zero-shot learning: 如GPT3这个模型做下游任务可以不需要微调

第1课 关于模型

Transformer架构的模型可以有以下分类

- 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)

根据transformer部件的择取 来分类

- **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.

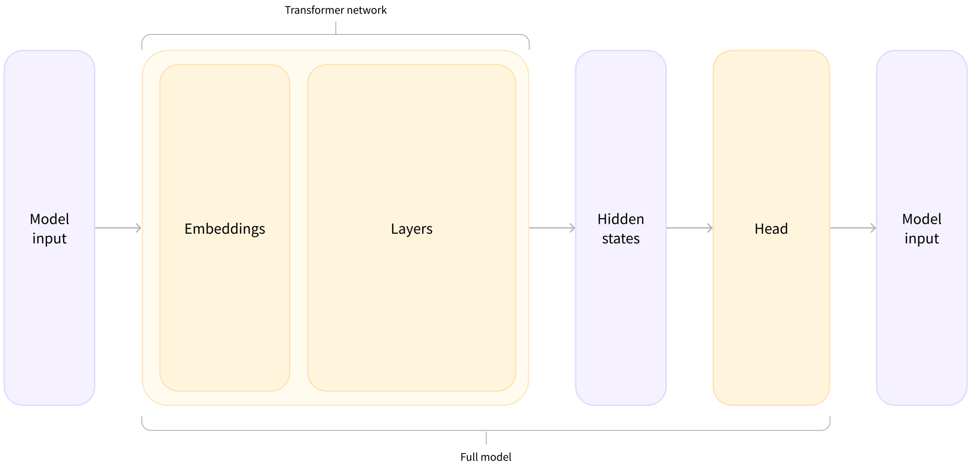
Sequence-to-sequence models are best suited for tasks revolving around generating new sentences depending on a given input, such as summarization, translation, or generative question answering.

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

第2课 使用模型

隐藏层的下一次层也被称为head

- **Batch size**: The number of sequences processed at a time.
- **Sequence length**: The length of the numerical representation of the sequence (16 in our example).
- **Hidden size**: The vector dimension of each model input.



1. model的输出类似于字典，可以用outputs["last_hidden_state"]，也可以直接用outputs[0]去查看想要看的内容。
2. 用 AutoModel实例化的model，其输出默认是关于last_hidden_state的；而AutoModelForSequenceClassification的输出是关于logit（logit还需要输入到softmax中才能更好地输出人能看懂的label）。后者的输出是前者的输出再加上head层（The model heads take the high-dimensional vector of hidden states as input and project them onto a different dimension.)

分词器

word-based 分词器的缺点:

1. 英文中单词在50万以上, 意味着要建立一个50万维的词向量
2. dog和dogs可能会由两个毫不相近的id来表示
3. 对于训练集没有出现过的单词 用UNK表示, 但是这个单词可能有非常重量级的含义, 却无法捕捉了(意思是如果能把尽量少的token变成UNK就好了)

Character-based分词器的特点:

1. 解决了单词表过大的问题, 以及不会有UNK的问题
2. 不过英文中一个字母的含义可能是无法体现的
3. 一个句子的长度会因为 拆成字母 长度会变成10倍左右

Subword tokenization

能更好的弥补上述两者的缺点, 同时兼顾到两者的优点

具体会有多个变种, 会在后续有更详细的介绍

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

第3课 关于fine-tune

#GLUE介绍

The GLUE Benchmark is a group of nine classification tasks on sentences or pairs of sentences which are:

- CoLA (Corpus of Linguistic Acceptability) 判断一个句子是否语法正确

- MNLI (Multi-Genre Natural Language Inference) 判断两个句子之间的三种关系, 第一句是假设, 第二句是一个推论, 关系是 合理, 矛盾, 毫无关联
- MRPC (Microsoft Research Paraphrase Corpus) 判断两个句子是否想表达同一个意思
- QNLI (Question-answering Natural Language Inference) 判断第一个句子的答案是否在第二个句子当中
- QQP (Quora Question Pairs2) 判断两个问句是否问的同一个意思
- RTE (Recognizing Textual Entailment) Determine if a sentence entails a given hypothesis or not.
- SST-2 (Stanford Sentiment Treebank) Determine if the sentence has a positive or negative sentiment.
- STS-B (Semantic Textual Similarity Benchmark) Determine the similarity of two sentences with a score from 1 to 5.
- WNLI (Winograd Natural Language Inference) Determine if a sentence with an anonymous pronoun and a sentence with this pronoun replaced are entailed or not. (This dataset is built from the Winograd Schema Challenge dataset.)

Task	Metric	Result	Training time
CoLA	Matthews corr	56.53	3:17
SST-2	Accuracy	92.32	26:06
MRPC	F1/Accuracy	88.85/84.07	2:21
STS-B	Pearson/Spearman corr.	88.64/88.48	2:13
QQP	Accuracy/F1	90.71/87.49	2:22:26
MNLI	Matched acc./Mismatched acc.	83.91/84.10	2:35:23
QNLI	Accuracy	90.66	40:57
RTE	Accuracy	65.70	57
WNLI	Accuracy	56.34	24

##

数据集:

MRPC(Microsoft Research Paraphrase Corpus) , 有5801个句子对, 同时有一个label, 显示这个句子对中的两句话是不是表达着同一个语义

它也是GLUE度量标准的10个数据集之一

用

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

即可导入

```
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
    })
})
```

导入数据

tokenizer是可以直接对数据集进行操作

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

这种操作比较方便, 但是有缺点

1. 你需要要有足够大的内存
2. it has the disadvantage of returning a dictionary (with our keys, `input_ids`, `attention_mask`, and `token_type_ids`, and values that are lists of lists).

另一种方式: (可保持着数据集本身的结构)

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

def tokenize_function(example): # 传进来的必须是字典
    return tokenizer(example["sentence1"], example["sentence2"],
                     truncation=True)
```

The `map()` method works by applying a function on each element of the dataset.

batched这个参数能使token化的过程大幅加速(but only if we give it lots of inputs at once.意思就是他会一次性对多个实例进行token化)

```
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
    })
})
```

这里变多了, 是因为我们自己写的map函数的tokenize_function参数返回的是 `input_ids`, `attention_mask`, and `token_type_ids`. 所以他会加上这些. 同理, 我们可以自定义返回值,来添加我们想要的字段.

动态填充(dynamic padding)

负责将样本放在一个批次中的函数称为**collate function**。这个是在构建 DataLoader 时可以传递的参数，默认值是将样本转换为 PyTorch 张量并将它们连接起来的函数（如果元素是列表、元组或字典，则递归）。

在这个例子中,句子长度不一,所以需要padding

当然也有提供这个api

```
from transformers import DataCollatorWithPadding
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

# 这里拿出训练集的前八个句子看看效果
samples = tokenized_datasets["train"][:8]
# samples是一个字典 我现在只拿出我感兴趣的三个字段
samples = {k: v for k, v in samples.items() if k not in ["idx", "sentence1",
"sentence2"]}
# 输出这八个句子的长度 会发现不一致
[len(x) for x in samples["input_ids"]]
# padding之后就一致
batch = data_collator(samples)
{k: v.shape for k, v in batch.items()}
```

动态的意思就是 会自动扫描batch中最大的句子的长度

一些细节:

1. bert中的句子级别的自监督任务, 提供两个句子时, 两个句子身上是有掩码的.
2. "您甚至可以通过传递 `num_proc` 参数在使用 `map()` 应用预处理函数时使用多处理。我们在这里没有这样做，因为 🤗 Tokenizers 库已经使用多个线程来更快地标记我们的样本，但是如果您没有使用该库支持的快速标记器，这可以加快您的预处理。"
3. 直接已tokenizer作为函数去调用可能会报错 需要3.0以上版本才行
4. 动态padding的好处之一

   Note that we've left the `padding` argument out in our tokenization function for now. This is because padding all the samples to the maximum length is not efficient: it's better to pad the samples when we're building a batch, as then we only need to pad to the maximum length in that batch, and not the maximum length in the entire dataset. This can save a lot of time and processing power when the inputs have very variable lengths!
5. 如果在线下载数据集失败了的话，可以先提前去github上档下来，参考链接

   https://blog.csdn.net/weixin_42655901/article/details/124246300

   记得在load_dataset的参数中加上cache_dir=，然后再去运行的时候，仍然会有进度条，不过这个反映的是加载和预处理数据的进度。

###

关于训练：

有两种fine-tune的训练方式：

1. Fine-tune a pretrained model with 🤗 Transformers Trainer.
2. Fine-tune a pretrained model in native PyTorch.

用Trainer训练

1. Prepare a dataset
2. 加载模型
3. 训练阶段
    1. 先定义一个TrainingArguments类，可以帮助我们记录所有训练器的超参'

    ```
    from transformers import TrainingArguments
    # 这里提供的参数是存放超参的地址路径
    # 第二个参数表示每一个epoch结束时都进行一次指标评估并打印
    training_args = TrainingArguments(output_dir="test-trainer" ,
    evaluation_strategy="epoch")
    ```

    2. 实例化计分类Metrics

    ```
    import numpy as np
    from datasets import load_metric
    # 这次的计分方式就打算用准确率
    metric = load_metric("accuracy")

    # 不过在调用metric的成员函数时，还得先稍稍预处理一下
    # 即对logits调用argmax
    def compute_metrics(eval_pred):
        logits, labels = eval_pred
        predictions = np.argmax(logits, axis=-1)
        return metric.compute(predictions=predictions, references=labels)
    ```

    3. 实例化Trainer

    ```
    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,
        compute_metrics=compute_metrics,
    )
    ```

    4. 开始finetune阶段的训练 使用trainer.train()即可
4. 不过现在的训练器只会自顾自训练并不会实时汇报损失函数的值（所以我们还需要定义评估器即compute_metrics()函数）

5. 不过现在也可以使用训练器进行预测了， 这个范例中的预测结果是（408,2）的维度，存着408个logits （all Transformer models return logits），2维是因为就两个label。因此还需要先扔进argmax才知道预测的是哪个label。

```
predictions = trainer.predict(tokenized_datasets["validation"])
print(predictions.predictions.shape, predictions.label_ids.shape)
preds = np.argmax(predictions.predictions, axis=-1)
```

关于评估：

```
def compute_metrics(eval_preds):
    metric = load_metric("glue", "mrpc")  # 官方都配好了专门的metric
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)
```

```
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,
)
```

一些细节：

metric = load_metric("glue", "mrpc") 必须要用sklearn库和scipy库

用pytorch fine-tune

因为是想着实现和Trainer训练的时候相同的效果, 所以用的优化器也是与Trainer中封装的相同.The optimizer used by the `Trainer` is `AdamW`, which is the same as Adam, but with a twist for weight decay regularization (see [“Decoupled Weight Decay Regularization”]() by Ilya Loshchilov and Frank Hutter)

Trainer的默认训练epoch是3个

training steps = the number of epochs * the number of training batches

加速器

用hug的官方的一个库,可以使得代码在多个GPU上进行训练

下面代码带有加号 就是新添加的 减号就是要删去的

```
+ 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)
```

第4课

介绍怎么上传东西到huggingface上 不重要

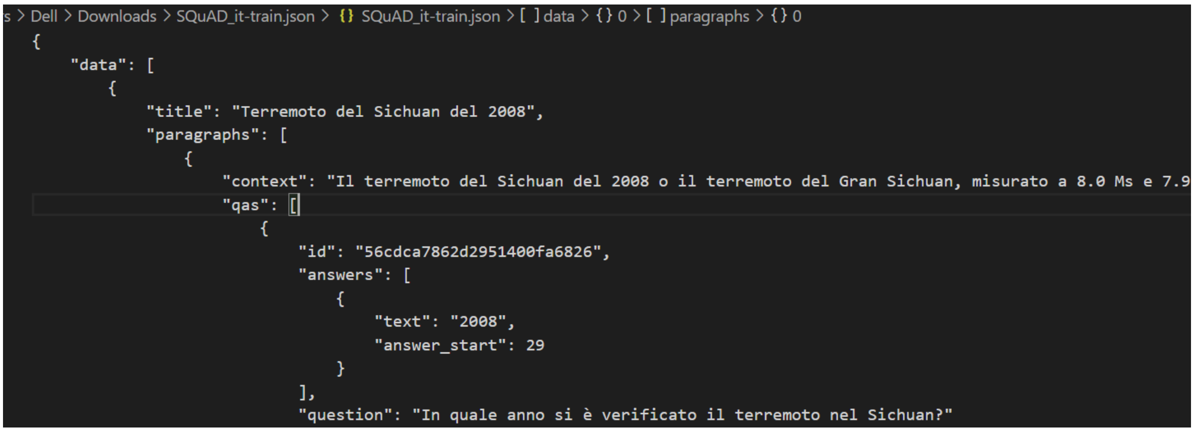
第5课

加载本地数据

Data format	Loading script	Example
CSV & TSV	`csv`	`load_dataset("csv", data_files="my_file.csv")`
Text files	`text`	`load_dataset("text", data_files="my_file.txt")`
JSON & JSON Lines	`json`	`load_dataset("json", data_files="my_file.jsonl")`
Pickled DataFrames	`pandas`	`load_dataset("pandas", data_files="my_dataframe.pkl")`

```
from datasets import load_dataset

squad_it_dataset = load_dataset("json", data_files="SQuAD_it-train.json",
field="data") # 这个field是因为数据里面就是这样的
```



s > Dell > Downloads > SQuAD_it-train.json > {} SQuAD_it-train.json > [ ] data > {} 0 > [ ] paragraphs > {} 0

```
{
    "data": [
        {
            "title": "Terremoto del Sichuan del 2008",
            "paragraphs": [
                {
                    "context": "Il terremoto del Sichuan del 2008 o il terremoto del Gran Sichuan, misurato a 8.0 Ms e 7.9
                    "qas": [
                        {
                            "id": "56cdca7862d2951400fa6826",
                            "answers": [
                                {
                                    "text": "2008",
                                    "answer_start": 29
                                }
                            ],
                            "question": "In quale anno si è verificato il terremoto nel Sichuan?"
```

```
squad_it_dataset["train"][0]
# 可以查看第一条数据， 这里之所以要先索引“train”，推测是因为通常都会像下面这种方式把训练集测试集捆绑读入，所以单个读入某个文件的时候默认是训练集（写test会报错，即使你加载的真的是测试集）

data_files = {"train": "SQuAD_it-train.json", "test": "SQuAD_it-test.json"}
squad_it_dataset = load_dataset("json", data_files=data_files, field="data")
squad_it_dataset
```

slice 与 dice

其实感觉还是像我们展示如何运用好map函数

```
drug_sample = drug_dataset["train"].shuffle(seed=42).select(range(1000))
# Peek at the first few examples
drug_sample[:3]
# 这样的输出不是简单的输出前三个样例，而是将前三个样例的按照每列组成元组
# 输出如下图
```

```
{'Unnamed: 0': [87571, 178045, 80482],
 'drugName': ['Naproxen', 'Duloxetine', 'Mobic'],
 'condition': ['Gout, Acute', 'ibromyalgia', 'Inflammatory Co
 'review': ['"like the previous person mention, I&#039;m a st
  '"I have taken Cymbalta for about a year and a half for fib
  '"I have been taking Mobic for over a year with no side eff
 'rating': [9.0, 3.0, 10.0],
 'date': ['September 2, 2015', 'November 7, 2011', 'June 5, 2
 'usefulCount': [36, 13, 128]}
```

Dataset.select()的参数得是可迭代的索引， 因为这里喂进的是0~999, 所以按照这个数字返回前1000个样例

可以发现已有几个地方值得我们去仔细检查:

- unnamed 0字段，指数据中第一个字段没有注明名称。但似乎是个唯一性的字段
- condition字段好像是大小写混合
- 有的样例的review字段非常长
- review字段还有爬虫的痕迹，即html字符

unique函数

```
# drug_dataset.keys() 只有train 和 test两个key
for split in drug_dataset.keys():
    #
    assert len(drug_dataset[split]) == len(drug_dataset[split].unique("Unnamed: 0"))
```

可以发现使用了unique函数之后 两个长度还是相等，所以确实有唯一标识的作用

同时可以给这个字段取个名字

```
drug_dataset = drug_dataset.rename_column(
    original_column_name="Unnamed: 0", new_column_name="patient_id"
)
```

map函数进行小写化 以及过滤

调用下面这个函数，去给该字段的字符串进行小写化

```
def lowercase_condition(example):
    return {"condition": example["condition"].lower()}

drug_dataset.map(lowercase_condition)
```

不过会报错， 因为有的样例正好没有这个字段，所以需要过滤一下

```
def filter_nones(x):
    return x["condition"] is not None
# 可以选择将上面这个函数作为参数送入filter 或者直接用lambda函数。
drug_dataset = drug_dataset.filter(lambda x: x["condition"] is not None)
# 再调用map（小写）即可
```

map函数进行字段增加

可以使用map函数统计review字段的长度，并把这个长度记录到数据中，只要让map的参数函数返回一个字典即可

```
def compute_review_length(example):
    return {"review_length": len(example["review"].split())}

drug_dataset = drug_dataset.map(compute_review_length)
# Inspect the first training example
drug_dataset["train"][0]
```

很多时候，太短的字串没有统计意义，这里就假设只提取30个字以上的样例

```
drug_dataset = drug_dataset.filter(lambda x: x["review_length"] > 30)
print(drug_dataset.num_rows)
```

map函数清洗html字串

```
drug_dataset = drug_dataset.map(lambda x: {"review":
html.unescape(x["review"])})
```

关于数据处理的速度的讨论

map函数中有个参数为可设置为 batched=True

AutoTokenizer中有个参数可设置为use_fast=True（默认就是True）

```
slow_tokenizer = AutoTokenizer.from_pretrained("bert-base-cased",
use_fast=False)

def slow_tokenize_function(examples):
    return slow_tokenizer(examples["review"], truncation=True)

tokenized_dataset = drug_dataset.map(slow_tokenize_function, batched=True,
num_proc=8)
```

Options	Fast tokenizer	Slow tokenizer
`batched=True`	10.8s	4min41s
`batched=False`	59.2s	5min3s

结论：batched会明显快很多，是因为一次对多个样例执行，能充分利用并发。

Fast tokenizer是因为底层用的Rust，能更好并发

map函数底层不是用Rust实现的，但可以使用多线程（进程），设置num_proc=

8

Options	Fast tokenizer	Slow tokenizer
`batched=True`	10.8s	4min41s
`batched=False`	59.2s	5min3s
`batched=True`, `num_proc=8`	6.52s	41.3s
`batched=False`, `num_proc=8`	9.49s	45.2s

但还是不建议，batched=True和multiprocess以及fast三个一起开。

return_overflowing_tokens

```
def tokenize_and_split(examples):
    return tokenizer(
        examples["review"],
        truncation=True,
        max_length=128,
        return_overflowing_tokens=True,
    )
result = tokenize_and_split(drug_dataset["train"][0])
[len(inp) for inp in result["input_ids"]]
# [128, 49]
# 开了return_overflowing_tokens之后，截断的句子会单独作为一个新的样例
# 即本来这个样例是128+49的token数， 但是上限是128所以截断成两个独立的
```

现在我们想对整个数据进行map改造，但是却报错了

```
tokenized_dataset = drug_dataset.map(tokenize_and_split, batched=True)  # 如果把batched去掉则可以运行 但是样例总数不会变多 这是我疑惑的地方，看了一下review部分没有切割。
```

```
ArrowInvalid: Column 1 named condition expected length 1463 but got length 1000
```

drug_dataset	drug_dataset.map(tokenize_and_split)
0 = {str} 'patient_id'	00 = {str} 'patient_id'
1 = {str} 'drugName'	01 = {str} 'drugName'
2 = {str} 'condition'	02 = {str} 'condition'
3 = {str} 'review'	03 = {str} 'review'
4 = {str} 'rating'	04 = {str} 'rating'
5 = {str} 'date'	05 = {str} 'date'
6 = {str} 'usefulCount'	06 = {str} 'usefulCount'
7 = {str} 'review_length'	07 = {str} 'review_length'
	08 = {str} 'input_ids'
	09 = {str} 'token_type_ids'
	10 = {str} 'attention_mask'
	11 = {str} 'overflow_to_sample_mapping'

（这里是个人理解，还是不能完全说通）

因为我们自定义的这个函数会拆分出许多东西，然后batched默认是一次对一堆样例操作，结果这些样例的特征不同，即有的特征是因裁剪多出来而新创造的样例，他会没有某些列，

而一个一个样例操作的时候就没有问题。就是新多出来的列的行数，与旧列的行数不一致

```
tokenized_dataset = drug_dataset.map(
    tokenize_and_split, batched=True,
remove_columns=drug_dataset["train"].column_names
)
# 样例数量会增加 上面报错的那种把
```

0 = {str} 'input_ids'
1 = {str} 'token_type_ids'
2 = {str} 'attention_mask'
3 = {str} 'overflow_to_sample_mapping'

另一种解决bug的方式，是使旧的列的行数增加至新列的行数

```
def tokenize_and_split(examples):
    result = tokenizer(
        examples["review"],
        truncation=True,
        max_length=128,
        return_overflowing_tokens=True,
    )
    # Extract mapping between new and old indices
    # 这个overflow_to_sample_mapping字段，存着新出来的样例的下标到原来下标的有映射，这样就
可以把他分出来之前的东西拿过来复制一份了。
    sample_map = result.pop("overflow_to_sample_mapping")
    for key, values in examples.items():
        result[key] = [values[i] for i in sample_map]
    return result
```

与pandas的api适配

```
drug_dataset.set_format("pandas")
# 复制的时候还是要带上最后这个冒号
train_df = drug_dataset["train"][:]

# 这样可以先利用pandas中强大的统计链式函数
frequencies = (
    train_df["condition"]
    .value_counts()  # 频率统计
    .to_frame()  # 转换成dataFrame
    .reset_index()  # 将原来的index（即condition）作为一个新的字段
    .rename(columns={"index": "condition", "condition": "frequency"})  # 将各字段
重命名
)
frequencies.head()

'''
reset_index之前
                condition
birth control   27655
depression      8023
acne            5209
anxiety         4991
pain            4744
改名之后
    condition    frequency
0   birth control   27655
1   depression  8023
2   acne     5209
3   anxiety 4991
4   pain     4744
'''
# 然后再转换成dataset格式
from datasets import Dataset
freq_dataset = Dataset.from_pandas(frequencies)

# 另外可以使用这个函数使其从panda格式恢复原状
drug_dataset.reset_format()
```

建立验证集

底层借助的是sklearn的api

```
drug_dataset_clean = drug_dataset["train"].train_test_split(train_size=0.8,
seed=42)
# Rename the default "test" split to "validation"
drug_dataset_clean["validation"] = drug_dataset_clean.pop("test")
# Add the "test" set to our `DatasetDict`
drug_dataset_clean["test"] = drug_dataset["test"]
drug_dataset_clean
```

```
DatasetDict({
    train: Dataset({
        features: ['patient_id', 'drugName', 'condition', 'review',
        num_rows: 110811
    })
    validation: Dataset({
        features: ['patient_id', 'drugName', 'condition', 'review',
        num_rows: 27703
    })
    test: Dataset({
        features: ['patient_id', 'drugName', 'condition', 'review',
        num_rows: 46108
    })
})
```

保存修改后的数据

Data format	Function
Arrow	`Dataset.save_to_disk()`
CSV	`Dataset.to_csv()`
JSON	`Dataset.to_json()`

关于Arrow

```
drug-reviews/
├── dataset_dict.json
├── test
│   ├── dataset.arrow
│   ├── dataset_info.json
│   └── state.json
├── train
│   ├── dataset.arrow
│   ├── dataset_info.json
│   ├── indices.arrow
│   └── state.json
└── validation
    ├── dataset.arrow
    ├── dataset_info.json
    ├── indices.arrow
    └── state.json
```

第6课 关于分词器

有时候我们需要一个全新的分词器，而不是现成的，比如拿着英语语料库上训练的分词器给中文分词肯定有问题。

本节课包含以下4点

1. 如何在新语料库中训练一个与要使用的checkpoint所调出来的模型相适应的分词器
2. fast分词器的特点
3. 主流Subword

4. 如何使用 Tokenizers 库从头开始构建标记器并在一些数据上对其进行训练

##训练新的tokenizer（当corpus不同时）

针对不同的语料库，可能预先训练好的不能直接用。举的例子是，如果corpus全都是python语言的代码，用一个旧的分词器，比如：old_tokenizer = AutoTokenizer.from_pretrained("gpt2")，他的分词效果如下：

```
example = '''def add_numbers(a, b):
    """Add the two numbers `a` and `b`."""
    return a + b'''

tokens = old_tokenizer.tokenize(example)
tokens
# This tokenizer has a few special symbols, like Ġ and Ċ, which denote spaces and newlines, respectively.
```

```
['def', 'Ġadd', '_', 'n', 'umbers', '(', 'a', ',', 'Ġb', '):', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġ"""', 'Add', 'Ġthe',
 'Ġnumbers', 'Ġ`', 'a', '`', 'Ġand', 'Ġ`', 'b', '`', '."', '""', 'Ċ', 'Ġ', 'Ġ', 'Ġ', 'Ġreturn', 'Ġa', 'Ġ+
```

当然也不可能重头开始训练，而是在旧的基础上再训练分词器。

```
def get_training_corpus():
    return (
        raw_datasets["train"][i : i + 1000]["whole_func_string"]
        for i in range(0, len(raw_datasets["train"]), 1000)
    )
# 用迭代器能更好的节省内存，使得并发效率更高

training_corpus = get_training_corpus()

tokenizer = old_tokenizer.train_new_from_iterator(training_corpus, 52000)
# 52000是词典大小
tokenizer.save_pretrained("code-search-net-tokenizer") # 可以将训练好的分词器保存下来
```

###

分词器的运行结果

{'input_ids': [101, 1422, 1271, 1110, 156, 7777, 2497, 1394, 1105, 146, 1250, 1120, 20164, 10932, 10289, 1107, 6010, 119, 102], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}

看起来像一个字典, 但其实是一个字典的子类,叫做BatchEncoding对象.

Besides their parallelization capabilities, the key functionality of fast tokenizers is that they always keep track of the original span of texts the final tokens come from — a feature we call *offset mapping*. 这段话的意思是说分词器可以把数字和原先的单词能牢牢地对应. 比如下例中的 ##yl就是Sylvain的一部分, 一堆3也表示都是下标3的单词的子词.

```
encoding.tokens()
# ['[CLS]', 'My', 'name', 'is', 'S', '##yl', '##va', '##in', 'and', 'I', 'work',
'at', 'Hu', '##gging', 'Face', 'in',
 'Brooklyn', '.', '[SEP]']
encoding.word_ids()
# [None, 0, 1, 2, 3, 3, 3, 3, 4, 5, 6, 7, 8, 8, 9, 10, 11, 12, None]
start, end = encoding.word_to_chars(3)
example[start:end]
# Sylvain
```

这个功能的好处在POS以及NER中都能体现, 还包括whole word masking, 后面会详说.

下面要讲解第一章中的调用起来非常便捷的pipeline内部是怎么工作的

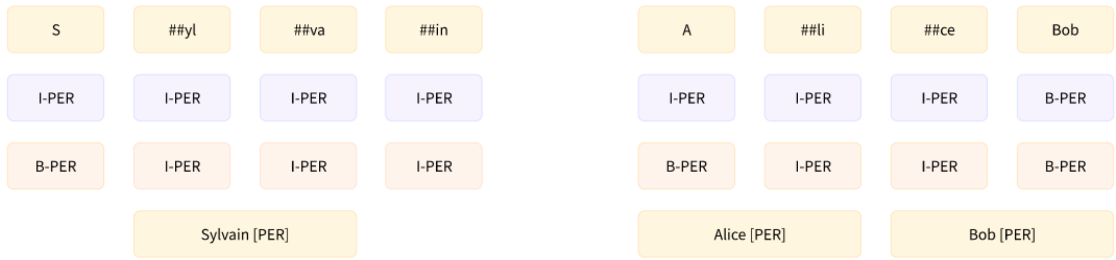
```
from transformers import AutoTokenizer, AutoModelForTokenClassification

model_checkpoint = "dbmdz/bert-large-cased-finetuned-conll03-english"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
model = AutoModelForTokenClassification.from_pretrained(model_checkpoint)

example = "My name is Sylvain and I work at Hugging Face in Brooklyn."
# 先分词
inputs = tokenizer(example, return_tensors="pt")
# 模型输出(因为选的是分类器 所以输出的会是logits)
outputs = model(**inputs)

print(inputs["input_ids"].shape)
print(outputs.logits.shape)
# 因为只有一个数据batch大小自然是1
# torch.Size([1, 19])
# torch.Size([1, 19, 9])'
# 将logist通过softmax
import torch

probabilities = torch.nn.functional.softmax(outputs.logits, dim=-1)[0].tolist()
predictions = outputs.logits.argmax(dim=-1)[0].tolist()
print(predictions)
print(model.config.id2label)
# [0, 0, 0, 0, 4, 4, 4, 4, 0, 0, 0, 0, 6, 6, 6, 0, 8, 0, 0]
# {0: 'O', 1: 'B-MISC', 2: 'I-MISC', 3: 'B-PER', 4: 'I-PER', 5: 'B-ORG', 6: 'I-
ORG', 7: 'B-LOC', 8: 'I-LOC'}
# 这里得解释一下为啥会是四个4, 下图紫色的就是现在的模式, 通常觉得红色的方块才是正常的, 但这个紫
色也是一种模式,他们的B是用来区分两个凑在一起的实体,就像右图一样
```




```
results = []
tokens = inputs.tokens()

for idx, pred in enumerate(predictions):
    label = model.config.id2label[pred]
    if label != "O":
        results.append(
            {"entity": label, "score": probabilities[idx][pred], "word":
tokens[idx]}
        )
# 上面这个循环把数字label映射到 真实的label, 且输出了概率
print(results)
```

```
[{'entity': 'I-PER', 'score': 0.9993828, 'index': 4, 'word': 'S', 'start': 11, 'end': 12},
 {'entity': 'I-PER', 'score': 0.99815476, 'index': 5, 'word': '##yl', 'start': 12, 'end': 14},
 {'entity': 'I-PER', 'score': 0.99590725, 'index': 6, 'word': '##va', 'start': 14, 'end': 16},
 {'entity': 'I-PER', 'score': 0.9992327, 'index': 7, 'word': '##in', 'start': 16, 'end': 18},
 {'entity': 'I-ORG', 'score': 0.97389334, 'index': 12, 'word': 'Hu', 'start': 33, 'end': 35},
 {'entity': 'I-ORG', 'score': 0.976115, 'index': 13, 'word': '##gging', 'start': 35, 'end': 40},
 {'entity': 'I-ORG', 'score': 0.98879766, 'index': 14, 'word': 'Face', 'start': 41, 'end': 45},
 {'entity': 'I-LOC', 'score': 0.99321055, 'index': 16, 'word': 'Brooklyn', 'start': 49, 'end': 57}]
```

在经过一些小处理, 利用前面提到的offset, 就能把这些分散的标识拼在一起, 代码如下

```
import numpy as np

results = []
inputs_with_offsets = tokenizer(example, return_offsets_mapping=True)
tokens = inputs_with_offsets.tokens()
offsets = inputs_with_offsets["offset_mapping"]

idx = 0
while idx < len(predictions):
    pred = predictions[idx]
    label = model.config.id2label[pred]
    if label != "O":
        # Remove the B- or I-
        label = label[2:]
        start, _ = offsets[idx]

        # Grab all the tokens labeled with I-label
        all_scores = []
        while (
            idx < len(predictions)
            and model.config.id2label[predictions[idx]] == f"I-{label}"
        ):
            all_scores.append(probabilities[idx][pred])
```

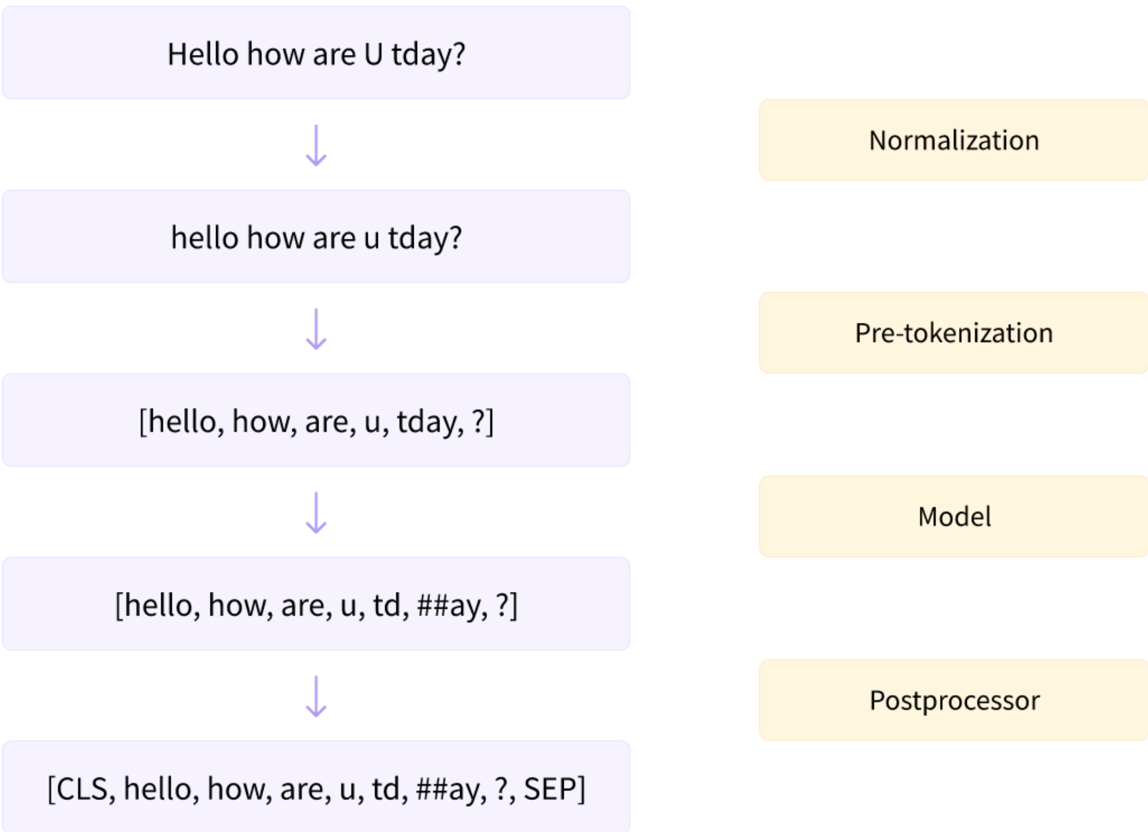
```
            _, end = offsets[idx]
            idx += 1

        # The score is the mean of all the scores of the tokens in that grouped entity
        score = np.mean(all_scores).item()
        word = example[start:end]
        results.append(
            {
                "entity_group": label,
                "score": score,
                "word": word,
                "start": start,
                "end": end,
            }
        )
    idx += 1

print(results)
```

Normalization and pre-tokenization

下图是high-level overview of the steps in the tokenization pipeline(下图的model指的就是分词器)



如图所示,分词器真正的工作之前还有两步: 规范化和预分词

规范化涉及的内容:

1. removing needless whitespace
2. lowercasing
3. removing accents.等

这些步骤自然 已经被集成到分词器中去了, 也可以自己调用

```
print(tokenizer.backend_tokenizer.normalizer.normalize_str("Héllò hôw are ü?"))
```

预分词的工作内容 类似于pyhton中的split函数

三种subword tokenization

BPE (used by GPT-2 and others)

WordPiece (used for example by BERT)

Unigram (used by T5 and others).

Model	BPE	WordPiece	Unigram
Training	Starts from a small vocabulary and learns rules to merge tokens	Starts from a small vocabulary and learns rules to merge tokens	Starts from a large vocabulary and learns rules to remove tokens
Training step	Merges the tokens corresponding to the most common pair	Merges the tokens corresponding to the pair with the best score based on the frequency of the pair, privileging pairs where each individual token is less frequent	Removes all the tokens in the vocabulary that will minimize the loss computed on the whole corpus
Learns	Merge rules and a vocabulary	Just a vocabulary	A vocabulary with a score for each token
Encoding	Splits a word into characters and applies the merges learned during training	Finds the longest subword starting from the beginning that is in the vocabulary, then does the same for the rest of the word	Finds the most likely split into tokens, using the scores learned during training

一些细节：

1. 训练分词器与训练模型不同！模型训练使用随机梯度下降来使每个批次的损失更小一些。它本质上是随机的。训练分词器是一个统计过程，它试图确定哪些子词是给定语料库的最佳选择，而用于选择它们的确切规则取决于分词算法。它是确定性的，这意味着在相同的语料库上使用相同的算法进行训练时，总是会得到相同的结果。
2. 如果数据集很大的话，应该把数据集转换成iterator格式，而不是直接是list格式，这样能避免一次性全部加载内存中来。
3. Slow tokenizers are those written in Python inside the 🤗 Transformers library, while the fast versions are the ones provided by 🤗 Tokenizers, which are written in Rust.