指令微调数据集中的指令是专门为特定领域设计的。例如，法律领域指令集包含法律考试、法律咨询、法律问答等任务的指令数据。

InstructGPT-sft[24] 就是典型的通用指令微调数据集，用于微调 InstructGPT 模型，在构建过程中将指令分为 10 个类别：生成、开放问答、头脑风暴、聊天、重写、总结、分类、其他、封闭问答以及提取。Firefly[230] 则进一步细化了指令类别，涵盖了 23 个类别。包括，故事生成、歌词生成、推理、数学、头脑风暴、封闭问答、开放问答、代码、提取、生成、重写、总结、翻译、角色扮演、社会规范等方面。2023 年以来，针对大模型指令微调所使用的领域数据集也非常多，特别是医疗、法律、教育、数据、编程等方面。本节将按照通用和领域分别进行介绍。

表5.2 给出了部分开源通用指令微调数据集的汇总信息。表5.3 给出了部分开源领域指令微调数据集的汇总信息。更多数据集以及数据集描述可以参考文献 [106]。

表 5.2 部分开源通用指令微调数据集的汇总信息

指令数据集名称	发布单位	指令数据集规模	语言	是否公开
Alpaca Data	Standford Alpaca	5.2 万条	英文	公开
Aya Collection	Cohere For AI 等	5.13 亿条	多语言	公开
Aya Dataset	Cohere For AI 等	20.4 万条	多语言	公开
BELLE	贝壳研究院	350 万条	中文	公开
COIG	北京智源研究院	19.11 万条	中文	公开
DialogStudio	Salesforce AI	87 个数据集	多语言	公开
Dolly	Databricks	1.5 万条	英语	公开
Firefly	YeungNLP	115 万条	中文	公开
Flan 2022	Google Research	1836 个数据集	多语言	部分
InstructionWild V2	新加坡国立大学	11 万条	中英文	公开
LCCC	清华大学	1200 万条	中文	公开
LMSYS-Chat-1M	加州大学伯克利分校	100 万条	多语言	公开
MOSS 003 SFT	复旦大学	107 万条	中英文	公开
OIG	LAION	388 万条	多语言	公开
Phoenix-sft-data-v1	香港中文大学等	46.45 万条	中英文	公开
PromptSource	布朗大学等	176 个数据集	多语言	公开
RedGPT-Dataset-V1-CN	DA-Southampton	5 万条	中文	部分
Self-Instruct	华盛顿大学	5.24 万条	英文	公开
ShareChat	Sharechat	9 万条	英文	公开
ShareGPT-Chinese-English	Sharechat	9 万条	中英文	公开
Super-Natural Instructions	Allen Institute for AI	1616 个数据集	多语言	公开
UltraChat	清华大学	147 万条	中英文	公开
WizardLM_evol_instruct_V2	微软等	14.3 万条	英文	公开

表 5.3 部分开源领域指令微调数据集的汇总信息

指令数据集名称	发布单位	指令数据集规模（条）	领域	是否公开
ChatDoctor	德克萨斯大学西南医学中心	11.5 万	医疗	公开
DISC-Med-SFT	复旦大学	46.49 万	医疗	公开
Huatuo-26M	香港中文大学等	265 万	医疗	公开
MedDialog	加州大学圣地亚哥分校	366 万	医疗	公开
Medical Meadow	亚琛大学医院等	16 万	医疗	公开
BELLE School Math	贝壳研究院	24.85 万	数学	公开
Goat	新加坡国立大学	175 万	数学	公开
OpenMathInstruct-1	NVIDIA	180 万	数学	公开
Code Alpaca 20K	Sahil Chaudhary	2 万	代码	公开
CodeContest	DeepMind	1.36 万	代码	公开
CommitPackFT	Bigcode	70.21 万	代码	公开
DISC-Law-SFT	复旦大学	40.3 万	法律	部分
HanFei 1.0	中国科学研究院	25.5 万	法律	公开
LawGPT	上海交通大学	10 万	法律	公开
Lawyer LLaMA_sft	北京大学	20 万	法律	公开
Child Chat Data	哈尔滨工业大学	5000	教育	公开
DISC-Fin-SFT	复旦大学	24.6 万	金融	部分
Owl-Instruction	北京航空航天大学	1.8 万	IT	公开
TaoLi Data	北京语言大学	8.8 万	教育	公开
TransGPT-SFT	北京交通大学	5.8 万	交通	公开

5.2 高效模型微调

由于大语言模型的参数量十分庞大，当将其应用到下游任务时，微调全部参数需要相当高的算力（全量微调的具体流程将在 5.5 节详细介绍）。为了节省成本，研究人员提出了多种参数高效（Parameter Efficient）的微调方法，旨在仅训练少量参数就使模型适应下游任务。本节将以 LoRA（Low-Rank Adaptation of Large Language Models，大语言模型的低秩适配器）[231] 为例，介绍高效模型微调方法。LoRA 方法可以在缩减训练参数量和 GPU 显存占用的同时，使训练后的模型具有与全量微调相当的性能。

5.2.1 LoRA

文献 [232] 的研究表明，语言模型针对特定任务微调之后，权重矩阵通常具有很低的本征秩（Intrinsic Rank）。研究人员认为，参数更新量即便投影到较小的子空间中，也不会影响学习的有

效性[231]。因此，提出固定预训练模型参数不变，在原本权重矩阵旁路添加低秩矩阵的乘积作为可训练参数，用以模拟参数的变化量。具体来说，假设预训练权重为 $\boldsymbol{W}_0 \in \mathbb{R}^{d\times k}$，可训练参数为 $\Delta\boldsymbol{W} = \boldsymbol{B}\boldsymbol{A}$，其中 $\boldsymbol{B} \in \mathbb{R}^{d\times r}$, $\boldsymbol{A} \in \mathbb{R}^{r\times d}$。初始化时，矩阵 $\boldsymbol{A}$ 通过高斯函数初始化，矩阵 $\boldsymbol{B}$ 为零初始化，使得训练开始之前旁路对原模型不造成影响，即参数变化量为 0。对于该权重的输入 $\boldsymbol{x}$ 来说，输出如下：

$$\boldsymbol{h} = \boldsymbol{W}_0\boldsymbol{x} + \Delta\boldsymbol{W}\boldsymbol{x} = \boldsymbol{W}_0\boldsymbol{x} + \boldsymbol{B}\boldsymbol{A}\boldsymbol{x} \tag{5.7}$$

LoRA 算法结构如图5.6 所示。

除 LoRA 外，也有其他高效微调方法，如微调适配器（Adapter）或前缀微调（Prefix Tuning）。微调适配器分别在 Transformer 层中的自注意力模块与多层感知（Multilayer Perceptron，MLP）模块之间，以及 MLP 模块与残差连接之间添加适配器层（Adapter Layer）作为可训练参数[233]，该方法及其变体会增加网络的深度，从而在模型推理时带来额外的时间开销。当没有使用模型或数据并行时，这种开销会较为明显。而对于使用 LoRA 的模型来说，由于可以将原权重与训练后权重合并，即 $\boldsymbol{W} = \boldsymbol{W}_0 + \boldsymbol{B}\boldsymbol{A}$，因此在推理时不存在额外的开销。前缀微调是指在输入序列前缀添加连续可微的软提示作为可训练参数。由于模型可接受的最大输入长度有限，随着软提示的参数量增多，实际输入序列的最大长度也会相应减小，影响模型性能。这使得前缀微调的模型性能并非随着可训练参数量单调上升。在文献 [231] 的实验中，使用 LoRA 方法训练的 GPT-2、GPT-3 模型在相近数量的可训练参数下，性能均优于或相当于使用上述两种微调方法。

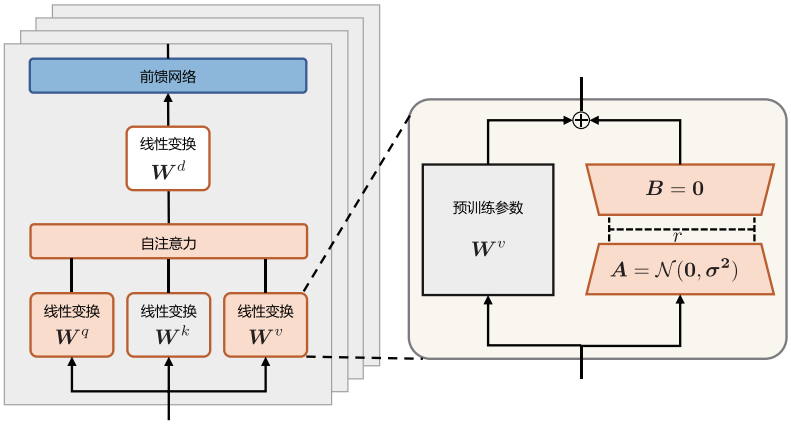

图 5.6　LoRA 算法结构[231]

peft 库中含有包括 LoRA 在内的多种高效微调方法，且与 transformers 库兼容。使用示例如下所示。其中，lora_alpha（α）表示放缩系数。表示参数更新量的 $\Delta\boldsymbol{W}$ 与 α/r 相乘后再与原本的模型参数相加。

```
from transformers import AutoModelForSeq2SeqLM
from peft import get_peft_config, get_peft_model, LoraConfig, TaskType
model_name_or_path = "bigscience/mt0-large"
tokenizer_name_or_path = "bigscience/mt0-large"

peft_config = LoraConfig(
    task_type=TaskType.SEQ_2_SEQ_LM, inference_mode=False, r=8, lora_alpha=32, lora_dropout=0.1
)

model = AutoModelForSeq2SeqLM.from_pretrained(model_name_or_path)
model = get_peft_model(model, peft_config)
```

接下来介绍 peft 库对 LoRA 的实现，也就是上述代码中 get_peft_model 函数的功能。该函数封装了基础模型并得到一个 PeftModel 类的模型。如果使用 LoRA 微调方法，则会得到一个 LoraModel 类的模型。

```
class LoraModel(torch.nn.Module):
    """
    从预训练的Transformer模型创建Lora模型

    Args:
        model ([`~transformers.PreTrainedModel`]): 要适配的模型
        config ([`LoraConfig`]): Lora模型的配置

    Returns:
        `torch.nn.Module`: Lora模型
    **Attributes**:
        - **model** ([`~transformers.PreTrainedModel`]) -- 要适配的模型
        - **peft_config** ([`LoraConfig`]): Lora模型的配置
    """

    def __init__(self, model, config, adapter_name):
        super().__init__()
        self.model = model
        self.forward = self.model.forward
        self.peft_config = config
        self.add_adapter(adapter_name, self.peft_config[adapter_name])

    # Transformer具有`.config`属性，后续假定存在这个属性
    if not hasattr(self, "config"):
            self.config = {"model_type": "custom"}

    def add_adapter(self, adapter_name, config=None):
        if config is not None:
            model_config = getattr(self.model, "config", {"model_type": "custom"})
            if hasattr(model_config, "to_dict"):
                model_config = model_config.to_dict()

            config = self._prepare_lora_config(config, model_config)
            self.peft_config[adapter_name] = config
        self._find_and_replace(adapter_name)
        if len(self.peft_config) > 1 and self.peft_config[adapter_name].bias != "none":
            raise ValueError(
                "LoraModel supports only 1 adapter with bias. When using multiple adapters, \
                set bias to 'none' for all adapters."
            )
        mark_only_lora_as_trainable(self.model, self.peft_config[adapter_name].bias)
        if self.peft_config[adapter_name].inference_mode:
            _freeze_adapter(self.model, adapter_name)
```

LoraModel 类通过 add_adapter 方法添加 LoRA 层。该方法包括 _find_and_replace 和 mark_only_lora_as_trainabl 个主要函数。mark_only_lora_as_trainable 的作用是仅将 Lora 参数设为可训练的，其余参数冻结；

_find_and_replace 会根据 config 中的参数从基础模型的 named_parameters 中找出包含指定名称的模块（默认为“q”“v”，即注意力模块的 $\boldsymbol{Q}$ 和 $\boldsymbol{V}$ 矩阵），创建一个新的自定义类 Linear 模块，并替换原来的。

```
class Linear(nn.Linear, LoraLayer):
    # Lora实现在一个密集层中
    def __init__(
        self,
        adapter_name: str,
        in_features: int,
        out_features: int,
        r: int = 0,
        lora_alpha: int = 1,
        lora_dropout: float = 0.0,
        fan_in_fan_out: bool = False,
        is_target_conv_1d_layer: bool = False,
        **kwargs,
    ):
        init_lora_weights = kwargs.pop("init_lora_weights", True)

        nn.Linear.__init__(self, in_features, out_features, **kwargs)
        LoraLayer.__init__(self, in_features=in_features, out_features=out_features)
        # 冻结预训练的权重矩阵
        self.weight.requires_grad = False

        self.fan_in_fan_out = fan_in_fan_out
        if fan_in_fan_out:
            self.weight.data = self.weight.data.T

        nn.Linear.reset_parameters(self)
        self.update_layer(adapter_name, r, lora_alpha, lora_dropout, init_lora_weights)
        self.active_adapter = adapter_name
        self.is_target_conv_1d_layer = is_target_conv_1d_layer
```

创建 Linear 模块时，会将原本模型的相应权重赋给其中的 nn.Linear 部分。另外的 LoraLayer 部分则是 Lora 层，在 update_adapter 中初始化。Linear 类的 forward 方法完成了对 LoRA 计算逻辑的实现。这里的 self.scaling[self.active_adapter] 即 lora_alpha/r。

```
result += (
self.lora_B[self.active_adapter](
self.lora_A[self.active_adapter(self.lora_dropout[self.active_adapter](x))
    )
    self.scaling[self.active_adapter]
)
```

在文献 [231] 给出的实验中，对于 GPT-3 模型，当 $r = 4$ 且仅在注意力模块的 $\boldsymbol{Q}$ 矩阵和 $\boldsymbol{V}$ 矩阵添加旁路时，保存的检查点大小减小为原来的 1/10000（从原本的 350GB 变为 35MB），训练时 GPU 显存占用从原本的 1.2TB 变为 350GB，训练速度相较全量参数微调提高了 25%。

5.2.2 LoRA 的变体

LoRA 算法不仅在 RoBERTa、DeBERTa、GPT-3 等大语言模型上取得了很好的效果，还应用到了 Stable Diffusion 等视觉大模型中，可以用小成本达到微调大语言模型的目的。LoRA 算法引起了企业界和研究界的广泛关注，研究人员又先后提出了 AdaLoRA[234]、QLoRA[235]、IncreLoRA[236] 及 LoRA-FA[237] 等算法。本节将详细介绍其中的 AdaLoRA 和 QLoRA 两种算法。

1. AdaLoRA

LoRA 算法给所有的低秩矩阵指定了唯一的秩，从而忽略了不同模块、不同层的参数对于微调特定任务的重要性差异。因此，文献 [238] 提出了 AdaLoRA（Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning）算法，在微调过程中根据各权重矩阵对下游任务的重要性动态调整秩的大小，用以进一步减少可训练参数量，同时保持或提高性能。

为了达到降秩且最小化目标矩阵与原矩阵差异的目的，常用的方法是对原矩阵进行奇异值分解并裁去较小的奇异值。然而，对于大语言模型来说，在训练过程中迭代地计算那些高维权重矩阵的奇异值是代价高昂的。因此，AdaLoRA 由对可训练参数 $\Delta \boldsymbol{W}$ 进行奇异值分解，改为令 $\Delta \boldsymbol{W} = \boldsymbol{P}\boldsymbol{\Gamma}\boldsymbol{Q}$（$\boldsymbol{P}$、$\boldsymbol{\Gamma}$、$\boldsymbol{Q}$ 为可训练参数）来近似该操作。其中 $\boldsymbol{\Gamma}$ 为对角矩阵，可用一维向量表示；$\boldsymbol{P}$ 和 $\boldsymbol{Q}$ 应近似为酉矩阵，需在损失函数中添加以下正则化项：

$$R(\boldsymbol{P}, \boldsymbol{Q}) = ||\boldsymbol{P}^\top \boldsymbol{P} - \boldsymbol{I}||_F^2 + ||\boldsymbol{Q}^\top \boldsymbol{Q} - \boldsymbol{I}||_F^2 \tag{5.8}$$

通过梯度回传更新参数，得到权重矩阵及其奇异值分解的近似解，然后为每一组奇异值及其奇异向量 $\{\boldsymbol{P}_{k,*i}, \boldsymbol{\lambda}_{k,i}, \boldsymbol{Q}_{k,i*}\}$ 计算重要性分数 $S_{k,i}^{(t)}$。其中，下标 k 是指该奇异值或奇异向量属于第 k 个权重矩阵，上标 t 指训练轮次为第 t 轮。接下来，根据所有组的重要性分数排序来裁剪权重矩阵以达到降秩的目的。有两种方法定义该矩阵的重要程度。一种方法是直接令重要性分数等于奇异值，另一种方法是用下式计算参数敏感性：

$$I(w_{ij}) = |w_{ij} \bigtriangledown_{w_{ij}} \mathcal{L}| \tag{5.9}$$

其中，w_{ij} 表示可训练参数。该式估计了某个参数变为 0 后，损失函数值的变化。因此，$I(w_{ij})$ 越大，表示模型对该参数越敏感，这个参数也就越应该被保留。然而，根据文献 [239] 中的实验结果，该敏感性度量受限于小批量采样带来的高方差和不确定性，因此并不完全可靠。相应地，文献 [239] 中提出了一种新的方案来平滑化敏感性，以及量化其不确定性。

$$\bar{I}^{(t)}(w_{ij}) = \beta_1 \bar{I}^{(t-1)} + (1-\beta_1) I^{(t)}(w_{ij}) \tag{5.10}$$

$$\bar{U}^{(t)}(w_{ij}) = \beta_2 \bar{U}^{(t-1)} + (1-\beta_2) |I^{(t)}(w_{ij}) - \bar{I}^{(t)}(w_{ij})| \tag{5.11}$$

$$s^{(t)}(w_{ij}) = \bar{I}^{(t)} \bar{U}^{(t)} \tag{5.12}$$

通过实验对上述几种重要性定义方法进行对比，发现由式 (5.11) 计算得到的重要性分数，即平滑后的参数敏感性，效果最优。故最终的重要性分数计算式为

$$S_{k,i} = s(\lambda_{k,i}) + \frac{1}{d_1} \sum_{j=1}^{d_1} s(P_{k,ji}) + \frac{1}{d_2} \sum_{j=1}^{d_2} s(Q_{k,ij}) \tag{5.13}$$

2. QLoRA

QLoRA[235] 并没有对 LoRA 的逻辑做出修改，而是通过将预训练模型量化为 4-bit 节省计算开销。QLoRA 可以将有 650 亿个参数的模型在一块 48GB GPU 上微调并保持原本 16-bit 微调的性能。QLoRA 的主要技术为：

（1）新的数据类型 4-bit NormalFloat（NF4）。

（2）双重量化（Double Quantization）。

（3）分页优化器（Paged Optimizer）。分页优化器指在训练过程中显存不足时自动将优化器状态移至内存，在需要更新优化器状态时再加载回来。

接下来将具体介绍 QLoRA 中的量化过程。

NF4 基于**分位数量化**（Quantile Quantization）构建而成，该量化方法使原数据经量化后，每个量化区间中的值的数量相同。具体做法是先对数据进行排序，然后找出所有数据中每个 k 分位的值，这些值组成了所需的数据类型（Data Type）。对于 4-bit 来说，$k = 2^4 = 16$。然而，该过程的计算代价对于大语言模型的参数来说是不可接受的。考虑到预训练模型参数通常呈均值为 0 的高斯分布，因此可以先对一个标准高斯分布 $N(0,1)$ 按上述方法得到其 4-bit 分位数量化数据类型，并将该数据类型的值缩放至 $[-1,1]$。随后，将参数也缩放至 $[-1,1]$ 即可按通常方法进行量化。该方法存在的一个问题是数据类型中缺少对 0 的表征，而 0 在模型参数中有表示填充、掩码等特殊含义。文献 [235] 中对此做出改进，分别对标准正态分布的非负和非正部分取分位数并取它们的并集，组合成最终的数据类型 NF4。

由于 QLoRA 的量化过程涉及放缩操作，当参数中出现一些离群点时会将其他值压缩在较小

的区间内。因此文献 [235] 中提出分块量化，以减小离群点的影响范围。为了恢复量化后的数据，需要存储每一块数据的放缩系数。如果用 32 位来存储放缩系数，块的大小设为 64，放缩系数的存储将为每一个参数平均带来 $\frac{32}{64} = 0.5$ 比特的额外开销，即 12.5% 的额外显存耗用。因此，需进一步对这些放缩系数进行量化，即双重量化。在 QLoRA 中，每 256 个放缩系数会进行一次 8 比特量化，最终每个参数的额外开销由原本的 0.5 比特变为 $\frac{8}{64} + \frac{32/256}{64} = 0.127$ 比特。

5.3 模型上下文窗口扩展

随着更多长文本建模需求的出现，多轮对话、长文档摘要等任务在实际应用中越来越多，这些任务需要模型能够更好地处理超出常规上下文窗口大小的文本内容。尽管当前的大语言模型在处理短文本方面表现出色，但在支持长文本建模方面仍存在一些挑战，这些挑战包括预定义的上下文窗口大小限制等。以 MetaAI 在 2023 年 2 月开源的 LLaMA 模型[34] 为例，其规定输入文本的词元数量不得超过 2048 个。这会限制模型对长文本的理解和表达能力。当涉及长时间对话或长文档摘要时，传统的上下文窗口大小可能无法捕捉到全局语境，从而导致信息丢失或模糊的建模结果。

为了更好地满足长文本需求，有必要探索如何扩展现有的大语言模型，使其能够有效地处理更大范围的上下文信息。具体来说，扩展语言模型的长文本建模能力主要有以下方法。

- **增加上下文窗口的微调**：采用直接的方式，即通过使用一个更大的上下文窗口来微调现有的预训练 Transformer，以适应长文本建模需求。
- **位置编码**：改进的位置编码，如 ALiBi[240]、LeX[241] 等能够实现一定程度上的长度外推。这意味着它们可以在小的上下文窗口上进行训练，在大的上下文窗口上进行推理。
- **插值法**：将超出上下文窗口的位置编码通过插值法压缩到预训练的上下文窗口中。

文献 [242] 指出，采用增大上下文窗口微调的方式训练的模型，对上下文的适应速度较慢。在经过了超过 10000 个批次的训练后，模型上下文窗口只有小幅度的增长，从 2048 增加到 2560。实验结果显示，这种朴素的方法在扩展到更大的上下文窗口时效率较低。因此，本节中主要介绍改进的位置编码和插值法。

5.3.1 具有外推能力的位置编码

位置编码的长度外推能力来源于位置编码中表征相对位置信息的部分，相对位置信息不同于绝对位置信息，对于训练时的依赖较少。位置编码的研究一直是基于 Transformer 结构模型的重点。2017 年 Transformer 结构[12] 提出时，介绍了两种位置编码，一种是 Naive Learned Position Embedding，也就是 BERT 模型中使用的位置编码；另一种是 Sinusoidal Position Embedding，通过正弦函数为每个位置向量提供一种独特的编码。这两种最初的形式都是绝对位置编码的形式，依赖于训练过程中的上下文窗口大小，在推理时基本不具有外推能力。随后，2021 年提出的 RoPE[48] 在一定程度上缓解了绝对位置编码外推能力弱的问题。关于 RoPE 位置编码的具体细节，已在 2.3.1 节进行了介绍，这里不再赘述。后续在 T5 架构[243] 中，研究人员又提出了 T5 Bias Position Embedding，直

接在 Attention Map 上操作，对于查询和键之间的不同距离，模型会学习一个偏置的标量值，将其加在注意力分数上，并在每一层都进行此操作，从而学习一个相对位置的编码信息。这种相对位置编码的外推性能较好，可以在 512 的训练窗口上外推 600 左右的长度。

ALiBi

受到 T5 Bias 的启发，Press 等人提出了 ALiBi[240] 算法，这是一种预定义的相对位置编码。ALiBi 并不在 Embedding 层添加位置编码，而是在 Softmax 的结果后添加一个静态的不可学习的偏置项：

$$\text{Softmax}\left(\boldsymbol{q}_i\boldsymbol{K}^\top + m\cdot[-(i-1),\cdots,-2,-1,0]\right) \tag{5.14}$$

其中 m 是对不同注意力头设置的斜率值，对于具有 8 个注意力头的模型，斜率定义为几何序列 $\frac{1}{2^1},\frac{1}{2^2},\cdots,\frac{1}{2^8}$，对于具有更多注意力头的模型，如 16 个注意力头的模型，可以使用几何平均对之前的 8 个斜率进行插值，从而变成 $\frac{1}{2^{0.5}},\frac{1}{2^1},\frac{1}{2^{1.5}},\cdots,\frac{1}{2^8}$。通常情况下，对于 n 个注意头，斜率值是从 $2^{\frac{-8}{n}}$ 开始，并使用相同的值作为其比率。ALiBi 的计算过程如图5.7 所示。

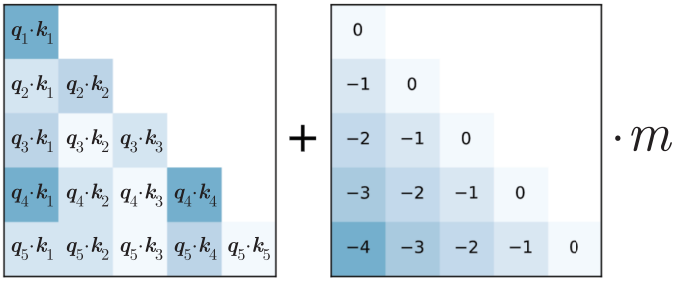

图 5.7 ALiBi 计算过程示例

ALiBi 对最近性具有归纳偏差，它对远程查询–键对之间的注意力分数进行惩罚，随着键和查询之间的距离增加，惩罚也增加。不同的注意力头以不同的速率增加其惩罚，这取决于斜率幅度。实验证明，这组斜率参数适用于各种文本领域和模型尺寸，不需要在新的数据和架构上调整斜率值。

5.3.2 插值法

不同的预训练大语言模型使用不同的位置编码，修改位置编码意味着重新训练，因此对于已训练的模型，通过修改位置编码扩展上下文窗口大小的适用性仍然有限。为了不改变模型架构而直接扩展大语言模型上下文窗口大小，文献 [242] 提出了位置插值法，使现有的预训练大语言模型（包括 LLaMA、Falcon、Baichuan 等）能直接扩展上下文窗口。其关键思想是，直接缩小位置索引，使最大位置索引与预训练阶段的上下文窗口限制相匹配。线性插值法的示意图如图5.8 所示。

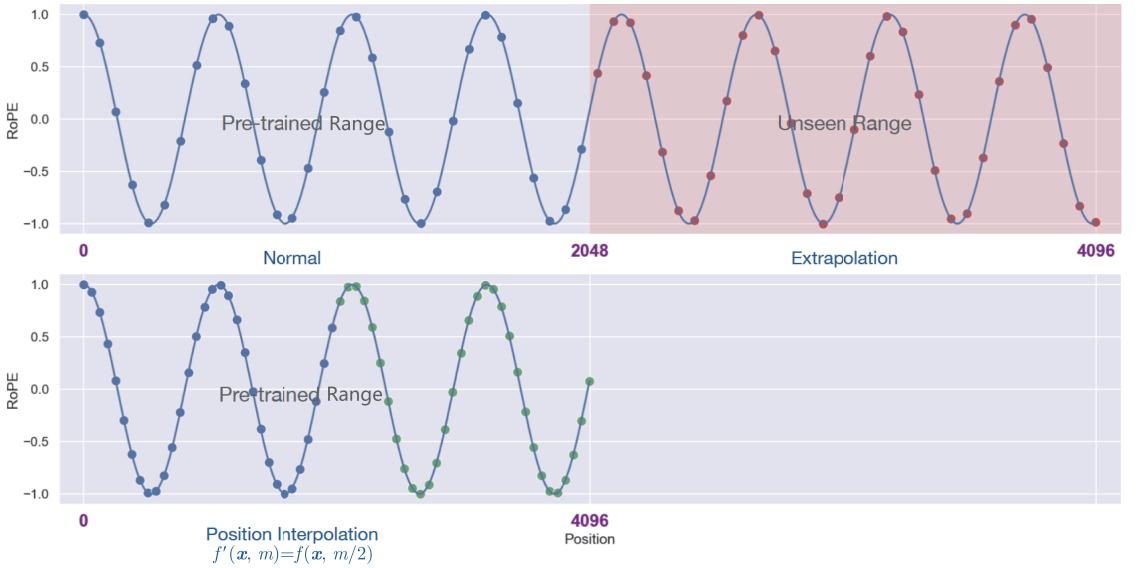

图 5.8 线性插值法的示意图[242]

给定一个位置索引 $m \in [0, c)$ 和一个嵌入向量 $\boldsymbol{x} := [x_0, x_1, \cdots, x_{d-1}]$，其中 d 是注意力头的维度，RoPE 位置编码定义为如下函数：

$$f(\boldsymbol{x}, m) = \left[(x_0 + \mathrm{i}x_1)\mathrm{e}^{\mathrm{i}m\theta_0}, (x_2 + \mathrm{i}x_3)\mathrm{e}^{\mathrm{i}m\theta_1}, \cdots, (x_{d-2} + \mathrm{i}x_{d-1})\mathrm{e}^{\mathrm{i}m\theta_{d/2-1}}\right]^\top \tag{5.15}$$

其中，$\mathrm{i} := \sqrt{-1}$ 是虚数单位，$\theta_j = 10000^{-2j/d}$。虽然 RoPE 位置编码所得的注意力分数只依赖于相对位置，但是其外推能力并不理想，当直接扩展上下文窗口时，模型的困惑度会飙升。具体来说，RoPE 应用于注意力分数可以得到以下结果：

$$\begin{aligned} a(m, n) &= \mathrm{Re}\langle f(\boldsymbol{q}, m), f(\boldsymbol{k}, m)\rangle \\ &= \sum_{j=0}^{d/2-1} (q_{2j} + \mathrm{i}q_{2j+1})(k_{2j} - \mathrm{i}k_{2j+1})\cos((m-n)\theta_j) \\ &\quad + (q_{2j} + \mathrm{i}q_{2j+1})(k_{2j} - \mathrm{i}k_{2j+1})\sin((m-n)\theta_j) \\ &= a(m-n) \end{aligned} \tag{5.16}$$

将所有三角函数视为基函数 $\phi_j(s) := \mathrm{e}^{\mathrm{i}s\theta_j}$，可以将式 (5.16) 展开为

$$a(s) = \mathrm{Re}\left[\sum_{j=0}^{d/2-1} h_j \mathrm{e}^{\mathrm{i}s\theta_j}\right] \tag{5.17}$$

其中 s 是查询和键之间的相对距离，$h_j := (q_{2j} + \mathrm{i}q_{2j+1})(k_{2j} - \mathrm{i}k_{2j+1})$ 是取决于查询和键的复系数。作为基函数的三角函数具有非常强的拟合能力，基本上可以拟合任何函数，因此在不训练的情况下，对于预训练 2048 的上下文窗口总会存在与 $[0, 2048]$ 中的小函数值相对应但在 $[0, 2048]$ 之外的区域中大很多的系数 h_j（键和查询），如图5.9(a) 所示，但线性插值法得到的结果平滑且数值稳定，如图5.9(b) 所示。

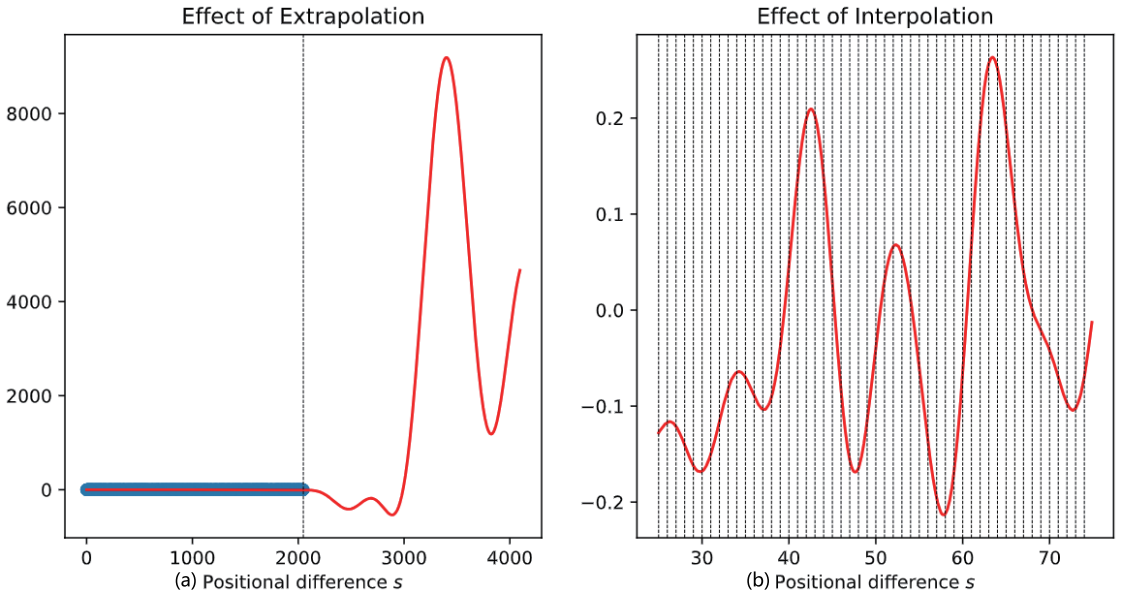

图 5.9　不同相对距离下外推法和线性插值法的注意力分数比较。

因此，可以利用位置插值修改式 (5.15) 的位置编码函数：

$$f'(\boldsymbol{x}, m) = f\left(\boldsymbol{x}, \frac{mL}{L'}\right) \tag{5.18}$$

这种方法对齐了位置索引和相对距离的范围，减小了上下文窗口扩展对注意力得分计算的影响，使得模型更容易适应。线性插值法具有良好的数值稳定性（具体推导请参考文献 [242]），并且不需要修改模型架构，只需要少量微调（例如，在 pile 数据集上进行 1000 步的微调）即可将 LLaMA

的上下文窗口扩展到 32768。

位置插值通过小代价的微调显著扩展 LLaMA 模型的上下文窗口，在保持原有扩展模型内任务能力的基础上，显著增加模型对长文本的建模能力。另外，通过位置插值扩展的模型可以充分重用现有的预训练大语言模型和优化方法，这在实际应用中具有很大吸引力。

5.4 DeepSpeed-Chat SFT 实践

ChatGPT 整体的训练过程复杂，虽然基于 DeepSpeed 可以通过单机多卡、多机多卡、流水线并行等操作来训练和微调大语言模型，但是没有端到端的基于人类反馈机制的强化学习的规模化系统，仍然会造成训练类 ChatGPT 系统非常困难。DeepSpeed-Chat[244] 是微软于 2023 年 4 月发布的基于 DeepSpeed 用于训练类 ChatGPT 模型的开发工具。基于 DeepSpeed-Chat 训练类 ChatGPT 对话模型的步骤框架如图5.10 所示，包含以下三个步骤。

（1）指令微调：使用精选的人类回答来微调预训练语言模型以应对各种查询。

（2）奖励模型微调：使用一个包含人类对同一查询的多个答案打分的数据集来训练一个独立的奖励模型。

（3）基于人类反馈的强化学习（Reinforcement Learning from Human Feedback，RLHF）训练：利用**近端策略优化**（Proximal Policy Optimization，PPO）算法，根据奖励模型的奖励反馈进一步微调 SFT 模型。

本节只针对步骤（1）指令微调的实践进行介绍，对于奖励模型微调和 RLHF 训练的实践会在后续对应章节中详细介绍。

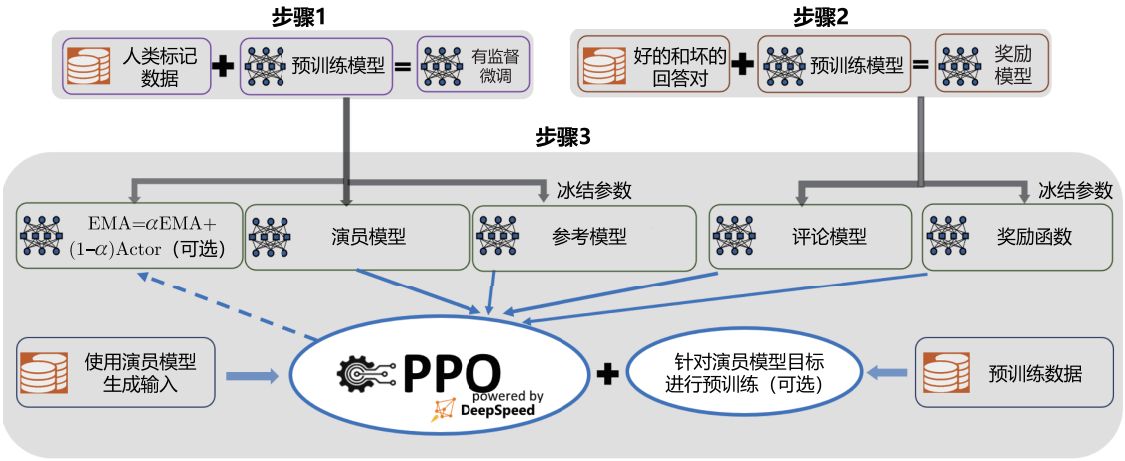

图 5.10　基于 DeepSpeed-Chat 训练类 ChatGPT 对话模型的三个步骤[244]

DeepSpeed-Chat 具有以下三大核心功能。

（1）易用的类 ChatGPT 模型的训练和强化推理：只需要一个脚本就可以实现多个训练步骤，包括使用 HuggingFace 预训练的模型，使用 InstructGPT 训练的所有三个步骤构建类 ChatGPT 模型。此外，还提供了一个易于使用的推理 API，用于用户在模型训练后进行对话式交互性测试。

（2）DeepSpeed-RLHF 管道：DeepSpeed-RLHF 复现了 InstructGPT[24] 论文中的训练模式，包括指令微调、奖励模型微调和基于人类反馈的强化学习三个步骤。此外，还提供了数据抽象和混合功能，以支持用户使用多个不同来源的数据源进行训练。

（3）DeepSpeed-RLHF 系统：将 DeepSpeed 的训练能力（Training Engine）和推理能力（Inference Engine）整合到统一的混合引擎（DeepSpeed Hybrid Engine，DeepSpeed-HE）中用于 RLHF 训练。DeepSpeed-HE 能够无缝地在推理和训练模式之间切换，使其能够利用来自 DeepSpeed-Inference 的各种优化。

5.4.1 代码结构

DeepSpeed-Chat 代码仓库位于微软官方 GitHub 仓库 DeepSpeedExamples/applications/DeepSpeed-Chat 路径下。在进行实际应用前，需要先对官方代码有一个全局的了解。DeepSpeed-Chat 代码的结构如下所示：

```
- DeepSpeed-Chat
  - inference                        # 模型测试、推理
  - training                         # 训练脚本
    - step1_supervised_finetuning    # 步骤一，指令微调
      - training_log_output          # 训练日志输出
      - evaluation_scripts           # 监督微调模型评测
      - training_scripts             # 模型训练脚本
      - main.py                      # 训练脚本
      - prompt_eval.py               # 评测脚本
      - README.md                    # 说明文档
    - step2_reward_model_finetuning # 步骤二，奖励模型微调
      - 省略
    - step3_rlhf_finetuning          # 步骤三，RLHF训练
      - 省略
    - utils                          # 模型训练与评价的相关函数库
      - data                         # 数据处理相关代码
      - model                        # 模型相关文件
      - module                       # 其他组件
      - ds_utils.py                  # DeepSpeed配置相关
      - utils.py                     # 其他相关函数
  - train.py                         # 三步骤集成训练入口
```

当需要完整微调一个模型时（包含所有步骤），可以直接运行 `train.py` 程序。训练中主要调整如下参数。

- `--step` 训练步骤参数，表示运行哪个步骤，可选参数为 1、2、3。本节介绍的内容只使用步骤一，指令微调。
- `--deployment-type` 表示分布式训练模型的参数，分别为单卡 single_gpu、单机多卡 single_node 和多机多卡 multi_node。
- `--actor-model` 表示要训练的模型，默认参数为训练 OPT 的 `"1.3b"`、`"6.7b"`、`"13b"`、`"66b"` 等各个参数量的模型。
- `--reward-model` 表示要训练的奖励模型，默认参数为 OPT 的 `"350m"` 参数量的模型。
- `--actor-zero-stage` 表示指令微调的 DeepSpeed 分布式训练配置。
- `--reward-zero-stage` 表示训练奖励的 DeepSpeed 分布式训练配置。
- `--output-dir` 表示训练过程和结果的输出路径。

在实践中，可以直接在代码根目录下输入命令`python3 train.py --step 1 2 --actor-model 1.3b --reward-model 350m`，表示通过 `train.py` 脚本进行步骤一和步骤二的训练，分别对 OPT-1.3b 模型进行监督微调和对 OPT-350m 模型进行奖励模型的训练。

当训练开始时，第一次运行会先下载 OPT-1.3b 模型和相应的数据集。

```
[2023-09-06 21:17:36,034] [INFO] [real_accelerator.py:110:get_accelerator] Setting ds_accelerator
to cuda (auto detect)
Detected CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 but ignoring it because one or several of --include/
--exclude/--num_gpus/--num_nodes cl args were used. If you want to use CUDA_VISIBLE_DEVICES don't
pass any of these arguments to deepspeed.
[2023-09-06 21:17:37,575] [INFO] [runner.py:555:main] cmd = /opt/miniconda3/envs/baichuan_sft/bin/
python -u -m deepspeed.launcher.launch --world_info=eyJsb2NhbGhvc3QiOiBbMF19 --master_addr=127.0.0.
1 --master_port=29500 --enable_each_rank_log=None main.py --model_name_or_path facebook/opt-1.3b
--gradient_accumulation_steps 8 --lora_dim 128 --zero_stage 0 --enable_tensorboard --tensorboard_pa
th /root/workspace/DeepSpeed-Chat/output/actor-models/1.3b --deepspeed --output_dir /root/workspace
/DeepSpeed-Chat/output/actor-models/1.3b
[2023-09-06 21:17:38,322] [INFO] [real_accelerator.py:110:get_accelerator] Setting ds_accelerator
to cuda (auto detect)
[2023-09-06 21:17:39,762] [INFO] [launch.py:145:main] WORLD INFO DICT: {'localhost': [0]}
[2023-09-06 21:17:39,762] [INFO] [launch.py:151:main] nnodes=1, num_local_procs=1, node_rank=0
[2023-09-06 21:17:39,762] [INFO] [launch.py:162:main] global_rank_mapping=defaultdict(<class 'list
'>,{'localhost': [0]})
[2023-09-06 21:17:39,762] [INFO] [launch.py:163:main] dist_world_size=1
[2023-09-06 21:17:39,762] [INFO] [launch.py:165:main] Setting CUDA_VISIBLE_DEVICES=0
[2023-09-06 21:17:41,099] [INFO] [real_accelerator.py:110:get_accelerator] Setting ds_accelerator
to cuda (auto detect)
[2023-09-06 21:17:43,194] [WARNING] [comm.py:152:init_deepspeed_backend] NCCL backend in DeepSpeed
not yet implemented
[2023-09-06 21:17:43,194] [INFO] [comm.py:594:init_distributed] cdb=None
[2023-09-06 21:17:43,194] [INFO] [comm.py:625:init_distributed] Initializing TorchBackend in
DeepSpeed with backend nccl

Downloading pytorch_model.bin:   0%|          | 0.00/2.63G [00:00<?, ?B/s]
Downloading pytorch_model.bin:   0%|          | 10.5M/2.63G [00:01<07:23, 5.91MB/s]
Downloading pytorch_model.bin:   1%|          | 21.0M/2.63G [00:02<04:38, 9.39MB/s]
Downloading pytorch_model.bin:   1%|          | 31.5M/2.63G [00:03<03:44, 11.6MB/s]
Downloading pytorch_model.bin:   2%|          | 41.9M/2.63G [00:03<03:18, 13.0MB/s]
...
Downloading pytorch_model.bin:  99%|      | 2.60G/2.63G [02:47<00:02, 14.9MB/s]
Downloading pytorch_model.bin:  99%|      | 2.61G/2.63G [02:48<00:01, 15.3MB/s]
Downloading pytorch_model.bin: 100%|      | 2.62G/2.63G [02:49<00:00, 15.6MB/s]
Downloading pytorch_model.bin: 100%|      | 2.63G/2.63G [02:49<00:00, 15.8MB/s]
Downloading pytorch_model.bin: 100%|      | 2.63G/2.63G [02:49<00:00, 15.5MB/s]

Downloading (…)neration_config.json:   0%|          | 0.00/137 [00:00<?, ?B/s]
Downloading (…)neration_config.json: 100%|      | 137/137 [00:00<00:00, 37.9kB/s]
Downloading readme: 100%|      | 530/530 [00:00<00:00, 2.48MB/s]
Downloading and preparing dataset None/None to /root/.cache/huggingface/datasets/Dahoas___parquet/
default-b9d2c4937d617106/0.0.0/14a00e99c0d15a23649d0db8944380ac81082d4b021f398733dd84f3a6c569a7...
Downloading data files:   0%|          | 0/2 [00:00<?, ?it/s]
Downloading data:   0%|          | 0.00/68.4M [00:00<?, ?B/s]
...
Downloading data: 100%|      | 68.4M/68.4M [00:05<00:00, 13.0MB/s]
```

此外，还可以只对模型进行指令微调。例如，通过路径 `training/ step1_supervised_finetuning/training_scripts/llama2/run_llama2_7b.sh`启动对应的脚本可以微调LLaMA-2 7B模型，脚本通过运行 `training/step1_supervised_finetuning/main.py` 启动训练。

5.4.2 数据预处理

训练一个属于自己的大语言模型，数据是非常重要的。通常，使用相关任务的数据进行优化的模型会在目标任务上表现得更好。在 DeepSpeed-Chat 中使用新的数据，需要进行如下操作。

（1）准备数据，并把数据处理成程序能够读取的格式，如 JSON、arrow。

（2）在数据处理代码文件 `training/utils/data/raw_datasets.py` 和 `training/utils/data/data_utils.py` 中增加对新增数据的处理。

（3）在训练脚本中增加对新增数据的支持，并开始模型训练。

在指令微调中，每条样本都有对应的 prompt 和 chosen（奖励模型微调中还有 rejected）。因此，需要将新增的数据处理成如下格式（JSON）：

```
[
  {
    "prompt": " 你是谁？",
    "chosen": " 我是你的私人小助手。",
    "rejected": "",
  },
  {
    "prompt": " 讲个笑话",
    "chosen": " 为什么有脚气的人不能吃香蕉？因为他们会变成香蕉脚！",
    "rejected": ""
  }
]
```

基于构建的数据，在 `raw_datasets.py` 和 `data_utils.py` 中增加对该数据的处理。

在 `raw_datasets.py` 中新增如下代码，其中 `load(dataset_name)` 为数据加载。

```
# 自定义load函数
def my_load(filepath):
    with open(filepath, 'r') as fp:
        data = json.load(fp)
    return data

# raw_datasets.py
class MyDataset(PromptRawDataset):
    def __init__(self, output_path, seed, local_rank, dataset_name):
        super().__init__(args, output_path, seed, local_rank, dataset_name)
        self.dataset_name = "MyDataset"
        # 加载数据集，其中load函数使用自定义的加载函数my_load()
        self.raw_datasets = my_load(dataset_name)

    # 获取训练数据
    def get_train_data(self):
        return self.raw_datasets["train"]

    # 获取验证数据
    def get_eval_data(self):
        return self.raw_datasets["eval"]

    # 得到一个样本的prompt
    def get_prompt(self, sample):
        return " Human: " + sample['prompt']

    # 得到一个样本的正例回答
    def get_chosen(self, sample):
        return " Assistant" + sample['chosen']

    # 得到一个样本的反例回答（在这里只进行步骤一的实践介绍，因此反例样本并不会被调用）
    def get_rejected(self, sample):
    return " Assistant: " + sample['rejected']

    # 得到一个样本的prompt和正例回答
    def get_prompt_and_chosen(self, sample):
        return " Human: " + sample['prompt'] + " Assistant: " sample['chosen']

    # 得到一个样本的prompt和反例回答
    def get_prompt_and_rejected(self, sample):
        return " Human: " + sample['prompt'] + " Assistant: " + sample['rejected']
```

```
# data_utils.py
def get_raw_dataset(dataset_name, output_path, seed, local_rank):

    # 加入之前构建的自定义数据类
    if "MyDataset" in dataset_name:
        return raw_datasets.MyDataset(output_path, seed,
                                                local_rank, dataset_name)
    elif "Dahoas/rm-static" in dataset_name:
        return raw_datasets.DahoasRmstaticDataset(output_path, seed,
                                                local_rank, dataset_name)
    elif "Dahoas/full-hh-rlhf" in dataset_name:
        return raw_datasets.DahoasFullhhrlhfDataset(output_path, seed,
                                                  local_rank, dataset_name)
```

数据处理完成后，读取到的数据格式如下：

```
# 原始样本
{
    "prompt": " 讲个笑话",
    "chosen": " 为什么有脚气的人不能吃香蕉？因为他们会变成香蕉脚！",
    "rejected": ""
}

# 调用my_dataset.get_prompt(sample)
Human: 讲个笑话

# 调用my_dataset.get_chosen(sample)
Human: 讲个笑话 Assistant: 为什么有脚气的人不能吃香蕉？因为他们会变成香蕉脚！
```

5.4.3 自定义模型

虽然 DeepSpeed-Chat 内置了在各项评估上都表现良好的 LLaMA-2 7B 模型，但是模型在预训练中并没有在足够的中文数据上训练，导致其中文能力并不强。当需要使用支持中文的预训练模型，或者更换其他模型时，就需要对 DeepSpeed-Chat 进行相应的更改来适配其他自定义的模型。

DeepSpeed-Chat 训练中默认使用的是基于 HuggingFace 格式的模型和数据，因此切换到 Transformer 和 HuggingFace 支持的模型非常简单，只需将 `model_name_or_path` 参数修改为要使用的模型即可。对于其他暂未支持的模型而言，则需要在代码层面做相应的修改。以下为基于百川智能发布的中文大语言模型 Baichuan 7B 进行自定义模型修改的具体过程。

首先进行模型结构相关的修改，在步骤一的 `main.py` 中进行如下修改来导入相应的类：

```
# main.py
# 导入本地存储的模型相关文件
modeling_baichuan = import_module("models.Baichuan-7B.modeling_baichuan")
tokenization_baichuan = import_module("models.Baichuan-7B.tokenization_baichuan")
# 获取Baichuan模型相关的类
BaiChuanForCausalLM = getattr(modeling_baichuan, "BaiChuanForCausalLM")
BaiChuanTokenizer = getattr(tokenization_baichuan, "BaiChuanTokenizer")
```

对模型代码文件路径做相应的修改，改为本地存储模型代码的路径。然后，同样在 `main.py` 中对对应的模型加载进行修改：

```
# main.py
# 原始代码
tokenizer = load_hf_tokenizer(args.model_name_or_path, fast_tokenizer=True)
model = create_hf_model(AutoModelForCausalLM,
                        args.model_name_or_path,
                        tokenizer,
                        ds_config,
                        disable_dropout=args.disable_dropout)

# 修改为支持Baichuan 7B的代码
tokenizer = BaiChuanTokenizer.from_pretrained(args.model_name_or_path)
model = create_hf_model(BaiChuanForCausalLM,
                        args.model_name_or_path,
                        tokenizer,
                        ds_config,
                        disable_dropout=args.disable_dropout)
```

最后，在训练脚本中将 `model_name_or_path` 参数修改为 Baichuan 7B 的模型路径即可开始模型的训练。训练脚本中以 DeepSpeed-Chat 中的 `run_llama2_7b.sh` 为模板进行修改：

```
# run_baichuan_7b.sh
#!/bin/bash
# Copyright (c) Microsoft Corporation.
# SPDX-License-Identifier: Apache-2.0

# DeepSpeed Team
OUTPUT=$1
ZERO_STAGE=$2
if [ "$OUTPUT" == "" ]; then
    OUTPUT=./output_step1_baichuan_7b
fi
if [ "$ZERO_STAGE" == "" ]; then
    ZERO_STAGE=3
fi
mkdir -p $OUTPUT

deepspeed main.py \
   --data_path <my_data>/my_dataset \ # 数据路径修改为本地的数据
   --data_split 10,0,0 \ # 由于只进行步骤一指令微调，因此不对数据进行切分，全部用于步骤一的训练
   --model_name_or_path <my_model>/baichuan_7b \ # 模型修改为本地存储的baichuan 7B模型路径
   --per_device_train_batch_size 4 \
   --per_device_eval_batch_size 4 \
   --max_seq_len 512 \
   --learning_rate 9.65e-6 \
   --weight_decay 0. \
   --num_train_epochs 2 \
   --gradient_accumulation_steps 1 \
   --lr_scheduler_type cosine \
   --num_warmup_steps 0 \
   --seed 1234 \
   --gradient_checkpointing \
   --zero_stage $ZERO_STAGE \
   --deepspeed \
   --output_dir $OUTPUT \
   &> $OUTPUT/training.log
```

5.4.4 模型训练

数据预处理和自定义模型的修改都完成后，就可以正式进行训练了。进入步骤一指令微调的路径 `training/step1_supervised_finetuning` 下，把上述构造的训练脚本放置到 `training/`

step1_supervised_finetuning/training_scripts/baichuan/run_baichuan_7b.sh，在命令行下可以运行以下代码启动训练：

```
# 在路径training/step1_supervised_finetuning下运行，示例中在一台8块NVIDIA A100机器下进行训练
CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 bash training_scripts/baichuan/run_baichuan_7b.sh
```

训练进行时会进行一次评估，计算**困惑度**（Perplexity，PPL）。然后继续训练，在每一轮训练结束后都会进行一次评估，PPL 也会随着训练的进行逐步下降。训练的过程如下：

```
[2023-09-07 10:31:52,575] [INFO] [real_accelerator.py:110:get_accelerator] Setting ds_accelerator
to cuda (auto detect)
[2023-09-07 10:31:57,019] [WARNING] [runner.py:196:fetch_hostfile] Unable to find hostfile, will
proceed with training with local resources only.
Detected CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7: setting --include=localhost:0,1,2,3,4,5,6,7
...
running - ***** Running training *****
running - ***** Evaluating perplexity, Epoch 0/2 *****
running - ppl: 6.88722562789917
running - Beginning of Epoch 1/2, Total Micro Batches 341
running - Rank: 0, Epoch 1/2, Step 1/341, trained samples: 128/341, Loss 1.916015625
running - Rank: 3, Epoch 1/2, Step 1/341, trained samples: 128/341, Loss 1.6083984375
running - Rank: 2, Epoch 1/2, Step 1/341, trained samples: 128/341, Loss 1.7587890625
running - Rank: 5, Epoch 1/2, Step 1/341, trained samples: 128/341, Loss 1.658203125
running - Rank: 4, Epoch 1/2, Step 1/341, trained samples: 128/341, Loss 1.6396484375
running - Rank: 6, Epoch 1/2, Step 1/341, trained samples: 128/341, Loss 1.94140625
...
running - Rank: 4, Epoch 1/2, Step 341/341, trained samples: 43584/341, Loss 2.005859375
running - Rank: 5, Epoch 1/2, Step 341/341, trained samples: 43584/341, Loss 1.6533203125
running - ***** Evaluating perplexity, Epoch 1/2 *****
running - Rank: 7, Epoch 1/2, Step 341/341, trained samples: 43584/341, Loss 2.076171875
running - ppl: 6.158349514007568
running - Beginning of Epoch 2/2, Total Micro Batches 341
running - Rank: 0, Epoch 2/2, Step 1/341, trained samples: 128/341, Loss 1.7919921875
running - Rank: 2, Epoch 2/2, Step 341/341, trained samples: 43584/341, Loss 1.291015625
running - ***** Evaluating perplexity, Epoch 2/2 *****
running - Rank: 5, Epoch 2/2, Step 341/341, trained samples: 43584/341, Loss 1.4794921875
running - Rank: 6, Epoch 2/2, Step 341/341, trained samples: 43584/341, Loss 2.017578125
running - Rank: 7, Epoch 2/2, Step 341/341, trained samples: 43584/341, Loss 1.748046875
running - ppl: 4.902741432189941
...
[2023-09-07 11:59:56,032] [INFO] [launch.py:347:main] Process 23957 exits successfully.
```

5.4.5 模型推理

模型训练完成后，可以使用 DeepSpeed-Chat 根路径下的 chat.py 进行推理。参数修改为已训练好的模型路径，具体执行方式如下：