

3

Huggingface Transformer Library

3.1 허깅페이스 트랜스포머란?
3.2 허깅페이스 허브 탐색하기
 3.2.1 모델 허브
 3.2.2 데이터셋 허브
 3.2.3 모델 데모를 공개하고 사용할 수 있는 스페이스
3.3 허깅페이스 라이브러리 사용법 익히기
 3.3.1 모델 활용하기
 3.3.2 토크나이저 활용하기
 3.3.3 데이터셋 활용하기
3.4 모델 학습하기
 3.4.1 데이터 준비
 3.4.2 Trainer API 사용해 학습하기
 3.4.3 트레이너 API 사용하지 않고 학습하기
 3.4.4 학습한 모델 업로드하기
3.5 모델 추론하기
 3.5.1 파이프라인을 활용한 추론
 3.5.2 직접 추론하기
3.6 정리



요약

- 등장 배경
 - 트랜스포머가 공개되면서 해당 아키텍처를 활용한 모델이 쏟아져 나왔으나 구현 방식에 차이가 있어 모델마다 활용법을 익혀야 했음
 - Huggingface팀이 개발한 Transformers라이브러리는 통합된 인터페이스로 제공됨
 - 트랜스포머 모델을 활용할 수 있도록 지원함으로써 해당 문제를 해결하며 현재는 핵심 라이브러리
- 구성
 - 무슨 라이브러리인지, 왜 각광받는지, 허깅페이스 허브를 알아봄
 - 모델학습과 활용에 꼭 필요한 데이터셋, 모델, 토크나이저를 각각 살펴봄
 - 한국어 데이터셋을 활용해 텍스트 분류모델을 만들어 활용하는 실습 진행
- 라이브러리 설치
 - ```
!pip install transformers==4.40.1 datasets==2.19.0 huggingface_hub==0.23.0 -qqq
```
  - `-qqq` : 거의 모든 메시지를 생략하고, 오직 필수적인 오류만 출력

### 3.1 허깅페이스 트랜스포머란?

트랜스포머 모델을 쉽게 학습하고 추론에 활용할 수 있도록 도움

- transformers 라이브러리 : model, tokenizer를 활용할 때 사용
- datasets 라이브러리 : 데이터셋을 공개하고 쉽게 가져다 쓸 수 있도록 지원하는 라이브러리

▼ ex. `from datasets import load_dataset`

```
from datasets import load_dataset

'imdb' 데이터셋을 불러오기
dataset = load_dataset("imdb")
```

- Huggingface에서 제공하며, 모델 학습을 위한 데이터를 간단한 코드로 가져올 수 있음
- `imdb`라는 영화 리뷰 데이터셋을 불러오며 Huggingface의 데이터셋 허브에 저장되어 있으며, 다양한 공개 데이터셋을 손쉽게 가져와 사용 가능

- ▣ 서로 다른 모델도 거의 동일한 인터페이스로 활용 가능하며 굉장히 쉽게 모델을 불러 올 수 있음

```
from transformers import AutoTokenizer, AutoModel

입력 데이터
text = "What is Huggingface Transformers?"

BERT 모델 활용
bert_model = AutoModel.from_pretrained("bert-base-uncased") # 모델 불러오기
bert_tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased') # 토크나이저 불러오기
encoded_input = bert_tokenizer(text, return_tensors='pt') # 입력 토큰화
bert_output = bert_model(**encoded_input) # 모델에 입력

GPT-2 모델 활용
gpt_model = AutoModel.from_pretrained('gpt2') # 모델 불러오기
gpt_tokenizer = AutoTokenizer.from_pretrained('gpt2') # 토크나이저 불러오기
encoded_input = gpt_tokenizer(text, return_tensors='pt') # 입력 토큰화
gpt_output = gpt_model(**encoded_input) # 모델에 입력
```

- BERT(Google) 와 GPT(OpenAI) 각각 다른 조직에서 개발한 모델
- 아래 문제들을 파악하는 데 많은 시간을 써야 했으나 Huggingface는 이런 불편함을 해소
  - 새로운 모델이 공개될 때마다 모델을 어떻게 불러올 수 있는지
  - 모델이 어떤 함수를 갖고 있는지
  - 어떻게 학습시킬 수 있는지

#### ▼ output

```
print('bert_output ', bert_output[0])
print('gpt_output ',gpt_output[0])

bert_output tensor([[[-0.3009, 0.0158, 0.0698, ..., -0.3406, 0.5976, 0.5820],
 [-0.1109, 0.0754, -0.1986, ..., 0.2970, 0.4278, -0.0391],
 [-0.5813, -0.0642, 0.4034, ..., -0.2549, 0.2216, 0.8121],
 ...
 [0.9971, 0.3301, -0.0688, ..., -0.4873, 0.0168, -0.0345],
 [-0.2394, -0.0573, -0.5885, ..., -0.0415, 0.3123, -0.0288],
 [0.7884, 0.4039, 0.0217, ..., 0.3869, -0.4785, -0.4116]]], grad_fn=<NativeLayerNormBackward0>)
gpt_output tensor([[-0.1643, 0.0957, -0.2844, ..., -0.1632, -0.0774, -0.2154],
 [-0.1234, 0.0442, 0.0246, ..., -0.2752, 0.1524, 0.3333],
 [-0.2047, -0.3868, -0.1290, ..., -0.6443, 0.0570, 0.0601],
 ...
 [-0.8368, 0.3412, -0.6933, ..., -0.2577, 0.3119, -0.1379],
 [-0.6274, -0.0040, -0.4727, ..., -0.4236, 0.3078, -0.0780],
 [-0.3692, -0.4871, -0.3275, ..., -0.2425, -0.3942, -0.0420]]], grad_fn=<ViewBackward0>)
```

## 3.2 허깅페이스 허브 탐색하기

**Hub Python Library documentation**에 따르면,,



Hugging Face Hub는 머신러닝 모델, 데모, 데이터 세트 및 메트릭을 공유할 수 있는 곳

- *huggingface\_hub* 라이브러리는 개발 환경을 벗어나지 않고도 Hub와 상호작용할 수 있도록 도와줌.
- 리포지토리를 쉽게 만들고 관리하거나, 파일을 다운로드 및 업로드하고, 유용한 모델과 데이터 세트의 메타데이터도 구할 수 있음

- ▶ [Model](#) / [Dataset](#) / [Space](#) 세 가지로 크게 진행

### 3.2.1 모델 허브

- ▣ 어떤 Task에 사용하는지, 어떤 Languages로 학습된 모델인지 등 다양한 기준으로 모델 분류

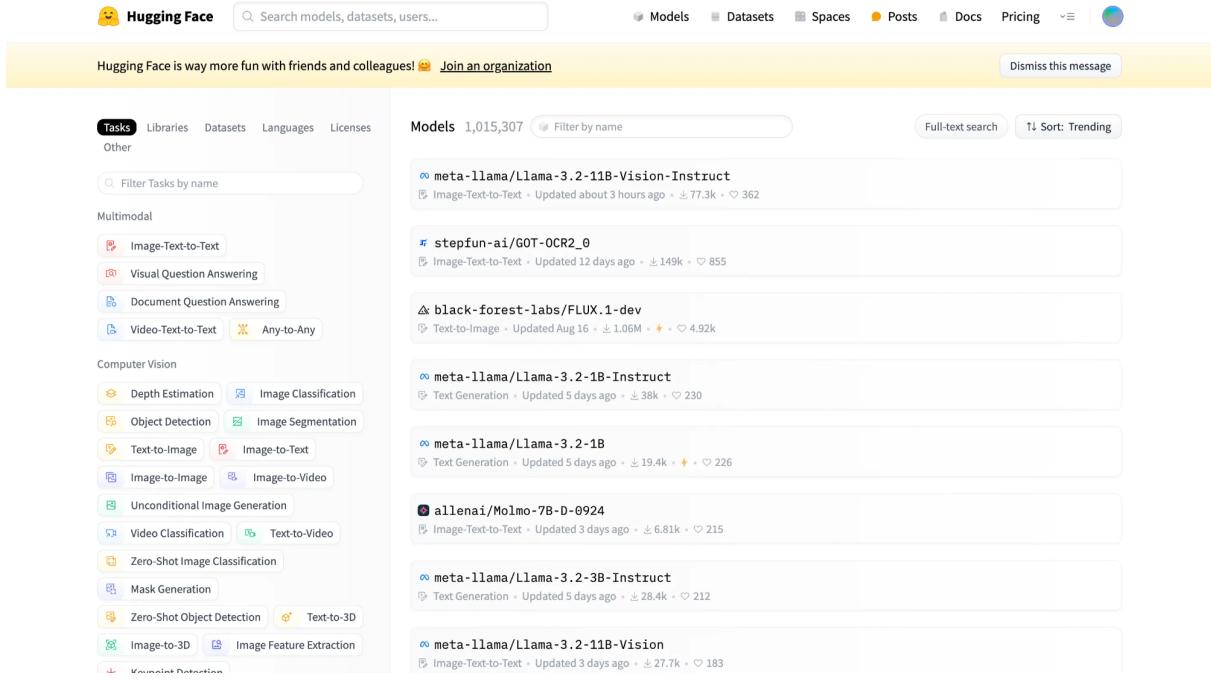


그림 3.1 허깅페이스 모델 허브 스크린샷 화면

- 모델분류
  - 모델 : 작업 종류에 따라 모델 필터링
    - ex. gemma 와 같이 검색 ok
  - 데이터셋 : 허깅페이스의 datasets library에서 제공하는 데이터셋을 탐색
  - 스페이스 : 공개된 스페이스를 탐색
- 전체검색
  - Model, Dataset, Space, User 등 검색

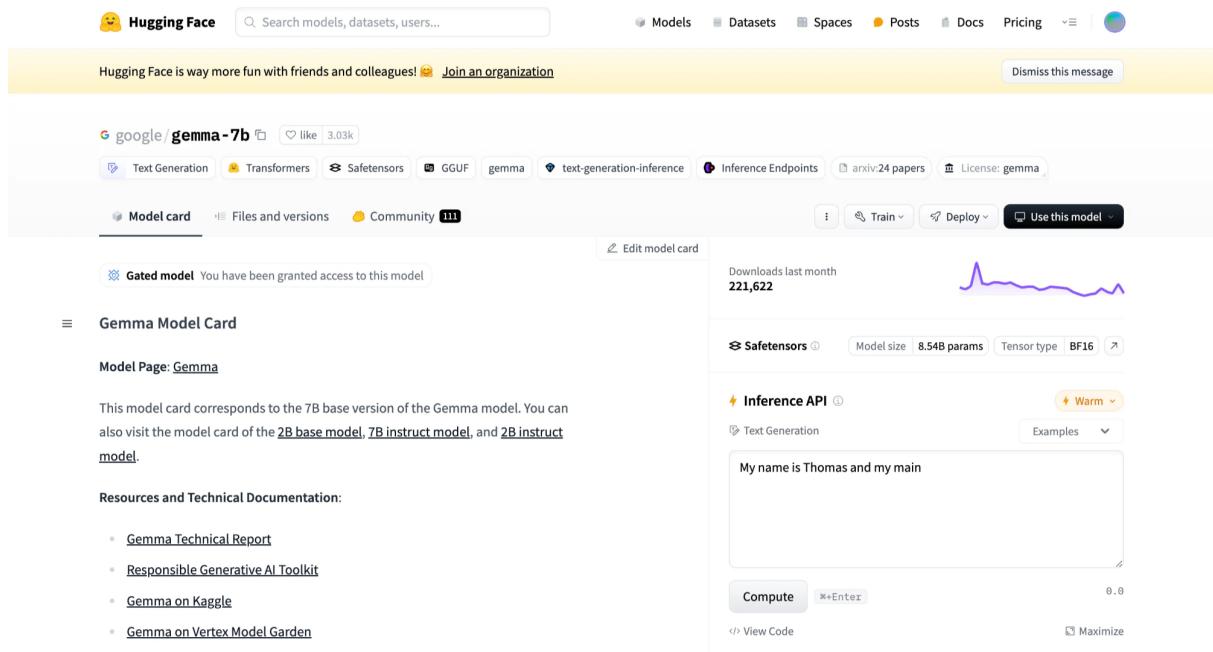


그림 3.2 google gemma

- model 이름 및 요약 정보  
어떤 Task를 위한 모델인지, 라이센스 유형등 파악 ok
- model card  
(optional) 모델 성능, 관련 논문 소개, 사용 방법 등 정보 제공
- model trend  
모델의 다운로드 수 추이를 볼 수 있는 모델 트랜드 그래프
- model test  
모델 간단히 테스트할 수 있는 inference API

### 3.2.2 데이터셋 허브

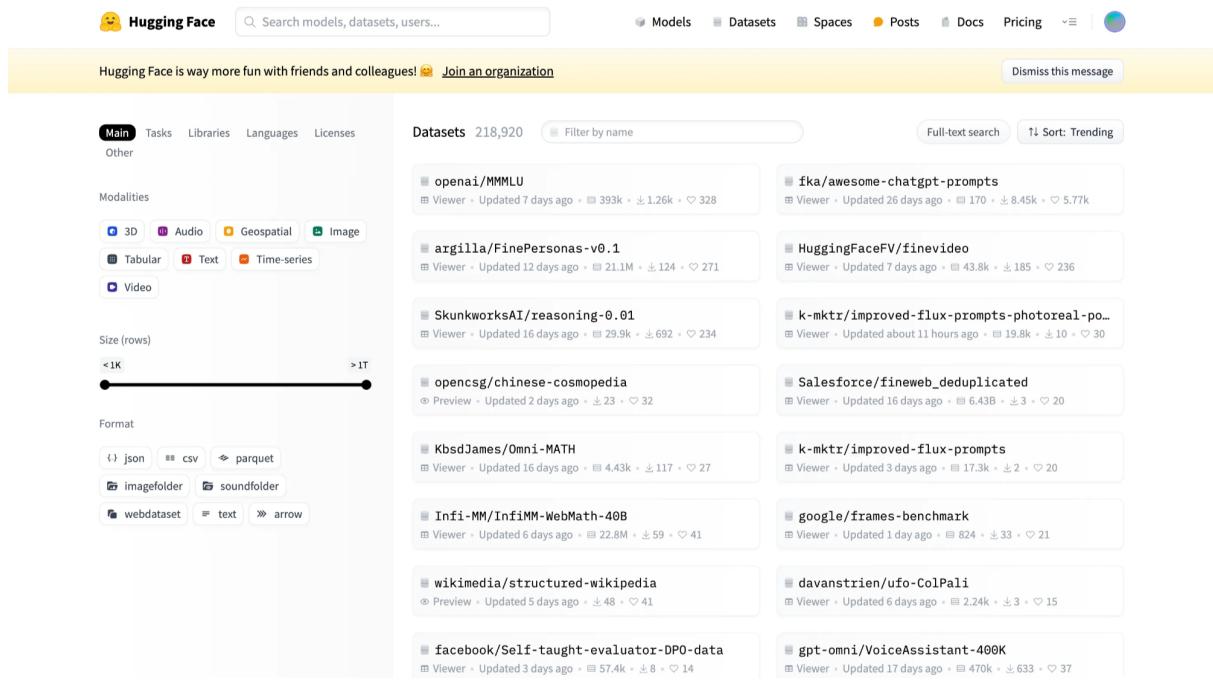


그림 3.3 허깅페이스 데이터셋 허브

- Dataset size, Format → 선택한 기준에 맞는 데이터셋 보여줌

KLUE Korean Language Understanding Evaluation

한국어 언어 이해 평가를 뜻하며 대표적인 한국어 데이터셋 중 하나

- 텍스트 분류, 기계 독해, 문장 유사도 판단 등 다양한 Task에 모델 성능을 평가하기 위해 개발된 Benchmark dataset
  - 데이터 종류는 총 8개
    - ex. MRC Machine Reading Comprehension  
기계 독해 능력 평가를 위한 데이터
    - ex. YNAT Younhap News Agency news headlines for Topic Classification  
토pic 분류 평가를 위한 데이터

- <https://huggingface.co/datasets/klue/klue>

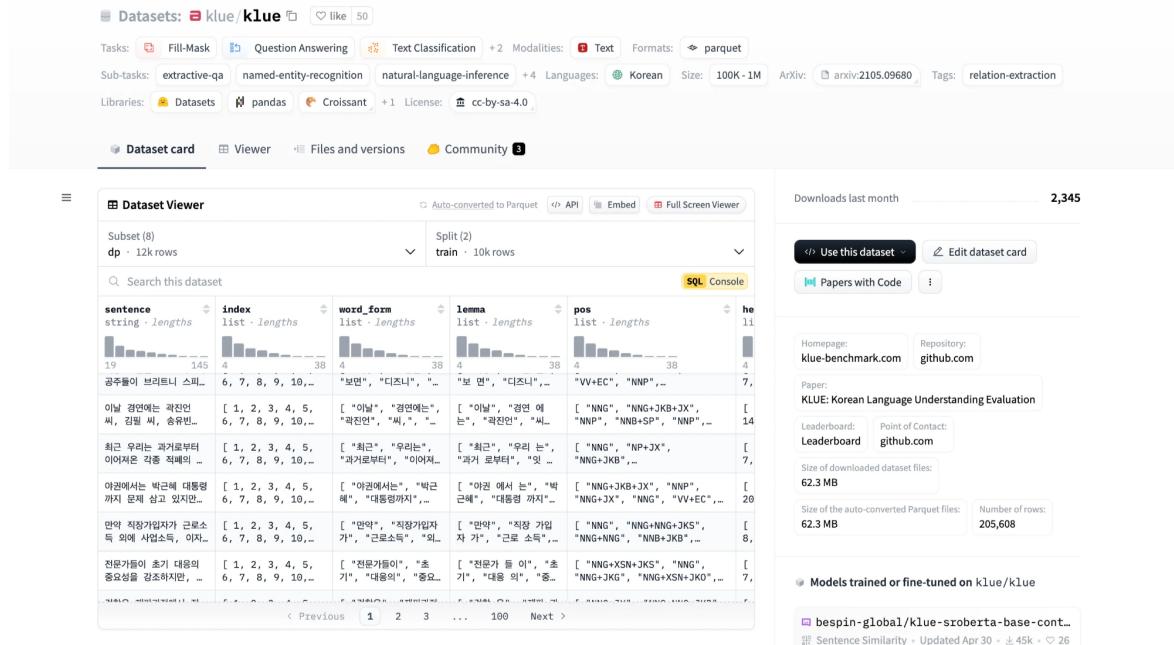
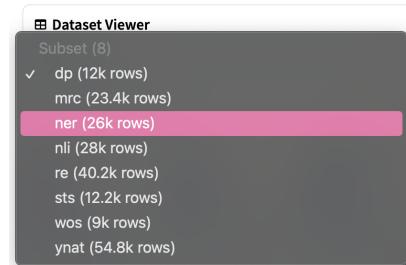


그림 3.4 KLUE 데이터셋

- split 은 train, validation, test 로 구분
  - 하나의 데이터셋에 여러 데이터셋이 포함된 경우 subset으로 구분



### 3.2.3 모델 데모를 공개하고 사용할 수 있는 스페이스

- ▼ 사용자가 자신의 모델 데모를 간편히 공개할 수 있는 기능인 **스페이스**

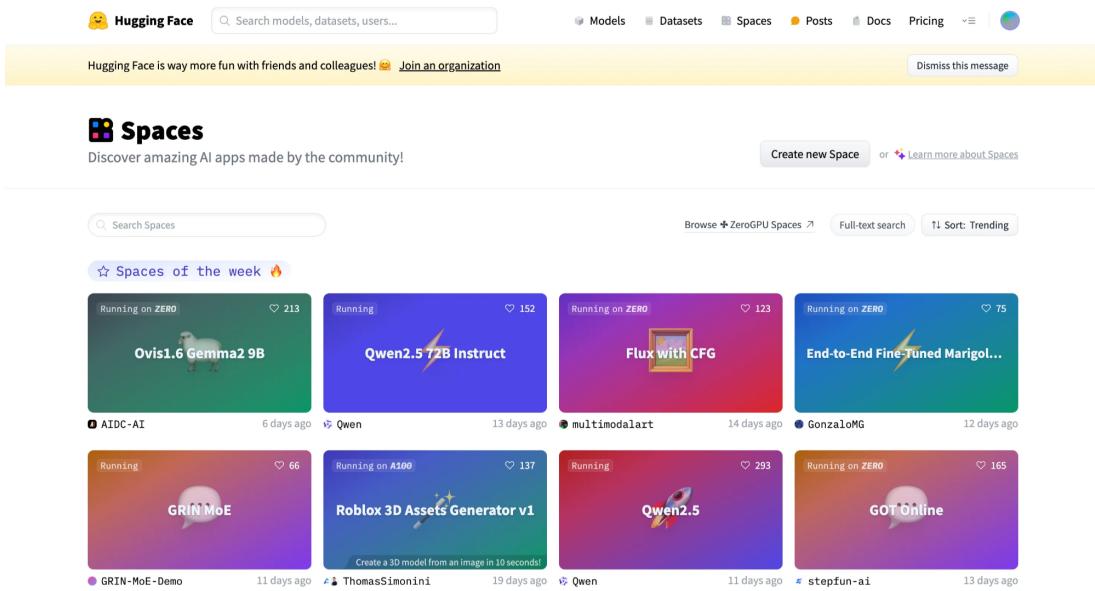


그림 3.5 허깅페이스 스페이스

- 별도의 복잡한 웹페이지 개발없이 모델 대모 공유 가능

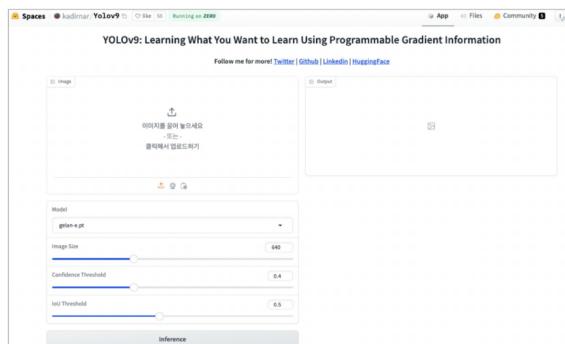


그림 3.6 YOLOv9 스페이스 스크린샷 화면(<https://huggingface.co/spaces/kadimir/Yolov9>)

- YOLOv9은 2024.02 공개된 object detection 모델을 활용
- 이미지 업로드 → 사용할 모델 종류와 추론에 사용할 모델 설정 선택 → inference
- YOLOv9 모델 결과로는 인식도 좋고 어떤 물체인지 구분 ok
- ▼ ! 실제로 적용해본 결과로는 에러 발생함

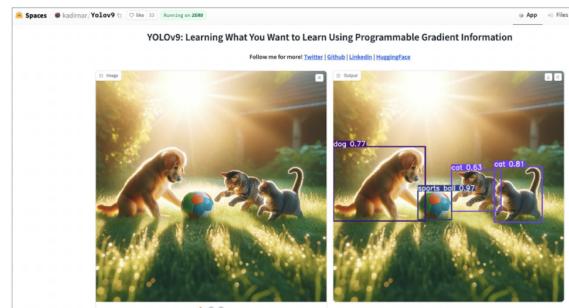
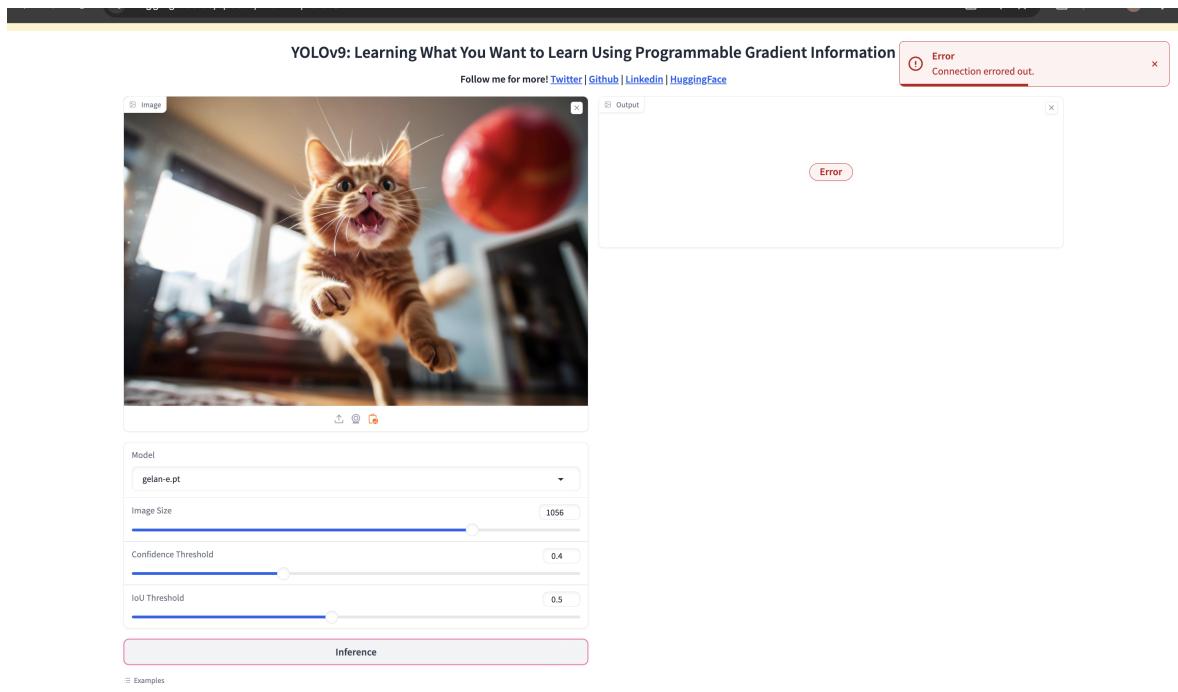


그림 3.7 YOLOv9 추론 예시 스크린샷 화면



## Leaderboard

- 다양한 오픈소스 LLM과 해당 성능 정보를 게시하는 리더보드로 모델 성능 비교하는 웹페이지
- 각 모델의 크기와 성능을 한번에 확인
- <https://huggingface.co/spaces/upstage/open-ko-llm-leaderboard>

| Model                                        | Average | Ko-GPQA | Ko-Winogrande | Ko-GSM8k | Ko-EQ Bench | Ko-IEFVal | Ko-NAT-CKA | Ko-NAT-SVA | Ko-Harmlessness |
|----------------------------------------------|---------|---------|---------------|----------|-------------|-----------|------------|------------|-----------------|
| ai-human-lab/EEVE-Korean_Instruct-10.8B-expo | 47.99   | 23.74   | 69.38         | 49.2     | 53.59       | 37.55     | 34.34      | 49.71      | 65.88           |
| BAAI/Infinity-Instruct-7M-Gen-Llama3_1-8B    | 46.38   | 28.79   | 58.64         | 53.75    | 40.47       | 51.58     | 25.25      | 45.73      | 64.02           |
| MaaData/Myrzhh_solar_10_7b_3.0               | 43.74   | 29.8    | 76.09         | 0.08     | 0           | 20.11     | 48.57      | 48.05      | 88.8            |
| princeton-nlp/Llama-3-Base-8B-SFT-RDPO       | 43.44   | 25.25   | 59.51         | 31.08    | 44.31       | 31.12     | 24.65      | 51.43      | 67.37           |

그림 3.8 한국어 LLM 리더보드

- 모델과 각 모델의 벤치마크 성능 표시
- 벤치마크 데이터셋 종류
  - ex. average : benchmark 평균 점수 의미
- 모델 정보
  - 모델 타입

ex. pretrained, instruction-tuned, RL-tuned

- precision

모델 파라미터의 데이터 형식에 따라 모델 필터링

- Model sizes

◀ 사용자에게 맞는 모델과 데이터셋 탐색 완료

▶ 모델과 데이터셋을 불러와 학습시키거나 추론

### 3.3 허깅페이스 라이브러리 사용법 익히기

| 모델 / 토크나이저 / 데이터 셋으로 학습시키거나 추론

#### 3.3.1 모델 활용하기

허깅페이스 모델을 `Body` + `Head`로 구분해 동일한 바디를 사용하면서 다른 task에 사용 가능

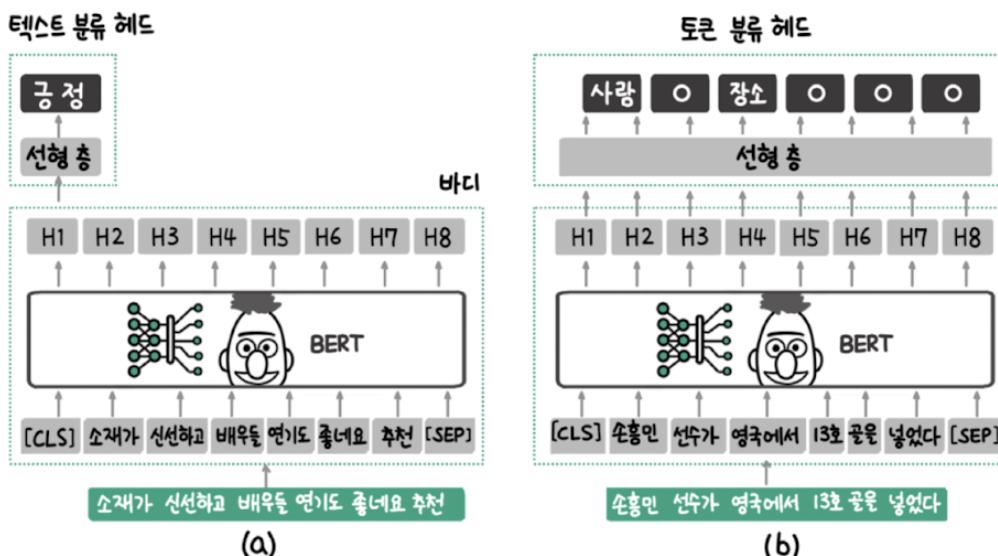


그림 3.9 바디와 헤드의 구분이 필요한 이유 → 긍정부정, NER 개체명 인식

- 모델의 본체 `Body`
  - 공통적인 특성 추출
  - 입력 데이터를 처리하고, 다양한 특성을 추출한 후 hidden state를 생성
- 작업에 맞는 최종 출력 부분 `Head`
  - 생성된 출력을 특정 task에 맞게 변환하는 부분
  - 다양한 task에는 분류, 질문-답변, 번역, 요약 등이 있을 수 있음

▼ 세번의 모델 불러오기

RoBERTa는 구글의 BERT를 개선한 모델

- only `Body`

- `AutoModel`을 사용하여 Transformer Body만 불러옴
- 이 모델은 기본적인 BERT 구조만 가지고 있고, 구체적인 작업을 수행하기 위한 Head는 없음

```
from transformers import AutoModel
model_id = 'klue/roberta-base'
model = AutoModel.from_pretrained(model_id)
```

▼ model architecture

```
RobertaModel(
 (embeddings): RobertaEmbeddings(
 (word_embeddings): Embedding(32000, 768, padding_idx=1)
 (position_embeddings): Embedding(514, 768, padding_idx=1)
 (token_type_embeddings): Embedding(1, 768)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
 (encoder): RobertaEncoder(
 (layer): ModuleList(
 (0-11): 12 x RobertaLayer(
 (attention): RobertaAttention(
 (self): RobertaSelfAttention(
 (query): Linear(in_features=768, out_features=768, bias=True)
 (key): Linear(in_features=768, out_features=768, bias=True)
 (value): Linear(in_features=768, out_features=768, bias=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
 (output): RobertaSelfOutput(
 (dense): Linear(in_features=768, out_features=768, bias=True)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
)
 (intermediate): RobertaIntermediate(
 (dense): Linear(in_features=768, out_features=3072, bias=True)
 (intermediate_act_fn): GELUActivation()
)
 (output): RobertaOutput(
 (dense): Linear(in_features=3072, out_features=768, bias=True)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
)
)
)
 (pooler): RobertaPooler(
 (dense): Linear(in_features=768, out_features=768, bias=True)
 (activation): Tanh()
)
)
```

```
only body
RobertaConfig {
 "_name_or_path": "klue/roberta-base",
 "architectures": [
 "RobertaForMaskedLM"
],
 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "classifier_dropout": null,
 "eos_token_id": 2,
 "gradient_checkpointing": false,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "initializer_range": 0.02,
```

```

 "intermediate_size": 3072,
 "layer_norm_eps": 1e-05,
 "max_position_embeddings": 514,
 "model_type": "roberta", # 모델 종류
 # 여러 설정 파라미터
 "num_attention_heads": 12,
 "num_hidden_layers": 12,
 "pad_token_id": 1,
 "position_embedding_type": "absolute",
 "tokenizer_class": "BertTokenizer", # 토크나이저 클래스
 "transformers_version": "4.44.2",
 "type_vocab_size": 1,
 "use_cache": true,
 "vocab_size": 32000 # 어휘 사전 크기
}

```

## 2. Body + Head

- Text classification 모델로 `AutoModelForSequenceClassification` 클래스 적용
- Text Sequence Classification을 위해 Head가 포함된 모델을 불러옴  
ex. 입력 문장이 어떤 sentiment인지 분류

```

from transformers import AutoModelForSequenceClassification
model_id = 'SamLowe/roberta-base-go_emotions'
classification_model = AutoModelForSequenceClassification.from_pretrained(model_id)

```

### ▼ model architecture

```

RobertaForSequenceClassification(
 (roberta): RobertaModel(
 (embeddings): RobertaEmbeddings(
 (word_embeddings): Embedding(50265, 768, padding_idx=1)
 (position_embeddings): Embedding(514, 768, padding_idx=1)
 (token_type_embeddings): Embedding(1, 768)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
 (encoder): RobertaEncoder(
 (layer): ModuleList(
 (0-11): 12 x RobertaLayer(
 (attention): RobertaAttention(
 (self): RobertaSelfAttention(
 (query): Linear(in_features=768, out_features=768, bias=True)
 (key): Linear(in_features=768, out_features=768, bias=True)
 (value): Linear(in_features=768, out_features=768, bias=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
 (output): RobertaSelfOutput(
 (dense): Linear(in_features=768, out_features=768, bias=True)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
)
 (intermediate): RobertaIntermediate(
 (dense): Linear(in_features=768, out_features=3072, bias=True)
 (intermediate_act_fn): GELUActivation()
)
 (output): RobertaOutput(
 (dense): Linear(in_features=3072, out_features=768, bias=True)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
)
)
)
)
)

```

```

)
)
(classifier): RobertaClassificationHead(
 (dense): Linear(in_features=768, out_features=768, bias=True)
 (dropout): Dropout(p=0.1, inplace=False)
 (out_proj): Linear(in_features=768, out_features=28, bias=True)
)
)

```

```

Body + Head
RobertaConfig {
 "_name_or_path": "SamLowe/roberta-base-go_emotions",
 "architectures": [
 "RobertaForSequenceClassification"
],
 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "classifier_dropout": null,
 "eos_token_id": 2,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "id2label": {
 "0": "admiration",
 "1": "amusement",
 "2": "anger",
 "3": "annoyance",
 "4": "approval",
 "5": "caring",
 "6": "confusion",
 "7": "curiosity",
 "8": "desire",
 "9": "disappointment",
 "10": "disapproval",
 "11": "disgust",
 "12": "embarrassment",
 "13": "excitement",
 "14": "fear",
 "15": "gratitude",
 "16": "grief",
 "17": "joy",
 "18": "love",
 "19": "nervousness",
 "20": "optimism",
 "21": "pride",
 "22": "realization",
 "23": "relief",
 "24": "remorse",
 "25": "sadness",
 "26": "surprise",
 "27": "neutral"
 },
 "initializer_range": 0.02,
 "intermediate_size": 3072,
 "label2id": {
 "admiration": 0,
 "amusement": 1,
 "anger": 2,
 "annoyance": 3,
 "approval": 4,
 "caring": 5,
 "confusion": 6,
 "curiosity": 7,
 "desire": 8,
 }
}

```

```

 "disappointment": 9,
 "disapproval": 10,
 "disgust": 11,
 "embarrassment": 12,
 "excitement": 13,
 "fear": 14,
 "gratitude": 15,
 "grief": 16,
 "joy": 17,
 "love": 18,
 "nervousness": 19,
 "neutral": 27,
 "optimism": 20,
 "pride": 21,
 "realization": 22,
 "relief": 23,
 "remorse": 24,
 "sadness": 25,
 "surprise": 26
 },
 "layer_norm_eps": 1e-05,
 "max_position_embeddings": 514,
 "model_type": "roberta",
 "num_attention_heads": 12,
 "num_hidden_layers": 12,
 "pad_token_id": 1,
 "position_embedding_type": "absolute",
 "problem_type": "multi_label_classification",
 "torch_dtype": "float32",
 "transformers_version": "4.44.2",
 "type_vocab_size": 1,
 "use_cache": true,
 "vocab_size": 50265
}

```

### 3. Body + Head but only body

- Text classification를 위한 architecture에 Model Body만 불러옴
- 텍스트 분류를 위한 모델이지만, 분류 층을 제거하고 Body만 사용

```

from transformers import AutoModelForSequenceClassification
model_id = 'klue/roberta-base'
classification_model = AutoModelForSequenceClassification.from_pretrained(model_id)

```

#### ▼ 경고

Some weights of RobertaForSequenceClassification were not initialized from the model checkpoint  
You should probably TRAIN this model on a down-stream task to be able to use it for predictions

- `klue/roberta-base` 의 사전 학습된 파라미터는 불러왔으나 모델 허브에서 Classification head에 대한 파라미터를 찾을 수 없음
- 따라서 랜덤으로 초기화하였기에 그대로 사용하면 안되고 추가 학습 이후 사용  
→ 지금 그대로 사용하면 의미있는 분류 불가능

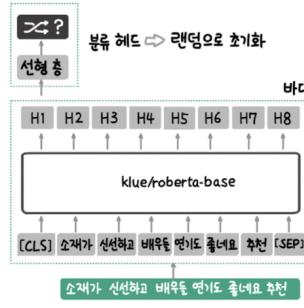


그림 3.11 분류 헤드가 랜덤으로 초기화된 모델

#### ▼ model architecture

```

RobertaModel(
 (embeddings): RobertaEmbeddings(
 (word_embeddings): Embedding(32000, 768, padding_idx=1)
 (position_embeddings): Embedding(514, 768, padding_idx=1)
 (token_type_embeddings): Embedding(1, 768)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
 (encoder): RobertaEncoder(
 (layer): ModuleList(
 (0-11): 12 x RobertaLayer(
 (attention): RobertaAttention(
 (self): RobertaSelfAttention(
 (query): Linear(in_features=768, out_features=768, bias=True)
 (key): Linear(in_features=768, out_features=768, bias=True)
 (value): Linear(in_features=768, out_features=768, bias=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
 (output): RobertaSelfOutput(
 (dense): Linear(in_features=768, out_features=768, bias=True)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
)
 (intermediate): RobertaIntermediate(
 (dense): Linear(in_features=768, out_features=3072, bias=True)
 (intermediate_act_fn): GELUActivation()
)
 (output): RobertaOutput(
 (dense): Linear(in_features=3072, out_features=768, bias=True)
 (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
 (dropout): Dropout(p=0.1, inplace=False)
)
)
)
)
 (pooler): RobertaPooler(
 (dense): Linear(in_features=768, out_features=768, bias=True)
 (activation): Tanh()
)
)
)

```

```

Body + Head but only body
RobertaConfig {
 "_name_or_path": "klue/roberta-base",
 "architectures": [
 "RobertaForMaskedLM"
],
}

```

```

 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "classifier_dropout": null,
 "eos_token_id": 2,
 "gradient_checkpointing": false,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "initializer_range": 0.02,
 "intermediate_size": 3072,
 "layer_norm_eps": 1e-05,
 "max_position_embeddings": 514,
 "model_type": "roberta",
 "num_attention_heads": 12,
 "num_hidden_layers": 12,
 "pad_token_id": 1,
 "position_embedding_type": "absolute",
 "tokenizer_class": "BertTokenizer",
 "transformers_version": "4.44.2",
 "type_vocab_size": 1,
 "use_cache": true,
 "vocab_size": 32000
}

```

▶ config 비교

```

only body
RobertaConfig {
 "_name_or_path": "klue/roberta-base",
 "architectures": [
 "RobertaForMaskedLM"
],
 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "classifier_dropout": null,
 "eos_token_id": 2,
 "gradient_checkpointing": false,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "initializer_range": 0.02,
 "intermediate_size": 3072,
 "layer_norm_eps": 1e-05,
 "max_position_embeddings": 514,
 "model_type": "roberta", # 모델 종류
 # 여러 설정 파라미터
 "num_attention_heads": 12,
 "num_hidden_layers": 12,
 "pad_token_id": 1,
 "position_embedding_type": "absolute",
 "tokenizer_class": "BertTokenizer", # 토크나이저
 "transformers_version": "4.44.2",
 "type_vocab_size": 1,
 "use_cache": true,
 "vocab_size": 32000 # 이후 사전 크기
}

```

```

Body + Head
RobertaConfig {
 "_name_or_path": "SamLowe/roberta-base-g
o_emotions",
 "architectures": [
 "RobertaForSequenceClassification"
],
 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "classifier_dropout": null,
 "eos_token_id": 2,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "id2label": {
 "0": "admiration",
 "1": "amusement",
 "2": "anger",
 "3": "annoyance",
 "4": "approval",
 "5": "caring",
 "6": "confusion",
 "7": "curiosity",
 "8": "desire",
 "9": "disappointment",
 "10": "disapproval",
 "11": "disgust",
 "12": "embarrassment",
 "13": "excitement",
 "14": "fear",
 "15": "gratitude",
 "16": "grief",
 "17": "joy",
 "18": "love",
 "19": "nervousness",
 "20": "optimism",
 "21": "pride",
 "22": "realization",
 "23": "relief",
 }
}

```

```

Body + Head but only body
RobertaConfig {
 "_name_or_path": "klue/roberta-base",
 "architectures": [
 "RobertaForMaskedLM"
],
 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "classifier_dropout": null,
 "eos_token_id": 2,
 "gradient_checkpointing": false,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "initializer_range": 0.02,
 "intermediate_size": 3072,
 "layer_norm_eps": 1e-05,
 "max_position_embeddings": 514,
 "model_type": "roberta",
 "num_attention_heads": 12,
 "num_hidden_layers": 12,
 "pad_token_id": 1,
 "position_embedding_type": "absolute",
 "tokenizer_class": "BertTokenizer",
 "transformers_version": "4.44.2",
 "type_vocab_size": 1,
 "use_cache": true,
 "vocab_size": 32000
}

```

### 3.3.2 토크나이저 활용하기

## 토크나이저 tokenizer

텍스트를 토큰 단위로 나누고 각 토큰을 대응하는 토큰 아이디로 변환

- 필요한 경우 특수 토큰을 추가하는 역할도 함
- 저장** | 토크나이저도 학습 데이터를 통해 어휘 사전을 구축하기 때문에 일반적으로 모델과 함께 저장
- 불러움** | 모델 불러올 때와 마찬가지로 model\_id를 통해 불러옴  
↳ 동일한 아이디로 동일 모델 불러오기

```
from transformers import AutoTokenizer
model_id = 'klue/roberta-base'
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

### ▼ 토크나이저의 종류나 설정에 대한 정보

```
tokenizer
BertTokenizerFast(name_or_path='klue/roberta-base', vocab_size=32000, model_max_length=512, is_fast=True, padding_side='right', truncation_side='right', special_tokens={'bos_token': '[CLS]', 'eos_token': '[SEP]', 'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True), added_tokens_decoder={0: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True), 1: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True), 2: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True), 3: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True), 4: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True)}
```

### ▼ 실제 어휘 사전 정보

```
tokenizer.get_vocab()
{'정보원': 12082,
'##을': 3042,
'다음': 11412,
'화상': 10948,
'1985': 10665,
'내정자': 11487,
'보여서': 18722,
'사단장': 29993,
'삿포로': 26651,
'GO': 22349,
'순경': 19420,
'스탠': 8411,
'외쳐': 22787,
'성현': 24074,
'[unused470]': 31970,
'##낫': 3166,
'##정보': 19861,
'분식': 3903,
'악학': 5915,
'##72': 23525,
'이주영': 26308,
'##르트': 6414,
'167': 17512,
'전방': 4016,
'심': 1332,
'##아들': 4383,
'南': 270,
'댐': 827,
'어려우': 18262,
'네이처': 18897,
'의료인': 29305,
'당근': 13469,
'스킬': 16920,
'그들': 9954,
'[unused499]': 31999,
'##습': 3178,
'천일염': 25355,
'독일군': 26177,
```

```
tokenized = tokenizer("토크나이저는 텍스트를 토큰 단위로 나눈다")
print(tokenized)
{'input_ids': [0, 9157, 7461, 2190, 2259, 8509, 2138, 1793, 2855, 5385, 2200, 20950, 2],
'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
#
print(tokenizer.convert_ids_to_tokens(tokenized['input_ids']))
['[CLS]', '토큰', '##나이', '##저', '##는', '텍스트', '##를', '토', '##큰', '단위', '##로', '나눈다', '[SEP]']
#
print(tokenizer.decode(tokenized['input_ids']))
[CLS] 토크나이저는 텍스트를 토큰 단위로 나눈다 [SEP]
```

```
print(tokenizer.decode(tokenized['input_ids'], skip_special_tokens=True))
토크나이저는 텍스트를 토큰 단위로 나눈다
```

#### ▼ output

```
{'input_ids': [0, 9157, 7461, 2190, 2259, 8509, 2138, 1793, 2855, 5385, 2200, 20950, 2],
'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}

[['CLS'], '토큰', '##나이', '##저', '##는', '텍스트', '##를', '토', '##큰', '단위', '##로', '나눈다', '[SEP]']

[CLS] 토크나이저는 텍스트를 토큰 단위로 나눈다 [SEP]

토크나이저는 텍스트를 토큰 단위로 나눈다
```

```
tokenizer("토크나이저는 텍스트를 토큰 단위로 나눈다")
```

tokenizer에 텍스트 입력

- input\_ids : 토큰 아이디 리스트
  - 토큰화했을 때 각 토큰이 토크나이저 사전의 몇 번째 항목인지 나타냄
  - ex. {0 : '[CLS]'}, {9157 : '토큰'}
- token\_type\_ids : 토큰이 속한 문장의 아이디 알려줌
  - 0이면 일반적으로 첫번째 문장임을 의미
  - 한번에 여러 문장 처리 가능
- attention\_mask : 토큰이 실제 텍스트인지 아니면 길이를 맞추기 위해 추가한 패딩인지 알려줌
  - 1이면 패딩이 아닌 실제 토큰을 의미

```
tokenizer.convert_ids_to_tokens(tokenized['input_ids'])
```

숫자 ID로 변환된 토큰들을 다시 원래의 텍스트 토큰으로 변환

- 각 숫자 ID는 토크나이저가 사용하는 어휘 사전에서 해당되는 단어 또는 서브워드에 대응

```
tokenizer.decode(tokenized['input_ids'])
```

- token\_id를 다시 텍스트로 돌리고 싶을 때 사용
- [CLS], [SEP] 와 같은 특수토큰이 추가됨  
→ 제거하고 싶으면 `skip_special_tokens=True` 설정 추가

#### + 여러 문장 입력하는 경우

```
tokenizer(['첫 번째 문장', '두 번째 문장'])
```

```
{'input_ids': [[0, 1656, 1141, 3135, 6265, 2],
[0, 864, 1141, 3135, 6265, 2]],
두 문장이 서로 별개로 처리되었기에 각각 문장으로 구분
'token_type_ids': [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]],
'attention_mask': [[1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]}
```

#### + 여러 문장 입력 + 한번에 입력

```
tokenizer([['첫 번째 문장', '두 번째 문장']]) # 한번 더 리스트로 감싸기
```

```
{'input_ids': [[0, 1656, 1141, 3135, 6265, 2, # 결과를 하나만 반환
864, 1141, 3135, 6265, 2]],
```

## 여러개의 입력(배치)일 때 토큰 아이디를 문자열로 복원하는 방법

```
두 문장 각각 입력
first_tokenized_result = tokenizer(['첫 번째 문장', '두 번째 문장'])['input_ids']
tokenizer.batch_decode(first_tokenized_result)
['[CLS] 첫 번째 문장 [SEP]', '[CLS] 두 번째 문장 [SEP]']

두 문장 한번에 입력
second_tokenized_result = tokenizer([['첫 번째 문장', '두 번째 문장']])['input_ids']
tokenizer.batch_decode(second_tokenized_result)
['[CLS] 첫 번째 문장 [SEP] 두 번째 문장 [SEP]']
```

## 모델 비교

```
bert_tokenizer = AutoTokenizer.from_pretrained('klue/bert-base')
bert_tokenizer([['첫 번째 문장', '두 번째 문장']])

roberta_tokenizer = AutoTokenizer.from_pretrained('klue/roberta-base')
roberta_tokenizer([['첫 번째 문장', '두 번째 문장']])

en_roberta_tokenizer = AutoTokenizer.from_pretrained('roberta-base')
en_roberta_tokenizer([['first sentence', 'second sentence']])
```

```
klue/bert-base
{'input_ids': [[2, 1656, 1141, 3135, 6265, 3, 864, 1141, 3135, 6265, 3]],
 # 학습할 때 2개의 문장이 이어지는지 맞추는 NSP 작업 활용 -> 문장 구분하는 토큰 타입 아이디 생성
 'token_type_ids': [[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]],
 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]}

klue roberta-base
{'input_ids': [[0, 1656, 1141, 3135, 6265, 2, 864, 1141, 3135, 6265, 2]],
 # 해당 모델의 경우 NSP 작업을 학습 과정에서 제거했기에 문장 토큰 구분이 필요 없음
 'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]}

roberta-base : 원본 영어에서는 아예 없음
{'input_ids': [[0, 9502, 3645, 2, 2, 10815, 3645, 2]],
 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1]]}
```

attention mask

```
tokenizer(['첫 번째 문장은 짧다.', '두 번째 문장은 첫 번째 문장 보다 더 길다.'],
 padding='longest')
```

```
{'input_ids': [[0, 1656, 1141, 3135, 6265, 2073, 1599, 2062, 18, 2, 1, 1, 1, 1, 1, 1, 1],
[0, 864, 1141, 3135, 6265, 2073, 1656, 1141, 3135, 6265, 3632, 831, 647, 2062, 18, 2]],
'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]}]
```

해당 토큰이 패딩 토크인지 실제 데이터인지에 대한 정보를 담고 있음

- 패딩은 모델에 입력하는 토큰 아이디의 길이를 맞추기 위해 추가하는 특수 토큰
  - `padding='longest'` : 긴 문장에 맞춰 패딩 토큰 추가

### 3.3.3 데이터셋 활용하기

```
from datasets import load_dataset
klue_mrc_dataset = load_dataset('klue', 'mrc')
klue_mrc_dataset_only_train = load_dataset('klue', 'mrc', split='train')
```

- klue 데이터셋에 접근해 서브셋인 mrc 다운

```
klue_mrc_dataset
DatasetsDict({
 train: Dataset({
 features: ['title', 'context', 'news_category', 'source', 'guid', 'is_impossible', 'question_type', 'question', 'answers'],
 num_rows: 17554
 })
 validation: Dataset({
 features: ['title', 'context', 'news_category', 'source', 'guid', 'is_impossible', 'question_type', 'question', 'answers'],
 num_rows: 5841
 })
})
```

예제 3.15 KLUE MRC 데이터셋 다운로드

- `split` : 유형 별로 택할 수 있음

```
from datasets import load_dataset
로컬의 데이터 파일을 활용
dataset = load_dataset("csv", data_files="my_file.csv")

파일 딕셔너리 활용
from datasets import Dataset
my_dict = {"a": [1, 2, 3]}
dataset = Dataset.from_dict(my_dict)

판다스 데이터프레임 활용
from datasets import Dataset
import pandas as pd
df = pd.DataFrame({"a": [1, 2, 3]})
dataset = Dataset.from_pandas(df)
```

## 3.4 모델 학습하기

### 한국어 뉴스 기사 제목 기반 기사의 카테고리 분류하는 텍스트 모델 학습

1. 데이터 준비
  - 데이터셋 준비
  - 모델과 토크나이저 불러옴
  - 모델 학습
- 2.1. Trainer API o
  - 간편하게 모델 학습 수행할 수 있도록 학습 과정을 추상화하기에 학습 간편
- 2.2. Trainer API X
  - 내부에서 어떤 과정을 거치는지 알 수 없다는 문제를 해결하기 위해 Trainer API 사용하지 않고 모델 학습
3. 학습한 모델 업로드
  - 학습을 마친 모델을 저장하거나 공유할 수 있도록 huggingface hub에 업로드

### 3.4.1 데이터 준비

| 데이터셋 준비 → 모델과 토크나이저 불러옴 → 모델 학습

## 1. 데이터 준비

```
from datasets import load_dataset
klue_tc_train = load_dataset('klue', 'ynat', split='train')
klue_tc_eval = load_dataset('klue', 'ynat', split='validation')
klue_tc_train
```

- subset 으로 'ynat'(연합뉴스 기사 제목과 기사가 속한 카테고리)

⇒ task. 연합 뉴스 기사의 제목을 바탕으로 카테고리 예측

```
klue_tc_train
Dataset({
 features: ['guid', 'title', 'label', 'url', 'date'],
 num_rows: 45678
})

klue_tc_train[0]
{'guid': 'ynat-v1_train_00000',
'title': '유튜브 내달 2일까지 크리에이터 지원 공간 운영',
'label': 3,
'url': 'https://news.naver.com/main/read.nhn?mode=LSD&mid=shm&sid1=105&sid2=227&oid=001&aid=0009588947',
'date': '2016.06.30 오후 10:36'}
```

그림 3.13 - 3.14

- 컬럼명 : 데이터 고유 ID, 뉴스 제목, 속한 category ID, 뉴스 링크, 뉴스 입력 시간

- label : 속한 카테고리 아이디

```
klue_tc_train.features['label'].names
['IT과학', '경제', '사회', '생활문화', '세계', '스포츠', '정치']

['IT과학', '경제', '사회', '생활문화', '세계', '스포츠', '정치']
```

## 2. 전처리

```
klue_tc_train = klue_tc_train.remove_columns(['guid', 'url', 'date'])
klue_tc_eval = klue_tc_eval.remove_columns(['guid', 'url', 'date'])
klue_tc_train
```

▼ 출력 결과 → 컬럼 수 줄었음

```
Dataset({
 features: ['title', 'label'],
 num_rows: 45678
})
```

### ▣ 카테고리 확인이 쉽도록 숫자형으로 변환

```
klue_tc_train.features['label']
ClassLabel(names=['IT과학', '경제', '사회', '생활문화', '세계', '스포츠', '정치'], id=None)

klue_tc_train.features['label'].int2str(1)
'경제'

klue_tc_label = klue_tc_train.features['label']

def make_str_label(batch):
 batch['label_str'] = klue_tc_label.int2str(batch['label'])
 return batch

klue_tc_train = klue_tc_train.map(make_str_label, batched=True, batch_size=1000)
```

```
klue_tc_train[0]
{'title': '유튜브 내달 2일까지 크리에이터 지원 공간 운영', 'label': 3, 'label_str': '생활문화'}
```

#### ▣ 데이터 일부 사용

```
train_dataset = klue_tc_train.train_test_split(test_size=10000, shuffle=True, seed=42)['test']

dataset = klue_tc_eval.train_test_split(test_size=1000, shuffle=True, seed=42)
test_dataset = dataset['test']
valid_dataset = dataset['train'].train_test_split(test_size=1000, shuffle=True, seed=42)['test']
```

### 3.4.2 Trainer API 사용해 학습하기

| 간편하게 모델 학습 수행할 수 있도록 학습 과정을 추상화한 Trainer API 제공

#### Trainer API

- huggingface는 학습에 필요한 학습 인자만으로 쉽게 사용할 수 있는 API 제공
- 데이터셋을 준비하고 학습 인자를 설정하는 데 필요한 몇 줄의 코드만으로 모델 학습 가능

#### 1. 준비

```
import torch
import numpy as np
from transformers import (
 Trainer,
 TrainingArguments,
 AutoModelForSequenceClassification,
 AutoTokenizer
)

데이터의 title 컬럼에 토큰화 적용
def tokenize_function(examples):
 return tokenizer(examples["title"], padding="max_length", truncation=True)

모델 불러옴
model_id = "klue/roberta-base"
model = AutoModelForSequenceClassification.from_pretrained(
 model_id,
 # 분류 헤드의 분류 클래스 수 지정
 num_labels=len(train_dataset.features['label'].names)
)
tokenizer = AutoTokenizer.from_pretrained(model_id)

train_dataset = train_dataset.map(tokenize_function, batched=True)
valid_dataset = valid_dataset.map(tokenize_function, batched=True)
test_dataset = test_dataset.map(tokenize_function, batched=True)
```

#### 2. 학습 인자와 평가 함수 정의

```
학습에 사용할 arguments 입력
training_args = TrainingArguments(
 output_dir=".results", # 결과 저장 폴더
 num_train_epochs=1, # 학습 에폭 수
 per_device_train_batch_size=8, # 배치크기
```

```

 per_device_eval_batch_size=8,
 evaluation_strategy="epoch", # 한 에폭 학습이 끝날 때마다 검증 데이터셋에 대한 평가 수행
 learning_rate=5e-5,
 push_to_hub=False
)

평가 지표 정의
def compute_metrics(eval_pred):
 logits, labels = eval_pred # 모델 예측 결과 저장
 predictions = np.argmax(logits, axis=-1) # 예측 결과 중 가장 큰 값인 클래스 저장
 return {"accuracy": (predictions == labels).mean()} # 정확도 반환

```

### 3. 학습 진행

```

trainer = Trainer(
 model=model,
 args=training_args,
 train_dataset=train_dataset,
 eval_dataset=valid_dataset,
 tokenizer=tokenizer,
 compute_metrics=compute_metrics,
)

trainer.train()

성능평가 수행
trainer.evaluate(test_dataset) # 정확도 0.84

```

## 3.4.3 트레이너 API 사용하지 않고 학습하기

| 내부 동작을 이해할 수 있도록 API를 사용하지 않고 모델 학습

### Trainer API

- huggingface는 학습에 필요한 학습 인자만으로 쉽게 사용할 수 있는 API 제공
  - 데이터셋을 준비하고 학습 인자를 설정하는 데 필요한 몇 줄의 코드만으로 모델 학습 가능
- ⇒ But. 내부 동작 파악하기 어려움

### 1. 학습을 위한 모델과 토크나이저 준비

```

import torch
from tqdm.auto import tqdm
from torch.utils.data import DataLoader
from transformers import AdamW

def tokenize_function(examples): # 제목(title) 컬럼에 대한 토큰화
 return tokenizer(examples["title"], padding="max_length", truncation=True)

모델과 토크나이저 불러오기
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model_id = "klue/roberta-base"
model = AutoModelForSequenceClassification.from_pretrained(
 model_id,
 num_labels=len(train_dataset.features['label'])
)
tokenizer = AutoTokenizer.from_pretrained(model_id)
model.to(device) # 직접 수행

```

## 2. 전처리

```
def make_dataloader(dataset, batch_size, shuffle=True):
 dataset = dataset.map(tokenize_function,
 batched=True).with_format("torch") # 데이터셋에 토큰화 적용
 dataset = dataset.rename_column("label", "labels") # 컬럼 이름 변경
 dataset = dataset.remove_columns(column_names=['title']) # 불필요한 컬럼 제거
 return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

데이터셋을 배치데이터로 만들어 데이터로더 만들기 (API 할 때와 달리 추가된 코드)
train_dataloader = make_dataloader(train_dataset, batch_size=8, shuffle=True)
valid_dataloader = make_dataloader(valid_dataset, batch_size=8, shuffle=False)
test_dataloader = make_dataloader(test_dataset, batch_size=8, shuffle=False)
```

## 3. 학습에 사용할 함수 정의

```
def train_epoch(model, data_loader, optimizer):
 model.train() # 학습 모드 변경
 total_loss = 0
 for batch in tqdm(data_loader):
 optimizer.zero_grad()
 input_ids = batch['input_ids'].to(device) # 모델에 입력할 토큰 아이디
 attention_mask = batch['attention_mask'].to(device) # 모델에 입력할 어텐션 마스크
 labels = batch['labels'].to(device) # 모델에 입력할 레이블
 outputs = model(input_ids, attention_mask=attention_mask, labels=labels) # 모델 계산
 loss = outputs.loss # 손실
 loss.backward() # 역전파
 optimizer.step() # 모델 업데이트
 total_loss += loss.item()
 avg_loss = total_loss / len(data_loader)
 return avg_loss
```

## 4. 평가에 사용할 함수 정의

```
def evaluate(model, data_loader):
 model.eval()
 total_loss = 0
 predictions = []
 true_labels = []
 with torch.no_grad():
 for batch in tqdm(data_loader):
 input_ids = batch['input_ids'].to(device)
 attention_mask = batch['attention_mask'].to(device)
 labels = batch['labels'].to(device)

 outputs = model(input_ids,
 attention_mask=attention_mask,
 labels=labels)

 # 로짓의 속성 가져와서 가장 큰 값으로 예측한 카테고리 정보 찾음
 logits = outputs.logits
 loss = outputs.loss
 total_loss += loss.item()
 preds = torch.argmax(logits, dim=-1)
 predictions.extend(preds.cpu().numpy())
 true_labels.extend(labels.cpu().numpy())
 avg_loss = total_loss / len(data_loader)
 # 실제 정답과 비교
```

```
accuracy = np.mean(np.array(predictions) == np.array(true_labels))
return avg_loss, accuracy
```

## 5. 학습 진행

```
num_epochs = 1
optimizer = AdamW(model.parameters(), lr=5e-5)

학습 루프
for epoch in range(num_epochs):
 print(f"Epoch {epoch+1}/{num_epochs}")
 train_loss = train_epoch(model, train_dataloader, optimizer)
 print(f"Training loss: {train_loss}")
 valid_loss, valid_accuracy = evaluate(model, valid_dataloader)
 print(f"Validation loss: {valid_loss}")
 print(f"Validation accuracy: {valid_accuracy}")

Testing
_, test_accuracy = evaluate(model, test_dataloader)
print(f"Test accuracy: {test_accuracy}") # 정확도 0.82
```

### 3.4.4 학습한 모델 업로드하기

| 학습 마친 모델을 저장하거나 공유할 수 있도록 허깅페이스 허브에 업로드

```
from huggingface_hub import login

login(token="본인의 허깅페이스 토큰 입력")
repo_id = f"본인의 아이디 입력/roberta-base-klue-ynat-classification"
Trainer를 사용한 경우
trainer.push_to_hub(repo_id)
직접 학습한 경우
model.push_to_hub(repo_id)
tokenizer.push_to_hub(repo_id)
```

- 업로드 방법은 trainer 사용 유무에 따라 달라짐

## 3.5 모델 추론하기

모델을 학습시킬 때 두가지 방법 존재

- ① 허깅페이스에서 제공하는 Trainer API를 활용하는 방법
- ② 직접 학습을 수행하는 방법

모델로 추론할 때도 두가지 방법 존재

- ① 모델을 활용하기 쉽도록 추상화한 pipeline 활용
- ② 직접 모델과 tokenizer를 불러와 활용하는 방법

### 3.5.1 파이프라인을 활용한 추론

#### ▣ 작업 종류 + 모델 + 설정

```
from transformers import pipeline

model_id = "본인의 아이디 입력/roberta-base-klue-ynat-classification"
```

```
model_pipeline = pipeline("text-classification", model=model_id)

model_pipeline(dataset["title"][:5])
```

▼ 결과

```
[{'label': '경제', 'score': 0.9940265417098999},
{'label': '사회', 'score': 0.9847791790962219},
{'label': 'IT과학', 'score': 0.9899107813835144},
{'label': '경제', 'score': 0.993854284286499},
{'label': '사회', 'score': 0.9936111569404602}]
```

파이프 라인에 추론하고자하는 텍스트 입력 시, 예측 확률이 가장 높은 레이블과 해당 확률 반환

### 3.5.2 직접 추론하기

```
import torch
from torch.nn.functional import softmax
from transformers import AutoModelForSequenceClassification, AutoTokenizer

class CustomPipeline:
 # 모델과 토크나이저 불러오기
 def __init__(self, model_id):
 self.model = AutoModelForSequenceClassification.from_pretrained(model_id)
 self.tokenizer = AutoTokenizer.from_pretrained(model_id)
 self.model.eval()

 # 인스턴스 호출할 때 내부적으로 사용
 def __call__(self, texts):
 # 토큰화
 tokenized = self.tokenizer(texts,
 return_tensors="pt",
 padding=True,
 truncation=True)

 # 모델 추론 수행
 with torch.no_grad():
 outputs = self.model(**tokenized)
 logits = outputs.logits

 # 가장 큰 예측 확률을 갖는 클래스 추출해 결과로 반환
 probabilities = softmax(logits, dim=-1)
 scores, labels = torch.max(probabilities, dim=-1)
 labels_str = [self.model.config.id2label[label_idx] for label_idx in labels.tolist()]

 return [{"label": label, "score": score.item()} for label, score in zip(labels_str, scores)]

custom_pipeline = CustomPipeline(model_id)
custom_pipeline(dataset['title'][:5])
```

## 3.6 정리

### 허깅페이스 트랜스포머 라이브러리

- 트랜스포머 모델을 활용할 수 있도록 통합된 인터페이스 제공
- 기본 제공

- Model hub
  - Tokenizer
  - Dataset
- 추가 제공 → 학습과 추론을 구현해보며 세부적인 작동 방식 확인
    - Trainer 클래스 : 간단하게 학습 수행
    - pipeline : 간단하게 추론 수행
  - ! 모델 크기가 작고 생성을 위한 모델이 아닌 분류 모델 선택

 다음 장부터 LLM 등장하며 어떤 과정을 거쳤는지 확인