微调过程

1.安装微调库

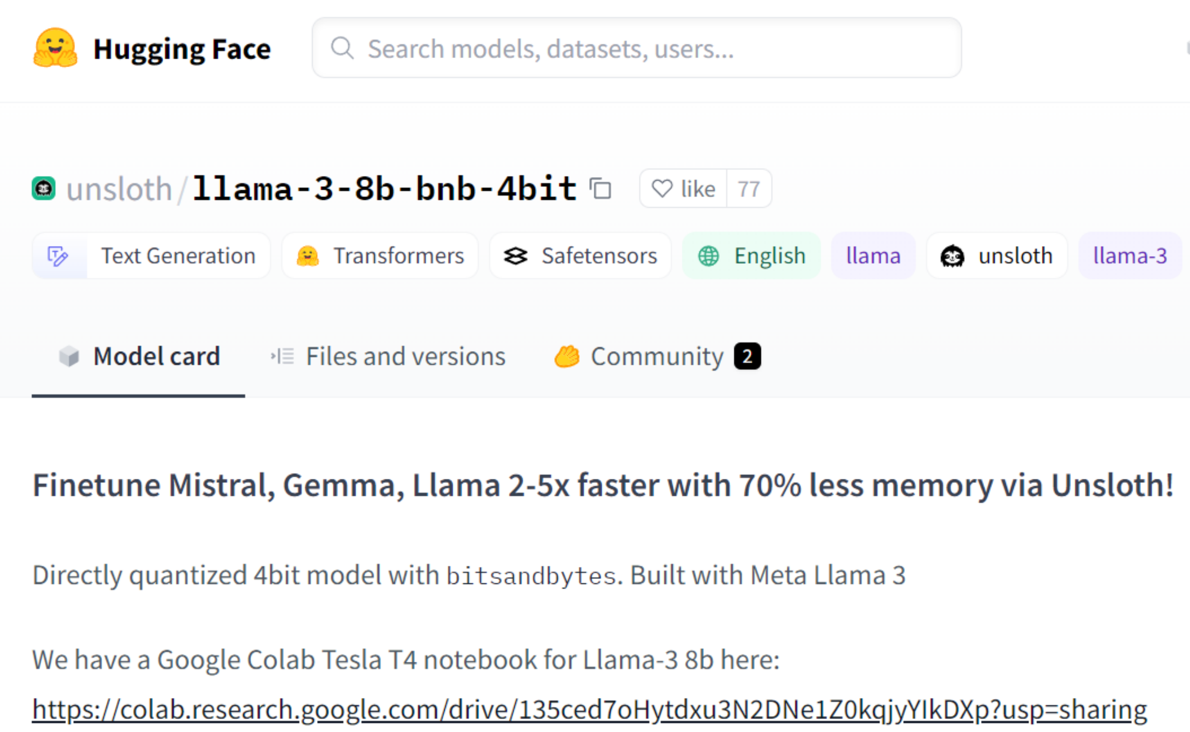
导入了PyTorch库并获取了当前设备的CUDA能力。然后，从GitHub仓库安装了Unsloth库（可以提高大型语言模型微调速度）。安装了一些其他的深度学习库，如`packaging`、`ninja`、`einops`、`flash-attn`、`xformers`、`trl`、`peft`、`accelerate`和`bitsandbytes`等。

```
%%capture
import torch
major_version, minor_version = torch.cuda.get_device_capability()
# 由于Colab有torch 2.2.1，会破坏软件包，要单独安装
!pip install "unsloth[colab-new] @ git+https://github.com/unslothai/unsloth.git"
if major_version >= 8:
    # 新GPU，如Ampere、Hopper GPU（RTX 30xx、RTX 40xx、A100、H100、L40）。
    !pip install --no-deps packaging ninja einops flash-attn xformers trl peft 
accelerate bitsandbytes
else:
    # 较旧的GPU（V100、Tesla T4、RTX 20xx）
    !pip install --no-deps xformers trl peft accelerate bitsandbytes
pass
```

2.加载模型

从Unsloth库中导入了FastLanguageModel类，可以使得微调大型语言模型的速度提高2-5倍，同时内存使用量减少80%。`load_in_4bit = True`：设置模型加载的精度为4位，量化减少模型的内存占用和计算量，提高模型的运行效率；`model_name = "unsloth/llama-3-8b-bnb-4bit"`指定了模型的位置。

模型仓库：



```python
from unsloth import FastLanguageModel
import torch
max_seq_length = 2048
dtype = None
load_in_4bit = True
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/llama-3-8b-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit
)
```

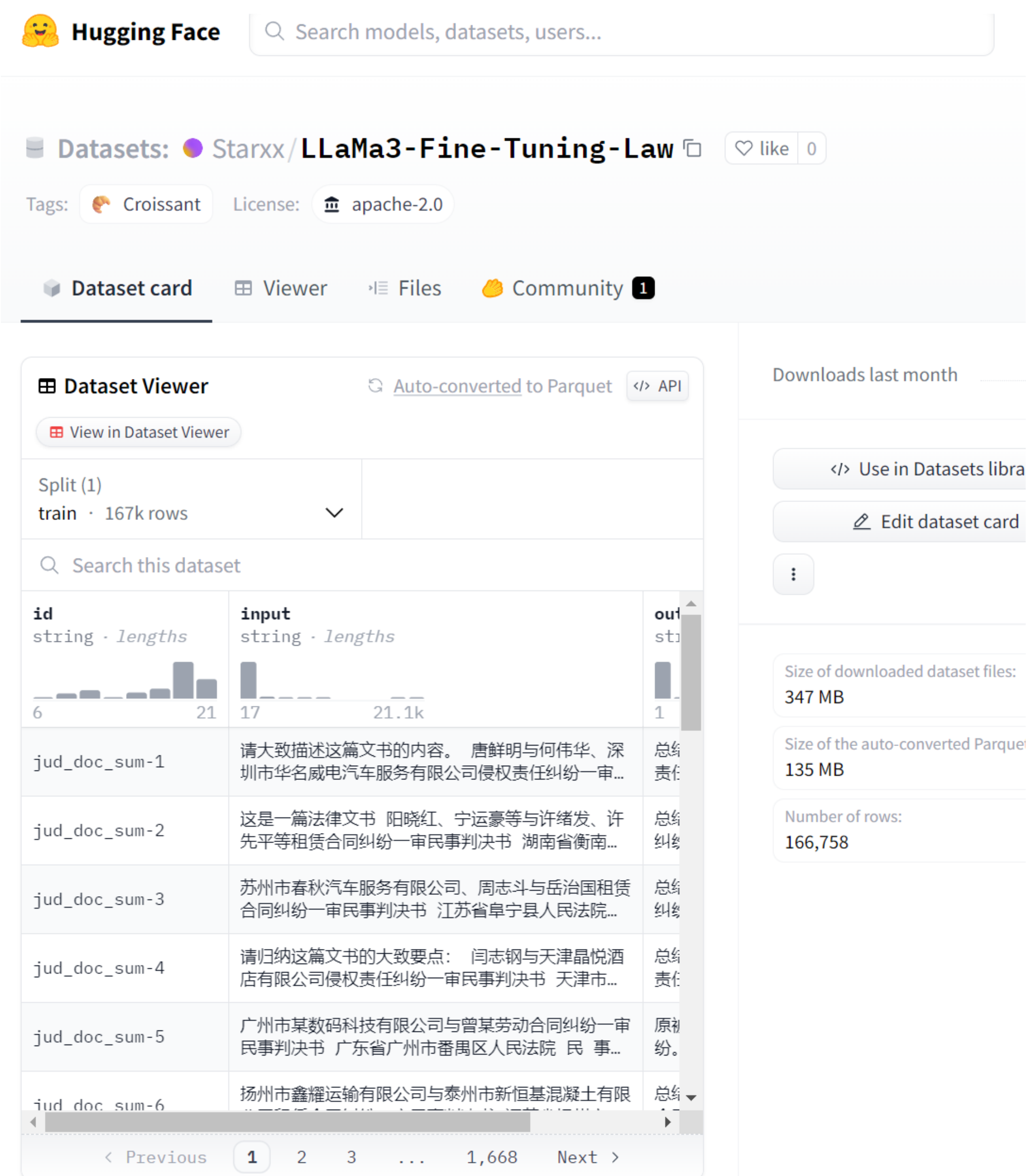
```python
#2加载模型
from unsloth import FastLanguageModel
import torch
max_seq_length = 2048
dtype = None
load_in_4bit = True
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name = "unsloth/llama-3-8b-bnb-4bit",
    max_seq_length = max_seq_length,
    dtype = dtype,
    load_in_4bit = load_in_4bit
)
```

```
config.json: 100%  1.14k/1.14k [00:00<00:00, 28.7kB/s]
==((====))==  Unsloth: Fast Llama patching release 2024.4
   \\   /|    GPU: Tesla T4. Max memory: 14.748 GB. Platform = Linux.
O^O/ \_/ \    Pytorch: 2.2.1+cu121. CUDA = 7.5. CUDA Toolkit = 12.1.
\        /    Bfloat16 = FALSE. Xformers = 0.0.25.post1. FA = False.
 "-____-"     Free Apache license: http://github.com/unslothai/unsloth
Unused kwargs: ['_load_in_4bit', '_load_in_8bit', 'quant_method']. These kwargs are not used ir
model.safetensors: 100%  5.70G/5.70G [00:47<00:00, 178MB/s]
generation_config.json: 100%  131/131 [00:00<00:00, 9.36kB/s]
tokenizer_config.json: 100%  50.6k/50.6k [00:00<00:00, 3.08MB/s]
tokenizer.json: 100%  9.09M/9.09M [00:00<00:00, 24.1MB/s]
special_tokens_map.json: 100%  449/449 [00:00<00:00, 23.5kB/s]
Special tokens have been added in the vocabulary, make sure the associated word embeddings are
Special tokens have been added in the vocabulary, make sure the associated word embeddings are
```

3.准备微调数据集

使用 `format` 函数将 “id” 、 “input” 和 “output” 组合成一个字符串，然后添加一个特殊的结束序列标记来表示序列的结束。使用 `map` 函数将这个函数应用到数据集的每个样本上，函数会遍历数据集中的每个样本，对每个样本调用这个函数，然后将结果保存在新的数据集中。最后得到一个经过预处理的数据集，可以直接用于模型训练。

数据集仓库地址：




```python
EOS_TOKEN = tokenizer.eos_token # 必须添加 EOS_TOKEN
def formatting_prompts_func(examples):
    instructions = examples["id"]
    inputs       = examples["input"]
    outputs      = examples["output"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        # 必须添加EOS_TOKEN，否则无限生成
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return { "text" : texts, }
pass

from datasets import load_dataset
dataset = load_dataset("Starxx/LLaMa3-Fine-Tuning-Law", split = "train")
dataset = dataset.map(formatting_prompts_func, batched = True,)
```

```python
#3准备微调数据集
EOS_TOKEN = tokenizer.eos_token # 必须添加 EOS_TOKEN
def formatting_prompts_func(examples):
    instructions = examples["id"]
    inputs       = examples["input"]
    outputs      = examples["output"]
    texts = []
    for instruction, input, output in zip(instructions, inputs, outputs):
        # 必须添加EOS_TOKEN，否则无限生成
        text = alpaca_prompt.format(instruction, input, output) + EOS_TOKEN
        texts.append(text)
    return { "text" : texts, }
pass

from datasets import load_dataset
dataset = load_dataset("Starxx/LLaMa3-Fine-Tuning-Law", split = "train")
dataset = dataset.map(formatting_prompts_func, batched = True,)
```

```
Downloading readme: 100% 28.0/28.0 [00:00<00:00, 1.47kB/s]
Downloading data: 100% 347M/347M [00:03<00:00, 119MB/s]
Generating train split: 100% 166758/166758 [00:02<00:00, 92093.68 examples/s
Map: 100% 166758/166758 [00:04<00:00, 27432.81 examples/s]
```

4.设置训练参数

1. 导入所需的库和类：`SFTTrainer` 用于训练模型，`TrainingArguments` 用于设置训练参数。
2. 使用 `FastLanguageModel.get_peft_model` 方法初始化模型。这个方法接收多个参数，包括模型的各种设置，如 `r`（推荐值为 8, 16, 32, 64, 128）、`target_modules`（目标模块）等。
3. 初始化 `SFTTrainer` 对象。这个对象负责管理模型的训练过程。它接收多个参数，包括模型、分词器、训练数据集、最大序列长度等。
4. 设置 `TrainingArguments`。这些参数用于控制训练过程，包括每设备的训练批次大小、梯度累积步数、预热步数、最大步数、学习率等。

`SFTTrainer` 对象用来进行模型微调的。它接收一个预训练模型和一组训练数据，然后通过反复的迭代和优化，逐渐调整模型的参数，使其能够更好地适应训练数据。

```python
from trl import SFTTrainer
from transformers import TrainingArguments

model = FastLanguageModel.get_peft_model(
    model,
    r = 16, #  建议 8, 16, 32, 64, 128
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0,
    bias = "none",
    use_gradient_checkpointing = "unsloth", # 检查点，长上下文度
    random_state = 3407,
    use_rslora = False,
    loftq_config = None,
)

trainer = SFTTrainer(
    model = model,
```

```python
    tokenizer = tokenizer,
    train_dataset = dataset,
    dataset_text_field = "text",
    max_seq_length = max_seq_length,
    dataset_num_proc = 2,
    packing = False, # 可以让短序列的训练速度提高5倍。
    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_steps = 5,
        max_steps = 60,  # 微调步数
        learning_rate = 2e-4, # 学习率
        fp16 = not torch.cuda.is_bf16_supported(),
        bf16 = torch.cuda.is_bf16_supported(),
        logging_steps = 1,
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
    ),
)
```

```python
    target_modules = ["q_proj", "k_proj", "v_proj", "o_proj",
                      "gate_proj", "up_proj", "down_proj",],
    lora_alpha = 16,
    lora_dropout = 0,
    bias = "none",
    use_gradient_checkpointing = "unsloth", # 检查点，长上下文度
    random_state = 3407,
    use_rslora = False,
    loftq_config = None,
)

trainer = SFTTrainer(
    model = model,
    tokenizer = tokenizer,
    train_dataset = dataset,
    dataset_text_field = "text",
    max_seq_length = max_seq_length,
    dataset_num_proc = 2,
    packing = False, # 可以让短序列的训练速度提高5倍。
    args = TrainingArguments(
        per_device_train_batch_size = 2,
        gradient_accumulation_steps = 4,
        warmup_steps = 5,
        max_steps = 60,  # 微调步数
        learning_rate = 2e-4, # 学习率
        fp16 = not torch.cuda.is_bf16_supported(),
        bf16 = torch.cuda.is_bf16_supported(),
        logging_steps = 1,
        optim = "adamw_8bit",
        weight_decay = 0.01,
        lr_scheduler_type = "linear",
        seed = 3407,
        output_dir = "outputs",
    ),
)
```

```
Unsloth 2024.4 patched 32 layers with 32 QKV layers, 32 O layers and 32 MLP layers.
/usr/local/lib/python3.10/dist-packages/multiprocess/popen_fork.py:66: RuntimeWarning: os.fork() was called. os.fork()
  self.pid = os.fork()
Map (num_proc=2): 100%  166758/166758 [03:55<00:00, 781.84 examples/s]
max_steps is given, it will override any value given in num_train_epochs
```

5.开始训练

在 60 步的训练过程中，模型的损失值从最初的 2.393900 逐渐下降到最后的 1.390100。这表明模型在学习过程中是在不断改进的。

```
trainer_stats = trainer.train()
```

```
==((====))==  Unsloth - 2x faster free finetuning | Num GPUs = 1
   \\   /|    Num examples = 166,758 | Num Epochs = 1
O^O/ \_/ \    Batch size per device = 2 | Gradient Accumulation steps = 4
\        /    Total batch size = 8 | Total steps = 60
 "-____-"     Number of trainable parameters = 41,943,040
```

[60/60 19:52, Epoch 0/1]

Step	Training Loss
1	2.393900
2	2.036200
3	2.079000
4	1.979000
5	2.168000
6	2.053400
7	1.595600
8	2.057000
9	1.945800
10	1.632500
11	1.499400
12	1.506100
13	1.480700
14	1.714200
15	1.569000
16	1.493200
17	1.680100
18	1.499100
19	1.221500
20	1.828600
21	1.262900
22	1.134500

49	1.156400
50	1.330100
51	1.265000
52	1.181700
53	1.252400
54	1.104800
55	1.318000
56	1.368400
57	1.510800
58	1.722500
59	1.182200
60	1.390100

6.保存LoRA模型

```
model.save_pretrained("lora_model") # Local saving
```