AI hub 일반상식 데이터 활용
QA Project

18기 분석 강하연
18기 분석 김다혜
18기 분석 박태남
18기 분석 서은유
18기 분석 한상범

INDEX

프로젝트 선정 이유

프로젝트 선정 이유

"왜 '한국어'만 못 알아볼까?"

```
tokenizer.decode(inputsShort["input_ids"])

'[CLS] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [SEP] [UNK] [UNK] [UNK] [UNK], [UNK] [UNK] [UNK] [UNK] [UNK]. [UNK] [U
NK] [UNK] [UNK] [UNK] [UNK] [UNK]. [UNK] [UNK]'[UNK] [UNK] [UNK] [UNK] [UNK] [UNK]'( Most Improved Player Of The Year )
[UNK] [UNK], [UNK] [UNK] [UNK] [UNK] ( Japan Men's Fashion Association ) [UNK] [UNK]'[UNK] [UNK] [UNK]'( Most Fashionab
le ) [UNK] [UNK]. [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK]. [UNK] [UNK] [UNK] [UNK] [UN
K] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK]. [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK],
[UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] [UNK] WTA [UNK] [UNK] [UNK] [UNK]. [UNK] [UNK]
[UNK] [UNK]! [SEP]'
```

<영어>
Most Improved Player of The Year
➡ 포착 성공

<한국어>
Unknown ➡ 포착 실패

프로젝트 선정 이유

“인공지능이 한국어를 더 어려워하는 이유는 무엇일까?”

1. 한국어는 ‘교착어 ***’ 이다.

● 교착어란? 어근과 접사에 의해 단어의 기능이 결정되는 언어를 의미한다
→ 따라서, 띄어쓰기 단위인 어절 단위로 토큰화할 경우 문장에서 발생 가능한 단어의 수가 폭발적으로 증가한다.

ex 단어 ‘**BOAZ**’

1. BOAZ랑 ~
2. BOAZ는 ~
3. BOAZ가 ~

2. 한국어는 어순이 중요하지 않다.

● 한국어는 어순이 바뀌어도 의미가 통하며 심지어 주어를 생략해도 문제가 없는 경우가 존재한다.
● 특정 단어 뒤에 어떤 단위가 나타나도 되기 때문에 확률에 기반한 언어 모델이 다음 단어를 예측하기 어렵다.

ex < 나는 학교에서 공부를 합니다. > VS < 나는 공부를 학교에서 합니다. >

3. 한국어는 띄어쓰기가 잘 지켜지지 않다.

● 한국어는 띄어쓰기를 전혀 하지 않더라도 의미 전달에 문제가 없다.
● 띄어쓰기는 한국인도 철저히 지키기 힘들만큼 까다롭고, 띄어쓰기 자체가 계속 변화되는 경향을 보인다.

데이터 소개 및 전처리

AI hub의 일반상식 데이터

- 한국어 위키백과내 주요 문서 15만개에 포함된 지식을 추출한 데이터이다.
- 위키백과 본문내용과 관련한 질문과 질문에 대응되는 위키백과 본문 내의 정답 쌍으로 구성되어 있다.
- 질문-답 쌍 데이터셋은 Context, question, answer를 각각 포함한다.
- Paragraphs는 가장 상위의 클래스이며 질문-답 세트와 질문-답 세트의 근거 단락인 contex를 하위 클래스로 가진다.
- qas에는 question과 answers가 포함된다.
- Answer_start는 Context에서 정답이 위치하는 글자 인덱스를 의미한다.

● **실제 데이터 구조**

```
- creator: MINDs Lab.
- version: 1
- data: [8538]
  - paragraphs: [1]
    - qas: [1]
      - question: 패밀리가 떴다는 어디에서 방송했어
      - answers: [1]
        - answer_start: 167
        - text: SBS
      - id: 8_C60_wiki_1818-1
    - context: 2010년 2월 14일 패밀리가 떴다가 종영되고, 패밀리가 떴다 2가 2월 21일 방영되었다. 《패밀리가 떴다》는 2008년 3월부터 제작을 시작하였으며, 20
  - title: 패밀리가 떴다
```

전처리

1) 띄어쓰기 단위 정보 관리

ex) '1839년 파우스트를 읽었다.'

- 띄어쓰기를 기준으로 단어를 분리한 후 영역을 표시한다.

→ 즉, 첫번째 어절에 해당하는 단어는 0, 두번째 어절에 해당하는 단어는 1이 기입된다.

```
'1' : ['1'] : [0]
'8' : ['18'] : [0, 0]
'3' : ['183'] : [0, 0, 0]
'9' : ['1839'] : [0, 0, 0, 0]
'년' : ['1839년'] : [0, 0, 0, 0, 0]
' ' : ['1839년'] : [0, 0, 0, 0, 0, 0]
'파' : ['1839년', '파'] : [0, 0, 0, 0, 0, 0, 1]
'우' : ['1839년', '파우'] : [0, 0, 0, 0, 0, 0, 1, 1]
'스' : ['1839년', '파우스'] : [0, 0, 0, 0, 0, 0, 1, 1, 1]
'트' : ['1839년', '파우스트'] : [0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
'를' : ['1839년', '파우스트를'] : [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
' ' : ['1839년', '파우스트를'] : [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
'읽' : ['1839년', '파우스트를', '읽'] : [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2]
'었' : ['1839년', '파우스트를', '읽었'] : [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2]
'다' : ['1839년', '파우스트를', '읽었다'] : [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2]
'.' : ['1839년', '파우스트를', '읽었다.'] : [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2]
```

전처리

2) Tokenize by Vocab

ex) '읽었다', '읽고', '읽으니', '읽어'

- 모든 단어를 단어 사전에 저장한다면 상당히 많은 단어가 저장되게 된다.

→ 읽 + 었다, 읽 + 고 로 나누어 처리하기 위해 단어를 나누어 처리하는 Subword Segmentation을 활용한다.

```
# vocab loading
vocab = spm.SentencePieceProcessor()
vocab.load(f"{model_dir}/ko_32000.model")

# word를 subword로 변경하면서 index 저장
word_to_token = []
context_tokens = []
for (i, word) in enumerate(word_tokens):
    word_to_token.append(len(context_tokens))
    tokens = vocab.encode_as_pieces(word)  # SentencePiece를 사용해 Subword로 쪼개기
    for token in tokens:
        context_tokens.append(token)

context_tokens, word_to_token
```

```
(['_1839', '년', '_', '파우스트', '를', '_읽', '었다', '.'], [0, 2, 5])
```

- 앞부분이 공백인 단어에는 앞에 '__ ' 을 포함
- [0, 2, 5] : 앞부분이 공백인 단어들의 위치

전처리

“ 데이터셋은 질문과 지문을 주고, 지문 영역에서 정답을 찾도록 구성되어있기 때문에 정답영역을 정확히 찾는 것이 중요하다. ”

3) 정답 영역을 정확히 찾아내기

- 띄어쓰기 단위로 쪼개진 context를 subword로 토큰화한다.

'우승'이 정답인 경우

띄어쓰기 단위로 쪼개진 단어는 '우승하였으며' 이므로

subword로 토큰화하여 정확하게 '우승' 만 추출해야 한다.

```
0 ['_재팬']
1 ['_오픈', '에서']
3 ['_4', '회']
5 ['_우승', '하였으며', ',']
8 ['_통산']
9 ['_단식']
10 ['_200', '승']
12 ['_이상을']
13 ['_거두었다', '.']
15 ['_1994', '년']
17 ['_생애']
18 ['_최초로']
19 ['_세계']
20 ['_랭킹']
21 ['_10', '위권', '에']
24 ['_진입', '하였다', '.']
27 ['_1992', '년에는']
29 ['_W', 'TA', '로부터']
32 ["_'", '올', '해']
35 ['_가장']
```

전처리

3) 정답 영역을 정확히 찾아내기

- context에 포함된 answer의 글자 단위 시작 인덱스 answer_start 와 종료 인덱스 answer_end 위치를 찾고, 어절(word) 단위로 변환한다.

```
[context]  재팬 오픈에서 4회 우승하였으며, 통산 단식 200승 이상을 거두었다. 1994년 생애 최초로 세계 랭킹 10위권에 진입하였다. 1992년에는 WTA로부터 '올해
[question]  다테 기미코가 최초로 은퇴 선언을 한게 언제지
[answer]  1996년 9월 24일
[answer_start] index:  260 character:  1
[answer_end]index:  271 character:  일
```

```
word_start = char_to_word[answer_start]
word_end = char_to_word[answer_end]
word_start, word_end, answer_text, word_tokens[word_start:word_end + 1]
```

```
(49, 51, '1996년 9월 24일', ['1996년', '9월', '24일'])
```

→ 260 ~ 271 인덱스를 어절 단위로 변환한 결과 49 ~ 51번째 어절에 정답이 포함되어 있음을 알 수 있다.

전처리

4) 데이터셋 분리

- 60,000 건의 데이터를 train 데이터로, 나머지 약 8000건의 데이터를 test 데이터로 분리한다.

```
with open('/content/drive/MyDrive/AI_hub/ko_wiki_v1_squad.json', 'r') as file:
    jsonRaw = json.load(file)
    print(type(jsonRaw))
    data = jsonRaw['data']
    print(len(data))

    new_train = {
        'creator': jsonRaw['creator'],
        'version': jsonRaw['version'],
        'data': data[:60000]
    }

    new_test = {
        'creator' : jsonRaw['creator'],
        'version' : jsonRaw['version'],
        'data': data[60000:]
    }
```

전처리

5) 데이터셋 전처리 후 저장

- context, questio을 공백 단위로 자르고 subword로 토큰화한다.
- context에서 answer의 위치를 어절단위로 찾아준다.
- 아래와 같이 train, test dataset 모두 전처리 후 json 파일로 저장한다.


```
{"qa_id": "6566495-0-0", "title": "파우스트_서곡", "question": ["_바그너", "는", "_괴테", "의", "_", "파우
스트", "를", "_읽고", "_무엇을", "_쓰고", "자", "_", "했", "는", "가", "?"], "context": ["_1839", "년", "_
바그너", "는", "_괴테", "의", "_", "파우스트", "을", "_처음", "_읽고", "_그", "_내용에", "_마음이", "_끌려", "_0
를", "_소재로", "_해서", "_하나의", "_교향곡", "을", "_쓰", "려는", "_뜻을", "_갖는다", ".", "_이", "_시기", "_
바그너", "는", "_1838", "년에", "_빛", "_독", "촉", "으로", "_산", "전", "수", "전을", "_다", "_", "걲",
"은", "_상황이", "라", "_좌절", "과", "_실망", "에", "_가득", "했으며", "_메", "피스", "토", "펠", "레스",
"를", "_만나는", "_", "파우스트", "의", "_심", "경에", "_공감", "했다고", "_한다", ".", "_또한", "_파리에서", "_
아브", "네", "크의", "_지휘", "로", "_파리", "_음악원", "_관현악단", "이", "_연주하는", "_베토벤", "의", "_교향
곡", "_9", "번을", "_듣고", "_깊은", "_감", "명을", "_받았는데", ",", "_이것이", "_이듬해", "_1", "월에", "_",
"파우스트", "의", "_서", "곡으로", "_쓰여진", "_이", "_작품에", "_조금", "이라도", "_영향을", "_끼", "쳤", "으리
라", "는", "_것은", "_의심", "할", "_여지가", "_없다", ".", "_여기", "의", "_라", "단", "조", "_조성", "의",
"_경우에도", "_그의", "_전기", "에", "_적혀", "_있는", "_것처럼", "_단순한", "_정신적", "_피로", "나", "_실",
"의", "가", "_반영", "된", "_것이", "_아니라", "_베토벤", "의", "_합창", "교", "향", "곡", "_조성", "의", "_영
향을", "_받은", "_것을", "_볼", "_수", "_있다", ".", "_그렇게", "_교향곡", "_작곡", "을", "_1839", "년부터",
"_40", "년에", "_걸쳐", "_파리에서", "_착수", "했으나", "_1", "악장", "을", "_쓴", "_뒤에", "_중단", "했다",
".", "_또한", "_작품의", "_완성", "과", "_동시에", "_그는", "_이", "_서", "곡", "(1", "악장", ")", "을", "_파
리", "_음악원", "의", "_연주회", "에서", "_연주", "할", "_파트", "보", "까지", "_준비", "하였으나", ",", "_실제로
는", "_이루어지지", "는", "_않았다", ".", "_결국", "_초연", "은", "_4", "년", "_반", "이", "_지난", "_후에",
"_드레스덴", "에서", "_연주", "되었고", "_재", "연", "도", "_이루어졌", "지만", ",", "_이후에", "_그대로", "_방
치", "되고", "_말았다", ".", "_그", "_사이에", "_그는", "_리", "엔", "치", "와", "_방", "황", "하는", "_네덜란
드", "인", "을", "_완성", "하고", "_탄", "호", "이", "저", "에도", "_착수", "하는", "_등", "_분", "주", "한",
"_시간을", "_보", "냈는데", ",", "_그런", "_바쁜", "_생활", "이", "_이", "_곡을", "_잊", "게", "_한", "_것이",
"_아닌", "가", "_하는", "_의견도", "_있다", "."], "answer": "교향곡", "token_start": 19, "token_end": 19}
{"qa_id": "6566495-0-1", "title": "파우스트_서곡", "question": ["_바그너", "는", "_교향곡", "_작곡", "을",
```

전처리

6) 전처리 완료한 데이터를 메모리에 로드, 데이터 확인

● Question과 Context가 포함된 입력데이터

```
array([    5, 13787, 24839, 10305,  1053,  2323, 18778,    58,   139,
        9675,    47,     4, 18404,  2804,    22,    46,   168,   908,
         450,     8,  4256, 11030,  1570,   470,  6298,  4280,     7,
        1218,    15,  6315,  1053,   238,  7410,    66, 14432,    13,
        3444,    54,     7,  1238,   308,   777,  5884,   420,    62,
        2194,    60,   141,   189, 18762, 16941,   611,   121,  3112,
         579,  8970,    11, 27598,  1772,  2613,  1181,   599, 27915,
       15461,  2007, 27103,    12,    16,  4652, 16634,     8,   146,
        1020,  5060,  2633,    14, 17052, 21492,   476,  1173,    56,
         162,   540, 13273,  3032, 20074,    12,    10,  6602,    62,
          19,   148,  1168,  2681,   827,  1761,    33,   611,  3112,
         579,  8970,   540, 13273,  3032, 12684,    12,    21,  2905,
          53,     7,  6315,    98,   407,   569,  2107, 22606,  1153,
          15,    81,    23,   544,    28,  1053,  2323,    20, 11152,
           7,    83,    89,    15,    63,    31,   386,    15,    46,
         443,  2898,   200,  1785,  2640, 18778,   367,  3937,    13,
       10254,  2722,     7,   386,    15,    64,    23,   182,    28,
        1120, 17040,   660, 12746,   651,  2804,    22,  2640,    79,
         216,  1415, 20613,     8,   429,    15,    81,    23,   645,
        1127,  6113,   705,    58,  3579,  8260,  2804,  2069,   908,
         246,  2640,    79,   216,   777,  5884,  3937,   442,   865,
        1415,  1625,     7,    58, 22608,  3653,  2132,   838,  4561,
         108,  1064,     4,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0,     0,     0,     0,     0,     0,     0,     0,     0,
           0.     0.     0.     0.     0.     0.     0.     0.     0.
```

● Question을 0으로, Context를 1로 구분해 준 Segment 데이터

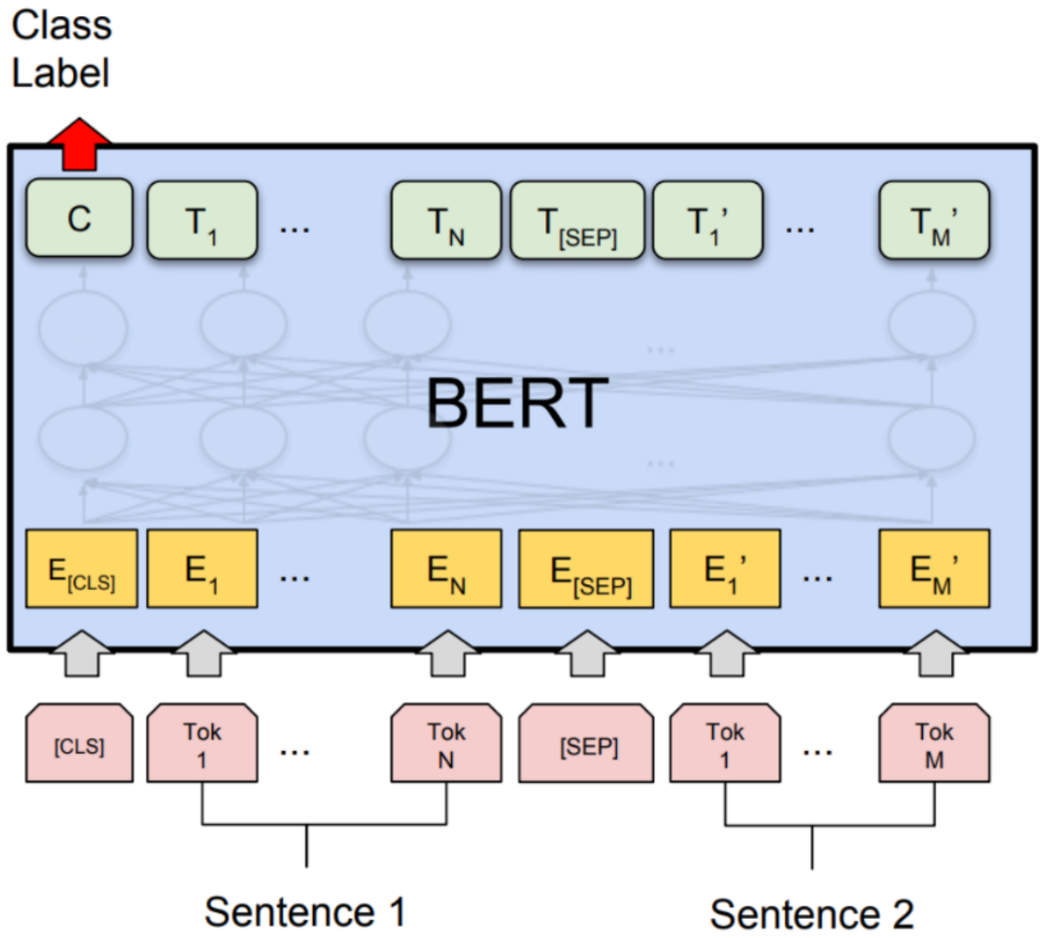
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

● Answer 위치의 시작점과 끝점 라벨

```
(116, 121)
```

모델링

Modeling
- BERT Tokenizer 개요

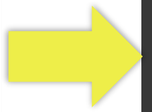


- **BERT 모델 구조**
트랜스포머의 self - attention으로
이루어진 encoder 구조만을 활용한다.

- **주목할 점**
입력이 주어지면 실제로 모델에 입력되는 것은 token, segment, position 임베딩이 더해진 형태이다.

BERT
- Input presentation

```python
# 생성한 데이터셋 파일을 메모리에 로딩하는 함수
def load_data(args, filename):
    inputs, segments, labels_start, labels_end = [], [], [], []

    n_discard = 0
    with open(filename, "r") as f:
        for i, line in enumerate(tqdm(f, desc=f"Loading ...")):
            data = json.loads(line)
            token_start = data.get("token_start")
            token_end = data.get("token_end")
            question = data["question"][:args.max_query_length]
            context = data["context"]
            answer_tokens = " ".join(context[token_start:token_end + 1])
            context_len = args.max_seq_length - len(question) - 3

            if token_end >= context_len:
                # 최대 길이내에 token이 들어가지 않은 경우 처리하지 않음
                n_discard += 1
                continue
            context = context[:context_len]
            assert len(question) + len(context) <= args.max_seq_length - 3

            tokens = ['[CLS]'] + question + ['[SEP]'] + context + ['[SEP]']
```



```
[CLS] 다테 기미코가 최초로 은퇴 선언을 한게 언제지[SEP] 재팬 오픈에서
 4회 우승하였으며, 통산 단식 200승 이상을 거두었다. 1994년 생애
 최초로 세계 랭킹 10위권에 진입하였다. 1992년에는 WTA로부터 '
올해 가장 많은 향상을 보여준 선수상'(Most Improv
ed Player Of The Year)을 수여받았으며, 일본 남자 패션 협회
(Japan Men's Fashion Association)는 그녀를 '
가장 패셔너블한 선수'(Most Fashionable
)로 칭했다. 생애 두 번째 올림픽 참가 직후인 1996년 9월
 24일 최초로 은퇴를 선언하였다. 이후 12년만인 2008년 4
월에 예상치 못한 복귀 선언을 하고 투어에 되돌아왔다. 2008년 6
월 15일 도쿄 아리아케 인터내셔널 여자 오픈에서 복귀 후 첫 우승을 기록했으며
, 2009년 9월 27일에는 한국에서 열린 한솔 코리아 오픈 대회에서 우승
하면서 복귀 후 첫 WTA 투어급 대회 우승을 기록했다. 한숨 좀
 작작 쉬어![SEP]
```

- NSP문제 (= Next Sentence Prediction)를 해결하기 위해 [CLS]+질문+[SEP]+지문+[SEP] 형태로 입력 모양을 정의하였다.

BERT
- Input Embedding

```
class SharedEmbedding(tf.keras.layers.Layer):
    """
    Weighed Shared Embedding Class
    """
    def __init__(self, config, name="weight_shared_embedding"):
        """
        생성자
        :param config: Config 객체
        :param name: layer name
        """
        super().__init__(name=name)

        self.n_vocab = config.n_vocab
        self.d_model = config.d_model
```

- **Token embedding**

- Token Embedding BERT는 텍스트의 tokenizer 로 Word Piece 모델이라는 subword tokenizer를 사용한다.

- tokenizer를 통해서 입력 문장을 토큰단위로 쪼개고, 해당 토큰을 vocab에 매칭하여 id(숫자)로 입력한다.

- vocab의 사이즈는 보통 5만정도를 사용하고 있고, 이것을 직접 입력하여 BERT 모델을 구축하면 학습해야하는 파라미터의 수가 엄청나게 많아지기 때문에 차원을 낮추기 위해서 embedding layer를 사용한다. 이러한 임베딩 작업을 통해 토큰을 벡터로서 표현하기 때문에 상대적으로 적은 차원의 수로 표현할 수 있게 된다.

BERT
- Input Embedding

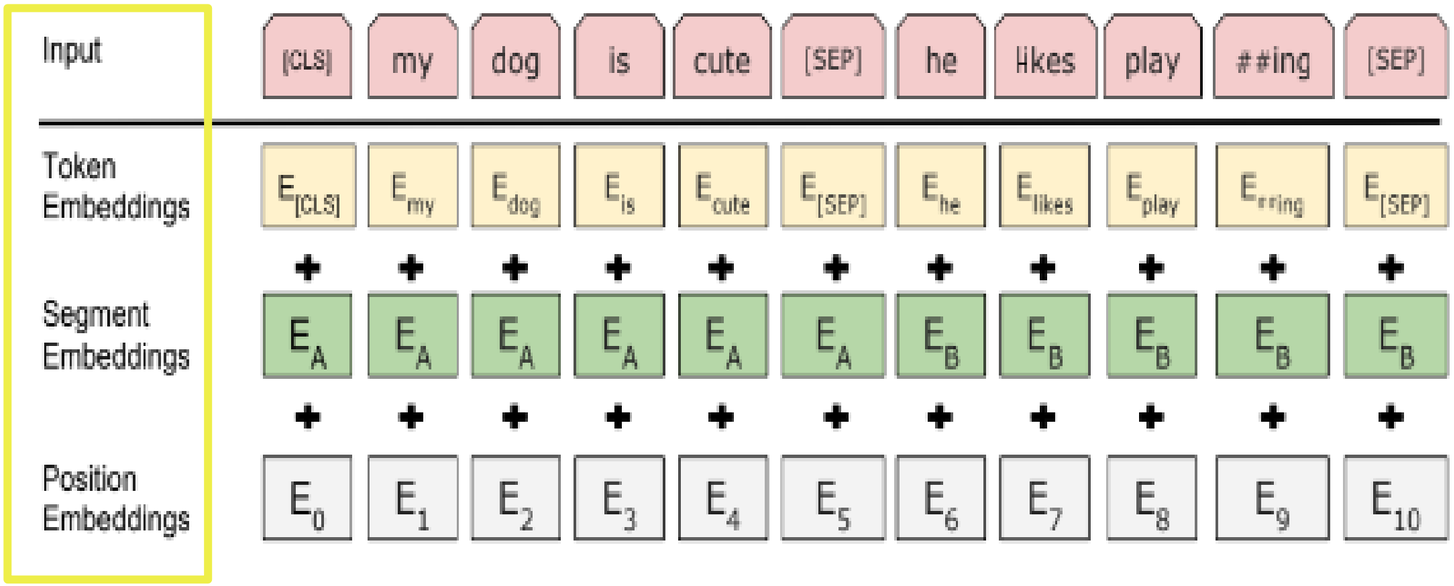
```python
class PositionalEmbedding(tf.keras.layers.Layer):
    """
    Positional Embedding Class
    """
    def __init__(self, config, name="position_embedding"):
        """
        생성자
        :param config: Config 객체
        :param name: layer name
        """
        super().__init__(name=name)

        self.embedding = tf.keras.layers.Embedding(config.n_seq, config.d_model, #
                                                   embeddings_initializer=kernel_initializer())
```

-

- **Positional Embedding**

- 토큰의 절대적 혹은 상대적인 위치에 대한 정보를 주입하는 것을 의미한다.
- 기존의 Transformer에서 사용되던 positional embedding을 의미한다.

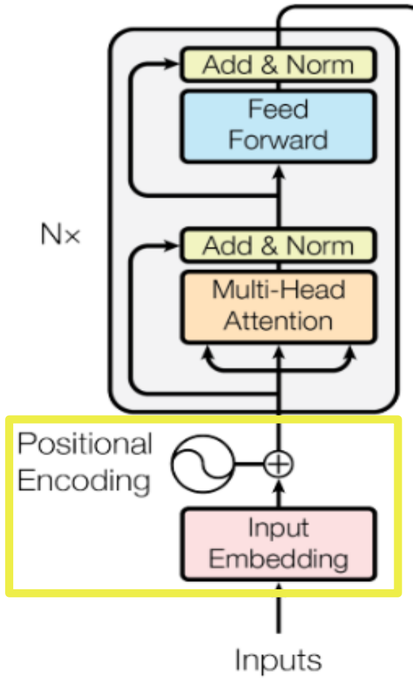
BERT
- Input Embedding



BERT는 Transformer와 달리 Positional Encoding을 사용하지 않고 대신 Position Embeddings를 사용한다. 여기에 문장을 구분해주는 Segment Embeddings를 추가해 각각의 임베딩, 즉 3개의 임베딩을 합산한 결과를 취한다.

● 계산식

```
embed = self.embedding(tokens) + self.position(tokens) + self.segment(segments)
```



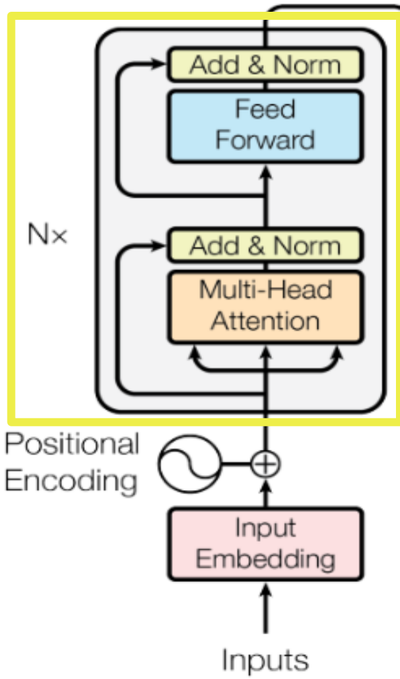
BERT
- Encoding Layer

```python
class EncoderLayer(tf.keras.layers.Layer):
    """
    Encoder Layer Class
    """
    def __init__(self, config, name="encoder_layer"):
        """
        생성자
        :param config: Config 객체
        :param name: layer name
        """
        super().__init__(name=name)

        self.self_attention = MultiHeadAttention(config)
        self.norm1 = tf.keras.layers.LayerNormalization(epsilon=config.layernorm_epsilon)

        self.ffn = PositionWiseFeedForward(config)
        self.norm2 = tf.keras.layers.LayerNormalization(epsilon=config.layernorm_epsilon)

        self.dropout = tf.keras.layers.Dropout(config.dropout)

    def call(self, enc_embed, self_mask):
        """
        layer 실행
        :param enc_embed: enc_embed 또는 이전 EncoderLayer의 출력
        :param self_mask: enc_tokens의 pad mask
        :return enc_out: EncoderLayer 실행 결과
        """
        self_attn_val = self.self_attention(enc_embed, enc_embed, enc_embed, self_mask)
        norm1_val = self.norm1(enc_embed + self.dropout(self_attn_val))

        ffn_val = self.ffn(norm1_val)
        enc_out = self.norm2(norm1_val + self.dropout(ffn_val))

        return enc_out
```

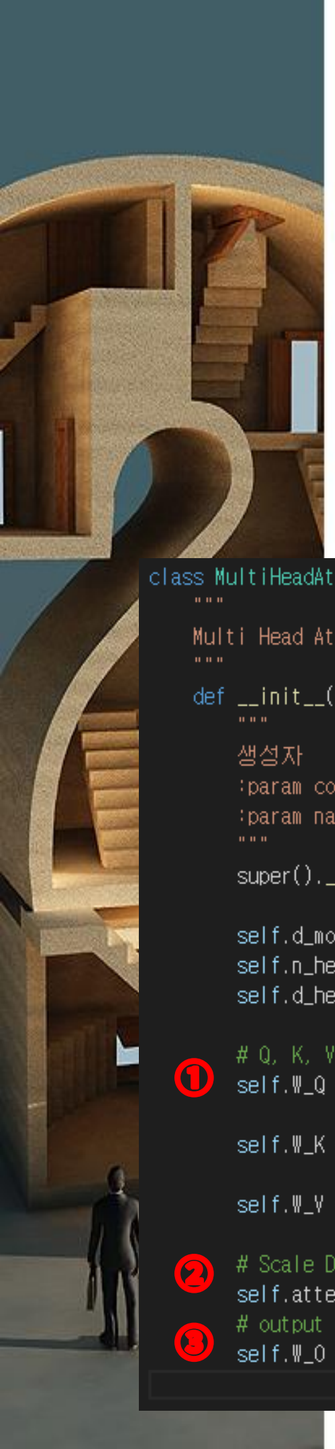
- **BERT의 Encoding Layer 정리**

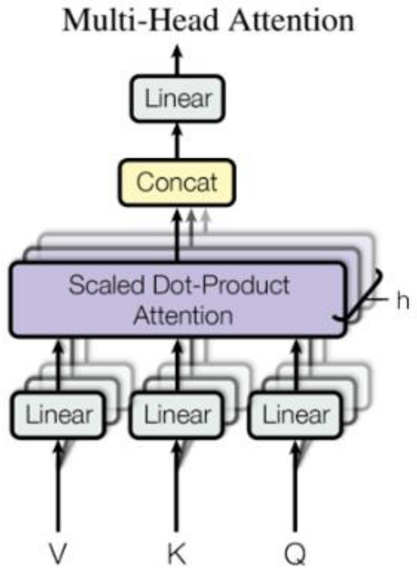
BERT는 N개의 인코더 블럭을 지니고 있다.

→ 이는 입력 시퀀스 전체의 의미를 N번 만큼 반복적으로 구축하는 것을 의미한다. 인코더 블럭의 수가 많을수록 단어 사이에 보다 복잡한 관계를 더 잘 포착할 수 있다.



BERT
- Encoding Layer





Multi Head Attention이란?

인코더 블럭의 가장 핵심적인 부분이며 말 그대로 헤드가 여러 개인 어텐션을 뜻한다.
서로 다른 가중치 행렬을 이용해 어텐션을 h번 계산한 다음 이를 서로 연결한 결과를 갖는다.

```python
class MultiHeadAttention(tf.keras.layers.Layer):
    """
    Multi Head Attention Class
    """
    def __init__(self, config, name="multi_head_attention"):
        """
        생성자
        :param config: Config 객체
        :param name: layer name
        """
        super().__init__(name=name)

        self.d_model = config.d_model
        self.n_head = config.n_head
        self.d_head = config.d_head

        # Q, K, V input dense layer
①       self.W_Q = tf.keras.layers.Dense(config.n_head * config.d_head, kernel_initializer=kernel_initializer(), #
                                         bias_initializer=bias_initializer())
        self.W_K = tf.keras.layers.Dense(config.n_head * config.d_head, kernel_initializer=kernel_initializer(), #
                                         bias_initializer=bias_initializer())
        self.W_V = tf.keras.layers.Dense(config.n_head * config.d_head, kernel_initializer=kernel_initializer(), #
                                         bias_initializer=bias_initializer())
②       # Scale Dot Product Attention class
        self.attention = ScaleDotProductAttention(name="self_attention")
③       # output dense layer
        self.W_O = tf.keras.layers.Dense(config.d_model, kernel_initializer=kernel_initializer(), #
                                         bias_initializer=bias_initializer())
```

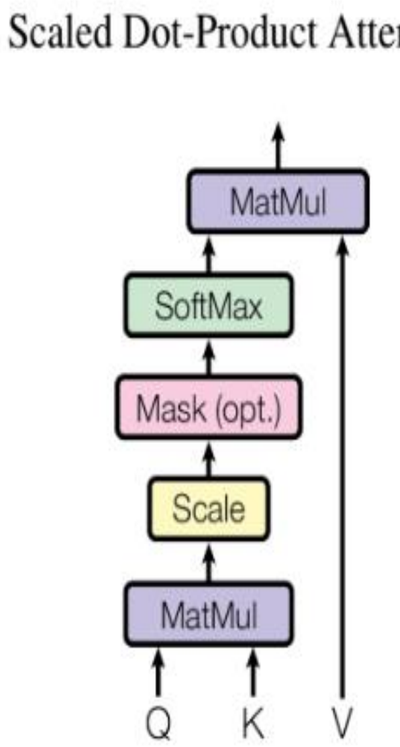
- **Multi Head Attention 과정**

① Query와 Key, Value 입력

② Q, K, V에 **ScaleDotProduct** 를 적용하여 다른 입력 차원으로 변환

③ **ScaleDotProduct**의 출력으로 나오는 여러개의 head를 concat > projection 수행

BERT
- Encoding Layer



Scale Dot Product Attention 이란?

- Scaled Dot-Product Attention은 입력값으로 **Q, K, V** 세 개를 받는다.
이는 입력값에 대한 플레이스 홀더로 맨 처음에는 임베딩의 fully-connected 결과를,
두 번째 부터는 RNN과 유사하게 이전 인코더 블록의 결과를 다음 인코더 블록의
입력으로 사용한다.
- Multi-Head Attention은 Scaled Dot-Product Attention을
H번 계산한 결과의 연결이다.

-> 원래 Transformer에서 Q는 주로 디코더의 히든 스테이트, K는 주로 인코더의 히든 스테이트, V는 K에 어텐션을 부여 받은 Normalized Weights가 되며,
초기값은 V와 K가 동일하다.
그러나 BERT는 디코더를 사용하지 않으며 Q, K, V의 초기값이 모두 동일하다.
물론 저마다 각각 다른 초기화로 인해 실제로는 서로 다른 값에서 출발하지만
입력값의 구성은 동일하다.
BERT는 이처럼 동일한 토큰이 문장내의 다른 토큰에 대한
Self-Attention 효과를 갖는다.

BERT
- Encoding Layer

```python
class ScaleDotProductAttention(tf.keras.layers.Layer):
    """
    Scale Dot Product Attention Class
    """
    def __init__(self, name="scale_dot_product_attention"):
        """
        생성자
        :param name: layer name
        """
        super().__init__(name=name)

    def call(self, Q, K, V, attn_mask):
        """
        layer 실행
        :param Q: Q value
        :param K: K value
        :param V: V value
        :param attn_mask: 실행 모드
        :return attn_out: attention 실행 결과
        """
①       attn_score = tf.matmul(Q, K, transpose_b=True)
        scale = tf.math.sqrt(tf.cast(tf.shape(K)[-1], tf.float32))
②       attn_scale = tf.math.divide(attn_score, scale)
        attn_scale -= 1.e9 * attn_mask
③       attn_prob = tf.nn.softmax(attn_scale, axis=-1)
④       attn_out = tf.matmul(attn_prob, V)
        return attn_out
```

- **Scale Dot Product Attention 과정**

① Query와 Key를 곱하여 attention scoer matrix 생성

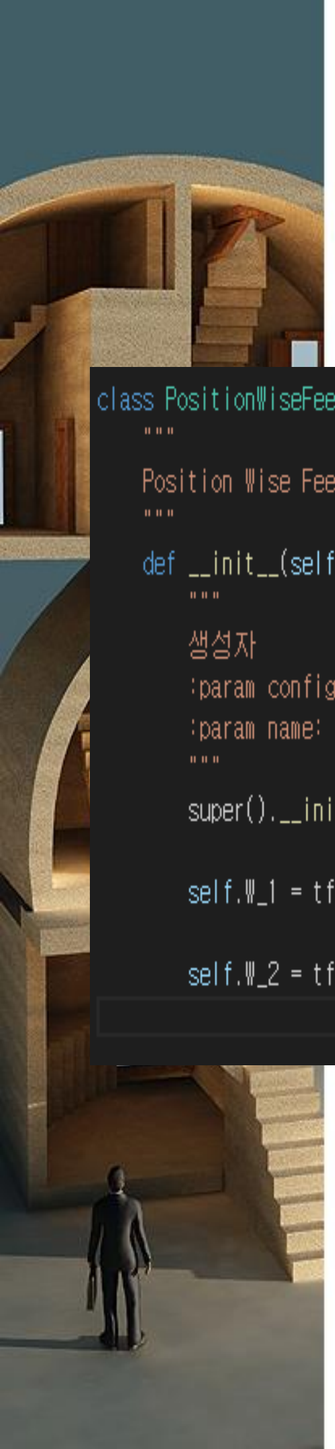
② depth의 루트값으로 나누어 scaling

③ attention weight 계산

④ Value와 Attention weight를 가중합하여 Attention value 계산

-> 각각의 토큰 벡터 768차원을 헤드 수 만큼인 12등분 (BERT based 기준) 하여 64개씩 12조각으로 차례대로 분리한다. 여기에 Scaled Dot-Product Attention을 적용하고 다시 768차원으로 합친다. 그렇게 되면 768차원 벡터는 각각 부위별로 12번 Attention 받은 결과가 된다.

BERT
- Encoding Layer



```python
class PositionWiseFeedForward(tf.keras.layers.Layer):
    """
    Position Wise Feed Forward Class
    """
    def __init__(self, config, name="feed_forward"):
        """
        생성자
        :param config: Config 객체
        :param name: layer name
        """
        super().__init__(name=name)

        self.W_1 = tf.keras.layers.Dense(config.d_ff, activation=gelu, #
                                         kernel_initializer=kernel_initializer(), bias_initializer=bias_initializer())
        self.W_2 = tf.keras.layers.Dense(config.d_model, #
                                         kernel_initializer=kernel_initializer(), bias_initializer=bias_initializer())
```

- **Position-wise Feed Forward Network**

앞선 어텐션의 결과를 Position-wise Feed-forward Network로 통과한다. Feed-forward Network Layer에서는 두번의 선형 변환을 하게 된다.

- **세부설명**

- **Attention function : GELU**

인코더 블락 안의 2-layer MLP 구조의 활성화 함수로 NLP, Vision 분야의 태스크에 대해 일관적으로 성능이 더 좋다는 실험 결과를 가진 GELU를 활성화 함수로 사용한다.
GELU는 ReLU보다 0주위에서 부드럽게 변화해 학습 성능을 높이며 원조 트랜스포머와 BERT 가 사용하는 트랜스포머 블록에서 가장 큰 차이점을 보이는 대목이다.

BERT
- BERT Structure 정리

```python
class BERT(tf.keras.layers.Layer):
    """
    BERT Class
    """
    def __init__(self, config, name="bert"):
        """
        생성자
        :param config: Config 객체
        :param name: layer name
        """
        super().__init__(name=name)

        self.i_pad = config.i_pad
        self.embedding = SharedEmbedding(config)
        self.position = PositionalEmbedding(config)
        self.segment = tf.keras.layers.Embedding(2, config.d_model, embeddings_initializer=kernel_initializer())
        self.norm = tf.keras.layers.LayerNormalization(epsilon=config.layernorm_epsilon)

        self.encoder_layers = [EncoderLayer(config, name=f"encoder_layer_{i}") for i in range(config.n_layer)]

        self.dropout = tf.keras.layers.Dropout(config.dropout)
```

- **BERT Structure 정리**

- **Token, Segment, Position Embedding의 입력 형태 조정한다.**

- **Segment Embedding 상세 설명**

- 각 단어가 어느 문장에 포함되는지 역할 규정한다.
(ex. Kowiki에서 해당 단어가 Question에 속하는지 Context에 속하는지 구분함)

- 임베딩 결과를 > Encoder 레이어로 입력한다.

BERT
- Fine Tuning

Fine-Tuning은 Pre-train된 BERT 모델을 이용해 수행하고자 하는 task를 추가 학습하는 것을 의미한다.
전이학습은 BERT의 언어 모델의 출력에 추가적인 모델을 쌓아 만든다.

```
class BERT4KorQuAD(tf.keras.Model):
    def __init__(self, config):
        super().__init__(name='BERT4KorQuAD')

        self.bert = BERT(config)
        self.dense = tf.keras.layers.Dense(2)
```

```
config = Config({"d_model": 256, "n_head": 4, "d_head": 64, "dropout": 0.1, #
                 "d_ff": 1024, "layernorm_epsilon": 0.001, "n_layer": 3, #
                 "n_seq": 384, "n_vocab": 0, "i_pad": 0})
config.n_vocab = len(vocab)
config.i_pad = vocab.pad_id()
config
```

```
{'d_ff': 1024,
 'd_head': 64,
 'd_model': 256,
 'dropout': 0.1,
 'i_pad': 0,
 'layernorm_epsilon': 0.001,
 'n_head': 4,
 'n_layer': 3,
 'n_seq': 384,
 'n_vocab': 32007}
```

조정 내용

- **BERT의 각 토큰마다 개별 Fully Connected** layer를 붙여 한국어 용으로 미세 조정하기 위한 모델 클래스를 정의하였다.

- **작은 실습 환경에 맞춰 크기 조정**
  - 레이어 : 12 ▷ 6
  - hidden neuron : 768 ▷ 512

일반 상식 질의 응답

일반 상식 질의 응답

```
def do_predict(model, question, context):
    """
    입력에 대한 답변 생성하는 함수
    :param model: model
    :param question: 입력 문자열
    :param context: 입력 문자열
    """
    q_tokens = vocab.encode_as_pieces(question)[:args.max_query_length]
    c_tokens = vocab.encode_as_pieces(context)[:args.max_seq_length - len(q_tokens) - 3]
    tokens = ['[CLS]'] + q_tokens + ['[SEP]'] + c_tokens + ['[SEP]']
    token_ids = [vocab.piece_to_id(token) for token in tokens]
    segments = [0] * (len(q_tokens) + 2) + [1] * (len(c_tokens) + 1)
```

pretrained model을 활용해서 입력에 대한 답변을 생성하는 함수를 정의한 후
결과 및 성능을 확인한다.

일반 상식 질의 응답

```
0
질문 :  패밀리가 떴다는 어디에서 방송했어
지문 :  2010년 2월 14일 패밀리가 떴다가 종영되고, 패밀리가 떴다 2가 2월 21일 방영되었다.
정답 :  SBS
예측 :  SBS
```

1번 문제 경우

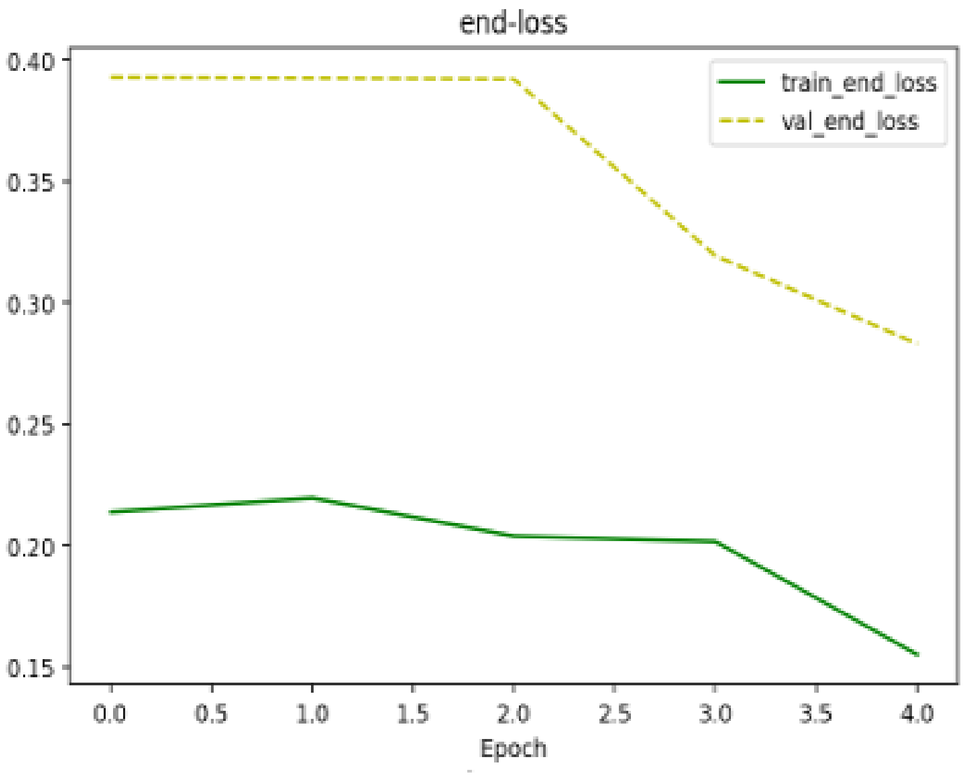
토큰화된 단어 단위의 정답을 제대로
유추하였다는 것을 확인할 수 있다.

2, 3번 문제 경우

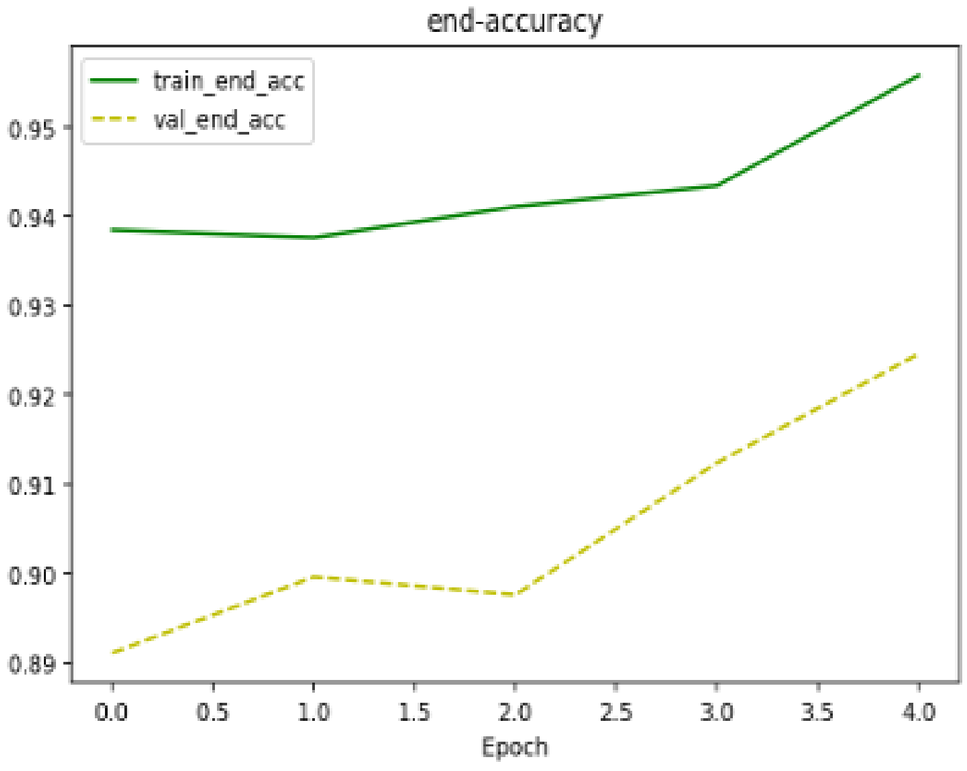
토큰화된 단어에서
subword segmentation 되지 않은 정답을
추측하였다는 것을 확인할 수 있다.

```
10
질문 :  다게스탄 공화국에 헌법이 만들어진 게 언제야
지문 :  1813년에 이 지역은 러시아에 합병되었다. 1830년대부터 1850년대까지 이곳은 계속 반러
정답 :  1994년
예측 :  1994년에는

11
질문 :  오키나와 사회대중당은 언제 창당했지
지문 :  오키나와 사회대중당은 류큐 열도 미국민정부 동안인 1950년 10월 31일에 다이라 다쯔오
정답 :  1950년 10월 31일
예측 :  1950년 10월 31일에
```

일반 상식 질의 응답
- 최종 성능



약 0.3의 오차



약 92%의 정확도

Thank You For Watching!