

자연어처리팀

2팀

한준호

윤세인

김나현

윤여원

권능주

INDEX

1. Attention
2. 트랜스포머
3. 트랜스포머 기반 사전 학습 모델
4. Appendix

1

Attention

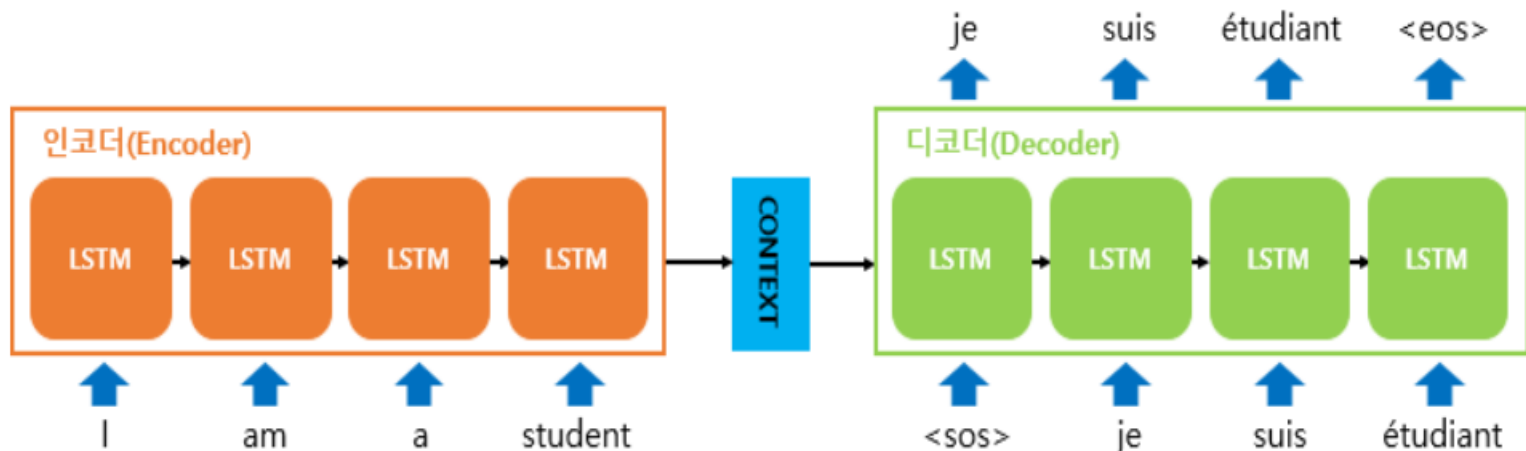
1

Attention

seq2seq

seq2seq

NMT에서 대표적으로 사용되는 Encoder-Decoder 모델



seq2seq의 한계점



고정된 길이의 context vector

정보의 손실 불가피



기울기 소실 문제

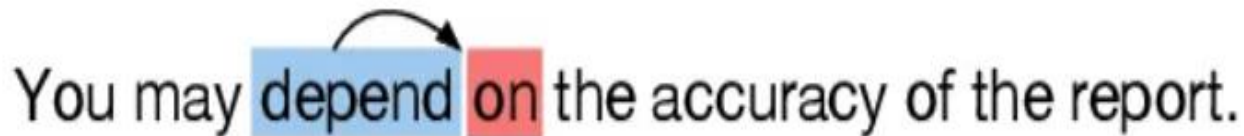
입력 시퀀스의 길이가 길수록 출력 시퀀스의 정확도 떨어짐

1

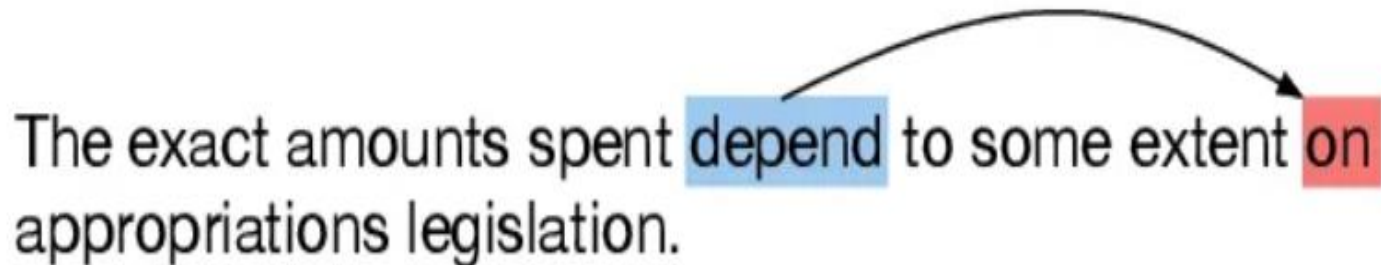
Attention

seq2seq의 한계점 예시

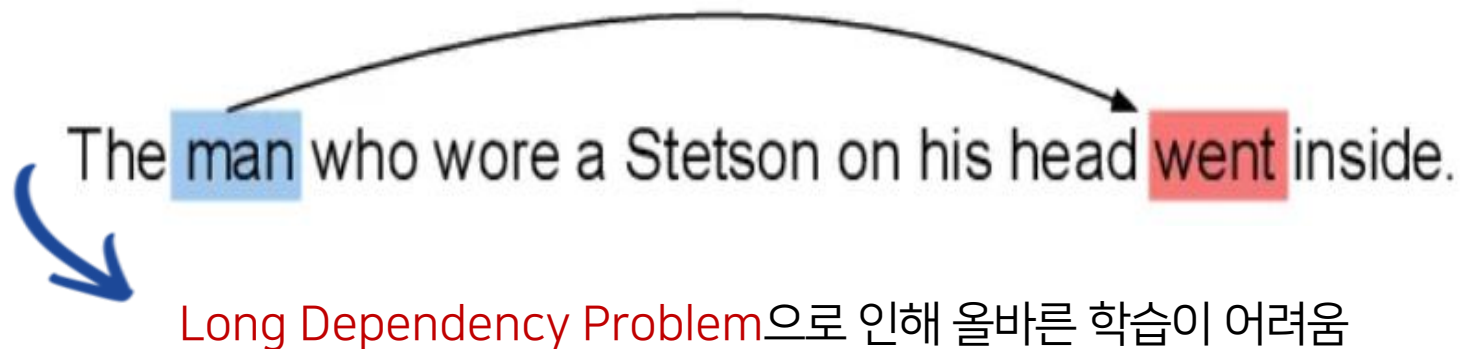
You may depend on the accuracy of the report.



The exact amounts spent depend to some extent on appropriations legislation.



The man who wore a Stetson on his head went inside.



Long Dependency Problem으로 인해 올바른 학습이 어려움

1

Attention

seq2seq의 한계점 예시



이를 극복하기 위해 **Attention**이 등장함

Long Dependency Problem으로 인해 올바른 학습이 어려움

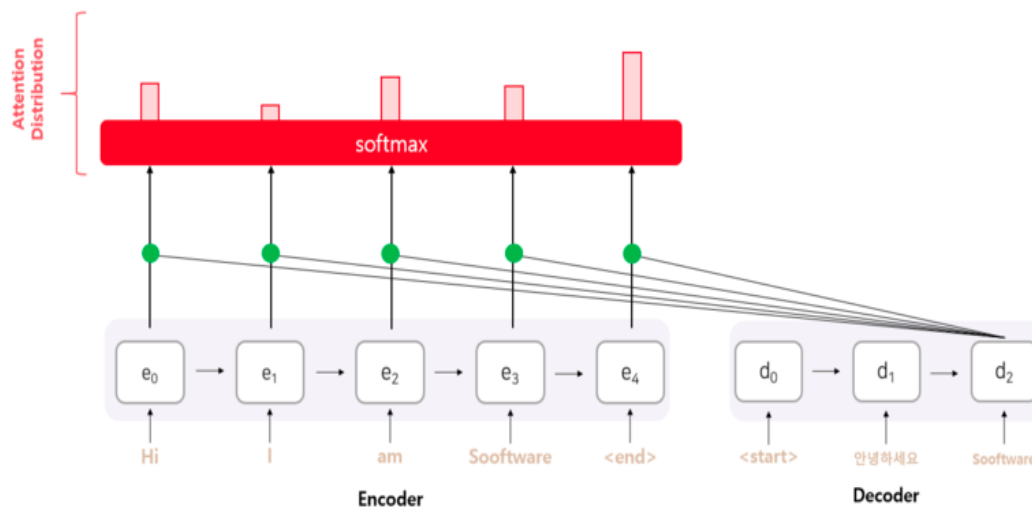
1

Attention

Attention

Attention

입력 시퀀스의 **특정한 부분에 집중**하는 방법

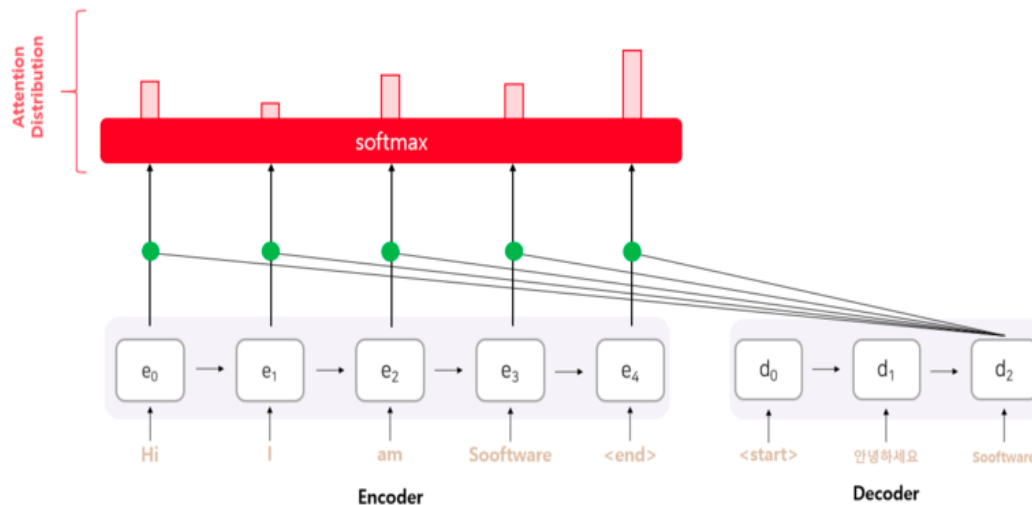


디코더에서 출력 단어를 예측하는 **매 시점마다**,
인코더에서의 전체 입력 문장을 다시 한번 **참고**하는 방식

Attention

Attention

입력 시퀀스의 **특정한 부분에 집중**하는 방법

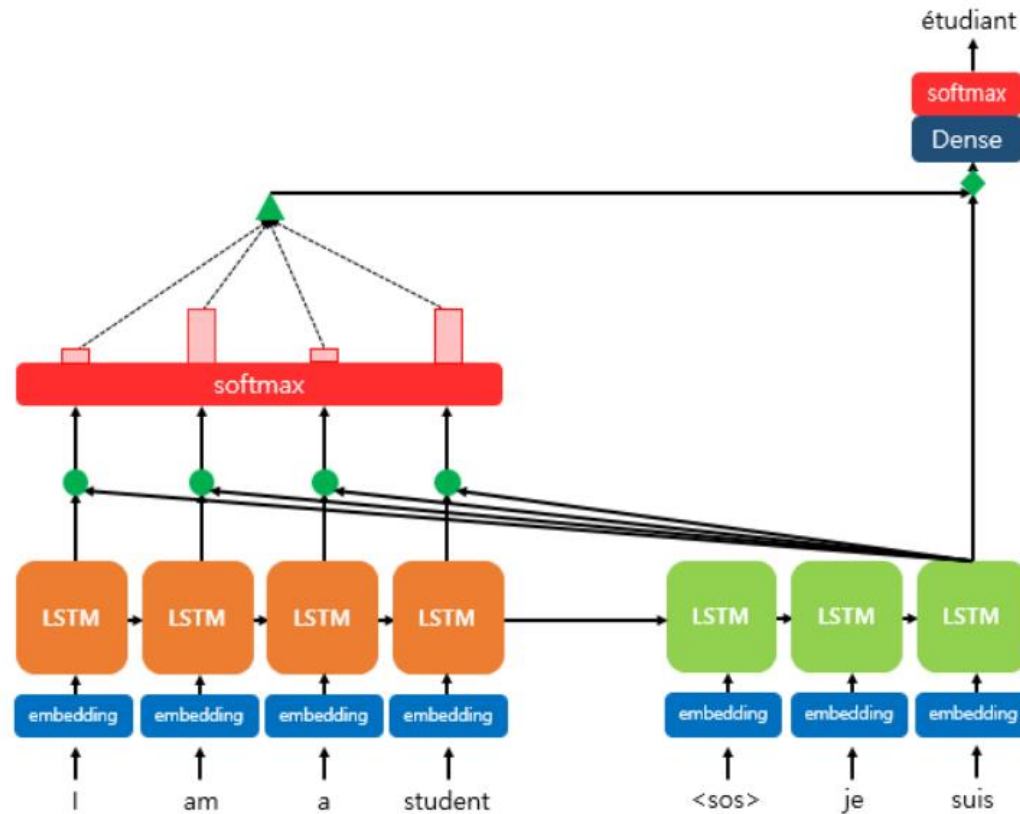


고정된 하나의 static context vector가 아니라
문맥에 맞는 새로운 attention vector (dynamic context vector)를 만듦

1

Attention

Attention mechanism 예시



프랑스어로 세 번째 단어를 만들기 위해
"suis" 토큰과 attention vector를 받고 있음

1

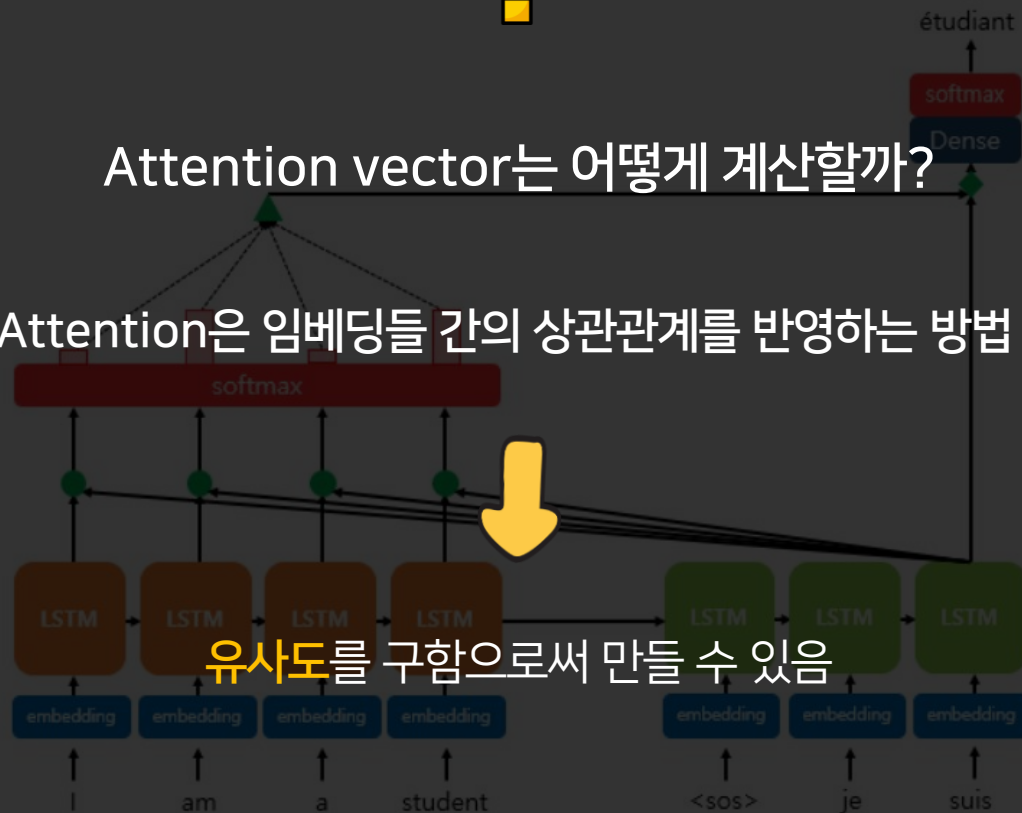
Attention

Attention mechanism 예시



Attention vector는 어떻게 계산할까?

Attention은 임베딩들 간의 상관관계를 반영하는 방법



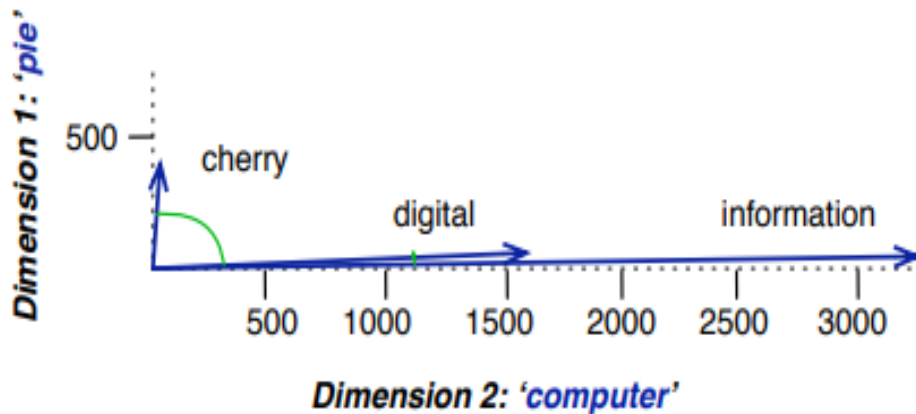
프랑스어로 첫 번째 단어를 만들어내기 위해

번역의 시작을 알리는 <s> 토큰과 attention vector를 받고 있음

1

Attention

유사도 구하기



	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

예시에서 세 개의 단어는 computer와 pie의 조합으로 표현



코사인 유사도로 단어들 사이의 유사도 계산 가능

유사도 구하기

코사인 유사도

두 벡터 사이의 내적(dot product)을 각각의 크기로 나눈 값

코사인 유사도

$$\cos\theta = \frac{x \cdot y}{|x||y|}$$

코사인 유사도와 내적은 스케일링 차이이므로
유사도를 위해 내적을 대신 사용할 수 있음

Scaled-Dot Product Attention

Scaled-Dot Product Attention

내적을 이용한 Attention 연산 방법

Scaled-Dot Product Attention

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

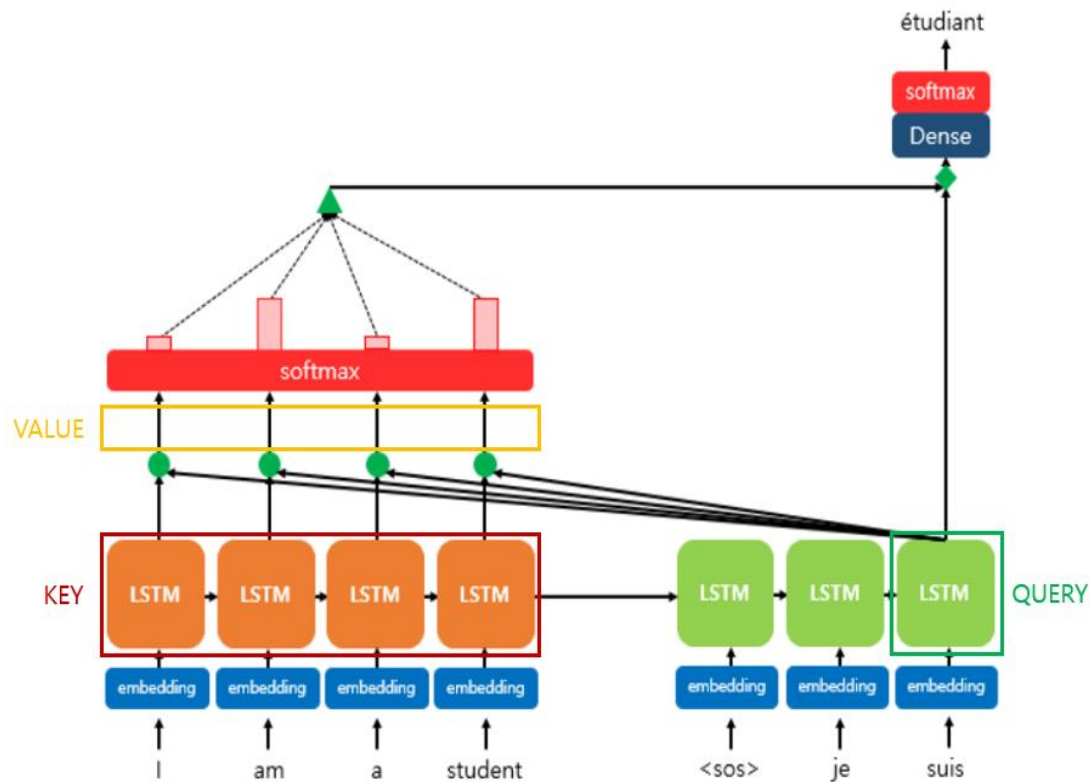
1

Attention

Query, Key, Value

Query, Key, Value

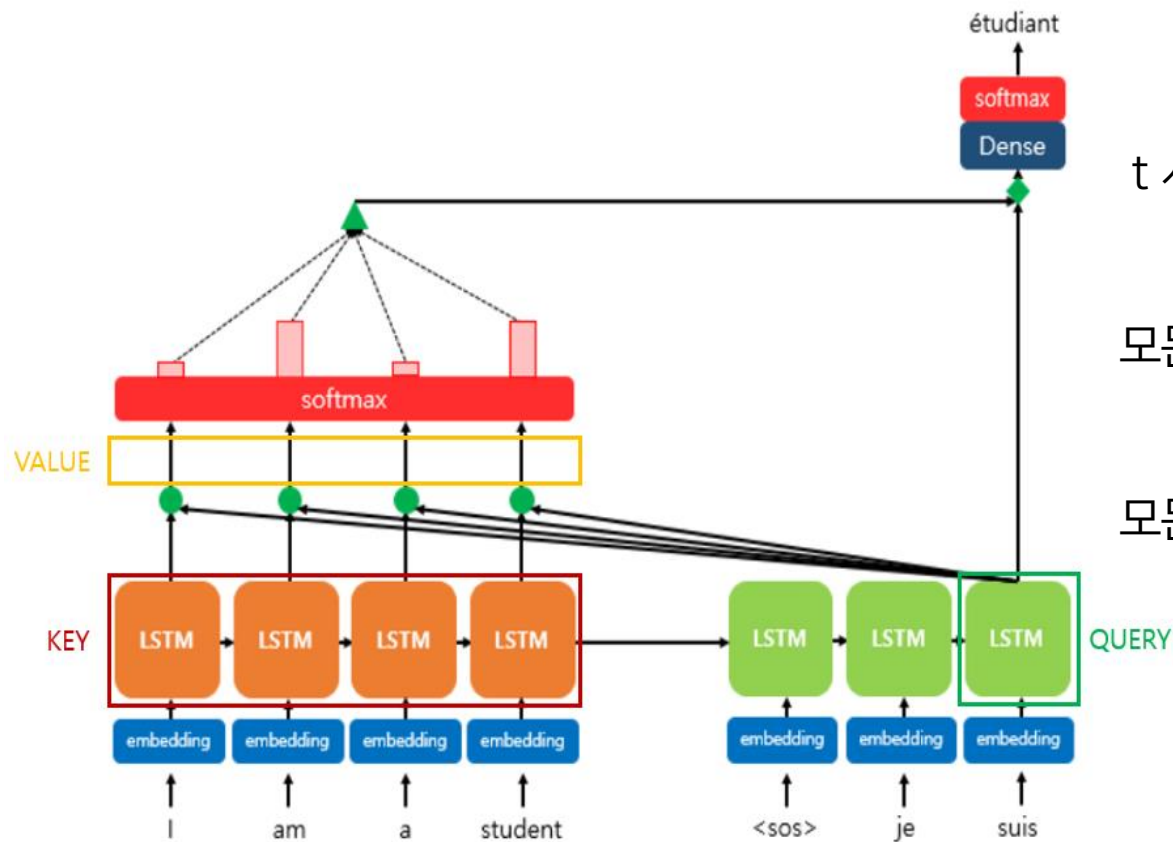
Attention 함수는 Query(질문)와 Key, Value 쌍의 대응 관계로 설명할 수 있음



1

Attention

Query, Key, Value



Query

t 시점의 디코더 셀에서의 은닉 상태

Key

모든 시점의 인코더 셀의 은닉 상태들

Value

모든 시점의 인코더 셀의 은닉 상태들

Attention 수식

Scaled-Dot Product Attention

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



i번째 단어에 대한 하나의 Attention vector 값

$$\alpha_i = \frac{\exp(k_i^T q)}{\sum_{j=1}^n \exp(k_j^T q)}$$

Attention 연산

 QK^T 유사도 계산

Query matrix인 Q 와 Key matrix인 K 에 대해 행렬 곱 수행



벡터 간의 내적은 유사도로 생각할 수 있음

Attention 연산

 QK^T 유사도 계산

Query matrix인 Q 와 Key matrix인 K 에 대해 행렬 곱 수행



예시를 통해 알아보자!

벡터 간의 내적은 유사도로 생각할 수 있음

Attention 연산 예시

He *left* his right shoe at home / 그는 그의 오른쪽 신발을 집에 *두고* 왔다

Turn *left* at the next intersection / 다음 교차로에서 *왼쪽으로* 돌아라



S1 : Query = "두고"

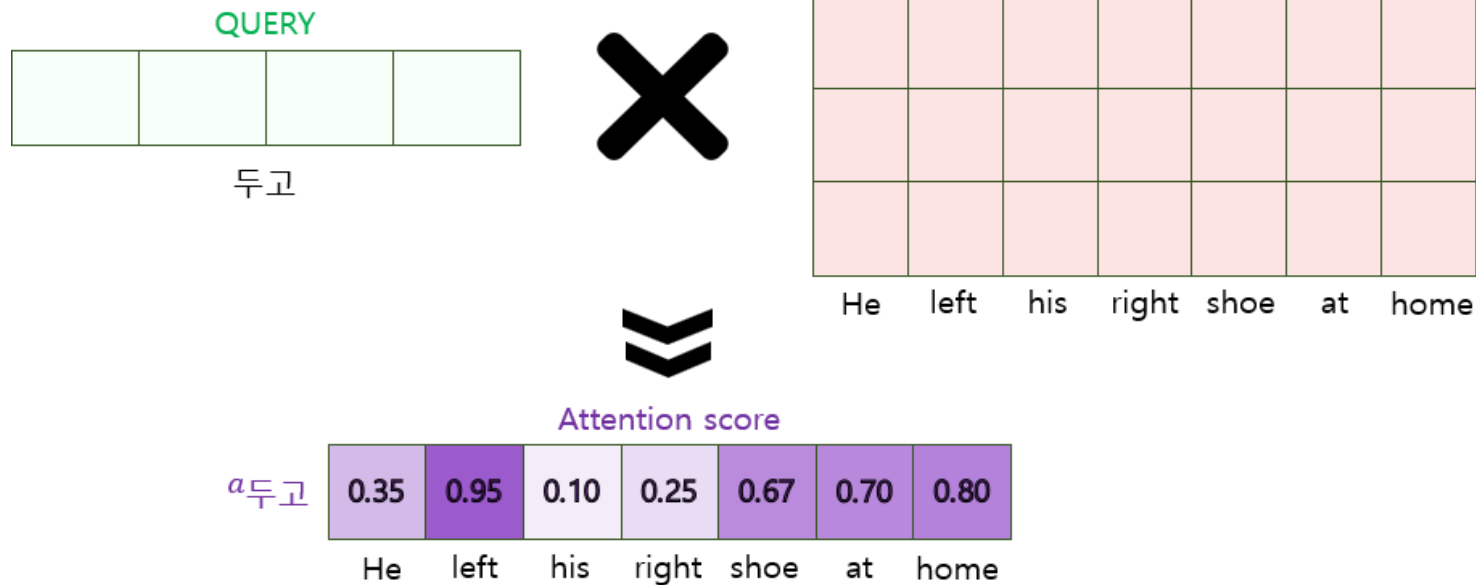
S2 : Query = "왼쪽으로"

1

Attention

Attention 연산 예시

S1 : 그는 그의 오른쪽 신발을 집에 두고 왔다



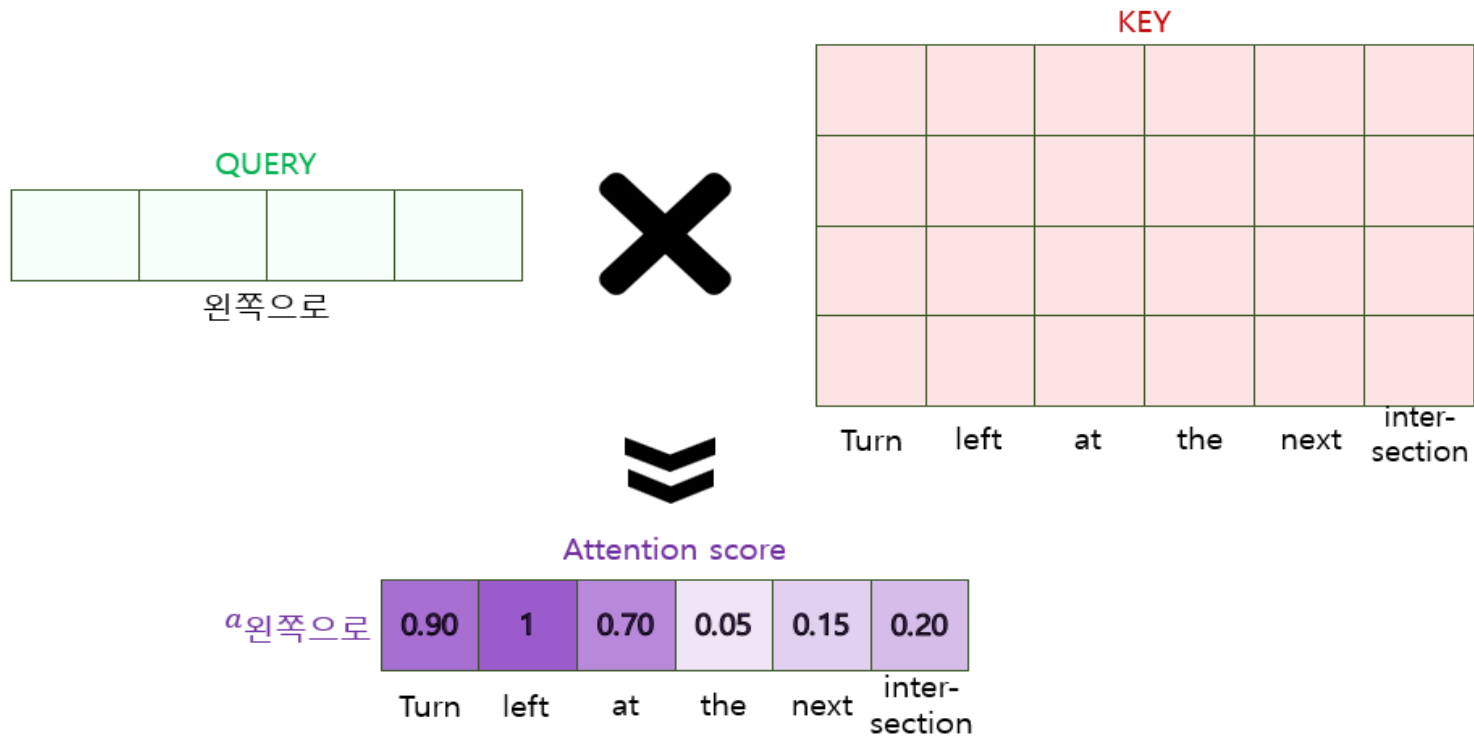
S1의 각 단어에 대한 QK^T 연산 결과

1

Attention

Attention 연산 예시

S2 : 다음 교차로에서 **왼쪽으로** 돌아라

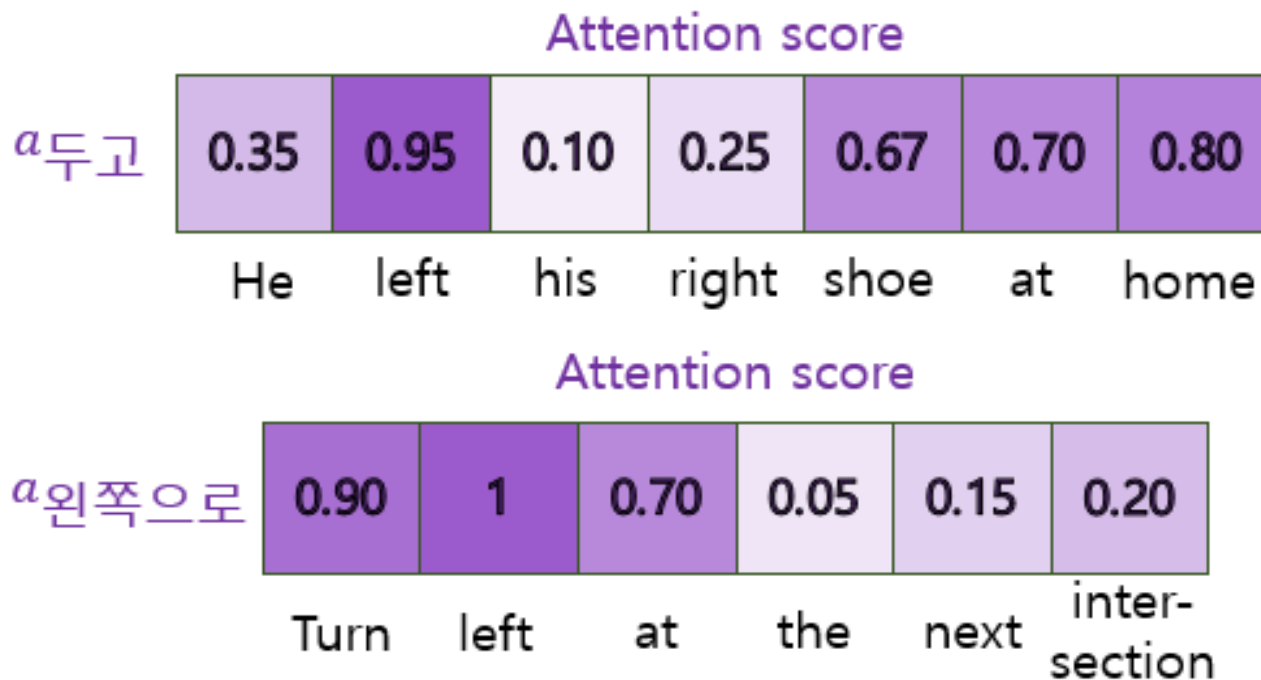


S2의 각 단어에 대한 QK^T 연산 결과

1

Attention

Attention 연산 예시



쿼리와 문장에 따라 서로 다른 단어에 집중함
 즉, QK^T 연산을 통해 각 단어마다 문장의 맥락에 맞는
 동적인 Attention score를 구할 수 있음

Attention 연산



softmax 와 $\sqrt{d_K}$ 표준화

유사도를 구한 후, Scaled Dot-Product Attention에서 표준화

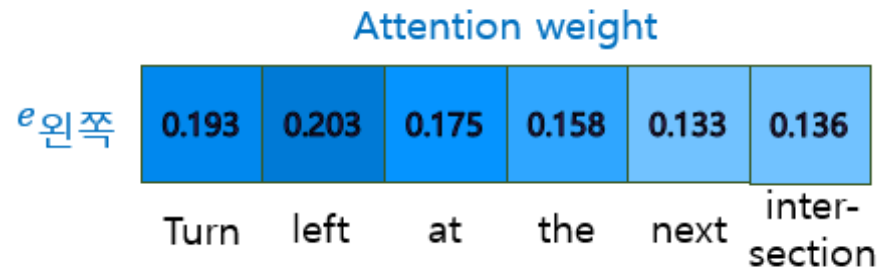
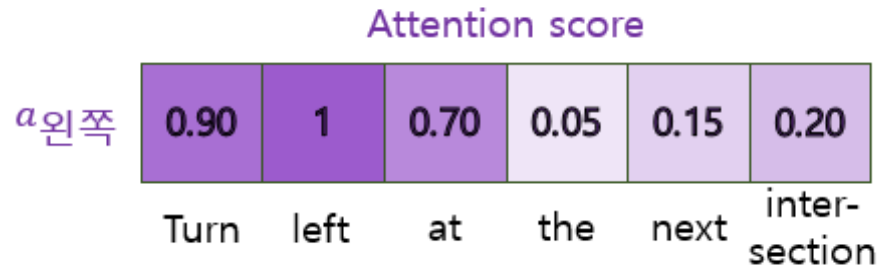


$\sqrt{d_K}$ 를 통한 스케일링과 softmax 함수를 통해
(0,1) 사이의 일종의 확률 값으로 바꿔줌으로써
Attention weight를 구할 수 있음

1

Attention

Attention 연산 예시



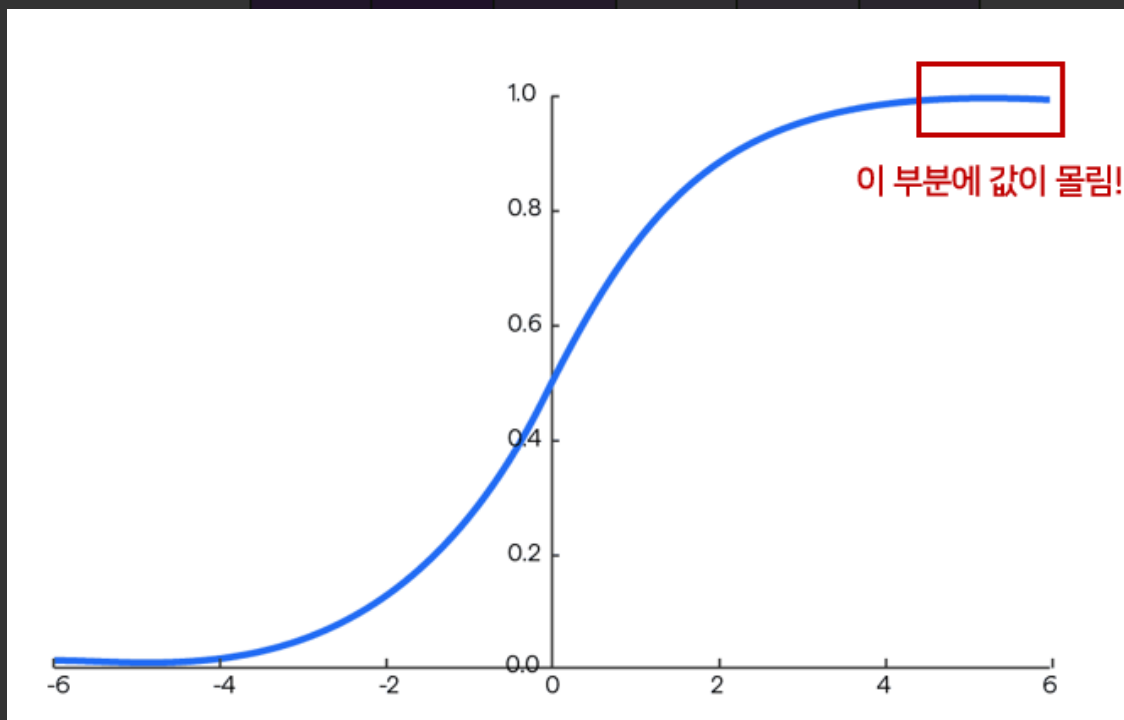
1

Attention



Attention 연산 예시

왜 $\sqrt{d_K}$ 로 나누는 스케일링이 필요할까?



임베딩 벡터의 차원이 커질수록, QK^T 는 점점 절대값이 큰 값을 갖게 되어
 softmax 함수가 매우 작은 기울기 값을 갖게 되기 때문

Attention 연산



QK^T 유사도 계산



softmax 와 $\sqrt{d_K}$ 표준화



2번까지의 과정을 통해 Attention weight,
각 단어마다 집중하는 정도를 구함

Attention 연산



QK^T 유사도 계산



softmax 와 $\sqrt{d_K}$ 표준화



단순히 단어 사이의 유사도를 구한 것이므로
임베딩으로 다시 나타내야 함

각 단어마다 집중하는 정도를 구함

1

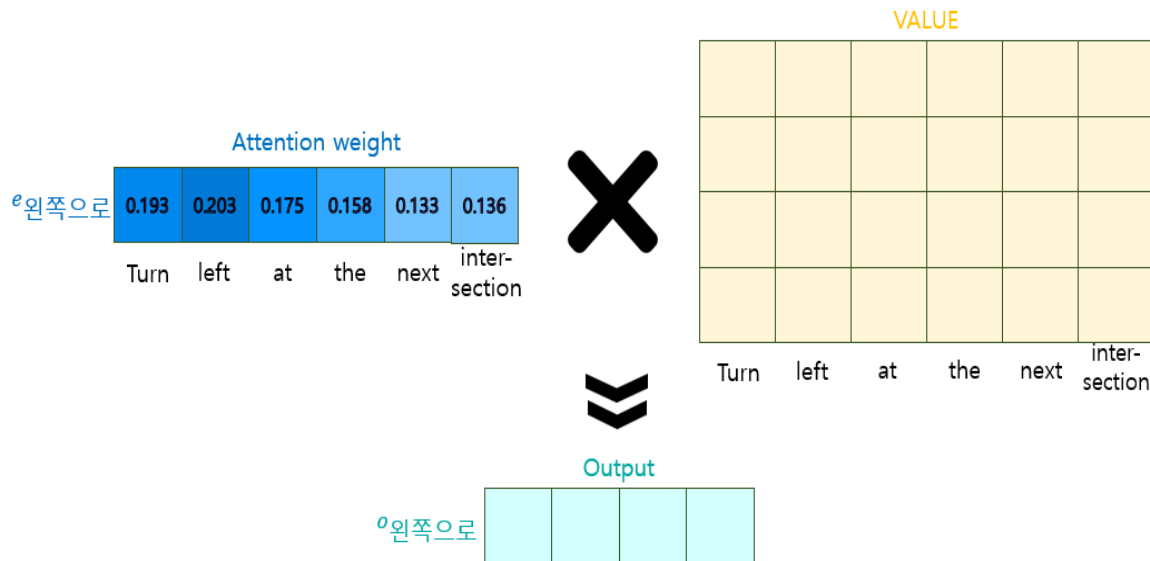
Attention

Attention 연산



V 새로운 임베딩 벡터

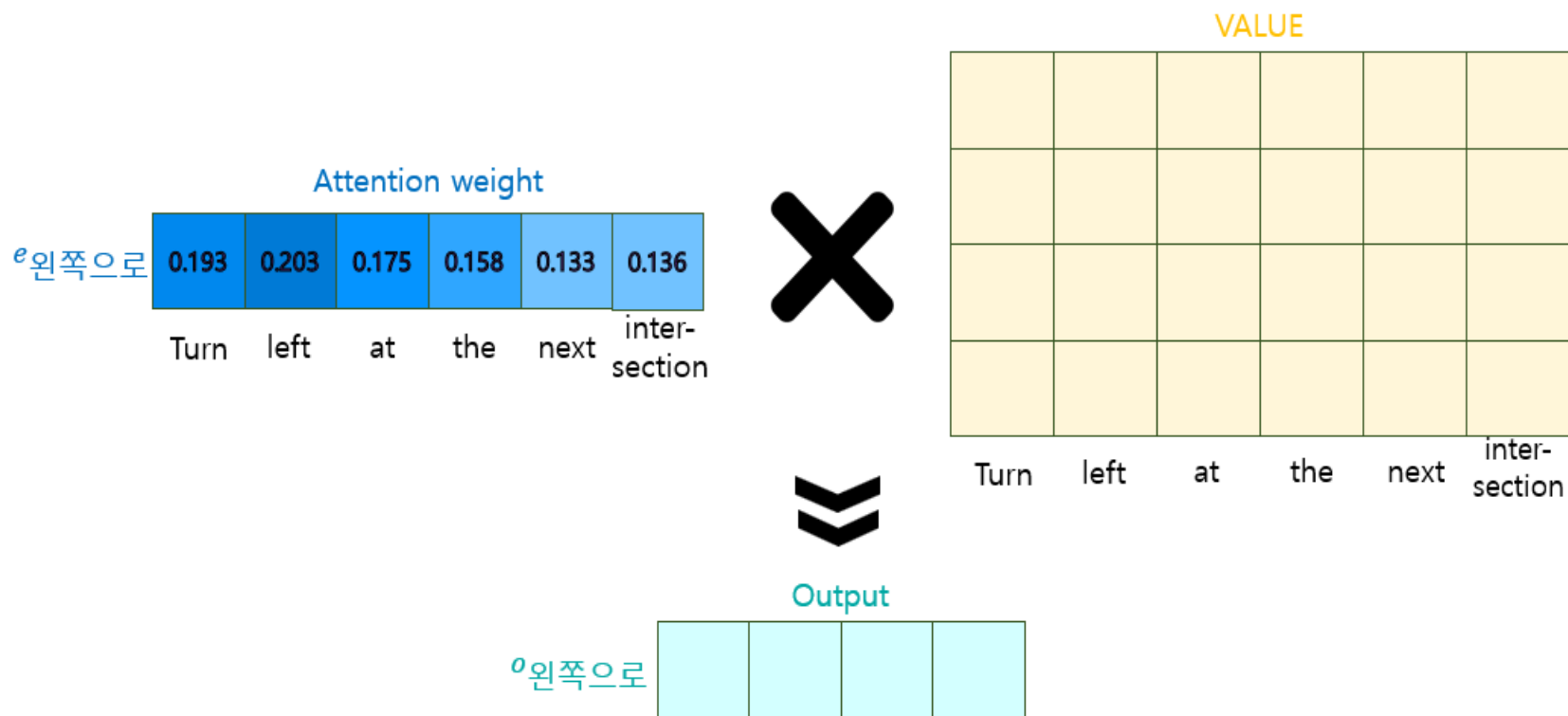
행렬 V 의 곱을 통해 새로운 임베딩 벡터를 만들어 냄



1

Attention

Attention 연산 예시



$$\begin{aligned}
 o_{\text{왼쪽}} &= 0.193 * \text{Turn} + 0.203 * \text{left} + 0.175 * \text{at} \\
 &\quad + 0.158 * \text{the} + 0.133 * \text{next} + 0.136 * \text{intersection}
 \end{aligned}$$

Self-Attention

Attention은 서로 다른 두 개의 시퀀스를 연결하여
상대적인 중요성을 가중치로 반영하는 기법



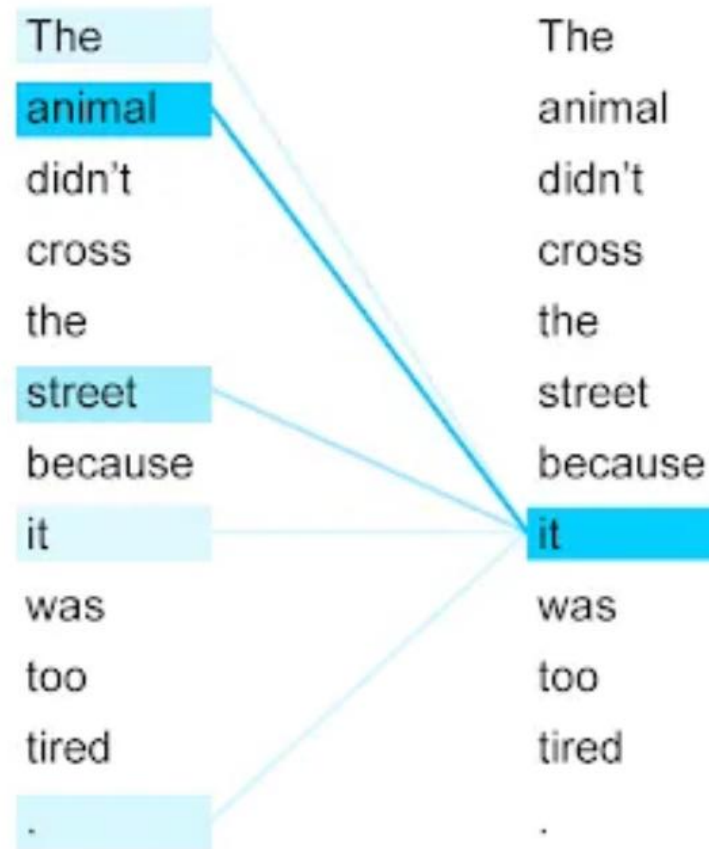
동일한 입력 시퀀스로부터 Q, K, V 행렬을 생성하는
self-attention (intra-attention) 으로 사용

Key, Value, Query 모두 같은 값을 사용

1

Attention

Self-Attention



"it"을 쿼리로 하는 self-attention

Self-Attention은 동일한 시퀀스의 서로 다른 위치에 있는 단어들을
연관시킨다는 점에서 **단어 간의 의존성**을 고려할 수 있음

Multi-Head Attention

Multi-Head Attention

각각 독립적인 Attention을 여러 번 병렬적으로 적용한 후,
이것들을 연결시켜 하나의 온전한 Attention weight를 만드는 방법

Multi-Head Attention

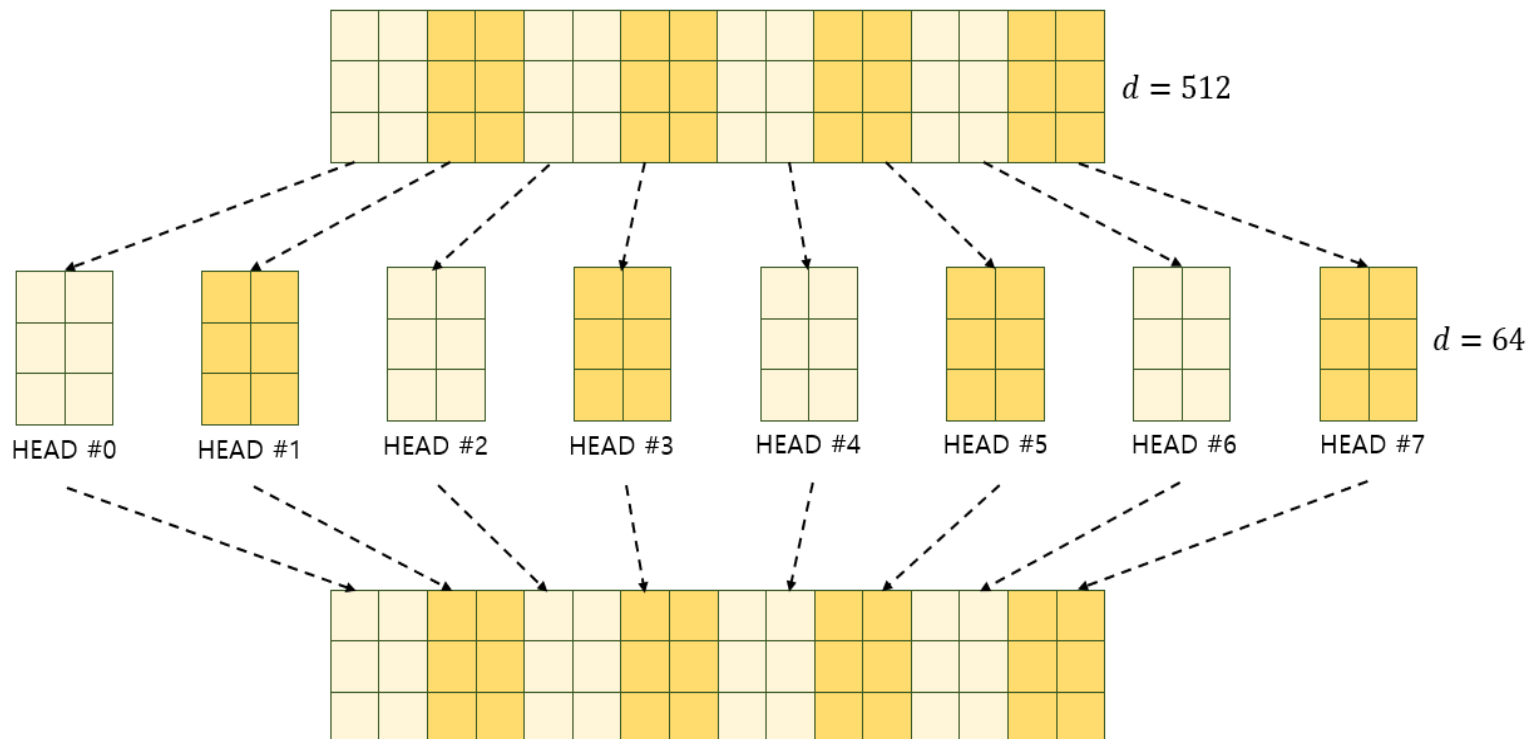
$$MultiHead(Q, K, V) = [head_1; head_2; \dots; head_h]W^O$$

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

1

Attention

Multi-Head Attention 예시

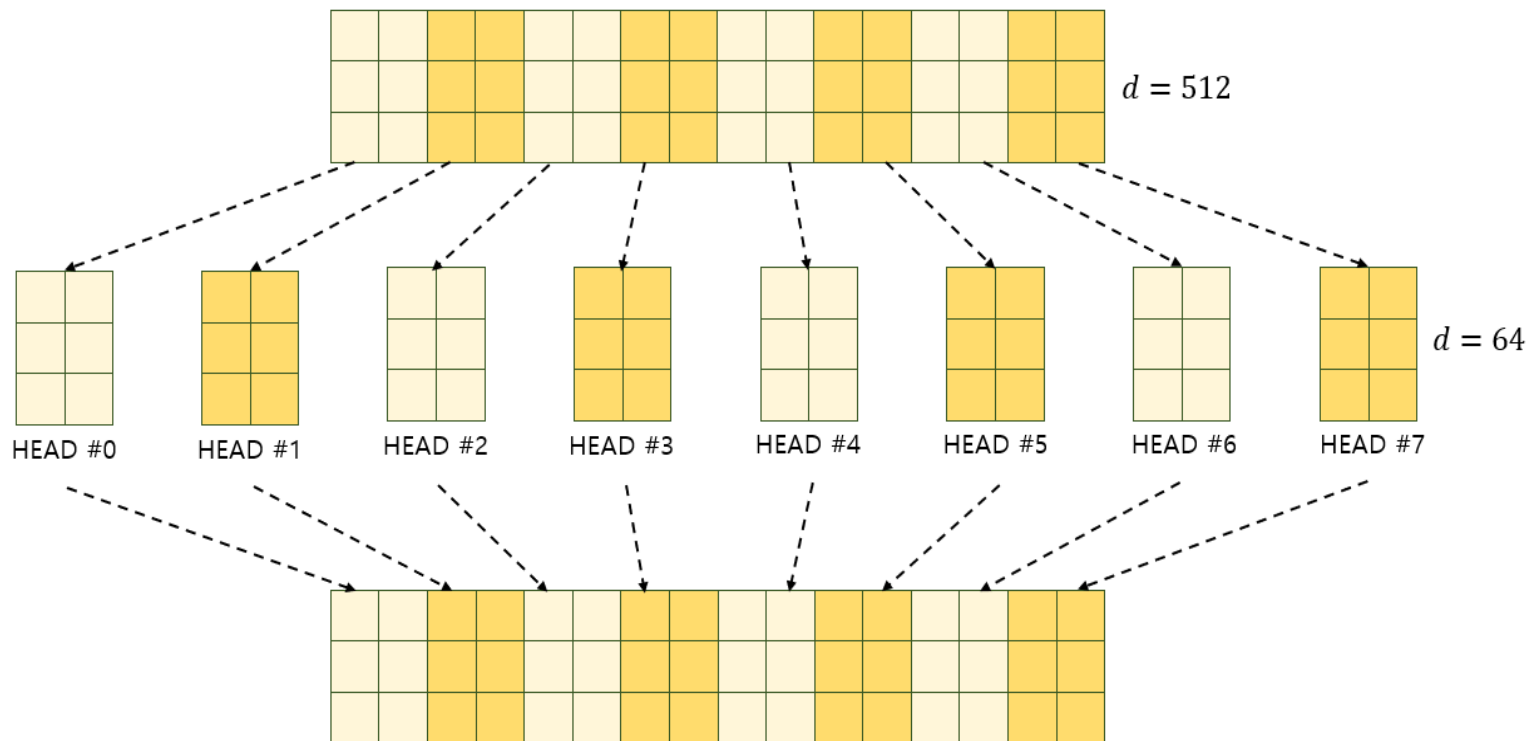


512차원의 임베딩 벡터($d_{model} = 512$)에 대해
8개의 독립적인 Attention($nhead = 8$) 적용

1

Attention

Multi-Head Attention 예시

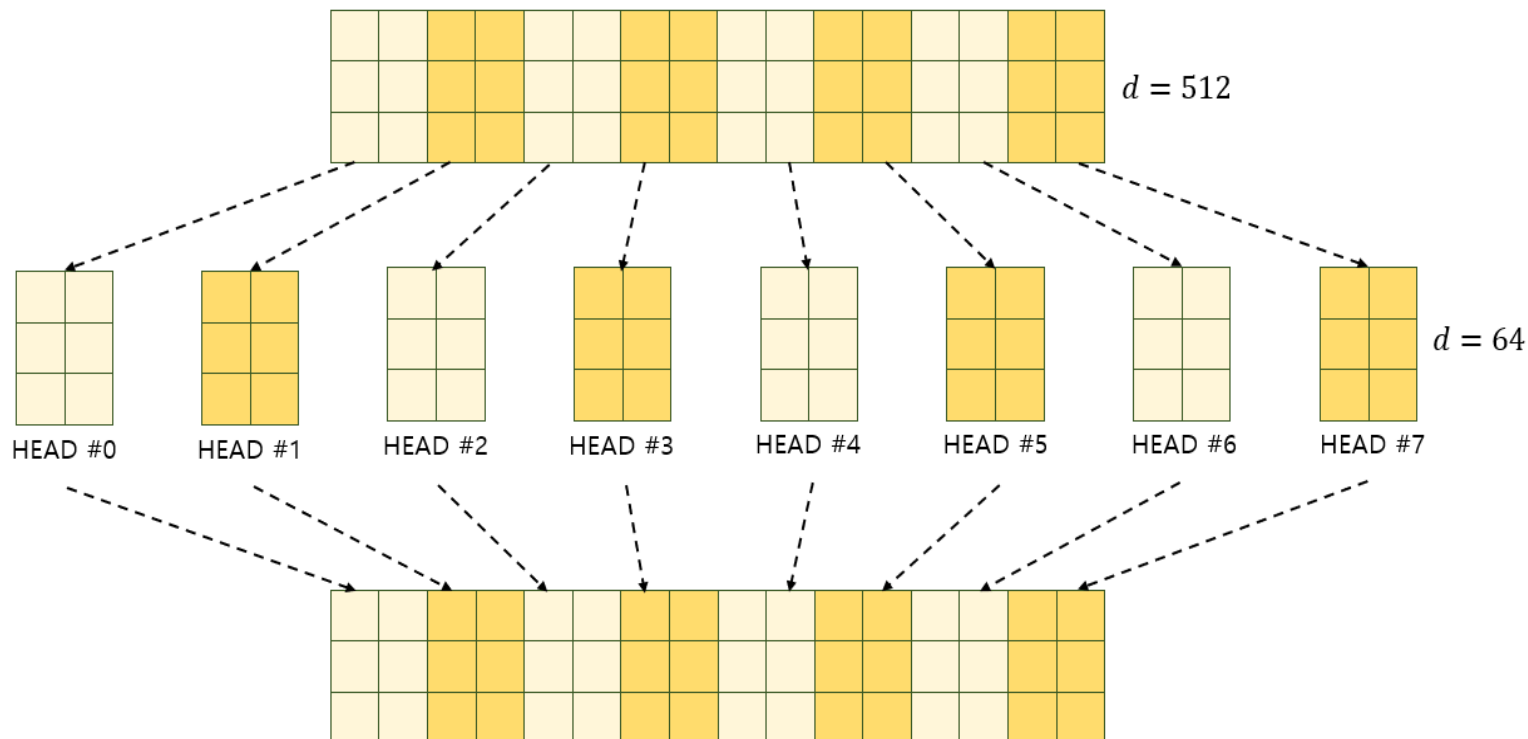


각각의 Attention Head는 $512/8 = 64$ 차원의
임베딩 벡터에 대해 Attention 적용하게 됨

1

Attention

Multi-Head Attention 예시



즉, 더 낮은 차원의 임베딩에 대해 Attention을 적용하기 때문에
고차원에서 학습하기 어려웠던 특징 더 잘 학습

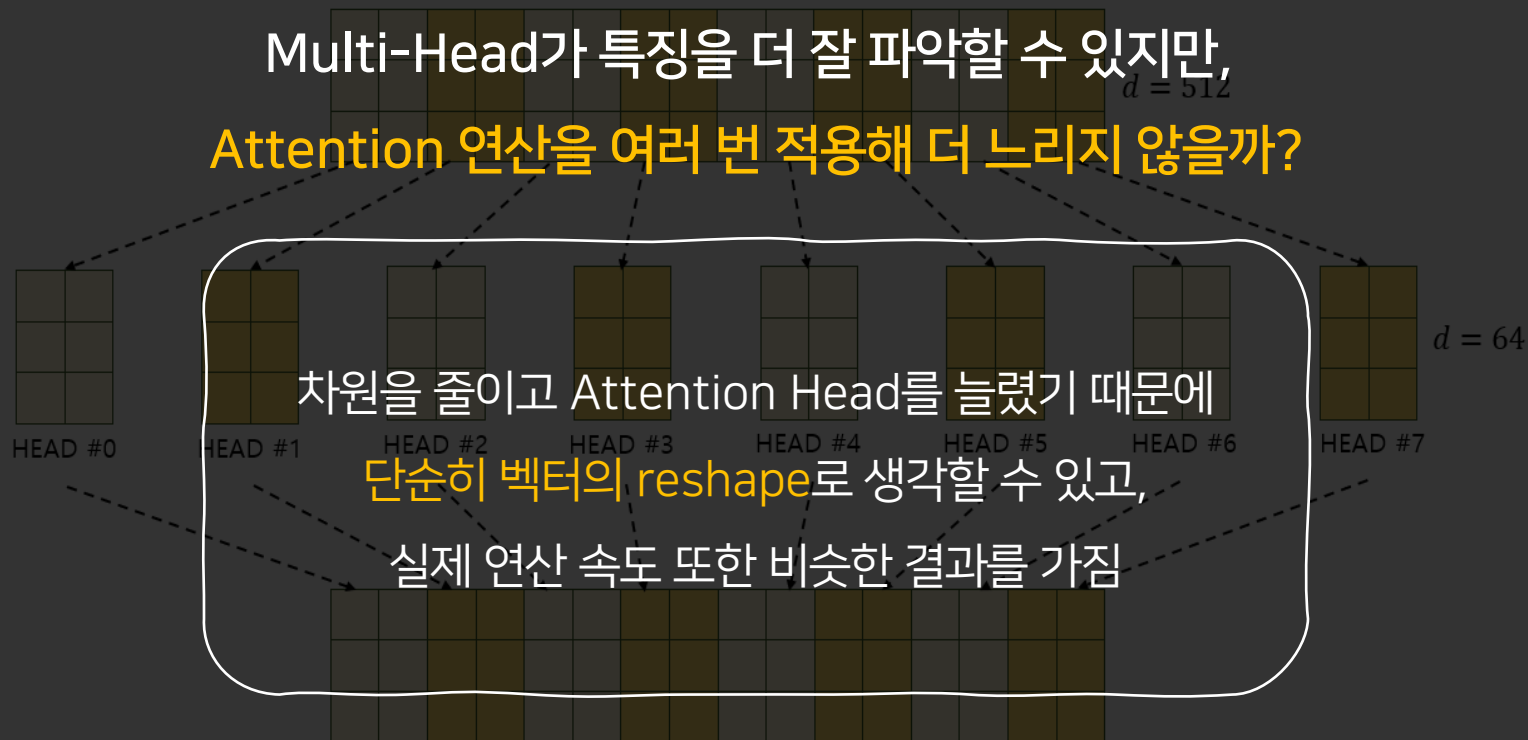
1

Attention

Multi-Head Attention 예시

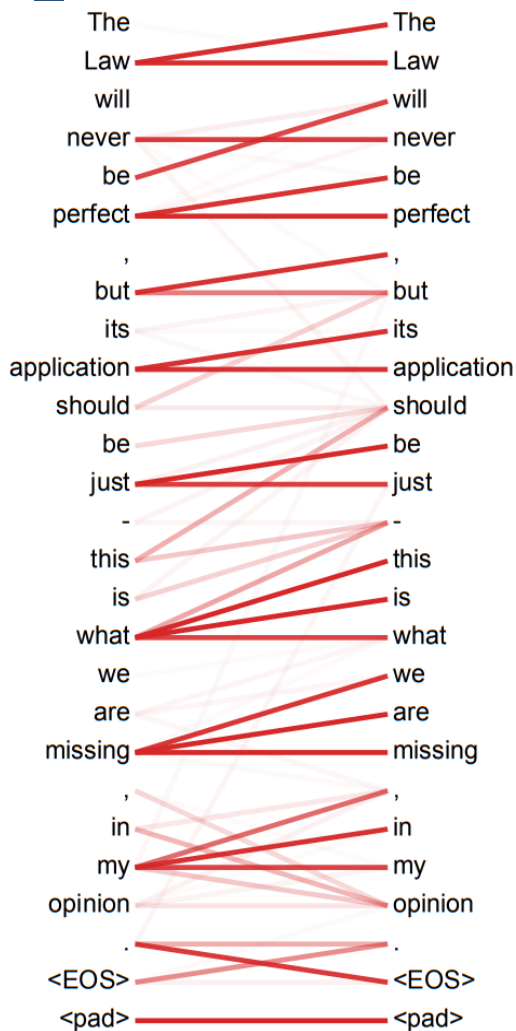


Multi-Head가 특징을 더 잘 파악할 수 있지만,
 Attention 연산을 여러 번 적용해 더 느리지 않을까?

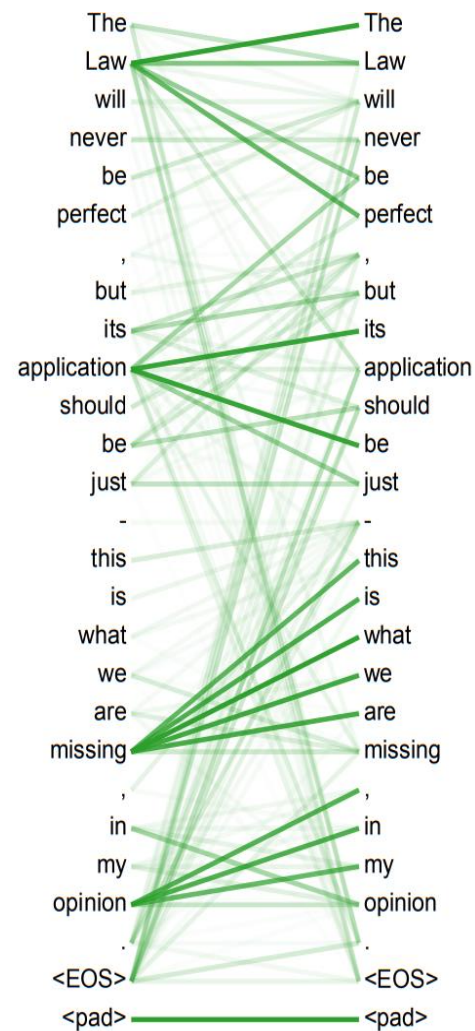


즉, 더 낮은 차원의 임베딩에 대해 Attention을 적용하기 때문
 에 고차원에서 학습하기 어려웠던 특징 더 잘 학습

Multi-Head Attention 예시



같은 문장이라도
서로 다른 구조로 학습하기 때문에
결과적으로 모델 성능 향상에 도움을 줌



2

Transformer

Transformer

Transformer

Attention 기반 Encoder-Decoder 구조를 갖는 모델

RNN에 기반하지 않고 오로지 Attention 기법만을 이용해 만든 새로운 모델

병렬 연산을 통해 학습 시간을 줄일 수 있음

*관련 논문: 2017년 구글의 "Attention is all you need"

Transformer

Transformer

Attention 기반 Encoder-Decoder 구조를 갖는 모델

기존의 RNN 기반 Encoder-Decoder에 비해 **모델의 성능이 향상됨**

현재 많은 모델이 이 모델로부터 파생됨

2

Transformer

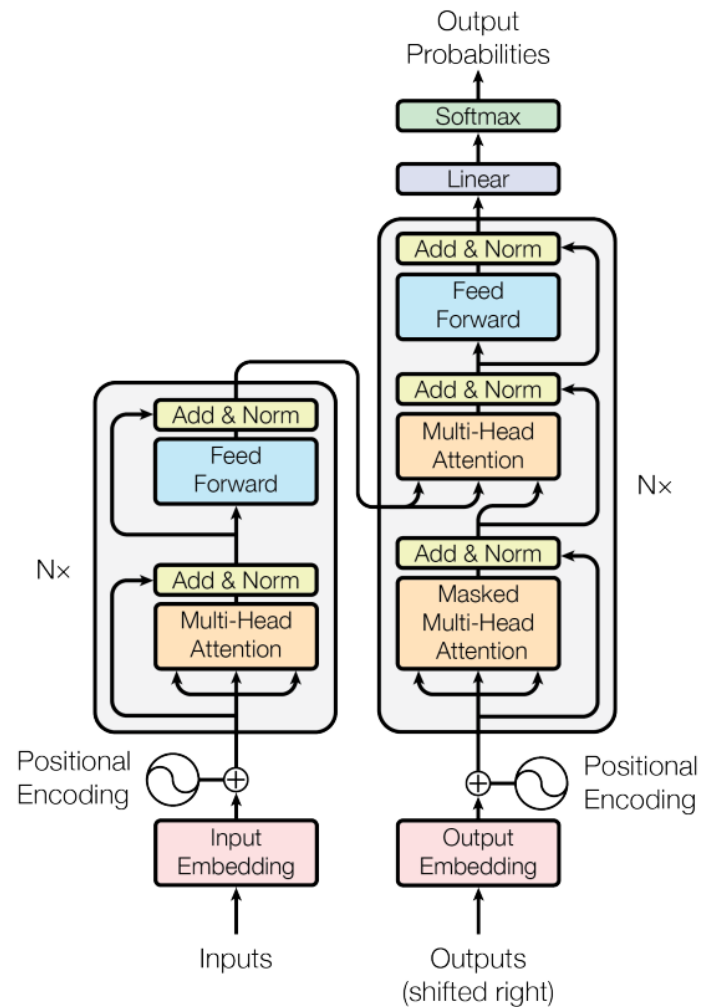
Transformer



일반적인 Encoder-Decoder 구조를 따름



인코더 또는 디코더는
각각의 부분 층을 여러 번 쌓고 있음



Transformer Architecture

Transformer



Attention의 문제점

자연어의 특성인 순차성 무시



Positional Encoding

비선형성 학습에 한계점 존재



FFNN



Positional Encoding

Attention의 첫 번째 한계점

내적 연산을 하므로 시퀀스에 대해 순차적으로 적용되지 않음
즉, 자연어의 특성인 **순차성 무시**

ex. [dogs chase cats] vs [cats chase dogs]

순서를 무시할 경우, 문맥 상 구별 불가

역으로 **위치 정보**를 넣어줄 필요가 있음 ➡ **Positional Encoding**

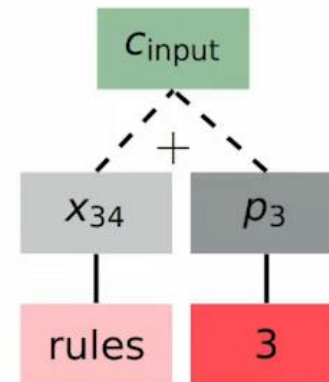
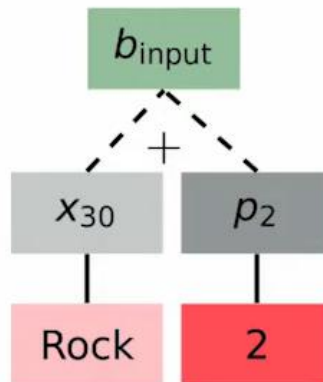
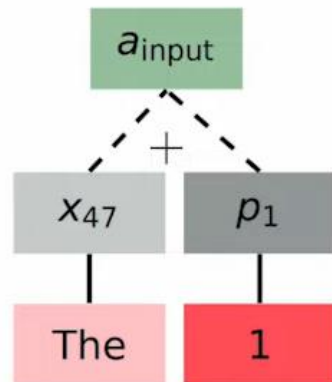
2

Transformer

Positional Encoding

Positional Encoding

각 토큰마다 위치에 대한 정보를 인코딩하는 것



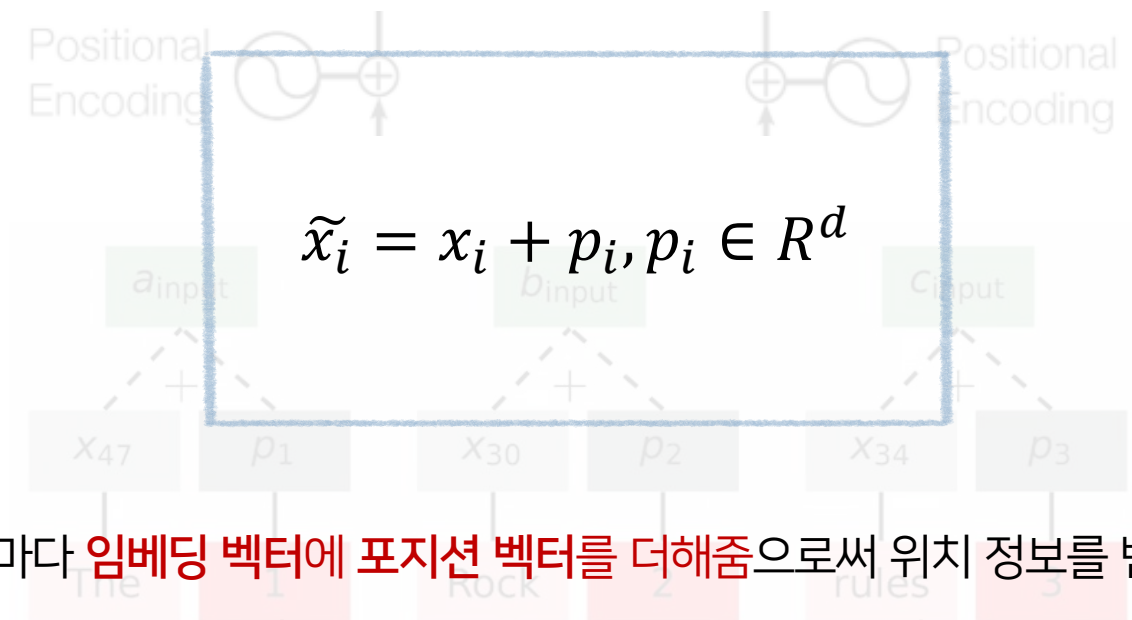
2

Transformer

Positional Encoding

Positional Encoding

각 토큰마다 위치에 대한 정보를 인코딩하는 것



단어마다 임베딩 벡터에 포지션 벡터를 더해줌으로써 위치 정보를 반영함
즉, 학습 이전에 입력 값의 위치를 반영함

Positional Encoding

Positional Encoding

각 토큰마다 위치에 대한 정보를 인코딩하는 것

⋮



Positional Encoding의 4가지 조건

각 위치의 Positional Encoding 값은 유일해야 함

서로 다른 길이의 시퀀스에 대해서도 time-step마다 동일한 간격을 가져야 함

시퀀스의 길이와 관계없이 적용 가능해야 함

각 위치의 Positional Encoding 값은 결정론적이어야 함

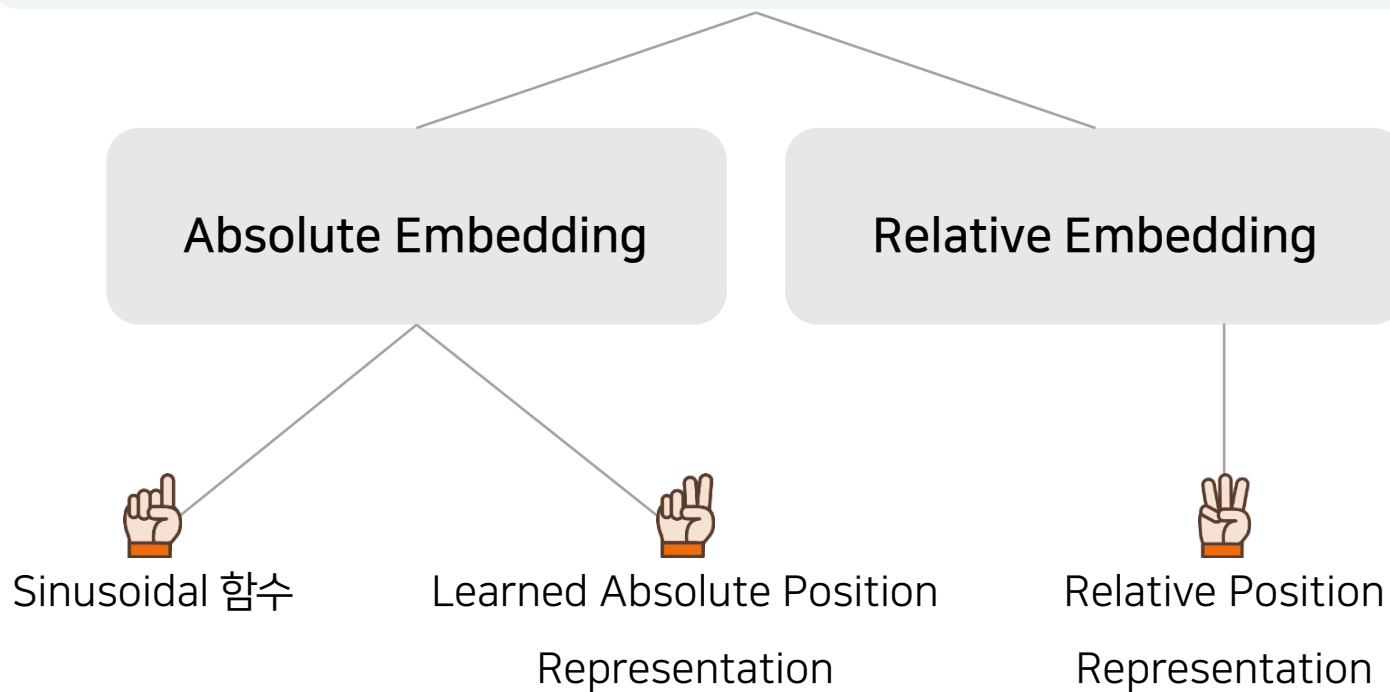
2

Transformer

Positional Encoding

Positional Encoding

각 토큰마다 위치에 대한 정보를 인코딩하는 것

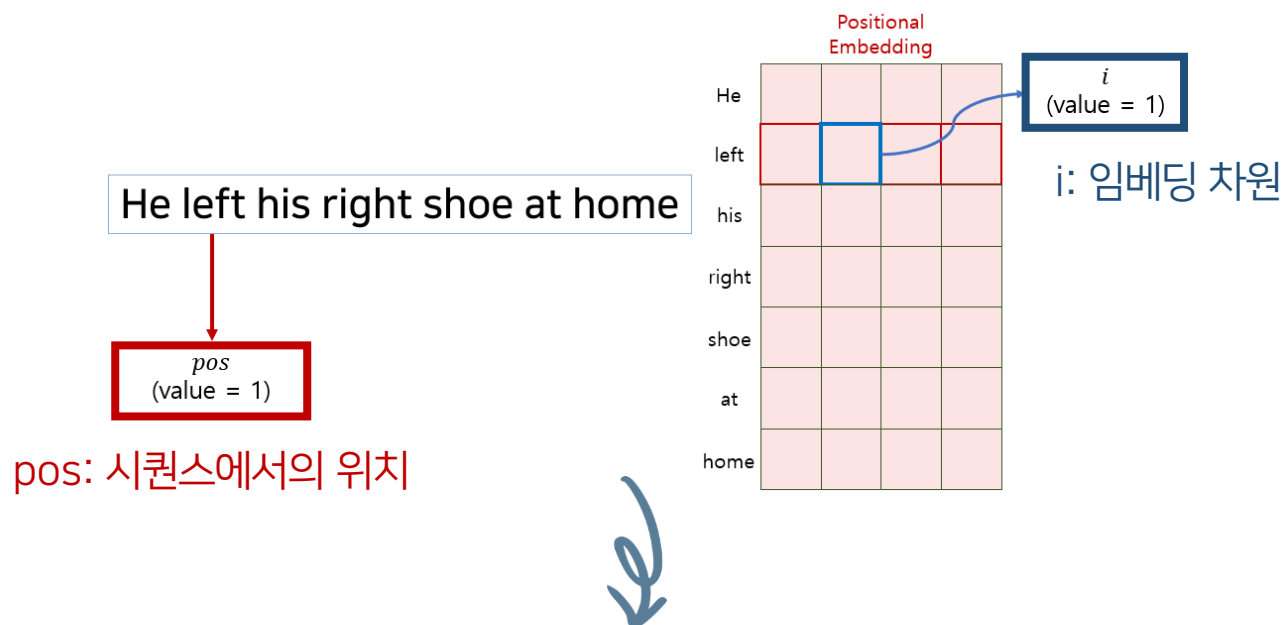


2

Transformer

Positional Encoding - ① Sinusoidal 함수

$$PE(pos, i) = \begin{cases} \sin(pos/10000^{2i/d_{model}}), i = 2k \\ \cos(pos/10000^{2i/d_{model}}), i = 2k + 1 \end{cases}$$



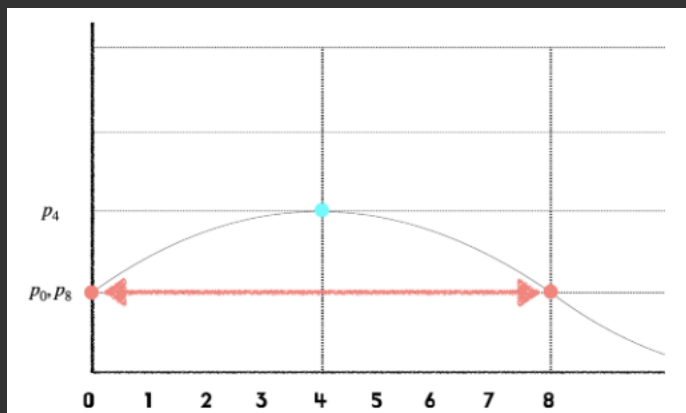
각 단어 위치마다 상수 값의 위치 임베딩 값을 갖기 때문에 **변하지 않음**(absolute)

2 Transformer

Positional Encoding - ① Sinusoidal 함수



임베딩 차원마다 다른 주기의 함수를 사용하는 이유



주기가 16인 sin 함수의 경우

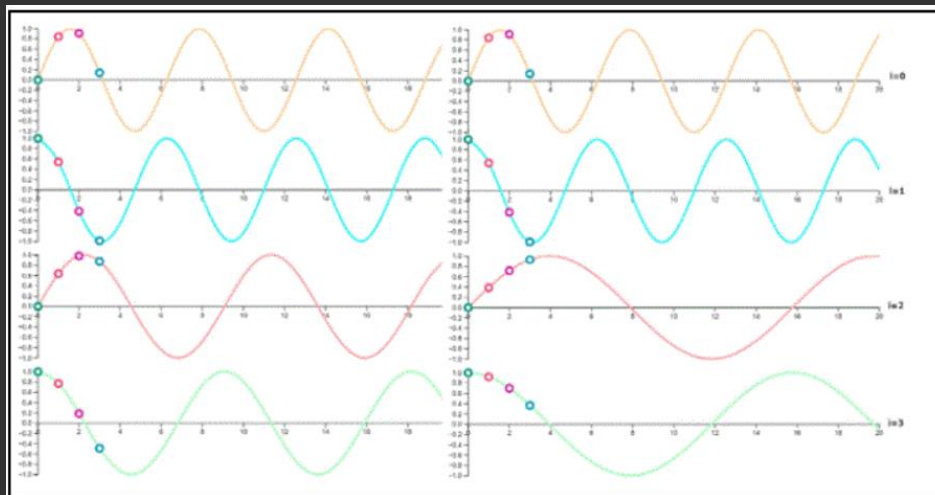
각 단어마다 위치 임베딩 값이 동일하다고 가정

동일한 위치의 임베딩 값이 나올 가능성 존재 → 구별 불가능

2 Transformer

Positional Encoding - ① Sinusoidal 함수

임베딩 차원마다 다른 주기의 함수를 사용하는 이유



 주기를 다르게

 사인/코사인 함수를 번갈아 사용

위치 정보의 차이 최대화

2

Transformer

Positional Encoding - ② Learned Absolute Position Representation

Learned Absolute Position Representation

절대적인 위치를 학습하는 방법



더 유연한 위치 임베딩 사용

```
self.pos_emb = nn.Parameter(torch.zeros(1, config.block_size, config.n_embd))
```

구현이 간단하며
성능 하락이 크지 않음

학습 중에 보지 않은
더 긴 길이의 입력으로의 확장 어려움

Positional Encoding - ③ Relative positional representation

Relative Positional Representation

문맥 내 단어들 간의 **상대적인 위치**를 고려하는 방법



Absolute Positional Encoding을 완벽하게 대체



문장 내 단어들의 위치 관계 고려 가능



기존의 Attention 연산을 확장하여 상대적인 위치를 고려할 수 있게 함

2

Transformer

Positional Encoding - ③ Relative positional representation

Relative Positional Representation

문맥 내 단어들 간의 **상대적인 위치**를 고려하는 방법

Attention

$$z_i = \sum_{j=1}^n \alpha_{ij} (x_j W^V)$$

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^n \exp e_{ik}}$$

$$e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$

기존의 Attention 연산과의 비교

Relative Positional Representations

$$z_i = \sum_{j=1}^n \alpha_{ij} (x_j W^V + a_{ij}^V)$$

$$\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^n \exp e_{ik}}$$

$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

Q: Query / K: Key / V: Value

즉, Attention 연산에서 추가적인 항으로 더해지는 **학습이 가능한 임베딩**

2

Transformer

Positional Encoding - ③ Relative positional representation

Relative Positional Representation

문맥 내 단어들 간의 **상대적인 위치**를 고려하는 방법

그녀와 나는 공원을 갔다

	Relative Positional Embedding						
	w_{-3}	w_{-2}	w_{-1}	w_0	w_1	w_2	w_3
그녀와	$a_{0,-3}$	$a_{0,-2}$	$a_{0,-1}$	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
나는	$a_{1,-2}$	$a_{1,-1}$	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$
공원을	$a_{2,-1}$	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
갔다	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$

자신보다 세 번째 앞 단어부터 세 번째 뒤 단어까지 표현할 수 있어야 함

단어 w 길이의 문장은 $(w, 2w-1)$ 크기의 임베딩을 형성함

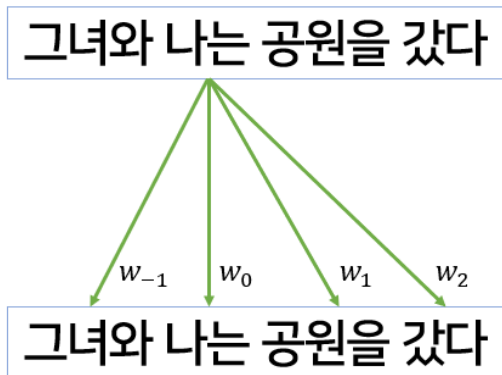
2

Transformer

Positional Encoding - ③ Relative positional representation

Relative Positional Representation

문맥 내 단어들 간의 **상대적인 위치**를 고려하는 방법



	w_{-3}	w_{-2}	w_{-1}	w_0	w_1	w_2	w_3
나는	$a_{1,-2}$	$a_{1,-1}$	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$

상대적인 위치를 고려하여 $[w_{-1}, w_0, w_1, w_2]$ 번째 값들이 위치 임베딩 됨



FFNN(Fully-connected Feedforward Neural Network)

Attention의 두 번째 한계점

Attention 메커니즘

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



만들어진 벡터는 행렬 곱을 통해 만들어진 것
즉, 선형 변환(Linear Transformation)의 일종

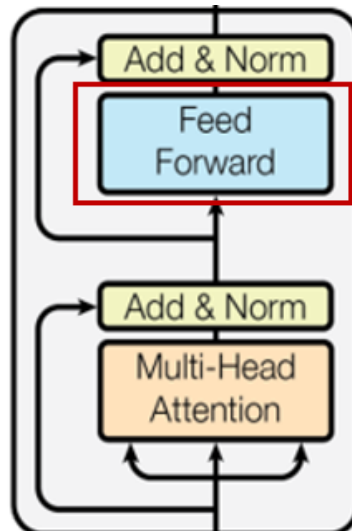
비선형성을 학습하는 데에 한계점 존재 ➡ FFNN

2 Transformer

FFNN(Fully-connected Feedforward Neural Network)

FFNN(Fully-connected Feedforward Neural Network)

모델 비선형 활성화함수 **ReLU**를 적용한 완전 연결층을 추가한 모델



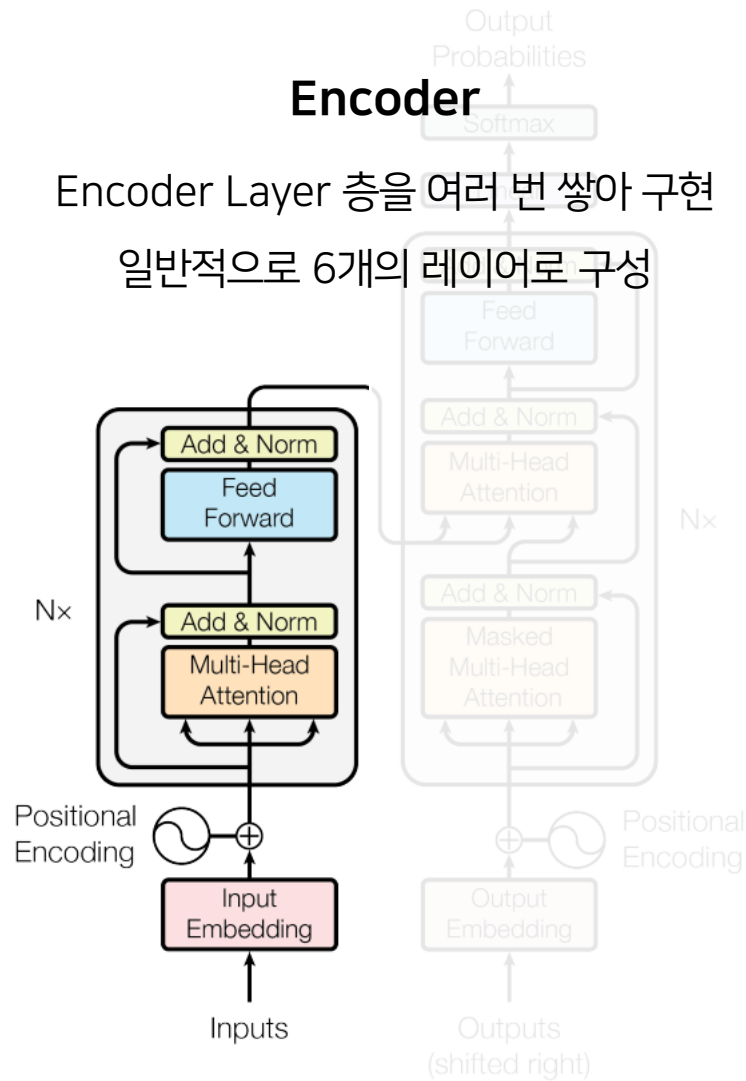
완전 연결층에서 비선형 활성화함수 ReLU를 적용하여 **비선형성** 학습 가능

2 Transformer

Encoder

Encoder

Encoder Layer 층을 여러 번 쌓아 구현
일반적으로 6개의 레이어로 구성



Encoder

① 입력 값에 대한 Multi-Head Attention 연산



② 잔차 연결과 레이어 별 정규화 수행



③ 비선형 구조를 학습하기 위해 FFNN을 학습



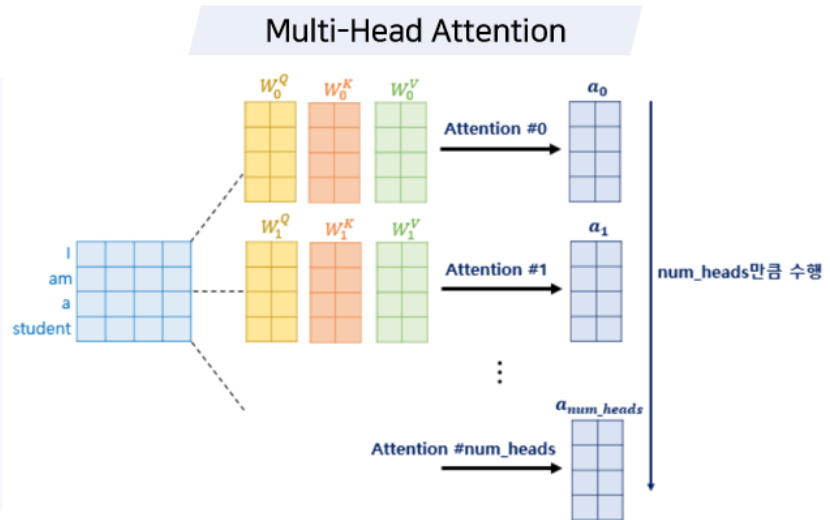
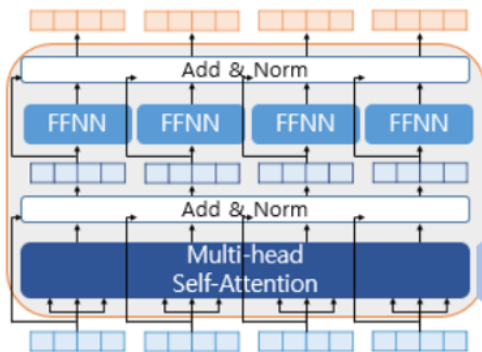
④ 잔차 연결과 레이어 별 정규화를 다시 한 번 수행

2

Transformer

Encoder

① 입력 값에 대한 Multi-Head Attention 연산

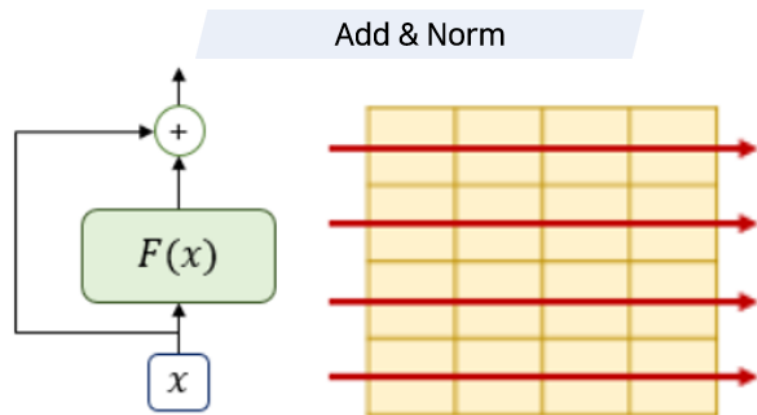
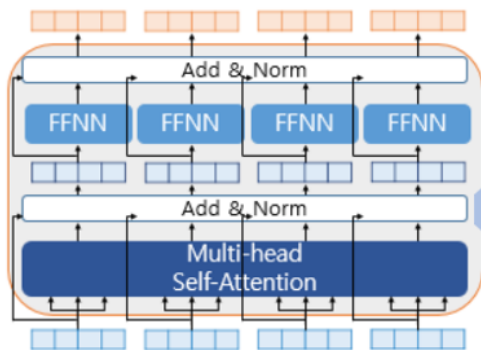


2

Transformer

Encoder

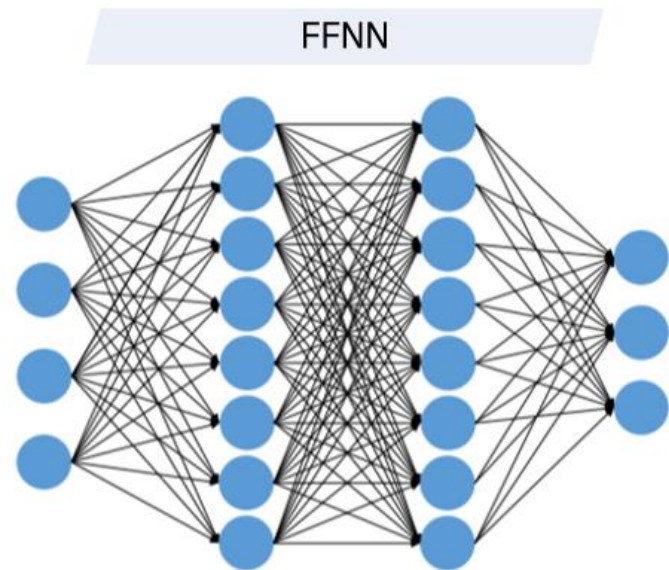
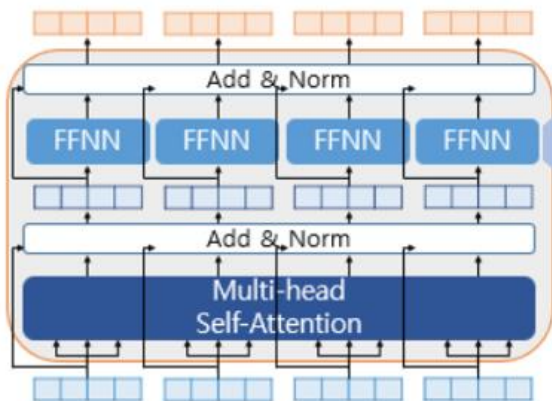
② 잔차 연결과 레이어 별 정규화 수행



2 Transformer

Encoder

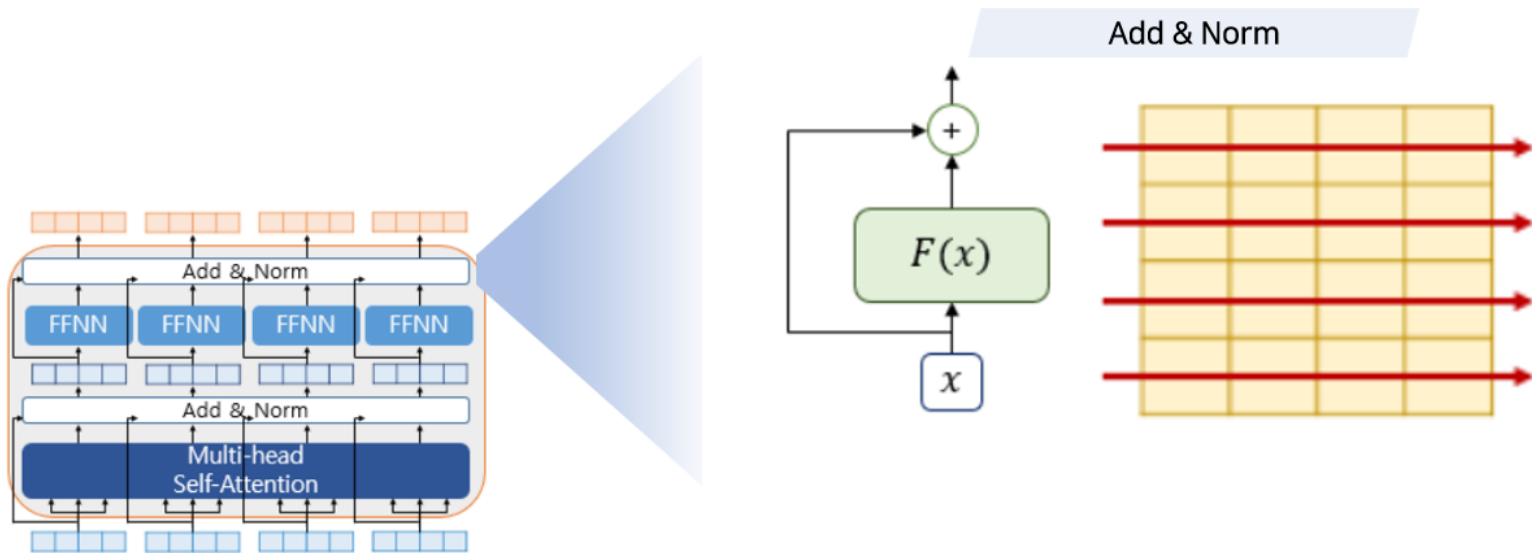
③ 비선형 구조를 학습하기 위해 FFNN을 학습



2 Transformer

Encoder

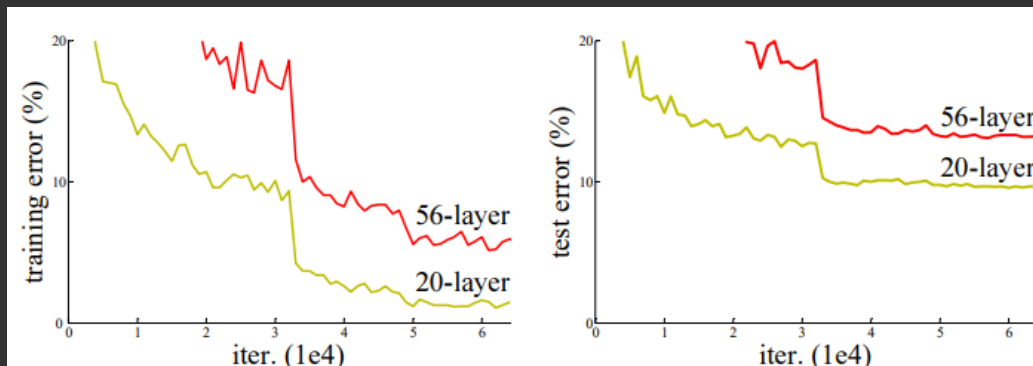
④ 잔차 연결과 레이어 별 정규화를 다시 한 번 수행





Degradation Problem

신경망은 그 깊이가 더 깊어질수록 다양한 특징을 학습할 수 있지만
너무 깊은 신경망은 학습이 지나치게 어려워짐



20-layer 신경망이 56-layer 신경망보다 더 낮은 에러를 만들어 냄



신경망의 깊이가 깊어질수록 **Train/Test 데이터에서의 성능이 악화되는 현상**

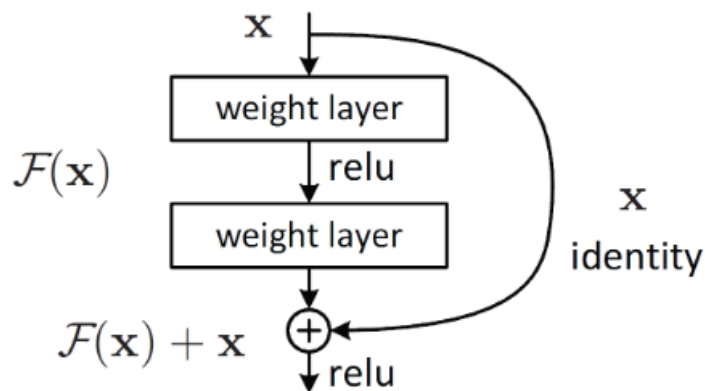
2

Transformer

잔차 연결(Residual Connection)

잔차 연결(Residual Connection)

모델의 새로운 입력으로 은닉층의 출력뿐만 아니라
입력 자기 자신 또한 더해주는 **Identity Mapping**을 하는 것



Residual Connection

모델은 입력 값과 출력 값의 **잔차**에 대해서 학습할 수 있음

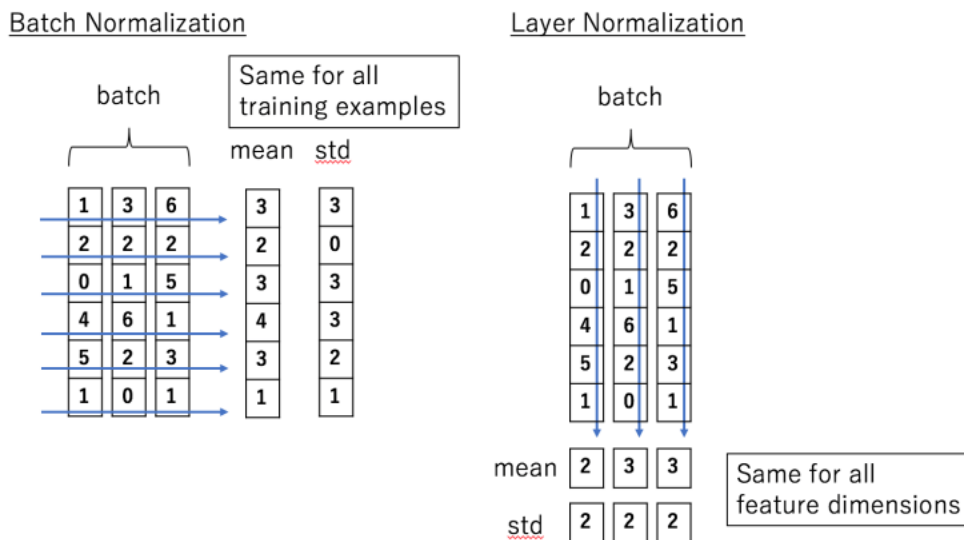
2

Transformer

레이어 별 정규화(Layer Normalization)

레이어 별 정규화(Layer Normalization)

데이터 샘플 단위로 정규화를 실시하는 것



Batch Normalization vs Layer Normalization

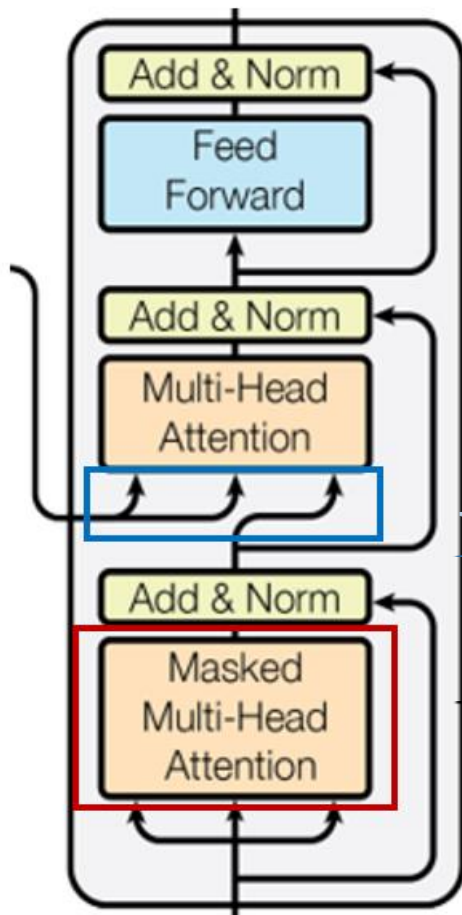
시퀀셜 데이터는 다양한 길이를 갖고, 레이어 별 정규화는 ICS 문제를 해결해 줌

딥러닝 1주차 클린업 참고!

2

Transformer

Decoder



Decoder

Encoder와 마찬가지로
잔차 연결, Layer Normalization, FFNN을 포함하지만
Encoder에 비해 구조가 복잡

Encoder - Decoder Attention과
Masked Multi - Head Attention이 추가

2

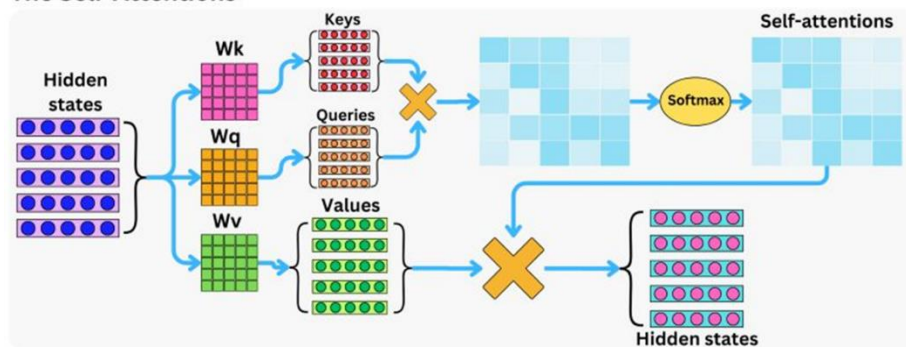
Transformer

Decoder : Encoder-Decoder Attention

Encoder – Decoder Attention

Key와 Value는 같은 값을 사용하지만
Query는 다른 값을 사용하는 **Cross Attention**

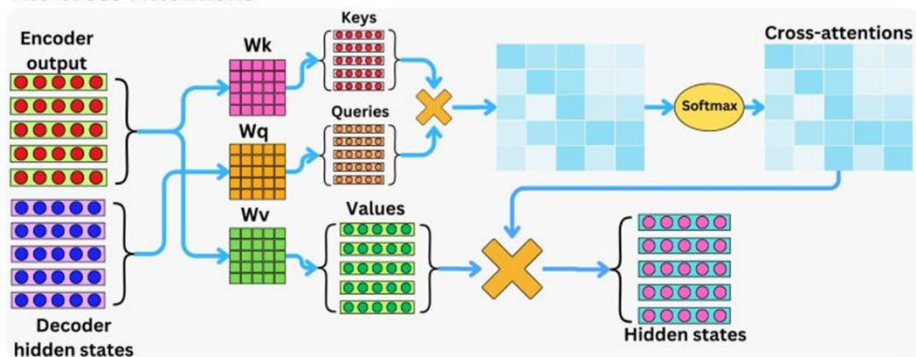
The Self-Attentions



Self – Attention

Query = Key = Value

The Cross-Attentions



Cross – Attention

Query \neq Key = Value

Decoder : Encoder-Decoder Attention

Encoder – Decoder Attention

Key와 Value는 같은 값을 사용하지만
Query는 다른 값을 사용하는 Cross Attention



Decoder의 각 시점마다 Encoder의 모든 Attention 값에 집중하는 효과

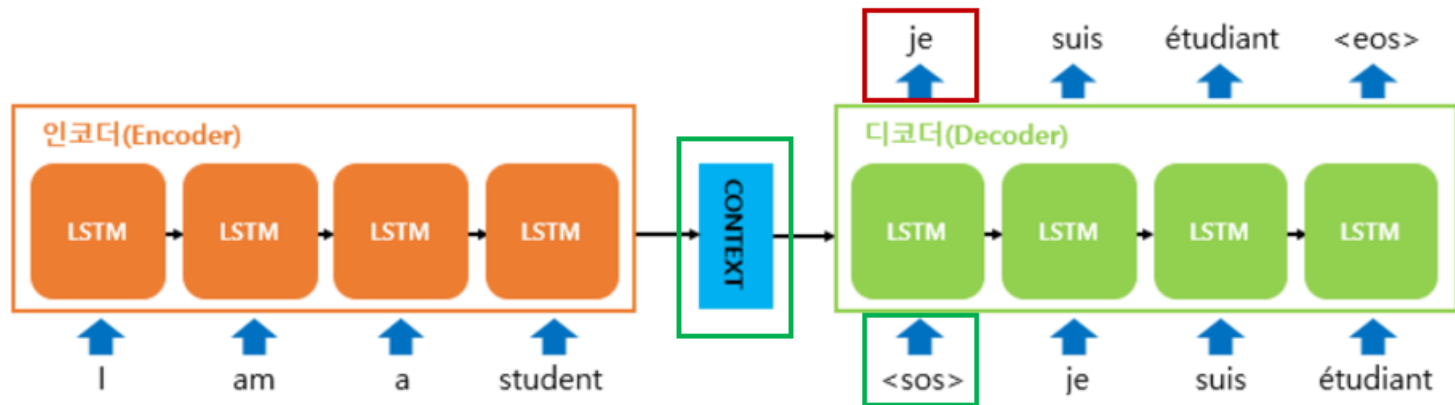
2

Transformer

Decoder : Masked Multi-Head Attention

Masked Multi-Head Attention

기존의 Multi-Head Attention 구조에
마스킹을 추가한 형태



시퀀셜 데이터를 입력으로 받아 예측하기 때문에

미래시점의 데이터를 사용 불가

2

Transformer

Decoder : Masked Multi-Head Attention

Scaled Scores

	<sos>	je	suis	étudiant
<sos>	0.7	0.1	0.1	0.1
je	0.1	0.6	0.2	0.1
suis	0.1	0.3	0.6	0.1
étudiant	0.1	0.3	0.3	0.3

+

Look-Ahead Mask

	<sos>	je	suis	étudiant
<sos>	0	$-\infty$	$-\infty$	$-\infty$
je	0	0	$-\infty$	$-\infty$
suis	0	0	0	$-\infty$
étudiant	0	0	0	0

=

Masked Scores

	<sos>	je	suis	étudiant
<sos>	0.7	$-\infty$	$-\infty$	$-\infty$
je	0.1	0.6	$-\infty$	$-\infty$
suis	0.1	0.3	0.6	$-\infty$
étudiant	0.1	0.3	0.3	0.3

⋮

미래 시점에 $-\infty$ 를 부여해 softmax 연산시
0을 만들어 예측을 위한 입력에서 제외

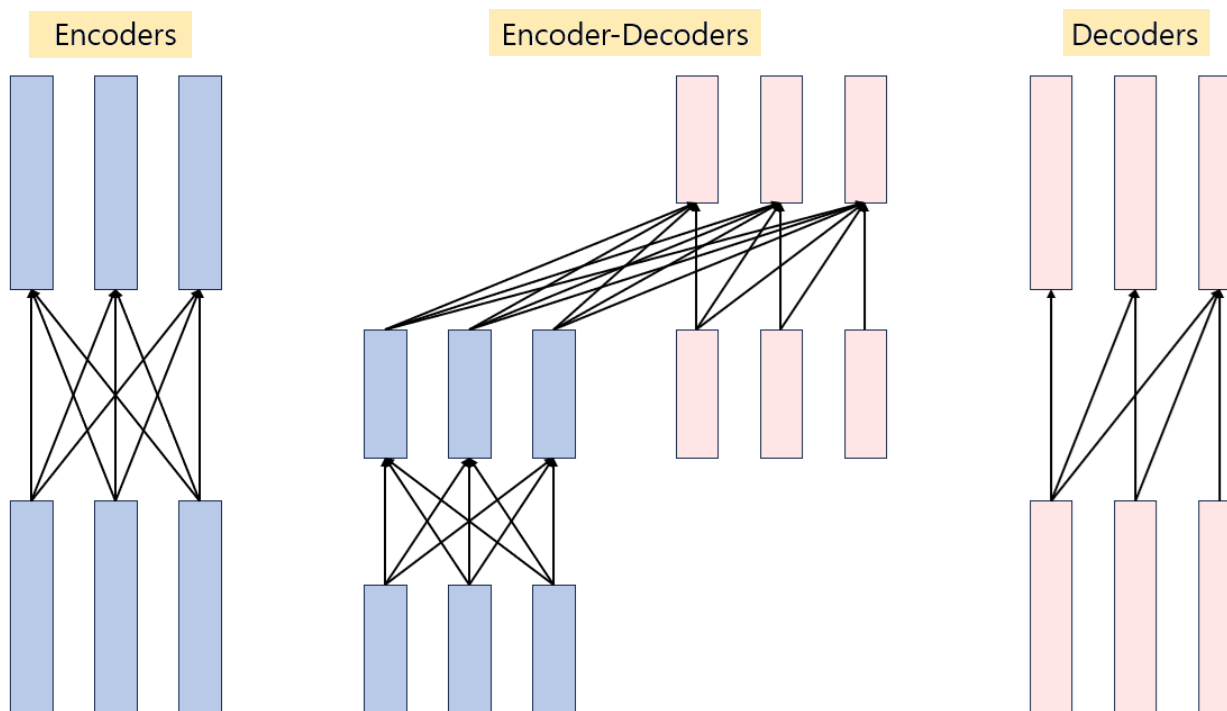
3

Pretrained Model

사전 학습 모델의 세 가지 타입

Transformer는 기본적으로 Encoder-Decoder 구조

→ 사전 학습 모델은 이를 분해한 세 가지 타입으로 구분



모두 비지도 학습 방법으로 사전 학습을 진행, 지도 학습으로 파인튜닝

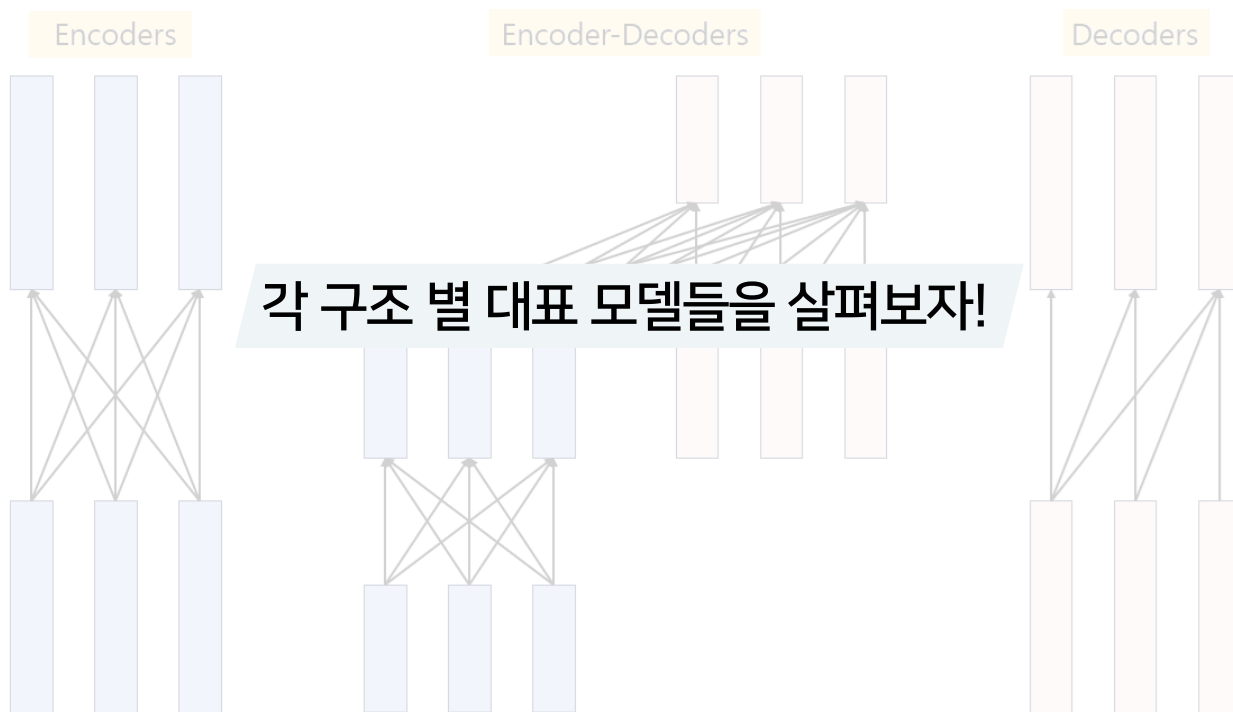
3

Transformer-based Pretrained Model

사전 학습 모델의 세 가지 타입

Transformer는 기본적으로 Encoder-Decoder 구조

→ 사전 학습 모델은 이를 분해한 세 가지 타입으로 구분



모두 비지도 학습 방법으로 사전 학습을 진행, 지도 학습으로 파인튜닝

Decoders - GPT

GPT(Generative Pretrained Transformer)

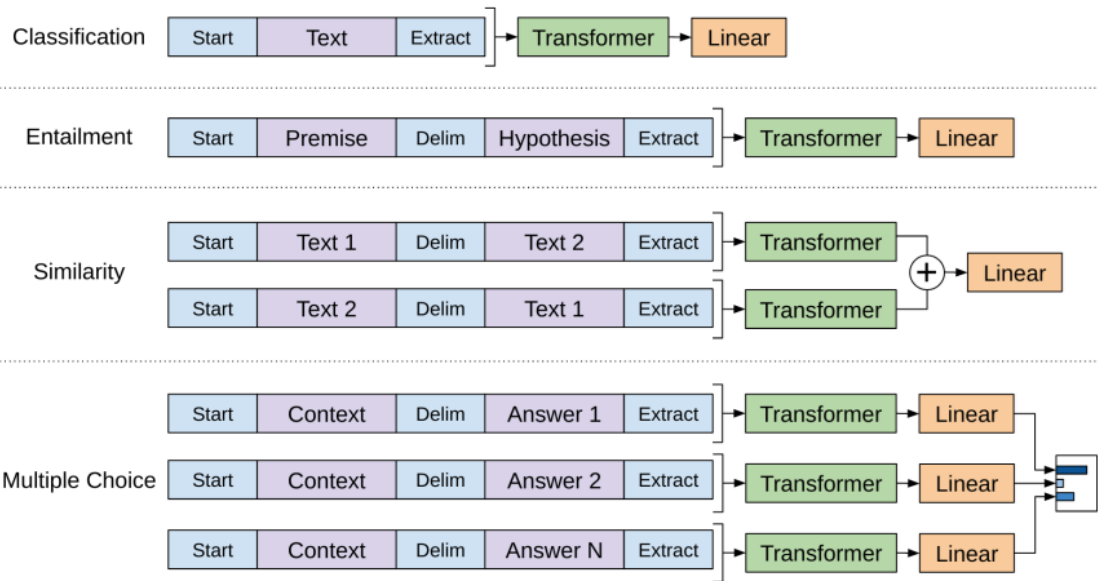
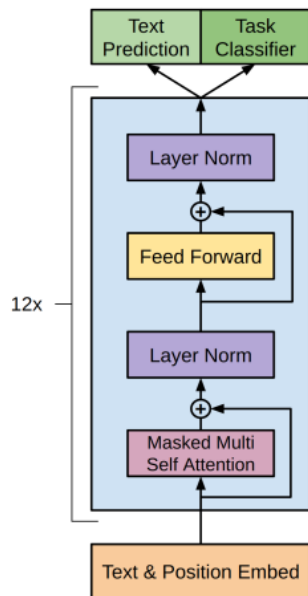
디코더 구조의 대표적인 사전 학습 모델로,
Transformer 구조를 사용하여 문장을 학습

디코더의 목표는 **문장 생성**이므로, 언어 모델링을 위한 구조로 적합

GPT 이전에는 LSTM 셀을 이용한 모델이 일반적

→ GPT는 Transformer 셀을 활용하여 크게 성능을 향상

Decoders - GPT



사전 학습이 끝난 GPT는 각 문제에 따라 입력을 바꾸어
다양한 문제에 적용할 수 있도록 파인 튜닝

Decoders – GPT-3

	GPT-1	GPT-2	GPT-3
Parameters	117 Million	1.5 Billion	175 Billion
Decoder Layers	12	48	96
Context Token Size	512	1024	2048
Hidden Layer	768	1600	12288
Batch Size	64	512	3.2M

GPT-3는 이전 모델들에 비해 전체적인 사이즈가 커졌음

Few-Shot(FS) Learning이라는 새로운 학습 방법 도입



파인 튜닝과 다른 새로운 학습 방법!

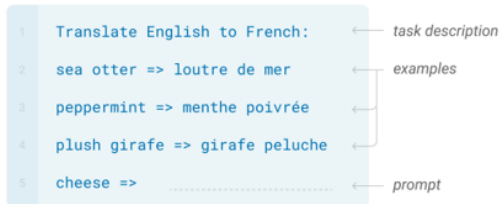
Decoders – GPT-3

Few-Shot(FS) Learning

몇 개의 예시를 instruction으로 주고, 이로부터 학습

Few-shot

In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

**Fine-tuning**

The model is trained via repeated gradient updates using a large corpus of example tasks.



3

Transformer-based Pretrained Model

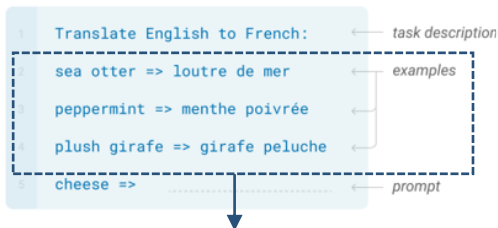
Decoders – GPT-3

Few-Shot(FS) Learning

몇 개의 예시를 instruction으로 주고, 이로부터 학습

Few-shot

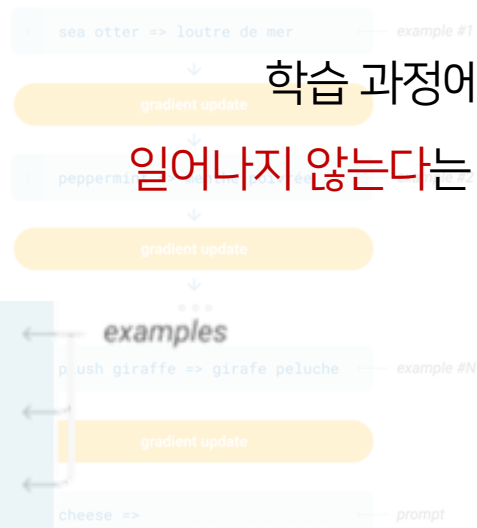
In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.



sea otter => loutre de mer
peppermint => menthe poivrée
plush girafe => girafe peluche

Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.



학습 과정에서 Gradient update가

일어나지 않는다는 점에서 파인 튜닝과 차이점 존재

Decoders – GPT-3

FS Learning이 파인 튜닝에 비해 가지는 장단점

장점

- ① 파인 튜닝을 위한 task-specific 데이터의 요구량이 확연히 줄어듦
- ② task-specific 데이터를 바탕으로 한 파인튜닝을 통해 모델이 지나치게 좁은 분포를 가지는 것을 방지

단점

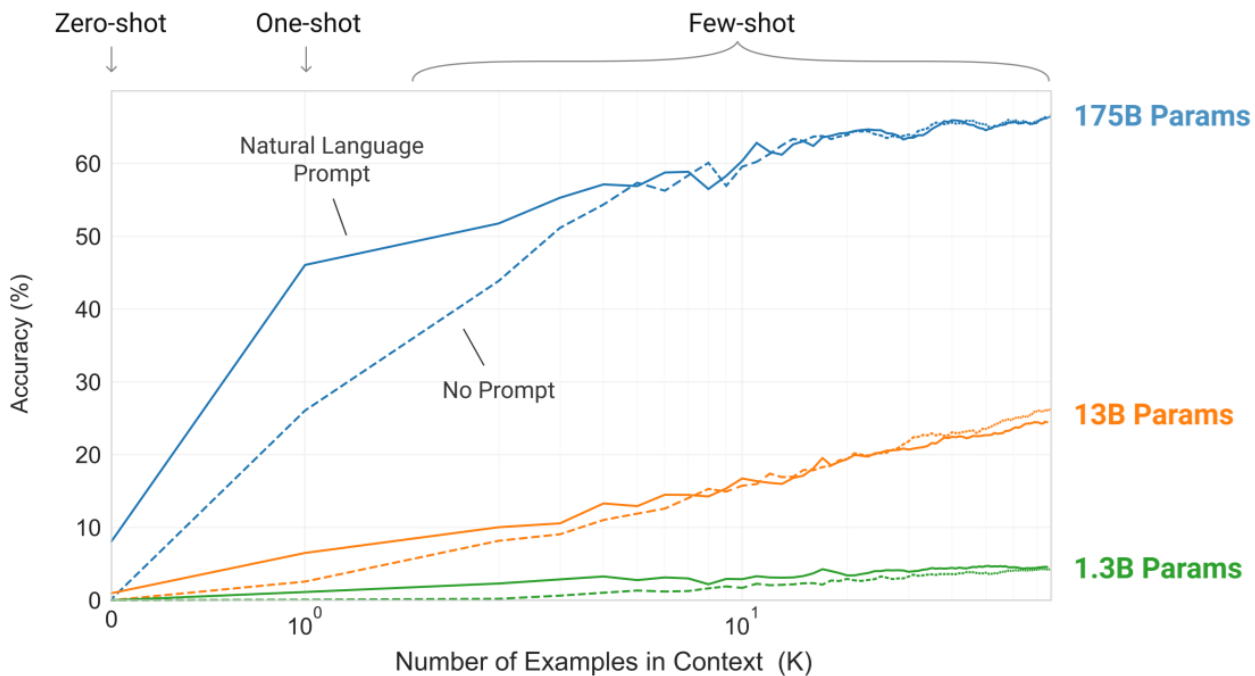
- ① 파인 튜닝에 비해 성능이 훨씬 안 좋음
- ② 여전히 task-specific 데이터를 조금은 필요로 함

3

Transformer-based Pretrained Model

Decoders – GPT-3

파인 튜닝보다는 성능이 안 좋지만, 특정 문제에 특화된 데이터셋이 거의 없더라도 파인 튜닝과 비슷하게 성능을 낼 수 있음



어떠한 예시도 주지 않은 Zero-shot에 비해 Few-Shot이 훨씬 좋은 성능을 가짐

3

Transformer-based Pretrained Model

Encoders

bidirectionality
(양방향성)

...

시퀀셜 데이터 전체의 맥락을 학습 가능
→ 자연어 추론 문제, 질의 응답 문제에 적용

자연어 추론 예시

P: 저는, 그냥 알아내려고 거기 있었어요.

H: 나는 처음부터 그것을 잘 이해했다.

→ CONTRADICTION

질의 응답 예시

Q: 1962년 아시안 게임이 어디서 열렸어

A: 자카르타

Encoders - BERT

BERT

Bidirectional Encoder Representations from Transformers,
2018년 구글에서 Transformer의 인코더를
바탕으로 학습한 사전 학습 모델

인코더의 양방향성으로 시퀀셜 데이터의 전체 맥락을 학습
→ 이는 언어모델링에 큰 도움이 될 것!

3

Transformer-based Pretrained Model

기존 언어모델링은 왜 양방향성을 활용하지 못 했을까?

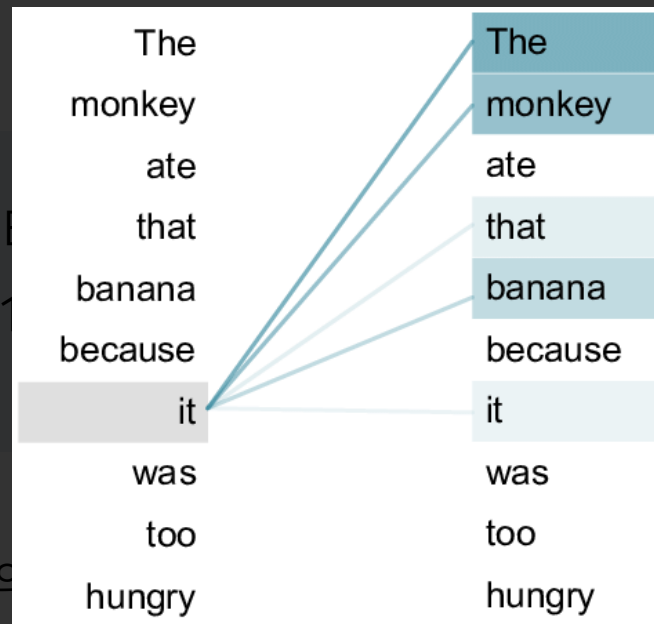
Encoders- BERT

BERT

Bidirectional

201

인코더



→ 이는 언어모델링에 큰 도움이 될 것!

문장 전체에 대한 attention 연산의 결과는 간접적으로 각 단어를 보는 것과 같음

→ 이는 미래 시점의 데이터를 예측해야 하는 데 있어 일종의 치팅이라고 간주

⋮

BERT는 이를 **Masked LM** 구조로 변형함으로써 해결

Encoders - BERT

Masked LM

기존 언어 모델링에서 입력 문장의 일부를
[MASK] 토큰으로 마스킹하고, 마스킹한 단어를 예측

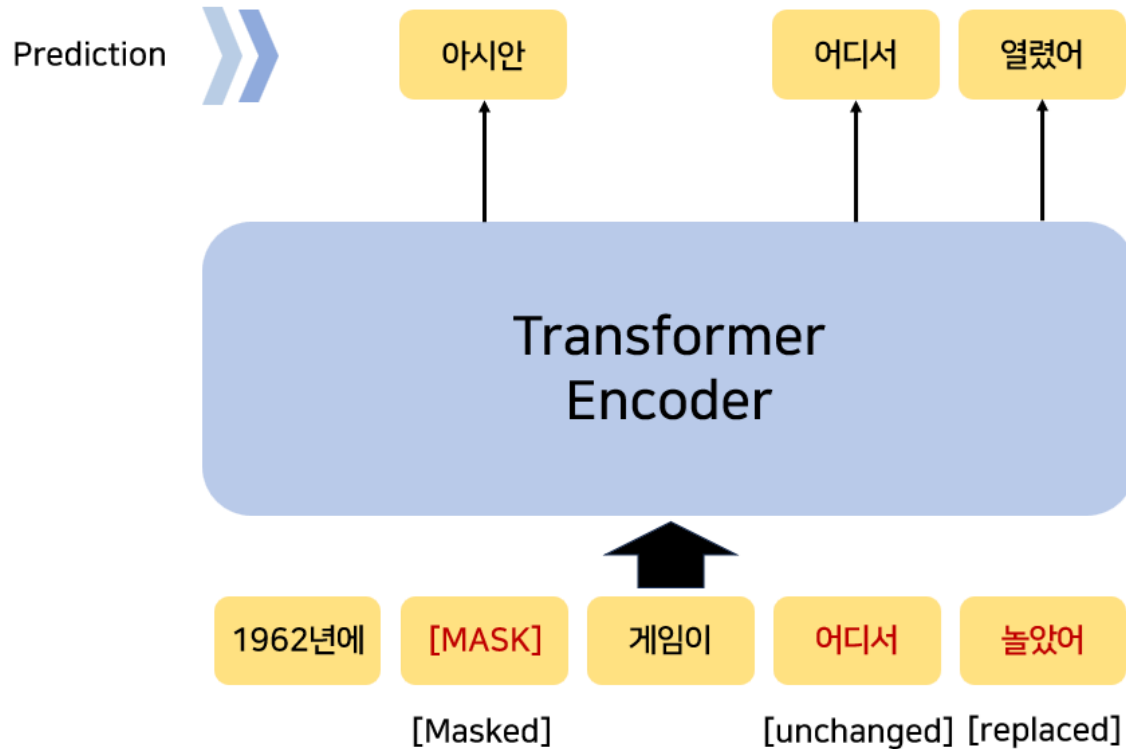
구체적으로, 단어의 부분 집합인 WordPiece를 마스킹하는 것

→ Appendix 참고!

3

Transformer-based Pretrained Model

Encoders - BERT



랜덤 마스킹 비율 = 문장의 15%

[MASK] = 80% / Random word = 10% / Unchanged = 10%

Encoders - BERT

NSP(Next Sentence Prediction)

랜덤한 두 개의 문장을 입력 받아서 앞뒤 문장이
연속되는 문장인지 분류하는 학습



자연어 추론, 질의 응답 등의 NLP 문제는
두 문장 사이의 관계를 이해하는 것이 중요하기 때문

Encoders - BERT

IsNext 예시

Input: [CLS] 나는 어제 마트에 갔다 [SEP] 저녁으로 순두부 찌개를 먹기 위해
재료들을 샀다 [SEP]

Label: IsNext

NotNext 예시

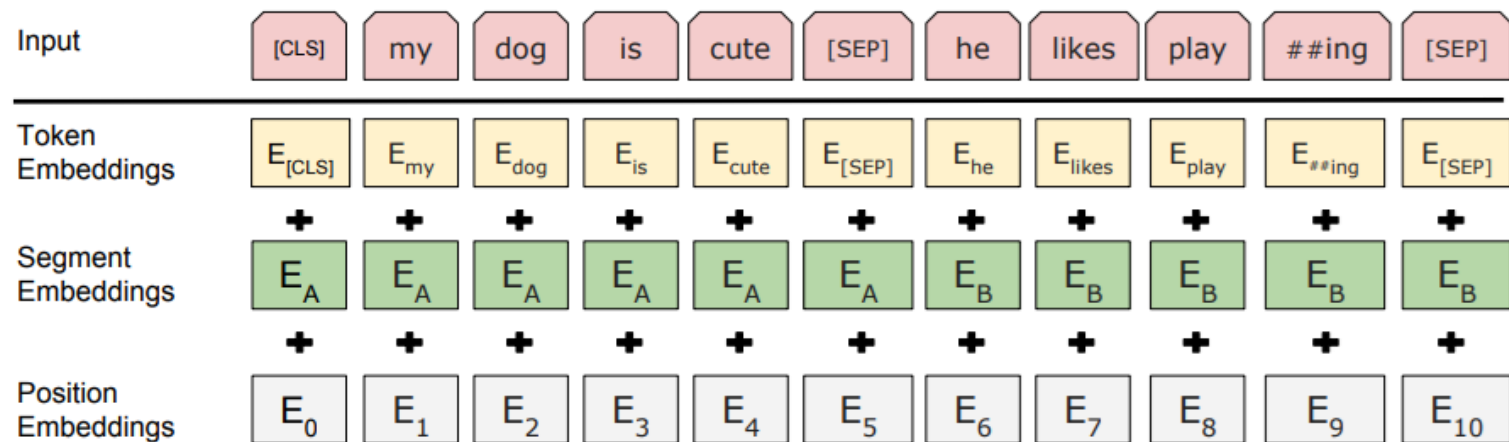
Input: [CLS] 나는 어제 마트에 갔다 [SEP] 도쿄 타워에서 보는 야경은 정말 예뻐다
[SEP]

Label: NotNext

3

Transformer-based Pretrained Model

Encoders - BERT



BERT는 토큰, 세그먼트, 위치를 더하여 최종적인 입력 임베딩을 만듦

Encoders - BERT

BERT의 최종적인 입력의 형태

[CLS] 토큰이 첫 번째로 나온 뒤, 두 개의 문장은 [SEP] 토큰으로 분리

- [CLS]: 모든 BERT 입력의 항상 처음에 등장하여 이 문제가 분류 문제라는 것을 알리는 토큰
- [SEP]: 두 개의 문장을 구분하기 위한 토큰



학습된 추가적인 임베딩인 Segment Embeddings를 추가하여
이 문장이 A인지, B인지 알려줌

- Segment Embeddings: BERT에서 사용되는 추가적인 임베딩.

Encoder-Decoders – T5

T5 (Text-To-Text Transfer Transformer)

기존의 BERT 모델이 특정 과제만 해결할 수 있는 것에 반해,

T5는 문장 분류, 번역, 요약 등의 모든 과제를

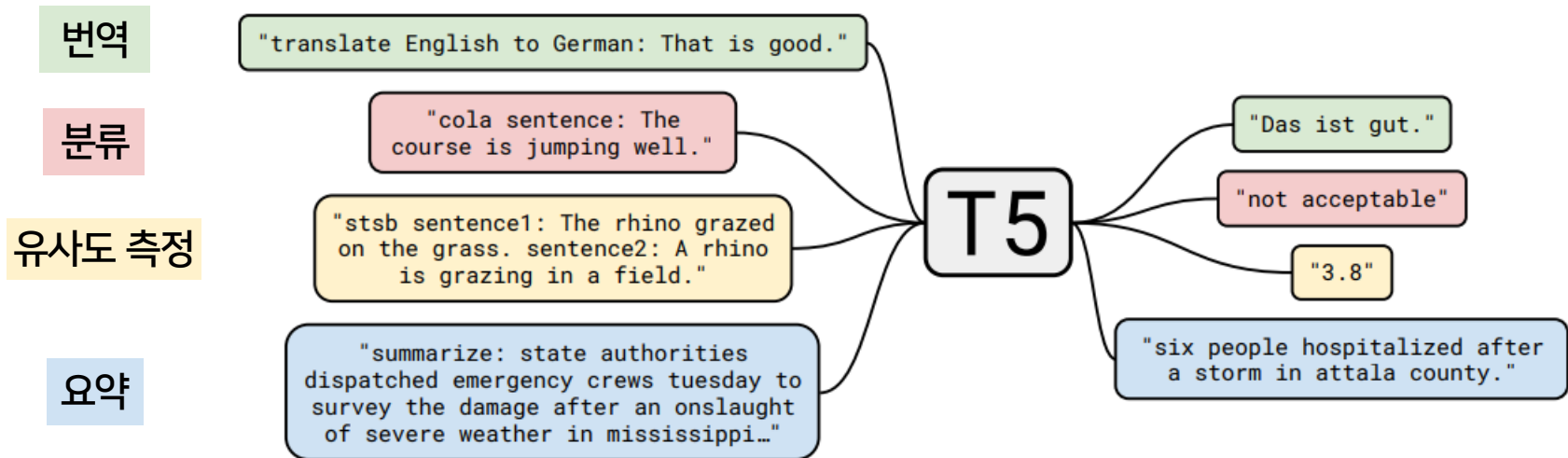
동일한 모델로 해결할 수 있음

폭넓은 과제에 적용하기 위해 비지도 학습으로

일반적인 지식들을 학습하기를 바랐음

→ 텍스트 입력으로 텍스트를 출력하는 “Text-To-Text” 를 진행

Encoder-Decoders – T5



T5 모델은 Text-To-Text를 학습하기 위해 🖐️ Prefix를 추가해주고,
BERT의 Masked LM을 변형한 🖐️ Span Corruption을 적용

Encoder-Decoders – T5



Prefix 추가

EX 1) Translation

Input – *Translation English to German*: That is good. → **Output** → Das ist gut.

EX 2) STS (Semantic Textual Similarity)

Input – *stsb sentence 1*: The rhino grazed on the grass.*stsb sentence 2*: A rhino is grazing in a field.**Output** → 3.8

작업에 대한 지시 사항을 줌으로써 모델이 문제를 파악할 수 있음

3

Transformer-based Pretrained Model

Encoder-Decoders – T5



Span Corruption

Original text

Thank you ~~for inviting~~ me to your party ~~last~~ week.

Inputs

Thank you <X> me to your party <Y> week.

Targets

<X> for inviting <Y> last <Z>

Masked LM과 동일하게 마스킹을 해주지만, 두 가지 다른 점 존재

Encoder-Decoders – T5



Span Corruption

① only dropout

마스킹한 단어를 오직 Dropout만 해줌.

이는 사전 학습 과정에서 계산 비용을 줄이기 위함임.

② 연속된 단어(consecutive span) 처리

연속된 단어는 하나의 토큰으로 처리하여 예측하고,
이렇게 예측한 새로운 스패는 Wordpiece 토큰으로 추가
단어가 아닌 구(phrase) 또한 Wordpiece로 추가됨

Masked LM과 동일하게 마스킹을 해주지만, 두 가지 다른 점 존재

Encoder-Decoders – T5



Span Corruption

Original text

Thank you ~~for inviting~~ me to your party ~~last~~ week.

Inputs

Thank you <X> me to your party <Y> week.

Targets

<X> for inviting <Y> last <Z>

위 예시의 경우, 토큰 "for inviting"은 연속되었기 때문에 하나의 토큰
<X>로 처리한 후 예측하고, Wordpiece로 추가해줌






4

Appendix

OOV(Out-of-Vocabulary)

OOV(Out-of-Vocabulary)

모델이 학습 과정에서 학습하지 못한 단어

	word		vocab mapping	embedding
Common words	hat	→	pizza (index)	
	learn	→	tasty (index)	
Variations	taaaaasty	→	UNK (index)	
misspellings	laern	→	UNK (index)	
novel items	Transformerify	→	UNK (index)	

⋮

학습 과정에서 보지 못했던

단어의 변형, 오타, 신조어 등 새로운 단어 등장










4

Appendix

BPE(Byte-Pair Encoding)

BPE(Byte-Pair Encoding)

subword로 분리한 후 학습을 통해
단어 집합을 만드는 알고리즘

	word		vocab mapping	embedding
Common words	hat	→	hat	
	learn	→	learn	
Variations	taaaaasty	→	taa## aaa## sty	  
misspellings	laern	→	la## ern##	 
novel items	Transformerify	→	Transformer## ify	 



예시의 OOV 문제를 해결 가능

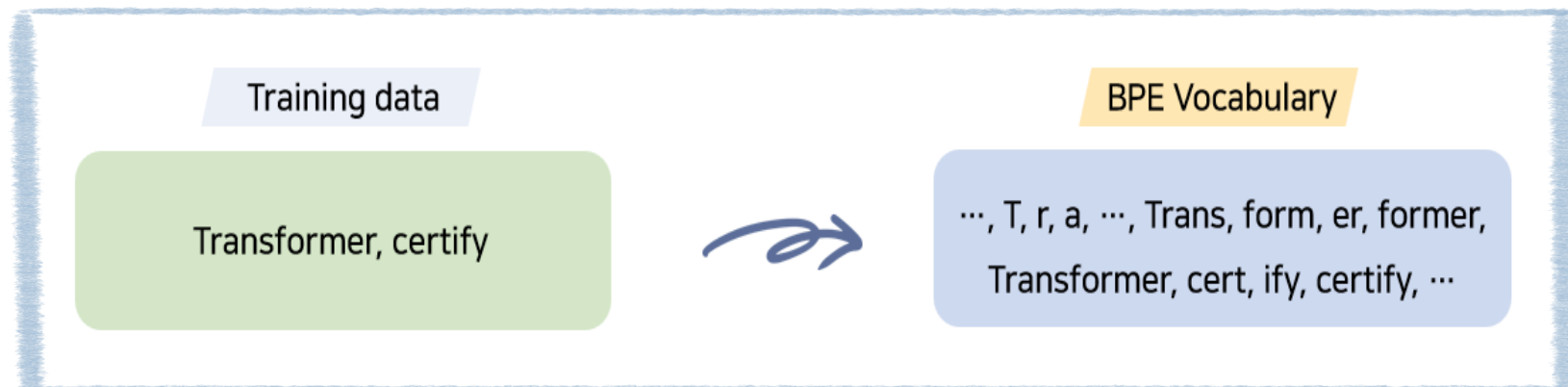
BPE(Byte-Pair Encoding)

BPE Vocabulary

a, b, c, d, e, ..., x, y, z, <eos>

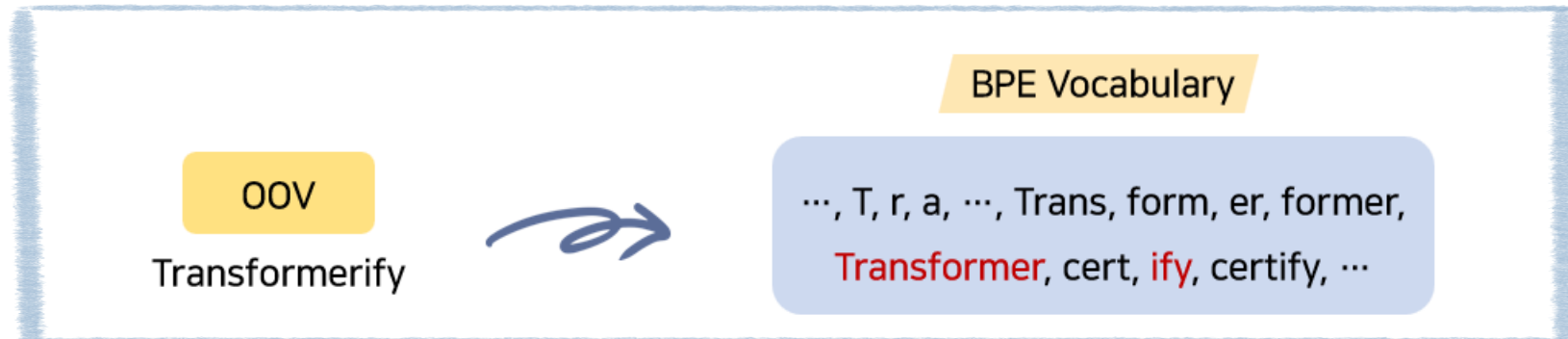
① 알파벳과 "end-of-word" 기호만 포함

BPE(Byte-Pair Encoding)



② 말뭉치 데이터를 이용해 가장 인접한 subword 찾기

BPE(Byte-Pair Encoding)



③ 찾은 subword와 연결하기

BPE(Byte-Pair Encoding)

BPE Vocabulary

..., T, r, a, ..., Trans, form, er, former,
Transformer, cert, **ify**, certify, ...



New subword

Transformer + ify

④ 해당 새로운 subword로 대체

WordPiece

WordPiece

구글에서 Bert 학습을 위해 개발한 알고리즘
subword의 빈도가 적을수록 높은 점수를 부여해
점수가 높은 우선순위로 병합하는 알고리즘

$$score = \frac{freq - of - pair}{freq - of - first - element \times freq - of - second - element}$$

WordPiece

WordPiece

구글에서 Bert 학습을 위해 개발한 알고리즘
subword의 빈도가 적을수록 높은 점수를 부여해
점수가 높은 우선순위로 병합하는 알고리즘



unable에서 un과 ##able은 출현빈도가 높으므로
hugging을 hu와 ##gging을 나눈 것에 비해
나중에 병합

감사합니다
