第3章

大模型微调

CHAPTER 3

通用预训练大模型虽然已经拥有大量知识，但其训练数据来自通用领域，难以直接适用于垂直领域。通过调整其参数，可以使它在保留通用知识和理解能力的同时，通过特定任务的数据驱动参数优化，精准适应垂直领域的任务需求。

本章介绍大模型微调的概念以及技术发展历程，分析大模型微调的核心理论，说明如何开展数据准备，给出全参数微调、部分参数微调、新增参数微调的方法和相应代码示例，阐述大模型微调策略。

3.1　大模型微调基础知识

3.1.1　大模型微调的基本概念

预训练大模型虽然具有强大的泛化能力，但其性能通常在特定任务中无法达到很好的效果。通过在特定任务数据上进行进一步训练，模型可以学习到与任务相关的细微特征和领域知识，从而显著提升其在该任务上的表现。

大模型微调(Fine-tuning)是指基于预训练大模型，通过特定领域或任务的数据进行二次训练，使模型适应特定领域或任务的过程。与从零开始训练相比，微调能够以较低成本实现模型的垂直领域适配，是大模型落地应用的核心技术路径。大模型微调示意图如图 3.1 所示。

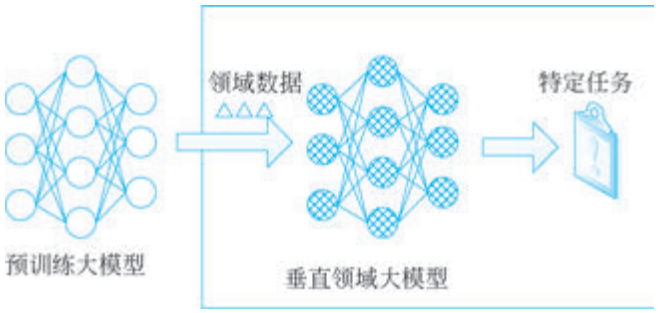

图 3.1　大模型微调示意图

1. 大模型微调的原因

虽然提示工程在某种程度上也能适配特定领域或任务，但仍需要对大模型进行微调，主要原因有三个：一是因为大模型的参数量非常大，训练成本非常高，用户从零开始训练一个的大模型的性价比非常低；二是虽然提示工程容易上手，但如果提示词过长，会增加大模型的推理成本，甚至会因模型输入长度超出限制而被截断，影响推理的准确性。对于需要考虑推理成本和减少对提示词依赖的场景，微调相对来说就是一个更优的方案；三是如果提示工程的效果不佳，用户可收集特定领域数据，通过微调提升大模型的能力，并形成私有的专用大模型。

2. 大模型微调技术的分类

根据微调参数的规模以及是否改变原模型参数，可以将大模型微调技术分为三类：全参数微调、部分参数微调和新增参数微调。具体内容将在 3.4 节～3.6 节中介绍。

3.1.2　大模型微调技术发展历程

大模型微调技术经历了从全参数微调、部分参数微调到参数高效微调的范式转变。

1. 技术萌芽期(2017—2018 年)

技术萌芽期是全参数微调的奠基阶段，主要有以下两个标志性事件。

1）Transformer 架构的诞生

2017 年，Transformer 架构这一创新模型基于自注意力机制彻底改变了自然语言处理和其他序列建模任务的范式，为后续预训练语言模型的发展奠定了基础。

2）BERT 模型的横空出世

2018 年 10 月，BERT（Bidirectional Encoder Representations from Transformers）模型的提出标志着“预训练-微调”范式的正式确立。BERT 是基于上下文的预训练模型，通过大量语料学习到每个词的一般性嵌入形式，学习到与上下文无关的语义向量表示。BERT 模型的训练分为两个阶段：第一阶段是预训练，第二阶段是微调。在预训练阶段，BERT 模型通过大规模未标注文本进行预训练。预训练采用两种任务：一是遮蔽语言模型（Masked Language Model，MLM），随机遮盖输入序列中的部分 Token，预测被遮盖的 Token；二是下一句预测（Next Sentence Prediction，NSP），预测两个句子是不是连续的。在微调阶段，BERT 模型通过反向传播更新所有参数以适应特定的下游任务。这种方法被称为全参数微调（Full Fine-tuning），是 BERT 时代标准的微调方法。对于一个具体任务（如问答、机器翻译等），在 BERT 模型的基础上添加一个输出层进行微调，即可得到一个针对该任务的模型。

该阶段的技术特点主要表现在整体性调节策略、粗粒度控制策略和数据密集型特性三个方面。

（1）整体性调节策略。

直接利用交叉熵损失优化全部参数，不加区分地对待模型各部分参数。这种策略简单直接，但缺乏针对性，未能充分利用模型各部分的特性。

（2）粗粒度控制策略。

简单采用分层学习率衰减策略，未能针对模型不同层次进行精细化调整。全参数微调通常采用统一的学习率或简单的分层学习率衰减，无法根据模型各层的重要性进行精细调整。

（3）数据密集型特性。

高度依赖大规模标注数据，对数据量和质量要求较高。全参数微调需要大量标注数据来更新整个模型，这对于数据资源有限的场景构成挑战。

尽管全参数微调在 BERT 时代取得了显著成功，但它也面临计算成本过高和灾难性遗忘现象两个挑战。

（1）计算成本过高。

例如 BERT- base 拥有 24 层 Transformer，拥有 1.1 亿个参数。对其进行全参数微调需要一定的算力支持（例如 4 块 TITAN Xp GPU）。这对于普通研究者来说，构成了难以逾越的门槛。

（2）灾难性遗忘现象。

大模型难以在保留通用语言能力与适应特定任务之间取得平衡。全参数微调容易导致灾难性遗忘，即在更新模型以适应特定任务时，模型可能忘记其在预训练阶段学习到的通用语言知识。

上述这些挑战促使研究者探索更加高效和精准的微调方法，为后续参数高效微调技术的兴起奠定了基础。

2. 效率革命期(2019 年至今)

效率革命期是参数高效微调的崛起阶段。随着大模型规模突破 10 亿参数,全参数微调的可行性受到了挑战。人们把研究重心转向参数高效微调,标志着技术范式的第一次重大转折。参数高效微调技术的主要目标是在保持或接近全参数微调性能的同时,显著减少计算资源需求和训练时间。

参数高效微调正在快速发展,层出不穷的方法被提出。接下来主要介绍比较常见的三种方法:适配器微调、软提示微调和低秩矩阵分解。这三种方法各有特点,针对不同的应用场景和需求提供了多样化的选择。

1) 适配器微调

2019 年首个 Transformer 适配器架构被提出。该方法的核心思想是通过在预训练模型的每层插入轻量级适配器模块,允许模型在不修改原始参数的情况下学习特定任务的表征,实现了对特定任务的快速适应。

随后,研究者提出了并行适配器、动态适配器等变体,进一步优化了适配器结构和训练策略。这些方法通过不同的方式扩展了原始适配器微调的概念,提供了更加灵活和高效的任务适应策略。

2) 软提示微调

GPT-3 的上下文学习(In-context Learning)启发了软提示(Soft Prompt)微调方法。相对应地,第 2 章提示工程中的 Prompt 可称为硬提示(Hard Prompt)。软提示是可学习的连续向量,可以通过梯度优化方法针对特定数据集进行优化。这种方法不需要人工设计提示词,可以自动优化以适应不同任务,计算效率高,支持多任务学习。软提示微调通过设计特定的提示结构,将任务指令编码作为输入序列的一部分,引导大模型生成期望的输出。这些方法包括 Prefix Tuning、Prompt Tuning 和 P-tuning。

Prefix Tuning 方法通过在模型输入的前缀位置添加可学习的提示向量来实现。这种方法的优势在于可以在不改变模型结构的情况下,为不同的任务提供不同的提示。

Prompt Tuning 方法通过将任务指令编码为可学习的向量,即只更新 Prompt Token 的嵌入向量的梯度,在不修改预训练大模型参数的情况下引导其生成期望的输出,仅训练 0.01%参数即可适配下游任务,大大降低了计算资源需求。

P-tuning 添加了一个可训练的嵌入张量,这个张量可以被优化以找到更好的提示,并且使用提示编码器(例如 LSTM)来优化提示参数,实现深度提示优化,在复杂任务上甚至超越了全参数微调效果。

3) 低秩矩阵分解

2021 年,研究人员提出了 LoRA(Low-Rank Adaptation,低秩适配)。该方法假设参数更新是低秩的,通过分解参数更新矩阵为两个低秩矩阵的乘积,大幅减少了需要训练的参数量(将参数更新量压缩至原模型的 0.1%),极大地降低了存储和计算需求。

除了上述三大类方法外,还有一些参数高效微调方法,例如 IA^3(Infused Adapter by Inhibiting and Amplifying Inner Activations)等。该方法通过引入重要性感知机制,对不同参数采用不同的学习率,用学习率向量对键值参数进行缩放优化。

从现有文献来看,参数高效微调技术的突破使微调效率普遍提升了 10～100 倍。这些

方法通过各种方式减少需要训练的参数量，显著降低了计算资源需求和训练时间，使得在资源受限的设备上部署大语言模型成为可能。

然而，这些方法也带来了新挑战。参数高效微调方法通常在推理时增加额外的计算开销，影响了模型的实时性能。此外，这些方法在跨任务泛化能力方面还存在不足，难以在多种任务之间共享学习到的知识。

3.2　大模型微调核心理论

3.2.1　高维非凸优化

大模型参数微调本质上是在高维参数空间（$d \geqslant 10^{10}$）中寻找最优约束优化问题，其数学形式可表述为式（3.1）：

$$\min_{\theta \in \Theta} E_{(x,y)} D_{\text{task}} [L(f_\theta(x), y) + \lambda R(\theta - \theta_p)] \tag{3.1}$$

式中，θ_p 为大模型预训练参数，正则化项 $R(\cdot)$ 用于控制参数偏移量。这一优化问题具有典型的带约束高维非凸优化特性，其复杂性源于以下三个主要方面。

1. 参数空间的维度灾难

大模型的参数空间维度通常高达数十亿，远超传统优化问题的处理范围。例如，GPT-3 模型拥有 1750 亿参数，这意味着在 1750 亿维的参数空间中进行优化。这种高维空间的性质与低维空间有着本质的不同，带来了所谓的“维度灾难”问题。在高维空间中，数据点之间的距离迅速增大，使得局部信息难以代表全局特征；同时，样本密度随着维度增加而呈指数级下降，导致在高维空间中很难找到足够多的样本来准确估计分布。这些特性使得传统的优化方法在面对大模型微调时面临巨大挑战。

维度灾难问题的一个直接后果是搜索空间极其庞大，难以高效探索。对于一个拥有数十亿参数的模型，即使每个参数只考虑少许可能取值，整个搜索空间的规模也是天文数字。例如，如果每个参数只需要考虑 10 种可能的取值，那么对于一个拥有 100 亿参数的模型，搜索空间的规模将达到 10^{100}，这远远超出当前计算能力的极限。因此，如何在如此庞大的搜索空间中有效地找到最优或近似最优的参数组合，成为大模型微调面临的核心挑战之一。

2. 非凸优化的复杂性

大模型微调的另一个核心特性是非凸性。表现在其损失函数在高维参数空间中通常呈现非凸性质，存在指数级数量的局部极小点，增加了找到全局最优解的难度。在非凸优化问题中，优化算法可能陷入局部极小点、鞍点等，从而无法达到全局最优解。对于大模型微调而言，损失函数的复杂性进一步加剧了这一问题，因为大模型通常具有非常复杂的损失景观，包含大量的局部极小点、鞍点和各种复杂的几何结构。

非凸性问题的显著性体现在优化过程的不确定性上，即不同的初始化点可能导致优化算法收敛到不同的局部极小点，而这些局部极小点的性能可能差异很大。在某些情况下，即使优化算法找到了一个局部极小点，该点也可能是非常差的，从而导致模型在测试集上表现

不佳。此外，非凸性还可能导致优化过程的不稳定，例如在训练过程中出现损失突然增加或振荡的现象。

3. 参数空间中的低损失路径

研究表明，参数空间中存在大量低损失路径（Low-loss Pathways），有效维度（Effective Dimensionality）仅为总维度的 0.1%～1%。这一特性意味着，虽然参数空间维度极高，但实际有用的参数变化方向非常有限。

低损失路径的存在是理解大模型微调有效性的关键之一。尽管参数空间维度很高，但模型性能主要取决于参数在某些特定方向上的变化，而不是所有方向。这些特定方向可能与任务相关，或者与模型的内部表示有关。通过只关注这些特定方向，可以大幅减少需要优化的参数数量，从而提高微调的效率。

在实际应用中，大模型微调的优化问题通常还包含各种约束条件，例如参数更新的稀疏性约束、低秩约束等。这些约束条件进一步增加了问题的复杂性。为了应对这些挑战，研究者们提出了各种优化算法和策略，如自适应学习率方法、动量方法以及各种参数高效微调方法等。

3.2.2　病态曲率分析

大模型微调的优化问题中，一个关键的挑战是黑塞矩阵（Hessian Matrix）的病态条件数导致传统优化算法收敛困难。通过深入分析大模型损失函数的曲率特性，可以更好地理解微调过程中的优化难题，并为开发更有效的微调方法提供理论基础。

1. 黑塞矩阵分析理论

黑塞矩阵的特征值分布反映了损失函数在不同方向上的曲率，而这些曲率特性直接影响了优化算法的行为。通过分析黑塞矩阵，可以了解大模型微调过程中的优化挑战，并为开发有效的微调方法提供指导。

大模型的黑塞矩阵通常具有少数大特征值和大量接近零的特征值的结构。这种结构表明，损失函数在某些方向上具有显著的曲率，而在其他方向上则接近平坦。这种各向异性的曲率特性是理解大模型微调挑战的关键之一。

此外，黑塞矩阵分析还揭示了大模型微调中的有效维度特性。通过分析黑塞矩阵的特征值分布，可以确定哪些方向对模型性能有显著影响，从而指导微调方法的设计。例如，如果某些方向上的特征值远大于其他方向，那么这些方向可能对模型性能有更重要的影响，因此在微调过程中应该给予更多的关注。

黑塞矩阵分析还为理解大模型的泛化能力提供了新的视角。研究表明，大模型的泛化能力可能与黑塞矩阵的谱特性有关。例如，如果黑塞矩阵的特征值分布表明模型在某些方向上过于敏感，那么它可能容易过拟合训练数据，从而降低泛化能力。因此，通过分析黑塞矩阵，可以了解微调过程如何影响模型的泛化能力，并为设计更好的微调策略提供指导。

2. 病态条件数与优化挑战

黑塞矩阵是函数二阶导数的矩阵，它描述了函数在不同方向上的曲率。黑塞矩阵的条

件数是其最大特征值与最小特征值的比值，它反映了函数在不同方向上的曲率差异。在优化算法中，用来描述函数局部曲率的黑塞矩阵的条件数通常超过 10^4，使得优化过程极其不稳定，导致梯度方向可能与实际下降方向偏差较大。条件数是描述一个矩阵“好坏”程度的指标，它表示矩阵在数值上的敏感度。对于优化问题而言，条件数反映了函数在不同方向上的曲率差异。条件数越大，函数在不同方向上的曲率差异越大，优化过程就越不稳定。

病态条件数（一般认为条件数大于 10 就是病态的）对优化过程的影响是多方面的。首先，它可能导致优化算法在某些方向上进展缓慢，而在其他方向上进展迅速，从而导致优化过程的不均衡。其次，它可能导致梯度方向与实际下降方向之间存在显著偏差，使得优化算法难以沿着最有效的方向前进。最后，病态条件数还可能导致优化算法对初始点和超参数（如学习率）非常敏感，使得找到合适的参数设置变得更加困难。

在大模型微调过程中，病态条件数的出现与模型的规模和复杂性密切相关。随着模型参数数量的增加，损失函数的复杂性也相应增加，导致黑塞矩阵的条件数增大。此外，大模型通常使用复杂的神经网络架构，如 Transformer。该架构本身就可能导致损失函数具有复杂的曲率特性，从而进一步加剧条件数问题。

3. 优化方向的各向异性

大模型微调中的另一个重要特性是优化方向的各向异性。在参数空间中，某些方向上的优化进展远快于其他方向，这种差异与黑塞矩阵的特征值分布密切相关。

各向异性的优化方向对微调方法的选择有重要影响。例如，某些微调方法通过在特定方向上进行参数更新，可以更有效地利用这些方向上的优化进展，从而提高微调效率。此外，优化算法的设计也需要考虑各向异性特性，例如使用自适应学习率的方法可以在不同方向上使用不同的学习率，从而更好地适应各向异性的优化环境。

各向异性还与大模型的泛化能力有关。研究表明，大模型在某些方向上的敏感性可能影响其泛化性能。例如，如果大模型在某个方向上过于敏感，那么它可能容易受到噪声的影响。因此，在微调过程中，如何平衡不同方向上的优化进展，成为提高大模型性能和泛化能力的关键问题。

3.2.3　流形嵌入假说

流形嵌入假说为理解大模型微调提供了一个直观的视角，它假设预训练模型参数位于高维参数空间中的低维光滑流形上，微调过程可视为在该流形上的局部探索。这一假说不仅解释了为什么大模型能够有效地进行微调，也为参数高效微调方法提供了理论基础。

1. 低维流形的存在性

流形嵌入假说的核心前提是预训练模型参数位于高维参数空间中的低维光滑流形上。研究表明，尽管参数空间维度高达数十亿，但实际有用的参数变化局限于低维流形上。这一低维流形的存在是理解大模型微调有效性的关键之一。

低维流形的存在可以通过参数空间的几何特性来理解。在高维参数空间中，尽管总维度很高，但实际有用的参数变化方向非常有限。这些有用的参数变化方向构成了一个低维流形，模型参数在这个流形上移动时，可以实现性能的显著提升。

低维流形的存在还与模型的结构和任务的性质有关。例如，对于特定任务，某些参数可能比其他参数更重要，因此在这些重要参数的方向上可能更容易找到性能提升的路径。此外，预训练过程也可能引导模型参数进入一个特定的低维流形，使得微调过程可以在这个流形上进行有效的探索。

2. 微调过程的流形探索

根据流形嵌入假说，微调过程可视为在低维流形上的局部探索。这一过程通过更新参数，使得模型沿着流形移动，从而找到在特定任务上性能更好的参数组合。由于流形的维度远低于参数空间的总维度，这种探索可以在相对较低的计算成本下完成。

微调过程的流形探索可以通过参数更新的方向来理解。在参数空间中，参数更新的方向通常沿着某些特定的方向，而这些方向可能与低维流形的方向一致。通过只关注这些特定方向的参数更新，可以有效地探索流形，从而找到性能更好的参数组合。

流形探索的一个重要特性是它允许只微调参数空间的一小部分。例如，LoRA、Adapter Tuning 等参数高效微调方法通过限制参数更新的方向或结构，可以在低维流形上进行有效的探索，而不需要更新全部参数。这不仅提高了微调效率，也为在资源受限的环境中部署大模型提供了可能。

3. 切线空间分析

切线空间分析是理解流形嵌入假说的重要工具。它通过分析参数空间中切线方向上的更新效果，揭示了有效更新方向与预训练梯度之间的关系。研究表明，有效更新方向与预训练梯度为 75°～85°夹角，这表明微调过程通常沿着与预训练梯度有一定偏差的方向进行。

切线空间分析的一个重要发现是有效更新方向通常与预训练梯度有一定偏差。这一偏差可能反映了特定任务与预训练任务之间的差异，使得微调过程需要沿着与预训练梯度不同的方向进行参数更新。例如，在网络安全领域的微调中，大模型需要学习与网络安全相关的概念和知识，这些概念和知识可能与预训练任务（如新闻阅读）中的概念和知识有所不同，因此微调过程可能需要沿着与预训练梯度不同的方向进行参数更新。

切线空间分析还揭示了参数更新方向的多样性。在参数空间中，不同的参数更新方向可能对应不同的任务或功能。通过分析这些不同方向上的更新效果，可以更好地理解模型参数的组织结构，以及微调过程如何影响模型的行为和性能。

4. 流形嵌入假说的验证

流形嵌入假说的验证可以通过实证研究和理论分析来完成。实证研究可以通过分析大模型微调过程中的参数变化轨迹，验证这些轨迹是否主要沿着低维流形进行。理论分析则可以通过数学模型来描述参数空间的几何特性，以及微调过程如何在这些几何特性下进行。

实证研究可以验证大模型参数确实位于低维流形上，并且微调过程主要沿着这个流形进行。例如，通过分析大模型参数在微调过程中的变化，研究人员发现参数更新通常集中在某些特定的方向上，而这些方向构成了一个低维流形。此外，通过改变微调方法或策略，可以观察到参数更新方向的变化，从而进一步验证流形嵌入假说。

理论分析则通过数学模型来描述参数空间的几何特性。例如，通过分析损失函数的特

性，可以推导出参数空间中流形的维度和曲率等特性。这些理论分析不仅验证了流形嵌入假说，还为理解大模型微调的优化过程提供了新的视角。

3.2.4　损失景观的多尺度特性

大模型参数微调的损失景观具有复杂的多尺度特性。通过深入分析损失地貌的多尺度特性，可以更好地理解微调过程中的各种现象，并为开发更有效的微调策略提供指导。

1. 全局尺度上的极小值盆地

在全局尺度上，大模型参数微调的损失景观通常存在多个极小值盆地。极小值盆地是指损失函数值较低的区域，模型参数在这个区域内时，模型在特定任务上表现良好。多个极小值盆地的存在意味着存在多种不同的参数配置，可以使模型在特定任务上表现良好，从而增加了模型的健壮性和泛化能力。

极小值盆地的存在对微调过程有重要影响。首先，它使得微调过程有多种可能的优化路径，增加了找到性能良好的参数配置的可能性。其次，它使得模型对参数初始化和优化算法的敏感性降低，因为即使从不同的起点开始，优化过程也可能收敛到性能良好的极小值盆地。最后，它增加了大模型的泛化能力，因为多个极小值盆地可能对应不同的表示方式，而这些不同的表示方式可以在不同的测试样本上表现良好。

全局尺度上的极小值盆地特性可以通过随机矩阵理论来分析。随机矩阵理论研究随机矩阵的特征值分布和相关统计特性，它可以用来描述大模型参数空间中的随机波动和统计特性。

2. 微观尺度上的高频振荡

在微观尺度上，大模型参数微调的损失景观通常具有噪声主导的高频振荡，表明损失函数在局部具有复杂的波动特性。这种高频振荡反映了损失函数在微观尺度上的噪声和不确定性，对微调过程的稳定性有重要影响。

微观尺度上的高频振荡可以通过分析损失函数在参数空间中的局部行为来理解。在参数空间的微观尺度上，损失函数可能受到各种因素的影响，例如训练数据的噪声、模型的随机初始化等，从而导致模型复杂的波动特性。这些波动特性使得优化过程在微观尺度上变得不稳定，增加了找到全局最优解的难度。

微观尺度上的高频振荡对微调过程有着重要影响。首先，它使得优化过程容易陷入局部极小点或鞍点，从而降低找到全局最优解的可能性。其次，它使得模型在训练集上的表现可能与在测试集上的表现有显著差异，从而影响模型的泛化能力。最后，它增加了微调过程的不确定性，使得多次运行的结果可能出现显著差异。

3. 多尺度特性对微调方法的影响

大模型参数微调的多尺度特性对微调方法有重要影响。不同的微调方法可能在探索损失景观的不同尺度时表现出不同的效果。例如，某些方法可能更擅长探索全局尺度上的极小值盆地，而其他方法则可能更擅长处理中观尺度上的各向异性特征或微观尺度上的高频振荡问题。

多尺度特性对微调方法的影响可以通过分析不同方法在不同尺度上的行为来理解。例如，全参数微调方法通过更新所有参数，可以在所有尺度上进行探索，但计算成本较高。参数高效微调方法通过限制参数更新的方向或结构，可以在特定尺度上进行更有效的探索，从而提高微调效率。

多尺度特性还为理解不同微调方法的有效性提供了框架。例如，某些微调方法可能在特定任务上表现良好，是因为它们能够有效地探索损失景观的特定尺度。通过理解多尺度特性，可以预测不同微调方法在特定任务上的表现，并为选择合适的微调方法提供指导。

3.2.5 微调技术价值

大模型微调技术的核心价值可以通过性能增益、训练效率和资源消耗等维度进行量化评估。

1. 性能增益

性能增益是微调技术价值的第一个维度，它衡量微调技术在特定任务上提升模型性能的能力。这种性能提升不仅体现在准确率等传统指标上，还可能包括模型对特定领域知识的掌握程度、对边缘案例的处理能力等多个方面。

性能增益的量化可以通过多种指标来衡量，如准确率、精确率、召回率、F1 值等。不同的任务可能需要不同的指标来评估性能增益。例如，在文本分类任务中，准确率和 F1 值是常用的评估指标；在命名实体识别任务中，精确率、召回率和 F1 值是常用的评估指标；在问答系统中，BLEU 分数、ROUGE 分数等是常用的评估指标。

除了量化指标外，性能增益还可能体现在模型对特定领域知识的掌握程度上。例如，在医疗领域，微调可以使模型更好地理解医疗术语和概念，从而提高诊断准确率。在代码生成任务中，微调可以使模型更好地理解编程语言和算法，从而生成更高质量的代码。这些特定领域知识的掌握程度可能难以通过单一的量化指标来衡量，但它们对模型的实际应用价值却至关重要。

此外，性能增益还可能体现在模型对边缘案例的处理能力上。边缘案例是指与训练数据分布不同的测试样本，它们通常对模型的健壮性和泛化能力有更高的要求。通过微调，模型可能会更好地处理这些边缘案例，从而提高整体的健壮性和泛化能力。

2. 训练效率

训练效率是微调技术价值的第二个维度，它衡量微调技术在训练过程中节省时间和计算资源的能力。参数高效微调技术显著提升了模型微调的效率，例如参数高效微调可以使 175B 模型的微调时间从三百多个小时降至三十几个小时。这种效率提升不仅体现在训练时间上，还包括训练所需的计算资源、数据量等方面。

训练效率的量化可以通过多种指标来衡量，如训练时间、计算资源消耗、数据需求等。不同的微调方法可能在这些指标上表现出不同的效率特性。例如，全参数微调虽然可能在性能上表现最好，但在训练时间和计算资源消耗上却可能是最高的；而参数高效微调方法虽然可能在性能上有所牺牲，但可能显著降低训练时间和计算资源的消耗。

微调过程的稳定性也是评估训练效率的重要因素。一个稳定且收敛快的微调过程不仅

可以节省时间和计算资源，还可以减少人工干预的需求，从而提高效率。

3. 资源消耗

资源消耗是微调技术价值的第三个维度，它衡量微调技术在模型部署和使用过程中节省资源的能力。资源消耗的量化可以通过多种指标来衡量，如显存需求、能耗等。不同的微调方法可能在这些指标上表现出不同的资源消耗特性。

此外，资源消耗还可能体现在模型的推理速度上。一个资源消耗较低的微调方法不仅可以节省硬件资源，还可以提高模型的推理速度，从而支持更高吞吐量的应用场景。

4. 微调技术价值的综合评估

微调技术的价值可以通过性能增益、训练效率和资源消耗三个维度进行综合评估。不同的应用场景可能对这三个维度有不同的重视程度。例如，在资源受限的移动设备上部署大模型，可能更重视资源消耗维度；在需要实时响应的场景中，可能更重视训练效率和推理速度；而在精度要求极高的专业领域应用中，可能更重视性能增益维度。

此外，综合评估微调技术价值还需要考虑实际应用的需求和约束。例如，在医疗、法律、政务等领域，大模型的性能和可靠性可能比训练效率和资源消耗更为重要；而在商用推荐、聊天娱乐系统中，训练效率和资源消耗可能比大模型性能更为重要。

3.3　数据准备

人工智能的三大核心要素是算法、算力和数据，其中数据是训练大模型的基础要素，也是大模型应用的核心资源。大模型微调的效果在很大程度上取决于所使用数据的质量和准备方式。本节深入探讨大模型微调数据准备的重要性、数据收集和预处理的方法。

3.3.1　数据准备的重要性

1. 数据质量对微调效果的影响

数据质量是微调成功的关键因素之一。在微调过程中，大模型会根据提供的数据调整其参数，以更好地适应特定任务。如果数据质量不高，大模型可能会学习到数据中的噪声或偏差，导致性能下降。

数据质量包括多个方面，如数据的准确性、完整性、一致性等。准确性指的是数据本身是否正确；完整性指的是数据是否涵盖所有必要的信息；一致性指的是数据在不同样本之间是否具有一致的格式和分布。只有在这些方面都达到较高水平的数据，才能为大模型微调提供良好的基础。

2. 数据多样性的重要性

数据多样性是指数据覆盖的范围和类型。多样化的数据可以帮助大模型学习到更广泛的模式和特征，从而提高其泛化能力。在实际应用中，大模型需要处理各种各样的输入，如果训练数据缺乏多样性，大模型可能无法应对这些不同的情况。

例如，在情感分析任务中，数据应涵盖不同情感倾向的文本，包括正面、负面和中性情感。此外，还应包括不同长度的文本、不同风格的文本（如专业性或者科普性）、不同领域的文本（如电影评论、产品介绍等）。只有这样，大模型才能学会在各种情况下准确地识别情感。同样，在问答系统中，使用涵盖不同问题类型和领域的问题数据集，可以提高大模型回答问题的能力。

3. 数据规模与微调效果的关系

虽然预训练模型已经具备很强的语言理解能力，但微调时的数据规模仍然对最终效果有重要影响。一般来说，数据量越大，大模型的性能提升越明显，但同时也需要更多的计算资源和时间。

然而，数据规模并不是唯一的因素。数据的质量和多样性同样重要。即使使用较小规模的数据集，如果数据质量高且具有良好的多样性，也可以获得较好的微调效果。因此，在准备微调数据时，需要平衡数据质量、规模和多样性。

3.3.2 数据的来源与收集

1. 公开数据

有许多公开的数据集可供选择，这些数据集通常经过了精心整理和标注，可以直接用于微调，覆盖了各种任务和领域，如文本分类、情感分析、命名实体识别、知识问答等。使用这些数据集可以节省数据收集和标注的时间和成本。

常用的公开数据集平台包括 Hugging Face、Kaggle、Google Dataset Search、ModelScope 等。这些平台提供了大量的高质量数据集，涵盖自然语言处理、计算机视觉、音频处理等多个领域。例如，Hugging Face 的 datasets 库提供了超过 5000 个高质量数据集，包括文本、图像、音频等多种类型的数据。

在选择公开数据集时，需要考虑数据集的适用性。数据集应该与微调任务相关，且数据格式和分布应该与实际应用场景相似。此外，还需要考虑数据集的规模和质量，以确保能够获得良好的微调效果。

例如，在情感分析任务中，可以选择像 SST-2、IMDB 等数据集。这些数据集包含不同领域和类型的情感文本。在命名实体识别任务中，可以选择像 CoNLL-2003、OntoNotes 等数据集，这些数据集包含丰富的命名实体标注，可以用于训练命名实体识别模型。在网络入侵检测领域，可以选择数据集 CIC-IDS2017，它包含正常的网络流量和各种攻击流量，如 DDoS、端口扫描、恶意软件等。

然而，公开数据集也存在一些局限性。首先，公开数据集可能与垂直领域的应用场景不完全匹配，导致模型在实际应用中的表现不达预期。其次，公开数据集可能缺乏某些特定领域的数据。最后，公开数据集可能包含一些噪声和错误，这会影响模型的训练效果。

为了克服上述局限性，可以考虑对公开数据集进行适当的预处理，除去其中的噪声和错误，保留高质量的数据；也可以考虑将多个公开数据集合并使用，以增加数据的多样性和规模；还可以考虑对公开数据集进行数据增强，生成新的样例，增加数据的有效规模。

2. 垂直领域数据

在某些情况下，公开数据集可能无法完全满足特定领域的需求。此时，可以收集特定领域的数据。这些数据有多种来源，如网络爬取、用户生成内容、专业数据库等。收集特定领域的数据可以确保数据与实际应用场景更加匹配，从而提高微调效果。

例如，在医学领域，可以从医学文献、病历记录中提取数据；在金融领域，可以从金融报告、股票评论中提取数据；在法律领域，可以从法律文书、案例判决中提取数据；在网络安全领域，可以从网络设备（如防火墙、路由器）收集日志数据，这些数据可以用于分析网络流量模式和检测异常行为。

在收集特定领域的数据时，需要注意数据的版权问题。有些数据可能受到版权保护，不能用于商业目的。此外，还需要考虑数据的隐私问题，特别是当数据包含个人身份信息或其他敏感信息时。在使用这些数据时，可能需要进行匿名化处理或获得相关方的许可。

3. 数据标注

模型微调有时需要标注数据。因此，对于一些收集的未标注数据，需要进行人工标注。数据标注的质量直接影响微调后模型的性能。因此，标注过程需要仔细管理，以确保标注结果的准确性和一致性。

标注过程通常包括以下步骤：首先，定义标注规则和标准，确保标注人员对任务有清晰的理解。其次，选择合适的标注工具。再次，招募具备相关领域的知识和技能的标注人员。最后，监控和审核标注结果，及时发现和纠正错误。

标注时需要注意以下几点：确保标注人员对标注标准有清晰的理解，避免标注结果的不一致性；标注的数据应涵盖各种情况，避免过于集中于某一类数据；标注的数据量应足够大，以满足模型训练的需求。

3.3.3　数据预处理

1. 文本清洗

文本清洗通常是数据预处理的第一步，也是至关重要的一步。它旨在去除无用信息，保留有价值的内容，提高模型的训练效率和准确性。

文本清洗的常见操作包括去除 HTML 标签、特殊字符、停用词等。这些操作可以减少噪声，提高模型的训练效率和准确性，需要根据具体场景和数据特性进行调整和优化。

去除 HTML 标签和特殊字符是文本清洗的基本操作。HTML 标签和特殊字符可能以不同的形式出现，如日志记录中的控制字符、制表符等。可以使用正则表达式来去除文本中的无关字符、特殊符号、HTML 标签、URL 和电子邮件地址。

去除停用词是另一种常见的文本清洗操作。停用词是指在文本中频繁出现但对语义贡献较小的词，例如中文的“的”“是”“在”等或英文的“the”“is”“and”等。

在网络安全数据中，某些常见词如“error”“warning”“alert”等可能具有重要意义，不应随意去除。因此，停用词列表应根据具体任务和领域进行调整。此外，还可以根据文本内容动态生成停用词列表，例如把文本中出现频率过高的词汇作为停用词。

在情感分析任务中，可以去除 HTML 标签、特殊字符和停用词，只保留对情感判断有价值的内容。在命名实体识别任务中，可以去除停用词，只保留可能包含命名实体的词语。在文本分类任务中，可以清洗文本，只保留与分类相关的特征。

此外，文本清洗在文本预处理中的作用不容忽视。通过文本清洗可以将原始文本转换为更干净、更一致的形式，便于后续的文本分析和模型的训练。

2. 分词

分词是自然语言处理中的一个基本任务，其目的是将连续的文本分割成有意义的词汇单元。分词的质量直接影响后续处理的效果，如文本分类、情感分析、命名实体识别等。在网络安全领域，分词面临一些独特的挑战，如网络安全特定术语的处理、混合语言的处理等。常见的分词方法有：基于规则的方法、基于统计的方法、基于深度学习的方法等。

基于规则的方法依赖于词典和语法规则，如最大匹配法、最小切分法等。这些方法简单直观，易于实现，但需要大量的领域知识和规则，难以处理未登录词(OOV)问题。在网络安全数据中，常常包含大量的特定术语和专有名词，如“DDoS”“SQL 注入”“蠕虫”等。此外，网络安全数据中还可能包含混合语言的情况，如英文和中文的混合使用，进一步增加了分词的难度。

基于统计的方法利用大量语料库统计信息，例如隐马尔可夫模型、条件随机场等。这些方法可以处理未登录词问题，但需要大量的标注语料库进行训练，其训练过程复杂，且对语料库的质量和多样性要求较高。在网络安全数据方面，高质量的标注语料库可能难以获取，这限制了基于统计的方法的应用。

基于深度学习的方法利用神经网络模型，如循环神经网络、卷积神经网络等。这些方法可以自动学习文本的特征和模式，处理未登录词问题，但需要大量的计算资源和标注数据，且模型的可解释性较差。然而，随着计算资源的增加和深度学习技术的发展，基于深度学习的分词方法在网络安全数据处理中变得越来越流行。

在实际应用中，分词方法的选择需要考虑多种因素，如数据特性、处理要求、计算资源等。例如，在处理包含大量特定术语和专有名词的网络安全数据时，可以结合使用基于规则的方法和基于统计或深度学习的方法，先使用基于规则的方法处理常见词和特定术语，再使用基于统计或深度学习的方法处理未登录词。此外，还可以使用后处理技术，例如基于上下文的词义消歧等，提高分词的准确性。

对于英文，分词相对简单，可以直接按照空格分割。对于中文，分词则需要借助分词工具，如 jieba。以下是一段使用 jieba 进行中文分词的代码：

```
import jieba
text = "自然语言处理是人工智能领域的一个重要方向"
words = jieba.cut(text)
print(list(words))
```

在网络安全数据中，常常需要处理混合语言的情况，如英文和中文的混合使用。对于这种情况，可以使用支持多语言的分词工具，如 spaCy 等。

3. 编码

在将文本输入模型前，需要将其转换为模型可以理解的数字形式，这就是编码过程。常见的编码方法有词嵌入(Word Embedding)和 BERT Tokenizer 等。

词嵌入是将单词映射为一个低维的密集向量，可以捕捉单词之间的语义关系。常用的词嵌入工具包括 Word2Vec、GloVe 等。Word2Vec 是一种基于神经网络的词嵌入方法，通过预测上下文单词或目标单词来学习词向量。GloVe 是一种基于统计的词嵌入方法，通过分析单词共现矩阵来学习词向量。这些词嵌入方法可以捕捉单词之间的语义关系，如相似性、类比等，对于自然语言理解和处理具有重要作用。

对于网络安全数据，词嵌入可以用于表示相关术语和概念，如“DDoS”“SQL 注入”“蠕虫”等，捕捉它们之间的语义关系，提高模型的训练效果和性能。

BERT Tokenizer 是 BERT 模型使用的一种特殊编码方式，将文本分割为子词(Subword)，可以更好地处理未登录词问题。BERT 是一种基于 Transformer 的预训练语言表示模型，可以用于各种自然语言处理任务，如文本分类、命名实体识别、情感分析等。BERT Tokenizer 是 BERT 模型使用的分词工具，它使用 WordPiece 算法将文本分割为子词，可以处理未登录词和罕见词。

在选择编码方法时，需要考虑模型类型和任务特点。例如，对于基于 BERT 的模型，应该使用 BERT Tokenizer 进行编码；对于基于传统神经网络的模型，可以使用词嵌入或 One-Hot 编码等方法进行编码。此外，还需要考虑数据特性，如数据量、数据多样性、数据质量等，选择最适合的编码方法。

编码过程还包括处理词向量的维度和范围。词向量的维度应该与模型的输入维度匹配，词向量的范围应该适当，以避免模型训练中的梯度爆炸或消失问题。

编码的质量对模型性能有重要影响。高质量的编码可以捕捉文本的语义信息，提高模型的准确性和效率。低质量的编码可能导致信息丢失或引入噪声，影响模型的性能。在网络安全领域中，高质量的编码对于准确识别和分类安全事件和威胁具有重要意义，可以提高入侵检测系统、异常检测系统等的安全性和有效性。

为提高编码质量，可以考虑使用专业的编码工具和库，如 Python 中的 Gensim、spaCy、Hugging Face Transformers 等。Gensim 是一个用于主题建模、文档索引和相似度检索的 Python 库，包含实现 Word2Vec、GloVe 等词嵌入方法的工具。spaCy 是一个工业强度的自然语言处理库，包含预训练的词嵌入和命名实体识别模型。Hugging Face Transformers 是一个用于 Transformer 预训练的模型库，包含 BERT、GPT-2、RoBERTa 等预训练模型和 Tokenizer。

此外，还可以结合多种编码方法，例如同时使用词嵌入和 BERT 编码，提取更丰富的特征。例如，可以使用词嵌入表示常见词和概念，使用 BERT 编码表示特定术语和复杂概念，结合两者的优势，提高编码的质量和效果。

4. 数据增强

数据增强是指通过一些方法增加数据的多样性，从而提高模型的泛化能力。常见的数据增强方法主要包括同义词替换、随机插入、随机删除、随机交换等。

同义词替换是指将文本中的某些单词替换为它们的同义词。例如，将“高兴”替换为“快乐”。在网络安全领域中，同义词替换可以用于创建攻击数据的变体，如将“DDoS”替换为“分布式拒绝服务”，或将“SQL 注入”替换为“SQL 注入攻击”等。这种方法可以增加攻击数据的多样性，提高模型对不同表述的识别能力。然而，在网络安全中，某些术语可能有特定的含义和用法，需要谨慎选择同义词，避免改变原意或引入歧义。

随机插入是指在文本中随机插入一些单词。在网络安全领域中，随机插入可以用于模拟攻击者的变异和变化，如在攻击描述中插入无关的词汇或短语，模拟攻击者的混淆和干扰。这种方法可以增加攻击数据的多样性和复杂性，提高模型的健壮性和适应性。然而，在网络安全中，随机插入可能引入噪声和干扰，影响模型的训练和性能，需要控制插入的频率和内容。

随机删除是指随机删除文本中的一些词。在网络安全领域中，随机删除可以用于模拟攻击描述的不完整或缺失，如删除攻击描述中的某些部分，模拟不完整的日志或报告。这种方法可以增加攻击数据的多样性和挑战性，提高模型的健壮性和适应性。然而，在网络安全中，随机删除可能导致信息丢失或不完整，影响模型的理解和识别，需要控制删除的频率和内容。

随机交换是指随机交换文本中的一些单词的位置。在网络安全领域中，随机交换可以用于模拟攻击描述的语法变化或重新表述，如交换攻击描述中的某些部分，模拟不同的表达方式或顺序。这种方法可以增加攻击数据的多样性和变异性，提高模型的适应性和泛化能力。然而，在网络安全领域应用中，随机交换可能导致语法错误或语义改变，影响模型的理解和识别，需要控制交换的频率和位置。

若采用的数据增强方法不合理，会破坏文本的语义或语法，生成无效的样本。其次，数据增强的强度和频率也很重要。如果增强过强或过频，可能会引入过多的噪声，影响模型的学习。最后，数据增强的多样性和覆盖范围也很重要。如果增强方法单一或覆盖范围有限，可能无法有效增加数据的多样性。

为了提高数据增强的质量和效果，可以考虑使用专业的数据增强工具和库，如 Python 中的 EDA(Easy Data Augmentation)库。可以设置合理的增强参数，如增强的概率和强度，平衡增强的效果和风险。可以先进行实验，以找到适合特定任务的数据增强策略，然后大规模实施数据增强。

3.4　全参数微调

全参数微调是一种相对彻底的微调方法，即大模型的原始权重参数全部参与更新。

对某个具体大模型进行全参数微调需要考虑三个重要因素。一是确定模型的训练环境，根据预训练大模型的规模选择合适的训练环境。这包括硬件配置和软件环境。二是组织微调数据，根据预训练大模型的训练数据格式，组织微调数据。一般来说，可以从公开渠道获得开源预训练大模型的训练数据格式，而非开源的预训练大模型的数据格式则只能通过相关的文献或者直接咨询模型研发者来获取。三是设定超参数和训练策略，根据微调数据的规模、该预训练大模型在训练阶段的超参数等，设定模型微调所采用的超参数以及训练策略(如学习率、批次大小、训练轮数等)。

上述三个要素准备好后，在资源充足的情况下，可以直接启动大模型训练；如果资源有限，则需要采取内存压缩等技巧来优化训练过程。对于资源充足的情况，大模型全参数微调方法与一般的深度神经网络模型训练类似，这里不做介绍；对于资源有限的情况，本节介绍几种优化方法。

3.4.1　节省外存方法

节省外存方法的代表是 Diff Pruning。Diff Pruning 通过学习一个稀疏的差值向量(Diff Vector)，只存储与原始权重相比变化的部分，从而大幅减少了模型的存储空间需求，同时保持了与全参数微调相当的性能。具体来说，对于每个任务，学习一个稀疏的差值向量，该向量表示相对于原始预训练参数的更改。这个差值向量在训练过程中通过 L0 范数惩罚(鼓励稀疏性)的可微近似进行自适应剪枝。随着任务数量的增加，Diff Pruning 保持了参数高效性，因为它只需要为每个任务存储一个小的差值向量，而原始预训练模型的存储成本保持不变。

Diff Pruning 在以下场景中具有显著优势。

1. 多任务学习

Diff Pruning 允许在不显著增加存储需求的情况下学习多个任务。这对于需要部署多个模型的场景特别有用，例如云服务或移动设备应用。

2. 在线学习

由于 Diff Pruning 不需要在训练期间访问所有任务，它特别适合任务流式到达的场景，例如在线推荐系统或实时翻译服务。

3. 资源受限环境

Diff Pruning 大幅减少了模型的存储需求，使其能够在资源受限的环境中部署，例如移动设备或嵌入式系统。

4. 模型版本控制

Diff Pruning 允许以紧凑的方式存储模型的不同版本之间的差异，这对于需要管理多个模型版本的场景特别有用。

尽管 Diff Pruning 具有许多优势，但它也存在以下局限性。

1. 计算开销

虽然 Diff Pruning 减少了存储需求，但它可能增加计算开销，特别是在需要频繁切换任务的场景中，因为每次任务切换都需要重新计算任务特定的参数。

2. 稀疏性与性能的权衡

Diff Pruning 通过 L0 范数正则化鼓励差分向量的稀疏性，但这可能导致性能下降。在某些任务上，可能需要较大的差分向量才能达到理想性能。

3. 超参数调整

Diff Pruning 引入了新的超参数，例如正则化强度 λ 和近似参数 σ，需要仔细调整。

3.4.2　节省内存方法

1. 混合精度训练

在深度学习中，数值精度是一个关键考虑因素。传统的深度学习框架主要使用单精度浮点数(FP32)来表示网络参数、激活值和梯度。FP32 提供约 7 位有效数字精度，适用于大多数深度学习任务。然而，FP32 的内存占用较大，计算效率也相对较低。为此，研究人员探索了使用更低精度的数值格式进行深度学习训练的可能性。其中，半精度浮点数(FP16)因其较小的内存占用和更高的计算效率而受到关注。FP16 的内存占用仅为 FP32 的一半，而在支持 FP16 的 GPU 上，FP16 的计算速度远快于 FP32 计算。

混合精度训练(Mixed-Precision Training)结合了 FP16 和 FP32 的优点，通过在适当的地方使用 FP16 以提高计算效率和减少内存占用，同时在需要高精度的关键操作中使用 FP32 以保持数值稳定性。这种混合使用不同精度格式的方法可以在不牺牲模型准确性的情况下，显著提高训练速度并减少内存消耗。

虽然使用 FP16 进行训练具有内存占用少、计算速度快、能源效率高等优势，但使用半精度训练也带来了一些挑战，特别是其较窄的动态范围可能导致数值下溢，影响训练的稳定性和模型的准确性。因此，如何在使用 FP16 的同时保持高精度，是混合精度训练需要解决的核心问题。可以使用完整精度权重副本、损失缩放和在完整精度下执行特定算术运算三种关键技术。

1) 维护 FP32 主副本

混合精度训练的第一项关键技术是维护 FP32 主副本，即为模型权重保留一个 FP32 主副本，用于精确地累积和更新梯度，而将 FP16 版本用于实际的前向和反向传播计算。这一技术解决了 FP16 精度有限导致的权重更新丢失问题，确保了模型训练的稳定性与准确性。

具体实现方法包括以下几个关键步骤：首先，在每次迭代开始时，将 FP32 主副本转换为 FP16 版本，用于前向传播计算。其次，在前向传播过程中，使用 FP16 版本的权重进行计算，生成激活值。再次，在反向传播过程中，计算基于 FP16 权重和激活值的梯度。最后，在优化步骤中，将 FP16 梯度累积到 FP32 主副本中，并使用优化算法更新 FP32 权重。完成更新后，将更新后的 FP32 权重转换回 FP16 版本，准备下一次迭代使用。

2) 损失缩放

混合精度训练的第二项关键技术是损失缩放，这是一种防止 FP16 表示中小梯度值下溢的有效方法。由于 FP16 的数值范围较窄，当梯度值非常小时，可能会下溢为零，导致模型无法有效学习。通过损失缩放技术可以放大梯度值，避免这种下溢问题，同时保持模型更新的准确性。

损失缩放的工作原理是通过将损失函数乘以一个固定的缩放因子放大梯度值，避免其下溢。具体实现方法包括：在前向传播过程中，将损失函数 L 乘以一个固定的缩放因子 s，得到缩放后的损失 $L'=sL$；在反向传播过程中，所有梯度值都会被相同的缩放因子 s 放

大，即$\nabla w' = s\nabla w$；在权重更新之前，将梯度除以缩放因子 s，恢复到原始的梯度规模，以确保更新与 FP32 训练一致。

这种机制确保了梯度在 FP16 表示中的稳定性。通过放大损失值，反向传播计算的梯度也会相应放大，避免了由于 FP16 精度限制导致的梯度信息丢失。在更新权重之前，再将梯度除以缩放因子，恢复到原始规模，确保了模型更新的准确性。损失缩放技术的关键在于选择合适的缩放因子 s。如果 s 太小，放大效果不明显，仍然可能导致梯度下溢；如果 s 太大，可能导致梯度过大，引起训练不稳定。因此，选择合适的 s 值对于混合精度训练的成功至关重要。在实践中，通常需要根据具体模型和任务调整 s 的值，以找到最佳平衡点。缩放因子的选择需要针对特定模型和任务进行调整；在分布式训练环境中，需要确保所有进程使用相同的缩放因子。这些问题在实际应用中需要仔细处理，以确保混合精度训练的有效性。

3）算术精度累积

混合精度训练的第三项关键技术是算术精度累积，这一技术通过在更高精度下累积特定算术运算的结果，避免了 FP16 精度限制导致的数值误差。在进行向量点积、求和等操作时，部分计算结果需要在 FP32 精度下累积，以保持计算的准确性。

具体实现方法为：在进行批量计算时，使用 FP16 进行乘法运算，在累积部分结果时采用 FP32 格式。例如，两个向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 的点积可以表示为 $\sum_{i=1}^{n} \boldsymbol{a}_i \cdot \boldsymbol{b}_i$，其中 $\boldsymbol{a}_i$ 和 $\boldsymbol{b}_i$ 为 FP16 格式，但累积 $\sum \cdot$ 的结果在 FP32 中计算。

这种机制确保了关键计算步骤的准确性。通过在 FP16 计算的基础上使用 FP32 进行累积，避免了 FP16 精度限制可能导致的数值误差。算术精度累积通过将计算密集但精度要求相对较低的部分使用 FP16 计算，而将精度要求较高的累积部分使用 FP32 计算，兼顾了计算效率和计算精度。

2. 内存优化

节约内存还可以从微调的计算过程着手，利用一些内存优化（Low-Memory Optimization，LOMO）方法改善全参数微调对计算资源的依赖。传统优化器如 Adam 或 SGD 通常分两步进行：首先计算所有参数的梯度，然后利用这些梯度信息更新模型参数。这种方法因为需要存储所有参数的梯度信息，在内存使用上效率不高。针对此问题，一种随机梯度下降优化器 LOMO 通过重新思考优化器的功能，大幅减少了内存使用量。

LOMO 的核心思想是“融合梯度计算与参数更新”。具体来说，LOMO 不是先计算所有参数的梯度，然后统一更新所有参数，而是对每一部分参数依次进行“梯度计算-参数更新”操作。这样，只需要存储当前正在处理的参数部分的梯度，而不需要同时存储所有参数的梯度，从而将梯度存储的复杂度从 $O(n)$ 降低为 $O(1)$。除此之外，LOMO 还采用了多种技术来确保训练的稳定性：一是通过梯度值裁剪防止梯度值过大导致参数更新不稳定；二是通过分离梯度范数计算，独立计算梯度范数，避免梯度爆炸或消失；三是动态损失缩放，自动调整损失缩放因子，防止梯度下溢。这种设计使得 LOMO 在内存使用上具有显著优势。当与其他内存节省技术结合使用时，LOMO 能够将内存使用量降低到标准方法的 10.8%。这一突破性进展使得在单台机器上使用 8 块 RTX 3090 GPU（每块 24GB 内存）进

行 650 亿参数规模模型的全参数微调成为可能。

LOMO 采用类似于 SGD 的优化策略，虽然简单有效，但在面对复杂的模型参数空间时，其性能往往不如 Adam 等自适应学习率优化器。这限制了 LOMO 在某些场景下的应用效果。为此，AdaLOMO 对 LOMO 进行了改进，结合了自适应学习率机制的低内存优化方法，在保持 LOMO 内存效率的同时，使其能够与主流优化器 AdamW 相媲美。

AdaLOMO 的核心思想是将自适应学习率机制引入低内存优化框架。研究人员通过实证分析发现，在 Adam 优化器中，相比于动量机制，自适应学习率对优化性能的提升更为关键。基于这一发现，AdaLOMO 引入了自适应学习率机制，同时通过非负矩阵分解（Non-negative Matrix Factorization）来估计优化器状态中的二阶矩，从而在不显著增加内存需求的情况下实现自适应学习率。

3.4.3 全参数微调示例

LOMO 和 AdaLOMO 集成到了 Transformers、Accelerate 和 CoLLiE 中。本小节介绍一个使用 LOMO 全参数微调技术微调 deepseek-r1-distill-qwen-1.5b 模型的示例。该示例实现往模型中注入网络安全专业知识。

首先创建虚拟环境。

```
conda create -n lomo-env python=3.12.3
source deepseek-env/bin/activate
```

接着安装必要的 Python 库，包括 PyTorch、Transformers、Datasets 和 LOMO 优化器等。

```
# 安装 PyTorch(确保与 CUDA 版本匹配)
pip install torch --index-url https://download.pytorch.org/whl/cu118
# 安装其他依赖
pip install transformers datasets wandb lomo-optim peft accelerate bitsandbytes
```

下面是全参数微调代码示例。

```
import os
import json
import torch
from transformers import (
    AutoTokenizer,
    AutoModelForCausalLM,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling,
    get_cosine_schedule_with_warmup
)
from datasets import Dataset
from tqdm.auto import tqdm
from lomo_optim import Lomo                    # 仅导入 Lomo

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
print(f"使用设备: {device}")

# 设置环境变量
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"          # 便于调试 CUDA 错误
os.environ["TOKENIZERS_PARALLELISM"] = "false"    # 禁用分词器并行以避免警告

# 模型和数据路径
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"
data_path = "./data/data.json"                    # 示例数据集路径

# 加载 tokenizer 和模型
print(f"加载模型: {model_name}")
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token         # 设置填充标记

# 以 FP16 精度加载模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto",                            # 自动处理多 GPU 分布
)

# 验证模型参数类型
print("\n检查模型参数类型:")
for name, param in list(model.named_parameters())[:5]:      # 只打印前 5 个参数
    print(f" {name}: {param.dtype}")

# 打印模型信息
print(f"\n模型参数数量: {sum(p.numel() for p in model.parameters())}")
print(f"使用的优化器: Lomo (全参数微调)")

# 加载数据
def load_data(file_path):
    print(f"加载数据: {file_path}")
    try:
        with open(file_path, "r", encoding="utf-8") as f:
            data = json.load(f)
    except Exception as e:
        print(f"加载数据时出错: {e}")
        # 创建示例数据用于测试
        print("创建示例数据用于测试…")
        data = [
            {"question": "什么是人工智能?", "answer": "人工智能是指计算机系统能够执行通
常需要人类智能才能完成的任务的能力。"},
            {"question": "什么是深度学习?", "answer": "深度学习是机器学习的一个分支领域,
它使用多层神经网络来学习数据的表示和特征。"}
        ]
    print(f"已加载 {len(data)} 条数据")
    return data

# 处理数据
def process_data(examples):
    # 构建指令模板
```

```
        prompts = []
        for q, a in zip(examples["question"], examples["answer"]):
            prompt = f"问:{q}\n答:{a}{tokenizer.eos_token}"
            prompts.append(prompt)
        # 编码文本
        tokenized = tokenizer(
            prompts,
            truncation=True,
            padding="max_length",
            max_length=512,
            return_tensors="pt"
        )
        # 对于自监督学习,标签与输入相同
        tokenized["labels"] = tokenized["input_ids"].clone()
        return tokenized

    # 加载并处理数据集
    data = load_data(data_path)
    dataset = Dataset.from_list(data)

    print("处理数据集...")
    tokenized_dataset = dataset.map(
        process_data,
        batched=True,
        remove_columns=dataset.column_names
    )

    # 配置训练参数
    training_args = TrainingArguments(
        output_dir="./autodl-tmp/results",
        overwrite_output_dir=True,
        num_train_epochs=2,
        per_device_train_batch_size=4,
        gradient_accumulation_steps=4,
        learning_rate=1e-4,
        weight_decay=0.01,
        warmup_ratio=0.1,
        logging_dir="./autodl-tmp/logs",
        logging_steps=10,
        save_strategy="epoch",
        report_to="none",
        disable_tqdm=False,
        dataloader_num_workers=2,
        gradient_checkpointing=True,
    )

    class CustomTrainer(Trainer):
        def create_optimizer(self):
            """创建优化器并避免参数冲突"""
            if self.optimizer is None:
                print("\n创建优化器 - 检查参数数据类型:")
                param_types = {}
                for param in self.model.parameters():
```

```
                dtype = str(param.dtype)
                if dtype not in param_types:
                    param_types[dtype] = 0
                param_types[dtype] += 1
            for dtype, count in param_types.items():
                print(f" {dtype}: {count} 个参数")
            # 关键修复:仅传递必要的位置参数,其他参数通过关键字传递
            # 假设 Lomo 的前两个位置参数是 params 和 model,其余参数用关键字传递
            self.optimizer = Lomo(
                self.model,
                lr = self.args.learning_rate,
            )
            print(f"优化器创建成功: {type(self.optimizer).__name__}")
        return self.optimizer
    def create_scheduler(self, num_training_steps: int = None, optimizer = None):
        """修复:接收 optimizer 参数以兼容基类调用"""
        # 优先使用传入的 optimizer,否则使用自身的 optimizer
        if optimizer is None:
            optimizer = self.create_optimizer()        # 确保优化器已初始化
        # 计算训练步数(如果未提供)
        if num_training_steps is None:
            train_dataloader = self.get_train_dataloader()
             num_update_steps_per_epoch = len(train_dataloader) // self.args.gradient_
accumulation_steps
            num_update_steps_per_epoch = max(num_update_steps_per_epoch, 1)
            num_training_steps = int(num_update_steps_per_epoch * self.args.num_train_epochs)
        # 初始化调度器
        self.lr_scheduler = get_cosine_schedule_with_warmup(
            optimizer = optimizer,                     # 使用传入的或自身的 optimizer
            num_warmup_steps = int(num_training_steps * self.args.warmup_ratio),
            num_training_steps = num_training_steps,
        )
        return self.lr_scheduler

# 初始化 Trainer
trainer = CustomTrainer(
    model = model,
    args = training_args,
    train_dataset = tokenized_dataset,
    data_collator = DataCollatorForLanguageModeling(tokenizer = tokenizer, mlm = False),
)

# 初始化调度器
trainer.create_scheduler()

# 开始训练
print("\n===== 开始训练 ===== ")
trainer.train()

# 保存模型
save_path = "./autodl-tmp/fine_tuned_deepseek"
trainer.save_model(save_path)
tokenizer.save_pretrained(save_path)
print(f"模型已保存到: {save_path}!")
```

下面是测试微调后的大模型的代码：

```
def generate_answer(question, model, tokenizer):
    model.eval()                    # 设置为评估模式
    prompt = f"问:{question}\n答:"
    inputs = tokenizer(prompt, return_tensors = "pt").to(device)
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length = 100,
            temperature = 0.7,
            do_sample = True,
            pad_token_id = tokenizer.eos_token_id
        )
    answer = tokenizer.decode(outputs[0], skip_special_tokens = True)
    return answer

# 测试
test_questions = [
    "什么是防火墙?",
    "如何提高网络安全?",
    "解释一下机器学习的基本概念"
]

print("\n===== 模型测试 =====")
for question in test_questions:
    answer = generate_answer(question, model, tokenizer)
    print(f"\n问题:{question}")
    print(f"回答:{answer.split('答:')[-1]}")
```

下面是代码中使用的网络安全数据集示例。

```
[
    {
        "question": "什么是防火墙?",
        "answer": "防火墙是一种网络安全设备,用于监控和控制网络流量,根据预定义的安全规则允许或阻止数据传输。它可以是硬件设备或软件程序,通常部署在内部网络和外部网络(如互联网)之间,防止未经授权的访问和数据泄露。"
    },
    {
        "question": "简述加密技术的基本原理。",
        "answer": "加密技术是将明文数据转换为密文的过程,通过使用加密算法和密钥来保护信息的机密性。基本原理包括:1) 使用加密算法对明文进行转换;2) 加密和解密过程需要使用密钥;3) 即使密文被截获,没有正确密钥也无法还原为明文。常见的加密算法有对称加密(如 AES)和非对称加密(如 RSA)。"
    },
    {
        "question": "什么是 SQL 注入攻击?如何防范?",
        "answer": "SQL 注入攻击是一种通过在应用程序的输入字段中注入恶意 SQL 代码,来篡改或获取数据库信息的攻击方式。防范措施包括:1) 使用参数化查询(Prepared Statements);2) 对用户输入进行严格的验证和过滤;3) 最小化数据库用户权限;4) 定期更新和维护应用程序和数据库;5) 对敏感数据进行加密存储。"
    },
```

```
    {
        "question": "简述零信任架构的核心原则。",
        "answer": "零信任架构的核心原则是\"默认不信任,始终验证\",具体包括:1) 不信任任何内部或外部的用户、设备和网络流量;2) 对任何试图访问资源的请求都进行严格身份验证和授权;3) 根据实时风险评估动态调整访问权限;4) 最小化权限原则,仅授予用户完成任务所需的最低权限;5) 对所有流量进行加密和监控;6) 持续验证系统和用户的安全性。"
    },
    {
        "question": "什么是双因素认证(2FA)?",
        "answer": "双因素认证(2FA)是一种增强身份验证方法,要求用户从①知识因素(如密码或PIN);②拥有因素(如手机或令牌);③固有因素(如指纹或面部识别)三种不同类型的身份验证因素中选用两种。通过结合两种因素,2FA 显著提高了账户安全性,即使密码泄露,攻击者也无法在没有第二个因素的情况下访问账户。"
    }
]
```

3.5　部分参数微调

部分参数微调是一种通过仅调整模型中的特定子集参数来适配下游任务的方法,其核心在于识别并优先更新对目标任务敏感的参数,从而在保留大部分预训练知识的同时降低计算开销。这种方法特别适用于资源受限的场景,或需要快速迭代开发的项目。

3.5.1　BitFit 方法

BitFit 方法仅微调模型中的偏置项(Bias Terms),而冻结所有权重矩阵。该方法在小规模-中等规模的训练数据上,性能与全参数微调相当,甚至有可能超过全参数微调;在大规模训练数据上,与其他微调方法也差不多。BitFit 方法的优势在于其简单性和高效性,仅调整 0.1%～1%的参数即可达到全参数微调 90%以上的性能。

3.5.2　Fish Mask 方法

Fish Mask 通过动态计算参数重要性得分(如梯度幅值或 Fisher 信息量),选择对当前任务影响最大的参数进行更新,从而避免全局调整的冗余计算。这类方法通常适用于预训练模型与目标任务领域较为接近的场景(如通用文本到垂直领域文本的迁移),其优势在于内存占用可减少 30%～50%,且训练速度更快,适合在单卡 GPU 甚至边缘设备上部署。其局限性在于,如果任务与预训练差异较大(如从代码生成任务转到医疗问答任务),仅调整局部参数可能不足以充分适配,导致性能瓶颈,且参数选择策略需要依赖经验或额外计算(如重要性评分),可能引入新的复杂度。Fish Mask 需要计算所有参数的全梯度,仍然需要消耗大量的训练时间,同时由于原始参数发生了改变,针对不同的下游任务仍需保存整个模型,因此依然存在存储资源占用大的问题,且效果没有全参数微调效果好。

3.5.3　部分参数微调示例

本小节介绍一个使用 BitFit 部分参数微调技术微调 deepseek-r1-distill-qwen-1.5b 模型的示例。和上一个例子一样，使用相同的数据集，实现往模型中注入网络安全专业知识。

首先创建虚拟环境。

```
python3 -m venv deepseek-env
source deepseek-env/bin/activate
```

接着安装必要的 Python 库，包括 PyTorch、Transformers、Datasets 和 PEFT。

```
# 安装 PyTorch(确保与 CUDA 版本匹配)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

# 安装其他依赖
pip install transformers datasets wandb peft accelerate bitsandbytes
```

下面是 BitFit 部分数微调代码示例。

```
import os
import torch
from datasets import load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling,
)
from peft import get_peft_model, LoraConfig, TaskType

# 设置环境变量
os.environ["WANDB_PROJECT"] = "security_knowledge_injection_bitfit"

# 模型和数据路径
model_name = "deepseek-ai/deepseek-r1-distill-qwen-1.5b"
dataset_path = "your_security_dataset_path"          # 替换为实际数据集路径

# 加载模型和分词器
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto",
    trust_remote_code=True
)

# BitFit 方法:冻结所有参数,仅解冻偏置项
```

```
def apply_bitfit(model):
    # 冻结所有参数
    for param in model.parameters():
        param.requires_grad = False
    # 解冻所有偏置项
    for name, param in model.named_parameters():
        if 'bias' in name:
            param.requires_grad = True
    # 打印可训练参数
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    total_params = sum(p.numel() for p in model.parameters())
    print(f"可训练参数: {trainable_params}/{total_params} ({trainable_params/total_params
*100:.2f}%)")
    return model

# 应用 BitFit 方法
model = apply_bitfit(model)

# 加载信息安全概论数据集
def load_security_dataset(path):
    dataset = load_dataset("json", data_files=path)
    return dataset

# 数据预处理函数
def preprocess_function(examples):
    texts = [f"问题:{q}\n答案:{a}" for q, a in zip(examples["question"], examples
["answer"])]
    tokenized = tokenizer(texts, padding="max_length", truncation=True, max_length=512)
    tokenized["labels"] = tokenized["input_ids"].copy()
    return tokenized

# 加载和预处理数据
dataset = load_security_dataset(dataset_path)
tokenized_dataset = dataset.map(preprocess_function, batched=True)

# 训练参数配置
training_args = TrainingArguments(
    output_dir="./results_bitfit",
    learning_rate=1e-4,                    # BitFit 通常使用较高的学习率
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=5,                    # 由于只训练偏置项,可能需要更多轮次
    weight_decay=0.0,                      # 不应用权重衰减到偏置项
    logging_dir="./logs_bitfit",
    logging_steps=10,
    save_strategy="epoch",
    fp16=True,
    gradient_accumulation_steps=4,
    report_to="wandb",
    warmup_ratio=0.1,
    lr_scheduler_type="cosine",
)
```

```
# 创建数据收集器
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)

# 创建 Trainer(使用默认的 AdamW 优化器)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    eval_dataset=tokenized_dataset.get("validation"),
    data_collator=data_collator,
)

# 开始训练
trainer.train()

# 保存微调后的模型
model.save_pretrained("./deepseek-r1-distill-qwen-1.5b-security-bitfit")
tokenizer.save_pretrained("./deepseek-r1-distill-qwen-1.5b-security-bitfit")
```

3.6　新增参数微调

随着模型规模的增大，传统的全参数微调和部分参数微调方法面临两大问题：一是基础模型太大，对于多种下游任务，多个微调后的大模型会占用存储空间；二是微调模型所需要的计算资源较大，全参数微调代价太高。新增参数微调方法可以较好地解决上述两大问题，并且不改变原模型权重，避免模型微调过程中出现灾难式遗忘。

新增参数微调是一种通过向预训练模型中引入少量可训练参数来适配任务的方法，其核心思想是冻结原始模型参数(不更新)，仅通过新增的轻量模块或结构对预训练模型进行干预，从而以极低资源成本实现任务适配。这类方法通常分为两类：模块插入型(如Adapter、LoRA 等)和输入干预型(如 Prefix Tuning、Prompt Tuning 和 P-tuning 等)。

3.6.1　Adapter

2019 年，适配器(Adapter)被提出，其思想是在 Transformer 模块内部增加一块可训练的参数(即 Adapter)。每个 Adapter 都由低秩权重矩阵组成，在 Transformer 层的注意力或前馈网络后插入小型全连接层(例如，将隐藏维度从 d 压缩到 r 再恢复)，仅训练这些插入的结构，保留原模型参数不变，训练参数量占比通常为 1%～5%，适合多任务场景。如图 3.2 所示，Adapter 一般添加在 Transformer 模块中的两个全连接层的后面。每个 Adapter 包括输入层、输出层、下投影前馈层、上投影前馈层、非线性层和从输入到输出的跳接。在训练过程中，一般只调整图中的黑色框部分，包括适配器的下投影前馈层、上投影前馈层、非线性层以及 Transformer 模块中的两个归一层的参数。Adapter 的工作原理是先把输入的 d 维特征向量通过下投影前馈层($d\times r$ 维矩阵)投影为 r 维向量($r\ll d$)，应用非线性层，再通过上投影前馈层($r\times d$ 维矩阵)投影回一个 d 维向量。

开源项目 Adapters(https://github.com/adapter-hub/adapters)已经实现了开箱即用

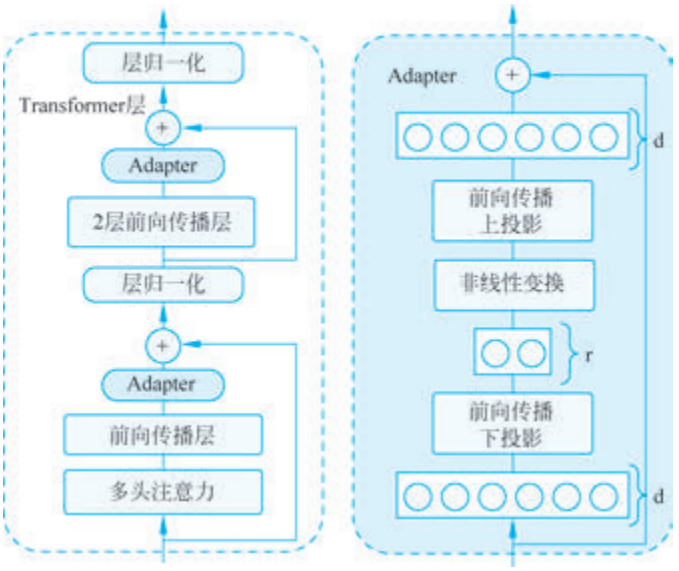

图 3.2　Adapter 模块的结构及其与 Transformer 模块的整合

的 Adapter 微调方法，也实现了本章后面会介绍的 LoRA、Prefix Tuning、Prompt Tuning 等微调方法。基于 Adapters，只需要稍加配置就可以自行训练或加载一些预训练过的 Adapter，还可以实现与 LoRA 等其他微调方法对模型实施混合训练和调用。Adapters 可以和 HuggingFace 的 Transformer 包无缝整合，即可以直接加载 HuggingFace 上的模型进行 Adapter 微调。Adapters 项目组提供并维护了一个使用教程（地址为 https://docs.adapterhub.ml/），读者可以查看 Adapters 最新的介绍。这里介绍一个使用 Adapters 在 deepseek-r1-distill-qwen-1.5b 模型中添加并训练适配器的示例。和前面的例子一样，使用相同数据集，实现往模型中注入网络安全专业知识。

首先创建虚拟环境。

```
python3 -m venv deepseek-env
source deepseek-env/bin/activate
```

接着安装必要的 Python 库，包括 PyTorch、Transformers、Datasets、Accelerate、trl、Adapters。

```
# 安装 PyTorch(确保与 CUDA 版本匹配)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
# 安装其他依赖
pip install transformers datasets accelerate trl adapters
```

下面是使用 Adapter 微调模型的示例。

```
import os
import json
import torch
from transformers import (
    AutoTokenizer,
```

```
    AutoModelForCausalLM,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling,
    get_cosine_schedule_with_warmup
)
from datasets import Dataset
import adapters
from adapters import AdapterModelInterface
from transformers import AutoModelForCausalLM

plugin_interface = AdapterModelInterface(
    # adapter 方法
    adapter_methods = ["lora", "reft", "bottleneck"],
    # 映射模型组件和 adapter 接口
    model_embeddings = "embed_tokens",              # 嵌入层
    model_layers = "layers",                        # Transformer 层
    layer_self_attn = "self_attn",                  # 每层的自注意模块
    layer_cross_attn = None,                        # Qwen 模型没有 cross-attention
    # 在注意模块中投影矩阵
    attn_k_proj = "k_proj",                         # Key 投影
    attn_q_proj = "q_proj",                         # Query 投影
    attn_v_proj = "v_proj",                         # Value 投影
    attn_o_proj = "o_proj",                         # Output 投影
    # 多层感知机投影
    layer_intermediate_proj = "mlp.up_proj",        # 多层感知机中的上投影
    layer_output_proj = "mlp.down_proj",            # 多层感知机中的下投影
    layer_pre_self_attn = "input_layernorm",        # 在自注意力前直接 Hook
    layer_pre_ffn = "post_attention_layernorm",     # 在多层感知机前直接 Hook
    #Qwen 在注意力和多层感知机前应用 norm 层,所以不需要再增加
    layer_ln_1 = None,
    layer_ln_2 = None,
)
# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")
# 设置环境变量
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"            # 便于调试 CUDA 错误
# 模型和数据路径
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"
data_path = "./data/data.json"                      # 示例数据集路径
# 加载 tokenizer 和模型
print(f"加载模型: {model_name}")
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token           # 设置填充标记
# 加载数据
def load_data(file_path):
    print(f"加载数据: {file_path}")
    try:
        with open(file_path, "r", encoding = "utf-8") as f:
            data = json.load(f)
    except Exception as e:
        print(f"加载数据时出错: {e}")
```

```
        # 创建示例数据用于测试
        print("创建示例数据用于测试…")
        data = [
            {"question": "什么是人工智能?", "answer": "人工智能是指计算机系统能够执行通常需要人类智能才能完成的任务的能力。"},
            {"question": "什么是深度学习?", "answer": "深度学习是机器学习的一个分支领域,它使用多层神经网络来学习数据的表示和特征。"}
        ]
    print(f"已加载 {len(data)} 条数据")
    return data

# 处理数据
def process_data(examples):
    # 构建指令模板
    prompts = []
    for q, a in zip(examples["question"], examples["answer"]):
        prompt = f"问:{q}\n答:{a}{tokenizer.eos_token}"
        prompts.append(prompt)
    # 编码文本
    tokenized = tokenizer(
        prompts,
        truncation=True,
        padding="max_length",
        max_length=512,
        return_tensors="pt"
    )
    # 对于自监督学习,标签与输入相同
    tokenized["labels"] = tokenized["input_ids"].clone()
    return tokenized

# 加载并处理数据集
data = load_data(data_path)
dataset = Dataset.from_list(data)
tokenized_dataset = dataset.map(
    process_data,
    batched=True,
    remove_columns=dataset.column_names
)

# 加载模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype="bfloat16",          # 使用半精度
)

# 设置 pad token ID 不同于模型的 EOS token
tokenizer.pad_token_id = 151645
model.config.pad_token_id = tokenizer.pad_token_id

# 用 plugin interface 初始化 adapter 框架
adapters.init(model, interface=plugin_interface)

# 增加 LoRA adapter
```

```
from adapters import SeqBnConfig
adapter_name = "qwen-adapter"
adapter_config = SeqBnConfig(
    reduction_factor=32,            # Bottleneck 大小
)
model.add_adapter(adapter_name, adapter_config)

# 激活 adapter
model.set_active_adapters(adapter_name)

# 设置模型只训练 adapter 参数
model.train_adapter(adapter_name)

# 打印 adapter,检验 adapter 是否成功
print(model.adapter_summary())

# 禁止 wandb
import wandb
wandb.init(project="config_example", mode="disabled")

from transformers import TrainingArguments
import numpy as np
# Set up training arguments
training_args = TrainingArguments(
    output_dir="./qwen-adapter",
    per_device_train_batch_size=2,          # 根据 GPU 显存大小设置
    per_device_eval_batch_size=2,
    learning_rate=1e-4,
    num_train_epochs=1,                     # 任务越复杂,值越大
    save_steps=30,
    eval_steps=30,
    logging_steps=10,
    metric_for_best_model="loss",           # 使用 loss 作用指标选择最好的模型
    greater_is_better=False,
    push_to_hub=False,
    gradient_accumulation_steps=8,          # 累积梯度
    bf16=True,                              # 使用混合精度
    )

# 初始化 trainer
from adapters import AdapterTrainer
from trl import DataCollatorForCompletionOnlyLM
trainer = AdapterTrainer(
    model=model,
    processing_class=tokenizer,
    args=training_args,
data_collator=DataCollatorForCompletionOnlyLM(response_template="Answer:", tokenizer=
tokenizer),
    train_dataset=tokenized_dataset,
)
# 训练
trainer.train()

# 只保存 adapter 的权重
model.save_adapter("./qwen-adapter", adapter_name)
```

3.6.2　LoRA

3.6.1 节中的 Adapter 方法增加了模型参数，改变了模型结构，这会导致模型训练、推理的计算成本和内存占用的增加。由于深度神经网络含有大量全连接层，这些参数矩阵通常是满秩的，根据流行嵌入假说，在适配具体下游任务时，微调得到的大模型的参数矩阵处在一个较低的内在维度上。因此，微调大模型只需要改变少部分的参数即可唤醒预训练模型对特定下游任务的适配。为此，微软研究团队在 2021 年推出 LoRA 方法，在预训练模型旁边增加一个旁路，通过进行降维再升维的操作来模拟内秩。具体做法如图 3-3 所示，在 Transformer 层的权重矩阵旁添加低秩矩阵，如原参数矩阵维度为 $d\times d$，LoRA 将其分解为 $d\times r$ 和 $r\times d$ 两个矩阵的乘积，其中 $r\ll d$。微调时固定预训练模型的参数，用随机高斯分布初始化 $\boldsymbol{A}$，用 $\boldsymbol{0}$ 矩阵初始化 $\boldsymbol{B}$（保证训练开始时，旁路矩阵是 $\boldsymbol{0}$ 矩阵），只训练降维矩阵 $\boldsymbol{A}$ 与升维矩阵 $\boldsymbol{B}$，训练完毕后，将两个矩阵的乘积 $\boldsymbol{BA}$ 与预训练模型的参数叠加使用。

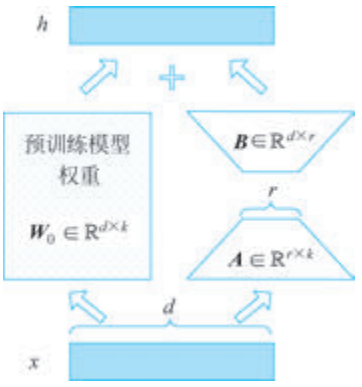

图 3.3　LoRA 微调示意图

假设要在下游任务微调一个预训练语言模型，需要更新预训练模型参数，公式为 $\boldsymbol{W}_0+\Delta\boldsymbol{W}$。其中，$\boldsymbol{W}_0$ 是预训练模型初始化的参数，$\Delta\boldsymbol{W}$ 是待更新的参数，如果是全参数微调，$\Delta\boldsymbol{W}$ 的参数量等于 $\boldsymbol{W}_0$。预训练的参数矩阵为 $\boldsymbol{W}_0\in\mathbb{R}^{d\times k}$。模型参数的更新可表示为 $\boldsymbol{W}_0+\Delta\boldsymbol{W}=\boldsymbol{W}_0+\boldsymbol{BA}$，其中 $\boldsymbol{B}\in\mathbb{R}^{d\times r}$，$\boldsymbol{A}\in\mathbb{R}^{r\times k}$，$r\ll\min(d,k)$。在前向过程中，$\boldsymbol{W}_0$ 和 $\Delta\boldsymbol{W}$ 乘以相同的输入 x，两者相加：$\boldsymbol{h}=\boldsymbol{W}_0x+\Delta\boldsymbol{W}x=\boldsymbol{W}_0x+\boldsymbol{BA}x$。当 $r=k$ 时，LoRA 微调等同于全参数微调。

目前，LoRA 已被 Hugging Face 集成到 PEFT（Parameter-Efficient Fine-Tuning）代码库中，使用非常方便。这里介绍一个使用 LoRA 微调 deepseek-r1-distill-qwen-1.5b 模型的示例。和上面的例子一样，使用相同的数据集，实现往模型中注入网络安全专业知识。

首先创建虚拟环境。

```
python3 -m venv deepseek-env
source deepseek-env/bin/activate
```

接着安装必要的 Python 库，包括 PyTorch、Transformers、Datasets、Accelerate、Peft。

```
# 安装 PyTorch(确保与 CUDA 版本匹配)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/
whl/cu118
# 安装其他依赖
pip install transformers datasets accelerate peft
```

下面是使用 LoRA 微调模型的示例。

```
import os
import torch
```

```
import json
from datasets import load_dataset, Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling,
)
from peft import (
    get_peft_model,
    LoraConfig,
    TaskType,
    prepare_model_for_kbit_training,
)
from tqdm.auto import tqdm

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")
# 设置环境变量
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"                    # 便于调试 CUDA 错误
# 模型和数据路径
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"
data_path = "./data/data.json"                              # 示例数据集路径

# 加载 tokenizer 和模型
print(f"加载模型: {model_name}")
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token                   # 设置填充标记

# 以 FP32 精度加载模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto",                                      # 自动处理多 GPU 分布
)

# 验证模型参数类型
print("\n检查模型参数类型:")
for name, param in list(model.named_parameters())[:5]:      # 只打印前 5 个参数
    print(f" {name}: {param.dtype}")

# 打印模型信息
print(f"\n模型参数数量: {sum(p.numel() for p in model.parameters())}")
print(f"使用的优化器: SGD (全参数微调)")

# 加载数据
def load_data(file_path):
    print(f"加载数据: {file_path}")
    try:
        with open(file_path, "r", encoding="utf-8") as f:
            data = json.load(f)
```

```
    except Exception as e:
        print(f"加载数据时出错：{e}")
        # 创建示例数据用于测试
        print("创建示例数据用于测试…")
        data = [
            {"question": "什么是人工智能?", "answer": "人工智能是指计算机系统能够执行通
常需要人类智能才能完成的任务的能力。"},
            {"question": "什么是深度学习?", "answer": "深度学习是机器学习的一个分支领域,
它使用多层神经网络来学习数据的表示和特征。"}
        ]
    print(f"已加载 {len(data)} 条数据")
    return data

# 处理数据
def process_data(examples):
    # 构建指令模板
    prompts = []
    for q, a in zip(examples["question"], examples["answer"]):
        prompt = f"问:{q}\n答:{a}{tokenizer.eos_token}"
        prompts.append(prompt)
    # 编码文本
    tokenized = tokenizer(
        prompts,
        truncation=True,
        padding="max_length",
        max_length=512,
        return_tensors="pt"
    )
    # 对于自监督学习,标签与输入相同
    tokenized["labels"] = tokenized["input_ids"].clone()
    return tokenized

# 加载并处理数据集
data = load_data(data_path)
dataset = Dataset.from_list(data)
print("处理数据集…")
tokenized_dataset = dataset.map(
    process_data,
    batched=True,
    remove_columns=dataset.column_names
)

import wandb
wandb.init(project="config_example", mode="disabled")
# 配置 LoRA
def setup_lora(model):
    # 冻结所有模型参数
    for param in model.parameters():
        param.requires_grad = False

    # 配置 LoRA
    config = LoraConfig(
        task_type=TaskType.CAUSAL_LM,
```

```
        r = 8,                                          # LoRA 秩
        lora_alpha = 32,                                # LoRA 缩放因子
        target_modules = ["q_proj", "v_proj"],          # 要应用 LoRA 的模块
        lora_dropout = 0.1,
        bias = "none",                                  # 不训练偏置项
        inference_mode = False
    )
    # 准备模型进行 LoRA 训练
    model = get_peft_model(model, config)
    # 打印可训练参数
    model.print_trainable_parameters()
    return model

# 应用 LoRA 配置
model = setup_lora(model)

# 加载信息安全概论数据集
def load_security_dataset(path):
    dataset = load_dataset("json", data_files = path)
    return dataset

# 训练参数配置
training_args = TrainingArguments(
    output_dir = "./autodl - tmp/results_LoRA",
    learning_rate = 3e - 4,
    per_device_train_batch_size = 4,
    per_device_eval_batch_size = 4,
    num_train_epochs = 1,
    weight_decay = 0.01,
    logging_dir = "./autodl - tmp/logs_lora",
    logging_steps = 10,
    save_strategy = "epoch",
    fp16 = True,
    gradient_accumulation_steps = 4,
    warmup_ratio = 0.1,
    lr_scheduler_type = "cosine",
)

# 创建数据收集器
data_collator = DataCollatorForLanguageModeling(tokenizer = tokenizer, mlm = False)

# 创建 Trainer
trainer = Trainer(
    model = model,
    args = training_args,
    train_dataset = tokenized_dataset,
    data_collator = data_collator,
)

# 训练
trainer.train()

# 保存最终模型
```

```
save_path = "./autodl-tmp/fine_tuned_deepseek_LoRA"
trainer.save_model(save_path)
tokenizer.save_pretrained(save_path)
```

在使用模型推理时需要加载 LoRA 权重，下面是调用模型的示例。

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import PeftModel, PeftConfig

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")

# 加载配置
peft_model_id = "./autodl-tmp/fine_tuned_deepseek_LoRA"      # LoRA 权重路径
config = PeftConfig.from_pretrained(peft_model_id)

# 加载基础模型
model = AutoModelForCausalLM.from_pretrained(
    config.base_model_name_or_path,
    torch_dtype=torch.float16,
    device_map="auto",
    trust_remote_code=True
)

# 加载 LoRA 权重
model = PeftModel.from_pretrained(model, peft_model_id)

# 加载分词器
tokenizer = AutoTokenizer.from_pretrained(config.base_model_name_or_path)
tokenizer.pad_token = tokenizer.eos_token

# 设置为评估模式
model.eval()

# 简单测试微调后的模型
def generate_answer(question, model, tokenizer):
    model.eval()              # 设置为评估模式
    prompt = f"问:{question}\n答:"
    inputs = tokenizer(prompt, return_tensors="pt").to(device)
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=100,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )
    answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return answer

# 测试
```

```
test_questions = [
    "什么是防火墙?"
]
print("\n===== 模型测试 =====")
for question in test_questions:
    answer = generate_answer(question, model, tokenizer)
    print(f"\n问题:{question}")
    print(f"回答:{answer.split('答:')[-1]}")
```

3.6.3　Prefix Tuning

传统提示工程是在模型输入上加提示词，再对模型输出进行映射。这种离散文本输入方式的健壮性较差。Prefix Tuning 采用连续输入方式来解决此问题，在模型的 Transformer 层的输入前添加一个称为前缀(Prefix)的连续的且任务特定的向量序列，如图 3.4 所示。微调时固定预训练模型的所有参数，只更新优化特定任务的前缀。除上述优点外，与传统微调方法相比，该方法还有其他好处：一是对不同的任务，只需训练和保存对应的前缀，大幅降低了存储和推理开销；二是可以为不同的用户训练前缀，实现不同用户间的训练数据隔离；三是可以实现在一个批次内训练多个用户/任务数据。

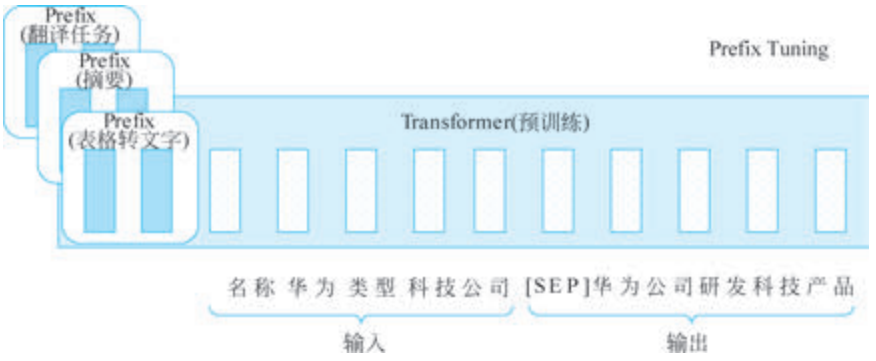

图 3.4　Prefix Tuning 示意图

如图 3.5 所示，Prefix Tuning 方法描述基于 Transformer 架构的自回归语言模型 $p_\phi(y|x)$。$z=[x;y]$表示拼接 x 和 y。X_{idx} 表示与 x 对应的索引，Y_{idx} 表示与 y 对应的索引。在时间步 i，激活态为 $\boldsymbol{h}_i \in \mathbb{R}^d$，其中 $\boldsymbol{h}_i=[h_i^{(1)},h_i^{(2)},\cdots,h_i^{(n)}]$，$h_i^{(j)}$ 是在第 i 个时间步、第 j 个 Transformer 层的激活态。在自回归模型中，$\boldsymbol{h}_i=\text{LM}_\phi(z_i,h_{<i})$。其中，$\boldsymbol{h}_i$ 的最后一层用来计算模型下一个 Token 的分布：$p_\phi(z_{i+1}|h_{\leqslant i})=\text{softmax}(W_\phi \boldsymbol{h}_i^{(n)})$。$P_{\text{idx}}$ 表示 Prefix 的索引，那么 $|P_{\text{idx}}|$ 就是 Prefix 的长度。Prefix Tuning 初始化训练矩阵 $\boldsymbol{P}_\theta$(维度为 $|P_{\text{idx}}|\times\dim(\boldsymbol{h}_i)$)，其中，$\boldsymbol{h}_i=\begin{cases}\boldsymbol{P}_\theta[i,:], & i\in P_{\text{idx}},\\ \text{LM}_\phi(z_i,h_{<i}), & \text{其他}\end{cases}$。模型训练的目标为 $\max\limits_{\phi}\log p_\phi(y|x)=\sum\limits_{i\in Y_{\text{idx}}}\log p_\phi(z_i,h_{<i})$。这里 $\boldsymbol{h}_i$ 是训练矩阵 $\boldsymbol{P}_\theta$ 的函数。

如果微调只有解码器的大模型(如 GPT)，Prefix 只加在句首，模型的输入表示为 $z=[\text{PREFIX};x;y]$；如果微调既有编码器又有解码器的大模型(如 BERT)，不同 Prefix 分别

加到编解码器的开头：$z=[\text{PREFIX};x;\text{PREFIX}';y]$。微调时，冻结预训练大模型的参数只更新 Prefix 部分的参数。

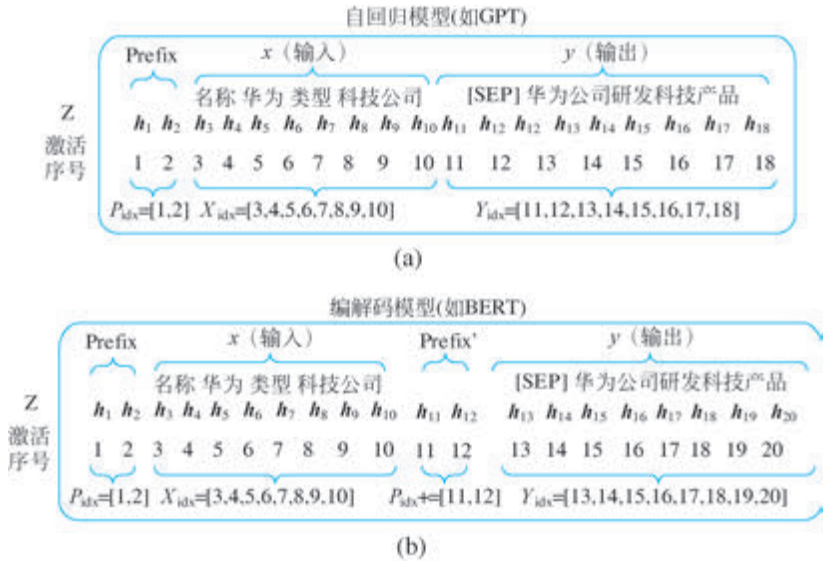

图 3.5　Prefix Tuning 的两个例子

具体实现是将前缀参数(可训练的张量)添加到所有的 Transformer 层，即将前缀向量与多注意力的 $\boldsymbol{K}$ 矩阵和 $\boldsymbol{V}$ 矩阵相乘，将相乘结果分别和预训练模型的 K 值和 V 值拼接，再计算注意力。

该方法其实和构造 Prompt 类似，只是 Prompt 是人为构造的“显示”提示，并且无法更新参数，而前缀完全由自由参数组成，可以学习“隐式”提示。相比之下，Prefix Tuning 只优化了前缀。因此，微调完成后，只需要存储预训练模型和已知任务特定前缀的副本，这使得在每个特定任务上的开销非常小。

为了防止直接更新 Prefix 的参数导致训练不稳定和性能下降的情况，如图 3.6 所示，可以在 Prefix 层前面加多层感知机(MLP)，重参数 $P_\theta[i,:]=\text{MLP}_\theta(P'_\theta[i,:])$。注意，$P_\theta$ 和 P'_θ 的行数相同(为 $|P_{\text{idx}}|$)，但列数不同(P_θ 的列数为 $\dim(h_i)$，P'_θ 的列数为 k，$k<\dim(h_i)$)。训练完成后，丢掉多层感知机，只保留 Prefix 的参数。

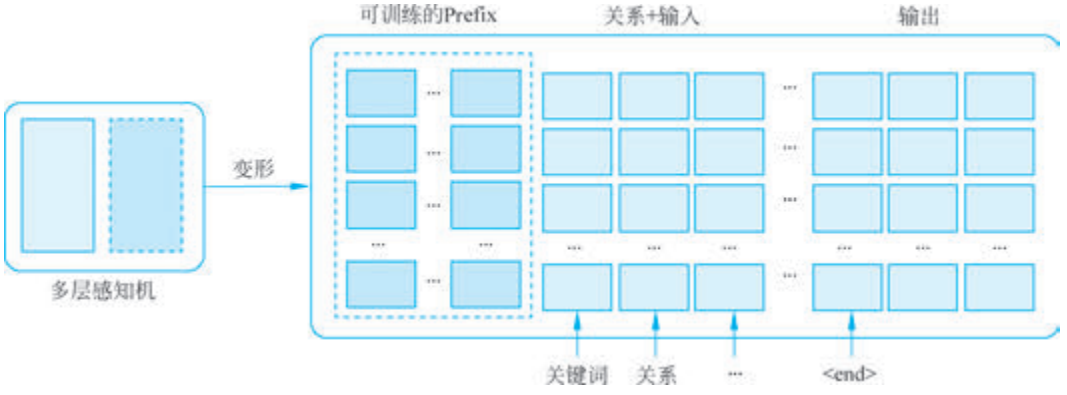

图 3.6　Prefix 层前加 MLP

需要注意两点：一是 Prefix 部分使用 Token 的数量直接影响模型微调的参数量级，以及处理长文本的能力，一般默认长度为 10；二是 Prefix Tuning 训练比较困难，且预留的向量序列挤占了下游任务的输入序列空间，在一定程度上影响了模型性能。

这里介绍一个使用 Prefix Tuning 微调 deepseek-r1-distill-qwen-1.5b 模型的示例。和上面的例子一样，使用相同的数据集实现往模型中注入网络安全专业知识。

首先创建虚拟环境。

```
python3 -m venv deepseek-env
source deepseek-env/bin/activate
```

接着安装必要的 Python 库，包括 PyTorch、Transformers、Datasets。

```
# 安装 PyTorch(确保与 CUDA 版本匹配)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
# 安装其他依赖
pip install transformers datasets
```

下面是使用 Prefix 微调模型的示例。

```
import os
import torch
import json
from datasets import load_dataset, Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
)
import torch.nn as nn                                    # 导入 PyTorch 神经网络模块

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")

# 设置环境变量
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"                 # 便于调试 CUDA 错误

# 模型和数据路径
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"
data_path = "./data/data.json"                           # 示例数据集路径

# 加载 tokenizer 和模型
print(f"加载模型: {model_name}")
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token                # 设置填充标记

# 加载模型(使用 Float16 精度)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,                           # 模型参数使用半精度
    device_map="auto",
```

```
)
model.requires_grad_(False)                    # 冻结模型参数
model = model.to(device)

# 验证模型参数类型
print("\n检查模型参数类型:")
for name, param in list(model.named_parameters())[:5]:
    print(f" {name}: {param.dtype}")

# 加载数据
def load_data(file_path):
    print(f"加载数据: {file_path}")
    try:
        with open(file_path, "r", encoding = "utf-8") as f:
            data = json.load(f)
    except Exception as e:
        print(f"加载数据时出错: {e}")
        print("创建示例数据用于测试…")
        data = [
            {"question": "什么是人工智能?", "answer": "人工智能是指计算机系统能够执行通
常需要人类智能才能完成的任务的能力。"},
            {"question": "什么是深度学习?", "answer": "深度学习是机器学习的一个分支领域,
它使用多层神经网络来学习数据的表示和特征。"}
        ]
    print(f"已加载 {len(data[:1000])} 条数据")
    return data[:1000]

# 处理数据
def process_data(examples):
    prompts = []
    for q, a in zip(examples["question"], examples["answer"]):
        prompt = f"问:{q}\n答:{a}{tokenizer.eos_token}"
        prompts.append(prompt)

    tokenized = tokenizer(
        prompts,
        truncation = True,
        padding = "max_length",
        max_length = 512,
        return_tensors = "pt"
    )
    tokenized["labels"] = tokenized["input_ids"].clone()
    return tokenized

# 加载并处理数据集
data = load_data(data_path)
dataset = Dataset.from_list(data)
print("处理数据集…")
tokenized_dataset = dataset.map(
    process_data,
    batched = True,
    remove_columns = dataset.column_names
)
```

```
import wandb
wandb.init(project="config_example", mode="disabled")

# 定义 MLP 层(仅训练时使用,最终会被丢弃)
class MLP(nn.Module):
    def __init__(self, hidden_size, mlp_ratio=4.0):
        super().__init__()
        self.hidden_size = hidden_size
        self.intermediate_size = int(512 * mlp_ratio)
        self.dense_in = nn.Linear(512, self.intermediate_size, dtype=torch.float16)
        self.act = nn.GELU()
        self.dense_out = nn.Linear(self.intermediate_size, hidden_size, dtype=torch.float16)
        self.projection = nn.Linear(hidden_size, 512, dtype=torch.float16)  # 从 hidden_
# size 映射到 512
    def forward(self, x):
        x_projected = self.projection(x)                # 映射到 512 维
        x = self.dense_in(x_projected)
        x = self.act(x)
        x = self.dense_out(x)
        return x

# 定义 PrefixTuningWithMLP(训练时用)
class PrefixTuningWithMLP(nn.Module):
    def __init__(self, model, prefix_length=10, hidden_size=None, mlp_ratio=4.0):
        super().__init__()
        self.prefix_length = prefix_length
        self.hidden_size = hidden_size or model.config.hidden_size
        # 可学习的原始前缀向量(未经过 MLP)
        self.raw_prefix = nn.Parameter(
            torch.randn(1, prefix_length, self.hidden_size, dtype=torch.float16)
        )
        # MLP 层(仅训练时用于优化前缀)
        self.mlp = MLP(self.hidden_size, mlp_ratio)

    def forward(self, attention_mask, input_embeddings):
        batch_size = attention_mask.shape[0]
        # 训练时:用 MLP 优化前缀向量
        optimized_prefix = self.mlp(self.raw_prefix)    # 经过 MLP 的前缀(训练核心)
        optimized_prefix = optimized_prefix.expand(batch_size, -1, -1)     # 匹配批次大小

        # 处理注意力掩码
        attention_mask = attention_mask.to(
            device=input_embeddings.device,
            dtype=torch.float16
        )
        if attention_mask.dim() == 1:
            attention_mask = attention_mask.unsqueeze(1)
        # 处理输入嵌入
        if input_embeddings.dim() == 2:
            input_embeddings = input_embeddings.unsqueeze(1)
        input_embeddings = input_embeddings.to(dtype=torch.float16)
        # 拼接前缀和输入
        input_with_prefix = torch.cat([optimized_prefix, input_embeddings], dim=1)
```

```
        # 生成前缀掩码
        prefix_attention_mask = torch.ones(
            batch_size,
            self.prefix_length,
            device = input_embeddings.device,
            dtype = torch.float16
        )
        new_attention_mask = torch.cat([prefix_attention_mask, attention_mask], dim = 1)
        return new_attention_mask, input_with_prefix
    def get_final_prefix(self):
        """训练完成后:获取经过 MLP 优化后的最终前缀向量(丢弃 MLP)"""
        with torch.no_grad():                                    # 不追踪梯度
            final_prefix = self.mlp(self.raw_prefix)     # 用训练好的 MLP 计算最终前缀
        return final_prefix  # 仅返回优化后的前缀向量(shape: [1, prefix_length, hidden_size])

# 初始化前缀调优模块
prefix_tuning = PrefixTuningWithMLP(model, prefix_length = 10, hidden_size = model.config.
hidden_size)
prefix_tuning = prefix_tuning.to(device, dtype = torch.float16)

# 定义前向过程
def forward_with_prefix(input_ids, attention_mask, labels = None):
    input_ids = input_ids.to(device, dtype = torch.long)
    attention_mask = attention_mask.to(device, dtype = torch.float16)
    labels = labels.to(device, dtype = torch.long) if labels is not None else None
    if input_ids.dim() == 1:
        input_ids = input_ids.unsqueeze(0)
    # 处理标签
    if labels is not None:
        if labels.dim() == 1:
            labels = labels.unsqueeze(0)
        seq_len_with_prefix = prefix_tuning.prefix_length + input_ids.shape[1]
        if labels.shape[1] != seq_len_with_prefix:
            prefix_labels = torch.full(
                (labels.shape[0], prefix_tuning.prefix_length),
                -100,
                dtype = torch.long,
                device = device
            )
            labels = torch.cat([prefix_labels, labels], dim = 1)
    # 获取输入嵌入
    input_embeddings = model.get_input_embeddings()(input_ids).to(dtype = torch.float16)
    # 获取带前缀的输入和掩码
    new_attention_mask, input_with_prefix = prefix_tuning(attention_mask, input_
embeddings)
    # 模型前向传播
    outputs = model(
        inputs_embeds = input_with_prefix,
        attention_mask = new_attention_mask,
        labels = labels
    )
    return outputs
```

```
# 训练
from torch.optim import AdamW
import tqdm
optimizer = AdamW(prefix_tuning.parameters(), lr = 1e - 5, eps = 1e - 4)
save_path = "./autodl - tmp/fine_tuned_deepseek_Prefix_MLP"
os.makedirs(save_path, exist_ok = True)
num_epochs = 1
log_interval = 10
best_loss = float('inf')
for epoch in range(num_epochs):
    total_loss = 0.0
    progress_bar = tqdm.tqdm(
        enumerate(tokenized_dataset),
        total = len(tokenized_dataset),
        desc = f"Epoch {epoch + 1}/{num_epochs}"
    )
    for batch_idx, batch in progress_bar:
        # 加载批次数据
        input_ids = torch.tensor(batch["input_ids"], device = device, dtype = torch.long)
        attention_mask = torch.tensor(batch["attention_mask"], device = device, dtype =
torch.float16)
        labels = torch.tensor(batch["labels"], device = device, dtype = torch.long)
        # 确保批次维度
        if input_ids.dim() == 1:
            input_ids = input_ids.unsqueeze(0)
            attention_mask = attention_mask.unsqueeze(0)
            labels = labels.unsqueeze(0)
        optimizer.zero_grad()
        outputs = forward_with_prefix(input_ids, attention_mask, labels)
        loss = outputs.loss
        current_loss = loss.item()
        loss.backward()
        optimizer.step()
        total_loss += current_loss
        # 打印中间结果
        if (batch_idx + 1) % log_interval == 0:
            avg_loss = total_loss / log_interval
            print(f"Batch {batch_idx + 1}, Loss: {current_loss:.4f}, Avg Loss: {avg_loss:.4f}")
            total_loss = 0.0
        # 保存最优模型(含 MLP,用于断点续训)
        if current_loss < best_loss:
            best_loss = current_loss
            checkpoint = {
                # 含 raw_prefix 和 MLP 参数
                "prefix_tuning_state_dict": prefix_tuning.state_dict(),
                "optimizer_state_dict": optimizer.state_dict(),
                "best_loss": best_loss
            }
            torch.save(checkpoint, os.path.join(save_path, f"best_checkpoint_loss_{best_
loss:.4f}.pt"))

    # 保存每轮检查点(用于续训)
    torch.save({
```

```
            "prefix_tuning_state_dict": prefix_tuning.state_dict(),
            "optimizer_state_dict": optimizer.state_dict(),
            "epoch": epoch
        }, os.path.join(save_path, f"epoch_{epoch + 1}_checkpoint.pt"))
        print(f"Epoch {epoch + 1} 检查点已保存")

# 训练完成:丢弃 MLP,仅保存经过优化的最终前缀向量
final_prefix = prefix_tuning.get_final_prefix()    # 核心:获取优化后的前缀(无 MLP)
final_prefix_path = os.path.join(save_path, "final_optimized_prefix.pt")
torch.save(final_prefix, final_prefix_path)        # 仅保存前缀向量(可直接用于推理)
print(f"训练完成!已丢弃 MLP 结构,仅保存优化后的前缀向量至:{final_prefix_path}")
print(f"最终前缀向量形状:{final_prefix.shape}")     # 应输出 [1, prefix_length, hidden_size]
```

下面是调用 Prefix 微调模型的示例。

```
import os
import torch
import json
from datasets import load_dataset, Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling,
)
import torch.nn as nn                              # 导入 PyTorch 神经网络模块

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")

# 设置环境变量
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"       # 便于调试 CUDA 错误
os.environ['TORCH_USE_CUDA_DSA'] = "1"         # 启用设备端断言详情

# 模型和数据路径
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"

# 加载 tokenizer 和模型
print(f"加载模型: {model_name}")
tokenizer = AutoTokenizer.from_pretrained(model_name)
# 确保 tokenizer 有 pad_token
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

# 以 FP32 精度加载模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto",                             # 自动处理多 GPU 分布
)
model.to(device)
```

```
model.requires_grad_(False)              # 冻结模型参数

# 定义 PrefixTuning 类(不含 MLP,用于推理)
class PrefixTuning(nn.Module):
    def __init__(self, model, prefix_length=10, hidden_size=None):
        super().__init__()
        self.prefix_length = prefix_length
        self.hidden_size = hidden_size or model.config.hidden_size
        # 直接定义优化后的前缀向量(无 MLP)
        self.prefix_embeddings = nn.Parameter(
            torch.randn(1, prefix_length, self.hidden_size, dtype=torch.float16)
        )
    def forward(self, attention_mask, input_embeddings):
        batch_size = attention_mask.shape[0]
        prefix = self.prefix_embeddings.expand(batch_size, -1, -1)
        # 处理注意力掩码和输入嵌入
        attention_mask = attention_mask.to(
            device=input_embeddings.device,
            dtype=torch.float16
        )
        if attention_mask.dim() == 1:
            attention_mask = attention_mask.unsqueeze(1)
        input_embeddings = input_embeddings.to(dtype=torch.float16)
        if input_embeddings.dim() == 2:
            input_embeddings = input_embeddings.unsqueeze(1)
        # 拼接前缀和输入
        input_with_prefix = torch.cat([prefix, input_embeddings], dim=1)
        # 生成前缀掩码
        prefix_attention_mask = torch.ones(
            batch_size,
            self.prefix_length,
            device=input_embeddings.device,
            dtype=torch.float16
        )
        new_attention_mask = torch.cat([prefix_attention_mask, attention_mask], dim=1)
        return new_attention_mask, input_with_prefix

# 加载模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto",
)
model = model.to(device)

# 创建 PrefixTuning 实例
prefix_tuning = PrefixTuning(model, prefix_length=10)
prefix_tuning = prefix_tuning.to(device)

# 加载最终的前缀参数(仅加载需要的键)
state_dict = torch.load(
    "./autodl-tmp/fine_tuned_deepseek_Prefix_MLP/final_optimized_prefix.pt",
    map_location=device,
```

```
    weights_only = True                    # 关键:启用安全加载模式
)

# 如果保存的是完整的 state_dict(包含 MLP),则过滤出需要的键
if isinstance(state_dict, dict):
    # 提取需要的键(只保留前缀向量)
    filtered_state_dict = {k: v for k, v in state_dict.items() if "prefix_embeddings" in k}
    prefix_tuning.load_state_dict(filtered_state_dict, strict = False)
else:
    # 如果保存的是直接的前缀张量
    prefix_tuning.prefix_embeddings.data = state_dict
print("前缀参数加载成功!")
prefix_tuning.to(device)
prefix_tuning.eval()
model.eval() # 设置为评估模式
def generate_with_prefix(model, tokenizer, prefix_tuning, prompt_text, max_length = 200):
    """使用 Prefix Tuning 生成回答"""
    # 准备输入
    inputs = tokenizer(prompt_text, return_tensors = "pt").to(device)

    # 手动添加前缀
    batch_size = inputs["input_ids"].shape[0]

    # 获取优化后的前缀(无 MLP)
    prefix = prefix_tuning.prefix_embeddings.expand(batch_size, -1, -1)

    # 处理注意力掩码
    attention_mask = inputs["attention_mask"].to(dtype = torch.float16)
    if attention_mask.dim() == 1:
        attention_mask = attention_mask.unsqueeze(1)

    # 扩展注意力掩码以包含前缀
    prefix_attention_mask = torch.ones(
        batch_size,
        prefix_tuning.prefix_length,
        device = device,
        dtype = torch.float16
    )
    new_attention_mask = torch.cat([prefix_attention_mask, attention_mask], dim = 1)

    # 获取输入嵌入并添加前缀
    input_embeddings = model.get_input_embeddings()(inputs["input_ids"]).to(dtype = torch.float16)
    input_with_prefix = torch.cat([prefix, input_embeddings], dim = 1)

    # 使用模型生成回答
    with torch.no_grad():
        outputs = model.generate(
            inputs_embeds = input_with_prefix,
            attention_mask = new_attention_mask,
            max_length = max_length,
            num_return_sequences = 1,
            temperature = 0.7,
            do_sample = True
```

```
            )

        # 处理生成的输出
        # 注意:outputs 包含整个序列(前缀 + prompt + 生成的回答)
        # 但由于我们使用了 inputs_embeds,生成的 outputs 只包含 prompt + 生成的回答

        # 解码生成的序列
        generated_text = tokenizer.decode(outputs[0], skip_special_tokens = True)

        # 提取回答部分(根据实际提示格式调整)
        # 例如,如果提示格式是"问:{问题}\n 答:",则回答从"答:"之后开始
        answer_start = generated_text.find("答:") + 2      # +2 跳过"答:"
        answer = generated_text[answer_start:]

        return answer

    # 示例推理
    prompt = "问:防火墙是什么?\n 答:"
    answer = generate_with_prefix(model, tokenizer, prefix_tuning, prompt)
    print("生成的回答:", answer)
```

3.6.4　Prompt Tuning

人工设计离散的提示词,需要领域专家手动设计,这种做法工作量大且耗时,成本高,并且可能无法充分捕捉任务的本质特征,导致模型性能不佳,而且针对不同任务需要重新设计提示词,无法进行有效复用。Prompt Tuning 通过反向传播更新参数来学习提示词,而不是人工设计提示词,使模型能够更好地适应特定任务;同时冻结模型原始权重,只训练提示词参数,减少计算开销。训练完以后,用同一个模型通过不同的提示词可以适应多种任务,从而提高了模型的泛化能力。该方法可以看作是 Prefix Tuning 的简化版本(只在模型输入层加前缀),同样冻结整个预训练模型,给每个任务定义 Soft Prompt,拼接到数据上作为输入。

在相同数据上集成不同神经模型可以提升模型预测准确性、增强模型稳定性并降低过拟合风险。然而,随着模型规模的扩大,模型集成变得困难重重。除了需要存储多个模型的空间外,还需要运行多个不同模型的推理成本。Prompt Tuning 为集成预训练大语言模型的多个适配版本提供了一种有效的方法。如图 3.7 所示,通过对同一任务训练 N 个提示,为任务创建 N 个独立的“模型”,同时还共享预训练大语言模型的参数。

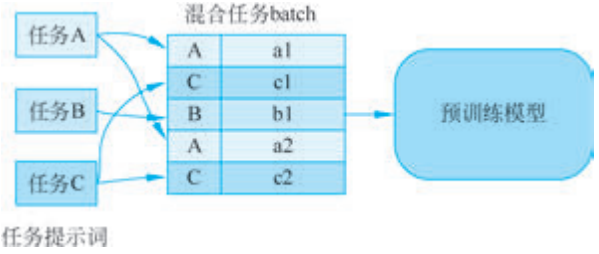

图 3.7　Prompt Tuning 示意图

当然,Prompt Tuning 也存在一定局限性。一是研究者的实验表明,Prompt Tuning 在模型规模超过 100 亿个参数时,提示优化可以与全量微调相媲美。但是对于较小的模型,

Prompt Tuning 和全量微调的表现有很大差异，这大大限制了 Prompt Tuning 的适用性。二是提示词只被插入模型第一层的输入 embedding 序列中，在接下来的 Transformer 层中，提示词的位置的 embedding 是由之前的 Transformer 层计算出来的，因此和 Prefix Tuning 相比，缺少深度提示优化。

这里介绍一个使用 Prompt Tuning 微调 deepseek-r1-distill-qwen-1.5b 模型的示例。和前面的例子一样，使用相同的数据集，实现往模型中注入网络安全专业知识。

首先创建虚拟环境。

```
python3 -m venv deepseek-env
source deepseek-env/bin/activate
```

接着安装必要的 Python 库，包括 PyTorch、Transformers、Datasets。

```
# 安装 PyTorch(确保与 CUDA 版本匹配)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/
whl/cu118
# 安装其他依赖
pip install transformers datasets
```

下面是使用 Prompt Tuning 微调模型的示例。

```
import os
import torch
import json
from datasets import Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
)
from torch.optim import AdamW
import tqdm

# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")

# 设置环境变量
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"

# 模型和数据路径
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"
data_path = "./data/data.json"

# 加载 tokenizer 和模型
print(f"加载模型: {model_name}")
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto",
```

```
)
model.requires_grad_(False)
print(f"模型参数数量：{sum(p.numel() for p in model.parameters())}")

# 数据加载与处理
def load_data(file_path):
    print(f"加载数据：{file_path}")
    try:
        with open(file_path, "r", encoding = "utf - 8") as f:
            data = json.load(f)
    except Exception as e:
        print(f"加载数据时出错：{e}")
        data = [
            {"question": "什么是人工智能?", "answer": "人工智能是指计算机系统能够执行通
常需要人类智能才能完成的任务的能力。"},
            {"question": "什么是深度学习?", "answer": "深度学习是机器学习的一个分支领域,
它使用多层神经网络来学习数据的表示和特征。"}
        ]
    print(f"已加载 {len(data[:1000])} 条数据")
    return data[:1000]
def process_data(examples):
    prompts = []
    for q, a in zip(examples["question"], examples["answer"]):
        prompts.append(f"问：{q}\n 答：{a}{tokenizer.eos_token}")

    tokenized = tokenizer(
        prompts,
        truncation = True,
        padding = "max_length",
        max_length = 512,
        return_tensors = "pt"
    )
    tokenized["labels"] = tokenized["input_ids"].clone()
    return tokenized

# 加载并处理数据
data = load_data(data_path)
dataset = Dataset.from_list(data)
print("处理数据集…")
tokenized_dataset = dataset.map(
    process_data,
    batched = True,
    remove_columns = dataset.column_names
)
tokenized_dataset = tokenized_dataset.with_format("torch", device = device)

# 核心：Prompt Tuning 实现
class PromptTuning(torch.nn.Module):
    def __init__(self, model, prompt_length = 20):
        super().__init__()
        self.model = model
        self.prompt_length = prompt_length
        self.hidden_size = model.config.hidden_size
```

```
        param = next(model.parameters())
        self.device = param.device
        self.dtype = param.dtype
        # 初始化可学习的 prompt 向量(三维:[1, prompt_length, hidden_size])
        self.prompt_embeds = torch.nn.Parameter(
            torch.randn(1, prompt_length, self.hidden_size,
                        device=self.device, dtype=self.dtype)
        )
    def forward(self, input_ids, attention_mask, labels=None):
        """强制确保所有张量维度正确"""
        # 1. 确保 input_ids 是二维 [batch_size, seq_len]
        if input_ids.dim() == 1:
            input_ids = input_ids.unsqueeze(0)            # 单样本时添加 batch 维度
        elif input_ids.dim() == 3:
            input_ids = input_ids.squeeze(1)              # 移除多余维度
        # 验证 input_ids 维度
        assert input_ids.dim() == 2, f"input_ids 维度错误: 预期二维,实际{input_ids.dim()}维"
        # 2. 确保 attention_mask 是二维 [batch_size, seq_len]
        if attention_mask.dim() == 1:
            attention_mask = attention_mask.unsqueeze(0)
        elif attention_mask.dim() == 3:
            attention_mask = attention_mask.squeeze(1)
        assert attention_mask.dim() == 2, f"attention_mask 维度错误: 预期二维,实际
{attention_mask.dim()}维"
        # 3. 确保 labels 是二维[batch_size, seq_len](如果存在)
        if labels is not None:
            if labels.dim() == 1:
                labels = labels.unsqueeze(0)
            elif labels.dim() == 3:
                labels = labels.squeeze(1)
            assert labels.dim() == 2, f"labels 维度错误: 预期二维,实际{labels.dim()}维"
        batch_size = input_ids.shape[0]
        # 4. 获取输入嵌入(必须是三维 [batch_size, seq_len, hidden_size])
        input_embeds = self.model.get_input_embeddings()(input_ids)
        # 强制确保 input_embeds 是三维的
        if input_embeds.dim() == 2:
            input_embeds = input_embeds.unsqueeze(1) # [batch_size, 1, hidden_size] -> 扩
展 seq_len 维度
        assert input_embeds.dim() == 3, f"input_embeds 维度错误: 预期三维,实际{input_
embeds.dim()}维"
        # 5. 扩展 prompt_embeds 至批次大小(三维 [batch_size, prompt_len, hidden_size])
        prompt_embeds = self.prompt_embeds.expand(batch_size, -1, -1)
        assert prompt_embeds.dim() == 3, f"prompt_embeds 维度错误: 预期三维,实际{prompt_
embeds.dim()}维"
        # 6. 拼接 prompt 和输入嵌入(三维拼接)
        input_with_prompt = torch.cat([prompt_embeds, input_embeds], dim=1)
        assert input_with_prompt.dim() == 3, f"拼接后维度错误: 预期三维,实际{input_with_
prompt.dim()}维"
        # 7. 调整注意力掩码(二维)
        prompt_attention_mask = torch.ones(
            batch_size, self.prompt_length,
            device=self.device, dtype=attention_mask.dtype
        )
```

```
        new_attention_mask = torch.cat([prompt_attention_mask, attention_mask], dim=1)
        assert new_attention_mask.dim() == 2, f"新 attention_mask 维度错误：预期二维，实际
{new_attention_mask.dim()}维"
        # 8. 调整标签(二维)
        if labels is not None:
            prompt_labels = torch.full(
                (batch_size, self.prompt_length), -100,
                device=self.device, dtype=labels.dtype
            )
            new_labels = torch.cat([prompt_labels, labels], dim=1)
            assert new_labels.dim() == 2, f"新 labels 维度错误：预期二维，实际{new_labels.
dim()}维"
        else:
            new_labels = None
        # 9. 调用模型 forward
        outputs = self.model(
            inputs_embeds=input_with_prompt,
            attention_mask=new_attention_mask,
            labels=new_labels
        )
        return outputs

# 初始化模型
prompt_tuning = PromptTuning(model=model, prompt_length=20)
prompt_tuning.to(device)

# 训练配置
save_path = "./autodl-tmp/fine_tuned_deepseek_Prompt"
os.makedirs(save_path, exist_ok=True)
num_epochs = 3
log_interval = 10
best_loss = float('inf')

# 优化器(仅优化 prompt 参数)
optimizer = AdamW([prompt_tuning.prompt_embeds], lr=5e-5)

# 训练循环(确保 batch 是正确的张量)
print("开始 Prompt Tuning 训练…")
for epoch in range(num_epochs):
    total_loss = 0.0
    # 转换为迭代器，每次取一个样本(适合小数据集)
    for batch_idx, batch in enumerate(tqdm.tqdm(
        tokenized_dataset,
        desc=f"Epoch {epoch+1}/{num_epochs}"
    )):
        optimizer.zero_grad()
        # 取出批次数据(已确保是张量)
        input_ids = batch["input_ids"]
        attention_mask = batch["attention_mask"]
        labels = batch["labels"]
        # 前向传播
        outputs = prompt_tuning(
            input_ids=input_ids,
```

```
            attention_mask = attention_mask,
            labels = labels
        )
        loss = outputs.loss
        # 反向传播与更新
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        if (batch_idx + 1) % log_interval == 0:
            avg_loss = total_loss / log_interval
            print(f"Batch {batch_idx + 1} | 当前 Loss: {loss.item():.4f} | 平均 Loss: {avg_
loss:.4f}")
            total_loss = 0.0
        if loss.item() < best_loss:
            best_loss = loss.item()
            torch.save(
                prompt_tuning.prompt_embeds,
                os.path.join(save_path, f"best_prompt_loss_{best_loss:.4f}.pt")
            )
    torch.save(
        prompt_tuning.prompt_embeds,
        os.path.join(save_path, f"epoch_{epoch + 1}_prompt.pt")
    )
    print(f"Epoch {epoch + 1} 保存完成 | 最佳 Loss: {best_loss:.4f}")

# 保存最终参数
final_save_path = os.path.join(save_path, "final_prompt_parameters.pt")
torch.save(prompt_tuning.prompt_embeds, final_save_path)
print(f"训练完成!最终 prompt 参数已保存至 {final_save_path}")

# 测试生成函数
def generate_answer(question, max_length = 200):
    prompt_tuning.eval()
    with torch.no_grad():
        input_text = f"问:{question}\n答:"
        inputs = tokenizer(
            input_text,
            return_tensors = "pt",
            truncation = True,
            max_length = 512
        ).to(device)
        outputs = prompt_tuning.model.generate(
            input_ids = inputs["input_ids"],
            attention_mask = inputs["attention_mask"],
            max_length = len(inputs["input_ids"][0]) + max_length,
            temperature = 0.7,
            do_sample = True,
            pad_token_id = tokenizer.eos_token_id
        )
    generated = tokenizer.decode(outputs[0], skip_special_tokens = True)
    return generated.split("答:")[ - 1].strip()
```

```
# 测试生成结果
print("\n测试生成示例:")
test_questions = [
    "什么是人工智能?",
    "机器学习和深度学习的关系是什么?"
]
for q in test_questions:
    print(f"问题:{q}")
    print(f"回答:{generate_answer(q)}\n")
```

下面是调用 Prefix 微调模型的示例。

```
import os
import torch
import json
from datasets import Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
)

# 核心:Prompt Tuning 实现
class PromptTuning(torch.nn.Module):
    def __init__(self, model, prompt_length=20):
        super().__init__()
        self.model = model
        self.prompt_length = prompt_length
        self.hidden_size = model.config.hidden_size
        param = next(model.parameters())
        self.device = param.device
        self.dtype = param.dtype
        # 初始化可学习的 prompt 向量(三维:[1, prompt_length, hidden_size])
        self.prompt_embeds = torch.nn.Parameter(
            torch.randn(1, prompt_length, self.hidden_size,
                        device=self.device, dtype=self.dtype)
        )
    def forward(self, input_ids, attention_mask, labels=None):
        """强制确保所有张量维度正确"""
        # 确保 input_ids 是二维[batch_size, seq_len]
        if input_ids.dim() == 1:
            input_ids = input_ids.unsqueeze(0)          # 单样本时添加 batch 维度
        elif input_ids.dim() == 3:
            input_ids = input_ids.squeeze(1)            # 移除多余维度
        # 验证 input_ids 维度
        assert input_ids.dim() == 2, f"input_ids 维度错误:预期二维,实际{input_ids.dim()}维"
        # 确保 attention_mask 是二维[batch_size, seq_len]
        if attention_mask.dim() == 1:
            attention_mask = attention_mask.unsqueeze(0)
        elif attention_mask.dim() == 3:
            attention_mask = attention_mask.squeeze(1)
        assert attention_mask.dim() == 2, f"attention_mask 维度错误:预期二维,实际
{attention_mask.dim()}维"
        # 确保 labels 是二维[batch_size, seq_len](如果存在)
```

```
        if labels is not None:
            if labels.dim() == 1:
                labels = labels.unsqueeze(0)
            elif labels.dim() == 3:
                labels = labels.squeeze(1)
            assert labels.dim() == 2, f"labels 维度错误：预期二维，实际{labels.dim()}维"
        batch_size = input_ids.shape[0]
        # 获取输入嵌入(必须是三维[batch_size, seq_len, hidden_size])
        input_embeds = self.model.get_input_embeddings()(input_ids)
        # 强制确保 input_embeds 是三维的
        if input_embeds.dim() == 2:
            input_embeds = input_embeds.unsqueeze(1)      # [batch_size, 1, hidden_size]
-> 扩展 seq_len 维度
        assert input_embeds.dim() == 3, f"input_embeds 维度错误：预期三维，实际{input_
embeds.dim()}维"
        # 扩展 prompt_embeds 至批次大小(三维[batch_size, prompt_len, hidden_size])
        prompt_embeds = self.prompt_embeds.expand(batch_size, -1, -1)
        assert prompt_embeds.dim() == 3, f"prompt_embeds 维度错误：预期三维，实际{prompt_
embeds.dim()}维"
        # 拼接 prompt 和输入嵌入(三维拼接)
        input_with_prompt = torch.cat([prompt_embeds, input_embeds], dim=1)
        assert input_with_prompt.dim() == 3, f"拼接后维度错误：预期三维，实际{input_with_
prompt.dim()}维"
        # 调整注意力掩码(二维)
        prompt_attention_mask = torch.ones(
            batch_size, self.prompt_length,
            device=self.device, dtype=attention_mask.dtype
        )
        new_attention_mask = torch.cat([prompt_attention_mask, attention_mask], dim=1)
        assert new_attention_mask.dim() == 2, f"新 attention_mask 维度错误：预期二维，实际
{new_attention_mask.dim()}维"
        # 调整标签(二维)
        if labels is not None:
            prompt_labels = torch.full(
                (batch_size, self.prompt_length), -100,
                device=self.device, dtype=labels.dtype
            )
            new_labels = torch.cat([prompt_labels, labels], dim=1)
            assert new_labels.dim() == 2, f"新 labels 维度错误：预期二维，实际{new_labels.
dim()}维"
        else:
            new_labels = None
        # 调用模型 forward
        outputs = self.model(
            inputs_embeds=input_with_prompt,
            attention_mask=new_attention_mask,
            labels=new_labels
        )
        return outputs
def load_finetuned_prompt(prompt_path):
    """加载微调后的 prompt 参数"""
    if not os.path.exists(prompt_path):
        raise FileNotFoundError(f"未找到微调参数文件：{prompt_path}")
```

```
    # 加载保存的 prompt 参数(是 torch.nn.Parameter 对象)
    prompt_embeds = torch.load(prompt_path, weights_only=True)
    print(f"已加载微调参数: {prompt_path},形状: {prompt_embeds.shape}")
    return prompt_embeds

def init_inference_model(base_model_path, prompt_path):
    """初始化推理模型(加载原始模型 + 微调的 prompt 参数)"""
    # 加载原始模型
    print(f"加载基础模型: {base_model_path}")
    model = AutoModelForCausalLM.from_pretrained(
        base_model_path,
        torch_dtype=torch.float32,
        device_map="auto",
    )
    model.requires_grad_(False)              # 推理时不训练
    model.eval()                             # 切换到评估模式
    # 加载微调的 prompt 参数
    prompt_embeds = load_finetuned_prompt(prompt_path)
    # 获取 prompt 长度(保存时的形状为[prompt_len, hidden_size])
    prompt_length = prompt_embeds.shape[0]
    # 初始化 PromptTuning 类并注入微调参数
    prompt_tuning = PromptTuning(
        model=model,
        prompt_length=prompt_length
    )
    # 替换 prompt_embeds 为微调后的参数
    prompt_tuning.prompt_embeds = torch.nn.Parameter(prompt_embeds)
    prompt_tuning.to(device)
    return prompt_tuning, model, tokenizer
def generate_answer(question, prompt_tuning, tokenizer, device, max_length=200):
    """使用加载了微调参数的模型生成回答"""
    prompt_tuning.eval()                     # 确保评估模式
    with torch.no_grad():                    # 关闭梯度计算,节省内存
        # 构建输入模板(与训练时一致)
        input_text = f"问:{question}\n答:"
        # 编码输入
        inputs = tokenizer(
            input_text,
            return_tensors="pt",             # 返回 PyTorch 张量
            truncation=True,
            max_length=512,
            padding="max_length"
        ).to(device)
        # 生成回答(使用微调后的 prompt 参数)
        outputs = prompt_tuning.model.generate(
            input_ids=inputs["input_ids"],
            attention_mask=inputs["attention_mask"],
            max_length=len(inputs["input_ids"][0]) + max_length,
            temperature=0.7,                 # 控制随机性(0 为确定性,1 为高随机)
            do_sample=True,                  # 启用采样生成
            top_p=0.9,                       # nucleus sampling 参数,控制候选词范围
            pad_token_id=tokenizer.eos_token_id,
            eos_token_id=tokenizer.eos_token_id,
```

```
            repetition_penalty = 1.2            # 惩罚重复生成的内容
        )
    # 解码并提取回答
    generated = tokenizer.decode(outputs[0], skip_special_tokens = True)
    # 从生成结果中提取"答:"之后的内容
    if "答:" in generated:
        answer = generated.split("答:")[ - 1].strip()
    else:
        answer = generated.strip()              # 兼容格式异常的情况
    return answer
base_model_path = "./model_path/DeepSeek - R1 - distill - Qwen - 1.5B_ori"     # 原始模型路径
prompt_path = "./autodl - tmp/fine_tuned_deepseek_Prompt/final_prompt_parameters.pt"
# 微调参数路径
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 加载 tokenizer(与训练时一致)
tokenizer = AutoTokenizer.from_pretrained(base_model_path)
tokenizer.pad_token = tokenizer.eos_token

try:
    # 初始化推理模型
    prompt_tuning, model, tokenizer = init_inference_model(
        base_model_path = base_model_path,
        prompt_path = prompt_path
    )
    # 测试推理
    print("\n===== 微调后模型推理结果 ===== ")
    test_questions = [
        "什么是防火墙?",
        "什么是人工智能?"
    ]
    for q in test_questions:
        print(f"问题:{q}")
        answer = generate_answer(
            question = q,
            prompt_tuning = prompt_tuning,
            tokenizer = tokenizer,
            device = device
        )
        print(f"回答:{answer}\n")
except Exception as e:
    print(f"推理过程出错:{e}")
```

3.6.5　P-tuning

P-tuning 是比 Prefix Tuning 稍晚一些的工作,它使用一个可训练的提示词编码器(如 LSTM)来映射提示词嵌入,将提示词嵌入转换为可以学习的嵌入层,允许根据输入数据的不同生成不同的嵌入,从而可以捕捉提示词嵌入之间的联系。P-tuning 的核心组件包括伪提示词嵌入和提示词编码器。伪提示词是一组可训练的参数,它们的嵌入表示是连续的。提示词编码器则是处理这些伪提示词嵌入的神经网络,它负责将伪提示词编码到高级表示,

然后再输入主模型中，如图 3.8 所示。

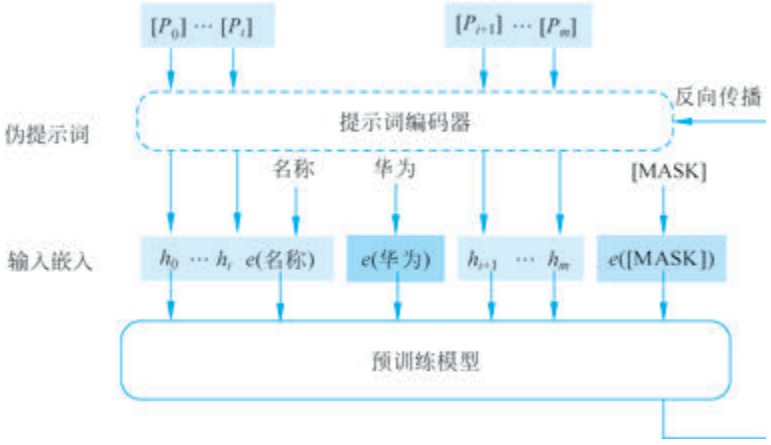

图 3.8　P-tuning 示意图

相比 Prefix Tuning，P-tuning 在输入层加入可微的伪提示词，另外，伪提示词的位置也不一定是前缀，可以插入输入的任何位置。因此，实际上是把传统人工设计模板中的真实提示词替换成可微的伪提示词。经过预训练的大语言模型的词嵌入已经变得高度离散化，若随机初始化伪提示词，容易优化到局部最优值。为此，研究者用一个提示词编码器编码这些伪提示词后，再输入模型，如图 3.9 所示。这种设计使得 P-tuning 相对于 Prefix Tuning 在处理需要精细控制和理解复杂上下文的任务时有更好的表现。

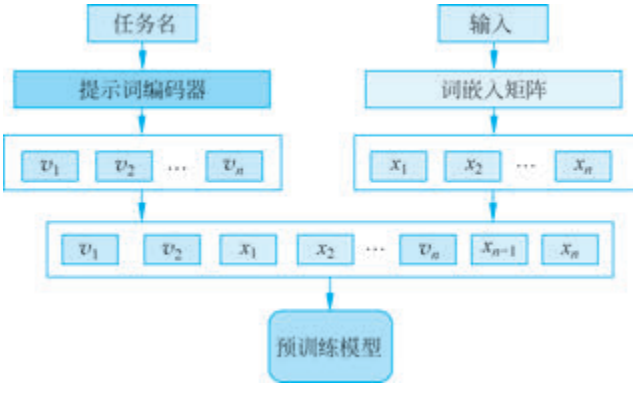

图 3.9　训练中 P-tuning 的前向传播

P-tuning 的关键技术细节如下。

1. 伪提示词嵌入

在 P-tuning 中，伪提示词嵌入是一组可训练的参数，它们的形状是[num_virtual_tokens, embedding_dim]，其中 num_virtual_tokens 是伪提示词的数量，embedding_dim 是嵌入维度。$[P_i]$ 是第 i 个伪提示词嵌入，则提示词模板为 $\{[P_{0:i}], x, [P_{i+1:j}], y, [P_{j+1:k}]\}$，通过一个嵌入函数 $f:[P_i]\rightarrow h_i$，将提示词模板映射为 $\{h_0,\cdots,h_i,e(x),h_{i+1},\cdots,h_j,e(y),h_{j+1},\cdots,h_k\}$。使用 f 有助于学习到不同提示词嵌入之间的联系。模型训练更新参数 $\{P_i\}_{i=0}^{k}$。也可以拼接伪提示词和离散提示词后，输入模型进行训练。

这些伪提示词嵌入是随机初始化的，然后通过优化过程学习。与直接使用预训练模型的词嵌入不同，这些伪提示词嵌入专用于特定任务，可以在训练过程中不断调整以找到最佳表示。

2. 提示词编码器

提示词编码器是一个处理伪提示词嵌入的神经网络，即上述的嵌入函数 f，通常是一个简单的 RNN 或 MLP。它的作用是将伪提示词嵌入编码为更高级别的表示，然后输入主模型中。提示词编码器的设计可以帮助捕捉伪提示词之间的语义关系，提供一个稳定的优化目标，减少随机初始化伪提示词嵌入带来的优化困难。

当然，P-tuning 也存在和 Prompt Tuning 相似的局限性。

这里介绍一个使用 P-tuning 微调 deepseek-r1-distill-qwen-1.5b 模型的示例。和前面的例子一样，使用相同的数据集，实现往模型中注入网络安全专业知识。

首先创建虚拟环境。

```
python3 -m venv deepseek-env
source deepseek-env/bin/activate
```

接着安装必要的 Python 库，包括 PyTorch、Transformers、Datasets。

```
# 安装 PyTorch(确保与 CUDA 版本匹配)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/
whl/cu118
# 安装其他依赖
pip install transformers datasets
```

下面是使用 P-tuning 微调模型的示例。

```
import os
import torch
import json
from datasets import load_dataset, Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling,
    get_linear_schedule_with_warmup
)
# 设置设备
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"使用设备: {device}")

# 设置环境变量
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"          # 便于调试 CUDA 错误

# 模型和数据路径
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"
data_path = "./data/data.json"                    # 示例数据集路径
```

```
# 加载 tokenizer 和模型
print(f"加载模型: {model_name}")
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token          # 设置填充标记

# 以 FP32 精度加载模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto",                             # 自动处理多 GPU 分布
)
model.requires_grad_(False)                        # 冻结模型参数

# 加载数据
def load_data(file_path):
    print(f"加载数据: {file_path}")
    try:
        with open(file_path, "r", encoding="utf-8") as f:
            data = json.load(f)
    except Exception as e:
        print(f"加载数据时出错: {e}")
        # 创建示例数据用于测试
        print("创建示例数据用于测试...")
        data = [
            {"question": "什么是人工智能?", "answer": "人工智能是指计算机系统能够执行通常需要人类智能才能完成的任务的能力。"},
            {"question": "什么是深度学习?", "answer": "深度学习是机器学习的一个分支领域,它使用多层神经网络来学习数据的表示和特征。"}
        ]
    print(f"已加载 {len(data[:1000])} 条数据")
    #return data
    return data[:1000]

# 处理数据
def process_data(examples):
    # 构建指令模板
    prompts = []
    for q, a in zip(examples["question"], examples["answer"]):
        prompt = f"问:{q}\n答:{a}{tokenizer.eos_token}"
        prompts.append(prompt)
    # 编码文本
    tokenized = tokenizer(
        prompts,
        truncation=True,
        padding="max_length",
        max_length=512,
        return_tensors="pt"
    )
    # 对于自监督学习,标签与输入相同
    tokenized["labels"] = tokenized["input_ids"].clone()
    return tokenized

# 加载并处理数据集
```

```
data = load_data(data_path)
dataset = Dataset.from_list(data)

print("处理数据集…")
tokenized_dataset = dataset.map(
    process_data,
    batched = True,
    remove_columns = dataset.column_names
)

import wandb
wandb.init(project = "config_example", mode = "disabled")

# 核心:P-tuning 实现
import transformers
class PromptEncoder(torch.nn.Module):
    """将离散的 Prompt Token 转换为连续嵌入的编码器"""
    def __init__(self, hidden_size, prompt_length, device, dtype):
        super().__init__()
        self.hidden_size = hidden_size
        self.prompt_length = prompt_length
        self.device = device
        self.dtype = dtype
        # LSTM 编码器参数 - 指定为 float32
        self.lstm_head = torch.nn.LSTM(
            input_size = hidden_size,
            hidden_size = hidden_size//2,
            num_layers = 2,
            bidirectional = True,
            batch_first = True,
            dtype = torch.float32        # 显式指定为 float32
        ).to(device)
        # MLP 映射层 - 指定为 float32
        self.mlp_head = torch.nn.Sequential(
            torch.nn.Linear(hidden_size, hidden_size, dtype = torch.float32),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_size, hidden_size, dtype = torch.float32)
        ).to(device)
        # 初始化参数
        for param in self.parameters():
            if param.dim() > 1:
                torch.nn.init.xavier_uniform_(param)
    def forward(self):
        # 创建位置索引
        indices = torch.arange(self.prompt_length, device = self.device).unsqueeze(0)
        # LSTM 编码 - 确保输入为 float32
        lstm_output, _ = self.lstm_head(
            torch.zeros(1, self.prompt_length, self.hidden_size, device = self.device,
dtype = torch.float32)
        )
        # MLP 映射
        output = self.mlp_head(lstm_output)
        # 转换回模型的原始数据类型
```

```
        return output.squeeze(0).to(self.dtype) # [prompt_length, hidden_size]
class Template(torch.nn.Module):
    """管理 prompt 模板,包括离散和连续部分"""
    def __init__(self, model, tokenizer, prompt_length=20, n_token=5):
        super().__init__()
        self.model = self._get_base_model(model)
        self.tokenizer = tokenizer
        self.prompt_length = prompt_length
        self.n_token = n_token if n_token is not None else 5
        self.continuous_length = prompt_length - self.n_token
        # 获取模型信息
        param = next(self.model.parameters())
        self.device = param.device
        self.dtype = param.dtype
        self.hidden_size = self.model.config.hidden_size

        # 检查 tokenizer 是否有 unk_token
        if tokenizer.unk_token_id is None:
            print("Tokenizer 没有定义 unk_token,使用 pad_token 代替")
            if tokenizer.pad_token_id is not None:
                unk_id = tokenizer.pad_token_id
            else:
                # 如果 pad_token 也没有,使用 eos_token 或任意一个特殊 Token
                unk_id = tokenizer.eos_token_id if tokenizer.eos_token_id is not None else 0
        else:
            unk_id = tokenizer.unk_token_id
        # 初始化离散 Token
        self.discrete_token_ids = torch.tensor(
            [unk_id] * self.n_token,
            device=self.device, dtype=torch.long
        )
        # 连续 Prompt 编码器
        self.prompt_encoder = PromptEncoder(
            self.hidden_size,
            self.continuous_length,
            self.device,
            self.dtype
        )
    def _get_base_model(self, model):
        """递归获取最内层的原始模型"""
        while hasattr(model, 'model') and isinstance(model, torch.nn.Module):
            model = model.model
        return model
    def forward(self):
        # 获取离散 Token 嵌入
        discrete_embeds = self.model.get_input_embeddings()(self.discrete_token_ids)
        # 获取连续 Prompt 嵌入
        continuous_embeds = self.prompt_encoder()
        # 组合离散和连续部分
        prompt_embeds = torch.cat([discrete_embeds, continuous_embeds], dim=0)
        return prompt_embeds

class PromptTuning(torch.nn.Module):
```

```
    """P-tuning 实现,基于论文【GPT Understands, Too】"""
    def __init__(self, model, tokenizer, prompt_length=20, n_token=5):
        super().__init__()
        self.model = model
        self.tokenizer = tokenizer
        # 创建 template - 使用递归方法获取原始模型
        self.template = Template(model, tokenizer, prompt_length, n_token)
        # 冻结原模型参数
        for param in model.parameters():
            param.requires_grad = False
    def forward(self, input_ids, attention_mask, labels=None):
        # 获取模型设备
        device = next(self.model.parameters()).device
        # 确保输入是正确类型的张量
        if not isinstance(input_ids, torch.Tensor):
            input_ids = self._convert_to_tensor(input_ids, device)
        if not isinstance(attention_mask, torch.Tensor):
            attention_mask = self._convert_to_tensor(attention_mask, device)
        if labels is not None and not isinstance(labels, torch.Tensor):
            labels = self._convert_to_tensor(labels, device)
        # 确保输入 ID 是 Long 类型
        if input_ids.dtype != torch.long:
            print(f"警告:将 input_ids 从{input_ids.dtype}转换为 torch.long")
            input_ids = input_ids.long()
        if attention_mask.dtype != torch.long:
            print(f"警告:将 attention_mask 从{attention_mask.dtype}转换为 torch.long")
            attention_mask = attention_mask.long()
        if labels is not None and labels.dtype != torch.long:
            print(f"警告:将 labels 从{labels.dtype}转换为 torch.long")
            labels = labels.long()
        # 获取原始输入嵌入
        base_model = self._get_base_model(self.model)
        input_embeds = base_model.get_input_embeddings()(input_ids)
        # 获取 Prompt 嵌入
        prompt_embeds = self.template()
        # 扩展 Prompt 嵌入以匹配批次大小
        batch_size = input_embeds.shape[0]
        prompt_embeds_expanded = prompt_embeds.unsqueeze(0).expand(batch_size, -1, -1)
        # 拼接 Prompt 和输入嵌入
        input_with_prompt = torch.cat([prompt_embeds_expanded, input_embeds], dim=1)
        # 调整注意力掩码
        prompt_attention_mask = torch.ones(
            batch_size, prompt_embeds.shape[0],
            device=input_embeds.device, dtype=attention_mask.dtype
        )
        new_attention_mask = torch.cat([prompt_attention_mask, attention_mask], dim=1)
        # 调整标签(如果提供)
        if labels is not None:
            # Prompt 部分的标签设为-100,忽略计算损失
            prompt_labels = torch.full(
                (batch_size, prompt_embeds.shape[0]), -100,
                device=input_embeds.device, dtype=labels.dtype
            )
```

```
            new_labels = torch.cat([prompt_labels, labels], dim=1)
        else:
            new_labels = None
        # 调用模型
        base_model = self._get_base_model(self.model)
        # 动态检查 Qwen 模型类名
        try:
            qwen_class = transformers.QWenLMHeadModel
        except AttributeError:
            try:
                qwen_class = transformers.QWenForCausalLM
            except AttributeError:
                qwen_class = None
        # 针对 Qwen2Model 的特殊处理
        if isinstance(base_model, transformers.models.qwen2.modeling_qwen2.Qwen2Model):
            print("检测到 Qwen2Model,使用专用逻辑处理 logits…")
            try:
                # 调用模型获取基础输出
                outputs = base_model(
                    inputs_embeds=input_with_prompt,
                    attention_mask=new_attention_mask,
                    return_dict=True
                )
                # 获取输出嵌入层权重
                output_weight = base_model.embed_tokens.weight
                # 获取最后一个隐藏状态
                final_hidden_state = outputs.last_hidden_state
                # 计算 logits(使用矩阵乘法替代直接调用嵌入层)
                logits = torch.matmul(final_hidden_state, output_weight.transpose(0, 1))
                # 手动计算损失
                if new_labels is not None:
                    loss_fct = torch.nn.CrossEntropyLoss()
                    loss = loss_fct(
                        logits.view(-1, logits.size(-1)),
                        new_labels.view(-1)
                    )
                else:
                    loss = None
            except Exception as e:
                # 打印详细的输入类型调试信息
                print(f"输入类型调试信息:")
                print(f" input_ids dtype: {input_ids.dtype}")
                print(f" attention_mask dtype: {attention_mask.dtype}")
                if labels is not None:
                    print(f" labels dtype: {labels.dtype}")
                # 仅在 input_with_prompt 存在时打印其类型
                if 'input_with_prompt' in locals() and input_with_prompt is not None:
                    print(f" input_with_prompt dtype: {input_with_prompt.dtype}")
                # 打印 Qwen2Model 的特殊调试信息
                print(f"Qwen2Model 调试信息:")
                print(f" embed_tokens 类型: {type(base_model.embed_tokens).__name__}")
                if hasattr(base_model, 'get_output_embeddings'):
                    output_embeddings = base_model.get_output_embeddings()
```

```
                    print(f" get_output_embeddings 返回类型: {type(output_embeddings).__
name__}")
                raise AttributeError(f"Qwen2Model 处理失败: {str(e)}")
        # 检查模型类型,确定输出结构
        elif isinstance(base_model, transformers.GPT2LMHeadModel) or \
             isinstance(base_model, transformers.LlamaForCausalLM) or \
             (qwen_class is not None and isinstance(base_model, qwen_class)):
            # 生成模型输出
            outputs = base_model(
                inputs_embeds = input_with_prompt,
                attention_mask = new_attention_mask,
                labels = new_labels,
                return_dict = True
            )
            loss = outputs.loss
            logits = outputs.logits
        else:
            # 基础模型输出(不含 logits)
            outputs = base_model(
                inputs_embeds = input_with_prompt,
                attention_mask = new_attention_mask,
                return_dict = True
            )
            # 尝试多种方式获取 logits
            logits = None
            try:
                # 尝试常见的 lm_head
                if hasattr(base_model, 'lm_head'):
                    lm_head = base_model.lm_head
                    if callable(lm_head):
                        logits = lm_head(outputs.last_hidden_state)
                    else:
                        raise AttributeError("lm_head 不可调用")
                elif hasattr(base_model, 'decoder') and hasattr(base_model.decoder, 'lm_head'):
                    lm_head = base_model.decoder.lm_head
                    if callable(lm_head):
                        logits = lm_head(outputs.last_hidden_state)
                    else:
                        raise AttributeError("decoder.lm_head 不可调用")
                elif hasattr(base_model, 'get_output_embeddings'):
                    # 某些模型需要通过此方法获取输出层
                    output_embeddings = base_model.get_output_embeddings()
                    if output_embeddings is not None and callable(output_embeddings):
                        logits = output_embeddings(outputs.last_hidden_state)
                    else:
                        raise AttributeError("get_output_embeddings()返回 None 或不可调用")
                else:
                    # 尝试获取最终层的输出
                    final_hidden_state = outputs.last_hidden_state
                    # 假设最后一个维度是隐藏状态大小
                    vocab_size = base_model.config.vocab_size
                    # 创建一个线性层作为最后的分类器
                    linear_layer = torch.nn.Linear(
```

```
                        final_hidden_state.shape[-1],
                        vocab_size,
                        bias=False,
                        dtype=final_hidden_state.dtype
                    ).to(final_hidden_state.device)
                    # 随机初始化权重
                    torch.nn.init.xavier_uniform_(linear_layer.weight)
                    logits = linear_layer(final_hidden_state)
                    print("警告:使用了默认的线性层来生成 logits,这可能不是最优的。")
                    print(f"模型类型:{type(base_model).__name__}")
                except Exception as e:
                    raise AttributeError(f"无法从基础模型中获取 logits: {str(e)}")
                # 手动计算损失
                if new_labels is not None and logits is not None:
                    loss_fct = torch.nn.CrossEntropyLoss()
                    loss = loss_fct(
                        logits.view(-1, logits.size(-1)),
                        new_labels.view(-1)
                    )
                else:
                    loss = None
            # 返回损失和其他输出
            return {
                'loss': loss,
                'logits': logits,
                'past_key_values': outputs.get('past_key_values', None)
            }
        def _convert_to_tensor(self, data, device):
            """递归转换嵌套列表为张量"""
            if isinstance(data, list):
                # 处理混合类型列表
                if len(data) == 0:
                    return torch.tensor([], dtype=torch.long, device=device).unsqueeze(0)
                converted = []
                for item in data:
                    converted.append(self._convert_to_tensor(item, device))
                # 检查所有元素是否为相同维度的张量
                if all(t.dim() == converted[0].dim() for t in converted):
                    try:
                        # 尝试堆叠张量
                        return torch.stack(converted, dim=0)
                    except RuntimeError:
                        # 堆叠失败,可能是形状不一致
                        pass
                # 如果无法堆叠,转换为列表形式的张量
                return torch.tensor([t.tolist() for t in converted], dtype=torch.long, device=device)
            elif isinstance(data, torch.Tensor):
                return data.to(device)
            elif isinstance(data, int):
                # 直接处理整数
                return torch.tensor([data], dtype=torch.long, device=device)
            else:
                # 尝试将其他类型转换为整数
```

```
            try:
                return torch.tensor([int(data)], dtype=torch.long, device=device)
            except (ValueError, TypeError):
                raise TypeError(f"无法将类型 {type(data).__name__} 转换为整数张量")
    def _get_base_model(self, model):
        """递归获取最内层的原始模型"""
        while hasattr(model, 'model') and isinstance(model, torch.nn.Module):
            model = model.model
        return model

from torch.optim import AdamW # 从 PyTorch 导入 AdamW
import tqdm # 用于显示进度条
# 训练函数
def train(model, tokenizer, train_dataset, args):
    """训练模型"""
    # 创建 dataloader
    train_dataloader = torch.utils.data.DataLoader(
        train_dataset,
        batch_size=args.batch_size,
        shuffle=True
    )
    # 优化器 - 只优化 prompt 参数
    optimizer = AdamW(
        [p for n, p in model.named_parameters() if p.requires_grad],
        lr=args.learning_rate
    )
    # 学习率调度器
    total_steps = len(train_dataloader) * args.num_epochs
    scheduler = get_linear_schedule_with_warmup(
        optimizer,
        num_warmup_steps=args.warmup_steps,
        num_training_steps=total_steps
    )
    # 创建保存路径
    os.makedirs(args.save_path, exist_ok=True)
    print("开始微调模型…")
    best_loss = float('inf')
    for epoch in range(args.num_epochs):
        model.train()
        total_loss = 0.0
        progress_bar = tqdm.tqdm(
            enumerate(train_dataloader),
            total=len(train_dataloader),
            desc=f"Epoch {epoch+1}/{args.num_epochs}"
        )
        for step, batch in progress_bar:
            # 将数据移至设备
            if isinstance(batch, list):
                batch = [item.to(args.device) if isinstance(item, torch.Tensor) else item
for item in batch]
            else:
                batch = {k: v.to(args.device) if isinstance(v, torch.Tensor) else v for k, v
in batch.items()}
```

```
            # 前向传播
            outputs = model(
                input_ids = batch["input_ids"],
                attention_mask = batch["attention_mask"],
                labels = batch["labels"]
            )
            loss = outputs['loss']        # 修改:从字典中获取损失
            # 反向传播
            loss.backward()
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()
            total_loss += loss.item()
            avg_loss = total_loss / (step + 1)
            progress_bar.set_postfix({"loss": avg_loss})
            # 保存最佳模型
            if loss.item() < best_loss:
                best_loss = loss.item()
                # 保存 prompt 参数
                torch.save(
                    model.template.prompt_encoder.state_dict(),
                    os.path.join(args.save_path, f"best_prompt_loss_{best_loss:.4f}.pt")
                )
                print(f"保存最佳 prompt: loss = {best_loss:.4f}")
        print(f"Epoch {epoch + 1}/{args.num_epochs} 完成 | 平均损失: {total_loss / len(train
_dataloader):.4f}")
    # 保存最终的 prompt 参数
    torch.save(
        model.template.prompt_encoder.state_dict(),
        os.path.join(args.save_path, "final_prompt_encoder.pt")
    )
    print(f"训练完成!最佳损失: {best_loss:.4f}")

class Args:
    def __init__(self):
        self.save_path = "./autodl-tmp/fine_tuned_deepseek"
        self.prompt_length = 10
        self.num_epochs = 1
        self.batch_size = 4
        self.learning_rate = 5e-5
        self.warmup_steps = 500
        self.max_length = 512

# 创建参数实例
args = Args()

# 应用 P-tuning
model = PromptTuning(model, tokenizer, args.prompt_length)
model.to(device)
print(f"应用 P-tuning 后,可训练参数数量: {sum(p.numel() for p in model.parameters() if p.
requires_grad)}")

# 训练模型
train(model, tokenizer, tokenized_dataset, args)
```

下面是调用 P-tuning 微调模型的示例。

```
import os
import torch
import json
from datasets import load_dataset, Dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling,
    get_linear_schedule_with_warmup
)
# 核心:P-tuning 实现
import transformers
class PromptEncoder(torch.nn.Module):
    """将离散的 Prompt Token 转换为连续嵌入的编码器"""
    def __init__(self, hidden_size, prompt_length, device, dtype):
        super().__init__()
        self.hidden_size = hidden_size
        self.prompt_length = prompt_length
        self.device = device
        self.dtype = dtype
        # LSTM 编码器参数 - 指定为 float32
        self.lstm_head = torch.nn.LSTM(
            input_size=hidden_size,
            hidden_size=hidden_size//2,
            num_layers=2,
            bidirectional=True,
            batch_first=True,
            dtype=torch.float32              # 显式指定为 float32
        ).to(device)
        # MLP 映射层 - 指定为 float32
        self.mlp_head = torch.nn.Sequential(
            torch.nn.Linear(hidden_size, hidden_size, dtype=torch.float32),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_size, hidden_size, dtype=torch.float32)
        ).to(device)
        # 初始化参数
        for param in self.parameters():
            if param.dim() > 1:
                torch.nn.init.xavier_uniform_(param)
    def forward(self):
        # 创建位置索引
        indices = torch.arange(self.prompt_length, device=self.device).unsqueeze(0)
        # LSTM 编码 - 确保输入为 float32
        lstm_output, _ = self.lstm_head(
             torch.zeros(1, self.prompt_length, self.hidden_size, device=self.device,
dtype=torch.float32)
        )
        # MLP 映射
        output = self.mlp_head(lstm_output)
```

```
            # 转换回模型的原始数据类型
            return output.squeeze(0).to(self.dtype)      # [prompt_length, hidden_size]

    class Template(torch.nn.Module):
        """管理 prompt 模板,包括离散和连续部分"""
        def __init__(self, model, tokenizer, prompt_length=20, n_token=5):
            super().__init__()
            self.model = self._get_base_model(model)
            self.tokenizer = tokenizer
            self.prompt_length = prompt_length
            self.n_token = n_token if n_token is not None else 5
            self.continuous_length = prompt_length - self.n_token
            # 获取模型信息
            param = next(self.model.parameters())
            self.device = param.device
            self.dtype = param.dtype
            self.hidden_size = self.model.config.hidden_size
            # 检查 tokenizer 是否有 unk_token
            if tokenizer.unk_token_id is None:
                print("Tokenizer 没有定义 unk_token,使用 pad_token 代替")
                if tokenizer.pad_token_id is not None:
                    unk_id = tokenizer.pad_token_id
                else:
                    # 如果 pad_token 也没有,使用 eos_token 或任意一个特殊 Token
                    unk_id = tokenizer.eos_token_id if tokenizer.eos_token_id is not None else 0
            else:
                unk_id = tokenizer.unk_token_id

            # 初始化离散 Token
            self.discrete_token_ids = torch.tensor(
                [unk_id] * self.n_token,
                device=self.device, dtype=torch.long
            )
            # 连续 Prompt 编码器
            self.prompt_encoder = PromptEncoder(
                self.hidden_size,
                self.continuous_length,
                self.device,
                self.dtype
            )
        def _get_base_model(self, model):
            """递归获取最内层的原始模型"""
            while hasattr(model, 'model') and isinstance(model, torch.nn.Module):
                model = model.model
            return model
        def forward(self):
            # 获取离散 Token 嵌入
            discrete_embeds = self.model.get_input_embeddings()(self.discrete_token_ids)
            # 获取连续 Prompt 嵌入
            continuous_embeds = self.prompt_encoder()
            # 组合离散和连续部分
            prompt_embeds = torch.cat([discrete_embeds, continuous_embeds], dim=0)
```

```
        return prompt_embeds

class PromptTuning(torch.nn.Module):
    """P-tuning 实现,基于论文【GPT Understands, Too】"""
    def __init__(self, model, tokenizer, prompt_length=20, n_token=5):
        super().__init__()
        self.model = model
        self.tokenizer = tokenizer
        # 创建 template - 使用递归方法获取原始模型
        self.template = Template(model, tokenizer, prompt_length, n_token)
        # 冻结原模型参数
        for param in model.parameters():
            param.requires_grad = False
    def forward(self, input_ids, attention_mask, labels=None):
        # 获取模型设备
        device = next(self.model.parameters()).device
        # 确保输入是正确类型的张量
        if not isinstance(input_ids, torch.Tensor):
            input_ids = self._convert_to_tensor(input_ids, device)
        if not isinstance(attention_mask, torch.Tensor):
            attention_mask = self._convert_to_tensor(attention_mask, device)
        if labels is not None and not isinstance(labels, torch.Tensor):
            labels = self._convert_to_tensor(labels, device)
        # 确保输入 ID 是 Long 类型
        if input_ids.dtype != torch.long:
            print(f"警告:将 input_ids 从{input_ids.dtype}转换为 torch.long")
            input_ids = input_ids.long()
        if attention_mask.dtype != torch.long:
            print(f"警告:将 attention_mask 从{attention_mask.dtype}转换为 torch.long")
            attention_mask = attention_mask.long()
        if labels is not None and labels.dtype != torch.long:
            print(f"警告:将 labels 从{labels.dtype}转换为 torch.long")
            labels = labels.long()
        # 获取原始输入嵌入
        base_model = self._get_base_model(self.model)
        input_embeds = base_model.get_input_embeddings()(input_ids)
        # 获取 Prompt 嵌入
        prompt_embeds = self.template()
        # 扩展 Prompt 嵌入以匹配批次大小
        batch_size = input_embeds.shape[0]
        prompt_embeds_expanded = prompt_embeds.unsqueeze(0).expand(batch_size, -1, -1)
        # 拼接 Prompt 和输入嵌入
        input_with_prompt = torch.cat([prompt_embeds_expanded, input_embeds], dim=1)
        # 调整注意力掩码
        prompt_attention_mask = torch.ones(
            batch_size, prompt_embeds.shape[0],
            device=input_embeds.device, dtype=attention_mask.dtype
        )
        new_attention_mask = torch.cat([prompt_attention_mask, attention_mask], dim=1)

        # 调整标签(如果提供)
        if labels is not None:
            # Prompt 部分的标签设为 -100,忽略计算损失
```

```
            prompt_labels = torch.full(
                (batch_size, prompt_embeds.shape[0]), -100,
                device=input_embeds.device, dtype=labels.dtype
            )
            new_labels = torch.cat([prompt_labels, labels], dim=1)
        else:
            new_labels = None
        # 调用模型
        base_model = self._get_base_model(self.model)
        # 动态检查 Qwen 模型类名
        try:
            qwen_class = transformers.QWenLMHeadModel
        except AttributeError:
            try:
                qwen_class = transformers.QWenForCausalLM
            except AttributeError:
                qwen_class = None
        # 针对 Qwen2Model 的特殊处理
        if isinstance(base_model, transformers.models.qwen2.modeling_qwen2.Qwen2Model):
            print("检测到 Qwen2Model,使用专用逻辑处理 logits…")
            try:
                # 调用模型获取基础输出
                outputs = base_model(
                    inputs_embeds=input_with_prompt,
                    attention_mask=new_attention_mask,
                    return_dict=True
                )
                # 获取输出嵌入层权重
                output_weight = base_model.embed_tokens.weight
                # 获取最后一个隐藏状态
                final_hidden_state = outputs.last_hidden_state
                # 计算 logits(使用矩阵乘法替代直接调用嵌入层)
                logits = torch.matmul(final_hidden_state, output_weight.transpose(0, 1))
                # 手动计算损失
                if new_labels is not None:
                    loss_fct = torch.nn.CrossEntropyLoss()
                    loss = loss_fct(
                        logits.view(-1, logits.size(-1)),
                        new_labels.view(-1)
                    )
                else:
                    loss = None
            except Exception as e:
                # 打印详细的输入类型信息
                print(f"输入类型调试信息:")
                print(f" input_ids dtype: {input_ids.dtype}")
                print(f" attention_mask dtype: {attention_mask.dtype}")
                if labels is not None:
                    print(f" labels dtype: {labels.dtype}")
                # 仅在 input_with_prompt 存在时打印其类型
                if 'input_with_prompt' in locals() and input_with_prompt is not None:
                    print(f" input_with_prompt dtype: {input_with_prompt.dtype}")
                # 打印 Qwen2Model 的特殊调试信息
```

```
                print(f"Qwen2Model 调试信息:")
                print(f" embed_tokens 类型: {type(base_model.embed_tokens).__name__}")
                if hasattr(base_model, 'get_output_embeddings'):
                    output_embeddings = base_model.get_output_embeddings()
                    print(f" get_output_embeddings 返回类型: {type(output_embeddings).__
name__}")
                raise AttributeError(f"Qwen2Model 处理失败: {str(e)}")
        # 检查模型类型,确定输出结构
        elif isinstance(base_model, transformers.GPT2LMHeadModel) or \
             isinstance(base_model, transformers.LlamaForCausalLM) or \
             (qwen_class is not None and isinstance(base_model, qwen_class)):
            # 生成模型输出
            outputs = base_model(
                inputs_embeds = input_with_prompt,
                attention_mask = new_attention_mask,
                labels = new_labels,
                return_dict = True
            )
            loss = outputs.loss
            logits = outputs.logits
        else:
            # 基础模型输出(不含 logits)
            outputs = base_model(
                inputs_embeds = input_with_prompt,
                attention_mask = new_attention_mask,
                return_dict = True
            )
            # 尝试多种方式获取 logits
            logits = None
            try:
                # 尝试常见的 lm_head
                if hasattr(base_model, 'lm_head'):
                    lm_head = base_model.lm_head
                    if callable(lm_head):
                        logits = lm_head(outputs.last_hidden_state)
                    else:
                        raise AttributeError("lm_head 不可调用")
                elif hasattr(base_model, 'decoder') and hasattr(base_model.decoder, 'lm_head'):
                    lm_head = base_model.decoder.lm_head
                    if callable(lm_head):
                        logits = lm_head(outputs.last_hidden_state)
                    else:
                        raise AttributeError("decoder.lm_head 不可调用")
                elif hasattr(base_model, 'get_output_embeddings'):
                    # 某些模型需要通过此方法获取输出层
                    output_embeddings = base_model.get_output_embeddings()
                    if output_embeddings is not None and callable(output_embeddings):
                        logits = output_embeddings(outputs.last_hidden_state)
                    else:
                        raise AttributeError("get_output_embeddings()返回 None 或不可调用")
                else:
                    # 尝试获取最终层的输出
                    final_hidden_state = outputs.last_hidden_state
```

```
                    # 假设最后一个维度是隐藏状态大小
                    vocab_size = base_model.config.vocab_size
                    # 创建一个线性层作为最后的分类器
                    linear_layer = torch.nn.Linear(
                        final_hidden_state.shape[-1],
                        vocab_size,
                        bias=False,
                        dtype=final_hidden_state.dtype
                    ).to(final_hidden_state.device)
                    # 随机初始化权重
                    torch.nn.init.xavier_uniform_(linear_layer.weight)
                    logits = linear_layer(final_hidden_state)
                    print("警告:使用了默认的线性层来生成 logits,这可能不是最优的。")
                    print(f"模型类型:{type(base_model).__name__}")
                except Exception as e:
                    raise AttributeError(f"无法从基础模型中获取 logits: {str(e)}")
                # 手动计算损失
                if new_labels is not None and logits is not None:
                    loss_fct = torch.nn.CrossEntropyLoss()
                    loss = loss_fct(
                        logits.view(-1, logits.size(-1)),
                        new_labels.view(-1)
                    )
                else:
                    loss = None
            # 返回损失和其他输出
            return {
                'loss': loss,
                'logits': logits,
                'past_key_values': outputs.get('past_key_values', None)
            }
        def _convert_to_tensor(self, data, device):
            """递归转换嵌套列表为张量"""
            if isinstance(data, list):
                # 处理混合类型列表
                if len(data) == 0:
                    return torch.tensor([], dtype=torch.long, device=device).unsqueeze(0)
                converted = []
                for item in data:
                    converted.append(self._convert_to_tensor(item, device))
                # 检查所有元素是否为相同维度的张量
                if all(t.dim() == converted[0].dim() for t in converted):
                    try:
                        # 尝试堆叠张量
                        return torch.stack(converted, dim=0)
                    except RuntimeError:
                        # 堆叠失败,可能是形状不一致
                        pass
                # 如果无法堆叠,转换为列表形式的张量
                return torch.tensor([t.tolist() for t in converted], dtype=torch.long, device=device)
            elif isinstance(data, torch.Tensor):
                return data.to(device)
            elif isinstance(data, int):
```

```
            # 直接处理整数
            return torch.tensor([data], dtype = torch.long, device = device)
        else:
            # 尝试将其他类型转换为整数
            try:
                return torch.tensor([int(data)], dtype = torch.long, device = device)
            except (ValueError, TypeError):
                raise TypeError(f"无法将类型 {type(data).__name__} 转换为整数张量")
    def _get_base_model(self, model):
        """递归获取最内层的原始模型"""
        while hasattr(model, 'model') and isinstance(model, torch.nn.Module):
            model = model.model
        return model
def load_model_and_infer(checkpoint_path, model, tokenizer, device, max_length = 512):
    """
    加载训练好的 prompt 参数并进行推理
    参数:
    - checkpoint_path: 保存的 prompt 参数路径 (final_prompt_encoder.pt)
    - model: 预训练模型实例
    - tokenizer: 分词器
    - device: 推理设备
    - max_length: 最大生成长度
    """
    # 创建 PromptTuning 模型
    prompt_tuning_model = PromptTuning(
        model = model,
        tokenizer = tokenizer,
        prompt_length = 20,                    # 应与训练时的设置一致
        n_token = 5                            # 应与训练时的设置一致
    )
    # 加载训练好的 prompt 参数
    prompt_tuning_model.template.prompt_encoder.load_state_dict(
        torch.load(checkpoint_path, map_location = device)
    )
    # 将模型移至设备
    prompt_tuning_model.to(device)
    prompt_tuning_model.eval()
    print(f"已加载模型参数: {checkpoint_path}")
    def infer(text, temperature = 0.7, top_p = 0.9):
        """执行推理"""
        # 准备输入
        inputs = tokenizer(
            text,
            return_tensors = "pt",
            padding = True,
            truncation = True,
            max_length = max_length
        )
        # 将输入移至设备
        inputs = {k: v.to(device) for k, v in inputs.items()}

        # 生成回答 - 修改:正确处理 Qwen2Model 的生成
        with torch.no_grad():
```

```
            # 获取基础模型
            base_model = prompt_tuning_model._get_base_model(prompt_tuning_model.model)
            # 获取原始输入嵌入
            input_embeds = base_model.get_input_embeddings()(inputs["input_ids"])
            # 获取 Prompt 嵌入
            prompt_embeds = prompt_tuning_model.template()
            # 扩展 Prompt 嵌入以匹配批次大小
            batch_size = input_embeds.shape[0]
            prompt_embeds_expanded = prompt_embeds.unsqueeze(0).expand(batch_size, -1, -1)
            # 拼接 Prompt 和输入嵌入
            input_with_prompt = torch.cat([prompt_embeds_expanded, input_embeds], dim=1)
            # 调整注意力掩码
            prompt_attention_mask = torch.ones(
                batch_size, prompt_embeds.shape[0],
                device=input_embeds.device, dtype=inputs["attention_mask"].dtype
            )
            new_attention_mask = torch.cat([prompt_attention_mask, inputs["attention_
mask"]], dim=1)

            # 检查模型类型,使用正确的生成方法
            if isinstance(base_model, transformers.models.qwen2.modeling_qwen2.Qwen2Model):
                # Qwen2Model 没有 generate 方法,需要手动处理
                # 这里假设外部传入的 model 是 Qwen2ForCausalLM 类型
                lm_model = prompt_tuning_model.model
                # 使用带有 Prompt 的嵌入进行生成
                outputs = lm_model.generate(
                    inputs_embeds=input_with_prompt,
                    attention_mask=new_attention_mask,
                    max_length=max_length,
                    temperature=temperature,
                    top_p=top_p,
                    pad_token_id=tokenizer.pad_token_id
                )
            else:
                # 其他模型类型使用基础模型的 generate 方法
                outputs = base_model.generate(
                    inputs_embeds=input_with_prompt,
                    attention_mask=new_attention_mask,
                    max_length=max_length,
                    temperature=temperature,
                    top_p=top_p,
                    pad_token_id=tokenizer.pad_token_id
                )
        # 解码生成的回答
        response = tokenizer.decode(outputs[0], skip_special_tokens=True)
        # 提取生成的部分(去除输入)
        input_length = len(tokenizer.decode(inputs["input_ids"][0], skip_special_tokens=True))
        generated_text = response[input_length:].strip()
        return generated_text
    return infer

# 加载预训练模型和分词器
model_name = "./model_path/DeepSeek-R1-distill-Qwen-1.5B_ori"
```

```
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype = torch.float16,
    device_map = "auto"
)

# 设备设置
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 加载模型并获取推理函数
infer_fn = load_model_and_infer(
    checkpoint_path = "./autodl - tmp/fine_tuned_deepseek/final_prompt_encoder.pt",
    model = model,
    tokenizer = tokenizer,
    device = device
)

# 推理
question_input = "什么是防火墙?"
response = infer_fn(question_input)
print(f"模型回答: {response}")
```

3.7　大模型微调策略

大模型参数微调本质上是在高维参数空间中寻找最优约束优化问题。我们可以采用有监督微调、强化学习微调等策略来定义和求解这个约束优化问题。读者可以根据自己的应用场景，选用相应的微调策略。

3.7.1　监督微调

监督微调(Supervised Fine-Tuning，SFT)是在预训练模型的基础上，利用标注的指令、答案等进行监督训练的方法。其核心目标是使模型适应特定任务，例如在 GPT-3 的基础上，通过监督微调生成符合人类指令的响应。有监督微调通过交叉熵损失函数优化模型参数，公式为 $L_{\mathrm{SFT}}=-\frac{1}{n}\sum_{i=1}^{n}\log P(y_i \vee x_i)$，其中 x_i 为当前 Token 之前所有输入序列，y_i 为目标输出 Token。

监督微调的优点是简单直接，适合用于有明确标注数据的任务，如文本分类和机器翻译等。其局限在于，一是依赖高质量标注数据；二是缺乏负反馈机制，无法考虑单个预测错误对全局所造成的影响，无法“向后看”整个生成序列，导致泛化能力有限。

3.7.2　强化学习微调

传统的有监督微调方法虽然能够提高模型生成内容的质量，但其主要关注即时准确性，而非长期回报和对齐性。相比之下，强化学习微调策略更加注重优化长期回报，而非即时准

确性，这使其在模型对齐方面具有独特优势。强化学习微调策略通过引入奖励信号或偏好反馈，使模型能够学习更符合人类期望的行为模式。这种方法允许模型在考虑当前决策的同时，也考虑未来可能的结果，从而生成更加连贯和符合期望的输出。在实际应用中，强化学习微调策略已被证明能够有效提升模型在各种任务中的表现，包括对话生成、代码生成和数学推理等。这些策略通过不断优化模型的行为，使其能够更好地适应不同的应用场景和用户需求。

与传统的有监督微调相比，强化学习微调策略在以下几个方面表现出显著优势：首先，强化学习微调策略能够更好地处理奖励稀疏和延迟的问题，使其在需要长期规划和决策的任务中表现更佳；其次，强化学习微调策略能够通过引入探索机制，使模型能够探索更多可能的生成路径，从而避免陷入局部最优；最后，强化学习微调策略能够更好地处理多目标优化问题，使模型能够在多个目标之间取得平衡，生成更加全面和符合期望的输出。这些优势使得强化学习微调策略成为提升大语言模型性能和对齐性的重要方法。

在众多强化学习微调策略中，近端策略优化（Proximal Policy Optimization，PPO）、人类反馈强化学习（Reinforcement Learning from Human Feedback，RLHF）、直接偏好优化（Direct Preference Optimization，DPO）和组相对策略优化（Group Relative Policy Optimization，GRPO）等方法因其在提升模型性能和对齐性方面的显著效果而备受关注。这些方法各有特点，适用于不同的应用场景，共同构成了强化学习微调策略的核心框架。通过深入研究这些方法的原理、优势和应用，可以更好地理解和应用强化学习微调策略，提升大语言模型的性能和对齐性。

1. 近端策略优化

近端策略优化是强化学习领域中一种广泛使用的策略优化算法，其核心思想是通过限制策略更新的幅度，确保每一步训练都不会偏离当前策略太多，同时高效利用采样数据。在强化学习中，直接优化策略会导致训练不稳定，模型可能因为过大的参数更新而崩溃。近端策略优化通过限制策略更新幅度，防止策略过度偏离，并使用优势函数 $A(s,a)$ 来评价某个动作的相对好坏，从而解决了这一问题。

近端策略优化通过更新策略网络的参数来提高策略在当前环境下的适应性。具体而言，近端策略优化算法采用剪切函数来限制新策略与旧策略之间的差异在给定范围内。这种剪切函数可以是线性、二次或指数函数等。通过使用剪切函数，近端策略优化算法能够平衡策略更新的剧烈程度，从而提升算法的稳定性和收敛速度。这种近端策略优化的方法使得算法在强化学习任务中表现出良好的性能和健壮性。近端策略优化算法的另一个重要特点是其简单性和高效性，不需要维护复杂的数据结构或额外的网络，只需要一个策略网络和一个价值函数网络。同时，近端策略优化算法的计算效率高，能够在较短的时间内完成训练，这对于需要快速迭代和测试的应用场景尤为重要。相比其他算法，近端策略优化具有简单、高效、稳定等优点，因此在学术界和工业界广泛应用。

然而，近端策略优化也面临着一些挑战和限制。首先，算法的性能对超参数设置非常敏感，需要仔细调整才能取得最佳效果。其次，算法在处理高维和连续状态空间时可能会遇到困难。最后，算法在处理非平稳环境时可能会表现不佳，需要特殊的处理方法。

2. 人类反馈强化学习

人类反馈强化学习通过引入人类反馈作为奖励信号，进一步优化模型生成质量，是ChatGPT 等模型的关键技术。其核心思想为：一是基于人类对模型生成结果的排序数据，训练奖励模型预测人类偏好评分；二是使用强化学习算法（如近端策略优化），以奖励模型输出为标量奖励，优化策略网络参数，公式为：$L_{\mathrm{RLHF}}=-E_{x\sim\pi_\theta}[R(x)]$，其中，$R(x)$为奖励模型对序列 x 的评分，π_θ 为策略网络。这种方法引入负反馈，通过奖励模型惩罚低质量输出，提升生成内容的安全性和相关性，从而可以评估整个生成序列，在一定程度上解决了有监督微调无法“向后看”的问题。

人类反馈强化学习的流程通常包括三个主要阶段：监督微调、奖励建模（Reward Modeling，RM）和强化学习（Reinforcement Learning，RL）。在监督微调阶段，模型首先在高质量下游任务数据集上通过监督学习对模型进行微调，获得初始策略 π_{SFT}。在奖励建模阶段，使用 π_{SFT} 生成一对答案 $(y_1,y_2)\pi_{\mathrm{SFT}}(y\vee x)$，然后通过人类标注，得到偏好标签 $(y_w>y_l)\vee x$，其中 y_w 是人类认为更好的答案。这些偏好数据被用于训练一个奖励模型。该奖励模型能够预测给定输入的人类偏好评分。在强化学习阶段，使用训练好的奖励模型作为新的奖励信号，通过强化学习算法（如近端策略优化）优化模型，使其最大化奖励，同时保持与原始模型的相似性。

人类反馈强化学习的优势在于其能够有效地将人类的主观判断和偏好融入模型的学习过程中，使其行为更加符合人类的期望和价值观。通过引入人类的直接反馈，人类反馈强化学习能够避免传统强化学习中奖励函数设计的困难和不确定性，使模型能够在更加复杂和模糊的任务中表现出色。此外，人类反馈强化学习还能够通过人类的反馈不断调整和优化模型，使其逐步逼近人类期望的行为模式，从而实现更好的性能和对齐性。

然而，人类反馈强化学习也面临着一些挑战和限制。首先，人类反馈强化学习通常需要大量的人类反馈数据，这可能会增加训练的成本。其次，人类的反馈可能存在主观性和不一致性，这可能会影响奖励模型的准确性和稳定性。最后，人类反馈强化学习中的强化学习阶段可能面临训练不稳定和样本效率低的问题，需要仔细调整和优化。

3. 直接偏好优化

直接偏好优化直接利用偏好数据优化模型，无须显式训练奖励模型。其核心思想是最大化偏好响应的相对概率，公式为 $L_{\mathrm{DPO}}=-E(x,y_w,y_l)\sim D\left[\log\dfrac{\pi_\theta(y_w\vee x)}{\pi_{\mathrm{ref}}(y_w\vee x)}-\log\dfrac{\pi_\theta(y_l\vee x)}{\pi_{\mathrm{ref}}(y_l\vee x)}\right]$，其中，$y_w$ 和 y_l 分别为偏好和非偏好响应，π_{ref} 为参考模型参数。它的优势在于跳过奖励模型的训练，直接通过分类损失优化策略，降低计算成本和训练复杂度，避免了人类反馈强化学习中奖励模型训练的不稳定性。

直接偏好优化基于奖励函数和最优策略之间的映射，以封闭形式提取出相应的最优策略，从而使标准的人类反馈强化学习问题仅通过简单的分类损失就能解决。直接偏好优化与人类反馈强化学习的优化问题相似，目标函数在初始形式上相同，都是通过最大化奖励函数并最小化 KL 散度，确保奖励最大化且不改变原始分布。直接偏好优化使用 Bradley-

Terry 模型，假设在给定 X 条件下，Y_1 大于 Y_2 的概率由固定奖励函数决定。通过一系列数学推导，直接偏好优化将公式转换为新的 KL 散度形式，从而得到优化后的最优策略。

在实际应用中，直接偏好优化已经被证明能够有效地将大语言模型与人类偏好对齐，甚至在某些方面超过了传统的人类反馈强化学习方法。特别是，在控制生成内容的情感方面，直接偏好优化微调超越了基于近端策略优化的人类反馈强化学习，并且其实施和训练过程更为简便。这使得直接偏好优化成为一种强大且高效的偏好对齐方法，适用于需要模型行为与人类偏好一致的场景。

近端策略优化的一个重要特点是其计算效率高，不需要在微调期间执行大量的超参数调整，大大降低了计算成本和训练复杂性。此外，近端策略优化还具有稳定性高的优势，由于直接优化，近端策略优化在训练过程中更加稳定，不容易出现训练不收敛或性能波动大的问题。这些优势使得近端策略优化成为一种适合大规模应用的偏好对齐方法，能够有效地提升模型的性能和对齐性。

然而，近端策略优化也面临着一些挑战和限制。首先，近端策略优化对偏好数据的质量和数量有较高的要求，需要足够的高质量偏好数据才能取得良好的效果。其次，近端策略优化在处理复杂的偏好关系时可能会遇到困难。最后，近端策略优化在处理非二元偏好数据时可能会表现不佳。

4. 组相对策略优化

组相对策略优化是 DeepSeek 专为大语言模型设计的新型强化学习算法，通过组内相对比较替代绝对价值估计。组相对策略优化的优势在于：一是去掉了近端策略优化中的价值网络，节省了内存并提高计算效率；二是通过组内对比学习，实现长链推理，增强模型在数学推理、代码生成等复杂任务中的表现。组相对策略优化的核心思想是通过在组内比较不同的策略输出，确定哪些策略在特定任务中表现更好，然后优化模型以提高这些策略的概率。这种方法避免了传统强化学习中对价值函数的依赖，简化了算法结构，同时提高了计算效率。

组相对策略优化的工作原理可以分为几个关键步骤：首先，模型根据当前策略生成多个候选输出；其次，使用奖励函数对每个输出进行评分；再次，计算组的平均奖励作为基线，计算每个输出相对于基线的优势；最后，基于计算出的优势更新策略，以最大化高于基线的输出的概率。通过这种方式，组相对策略优化能够有效地优化模型的策略，使其在特定任务中表现更好。

组相对策略优化的一个重要特点是其高效性。相比传统的强化学习算法，组相对策略优化不需要维护复杂的价值网络，这大大简化了算法结构，提高了计算效率。此外，组相对策略优化还通过组内比较的方式，减少了对样本的需求量，进一步提高了训练效率。这些特点使组相对策略优化成为一种适合训练大语言模型的高效算法。

组相对策略优化的另一个重要特点是擅长处理长链推理任务。在数学推理和代码生成等需要长链推理的任务中，传统的强化学习算法可能会因为价值估计的不准确或不稳定而表现不佳。而组相对策略优化通过组内相对比较，避免了绝对价值估计的困难，能够更准确地评估策略的好坏，从而更好地优化模型的推理能力。这使得组相对策略优化在处理复杂的推理任务时表现出色，能够有效地提升模型的性能和准确性。

然而，组相对策略优化也面临着一些挑战和限制。首先，组相对策略优化对奖励函数的设计有较高的要求，需要设计能够准确反映任务目标的奖励函数。其次，组相对策略优化在处理高维和连续状态空间时可能会遇到困难。最后，组相对策略优化在处理非平稳环境时可能会表现不佳。

5. 强化学习微调策略的比较与选择

在选择强化学习微调策略时，需要考虑多个因素，包括任务特性、数据可用性、计算资源和预期效果等。对于需要快速迭代和测试的任务，近端策略优化可能是一个更好的选择，因为其简单性和高效性使其易于实现和维护。对于需要将人类偏好和价值观融入模型学习过程的任务，人类反馈强化学习可能是一个更好的选择，因为其能够有效地捕捉人类的主观判断和偏好。对于需要直接在人类偏好数据上优化模型的任务，近端策略优化可能是一个更好的选择，因为其能够简化偏好对齐的过程，降低计算成本和训练复杂度。对于需要在长链推理任务中表现良好的任务，组相对策略优化可能是一个更好的选择，因为其通过组内相对比较，能够更准确地评估策略的好坏，从而更好地优化模型的推理能力。

3.8　本章小结

本章围绕大模型微调展开阐述，系统给出了大模型微调的基础知识、核心理论、数据准备、技术方法、实施策略及工具框架。

首先，明确了大模型微调的基本概念，给出从全参数微调、部分参数微调到参数高效微调的发展历程。其次，剖析了支撑微调技术的核心理论，高维非凸优化揭示了微调过程中参数优化的复杂特性，病态曲率分析解释了模型训练的挑战，流形嵌入假说指出微调可使模型参数在特定流形上找到更优解，损失景观的多尺度特性为优化策略提供了理论依据。再次，给出了数据准备的重要性、来源与收集、数据预处理方法，为微调提供可靠输入。然后，详细介绍了全参数微调、部分参数微调和新增参数微调 3 类方法。最后，介绍了监督微调与强化学习两种大模型微调策略。

思考题

1. 大模型微调核心理论对垂直领域大模型的训练和部署有哪些指导意义？

2. 结合具体应用场景，说明如何准备垂直领域大模型训练数据集，以及选择适合的预训练模型？

3. 结合具体应用场景，选择哪种微调方式（全参数微调、部分参数微调和新增参数微调）来训练垂直领域大模型？

4. 试用数学表达式描述新增参数微调（如适配器），能够既保持预训练模型的原始能力，又能获得在垂直领域的新能力。

5. 当目标任务与预训练模型使用的数据差异很大的情况下，LoRA 方法的低秩假设会失效。这种情况下，LoRA 方法会显著劣于全参数微调。那么，如何判断和验证低秩假设

失效？

6. 比较分析 Prefix Tuning、Prompt Tuning 和 P-tuning 三者的异同之处，指出它们的优势、劣势以及适合的应用场景。

练习题

1. 大模型微调的几个关键步骤和相应的内容是什么？

2. 在 HuggingFace 或 ModelScope 社区查看有关 deepseek-r1-distill-qwen：1.5b 模型的描述，请描述模型的 Token 数量、模型预训练的数据格式。

3. 用自建的知识库，采用 AdaLOMO 全参数微调方法微调 deepseek-r1-distill-qwen：1.5b 模型。若出现梯度消失的现象，可以采用什么方式解决？

4. 使用 LoRA 微调方法时，如何选择设置关键超参数秩 r？

5. 用 Prefix Tuning 微调方法，如何处理训练数据？关键技术有哪些？