한눈에 보는 딥러닝 기반 기계번역 / 작성자: 박찬준 (bcj1210@naver.com)

기계번역 소개 및 흐름

기계번역이란 소스문장(Source Sentence)을 타겟문장(Target Sentence)로 컴퓨터가 번역하는 시스템을 의미한다. 기계번역은 크게 3 가지 흐름으로 변화되어 왔다.

> Machine translation is a sub-field of computational linguistics that investigates the use of software to translate text or speech from one language to another.
>
> From Wikipedia

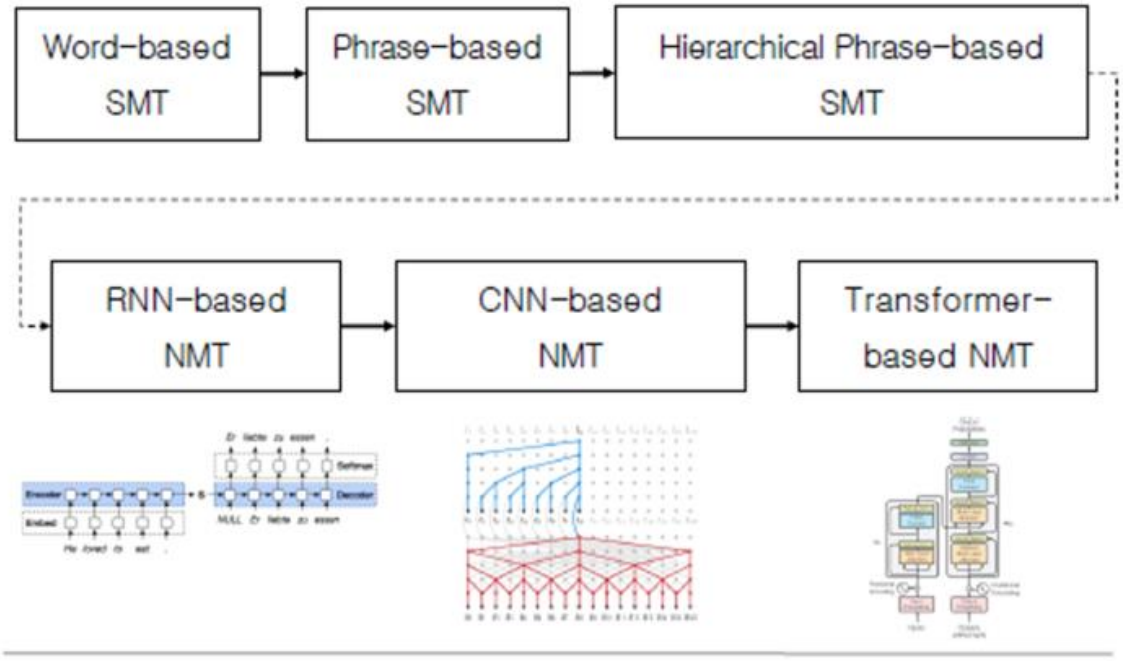
<그림 1 기계번역의 정의>

규칙기반 기계번역은(RBMT) 간단히 말해 언어학적, 문법적으로 언어를 번역하는 것을 의미하며 소스 문장에 해당하는 언어를 형태소, 구문분석 등의 과정을 거쳐 타겟 언어의 문법적 규칙에 맞게 변환시켜서 번역을 진행하는 방법이라 할 수 있다. 과거에 사용되던 방식으로 문법 규칙 추출이 어렵고, 언어학적인 지식이 많이 필요하다. 또한 번역 언어 확장이 어려우며 코드가 길다는 단점이 있다

통계기반 기계번역(SBMT)은 대용량 코퍼스(corpus)로부터 학습된 통계정보를 통하여 번역을 진행하는 방식이다. 간단히 생각해 확률을 이용하는 방식이라고 생각할 수 있다. 어느 정도 정확성 있는 기계번역기를 만들려면 최소 200 만개 이상의 말뭉치가 필요하며 말뭉치가 많으면 많을수록 성능이 향상된다. 번역모델과 언어모델로 구성되어 있으며 번역모델을 통하여 소스문장과 타겟문장의 Alignment 를 추출하고 언어모델을 통하여 타겟문장의 확률을 예측하게 된다.

인공신경망 기계번역(NMT)은 가장 최신의 기술로서 알파고에 쓰인 기술로도 유명한 딥러닝(Deep Learning)을 이용하여 기계번역을 진행하는 방식이다. 딥러닝을 이용하여 입력과 출력의 문장을 하나의 쌍으로 두고 가장 적합한 표현과 번역결과물을 찾는 방식이라고 할 수 있습니다. 즉 딥러닝을 이용한 end-to-end 번역시스템이다.

딥러닝 기반 기계번역의 흐름



<그림 2 딥러닝 기반 기계번역의 흐름>

기계번역은 규칙기반 기계번역을 거쳐 단어 단위, 구 단위(Phrase-based), 계층적 구 단위(Hierarchical Phrase-based), 구문 단위(Syntax based) 통계기반 기게번역(Word-based SMT)을 거치며 발전되어 왔다. 여기까지의 흐름이 RBMT 와 SMT 이다.

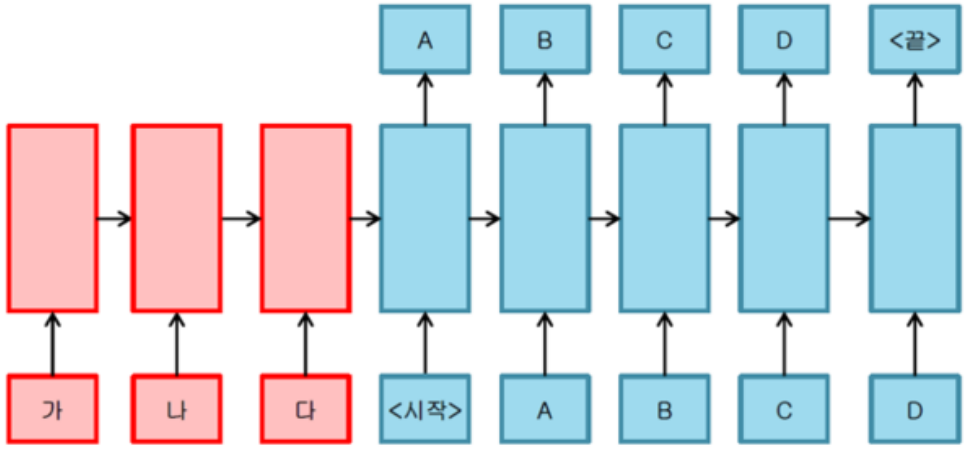
이후 Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation(cho. 2014)논문을 통하여 딥러닝 기반인 RNN 을 사용하여 기계번역을 개발하기 시작했으며 2016 년부터 본격적으로 NMT 의 시대에 접어들게 되었다. Neural Machine Translation By Jointly Learning to alingn and translate(Bahanau,2016)의 논문에서 소개된 Attention 이라는 개념을 통하여 SMT 와 비교하여 눈에 띄는 성능차이를 보이기 시작했다. 이후 GNMT, Papago, PNMT 등 실제 NMT 를 이용한 상용화 기계번역 시스템이 개발되기 시작했다. 또한 최근에는 Attention is all you need 라는 논문을 통해 소개된 Transformer 라는 모델이 가장 좋은 성능을 보이고 있으며 현재 대부분의 기업에서 해당 모델을 채택하여 서비스를 진행하고 있다. 뿐만 아니라 Transformer 를 통하여 Q&A, Languge Representation, Summarization 등 다양한 자연어처리 분야에서 좋은 성과를 보이고 있다. 이제부터 RNN 기반 NMT 부터 Transformer 까지 NMT 의 기술 변화의 흐름을 자세히 살펴본다.

Sequence to Sequence 구조와 인코더 디코더

Neural Machine Translation(이하 NMT)는 딥러닝 기반 기계번역기이다.
학습에 필요한 것은 오직 Parallel Corpus 즉 병렬 코퍼스(Corpus)이다.

NMT 는 Sequence to Sequence 구조라고 불리운다. 간단히 생각하여 문장 to 문장이라는 표현이다. 즉 한국어- 영어 번역기라면 "한국어 문장 to 영어 문장" 으로 번역하니 문장 to 문장 즉 Sequence to Sequence 인 것이다.

- 구조



- Encoder와 Decoder로 구성.
- Encoder에서 입력 문장 Vector화.
- Decoder에서 Vector화된 입력 문장 복호화.

<그림 3 인코더-디코더 구조>

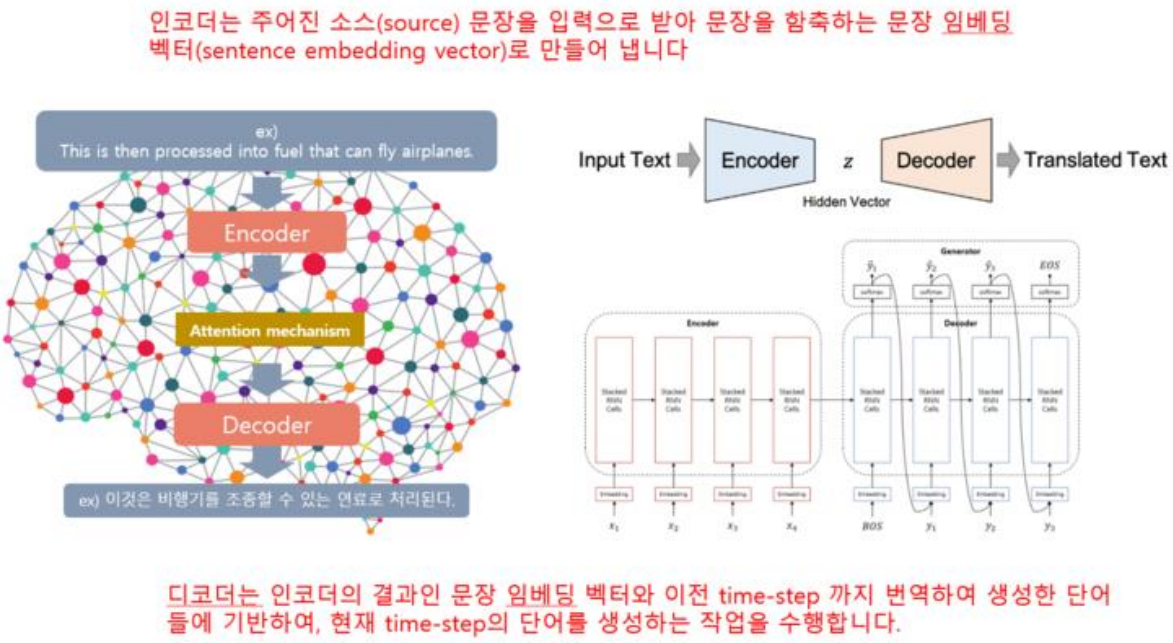
Sequence 2 Sequence 를 다르게 표현하면 바로 인코더-디코더 구조이다.
인코더란 간단히 생각하면 컴퓨터가 이해할 수 있는 형태로 인간의 언어를 바꾸어주는 것이다.
컴퓨터는 숫자만 이해할 수 있으며 사람의 언어를 이해할 수 없다. 따라서 컴퓨터가 이해할 수 있는 지식표현 체계인 숫자로 바꾸어 주어야 한다. 그 역할을 Encoder 에서 진행하게 된다.

그렇다면 어떠한 숫자로 바꾸어 주는가?
바로 고차원의 Vector 형태로 바꾸어 준다.

디코더는 인코더의 결과인 문장 임베딩 벡터와 이전 time-step 까지 번역하여 생성한 단어들에 기반하여, 현재 Time-step 의 단어를 생성하는 작업이다. 간단히 생각하면 그냥 "번역을 한다" 는 것을 의미한다.

Encoder 에서 딥러닝 학습을 할 수 있게 학습데이터를 변경해주고(고차원 벡터로, 워드임베딩) 그 다음 RNN 을 통하여 Context Vector 를 생성하고 이를 기반으로 디코더에서 번역을 진행하게 되는 것 이다.

RBMT, SMT 처럼 복잡한 구조가 아닌 단순하게 병렬 코퍼스 데이터만 있으면 컴퓨터가 알아서 학습을 하게 된다. 즉 End to End 방식인 것이다.



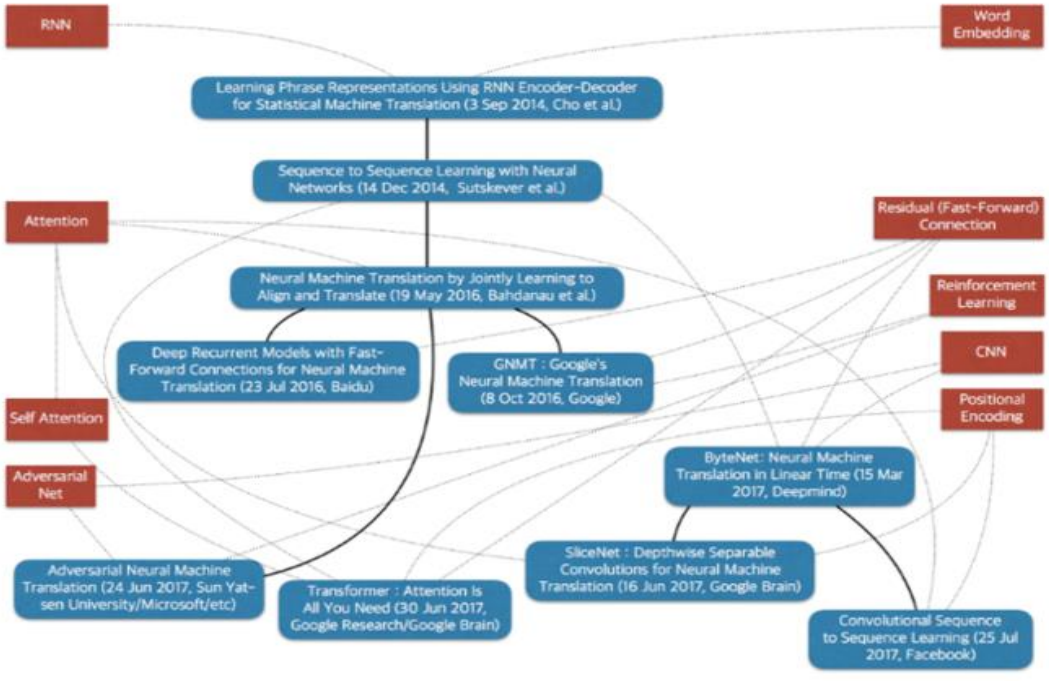
<그림 4 기계번역에서 인코더-디코더 구조>

인코더- 디코더 구조는 기계번역 뿐만 아니라 챗봇, 요약, 음성인식 등 다양한 분야에서 활용할 수 있다.

기계번역	특정 언어 문장을 입력으로 받아 다른 언어의 문장으로 출력
챗봇	사용자의 문장 입력을 받아 대답을 출력
요약(Summarization)	긴 문장을 입력으로 받아 같은 언어의 요약된 문장으로 출력
기타 자연어처리 분야	사용자의 문장 입력을 받아 프로그래밍 코드로 출력 등
음성인식	사용자의 음성을 입력으로 받아 해당 언어의 문자열(문장)으로 출력
독순술	입술 움직임의 동영상을 입력으로 받아 해당 언어의 문장으로 출력
이미지 캡셔닝 (captioning)	변형된 seq2seq를 사용하여 이미지를 입력으로 받아 그림을 설명하는 문장을 출력

RNN 기반 Neural Machine Translation

본격적으로 딥러닝 기반 NMT 에 대해서 설명한다. 현재 가장 많이 사용되면서 뛰어난 성능을 보이는 모델은 바로 Google 에서 개발한 Transformer 이다. 그 이전에는 RNN 과 CNN 기반의 NMT 를 많이 사용했기에 본서는 바로 Transformer 를 설명하는 것이 아닌 2014 년부터 시작된 RNN 및 CNN 기반 NMT 부터 Attention 기반 NMT 그리고 Transformer 에 대해서 자세히 설명한다.

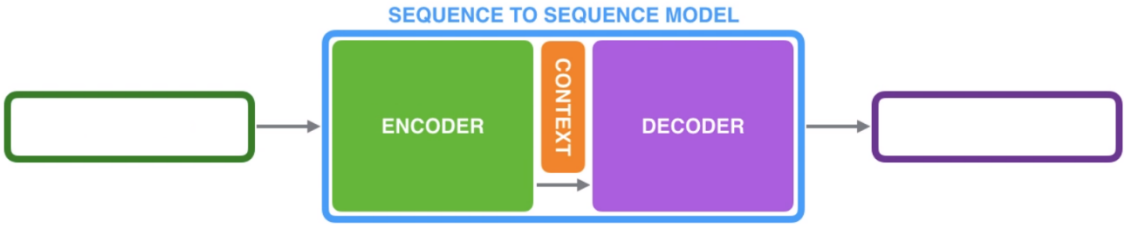


<그림 5 NMT 흐름도>

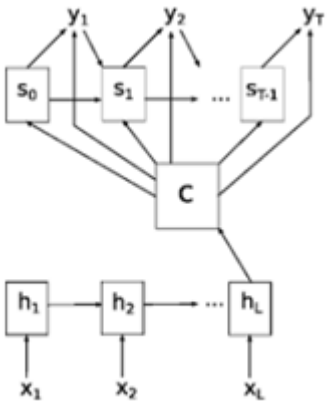
이번 챕터에서는 RNN 기반의 기계번역에 대해서 설명한다.

NMT 의 시작점이 한국인으로부터 시작되었다는 사실을 알고 있었는가?
현재 뉴욕대학교 교수이자 페이스북에서 근무하고 계시는 조경현의 논문이 NMT 의 시작점이다.
아래의 그림 RNN 기반 기계번역 모델을 설명하는 그림이다.

encoder 는 입력의 각 아이템을 처리하여 거기서 정보를 추출한 후 그것을 하나의 벡터로 만들어낸다 (흔히 context 라고 불림). 입력의 모든 단어에 대한 처리가 끝난 후 encoder 는 context 를 decoder 에게 보내고 출력할 아이템이 하나씩 선택되기 시작한다.



<그림 6 RNN 기반 기계번역 1>



<그림 7 RNN 기반 기계번역 2>

그러나 단순 RNN 기반 Sequence to Sequence 방식은 Encoder 의 마지막 vector 만의 정보를 이용한다는 한계점이 있다. 위 그림을 보면 X1 부터 XL 까지 모든 입력이 C 라는 벡터 한개로 표현이 된다. 그 C 를 Context Vector 라고 한다.

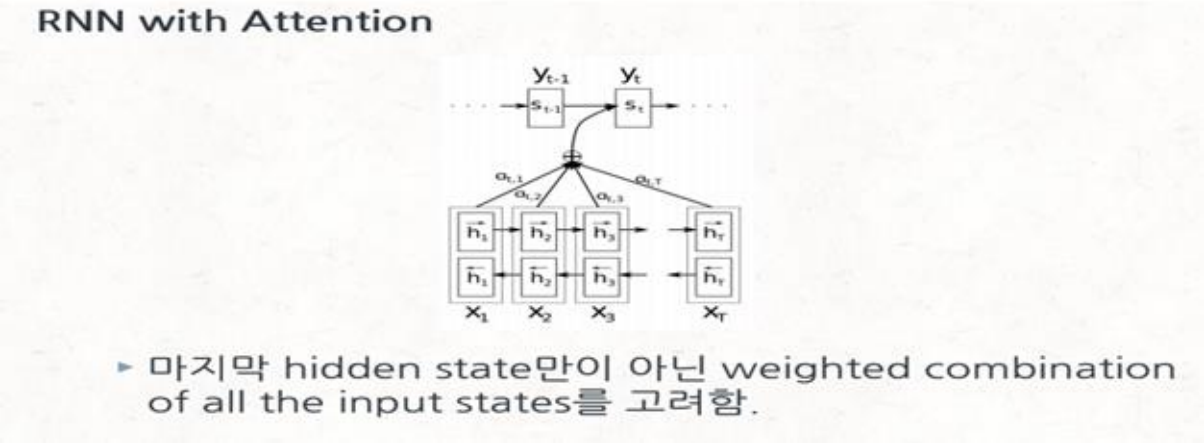
위 방식은 긴 문장이 들어오면 번역이 잘 안되는 문제점이 있으며 학습데이터에 존재하지 않는 단어가 나오면 생략해버리거나, 앞부분에 대해 반복 번역을 진행할 때가 있다. 또한 Context

Vector 는 고정된 벡터인데 이러한 점 때문에 긴 문장을 잘 번역하지 못한다는 치명적인 한계점이 있다. 여기까지가 2014 년에 나온 논문인 "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation"에 대한 내용이다.

결론적으로 단순 RNN 및 LSTM 을 기반으로 하는 NMT 는 SMT 와 비교하여 경쟁력있는 성능차이를 보여주지 못하였다. 그러나 2015 년 Bahadanau 가 발표한 Neural Machine Translation By Jointly Learning To Align And Transaltion 이라는 논문에서 Attention 이 발표되면서 본격적으로 NMT 가 빛을 보게 된다.

Attention 의 등장

단순 RNN 기반 NMT 는 Context 벡터가 가장 큰 걸림돌인 것으로 밝혀졌다. 이렇게 하나의 고정된 벡터로 전체의 맥락을 나타내는 방법은 특히 긴 문장들을 처리하기 어렵게 만들었다. 이에 대한 해결 방법으로 제시된 것이 바로 "Attention" 이다 Bahdanau et al., 2014 논문이 소개한 attention 메커니즘은 seq2seq 모델이 디코딩 과정에서 현재 스텝에서 가장 관련된 입력 파트에 집중할 수 있도록 해줌으로써 기계 번역의 품질을 매우 향상 시켰다.

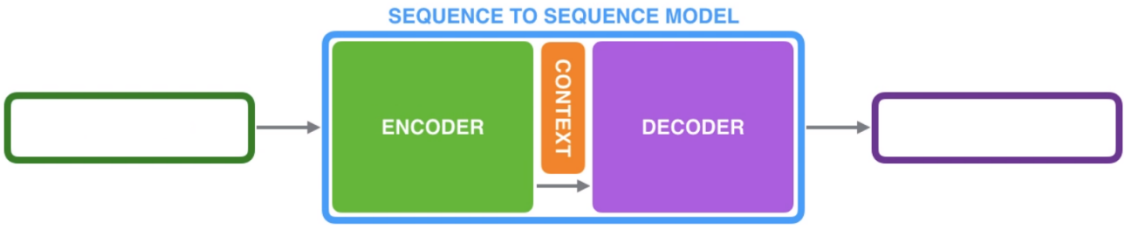


<그림 8 RNN with Attention 1>

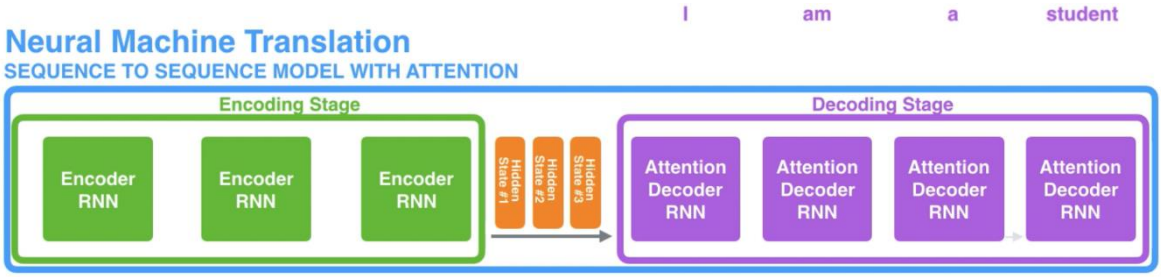
스텝마다 관련된 부분에 더 집중할 수 있게 해주는 attention model 은 attention 이 없는 모델보다 훨씬 더 좋은 결과를 생성한다. 즉 특정 타임프레임에 Weight 를 높게 주는 방식이다. 어떠한 시점에 어떤 부분을 집중할지 그때 그때 동적으로 정하게 되는 것이다.

그렇다면 단순 RNN 기반 NMT 와 Attention 기반 NMT 는 어떠한 차이가 존재할까?

기존 RNN 모델에서는 그저 마지막 아이템의 hidden state 즉 Context 벡터를 디코더에 넘겼던 반면 attention 모델에서는 *모든* 스텝의 hidden states 를 decoder 에게 넘겨준다.



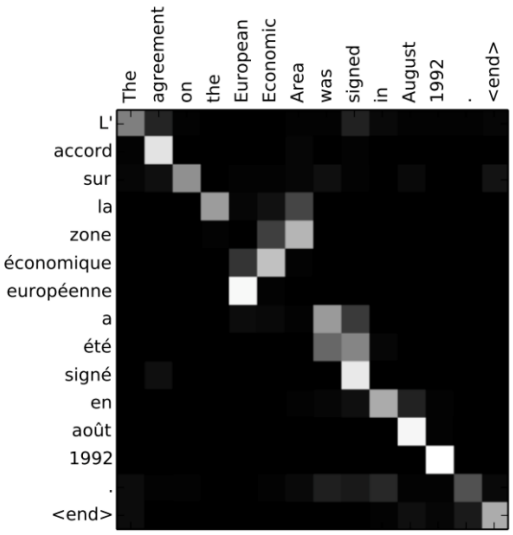
<그림 9 RNN without Attention>



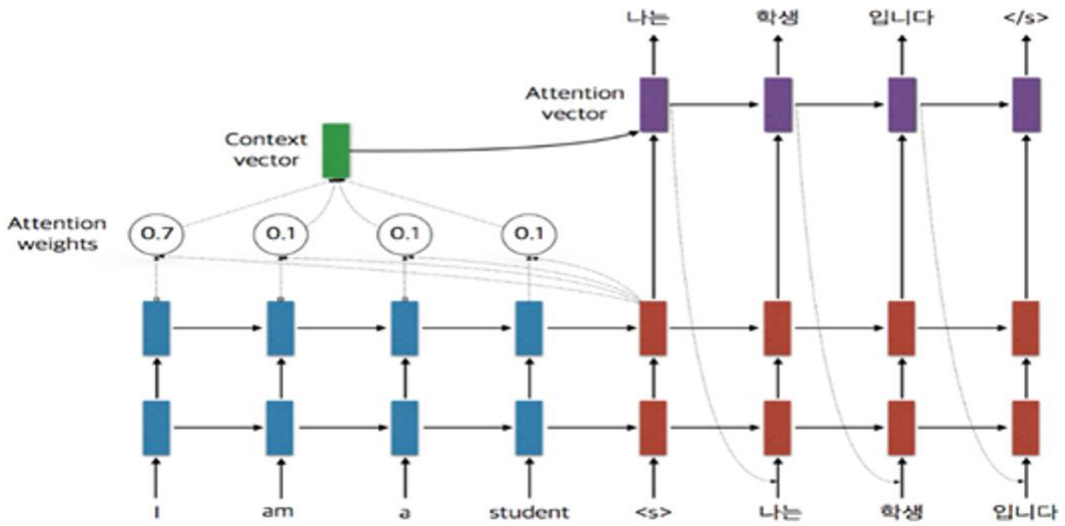
<그림 10 RNN with Attention>

두 그림의 차이에서 알 수 있듯 Attention 을 사용한 모델은 각 타임스텝마다 동적으로 생성된 Context vector 모두를 디코더에 넘겨주게 된다.

Attention 을 이용하면 각 decoding 스텝에서 입력 문장에서 어떤 부분을 집중하고 있는지에 대해 볼 수 있다.



<그림 11 Attention Visualization1>



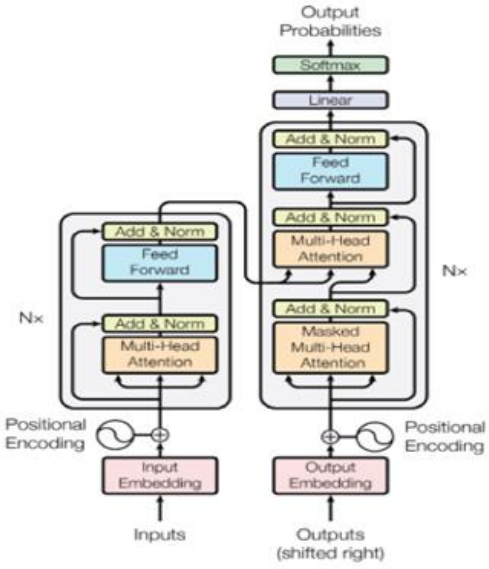
<그림 12 Attention Visualization2>

결론적으로 Attention 을 사용하여 NMT 가 가지고 있던 기존의 단점들의 많은 부분을 극복할 수 있었다.

여기에서 더 나아간 것이 Attention is all you need 논문에서 소개된 Transformer 이다..

Transformer 는 RNN 은 철저히 배제하고 Attention 만을 이용한 방식이다.

Transformer

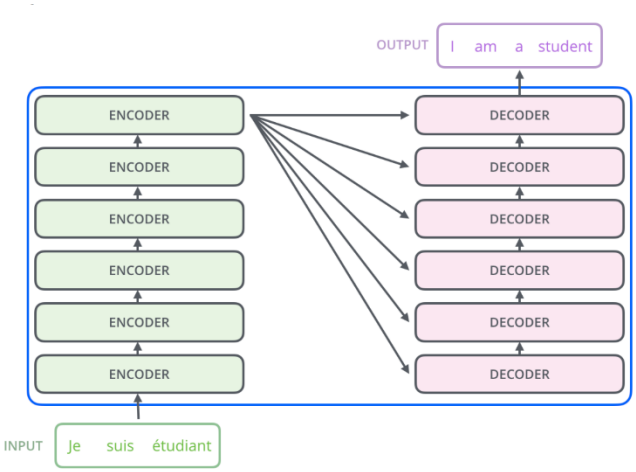


<그림 13 Transformer>

Transformer 은 Attention is All You Need 이라는 논문을 통해 처음 발표되었다.
논문제목에서 알 수 있듯 RNN 과 CNN 에 사용 없이 오직 Attention 만을 이용한 모델이다.

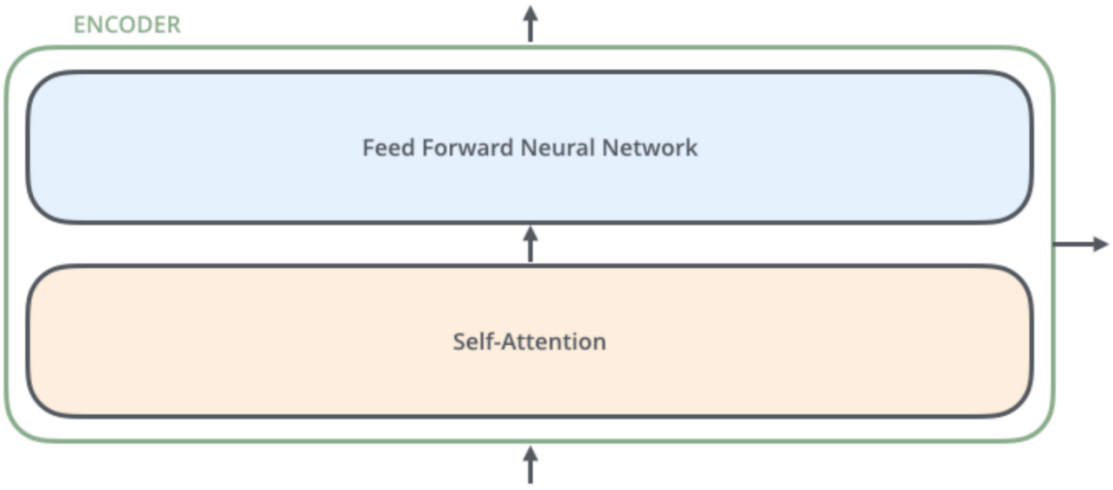
Transformer 는 병렬화가 가능하고 학습속도가 상당히 빠르다. 기존에 1 주일 걸리던 것이 2~3 일이면 해결된다.

Transformer 도 결국 Encoder-Decode 구조이다.



<그림 14 Transformer Encoder-Decoder>

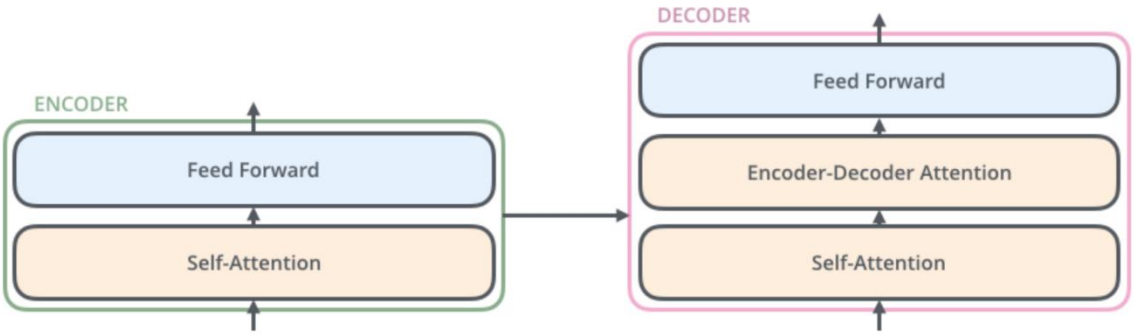
먼저 인코더의 구조를 살펴본다.



<그림 15 Transformer Encoder>

인코더에 들어온 입력은 일단 먼저 self-attention 블럭 을 지나가게 된다. 이 블럭은 encoder 가 하나의 특정한 단어를 encode 하기 위해서 입력 내의 모든 다른 단어들과의 관계를 살펴본다. 간단히 생각하여 Self-Attention 이란 input 단어들 간의 Attention, 즉 자기들끼리의 Attention 이다.

입력이 self-attention 층을 통과하여 나온 출력은 다시 feed-forward 신경망으로 들어가게 된다. 똑같은 feed-forward 신경망이 각 위치의 단어마다 독립적으로 적용돼 출력을 만들게 된다.



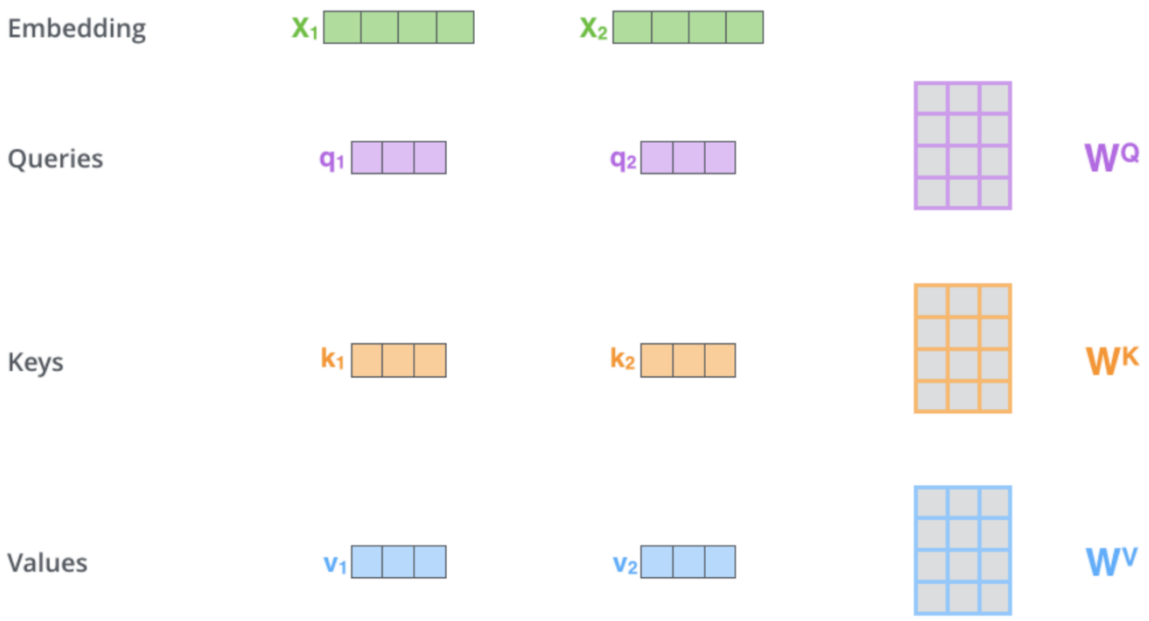
<그림 16 Transformer Decoder>

decoder 또한 encoder 에 있는 두 layer 모두를 가지고 있다. 그러나 그 두 층 사이에 seq2seq 모델의 attention 과 비슷한 encoder-decoder attention 이 포함되어 있다. 이는 decoder 가 입력 문장 중에서 각 타임 스텝에서 가장 관련 있는 부분에 집중할 수 있도록 해준다.

Self- Attention

self-attention 계산의 가장 첫 단계는 encoder 에 입력된 벡터들에게서 부터 각 3 개의 벡터를 만들어내는 일이다.

각 단어에 대해서 Query 벡터, Key 벡터, 그리고 Value 벡터를 생성한다. 이 벡터들은 입력 벡터에 대해서 세 개의 학습 가능한 행렬들을 각각 곱함으로써 만들어진다.



<그림 17 Query,Key,Value>

x1 를 weight 행렬인 WQ 로 곱하는 것은 현재 단어와 연관된 query 벡터인 q1 를 생성한다. 같은 방법으로 입력 문장에 있는 각 단어에 대한 query, key, value 벡터를 만들 수 있다.

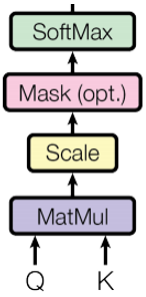
self-attention 계산의 두 번째 스텝은 점수를 계산하는 것이다.
A 라는 단어와 입력 문장 속의 다른 모든 단어들에 대해서 각각 점수를 계산하여야 한다. 이 점수는 현재 위치의 이 단어를 encode 할 때 다른 단어들에 대해서 얼마나 집중을 해야 할지를 결정한다.

점수는 현재 단어의 Query vector 와 점수를 매기려 하는 다른 위치에 있는 단어의 Key vector 의 내적으로 계산된다. 즉 Query 와 Key 벡터를 곱해준다.

세 번째와 네 번째 단계는 이 점수들을 8 로 나누는 것이다.

왜 8 인가? 이 8 이란 숫자는 key 벡터의 사이즈인 64 의 제곱 근이라는 식으로 계산이 된 것이다. 이 나눗셈을 통해 더 안정적인 gradient 를 가지게 된다. 그리고 난 다음 이 값을 softmax 계산을 통과시켜 모든 점수들을 양수로 만들고 그 합을 1 으로 만들어 준다.

이 softmax 점수는 현재 위치의 단어의 encoding 에 있어서 얼마나 각 단어들의 표현이 들어갈 것인지를 결정한다. 당연하게 현재 위치의 단어가 가장 높은 점수를 가지며 가장 많은 부분을 차지하게 되겠지만, 가끔은 현재 단어에 관련이 있는 다른 단어에 대한 정보가 들어가는 것이 도움이 된다. 현재까지 진행과정을 그림으로 표현하면 아래와 같다.

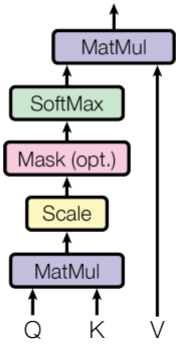


<그림 18 Query,Key 내적>

다섯 번째 단계는 이제 입력의 각 단어들의 value 벡터에 이 점수를 곱하는 것이다.. 이것을 하는 이유는 우리가 집중을 하고 싶은 관련이 있는 단어들은 그대로 남겨두고, 관련이 없는 단어들은 0.001 과 같은 작은 숫자 (점수)를 곱해 없애버리기 위함이다.

마지막 여섯 번째 단계는 이 점수로 곱해진 weighted value 벡터들을 다 합해 버리는 것이다. 이 단계의 출력이 바로 현재 위치에 대한 self-attention layer 의 출력이 된다. 현재까지 모든 진행과정을 그림으로 표현하면 아래와 같으며 현재까지의 작업을 Scale Dot Product Attention 이라고 한다.

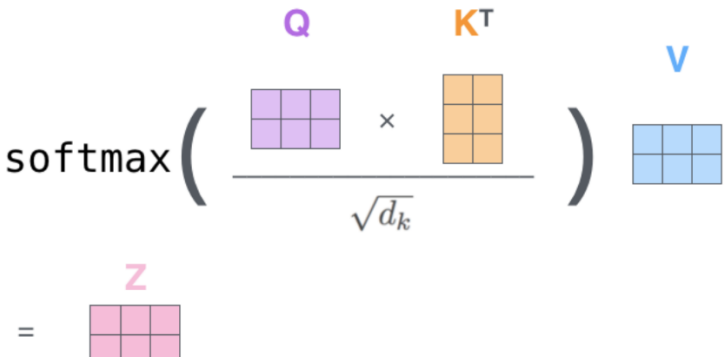
Scaled Dot-Product Attention



<그림 19 Scaled Dot-Product Attention>

이제까지의 모든 과정을 수식으로 정리하면 아래와 같다.

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



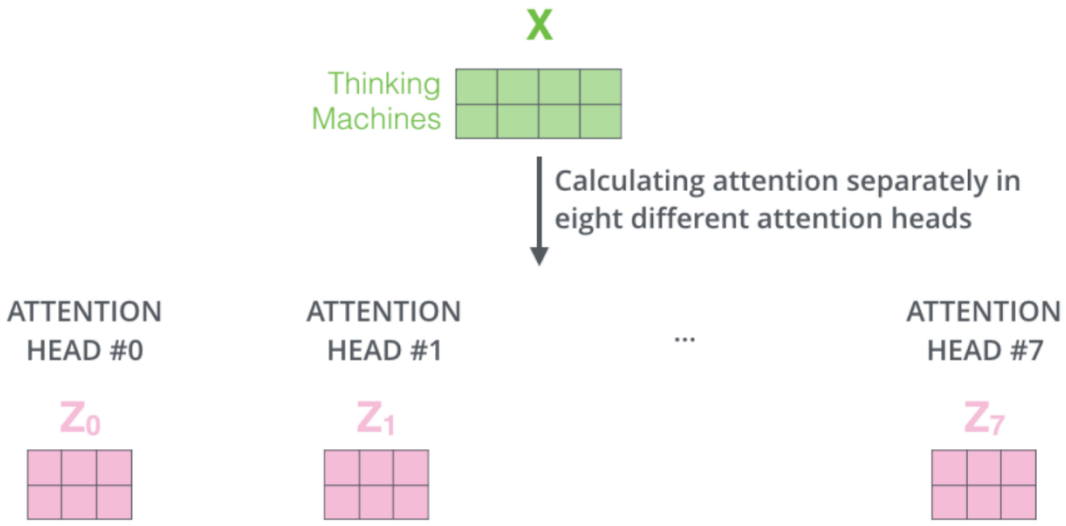
이 여섯 가지 과정이 바로 self-attention 의 계산 과정이다. 우리는 이 결과로 나온 벡터를 feed-forward 신경망으로 보내게 된다.

Multi Head Attention

Self-attention layer 에 multi-head attention 이라는 메커니즘을 더해 성능을 더욱 더 개선할 수 있다.

Multi-head attention 을 이용함으로써 여러 개의 Query, key, Value weight 행렬들을 가지게 된다. 논문에서 제안된 구조는 8 개의 Attention heads 를 가지므로 각 encoder/decoder 마다 이런 8 개의 세트를 가지게 되는 것이다. 이러한 세트가 여러 개 있다는 것은 각 벡터들을 각각 다른 representation 공간으로 나타낸 다는 것을 의미한다.

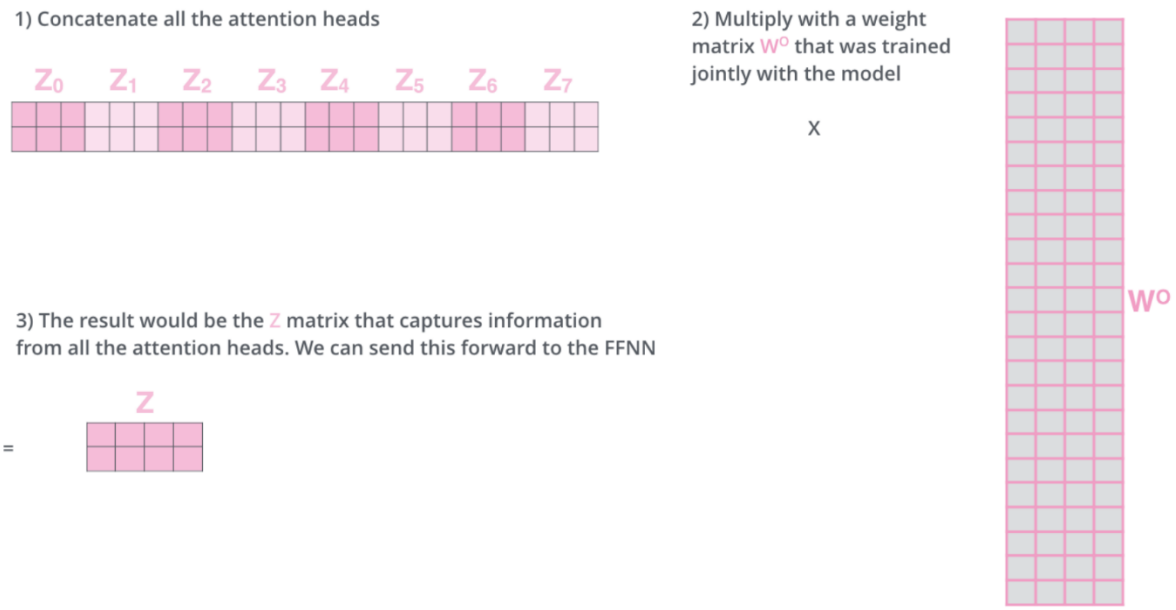
Multi-headed attention 을 이용하기 위해서 각 head 를 위해서 각각의 다른 Query, key, Value weight 행렬들을 모델에 가지게 된다. 이전에 설명한 것과 같이 입력 벡터들의 모음인 행렬 X 를 WQ,WK,WV 행렬들로 곱해 각 head 에 대한 Query, key, Value 행렬들을 생성한다



<그림 20 Multi Head Attention>

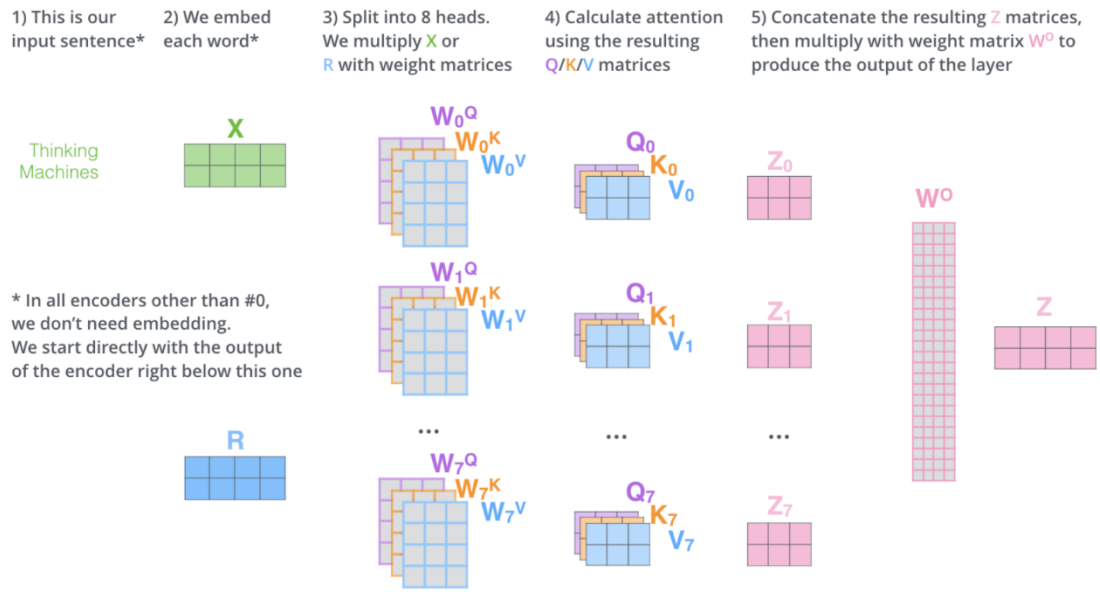
위에 설명했던 대로 같은 self-attention 계산 과정을 8 개의 다른 weight 행렬들에 대해 8 번 거치게 되면, 8 개의 서로 다른 Z 행렬을 가지게 된다. 간단히 생각하여 Self Attention 을 8 번 진행하여 각각 다르게 표현된 Attention Head 를 얻게 되는 것이다.

그러나 문제는 이 8 개의 행렬을 바로 feed-forward layer 으로 보낼 수 없다.
feed-forward layer 은 한 위치에 대해 오직 한 개의 행렬만을 input 으로 받을 수 있다. 그러므로 이 8 개의 행렬을 하나의 행렬로 합쳐야 한다.
어떻게 할 수 있을까? 일단 모두 이어 붙여서 하나의 행렬로 만들어버리고, 그 다음 하나의 또 다른 weight 행렬인 W0 을 곱해버리면 된다.

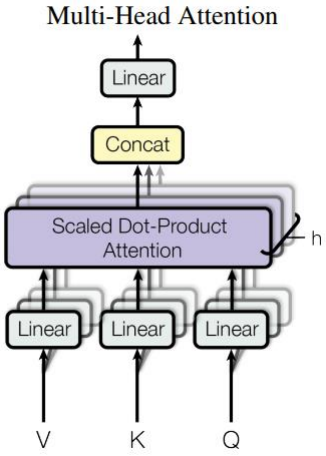


<그림 21 Multi Head Attention>

결국 1 개의 행렬로 Multi Head Attention 값이 표현되게 된다.
지금까지 진행된 과정을 총 정리해보면 아래 그림과 같다.



<그림 22 Multi Head Attention>



<그림 23 Multi Head Attention>

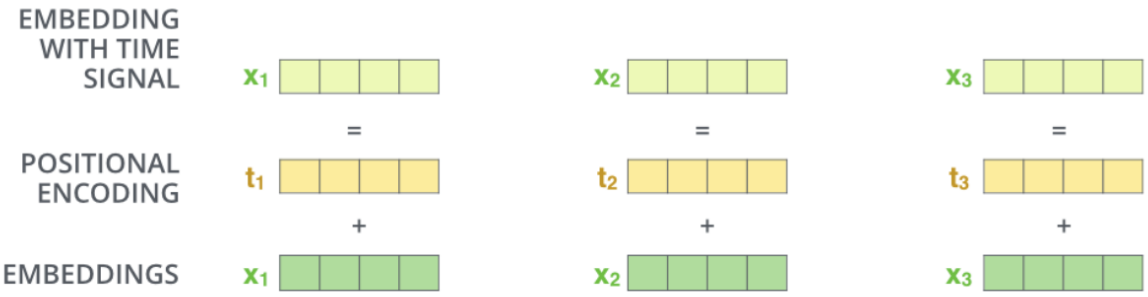
Multi Head Attention 의 수식은 아래와 같다.

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

Positional Encoding

Transformer 모델에서 입력 문장에서 단어들의 순서에 대해서는 어떻게 고려할까?

Transformer 모델은 각각의 입력 embedding 에 "positional encoding"이라고 불리는 하나의 벡터를 추가한다. 이 벡터들은 모델이 학습하는 특정한 패턴을 따르는데, 이러한 패턴은 모델이 각 단어의 위치와 시퀀스 내의 다른 단어 간의 위치 차이에 대한 정보를 알 수 있게 해준다.

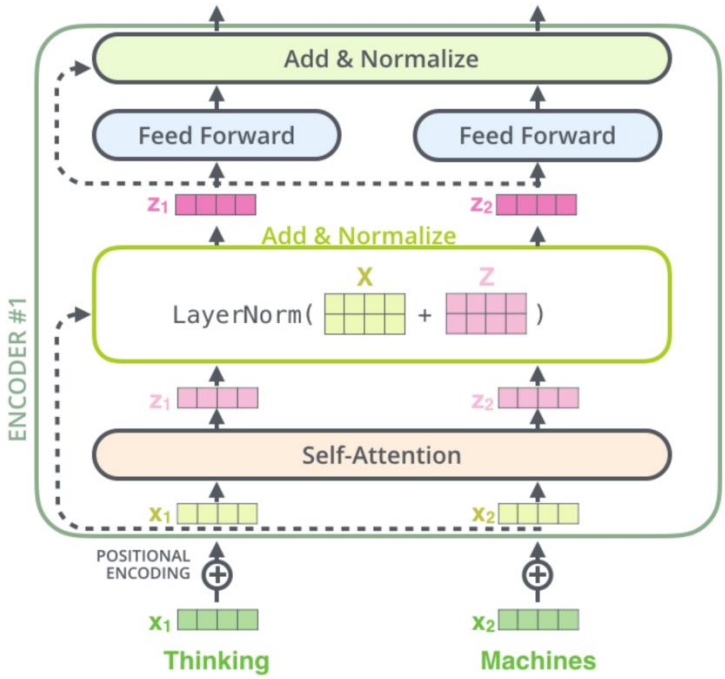


<그림 24 Positional Encoding>

Residual & Normalization

각 encoder 내의 sub-layer 가 residual connection 으로 연결되어 있으며, 그 후에는 layer-normalization 과정을 거친다는 거치게 된다.

이 벡터들과 layer-normalization 과정을 시각화해보면 다음과 같다.



<그림 25 Residual & Normalization >

이것은 decoder 내에 있는 sub-layer 들에도 똑같이 적용되어 있다.

Decoder

이제 Decoder 에 대해서 알아본다. 사실 이제까지 Encoder 와 Decoder 에 차이는 거의 없고 딱 2 가지 면에서 차이가 존재한다.

<1> Decoder 에서의 self-attention layer 은 output sequence 내에서 현재 위치의 이전 위치들에 대해서만 Attention 을 진행 할 수 있다. 이것은 self-attention 계산 과정에서 softmax 를 취하기 전에 현재 스텝 이후의 위치들에 대해서 masking 을 해줌으로써 가능해진다.

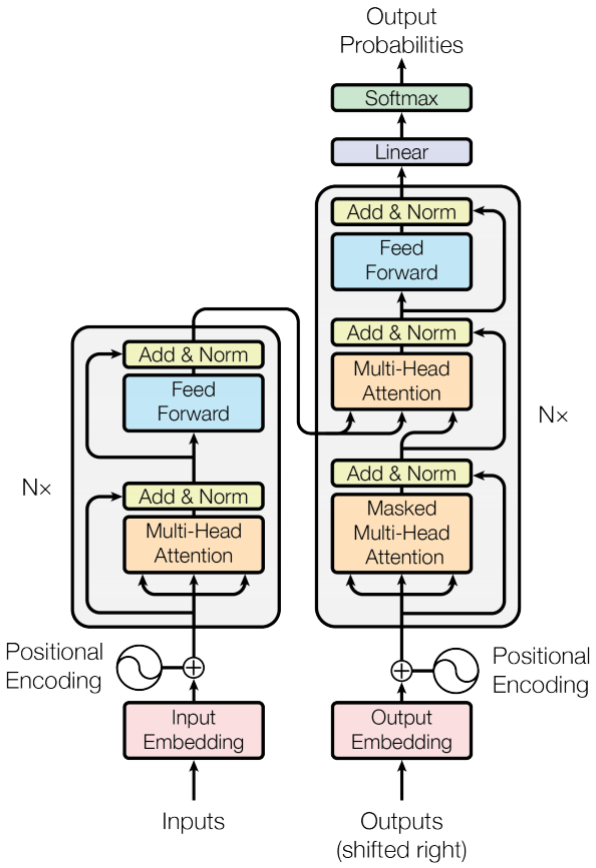
<2> "Encoder-Decoder Attention" layer 은 multi-head self-attention 과 한 가지를 제외하고는 똑같은 방법으로 작동한다. 그 한가지 차이점은 Query 행렬들을 그 밑의 layer 에서 가져오고 Key 와 Value 행렬들을 encoder 의 출력에서 가져온다는 점이다.

Linear Layer & Softmax

여러 개의 decoder 를 거치고 난 후에는 소수로 이루어진 벡터 하나가 남게 된다. 어떻게 이 하나의 벡터를 단어로 바꿀 수 있을까?
이것이 바로 마지막에 있는 Linear layer 과 Softmax layer 가 하는 일이다.
Linear layer 은 fully-connected 신경망으로 decoder 가 마지막으로 출력한 벡터를 그보다 훨씬 더 큰 사이즈의 벡터인 logits 벡터로 투영시킨다.
우리의 모델이 training 데이터에서 총 10,000 개의 영어 단어를 학습하였다고 가정한다. 그렇다면 이 경우에 logits vector 의 크기는 10,000 이 될 것이다. 이 벡터의 각 셀은 그에 대응하는 각 단어에 대한 점수가 된다. 이렇게 되면 우리는 Linear layer 의 결과로서 나오는 출력에 대해서 해석을 할 수 있게 된다.

softmax layer 는 이 점수들을 확률로 변환해주는 역할을 한다. 셀들의 변환된 확률 값들은 모두 양수 값을 가지며 다 더하게 되면 1 이 된다. 가장 높은 확률 값을 가지는 셀에 해당하는 단어가 해당 스텝의 최종 결과물로서 출력되게 된다.

이제까지 Transformer 에 대해서 모든 설명을 진행해보았다. 이제 아래 그림을 보고 본인의 표현으로 직접 Transformer 를 설명해보기 바란다.



이제까지 딥러닝 기반 기계번역에 대해서 살펴보았다. 본서로 학습한 이론을 바탕으로 직접 구현을 진행해보거나 오픈소스를 활용하여 직접 NMT 를 만들어보기를 추천한다.

실습

1. https://github.com/tensorflow/nmt
2. https://github.com/Parkchanjun/OpenNMT-Colab-Tutorial