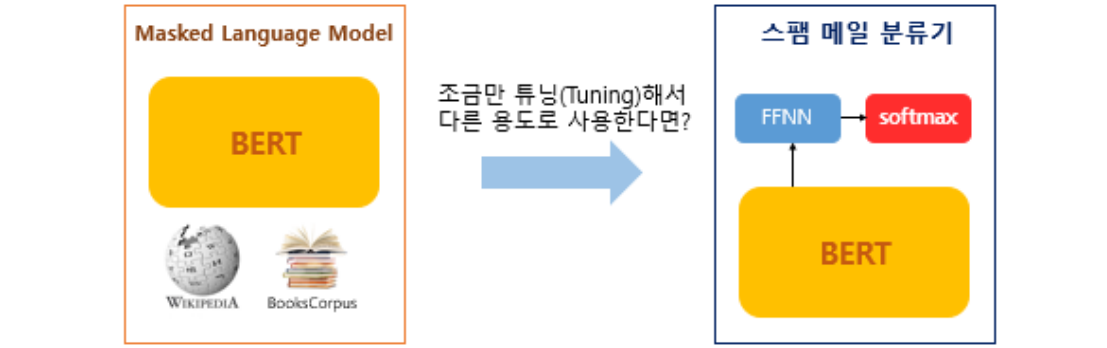
BERT

BERT 개요

BERT(Bidirectional Encoder Representations from Transformers)는 2018년에 구글이 공개한 사전 훈련된 모델, **트랜스포머**를 이용하여 구현, 위키피디아(25억 단어)와 BooksCorpus(8억 단어)와 같은 레이블이 없는 텍스트 데이터로 **사전 훈련된 언어 모델**

레이블이 없는 방대한 데이터로 사전 훈련된 모델을 가지고, 레이블이 있는 다른 작업(Task)에서 추가 훈련과 함께 하이퍼파라미터를 재조정: **파인 튜닝(Fine-tuning)**



33억 단어에 대해서 4일간 학습시킨 언어 모델

BERT의 지식을 이용한 스팸 메일 분류기

하고 싶은 태스크가 스팸 메일 분류라고 하였을 때, 이미 위키피디아 등으로 사전 학습된 BERT 위에 분류를 위한 **신경망을 한 층 추가**

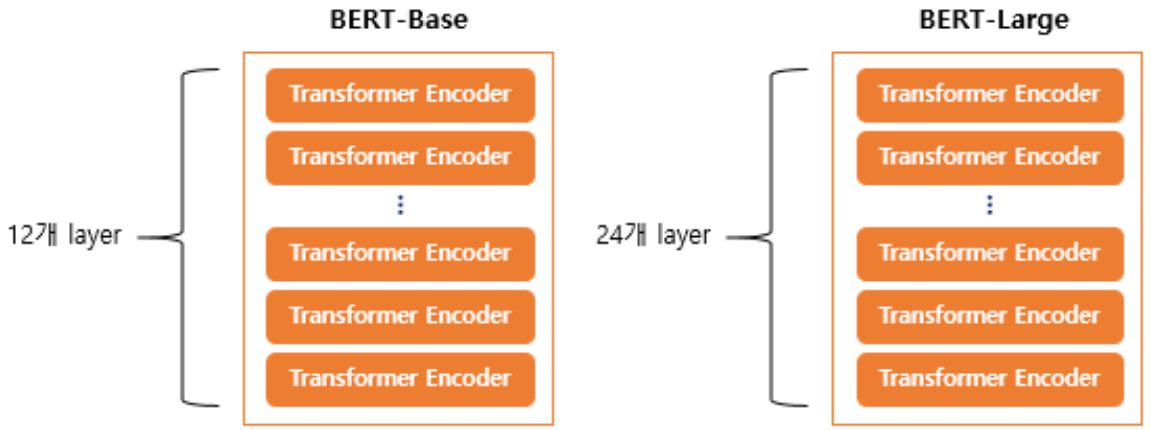
BERT의 크기

BERT의 기본 구조: **트랜스포머의 인코더를 쌓아 올린 구조** (Base 버전: 총 12개, Large 버전: 총 24개)

L: 트랜스포머 인코더 층의 수
D: d_model의 크기
A: 셀프 어텐션 헤드의 수

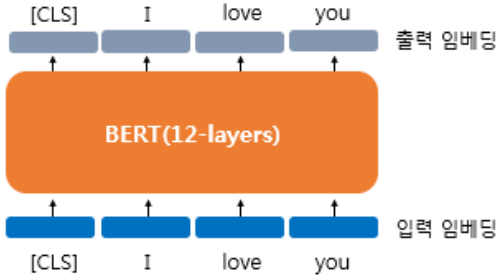
BERT-Base : L=12, D=768, A=12 : 110M개의 파라미터
BERT-Large : L=24, D=1024, A=16 : 340M개의 파라미터

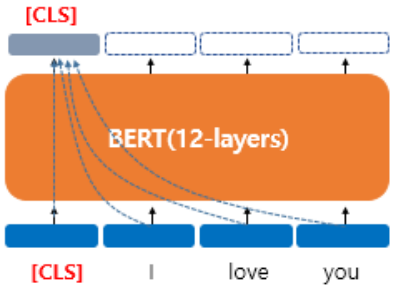
- Base는 GPT-1과의 비교를 위해 같은 파라미터 크기, Large는 최대 성능을 보여주기 위한 파라미터 크기

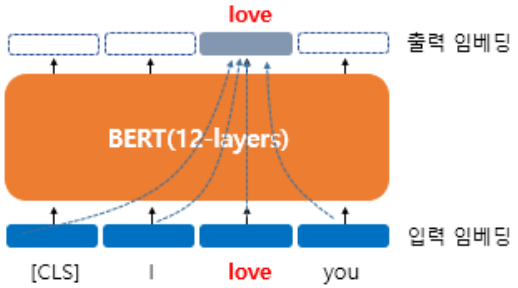


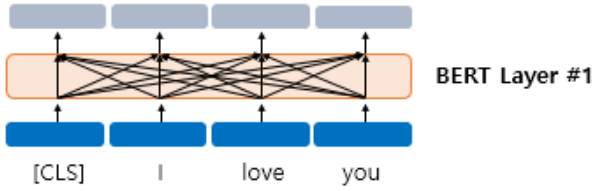
Contextual Embedding

BERT는 ELMo나 GPT-1과 마찬가지로 **문맥을 반영한 임베딩(Contextual Embedding)**을 사용

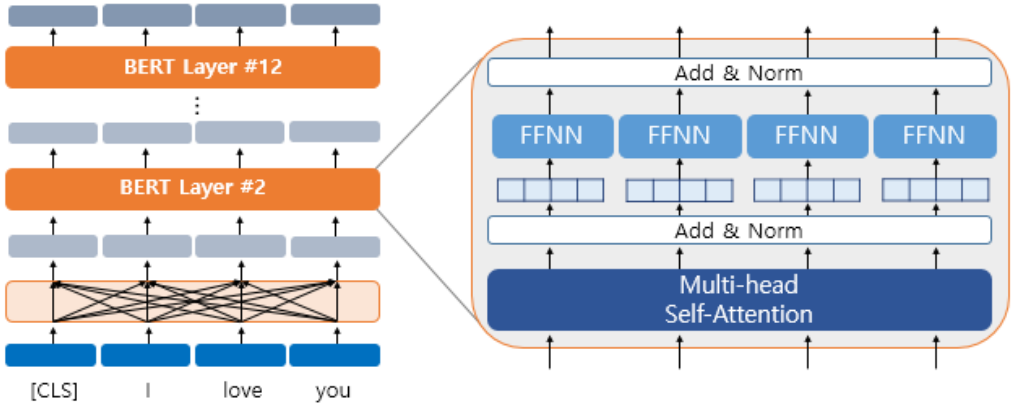








BERT의 입력은 단순히 임베딩 층(embedding layer)를 지난 임베딩 벡터였지만, BERT를 지나고 나서는 [CLS], I, love, you라는 **모든 단어 벡터들을 참고한 후에 문맥 정보를 가진 벡터가 됨**



- 12개의 층을 지난 후에 최종적으로 출력 임베딩을 얻게 됨
- BERT의 첫번째 층의 출력 임베딩은 BERT의 두번째 층에서는 입력 임베딩이 됨
- 내부적으로 각 층마다 멀티 헤드 셀프 어텐션과 포지션 와이즈 피드 포워드 신경망을 수행

서브워드 토크나이저 : WordPiece

- BERT는 **단어보다 더 작은 단위로 쪼개는** 서브워드 토크나이저를 사용
- BERT가 사용한 토크나이저는 **WordPiece**
  - 바이트 페어 인코딩(Byte Pair Encoding, BPE)의 유사
  - 글자로부터 서브워드들을 병합해가는 방식으로 최종 단어 집합(Vocabulary)을 만듦

- 서브워드 토크나이저는 자주 등장하지 않는 단어의 경우에는 더 작은 단위인 서브워드로 분리되어 서브워드들이 단어 집합에 추가. 이 단어 집합을 기반으로 토큰화를 수행.

```
준비물 : 이미 훈련 데이터로부터 만들어진 단어 집합

1. 토큰이 단어 집합에 존재한다.
=> 해당 토큰을 분리하지 않는다.

2. 토큰이 단어 집합에 존재하지 않는다.
=> 해당 토큰을 서브워드로 분리한다.
=> 해당 토큰의 첫번째 서브워드를 제외한 나머지 서브워드들은 앞에 "##"를 붙인 것을 토큰으로 한다.
```

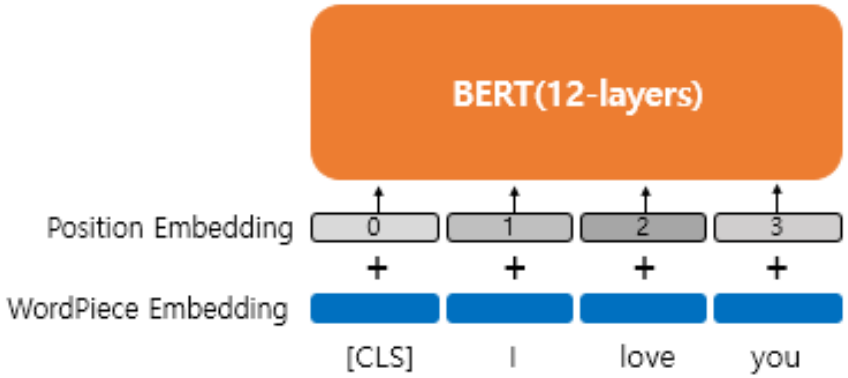
```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased") # Bert-base의 토크나이저
```

```
result = tokenizer.tokenize('Here is the sentence I want embeddings for.')
print(result)
```

```
['here', 'is', 'the', 'sentence', 'i', 'want', 'em', '##bed', '##ding', '##s', 'for', '.']
```

포지션 임베딩(Position Embedding)

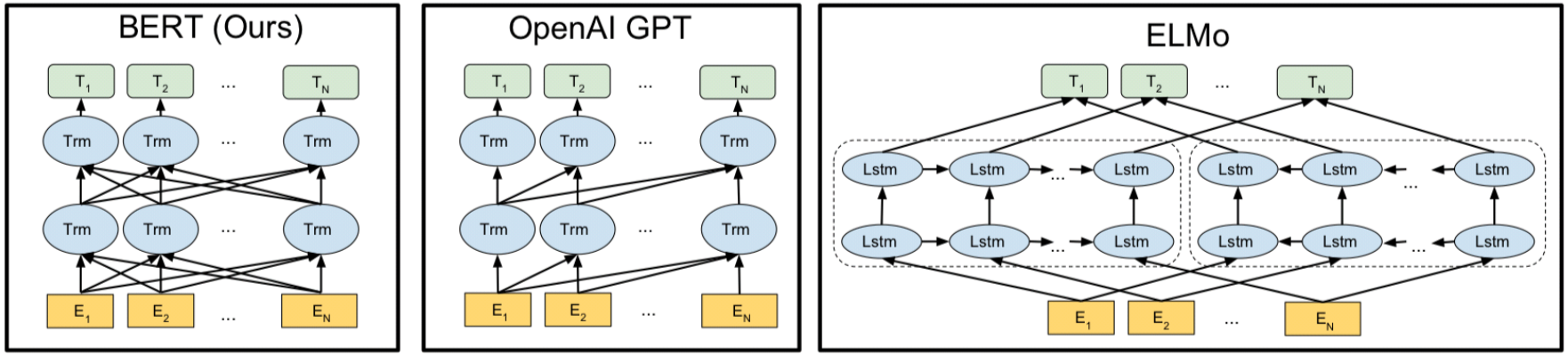
- 입력(Wordpiece Embedding)에 포지션 임베딩을 통해서 **위치 정보**를 더해 줌
- **위치 정보를 위한 임베딩 층(Embedding layer)**을 하나 더 사용하는 것
- 가령, 문장의 길이가 4라면 4개의 포지션 임베딩 벡터를 학습시키고, BERT의 입력마다 다음과 같이 포지션 임베딩 벡터를 더해 줌



- 첫번째 단어의 임베딩 벡터 + 0번 포지션 임베딩 벡터
- 두번째 단어의 임베딩 벡터 + 1번 포지션 임베딩 벡터
- 세번째 단어의 임베딩 벡터 + 2번 포지션 임베딩 벡터
- 네번째 단어의 임베딩 벡터 + 3번 포지션 임베딩 벡터

실제 BERT에서는 문장의 최대 길이를 512로 하고 있으므로, 총 512개의 포지션 임베딩 벡터가 학습됨

사전 훈련(Pre-training)



ELMo는 정방향 LSTM과 역방향 LSTM을 각각 훈련시키는 방식의 양방향 언어 모델

GPT-1은 트랜스포머의 디코더를 이전 단어들로부터 다음 단어를 예측하는 방식의 단방향 언어 모델을

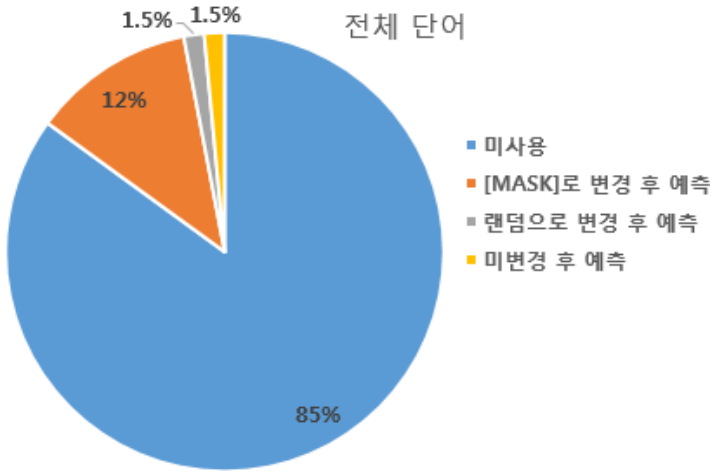
BERT는 GPT와 달리 화살표가 **양방향**으로 뻗어 나감. 이는 마스크드 언어 모델(Masked Language Model)을 통해 양방향성을 얻었기 때문.

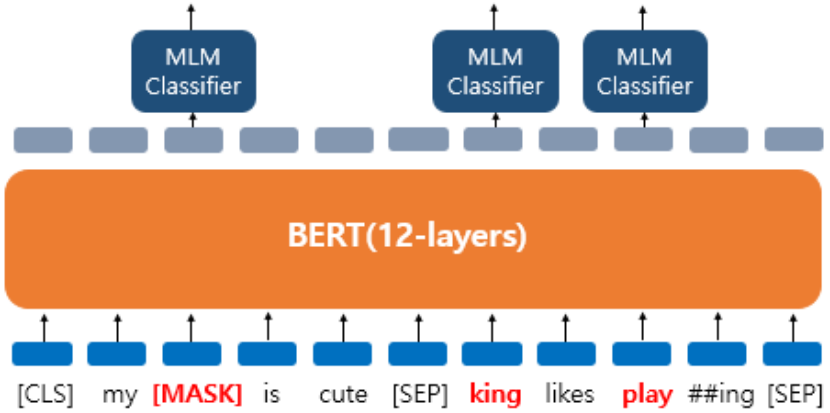
BERT의 사전 훈련 방법은 크게 두 가지
- 첫번째는 **마스크드 언어 모델(Masked Language Model, MLM)**
- 두번째는 **다음 문장 예측(Next sentence prediction, NSP)**

사전 훈련 - Masked Language Model(MLM)

BERT는 사전 훈련을 위해서 인공 신경망의 입력으로 들어가는 입력 텍스트의 15%의 단어를 랜덤으로 마스킹(Masking)하고 이 단어들을 예측하도록 함.

- 단, 전부 [MASK]로 변경하지는 않고, 그 중 80%는 [MASK]로, 10%는 랜덤의 다른 단어로, 10%는 동일하게 놔둠





- 'dog' 토큰은 [MASK]로 변경되었습니다.
- 'he'는 랜덤 단어 'king'으로 변경되었습니다.
- 'play'는 변경되진 않았지만 예측에 사용됩니다.

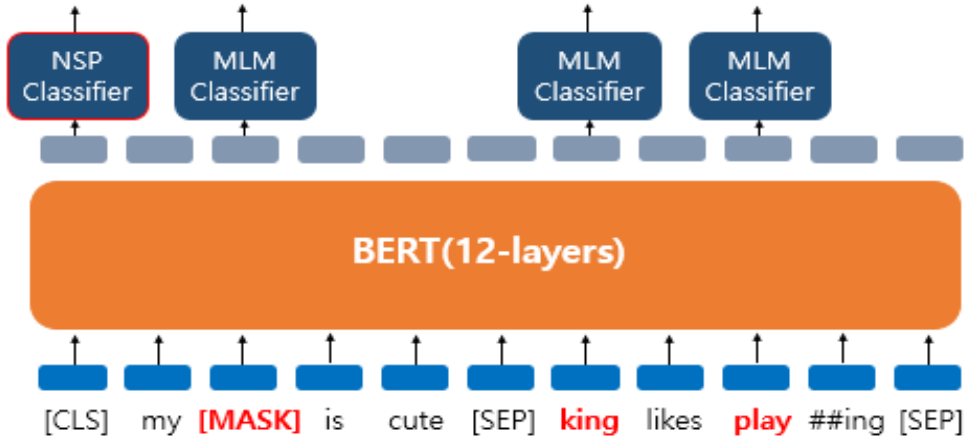
사전 훈련 - 다음 문장 예측(NSP)

BERT는 두 개의 문장을 준 후에 이 문장이 **이어지는 문장**인지 **아닌지**를 맞추는 방식으로 훈련

반은 실제 이어지는 두 개의 문장, 반은 랜덤으로 이어 붙인 두 개의 문장으로 훈련

이어지는 문장의 경우
Sentence A : The man went to the store.
Sentence B : He bought a gallon of milk.
Label = IsNextSentence

이어지는 문장이 아닌 경우 경우
Sentence A : The man went to the store.
Sentence B : dogs are so cute.
Label = NotNextSentence



- BERT의 입력으로 넣을 때에는 **[SEP]**라는 특별 토큰을 사용해서 문장을 구분
- 첫번째 문장의 끝에 [SEP] 토큰을 넣고, 두번째 문장이 끝나면 역시 [SEP] 토큰을 붙여줌
- 그리고 이 두 문장이 실제 이어지는 문장인지 아닌지를 **[CLS] 토큰**(BERT가 분류 문제를 풀기 위해 추가된 특별 토큰)의 위치의 출력층에서 이진 분류 문제를 풀도록 함

MLM과 **NSP**는 따로 학습하는 것이 아니라 loss를 합하여 학습이 **동시에** 이루어짐

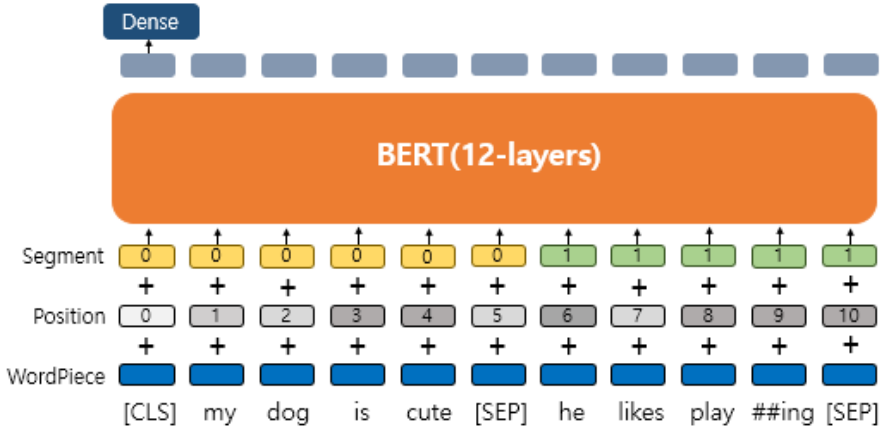
세그먼트 임베딩(Segment Embedding)

BERT는 QA(Question Answering) 같은 두 개의 문장 입력이 필요한 태스크를 풀기도 함

이 때, 문장 구분을 위해서 BERT는 **세그먼트 임베딩**이라는 또 다른 임베딩 층(Embedding layer)을 사용

첫번째 문장에는 Sentence 0 임베딩, 두번째 문장에는 Sentence 1 임베딩을 더해주는 방식

단, 여기서 두 개의 '문장'이라는 표현에서 실제로 문장은 두 종류의 텍스트, 두 개의 문서일 수 있음



두 개의 문장을 입력 받을 필요가 없는 경우도 있음.
예를 들어 감성 분류 태스크에서는 한 개의 문서에 대해서만 분류를 하는 것이므로, 이 경우에는 BERT의 전체 입력에 Sentence 0 임베딩만 더함.

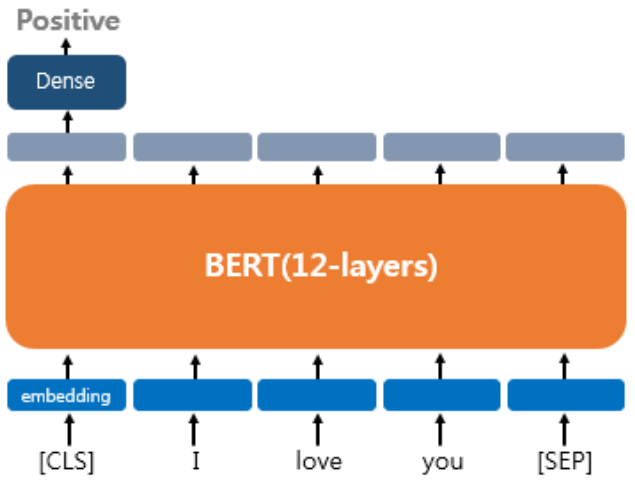
- ✓ WordPiece Embedding : 실질적인 입력이 되는 워드 임베딩. 임베딩 벡터의 종류는 단어 집합의 크기로 30,522개
- ✓ Position Embedding : 위치 정보를 학습하기 위한 임베딩. 임베딩 벡터의 종류는 문장의 최대 길이인 512개
- ✓ Segment Embedding : 두 개의 문장을 구분하기 위한 임베딩. 임베딩 벡터의 종류는 문장의 최대 개수인 2개

파인 튜닝(Fine-Tuning)

Single Text Classification

하나의 문서에 대한 텍스트 분류
- 이 유형은 영화 리뷰 감성 분류, 로이터 뉴스 분류 등과 같이 입력된 문서에 대해서 분류를 하는 유형

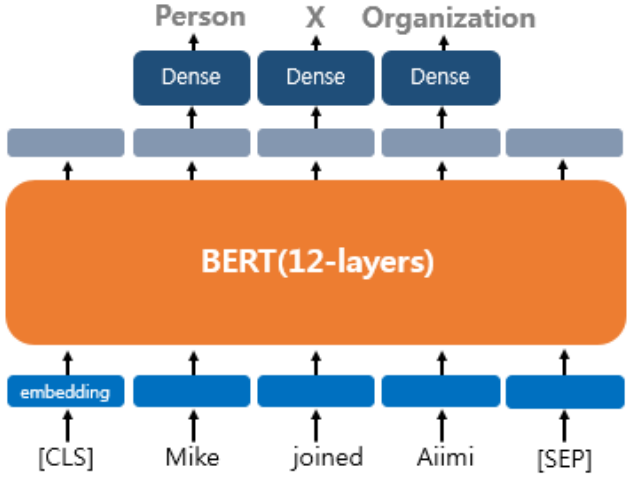
- 문서의 시작에 [CLS] 라는 토큰을 입력
- 텍스트 분류 문제를 풀기 위해서 [CLS] 토큰의 위치의 출력층에서 밀집층(Dense layer) 또는 완전 연결층(fully-connected layer)이라고 불리는 층들을 추가하여 분류에 대한 예측을 함



Tagging

- 문장의 각 단어에 품사를 태깅하는 품사 태깅 작업과 개체를 태깅하는 개체명 인식 작업이 있음

- 출력층에서는 입력 텍스트의 각 토큰의 위치에 밀집층을 사용하여 분류에 대한 예측을 함

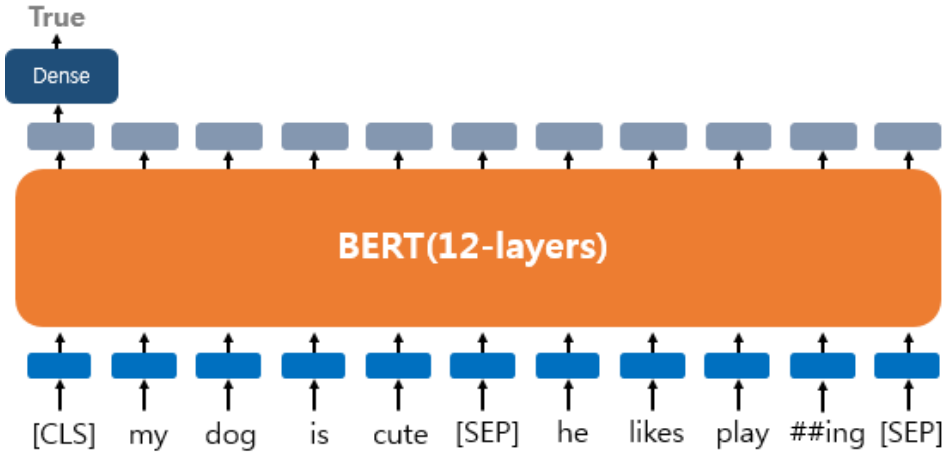


파인 튜닝(Fine-Tuning)

Text Pair Classification or Regression

텍스트의 쌍을 입력으로 받는 태스크(예: 자연어 추론(Natural language inference))
- **자연어 추론 문제**란, 두 문장이 주어졌을 때, 하나의 문장이 다른 문장과 논리적으로 **어떤 관계**에 있는지를 분류하는 것
- 유형으로는 모순 관계(contradiction), 함의 관계(entailment), 중립 관계(neutral)가 있음

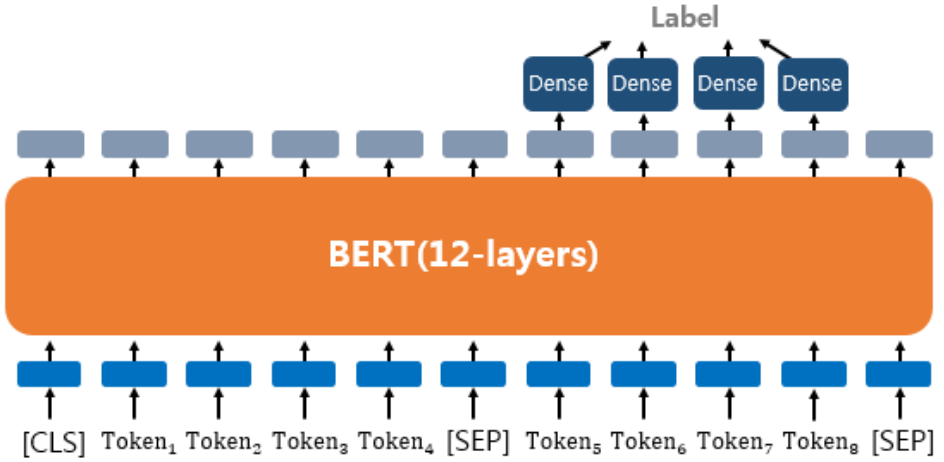
- 이 때 입력 텍스트가 1개가 아니므로, 텍스트 사이에 [SEP] 토큰을 집어넣고, **세그먼트 임베딩**을 사용하여 문서를 구분



Question Answering

질문과 본문이라는 두 개의 텍스트의 쌍을 입력
- 질문과 본문을 입력 받으면, 본문의 일부분을 추출해서 질문에 답변

- Ex."강우가 떨어지도록 영향을 주는 것은?" 라는 질문이 주어지고, "기상학에서 강우는 대기 수증기가 응결되어 중력의 영향을 받고 떨어지는 것을 의미합니다 ..." 라는 본문이 주어졌을 때, 정답은 "중력"



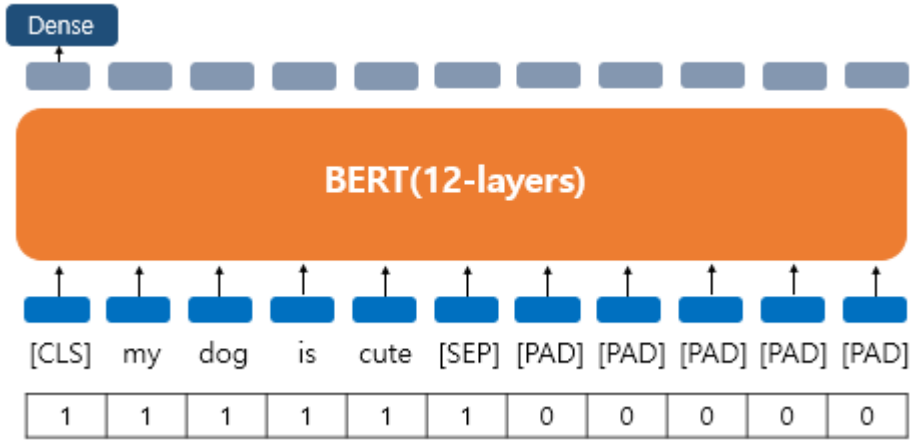
어텐션 마스크(Attention Mask)

BERT를 실제로 실습하게 되면 어텐션 마스크라는 시퀀스 입력이 추가로 필요함

어텐션 마스크는 BERT가 어텐션 연산을 할 때, 불필요하게 패딩 토큰에 대해서 어텐션을 하지 않도록 실제 단어와 패딩 토큰을 구분할 수 있도록 알려주는 입력

이 값은 0과 1 두 가지 값을 가지는데,
숫자 1은 해당 토큰은 실제 단어이므로 마스킹을 하지 않는다라는 의미, 숫자 0은 해당 토큰은 패딩 토큰이므로 마스킹을 한다는 의미

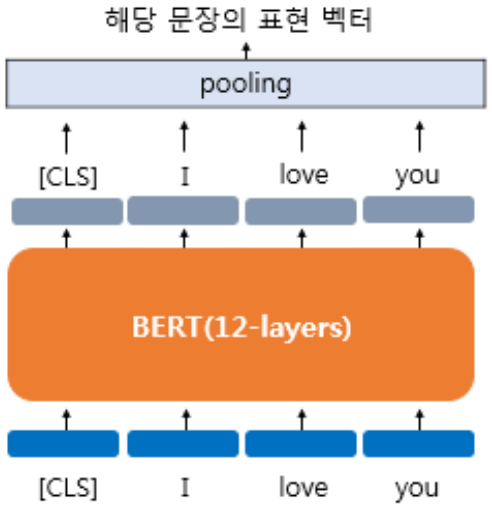
실제 단어의 위치에는 1, 패딩 토큰의 위치에는 0의 값을 가지는 시퀀스를 만들어 BERT의 또 다른 입력으로 사용



기타 사항

- 훈련 데이터는 위키피디아(25억 단어)와 BooksCorpus(8억 단어) ≈ 33억 단어
- WordPiece 토크나이저로 토큰화를 수행 후 15% 비율에 대해서 마스크드 언어 모델 학습
- 두 문장 Sentence A와 B의 합한 길이. 즉, 최대 입력의 길이는 512로 제한
- 100만 step 훈련 ≈ (총 합 33억 단어 코퍼스에 대해 40 에포크 학습)
- 옵티마이저 : 아담(Adam)
- 학습률(learning rate) : 10^{-4}
- 가중치 감소(Weight Decay) : L2 정규화로 0.01 적용
- 드롭 아웃 : 모든 레이어에 대해서 0.1 적용
- 활성화 함수 : relu 함수가 아닌 gelu 함수
- 배치 크기(Batch size) : 256

센텐스버트(Sentence BERT, SBERT)



문장 벡터를 얻는 방법:

1. [CLS] 토큰 자체를 입력 문장의 벡터로 간주
2. BERT의 각 단어에 대한 출력 벡터들에 대해서 평균을 냄(평균 풀링, mean pooling)
3. 각 단어의 출력 벡터들에 대해서 평균 풀링 대신 맥스 풀링(max pooling)

평균 풀링을 얻은 문장 벡터의 경우에는 모든 단어의 의미를 반영
맥스 풀링을 얻은 문장 벡터의 경우에는 중요한 단어의 의미를 반영

SBERT(센텐스버트, Sentence-BERT)는 기본적으로 BERT의 문장 임베딩의 성능을 우수하게 개선시킨 모델

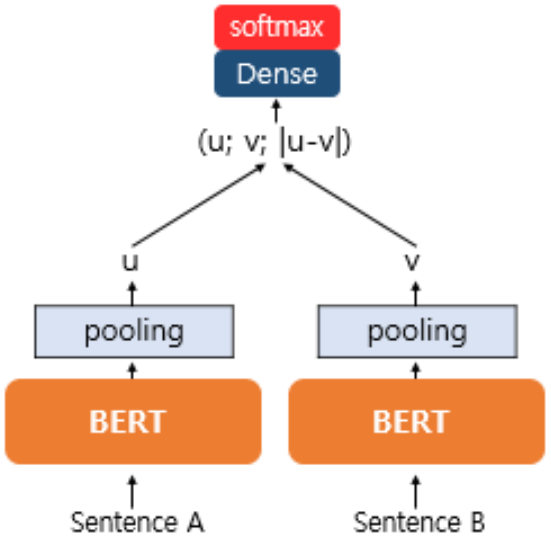
BERT의 문장 임베딩을 응용하여 BERT를 파인 튜닝

센텐스버트(Sentence BERT, SBERT)

문장 쌍 분류 태스크로 파인 튜닝

대표적으로는 NLI(Natural Language Inferencing) 문제

문장 A	문장 B	레이블
A lady sits on a bench that is against a shopping mall.	A person sits on the seat.	Entailment
A lady sits on a bench that is against a shopping mall.	A woman is sitting against a building.	Entailment
A lady sits on a bench that is against a shopping mall.	Nobody is sitting on the bench.	Contradiction
Two women are embracing while holding to go packages.	The sisters are hugging goodbye while holding to go packages after just eating lunch.	Neutral



문장 A,B BERT 입력 → 평균 풀링 or 맥스 풀링 → 문장 임베딩 벡터 u, v
→ 벡터간 차이 $|u-v|$ → 벡터 연결 $h = (u;\ v;\ |u-v|)$
→ 출력층으로 보내 다중 클래스 분류 문제 풀기 → 소프트맥스 함수 통과
→ 레이블과의 오차 줄이며 학습

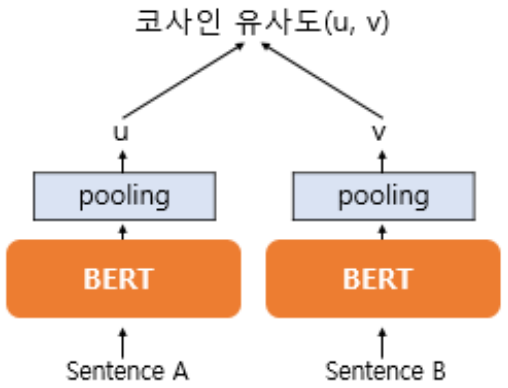
$$o = softmax(W_y h)$$

센텐스버트(Sentence BERT, SBERT)

문장 쌍 회귀 태스크로 파인 튜닝

대표적으로 STS(Semantic Textual Similarity) 문제(두 개의 문장으로부터 의미적 유사성을 구하는 문제)

문장 A	문장 B	레이블
A plane is taking off.	An air plane is taking off.	5.00
A man is playing a large flute.	A man is playing a flute.	3.80
A man is spreading shreded cheese on a pizza.	A man is spreading shredded cheese on an uncoo...	3.80
Three men are playing chess.	Two men are playing chess.	2.60
A man is playing the cello.	A man seated is playing the cello.	4.25



문장 A,B BERT 입력 → 평균 풀링 or 맥스 풀링 → 문장 임베딩 벡터 u, v
→ 두 벡터의 코사인 유사도
→ 해당 유사도와 레이블 유사도와의 평균 제곱 오차(Mean Squared Error, MSE)를 최소화

한국어 BERT 실습

```
from transformers import TFBertForMaskedLM
from transformers import AutoTokenizer
```

```
model = TFBertForMaskedLM.from_pretrained('klue/bert-base', from_pt=True)
tokenizer = AutoTokenizer.from_pretrained("klue/bert-base")
```

MLM

```
from transformers import FillMaskPipeline
pip = FillMaskPipeline(model=model, tokenizer=tokenizer)
pip('축구는 정말 재미있는 [MASK]다.')
```

```
[{'score': 0.8963505625724792,
  'sequence': '축구는 정말 재미있는 스포츠 다.',
  'token': 4559,
  'token_str': '스포츠'},
 {'score': 0.02595764957368374,
  'sequence': '축구는 정말 재미있는 거 다.',
  'token': 568,
  'token_str': '거'},
 {'score': 0.010033931583166122,
  'sequence': '축구는 정말 재미있는 경기 다.',
  'token': 3682,
  'token_str': '경기'},
 {'score': 0.007924391888082027,
  'sequence': '축구는 정말 재미있는 축구 다.',
```

NSP

```
# 이어지는 두 개의 문장
prompt = "2002년 월드컵 축구대회는 일본과 공동으로 개최되었던 세계적인 큰 잔치입니다."
next_sentence = "여행을 가보니 한국의 2002년 월드컵 축구대회의 준비는 완벽했습니다."
encoding = tokenizer(prompt, next_sentence, return_tensors='tf')

logits = model(encoding['input_ids'], token_type_ids=encoding['token_type_ids'])[0]

softmax = tf.keras.layers.Softmax()
probs = softmax(logits)
print('최종 예측 레이블 :', tf.math.argmax(probs, axis=-1).numpy())
```

```
최종 예측 레이블 : [0]
```

참고자료

- 딥 러닝을 이용한 자연어 처리 입문(https://wikidocs.net/book/2155)