Attention Is All You Need

Abstract

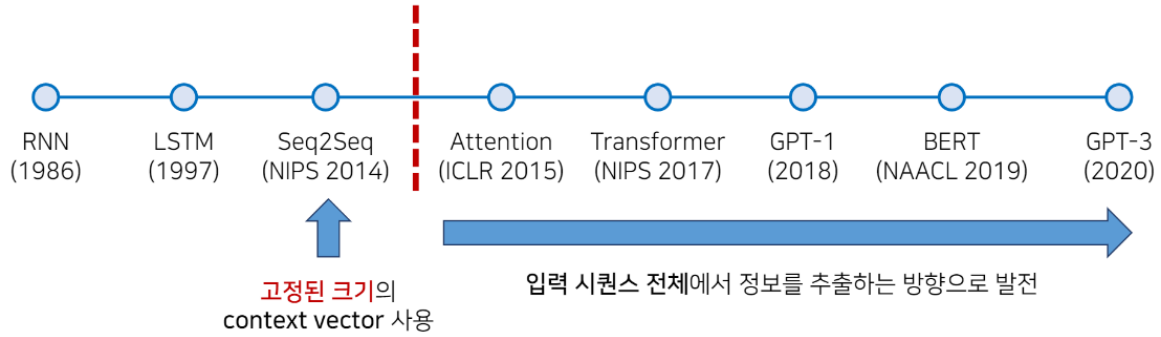
우수한 Sequence transduction 모델들은 대체로 complex rnn 혹은 cnn에 기반을 두고 있으며, encoder와 decoder의 구조로 이루어져 있다. 그 중에서도 가장 좋은 성능의 모델들은 encoder와 decoder를 attention 메커니즘으로 연결하고 있다. 이 논문에서는 새로운 단순한 네트워크 구조를 제안하는데, 이것이 바로 transformer다.

transformer의 아키텍처의 핵심이 되는 것이 바로 Attention 메커니즘이다.

1. Introduction

Recurrent neural networks(RNN)은 sequence modeling과 transduction 문제(language modeling이나 기계 번역 등)를 다루는 데 탁월한 솔루션으로 자리잡았다.

딥러닝 기반의 기계 번역 발전 과정을 다음과 같이 정리해볼 수 있다.



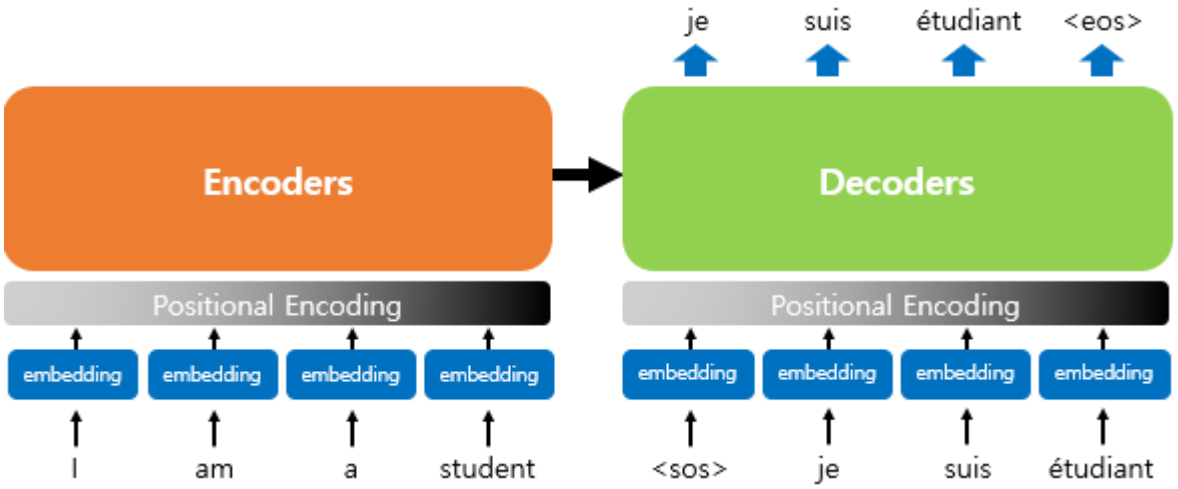
(*출처: 나동빈님의 Attention Is All You Need 발표자료)

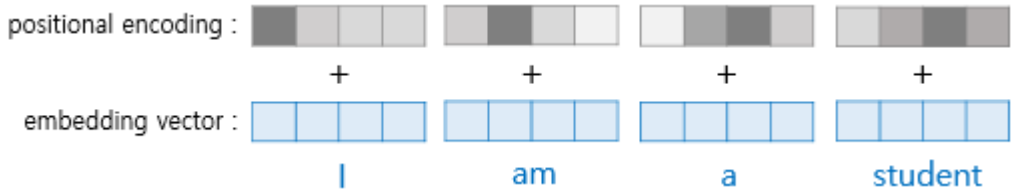
Seq2Seq까지만 해도, 고정된 크기의 context vector에 문장을 압축해서 집어넣는 과정을 거쳐야했기 때문에 성능적인 한계점(: Bottleneck이 발생해서 성능 하락)이 존재했다.

단어가 입력될 때마다 이전까지 입력되었던 단어들에 대한 정보를 포함한 hidden state값을 받고, hidden state 값을 갱신해나가는 방식. 마지막 hidden state 값을 하나의 context vector로서, 전체 문맥을 담고 있다고 여긴다. 하나의 context vector가 소스 문장의 모든 정보를 가지고 있어야 하기 때문에 성능이 저하될 수밖에 없다.

다양한 크기를 가진 경우들을 모두 고정된 크기의 context vector로 압축한다는 점에서 bottleneck 문제가 발생하게 된다. 이후 Seq2Seq 모델에 Attention 메커니즘을 도입하는 방식도 제안되었고, context vector와 함께 Decoder가 Encoder의 모든 출력을 참고하도록 만듦으로써, source 문장의 정보를 모두 고려할 수 있게 만들어 성능을 높이기도 했다.

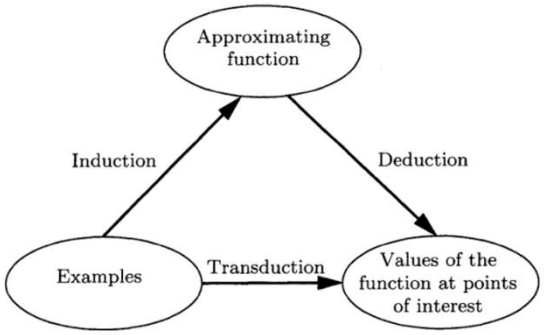
Transformer에서는 RNN을 사용할 필요 없이 오직 Attention 기법만을 사용하는 아키텍처를 제안하여 더욱 높은 성능의 모델을 얻어낼 수 있었다. RNN 기법을 사용하지 않으므로, transformer에서는 문장의 구성성분들의 각 순서(위치) 정보를 포함하고 있는 임베딩을 사용하는데, 이를 위해 positional Encoding을 사용한다. 문장의 정보 및 위치 정보까지 포함한 입력값을 attention 구조에 넣고 진행된다.





Attention 모델의 등장 이후로 입력 시퀀스 전체에서 정보를 추출하는 방향으로 발전하는 경향으로 여러 논문이 발표됐다.

*transduction

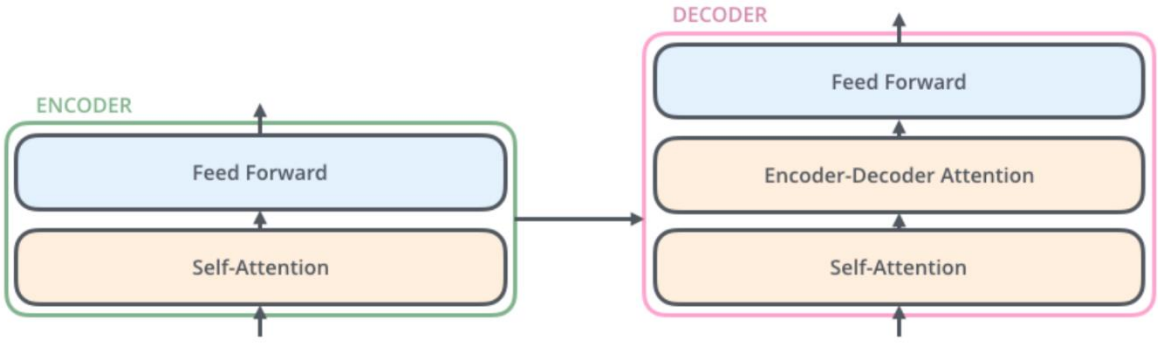


* 변환(transduction)은 주어진 데이터에서 관심 있는 점에 대해 알려지지 않은 함수의 값을 유도한다.

2. Background

Self attention 개념

- self-attention은, encoder에 입력 벡터가 들어가기 전, 입력 벡터와 입력 벡터 자신 사이의 attention을 구해서, encoder의 입력 자체에 "상황"을 인코딩하는 방법이다.



3. Model Architecture

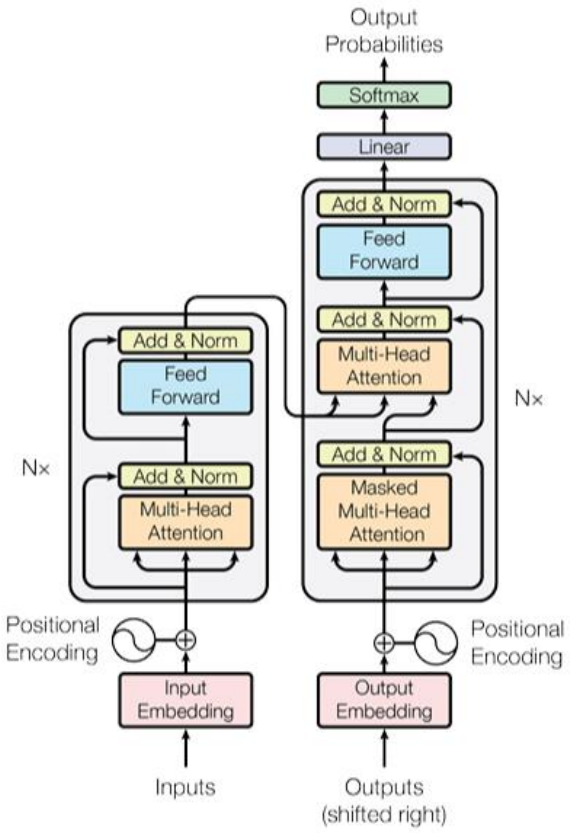


Figure 1: The Transformer - model architecture.

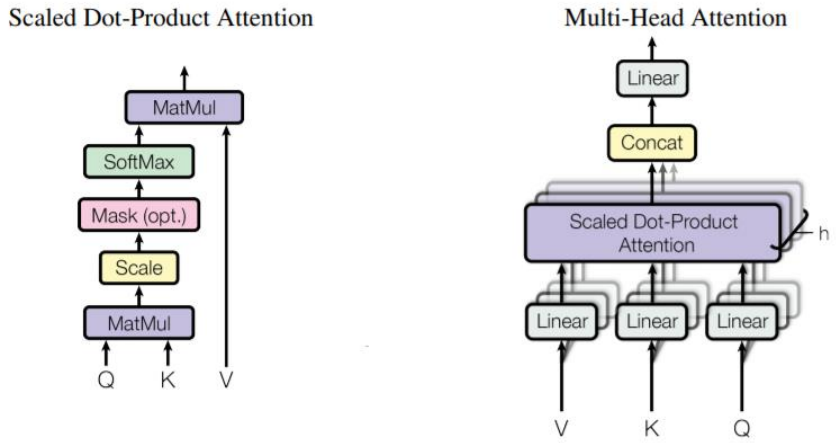
3.1 Encoder and Decoder Stacks

Encoder와 Decoder의 경우 둘 모두 크게 3가지로 Positional Encoding, Multi-Head Attention과 Feed Forward NN이 사용된다. 논문에서는 Encoder와 Decoder의 N 값을 6으로 주어 Transformer 모델을 구성했다.

3.2 Attention

Attention function은 vector query와 vector key-value 집합 쌍을 query, keys, values, output이 모두 벡터인 output에 mapping하는 것이다. Output은 각 values의 weighted sum으로 계산되며, 여기서 각 value에 할당된 weight는 query와 해당 key의 compatibility function에 의해 계산된다.

Q: 영향을 받는 벡터 K: 영향을 주는 벡터 V: 주는 영향의 가중치 벡터



3.2.1 Scaled Dot-Product Attention

Scaled Dot-Production attention은 위 그림의 왼쪽에 나와있다. **Input은 d_k차원의 query, key와 d_v차원의 value**로 구성된다. Key와 query의 dot product를 계산하고, 각각의 값을 √(d_k)로 나눈 후 softmax 함수를 적용하여 value에 대한 weight를 구한다.

Output matrix는 다음과 같이 계산한다.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

주로 사용되는 attention function은 additive attention과 dot-product (multiplicative) attention이다.

dot-product 계산법

values의 weight를 구하기 위해 query와 all key를 곱하고 d_k 를 각각 나누고 softmax 함수를 적용한다. 실제, Matrix Q로 변환해서 queries set를 동시에 attention function을 계산한다. Keys와 values 역시 matrices K와 V로 계산한다.

softmax를 적용하기 전 작은 gradients를 가지기 위해 $1/\sqrt{d_k}$ 을 곱해서 dot product을 조정한다.

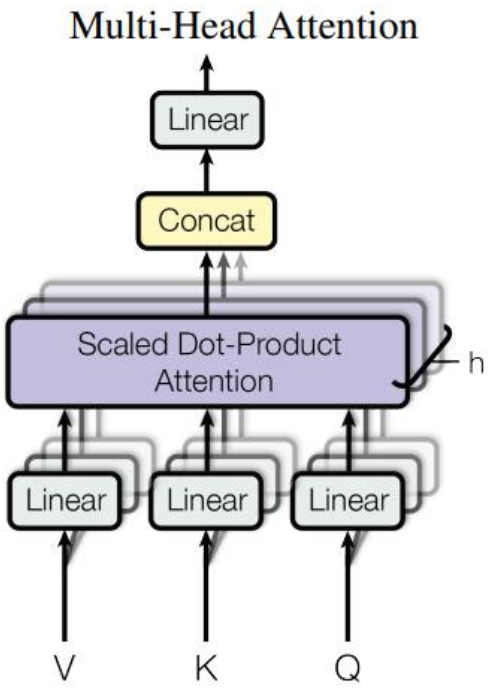
(논문에서는 dot products는 크기를 키우고, 큰 dk 는 좋지 않은 영향이 있을거라 의심하고 있다.)

두 방법은 이론적으로 복잡성이 비슷하지만, **dot-product attetion은 matrix를 통해 최적화된 연산을 구현할 수 있기 때문에 훨씬 빠르고 공간 효율적**이다.

d_k가 값이 작은 경우에는 dot-product와 scaled dot-product가 유사하게 수행되지만, 값이 큰 경우에는 후자가 우수하다. d_k 값이 클 때, dot-product의 size가 커지면서 softmax를 극도로 작은 gradient를 갖게끔 한다고 생각한다. 이를 개선하기 위해 **1/√(dk)만큼 스케일링**한다.

3.2.2 Multi-Head Attention

d_model 차원의 query, key, value를 사용하여 single attention을 수행하는 대신, 각각 d_k, d_k, d_v 차원에 대해 학습된 서로 다른 linear projection을 사용하여 query, key, value를 h회 linear projection하는 것이 유익하다는 것을 발견했다. 이러한 query, key, value의 각 projection version에서 attention funtction을 **병렬로 수행하여 d_v차원 output을 생성하고, 이를 concat하여 다시 d_model 차원의 output이 생성**된다.



Multi-head attention을 통해 모델은 다른 position의 서로 다른 representation subspaces의 정보에 공동으로 attend할 수 있다.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

h = 8개의 병렬 attention layer 혹은 head를 사용한다. d_k = d_v = d_model/h = 64를 사용한다. 각 head의 축소된 차원으로 인해 총 계산 비용은 전체 차원을 갖는 single-head attention 비용과 유사하다.

3.2.3 Applications of Attention in our Model

Transformer는 세 가지 방식으로 multi-head attnetion을 사용한다.

1. **"Encoder-Decoder Attention layer"** 에서 **query는 이전 decoder layer에서 얻고, key와**

value는 encoder의 output에서 얻는다. 이를 통해 decoder의 모든 position이 input sequence의 모든 position에 배치될 수 있다. 이는 sequence-to-sequence 모델에서 **일반적인 encoder-decoder attention 메커니즘과 동일**하다.

2. **"Encoder Self-Attention layer"** 는 **encoder에 존재하며 query, key, value가 동일**하며, 이는 encoder에 있는 이전 layer의 output이다. Encoder의 각 position은 이전 layer의 모든 position에 attend 할 수 있다.

3. **"Masked Decoder Self-Attention layer"**는 decoder에 존재하며, 마찬가지로, decoder의 self-attention layer는 각 position이 해당 position의 위치까지 docoder의 모든 position에 attned하도록 한다. auto-regressive propert를 유지하기 위해 decoder에서 leftward information flow을 막아야 한다 **(미래 시점의 단어를 볼 수 없도록 하는 것)**. 이를 위해 매우 작은 수를 부여하여 softmax 결과 0에 수렴하도록 하여 **masking을 수행**한다.우리는 잘못된 연결에 해당하는 소프트맥스 입력의 모든 값을 마스킹(-)로 설정)하여 스케일링된 도트 제품 주의의 내부에서 이를 구현한다.

3.3 Position-wise Feed-Forward Networks

attention sub-layer 외에도 encoder와 decoder의 각 layer에는 FFN가 포함되어 있으며, 이는 각 position에 개별적으로 동일하게 적용된다. 이것은 중간에 ReLU activation이 있는 두 가지 linear transformation으로 구성된다.

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{2}$$

linear transformation은 여러 position에서 동일하지만 layer마다 다른 파라미터를 사용한다. 이것을 설명하는 또 다른 방법은 커널 크기가 1인 두 개의 convolution을 사용하는 것이다. Input과 output의 차원은 d_model=512이고, 내부 layer의 차원은 d_ff=2048이다.

3.4 Embeddings and Softmax

다른 sequence trasduction 모델과 유사하게, 학습된 embedding을 사용하여 input token과 output token을 d_model차원의 벡터로 변환한다. 또한, 일반적으로 학습된 linear transformation과 softmax 함수를 사용하여 decoder output을 예측된 다음 token 확률로 변환한다. 본 모델에서, 두 embedding layer와 softmax 이전 linear transformation 사이에서 동일한 weight matrix를 공유한다. Inner layer에서 이러한 weight를 가중치에 √(d_model)을 곱한다.

3.5 Positional Encoding

Transformer는 순차적인 특성이 없고 이에 따라 **sequence의 위치 정보가 없기 때문에 positional 정보를 추가**해줘야한다. 이를 위해, **encoder와 decoder input embedding에 "positional encoding'을 추가**한다. Positional encoding은 embedding과 동일한 차원을 가진다.

본 연구에서는 다음과 같이 sin, cos 함수를 사용한다.

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

여기서 *pos*는 token의 위치를 의미하고, i는 차원을 의미한다. 즉, positional encoding의 각 차원은 사인파에 해당한다. geometric progression wavelengths 은 2π에서 10000까지이다. **PE_pos+k가 PE_pos의 선형 함수로 표현**될 수 있기에 모델이 상대적인 위치를 쉽게 학습할 수 있을 것이라 가정했기 때문에 이 함수를 사용한다.

학습가능한 positional embedding을 사용해 봤지만 동일한 성능을 보이는 것을 확인할 수 있었고, 더 긴 시퀀스에서도 추론이 가능한 사인파 버전을 사용한다.

4. Why Self-Attention

Self-attention layers와 recurrent and convolution layers와의 비교를 수행한다. 비교 방법은 symbol representations(x1, ..., xn)의 one variable-length sequence를 같은 길이 (z1, ..., zn)으로 mapping 이다. 우리가 self-attetion을 사용하는 이유는 세가지이다.

layer별 총 연산의 complexity.

1. 필요한 최소 sequential operations으로 측정한 병렬처리 연산량.
2. Network에서 long-range dependencies(장거리 의존성) 사이 path length. long-range dependencies의 학습은 번역 업무에서 핵심 task이다. 이러한 dependencies을 학습하는 능력에 영향을 미치는 한 가지 핵심 요소는 전달해야 하는 forward 및 backward signal의 길이다. Input의 위치와 output의 위치의 길이가 짧을수록 dependencies 학습은 더욱 쉬워진다. 그래서 서로 다른 layer types로 구성된 네트워크에서 input과 output 위치 사이 길이가 maximum 길이를 비교한다.