Transformer

2017년 구글의 "Attention is all you need"논문 발표

기존 seq2seq 구조인 encoder-decoder를 따르면서 RNN을 사용하지 않고, Attention mechanism을 사용

- 기존 seq2seq 모델의 한계

encoder-decoder구조로 구성되어 encoder는 입력문장을 하나의 vector로 표현해서 압축, decoder는 이 vector를 통해 출력 문장을 만들어냄

→ 이 방법은 입력 문장을 하나의 vector로 압축해서 입력 문장의 일부 정보가 손실됨

∴ Attention mechanism이 등장해서 이를 보정

Attention Mechanism

- transformer의 하이퍼파라미터
  1. encoder와 decoder의 입출력 크기(신경망 크기)

$$d_{model} = 512$$

  2. encoder와 decoder를 각각 6개씩

$$\text{num\_layers} = 6$$

  3. Attention을 8개로 분할해서 합침

$$\text{num\_heads} = 8$$

  4. feed forward 신경망의 은닉층 크기

$$d_{ff} = 2048$$

- Transformer 구조

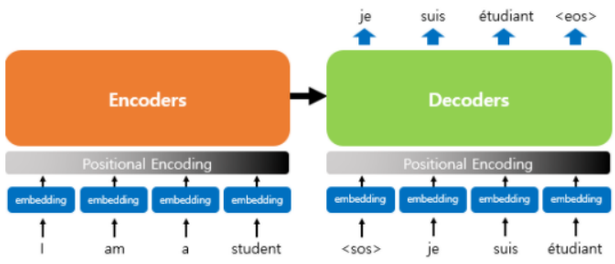
  Positional Encoding 사용

  → 단어 임베딩 벡터에 RNN이나 CNN을 사용하지 않고 대신 위치 정보를 포함하는 임베딩인 positional encoding 값을 더함

  pos : 입력 문장에서 임베딩 벡터의 위치, i : 임베딩 벡터 내 차원 인덱스 (짝수 : sin, 홀수 : cos)

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

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



  encoder & decoder의 Attention 3가지

  1. 인코더의 multi-head self-attention

     Query = Key = Value = 인코더 벡터

  2. 디코더의 multi-head masked self-attention

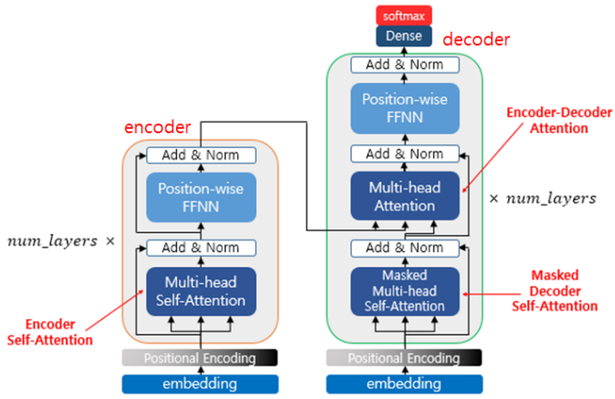
     Query = Key = Value = 디코더 벡터

  3. 디코더의 multi-head encoder-decoder attention

     Query = 디코더 벡터, Key = Value = 인코더 벡터

  multi-head → attention을 병렬적으로 수행함

  self-attention → Query, Key, Value가 같은 벡터의 출처 일 때



Encoder의 구조

- 인코더의 self-attention 동작 매커니즘
  1. Q, K, V 벡터 얻기

     dmodel의 차원을 가지는 단어 벡터들을 64차원의 Q, K, V 벡터로 변환 → 각 단어에 가중치 행렬 WQ, WK, WV를 곱해줌

     dmodel를 num_head로 나눈 값을 Q, K, V 벡터 차원으로 사용 (512/8=64)

  2. scaled dot-product attention

     각 Q벡터는 모든 K 벡터에 대해 attention score를 구하고, attention 분포를 구한 뒤에 모든 V벡터를 가중합 하여 attention value를 구함

     transformer에서는 scaled dot-product attention을 사용

     $$score\ function(q,k) = q \cdot k/\sqrt{n}$$

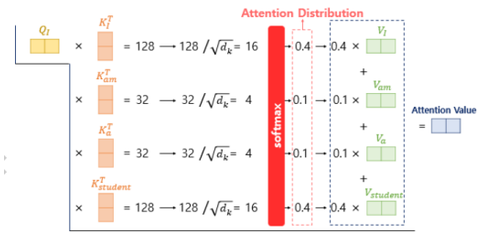
     ex) "I am a student"에서 I에 대한 Q벡터

     Q벡터에 대해 모든 K벡터에서 attention score를 구함(각 단어와 얼마나 연관되어 있는지)

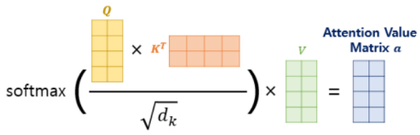
     attention score = q벡터(64) x k벡터(64) / √k벡터 차원(8)

     attention score에 softmax함수 적용해 attention 분포를 구함

     attention 분포와 각 V벡터를 가중합하여 attention value(=context vector)를 구함



     실제로는 벡터 연산이 아니라 행렬 연산으로 각 Q행렬을 일렬 계산

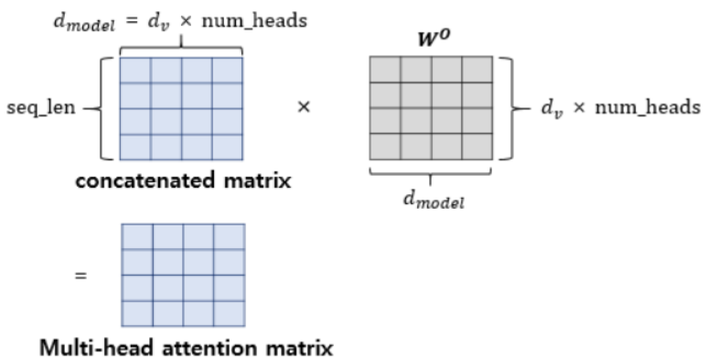


$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

  3. multi-head attention

     transformer에서는 dmodel의 차원을 num_head개로 나눈 Q, K, V에 대해 num_haed개의 병렬 attention 수행 ⇒ 8개의 attention을 concatenate해줌 : (입력 단어 수, dmodel) 크기가 됨

     concatenate 매트릭스에 가중치 행렬 W0를 곱한 것이 multi-head attention의 최종 결과

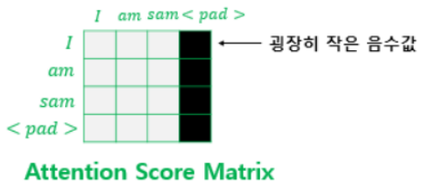


- 패딩 마스크

  scaled-dot product attention 함수 내에서 수행

  ⇒ <pad>라는 공백 토큰이 있는 경우 attention에서 제외하기 위해 유사도를 구하지 않도록 masking을 해줌

  → key에 <pad>가 있는 경우 해당 열 전체를 -1e9를 곱해줌 (행:query, 열: key)

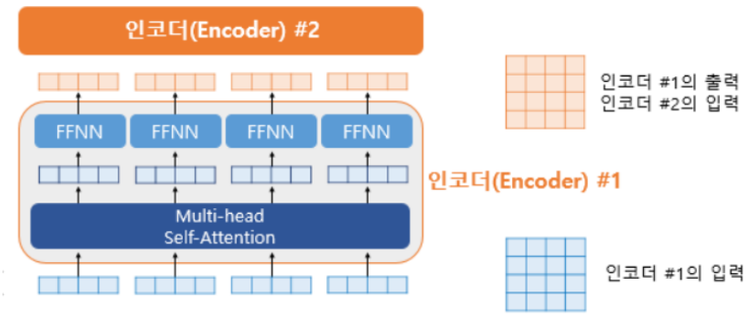


- 포지션-와이즈 피드 포워드 신경망 (Position-wise FFNN)

$$FFNN(x) = MAX(0, xW_1 + b_1)W_2 + b_2$$

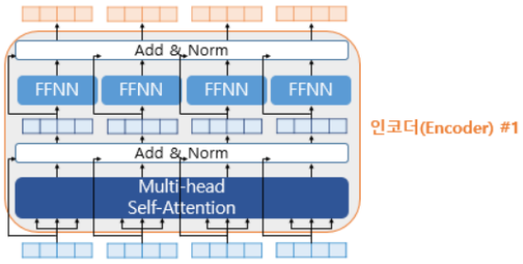
  x = multi-head attention의 결과 행렬

  W1, W2, b1, b2는 한 인코더 층 안에서는 동일하지만 인코더 층마다는 다른 값을 가짐



- Residual connection(잔차 연결)과 layer normalization(층 정규화)

  ⇒ Add & Norm (두 연산 수행 후 결과 : LN=LayerNorm(x+Sublayer(x))



  Residual connection : multi-head attention의 입력 + multi-head attention의 출력 (x+Sublayer(x))

  layer normalization : 텐서의 마지막 차원에 대해 평균과 분산을 구하고 정규화 (LayerNorm(xi))

Decoder의 구조

- 디코더의 masked multi-head self attention

  transformer에서는 전체 문장을 한번에 입력받아 각 시점의 단어를 예측

  → 현재 시점의 단어를 예측하고자 할 때, 미래 시점의 단어까지 참고할 수 있는 현상이 발생 ⇒ 현재 시점보다 미래에 있는 단어들을 참고하지 못하도록 look-ahead mask 도입

  attention score 행렬에서 masking을 적용, 나머지는 인코더의 self-attention과 동일 + 패딩 마스크도 동일하게 적용

- 인코더 - 디코더 attention

  Query는 디코더, Key와 Value는 인코더의 마지막 층의 행렬에서 얻어짐

  나머지 계산은 다른 attention과 동일

- position-wise FFNN