

[SUM]Huggingface Tutorial | Transformers

- HuggingFace link ([Link](#))

Contents

- 1) [HuggingFace 01 | Transformer Models](#)
- 2) [HuggingFace 02 | Using Transformers](#)
- 3) [HuggingFace 03 | Fine-Tuning a Pretrained Model](#)
- 4) [HuggingFace 04 | Sharing Models and Tokenizers](#)

HuggingFace 01 | Transformer Models

[Introduction]

- Requires a good knowledge of Python
- Is better taken after an introductory deep learning course, such as [fast.ai's Practical Deep Learning for Coders](#) or one of the programs developed by [DeepLearning.AI](#)
- Does not expect prior [PyTorch](#) or [TensorFlow](#) knowledge, though some familiarity with either of those will help
- WHAT WE WILL LEARN
 - How to use the `pipeline()` function to solve NLP tasks such as text generation and classification
 - About the Transformer architecture
 - How to distinguish between encoder, decoder, and encoder-decoder architectures and use cases

[Natural Language Processing]

- What is NLP?
 - : NLP is a field of linguistics and machine learning focused on understanding everything related to human language.
 - The aim of NLP tasks is not only to understand single words individually, but to be able to understand the context of those words.
- list of common NLP tasks
 - **classifying whole sentences:** Getting the sentiment of a review, detecting if an email is spam, determining if a sentence is grammatically correct or whether two sentences are logically related or not
 - **Classifying each word in a sentence:** Identifying the grammatical components of a sentence (noun, verb, adjective), or the named entities (person, location, organization)
 - **Generating text content:** Completing a prompt with auto-generated text, filling in the blanks in a text with masked words
 - **Extracting an answer from a text:** Given a question and a context, extracting the answer to the question based on the information provided in the context
 - **Generating a new sentence from an input text:** Translating a text into another language, summarizing a text

[Transformers, what can they do?]

There are three main steps involved when you pass some text to a pipeline:

1. The text is preprocessed into a format the model can understand.
2. The preprocessed inputs are passed to the model.
3. The predictions of the model are post-processed, so you can make sense of them.

• Some of the currently available pipelines are:


- `feature-extraction` (get the vector representation of a text)
- `fill-mask`
- `ner` (named entity recognition)
- `question-answering`
- `sentiment-analysis`
- `summarization`
- `text-generation`
- `translation`
- `zero-shot-classification`

: We'll start by tackling a more challenging task

where we need to classify texts that haven't been labelled.

Let's have a look at a few of these!

Google Colaboratory

 <https://colab.research.google.com/github/huggingface/notebooks/blob/master/course/chapter1/section3.ipynb#scrollTo=vs0gl7TuMyhm>



[How do Transformers work?]

• A bit of Transformer history

: The Transformer architecture was introduced in June 2017. The focus of the original research was on translation tasks. This was followed by the introduction of several influential models, including

- **June 2018:** GPT, the first pretrained Transformer model, used for fine-tuning on various NLP tasks and obtained state-of-the-art results
- **October 2018:** BERT, another large pretrained model, this one designed to produce better summaries of sentences (more on this in the next chapter!)
- **February 2019:** GPT-2, an improved (and bigger) version of GPT that was not immediately publicly released due to ethical concerns
- **October 2019:** DistilBERT, a distilled version of BERT that is 60% faster, 40% lighter in memory, and still retains 97% of BERT's performance
- **October 2019:** BART and T5, two large pretrained models using the same architecture as the original Transformer model (the first to do so)
- **May 2020:** GPT-3, an even bigger version of GPT-2 that is able to perform well on a variety of tasks without the need for fine-tuning (called *zero-shot learning*)

: This list is far from comprehensive, and is just meant to highlight a few of the different kinds of Transformer models. Broadly, they can be grouped into three categories

- GPT-like (also called *auto-regressive* Transformer models)
- BERT-like (also called *auto-encoding* Transformer models)
- BART/T5-like (also called *sequence-to-sequence* Transformer models)

- 참고!!

- Transformers for Language Modeling

어텐션(Attention) 기법이 개발되고 난 뒤로, 뛰어난 성능을 가진 많은 언어 모델들이 생겨났습니다. 처음 어텐션이 적용된 분야는 기계번역(Machine Translation)이었습니다. 기존에 사용하던 seq2seq에 어텐션을 적용함으로써 성능을 높였죠.

<https://wikidocs.net/24996>

<https://wikidocs.net/22893>

어텐션만으로 구성된 트랜스포머(Transformer)도 등장했습니다.

<https://wikidocs.net/31379>

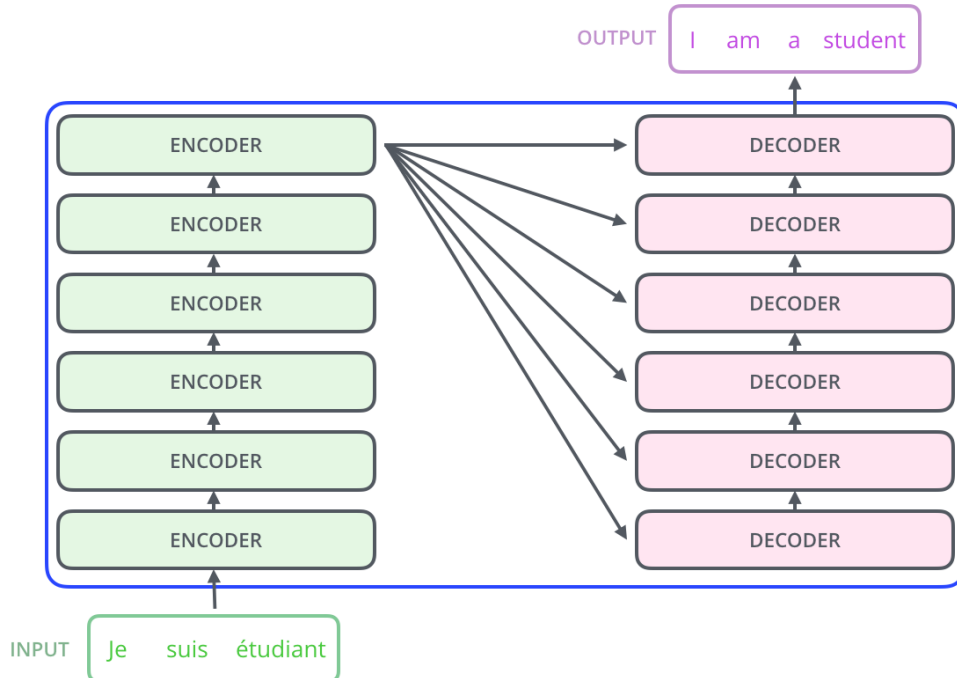
<https://aimb.tistory.com/182>

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2c23a11a-69d9-4aa5-b02a-5ff1c6ab0a92/1주차_발표transformers attention is all you need is love.pdf

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/defebe72-fd46-414c-b762-4ff5d151348f/2주차발표Attention Positional Encoding.pdf>

<https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2156304a-f5b4-46f2-9640-283efa01394c/2주차발표Attention is all you need - 코드 리뷰.pdf>

- 트랜스포머는 인코더와 디코더 스택으로 구성되어 있습니다.

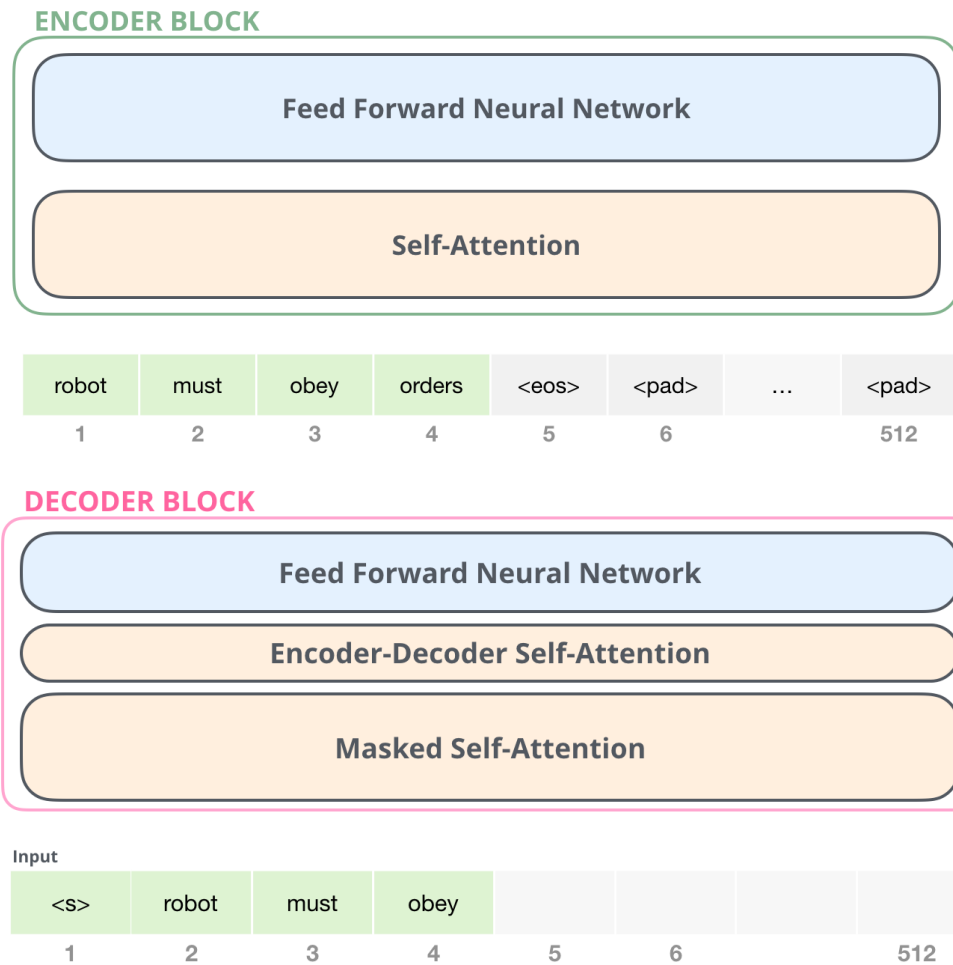


: 이에 그치지 않고, 트랜스포머의 인코더 혹은 디코더만 사용해야한다는 주장들이 등장했습니다. 그리고, 이렇게 등장한 것이 그 유명한 BERT와 GPT-2입니다. BERT는 트랜스포머의 인코더 스택만 사용한 모델이고, GPT-2는 디코더 스택만 사용한 모델이죠.

각각에 대한 간단한 내부 구조와 그에 따른 차이점을 살펴봅시다.

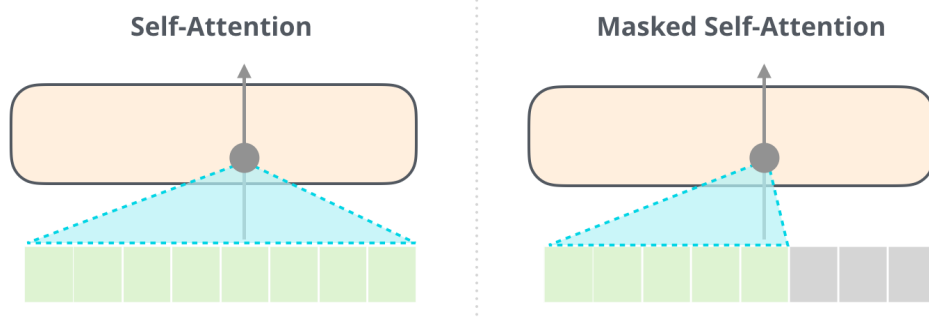
◦ BERT vs GPT-2

Encoder & Decoder cell



: 인코더셀과 디코더셀의 차이점 중 가장 주목해야하는 것은 제일 첫번째 레이어입니다.

- Encoder는 단순한 self-attention 레이어를 사용하는 반면, Decoder는 **Masked Self Attention**을 사용합니다.
- 이 말을 즉, BERT와는 다르게 gpt-2는 셀프 어텐션을 계산할 때 해당 스텝의 오른쪽에 있는 단어들은 고려하지 않는다는 것을 의미합니다.



◦ Auto-Regressive

: GPT-2는 자기 회귀 모델(auto-regressive model)입니다. 자기 회귀 모델이란 이전의 출력이 다음의 입력이 되는 모델을 의미합니다. 즉, RNN과 같은 것이죠! 반면, BERT는 자기 회귀 모델이 아닙니다.

- 이 둘의 장단점은 매우 확실합니다:

- 평범한 Self-Attention을 사용하며 자기 회귀 능력을 포기한 BERT는 다음 단어의 예측 능력은 덜 하지만, 맥락 정보를 충분히 고려할 수 있습니다.
- 반면, Masked Self-Attention을 사용하는 자기회귀 모델인 GPT-2는 다음 단어의 예측 능력은 뛰어나지만, 해당 단어의 이후에 있는 맥락 정보들을 이용할 수 없습니다.

• Architectures vs. checkpoints

: As we dive into Transformer models in this course, you'll see mentions of *architectures* and *checkpoints* as well as *models*. These terms all have slightly different meanings

- **Architecture**: This is the skeleton of the model – the definition of each layer and each operation that happens within the model.
- **Checkpoints**: These are the weights that will be loaded in a given architecture.
- **Model**: This is an umbrella term that isn't as precise as "architecture" or "checkpoint": it can mean both. This course will specify *architecture* or *checkpoint* when it matters to reduce ambiguity.

For example, BERT is an architecture while `bert-base-cased`, a set of weights trained by the Google team for the first release of BERT, is a checkpoint. However, one can say "the BERT model" and "the `bert-base-cased` model."

• Transformers are big models

- Apart from a few outliers (like DistilBERT), the general strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pretrained on.
- Unfortunately, training a model, especially a large one, requires a large amount of data. This becomes very costly in terms of time and compute resources. It even translates to environmental impact, as can be seen in the following graph.

◦ Transfer Learning(전이학습)

- *Pretraining(사전학습)*

: is the act of training a model from scratch: the weights are randomly initialized, and the training starts without any prior knowledge. This pretraining is usually done on very large amounts of data. Therefore, it requires a very large corpus of data, and training can take up to several weeks.

- *Fine-tuning*

: on the other hand, is the training done **after** a model has been pretrained. To perform fine-tuning, you first acquire a pretrained language model, then perform additional training with a dataset specific to your task. Wait – why not simply train directly for the final task? There are a couple of reasons:

1. The pretrained model was already trained on a dataset that has some similarities with the fine-tuning dataset. The fine-tuning process is thus able to take advantage of knowledge acquired by the initial model during pretraining (for instance, with NLP problems, the pretrained model will have some kind of statistical understanding of the language you are using for your task).
2. Since the pretrained model was already trained on lots of data, the fine-tuning requires way less data to get decent results.
3. For the same reason, the amount of time and resources needed to get good results are much lower.

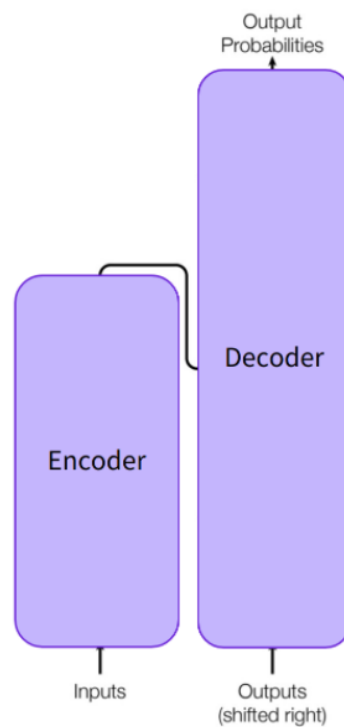
: Fine-tuning a model therefore has lower time, data, financial, and environmental costs. It is also quicker and easier to iterate over different fine-tuning schemes, as the training is less constraining than a full pretraining.

This process will also achieve better results than training from scratch (unless you have lots of data), which is why you should always try to leverage a pretrained model – one as close as possible to the task you have at hand – and fine-tune it.

- **General architecture**

The model is primarily composed of two blocks:

- **Encoder (left):** The encoder receives an input and builds a representation of it (its features). This means that the model is optimized to acquire understanding from the input.
- **Decoder (right):** The decoder uses the encoder's representation (features) along with other inputs to generate a target sequence. This means that the model is optimized for generating outputs.



[ABOUT MODELS]

- **Encoder-only models:** Good for tasks that require understanding of the input, such as sentence classification and named entity recognition.
- **Decoder-only models:** Good for generative tasks such as text generation.
- **Encoder-decoder models or sequence-to-sequence models:** Good for generative tasks that require an input, such as translation or summarization.

- **Encoder models**
- **Decoder models**
- **Sequence-to-sequence models**

[Summary]

Aa Model	≡ Examples	≡ Tasks
<u>Encoder</u>	ALBERT, BERT, DistilBERT, ELECTRA, RoBERTa	Sentence classification, named entity recognition, extractive question answering
<u>Decoder</u>	CTRL, GPT, GPT-2, Transformer XL	Text generation
<u>Encoder-decoder</u>	BART, T5, Marian, mBART	Summarization, translation, generative question answering

HuggingFace 02 | Using Transformers

[Introduction]

- 트랜스포머 모델은 사이즈가 매우 크고, 그 파라미터 또한 수백, 수천만 개에 달함
- “**Transformer Library**” ⇒ 트랜스포머 모델을 쉽게 로드하고, 학습하고, 저장할 수 있도록 API 제공
 - **Main Features**
 - 1) 쉬운 사용 : 단순히 코드 두 줄로 NLP 트랜스포머 모델 로드 및 사용 가능
 - 2) 유연성 : PyTorch `nn.Module` or TensorFlow `tf.keras.Model`
⇒ ML 프레임워크의 다른 모델과 동일하게 처리됨
 - 3) **단순함** : concept - “All in one file”
⇒ 모델의 forward pass는 하나의 파일로 처리되고 관리할 수 있음
 - 다른 ML 라이브러리와의 차별성 ⇒ 하나의 파일로 저장되는 특징
 - other ML libraries : 파일 간 공유되는 모듈 기반으로 모델 구축
 - Transformer : 각 모델에 고유한 계층 존재
 - ⇒ 모델의 접근성, 이해도 향상 & 다른 모델에 영향을 미치지 않음
- contents
 - 1) end-to-end example : 모델과 토큰라이저를 함께 사용해 1장에서 언급한 `pipeline()` fn 복제
 - End-to-End : 처음부터 끝까지, 보통 제3자로부터 아무 것도 얻지 않고 완전한 기능적 해결책을 전달하는 시스템 또는 서비스하는 과정을 설명 (input → output)
 - 2) 모델 API : 모델과 구성 클래스에 대해 학습하고, 모델을 로드하는 방법과 예측 값에 대한 수치 입력 처리 과정 소개
- Tokenizer API
 - : `pipeline()` fn의 메인 구성요소
 - 과정의 처음과 끝에 활용
 - 대화 속 문자를 신경망에 사용될 수치화 된 input으로 변경
 - 필요한 경우 다시 문자화
 - 다중 문장 처리 과정 ⇒ high-level `tokenizer()` fn 사용

[Behind the pipeline]

[PyTorch] or [TensorFlow]

[Example]

- Transformers Library import Pipeline

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
classifier([
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
```

```
]
)

# Result

[{'label': 'POSITIVE', 'score': 0.9598047137260437},
 {'label': 'NEGATIVE', 'score': 0.9994558095932007}]
```

- Three steps :



- **Pre-processing with a "Tokenizer"**

: 다른 뉴럴 네트워크와 마찬가지로 텍스트 값을 수치화하는 전처리 과정 필요

- **Tokenizer 사용**

- "tokens"로 분할 → 단어, 하위단어, 기호
- 각각의 token을 수치로 변환
- 모델에 필요한 추가적인 input 값 입력

- 모든 과정은 모델이 학습되었던 방식과 동일하게 진행되어야 함

- **Model Hub** 에서 해당 모델 정보를 다운

- **AutoTokenizer** 클래스와 그 **from_pretrained()** 메서드 사용

- 모델의 체크포인트를 사용해 해당 모델의 토크나이저와 관련된 데이터를 가져와 캐싱 (처음 한 번만 다운로드하면 사용가능)

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

# "sentiment-analysis"의 checkpoint 디폴트 값
# => "distilbert-base-uncased-finetuned-sst2-english"
# 해당 모델 정보 페이지에서 확인하기(Link)
```

- **Tokenizer** ~ sentence ⇒ dictionary 형태로 반환 (→ model 학습 가능)

- input 값의 list를 tensor 형태로 변환하는 작업 진행

- **Transformer models** ~ input 값은 tensor만 가능

- tensors = NumPy arrays
 - ⇒ a scalar (0D), a vector (1D), a matrix (2D), or have more dimensions
 - : 다른 ML framework의 tensor도 유사하게 동작함
 - : NumPy array 만큼 인스턴스화 용이

```
# "return_tensor" -> 반환할 tensor의 유형 지정

raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
]
inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="pt")
print(inputs)
```

- input 값 ⇒ 한 문장 혹은 문장 여러 개도 가능

- **return_tensor** 에서 특정 값을 지정하지 않을 경우 ⇒ default 값 : list of lists

```
# Result
```



```
{
  'input_ids': tensor([
    [ 101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166, 1012, 102],
    [ 101, 1045, 5223, 2023, 2061, 2172, 999, 102, 0, 0, 0, 0, 0, 0, 0, 0]
  ]),
  'attention_mask': tensor([
    [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]
  ])
}
```

- output

- 1) `input_ids`
: input에 포함된 문장의 수만큼 (여기 예제에서는 두 개) 각 문장의 토큰의 고유 식별자가 list형태로 포함
- 2) `attention_mask`

- **Going through the “Model”**

- Pretrained Model Download (이전 방법과 동일)

```
from transformers import AutoModel

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModel.from_pretrained(checkpoint)
```

- 기본적인 Transformer module 포함

- inputs
→ high-dimensional vector
(representing the **contextual understanding of that input by the Transformer model**)
- outputs (=hidden states or features)
→ use as another input of the another part of the model

- **A high-dimensional vector? (고차원벡터)**

- **Batch size (numb of inputs)**
: The number of sequences processed at a time (예제의 경우 2)
- **Sequence length (len of each input)**
: The length of the numerical representation of the sequence (예제의 경우 16)
- **Hidden size:** The vector dimension of each model input
⇒ “high dimensional” 이라고 하는 이유에 해당
: Generally, 768 is common for smaller models, and in larger models this can reach 3072 or more

```
outputs = model(**inputs)
print(outputs.last_hidden_state.shape)
```

```
# Result

torch.Size([2, 16, 768])
```

- format of outputs

- ⇒ `namedtuples` or `dictionaries`
- by the elements by attributes
 - by key : `outputs["last_hidden_state"]`
 - by index in list: `outputs[0]`

- **Model heads: Making sense out of numbers**

- input
 - = high-dimensional vector of hidden states
 - 하나 혹은 그 이상의 선형층으로 구성



- Model
 - 1) embeddings layer
 - : 토큰화된 input의 각 input ID를 벡터로 변환
 - 2) subsequent layers
 - : 최종 output을 생성하기 위해 embeddings layer에 입력된 벡터값 조작
- non-exhaustive list :
 - `Model` (retrieve the hidden states)
 - `ForCausalLM`
 - `ForMaskedLM`
 - `ForMultipleChoice`
 - `ForQuestionAnswering`
 - `ForSequenceClassification`
 - `ForTokenClassification`
 - and others ...

```
# Use "AutoModelForSequenceClassification"
# to be able to classify the sentences as positive or negative

from transformers import AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
outputs = model(**inputs)
```

```
print(outputs.logits.shape)
```

```
# Result (shape => 2 x 2)

torch.Size([2, 2])
```

- **"Post-processing" the Output**

```
print(outputs.logits)
```

```
# Result

tensor([[ -1.5607,  1.6123],
        [ 4.1692, -3.3464]], grad_fn=<AddmmBackward>)
```

- 확률이 출력되는 것이 아니라 모델의 마지막 계층에서 출력되는 비정규 점수인 *logit*
- 확률로 변환하기 위해서는 **SoftMax** layer를 거쳐야 함
 - activation fn(SoftMax) + loss fn(cross entropy) 의 결합으로 나타남

```
import torch

predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
print(predictions)
```

```
# Result => convert to probability score
tensor([[4.0195e-02, 9.5980e-01],
        [9.9946e-01, 5.4418e-04]], grad_fn=<SoftmaxBackward>)Copied
```

```
# 각 위치에 해당하는 label 가져오기
# "id2label" 사용

model.config.id2label
```

```
# Result

{0: 'NEGATIVE', 1: 'POSITIVE'}
```

- output
 - First sentence : **NEGATIVE**: 0.0402, **POSITIVE**: 0.9598
 - Second sentence : **NEGATIVE**: 0.9995, **POSITIVE**: 0.0005

[Models]

• Creating a Transformer

- BERT Model 초기화

: configuration object(구성개체) 로드하기

```
from transformers import BertConfig, BertModel

# Building the config
config = BertConfig()

# Building the model from the config
model = BertModel(config)
```

```
print(config)
```

```
# Result

BertConfig {
  [...]
  "hidden_size": 768,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  [...]
}
```

- `hidden_size` : `hidden_states` vector 의 크기 정의
- `num_hidden_layers` : Transformer model 의 layer 수 정의

Different loading methods

```
from transformers import BertConfig, BertModel

config = BertConfig()
model = BertModel(config)

# Model is randomly initialized!
```

- 모델 생성 후 학습을 먼저 진행
 - ⇒ 오랜 시간과 많은 데이터를 필요로 함 + 환경의 영향도 받을 수 밖에 없음
 - ⇒ Pretrained Model을 공유하고 재사용할 수 있어야 함

```
# "from_pretrained()" method를 통해 이미 학습된 모델 불러오기

from transformers import BertModel

model = BertModel.from_pretrained("bert-base-cased")
```

- `AutoModel` 클래스를 `BertModel` 로 대체 가능
⇒ checkpoint에 의존하지 않는 코드가 생성되기 때문에 이에 상관없이 동등하게 모델 실행 가능
- 위의 예제에서 `BertConfig` 대신 pretrained 된 `bert-base-cased` identifier 사용
 - more details ([Link](#))
- pretrained model ⇒ 처음부터 학습할 필요없이 바로 task 진행 가능
 - 가중치는 다운로드 후 캐시되어 지정된 폴더에 저장(`~/.cache/huggingface/transformers`)
- 모델을 로드하는 identifier
⇒ BERT 아키텍처와 호환되는 경우 허브에 있는 모든 모델의 identifier로 사용 가능
: 사용 가능한 BERT 체크포인트 목록 ([List](#))

Saving methods

```
# 모델 저장 => "save_pretrained()" method 사용

model.save_pretrained("directory_on_my_computer")
```

```
# save 2 files on device

ls directory_on_my_computer

config.json
pytorch_model.bin
```

- 1) `config.json`
: 모델 아키텍처 구축에 필요한 속성
 - 체크포인트의 출처
 - 메타데이터 (체크포인트를 마지막으로 저장할 때 사용하던 Transformers 버전 등)
 - 2) `pytorch_model.bin`
: known as state dictionary
: 모델의 모든 무게 (가중치) ⇒ 모델의 parameter
- Using a Transformer model for inference
 - Transformer Models can **only process numbers** that Tokenizer generates !

```
sequences = ["Hello!", "Cool.", "Nice!"]
```

```
# Result
# Tokenizer converts inputs to input IDs (vocab indices)
# list format => easy to convert tensors

encoded_sequences = [
    [101, 7592, 999, 102],
    [101, 4658, 1012, 102],
    [101, 3835, 999, 102],
]
```

```
# convert to tensors

import torch

model_inputs = torch.tensor(encoded_sequences)
```

Using the tensors as inputs to the model

```
# Model 호출

output = model(model_inputs)
```

- 모델의 다양한 input 값들 중 input IDs는 필수적 요소
 - input IDs ⇒ generated by the Tokenizer

[Tokenizers]

- Tokenizer ⇒ NLP 파이프라인의 핵심 구성요소
- purpose ⇒ 텍스트 input을 모델에서 처리할 수 있는 수치형 데이터(numerical data)로 변환

```
# input text

Jim Henson was a puppeteer
```

- To find the most meaningful representation
 - 가장 의미있는 표현 찾기
- To find the smallest representation
 - eliminate stopwords (불용어 제거)
- **Word-based**
 - input(raw text)을 단어 기반으로 분리 (split into words)
 - find a numerical representation for each of them



```
# python "split()" function으로
# 공백(whitespace) 기준 문장 구분

tokenized_text = "Jim Henson was a puppeteer".split()
print(tokenized_text)
```

```
# Result

['Jim', 'Henson', 'was', 'a', 'puppeteer']
```

- 각 단어는 0부터 해당 문장의 vocabs 만큼의 IDs를 할당받는데, 모델은 이 ID를 기반으로 단어 식별
- Disadvantages
 - punctuations(기호)에 따라 다르게 인식되는 단어 존재
 - ⇒ can get “large vocabularies” (어휘 수가 매우 커짐)
 - [example]
 - “dog”과 “dogs”는 다른 단어 개체로 식별됨 ⇒ 모델은 유사하다는 것을 사전에 알 수 없음
 - 어휘에 없는 단어들을 표현하기 위한 맞춤 토큰이 필요함
 - ⇒ 종종 “[UNK]” or “ ” (blank) 로 표기
- **Character-based**
 - input text를 단어가 아닌 특징에 따라(character-based) 분리 및 분류
 - Benefits :
 - vocabulary is much smaller
 - 단어보다 큰 단위로 분류하기 때문에 어휘목록의 수가 적어짐
 - out-of-vocabulary (unknown) tokens의 수가 훨씬 적어짐
 - 공백(spaces)과 구두점(punctuations)에 따른 문제점
 - 1) less meaningful : each character doesn't mean a lot (단순 단어가 가진 의미 없음)

[example]

Chinese have more information in each character than Latin languages

2) much more amount of tokens to be processed

- the numb of vocabs 는 줄어들지만 처리해야하는 token의 수가 증가

[example]

10개의 문자로 이루어진 token(word)은 word-based일 경우 1개의 token으로 변환,

하지만 character-based일 경우 10개의 tokens로 processing 과정 필요

• Subword tokenization

= "word-based" + "character-basd"

- Principles

- words should not be spit into smaller subwords
- rare words should be decomposed into meaningful subwords

[example]

"annoyingly"

⇒ "annoying" + "ly" 로 분리해서 복합어로 인식

- "Tokenization"

⇒ "Token" + "ization" 으로 분리 (형태소로 분석 ⇒ 실질적 의미 + 문법적 의미)

1) vocabulary의 수를 적게하면서도 명확한 의미로 분석 가능

2) 불명확한 토큰이 상대적으로 적음

⇒ useful for agglutinative languages (Turkish) - 복합어

And more!

- other techniques :

- Byte-level BPE, as used in GPT-2
- WordPiece, as used in BERT
- SentencePiece or Unigram, as used in several multilingual models

• Loading and saving

- loading and saving methods : `from_pretrained()` and `save_pretrained()`
 - tokenizer = architecture of the model
 - vocabulary = weights of the model

```
# using "BertTokenizer" class
# BERT와 동일한 checkpoint로 학습됨

from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-cased")
```

```
# "AutoTokenizer"
# => "AutoModel"과 유사하게 checkpoint 이름을 기반으로 사용 가능

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

```
tokenizer("Using a Transformer network is simple")
```

```
# Result

{'input_ids': [101, 7993, 170, 11303, 1200, 2443, 1110, 3014, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

```
# save model

tokenizer.save_pretrained("directory_on_my_computer")
```

• Encoding

= Translate text to numbers

◦ Process

1) Tokenization

: to split the text into words (or parts of words, punctuation symbols)

= tokens

- 모델이 사전 학습될 때와 동일한 규칙을 사용하여 토크나이저 진행

2) Conversion to input IDs

: convert tokens into numbers, so build a tensor for feeding to model

= vocabulary

- 모델이 사전 훈련될 때 사용했던 것과 같은 어휘를 사용해야 함 (= "from_pretrained()")

1) Tokenization

```
# using "tokenize()" method

from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

sequence = "Using a Transformer network is simple"
tokens = tokenizer.tokenize(sequence)

print(tokens)
```

```
# Result => list of strings (=tokens)

['Using', 'a', 'transform', '-er', 'network', 'is', 'simple']
```

◦ "transformer"

- "transform" + "-er" ⇒ subword tokenizer

2) From tokens to input IDs

```
# using "convert_tokens_to_ids()"

ids = tokenizer.convert_tokens_to_ids(tokens)

print(ids)
```

```
# Result => convert to appropriate tensor

[7993, 170, 11303, 1200, 2443, 1110, 3014]
```

• Decoding

= Get a string from vocabulary indices

```
# using "decode()" method

decoded_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110, 3014])
print(decoded_string)
```

```
# Result

'Using a Transformer network is simple'
```

- Process
 - 1) converts the indices back to tokens
 - 2) groups together the tokens (하나의 문장으로 생성)
- ⇒ useful for translation or summarization

[Handling Multiple Sequences]

- 실전에 적용
 - multiple sequences
 - multiple sequences of different lengths
 - other inputs except vocabulary indices
 - such a thing as too long as sequence

⇒ Using Transformers API

• Models expect a batch of inputs

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

sequence = "I've been waiting for a HuggingFace course my whole life."

tokens = tokenizer.tokenize(sequence)
ids = tokenizer.convert_tokens_to_ids(tokens)
input_ids = torch.tensor(ids)

# This line will fail.
model(input_ids)
```

```
# Result => error

IndexError: Dimension out of range (expected to be in range of [-1, 0], but got 1)
```

: Transformers models expect “multiple sentences” by default
 → just input “a sentence”
 ~ mismatch dimension

```
tokenized_inputs = tokenizer(sequence, return_tensors="pt")
print(tokenized_inputs["input_ids"])
```

```
# Result

tensor([[ 101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172,
          2607, 2026, 2878, 2166, 1012, 102]])
```

⇒ Identifying “input IDs” & “Logits”

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

sequence = "I've been waiting for a HuggingFace course my whole life."

tokens = tokenizer.tokenize(sequence)
```



```
ids = tokenizer.convert_tokens_to_ids(tokens)
```

```
input_ids = torch.tensor([ids])  
print("Input IDs:", input_ids)
```

```
output = model(input_ids)  
print("Logits:", output.logits)
```

```
# Result
```

```
Input IDs: [[ 1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662, 12172,  2607, 2026,  2878,  2166, 1012]]  
Logits: [[-2.7276,  2.8789]]
```

◦ **Batching**

: the act of **sending multiple sentences** through the model, all at once

- only one sentence ⇒ just build a batch with a single sequence

```
# batch of two identical sequences  
# convert "batched_ids" list into a tensor for model
```

```
batched_ids = [ids, ids]
```

◦ **Limit of Batching**

: 한 개 혹은 그 이상의 문장을 처리할 때 각각의 길이가 다를 수 있음 !

⇒ Need to be of rectangular shape, so it cannot convert the list of input IDs into a tensor directly

• **Padding the inputs**

```
# Each of them differs their lengths
```

```
batched_ids = [  
    [200, 200, 200],  
    [200, 200]  
]
```

⇒ **Using padding** for making our tensors have a rectangular shape

: “padding token”이라고 불리는 값을 추가함으로써 길이를 동일하게 함

```
# example
```

```
padding_id = 100
```

```
batched_ids = [  
    [200, 200, 200],  
    [200, 200, padding_id],  
]
```

```
# using "tokenizer.pad_token_id"
```

```
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
```

```
sequence1_ids = [[200, 200, 200]]  
sequence2_ids = [[200, 200]]  
batched_ids = [  
    [200, 200, 200],  
    [200, 200, tokenizer.pad_token_id],  
]
```

```
print(model(torch.tensor(sequence1_ids)).logits)  
print(model(torch.tensor(sequence2_ids)).logits)  
print(model(torch.tensor(batched_ids)).logits)
```

```
# Result
```

```
tensor([[ 1.5694, -1.3895]], grad_fn=<AddmmBackward>)  
tensor([[ 0.5803, -0.4125]], grad_fn=<AddmmBackward>)  
tensor([[ 1.5694, -1.3895,  
         [ 1.3373, -1.2163]], grad_fn=<AddmmBackward>)
```

: sequence2_ids의 logits 값과 batched_ids의 logits 값이 전혀 다름
([0.5803, -0.4125] or [1.3373, -1.2163])

⇒ **Because** ! key feature of Transformer models is attention layers that **contextualize** each token.

: 임의로 지정했던 "padding token"까지 고려하기 때문에 결과 값이 달라짐
따라서, "attention masks"를 사용해서 이를 무시하도록 선언해야 함

• Attention masks

= tensors with **the exact same shape as the input IDs tensor**

But, filled with 0s and 1s

- 1s ⇒ should be attended to
- 0s ⇒ should not be attended to
(= should be ignored by the attention layers of the model)

```
# using "attention masks"

batched_ids = [
    [200, 200, 200],
    [200, 200, tokenizer.pad_token_id],
]

attention_mask = [
    [1, 1, 1],
    [1, 1, 0],
]

outputs = model(torch.tensor(batched_ids), attention_mask=torch.tensor(attention_mask))
print(outputs.logits)
```

```
# Result => get the same logits

tensor([[ 1.5694, -1.3895],
        [ 0.5803, -0.4125]], grad_fn=<AddmmBackward>)
```

• Longer sequences

- **have limits** to the lengths of the sequences(= tensors)
- Most models handle **up to 512 or 1024 tokens**
- Two Solutions :
 - Use a model with a longer supported sequence length (better one)
⇒ Longformer or LED
 - Truncate sequences (cut off)
⇒ using `max_sequence_length` parameter

```
# using "max_sequence_length" for truncating

sequence = sequence[:max_sequence_length]
```

[Putting it all together]

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

sequence = "I've been waiting for a HuggingFace course my whole life."

model_inputs = tokenizer(sequence)
```

: `model_inputs` ⇒ contains everything that's necessary for a model to operate

- `DistilBERT` : input IDs with attention mask
- other models : also have those output by the `tokenizer` object

```
# tokenize a single sequence

sequence = "I've been waiting for a HuggingFace course my whole life."

model_inputs = tokenizer(sequence)
```

```
# tokenize multiple sequences at a time
# without changing API

sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]

model_inputs = tokenizer(sequences)
```

```
# padding inputs

# up to the maximum sequence length
model_inputs = tokenizer(sequences, padding="longest")

# up to the model max length that can handle (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, padding="max_length")

# up to the specified max length
model_inputs = tokenizer(sequences, padding="max_length", max_length=8)
```

```
# truncate sequences

sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]

# the sequences that are longer than the model max length that can handle (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, truncation=True)

# the sequences that are longer than the specified max length
model_inputs = tokenizer(sequences, max_length=8, truncation=True)
```

- can convert to specific framework tensors

```
# convert to specific framework tensors

sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]

# Returns PyTorch tensors
model_inputs = tokenizer(sequences, padding=True, return_tensors="pt")

# Returns TensorFlow tensors
model_inputs = tokenizer(sequences, padding=True, return_tensors="tf")

# Returns NumPy arrays
model_inputs = tokenizer(sequences, padding=True, return_tensors="np")
```

- **Special tokens**

```
sequence = "I've been waiting for a HuggingFace course my whole life."

model_inputs = tokenizer(sequence)
print(model_inputs["input_ids"])

tokens = tokenizer.tokenize(sequence)
ids = tokenizer.convert_tokens_to_ids(tokens)
print(ids)
```

```
# Result

[101, 1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166, 1012, 102]
[1045, 1005, 2310, 2042, 3403, 2005, 1037, 17662, 12172, 2607, 2026, 2878, 2166, 1012]
```

: 제일 앞과 제일 뒤에 token이 추가된 것을 확인할 수 있음

```
# Decode using "decode()" method

print(tokenizer.decode(model_inputs["input_ids"]))
print(tokenizer.decode(ids))
```

```
# Result

"[CLS] i've been waiting for a huggingface course my whole life. [SEP]"
"i've been waiting for a huggingface course my whole life."
```

⇒ [CLS]와 [SEP]가 추가됨

: 동일한 결과 값을 도출하기 위해 앞과 뒤에 특정 단어를 추가함

- 모델에 따라 특정 단어를 추가하거나 그렇지 않은 경우도 있음
- 모델에 따라 제일 앞에만 추가할 수도, 제일 마지막에만 추가할 수도 있음

• Wrapping up: From tokenizer to model

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]

tokens = tokenizer(sequences, padding=True, truncation=True, return_tensors="pt")
output = model(**tokens)
```

[Basic Usage Completed]

- Learned about :
 - basic building blocks of a **Transformer Model**
 - components of a **tokenization pipeline**
 - Transformer model in practice
 - conversion text to tensors **using Tokenizer**
 - Setting tokenizer and model together for getting predictions from text
 - **Limits of input IDs** and **attention masks**
 - A variety of **Tokenizer methods**

[End-of-Chapter Quiz]

1. What is the order of the language modeling pipeline?

⇒ The tokenizer handles text and returns IDs. The model handles these IDs and outputs a prediction. The tokenizer can then be used once again to convert these predictions back to some text.

(Tokenizer → 텍스트를 처리하고 ID 반환 & Model → 이러한 ID를 처리하고 예측 값 출력 & Tokenizer → 예측을 다시 텍스트로 변환)

2. How many dimensions does the tensor output by the base Transformer model have, and what are they?

⇒ 3 : The sequence length, the batch size, and the hidden size

3. Which of the following is an example of subword tokenization?

⇒ WordPiece / BPE / Unigram

4. What is a model head?

⇒ An additional component, usually made up of one or a few layers, to convert the transformer predictions to a task-specific output

(일반적으로 하나 혹은 그 이상의 계층으로 구성되며, Transformer의 예측 값을 "task-specific" 출력 값으로 변환하기 위한 추가 구성 요소)

5. What is an AutoModel?

⇒ An object that returns the correct architecture based on the checkpoint
(checkpoint를 기반으로 올바른 아키텍처를 반환)

6. What are the techniques to be aware of when batching sequences of different lengths together?

⇒ Truncating / Padding / Attention masking

7. What is the point of applying a SoftMax function to the logits output by a sequence classification model?

⇒ It applies a lower and upper bound so that they're understandable
(이해하기 쉽도록 하한선과 상한선을 적용)

⇒ The total sum of the output is then 1, resulting in a possible probabilistic interpretation
(출력의 총합이 1로, 확률론적 해석 가능)

8. What method is most of the tokenizer API centered around?

⇒ Calling the tokenizer object directly

(Tokenizer의 "__call__" 메서드는 거의 모든 것을 처리할 수 있는 매우 강력한 method. model에서 예측 값을 검색하는데 사용되기도 함)

9. What does the `result` variable contain in this code sample?

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
result = tokenizer.tokenize("Hello!")
```

⇒ A list of strings, each string being a token
(convert this to IDs, and send them to a model)

10. Is there something wrong with the following code?

```
from transformers import AutoTokenizer, AutoModel

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
model = AutoModel.from_pretrained("gpt2")

encoded = tokenizer("Hey!", return_tensors="pt")
result = model(**encoded)
```

⇒ The tokenizer and model should always be from the same checkpoint.

(coupling a model with a tokenizer that was trained with a different checkpoint is rarely a good idea.)

HuggingFace 03 | Fine-Tuning a Pretrained Model

[Introduction]

- Hub에서 대규모 데이터셋을 준비하는 방법
- 모델을 fine-tune 하기 위해 고급 Trainer API를 사용하는 방법
- custom training loop를 사용하는 방법
- Accelerate 라이브러리를 활용하여 모든 분산된 설정에서 해당 custom training loop를 쉽게 실행하는 방법

▶ trained checkpoint들을 Hugging Face Hub에 업로드하기 위해서는 [hugging face 계정](#)이 필요!

[Processing the data]

- INTRO

(이전 예제에 이어서) PyTorch의 한 배치에서 시퀀스 분류기를 훈련하는 방법

```
import torch
from transformers import AdamW, AutoTokenizer, AutoModelForSequenceClassification

# Same as before
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
sequences = [
    "I've been waiting for a HuggingFace course my whole life.",
    "This course is amazing!",
]
batch = tokenizer(sequences, padding=True, truncation=True, return_tensors="pt")

# This is new
batch["labels"] = torch.tensor([1, 1])

optimizer = AdamW(model.parameters())
loss = model(**batch).loss
loss.backward()
optimizer.step()
```

물론, 위 두 문장만 훈련시킨다면 좋은 결과를 낼 수 없기 때문에, 더 큰 dataset을 준비

⇒ **MRPC (Microsoft Research Paraphrase Corpus) dataset** 을 사용할 예정

- William B. Dolan and Chris Brockett 의 논문에서 소개됨
- 5801 쌍의 문장으로 이루어져 있음 & paraphrase인지 아닌지에 대한 라벨이 있음

• Loading a dataset from the Hub

Hub에는 모델 뿐만 아니라, 다양한 언어로 된 여러 dataset이 있음

- dataset 모음집 : <https://huggingface.co/datasets>
- 새 dataset을 로드하고 처리하는 법 (general docs) : <https://huggingface.co/docs/datasets/loading#from-the-huggingface-hub>

MRPC dataset

- GLUE 벤치마크를 구성하는 10개의 dataset 중 하나
- GLUE 벤치마크는 10개의 서로 다른 텍스트 분류 작업에 대한 ML 모델의 성능을 측정하는 데 사용되는 학문적 벤치마크
- MRPC dataset 다운로드 받는 법

```
from datasets import load_dataset

raw_datasets = load_dataset("glue", "mrpc")
raw_datasets
```

- 이 명령은 기본적으로 `~/.cache/huggingface/dataset` 에 dataset과 cache를 다운로드 함
- cache 폴더를 변경하고 싶다면, `HF_GOME` 환경변수를 세팅하면 됨 (Chap.2 참고)
- MRPC dataset 살펴보기

```
DatasetDict({
  train: Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 3668
  })
  validation: Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 408
  })
  test: Dataset({
    features: ['sentence1', 'sentence2', 'label', 'idx'],
    num_rows: 1725
  })
})
```

- DatasetDict 객체 : training, validation, test set 이 있고, 각각 (`sentence1`, `sentence2`, `label`, `idx`)열을 가지고 있음
- training set : 3668 쌍, validation set : 408 쌍, test set : 1725 쌍의 문장 이 들어있음

- indexing을 통해 각각의 문장 쌍에 접근하는 법

```
raw_train_dataset = raw_datasets["train"]
raw_train_dataset[0]
```

```
{'idx': 0,
 'label': 1,
 'sentence1': 'Amrozi accused his brother , whom he called " the witness " , of deliberately distorting his evidence .',
 'sentence2': 'Referring to him as only " the witness " , Amrozi accused his brother of deliberately distorting his evidence .'}
}
```

- 이미 label이 정수이기 때문에, 전처리를 더 하지 않아도 됨
- 어떤 정수가 어떤 label인지 알려면,

```
raw_train_dataset.features
```

```
{'sentence1': Value(dtype='string', id=None),
 'sentence2': Value(dtype='string', id=None),
 'label': ClassLabel(num_classes=2, names=['not_equivalent', 'equivalent'], names_file=None, id=None),
 'idx': Value(dtype='int32', id=None)}
```

- 레이블은 ClassLabel 유형이고 레이블 이름에 대한 정수 매핑은 이름 폴더에 담겨있음.
- 0은 not_equivalent에 해당하고 1은 equivalent에 해당함.



Try it out!

Look at element 15 of the training set and element 87 of the validation set. What are their labels?

[Preprocessing a dataset]

- 데이터 세트를 사전 처리하려면 텍스트를 모델이 이해할 수 있는 **숫자로 변환**해야 함.

⇒ 토큰라이저 사용

- 토큰라이저에 한 문장 또는 문장 목록을 제공할 수 있으므로 다음과 같이 각 쌍의 모든 첫 번째 문장과 두 번째 문장을 모두 직접 토큰화할 수 있음.

```
from transformers import AutoTokenizer

checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
tokenized_sentences_1 = tokenizer(raw_datasets["train"]["sentence1"])
tokenized_sentences_2 = tokenizer(raw_datasets["train"]["sentence2"])
```

- 그러나 두 개의 시퀀스를 모델에 전달하고 두 문장이 의역인지 아닌지에 대한 예측을 얻을 수는 없음. **두 시퀀스를 쌍으로 처리하고 적절한 전처리를 적용해야 함.**
- 토큰라이저는 한 쌍의 시퀀스를 가져와 BERT 모델이 예상하는 방식으로 준비할 수도 있음.

```
inputs = tokenizer("This is the first sentence.", "This is the second one.")
inputs
```

```
{
  'input_ids': [101, 2023, 2003, 1996, 2034, 6251, 1012, 102, 2023, 2003, 1996, 2117, 2028, 1012, 102],
  'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
  'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
}
```

- **token_type_ids** : 이 예시에서 입력의 어느 부분이 첫 번째 문장이고 어느 것이 두 번째 문장인지 모델에 알려줌

Try it out!

Take element 15 of the training set and tokenize the two sentences separately and as a pair. What's the difference between the two results?

- `input_ids` 내부의 ID를 다시 단어로 디코딩하면:

```
tokenizer.convert_ids_to_tokens(inputs["input_ids"])
```

```
['[CLS]', 'this', 'is', 'the', 'first', 'sentence', '.', '[SEP]', 'this', 'is', 'the', 'second', 'one', '.', '[SEP]']
```

- 모델은 두 개의 문장이 있을 때 입력이 `[CLS]` 문장1 `[SEP]` 문장2 `[SEP]` 형식이 될 것으로 예상함. 이를 `token_type_ids`와 정렬하면 아래와 같이 나옴.

```
['[CLS]', 'this', 'is', 'the', 'first', 'sentence', '.', '[SEP]', 'this', 'is', 'the', 'second', 'one', '.', '[SEP]']  
[ 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]
```

- `[CLS]` 문장1 `[SEP]`에 해당하는 입력 부분은 모두 토큰 유형 ID가 0이고 문장2 `[SEP]`에 해당하는 다른 부분은 모두 토큰 유형 ID가 1임
 - 다른 체크포인트를 선택하는 경우 : 토큰화된 입력에 반드시 `token_type_ids`가 있는 것은 아님 (예: DistilBERT 모델을 사용하는 경우 반환되지 않음).
 - 모델이 사전 훈련 중에 봐서, 모델이 무엇을 해야 하는지 알 수 있을 때만 반환됨.
- 여기서 BERT는 `token type IDs`로 사전 학습되며, 1장에서 설명한 '마스크된 언어 모델링' 목표 외에 '다음 문장 예측'이라는 추가 목표가 있음.
⇒ 이 작업의 목표는 문장 쌍 간의 관계를 모델링하는 것.
 - '다음 문장 예측'을 사용하면 모델에 문장 쌍(무작위로 마스킹된 토큰 포함)이 제공되고 두 번째 문장이 첫 번째 문장을 따르는지 여부를 예측하도록 요청됨.
 - 작업을 간단하지 않게 하기 위해 절반은 문장이 추출된 원본 문서에서 follow each other하고, 나머지 절반은 두 문장이 두 개의 다른 문서에서 나옴.
 - 일반적으로 토큰화된 input에 `token_type_ids`가 있는지 여부에 대해 걱정할 필요가 없음. 토크나이저와 모델에 대해 동일한 체크포인트를 사용하는 한 토크나이저가 알아서 해줌.
- 토크나이저가 한 쌍의 문장을 어떻게 처리하는지 봤으니, 이제 이를 전체 데이터 세트를 토큰화하는 데 사용할 수 있음.
 - 첫 번째 문장들의 목록과 두 번째 문장들의 목록을 줌으로써 문장 쌍을 토크나이저에 넣어줄 수 있음. (이는 Chap.2에서 봤던 padding & truncation 옵션들에서도 호환가능)

```
tokenized_dataset = tokenizer(  
    raw_datasets["train"]["sentence1"],  
    raw_datasets["train"]["sentence2"],  
    padding=True,  
    truncation=True,  
)
```

⇒ 잘 작동하지만 다음과 같은 단점이 있음.

1. 사전을 반환한다. (`keys`, `input_ids`, `Attention_mask`, `token_type_ids`, and `values that are list of lists`)
2. 토큰화하는 동안 전체 데이터 세트를 저장할 수 있는 RAM이 충분한 경우에만 작동한다.

- 데이터를 데이터 세트로 유지하기 위해 `Dataset.map()` 메서드를 사용함.

⇒ 이는 토큰화보다 더 많은 사전 처리가 필요한 경우 추가 유연성을 허용함. `map()` 메서드는 데이터 세트의 각 요소에 함수를 적용하여 작동하므로, `inputs`을 토큰화하는 함수를 정의해야 함.

```
def tokenize_function(example):  
    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)
```


- 이 함수는 사전을 사용하고(dataset의 item처럼) key(`input_ids`, `Attention_mask`, `token_type_ids`)가 있는 새 사전을 반환함.
- 이전에 본 것처럼 토큰라이저는 문장 쌍 목록에서 작동하기 때문에 예제 사전에 여러 샘플(each key as a list of sentences)이 포함된 경우에도 작동함.
- 이렇게 하면 `map()` 호출에서 `batched=True` 옵션을 사용할 수 있어 토큰화 속도가 크게 빨라짐.
- 토큰라이저는 `Tokenizers` 라이브러리에서 Rust로 작성된 토큰라이저에 의해 지원되며, 매우 빠를 수 있지만 한 번에 많은 입력을 제공하는 경우에만 가능함.

◦ 토큰화 함수에서 패딩 인수를 생략한 이유:

모든 샘플을 최대 길이로 채우는 것이 효율적이지 않기 때문임.

배치를 빌드할 때만 샘플을 채우는 것이 더 좋은데, 그 이유는 전체 dataset의 최대길이가 아닌 해당 배치의 최대 길이로 채우기만 하면 되기 때문임.

→ input의 길이가 매우 가변적일 때 많은 시간과 processing power을 절약할 수 있음.

◦ 한 번에 모든 데이터 세트에 tokenization function을 적용하는 방법

- `map` 호출에서 `batched=True` 를 사용하므로 함수가 각 요소에 개별적으로 적용되지 않고 데이터 세트의 여러 요소에 한 번에 적용됩니다. 이것은 더 빠른 전처리를 가능하게 함.

```
tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
tokenized_datasets
```

- `Datasets` 라이브러리가 이 처리를 적용하는 방법은 데이터 세트에 새 필드를 추가하는 것. `preprocessing function`에서 반환된 사전의 각 키에 대해 하나씩 적용됨

```
DatasetDict({
  train: Dataset({
    features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'sentence2', 'token_type_ids'],
    num_rows: 3668
  })
  validation: Dataset({
    features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'sentence2', 'token_type_ids'],
    num_rows: 408
  })
  test: Dataset({
    features: ['attention_mask', 'idx', 'input_ids', 'label', 'sentence1', 'sentence2', 'token_type_ids'],
    num_rows: 1725
  })
})
```

- `num_proc` 인수를 전달하여 `map()`으로 전처리 기능을 적용할 때 다중 처리를 사용할 수도 있음. `Tokenizers` 라이브러리는 샘플을 더 빠르게 토큰화하기 위해 이미 여러 스레드를 사용하지만, 이 라이브러리에서 지원하는 빠른 토큰라이저를 사용하지 않는 경우 이 방법을 통해 사전 처리 속도가 빨라질 수 있음.

◦ 우리의 tokenize_function은 `input_ids`, `Attention_mask` 및 `token_type_ids` 키가 있는 사전을 반환하므로 이 세 필드는 데이터 세트의 모든 분할에 추가됨.

- 전처리 함수가 `map()`을 적용한 데이터 세트의 기존 키에 대한 새 값을 반환한 경우 기존 필드를 변경할 수도 있음.

◦ 마지막으로, 요소를 함께 배치할 때 가장 긴 요소의 길이로 모든 예제를 채워야 함. (동적 패딩)

• Dynamic padding

- `collate function` : 배치 내부에 샘플을 모으는 역할을 하는 함수
 - `DataLoader`를 빌드할 때 전달할 수 있는 인수
 - 기본값 : 샘플을 PyTorch 텐서로 변환하고 연결하는 함수(요소가 목록, 튜플 또는 사전인 경우 재귀적으로)
 - ** 입력값이 모두 같은 크기가 아니기 때문에 이 예시에서는 불가능.
 - ** 우리는 각 배치에 필요한 만큼만 패딩을 적용하고 패딩이 많은 너무 긴 입력을 방지하기 위해 의도적으로 패딩을 연기했음.
 - 참고로, 이렇게 하면 훈련 속도가 상당히 빨라지지만 TPU에서 훈련하는 경우 문제가 발생할 수 있음. (TPU는 extra padding이 필요한 경우에도 fixed shape을 선호)

- 실제로 동적패딩을 수행하려면 함께 일괄 처리(batch)하려는 데이터 세트의 items에 정확한 양의 패딩을 적용하는 `collate function` 을 정의해야 함.
- 😊 Transformers 라이브러리는 `DataCollatorWithPadding` 을 통해 이러한 기능을 제공
→ 인스턴스화할 때 토큰라이저가 필요하며(사용할 패딩 토큰과 모델에서 패딩이 입력의 왼쪽 또는 오른쪽에 있을 것으로 예상하는지 여부를 알기 위해) 필요한 모든 작업을 수행할 것.

```
from transformers import DataCollatorWithPadding

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

◦ 테스트하기 위해 함께 배치하려는 훈련 세트에서 몇 가지 샘플을 가져오기.

- 여기에서 `idx`, `sentence1`, `sentence2` 열은 필요하지 않고
- 문자열을 포함하고 있으니, 제거하고(문자열로 텐서를 만들 수 없음)
- 일괄 처리의 각 항목 길이를 살펴봅시다.

```
samples = tokenized_datasets["train"][:8]
samples = {k: v for k, v in samples.items() if k not in ["idx", "sentence1", "sentence2"]}
[ len(x) for x in samples["input_ids"] ]
```

```
[50, 59, 47, 67, 59, 50, 62, 32]
```

→ 32에서 67까지 다양한 길이의 샘플들을 얻는데, 동적 패딩은 이 배치의 샘플이 모두 배치 내부의 최대 길이인 67의 길이로 채워져야 함을 의미함.

** 동적 패딩이 없으면 모든 샘플은 전체 데이터 세트의 최대 길이 또는 모델이 허용할 수 있는 최대 길이로 채워져야 함.

- `data_collator` 가 동적으로 배치를 적절하게 패딩하는지 다시 확인!

```
batch = data_collator(samples)
{k: v.shape for k, v in batch.items() }
```

```
{ 'attention_mask': torch.Size([8, 67]),
  'input_ids': torch.Size([8, 67]),
  'token_type_ids': torch.Size([8, 67]),
  'labels': torch.Size([8]) }
```

이제 원시 텍스트에서 모델이 처리할 수 있는 배치로 이동했으므로 미세 조정할 준비가 되었음!

Try it out!

Replicate the preprocessing on the GLUE SST-2 dataset. It's a little bit different since it's composed of single sentences instead of pairs, but the rest of what we did should look the same. For a harder challenge, try to write a preprocessing function that works on any of the GLUE tasks.

[Fine-tuning a model with the Trainer API]

😊 Transformers는 `Trainer` 클래스를 제공하여 데이터 세트에서 제공하는 사전 훈련된 모델을 미세 조정할 수 있도록 도와줌.

마지막 섹션에서 모든 데이터 전처리 작업을 완료하면 `Trainer`를 정의하는 데 몇 단계만 남는데, 가장 어려운 부분은 `Trainer.train()`을 실행할 환경을 준비하는 것. (CPU에서 매우 느리게 실행되기 때문. GPU를 설정하지 않은 경우 Google Colab에서 무료 GPU 또는 TPU에 액세스 가능)

📌 아래 코드 예제는 이전 섹션의 예제를 이미 실행했다고 가정하고, 필요한 사항을 간략하게 요약함

```
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("glue", "mrpc")
```

```
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

def tokenize_function(example):
    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

• Training

◦ Trainer를 정의하기 전에,

1 첫 번째 단계 : Trainer가 훈련 및 평가에 사용할 모든 하이퍼파라미터를 포함하는 `TrainingArguments` 클래스를 정의

- 제공해야 하는 인수는 1) 학습된 모델이 저장될 디렉토리나 2) 그 과정에 따른 체크포인트
- 나머지는 모두 기본값을 그대로 둘 수 있으며 이는 기본 미세 조정에 적합함.

```
from transformers import TrainingArguments

training_args = TrainingArguments("test-trainer")
```

💡 If you want to automatically upload your model to the Hub during training, pass along `push_to_hub=True` in the `TrainingArguments`. We will learn more about this in [Chapter 4](#)

2 두 번째 단계 : 모델 정의

- 이전 장에서와 같이 두 개의 레이블이 있는 `AutoModelForSequenceClassification` 클래스를 사용

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

- 실행해보면, 2장과 달리 이 사전 훈련된 모델을 인스턴스화한 후 경고를 받는 것을 알 수 있는데, 이는 BERT가 문장 쌍 분류에 대해 사전 학습되지 않았기 때문에 사전 학습된 모델의 헤드를 버리고 시퀀스 분류에 적합한 새로운 헤드를 대신 추가했기 때문
- 경고는 일부 가중치가 사용되지 않았으며(떨어진 사전 훈련 헤드에 해당하는 가중치) 다른 가중치가 무작위로 초기화되었음을(새 헤드에 대한 가중치) 나타냄.

◦ 이제 모델이 있으므로, 지금까지 구성된 모든 객체를 전달하여 **Trainer**를 정의 가능 (`model`, `training_args`, `the training and validation datasets`, `data_collator`, `tokenizer`)

```
from transformers import Trainer

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
)
```

- 토크나이저를 전달할 때 Trainer가 사용하는 기본 `data_collator`는 이전에 정의된 `DataCollatorWithPadding`이 되므로 이 호출에서 `data_collator=data_collator` 줄을 건너뛸 수 있음.

◦ 데이터 세트의 모델을 미세 조정하려면 **Trainer**의 `train()` 메서드를 호출하기만 하면 됨.

```
trainer.train()
```

- 이렇게 하면 미세 조정이 시작되고(GPU에서 몇 분 정도 소요됨) 500단계마다 훈련 손실이 보고되지만, 모델이 얼마나 잘(또는 나쁘게) 수행되고 있는지는 알려주지 않음.
- 그 이유 :
 1. 우리는 `evaluation_strategy`를 "`steps`"(모든 `eval_steps` 평가) 또는 "`epoch`"(각 `epoch`의 끝에서 평가)로 설정하여 훈련 중에 평가하도록 Trainer에게 지시하지 않았음.
 2. 우리는 평가하는 동안 메트릭을 계산하기 위해 Trainer에 `compute_metrics()` 함수를 제공하지 않았음. (그렇지 않으면 평가에서 손실을 출력했을 것인데, 이는 매우 직관적인 수치가 아님.)

• Evaluation

◦ 유용한 `compute_metrics()` 함수를 빌드하고 다음에 훈련할 때 사용하는 방법

- 이 함수는 `EvalPrediction` 객체(예측 필드와 `label_ids` 필드가 있는 명명된 튜플)를 가져와야 하며 사전 매핑 문자열을 `float`로 반환함(문자열은 반환된 메트릭의 이름이고 `float`는 해당 값). 모델에서 일부 예측을 얻으려면 `Trainer.predict()` 명령을 사용할 수 있음.

```
predictions = trainer.predict(tokenized_datasets["validation"])
print(predictions.predictions.shape, predictions.label_ids.shape)
```

```
(408, 2) (408,)
```

- `predict()` 메서드의 output은 3개의 필드(`predictions`, `label_ids`, and `metrics`)가 있는 또 다른 명명된 튜플.
- 메트릭 필드에는 전달된 데이터 세트의 손실과 일부 시간 메트릭(전체 및 평균적으로 예측하는 데 걸린 시간)만 포함됨.
- `compute_metrics()` 함수를 완료하고 Trainer에 전달하면 해당 필드에는 `compute_metrics()`에서 반환한 메트릭도 포함됨.
- 예측은 모양이 408 x 2인 2차원 배열(408은 우리가 사용한 데이터세트의 요소 수).
- 이는 우리가 `predict()`에 전달한 데이터 세트의 각 요소에 대한 `logits`임. (이전 장에서 보았듯이 모든 Transformer 모델은 `logits`을 반환함).
- 레이블과 비교할 수 있는 예측으로 변환하려면 두 번째 축에 최대값이 있는 인덱스를 가져와야 함.

```
import numpy as np

preds = np.argmax(predictions.predictions, axis=-1)
```

◦ 이제 해당 `pred`를 레이블과 비교 가능함.

- `compute_metric()` 함수를 빌드하기 위해 🤗 Datasets 라이브러리의 메트릭을 사용
- 이번에는 `load_metric()` 함수를 사용하여 데이터 세트를 로드하는 것처럼 쉽게 MRPC 데이터 세트와 관련된 메트릭을 로드할 수 있음.
- 반환된 객체에는 메트릭 계산을 수행하는 데 사용할 수 있는 `compute()` 메서드가 있음.

```
from datasets import load_metric

metric = load_metric("glue", "mrpc")
metric.compute(predictions=preds, references=predictions.label_ids)
```

```
{'accuracy': 0.8578431372549019, 'f1': 0.8996539792387542}
```

*** 모델 헤드를 무작위로 초기화하면 달성한 메트릭이 변경될 수 있으므로 정확한 결과는 다를 수 있음.

- 우리 모델이 검증 세트에서 85.78%의 정확도와 89.97의 F1 점수를 가지고 있음을 알 수 있음.
- 정확도와 F1 score는 GLUE 벤치마크에 대한 MRPC 데이터 세트의 결과를 평가하는 데 사용되는 두 가지 메트릭
- BERT 논문의 표에서는 기본 모델에 대해 F1 점수가 88.9임.

◦ 모든 것을 함께 wrapping하면 `compute_metrics()` 함수를 얻을 수 있음.

```
def compute_metrics(eval_preds):
    metric = load_metric("glue", "mrpc")
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)
```

- 다음은 각 에포크가 끝날 때 메트릭을 보고하기 위해 실제로 사용되는 것을 보기 위해 이 `compute_metrics()` 함수를 사용하여 새 `Trainer`를 정의하는 방법

```
training_args = TrainingArguments("test-trainer", evaluation_strategy="epoch")
model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)
```

- `evaluation_strategy`이 `"epoch"`로 설정된 새 `TrainingArguments`와 새 모델을 생성 (그렇지 않으면 이미 훈련한 모델의 훈련을 계속할 것)
- 새로운 훈련 실행을 시작하기 위해 다음을 실행

```
trainer.train()
```

- 이번에는 훈련 손실 외에 각 에포크가 끝날 때 유효성 검사 손실 및 메트릭을 보고함.
- 도달하는 정확한 정확도/F1 점수는 모델의 무작위 헤드 초기화로 인해 우리가 찾은 것과 약간 다를 수 있지만 거의 근접한 결과가 나와야 함.
- `Trainer`는 여러 GPU 또는 TPU에서 즉시 사용할 수 있으며 혼합 정밀도 훈련과 같은 다양한 옵션을 제공(`training arguments`에서 `fp16 = True` 사용).

이것으로 `Trainer` API를 사용한 미세 조정 소개를 마칩니다. 가장 일반적인 NLP 작업에 대해 이 작업을 수행하는 예는 7장에서 제공되지만 지금은 순수한 PyTorch에서 동일한 작업을 수행하는 방법을 살펴보겠습니다.

Try it out!

Fine-tune a model on the GLUE SST-2 dataset, using the data processing you did in section 2.

[A full training]

`Trainer` 클래스를 사용하지 않고 마지막 섹션에서 했던 것과 동일한 결과를 얻는 방법을 살펴보기

- 섹션 2에서 데이터 처리를 완료했다고 가정

📌 필요한 모든 것을 다루는 짧은 요약

```
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("glue", "mrpc")
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

def tokenize_function(example):
    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

• Prepare for training

실제로 훈련 루프를 작성하기 전 준비

- **tokenized_datasets에 약간의 후처리 적용** (Trainer가 자동으로 수행한 몇 가지 작업을 처리하기 위함)

1. 모델이 기대하지 않는 값에 해당하는 열을 제거(예: sentence1 과 sentence2 열)
2. column label을 labels로 변경(모델은 인수가 named labels일 것으로 예상하기 때문)
3. lists 대신 PyTorch 텐서를 반환하도록 데이터셋의 형식을 설정

** 우리의 tokenized_datasets에는 이러한 각 단계에 대해 하나의 메서드가 있음

```
tokenized_datasets = tokenized_datasets.remove_columns(["sentence1", "sentence2", "idx"])
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
tokenized_datasets.set_format("torch")
tokenized_datasets["train"].column_names
```

그런 다음 결과에 모델이 허용하는 열만 있는지 확인

```
["attention_mask", "input_ids", "labels", "token_type_ids"]
```

◦ 객체 정의

- **1 batch마다 일괄 처리를 반복하는 데 사용할 데이터 로더 정의**

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(
    tokenized_datasets["train"], shuffle=True, batch_size=8, collate_fn=data_collator
)
eval_dataloader = DataLoader(
    tokenized_datasets["validation"], batch_size=8, collate_fn=data_collator
)
```

- 데이터 처리에 오류가 없는지 빠르게 확인하기 위해 다음과 같이 배치를 검사 가능

```
for batch in train_dataloader:
    break
{k: v.shape for k, v in batch.items()}
Copied
{'attention_mask': torch.Size([8, 65]),
 'input_ids': torch.Size([8, 65]),
 'labels': torch.Size([8]),
 'token_type_ids': torch.Size([8, 65])}
```

** 훈련 데이터 로더에 대해 shuffle=True를 설정하고 배치 내에서 최대 길이로 패딩하기 때문에 실제 shape은 약간 다를 수 있음.

- **2 이제 데이터 전처리가 완전히 끝났으므로 모델로 돌아가서, 이전 섹션에서 했던 것처럼 정확히 인스턴스화**

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
```

- 훈련 중에 모든 것이 원활하게 진행되도록 하기 위해 배치를 이 모델에 전달

```
outputs = model(**batch)
print(outputs.loss, outputs.logits.shape)
```

```
tensor(0.5441, grad_fn=<NllLossBackward>) torch.Size([8, 2])
```

- 모든 😊 Transformers 모델은 레이블이 제공되면 손실을 반환하고 로짓도 얻음(배치의 각 입력에 대해 2개이므로 텐서는 8 x 2 크기임).

◦ 최적화 프로그램 설정

- Trainer가 직접 수행한 작업을 복제하려고 하므로, 동일한 기본값을 사용
- Trainer에서 사용하는 옵티마이저는 AdamW로 Adam과 동일하지만 가중치 감소 정규화에 약간의 변형이 있음(Ilya Loshchilov 및 Frank Hutter의 "[Decoupled Weight Decay Regularization](#)" 참조).

```
from transformers import AdamW

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

◦ 학습률 스케줄러 설정

- 기본적으로 사용되는 학습률 스케줄러는 최대값(5e-5)에서 0으로의 linear decay
- 이를 올바르게 정의하려면 수행할 학습 단계 수, 즉 Epoch 수를 알아야 하는데, 우리는 훈련 배치 수(훈련 데이터 로더의 길이)를 곱하여 실행하려고 함
- Trainer는 기본적으로 3개의 Epoch를 사용하므로 이를 따름 :

```
from transformers import get_scheduler

num_epochs = 3
num_training_steps = num_epochs * len(train_dataloader)
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)
print(num_training_steps)
```

```
1377
```

• The training loop

- GPU에 액세스할 수 있는 경우 GPU를 사용
(CPU에서는 훈련에 몇 분이 아닌 몇 시간이 걸릴 수 있음).
- 이를 위해 모델과 배치를 배치할 장치를 정의

```
import torch

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)
device
```

```
device(type='cuda')
```

- 훈련이 언제 끝날지 알기 위해 tqdm 라이브러리를 사용하여 훈련 단계 수에 진행률 표시줄을 추가

```
from tqdm.auto import tqdm

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

보고를 요청하지 않았으므로 이 training loop는 모델 성능에 대해 아무 것도 알려주지 않음.
→ 평가 루프를 추가해야 함

• The evaluation loop

이전에 했던 것처럼 😊 Datasets 라이브러리에서 제공하는 메트릭을 사용.

- `add_batch()` 메서드로 예측 루프를 진행하면서 메트릭이 실제로 배치를 누적할 수 있음.
- 모든 배치를 누적하면 `metric.compute()` 로 최종 결과를 얻을 수 있음.

👉 평가 루프에서 이 모든 것을 구현하는 방법 :

```
from datasets import load_metric

metric = load_metric("glue", "mrpc")
model.eval()
for batch in eval_dataloader:
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model(**batch)

    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)
    metric.add_batch(predictions=predictions, references=batch["labels"])

metric.compute()
```

```
{'accuracy': 0.8431372549019608, 'f1': 0.8907849829351535}
```

모델 헤드 초기화 및 데이터 셔플링의 임의성으로 인해 결과가 약간 다르겠지만, 어느 정도는 비슷한 결과가 나와야 함.

🖋 Try it out!

Modify the previous training loop to fine-tune your model on the SST-2 dataset.

• Supercharge your training loop with 😊 Accelerate

앞에서 정의한 훈련 루프는 단일 CPU 또는 GPU에서 제대로 작동하지만, 😊 Accelerate 라이브러리를 사용하여 몇 가지만 조정하면 여러 GPU 또는 TPU에서 분산 교육을 활성화할 수 있음.

- 훈련 및 검증 데이터 로더 생성부터 수동 훈련 루프의 모습은 다음과 같음

```
from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
optimizer = AdamW(model.parameters(), lr=3e-5)

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)

num_epochs = 3
num_training_steps = num_epochs * len(train_dataloader)
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps,
)

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```


- 변경 사항은 다음과 같음.

```
+ from accelerate import Accelerator
+ from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

+ accelerator = Accelerator()

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
optimizer = AdamW(model.parameters(), lr=3e-5)

- device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
- model.to(device)

+ train_dataloader, eval_dataloader, model, optimizer = accelerator.prepare(
+   train_dataloader, eval_dataloader, model, optimizer
+ )

num_epochs = 3
num_training_steps = num_epochs * len(train_dataloader)
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps
)

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        - batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        - loss.backward()
        + accelerator.backward(loss)

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

- 추가할 첫 번째 줄은 import line
- 두 번째 줄은 환경을 살펴보고 적절한 분산 설정을 초기화하는 Accelerator 개체를 인스턴스화함.
- 🤗 Accelerate가 device placement를 대신 처리하므로 장치에 모델을 표시하는 라인을 제거가능(or 원하는 경우 device 대신 accelerator.device를 사용하도록 변경).
- 그런 다음 `dataloaders`, `model` 및 `optimizer`를 `accelerator.prepare()`로 보내는 라인에서 대부분의 작업이 수행됨. 이는 분산된 학습이 의도대로 작동하도록 만들기 위해 이러한 객체들을 적절한 컨테이너 안에 모음
- 나머지 변경 사항은
 - device에 batch를 넣는 줄을 제거하는 것 (이것을 유지하려면 가속기를 사용하도록 변경할 수 있음)
 - `loss.backward()`를 `accelerator.backward(loss)`로 교체하는 것

⚠ In order to benefit from the speed-up offered by Cloud TPUs, we recommend padding your samples to a fixed length with the `padding="max_length"` and `max_length` arguments of the tokenizer.

📌 완전한 훈련 루프 🤗 Accelerate:

```
from accelerate import Accelerator
from transformers import AdamW, AutoModelForSequenceClassification, get_scheduler

accelerator = Accelerator()

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
optimizer = AdamW(model.parameters(), lr=3e-5)

train_dl, eval_dl, model, optimizer = accelerator.prepare(
    train_dataloader, eval_dataloader, model, optimizer
)

num_epochs = 3
num_training_steps = num_epochs * len(train_dl)
lr_scheduler = get_scheduler(
    "linear",
```

```

optimizer=optimizer,
num_warmup_steps=0,
num_training_steps=num_training_steps,
)

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dl:
        outputs = model(**batch)
        loss = outputs.loss
        accelerator.backward(loss)

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)

```

- 이것을 train.py 스크립트에 넣으면 모든 종류의 분산 설정에서 스크립트를 실행할 수 있음.
- 분산 설정에서 사용해 보려면 다음 명령을 실행

```
accelerate config
```

몇 가지 질문에 답하고 이 명령이 사용하는 구성 파일에 답을 덤프하라는 메시지가 표시됨

```
accelerate launch train.py
```

그러면 분산 교육이 시작됨.

- 노트북에서 이것을 시도하려면(예를 들어 Colab의 TPU로 테스트하기 위해) training_function()에 코드를 붙여넣고 다음을 사용하여 마지막 셀을 실행

```

from accelerate import notebook_launcher

notebook_launcher(training_function)

```

You can find more examples in the [🤗 Accelerate repo](#)

[Fine-tuning, Check!]

- 허브의 데이터세트에 대해 알아볼.
- dynamic padding 및 collators 사용을 포함하여 데이터 세트를 로드하고 사전 처리하는 방법을 배움
- 모델에 대한 자체 미세 조정 및 평가 구현
- 낮은 수준의 training loop를 구현
- multiple GPUs나 TPUs에서 작동하도록 training loop을 쉽게 조정하기 위해 🤗 Accelerate 사용

[End-of-chapter quiz]

1. The `emotion` dataset contains Twitter messages labeled with emotions. Search for it in the [Hub](#), and read the dataset card. Which of these is not one of its basic emotions?
 - ☐ Joy Love Confusion SurpriseSubmit
2. Search for the `ar_sarcasm` dataset in the [Hub](#). Which task does it support?
 - ☐ Sentiment classification Machine translation Named entity recognition Question answeringSubmit
3. How does the BERT model expect a pair of sentences to be processed?
 - ☐ Tokens_of_sentence_1 [SEP] Tokens_of_sentence_2 [CLS] Tokens_of_sentence_1 Tokens_of_sentence_2 [CLS] Tokens_of_sentence_1 [SEP] Tokens_of_sentence_2 [SEP] [CLS] Tokens_of_sentence_1 [SEP] Tokens_of_sentence_2Submit
4. What are the benefits of the `Dataset.map()` method?
 - ☐ The results of the function are cached, so it won't take any time if we re-execute the code. It can apply multiprocessing to go faster than applying the function on each element of

the dataset. It does not load the whole dataset into memory, saving the results as soon as one element is processed.Submit

5. What does dynamic padding mean?

- ☐ It's when you pad the inputs for each batch to the maximum length in the whole dataset. It's when you pad your inputs when the batch is created, to the maximum length of the sentences inside that batch. It's when you pad your inputs so that each sentence has the same number of tokens as the previous one in the dataset.Submit

6. What is the purpose of a collate function?

- ☐ It ensures all the sequences in the dataset have the same length. It puts together all the samples in a batch. It preprocesses the whole dataset. It truncates the sequences in the dataset.Submit

7. What happens when you instantiate one of the `AutoModelForXxx` classes with a pretrained language model (such as `bert-base-uncased`) that corresponds to a different task than the one for which it was trained?

- ☐ Nothing, but you get a warning. The head of the pretrained model is discarded and a new head suitable for the task is inserted instead. The head of the pretrained model is discarded. Nothing, since the model can still be fine-tuned for the different task.Submit

8. What's the purpose of `TrainingArguments` ?

- ☐ It contains all the hyperparameters used for training and evaluation with the `Trainer`. It specifies the size of the model. It just contains the hyperparameters used for evaluation. It just contains the hyperparameters used for training.Submit

9. Why should you use the 🚀 Accelerate library?

- ☐ It provides access to faster models. It provides a high-level API so I don't have to implement my own training loop. It makes our training loops work on distributed strategies It provides more optimization functions.

HuggingFace 04 | Sharing Models and Tokenizers

[The Hugging Face Hub]

- Hugging Face Hub :
 - SOTA 모델과 데이터셋을 사용하고 발견할 수 있는 웹 사이트로, 다양한 모델들을 제공하고 있다.
 - 다양한 모델들은 Git repository로 host되어 있으며 자유롭게 모델을 공유하고 사용할 수 있다.

[Using pretrained models]

: Model hub를 통해서 간단하게 적합한 모델을 찾고, 몇 줄의 코드 만으로 그 모델을 사용할 수 있도록 한다. 이 챕터에서는 그런 모델을 어떻게 실제로 사용하는지 배우고 다시 community에 contribute back하는 법에 대해 살펴본다.

```
from transformers import pipeline

camembert_fill_mask = pipeline("fill-mask", model="camembert-base")
results = camembert_fill_mask("Le camembert est <mask> :)")
```

- `camembert-base` 모델을 사용하기 위해서는 `pipeline()` 에 파라미터로 넣어주기만 하면 된다. `camembert-base` 모델은 text-classification에는 적합하지 않은 모델이기 때문에 만약 그렇게 넣어서 실행하면 말도 안 되는 결과가 나올 것! 그러니까 hugging face 사이트에서 task에 적합한 모델을 찾아서 사용하자

```
from transformers import CamembertTokenizer, CamembertForMaskedLM

tokenizer = CamembertTokenizer.from_pretrained("camembert-base")
model = CamembertForMaskedLM.from_pretrained("camembert-base")
```

이렇게 해서 model architecture을 이용해서 checkpoint를 인스턴스화 시킬 수 있는데, 이렇게 하는 것 보다는 Auto*classes를 이용해서 하는 걸 추천한다.

- Auto*classes 란

: 많은 경우의 모델 구조는 그 from_pretrained()에 제공하는 pretrained model의 이름이나 path로 추측할 수 있음. AutoClasses는 pretrained weights/config/vocabulary에 given된 name/path 를 통해 자동적으로 관련 있는 모델을 검색하는 역할을 한다.

```
model = AutoModel.from_pretrained("bert-base-cased")
```

이렇게 AutoConfig/AutoModel/AutoTokenizer을 instantiating하면 자동적으로 그 architecture의 클래스를 생성한다. 위의 코드는 BertModel의 instance인 모델을 생성한다.

각 task와 각 backend에 하나의 AutoModel에 대해서 한 개의 클래스가 생성된다.

```
from transformers import AutoTokenizer, AutoModelForMaskedLM

tokenizer = AutoTokenizer.from_pretrained("camembert-base")
model = AutoModelForMaskedLM.from_pretrained("camembert-base")
```

이렇게 하면 checkpoint를 바꾸는게 더 simple하고, checkpoints를 camemBERT 구조를 통해서만 읽어들 수 있는 한계에서 벗어날 수 있음



Pretrained model을 사용할 때는 꼭 그 pretrained model이 어떤 데이터셋으로 학습되었고, 어떤 한계를 가지고 있고 어떤 편향을 가지고 있는지를 확인하고 사용하자!

[Sharing pretrained models]

: pretrained model을 공유하여 community에 기여하는 것은, 그 모델이 specific한 데이터셋으로 학습되었더라도 다른 사람의 시간과 노력을 아끼게 해 줄 수 있다!

새로운 모델 repository를 만드는 방법은 세 가지가 있다.

- push_to_hub API
- huggingface_hub Python library
- Using the web interface

일단 repository를 생성하고 나면 git과 git-lfs를 통해서 file들을 upload할 수 있다.

[Building a model card]

: training과 evaluation과정을 문서화함으로써 다른 사람들이 모델을 이해하는데 도움을 줄 수 있다. preprocessing과 postprocessing에 사용된 데이터에 대한 충분한 정보를 제공하는 것은 그 모델의 한계, 편향, 그리고 어떤 맥락에서 이 모델이 유용할 지 유용하지 않을 지를 이해할 수 있게 도와준다.

- Model card는 README.md를 통해서 생성할 수 있고 아래와 같은 구성을 따른다.
 - Model의 very brief, high-level overview of 이 모델이 무엇인가
 - Model description : architecture, version, original implementation, author, general information about the model. Copyright, general information about training procedures, parameters, and important disclaimers
 - Intended uses & limitations : describe the use cases the model is intended for, including the languages, fields, and domains where it can be applied. This section of the model card can also document areas that are known to be out of scope for the model, or where it is likely to perform
 - How to use : examples of how to use the model. Showcase usage of the pipeline() function.
 - Limitations and bias
 - Training data : indicate which dataset(s) the model was trained on
 - Training procedure : describe all the relevant aspects of training that are useful from a reproducibility perspective. preprocessing and postprocessing that were done on the data, as

well as details such as the number of epochs the model was trained for, the batch size, the learning rate, and so on.

- **Variable and metrics** : describe the metrics you use for evaluation, and the different factors you are measuring.
- **Evaluation results** : how well the model performs on the evaluation dataset