3장

BERT 활용하기

사전학습 된 BERT 모델

- 토큰에 대해 소문자화를 한 uncased 모델과
  그렇지 않은 cased 모델 존재

(인코더 레이어 수, 은닉 유닛 크기)가
(2, 128)인 tiny,
(4, 256)인 mini
(4, 512)인 small
(8, 512)인 base
(12, 768)인 large 가 있다

- BERT uncased 모델의 경우 가장 일반적으로 사용됨
- 개체명 인식(NER)의 경우 cased 모델을 사용해야 한다

- 사전 학습된 모델의 사용 방법
  - 임베딩을 추출해 특징 추출기로 사용한다
  - 다운스트림 태스크에 맞게 파인 튜닝한다

임베딩을 추출하는 방법

- 예시 문장) I love Paris.

1. 토큰화 — ['i', 'love', 'paris']
2. [CLS], [SEP] 토큰 추가 — ['[CLS]', 'i', 'love', 'paris', '[SEP]']
3. 길이를 맞추기 위해 [PAD] 토큰 추가
   (길이 = 7이라 가정) — ['[CLS]', 'i', 'love', 'paris', '[SEP]', '[PAD]', '[PAD]']
4. 어텐션 마스크 (패딩과 토큰 구분)추가 — [1, 1, 1, 1, 1, 0, 0]
5. 각 토큰을 고유한 토큰ID에 매핑 — [101, 1045, 2293, 3000, 102, 0, 0]

- 각 인코더는 자신의 표현을 다음 인코더로 전송
- 최종 인코더는 문장에 있는 모든 토큰의
  최종 표현 벡터 (임베딩)을 반환

각 토큰의 표현 크기는 은닉 유닛 크기와 같다

[CLS] 토큰은 전체 문장의 집계 표현을 보유한다
→ $R_{[CLS]}$ 토큰은 'I love Paris'의 표현 벡터가 된다

허깅페이스 트랜스포머

In [40]:
```
from transformers import BertModel, BertTokenizer
import torch
```

In [88]:
```
model = BertModel.from_pretrained('bert-base-uncased', output_hidden_states = True)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertModel: ['cls.predictions.decoder.weight', 'cls.predictions.transform.dense.bias', 'cls.seq_relationship.bias', 'cls.predictions.bias', 'cls.seq_relationship.weight', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.transform.dense.weight', 'cls.predictions.transform.LayerNorm.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

In [89]:
```
sentence = 'I love Paris'
```

In [90]:
```
tokens = tokenizer.tokenize(sentence)
print(tokens)
```

```
['i', 'love', 'paris']
```

In [91]:
```
tokens = ['[CLS]'] + tokens + ['[SEP]']
print(tokens)
```

```
['[CLS]', 'i', 'love', 'paris', '[SEP]']
```

In [92]:
```
tokens = tokens + ['[PAD]']*2 # token len=7
print(tokens)
```

```
['[CLS]', 'i', 'love', 'paris', '[SEP]', '[PAD]', '[PAD]']
```

In [93]:
```
attention_mask = [1 if i!='[PAD]' else 0 for i in tokens]
print(attention_mask)
```

```
[1, 1, 1, 1, 1, 0, 0]
```

In [94]:
```
token_ids = tokenizer.convert_tokens_to_ids(tokens)
print(token_ids)
```

```
[101, 1045, 2293, 3000, 102, 0, 0]
```

-

```
In [95]: token_ids = torch.tensor(token_ids).unsqueeze(0)
         attention_mask = torch.tensor(attention_mask).unsqueeze(0)

In [54]: hidden_rep, cls_head = model(token_ids, attention_mask = attention_mask)

In [59]: print(hidden_rep)

         last_hidden_state

In [96]: output = model(token_ids, attention_mask = attention_mask)

In [97]: hidden_rep = output.last_hidden_state
         hidden_rep

Out[97]: tensor([[[-0.0719,  0.2163,  0.0047,  ..., -0.5865,  0.2262,  0.1981],
                  [ 0.2236,  0.6536, -0.2294,  ..., -0.3547,  0.5517, -0.2367],
                  [ 1.0410,  0.7755,  1.0335,  ..., -0.5621,  0.5218, -0.0852],
                  ...,
                  [ 0.6156,  0.1036, -0.1875,  ..., -0.3799, -0.7008, -0.3500],
                  [ 0.0791,  0.4287,  0.4147,  ..., -0.2417,  0.2403,  0.0378],
                  [-0.0165,  0.2459,  0.4566,  ..., -0.2179,  0.1876,  0.0228]]],
                grad_fn=<NativeLayerNormBackward0>)

In [98]: hidden_rep.shape

Out[98]: torch.Size([1, 7, 768])

In [99]: cls_head = output.pooler_output
         cls_head

Out[99]: tensor([[-9.0660e-01, -3.4189e-01, -3.3729e-01,  7.7140e-01,  6.0977e-02,
                  -1.0525e-01,  9.0143e-01,  2.5822e-01, -2.7881e-01, -9.9997e-01,
                  -1.0322e-01,  7.4773e-01,  9.8521e-01,  5.9799e-02,  9.4447e-01,
                  -5.9859e-01, -2.0539e-01, -5.7386e-01,  3.7684e-01, -7.5183e-01,
                   6.6604e-01,  9.9584e-01,  4.2231e-01,  2.2824e-01,  4.9139e-01,
                   9.2378e-01, -6.6123e-01,  9.3111e-01,  9.6157e-01,  6.8816e-01,
                  -6.7706e-01,  1.2696e-01, -9.8735e-01, -1.3884e-01, -4.2742e-01,
                  -9.9151e-01,  3.1577e-01, -7.9245e-01,  1.1233e-01,  2.5441e-02,
                  -9.0001e-01,  2.9572e-01,  9.9972e-01,  2.9508e-02,  9.4610e-02,
                  -2.3375e-01, -1.0000e+00,  2.1042e-01, -8.8451e-01,  4.4717e-01,
                   2.8363e-01,  2.3094e-01,  1.6396e-01,  4.6484e-01,  4.1578e-01,

n [103]: output.hidden_states[0].shape

ut[103]: torch.Size([1, 7, 768])
```

Output에는 3가지가 저장된다

last_hidden_state :
최종 인코더 계층에서 얻은 모든 토큰의 표현

pooler_output :
최종 인코더 계층의 [CLS]표현 토큰을 나타내며 선형 및 tanh 함수에 의해 계산됨

hidden_states :
모든 인코더 계층에서 얻은 모든 토큰의 표현

- Size[1, 7, 768] = [batch_size, sequence_length, hidden_size]

BERT의 모든 인코더 레이어에서 임베딩 추출하기

```
In [82]: output.pooler_output.shape
Out[82]: torch.Size([1, 768])

In [102]: len(output.hidden_states)
Out[102]: 13

In [103]: output.hidden_states[0].shape
Out[103]: torch.Size([1, 7, 768])
```

- Hidden_states[n] = 입력 임베딩 레이어 h_n에서 얻은 모든 토큰의 표현 벡터
- Hidden_states[n][m] = hidden_states[n]의 m번째 단어의 표현 벡터

다운스트림 태스크를 위한 파인 튜닝

- 사전 학습된 BERT를 기반으로 태스크에 맞게 가중치 업데이트

텍스트 분류

- 감정 분석을 한다고 가정
- 문장을 전처리 후 BERT에 토큰을 입력하고 임베딩을 가져온다
- [CLS] 토큰의 임베딩인 $R_{[CLS]}$만 취한다.
- $R_{[CLS]}$를 분류기에 넣고 학습시켜 감정 분석 수행

- 사전 학습된 BERT모델을 파인 튜닝하는건 특징 추출기로 사용하는 것과 어떻게 다를까?

- 파인 튜닝할 때 분류기와 함께 모델의 가중치를 업데이트 함
- 특징 추출기로 사용하면 분류기의 가중치만 업데이트

- 파인 튜닝 중 모델의 가중치를 조정하는 방법
  - 분류 계층과 함께 사전 학습된 BERT모델의 가중치를 업데이트
  - 분류 계층의 가중치만 업데이트
    -> 사전 학습된 BERT를 특징 추출기로 사용하는 것과 같다.

다운스트림 태스크를 위한 BERT 파인튜닝

텍스트 분류 - 감성분석

In [2]:
```
from transformers import BertForSequenceClassification, BertTokenizerFast, Trainer, TrainingArguments
#from nlp import dataset
import torch
import numpy as np
```

In [3]:
```
from nlp import load_dataset
```

In [4]:
```
import matplotlib.pyplot as plt
```

In [4]:
```
dataset = load_dataset('csv', data_files='./imdbs.csv', split='train')
```

```
Using custom data configuration default
```

In [5]:
```
type(dataset)
```

Out[5]:
```
nlp.arrow_dataset.Dataset
```

In [6]:
```
dataset = dataset.train_test_split(test_size=0.3)
```

In [7]:
```
dataset
```

Out[7]:
```
{'train': Dataset(features: {'text': Value(dtype='string', id=None), 'label': Value(dtype='int64', id=None)}, num_rows: 70),
 'test': Dataset(features: {'text': Value(dtype='string', id=None), 'label': Value(dtype='int64', id=None)}, num_rows: 30)}
```

In [44]:
```
train_set = dataset['train']
test_set = dataset['test']
```

토큰화

In [9]:
```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertForSequenceClassification: ['cls.seq_relationship.bias', 'cls.predictions.decoder.weight', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.bias', 'cls.predictions.transform.LayerNorm.bias', 'cls.seq_relationship.weight', 'cls.predictions.transform.dense.weight', 'cls.predictions.transform.dense.bias']
- This IS expected if you are initializing BertForSequenceClassification from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForSequenceClassification from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

In [10]:
```
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

In [11]:
```
tokenizer('I love Paris')
```

Out[11]:
```
{'input_ids': [101, 1045, 2293, 3000, 102], 'token_type_ids': [0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1]}
```

In [12]:
```
tokenizer(['I love Paris', 'birds fly', 'snow fall'], padding = True, max_length=5)
```

```
C:₩Users₩HUMANE-5₩Anaconda3₩envs₩venv_1₩lib₩site-packages₩transformers₩tokenization_utils_base.py:2317: UserWarning: `max_length` is ignored when `padding`=`True` and there is no truncation strategy. To pad to max length, use `padding='max_length'`.
  warnings.warn(
```

Out[12]:
```
{'input_ids': [[101, 1045, 2293, 3000, 102], [101, 5055, 4875, 102, 0], [101, 4586, 2991, 102, 0]], 'token_type_ids': [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]], 'attention_mask': [[1, 1, 1, 1, 1], [1, 1, 1, 1, 0], [1, 1, 1, 1, 0]]}
```

In [45]:
```
def preprocess(data):
    return tokenizer(data['text'], padding=True, truncation=True)
```

In [46]:
```
train_set = train_set.map(preprocess, batched=True, batch_size = len(train_set))
test_set = test_set.map(preprocess, batched=True, batch_size = len(test_set))
```

In [47]:
```
train_set.set_format('torch', columns=['input_ids', 'attention_mask', 'label'])
test_set.set_format('torch', columns=['input_ids', 'attention_mask', 'label'])
```

In [48]:
```
batch_size = 8
epochs = 2
```

In [49]:
```
warmup_steps = 500
weight_decay = 0.01
```

In [92]:
```
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs = epochs,
    per_device_train_batch_size = batch_size,
    per_device_eval_batch_size = batch_size,
    warmup_steps = warmup_steps,
    weight_decay = weight_decay,
    evaluation_strategy='epoch',
    logging_strategy='epoch',
    logging_dir = './logs',
)
```

```
PyTorch: setting up devices
The default value for the training argument `--report_to` will change i
d to use `--report_to all` to get the same behavior as now. You should
```

In [97]:
```
torch.cuda.empty_cache()
```

In [100]:
```
trainer = Trainer(
    model = model,
    args = training_args,
    train_dataset = train_set,
    eval_dataset = test_set,
)
```

In [94]:
```
output= trainer.train()
```

```
C:₩Users₩HUMANE-5₩Anaconda3₩envs₩venv_1₩lib₩site-packages₩transformers₩optimization.py:306: FutureWarning: This implementation of AdamW is deprecated and will be removed in a future version. Use the PyTorch implementation torch.optim.AdamW instead, or set `no_deprecation_warning=True` to disable this warning
  warnings.warn(
***** Running training *****
  Num examples = 70
  Num Epochs = 2
  Instantaneous batch size per device = 8
  Total train batch size (w. parallel, distributed & accumulation) = 8
  Gradient Accumulation steps = 1
  Total optimization steps = 18
```

Epoch	Training Loss	Validation Loss
1	0.579900	0.711291
2	0.593600	0.701339

In [96]:
```
output
```

Out[96]:
```
TrainOutput(global_step=18, training_loss=0.5867517789204916, metrics={'train_runtime': 8.5599, 'train_samples_per_second': 16.355, 'train_steps_per_second': 2.103, 'total_flos': 36835547750400.0, 'train_loss': 0.5867517789204916, 'epoch': 2.0})
```

In [54]:
```
eval_output=trainer.evaluate()
```

```
***** Running Evaluation *****
  Num examples = 30
  Batch size = 8
```

In [55]:
```
eval_output
```

Out[55]:
```
{'eval_loss': 0.7724966406822205,
 'eval_runtime': 0.7152,
 'eval_samples_per_second': 41.945,
 'eval_steps_per_second': 5.593,
 'epoch': 2.0}
```

질문-응답

- 질문에 대한 응답이 포함된 단락과 함께 질문이 제공됨
- 목표는 주어진 질문에 대한 단락에서 답 추출

- BERT 입력은 질문-단락 쌍
- BERT는 단락에서 응답에 해당하는 텍스트 범위를 반환해야 함

- 텍스트 범위의 시작과 끝 찾기
- 시작 벡터 S와 끝 벡터 E

- 단락 내 각 토큰이 응답의 시작 토큰이 될 확률 계산
- 각 토큰 i에 대해 R_i와 S간의 내적에 소프트맥스를 적용
- 시작 토큰이 될 확률이 높은 토큰의 인덱스를 선택

- 유사한 방식으로 끝 토큰 계산

질문-응답

```
In [5]: from transformers import AutoTokenizer, AutoModelForQuestionAnswering

        tokenizer = AutoTokenizer.from_pretrained("bert-large-uncased-whole-word-masking-finetuned-squad")

        model = AutoModelForQuestionAnswering.from_pretrained("bert-large-uncased-whole-word-masking-finetuned-squad")
```

```
In [6]: question = "면역 체계는 무엇입니까?"
        paragraph = "면역 체계는 질병으로부터 보호하는 유기체 내의 다양한 생물학적 구조와 과정의 시스템입니다. 제대로 기능하려면 면역 체계가 바이러
```

토큰화

```
In [7]: question = '[CLS]' + question + '[SEP]'
        paragraph = paragraph + '[SEP]'
```

```
In [8]: question_tokens = tokenizer.tokenize(question)
        paragraph_tokens = tokenizer.tokenize(paragraph)
```

```
In [9]: tokens = question_tokens + paragraph_tokens
        input_ids = tokenizer.convert_tokens_to_ids(tokens)
```

세그먼트 임베딩

```
In [10]: segment_ids = [0]* len(question_tokens)
         segment_ids = segment_ids + [1] * len(paragraph_tokens)
```

```
In [13]: input_ids = torch.tensor([input_ids])
         segment_ids = torch.tensor([segment_ids])
```

```
In [14]: outputs = model(input_ids, token_type_ids = segment_ids)
```

응답 얻기

```
In [15]: start_scores = outputs.start_logits
         end_score = outputs.end_logits
```

```
In [16]: start_index = torch.argmax(start_scores)
```

한글은 자모음이 분리되어 나옴

```
In [16]: start_index = torch.argmax(start_scores)
         end_index = torch.argmax(end_score)

In [17]: print(' '.join(tokens[start_index:end_index+1]))
```

```
ㅈ ##ㅣ ##ㄹ ##ㅂ ##ㅕ ##ㅇ ##ㅇ ##ㅡ ##ㄹ ##ㅗ ##ㅂ ##ㅜ ##ㅌ ##ㅓ ㅂ ##ㅗ ##ㅎ ##ㅗ ##ㅎ ##ㅏ ##ㄴ ##ㅡ ##ㄴ ㅇ ##ㅠ ##ㄱ ##ㅣ ##ㅊ ##ㅖ
ㄴ ##ㅐ ##ㅇ ##ㅢ [UNK] ㅅ ##ㅐ ##ㅇ ##ㅁ ##ㅜ ##ㄹ ##ㅎ ##ㅏ ##ㄱ ##ㅈ ##ㅓ ##ㄱ ㄱ ##ㅜ ##ㅈ ##ㅗ ##ㅇ ##ㅘ ㄱ ##ㅘ ##ㅈ ##ㅓ ##ㅇ ##ㅇ
##ㅢ ㅅ ##ㅣ ##ㅅ ##ㅡ ##ㅌ ##ㅖ ##ㅁ ##ㅇ ##ㅣ ##ㅂ ##ㄴ ##ㅣ ##ㄷ ##ㅏ
```

```
In [19]: print(tokens[start_index:end_index+1])
```

```
['ㅈ', '##ㅣ', '##ㄹ', '##ㅂ', '##ㅕ', '##ㅇ', '##ㅇ', '##ㅡ', '##ㄹ', '##ㅗ', '##ㅂ', '##ㅜ', '##ㅌ', '##ㅓ', 'ㅂ', '##ㅗ', '##ㅎ', '##
ㅗ', '##ㅎ', '##ㅏ', '##ㄴ', '##ㅡ', '##ㄴ', 'ㅇ', '##ㅠ', '##ㄱ', '##ㅣ', '##ㅊ', '##ㅖ', 'ㄴ', '##ㅐ', '##ㅇ', '##ㅢ', '[UNK]', 'ㅅ', '#
#ㅐ', '##ㅇ', '##ㅁ', '##ㅜ', '##ㄹ', '##ㅎ', '##ㅏ', '##ㄱ', '##ㅈ', '##ㅓ', '##ㄱ', 'ㄱ', '##ㅜ', '##ㅈ', '##ㅗ', '##ㅇ', '##ㅘ', 'ㄱ',
'##ㅘ', '##ㅈ', '##ㅓ', '##ㅇ', '##ㅇ', '##ㅢ', 'ㅅ', '##ㅣ', '##ㅅ', '##ㅡ', '##ㅌ', '##ㅖ', '##ㅁ', '##ㅇ', '##ㅣ', '##ㅂ', '##ㄴ', '##
ㅣ', '##ㄷ', '##ㅏ']
```

```
In [24]: answer = tokenizer.convert_tokens_to_string(tokens[start_index:end_index+1])

In [25]: answer
```

```
Out[25]: '질병으로부터 보호하는 유기체 내의 [UNK] 생물학적 구조와 과정의 시스템입니다'
```

개체명 인식

- 개체명을 미리 정의된 범주로 분류하는 것
- ‘Jeremy lives in Paris’에서 Jeremy는 사람, Paris는 위치로 분류
- 문장 전처리 후 BERT 모델에 토큰을 입력하고 모든 토큰의 표현 벡터 획득
- 이러한 토큰 표현을 분류기에 입력 -> 개체명 범주 반환