BERT

CONTENTS

1. 기존 모델과의 차이점

01

* NLP에서 Pre-training model을 사용하는 방법 2가지

1. Feature based - ELMo

- Task-specific한 architecture
- Pre-train한 representation을 additional feature로 넣는 방법
  (=2개의 network를 붙여서 사용)

2. fine-tuning – OpenAI GPT

- BERT 이전 SOTA를 달성한 model인 GPT가 사용한 방법
- Task-specific한 parameter를 최소화하여 fine-tuning
- bidirectional하려고 노력했으나, 결국은 [단방향 concat 단방향]

1. 기존 모델과의 차이점

01

- **BERT**

→ 이전의 언어모델과는 달리,

unlabeled data(wiki, book data 등)으로 model을 pre-training한 후

특정 Task를 가진 labeled data로 fine-tuning

→ 특정 Task를 위한 network 붙일 필요 X

→ pre-trained BERT Model은 1개의 output layer로 fine-tuning할 수 있다

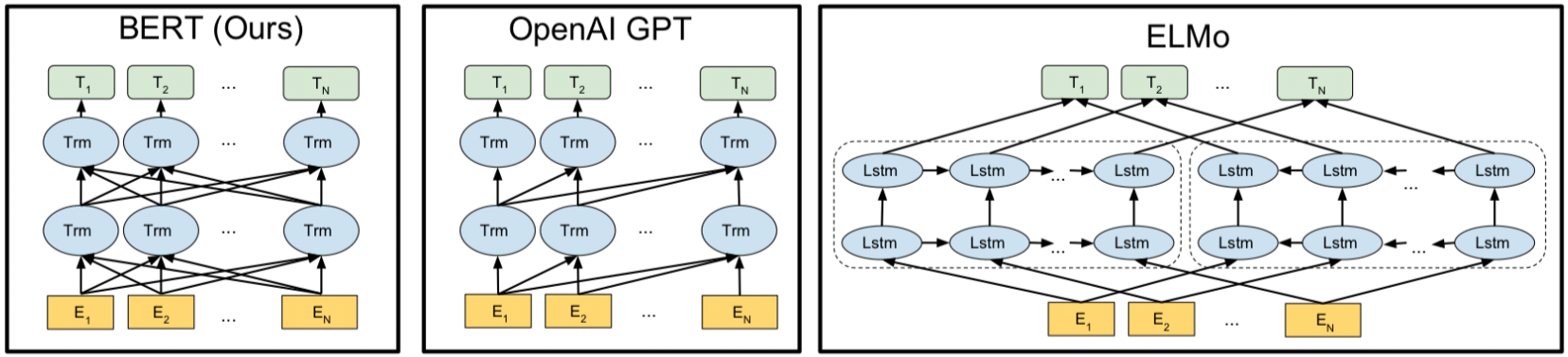
1. 기존 모델과의 차이점

- **BERT**

1) unidirectional이 아닌 bidirectional

2) pretraining의 새로운 방법론 2가지 제시

3) 대규모 dataset으로 pre-training, 적은 양의 labeled dataset에 대해 fine-tuning



1. 기존 모델과의 차이점

02

1) unidirectional이 아닌 bidirectional

- 기존

: 앞의 단어 n개의 단어를 가지고 뒤의 단어를 예측 → unidirectional

- BERT

: 주변 단어들을 보고 Masked 단어를 예측 → bidirectional

2) BERT의 Pre-training의 새로운 방법론 2가지

1. Masked Language Model

: 주변 단어들을 보고 Masked 단어를 예측

2. Next Sentence Prediction Task

: 문장들 사이의 관계를 학습하기 위해 "다음 문장이라는 Label" 추가

1. 기존 모델과의 차이점

02

2-1. Masked Language Model

: input 에서 random하게 몇 개의 Token을 Mask

: Mask 처리한 sequence를 Transformer의 Encoder에 넣어서 주변 단어 Context를 보고 Mask 된 Token을 예측하는 Model

2-2. Next Sentence Prediction Task

: 두 문장을 Pre-training때 함께 넣어서 두 문장이 이어지는 문장인지 아닌지 맞추게 하는

: Pre-training시,

실제로 이어지는 두 문장 : 랜덤한게 추출한 두 문장 = 50 : 50

2. Preprocessing

02

2.1. Segmentation

: input 은 Token의 집합

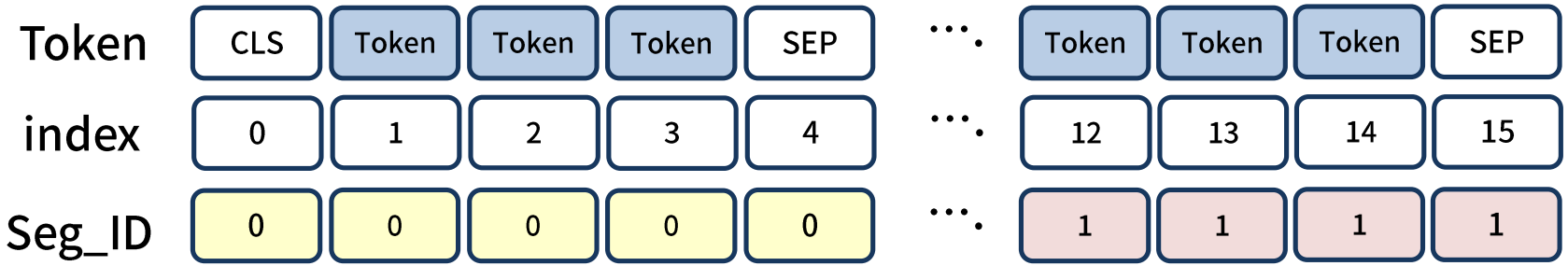
→ 한 batch에서 Token A, Token B 2가지로 나뉘어서 input으로!

: sequence에는 [CLS],[SEP],[SEP]이 포함되어야 함

→ [CLS] : Task-specific한 정보를 주기위한 Token

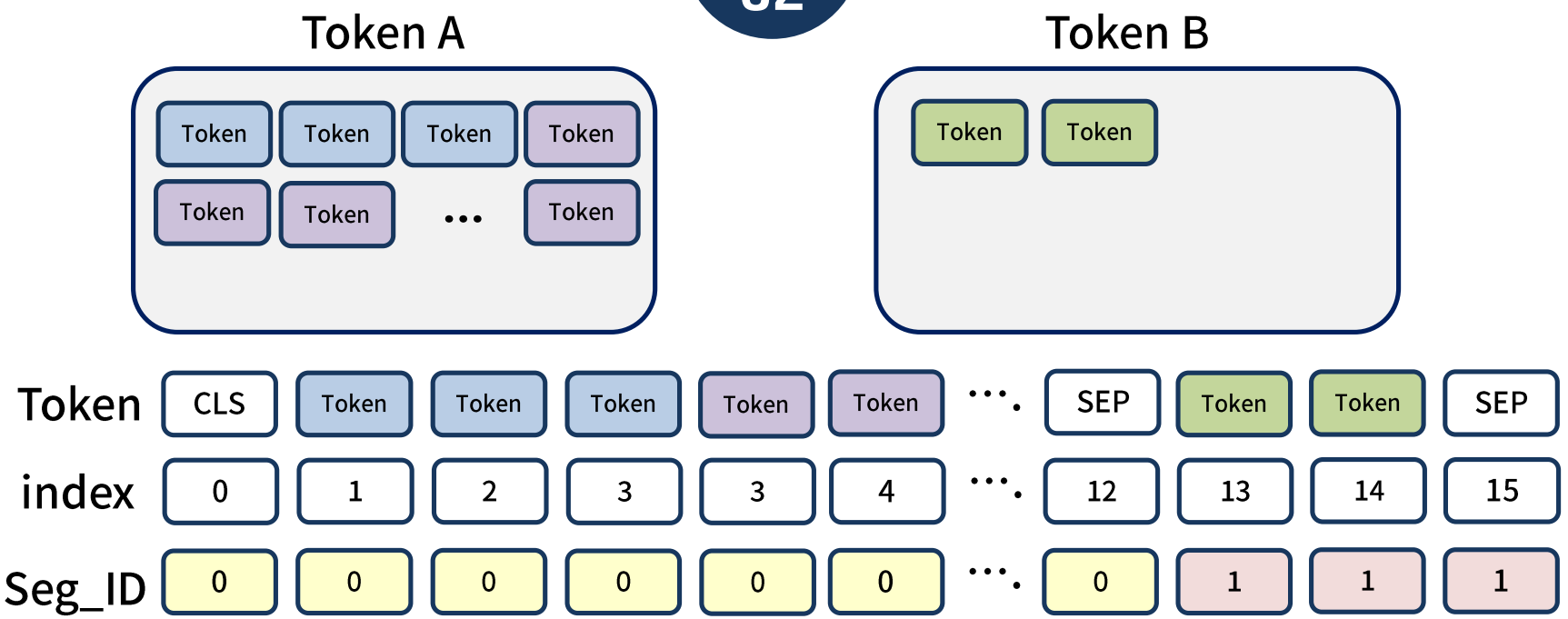
→ [SEP] : Token A,B를 구분하는 Token

→ [SEP] : Token A, B의 끝을 알리는 Token



2. Preprocessing

02



*** index**

: positional encoding할 때 사용되는 index

→ 어느 부분이 masked token인지(예측해야 하는 부분)알 수 있다

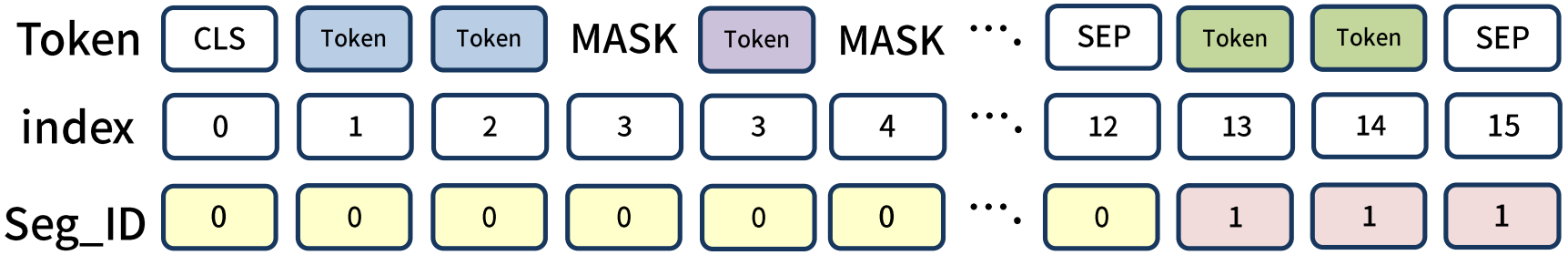
***Seg_ID**

: Token A → 0, Token B → 1

2. Preprocessing

02

2-2. Next Sentence



- **is_next**

  : True → 다음 문장인지 아닌지

- **Masked_P**

  : 4, 10 → MASK된 Token의 index

- **Label**

  : Single, double

2.3 Length & Mask

- **Mask Prediction**

  : 15%를 Mask 하지만 예측할 Mask는 최대 20개로 설정

  # of masks = Min(20, Max(1, (# of Tokens) x 0.15))

  : 15% 중에서, 80%→ MASK

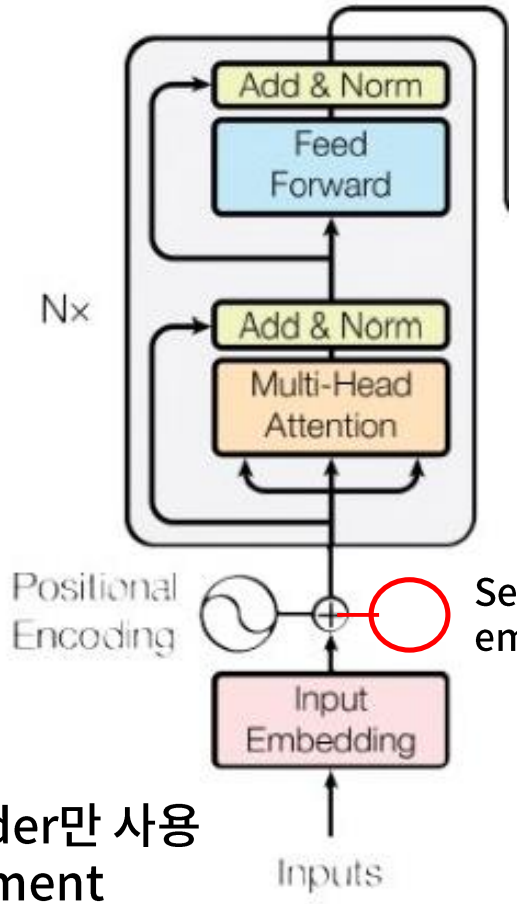
  10%→ random Token으로 대체

  10%→ 원래 Token

3. BERT 구조

03

: Transformer의 Encoder 부분만 사용



[BERT]
: Encoder만 사용
(+ Segment Embedding)

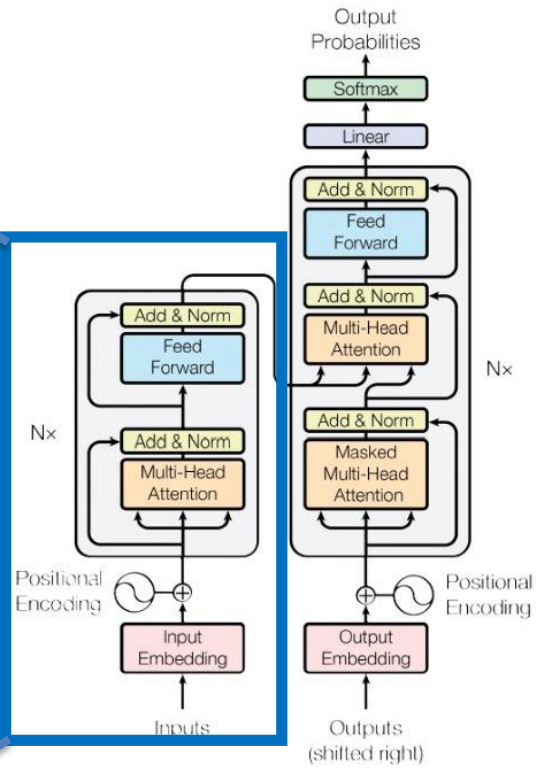
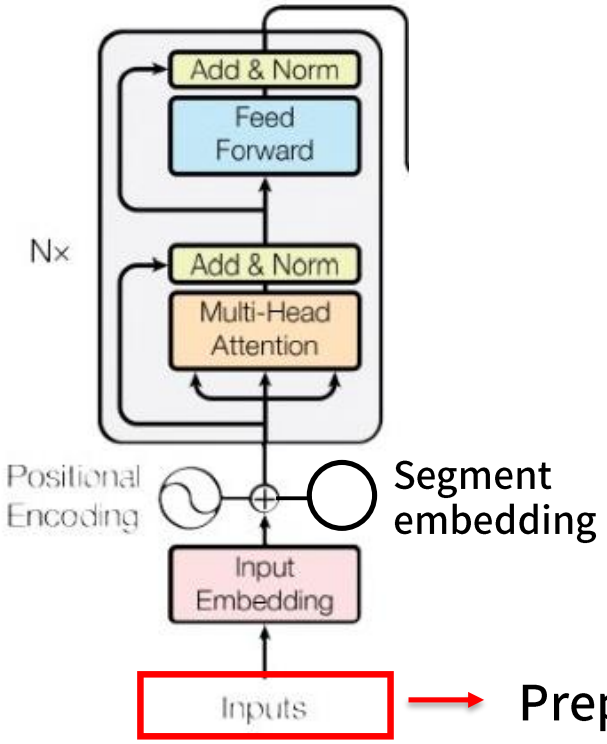


Figure 1: The Transformer - model architecture.

[transformer Model architecture]
: Encoder-Decoder

3. BERT 구조

03



Preprocessing

→ [CLS] : Task-specific한 정보를 주기위한 Token

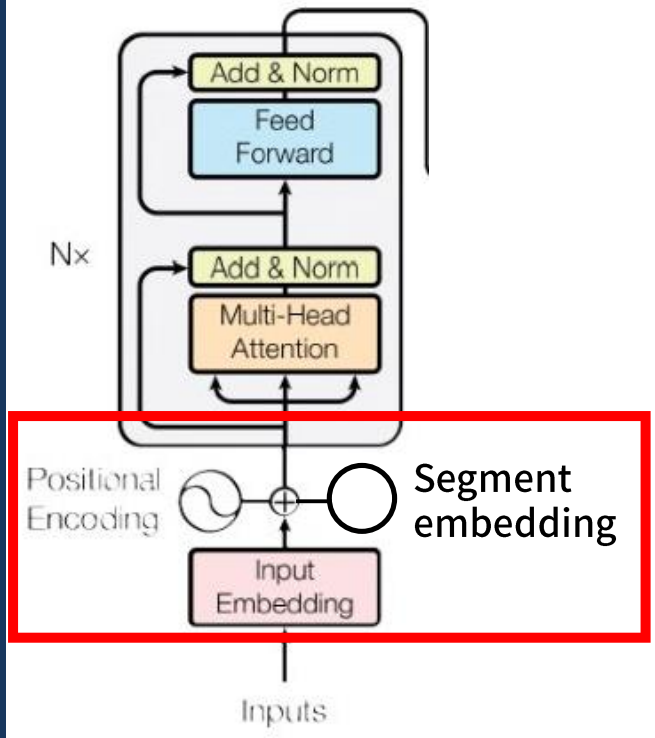
→ [SEP] : Token A,B를 구분하는 Token

→ [SEP] : Token A, B의 끝을 알리는 Token

→ [MASK] : 예측하는 Target

3. BERT 구조

03

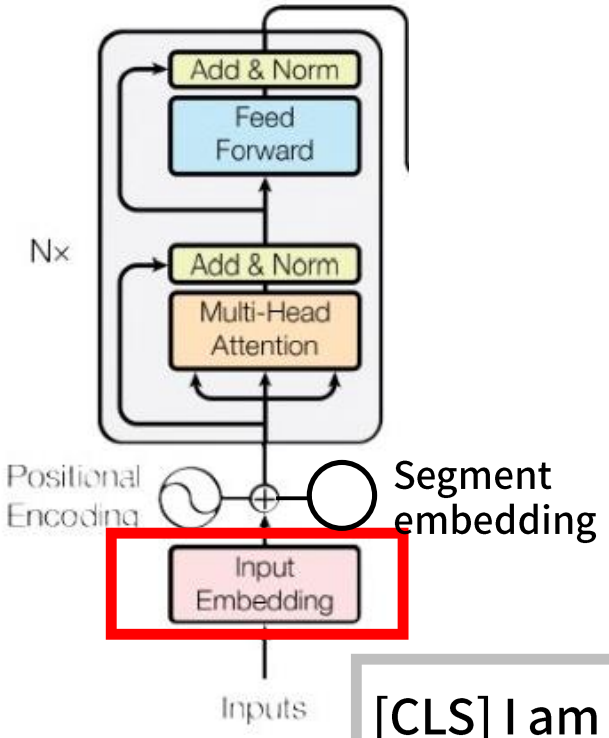


3가지의 Embedding

1) Input embedding
2) Positional embedding
3) Segment embedding

3. BERT 구조

03



1) Input embedding

: one-hot vector를 Weight matrix랑 곱한다.

: 그 결과 → embedding vector

[CLS] I am looking for happiness [SEP] B type sentence [SEP]

looking x_j

Vocab size(Encoder)

⟨PAD⟩	⟨S⟩	⟨UNK⟩	am	for	···	i	looking	···	w	···	z
0	0	0	0	0	0	0	1	0	0	0	0

Padding값의 경우 코드상에서 학습 되지 않도록 일정한 값으로 고정하고!

Embed size

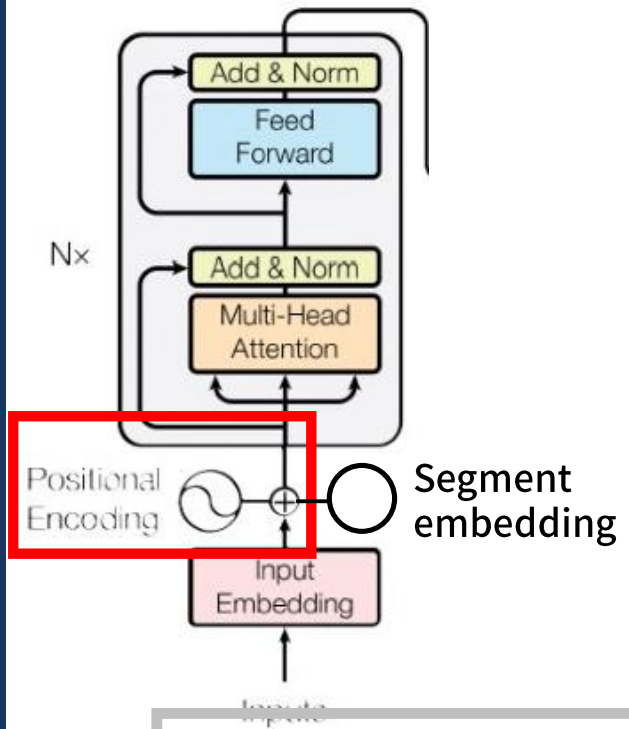
⟨PAD⟩	0.0	0.0	0.0	0.0
⋮	0.0	0.3	0.7	0.0
looking	0.2	0.1	0.7	0.2
⋮	0.9	0.1	0.7	0.9
	0.5	0.7	0.1	0.5
	0.2	0.9	0.4	0.2
z	0.2	0.0	0.7	0.2

0.2 0.1 0.7 0.2 w_j looking

Input embedding 결과

3. BERT 구조

03



2) **Positional encoding**

: 각 단어의 위치정보를 저장하기 위해

: 단어의 embedding vector+위치정보

: position에 대한 one-hot vector를 Weight matrix와 곱한다

[CLS] I am looking for happiness [SEP] B type sentence [SEP]

3th position

Position vocab

0	1	2	3	4	5
0	0	1	0	0	0

Embed size

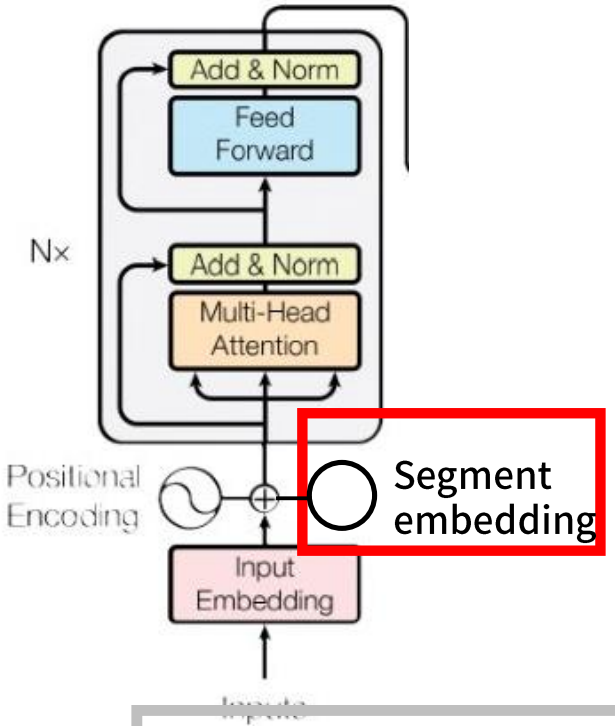
0	0.1	0.4	0.7	0.1
⋮	0.0	0.3	0.7	0.0
2	0.1	0.1	0.9	0.1
⋮	...	...	...	...
5	0.2	0.0	0.7	0.2

3th position embedding

0.1	0.1	0.9	0.1

3. BERT 구조

03



3) **Segment embedding (Token embedding)**

:Token A group, Token B group을 구분하는 정보

[CLS] I am looking for happiness [SEP] B type sentence [SEP]

looking x_j

Vocab size(Encoder)

A	B
1	0

Embed size

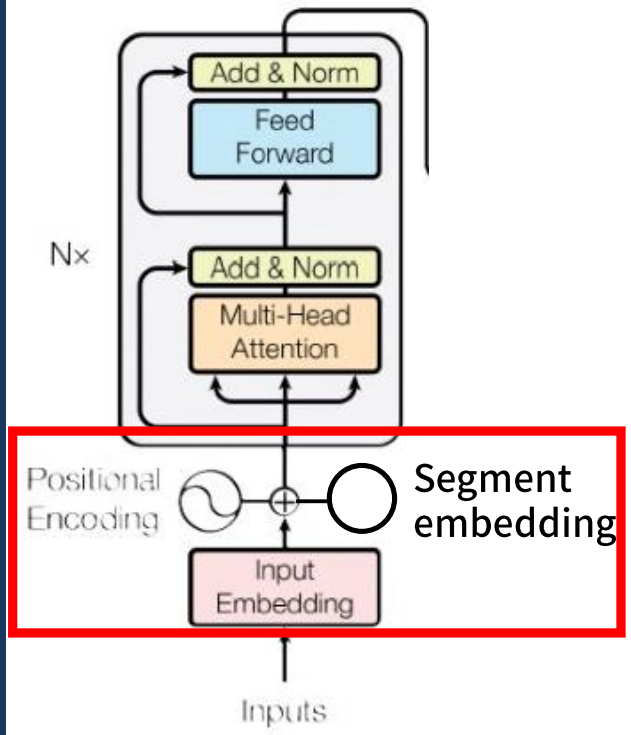
A				
B				

0.1	0.1	0.1	0.3

w_j looking

3. BERT 구조

03



Input embedding 결과

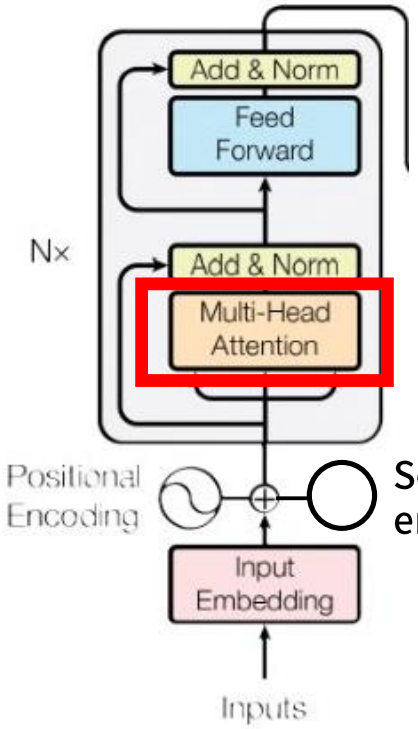
+ Positional embedding 결과

+ Segment embedding 결과

→ Encoder의 입력

3. BERT 구조

03



: Attention(Q*K)을 scaling해준다.

: Key Masking만 사용

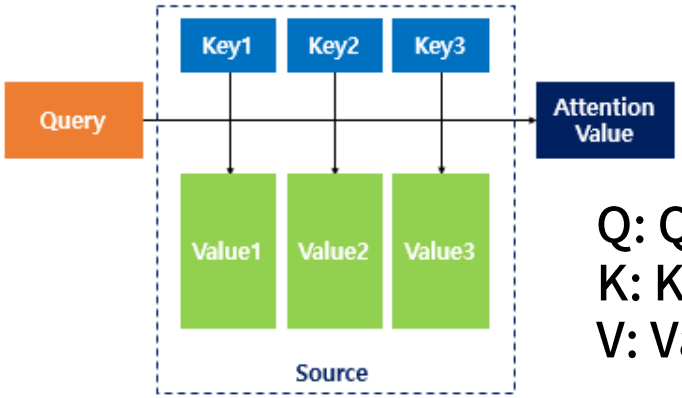
Attention

03

- **Attention (transformer)**

  : decoder에서 매 시점마다 Encoder의 전체 입력문장을 다시한번 참고하는 것

  : 문장 전체 참고 X → 해당 시점에서 예측해야 할 단어와 연관이 있는 부분만



Q: Query→ t시점의 decoder 셀에서의 은닉상태
K: Key → 모든 시점의 encoder 셀의 은닉상태
V: Value → 모든 시점의 encoder 셀의 은닉 상태

- **Self-Attention**

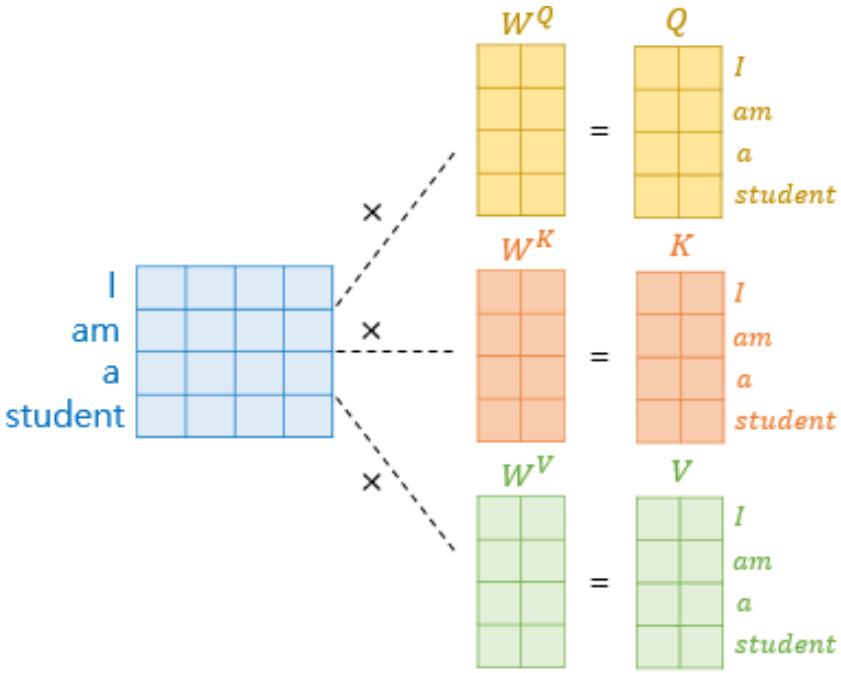
  : attention을 자기 자신에게 수행하는 것

  : 효과→ self-attention은 입력문장내의 단어들끼리 유사도를 구할 수 있음

Attention

03

- 각 Q 벡터는 모든 K벡터에 대한 Attention Score 계산→ Attention distribution 계산→ 모든 V 벡터를 가중합 계산→ Attention value(context vector) 계산
- 각 단어에 대한 Q, K, V vector를 구해야 하는데, 이를 행렬연산으로 동시에 구할 수 있다.

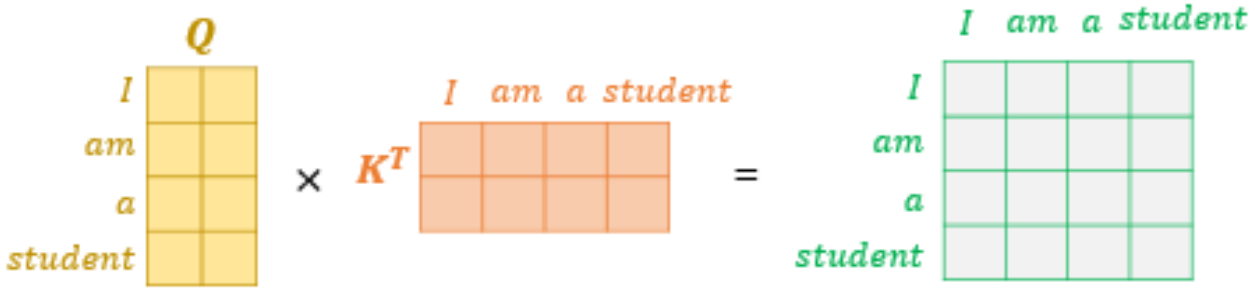


문장 행렬에 가중치 행렬을 곱해서 Q, K, V 행렬을 구함

Attention

03

- **Attention score 계산**



$Q \times K^T$ → 각 단어의 Q 벡터와 K 벡터의 내적이 각 행렬의 원소가 되는 행렬

: 이 행렬의 값에 전체적으로 $\sqrt{d_k}$를 나누어 주면 각 행, 열이 Attention score를 가지게 됨

: d_k→ d_{model}/num_heads

Attention

03

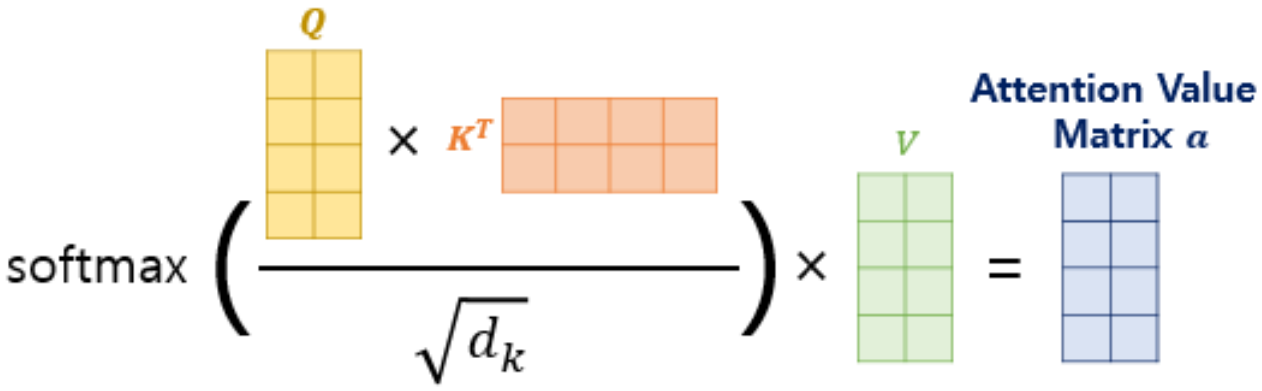
- **Attention distribution 계산**

  : 이를 이용해 모든 단어에 대한 Attention 값을 구해야 함

  : Attention Score에 Softmax를 적용하면 그 결과가 Attention distribution

- **Attention value Matrix 계산**

  : Attention distribution에 V행렬을 곱하면 그 결과가 Attention Value Matrix



Attention

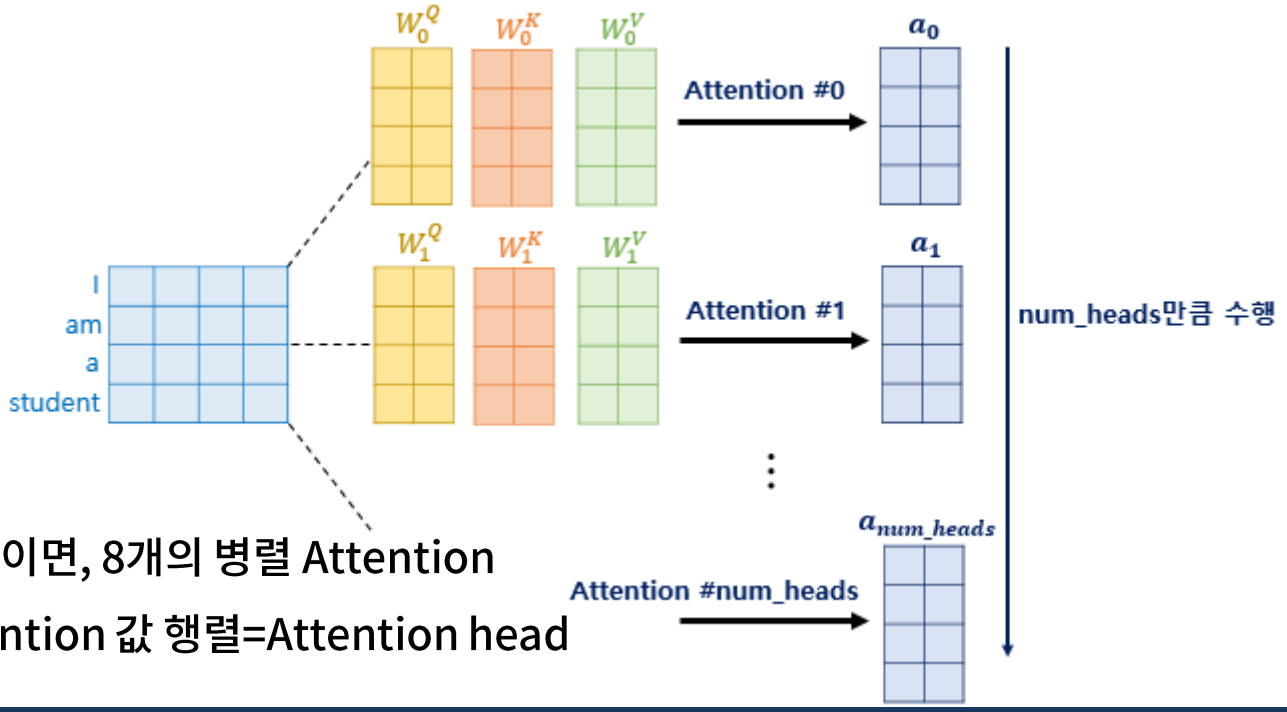
03

- **Multi-head Attention**

  : 한번의 Attention보다 여러 번 하는 것이 더 효율적

  : dmodel의 차원을 num_heads개로 나누어 (dmodel/num_heads)차원을 가지는 Q, K, V에 대해 num_heads개의 병렬 Attention을 수행

  : 가중치 행렬 W^Q, W^K, W^V 도 Attention head마다 다 다름



*Attention head

: num_heads가 8이면, 8개의 병렬 Attention

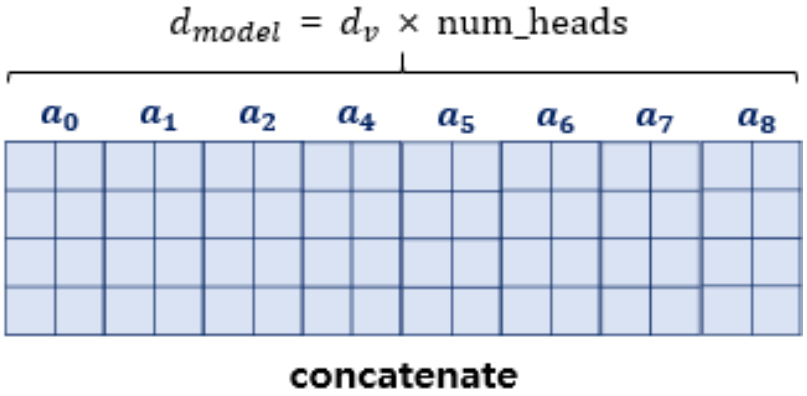
→ 이때 각각의 Attention 값 행렬=Attention head

Attention

03

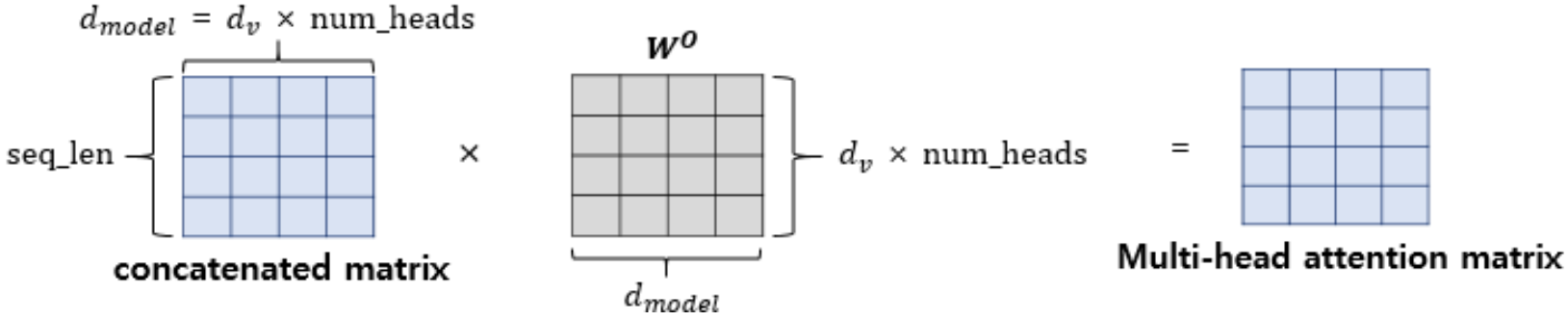
- **Multi-head Attention**

  : 병렬 Attention을 모두 수행했으면 모든 Attention head를 concatenate



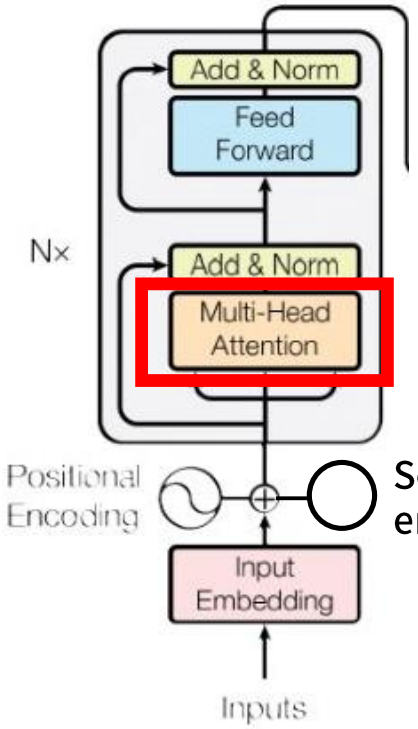
- **Concat한 행렬을 또 다른 가중치 W^0와 곱한다**

  : 그 결과 행렬이 Multi Attention matrix



3. BERT 구조

03



: Attention(Q*K)을 scaling해준다.

: Key Masking만 사용

Masking

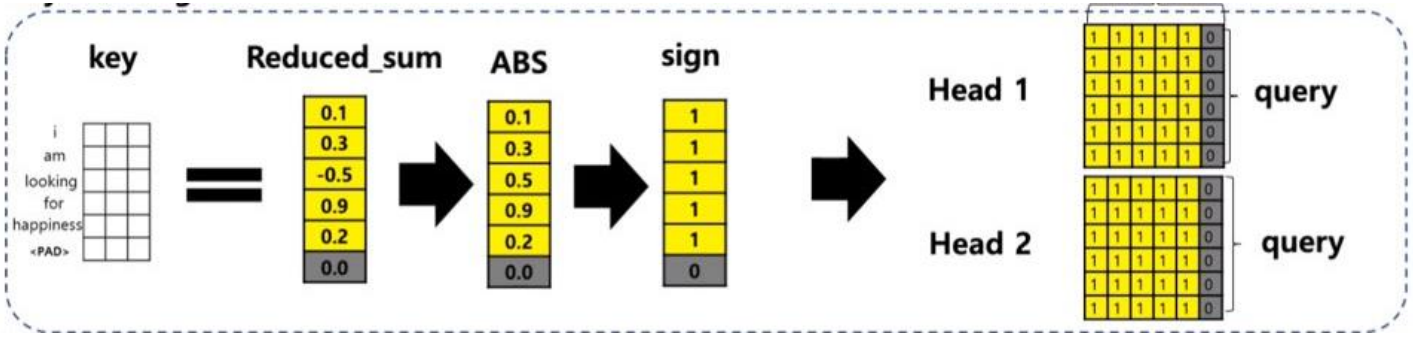
03

: [PAD]인 부분은 0, [PAD]가 아닌 부분은 1로 Masking

: Transformer에서는 Key, Query에 Masking

BERT에서는 Key에만 Masking

Key Masking



Masking

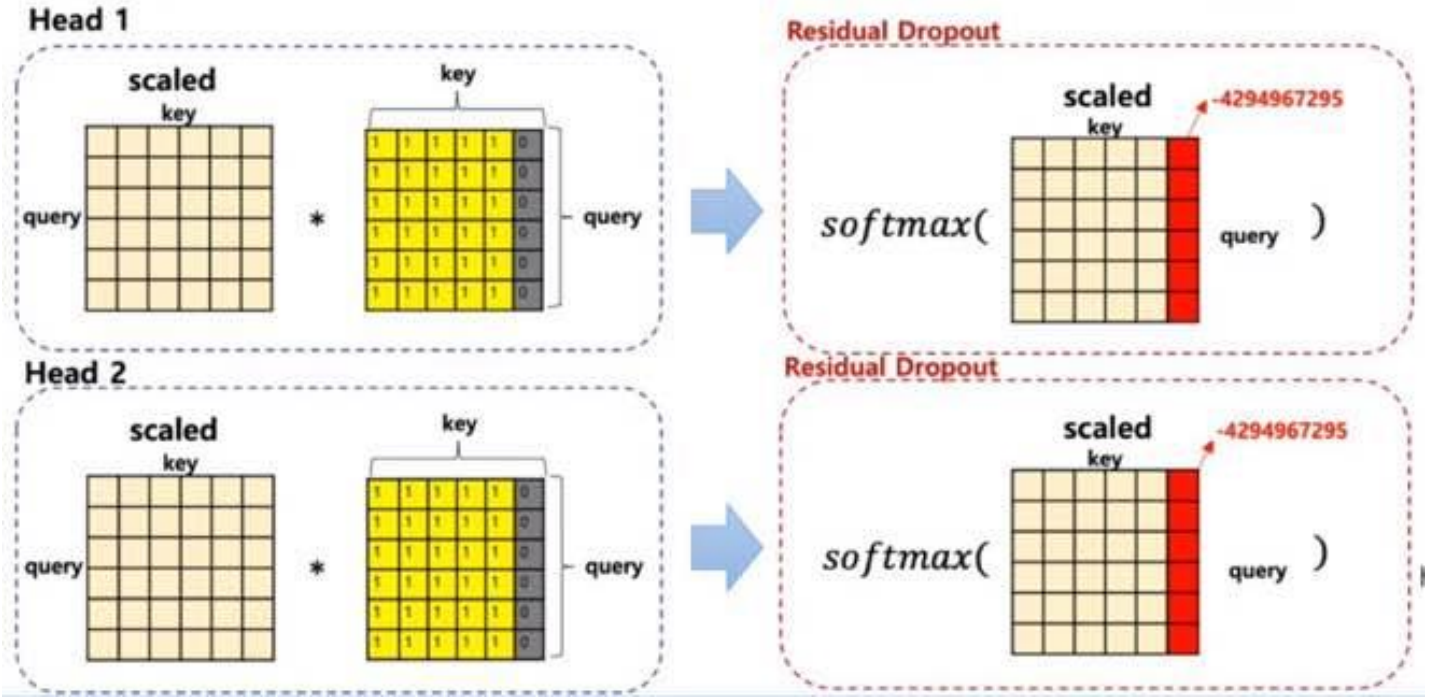
03

: Key Masking한 결과 = [PAD]가 아닌 부분만 1인 matrix

: 이 matrix를 scaled attention weight에 적용

: 0인 부분에 큰 음수를 넣어준다 → softmax를 할때 0과 가깝게 나오게 하기 위해

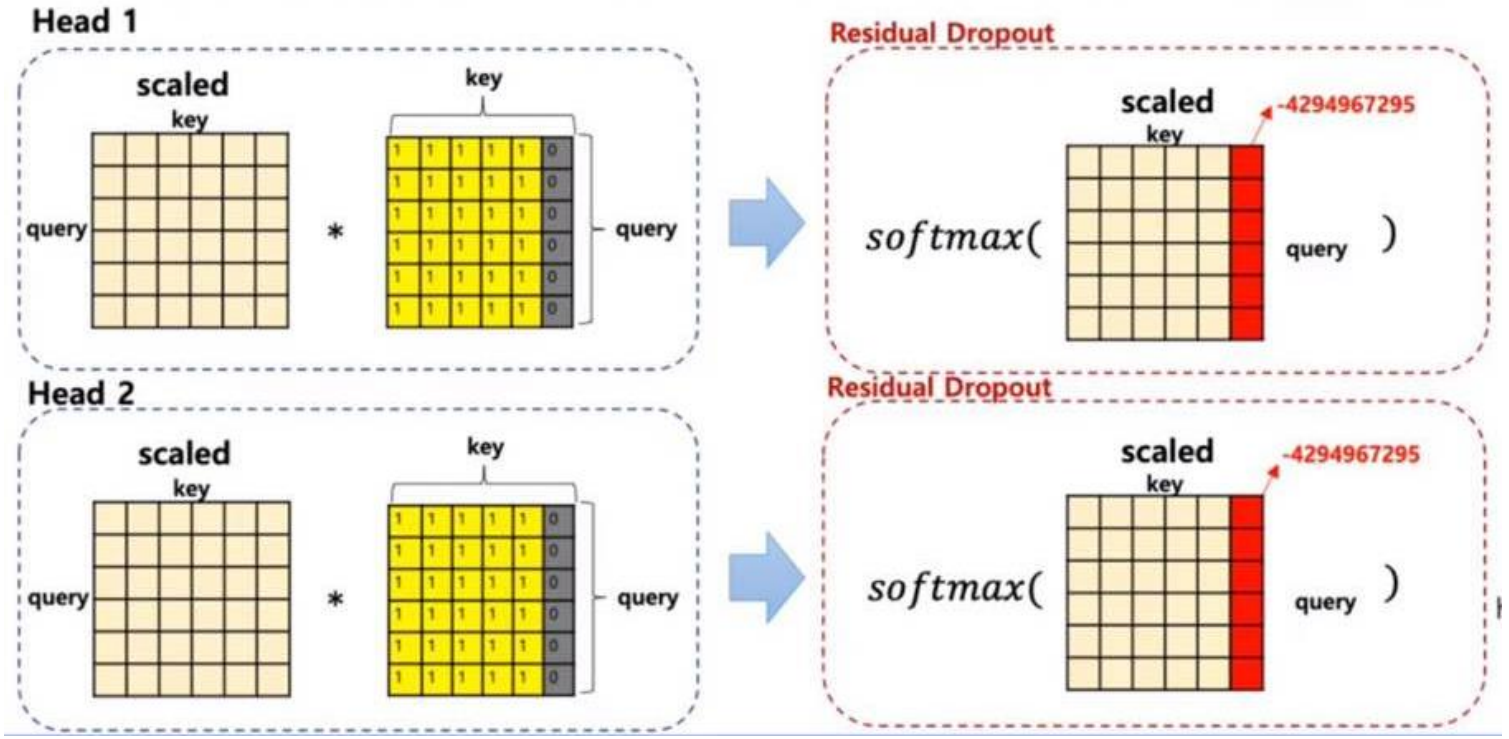
: 그 결과에 residual Dropout → attention에 대한 weight를 다양하게 하기 위해



Masking

03

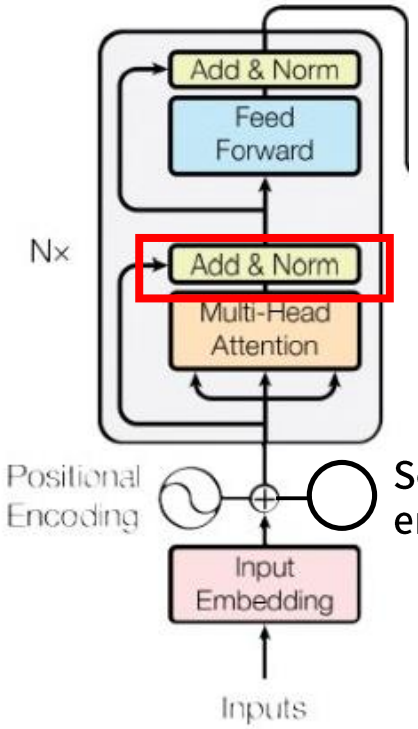
: 자기 자신에 대한 Attention을 하고 그 결과를 concat



: concat한 결과→ Multi-Head Attention의 결과

3. BERT 구조

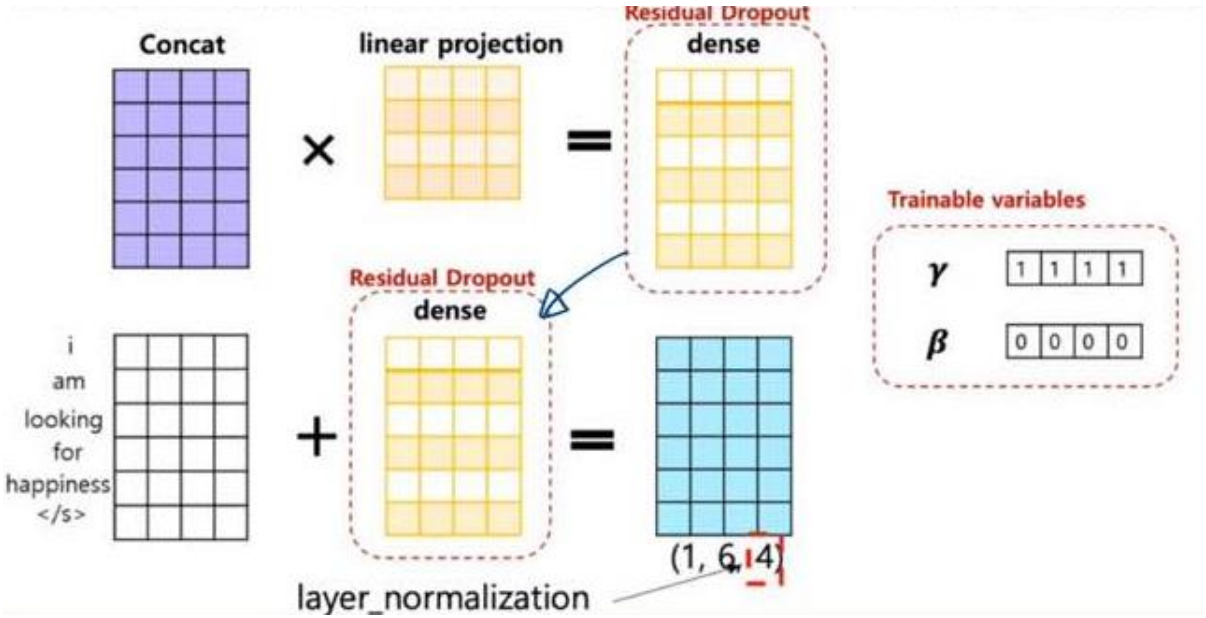
03



:(Multi-Attention head결과 matrix X linear projection)

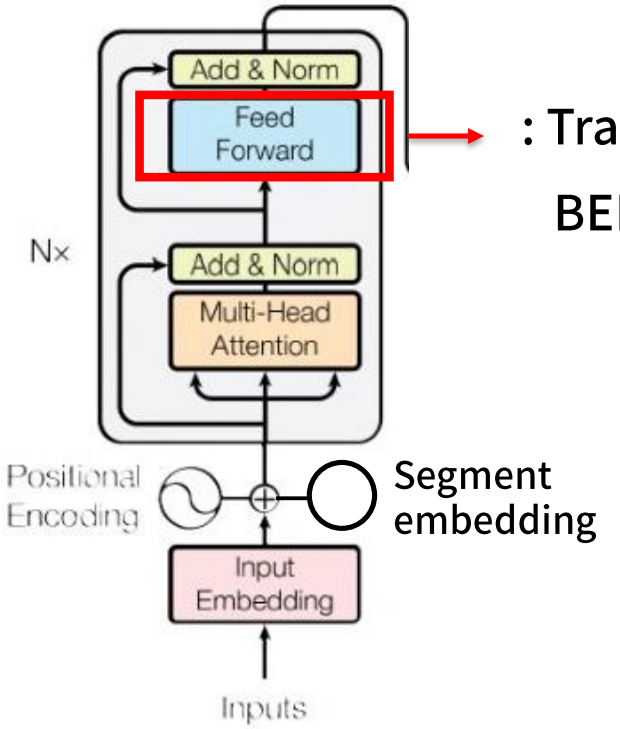
+ 원래의 input

: 위의 결과 matrix에 layer_normalization



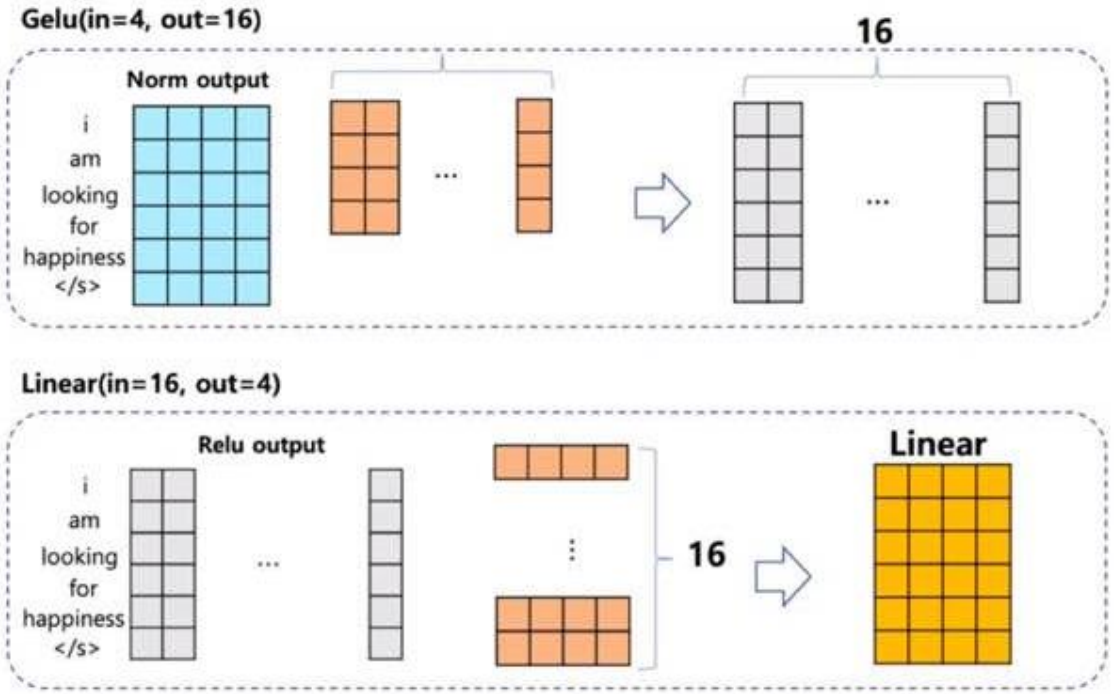
3. BERT 구조

03



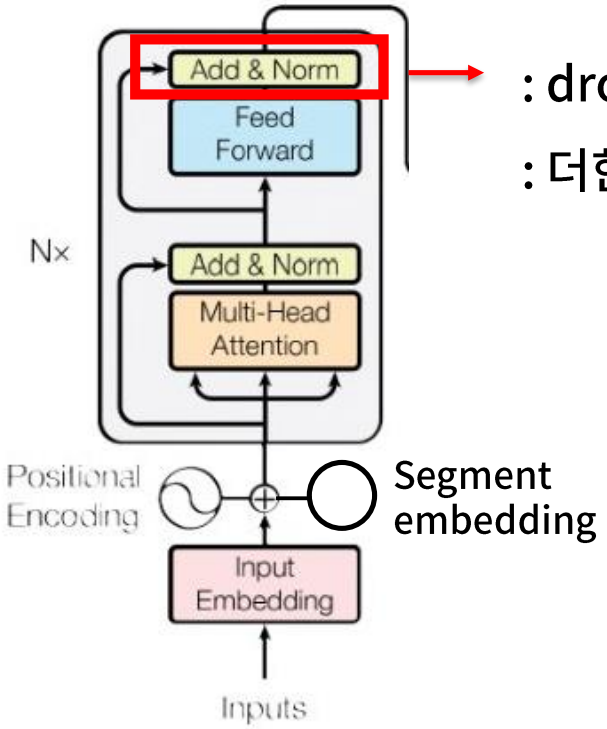
: Transformer에서는 Relu → linear

BERT에서는 보다 부드러운 Gelu → linear



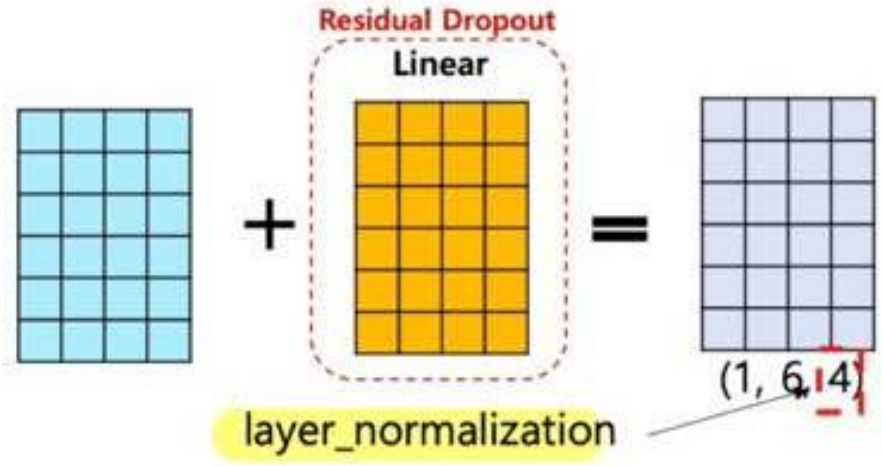
3. BERT 구조

03



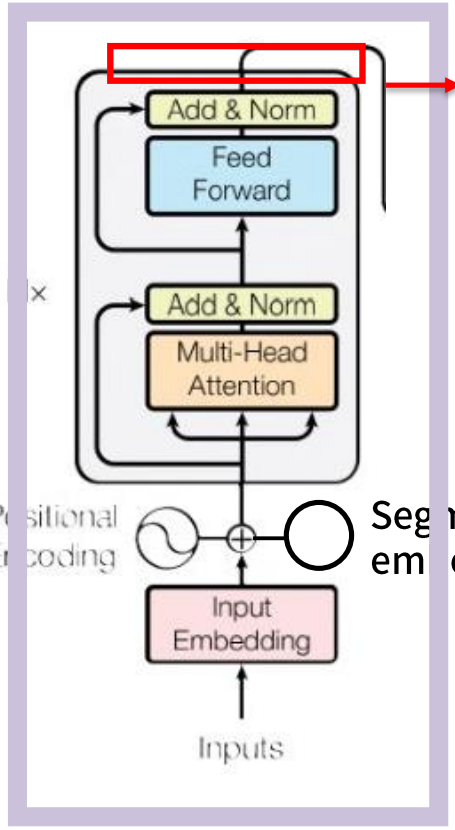
: dropout한 후 이전의 Add&Norm에서 나온 값과 더함

: 더한 결과에 layer_normalization



3. BERT 구조

03



BERT의 단어 embedding vector

Segment embedding

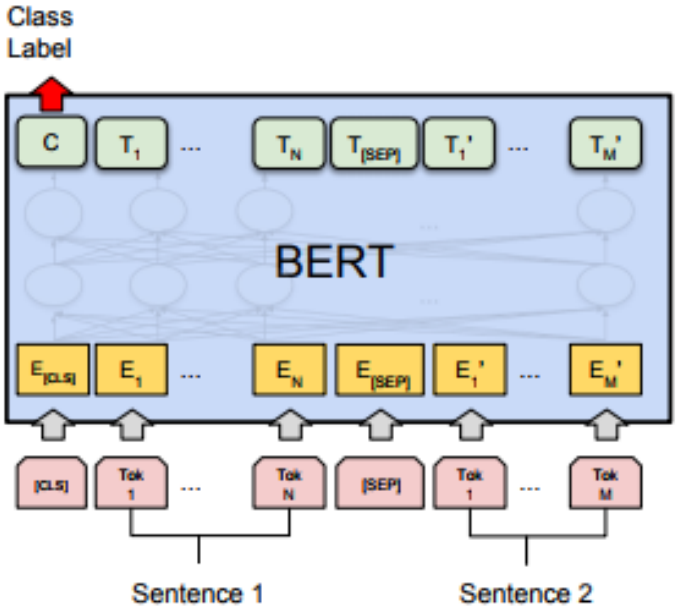
이 과정을
Block 수만큼 반복

4. Experiments

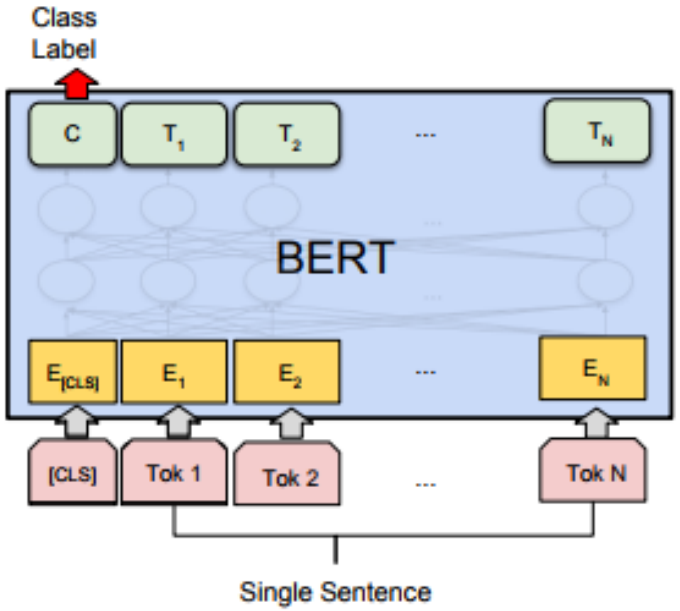
04

4.1 GLUE

: GLUE에 fine-tuning하는 방식 그림 ↓



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tasks: SST-2, CoLA

4. Experiments

04

4.1 GLUE

: 첫번째 input Token([CLS])에 해당하는 final hidden vector C를 aggregate representation으로 사용

: 이 C에 Classification loss를 계산

: BERT_LARGE의 경우 small data set에 안정적이지 않아, 여러 번 random restart하여 dev set에 가장 성능이 좋은 모델 선택

: 결과

System	MNLI-(m/mm)	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	**Average**
	392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.9	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	88.1	91.3	45.4	80.0	82.3	56.0	75.2
$\text{BERT}_{\text{BASE}}$	84.6/83.4	71.2	90.1	93.5	52.1	85.8	88.9	66.4	79.6
$\text{BERT}_{\text{LARGE}}$	**86.7/85.9**	**72.1**	**91.1**	**94.9**	**60.5**	**86.5**	**89.3**	**70.1**	**81.9**

4. Experiments

04

4.2 SQuAD v1.1

: paragraph와 question pair가 주어지면 정답을 포함하는 text span을 찾는 문제

: question과 paragraph을 각각의 single sentence로 보고 (즉, 각각 A, B embedding으로 구분한다.)

: 새롭게 학습되는 parameter→start vector S, end vector E

: ith input token의 마지막 hidden vector를 Ti라 할때, ith token이 start일 확률은 softmax로 구해짐

: answer span의 end도 위와 같이 구함

4. Experiments

04

4.2 SQuAD v1.1

: 결과

System	Dev		Test	
	EM	F1	EM	F1
Leaderboard (Oct 8th, 2018)				
Human	-	-	82.3	91.2
#1 Ensemble - nlnet	-	-	86.0	91.7
#2 Ensemble - QANet	-	-	84.5	90.5
#1 Single - nlnet	-	-	83.5	90.1
#2 Single - QANet	-	-	82.5	89.3
Published				
BiDAF+ELMo (Single)	-	85.8	-	-
R.M. Reader (Single)	78.9	86.3	79.5	86.6
R.M. Reader (Ensemble)	81.2	87.9	82.3	88.5
Ours				
$\text{BERT}_{\text{BASE}}$ (Single)	80.8	88.5	-	-
$\text{BERT}_{\text{LARGE}}$ (Single)	84.1	90.9	-	-
$\text{BERT}_{\text{LARGE}}$ (Ensemble)	85.8	91.8	-	-
$\text{BERT}_{\text{LARGE}}$ (Sgl.+TriviaQA)	**84.2**	**91.1**	**85.1**	**91.8**
$\text{BERT}_{\text{LARGE}}$ (Ens.+TriviaQA)	**86.2**	**92.2**	**87.4**	**93.2**

→ TriviaQA data를 추가하여 성능을 높임

5. 참고자료

05

김동화-Transformer & BERT

https://www.youtube.com/watch?v=xhY7m8QVKjo

Wikidocs- 딥러닝을 이용한 자연어처리 입문

https://wikidocs.net/31379

https://vanche.github.io/NLP_Pretrained_Model_BERT(2)/

5. 참고자료

05

- 우리의 질문

1) Pre-training~Fine-tuning할 때

Mismatch를 최소화하기 위해 sequence length를 임의적으로(10%) 짧게 수정했다고 하는데 왜?

→ 영상에선 길이가 짧은 문장들 사이에서 교집합을 찾는게 더 쉽기때문이라고 함

→ 이해 X

2) SQuAD v1.1로 fine-tuning을 할때

왜 start 부분 찾는 과정을 Question도 포함해서 하는지 paragraph만 하면 되는 거 아닌가?

THANK YOU

2. Preprocessing

02

2.4 학습되는 데이터

- Placeholder for data
  - sequence length = 128
  - 최대 masking 개수=20

Placeholde	Size	
input_ids	128	Vocab에서 해당 token이 어느 위치에 있는지
input_mask	128	Padding인지 아닌지 구분하는
segment_ids	128	Token A인지 B인지(0/1)
masked_lm_positions	20	Sequence에서 mask의 위치
masked_lm_ids	20	Vocab에서 해당 token이 어느 위치에 있는지
masked_lm_weights	20	Padding인지 아닌지 구분하는
next_sentence_labels	1	연속된 문장인지(True/Flase)