Transformer

Attention Is All You Need (어텐션이면 됩니다)

- Attention만 잘 활용해도 다양한 자연어 처리 task에서 높은 성능을 얻을 수 있다는 것을 의미
- Transformer에서 Attention이 핵심 아이디어로 사용

Transformer

- 딥러닝 기반 자연어 처리 기술에서 핵심 아키텍처
- RNN이나 CNN을 전혀 필요로 하지 않으며, Attention 기법만 사용
- 2021년 기준으로 최신 고성능 모델들은 Transformer 아키텍처를 기반으로 함
  - GPT: Transformer의 Decoder 아키텍처를 활용
  - BERT: Transformer의 Encoder 아키텍처를 활용

기계번역 기술의 발전 과정

- RNN (1986)
- LSTM (1997)
  - 주가예측, 주기함수 예측 등 다양한 Sequence 정보를 모델링할 수 있음
- Seq2Seq (2014)
  - 현대의 딥러닝 기술들이 다시 빠르게 나오기 시작한 시점에 LSTM을 활용하여 고정된 크기의 Context Vector를 사용하는방식으로 번역을 수행하는 방법을 제안
  - Source 문장을 전부 고정된 크기의 한 Vector에 압축할 필요가 없다는 점에서 성능적인 한계가 존재
- Attention (2015)
  - Seq2Seq 모델에 Attention 기법을 적용하여 성능 향상

- Transformer (2017)
  - RNN, CNN을 사용할 필요가 없고 오직 Attention 기법에만 의존하는 아키텍처를 설계하여 성능이 훨씬 좋아짐
  - Transformer를 기점으로 하여 더 이상 다양한 자연어처리 Task에 대해 RNN 기반의 아키텍처를 사용하지 않고, Attention 메커니즘을 더욱 더 많이 사용하게 됨
- GPT-1 (2018)
- BERT (2019)

→ 입력 시퀀스 전체에서 정보를 추출하는 방향으로 발전

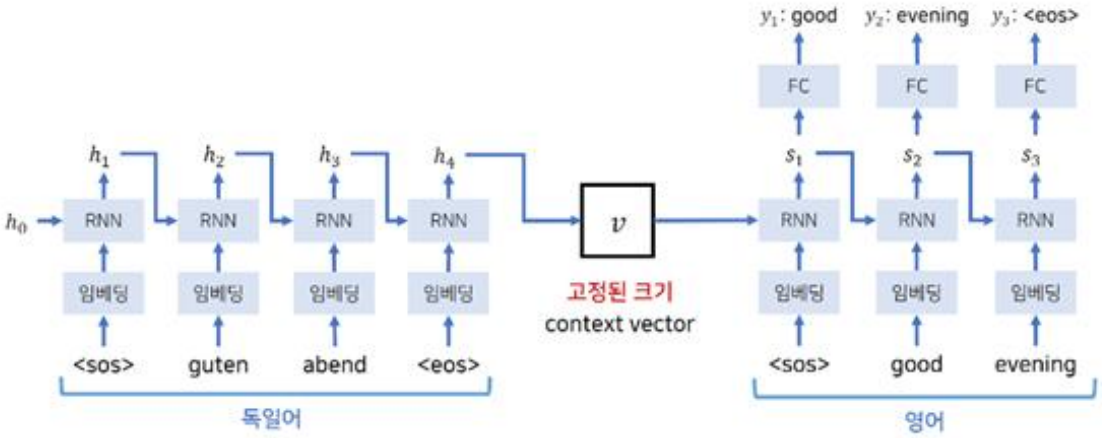
→ 전반적인 추세는 Attention 기법을 활용하는 Transformer의 아키텍처를 따르는 방식으로 진행되며 다양한 고성능 모델을 제안

기존의 Seq2Seq 모델들의 한계점

- Context Vector v에 소스 문장의 정보를 압축
  - 이는 병목현상을 야기하며, 성능 하락의 원인

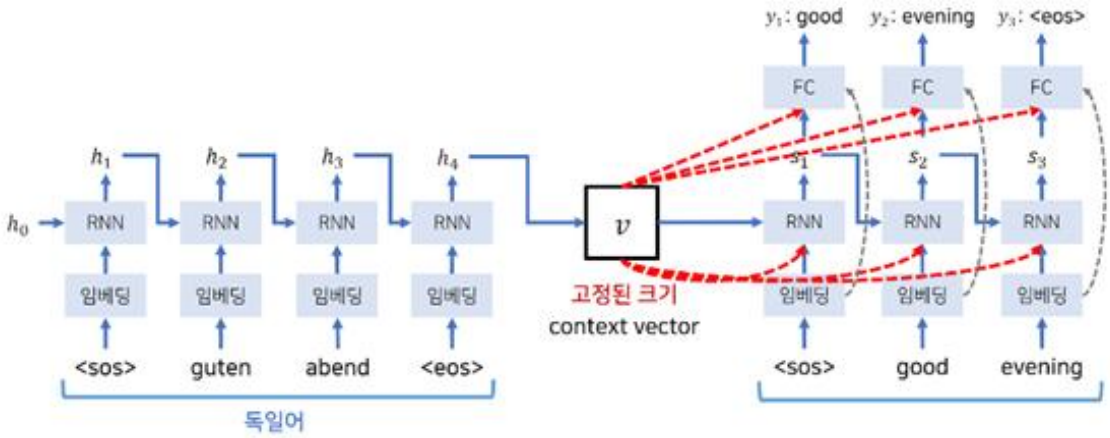
  → 마지막 단어가 들어왔을 때의 hidden state 값이 Souce 문장을 대표하는 하나의 Context Vector로 사용될 수 있다.

  → Source 문장을 대표하는 하나의 Context Vector를 만들어야 한다는 점에서 고정된 크기의 Context Vector에 모든 정보를 압축하려고 하면, 다양한 경우에 대해 Source 문장의 정보를 고정된 크기로 가지고 있는 것은 전체 성능에서 병목현상의 원인이 될 수 있다.



→ Decoder 부분에서 매번 출력단어가 들어올 때마다 이러한 Context Vector로 출발해서 hidden state를 만든 뒤에 출력을 만든다. <eos>가 나올때까지 반복하는 구조로 설계

→ 이러한 방법은 Context가 한번만 참고된다는 단점이 존재



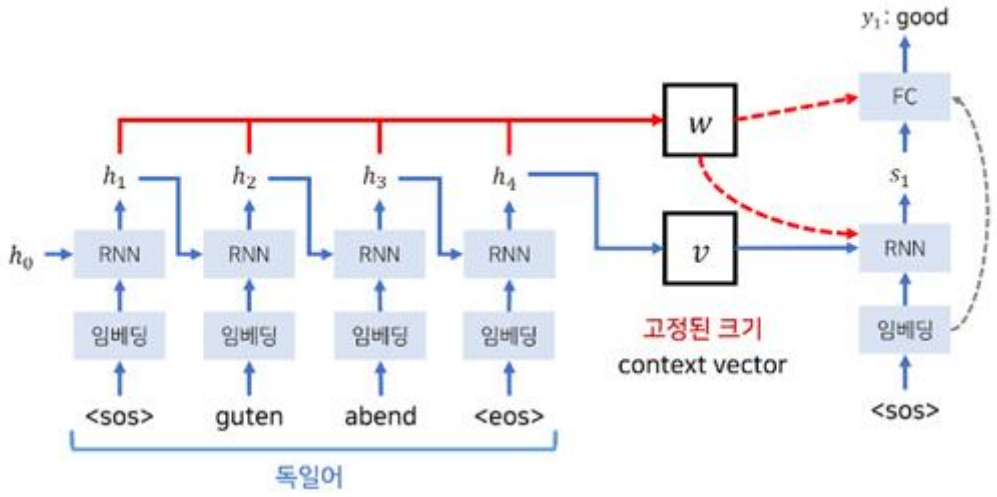
→ 매번 Decoder의 RNN Cell에서 Context Vector를 매번 참고하도록 설계하면 출력되는 문장이 길어진다고 하더라고 각각의 출력되는 단어에 Context Vector에 대한 정보를 다시 한번 넣어주어 정보가 손실되는 정도를 줄이며 기존보다 성능이 향상될 수 있다.

→ 하지만 이 방법 또한 Source 문장을 하나의 Context Vector로 압축해야 하기 때문에 병목현상이 발생한다.

Seq2Seq with Attention

- 하나의 Context Vector가 Source 문장의 모든 정보를 가지고 있어야 하기 때문에 성능이 저하되는 문제가 발생하는데, 이를 하나의 Context Vector에 담지 말고 Source 문자에서 나왔던 출력값을 매번 입력으로 받아서 일련의 처리과정을 거쳐 출력단어를 만드는 아이디어에서 Attention 모델이 제안

→ Source 문장에서 출력되었던 모든 hidden state 값을 전부 반영하여 Source 문장의 단어들 중 어떠한 단어에 주의 집중 하여 출력 단어를 만들까?를 모델이 고려하도록 만들어서 성능을 높이는 방식



- Seq2Seq 모델에 Attention 매커니즘을 사용

→ 출력 값들을 별도의 배열에 기록하며, 각각의 단어를 거치며 갱신되는 hidden state 값들을 매번 가지고 있으며, 출력 값들을 전부 고려한 Weighted Sum Vector w에 저장

→ Decoder는 Encoder의 모든 출력을 참고할 수 있다.

→ 출력 단어가 생성될 때마다 Source 문장 전체를 반영

Decoder에서는 매번 Encoder의 모든 출력 중에서 어떤 정보가 중요한지를 계산

- Energy

  Decoder가 이전에 출력했던 단어를 만들기 위해 사용했던 hidden state와, Encoder 부분의 각각의 hidden state를 비교해서 Energy를 계산

  이는 Encoder 부분의 어떠한 hidden state와 가장 많은 연관성을 가지는지를 구하는 과정

  즉, Source 문장에서 나왔던 모든 출력값들 중 어떠한 값과 가장 연관성이 있는 지를 수치화

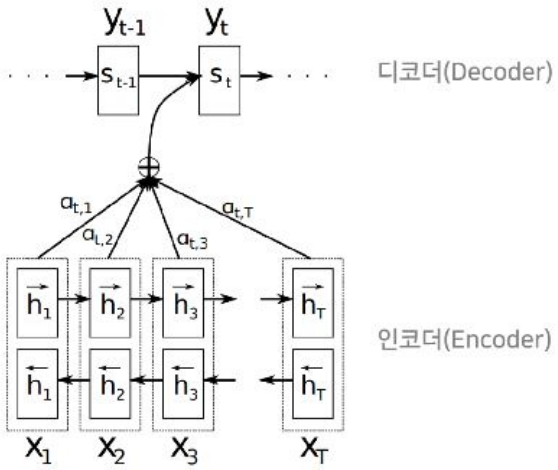
- Weight

  에너지 값들을 softmax에 넣어서 상대적인 확률 값을 계산

- 에너지(Energy) $e_{ij} = a(s_{i-1}, h_j)$
- 가중치(Weight) $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$

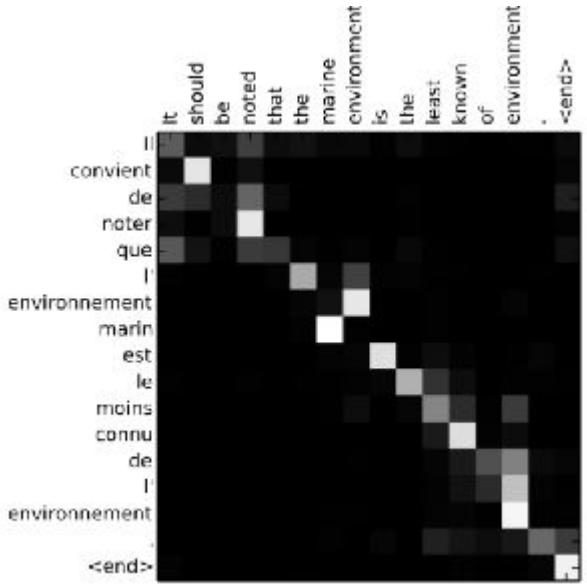
Weighted sum 이용

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$



- 가중치 값들을 시레로 각각의 hidden state와 곱해서 전부 더해준 값을 실제로 Decoder의 입력으로 넣어주기 위해 Weighted Sum을 계산
- t번째 단어를 출력하기 위해서 Encoder 부분에서 각각의 모든 hidden state 값을 같이 묶어서 Energy 값을 계산하고 softmax로 구한 가중치 값과, 이전에 사용했던 hidden state St-1을 이용
- 가중치는 각각의 단어들 중에서 어떠한 값을 가장 많이 참고하면 되는지 비율 값을 percentage로 구해서 비율만큼 hidden state를 구한 값을 Context Vector처럼 사용
- 매번 출력할 때마다 Source 문장에서 출력되었던 모든 hidden state 값을 전부 반영해서 다음에 무엇을 출력할 지 만들 수 있음

Attention 가중치 시각화

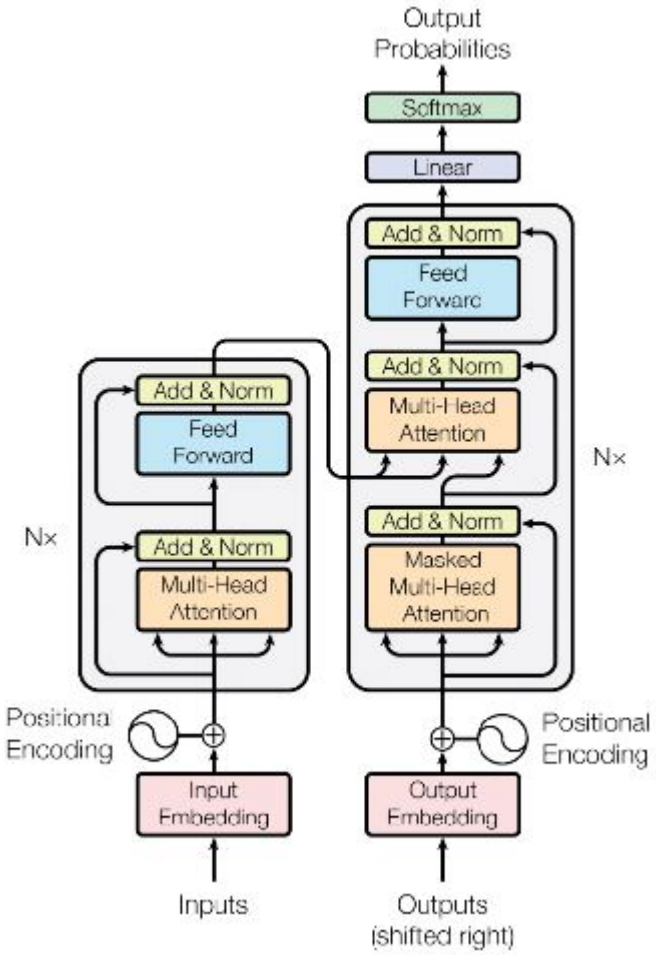


- 해당 방식은 영어를 불어로 번역하는 과정에서 Attention 가중치를 시각화한 것
  - 기본적으로 딥러닝은 많은 파라미터를 보유하고 있기 대문에 세부적인 파라미터를 하나씩 분석하면서 어떤 원리로 동작했는지 파악하는 것은 어려움
  - Attention 매커니즘의 경우 딥러닝이 어떤 요소에 더욱 더 많은 초점을 두어서 분류/생성을 했는지 분석하는데 용이하도록 시각화가 가능
- 매번 불어의 단어를 출력할 때마다 이러한 입력 단어들 중에서 어떤 단어에 가장 많은 단어에 초점을 두었는지 파악하는 것이 가능
  - 밝은 부분이 확률값이 높은 부분을 의미하며, 해당 단어에 많은 초점을 두고 번역했음을 의미
  - 즉, 입력 단어 중에 어떠한 단어에 더욱 많은 가중치를 두어 Attention을 수행했는지 밝기를 통해 알 수 있음

Transformer

- RNN이나 CNN을 사용하지 않기 때문에 각각의 단어들의 순서에 대한 정보를 주는 것이 어려움

  → Positional Encoding을 사용하여 문장 내 각각의 단어들에 대한 순서 정보를 알려줌
- Encoder와 Decoder로 구성되며, Attention 과정을 여러 Layer에서 반복

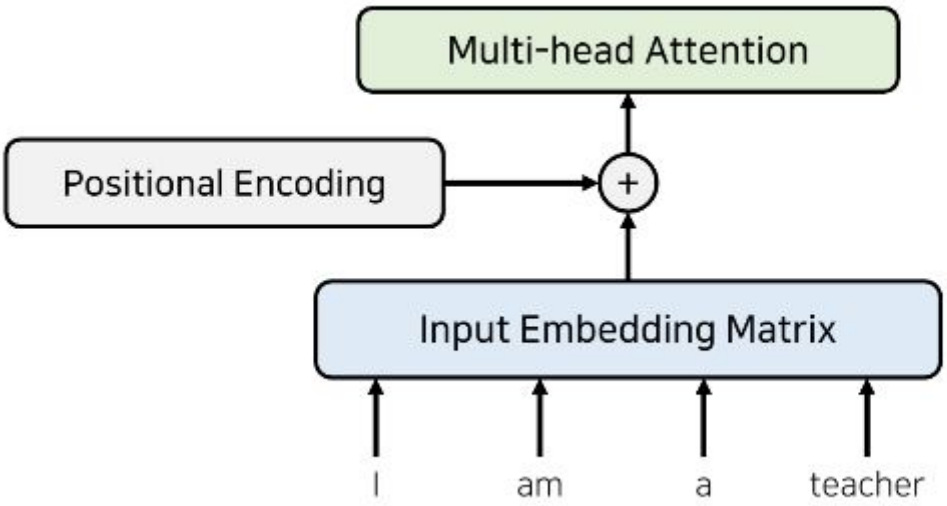


Embedding

- 어떠한 단어정보를 네트워크에 넣기 위해서는 일반적으로 임베딩 과정을 거쳐야 한다.
  - 입력차원 = 특정 언어에서 존재할 수 있는 단어의 갯수
- Seq2Seq와 같은 RNN 기반의 아키텍처를 사용한다고 하면, RNN을 사용하는 것만으로도 각각의 단어가 RNN에 들어갈 때, 순서에 맞게 들어가기 때문에 자동으로 각각의 hidden state값은 순서에 대한 정보를 가진다.

  → 하지만 RNN 계열을 사용하지 않는다면, 별도의 위치정보를 포함하고 있는 임베딩을 사용해야 한다. Positional Encoding

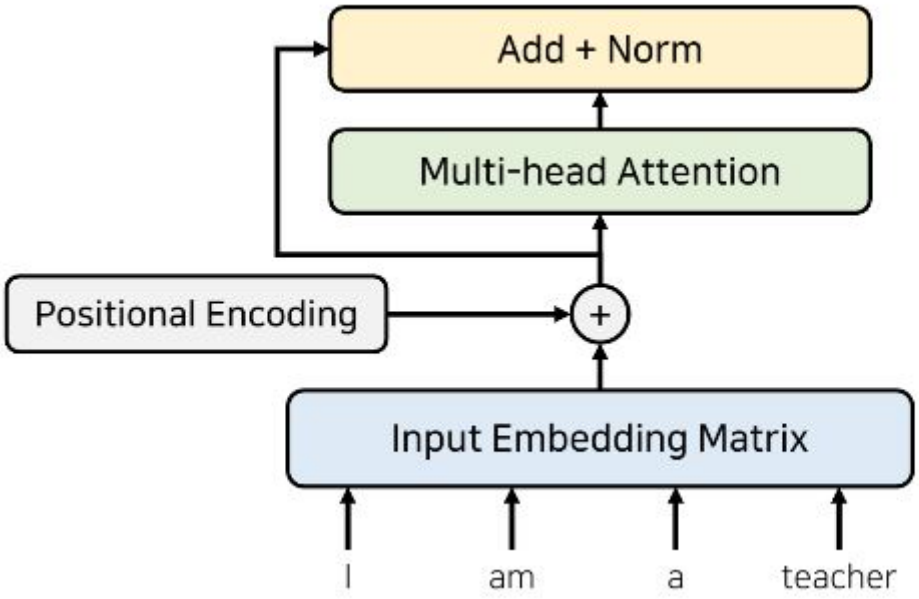
Positional Encoding



- Input Embedding과 같은 크기, 차원을 가짐
- 별도의 위치에 대한 정보를 가지고 있는 Embedding을 넣어주어 각각 elementwise로 더해주어 각각의 단어가 어떤 순서를 가지는 지에 대한 정보를 네트워크가 알 수 있도록 만드는 역할
- 보통은 sin, cos 함수 등 주기함수를 이용하여 하나의 문장에 포함된 각각의 단어들에 대한 상대적인 위치에 대한 정보를 구하지만 주기성을 학습할 수 있도록 하기만 한다면 주기함수가 아닌 다른 함수를 이용해도 무관

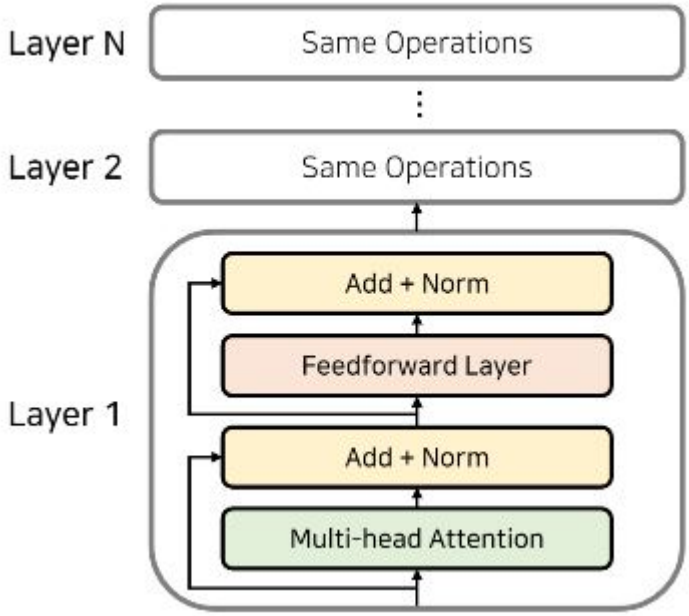
  → Transformer 이후에 주기함수가 아닌 학습이 가능한 형태로 별도의 임베딩 레이어를 사용

Encoder



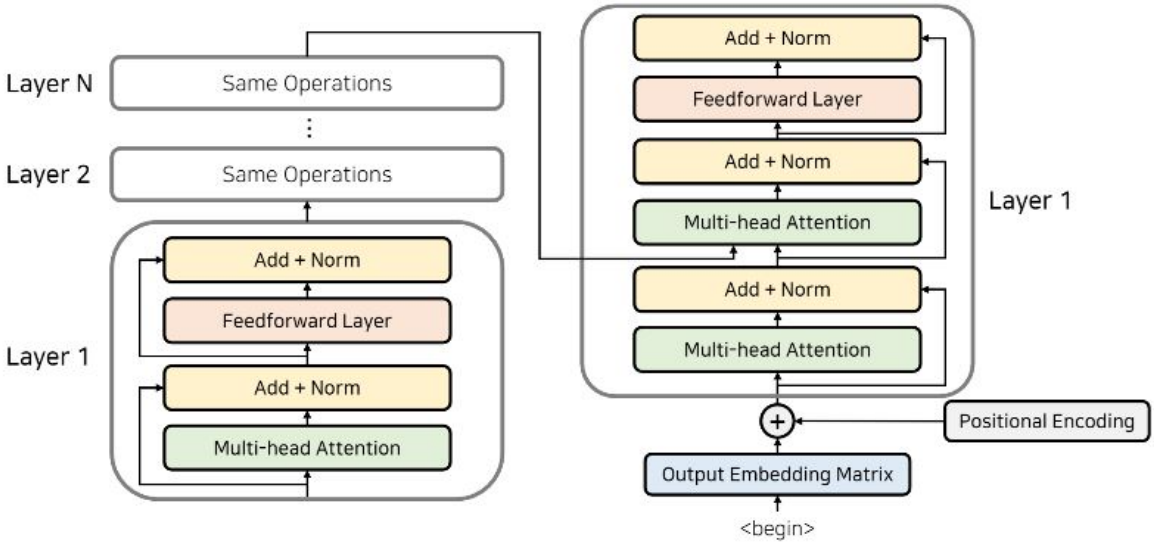
- 임베딩이 끝난 후에 Attention을 진행
  - 입력 문장에 대한 정보 + 실제 위치에 대한 정보를 input으로 받음
  - 실제 위치에 대한 정보가지 포함하고 있는 입력값을 실제 Attention에 넣어줄 수 있도록 함
  - 이러한 입력을 받아서 각각의 단어들을 이용해 Attention을 수행
  - 첫번째 Attention: Self-Attention = 각각 서로에게 Attention Score를 구해서 각각의 단어는 어떠한 단어와 높은 연관성을 가지는 지에 대한 정보를 학습할 수 있도록 만드는 역할
  - Attention은 전반적인 입력문장에 대한 문맥에 대한 정보를 잘 학습하도록 만드는 것
- 성능 향상을 위해 잔여학습을 진행
  - 잔여학습이란 대표적인 이미지 분류 네트워크 ResNet과 같은 네트워크에서 사용되고 있는 기법
  - 어떠한 값을 레이어를 거쳐서 반복적으로 단순하게 갱신하는 것이 아니라 특정 레이어를 건너 뛰어서 복사가 된 값을 그대로 넣어주는 것을 의미
  - 잔차 네트워크는 기존 정보를 입력 받으면서 추가적으로 잔여된 부분만 학습되도록 만들기 때문에 전반적인 학습 난이도가 낮고 초기의 모델 수렴속도가 높아지게 된다.
  - global optimal을 찾을 확률이 높아지기 때문에 전반적으로 다양한 네트워크에서 Residual Learning을 사용했을 때 성능이 좋아지므로 Transformer에서 채택

- Attention을 수행한 결과값과 Residual Connection을 이용해서 바로 더해진 값을 같이 받아서 Noramlization까지 수행한 뒤에 결과를 내보낼 수 있도록 만드는 것이 Encoder의 동작과정



- Attention과 Noramlization 과정을 여러 개의 Layer에서 중첩하여 사용
- 각 레이어는 서로 다른 파라미터를 가지게 됨
- 입력되는 값과 출력되는 값의 dimension이 동일하기 때문에 레이어를 중첩하여 사용

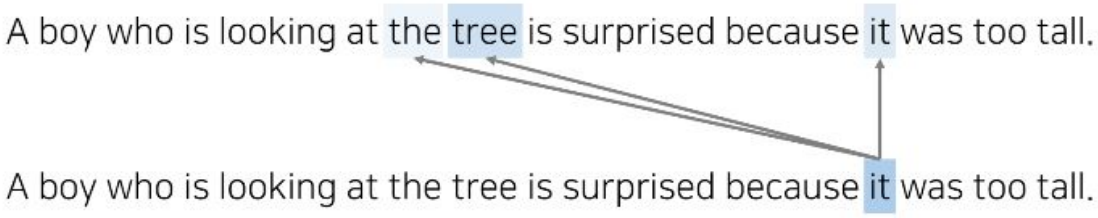
Decoder



- 기본적인 Attention Layer의 동작 방식은 동일

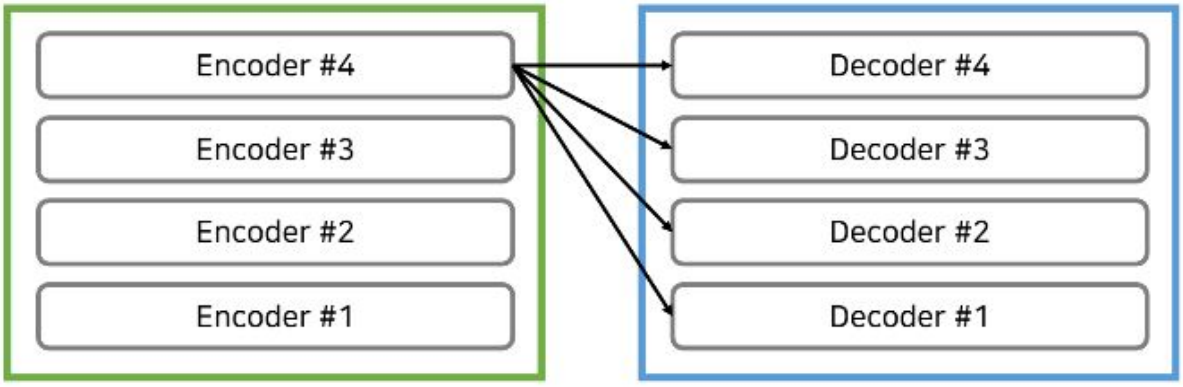
- 가장 마지막의 Encoder에서 나오게 된 출력 값이 Decoder로 들어가게 됨
    - 각각 단어정보를 받아서 각 단어의 상대적인 위치에 대한 정보를 알려주기 위하여 Encoding 값을 추가한 뒤에 입력을 받게 된다.
- 결국 Decoder 부분에서는 매번 출력 시, 입력 Source 문장 중 어떤 단어에게 가장 많은 초점을 주는지를 고려하여 출력
- 하나의 Decoder Layer에서는 두개의 Attention을 사용

  1) Self - Attetion: Encoder 부분과 마찬가지로 각각의 단어들이 서로가 서로에게 어떠한 가중치를 가지는지 구하도록 만들어서 출력되고 있는 문장에 대한 전반적인 표현 학습



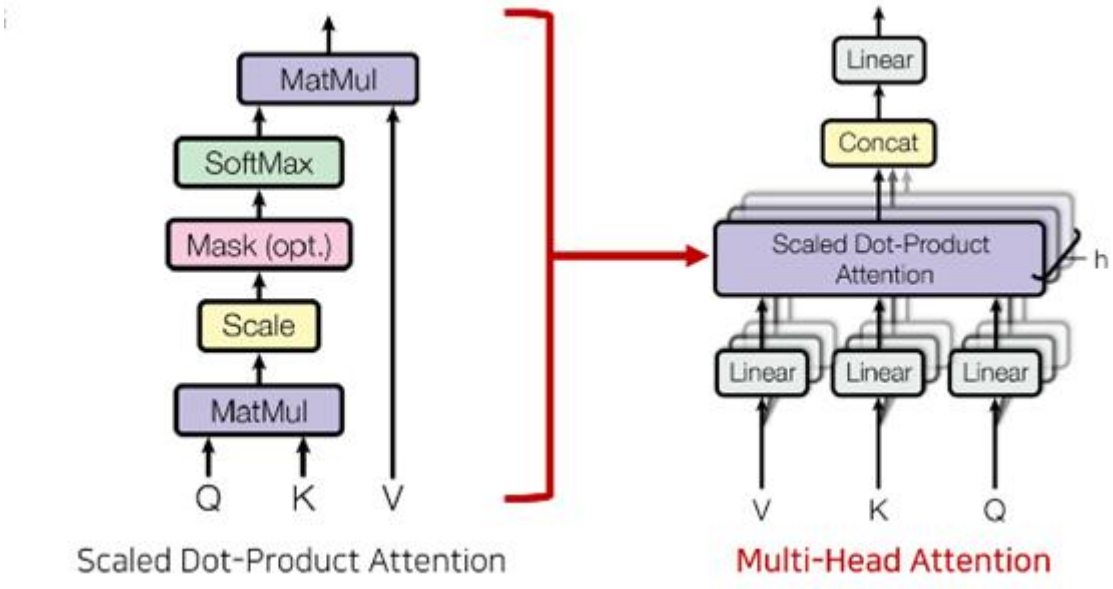
    - 해당 경우에는 대명사 it이 tree를 가리키기 때문에 tree와 유사함을 알 수 있음

  2) Encoder - Decoder Attention: Encoder에 대한 정보를 Attention 할 수 있도록 만드는 역할을 하는데 각각의 출력되고 있는 단어가 Source 문장에서의 어떠한 단어와 연관성이 있는 지를 구하는 과정이며, Encoder 부분에서 나왔던 출력결과를 전적으로 활용하도록 네트워크를 설계
- 사용되는 위치마다 Query, Key, Value를 어떻게 사용할지 달라질 수 있음
    - 예를 들어 Encoder-Decoder Attention의 경우에는 Decoder 부분의 단어가 Query, Encoder 파트의 각각의 값들이 : Key, Value로 이용



- 마지막 Encoder Layer의 출력 값이 모든 Decoder Layer의 입력으로 들어감
- Decoder 부분의 두번째 Attention에서 각각의 출력 단어가 입력 단어 중 어떠한 정보와 높은 연관성을 가지는 지를 계산하도록 만드는 부분에서 사용

Multi-Head Attention



- Encoder와 Decoder는 Multi-Head Attention Layer를 사용
- Attention을 위한 3가지 입력 요소
  - Query (물어보는 주체), Key (물어보는 대상), Value
- 예를 들어 I am a student 라는 문장이 있을 때, I가 각각의 단어에 대해 어떠한 연관성이 알고 싶다면 Query = I, Key = I, am, a, student 각각의 단어
- 입력으로 들어온 값은 3개로 복제되어, Key, Value, Query로 들어감
- 이러한 Value, Key, Query 값들은 Linear Layer(행렬곱 수행)를 통과하여 각각의 Query쌍을 만들어냄
- 서로 다른 head끼리 Value, Key, Query의 쌍을 받아서 Attention을 수행하고 concat
  - 입력 값과 출력 값의 차원을 동일하게 하기 위해 각각의 head로 부터 나오게 된 Attention 값들을 concat
  - h개의 서로 다른 Attention Concept을 학습하도록 만들어 더욱더 구분된 다양한 특징을 학습할 수 있도록 유도
- 최종적으로 Linear Layer를 거쳐 출력값을 내보냄

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

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

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

- 각 Query에 대하여 각각의 Key에 대한 Energy 값을 계산한 뒤에 확률값으로 계산하여 실제로 어떠한 Key에 대해 높은 가중치를 가지는지를 계산할 수 있음

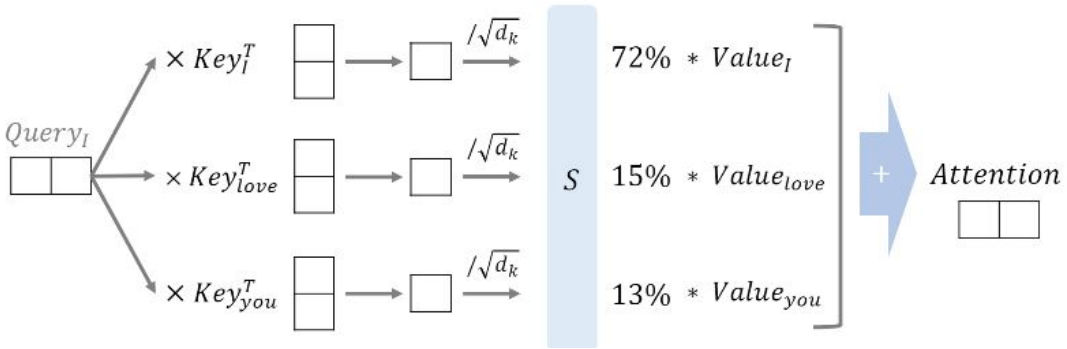
  → Score*Value = Attenion Value

- Score에 softmax 전에 scale을 진행한 이유는 softmax의 특성 상 값이 너무 커질 경우 gradient vanishing 문제가 발생할 수 있기 때문에 Key의 차원의 제곱근 값으로 scaling을 진행

- 입력으로 들어오는 각각의 값에 대해서 서로 다른 Linear Layer를 거치도록 만들어서 h개의 서로 다른 Query, Key, Value를 만든다.

  → 이는 Attention을 수행하기 위한 다양한 feature들을 학습 하도록 만든다.

- 입력으로 들어왔던 값과 동일한 차원의 출력을 위해 concat을 진행한 뒤에 output matrix와 행렬곱을 수행하여 최종적인 Multi-Head Attention의 결과값을 구할 수 있음



  - 예를 들어 I love you라는 문장에서 I라는 Query에 대해 Attention Score를 구하고 싶다면, 다음 사진과 같이 구할 수 있다.

Mask Matrix

마스크 행렬을 통하여 특정 단어는 무할 수 있다.

마스크 값으로 음수 무한의 값을 넣어서 softmax 함수의 출력이 0%에 가까워지도록 한다.

- Attention Energy와 같은 크기의 Mask Matrix를 생성하여 elementwise하여 어떠한 단어는 참고하지 않도록 함

Attention 종류

1) Encoder Self Attention

- 각각의 단어가 서로에게 어떠한 연관성을 가지는지 Attention을 통해 구하도록 함
- 전체 문장에 대한 Representation을 Learning할 수 있도록 만든다는 점이 특징

2) Masked Decoder Self-Attention

- 각각의 출력 단어가 앞쪽에 등장했던 단어들만 참고할 수 있도록 만들어서 cheating(출력할 때 뒤의 문장을 참고하는 것)을 하지 않고 정상적으로 모델이 학습될 수 있도록 만든다.

3) Encoder-Decoder Attention

- Query는 Decoder에 존재
- Key, Value는 Encoder에 존재
- 각각의 출력 단어들이 이러한 입력 단어들 중에서 어떠한 정보에 더욱 더 많은 가중치를 두는지 구할 수 있게 함