Thesis review study

Attention Is All You Need

조은비, 전민하

Abstract

- 기존에는 시퀀스간 변형을 활용한 RNN, CNN 기반 모델이 많이 사용됐다.
- 그중 가장 좋은 성능을 보인 모델은 인코더, 디코더 아키텍처에 Attention 매커니즘을 활용한 모델이었다.
- 이에 본 논문은 recurrence, convolutions를 제거하고 오직 Attention 매커니즘에 기반한 ‘Transformer’ 아키텍처를 소개한다.
- Transformer는 영어를 독일어, 불어로 번역하는 과정에서 기존 모델보다 더 개선된 성능을 보여줬다.

AGENDA

01

Introduction

Recurrent models

시퀀스에 포함된 각각의 토큰들에 대한 순서 정보를 먼저 정렬시킨 뒤에 반복적으로 입력으로 넣어서 hidden state 값을 갱신시키는 방식. 토큰 개수만큼 뉴럴 네트워크에 입력을 넣어야 해서 병렬적 처리가 어렵다. 메모리, 속도 측면 비효율성 야기.

Attention mechanisms

출력 단어를 만들때마다 어떤 정보가 가장 중요한지 가중치를 부여하고, 가중치가 적용돼 곱해진 hidden state 값을 사용한다.

Transformer

Recurrence 속성을 완전히 없애버리고, 완전히 Attention mechanisms에 의존했다. 한번의 행렬곱에 위치정보가 포함된 전체 시퀀스를 처리할 수 있다. 순차적인 입력이 아니라서 병렬처리가 가능하다. 성능이 훨씬 좋아졌다.

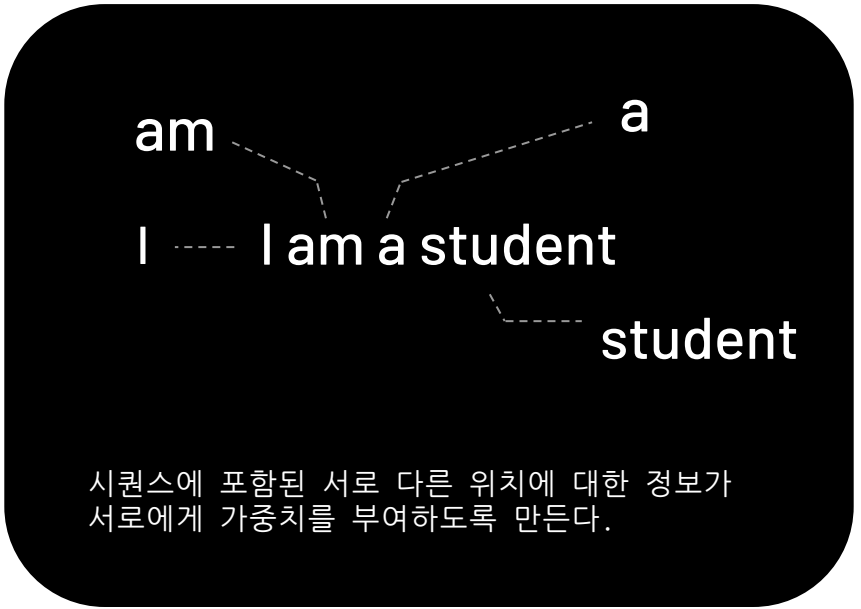
02

Backgroud

Self-Attention

본 논문에서는 전적으로 Self-Attention에 의존해 최초로 시퀀스간 변형이 가능한 네트워크를 만들었다.

- Self-Attention은 문장 스스로 Attention을 수행해서 representation을 학습하게 한다.



03

Model Architecture

3.1 Encoder and Decoder Stacks

참고) 본 논문에서는 인코더와 디코더 레이어를 6번 쌓을 수 있게 만들었고, multi-head attention 수행 시 인코더의 output 값에 attention을 수행할 수 있게 했다.

1-5 과정을 N번만큼 반복. 총 N개의 인코더 레이어가 반복 수행된다. 이후 마지막 출력값을 디코더에서 사용할 수 있도록 한다.

이후 Feed Forward 레이어, Linear 레이어를 거치고 Softmax를 해서 각각의 출력 문장에 포함된 단어들이 어떤 단어에 해당하는지 구할 수 있도록 한다.

5 Add&Norm 후 정규화를 한다. 이후 Feed Forward 레이어를 거치고 정규화를 하기 전까지는 입출력 디멘젼이 같다.

4 Multi-Head Attention은 쿼리, 키, 벨류 값이 동일하며 self attention으로 동작한다.

3 쿼리, 키, 벨류로 복제돼 입력된다.

2 문장에 포함된 각 단어의 위치정보를 인코딩해서 입력하기 위함

1 입력 임베딩과 같은 디멘젼으로 합쳐서 임베딩 벡터로 사용

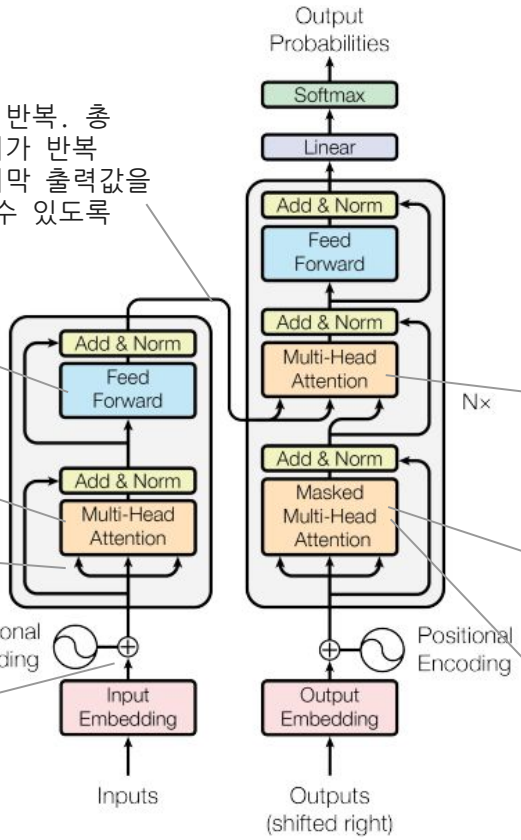


Figure 1: The Transformer - model architecture.

디코더 파트의 두번째 Attention에서는 키, 벨류값이 인코더 파트에서 왔고, 쿼리 값은 디코더에 있기 때문에 인코더 파트에서 어떤 정보를 참고해야 하는지에 대해 Attention을 수행한다.

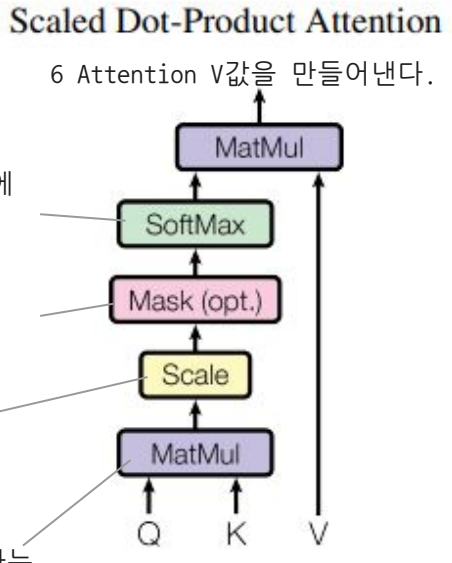
학습 수행 시 Mask를 씌워서 뒤쪽에 있는 단어를 미리 알지 못하도록 한다. 이전에 등장한 단어만 참고해서 Attention할 수 있다.

디코더 파트의 첫번째 Attention은 쿼리, 키, 벨류 값이 같아서 Self Attention이 수행된다.

3.2 Attention

쿼리는 특정 키에게 질문을 하는 주체. 키는 attention을 수행할 대상을 의미한다. 쿼리가 키에게 ‘I am happy’ 중에 어떤 단어가 제일 중요해? 라고 질문하는 것과 같다.

3.2.1 Scaled Dot-Product Attention



6 Attention V값을 만들어낸다.

4 Softmax를 통해 각각의 키에 대해 중요도를 파악하는 값을 확률 형태로 바꿔준다.

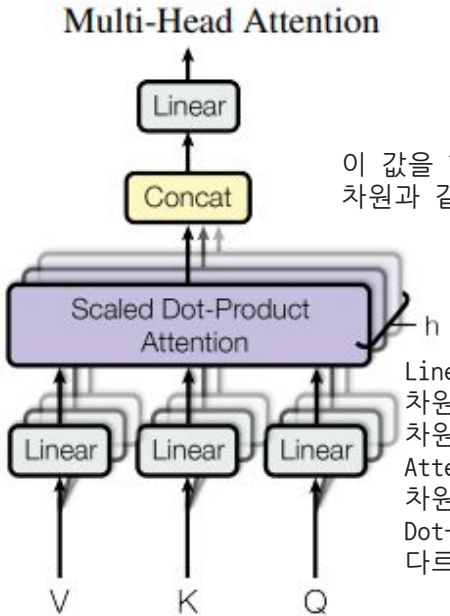
3 필요할 때 선택적으로 사용

2 Softmax에 들어가는 값을 스케일링 하기 위해 scale 레이어를 포함한다.

1 쿼리가 키에게 질문하는 게 행렬곱으로 이뤄진다.

5 그렇게 나온 확률값을 각각 벨류와 곱해서

3.2.2 Multi-Head Attention



이 값을 합쳐서 기존의 임베딩 차원과 같게 만든다.

Linear 레이어는 임배딩 차원을 키, 벨류, 쿼리의 차원으로 바꾼다. 이후 Attention을 수행해서 동일한 차원이 나오게 되면 Scaled Dot-Product를 head마다 서로 다르게 수행한다.

입력값은 벨류, 키, 쿼리에 각각 복제가 돼 들어갈 수 있다.

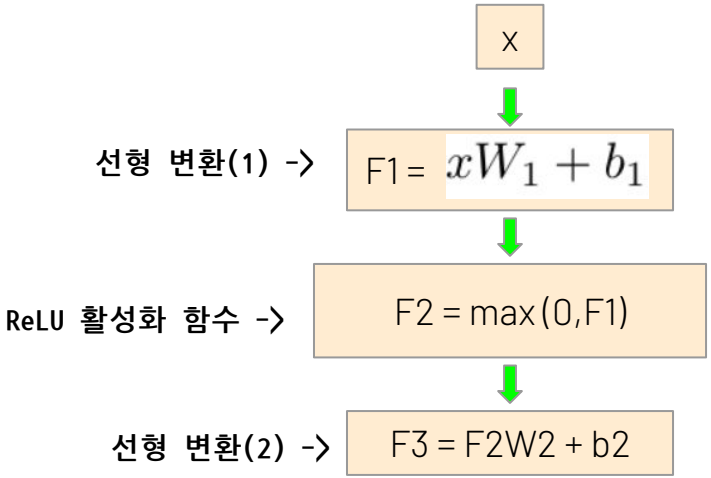
multi-head attention은 Transformer에서 위치마다 다른 방식으로 쓰였다.

encoder-decoder attention	• 쿼리는 디코더에서 오고, 키, 벨류값은 인코더 출력에서 가져온다. • 소스문장의 문장들 중에 어떤 단어에 집중해야 하는지를 계산하는 과정
encoder contains self-attention	• 인코더 파트에서 사용. • 쿼리, 키, 벨류가 모두 같다.
self-attention	• 디코더 파트. 맨 처음에 입력 임베딩 들어왔을 때 self-attention 수행. • 마스크 씌워서 소프트맥스에 들어가는 값이 -무한대가 될 수 있도록 하고, 각 단어가 미리 등장했던 단어만 참고할 수 있도록 했다.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- FFN: 인코더와 디코더에서 공통으로 가지고 있는 서브층
- 각 단어의 위치별로 독립적이고 동일하게 적용

  -> 모든 단어가 같은 FFN 네트워크를 거침
- 차원: 인코더에 처음 넣었을 때의 input과 동일

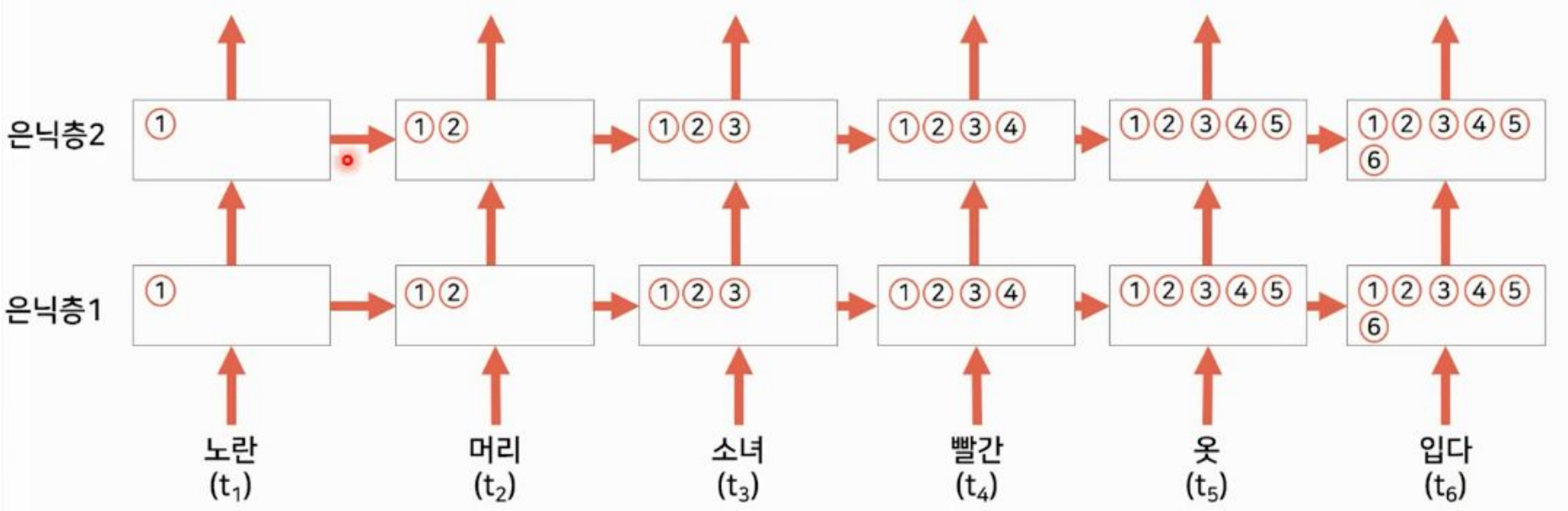


- Parameter (W, b)
- 각 레이어마다 다른 값 사용
- 같은 레이어 내에서는 동일한 값을 사용

input과 output의 token들을 embedding

- 입력과 출력 토큰을 dmodel 차원의 벡터로 변환하기 위해 학습된 임베딩을 사용
- decoder 출력에서 다음 토큰 예측 확률을 얻기 위해 학습된 선형 변환과 softmax 함수를 사용.
- 두 embedding layer와 softmax 이전 선형 변환 사이에서 동일한 가중치 행렬을 공유.

RNN: 순차적으로 cell에 입력되므로, 토큰의 위치(순서)정보가 보존됨



Input Embedding: Input에 입력된 데이터를 컴퓨터가 이해할 수 있도록 행렬 값으로 변환

- 각각의 벡터 차원은 해당 단어의 피처 값을 보유함
- 서로 다른 단어의 피처값이 유사할 수록 벡터 공간의 임베딩 벡터는 가까워짐

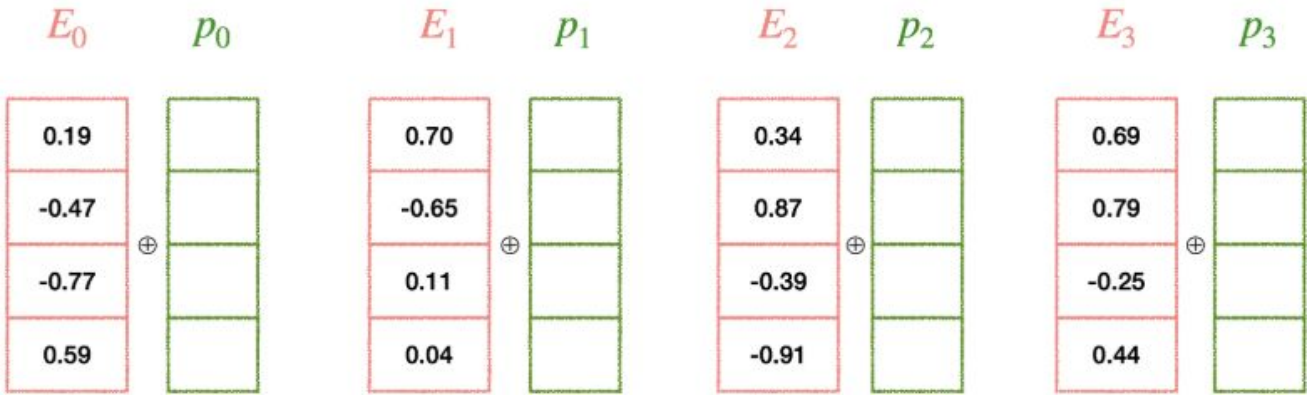
주기함수인 사인 코사인 함수 이용 *-> 상대적인 위치 정보를 얻기 용이함*

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

i가 짝수: sin 함수 사용
i가 홀수: cos 함수 사용

- pos: position
- i: 차원
  - 위치 정보를 표현하는 positional embedding의 차원 == 단어 임베딩의 차원
- d_model: transformer의 인코더, 디코더에서 정해진 입출력의 크기

Transformer - 토큰을 한번에 병렬로 처리하므로 단어의 순서를 **알 수 없음!**



- 각 단어 벡터 + positional encoding을 통해 얻은 위치 정보
  -> 시공간적 속성이 임베딩된 벡터가 생성, 시퀀스 정보가 보존!

Self-Attention
VS
Recurrent Layer
VS
Convolutional Layer

〈주요 비교 요소〉

1. **계산 복잡도**: 각 레이어에서 필요한 계산량.
2. **병렬 처리 가능성**: 계산을 병렬화할 수 있는 능력, 최소 순차적 연산 횟수로 측정.
3. **장기 의존성 학습의 경로 길이**: 입력과 출력 시퀀스 간의 신호가 얼마나 빨리 전달될 수 있는지,
   즉 장기 의존성을 학습할 수 있는 용이성.

입력 데이터의 첫 단어가 출력 데이터의 마지막에 영향을 미치는 정도

4. Why Self-Attention?

	Self-Attention	Recurrent Layer	Convolutional Layer
계산 복잡도	일정한 수의 연산으로 모든 위치 연결	$O(n)O(n)O(n)$ 순차적 연산 필요	합성곱 커널에 따라 다름. $O(nk)O(nk)O(nk)$ ¦ $O(\log_k n)O(\log n)$
병렬화 가능성	높음	낮음 (순차적 처리)	병렬화 가능, but 순차적 처리로 인해 복잡도 증가
장기 의존성 학습	경로가 짧아서 용이	경로가 길어서 어려움	경로가 길어서 어려움
최대 경로 길이	일정한 수의 연산으로 연결됨	$O(n)O(n)O(n)$ 순차적 경로	$O(nk)O(nk)O(nk)$ ¦ $O(\log_k n)O(\log n)$
계산 효율성	시퀀스 길이가 짧을 때 효율적, 매우 긴 시퀀스에서 이웃만 고려 시 성능 개선 가능	비교적 느림 (순차적 연산)	많은 연산 필요
해석 가능성	각 주의 헤드가 구문적 및 의미적 구조와 관련된 행동을 학습	어려움	어려움

- **Self-Attention**은 계산 효율성, 병렬화가 용이하고, **장기 의존성 학습에 유리함**.

  시퀀스의 모든 단어를 한번에 고려할 수 있기 때문