.
  - 이 과정에서 학습된 가중치들은 입력 시퀀스의 각 단어의 의미를 잘 파악할 수 있는 방식으로 갱신됨.
- 입력 시퀀스 데이터를 **소스(source)**와 **타깃(Target)** 데이터로 나눠 처리함.
  - 예) 영어→한글 번역하는 경우, 생성하는 언어인 한글을 타깃 데이터로 정의하고 참조하는 언어인 영어를 소스 데이터로 정의함.
- 인코더는 소스 시퀀스 데이터를 **위치 인코딩(Positional Encoding)**된 입력 임베딩으로 표현해 트랜스포머 블록의 출력 벡터 생성함. → 입력 시퀀스 데이터 관계를 잘 표현할 수 있게 구성.
- 디코더도 유사하게 트랜스포머 블록으로 구성, 마스크 멀티 헤드 어텐션 **(Masked Multi-Head Attention)** 사용하여 타깃 시퀀스 데이터 순차적으로 생성시킴. → 디코더 입력 시퀀스들의 관계 고도화하기 위해 인코더 출력 벡터 정보 참조.
- 최종적으로 생성된 디코더 출력 벡터는 선형 임베딩으로 재표현되어 이미지나 자연어 모델에 활용됨.

- **입력 임베딩과 위치 인코딩**
  - 입력 시퀀스의 각 단어는 임베딩 처리되어 벡터 형태로 변환됨.
    - 트랜스포머 모델은 순환 신경망과 달리 입력 시퀀스를 **병렬 구조**로 처리하기 때문에 단어의 순서 정보 제공하지 x.

    ⇒ **위치 정보**를 임베딩 벡터에 추가하여 단어의 순서 정보를 모델에 반영해야 됨. 이를 위해 트랜스포머는 **위치 인코딩 방식** 사용함.
  - 위치 인코딩

        - 입력 시퀀스의 순서 정보를 모델에 전달하는 방법.
        - 각 단어의 위치 정보를 나타내는 벡터 더하여 임베딩 벡터에 위치 정보 반영.
        - 위치 인코딩 벡터 : sin 함수와 cos 함수 사용하여 생성, 이를 통해 임베딩 벡터와 위치 정보가 결합된 최종 입력 벡터 생성함. ⇒ 단어의 순서 정보 학습 가능하게 됨.
        - 각 토큰 위치를 각도로 표현해 sin 함수와 cos 함수로 위치 인코딩 벡터 계산함.
            - 토큰의 위치마다 동일한 임베딩 벡터 사용하지 않기 때문에 각 토큰의 위치 정보를 모델이 학습 가능함.
- **특수 토큰**
    - 단어 토큰 이외의 특수 토큰을 활용하여 문장 표현함.
    - 특수 토큰 - 입력 시퀀스의 시작과 끝을 나타내거나 마스킹(Masking) 영역으로 사용됨.
    - 모델이 입력 시퀀스의 시작과 끝을 인식할 수 있게 하며, 마스킹 통해 일부 입력 무시할 수 있음.
        - 예) 번역 모델 - 디코더의 입력 시퀀스에서 현재 위치 이후의 토큰 마스크하여 이전 토큰만을 참조하도록 함.
    - BOS(Beginning of Sentence)
        - 문장의 시작
    - EOS(End of Sentence)
        - 문장의 종료
    - UNK(Unknown)
        - 어휘 사전에 없는 단어, 모르는 단어 의미함.
        - 모델이 이전에 본 적 없는 단어 처리할 때 사용됨.
    - PAD(Padding)
        - 모든 문장을 일정한 길이로 맞추기 위해 사용됨.
        - 짧은 문장의 빈 공간 채울 때 사용.
- **트랜스포머 인코더**
    - 입력 시퀀스 받아 여러 개의 계층으로 구성된 인코더 계층을 거쳐 연산 수행.

- 각 인코더 계층은 멀티 헤드 어텐션과 순방향 신경망으로 구성, 입력 데이터에 대한 정보 추출하고 다음 계층으로 전달함.

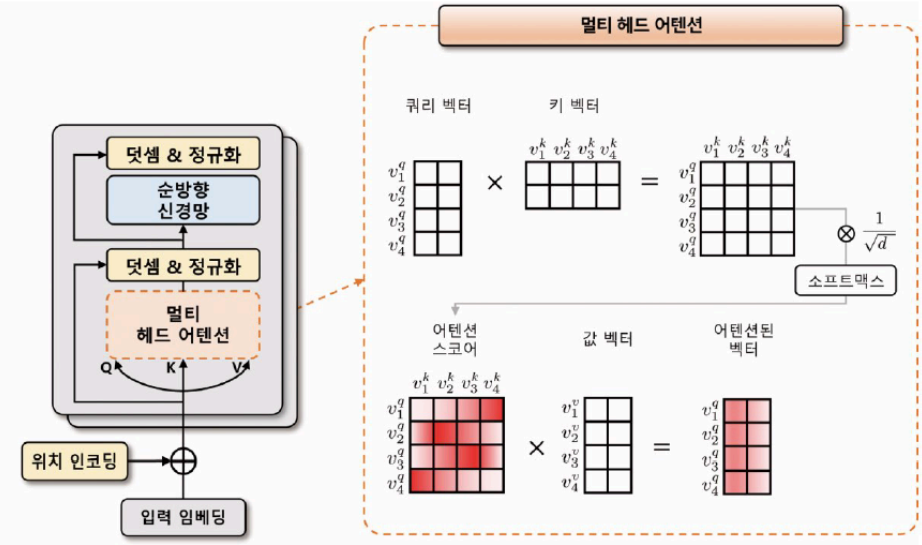


그림 7.5 트랜스포머 인코더 블록 연산 과정

- 위치 인코딩 적용된 소스 데이터의 입력 임베딩 입력 받음.
- 멀티 헤드 어텐션 단계에서 입력 텐서 차원 [N, S, d]라고 하면 입력 임베딩은 선형 변환 통해 3개의 임베딩 벡터 생성. (**쿼리(Q), 키(K), 값(V)**)
  - 쿼리 벡터: 현재 시점에서 참조하고자 하는 정보의 위치 나타내는 벡터
    - 인코더의 각 시점마다 생성됨.
    - 현재 시점에서 질문이 되는 벡터, 이 벡터 기준으로 다른 시점 정보 참조.
  - 키 벡터: 쿼리 벡터와 비교되는 대상, 쿼리 벡터를 제외한 입력 시퀀스에서 탐색되는 벡터.
    - 인코더의 각 시점에서 생성됨.
  - 값 벡터: 쿼리 벡터와 키 벡터로 생성된 어텐션 스코어를 얼마나 반영할지 설정하는 가중치 역할.

수식 7.2 어텐션 스코어 계산식

$$score(v^q, v^k) = \mathrm{softmax}\left(\frac{(v^q)^T \cdot v^k}{\sqrt{d}}\right)$$

- 쿼리와 키 벡터의 연관성은 내적 연산으로 [N, S, S] 어텐션 스코어 맵 사용함.
  - 셀프 어텐션 과정에서는 3개 벡터를 내적하여 어텐션 스코어 구하고 이 스코어값에 벡터 차원의 제곱근($\sqrt{d}$)만큼 마눠 보정함.(벡터 차원 커질 때 스코어값이 같이 커지는 문제 완화.)
  - 보정된 어텐션 스코어를 소프트 맥스 함수 이용하여 확률적 재표현하고, 이를 값 벡터와 내적하여 셀프 어텐션된 벡터 생성함.
- **멀티 헤드(Multi-Head)**
  - 이러한 셀프 어텐션 여러 번 수행하여 여러 개의 헤드 만듦.
  - 각각의 헤드가 독립적으로 어텐션 수행하고 그 결과를 합침.
- 입력받는 [N, S, d] 텐서에 k개의 셀프 어텐션 벡터 생성하고자 한다면 헤드에 대한 차원 축 생성해 [N, k, S, d/k] 텐서 형태 구성. (k개의 셀프 어텐션된 [N, S, d/k] 텐서 의미)
- k개의 셀프 어텐션 벡터는 임베딩 차원 축으로 다시 병합(concatenation)되어 [N, S, d] 형태로 출력됨.
- 덧셈&정규화 : 멀티 헤드 어텐션 통과하기 이전의 입력값 [N, S, d] 텐서와 통과 이후의 출력값 [N, S, d] 텐서를 더함으로써 학습 시 발생하는 기울기 소실 완화. 임베딩 차원 축으로 정규화하는 계층 정규화 적용.
- 순방향 신경망
  1. 선형 임베딩과 ReLU로 이루어진 인공 신경망
  2. 1차원 합성곱
- 이어서 다시 덧셈&정규화 과정 수행.
- 인코더는 여러 개의 트랜스포머 인코더 블록으로 구성.
  - 이전 블록에서 출력된 벡터는 다음 블록의 입력으로 전달되어 인코더 블록 통과하면서 점차 입력 시퀀스 정보 추상화됨.
  - 마지막 인코더 블록에서 출력된 벡터는 디코더에서 사용되며, 디코더의 멀티 헤드 어텐션 모듈에서 참조되는 키, 값 벡터로 활용됨.

- **트랜스포머 디코더**
  - 위치 인코딩 적용된 타깃 데이터의 입력 임베딩을 입력받음.
  - 디코더 입력 임베딩에 위치 정보 추가함으로써 디코더가 입력 시퀀스 순서 정보 학습 가능해짐.

- 인코더의 멀티 헤드 어텐션 모듈 → 인과성(Casuality) 반영한 **마스크 멀티 헤드 어텐션 모듈**로 대체됨.
  - 어텐션 스코어 맵을 계산할 때 첫 번째 쿼리 벡터가 첫 번째 키 벡터만을 바라볼 수 있게 마스크를 씌움.
  - 두 번째 쿼리 벡터는 첫 번째와 두 번째 키 벡터를 바라보게 마스크 씌움.

    ⇒ 셀프 어텐션에서 현재 위치 이전의 단어들만 참조 가능, 인과성 보장됨.
  - 마스크 영역에 수치적으로 굉장히 작은 값 -inf 마스크 더해줌으로써 해당 영역의 어텐션 스코어 값이 0에 가까워짐.
  - 소프트맥스 계산 시 해당 영역 어텐션 가중치가 0이 되므로, 마스크 영역 이전에 있는 입력 토큰들에 대한 정보 참조하지 x.

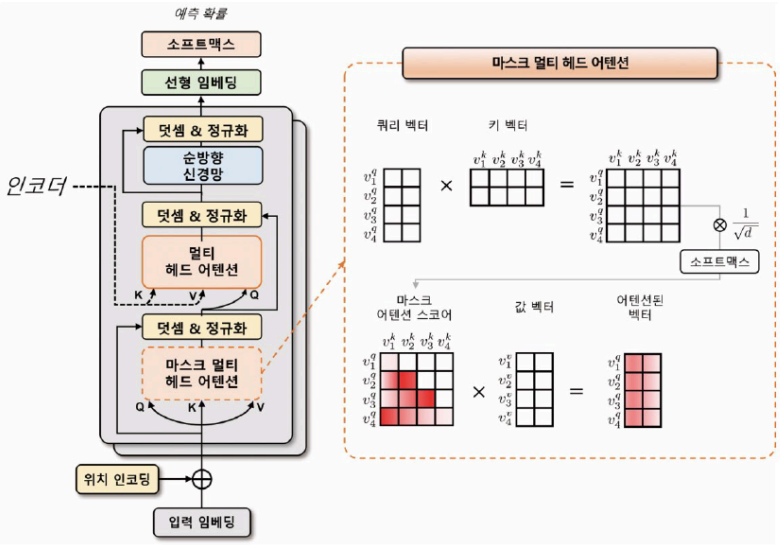


그림 7.6 트랜스포머 디코더 블록 연산 과정

- 디코더 블록 연산 과정
  - 멀티 헤드 어텐션 : 타깃 데이터가 쿼리 벡터로, 인코더의 소스 데이터가 키와 값 벡터로 사용됨.

    ⇒ 쿼리 벡터는 타킷 데이터의 위치 정보 포함한 입력 임베딩과 위치 인코딩을 더한 벡터가 됨.
  - 이후 쿼리, 키, 값 벡터 이용하여 어텐션 스코어 맵 계산 → 소프트맥스 함수 적용하여 어텐션 가중치 구함. → 최종적으로 어텐션 가중치와 값 벡터 가중합하여 멀티 헤드 어텐션의 출력 벡터 얻음.
  - 셀프 어텐션 방지하기 위해 마스킹 적용.

        - 인코더처럼 여러 개의 트랜스포머 디코더 블록으로 구성, 이전 트랜스포머 디코더 블록의 산출물은 다음 트랜스포머 디코더 블록 입력으로 전달됨.
        - 이로써 디코더는 이전 시점에서의 정보를 현재 시점에서 활용할 수 있게 됨.
        - 또한 마지막 디코더 블록 출력 텐서 [N, S, d]에 선형 변환 및 소프트맥스 함수 적용하여 각 타깃 시퀀스 위치마다 예측 확률 계산 가능.
    - 디코더 - 타깃 데이터 추론할 때 토큰 또는 단어를 순차적으로 생성시킴.
        - 입력 토큰 순차적으로 나타내는 방법은 빈 값 의미하는 PAD 토큰 사용.

표 7.2 인과성을 고려한 디코더 입력과 출력 예시

추론 순서	디코더 입력	디코더 출력
1	[BOS], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]
2	[BOS], 'ChatGPT', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]
3	[BOS], 'ChatGPT', '는', [PAD], [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', [PAD], [PAD], [PAD], [PAD], [PAD]
4	[BOS], 'ChatGPT', '는', '트랜스포머', [PAD], [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', [PAD], [PAD], [PAD], [PAD]
5	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', [PAD], [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', [PAD], [PAD], [PAD]
6	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', [PAD], [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', '있다', [PAD], [PAD]
7	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', '있다', [PAD], [PAD]	[BOS], 'ChatGPT', '는', '트랜스포머', '모델로', '이뤄져', '있다', [EOS], [PAD]

0. Abstract

- 기존 시퀀스 변환 모델
    - 복잡한 순환 신경망(RNN) 또는 합성곱 신경망(CNN)에 기반하여 인코더-디코더 구조 사용, 주로 어텐션 메커니즘 포함.
- 본 논문에서는 새로운 네트워크 구조 **Transformer(트랜스포머)** 제안
    - 이는 순환성과 합성곱을 완전히 제거하고 오직 **어텐션 메커니즘**에만 기반함.
- 실험 결과, Transformer는 기계 번역 작업에서 더 높은 성능과 병렬 처리 효율성 보이며 훈련 시간 크게 단축됨을 확인.
    - WMT 2014 영어-독일어 번역 : 28.4 BLEU, 영어-프랑스어 번역 : 41.0 BLEU 기록하며 기존 최고 성능 초과.

▼ ❗ **BLEU(BiLingual Evaluation Understudy)**란?

- 기계 번역(Machine Translation) 모델의 성능 평가하는 대표적인 지표.
- 0~100 사이의 값을 가지며, 점수가 높을수록 기계 번역이 사람이 번역한 문장과 유사함을 의미함.
- 일반적으로 30~40 정도가 높다고 평가되며, 50 이상은 거의 완벽한 번역에 가까운 수준.

1. Introduction

논문이 다루는 분야

- **자연어 처리(NLP)**와 **기계 번역** 분야에서 시퀀스 변환(sequence transduction) 다룸.
- 기존 연구들은 주로 순환 신경망(RNN), 장단기 기억 네트워크(Long Short-Term Memory, LSTM), 게이트 순환 유닛(Gated Recurrent Units, GRU) 등의 모델을 기반으로 함.
  - 언어 모델링과 기계 번역과 같은 시퀀스 모델링 및 변환 문제에 주로 쓰였던 기술들.

해당 task에서 기존 연구 한계점

- 기존 RNN 기반 모델 - 순차적으로 연산해야 했기 때문에 병렬화가 어렵고, 긴 문장에서는 학습 속도가 느려지는 문제 존재.
  - 특히, 긴 시퀀스를 처리할 때 메모리 제한으로 인한 배치(batch) 크기 늘리는 것이 어려움.
- 최근 연구에서는 **연산 최적화(factorization tricks)**와 **조건부 연산(conditional computation)** 통해 계산 효율성 높이는 방법 제안 → 여전히 순차적 연산의 근본적인 제약 존재함.
- 어텐션(Attention) 메커니즘 → 다양한 시퀀스 모델링 및 변환 모델에서 중요한 역할 해왔으며, 입력 또는 출력 시퀀스 내에서 **토큰 간 거리에 관계없이 의존성을 모델링할 수 있도록** 해줌.
  - 그러나, 대부분 RNN과 결합된 형태.

논문의 contributions

- **'Transformer'**라는 새로운 모델 구조 제안.

- **RNN 사용하지 않고 오직 어텐션 메커니즘만을 활용**하여 입력과 출력 간 **전역적인 의존성을 학습**함.
- RNN보다 훨씬 높은 병렬 처리 성능을 제공, 8개의 P100 GPU에서 단 12시간만의 학습으로 새로운 최첨단 번역 성능 달성할 수 있었음.

2. Related Work(Background)

- 순차적 연산을 줄이려는 시도는 과거에도 존재함.

  ✅ **Extended Neural GPU**

  ✅ **ByteNet**

  ✅ **ConvS2S (Convolutional Sequence-to-Sequence)**

  - 이 모델들은 기본적인 연산 단위로 CNN 사용하며, 입력 및 출력의 모든 위치에서 병렬적으로 은닉 표현(hidden representation) 계산함.
  - 그러나 **멀리 떨어진 두 입력 간의 관계를 학습**하는 데 필요한 **연산량 증가**하는 문제.
    - ConvS2S 모델) **입력 간 거리에 따라 연산량이 선형적으로 증가**
    - ByteNet 모델) **로그 형태로 증가**.
  - 멀리 떨어진 단어들 간 의존성 학습하는 것이 어려워지는 경향.
- Transformer에서는 **셀프-어텐션(Self-Attention) 메커니즘** 활용하여 이 문제 해결. 두 입력 위치 간 연산량 일정하게 유지함.

  ✳️ **셀프-어텐션(Self-Attention) 메커니즘?**

  - 하나의 시퀀스 내에서 서로 다른 위치를 연결하여 새로운 표현 생성하는 메커니즘.
  - 아래의 NLP 작업에서 효과적으로 사용된 바가 있음.
    - 독해(Reading Comprehension)
    - 추상적 요약(Abstractive Summarization)
    - 텍스트 의미 관계 분석(Textual Entailment)
    - 문장 표현 학습(Task-Independent Sentence Representation)
- 기존의 End-to-End 메모리 네트워크(end-to-end memory networks)는 전통적인 RNN 대신 **반복적인 어텐션 메커니즘 사용**하여 간단한 언어 기반 질의응답과 언어 모델링에서 우수한 성능 보임.

- 하지만, Transformer는 **기존의 RNN 및 CNN을 완전히 배제**하고, 순수한 **셀프-어텐션(Self-Attention)만을 활용**하여 시퀀스 변환을 수행하는 최초의 모델.

3. 제안 방법론(Model Architecture)

- 신경망 기반 시퀀스 변환(sequence transduction) 모델 대부분 **인코더-디코더 구조**를 따름.
  - **인코더(Encoder)**: 입력 시퀀스 $(x_1, \ldots, x_n)$를 연속적인 표현(continuous representation) **z** = $(z_1, \ldots, z_n)$으로 변환.
  - **디코더(Decoder)**: 인코더의 출력을 기반으로 출력 시퀀스 $(y_1, \ldots, y_n)$를 생성.
- Transformer는 이러한 인코더-디코더 구조 유지하면서도, 기존의 RNN 또는 CNN 사용하지 않고 오직 **Self-Attention과 완전연결(Feed-Forward) 네트워크**만을 활용함.
- 인코더와 디코더 모두 동일한 기본 구조를 **여러 층(Stack)으로 쌓아** 구성.

3.1. Encoder and Decoder Stacks

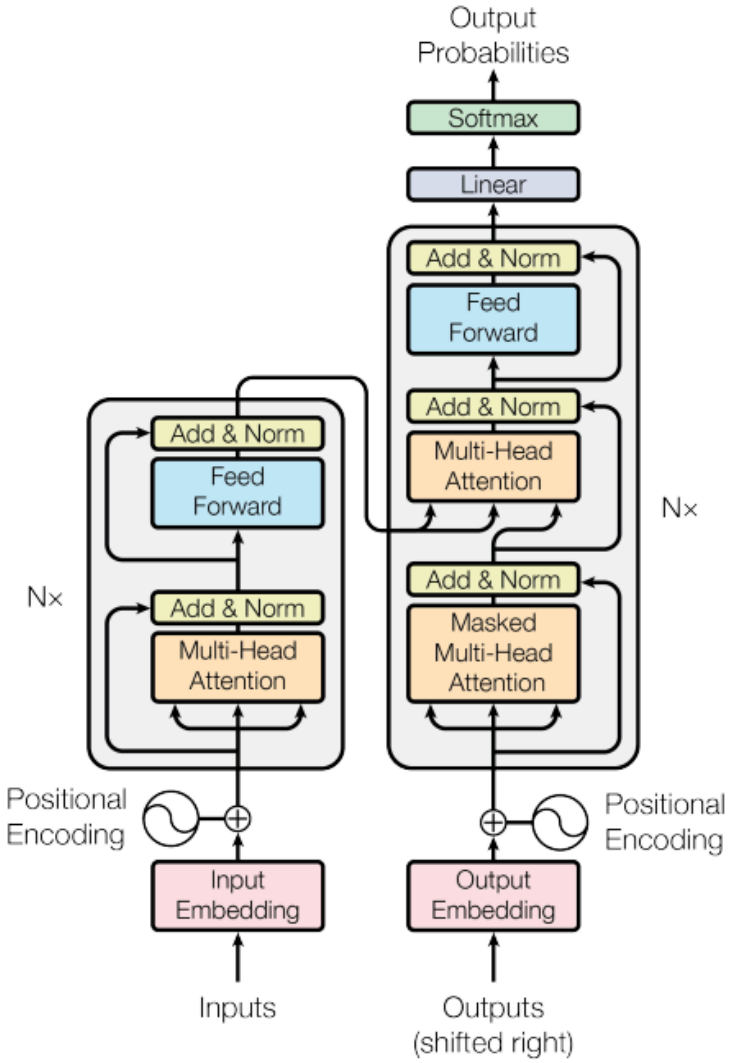


Figure 1: The Transformer - model architecture.

(1) 인코더 (Encoder)

- Transformer의 인코더는 **N=6개의 동일한 layer로 구성**된 스택.
- 각 layer는 다음과 같은 두 개의 sub-layer로 이루어짐.
  1. **멀티헤드 셀프-어텐션(Multi-Head Self-Attention) 메커니즘**
  2. **위치별 완전연결 신경망(Position-wise fully connected Feed-Forward Network, FFN)**
     - 각 sub-layer에는 잔차 연결(Residual Connection)과 층 정규화(Layer Normalization) 적용됨.
     - 각 sub-layer의 출력은 다음과 같이 정규화됨.

$$LayerNorm(x + Sublayer(x))$$

        - 잔차 연결 적용을 위해, Transformer의 모든 sub-layer와 임베딩 층의 출력 차원은 **512**로 고정됨.

(2) 디코더 (Decoder)

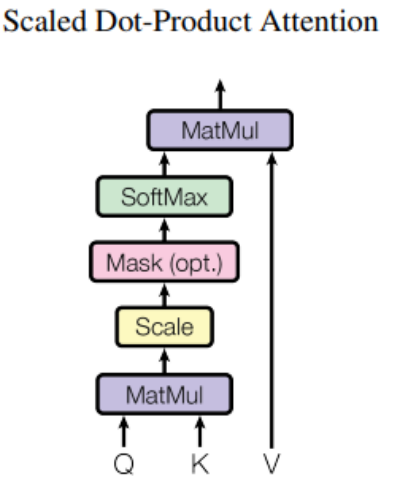
- 디코더 또한 **N=6개의 동일한 layer로 구성,** 인코더 구조와 유사함.
- 인코더와 다른 점 존재. (3개의 sub-layer)
    1. **멀티헤드 셀프-어텐션(Multi-Head Self-Attention)**
    2. **인코더 출력에 대한 멀티헤드 어텐션(Multi-Head Attention over Encoder Output)**
    3. **위치별 완전연결 신경망(Position-wise fully connected Feed-Forward Network, FFN)**
    - 디코더의 self-attention layer는 미래 정보를 참조하지 않도록 마스킹(Masking) 적용함. ⇒ 현재 위치 i에서의 예측은 **이전 위치** $< i$ **의 정보만을 사용**할 수 있도록 설계.

      ⇒ 이러한 마스킹 기법 통해 디코더는 **자동회귀적(auto-regressive) 특성 유지**할 수 있음.

3.2. Attention (어텐션 메커니즘)

- **쿼리(Query), 키(Key), 값(Value)를 매핑하여 출력을 생성하는 함수**로 정의.
    - 이때, **쿼리(Query)와 키(Key) 간의 연관도(compatibility)** 계산하여 값(Value)에 가중치를 부여.

3.2.1. Scaled Dot-Product Attention

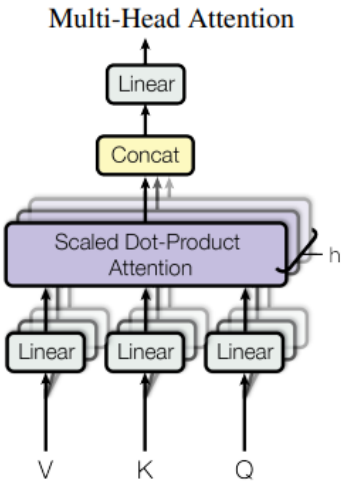


- 공식

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

  - Q(쿼리), K(키), V(값) → 행렬로 표현됨.
  - d_k: 키(Key)의 차원, 값이 너무 커지면 Softmax가 매운 작은 기울기 가지게 되므로 $\sqrt{d_k}$로 나누어 스케일 조정함.

3.2.2. Multi-Head Attention (멀티헤드 어텐션)



- 하나의 어텐션만 수행하는 것이 아니라, **여러 개의 어텐션을 병렬적으로 수행**하는 멀티헤드 어텐션 기법 사용함.

  ✳️ **장점** ✳️

  - 각각의 어텐션 헤드가 서로 다른 정보(subspace) 학습할 수 있도록 함.
  - 단일 어텐션보다 표현력 증가하여, 문장에서 다양한 의미적 패턴 학습할 수 있음.
- 공식

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

  - W_i^Q, W_i^K, W_i^V : 각 헤드에 대한 학습 가능한 가중치 행렬.
- $h = 8$개의 헤드 사용하며, 각 헤드는 $d_k = d_v = d_{model}/h = 64$ 차원을 가짐.

3.2.3. Applications of Attention in our Model

- 3가지 다른 방식으로 사용됨.
  1. ‘인코더-디코더 어텐션’ 레이어 → **쿼리(Query)**는 **이전 디코더**에서 가져오며, **키(Key)와 값(Value)**는 **인코더의 출력**에서 가져옴. 이를 통해 디코더의 모든 위치가 입력 시퀀스의 모든 위치를 참조(어텐션)할 수 있도록 함. (기존의 sequence-to-sequence 모델에서 사용되는 전형적인 인코더-디코더 어텐션 메커니즘 모방한 것.)
  2. 인코더(Encoder)에는 **셀프-어텐션 레이어**가 포함됨.
     - 여기서는 쿼리, 키, 값이 모두 동일한 곳(**이전 인코더 레이어의 출력**)에서 가져옴.

       ⇒ 이를 통해 인코더 각 위치는 이전 레이어의 모든 위치 참조할 수 있도록 설계됨.
  3. 디코더(Decoder)에서도 **셀프-어텐션 레이어** 포함됨.
     - 디코더의 각 위치는 해당 위치까지의 모든 디코더 위치 참조할 수 있음.
     - But, auto-regressive 특성 유지하기 위해 미래 정보가 디코더로 전달되지 않도록 차단해야 함. ⇒ 이를 위해 Scaled Dot-Product Attention 내부에서 masking 기법 적용함. (소프트맥스 입력에서 불가능한 연결에 해당하는 값을 $-\infty$로 설정하여 마스킹 처리함.)

3.3. Position-wise Feed-Forward Networks

- 각 인코더 및 디코더 레이어에는 **완전연결 신경망(Feed-Forward Network, FFN)**이 포함됨.
- 구조

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

  - 첫 번째 선형 변환 후 ReLU 활성화 함수 적용
  - 두 번째 선형 변환 거쳐 최종 출력 생성
- 각 위치에서 독립적으로 적용되며, 커널 크기가 1인 2개의 convolution(1D-CNN)과 유사한 방식으로 동작함.

3.4. Embeddings and Softmax

- 입력 및 출력 토큰을 벡터로 변환하는 임베딩 층 사용.
- 임베딩 층과 출력층의 가중치를 공유(weight tying)하여 모델 성능을 향상시킴.
- 최종 출력은 **소프트맥스(Softmax) 함수**를 통해 확률 값으로 변환함.

3.5. Positional Encoding

- Transformer는 RNN이 없기 때문에, 단어의 순서를 고려하는 방법이 필요함.

  ⇒ 이를 위해 인코더 및 디코더 스택의 하단에 **Positional Encoding (위치 인코딩)** 추가함.

- 정의

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

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

  - pos : 위치, i : 차원
  - sine & cosine 함수 사용
    - 고정된 offset k에 대해 PE_{pos+k}가 PE_{pos}의 선형 함수로 표현될 수 있기 때문에 모델이 쉽게 상대적인 위치 참조할 수 있을 것이라 가정함.

  🤔 홀수, 짝수 나누어 함수 2개(사인, 코사인) 사용하는 이유?

    - 하나의 함수만 사용할 경우, 위치가 커질 때마다 그 값이 커지고 작아짐을 반복하여 어떤 특정 두 토큰의 위치값이 동일해질 수 있기 때문. (주기성 때문에 특정 위치의 값이 이전의 위치 값과 동일하게 나타나는 문제.)
    - 각각의 고유한 토큰 위치값은 유일한 값 가져야 하고, 서로 다른 두 토큰이 떨어져 있는 거리가 일정해야 함.
    - 사인, 코사인 같이 이용하면 서로 다른 주기성을 가지도록 조정되어 다른 값 생성 가능함.

4. Why Self-Attention

- 왜 Self-Attention이 기존의 순환 및 합성곱 계층보다 우수한가?

- 세 가지 기준에서 비교해보자.

1. **각 층에서의 연산 복잡도 (Computational Complexity per Layer)**
2. **병렬화 가능성 (Parallelizability)**
3. **장거리 의존성 학습 (Ability to Model Long-Range Dependencies)**

- **연산 복잡도 비교**

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

  - **RNN** : 입력 길이 n에 비례하여 연산량 증가하므로, 긴 문장 처리 시 비용이 큼.
  - **CNN** : RNN보다 빠르지만, 원거리 단어 간 관계 학습이 어려움. (큰 커널 사용 시 연산량 증가함.)
  - **Self-Attention** : 입력 길이 n에 따라 $O(n^2)$의 연산량 필요하지만, 병렬 처리가 가능하여 빠르게 학습 가능함.

- **병렬화 가능성**
  - **RNN** : 이전 타임스텝의 출력 사용해야 하므로 순차적 연산이 필수, 병렬화 어려움.
  - **CNN** : 필터를 사용해 여러 위치를 동시에 계산 가능하나, **멀리 떨어진 단어 간 관계를 학습하는 데 여러 계층이 필요**.
  - **Self-Attention** : 모든 토큰이 동시에 계산되므로 **완전 병렬화 가능**.

- **장거리 의존성 학습**
  - **RNN : 최대 경로 길이가 O(n), 입력 길이가 길어질수록 학습이 어려워짐.** (기울기 소실 문제)
  - **CNN** : **커널 크기 k에 따라 $O(\log_k(n)$의 경로 길이가 필요**, 깊은 네트워크가 요구됨.
  - **Self-Attention** : **모든 단어가 한 번의 연산으로 서로 연결될 수 있어 $O(1)$의 경로 길이를 가지며, 장거리 의존성 학습이 용이**.

⇒ 👍 Self-Attention은 **병렬화가 가능**하고 **장거리 의존성을 쉽게 학습**할 수 있어, RNN 및 CNN보다 **효율적**임.

5. 실험 (Training)

Dataset

- 다음 데이터셋 사용하여 학습됨.
  - **WMT 2014 영어-독일어 번역 데이터셋** (약 450만 문장)
  - **WMT 2014 영어-프랑스어 번역 데이터셋** (약 3600만 문장)
  - **Byte-Pair Encoding (BPE) 적용**: 약 37,000개의 서브워드(subword) 단위를 사용
- 배치(batch) 크기
  - 비슷한 문장 길이를 가진 문장들끼리 그룹화하여 학습 속도를 높임.
  - 각 배치는 **약 25,000개의 입력 토큰과 25,000개의 출력 토큰**을 포함.

Baseline

- 하드웨어 및 학습 시간
  - **GPU 환경**: 8개의 NVIDIA P100 GPU에서 훈련
  - **학습 시간**:
    - 기본 모델(Base) → **12시간 (100,000 스텝)**
    - 큰 모델(Big) → **3.5일 (300,000 스텝)**
- 옵티마이저
  - **Adam 옵티마이저** 사용
  - **학습률 스케줄링** 적용
    - 초기 4,000 스텝 동안 선형 증가, 이후 스텝 수의 제곱근에 반비례하여 감소.
    - $warmup\_steps$ = 4000

$$lrate = d_{model}^{-0.5} \cdot min(step\_num \cdot warmup\_steps^{-1.5})$$

- 정규화 기법 (Regularization)
  - **Dropout**: 각 서브 레이어의 출력에 dropout 적용 (Base 모델에서는 10%)
  - **Label Smoothing**: $\epsilon_{ls} = 0.1$ 적용

- 정확도가 높은 모델일수록 오버피팅 방지 효과가 있음.

6. 결과 (Results)

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

- **기계 번역 성능**
  - 기존의 RNN 및 CNN 기반 모델보다 뛰어난 성능을 보임.

    ✅ Transformer (Big)가 기존 최고 성능 대비 BLEU 점수 2.0 이상 향상.

    ✅ 훈련 비용(연산량)은 기존 모델보다 적음.

Table 3: Variations on the Transformer architecture. Unlisted values are identical to those of the base model. All metrics are on the English-to-German translation development set, newstest2013. Listed perplexities are per-wordpiece, according to our byte-pair encoding, and should not be compared to per-word perplexities.

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps	PPL (dev)	BLEU (dev)	params $\times 10^6$
base	6	512	2048	8	64	64	0.1	0.1	100K	4.92	25.8	65
(A)				1	512	512				5.29	24.9	
				4	128	128				5.00	25.5	
				16	32	32				4.91	25.8	
				32	16	16				5.01	25.4	
(B)					16					5.16	25.1	58
					32					5.01	25.4	60
(C)	2									6.11	23.7	36
	4									5.19	25.3	50
	8									4.88	25.5	80
		256			32	32				5.75	24.5	28
		1024			128	128				4.66	26.0	168
			1024							5.12	25.4	53
			4096							4.75	26.2	90
(D)							0.0			5.77	24.6	
							0.2			4.95	25.5	
								0.0		4.67	25.3	
								0.2		5.47	25.7	
(E)	positional embedding instead of sinusoids									4.92	25.7	
big	6	1024	4096	16			0.3		300K	**4.33**	**26.4**	213

- **모델 변형 실험**
  - **어텐션 헤드 개수 변화**
    - 8개 → 최적 성능
    - 너무 많거나 적으면 성능 저하
  - **위치 인코딩 방식 비교**
    - 학습된 임베딩 vs 사인-코사인 함수 → 성능 차이 거의 없음
  - **드롭아웃 및 모델 크기 증가**
    - 모델이 커질수록 성능 향상
    - 드롭아웃 적용 시 BLEU 점수 상승

7. 결론 (Conclusion)

7.1. 의의

✳️ RNN 제거하고 완전한 병렬 처리가 가능한 Transformer 제안함.

- 이전까지 기계 번역 및 자연어 처리(NLP)에서 가장 강력한 모델 → RNN 기반의 구조

- 순차적 연산이 필수적이어서 병렬화가 어렵고 학습 속도가 느리다는 단점.

⇒ 이러한 문제를 해결하기 위해 **순환 구조를 완전히 제거하고, 오직 Self-Attention) 메커니즘만을 활용**하는 새로운 모델을 제안하여 병렬 연산 가능하고 RNN 기반 모델보다 훨씬 빠른 학습 속도를 보이며 더 우수한 성능을 기록함.

✳️ Self-Attention만을 이용한 최초의 시퀀스 변환 모델

- 기존 연구에도 Attention 메커니즘 존재했지만, 대부분 RNN 또는 CNN과 함께 사용되는 보조적 역할.
- Transformer는 RNN 없이도 시퀀스 변환 가능함을 처음으로 입증함.

✳️ 이후 NLP 모델 발전에 미친 영향

- Transformer의 성공 이후, NLP에서는 기존의 RNN 기반 모델들이 점차 사라지고, **Transformer 기반 모델이 표준이 됨**.
  - **BERT (2018)** → 사전 학습(Pre-training) 기법 도입, 문맥을 양방향으로 학습 (Bidirectional Encoding)
  - **GPT (2018~2023)** → 대규모 언어 모델(LLM)의 기초가 된 모델, 텍스트 생성 최적화
  - **T5, BART, T-GPT, LLaMA 등** → 다양한 NLP 태스크에 맞춰 Transformer 기반 모델들이 발전

7.2. 한계점

⛔ 입력 길이가 길어질수록 계산량이 급격히 증가. ($O(n^2)$ 문제)

- Self-Attention→ 입력 길이 n에 따라 $O(n^2)$의 연산량이 필요

  ⇒ 입력 시퀀스가 길어질수록 계산량이 빠르게 증가해서, 긴 문장을 처리하는 데 한계.
- 대규모 데이터 또는 긴 문장을 다루는 경우 메모리 사용량이 매우 커지는 문제가 발생.

💡 해결책

- 이후 연구에서 **효율적인 어텐션 기법 (Sparse Attention, Longformer, Performer 등)이 등장**하여 이 문제를 개선하고 있음.

⛔ 문장 길이 정보를 직접 학습하지 않음. (Positional Encoding 한계)

- RNN처럼 순차적으로 데이터를 처리하지 않기 때문에, 단어의 순서를 학습할 수 있도록 **Positional Encoding (위치 인코딩)**을 추가

⇒ 학습을 통해 최적화되지 않으며, 단순한 수학적 함수(사인, 코사인)로 정해진다는 한계.

- 문장 구조를 충분히 잘 학습하지 못할 수 있다는 문제를 유발할 수 있음.
- 이후 연구에서 학습 가능한 위치 임베딩 (Learnable Positional Embedding) 방식이 제안되어, 이 문제를 해결하려는 시도가 이루어짐.

⛔ 대규모 데이터 학습 필요 → 작은 데이터셋에서 과적합 위험.

- 병렬 연산과 높은 모델 용량을 활용하여 강력한 성능을 발휘하지만, 많은 양의 데이터를 학습해야 최상의 성능을 낼 수 있음.
- 데이터셋이 작으면 **과적합(Overfitting) 위험**이 커질 수 있음.

💡 해결책

- 사전 학습된 Transformer 모델(BERT, GPT 등)을 활용하여 적은 데이터에서도 높은 성능을 낼 수 있도록 함.