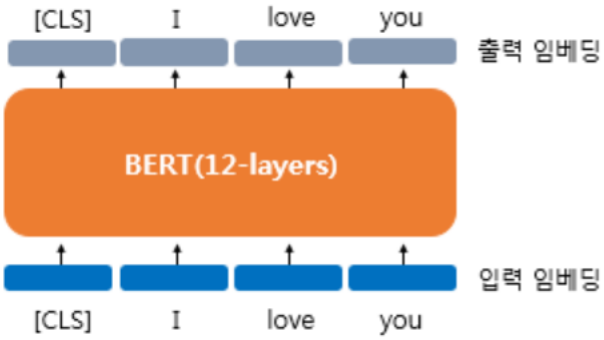
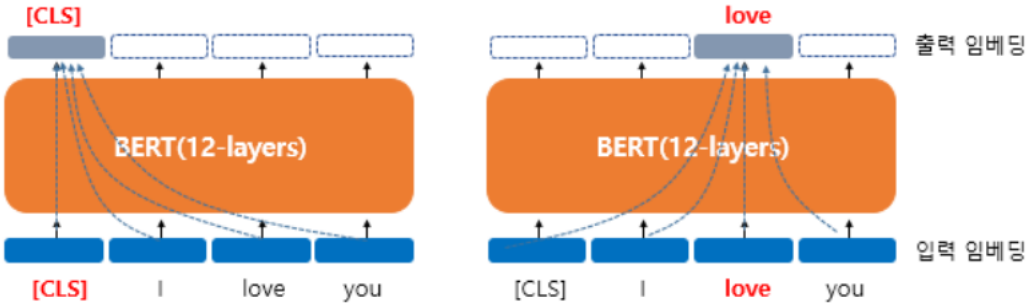
BERT (Bidirectional Encoder Representations from Transformers, 트랜스포머의 양방향 인코더 표현)

- BERT는 트랜스포머를 이용하여 구현되었다.
- 수많은 레이블이 없는 텍스트 데이터로 사전 훈련된 언어 모델이다.
- BERT가 높은 성능을 가진다.
  - ■ 레이블이 없는 방대한 데이터로 사전 훈련된 기존 모델을 가지고, 레이블이 있는 다른 작업에서 추가 훈련과 함께 하이퍼파라미터를 재조정하여 이 모델을 사용하면 성능이 높게 나온다.
    - ◆ 즉, 파인 튜닝(Fine-tuning) 과정을 기친 BERT는 다른 용도로도 사용할 수 있다. -> 뒤에서 설명

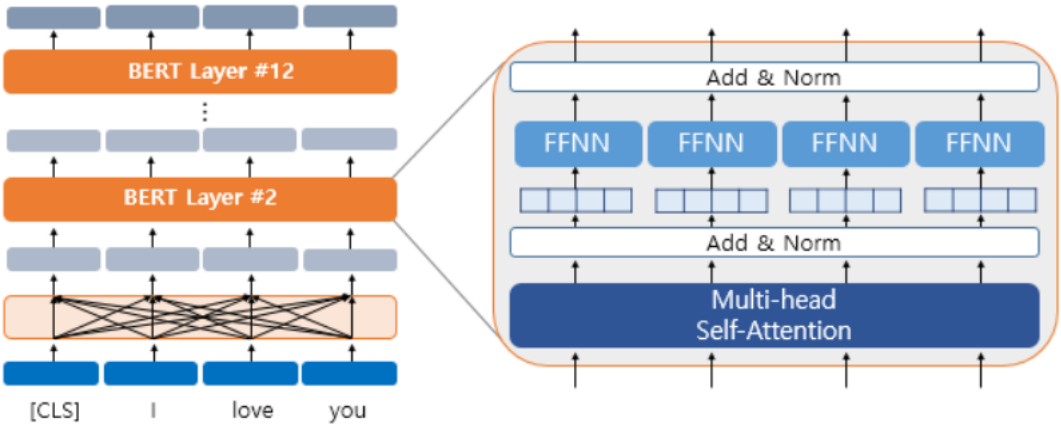
- BERT는 문맥을 반영한 임베딩을 사용하고 있다.



- BERT의 입력은 딥 러닝 모델들과 마찬가지로 임베딩 층을 지난 임베딩 벡터(단어의 정보가 담긴 벡터)이다.
  - ■ 모든 단어들은 하이퍼파라미터에서 지정한 차원의 임베딩 벡터가 되어 BERT의 입력으로 사용된다.
  - ■ BERT는 내부적인 연산을 거친 후, 동일하게 각 단어에 대해 N차원의 벡터(이것이 임베딩 벡터)를 출력한다.
- 위 그림에서는 BERT가 각 768차원의 [CLS], I, love, you라는 4개의 벡터를 입력으로 받아서 (입력 임베딩) 동일하게 768차원의 4개의 벡터를 출력하는 모습(출력 임베딩)을 보여준다.



- BERT의 연산을 거친 후 출력 임베딩은 문장의 문맥을 모두 참고한 문맥을 반영한 임베딩이 된다.
    - 문장의 문맥을 참고할 수 있는 이유는 트랜스포머 층을 거쳤기 때문
- 왼쪽 그림의 [CLS]라는 벡터는 BERT의 초기 입력으로 사용되었을 임베딩 당시에는 단순히 임베딩 층을 지난 임베딩 벡터였지만, BERT를 지나고 나서는 모든 단어 벡터들을 참고한 후 문맥 정보를 가진 벡터가 된다.
    - 점선 화살표는 모든 단어를 참고하였음을 나타낸다.
    - 이는 [CLS]라는 단어 벡터뿐만 아니라 오른쪽 그림에서 love라는 단어 벡터도 모든 단어를 참고함을 보여준다.



- 하나의 단어가 모든 단어를 참고하는 연산은 BERT의 12개 층에서 전부 이루어지는 연산이다.
- 12개 층을 모두 지난 후 최종적으로 출력 임베딩을 얻게 되는 것
- 위 그림에서 첫번째 층에 입력된 단어가 모든 단어를 참고한 후 다른 층으로 넘어가는 것을 볼 수 있다.
    - 즉, BERT의 첫번째 층의 출력 임베딩은 BERT의 두번째 층에서는 입력 임베딩이 된다.

- BERT가 모든 단어를 참고하여 문맥을 반영한 출력 임베딩을 얻을 수 있는 이유는 셀프 어텐션 때문이다.

  - BERT는 기본적으로 **트랜스포머 인코더를 12번 쌓은 것**이므로 내부적으로 각 층마다 멀티 헤드 어텐션과 포지션 와이즈 피드 포워드 신경망을 수행하고 있다.

  - 멀티 헤드 어텐션이란 트랜스포머에서 한 번의 어텐션을 하는 것 보다 어텐션을 병렬로 여러 번 사용하는 것이 더욱 효과적으로 성능이 올라가기 때문에 사용한다.

    - d_model(트랜스포머의 인코더와 디코더에서 정해진 입력과 출력의 크기, 여기서는 인코더의 크기)의 차원을 num_heads(병렬로 처리할 어텐션의 수)개로 나누어 d_model/num_heads의 차원을 가지는 Q, K, V에 대해 num_heads개의 병렬 어텐션을 수행

    - 논문에서는 num_heads를 8개로 정했고, 8개의 병렬 어텐션이 이루어짐

    - 즉, 어텐션이 8개로 병렬로 이루어지며, 이때 각각의 어텐션 값 행렬을 어텐션 헤드라고 부른다.

      - 이때 가중치 행렬의 값은 8개의 어텐션 헤드마다 전부 다르다.

    - 어텐션을 병렬로 수행하는 이유는 다른 시각으로 정보들을 수집하겠다는 의미이다.

    - '그 동물은 길을 건너지 않았다, 왜냐하면 그것은 너무 피곤했기 때문이다' 라는 문장을 예로 들면

      - 단어 '그것'이 쿼리였다고 하면, Q 벡터로부터 다른 단어와의 연관도를 구하였을 때 첫번째 어텐션 헤드는 '그것'과 '동물'의 연관도를 높게 본다면

      - 두 번째 어텐션 헤드는 '그것'과 '피곤하였기 때문이다'의 연관도를 더 높게 볼 수 있다.

      - 각 어텐션 헤드는 전부 다른 시각에서 보고 있기 때문

    - 병렬 어텐션을 모두 수행하였다면 모든 어텐션 헤드를 연결한다.

  - 포지션 와이즈 피드 포워드 신경망은 멀티 헤드 어텐션의 결과로 나온 가중치 행렬에 대해 연산을 수행

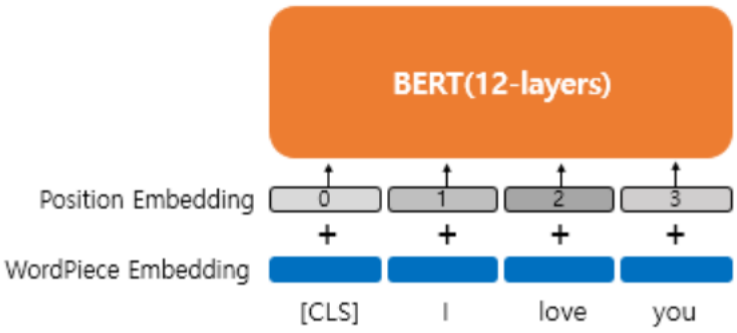
    - 각 개별 단어마다 신경망이 적용된다. (position-wise인 이유)

    - FFNN은 가중치의 반복적인 업데이트를 통해 출력 값의 에러를 최소화한다.

    - 가중치 업데이트는 경사 하강법을 이용하여 에러를 역전파 하여 구한다.

- BERT는 단어보다 더 작은 단위로 쪼개는 **서브워드 토크나이저**를 사용한다.
- BERT가 사용한 토크나이저는 WordPiece 토크나이저로 바이트 페어 인코딩(BPE)의 유사 알고리즘이다.
    - 동장 방식은 BPE와 조금 다르지만, 글자로부터 서브 워드를 병합해가는 방식으로 최종 단어 집합(Vocabulary)을 만드는 것은 BPE와 유사함
- 서브워드 토크나이저는 기본적으로 자주 등장하는 단어는 그대로 단어 집합에 추가
- 자주 등장하지 않는 단어의 경우 더 작은 단위인 서브 워드로 분리하여 단어 집합에 추가
- 단어 집합이 만들어지면, 이 단어 집합을 기반으로 토큰화를 수행
    - 즉, 입력한 단어가 BERT의 단어 집합에 해당 단어가 존재하지 않으면 더 작은 단위로 쪼갠다.

- 트랜스포머에서는 **포지셔널 인코딩**이라는 방식으로 단어의 위치 정보를 표현
- 포지셔널 인코딩은 sin함수와 cos함수를 사용하여 위치에 따라 다른 값을 가지는 행렬을 만들어 이를 단어 벡터들과 더하는 방식
- BERT에서는 이와 유사하지만, 위치 정보를 sin함수와 cos함수로 만드는 것이 아닌 학습을 통해 얻은 **포지션 임베딩**이라는 방법을 사용

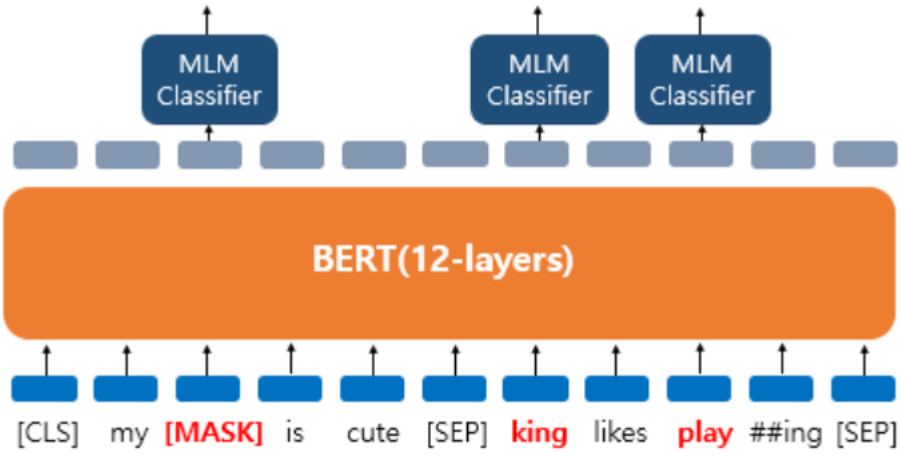


- 위 그림은 포지션 임베딩을 사용하는 방법을 보여줌
- wordPiece Embedding이란 평소에 알고 있는 단어 임베딩으로 실질적인 입력이다.
- 이 입력에 포지션 임베딩을 통해 위치 정보를 더해줌
- 따라서 위치 정보를 위한 임베딩 층을 하나 더 사용
    - 예를 들어, 문장의 길이가 4개라면 4개의 포지션 임베딩 벡터를 학습시킨다.

- BERT의 입력마다 포지션 임베딩 벡터를 더해준다.
  - 첫번째 단어의 임베딩 벡터 + 0번 포지션 임베딩 벡터
  - N번째 단어의 임베딩 벡터 + N-1번 포지션 임베딩 벡터

- 실제 BERT에서는 문장의 최대 길이를 512로 하고 있으므로, 총 512개의 포지션 임베딩 벡터가 학습됨
- 결론적으로 BERT에서는 총 두 개의 임베딩 층이 사용된다.

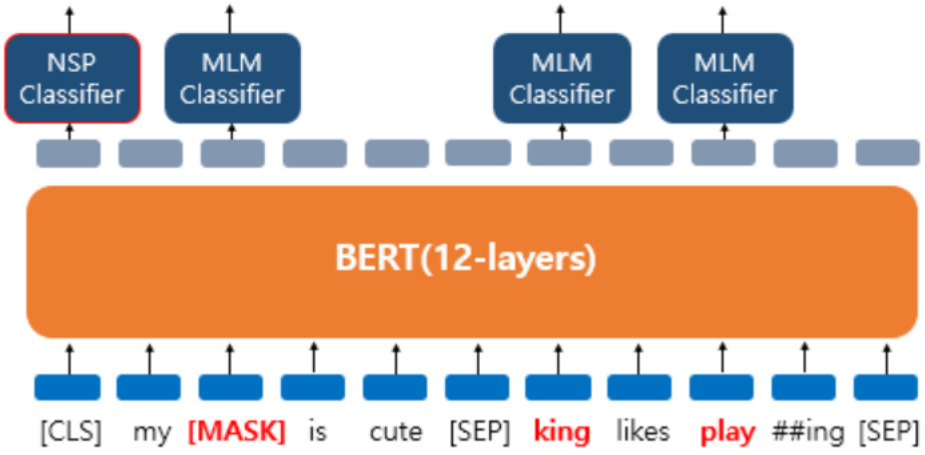
- BERT는 사전 학습에서 양방향 학습을 사용함
  - 이는 마스크드 언어 모델(MLM)과 다음 문장 예측(NSP)을 사용하기 때문
- **MLM**



  - BERT는 사전 훈련을 위해 인공 신경망의 입력으로 들어가는 입력 텍스트의 15%의 단어를 랜덤으로 Masking한다.
  - 그 후 인공 신경망에게 이 가려진 단어들을(Masked words) 예측하도록 한다.
    - 문장 중간에 구멍을 뚫어([Mask]토큰) 구멍에 들어갈 단어를 예측하게 하는 방식
  - 더 정확히는 전부 구멍([Mask])으로 변경하는 것이 아닌 랜덤으로 선택된 15%의 단어는 다음과 같은 비율로 규칙이 적용됨
    - 80%의 단어는 [Mask]로 변경됨
    - 10%의 단어는 랜덤으로 단어가 변경됨
    - 10%의 단어는 동일하게 둔다.

- 이렇게 하는 이유는 Mask만 사용할 경우 [Mask]토큰이 파인 튜닝 단계에서 나타나지 않으므로 사전 학습 단계와 파인 튜닝 단계에서의 불일치가 발생할 수 있다.
- 이를 완화하기 위해 모든 단어에 [Mask] 토큰을 사용하지 않는 것
- BERT는 단어가 변경된 경우 이 단어가 변경된 단어인지 원래 단어가 맞는지 알 수 없다.
  - 이 경우에도 원래 단어가 무엇인지 예측하도록 한다.
  - 따라서 BERT는 모든 단어에 MLM을 적용해 원래 단어를 예측한다.
- 위 그림은 단어가 변경되는 것의 예시이다.
  - BERT는 원래 단어가 무엇인지 출력층에 있는 다른 위치의 벡터들을 예측과 학습에 사용하지 않는다.
  - 즉, [Mask] 토큰 위치의 출력층 벡터만이 사용된다.
  - 구체적으로 BERT의 손실 함수에서 다른 위치에서의 예측은 무시된다.

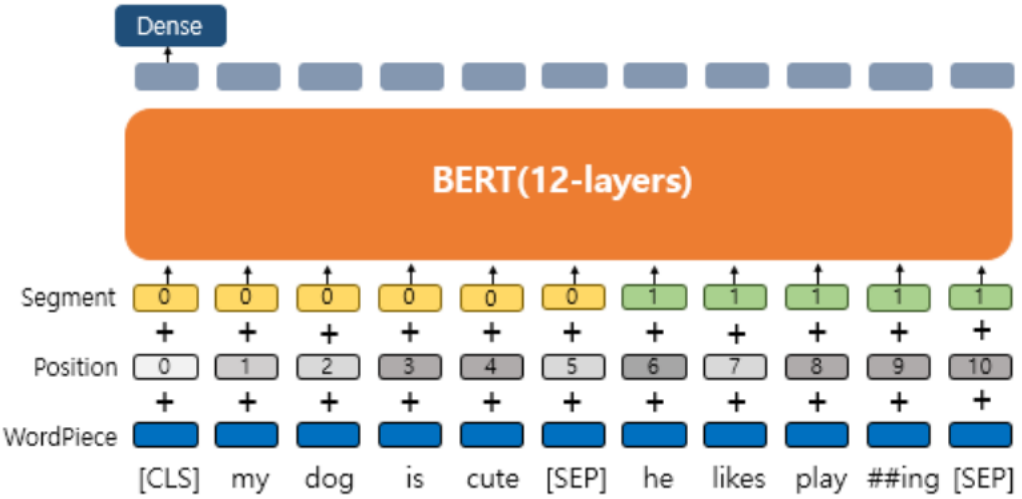
- **NSP**



- NSP를 사용하는 이유는 두 문장의 관계를 이해하는 것이 필요하기 때문
- 두 개의 문장을 주어 이 문장이 이어지는 문장인지 아닌지를 맞추는 방식 사용
- 이를 위해 50:50 비율로 실제 이어지는 두 개의 문장과 랜덤으로 이어지는 두 개의 문장을 주고 훈련시킨다.
  - 이어지는 문장의 경우 Label을 IsNextSentence를 준다.
  - 이어지는 문장이 아닌 경우 NotNextSentence를 준다.
- BERT의 입력으로 넣을 때 [SEP]라는 특별 토큰을 사용하여 문장을 구분

- 첫번째 문장의 끝에 [SEP] 토큰을 넣고, 두번째 문장이 끝나면 역시 [SEP] 토큰을 붙임
- 두 문장이 실제 이어지는 문장이 아닌지를 [CLS] 토큰의 위치의 출력층에서 이진 분류 문제를 풀도록 한다.
  - [CLS] 토큰은 BERT가 분류 문제를 풀기위해 추가된 특별 토큰이다.
- 위 그림에 나타난 것과 같이 MLM과 NLP는 따로 학습하는 것이 아닌 loss를 합하여 학습이 동시에 이루어진다.
- BERT가 NLP를 학습하는 이유는 BERT가 풀고자 하는 태스크 중 QA(주어진 질문에 대한 단락에서 답을 추출하는 것, 챗봇 같은 것)나 NLI(전제와 가설이 주어졌을 때 참 거짓 중립을 판별하는 작업)와 같이 두 문장의 관계를 이해하는 것이 중요한 태스크가 있기 때문이다.
  - NLI 예 – 전제: 여러 남성이 플레이하는 축구 게임, 가설: 어떤 남자들이 스포츠를 하고 있다. 결론: 참

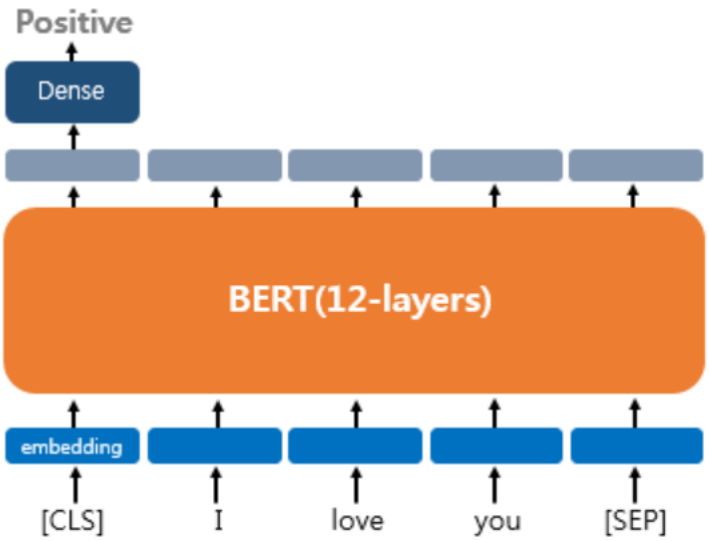
- **세그먼트 임베딩**



- BERT는 QA나 NLI와 같은 두 개의 문장 입력이 필요한 태스크를 풀기도 한다.
- 문장 구분을 위해 BERT는 세그먼트 임베딩이라는 또 다른 임베딩 층을 사용
  - 실제로는 두 개의 층이 아닌 세 개의 임베딩 층이 사용됨
  - 첫번째 문장에는 Setence 0 임베딩, 두번째 문장에는 Sentence 1 임베딩을 더해주는 방식
  - 임베딩 벡터는 두 개만 사용된다.

- BERT가 두 개의 문장을 입력받을 필요가 없는 경우
  - 네이버 영화 리뷰 분류와 같은 감성 분류 태스크에서는 한 개의 문서에 대해서만 분류를 진행
  - 이 경우 BERT의 전체 입력에 Sentence 0 임베딩만을 더해준다.

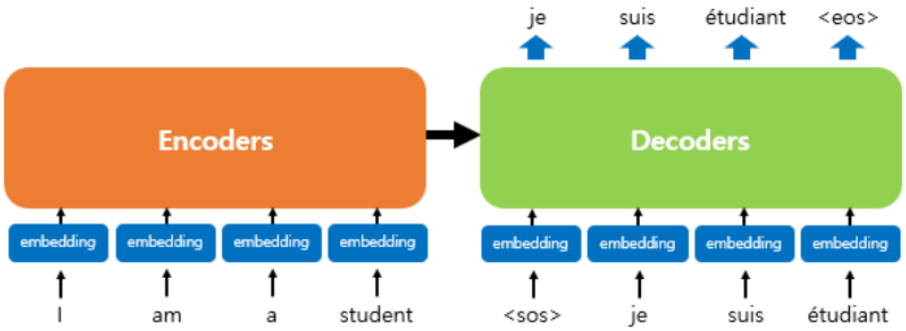
- **BERT를 파인 튜닝 하기**
  - 사전 학습된 BERT에 풀고자 하는 태스크의 데이터를 추가로 학습시켜서 테스트하는 단계
  - 실질적으로 BERT를 사용하는 단계에 해당
- 하나의 텍스트에 대한 텍스트 분류 유형 (Single Text Classification)



  - 이 유형은 리뷰 감성 분류 등과 같이 입력된 문서에 대해서 분류를 하는 유형
  - 문서의 시작에 [CLS]라는 토큰을 입력한다.
  - 사전 훈련 단계에서 다음 문장을 예측 설명할 때, [CLS] 토큰은 BERT가 분류 문제를 풀기 위한 특별 토큰이다.
    - 이는 파인 튜닝 과정에서도 동일함
  - 텍스트 분류 문제를 풀기 위해서 [CLS] 토큰의 위치의 출력층에서 Dense층을 추가하여 분류에 대한 예측을 하게된다.
- 이 외에 다른 파인 튜닝도 존재 -> QA, Text Pair Classification or Regression, Text Tagging

트랜스포머(Transformer)란?

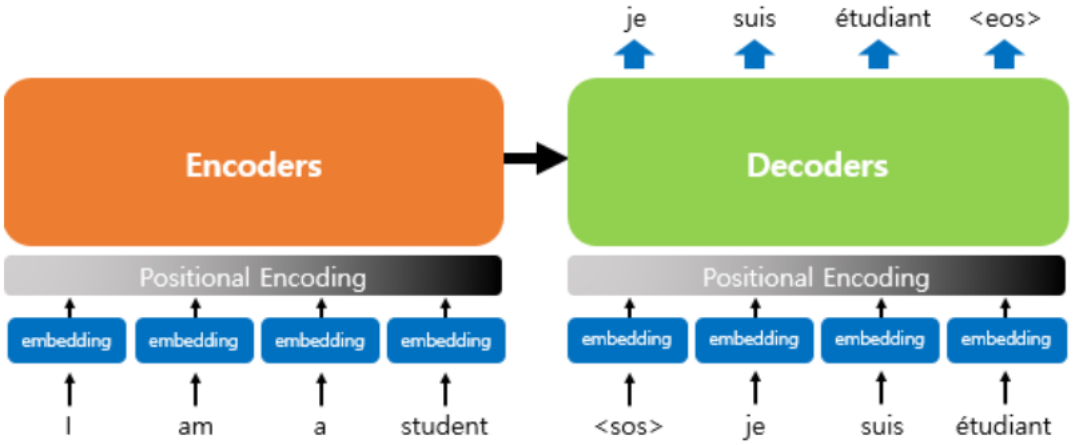
- 기존의 seq2seq의 구조인 인코더 디코더를 따르면서도, 어텐션(Attention)만으로 구현한 모델
- 이 모델은 RNN을 사용하지 않고, 인코더 디코더 구조를 설계하였음에도 RNN보다 우수한 성능을 보여준다
    - RNN이란 히든 노드가 방향을 가진 엣지로 연결되어 순환 구조를 이루는 인공신경망의 한 종류
    - 음성, 문자 등 순차적으로 등장하는 데이터 처리에 적합한 모델
    - RNN은 입력과 출력의 길이를 다르게 설계할 수 있어 다양한 용도로 사용할 수 있다.
- Seq2seq 구조에서 인코더가 입력 사퀀스를 하나의 벡터로 압축하는 과정에서 입력 시퀀스의 정보 일부가 손실된다는 단점이 있다.
    - 이를 보정하기 위해 어텐션이 사용됨
    - 이때 어텐션을 RNN의 보정을 위한 용도로 사용하는 것이 아닌 어텐션만으로 인코더와 디코더를 생성하는 것이 트랜스포머이다.
- 트랜스포머는 RNN을 사용하지 않지만 기존의 seq2se2처럼 인코더에서 입력 시퀀스를 입력받고, 디코더에서 출력 시퀀스를 출력하는 인코더-디코더 구조를 유지하고 있다.
- Seq2seq 구조는 인코더와 디코더에서 각각 하나의 RNN이 t개의 time step을 가지는 구조였다면, 트랜스포머는 인코더와 디코더라는 단위가 N개로 구성되는 구조이다.



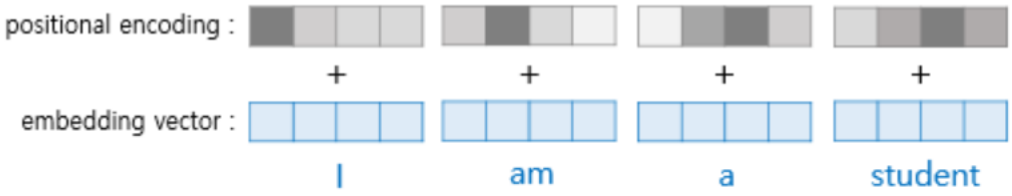
- 위 그림은 인코더로부터 정보를 전달받아 디코더가 출력 결과를 만들어내는 트랜스포머 구조이다.
    - Encoders와 Decoders인 이유는 인코더와 디코더가 6개 존재하는 구조이기 때문이다.
    - 인코더와 디코더가 6개 존재하는 이유는 논문에서 최적의 결과가 6개이기 때문

- 디코더는 기존의 seq2seq 구조처럼 시작 심볼 <sos>를 입력으로 받아 종료 심볼 <eos>가 나올 때까지 연산을 진행
- RNN이 사용되지 않아도 인코더-디코더 구조가 유지

- RNN이 자연어 처리에서 유용한 이유는 단어의 위치에 따라 단어를 순차적으로 입력 받아서 처리하는 RNN의 특성으로 인해 각 단어의 위치 정보를 가질 수 있다는 점에 있다.
  - 트랜스포머에서는 단어 입력을 순차적으로 받는 방식이 아니므로 단어의 위치 정보를 다른 방식으로 알려줄 필요가 있다.
  - 트랜스포머는 단어의 위치 정보를 얻기 위해서 각 단어의 임베딩 벡터에 위치 정보를 더하여 모델의 입력으로 사용 – 이를 포지셔널 인코딩이라고 한다.



- 위 그림은 입력으로 사용되는 임베딩 벡터들이 트랜스포머의 입력으로 사용되기 전 포지셔널 인코딩의 값이 더해지는 것을 보여줌
- 임베딩 벡터가 인코더의 입력으로 사용되기 전 포지셔널 인코딩 값이 더해지는 과정은 아래와 같다.

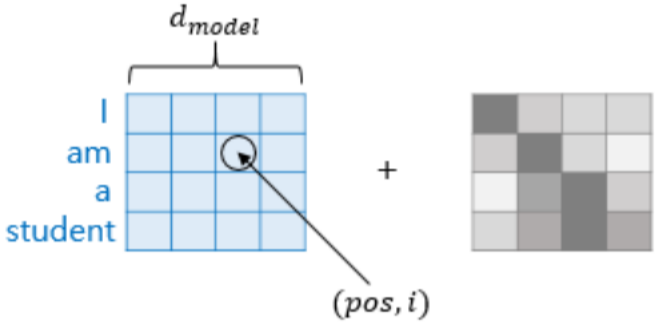


- 포지셔널 인코딩의 값은 아래의 함수를 이용하여 구한다. 따라서 위치 정보를 가진 값을 만들 수 있다.

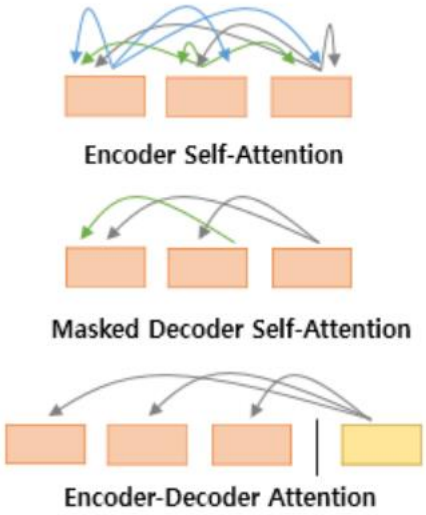
$$PE_{(pos,\ 2i)} = sin(pos/10000^{2i/d_{model}})$$

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

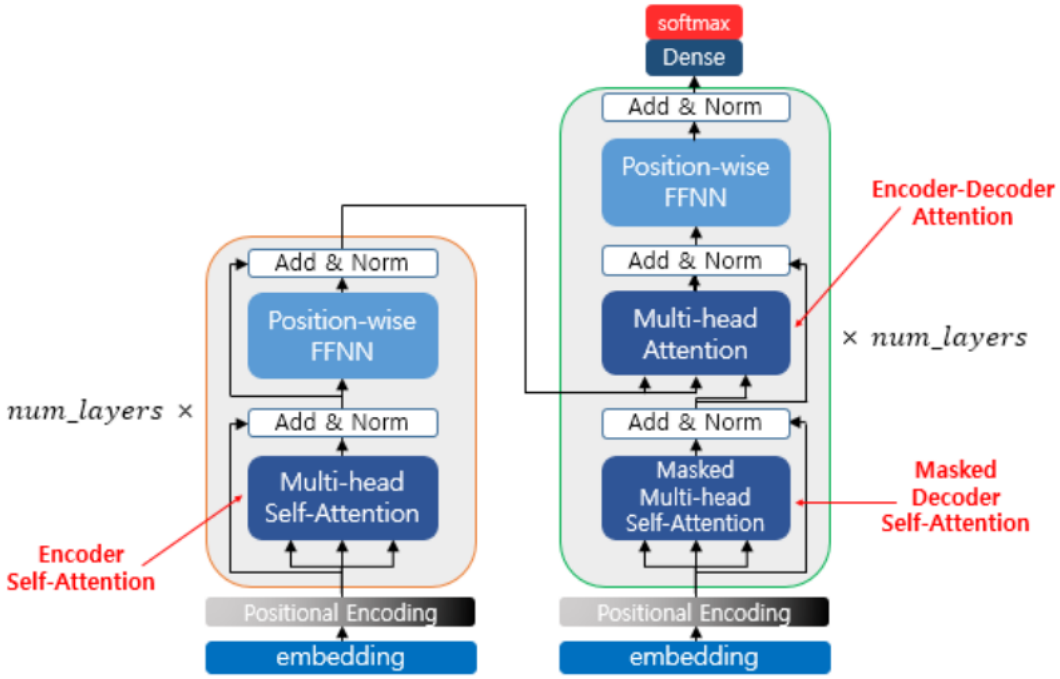
- Sin함수와 cos 함수의 그래프는 요동치는 값의 형태를 띈다(포물선 모양이 반복됨)
- 트랜스포머는 사인 함수와 코사인 함수의 값을 임베딩 벡터에 더해주어 단어의 순서 정보를 더해준다.
- 위의 두 함수는 pos, I, d와 같은 변수들이 존재
  - 임베딩 벡터와 포지셔널 인코딩의 덧셈은 임베딩 벡터가 모여 만들어진 문장 행렬과 포지셔너널 인코딩 행렬의 덧셈 연산을 통해 이루어짐을 기억해야함



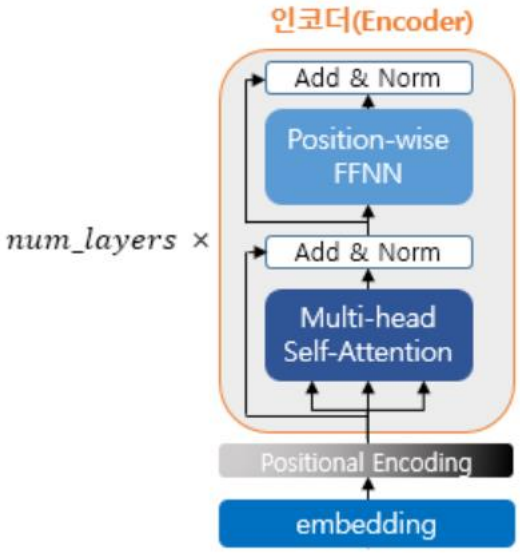
- Pos는 입력 문장에서의 임베딩 벡터의 위치를 나타낸다.
- I는 임베딩 벡터 내의 차원의 인덱스를 나타낸다.
- 임베팅 벡터 내의 각 차원의 인덱스가 짝수인 경우 sin을 사용, 홀수인 경우 cos를 사용
- d는 트랜스포머의 모든 층의 출력 차원을 의미하는 하이퍼파라미터이다. 트랜스포머의 각종 구조에서 d의 값이 계속 등장함
- 임베딩 벡터 또한 d의 차원을 가진다. 그림에서는 d는 4이지만 논문에서는 d의 값은 512를 가짐
- 포지셔널 인코딩 방법을 사용하면 순서 정보가 보존이 된다.
  - 예를 들어 각 임베딩 벡터에 포지셔널 인코딩 값을 더하면 같은 단어라고 하더라도 문장 내의 위치에 따라서 트랜스포머의 입력으로 들어가는 임베딩 벡터의 값이 달라짐
  - 따라서 트랜스포머의 입력은 순서 정보가 고려된 임베딩 벡터가 된다.



- 트랜스포머에서 사용되는 세 가지 어텐션을 나타내는 그림
  - 첫 번째 Self-Attention은 인코더에서 이루어진다.
  - 두 번째 Self-Attention과 Encoder-Decoder Attention은 디코더에서 이루어진다.
- Self-Attention은 Query, Key, Value가 동일한 경우를 말함
- Encoder-Decoder Attention은 Query가 디코더의 벡터인 반면 Key, Value는 인코더의 벡터이므로 Self-Attention이라고 부르지 않는다.
  - Query, Key, Value가 같다는 것은 벡터의 값이 같은 것이 아닌 벡터가 존재하는 공간이 다르다는 의미이다. (인코더에서 뽑은 값인지 디코더에서 뽑은 값인지)



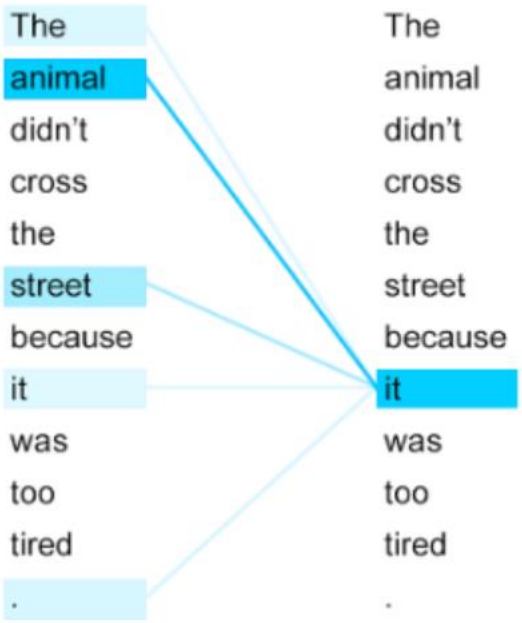
- 위 그림은 트랜스포머 구조에서 세 가지 어텐션의 위치를 보여줌
  - 멀티 헤드라는 이름이 붙어있는 이유는 어텐션을 병렬적으로 수행하기 때문



- 위 그림은 인코더의 구조이다.
- 하이퍼파라미터인 num_layers의 값에 따라 인코더의 층이 결정됨
  - 논문에서는 6개의 인코더 층을 사용

- 인코더를 하나의 층이라는 개념으로 생각한다면, 하나의 인코더 층은 크게 2개의 서브 layer로 구성된다.
    - Self-Attention과 피드 포워드 신경망(FFNN)으로 층이 나눠져 있음
    - 그림에서 멀티 헤드 셀프 어텐션이라고 되어 있는 이유는 셀프 어텐션을 병렬 적으로 사용했다는 의미
    - 포지션 와이즈 피드 포워드 신경망은 일반적인 피드 포워드 신경망이다.

- 어텐션이란 쿼리(Q)에 대해서 모든 키(K)와의 유사도를 각각 구하고, 이 유사도를 가중치로 하여 키와 매핑되어있는 각각의 값(V)에 반영하고, 유사도가 반영된 값을 모두 더한 후 반환하는 함수이다.
- 이러한 어텐션 중 self-Attention이 존재하며, 그 뜻은 어텐션을 자기 자신에게 수행한다는 의미
- Self-Attention에서 Q,K,V는 토큰화된 입력 문장으로부터 Q는 단어에 대한 가중치, K는 단어가 Q와 얼마나 연관되었는지 비교하는 가중치. V는 의미에 대한 가중치를 생성



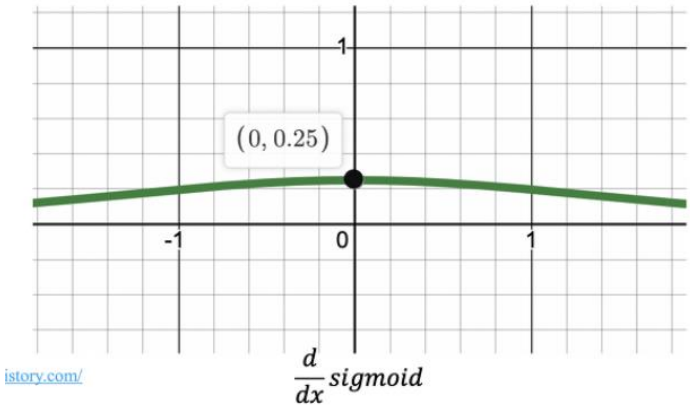
- Self-Attention을 사용하면 it에 해당하는 것이 동물에 연관되었다는 것을 알 수 있다.
    - It에 해당하는 것이 동물인지 길인지 기계는 알 수 없지만 샐프 어텐션을 사용하면 문장 내의 단어들끼리 유사도를 구하므로 it이 동물과 연관되었음을 알 수 있다.
- Self-Attention의 세부 과정은 생략 -> 이해하기 어려움

- 간단하게 말하면 Self-Attention을 사용하면 토큰화된 입력 문장으로부터 Q벡터와 K벡터를 내적하여 Score를 계산하고, K벡터의 사이즈에 루트를 취한 값을 나눠준 후, Softmax 함수를 통과시켜 확률 값을 얻는 과정이다. -> softmax를 통과한 점수는 현재 단어와의 연관성을 보여주는 수치

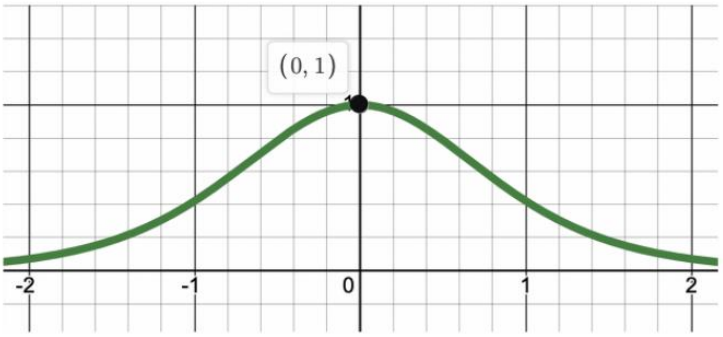
- 인코더에서 모든 층을 순차적으로 연산한 후 마지막 층의 출력을 디코더에 전달
- 인코더 연산이 끝나면 설정한 파라미터만큼(num_layers의 수) 디코더에서 연산을 진행
- 연산을 수행할 때 마다 인코더가 보낸 출력을 각 디코더 층 연산에서 사용
- 디코더는 BERT에서 사용하지 않음 -> 생략

어텐션(Attention)이란?

- Seq2seq 모델의 정확도가 떨어지는 문제를 보정해주는 기법
  - Seq2seq의 문제
    1. 하나의 고정된 크기의 벡터에 모든 정보를 압축하기 때문에 정보 손실 발생
    2. RNN의 고질적인 문제인 기울기 소실(Vanishing Gradient) 문제 발생
       A. 딥러닝 과정 중 layer가 많아질 수 록 학습이 잘 되지 않는 문제 발생
       B. 기울기 소실의 발생 원인은 활성화 함수(sigmoid, tanh)와 관련이 깊다

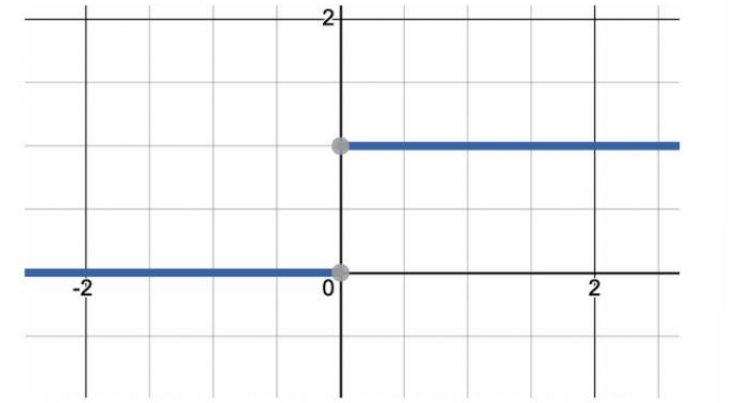


i. Sigmoid 함수로 학습 진행 시 sigmoid는 정확한 값이 아닌 근사 값으로 계산하고 층이 깊어질 수 록 기울기가 낮아지는 문제가 있다.



ech.tistory.com/ $\frac{d}{dx}tanh$

ii. Tanh 함수 역시 x 값이 크거나 작아짐에 따라 기울기 크기가 작아짐

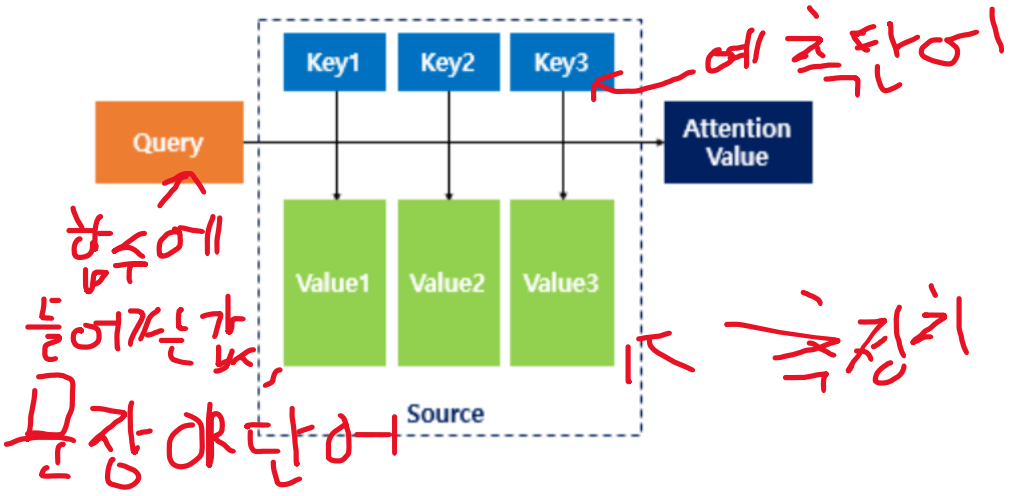


heytech.tistory.com/ $\frac{d}{dx}ReLU$

C. 기울기 소실 문제는 ReLU로 해결 가능

i. ReLU의 음수가 될 경우 생기는 문제는 Leaky ReLU로 해결 가능

- 어텐션은 디코더에서 출력 단어를 예측하는 매 시점(다음 셀로 넘어가는 시점)마다 인코더에서의 전체 입력 문장을 다시 한 번 참고한다
  - ■ 단, 전체 문장을 동일한 비율로 참고하는 것이 아닌, 해당 시점에서 예측하는 단어와 연관이 있는 입력 단어 부분을 조금 더 집중(attention)해서 보게 된다.
- 어텐션 함수의 표현 식
  - ■ Attention(Q, K, V) = Attention Value

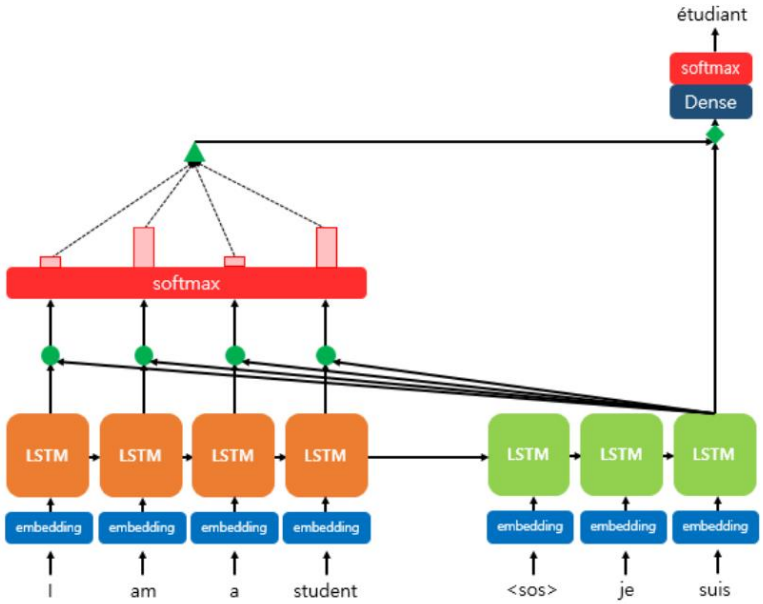


- 어텐션 함수는 주어진 쿼리(Q)에 대해서 모든 키(K)와의 유사도를 각각 구한다.
- 구해낸 유사도를 키와 메핑되어있는 각각의 값(V)에 반영해준다.
- 유사도가 반영된 값을 모두 더해서 Attention Value를 구한다.
- K, V는 Key-Value 자료형이라고 생각(딕셔너리, map 등)

```
Q = Query : t 시점의 디코더 셀에서의 은닉 상태
K = Keys : 모든 시점의 인코더 셀의 은닉 상태들
V = Values : 모든 시점의 인코더 셀의 은닉 상태들
```

- t 시점이라는 것은 함수에 주어지는 쿼리가 반복해서 불린다는 의미
- 어텐션은 디코더에서 출력 단어를 예측하는 매 시점마다 인코더에서의 전체 입력 문장을 다시 한번 참고한다. 따라서 매 시점의 인코더의 셀의 은닉 상태를 알아야함

- **Dot-Product Attention** (수식적으로 이해하기 쉬운 어텐션)

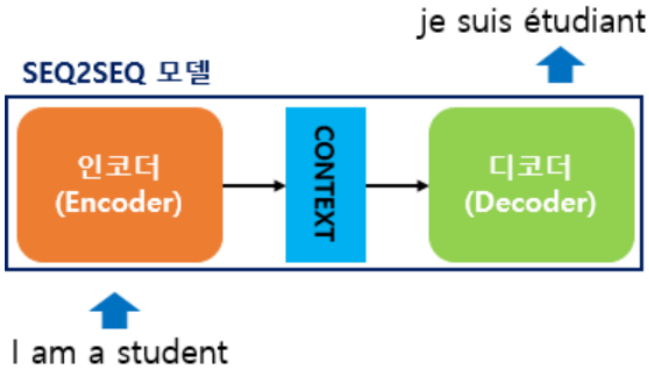


  - 세번째 LSTM 셀(현재 그림에는 3번째 셀 없음)에서 출력 단어를 예측할 때 어텐션 메커니즘을 사용
  - 디코더의 1, 2번째 셀은 이미 어텐션 메커니즘을 통해 je와 suis를 예측
  - 세번째 셀은 출력 단어를 예측하기 위해 인코더의 모든 입력 단어들의 정보를 참조
  - 인코더의 softmax 함수를 통해 나온 결과 값은 인코더의 input으로 사용하는 각 단어가 출력 단어(je, suis 등)를 예측할 때 얼마나 도움이 되는지 수치화한 값
    - 그림에서는 막대 그래프로 표현, 그래프가 클수록 도움이 되는 단어
    - 디코더에 수치화한 값을 하나의 정보로 담아(초록 색 삼각형) 디코더로 전송
  - 이를 통해 디코더는 출력 단어를 정확하게 예측할 수 있다.

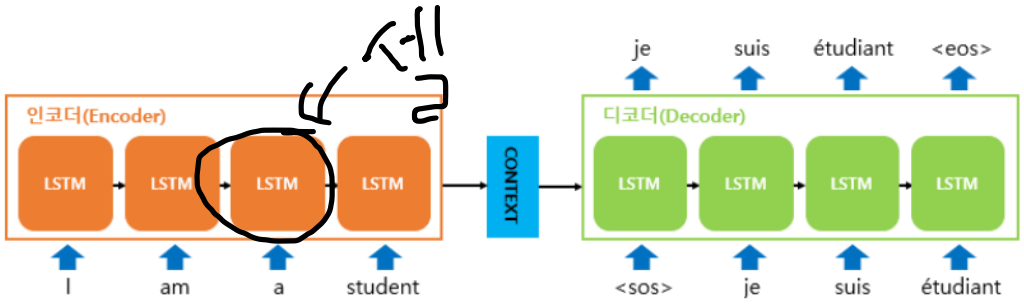
시퀀스-투-시퀀스(seq2seq)란?

- Seq2seq는 입력된 시퀀스로부터 다른 도메인의 시퀀스를 출력하는 다양한 분야(챗봇, 기계 번역 등)에서 사용되는 모델
- 입력 시퀀스와 출력 시퀀스를 각각 질문과 대답으로 구성하면 챗봇으로 만들 수 있다
- 입력 시퀀스와 출력 시퀀스를 각각 입력 문장과 번역 문장으로 만들면 번역기로 만들 수 있다.

■ 시퀀스 : 하나의 문장



- Seq2seq는 크게 인코더와 디코더라는 두 개의 모듈로 구성된다.
- 인코더는 입력 문장의 모든 단어들을 순차적으로 입력 받는다.
- 마지막에 모든 단어 정보들을 압축해서 하나의 벡터로 만든다.
  - 이를 컨텍스트 벡터라고 한다.
- 입력 문장의 정보가 하나의 컨텍스트 벡터로 모두 압축되면 인코더는 컨텍스트 벡터를 디코더로 전송한다.
- 디코더는 컨텍스트 벡터를 받아서 번역된 단어를 하나씩 순차적으로 출력한다.



- 인코더와 디코더의 내부 구조는 RNN으로 구성된다.
- 실제로는 성능 문제로 인해 LSTM이나 GRU(게이트 순환 유닛)를 이용한다.
- 입력 문장은 단어 토큰화를 통해 단어 단위로 쪼개진다.
- 단어 토큰은 각각 인코더 셀에 입력된다.
- 인코더 셀은 모든 단어를 입력받은 뒤 인코더 셀의 마지막 시점의 은닉 상태를 디코더의 셀로 넘겨준다. 이를 컨텍스트 벡터라고 한다.

- 컨텍스트 벡터는 디코더의 첫번째 은닉 상태에 사용된다.
- 디코더는 초기 입력으로 문장의 시작을 의미하는 심볼인 <sos>가 들어간다.
- <sos>가 입력되면, 다음에 등장할 확률이 높은 단어를 예측한다.
- 첫번째 시점의 디코더 셀이 예측한 단어는 다음 시점의 디코더 셀의 입력으로 사용
    - 즉, 앞애 있는 신경망이 예측한 단어를 다음 신경망의 입력으로 사용해 예측
- 이 과정은 문장의 끝을 의미하는 심볼인 <eos>가 예측될 때까지 반복