4.lora

1.LoRA

1.1 背景

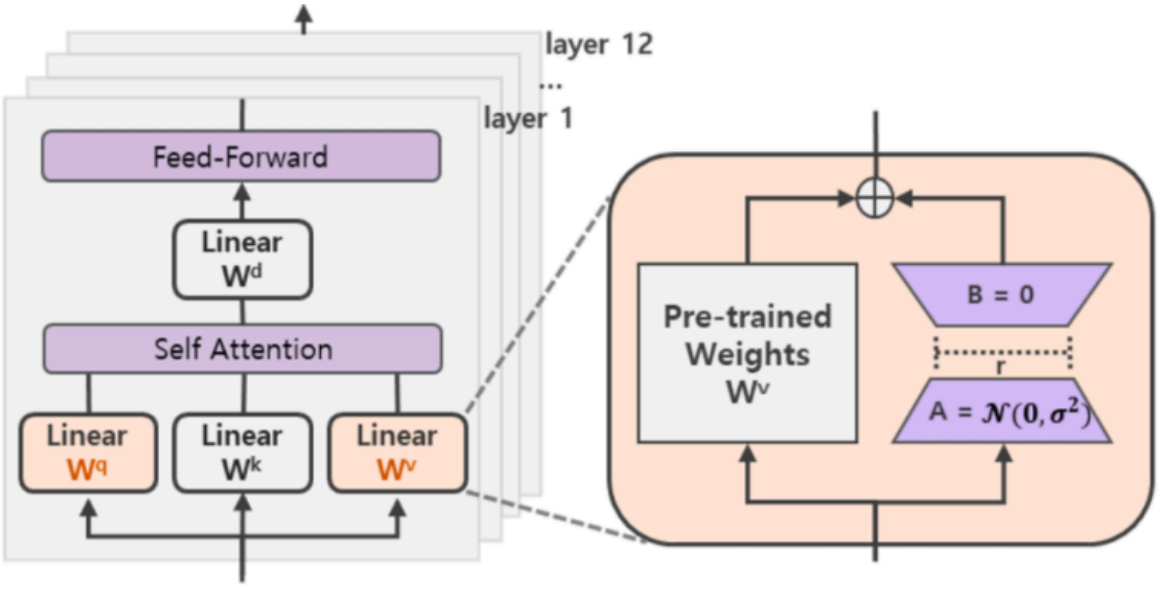
神经网络包含很多全连接层，其借助于矩阵乘法得以实现，然而，很多全连接层的权重矩阵都是满秩的。当针对特定任务进行微调后，**模型中权重矩阵其实具有很低的本征秩**（intrinsic rank），因此，论文的作者认为**权重更新的那部分参数矩阵尽管随机投影到较小的子空间，仍然可以有效的学习，可以理解为针对特定的下游任务这些权重矩阵就不要求满秩。**

> **Note**
>
> 什么是“本征秩”?
>
> - 矩阵的“秩”表示它能张成的空间维度
> - 如果一个 768×768 的 ΔW 实际上只在 8 维子空间中有变化，其余方向几乎为 0，
>   那么它的**有效秩 ≈ 8**（远小于 768）

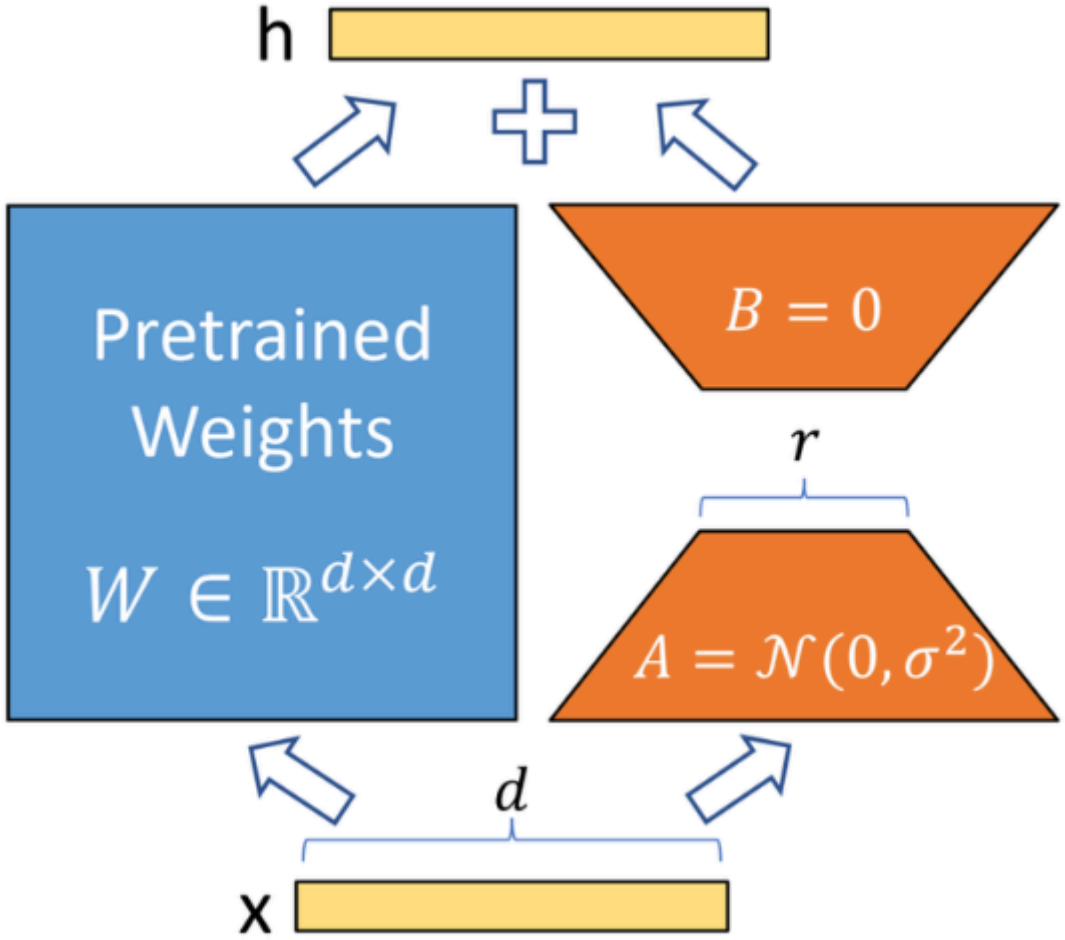
1.2 技术原理

LoRA（论文：**LoRA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS**），该方法的核心思想就是**通过低秩分解来模拟参数的改变量，从而以极小的参数量来实现大模型的间接训练。**

在涉及到矩阵相乘的模块，在原始的PLM旁边增加一个新的通路，通过前后两个矩阵A,B相乘，第一个矩阵A负责降维，第二个矩阵B负责升维，中间层维度为r，从而来模拟所谓的本征秩（intrinsic rank）。



可训练层维度和预训练模型层维度一致为`d`，先将维度`d`通过全连接层降维至`r`，再从`r`通过全连接层映射回`d`维度，其中，`r<<d`，r是矩阵的秩，这样矩阵计算就从`d x d`变为`d x r + r x d`，参数量减少很多。



在下游任务训练时，固定模型的其他参数，只优化新增的两个矩阵的权重参数，将PLM跟新增的通路两部分的结果加起来作为最终的结果（两边通路的输入跟输出维度是一致的），即 $h = Wx + BAx$。第一个矩阵的A的权重参数会通过高斯函数初始化，而第二个矩阵的B的权重参数则会初始化为零矩阵，这样能保证训练开始时新增的通路 $BA = 0$ 从而对模型结果没有影响。

$$h = W_0x + \Delta Wx = W_0x + BAx$$

在推理时，将左右两部分的结果加到一起即可，$h = Wx + BAx = (W + BA)x$，所以只要将训练完成的矩阵乘积 BA 跟原本的权重矩阵 W 加到一起作为新权重参数替换原本PLM的 W 即可，对于推理来说，不会增加额外的计算资源。

此外，Transformer的权重矩阵包括Attention模块里用于计算query, key, value的 W_q，W_k，W_v 以及多头attention的 W_o，以及MLP层的权重矩阵，LoRA只应用于Attention模块中的4种权重矩阵，而且通过消融实验发现同时调整 W_q 和 W_v 会产生最佳结果。

实验还发现，**保证权重矩阵的种类的数量比起增加隐藏层维度 r 更为重要，增加 r 并不一定能覆盖更加有意义的子空间。**

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL ($\pm$0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI ($\pm$0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

	Weight Type	$r=1$	$r=2$	$r=4$	$r=8$	$r=64$
WikiSQL(±0.5%)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI (±0.1%)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

那么关于秩的选择，通常情况下，rank为4，8，16即可。

通过实验也发现，在众多数据集上LoRA在只训练极少量参数的前提下，最终在性能上能和全量微调匹配，甚至在某些任务上优于全量微调。

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm.2}$	$\mathbf{91.9}_{\pm.1}$	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
$\text{RoB}_{\text{large}}$ (Adpt^{P})†	0.8M	$\mathbf{90.5}_{\pm.3}$	$\mathbf{96.6}_{\pm.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm.3}$	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	6.0M	$89.9_{\pm.5}$	$96.2_{\pm.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm.2}$	$92.1_{\pm.1}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	87.8
$\text{RoB}_{\text{large}}$ (Adpt^{H})†	0.8M	$90.3_{\pm.3}$	$96.3_{\pm.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm.2}$	$91.5_{\pm.1}$	$72.9_{\pm2.9}$	$91.5_{\pm.5}$	86.4
$\text{RoB}_{\text{large}}$ (LoRA)†	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.2}_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm.3}$	$91.6_{\pm.2}$	$\mathbf{85.2}_{\pm1.1}$	$\mathbf{92.3}_{\pm.5}$	**88.6**
DeB_{XXL} (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeB_{XXL} (LoRA)	4.7M	$\mathbf{91.9}_{\pm.2}$	$96.9_{\pm.2}$	$\mathbf{92.6}_{\pm.6}$	$\mathbf{72.4}_{\pm1.1}$	$\mathbf{96.0}_{\pm.1}$	$\mathbf{92.9}_{\pm.1}$	$\mathbf{94.9}_{\pm.4}$	$\mathbf{93.0}_{\pm.2}$	**91.3**

. $\text{RoBERTa}_{\text{base}}$, $\text{RoBERTa}_{\text{large}}$, and $\text{DeBERTa}_{\text{XXL}}$ with different adaptation methods on the GLUE benchmark. We report the overall (matched and mismatched) accuracy for MNLI, Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for other tasks. Higher is better for all metrics. * indicates numbers published in prior works. † indicates runs configured in a setup similar to Houlsby et al. (2019) for a fair comparison.

2.AdaLoRA

2.1 背景

在NLP领域，对于下游任务进行大型预训练语言模型的微调已经成为一种重要的做法。一般而言，我们会采用对原有的预训练模型进行全量微调的方法来适配下游任务，但这种方法存在两个问题。

- **训练阶段**。对于预训练模型进行微调的时候，为了更新权重参数，需要大量的显存来存储参数的梯度和优化器信息，在当今预训练模型的参数变得越来越大的情况下，针对下游任务微调门槛变得越来越高。
- **推理阶段**。由于我们训练的时候是对于模型参数进行全量的更新，所以多个下游任务需要为每个任务维护一个大型模型的独立副本，这样就导致我们在实际应用的时候浪费了不必要的存储。

为了解决这些问题，研究者提出了两个主要研究方向，以减少微调参数的数量，同时保持甚至提高预训练语言模型的性能。

- **方向一：添加小型网络模块**：将小型网络模块添加到PLMs中，保持基础模型保持不变的情况下仅针对每个任务微调这些模块，可以用于所有任务。这样，只需引入和更新少量任务特定的参数，就可以适配下游的任务，大大提高了预训练模型的实用性。如：Adapter tuning、Prefix tuning、Prompt Tuning等，这类方法虽然大大减少了内存消耗。但是这些方法存在一些问题，比如：

Adapter tuning引入了推理延时；Prefix tuning或Prompt tuning直接优化Prefix和Prompt是非单调的，比较难收敛，并且消耗了输入的token。

- **方向二：下游任务增量更新**：对预训练权重的增量更新进行建模，而无需修改模型架构，即W=W0+△W。比如：Diff pruning、LoRA等， 此类方法可以达到与完全微调几乎相当的性能，但是也存在一些问题，比如：Diff pruning需要底层实现来加速非结构化稀疏矩阵的计算，不能直接使用现有的框架，训练过程中需要存储完整的ΔW矩阵，相比于全量微调并没有降低计算成本。LoRA则需要预先指定每个增量矩阵的本征秩 r 相同，忽略了在微调预训练模型时，权重矩阵的重要性在不同模块和层之间存在显著差异，并且只训练了Attention，没有训练FFN，事实上FFN更重要。

方法类型	怎么做	优点	缺点
加小模块（Adapter/Prompt）	在模型上“插件”，只训插件	参数少、省显存、多任务共享主干	推理慢 / 训练难收敛 / 占输入长度
增量更新（LoRA/Diff）	原始权重 + 小调整量 ΔW	不改结构、效果接近全量微调	LoRA 秩固定、忽略 FFN；Diff 难加速

LoRA 的两个主要局限确实是：

1. **统一秩 r** → 无法适应不同层/模块的差异化需求；
2. **只更新 Attention** → 忽略了可能更重要的 FFN 层。

> 问题 1：**Diff pruning 指的是什么技术?**
>
> **Diff pruning**（差异剪枝）是一种“增量微调”方法，它的核心思想是：
>
> > **只让模型的权重发生“稀疏的、微小的变化”**，而不是全部更新。
>
> 具体来说：
>
> - 假设原始预训练模型的权重是 **W_0**（冻结不动）。
> - 微调时，我们学习一个**增量变化量 ΔW**，最终用的权重是：**$W = W_0 + \Delta W$**。
> - 但 Diff pruning 要求 **ΔW 是“稀疏”的**——也就是说，ΔW 中**绝大多数元素是 0**，只有极少数位置是非零的（比如 0.1% 的参数被修改）。
>
> 这就像你在一本厚厚的书中，只允许用荧光笔标出**几个关键错别字**来修正，其他地方一律不能动。
>
> **为什么叫 “pruning”（剪枝）?**
>
> 因为“稀疏”本质上就是一种剪枝：把不重要的更新“剪掉”（设为 0），只保留重要的。
>
> **但为什么说它“计算成本没降”?**
>
> 虽然 ΔW 很稀疏（比如 99% 是 0），但在**训练过程中**：
>
> - 你仍然需要**存储整个 ΔW 矩阵**（哪怕全是 0，也要占内存位置）。
> - 而且，**普通深度学习框架**（如 PyTorch、TensorFlow）**默认不支持高效的稀疏矩阵运算。**它们是为“稠密矩阵”（dense）优化的。
> - 所以即使 ΔW 很稀疏，GPU 还是会像处理普通矩阵一样去算，**既没省显存，也没提速。**
> - 只有在**推理阶段**，如果你有专门的稀疏计算硬件或编译器（比如 NVIDIA 的稀疏 Tensor Core），才能真正加速。

基于以上问题进行总结：

- 第一，我们不能预先指定矩阵的秩，需要动态更新增量矩阵的R，因为权重矩阵的重要性在不同模块和层之间存在显著差异。
- 第二，需要找到更加重要的矩阵，分配更多的参数，裁剪不重要的矩阵。找到重要的矩阵，可以提升模型效果；而裁剪不重要的矩阵，可以降低参数计算量，降低模型效果差的风险。

为了弥补这一差距，作者提出了AdaLoRA，它根据权重矩阵的重要性得分，在权重矩阵之间自适应地分配参数预算。

2.2 技术原理

AdaLoRA（论文：*ADAPTIVE BUDGET ALLOCATION FOR PARAMETER-EFFICIENT FINE-TUNING*），是对LoRA的一种改进，它根据重要性评分动态分配参数预算给权重矩阵。具体做法如下：

- **调整增量矩阵分配**：AdaLoRA将关键的增量矩阵分配高秩以捕捉更精细和任务特定的信息，而将较不重要的矩阵的秩降低，以防止过拟合并节省计算预算。
- **以奇异值分解（SVD）的形式对增量更新进行参数化**，并根据重要性指标裁剪掉不重要的奇异值，同时保留奇异向量。由于对一个大矩阵进行精确SVD分解的计算消耗非常大，这种方法通过减少它们的参数预算来加速计算，同时保留未来恢复的可能性并稳定训练。

$$W = W^{(0)} + \Delta = W^{(0)} + P\Lambda Q$$

- 在训练损失中添加了额外的惩罚项，以规范奇异矩阵 P 和 Q 的正交性，从而避免SVD的大量计算并稳定训练。

通过实验证明，AdaLoRA 实现了在所有预算、所有数据集上与现有方法相比，性能更好或相当的水平。例如，当参数预算为 0.3M 时，AdaLoRA 在 RTE 数据集上，比表现最佳的基线（Baseline）高 1.8%。

> **Tip**
>
> SVD（Singular Value Decomposition，奇异值分解）是线性代数中一种非常强大的矩阵分解方法。
>
> 对于任意一个实数矩阵 Δ（比如 LoRA 中的增量权重），SVD 可以把它拆成三个部分：
>
> $$\Delta = P\Lambda Q^{\top}$$
>
> 其中：
>
> - P 是一个左奇异向量矩阵（列向量正交），
> - Q 是一个右奇异向量矩阵（列向量也正交），
> - Λ（Lambda）是一个对角矩阵，对角线上的数叫奇异值（singular values），通常从大到小排列：$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$。
>
> 你可以把 SVD 想象成：把一个复杂的“变化”（Δ）拆解成“方向 × 强度 × 方向”的组合。
> 奇异值 σ 越大，说明这个方向上的变化越重要！

> **Note**
>
> 在标准 **LoRA** 中，我们假设：
>
> > $\Delta \approx A \times B$，其中 A 和 B 是随机初始化的小矩阵（秩为 r）。
>
> 但这里有个问题：**A 和 B 是任意的低秩表示**，不一定对应“最重要的变化方向”。
>
> 而 **AdaLoRA 的核心思想是**：

> **不是所有方向都 equally important**（同等重要）！
> 我们应该把有限的参数预算（比如总共只能用 0.3M 个参数），**优先分配给那些"变化最大、对任务最关键"的方向。**

于是，AdaLoRA **用 SVD 的视角来看待 Δ**：

- 把 Δ 看作是由多个"奇异方向"组成的；
- 每个方向的重要性由对应的**奇异值 σ 的大小**决定；
- **大的 σ → 重要 → 分配高秩（保留）**；
  小的 σ → 不重要 → 降低秩甚至裁剪掉。

Results with DeBERTaV3-base on GLUE development set. The best results on each dataset are shown in **bold**. We report the average correlation for STS-B. *Full FT*, *HAdapter* and *PAdapter* represent full fine-tuning, Houlsby adapter, and Pfeiffer adapter respectively. We report mean of 5 runs using different random seeds.

Method	# Params	MNLI m/mm	SST-2 Acc	CoLA Mcc	QQP Acc/F1	QNLI Acc	RTE Acc	MRPC Acc	STS-B Corr	All Ave.
Full FT	184M	89.90/90.12	95.63	69.19	**92.40/89.80**	94.03	83.75	89.46	91.60	88.09
BitFit	0.1M	89.37/89.91	94.84	66.96	88.41/84.95	92.24	78.70	87.75	91.35	86.02
HAdapter	1.22M	90.13/90.17	95.53	68.64	91.91/89.27	94.11	84.48	89.95	91.48	88.12
PAdapter	1.18M	90.33/90.39	95.61	68.77	92.04/89.40	94.29	85.20	89.46	91.54	88.24
LoRA$_{r=8}$	1.33M	90.65/90.69	94.95	69.82	91.99/89.38	93.87	85.20	89.95	91.60	88.34
AdaLoRA	1.27M	**90.76/90.79**	**96.10**	**71.45**	**92.23/89.74**	**94.55**	**88.09**	**90.69**	**91.84**	**89.31**
HAdapter	0.61M	90.12/90.23	95.30	67.87	91.65/88.95	93.76	85.56	89.22	91.30	87.93
PAdapter	0.60M	90.15/90.28	95.53	69.48	91.62/88.86	93.98	84.12	89.22	91.52	88.04
HAdapter	0.31M	90.10/90.02	95.41	67.65	91.54/88.81	93.52	83.39	89.25	91.31	87.60
PAdapter	0.30M	89.89/90.06	94.72	69.06	91.40/88.62	93.87	84.48	89.71	91.38	87.90
LoRA$_{r=2}$	0.33M	90.30/90.38	94.95	68.71	91.61/88.91	94.03	85.56	89.71	**91.68**	88.15
AdaLoRA	0.32M	**90.66/90.70**	**95.80**	**70.04**	**91.78/89.16**	**94.49**	**87.36**	**90.44**	91.63	**88.86**

3.QLoRA

3.1 背景

微调大型语言模型 (LLM) 是提高其性能以及添加所需或删除不需要的行为的一种非常有效的方法。然而，微调非常大的模型非常昂贵；以 LLaMA 65B 参数模型为例，常规的 16 bit微调需要超过 780 GB 的 GPU 内存。

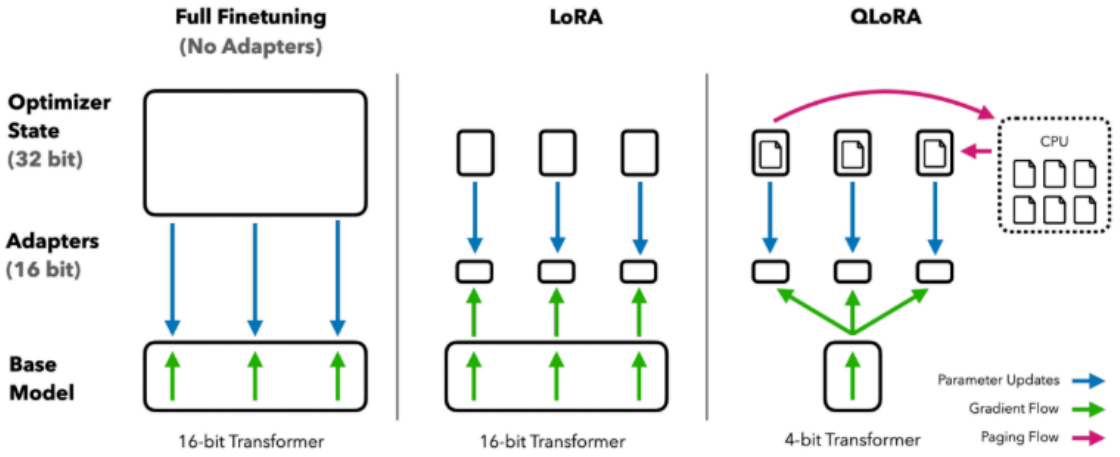
虽然最近的量化方法可以减少 LLM 的内存占用，但此类技术仅适用于推理场景。

基于此，作者提出了QLoRA，并首次证明了**可以在不降低任何性能的情况下微调量化为 4 bit的模型。**

3.2 技术原理

- QLoRA（论文： **QLORA: Efficient Finetuning of Quantized LLMs**），使用**一种新颖的高精度技术将预训练模型量化为 4 bit，然后添加一小组可学习的低秩适配器权重，这些权重通过量化权重的反向传播梯度进行微调。**QLORA 有一种低精度存储数据类型（4 bit），还有一种计算数据类型（BFloat16）。实际上，这意味着无论何时使用 QLoRA 权重张量，我们都会将张量反量化为 BFloat16，然后执行 16 位矩阵乘法。QLoRA提出了两种技术实现高保真 4 bit微调——4 bit NormalFloat(NF4) 量化和双量化。此外，还引入了分页优化器，以防止梯度检查点期间的内存峰值，从而导致内存不足的错误，这些错误在过去使得大型模型难以在单台机器上进行微调。具体说明如下：
  - **4bit NormalFloat**（NF4）：对于正态分布权重而言，一种信息理论上最优的新数据类型，该数据类型对正态分布数据产生比 4 bit整数和 4bit 浮点数更好的实证结果。
  - **双量化**：对第一次量化后的那些常量再进行一次量化，减少存储空间。

- **分页优化器**：使用NVIDIA统一内存特性，该特性可以在在GPU偶尔OOM的情况下，进行CPU和GPU之间自动分页到分页的传输，以实现无错误的 GPU 处理。该功能的工作方式类似于 CPU 内存和磁盘之间的常规内存分页。使用此功能为优化器状态（Optimizer）分配分页内存，然后在 GPU 内存不足时将其自动卸载到 CPU 内存，并在优化器更新步骤需要时将其加载回 GPU 内存。



Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

> **Note**
>
> QLoRA 的目标是：**让超大语言模型**（比如 Llama-7B、13B）
>
> 它在 LoRA 的基础上，加了**三层关键技术**：
>
> | 技术 | 作用 |
> | --- | --- |
> | **4-bit 量化**（NF4） | 把原始模型 W_0 从 16 位压缩到**4 位**，大幅降低显存占用 |
> | **双量化**（Double Quantization） | 进一步压缩量化过程中的"缩放因子"，省更多内存 |
> | **分页优化器**（Paged Optimizer） | 防止训练时因显存峰值导致 OOM（内存溢出） |
>
> 最关键的是：
>
> > **QLoRA = 4-bit 量化的大模型 + LoRA 微调**
>
> 也就是说：
>
> - **原始模型 W_0 被量化成 4-bit**（用 NF4 存储，非常省显存）；
> - **LoRA 的增量矩阵 ΔW 仍然用 16 位**（bfloat16）训练；
> - **计算时，4-bit 的 W_0 会被临时"反量化"成 16 位**，再和 LoRA 一起做矩阵乘法。

> **Tip**
>
> 先回顾：**普通量化是怎么做的**？
>
> 在 QLoRA 中，我们要把一个大模型的权重（比如 float16 的张量 **W**）压缩成 **4-bit**，以节省显存。
>
> 最常用的量化方法是 **分组量化**（Group-wise Quantization），比如：
>
> 1. 把权重 **W** 分成很多小组（比如每组 128 个数）；
> 2. 对每一组：
>   - 找到最大值和最小值（或标准差）→ 得到一个**缩放因子**（scale）；

- 用这个 scale 把这组数映射到 4-bit 整数（比如 0~15）；

3. 存储两样东西：
   - **量化后的 4-bit 整数**（占 4 bit/数）；
   - **对应的 scale**（通常是 float16，占 16 bit）。

这样做可以大幅压缩权重（比如从 16 bit → 4 bit），但 **scale 本身也需要存储**！

比如：

- 一个 7B 模型有 70 亿个参数；
- 如果每 128 个参数一组，就有约 5500 万个 scale；
- 每个 scale 是 float16（2 字节）→ 光 scale 就要 **110 MB**！

虽然比原始模型小很多，但**scale 的存储开销仍然不可忽略**，尤其当模型更大时。

那"双量化"是怎么优化的?

双量化（Double Quantization）的核心思想是：

> **既然 scale 本身也是一堆浮点数，而且它们通常比较集中（比如都接近某个值），那我们也可以对 scale 再做一次量化！**

也就是说：**量化两次**！

第一次量化（主权重）：

- 把原始权重 **W**（float16）→ 量化为 **4-bit 整数 + 第一层 scale（S_1）**

第二次量化（对 scale 本身）：

- 把第一层的 scale **S_1**（float16）→ 再量化为 **8-bit 整数 + 第二层 scale（S_2）**

这样，我们最终存储的是：

- 4-bit 的权重；
- 8-bit 的第一层 scale（S_1 被压缩了）；
- 一个很小的第二层 scale（S_2，通常只有一个或几个值）。

实验证明，无论是使用16bit、8bit还是4bit的适配器方法，都能够复制16bit全参数微调的基准性能。这说明，尽管量化过程中会存在性能损失，但通过适配器微调，完全可以恢复这些性能。

Experiments comparing 16-bit BrainFloat (BF16), 8-bit Integer (Int8), 4-bit Float (FP4), and 4-bit NormalFloat (NF4) on GLUE and Super-NaturalInstructions. QLoRA replicates 16-bit LoRA and full-finetuning.

Dataset	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
Model	RoBERTa-large	T5-80M	T5-250M	T5-780M	T5-3B	T5-11B
BF16	88.6	40.1	42.1	48.0	54.3	62.0
BF16 replication	88.6	40.0	42.2	47.3	54.9	-
LoRA BF16	88.8	40.5	42.6	47.1	55.4	60.7
QLoRA Int8	88.8	40.4	42.9	45.4	56.5	60.7
QLoRA FP4	88.6	40.3	42.4	47.5	55.6	60.9
QLoRA NF4 + DQ	-	40.4	42.7	47.7	55.3	60.9

实验还比较了不同的4bit数据类型对效果（zero-shot均值）的影响，其中，NFloat 显著优于Float，而NFloat + DQ略微优于NFloat，虽然DQ对精度提升不大，但是对于内存控制效果更好。

除此之外，论文中还对不同大小模型、不同数据类型、在 MMLU数据集上的微调效果进行了对比。使用QLoRA（NFloat4 + DQ）可以和Lora(BFloat16)持平，同时， 使用QLORA（ FP4）的模型效果落后于前两者一个百分点。

类型	设计理念	适合场景
FP4	通用浮点格式	硬件友好，但不适合 LLM 权重
NF4	为正态分布定制	**LLM 量化首选**，精度高
NF4 + DQ	NF4 + 压缩 scale	实际部署首选，省内存不损精度

Mean 5-shot MMLU test accuracy for LLaMA 7-65B models finetuned with adapters on Alpaca and FLAN v2 for different data types. Overall, NF4 with double quantization (DQ) matches BFloat16 performance, while FP4 is consistently one percentage point behind both.

	Mean 5-shot MMLU Accuracy								
LLaMA Size	7B		13B		33B		65B		Mean
Dataset	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	
BFloat16	38.4	45.6	47.2	50.6	57.7	60.5	61.8	62.5	53.0
Float4	37.2	44.0	47.3	50.0	55.9	58.5	61.3	63.3	52.2
NFloat4 + DQ	39.0	44.5	47.5	50.7	57.3	59.2	61.8	63.9	53.1

作者在实验中也发现了一些有趣的点，比如：指令调优虽然效果比较好，但只适用于指令相关的任务，在聊天机器人上效果并不佳，而聊天机器人更适合用Open Assistant数据集去进行微调。通过指令类数据集的调优更像是提升大模型的推理能力，并不是为聊天而生的。

总之，QLoRA的出现给大家带来一些新的思考，**不管是微调还是部署大模型，之后都会变得更加容易。**每个人都可以快速利用自己的私有数据进行微调；同时，又能轻松的部署大模型进行推理。