NLP

- Seq2seq 모델

- sequence 목록을 Input 받게되는데 단어, 문자, 이미지의특징 등 여러 가지 형태의 Input을 받게됨.

- Input length와 Output length의 크기가 같은 필요 없다

- Main idea

● 두가지의 구조로 결합되어있다 (encoder / decoder)

▶ Encoder : 입력된 정보를 어떻게 처리해서 저장할까?

▶ Decoder : Encoder로부터 입력받은 정보를 어떻게 풀어서 반환해줄까?

● Encoder는 input sequence에서의 각 item을 가공하고, 정보들을 capture한 vector로 만듦, 이를 'Context Vector'라고 한다. 이를 'Decoder'에게 넘겨주고,

● Decoder는 context vector를 받아서 (item by item) 형태의 Output sequence를 만듦

- Encoder-Decoder의 구조

● RNN기반

▶ machine translation에서 'Context'는 **Vectorization**됨

▶ hidden state는 input이 들어올때마다 한번씩 update되고, 최종적으로 모든 input이 들어온 후의 hidden state는 'context vector'가 되어 decoder로 전달된다

▶ 그후, decoder는 가장마지막 hidden state vector를 통해서 output 생성

▶ 즉, 인코더에서의 **"final hidden state = context vector"**

- Attention 모델

▶ RNN은 구조적으로 gradient가 vanishing 되기에 long-term dependency >> LSTM이나 GRU가 완화시켜줄 순 있지만, 해결은 불가능하다 >> Attention

▶ Context Vector는 긴 문장(long sentence)을 다루는 모델에서 사용되면 **bottleneck**(병목현상)이 나타남

▶ Input sequence가 output sequence에 필요한 정보들을 직접적으로 전달(=가중치를 주어)해주는 것을 가능케한다

- 종류

● Bahadanau Attention

▶ 어텐션 스코어 자체를 학습하는 모델이 존재

● Luong Attention

▶ 어텐션 스코어는 따로 학습을 시키지 않고, 현재의 hidden state와 과거의 hidden state 간의 유사도를 측정해 어텐션 스코어 산출

✓ 두 모델의 성능 차이는 거의 없다. 이는 즉, 별도로 모델을 학습시켜야하는 메커니즘(bahadanau attention)과 모델을 학습시킬 필요 없고 단순한 곱셈연산을 통해 스코어값을 도출해내는 메커니즘(Luong attention)중에 더 효율적인(실용적인) 모델은 당연하게도 후자의 모델이 될 것이다.

- Attention모델이 기존 Seq2seq모델과의 다른점

● 디코더에 더 많은 정보(데이터)를 넘겨줌으로써 성능↑

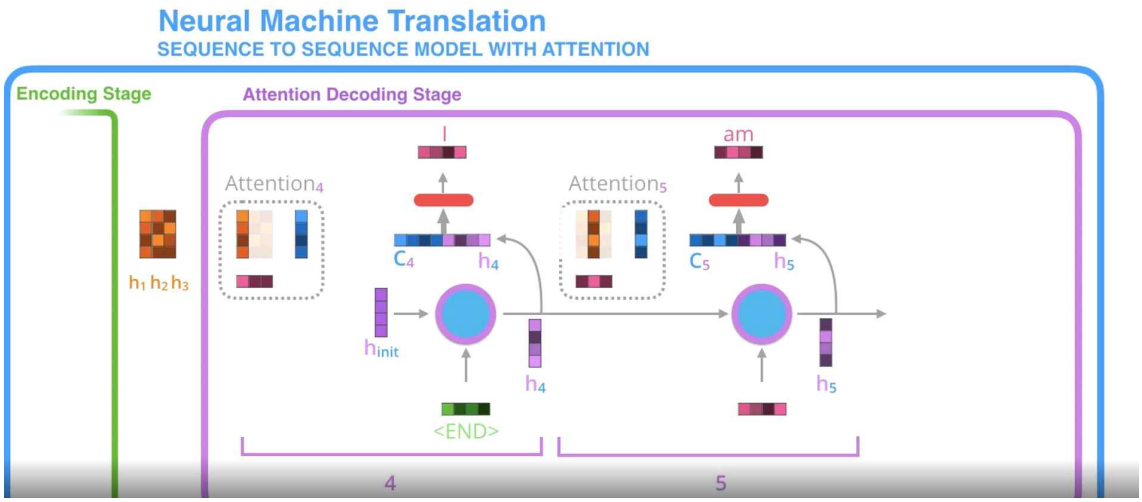
▶ 가장 마지막 hidden state(Context Vector)만을 넘겨주는 것이 아니라, 모든 hidden state를 디코더에 넘겨준다

▶ **Decoder는 필요한 hidden state를 취사선택해 서로 다른 가중치를 이용해 활용가능하다**

● 디코더는 Output을 처리하기전에 인코더가 전해준 정보들을 바탕으로 'Extra Step'을 수행한다

1. 인코더가 전해준 모든 hidden state를 받고
2. 각각의 hidden state에게 score값을 산출 (Input hidden state와 Decoder의 hidden state간의 내적dot-product) >> Context vector 생성
3. 디코더의 hidden state와 앞서 생성된 Context vector를 결합(Concatenate)
3. score에 대해 softmax를 수행한 후, 결합하여 하나의 'weighted-vector'를 만듦 >> Context Vector
4. Context Vector와 Decoder의 hidden state를 다시 결합해 최종 output 산출

>> 스코어값이 낮으면 중요하지 않은 정보, 높으면 중요한 정보로 인식



- Transformer 모델

: 기존의 RNN처럼 하나씩 처리하는게 아닌 한꺼번에 처리 / 기존의 Attention메커니즘을 사용하면서 학습과 병렬화(parallelize)가 쉽게 되게하는 모델

- 기존의 모델과 다른점

1. Trnasformer 내부에 'encoding component'와 'decoding component'가 따로 존재
2. 연결시키는 구조가 다름

- **Self-Attention**

● 'Encoding Component = stack of encoders'

- 인코더를 그냥 쌓은거임
- 이를 소개하는 논문에선 6개를 stack했고, 최적의 값은 아니다
- 한번에 모든 시퀀스를 사용해서 masking을 사용하지 않는 "Unmasked"

Input → Self-Attention → Feed Forward Neural Network를 거치는 2단구조

● 'Decoding Component = stack of decoders'

- 이 또한, 디코더를 그냥 쌓은거임
- 어쨌든 output은 순차적으로 내야하기 때문에 순서에 따라 masking하는 "Masked"

Masked Self-Attention → Encoder-Decoder Self-Attention → Feed Forward Neural Network를 거치는 3단구조

✓ RNN계열 모델에선 encoder / decoder가 하나씩 존재했지만 transformer는 다수의 encoder /decoder가 존재

● Transformer에서는 512개의 TOKEN을 사용하고 이 길이를 못채우면 Padding

Encoder Block

- Encoder

● 각각의 인코더는 구조적으로 전부 동일하다. (단, 구조적으로 같다고해서 weight를 share하는 것은 아님) / 첫 번째 encoder block과 두번째 encoder block의 가중치는 얼마든지 달라질 수 있다.

● 이 encoder block은 두 개의 하위계층(sub-layer)으로 구성되어있다.

① Self-attention Layer

- 한 단어의 정보를 처리할 때, 함께 주어진 input sequence의 다른 단어들을 얼마만큼 중요하게 볼것인가를 계산하는 layer

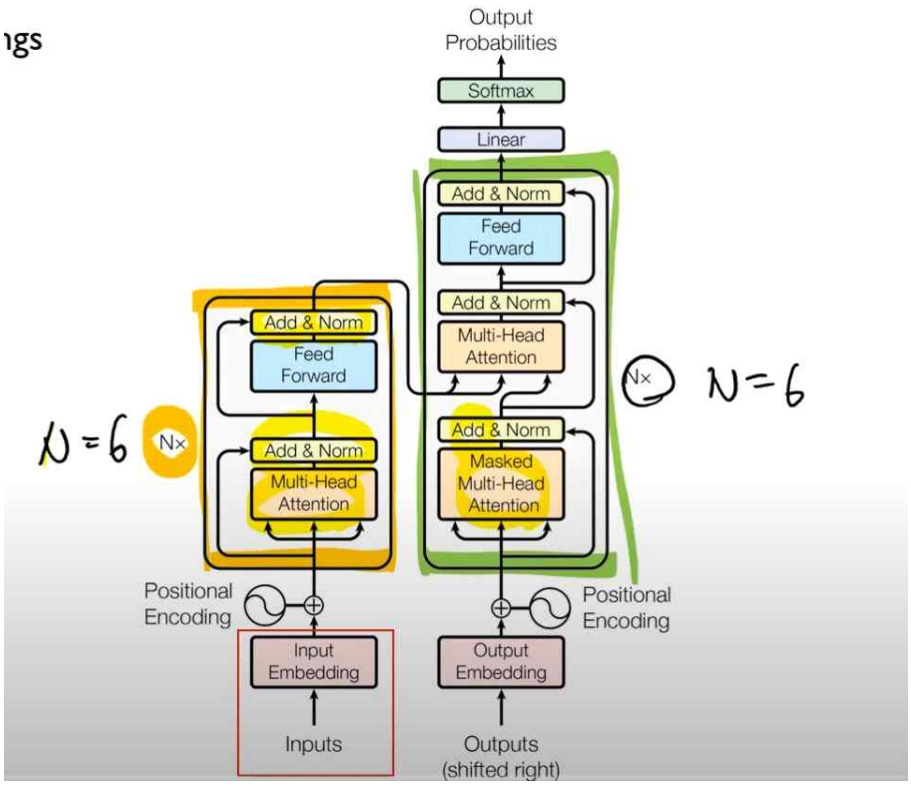
② FFNN(Feed Forward Neural Network)

- 매 input에 각각의 neural network를 독립적으로 적용하여 output값 산출

- Decoder

● 인코더와 다르게 하나의 sub-layer를 더 가지고 있는데 그것이 바로 '**Encoder-Decoder Attention**'

최종 output을 산출할 때, 인코더에서 주어지는 정보를 어떻게 반영할것인가를 결정



- Input Embedding

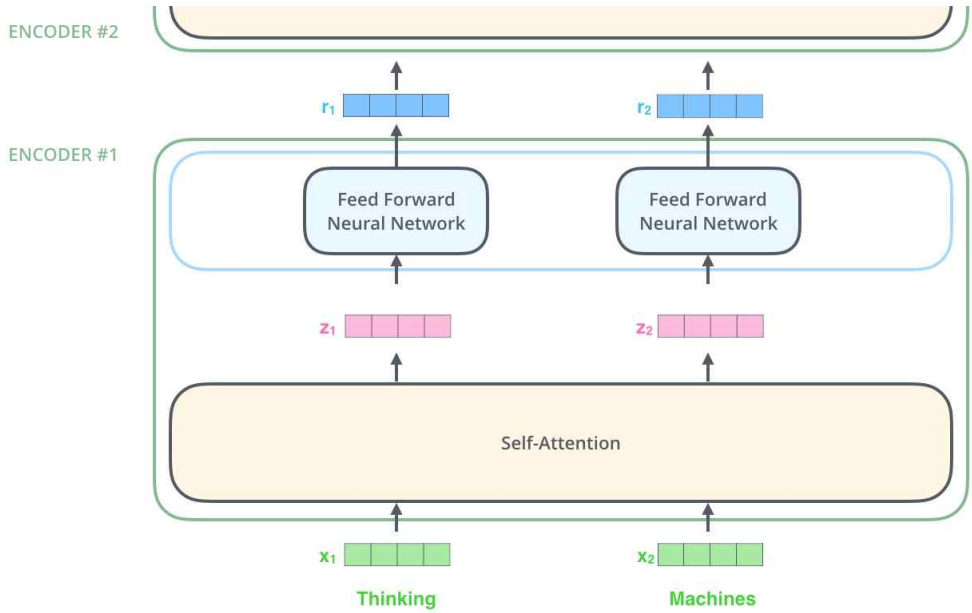
- 인코더의 최하단에서 사용된다. 밑단에 있는 인코더는 word embedding을 수행. 그 외의 인코더는 바로 아랫단의 인코더들의 output으로 받는다. 동시에 이들의 사이즈는 동일하게 유지된다
- 한 sequence의 길이를 최대 몇으로 가져갈까 결정하는 하이퍼파라미터이다
- 논문에선 512로 설정해줬다
- ex) 상위 90%에 해당하는 토큰의 개수로 설정, 학습데이터셋에서 가장 긴 문장의 길이로 설정

- Positional Encoding

- RNN은 하나의 sequence만을 입력으로 받기에, 위치정보를 그대로 저장하나, Transformer에선 모든 sequence를 한번에 입력으로 받기에, 위치정보가 손실되는 단점이 있다. 그 정보의 완전한 복원은 불가능하니 각각의 위치정보를 어느정도 복원해주자는 목적으로 만들어진게 **Positional Encoding**
- 위 그림에 나왔듯이, Input Embedding과 Positional Encoding을 더해준다. **concat의 개념 x, 그러므로 결과물도 같은 차원의 형태를 띌 것이다
- 목적 : Input Sequence의 단어 순서를 어느정도 반영해주자!
- Embedding + Positional Encoding = Embedding with Time Signal (between Emb and P.E with same dimension)
- 해당하는 Encoding vector의 크기는 동일해야한다
- 위치관계를 표현하고 싶은것이니, 두 단어의 거리가 Input sequence에 멀어지게되면 Positional

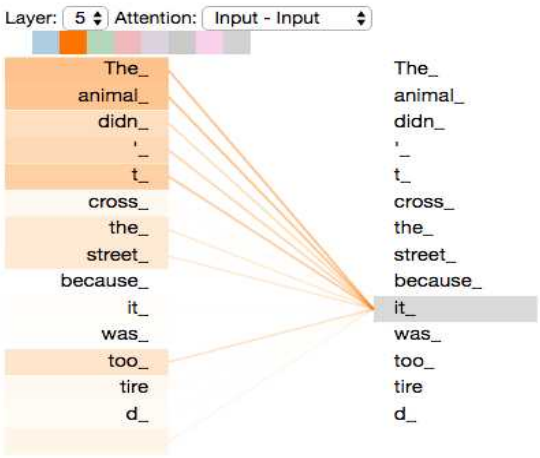
Encoding 사이의 거리도 멀어져야한다

- Self-Attention



● Self-Attention층에서 이 위치에 따른 path들 사이에 모두 dependency가 존재하는데, FFNN는 dependency가 없기에 FF layer 내의 다양한 path들은 병렬처리(parallelize)가 불가능하다
● FFNN의 weight들은 모두 동일.
● 위 그림의 r1과 r2는 첫 번째 인코더의 output인과 동시에 두 번째 인코더의 input값이다

● Input Sequence : 'The animal didn't cross the street because it was too tired.'
● 사람들은 "it"이 가리키는 정보가 앞에 나와있는 "street"와 "The animal" 중 "The animal" 이라는 것을 단번에 알 수 있지만, AI는 그것이 불가능하다.
● 그렇기에 각각의 단어들을 훑어가면서 it과 연관이 있는 단어는 무엇일까? 라는 질문에 답을 구하고자하는 것
● 모델이 입력 문장내의 각 단어를 처리해 나감에따라, self-attention은 입력문장내의 다른 위치에 있는 단어들을 보고 거기서 힌트를 받아 현재타겟위치의 단어를 더 잘 encoding 할 수 있다



- "it"이라는 단어를 encoding 할 때, attention 메커니즘은 입력의 여러 단어들 중에서 "the animal"이라는 단어에 집중하고 이 단어의 의미 중 일부를 "it"이라는 단어를 encoding 할 때 이용한다
- Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing

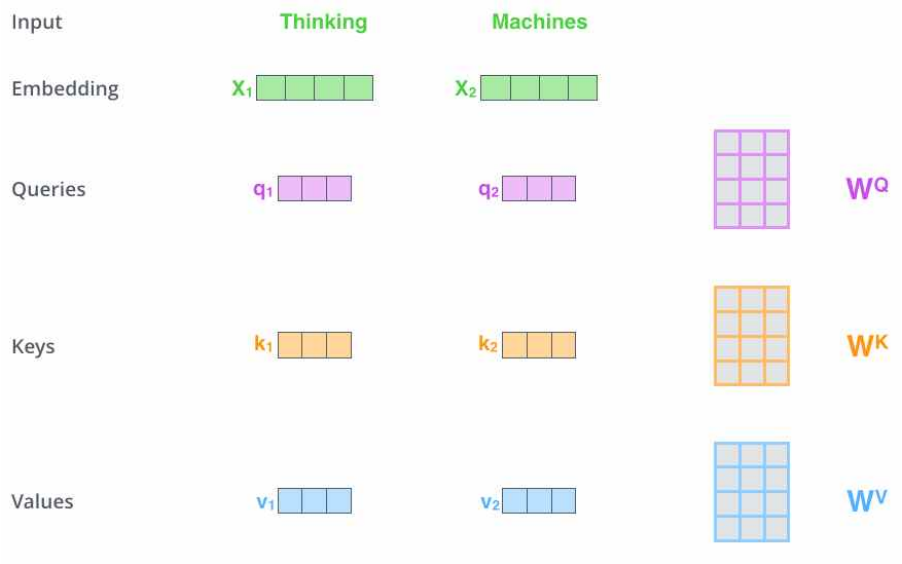
Self-attention Mechanism

- self-attention을 하기 위해 세 종류의 벡터를 만든다

1. Query : 현재 내가 보고 있는 단어의 representation, 다른 단어를 스코어링 하기 위한 기준값
2. Key : 모든 단어들에 대한 label과 같은 의미를 지닌다. Query가 주어졌을 때, Key값을 참고해 relevant한 단어를 추출해낸다
3. Value : 실제 단어가 나타내는 값 (actual word representation)

✓ 즉, Query와 Key를 통해 적절한 Value값을 찾아, 연산을 진행하겠다!

위 세 개의 값은 Input Embedding을 통해 만들어지며, 각각에 해당하는 matrix가 존재한다.



● $X_1 * W^Q = q_1$ (W^Q, W^K, W^V는 학습을 통해 찾아야하는 미지수)

Step 1 : 각각의 인코더 input으로부터 3개의 벡터(Q, K, V)를 만들어내자

▶ 일반적으로 Q, K, V의 차원은 인코더의 input/output보다 작게 만든다

▶ 굳이 더 작게 만들 필욘 없지만, multi-head attention의 연산에 있어서 유리하게 적용시키기 위해 대개 그렇게 만듦

Step 2 : 스코어값을 계산하자

▶ Q 벡터와 K 벡터 각각에 내적(dot-product)를 수행해 스코어값을 산출

▶ 단어와 Input sequence 속의 다른 모든 단어들에 대해서 각각 점수를 계산해야한다

▶ 이 점수는 현재 위치의 이 단어를 encode할 때, 다른 단어들에 대해서 얼마나 집중을 해야 할지를 결정한다

Step 3 : 스코어값을 $\sqrt{d_k}$ 로 나눠준다

▶ 논문에선 8로 나눠줬음 (d_k = 64)

▶ 이 과정은 gradient를 안정화시켜준다

Step 4 : softmax함수를 통해 결과값을 만들어준다

▶ softmax 함수를 통해 각각의 단어들은 매 위치에서 스코어값이 표현될 것 >> 매 위치에서의 중요도를 나타내는 값

▶ 이 결과값은 현재 위치의 단어의 encoding에 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.

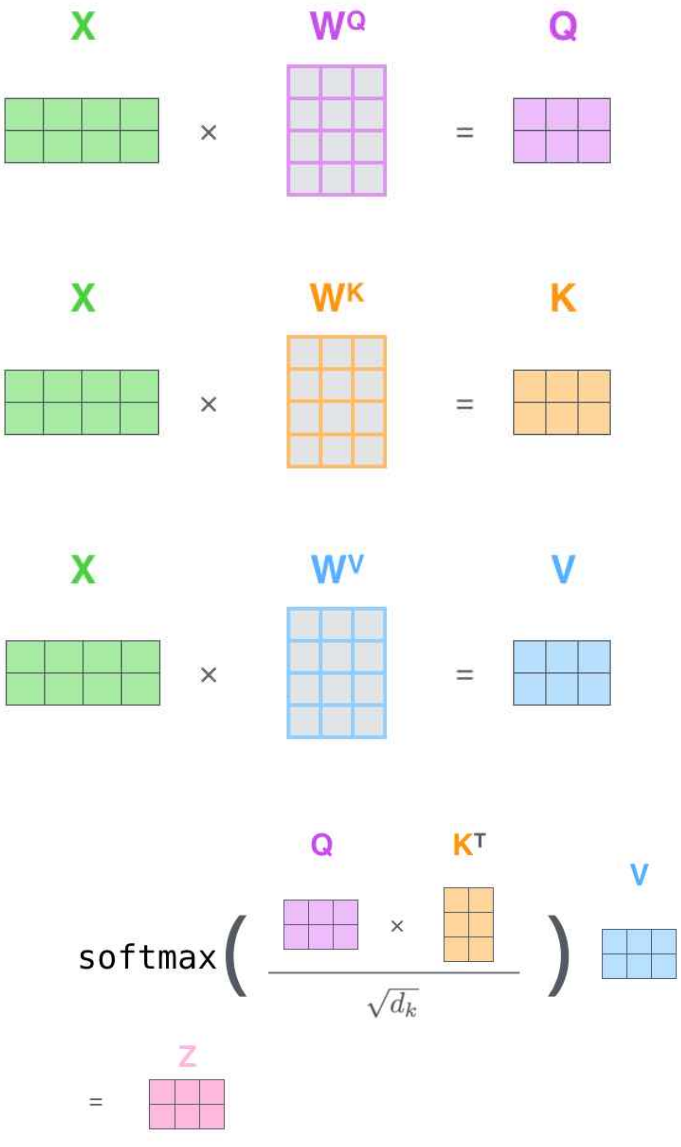
Step 5 : softmax함수를 통해 도출된 결과값과 V 벡터를 곱해주자

▶ 집중하고싶은 관련이 있는 단어들은 그대로 남겨두고, 관련이 없는 단어들은 0.001과 같은

작은 숫자(score)를 곱해 없애버리기 위함이다.

Step 6 : Step 5에서 나타난 값을 더해주자 ⇒ 현재 위치에 대한 self-attention layer의 출력

▶ softmax에 의해서 가중합이된 value값들을 self-attention layer의 output 값으로 사용



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

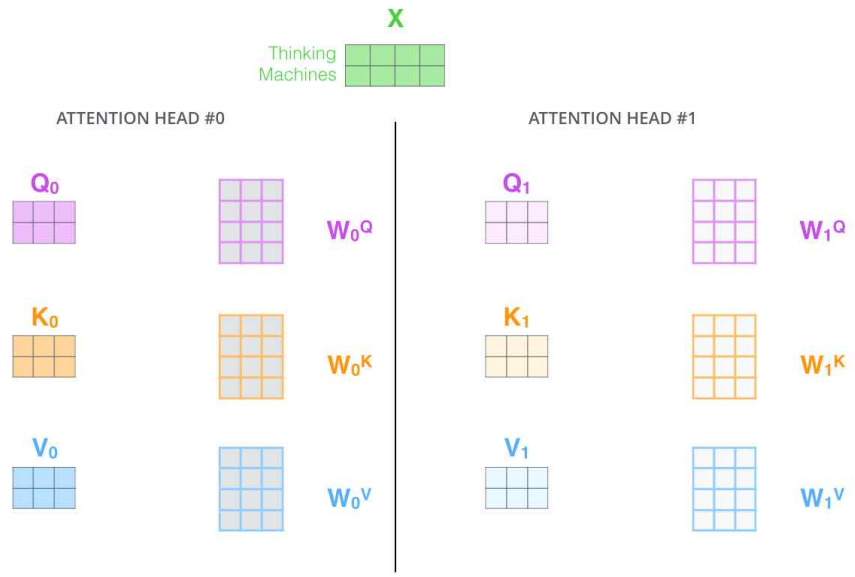
요약하자면 위 그림과 같다

- Multi-Head Attention

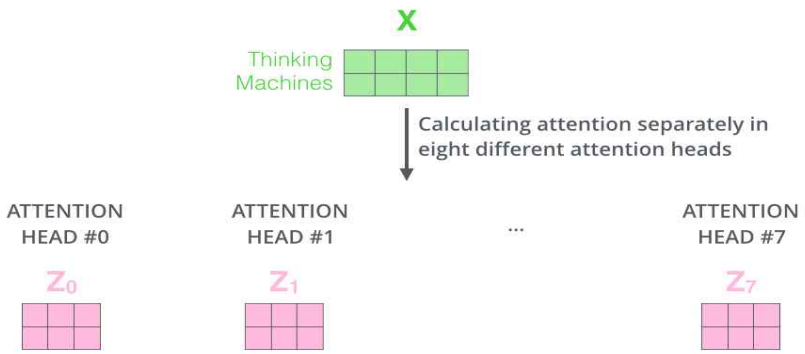
● 지금까지 본 것은 single-head attention. Multi-head Attention은 하나의 경우의 수가 아닌 여러

개의 경우의 수를 허용해 다수의 attention head 값(위 그림에선 Z 값)을 만들어내자 >> 여러개의 attention을 사용

- 모델이 다른 위치에 집중하는 능력을 확장시킨다. 위의 예시 문장을 번역할 때, “it”이 무엇을 가리키는지에 대해 알아낼 때 유용할 것이다
- attention layer가 여러개의 “representation 공간”을 가지게 해준다. multi-head attention을 이용함으로써 여러개의 Q, K, V weight 행렬들을 가지게된다. 이 각각의 Q, K, V set은 랜덤으로 초기화되어 학습된다. 학습이 된 후, 각각의 세트는 입력벡터들에 곱해져 벡터들을 각 목적에 맞게 투영시키게 된다. 이러한 세트가 여러개 있다는 것은 각 벡터들을 각각 다른 representation 공간으로 나타낸다는 것을 의미한다.

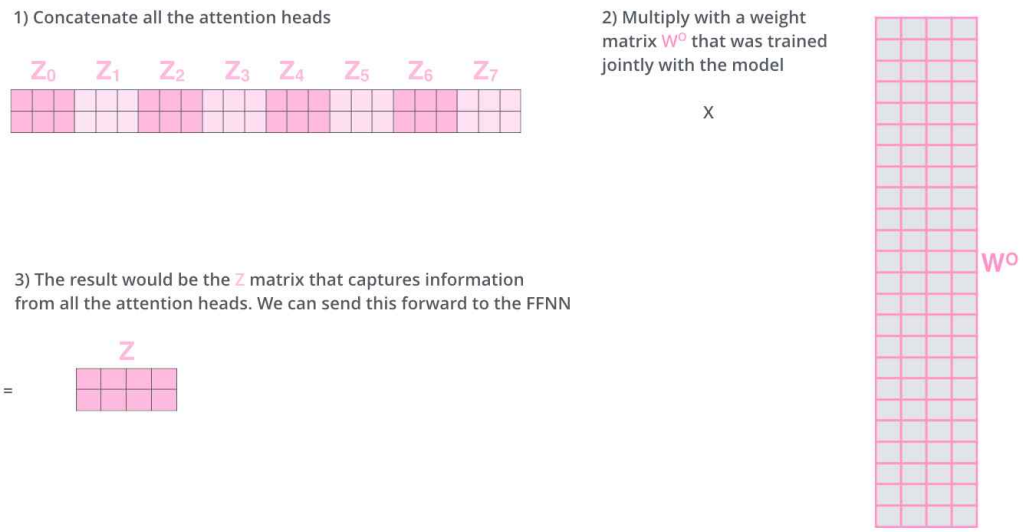


- multi-head atttention을 이용하기 위해서는 각각의 head를 위해서 각각의 다른 Q, K, V weight 행렬들을 모델에 가지게 된다. 벡터들의 모음인 행렬 X를 W^Q, W^K, W^V 행렬들로 곱해 각 head에 대한 Q, K, V 행렬을 생성한다



- self-attention 계산 과정을 8개의 다른 weight 행렬들에 대해 8번 거치게되면, 8개의 서로 다른 Z 행렬을 가지게 된다

● 그러나 문제는 이 8개의 행렬을 바로 FF layer로 보낼 수 없다. FF layer는 한 위치에 대해 오직 한 개의 행렬만을 input으로 받을수 있기 때문이다. 그렇기 때문에 이 8개의 행렬을 하나의 행렬로 합치는 방법을 고안해야한다. 일단 Z행렬을 하나의 행렬로 만들기 위해 concatenate하고 난 후, 하나의 또 다른 weight 행렬인 W^O을 곱하면 된다. 이 과정은 다음의 그림과 같다.

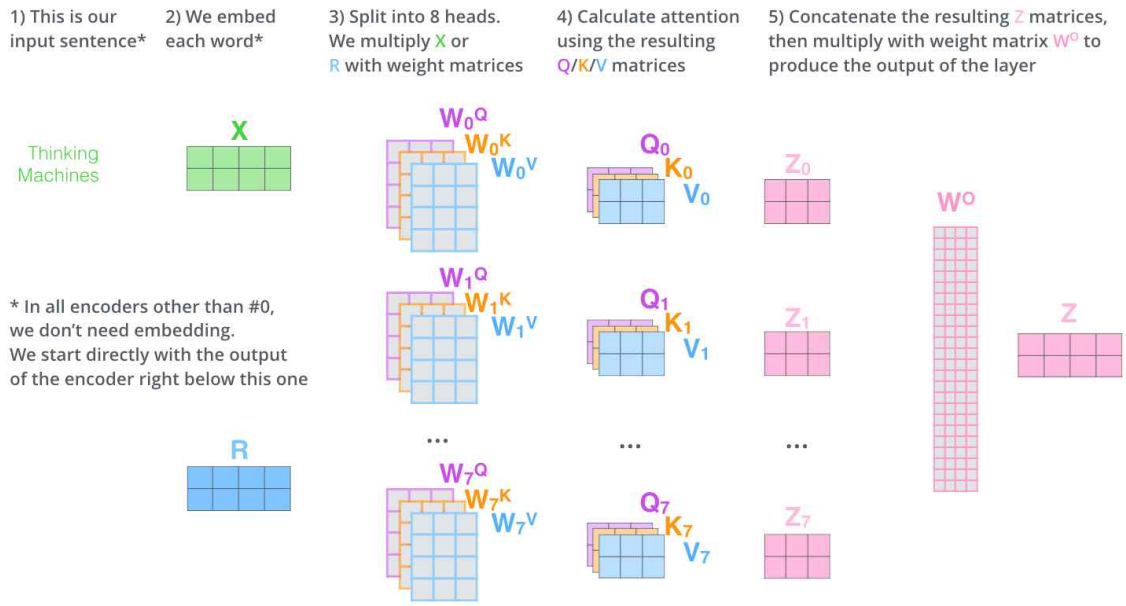


● Multi-Head Attention 과정 정리

Step 1 : 모든 attention head를 concat한다

Step 2 : (attention head와 동일한 len, input embedding len) 의 크기를 가지는 W^O를 생성하여 step 1의 값과 곱한다

Step 3 : 모든 attention head의 정보를 가지고 있는 결과값을 도출. 이는 최초 Input Embedding의 사이즈와 동일함. 또한, 이후 FFNN에 보내질 것이다

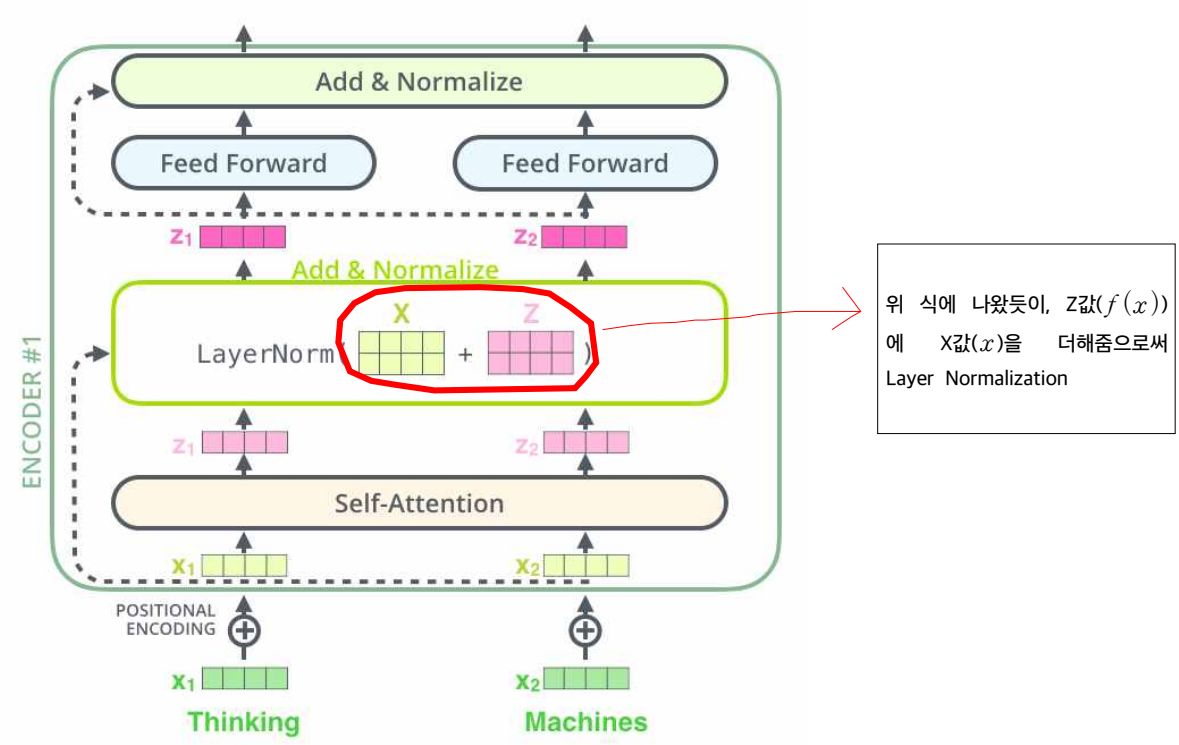


기존 인코더 인풋의 차원을 유지하면서 multi-head attention을 수행할 수 있음을 보여주는 그림

✓ self-attention 이후, Residual block을 더해주고 Normalization을 수행

- Residual Connection

- $f(x)+x$ 미분하면 $f'(x)+1$이라는 결과값이 나올테고 $f'(x)$값이 매우 작은 값일지라도 " +1 " 이라는 상수를 통해 gradient가 최소 1만큼의 값을 나타내기 때문에, 학습에 굉장히 유리하게 작용할 것
  - ▶ $f(x)+x$ 에서 $f(x)$는 self-attention을 거친 output값이고, x는 Input 값
- 이 과정을 시각화해보자면 다음과 같을 것이다

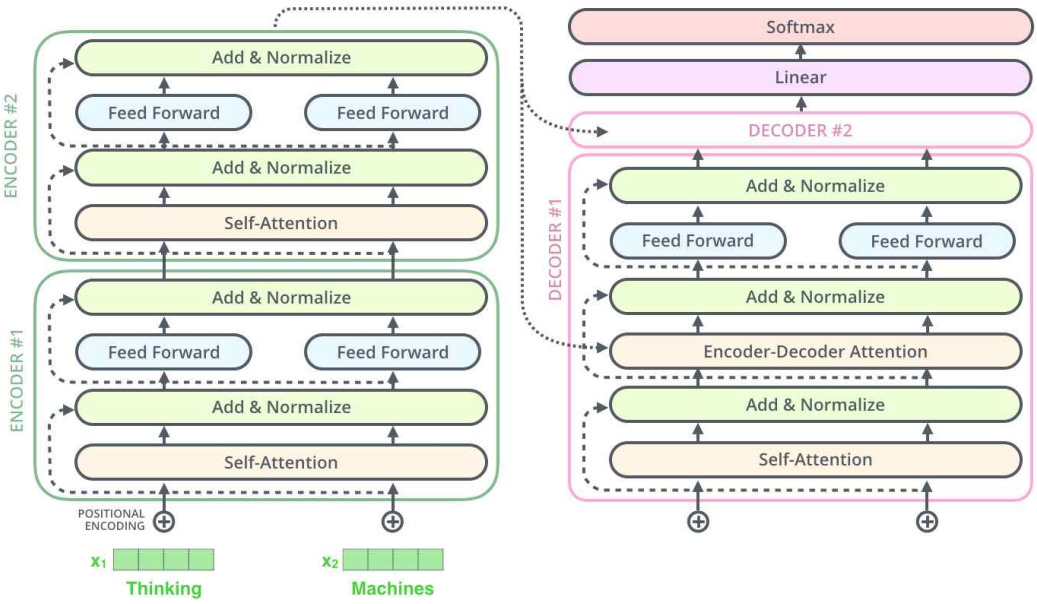


- Layer Normalization

● 해당 논문에선 Layer Normalization 사용

✓ 이후의 output값은 FFNN에 보내짐

✓ 위 2개의 과정은 Encoder, Decoder에서 default값으로 자주 사용



- Position-wise FFN

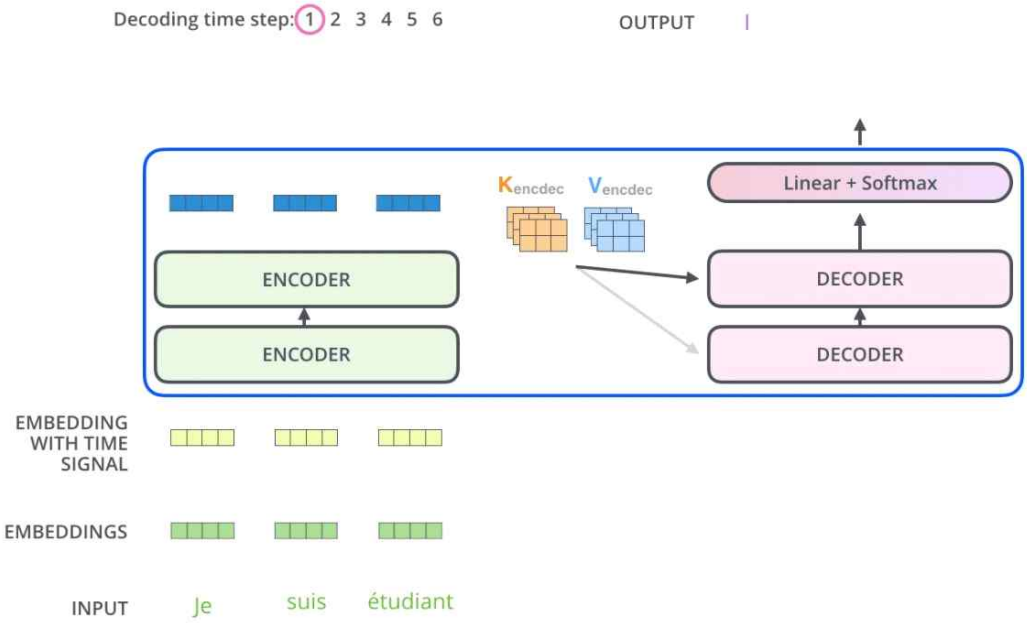
- Fully-connected feed-forward network
- 각각의 위치에 대해 개별적으로 적용

$$FFN(x) = \max(0, x W_1 + b_1) W_2 + b_2$$

▶ ReLU 함수 적용

- 각각의 layer마다 서로 다른 파라미터값을 사용한다

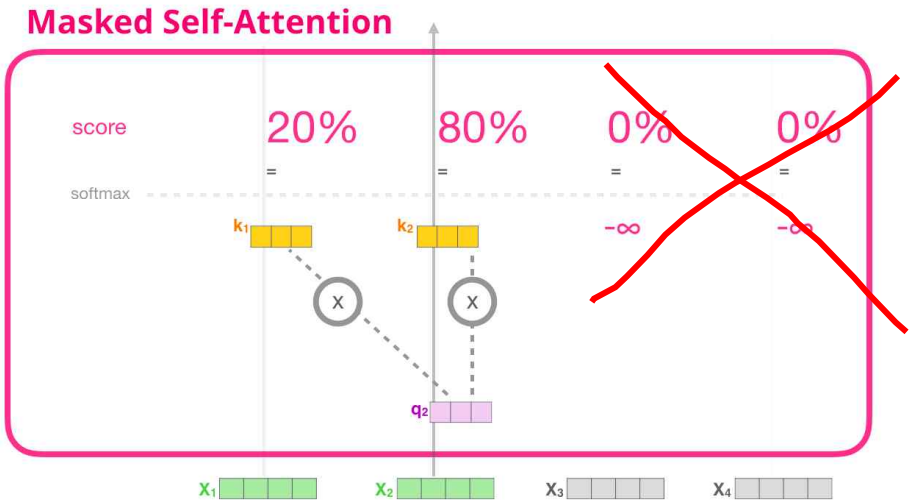
Decoder Block



- 디코더가 출력을 생성할 때 다음 출력에서 정보를 얻는 것을 방지하기 위해 masking을 사용한다. 이는 I번째 원소를 생성할 때는 1 ~ I-1 번째 원소만 참조할 수 있도록 하는 것

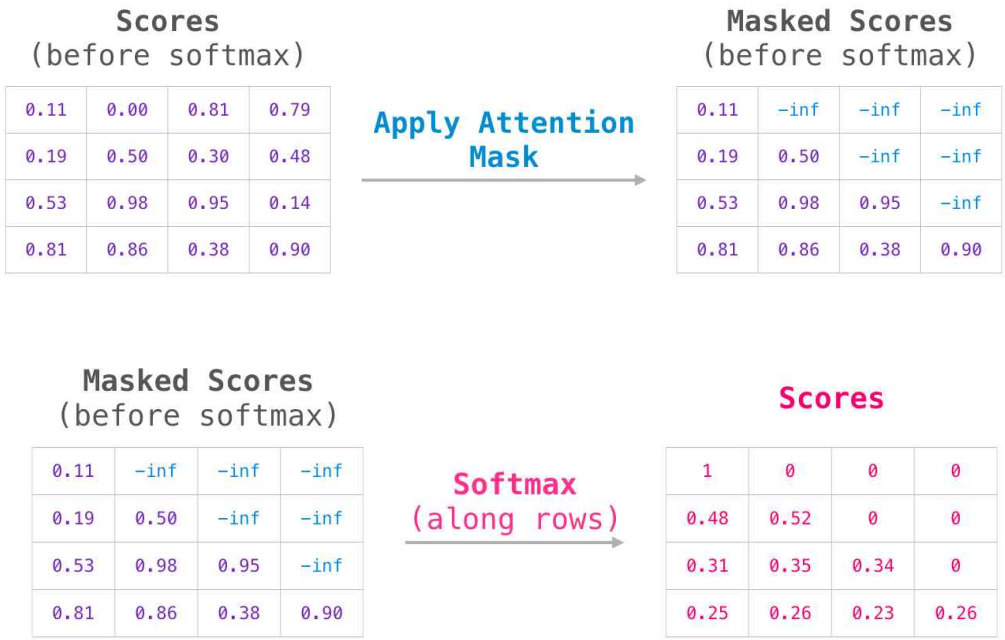
- Masked Multi-head Attention

- encoder에선 스코어값이 Input sequence모두에 대해 나타났지만, decoder에선 오직 전 포지션에서의 output값만을 가져와야하기 때문에, 다른 Input sequence에 대한 어텐션 스코어값을 $-\infty$ 값으로 masking해준다. 이 값들은 이후 softmax함수를 거치게 되면 어텐션 스코어 값은 0을 가지게 될 것이고, 오직 자기 자신의 스코어값 또는 그보다 앞에 해당하는 어텐션 스코어 값만을 사용하게 될 것



- 자신보다 먼저 들어온 sequence에 해당하는 K, V 값만을 사용 할 수 있고, 그 이후의 값들은

— ∞로 masking처리



- Multi-Head Attention

- Encoder로부터 전달받은 결과값과의 연산
- Encoder의 K, V 벡터값이 Decoder와의 attention score를 계산할 때, 영향을 준다

✓ Transformer의 attention은 총 3단계에 걸쳐 진행된다

① encoder의 self-attention
② decoder의 masked self-attention
③ encoder의 output과 decoder 사이의 attention = “Encoder-Decoder Attention”

- Final Linear and Softmax layer

- Linear Layer : 단순한 FC Neural Network로 decoder가 마지막으로 출력한 벡터를 그보다 훨씬 더 큰 사이즈의 벡터인 logits 벡터로 투영시킨다
- Softmax Layer : Linear Layer를 거친 점수들을 확률로 변환해주는 역할을 한다. 당연하게도 가장 높은 확률 값을 가지는 셀에 해당하는 단어가 Output으로 출력된다

✓ decoder의 FFNN을 거친 값 통과 > Linear layer에서 vocab_size만큼의 실수값 도출 > softmax 함수를 취한 후, 최종 output값 산출

Transformer 의의

- Attention 메커니즘만을 사용하는 Transformer라는 새로운 구조를 제안

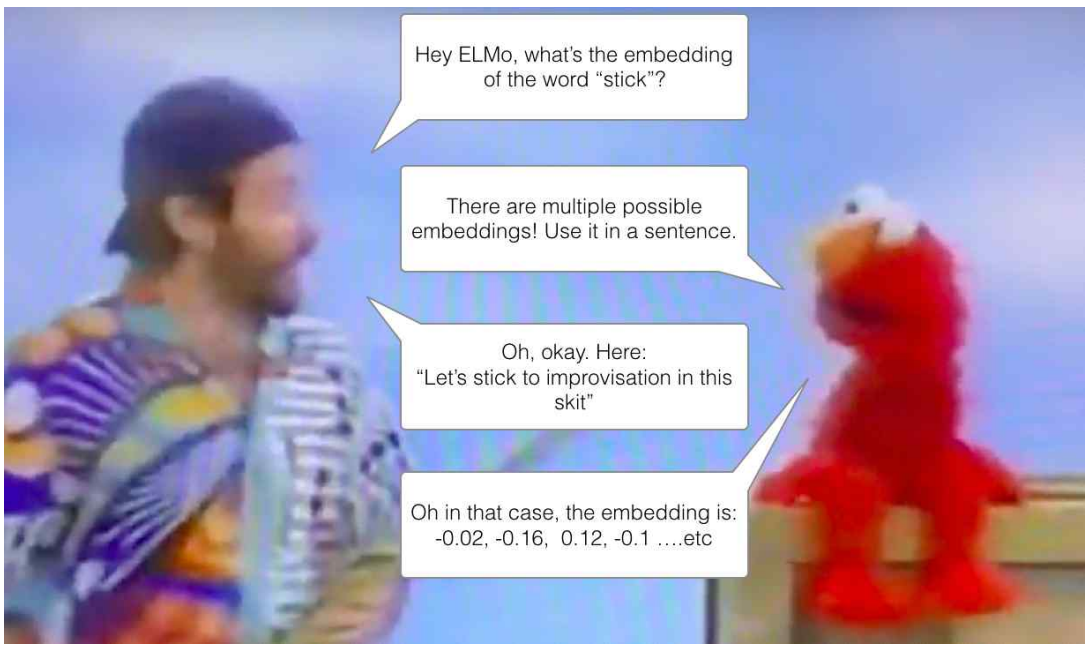
1. 기계번역 Task에서 매우 좋은 성능
2. 학습 시, 우수한 병렬화(Parallelizable) 및 훨씬 더 적은 시간 소요
3. 구문분석(Constituency Parsing) 분야에서도 우수한 성능 > 일반화도 잘됨

- ELMO (Embeddings from Language Models)

- 기존 word2vec, Glove 등에 비해 context 정보를 반영하는 향상된 Pre-training 방법을 제시한다. 여러층의 BiLSTM을 이용해 Language Model을 하며, 만들어진 hidden state를 적절하게 조합해 새로운 word representation을 만들어낸다.
- 사전 훈련된 모델을 사용한다
    - ▶ 사전훈련과 문맥을 고려하는 문맥반영 언어모델Contextualized LM이다.
- representation은 문장 내 각 token이 전체입력 시퀀스의 함수인 representation을 할당받는다는 점에서 전통적인 단어임베딩과 다르다. 이를 위해 이어붙여진 LM으로 학습된 BiLSTM으로부터 얻은 vector를 사용한다.
    - ▶ 복잡한 형태를 가지는 데이터(word)를 모델링
    - ▶ 다양성을 가지는 언어(동음이의어=polysemy) catch 가능

✓ 다소 복잡한 특징을 가지는 데이터에 대해 모델링을 하고, polysemy와 같은 다양성에 대해 나타나는 언어들에 대해 각각 다른 Embedding Vector를 구성가능하다. 즉, 단어를 임베딩하기 전에 전체 문장을 고려해서 임베딩을 하겠다.

- Context Matters



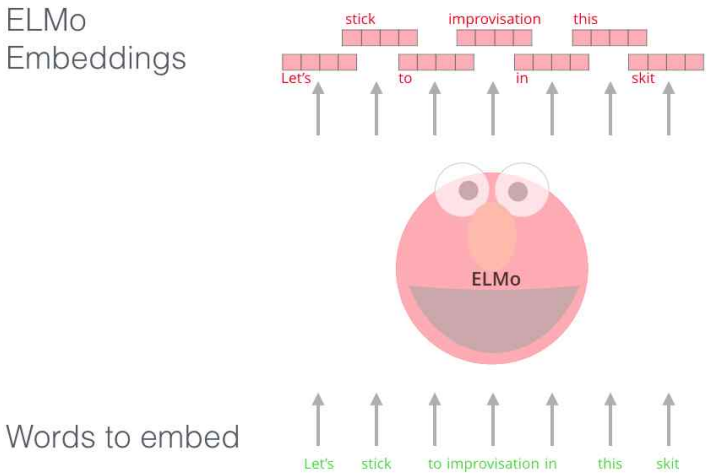
● "눈" 이란 단어에 대해 2가지 해석이 가능하다. 첫 번째, 신체기관 중 하나인 eye의 뜻을 가지는

눈. 두 번째, 겨울철 나타나는 기상현상 snow의 뜻을 가지는 눈. ELMo는 한 단어가 가지는 다른 뜻에 대한 정보를 모델링 가능하다. 그 후, contextualized word-embedding vector가 도출될 것이다.

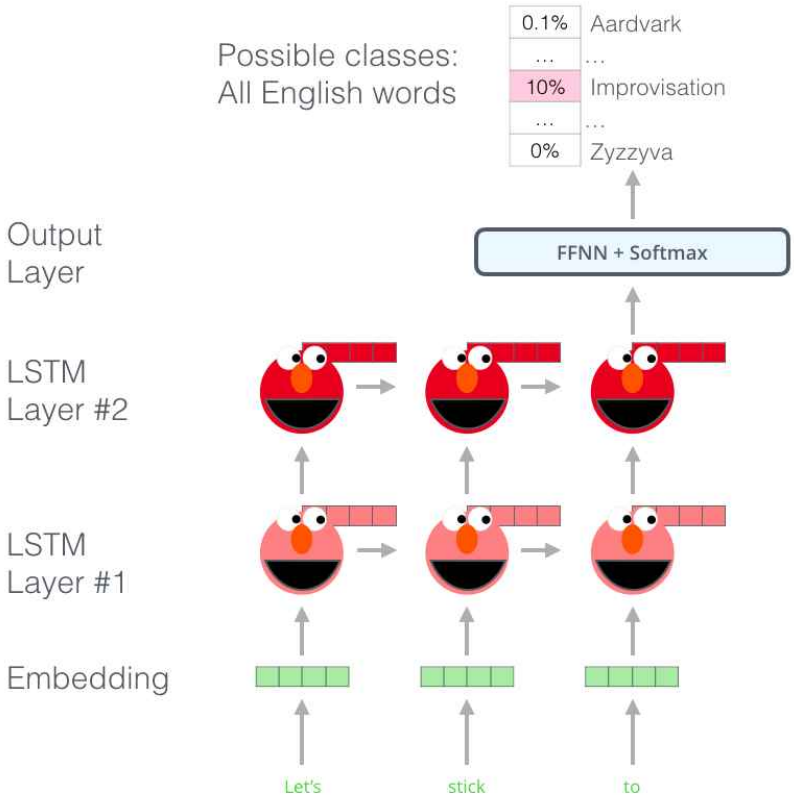
- contextualized word-embedding은 단어가 가지는 각기 다른 의미에 대해 embedding 할 것이고, 이는 context의 문장에 전달될 것이다.

- 특징

- 각각의 토큰은 representation을 할당받는데, 이는 전체의 입력 문장의 함수다.
- BiLSTM을 사용해 언어모델을 학습시켰다. 이를 통해, 도출되는 내부 레이어에서의 hidden vector들을 결합하는 방식을 차용 / 기존의 방법들과는 다르게 Top level의 LSTM정보만을 이용하지 않고, 각 level의 hidden state들을 조합해 새로운 단어 representation을 만들어낸다.



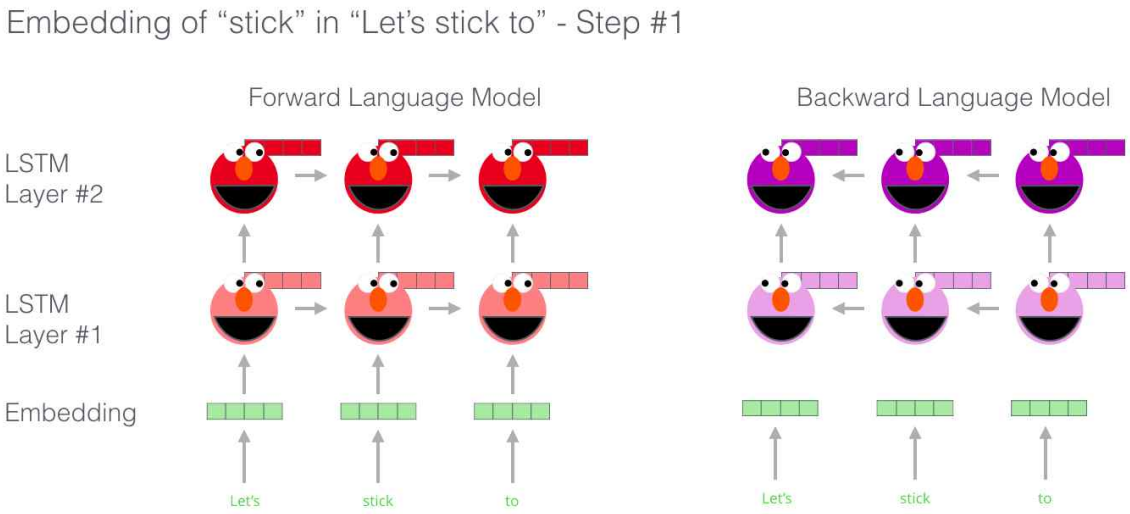
- 단어 각각에 ELMo를 적용시켜 Embedded Vector를 산출



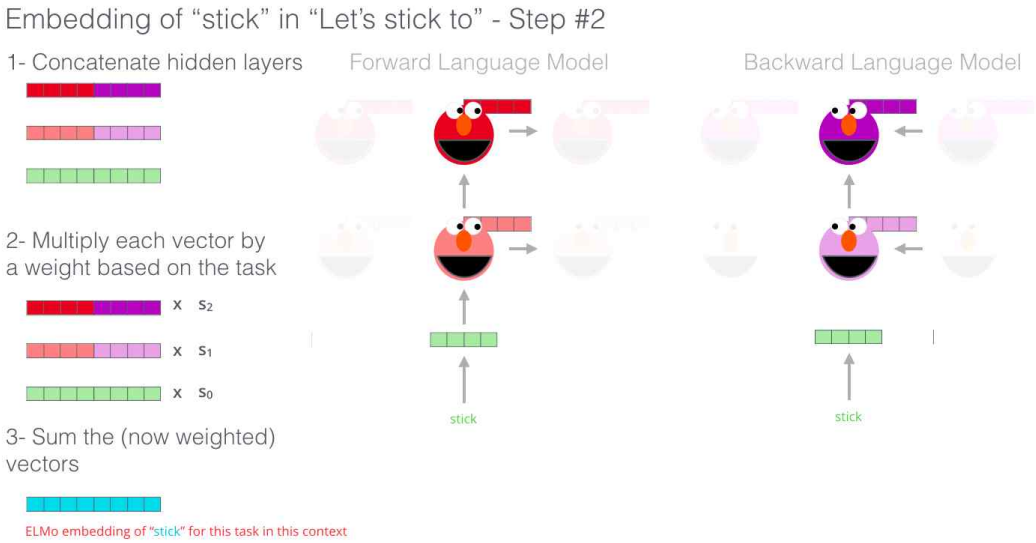
● A step in the pre-training process of ELMo: Given "Let's stick to" as input, predict the next most likely word? a language modeling task. When trained on a large dataset, the model starts to pick up on language patterns. It's unlikely it'll accurately guess the next word in this example. More realistically, after a word such as "hang", it will assign a higher probability to a word like "out" (to spell "hang out") than to "camera".

● ELMo는 기본적으로 sequence의 단어속에서 다음 단어를 예측하는 Language Model이다. 각각의 입력 임베딩벡터는 2개의 LSTM layer를 거쳐 10%의 확률값을 가지는 'Improvisation'을 최종 예측

● ELMo는 LM이 다음단어의 정보를 가지고있지 않기 때문에 step을 통해 BiLSTM을 학습시킬 것이다.



- ELMo는 Bi-directional LSTM을 통해 학습하기 때문에, Forward L.M, Backward L.M 둘 다 학습
  - ▶ Forward L.M : 앞방향, 순차적으로 학습시키는 모델
  - ▶ Backward L.M : 뒤쪽 단어부터 역방으로 학습시키는 모델
- 그룹화시킨 hidden state를 통해 문맥을 반영한 embedding을 제시한다. 이는 가중합을 위해 concat할 예정.



- 두 가지 L.M을 통해 나타난 ELMo의 Vector
  - ▶ 두 모델에서의 시점이 같아야 함
  - ▶ Concatenate hidden layers : 동일한 level에 있는 embedding 값(=hidden state 값)을 concat
  - ▶ task에 대해서 적절한 가중합(그림의 s_0, s_1, s_2)을 계산하여 최종 embedding 값을 도출

- layer를 어떻게 weighting 시킬까?
  - ▶ task에 맞게 적절한 조정
  - ▶ 또는, 모두 똑같은 가중치로 설정

- 실험결과

	Source	Nearest Neighbors
GloVe	play	playing, game, games, played, players, plays, player, Play, football, multiplayer
biLM	Chico Ruiz made a spectacular play on Alusik 's grounder {...}	Kieffer , the only junior in the group , was commended for his ability to hit in the clutch , as well as his all-round excellent play .
	Olivia De Havilland signed to do a Broadway play for Garson {...}	{...} they were actors who had been handed fat roles in a successful play , and had talent enough to fill the roles competently , with nice understatement .

Table 4: Nearest neighbors to "play" using GloVe and the context embeddings from a biLM.

- Glove를 통해 얻은 “play”에 대한 벡터와 ELMo를 통해 얻은 “play”에 대한 벡터를 비교한 표이다. Glove는 단어의 representation이 고정되어 있기 때문에 play처럼 여러 뜻을 가지고 있는 단어의 경우 여러 의미를 단어 representation에 담기 어렵다.
- 하지만, ELMo의 경우, LSTM의 이전 step에 입력된 단어에 따라 현재단어에 해당하는 projection vector들이 변하기 때문에, context에 따라 얻어지는 “play”에 대한 벡터가 다르다. 첫 번째 source 문장을 통해 얻어진 “play”의 벡터는 운동경기에 대한 의미를 나타내어야하는데, nearest neighbor를 구해봤을 때, 두 “play”의 의미가 비슷하다는 것을 알 수 있다. 반면, 두 번째 source문장은 연극에 대한 의미를 반영하는 것을 볼 수 있다.

- GPT

- BERT

- GPT-1의 트랜스포머 디코더를 사용한 자연어 처리능력은 문장을 처리하는데 부족함이 있을 수 있다라는 문제를 제기한다. 더불어 질의 및 응답영역은 문맥이외의 능력이 상당히 중요한데, 단순히 왼쪽

에서 오른쪽으로 읽어나가는 방식은 문맥이외의 약점이 있을 수 있다고 지적한다.
- 이러면서 단순히 왼->오의 방향으로 읽어나가는 디코더보다 양방향으로 문맥을 이해할 수 있는 인코더를 활용한 LM을 BERT라는 이름으로 발표한다.

- 인코더는 ①모든 토큰을 한번에 계산하며 ②왼쪽에서 오른쪽으로 하나씩 읽어가는 과정이 없음
- 디코더는 왼쪽부터 오른쪽으로 순차적으로 출력값을 생성한다, 이전 생성된 디코더의 출력값과 인코더의 출력값을 사용하여 현재의 출력값을 생성한다.

- BERT는 양방향으로 학습되는 기존 LM과 조금 다른 형태이다
- BERT는 동일한 문장을 그대로 학습하되 가려진 단어를 예측하도록 학습된다. 여기서 가려진 단어는 “Mask Token”이라고 불린다.
- 사람이 직접 labeling 할 필요가 없다. 단순히, 문장 속 단어만 가려주고 가려진 단어를 맞추도록 학습하면 되는 것
- BERT는 한 문장뿐만 아니라 두 개의 문장을 입력값으로 받을 수 있다. 예를 들면, 입력값으로 질문과 정답이 들어있는 문맥을 받아 정답을 출력하는 형태가 될 수 있겠다. BERT는 한문장 또는 두문장의 학습 데이터를 통해 token간의 상관관계 뿐만 아니라 문장 간의 상관관계도 학습하게된다.
    - 한 문장 : <CLS>
    - 두 문장을 구별해주는 Special Token : <SEP>

▶ 구조

1. Pre-train (입력값)

- 입력토큰들은 Positional Encoding, Segment Embedding 등과 더해진다.
- **“WordPiece Embedding”** : 문장을 토큰단위로 분리한다. 이는 단순히 띄어쓰기로 토큰을 나누는것보다 효과적으로 토큰을 구분한다.

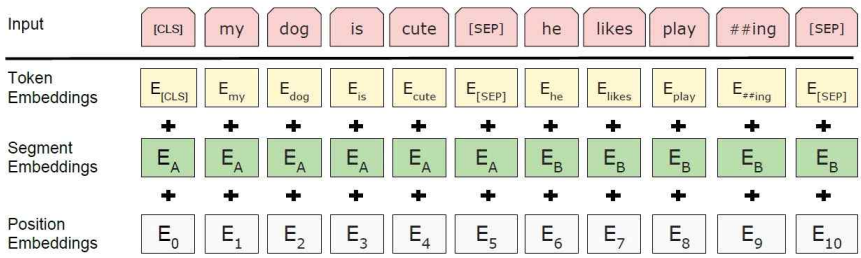


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

- “play”라는 단어는 1개이지만, “play”와 “playing”으로 토큰이 나뉘었다.
- 장점1. ‘play’와 ‘playing’, 두 단어는 뜻이 명확하게 다르다. 이러한 뜻을 모델에 명확히 전달 가능.
- 장점2. 위와같이 단어를 split해서 토큰을 나눌 때, 신조어 또는 오탈자가 있는 입력값에 대한 예측이 향상 될 수 있다. (dictionary에 단어를 추가해주는 느낌)
- **“Segment Embeddings”** : 두 개의 문장이 입력될 경우, 각각의 문장에 서로 다른 숫자들을 더해주

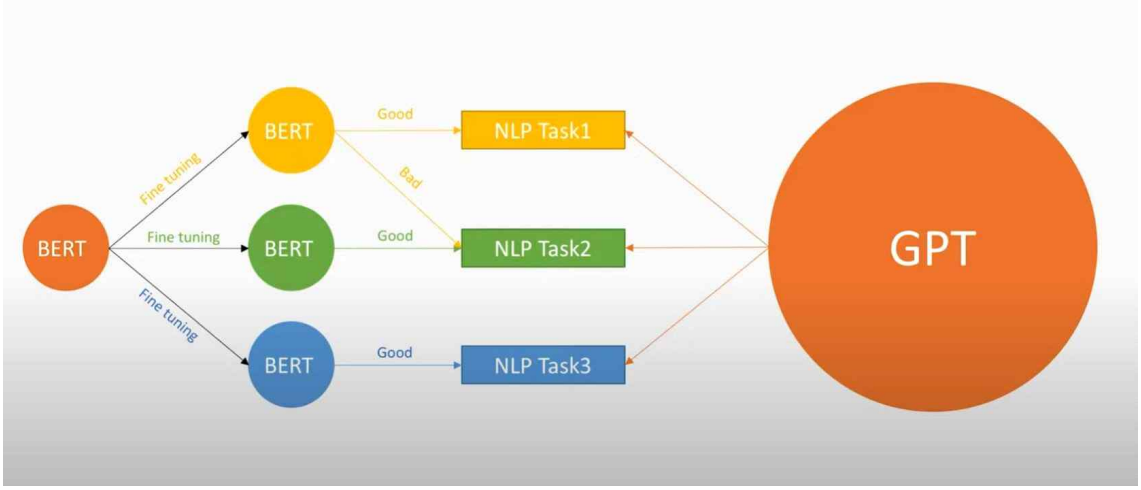
는 것. 모델에게 두 개의 문장이 있다는 것을 쉽게 알려주기위해 사용되는 임베딩이다.

- **"Positional Embeddings"** : 토큰들의 상대적 위치정보를 알려준다. 모델은 Positonal Embedding을 통해, E_1 다음에 E_2가 있음을 알 수 있다. 더불어서 sin, cos함수를 사용하는데 3가지 이유가있다. ① sin과 cos의 출력값은 입력값에 따라 달라진다. 따라서 두 함수는 입력값의 상대적인 위치를 알 수 있는 숫자로 사용이 가능하다. ② 두 함수의 출력값은 규칙적으로 증가 또는 감소한다. 따라서 모델이 이 규칙을 사용해 입력값의 상대적 위치를 쉽게 계산 가능하다. ③ 두 함수는 무한대 길이를 가지는 입력값에 대해서도 상대적인 위치값을 출력가능하다. 어떤 위치의 입력값이라도 -1 ~ 1 사이의 값을 출력하게 되어있다.

※ 왜 BERT는 절대적 위치정보값이 아닌 상대적 위치정보값을 계산하는 P.E를 채택한걸까? 만약, 절대적 위치정보값을 사용할 경우에는 최장 길이 문장을 세팅해야한다. 즉, 학습시에 사용했던 최장길이의 문장보다 더 큰 문장을 받을 수가 없게 되는 것! 그렇기 때문에 상대적 위치정보값이 P.E에서 더 선호된다.

2. Fine-tunning

- BERT는 **양방향 LM**이며 **fine-tunning을 하기 위해 만들어졌다**.



- 그림과같이 GPT는 Pre-train된 모델을 통해 task에 바로 적용이 가능하지만 모델이 상당히 크다는 것을 알 수 있다. 반면에, BERT는 모델이 상대적으로 작으며 각각의 다른 task를 수행하기위해 따로 Fine-tunning이 필요하다. 당연하게도, 각 목적에 맞게 만들어진 모델은 해당 목적과 부합하는 학습이 불가능하다. 또한 GPT에 비해 적은 시간과 돈이 들지만, fine-tunning을 직접 해줘야한다.

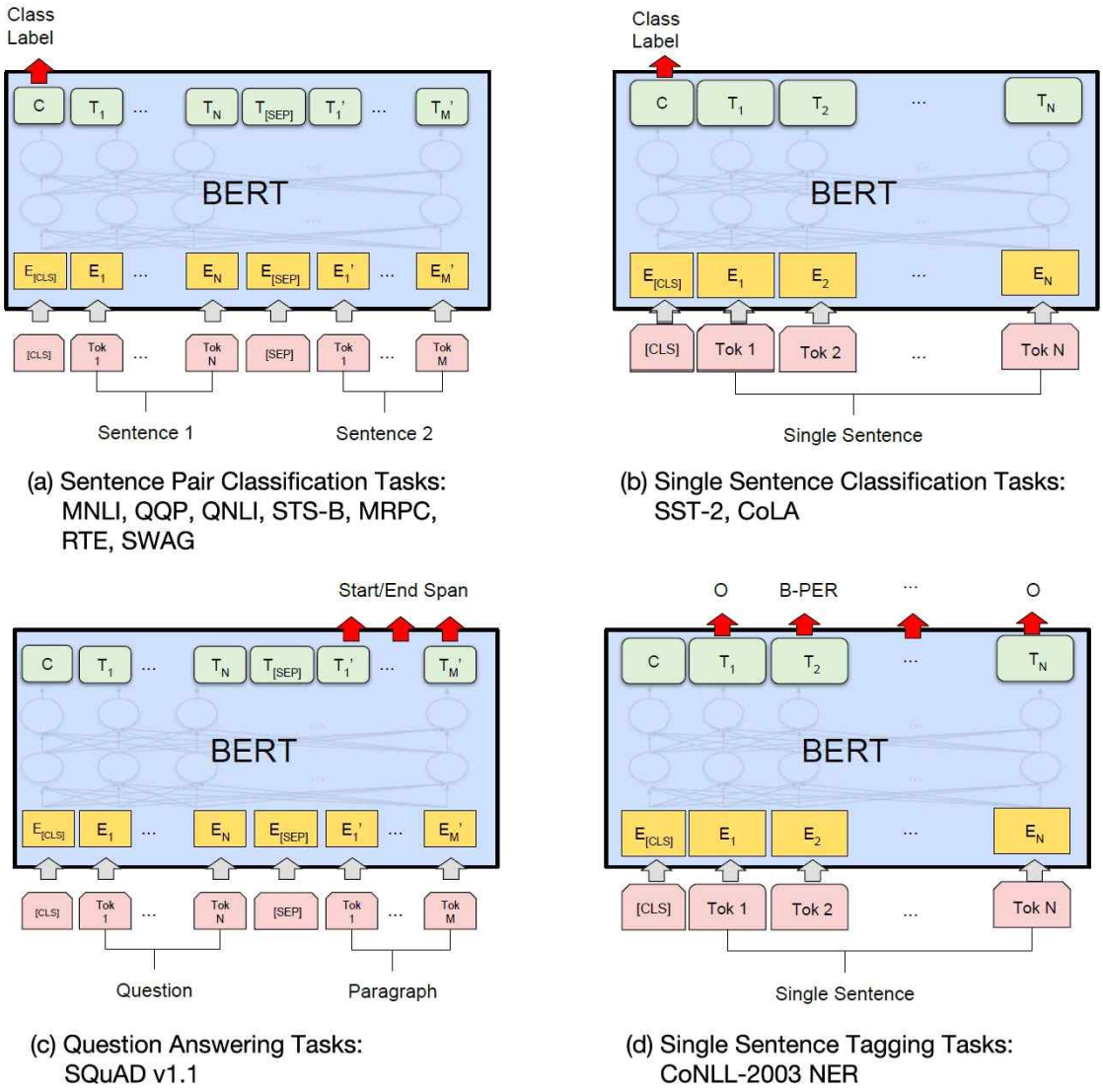


Figure 4: Illustrations of Fine-tuning BERT on Different Tasks.

1. 그림 a는 두 문장의 관계를 예측하기 위해 모델을 만드는 방법이다. 2개의 문장을 <SEP> 토큰으로 구분해 BERT에 입력하여 출력값의 첫 번째 <CLS> 토큰을 두 문장의 관계를 나타내도록 학습시킨다.
2. 그림 b는 문장을 분류하는 모델이다. 1개의 문장을 입력으로 받고 <CLS> 토큰이 분류값 중 하나가 되도록 학습시킨다.
3. 그림 c는 Q&A (지배 및 응답) 모델이다. 입력값으로 질문과 정답이 포함된 장문을 <SEP> 토큰으로 구분하여 전달한다. 그리고 BERT 출력값의 마지막 토큰들이 장문 속에 위치한 정답의 시작인덱스와 마지막인덱스를 출력하도록 학습시킨다.
4. 그림 d는 문장속 단어를 태깅하는 모델이다. 각각의 입력토큰에 대한 출력값이 있기 때문에 이 출력값이 원하는 태깅으로 출력되도록 학습시킨다.