

1. 问题背景

问题-1: 大模型通常包含数亿甚至数百亿个参数，对其进行微调需要大量的计算资源和存储空间。

问题-2: 在微调过程中，直接修改预训练模型的所有参数可能会破坏模型的原始性能。

问题-3: 存储和部署微调后的大模型需要大量存储空间，尤其是当需要在多个应用场景中部署不同微调版本时。

问题-4: 许多微调方法会增加推理阶段的计算延迟，影响模型的实时性应用。

2. 解决措施

LoRA (Low-Rank Adaptation) 通过引入低秩矩阵分解，在减少计算资源和存储需求的同时，保持了预训练模型的初始性能，稳定了微调过程，并降低了存储和部署成本。它特别适用于大规模模型的微调，在资源有限的环境中具有显著的优势。

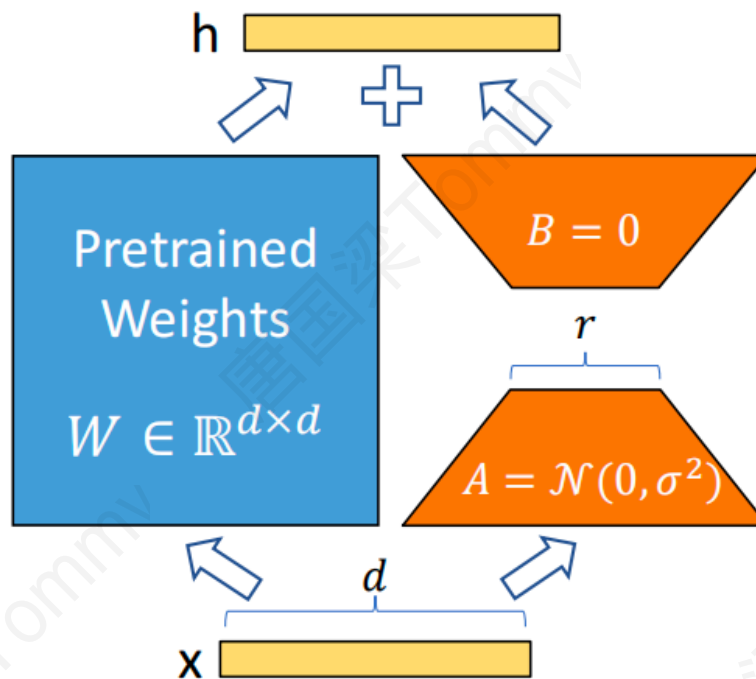


Figure 1: Our reparametrization. We only train A and B .

3. LoRA 优势

- **存储与计算效率:** 通过低秩适应 (LoRA)，可以显著减少所需存储的参数数量，并减少计算需求。

- **适应性与灵活性**：LoRA方法允许模型通过只替换少量特定的矩阵A和B来快速适应新任务，显著提高任务切换的效率。
- **训练与部署效率**：LoRA的简单线性设计允许在不引入推理延迟的情况下，与冻结的权重结合使用，从而提高部署时的操作效率。

GPT-3 175B

Model&Method	# Trainable Parameters	WikiSQL	MNLI-m	SAMSum
		Acc. (%)	Acc. (%)	R1/R2/RL
GPT-3 (FT)	175,255.8M	73.8	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 (Adapter ^H)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 (Adapter ^H)	40.1M	73.2	91.5	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	91.7	53.8/29.8/45.9
GPT-3 (LoRA)	37.7M	74.0	91.6	53.4/29.2/45.1

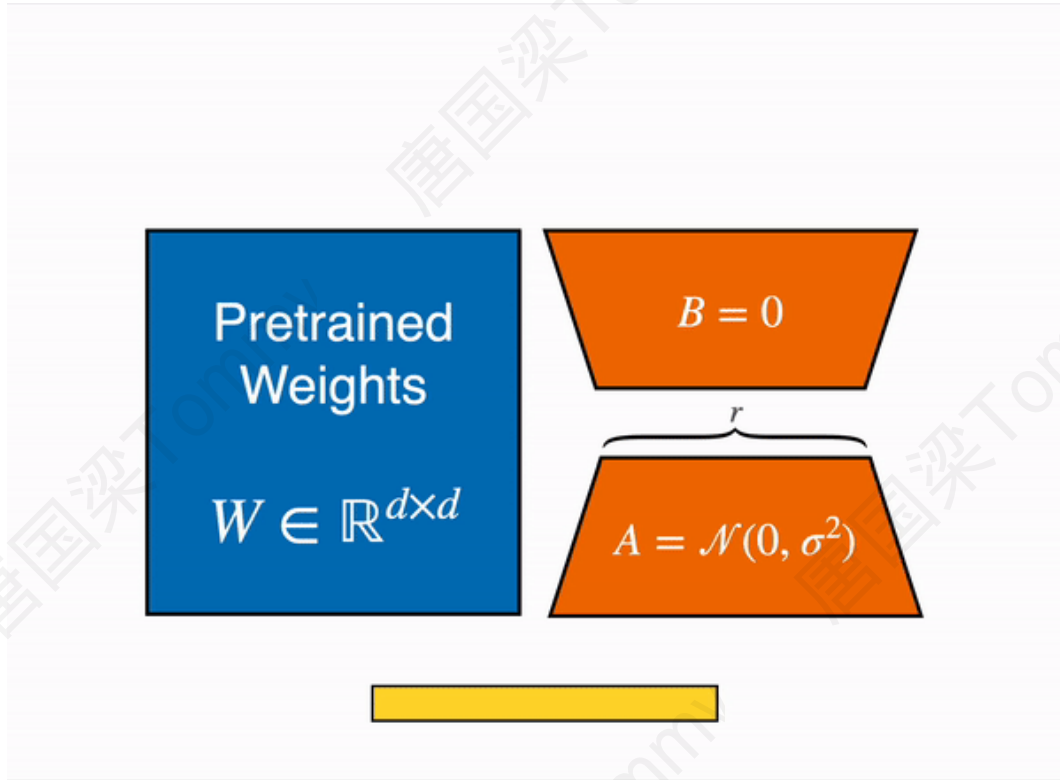
表4：在GPT-3 175B上不同适配方法的表现。我们报告了WikiSQL上的逻辑形式验证准确率、MultiNLI-matched上的验证准确率，以及SAMSum上的Rouge-1/2/L。LoRA的表现超过了之前的方法，包括完整的微调。WikiSQL的结果波动在 $\pm 0.5\%$ ，MNLI-m在 $\pm 0.1\%$ ，而SAMSum的三个指标分别在 $\pm 0.2/\pm 0.2/\pm 0.1$ 。

4. 深入理解 LoRA

4.1 为什么需要低秩分解？

- 现代预训练模型虽然是过参数化的，但在微调时参数更新主要集中在一个低维子空间中。
2. 参数更新 ΔW 可以在低维度中进行优化，高维参数空间中的大部分参数在微调前后几乎没有变化。
 3. 低秩分解使参数优化更高效，但如果参数更新实际上在高维子空间中发生，可能会导致重要信息遗漏和LoRA方法失效。

4.2 LoRA算法原理



LoRA 通过冻结预训练的权重矩阵 $W_{\text{pretrained}} \in \mathbb{R}^{d \times d}$ ，仅学习一个较小的偏置矩阵 ΔW ，公式如下：

$$W_{\text{finetuned}} = W_{\text{pretrained}} + \Delta W$$

$$\Delta W = BA, \quad B \in \mathbb{R}^{d \times r}, \quad A \in \mathbb{R}^{r \times d}$$

A 是一个随机初始化的矩阵，且服从正态分布 $A \sim \mathcal{N}(0, 1)$

B 初始化为零矩阵，即 $B = 0$ 。

作用：这种初始化方法使得在训练初期，新增的部分 $\Delta W = BA$ 对原始权重 $W_{\text{pretrained}}$ 的影响为零，从而不会破坏预训练模型的初始性能。

秩 $r \ll d$ ，只有 $d \times r + r \times d$ 参数需要训练，减少了计算梯度所需的内存和浮点运算量（FLOPS）。

例如，应用 $r = 16$ 的 LoRA 到一个 $d = 4096$ 的 7B 权重矩阵，仅训练不到1%的原始参数量。

假设我们有一个预训练的大模型，其中某个权重矩阵 W 的维度为 $d \times d$ 。假设 $d = 4096$ ，即这个权重矩阵的尺寸为 4096×4096 。

原始权重矩阵 W 的参数数量为：

$$d \times d = 4096 \times 4096 = 16,777,216$$

使用 LoRA 方法

选择一个较小的秩 r ，例如 $r = 16$ 。在 LoRA 中，我们将权重矩阵分解为两个低秩矩阵 A 和 B ，其中：

- A 的维度为 $r \times d$
- B 的维度为 $d \times r$

使用 LoRA 方法后，需要训练的参数数量为：

$$d \times r + r \times d = 4096 \times 16 + 16 \times 4096 = 131,072 + 131,072 = 262,144$$

与原始模型相比，使用 LoRA 后的参数数量显著减少：

$$\frac{262,144}{16,777,216} \approx 0.0156$$

也就是说，只需要训练原始参数数量的约 1.56%。

无需完全微调：LoRA方法允许在不累积对所有权重矩阵的全秩梯度更新的情况下，通过调整LoRA的秩 r 来逼近原始模型的表达能力。

无额外推理延迟：在生产中部署时，可以显式计算并存储 $W_{\text{finetuned}} = W_{\text{pretrained}} + \Delta W$ ，并像往常一样执行推理。这保证了与细调的模型相比，不会引入任何额外的延迟。

知识补充：

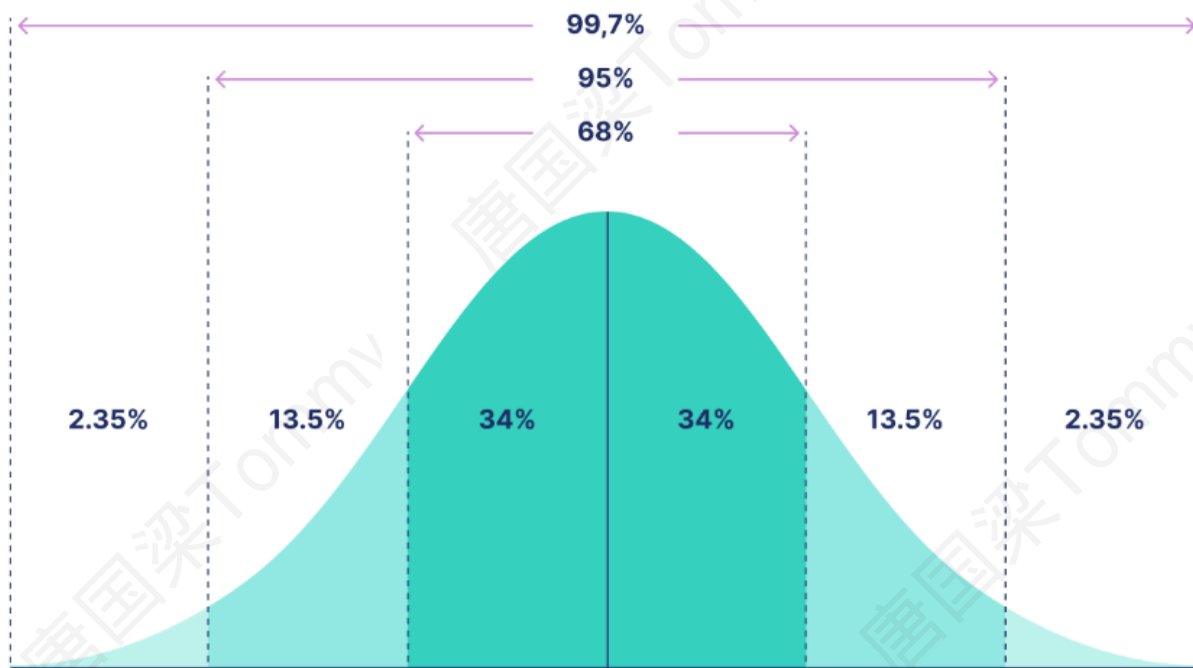
问-1：为什么初始化参数使用正态分布？

答-1：这样做的原因包括：

① **确保初始梯度的有效传播：**正态分布初始化有助于在训练初期确保梯度有效传播，避免梯度消失或爆炸的问题。

② **提供足够的随机性：**正态分布的随机初始化为模型提供了足够的随机性，从而能够探索更广泛的参数空间，增加了模型找到最优解的可能性。

③ **平衡训练初期的影响：**正态分布初始化的值一般较小，结合 B 初始化为零矩阵，可以在训练初期确保新增的偏置矩阵对原始预训练权重的影响为零，从而避免破坏预训练模型的初始性能。



问-2：为什么 A 初始化服从正态分布？而 B 初始化为零矩阵？

答-2：（1）如果 B 和 A 全部初始化为零矩阵，缺点是很容易导致梯度消失；（2）如果 B 和 A 全部正态分布初始化，那么在模型训练开始时，就会容易得到一个过大的偏移值 ΔW ，从而引起太多噪声，导致难以收敛。

>>> 举例讲解 <<<

假设我们有一个预训练的权重矩阵 $W_{\text{pretrained}}$ ，其维度为 $d \times d$ ，我们想应用 LoRA 进行微调。

1. 初始化：

- 我们选择一个较小的秩 r ，例如 $r = 4$ 。
- 初始化矩阵 A 和 B ，其中 $A \in \mathbb{R}^{r \times d}$ ， $B \in \mathbb{R}^{d \times r}$ 。

- A 的元素服从正态分布 $\mathcal{N}(0, 1)$ ，例如：

$$A = \begin{pmatrix} 0.5 & -0.2 & 0.1 & 0.3 & \cdots & 0.7 \\ -0.4 & 0.3 & -0.2 & 0.5 & \cdots & -0.6 \\ 0.6 & 0.4 & -0.3 & -0.1 & \cdots & 0.2 \\ -0.5 & 0.1 & 0.4 & -0.3 & \cdots & 0.5 \end{pmatrix}$$

- B 初始化为零矩阵：

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

2. 训练：

- 在训练过程中，仅更新 A 和 B 的值。
- 随着训练的进行， B 的值逐渐变得非零。

3. 微调权重计算：

- 训练结束后，我们得到 A 和 B 的最终值，计算 $\Delta W = BA$ 。
- 将 ΔW 加到 $W_{\text{pretrained}}$ 上，得到微调后的权重矩阵 $W_{\text{finetuned}}$ 。

4.3 选择哪些权重矩阵进行适配？

在有限参数预算下，应选择哪些权重矩阵进行适配以最大化下游任务性能？

	# of Trainable Parameters = 18M						
Weight Type Rank r	W_q 8	W_k 8	W_v 8	W_o 8	W_q, W_k 4	W_q, W_v 4	W_q, W_k, W_v, W_o 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

表5：在GPT-3中应用LoRA到不同类型的注意力权重后，在WikiSQL和MultiNLI上的验证准确性，考虑到可训练参数的数量相同。同时调整 W_q 和 W_v 能够获得最佳的整体表现。我们发现给定数据集的随机种子间的标准偏差是一致的，我们在第一列中报告了这一点。

主要发现：

- 同时适配 W_q 、 W_k 、 W_v 、 W_o 提供了最佳性能。
- 低秩（例如rank为4）足以在 ΔW 中捕获足够信息，表现优于单一类型权重适配但具有更高秩的策略。

4.4 为什么 LoRA 在 Q, K, V, O 上有效？

LoRA (Low-Rank Adaptation) 在 Transformer 模型的 Q (Query)、K (Key)、V (Value) 和 O (Output) 矩阵上有效的原因可以归结为这些矩阵在注意力机制中的核心作用以及 LoRA 方法在降低参数数量的同时保持或提升模型性能的能力。具体原因如下：

4.4.1 Self-Attention

- **Q (Query) 矩阵**：用于生成查询向量，决定模型在注意力机制中对输入的关注程度。
- **K (Key) 矩阵**：用于生成键向量，与查询向量计算相似度，帮助确定注意力分布。
- **V (Value) 矩阵**：用于生成数值向量，实际传递注意力机制计算的输出。
- **O (Output) 矩阵**：用于将多头注意力的输出合并并映射回原始维度。

4.4.2 信息传播的关键路径

Q 、 K 、 V 和 O 矩阵在信息传播和特征表示中起着关键作用：

- **查询与键的交互**： Q 和 K 的交互决定了注意力分布，影响模型对输入序列的不同部分的关注度。
- **数值的加权求和**： V 矩阵通过加权求和操作，将注意力分布转化为具体的输出。
- **多头输出的整合**： O 矩阵整合多头注意力的输出，提供最终的特征表示。

4.4.3 LoRA 的低秩近似

LoRA 通过将权重矩阵分解为两个低秩矩阵（例如 $W \approx BA$ ），减少了参数数量，降低了计算和存储成本，同时保持模型性能：

- **参数压缩**： Q 、 K 、 V 和 O 矩阵通常包含大量参数，LoRA 的低秩分解显著减少了需要优化的参数数量。
- **性能保持**：低秩矩阵能够捕捉到原始矩阵的主要信息，确保模型性能不受显著影响。

4.5 LoRA中的最优秩 r 的选择

作者探讨了不同秩 r 对模型性能的影响，并确定最小的有效秩，即“内在秩”。

	# 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

表5：在GPT-3中应用LoRA到不同类型的注意力权重后，在WikiSQL和MultiNLI上的验证准确性，考虑到可训练参数的数量相同。同时调整 W_q 和 W_v 能够获得最佳的整体表现。我们发现给定数据集的随机种子间的标准偏差是一致的，我们在第一列中报告了这一点。

	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL($\pm 0.5\%$)	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 ($\pm 0.1\%$)	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

表6：在WikiSQL和MultiNLI上使用不同等级 r 的验证准确性。令我们惊讶的是，等级为一就足以适应这些数据集上的 W_q 和 W_v ，而单独训练 W_q 则需要更高的 r 。我们在第H.2节中对GPT-2进行了类似的实验。

主要发现：

- LoRA 即使在非常小的秩 r 下也展现出了竞争力的性能。
- 在同时适配 W_q 和 W_v 时比仅适配 W_q 表现更好。
- 增加 r 并没有覆盖更多有意义的子空间，说明低秩适配矩阵已足够。

4.6 LoRA可以应用到模型中的哪些层？

LoRA (Low-Rank Adaptation) 可以插入到模型的多个地方，具体取决于需要微调的模型部分和任务要求。以下是一些常见的插入位置及其原因：

4.6.1 线性层（全连接层）

位置：

在神经网络的全连接层（Linear Layer）中，通常会使用线性变换 $Wx + b$ 。

原因：

- **主要参数集中**：全连接层通常包含大量参数，通过在这些层中应用LoRA，可以显著减少需要微调的参数数量。
- **计算密集型**：全连接层的计算量较大，通过低秩近似可以有效降低计算复杂度。

4.6.2 注意力层

位置：

在Transformer模型的多头自注意力（Multi-head Self-Attention）机制中，包括查询（Query）、键（Key）和值（Value）矩阵的线性投影部分。

原因：

- **关键功能组件**：自注意力机制是Transformer模型的核心组件，对模型性能影响重大。对这些矩阵进行低秩近似可以显著影响模型的表达能力。
- **参数量大**：这些投影矩阵包含大量参数，使用LoRA可以减少参数数量，降低计算和存储需求。

4.6.3 嵌入层

位置：

在NLP任务中，嵌入层用于将离散的词汇表映射到连续的向量空间。

原因：

- **高维稀疏表示**：嵌入层通常包含大量高维向量，通过LoRA可以有效降低维度，减少计算量和内存占用。
- **提升训练效率**：低秩分解可以使嵌入层的训练更加高效。

5. 案例实战

本机实验环境

1. ubuntu20.04
2. Python 3.10.14
3. pytorch 1.13.0
4. CUDA

Cuda compilation tools, release 11.8, V11.8.89

Build cuda_11.8.r11.8/compiler.31833905_0

01_LoRA_案例实战.ipynb

02_LoRA_微软开源项目

第1步：环境配置

```
conda create -n lora python=3.10
# conda init bash && source /root/.bashrc
conda activate lora
# conda install ipykernel
# ipython kernel install --user --name=lora # 设置kernel, --user表示当前用户, lora为虚拟环境名称
```

```
pip install torch==1.13.0
pip install progress
pip install transformers==4.39.3 deepspeed accelerate datasets==2.18.0 peft bitsandbytes

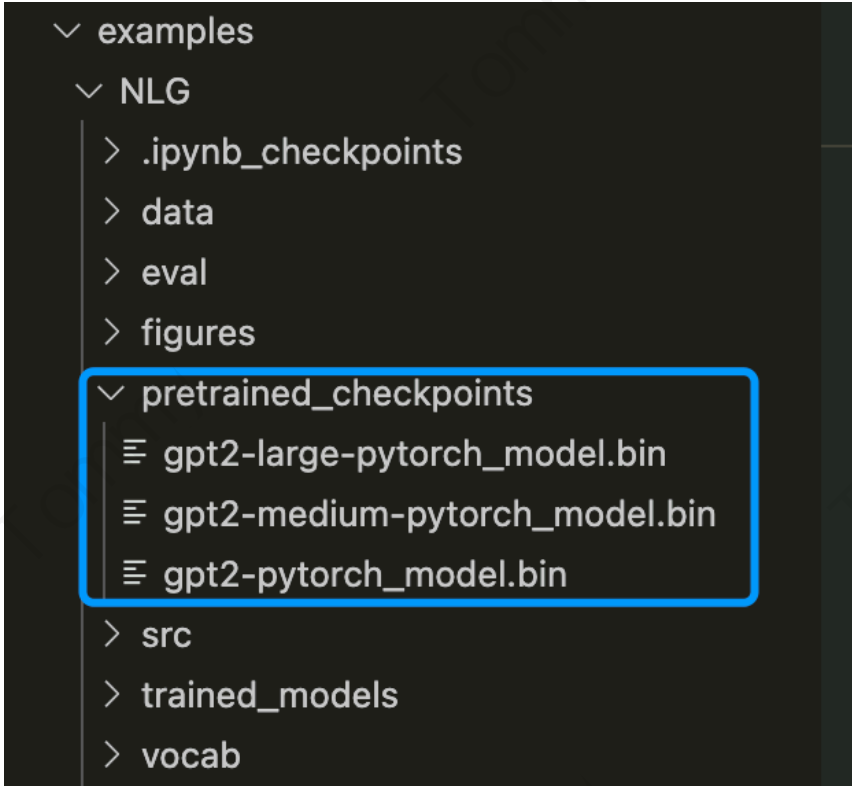
# 安装loralib
# 方式-1
pip install loralib

## 方式-2 (开发者模式)
pip install -e .
```

第2步：下载预训练模型

```
# 在路径：/LoRA/examples/NLG/ 下运行命令

bash download_pretrained_checkpoints.sh
```



```
examples
├── NLG
│   ├── .ipynb_checkpoints
│   ├── data
│   ├── eval
│   ├── figures
│   └── pretrained_checkpoints
│       ├── gpt2-large-pytorch_model.bin
│       ├── gpt2-medium-pytorch_model.bin
│       └── gpt2-pytorch_model.bin
│   ├── src
│   ├── trained_models
│   └── vocab
```

第3步：下载训练数据集

```
# 在路径：/LoRA/examples/NLG/ 下运行命令

bash create_datasets.sh
```

第4步：在数据集E2E上训练

```
python -m torch.distributed.launch --nproc_per_node=1 src/gpt2_ft.py \  
    --train_data ./data/e2e/train.jsonl \  
    --valid_data ./data/e2e/valid.jsonl \  
    --train_batch_size 8 \  
    --grad_acc 1 \  
    --valid_batch_size 4 \  
    --seq_len 512 \  
    --model_card gpt2.md \  
    --init_checkpoint ./pretrained_checkpoints/gpt2-medium-pytorch_model.bin \  
    --platform local \  
    --clip 0.0 \  
    --lr 0.0002 \  
    --weight_decay 0.01 \  
    --correct_bias \  
    --adam_beta2 0.999 \  
    --scheduler linear \  
    --warmup_step 500 \  
    --max_epoch 5 \  
    --save_interval 1000 \  
    --lora_dim 4 \  
    --lora_alpha 32 \  
    --lora_dropout 0.1 \  
    --label_smooth 0.1 \  
    --work_dir ./trained_models/GPT2_M/e2e \  
    --random_seed 110
```

参数解析：

python -m torch.distributed.launch：使用PyTorch的分布式启动模块来启动脚本。

--nproc_per_node=1：每个节点上的进程数为1，即不进行实际的分布式训练，只在单个进程上运行。

--grad_acc：梯度累积步数

--platform：指定平台

--clip：梯度裁剪阈值

--correct_bias：是否修正偏差

--adam_beta2：Adam优化器的beta2参数

--label_smooth：标签平滑系数

```

p.data.addcddiv_(-step_size, exp_avg, denom)
| epoch 1 step 100 | 100 batches | lr 4e-05 | ms/batch 443.58 | loss 5.18 | avg loss 5.56 | ppl 259.01
| epoch 1 step 200 | 200 batches | lr 8e-05 | ms/batch 417.75 | loss 3.21 | avg loss 3.75 | ppl 42.48
| epoch 1 step 300 | 300 batches | lr 0.00012 | ms/batch 418.06 | loss 2.97 | avg loss 3.08 | ppl 21.72
| epoch 1 step 400 | 400 batches | lr 0.00016 | ms/batch 418.43 | loss 3.11 | avg loss 2.98 | ppl 19.59
| epoch 1 step 500 | 500 batches | lr 0.0002 | ms/batch 418.79 | loss 2.84 | avg loss 2.89 | ppl 17.98
| epoch 1 step 600 | 600 batches | lr 0.000199 | ms/batch 418.96 | loss 2.77 | avg loss 2.83 | ppl 16.89
| epoch 1 step 700 | 700 batches | lr 0.000198 | ms/batch 418.99 | loss 2.88 | avg loss 2.79 | ppl 16.29
| epoch 1 step 800 | 800 batches | lr 0.000198 | ms/batch 418.96 | loss 2.47 | avg loss 2.75 | ppl 15.72
| epoch 1 step 900 | 900 batches | lr 0.000197 | ms/batch 419.01 | loss 2.50 | avg loss 2.74 | ppl 15.51
| epoch 1 step 1000 | 1000 batches | lr 0.000196 | ms/batch 418.76 | loss 3.18 | avg loss 2.76 | ppl 15.87
saving checkpoint ./trained_models/GPT2_M/e2e/model.1000.pt
| epoch 1 step 1100 | 1100 batches | lr 0.000195 | ms/batch 419.03 | loss 2.84 | avg loss 2.76 | ppl 15.78
| epoch 1 step 1200 | 1200 batches | lr 0.000195 | ms/batch 419.08 | loss 2.58 | avg loss 2.75 | ppl 15.67
| epoch 1 step 1300 | 1300 batches | lr 0.000194 | ms/batch 418.97 | loss 2.62 | avg loss 2.72 | ppl 15.15
| epoch 1 step 1400 | 1400 batches | lr 0.000193 | ms/batch 418.97 | loss 2.70 | avg loss 2.72 | ppl 15.16
| epoch 1 step 1500 | 1500 batches | lr 0.000192 | ms/batch 419.26 | loss 2.66 | avg loss 2.73 | ppl 15.28
| epoch 1 step 1600 | 1600 batches | lr 0.000191 | ms/batch 419.39 | loss 2.69 | avg loss 2.68 | ppl 14.59
| epoch 1 step 1700 | 1700 batches | lr 0.000191 | ms/batch 419.29 | loss 2.57 | avg loss 2.69 | ppl 14.79
| epoch 1 step 1800 | 1800 batches | lr 0.00019 | ms/batch 420.06 | loss 2.54 | avg loss 2.69 | ppl 14.67
| epoch 1 step 1900 | 1900 batches | lr 0.000189 | ms/batch 419.09 | loss 2.69 | avg loss 2.68 | ppl 14.65
| epoch 1 step 2000 | 2000 batches | lr 0.000188 | ms/batch 419.38 | loss 2.51 | avg loss 2.67 | ppl 14.44
saving checkpoint ./trained_models/GPT2_M/e2e/model.2000.pt

```

跑完第1个epoch后的输出模型

📁 / ... / GPT2_M / e2e /

名称

- 📄 model.1000.pt
- 📄 model.2000.pt
- 📄 model.3000.pt
- 📄 model.4000.pt
- 📄 model.5000.pt
- 📄 model.5258.pt

第5步：基于上一步训练之后的输出模型进行推理

```

python -m torch.distributed.launch --nproc_per_node=1 src/gpt2_beam.py \
    --data ./data/e2e/test_200.jsonl \
    --batch_size 8 \
    --seq_len 512 \
    --eval_len 64 \
    --model_card gpt2.md \
    --init_checkpoint ./trained_models/GPT2_M/e2e/model.5258.pt \
    --platform local \
    --lora_dim 4 \
    --lora_alpha 32 \

```

```
--beam 10 \
--length_penalty 0.8 \
--no_repeat_ngram_size 4 \
--repetition_penalty 1.0 \
--eos_token_id 628 \
--work_dir ./trained_models/GPT2_M/e2e \
--output_file predict.5258.s200.jsonl
```

参数解析：

--beam：束搜索的束宽，指在搜索过程中保留的候选序列数量。更大的束宽通常会导致更好的搜索结果，但也会增加计算成本。

--length_penalty：长度惩罚系数，避免生成过短或过长的序列。较低的值会惩罚较长的序列，较高的值会鼓励生成更长的序列。

示例：

--length_penalty 0.8：适度惩罚较长的序列，生成结果不会过长。

--length_penalty 1.0：不进行长度惩罚，生成结果的长度将仅由概率决定。

--length_penalty 1.2：鼓励生成更长的序列，减少生成短序列的可能性。

--no_repeat_ngram_size：禁止重复的n-gram大小，用于避免生成结果中出现重复的n-gram。n-gram是指连续的n个词，设置这个参数可以提高生成文本的多样性。

示例：

--no_repeat_ngram_size 2：禁止生成结果中出现重复的二元组 (bigram)，例如“the cat the cat”。

--no_repeat_ngram_size 3：禁止生成结果中出现重复的三元组 (trigram)，例如“the cat is the cat is”。

--repetition_penalty：重复惩罚系数，用于降低生成过程中重复词或短语的概率。较高的值会强烈惩罚重复，鼓励生成更多样化的文本。

示例：

--repetition_penalty 1.0：不进行重复惩罚。

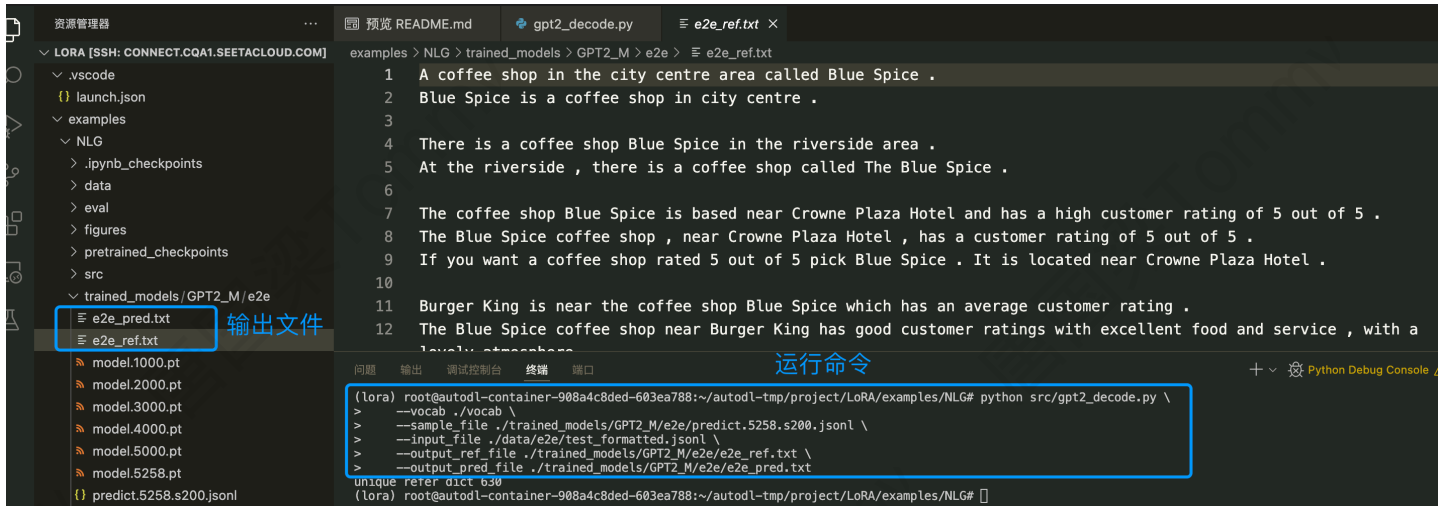
--repetition_penalty 1.2：轻微惩罚重复的词或短语，增加生成文本的多样性。

--repetition_penalty 1.5：强烈惩罚重复的词或短语，显著增加生成文本的多样性。

```
=====
Experiment dir : ./trained_models/GPT2_M/e2e
loading model pretrained weight.
model sampling ...
inference samples 0
inference samples 10
inference samples 20
saving prediction file ./trained_models/GPT2_M/e2e/predict.5258.s200.jsonl
cleanup dist ...
```

第6步：基于第5步的输出进行解码

```
python src/gpt2_decode.py \
  --vocab ./vocab \
  --sample_file ./trained_models/GPT2_M/e2e/predict.5258.s200.jsonl \
  --input_file ./data/e2e/test_formatted.jsonl \
  --output_ref_file ./trained_models/GPT2_M/e2e/e2e_ref.txt \
  --output_pred_file ./trained_models/GPT2_M/e2e/e2e_pred.txt
```



第7步：在E2E测试集上进行评估

```
python eval/e2e/measure_scores.py ./trained_models/GPT2_M/e2e/e2e_ref.txt
./trained_models/GPT2_M/e2e/e2e_pred.txt -p
```

踩坑记录

运行上面的命令后，抛出Error。如下图：

```
Traceback (most recent call last):
  File "/root/autodl-tmp/project/LoRA/examples/NLG/eval/e2e/measure_scores.py", line 380, in <module>
    evaluate(data_src, data_ref, data_sys, args.table, args.header, args.sys_file, args.python)
  File "/root/autodl-tmp/project/LoRA/examples/NLG/eval/e2e/measure_scores.py", line 234, in evaluate
    coco_eval = run_coco_eval(data_ref, data_sys)
  File "/root/autodl-tmp/project/LoRA/examples/NLG/eval/e2e/measure_scores.py", line 323, in run_coco_eval
    coco_eval.evaluate()
  File "/root/autodl-tmp/project/LoRA/examples/NLG/eval/e2e/pycocoevalcap/eval.py", line 36, in evaluate
    gts = tokenizer.tokenize(gts)
  File "/root/autodl-tmp/project/LoRA/examples/NLG/eval/e2e/pycocoevalcap/tokenizer/ptbtokenizer.py", line 54, in tokenize
    p_tokenizer = subprocess.Popen(cmd, cwd=path_to_jar_dirname, \
  File "/root/miniconda3/envs/lora/lib/python3.10/subprocess.py", line 971, in __init__
    self._execute_child(args, executable, preexec_fn, close_fds,
  File "/root/miniconda3/envs/lora/lib/python3.10/subprocess.py", line 1863, in _execute_child
    raise child_exception_type(errno_num, err_msg, err_filename)
FileNotFoundError: [Errno 2] No such file or directory: 'java'
```

解决措施：

在系统上安装JAVA,

```
sudo apt update
sudo apt install default-jre
```

解决Error后，再次运行命令，输出如下：

```
(lora) root@autodl-container-908a4c8ded-603ea788:~/autodl-tmp/project/LoRA/examples/NLG#
(lora) root@autodl-container-908a4c8ded-603ea788:~/autodl-tmp/project/LoRA/examples/NLG# python eval/e2
e/measure_scores.py ./trained_models/GPT2_M/e2e/e2e_ref.txt ./trained_models/GPT2_M/e2e/e2e_pred.txt -p
Running MS-COCO evaluator...
creating index...
index created!
Loading and preparing results...
DONE (t=0.00s)
creating index...
index created!
tokenization...
PTBTokenizer tokenized 132573 tokens at 506481.96 tokens per second.
PTBTokenizer tokenized 1141 tokens at 13036.07 tokens per second.
setting up scorers...
computing METEOR score...
METEOR: 0.017
computing Rouge score...
ROUGE_L: 0.031
computing CIDEr score...
CIDEr: 0.133
Running Py-MTEval metrics...
SCORES:
=====
BLEU: 5.4198
NIST: 0.0000
METEOR: 0.0165
ROUGE_L: 0.0312
CIDEr: 0.1326
```

指标含义：

1. BLEU

含义：

BLEU(Bilingual Evaluation Understudy) 是一种用于评估机器翻译和自然语言生成模型的精确度指标。它通过计算生成的文本与参考文本之间的 n-gram 匹配程度来评估生成文本的质量。

计算方法：

- 计算生成文本和参考文本中 n-gram 的精确匹配数。
- 计算不同 n-gram（通常是 1-gram 到 4-gram）的加权几何平均值。
- 结合长度惩罚项，惩罚生成的文本长度与参考文本长度不一致的情况。

公式：

$$\text{BLEU} = \text{BP} \cdot \exp \left(\sum_{n=1}^N w_n \log p_n \right)$$

其中：

- BP 是长度惩罚（Brevity Penalty）。
- w_n 是 n-gram 的权重（通常等权重）。

- p_n 是生成文本和参考文本之间 n-gram 的精确匹配比例。

2. NIST

含义：

NIST(NIST Machine Translation Evaluation) 是 BLEU 的变种，它不仅考虑 n-gram 的精确匹配，还考虑匹配 n-gram 的信息量（即罕见 n-gram 的匹配会得到更高的得分）。

计算方法：

- 计算生成文本和参考文本中 n-gram 的精确匹配数。
- 计算每个匹配 n-gram 的信息量。
- 结合长度惩罚项。

公式：

与 BLEU 类似，但对 n-gram 匹配的权重进行了加权处理，强调罕见 n-gram 的匹配。

3. METEOR

含义：

METEOR 通过考虑词形变化、同义词匹配和词序来评估生成文本的质量。

计算方法：

- 计算生成文本和参考文本之间的 unigram 精确匹配。
- 计算词形变化、同义词匹配的匹配数。
- 结合词序惩罚项。

公式：

$$\text{METEOR} = F_{\text{mean}} \cdot (1 - \text{Pen})$$

其中：

- F_{mean} 是精确度和召回率的调和平均。
- Pen 是词序惩罚项。

4. ROUGE_L

含义：

ROUGE_L 基于最长公共子序列（LCS）来评估生成文本的覆盖度，强调生成文本中有多少部分与参考文本的顺序匹配。

计算方法：

- 计算生成文本和参考文本之间的最长公共子序列。

- 结合精确度和召回率。

公式：

$$\text{ROUGE}_L = \frac{(1+\beta^2) \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \beta^2 \cdot \text{Recall}}$$

其中：

- 精确度 (Precision) 和召回率 (Recall) 基于 LCS 计算。
- β 是精确度和召回率的权重系数。

5. CIDEr

含义：

CIDEr (Consensus-based Image Description Evaluation) 主要用于图像描述生成任务，通过评估生成文本与参考文本的共识度来评估生成文本的质量。

计算方法：

- 计算生成文本和参考文本之间的 TF-IDF 加权 n-gram 相似度。
- 结合多个参考文本的共识度。

公式：

$$\text{CIDEr} = \frac{1}{m} \sum_{i=1}^m \frac{\sum_{n=1}^N \text{CIDEr}_n(i)}{N}$$

其中：

- m 是参考文本数量。
- $\text{CIDEr}_n(i)$ 是第 i 个参考文本的第 n -gram 相似度。