第 5 章 大模型的预训练

预训练是大模型训练的第 1 步，它从大量的文本中通过学习预测下一个 token 的方式来学习知识。预训练阶段是大模型获取能力最重要的阶段。应该给大模型提供尽可能多样性的高质量文本，让它学习各种知识，例如 Llama3 的预训练就用到了 15T 个 token。训练如此多的 token，自然代价也是非常大的，很少有公司可以负担得起。

5.1 预训练的作用

开源模型为了提供大模型的通用能力会在各种各样的文本上进行预训练，提高大模型的综合能力。如果想将大模型引入特定的行业或某个领域，就需要在开源预训练大模型的基础上继续进行预训练(Post Pre-train)，以此来让它学习特定领域的知识。

想让大模型在一些专业领域的问题上有好的表现，进行预训练是必不可少的。特别是当这些领域在互联网上资料较少时。也有人尝试不通过预训练，只在后续微调时引入新的知识，但实践效果都不好。

假如你的项目中需要训练一个辅助程序员生成 Scala 代码的大模型。这就需要去 GitHub 收集大量注释良好的 Scala 代码，进行预训练。让大模型学习功能描述和 Scala 代码实现之间的联系。

实验表明，预训练是大模型学习知识的环节，而微调只是改进大模型的输出方式，让它的输出更符合人类的期望。

5.2 预训练的数据

预训练一个大模型，获得高质量的预训练数据是首先要考虑的问题。大语言模型的预训练数据是由大量的非结构化的文本构成的。高质量的预训练数据应该主要包含以下特征。

(1) 多样性：数据应涵盖各种主题和语言风格，以确保模型在不同的场景中具有良好的泛化能力。这包括书籍、新闻文章、科学论文、社交媒体帖子、对话记录等。同时应该尽量

地避免在训练集里出现重复的文本。

(2) 大规模：预训练数据集应足够大，以便模型能够学习到丰富的语言模式和知识。通常，数据量越大，模型的性能越好。

(3) 高质量：数据应经过严格的筛选和清洗，尽量减少噪声和错误信息。高质量的数据能帮助模型学到更准确和更有用的知识。

(4) 时效性：数据应包含最新的信息和事件，以便模型能够理解和生成当前最新的内容和知识。

(5) 平衡性：数据集应在不同类别、话题和观点之间保持平衡，避免模型产生偏见，包括性别、种族、文化、地域等方面的平衡。

(6) 合法性和伦理性：数据的收集和使用应遵循相关法律法规和伦理准则，确保数据来源合法，保护用户隐私。

高质量的训练数据收集和准备通常要花费大量的人力，但是这些努力都是值得的。只有高质量的数据才能训练出高质量的模型。

HuggingFace Datasets 库提供了便捷的数据读取、处理和缓存功能。特别是对于训练大模型所需的大规模数据集，该库得益于其背后的 Apache Arrow 支持，可以实现零复制读取，而不受内存大小的限制。同时 Datasets 库和 HuggingFace Hub 做了深度集成，方便用户下载和分享数据。

如果之前没有安装 Datasets 库，则安装命令如下：

```
pip install datasets
```

Datasets 库提供了多种数据加载的方法，最简单的加载方式是从 HuggingFace Hub 来加载数据。首先，可以在网页 https://huggingface.co/datasets 寻找合适的数据，并且可以对数据进行预览，例如对于 cornell-movie-review-data/rotten_tomatoes 数据集，如图 5-1 所示。

通过 Datasets 库的 load_dataset 来从 Hub 加载数据，代码如下：

```
from datasets import load_dataset
dataset = load_dataset("cornell-movie-review-data/rotten_tomatoes",
split="train")
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text', 'label'],
    num_rows: 8530
})
```

Dataset 对象里的数据是由列构成的，每列可以是不同的类型。可以按行的索引来访问数据中的样本，代码如下：

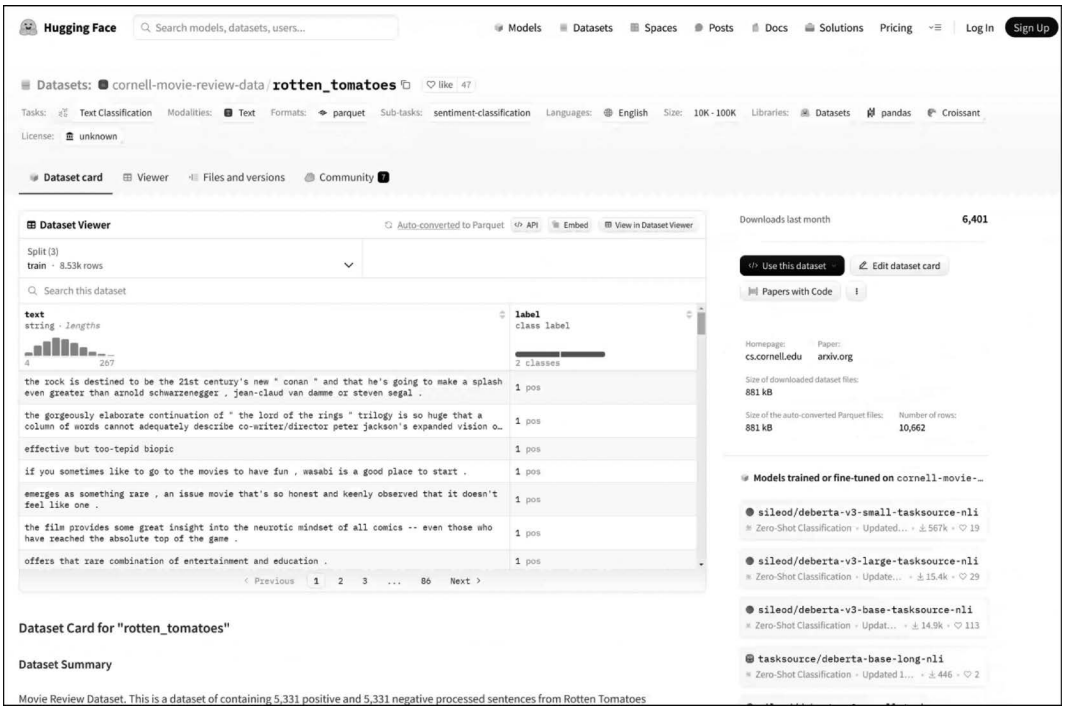

图 5-1 rotten_tomatoes 数据集

```
print(dataset[0])
```

输出如下：

```
{'text': 'the rock is destined to be the 21st century\'s new " conan " and that he\'s going to make a splash even greater than arnold schwarzenegger , jean-claud van damme or steven segal .', 'label': 1}
```

输出是一个字典类型的样本。同时也可以通过切片的方式来访问一组数据，代码如下：

```
print(dataset[-5:])
```

输出如下：

```
{'text': ['any enjoyment will be hinge from a personal threshold of watching sad but endearing characters do extremely unconventional things .', "if legendary shlockmeister ed wood had ever made a movie about a vampire , it probably would look a lot like this alarming production , adapted from anne rice's novel the vampire chronicles .", "hardly a nuanced portrait of a young woman's breakdown , the film nevertheless works up a few scares .", 'interminably bleak , to say nothing of boring .', 'things really get weird , though not particularly scary : the movie is all portent and no content .'], 'label': [0, 0, 0, 0, 0]}
```

对于这个数据集，如果只想利用 text 列进行模型预训练，而不关心 label 列，则可以通过列名来选择，代码如下：

```
print(dataset["text"])
```

输出为一个字符串列表：

```
['the rock is destined to be the 21st century\'s new " conan " and that he\'s going
to make a splash even greater than arnold schwarzenegger , jean-claud van damme or
steven segal .',
'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so
huge that a column of words cannot adequately describe co-writer/director peter
jackson\'s expanded vision of j . r . r . tolkien\'s middle-earth .',
'effective but too-tepid biopic',
...,
'things really get weird , though not particularly scary : the movie is all
portent and no content .']
```

也可以通过组合行和列来返回一个确定的数据元素，但是为了更有效地访问数据，最好先指定行的 index，然后指定列名，代码如下：

```
print(dataset[0]["text"])
```

输出如下：

```
'the rock is destined to be the 21st century\'s new " conan " and that he\'s going
to make a splash even greater than arnold schwarzenegger , jean-claud van damme or
steven segal .'
```

在更多的情况下，需要从本地加载自己的数据，Datasets 库也提供了相应的支持，例如对于 txt 文件，加载数据的代码如下：

```
from datasets import load_dataset
dataset = load_dataset("text", data_files=r"D:\news1.txt")
print(dataset)
```

Datasets 会按行加载 txt 文件中的数据，默认列名为 text，得到的输出如下：

```
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 13
    })
})
```

如果有多个文件，则可以给 data_files 参数传入多个文件的列表，代码如下：

```
from datasets import load_dataset
dataset = load_dataset("text", data_files=[r"D:\news1.txt",r"D:\news2.txt",r"
D:\news3.txt"])
print(dataset)
```

num_rows 是所有文件的行数之和,输出如下:

```
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 50
    })
})
```

在训练大模型时,另一种常用的文件格式是 JSON 数据,例如对于如下格式的数据:

```
{"id": 1, "query": "保持健康的 3 个提示。", "answer": "以下是保持健康的 3 个提示:\n\
n1. 保持身体活动。每天做适当的身体运动,如散步、跑步或游泳,能促进心血管健康,增强肌肉力
量,并有助于减少体重。\n\n2. 均衡饮食。每天食用新鲜的蔬菜、水果、全谷物和脂肪含量低的蛋
白质食物,避免高糖、高脂肪和加工食品,以保持健康的饮食习惯。\n\n3. 睡眠充足。睡眠对人体
健康至关重要,成年人每天应保证 7~8 小时的睡眠。良好的睡眠有助于减轻压力,促进身体恢复,
并提高注意力和记忆力。"}
{"id": 2, "query": "三原色是什么? ", "answer": "三原色通常指的是红色、绿色和蓝色
(RGB)。它们是通过加色混合原理创建色彩的 3 种基础颜色。在以发光为基础的显示设备中(如电
视、计算机显示器、智能手机和平板电脑显示屏),三原色可混合产生大量色彩,其中红色和绿色可
以混合生成黄色,红色和蓝色可以混合生成品红色,蓝色和绿色可以混合生成青色。当红色、绿色
和蓝色按相等比例混合时,可以产生白色或灰色。\n\n 此外,在印刷和绘画中,三原色指的是以颜
料为基础的红、黄和蓝颜色(RYB)。这 3 种颜色用以通过减色混合原理来创建色彩。不过,三原色
的具体定义并不唯一,不同的颜色系统可能会采用不同的三原色。"}
…
```

对 JSON 类型数据的读入,代码如下:

```
from datasets import load_dataset
dataset = load_dataset("json", data_files=r"D:\gpt4.jsonl")
print(dataset)
```

Datasets 库会自动解析 JSON 格式的数据,并把 JSON 里的 key 作为列名,方便进一步读取数据,输出如下:

```
DatasetDict({
    train: Dataset({
        features: ['id', 'query', 'answer'],
        num_rows: 48647
    })
})
```

可以先将多个文件放入一个文件夹,然后将 load_dataset()里的 data_files 参数替换成

data_dir 参数来一次性地读取文件夹内所有的文件，代码如下：

```
from datasets import load_dataset
dataset = load_dataset("text", data_dir=r'D:\news')
print(dataset)
```

Datasets 库也支持利用不同结构的内存数据来创建 Dataset。从一个字典类型对象来创建 Dataset，代码如下：

```
from datasets import Dataset
my_dict = {"text": ["text1", "text2", "text3"]}
dataset = Dataset.from_dict(my_dict)
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text'],
    num_rows: 3
})
```

从列表对象创建一个 Dataset 对象，代码如下：

```
from datasets import Dataset
my_list = [{"text": "text1"}, {"text": "text2"}, {"text": "text3"}]
dataset = Dataset.from_list(my_list)
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text'],
    num_rows: 3
})
```

从一个 Python 的生成器来创建一个 Dataset 对象，代码如下：

```
from datasets import Dataset

def my_gen():
    for i in range(1, 4):
        yield {"text": f"text{i}"}

dataset = Dataset.from_generator(my_gen)
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text'],
```

```
    num_rows: 3
})
```

有了上边丰富的创建 Dataset 的方法,可以方便地创建一个 Dataset 对象。同时,Datasets 库也提供了丰富的对数据进行处理的方法。

(1) sort 可以按照数字类型的列对整个数据集进行排序,代码如下:

```
sorted_dataset = dataset.sort("numerical_column")
```

(2) shuffle 用于打乱数据的顺序,代码如下:

```
shuffled_dataset = sorted_dataset.shuffle(seed=123)
```

(3) select 用于选择指定 index 的数据,代码如下:

```
small_dataset = dataset.select([0, 10, 20, 30, 40, 50])
```

(4) filter 用于对数据进行过滤,例如下边的代码用于演示如何过滤出 text 列以 Abc 开头的句子,代码如下:

```
start_with_Abc = dataset.filter(lambda example: example["text"].startswith("
Abc"))
```

(5) split 用于对数据集按训练集和测试集进行划分,代码如下:

```
dataset.train_test_split(test_size=0.1)
```

数据集划分后返回的结果为一个字典,里边包含 train 和 test 的 Dataset,结果如下:

```
{'train': Dataset(schema: {'text': 'string','idx': 'int32'}, num_rows: 3301),
'test': Dataset(schema: {'text': 'string','idx': 'int32'}, num_rows: 367)}
```

(6) shard 用于对一个大的数据集进行分块,通过指定 num_shards 来确定将数据分为多少块,通过提供 index 参数来指定要返回的数据分块,代码如下:

```
from datasets import load_dataset
dataset = load_dataset("imdb", split="train")
print(dataset)
sub_dataset = dataset.shard(num_shards=4, index=0)
print(sub_dataset)
```

输出如下:

```
Dataset({
    features: ['text', 'label'],
    num_rows: 25000
})
```

```
Dataset({
    features: ['text', 'label'],
    num_rows: 6250
})
```

（7）map：可以通过 map()方法来完成一些更复杂的对数据的操作，可以提供一个对每个样本进行处理的方法。这种方法可以是对单个样本或者对一个批次样本的操作，代码如下：

```
from datasets import Dataset

text_dict = {"text":["苹果","橘子","香蕉"]}
dataset = Dataset.from_dict(text_dict)
print(dataset)

def add_prompt(example):
    example["text"]="翻译下边这句话为英语:\n"+example["text"]
    return example
dataset = dataset.map(add_prompt)
print(dataset)
print(dataset["text"])
```

输出如下：

```
Dataset({
    features: ['text'],
    num_rows: 3
})
Dataset({
    features: ['text'],
    num_rows: 3
})
['翻译下边这句话为英语:\n 苹果', '翻译下边这句话为英语:\n 橘子', '翻译下边这句话为英语:\n 香蕉']
```

通过将 batched 指定为 True，可以对一批数据进行操作，并且返回的行数不必等于输入的行数，因此可以在这里进行改变数据集样本个数的操作，例如对长的文本进行切分，代码如下：

```
from datasets import Dataset

text_dict = {"text": ["苹果", "橘子", "香蕉"]}
dataset = Dataset.from_dict(text_dict)
print(dataset)


def add_prompt(example):
```

```
    text = []
    for e in example["text"]:
        text.append("翻译下边这句话为英语:\n" + e)
        text.append("翻译下边这句话为法语:\n" + e)
    return {"text": text}


dataset = dataset.map(add_prompt, batched=True)
print(dataset)
print(dataset["text"])
```

输出如下：

```
Dataset({
    features: ['text'],
    num_rows: 3
})
Dataset({
    features: ['text'],
    num_rows: 6
})
['翻译下边这句话为英语:\n苹果', '翻译下边这句话为法语:\n苹果', '翻译下边这句话为英语:\n橘子', '翻译下边这句话为法语:\n橘子', '翻译下边这句话为英语:\n香蕉', '翻译下边这句话为法语:\n香蕉']
```

Dataset 对象也可以和 PyTorch 里的 DataLoader 来配合使用，代码如下：

```
dataset.set_format(type="torch",columns=["text"])
data_loader = DataLoader(dataset, batch_size=2)
for batch_data in data_loader:
    print(batch_data)
```

输出如下：

```
{'text': ['翻译下边这句话为英语:\n苹果', '翻译下边这句话为法语:\n苹果']}
{'text': ['翻译下边这句话为英语:\n橘子', '翻译下边这句话为法语:\n橘子']}
{'text': ['翻译下边这句话为英语:\n香蕉', '翻译下边这句话为法语:\n香蕉']}
```

处理完的数据可以对数据进行保存和加载，示例代码如下：

```
dataset.save_to_disk("path/of/my/dataset/directory")

from datasets import load_from_disk
reloaded_dataset = load_from_disk("path/of/my/dataset/directory")
```

另外也可以将处理好的数据以 CSV 或者 JSON 格式导出，示例代码如下：

```
dataset.to_csv("path/of/my/dataset.csv")
dataset.to_json("path/of/my/dataset.json")
```

HuggingFace 的 Dataset 库采用了 Arrow 作为本地缓存系统，它允许对磁盘上的数据通过虚拟内存映射的方式进行快速查找，这种架构使在设备内存相对较小的机器上也可以使用大型数据集进行模型训练。

另一个让 Dataset 高效的原因是它内部采用了缓存技术，它会缓存之前下载的或者处理过的数据集，所以当再次访问这些数据时会从缓存直接加载，避免了重新下载或者对数据的再次处理，并且这个缓存是全局的，即使关闭当前 Python 进程，重新在一个新的 Python 进程里去访问这些数据，缓存也是有效的。

Dataset 的缓存是如何知道这些缓存的数据都应用了哪些操作呢？每个缓存都有一个指纹。这个指纹描述了这个缓存的当前状态，最初的指纹是通过原始的数据的哈希来计算的，获取 Dataset 指纹的代码如下：

```
from datasets import Dataset
dataset1 = Dataset.from_dict({"a": [0, 1, 2]})
dataset2 = dataset1.map(lambda x: {"a": x["a"] + 1})
print(dataset1._fingerprint, dataset2._fingerprint)
```

执行结果如下：

```
d19493523d95e2dc 5b86abacd4b42434
```

后续的缓存指纹是通过之前的指纹加上当前对数据的操作生成的。当前对数据的操作的哈希包含了操作的方法和传入的参数，这样保证了一个缓存的指纹对应一个确定的数据状态，代码如下：

```
from datasets.fingerprint import Hasher
my_func = lambda example: {"length": len(example["text"])}
print(Hasher.hash(my_func))
```

执行结果如下：

```
'3d35e2b3e94c81d6'
```

5.3　预训练的方法

为了简化模型的训练，HuggingFace 里提供了 Trainer 类，它包含了完整的训练和评估流程的实现。开发人员只需传入训练时必要的模块，例如模型、Tokenizer、数据集、评估方法、训练的超参数等。剩下的工作都会由 Trainer 来完成。这让开发人员可以快速地开始训练，而不必重复地去写那些训练、评估、模型保存等代码。同时 Trainer 也是非常灵活的，可以根据开发人员的需求去定制。

Trainer 里包含了一个标准训练流程所需的基本模块：

(1) 执行训练的一步，计算 loss。

(2) 通过调用 loss 的 backward()来计算梯度。

(3) 根据梯度更新参数的权重。

(4) 重复上述步骤,直到完成指定的迭代。

Trainer 类对上述的过程进行了抽象,从而不用每次都重复写这个训练循环,开发人员只需提供像模型和数据这样训练时必要的对象就可以了。

可以通过一个 TrainingArguments 对象来指定训练时的超参数,例如可以通过 output_dir 来指定模型保存在哪里,代码如下:

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="your-model-path",
    learning_rate=2e-5,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=2,
    num_train_epochs=2,
    weight_decay=0.01,
    eval_strategy="epoch",
    save_strategy="epoch",
    load_best_model_at_end=True,
)
```

然后把 training_args、模型、数据一起传递给 Trainer 对象。同时还可以给 Trainer 对象传入对数据进行处理的 Data Collator,还有在训练过程中监控的指标项等,然后调用 trainer 对象的 train()方法就可以进行训练了,代码如下:

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

trainer.train()
```

Trainer 对象在训练过程中会在通过 TrainingArguments 里指定的 output_dir 路径保存模型的 Checkpoints。Checkpoints 都被保存在类似 checkpoint-000 这样命名的文件夹内。后边的数字表示训练的 step。通过 Checkpoints 可以方便地恢复中断的训练,代码如下:

```
#从最近的 checkpoint 恢复训练
trainer.train(resume_from_checkpoint=True)

#从指定的 checkpoint 恢复训练
trainer.train(resume_from_checkpoint="your-model/checkpoint-1000")
```

Trainer 类同时也支持定制化的训练。Trainer 的很多方法支持通过子类重载的方式来实现开发人员想要定制的功能。可以被重载的方法包如下。

(1) get_train_dataloader()：创建一个训练的 DataLoader。

(2) get_eval_dataloader()：创建一个评估的 DataLoader。

(3) get_test_dataloader()：创建一个测试的 DataLoader。

(4) log()：记录日志。

(5) create_optimizer_and_scheduler()：创建一个优化器和一个学习率规划器。

(6) create_optimizer()：创建一个优化器。

(7) create_scheduler()：创建一个学习率规划器。

(8) compute_loss()：根据一个 batch 的训练数据来计算 loss。

(9) training_step()：对一个 batch 的数据进行一次训练。

(10) prediction_step()：执行预测和测试的一步。

(11) evaluate()：评估模型并返回评估的指标值。

(12) predict()：在测试集上进行预测，如果提供了 label，则要进行评估指标的计算。

通过自定义 compute_loss()方法来重写一个带权重 loss 的代码如下：

```
from torch import nn
from transformers import Trainer

class CustomTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False):
        labels = inputs.pop("labels")
        #前向传播
        outputs = model(**inputs)
        logits = outputs.get("logits")
        #自定义 loss,不同的 label 有不同的权重
         loss_fct = nn.CrossEntropyLoss(weight=torch.tensor([1.0, 2.0, 3.0],
device=model.device))
        loss = loss_fct(logits.view(-1, self.model.config.num_labels), labels.
view(-1))
        return (loss, outputs) if return_outputs else loss
```

对 Trainer 的另一种定制化方式是通过 Callbacks 实现。Callbacks 不会改变训练过程，但它会检查训练过程的状态，然后根据状态执行某些操作，例如提前停止训练，或者记录一些信息等。在训练 10 步后停止训练的回调函数实现，代码如下：

```
from transformers import TrainerCallback

class EarlyStoppingCallback(TrainerCallback):
    def __init__(self, num_steps=10):
        self.num_steps = num_steps

    def on_step_end(self, args, state, control, **kwargs):
        if state.global_step >= self.num_steps:
            return {"should_training_stop": True}
        else:
            return {}
```

然后把这个回调函数传给 Trainer 的 callback 参数,代码如下:

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
    callback=[EarlyStoppingCallback()],
)
```

Trainer 可以和 Accelerate 库很好地进行配合。开发人员不需要修改代码,只需通过 Accelerate 的 config 命令来配置分布式框架和参数,然后通过 Accelerate 的 launch 命令来运行训练脚本就可以分布式地训练通过 Trainer 实现的代码。

5.4 预训练 Llama3.1

下边的章节以预训练 Llama3.1 模型为例,讲解从数据准备到模型预训练的全过程。

5.4.1 最简训练代码

在代码变得复杂之前,不考虑数据处理、显存优化、分布式训练等,对于一个大模型预训练最简单的代码如下:

```
//Chapter5/clm_pt_simple.py
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

model_path = r'Your-Model-Path\Meta-Llama-3.1-8B-Instruct'
```

```
tokenizer = AutoTokenizer.from_pretrained(model_path, use_fast=False)
model = AutoModelForCausalLM.from_pretrained(model_path).to("cuda")

optimizer = torch.optim.AdamW(model.parameters())

text = "今天天气不错。"
input = tokenizer(text, return_tensors="pt", add_special_tokens=True)
input = {k: v.to("cuda") for k, v in input.items()}

#设置 labels 和 inputs,使它们一致
input["labels"] = input["input_ids"].clone()

output = model(**input)

#获取模型的 loss
loss = output.loss
loss.backward()
optimizer.step()
optimizer.zero_grad()

#保存模型
model.save_pretrained("output_dir")
```

可以发现,利用 HuggingFace 库来实现对大模型的预训练是很简单的。训练输入文本是“今天天气不错。”。首先对这句话进行分词。因为我们期望模型能根据前边的 token 预测出后边的 token,所以设置 labels,使其等于 inputs,然后传入模型,这样就可以得到损失值。剩下的步骤就和训练一个普通小模型一样了。

因为一般在训练小模型时,损失函数都是自己定义的,那么 Llama 模型内部是如何计算损失的呢?通过查看源码可以发现,它将 labels 向左移一位,去掉输出 logits 的最后一个 token,然后进行交叉熵损失计算。HuggingFace 的 Llama 模型的损失函数源码如下:

```
//transformers/models/llama/modeling_llama.py
...
if labels is not None:
    #Shift so that tokens < n predict n
    shift_logits = logits[..., :-1, :].contiguous()
    shift_labels = labels[..., 1:].contiguous()
    #Flatten the tokens
    loss_fct = CrossEntropyLoss()
    shift_logits = shift_logits.view(-1, self.config.vocab_size)
    shift_labels = shift_labels.view(-1)
    #Enable model parallelism
    shift_labels = shift_labels.to(shift_logits.device)
    loss = loss_fct(shift_logits, shift_labels)
...
```

5.4.2 数据清洗

高质量的训练数据对训练一个好的大模型是至关重要的。需要根据收集数据的情况来制定训练数据的处理策略,例如收集到如下的数据:

```
北京时间 11 月 10 日凌晨,郑钦文在 2024WTA 年终总决赛冠军战中 1-2 不敌高芙,没能拿下首个总决赛冠军。尽管略显遗憾但郑钦文本赛季的表现足够出色,从首次进入大满贯决赛到拿下奥运女单金牌,到如今首进总决赛冠军战,郑钦文已经成为世界女子网坛正在升起的一颗巨星。
<meta http-equiv=Content-Type content="text/html;charset=utf-8">
美国债务总额日前突破 35 万亿美元的史上新高,在让人们进一步担忧华盛顿扭曲的赤字财政政策不可持续的同时,也触发了各方对美国两党新一轮“债务上限”之争的预警。
日経平均株価は今年最大となる一時 2000 円以上の暴落です。終値との比較では過去 2 番目の下げ幅となる可能性も出ています。東証から中継です。
孙颖莎一天双赛稳定发挥都是 4 比 0 横扫“光速”下班。北京时间 8 月 1 日凌晨,巴黎奥运会乒乓球女单 1/8 决赛,孙颖莎 4-0 战胜印度选手阿库拉,顺利晋级女单八强。
孙颖莎一天双赛稳定发挥都是 4 比 0 横扫“光速”下班。北京时间 8 月 1 日凌晨,巴黎奥运会乒乓球女单 1/8 决赛,孙颖莎 4-0 战胜印度选手阿库拉,顺利晋级女单八强。
人民日报
2024-08-02
体育
财经
NEW YORK - In 2006, the hulking office building at 135 W. 50th St. in midtown Manhattan sold for $ 332 million. Tenants occupied nearly every floor; offices were in demand; real estate was booming.
中新网台州 8 月 1 日电(傅飞扬)“我认为草编帽的发展趋势就是不断创新,紧跟世界时尚潮流步伐。企业不能盲目地模仿、复制前沿设计,而是需要建立自己的研发设计团队,提高自身‘硬实力’,将传统工艺和时尚元素相结合,打造出独具特色的时尚产品。”
```

上边的数据中包含了常见的一些数据质量问题。有的文本太短、有的文本重复、包含网页脚本等。另外只想训练中文大语言模型,但是采集的数据内混入了其他语言的数据。首先过滤掉太短的段落,代码如下:

```
//Chapter5/data_clean.py
from datasets import Dataset, load_dataset

dataset = load_dataset("text", data_files="../data/raw_news.txt", split="train")
print(dataset)


def length_filter(x):
    if len(x['text']) < 20:
        return False
    return True


dataset = dataset.filter(length_filter)
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text'],
    num_rows: 12
})
Dataset({
    features: ['text'],
    num_rows: 8
})
```

接下来需要去除重复的段落，代码如下：

```
//Chapter5/data_clean.py

def deduplication(ds):
    def dedup_func(x):
        """移除重复的段落"""
        if x['text'] in unique_text:
            return False
        else:
            unique_text.add(x['text'])
            return True

    unique_text = set()

    ds = ds.filter(dedup_func, load_from_cache_file=False, num_proc=1)
    return ds


dataset = deduplication(dataset)
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text'],
    num_rows: 7
})
```

最后，过滤掉非中文的段落，这里需要使用一个 langdetect 库来进行文本语言识别。如果之前没有安装 langdetect，则安装命令如下：

```
pip install langdetect
```

通过定义一种语言类型的过滤器来对非中文语言的文本进行过滤，代码如下：

```
//Chapter5/data_clean.py

def lang_filter(x):
```

```
    if detect(x["text"]) == "zh-cn":
        return True
    else:
        return False


dataset = dataset.filter(lang_filter)
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text'],
    num_rows: 4
})
```

最终对清理后的数据进行导出，用于大模型的预训练，代码如下：

```
dataset.to_parquet ("../data/clean_data.parquet")
```

最终清洗后的数据如下：

```
北京时间 11 月 10 日凌晨，郑钦文在 2024WTA 年终总决赛冠军战中 1-2 不敌高芙，没能拿下首个总决赛冠军。尽管略显遗憾但郑钦文本赛季的表现足够出色，从首次进入大满贯决赛到拿下奥运女单金牌，到如今首进总决赛冠军战，郑钦文已经成为世界女子网坛正在升起的一颗巨星。
美国债务总额日前突破 35 万亿美元的史上新高，在让人们进一步担忧华盛顿扭曲的赤字财政政策不可持续的同时，也触发了各方对美国两党新一轮“债务上限”之争的预警。
孙颖莎一天双赛稳定发挥都是 4 比 0 横扫“光速”下班。北京时间 8 月 1 日凌晨，巴黎奥运会乒乓球女单 1/8 决赛，孙颖莎 4-0 战胜印度选手阿库拉，顺利晋级女单八强。
中新网台州 8 月 1 日电(傅飞扬)“我认为草编帽的发展趋势就是不断创新，紧跟世界时尚潮流步伐。企业不能盲目地模仿、复制前沿设计，而是需要建立自己的研发设计团队，提高自身‘硬实力’，将传统工艺和时尚元素相结合，打造出独具特色的时尚产品。”
```

5.4.3 数据准备

清洗后的数据还不能直接送入大模型进行训练，还需要经过分词，添加特殊符号和组成一个批次才可以输入大模型进行训练。

首先对原始的段落文本进行分词和添加表示文本开始和结束的特殊 token，代码如下：

```
//Chapter5/data_prepare.py

from transformers import AutoTokenizer
from datasets import load_dataset


model_path = r"Meta-Llama-3.1-8B-Instruct"
```

```
tokenizer = AutoTokenizer.from_pretrained(model_path)
#加载之前清洗好的数据
dataset = load_dataset("parquet", data_files="../data/clean_data.parquet",
split="train")

#定义分词和增加特殊符号的方法
def tokenization(example):
    #分词
    tokens = tokenizer.tokenize(example["text"])

    #转换为 ID
    token_ids = tokenizer.convert_tokens_to_ids(tokens)

    #文章前添加<bos>,后边加<eos>
    token_ids = [tokenizer.bos_token_id]+token_ids+[tokenizer.eos_token_id]
    example["input_ids"] = token_ids


#对所有的样本进行分词和增加特殊符号的操作
dataset = dataset.map(tokenization)
print(dataset)
```

输出如下：

```
Dataset({
    features: ['text', 'input_ids'],
    num_rows: 4
})
```

接下来需要先对多段文字进行拼接，然后按照最长长度进行打包，它的过程如图 5-2 所示。

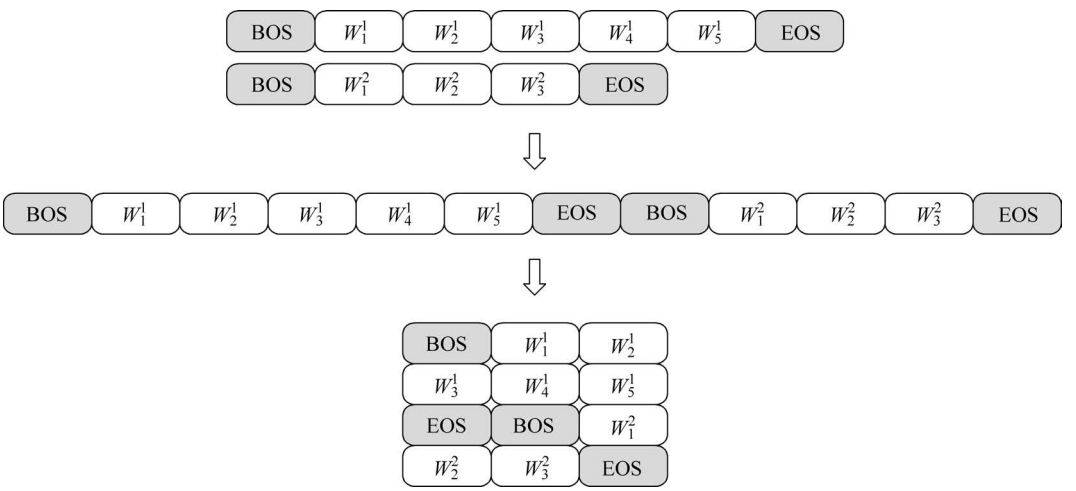

图 5-2　预训练数据拼接

代码如下：

```
//Chapter5/data_prepare.py

import numpy as np

input_ids = np.concatenate(dataset["input_ids"])
print("total length:", len(input_ids))
#定义序列最大长度
max_seq_length = 32
#丢弃最后一行的余数部分
total_length = len(input_ids) - len(input_ids) % max_seq_length
input_ids = input_ids[:total_length]
print("shape", input_ids.shape)

#按照 max_seq_length 将数据打包为 batch 数据
input_ids_reshaped = input_ids.reshape(-1, max_seq_length).astype(np.int32)
print("shape", input_ids_reshaped.shape)

input_ids_list = input_ids_reshaped.tolist()
packaged_pretrain_dataset = datasets.Dataset.from_dict(
    {"input_ids": input_ids_list}
)
print(packaged_pretrain_dataset)
```

输出如下：

```
total length: 352
shape (352,)
shape (11, 32)
Dataset({
    features: ['input_ids'],
    num_rows: 11
})
```

将最终处理好的数据存入文件，代码如下：

```
packaged_pretrain_dataset.to_parquet("../data/pretrain_data.parquet")
```

5.4.4 初始化模型

如果想从头训练一个全新的大模型，则可以通过 LlamaConfig 对象来初始化一个大模型，示例代码如下：

```
from transformers import LlamaConfig
config = LlamaConfig()
print(config)
```

输出如下：

```
LlamaConfig {
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 1,
  "eos_token_id": 2,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 11008,
  "max_position_embeddings": 2048,
  "mlp_bias": false,
  "model_type": "llama",
  "num_attention_heads": 32,
  "num_hidden_layers": 32,
  "num_key_value_heads": 32,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-06,
  "rope_scaling": null,
  "rope_theta": 10000.0,
  "tie_word_embeddings": false,
  "transformers_version": "4.43.1",
  "use_cache": true,
  "vocab_size": 32000
}
```

这是一个默认的配置，可以根据自己的需要来修改，例如通过修改网络结构来减少网络参数，示例代码如下：

```
from transformers import LlamaConfig

config = LlamaConfig()
config.num_hidden_layers = 12
config.hidden_size = 1024
config.intermediate_size = 4096
config.num_key_value_heads = 8
print(config)
```

输出如下：

```
LlamaConfig {
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 1,
  "eos_token_id": 2,
  "hidden_act": "silu",
  "hidden_size": 1024,
  "initializer_range": 0.02,
```

```
  "intermediate_size": 4096,
  "max_position_embeddings": 2048,
  "mlp_bias": false,
  "model_type": "llama",
  "num_attention_heads": 32,
  "num_hidden_layers": 12,
  "num_key_value_heads": 8,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-06,
  "rope_scaling": null,
  "rope_theta": 10000.0,
  "tie_word_embeddings": false,
  "transformers_version": "4.43.1",
  "use_cache": true,
  "vocab_size": 32000
}
```

有了模型的配置,就可以根据配置来创建一个大模型,代码如下:

```
from transformers import LlamaForCausalLM
model = LlamaForCausalLM(config)
print(model)
```

当然这个模型目前的参数都是随机初始化的,它还需要大量文本的训练才可以进行语言生成。在大多数情况下,我们需要从一个开源预训练好的大模型开始,继续在特定领域进行预训练,加载预训练大模型的代码如下:

```
model_path = "your_model_path"
model = AutoModelForCausalLM.from_pretrained(model_path)
tokenizer = AutoTokenizer.from_pretrained(model_path)
```

5.4.5 模型预训练

首先通过自定义 Dataset 来加载之前处理好的数据,将 token id 转换为 PyTorch 对象,代码如下:

```
import datasets
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, args, split="train"):
        """初始化用户自定义 Dataset"""
        self.args = args
        self.dataset = datasets.load_dataset(
            "parquet",
            data_files=args.dataset_name,
```

```
            split=split
        )

    def __len__(self):
        """返回 Dataset 的长度"""
        return len(self.dataset)

    def __getitem__(self, idx):
        """根据指定的 idx,返回特定的样本"""
        #将 token id 的 list 转换为 PyTorch 的 LongTensor
        input_ids = torch.LongTensor(self.dataset[idx]["input_ids"])
        labels = torch.LongTensor(self.dataset[idx]["input_ids"])

        #同时包含 input_ids 和 labels,HuggingFace 模型会输出 loss
        return {"input_ids": input_ids, "labels": labels}
```

使用量化加载 Llama3.1 预训练模型并准备 LoRA 模型,代码如下:

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, TextStreamer,
BitsAndBytesConfig

model_path = r'Your_path_to\Meta-Llama-3.1-8B-Instruct'
#4 位 加载
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

#8 位 加载
#bnb_config = BitsAndBytesConfig(
#load_in_8bit=True
#)

tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModelForCausalLM.from_pretrained(
    model_path,
    quantization_config=bnb_config
)
peft_config = LoraConfig(
        r=8,
        target_modules=["q_proj",
                        "v_proj",
                        "k_proj",
                        "o_proj",
                        "gate_proj",
                        "down_proj",
```

```
                        "up_proj"
                        ],
        task_type=TaskType.CAUSAL_LM,
        lora_alpha=16,
        lora_dropout=0.05
    )
model = get_peft_model(model, peft_config)
```

定义一个训练参数的数据类,用来统一管理所有的训练参数,代码如下:

```
from dataclasses import dataclass, field
import transformers

@dataclass
class CustomArguments(transformers.TrainingArguments):
    #训练数据集
    dataset_name: str = field(default="../data/pretrain_data.parquet")
    #数据处理时的并行进程数
    num_proc: int = field(default=1)
    #最大序列长度
    max_seq_length: int = field(default=32)

    #随机种子
    seed: int = field(default=0)
    #优化器
    optim: str = field(default="adamw_torch")
    #训练轮数
    num_train_epochs: int = field(default=2)
    #每台设备上的批量大小
    per_device_train_batch_size: int = field(default=2)

    #学习率
    learning_rate: float = field(default=5e-5)
    #权重衰减
    weight_decay: float = field(default=0)
    #预热步数
    warmup_steps: int = field(default=10)
    #学习率规划期类型
    lr_scheduler_type: str = field(default="linear")
    #是否使用梯度检查点
    gradient_checkpointing: bool = field(default=False)
    #数据加载并行进程数
    dataloader_num_workers: int = field(default=2)
    #是否使用 BF16 作为混合精度训练类型
    bf16: bool = field(default=True)
    #梯度累加步数
    gradient_accumulation_steps: int = field(default=1)

    #日志记录的步长频率
```

```
    logging_steps: int = field(default=3)
    #checkpoint 保存策略
    save_strategy: str = field(default="steps")
    #checkpoint 保存的步长频率
    save_steps: int = field(default=3)
    #总的保存 checkpoint 的数量
    save_total_limit: int = field(default=2)
```

HfArgumentParser 是来自 HuggingFace Transformers 库的一个实用工具类，用于解析命令行参数并将它们转换为数据类实例。它简化了命令行参数的处理，使将参数传递给脚本更加直观和类型安全，解析代码如下：

```
parser = transformers.HfArgumentParser(CustomArguments)
args, = parser.parse_args_into_dataclasses()
```

这样当从命令行调用训练脚本时，就可以传入参数来修改训练脚本的配置项。下边增加创建数据集和训练器的逻辑，代码如下：

```
train_dataset = CustomDataset(args=args)
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=args,
    train_dataset=train_dataset,
    eval_dataset=None
)

if __name__ == '__main__':
    trainer.train()
```

最后通过命令行传递参数来启动训练脚本，命令如下：

```
python pretrain_model.py --output_dir output_dir --num_train_epochs 10 --per_
device_train_batch_size 8
```

完整的训练代码如下：

```
//Chapter5/pretrain_model.py
import datasets
from peft import TaskType, LoraConfig, get_peft_model
from torch.utils.data import Dataset
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from dataclasses import dataclass, field
import transformers
```

```
class CustomDataset(Dataset):
    def __init__(self, args, split="train"):
        """初始化用户自定义 Dataset"""
        self.args = args
        self.dataset = datasets.load_dataset(
            "parquet",
            data_files=args.dataset_name,
            split=split
        )

    def __len__(self):
        """返回 Dataset 的长度"""
        return len(self.dataset)

    def __getitem__(self, idx):
        """根据指定的 idx,返回特定的样本"""
        #将 token id 的 list 转换为 PyTorch 的 LongTensor
        input_ids = torch.LongTensor(self.dataset[idx]["input_ids"])
        labels = torch.LongTensor(self.dataset[idx]["input_ids"])

        #同时包含 input_ids 和 labels,HuggingFace 模型会输出 loss
        return {"input_ids": input_ids, "labels": labels}


@dataclass
class CustomArguments(transformers.TrainingArguments):
    #训练数据集
    dataset_name: str = field(default="../data/pretrain_data.parquet")
    #数据处理时的并行进程数
    num_proc: int = field(default=1)
    #最大序列长度
    max_seq_length: int = field(default=32)

    #随机种子
    seed: int = field(default=0)
    #优化器
    optim: str = field(default="adamw_torch")
    #训练轮数
    num_train_epochs: int = field(default=2)
    #每台设备上的批量大小
    per_device_train_batch_size: int = field(default=2)

    #学习率
    learning_rate: float = field(default=5e-5)
    #权重衰减
    weight_decay: float = field(default=0)
    #预热步数
    warmup_steps: int = field(default=10)
    #学习率规划期类型
```

```
    lr_scheduler_type: str = field(default="linear")
    #是否使用梯度检查点
    gradient_checkpointing: bool = field(default=False)
    #数据加载并行进程数
    dataloader_num_workers: int = field(default=2)
    #是否使用 BF16 作为混合精度训练类型
    bf16: bool = field(default=True)
    #梯度累加步数
    gradient_accumulation_steps: int = field(default=1)

    #日志记录的步长频率
    logging_steps: int = field(default=3)
    #checkpoint 保存策略
    save_strategy: str = field(default="steps")
    #checkpoint 保存的步长频率
    save_steps: int = field(default=3)
    #总的保存 checkpoint 的数量
    save_total_limit: int = field(default=2)


model_path = r'Your_model_path/Yi-1.5-6B-Chat'
#4 位 加载
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

#8 位 加载
#bnb_config = BitsAndBytesConfig(
#load_in_8bit=True
#)

tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModelForCausalLM.from_pretrained(
    model_path,
    quantization_config=bnb_config
)
peft_config = LoraConfig(
        r=8,
        target_modules=["q_proj",
                        "v_proj",
                        "k_proj",
                        "o_proj",
                        "gate_proj",
                        "down_proj",
                        "up_proj"
                        ],
```

```
        task_type=TaskType.CAUSAL_LM,
        lora_alpha=16,
        lora_dropout=0.05
    )
model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
parser = transformers.HfArgumentParser(CustomArguments)
args, = parser.parse_args_into_dataclasses()

train_dataset = CustomDataset(args=args)
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=args,
    train_dataset=train_dataset,
    eval_dataset=None
)

if __name__ == '__main__':
    trainer.train()
```