OH报告

——终端大模型推理性能优化

李乐琪　龙浩泉

技术预研

OpenHarmony AI子系统包含四个部件:

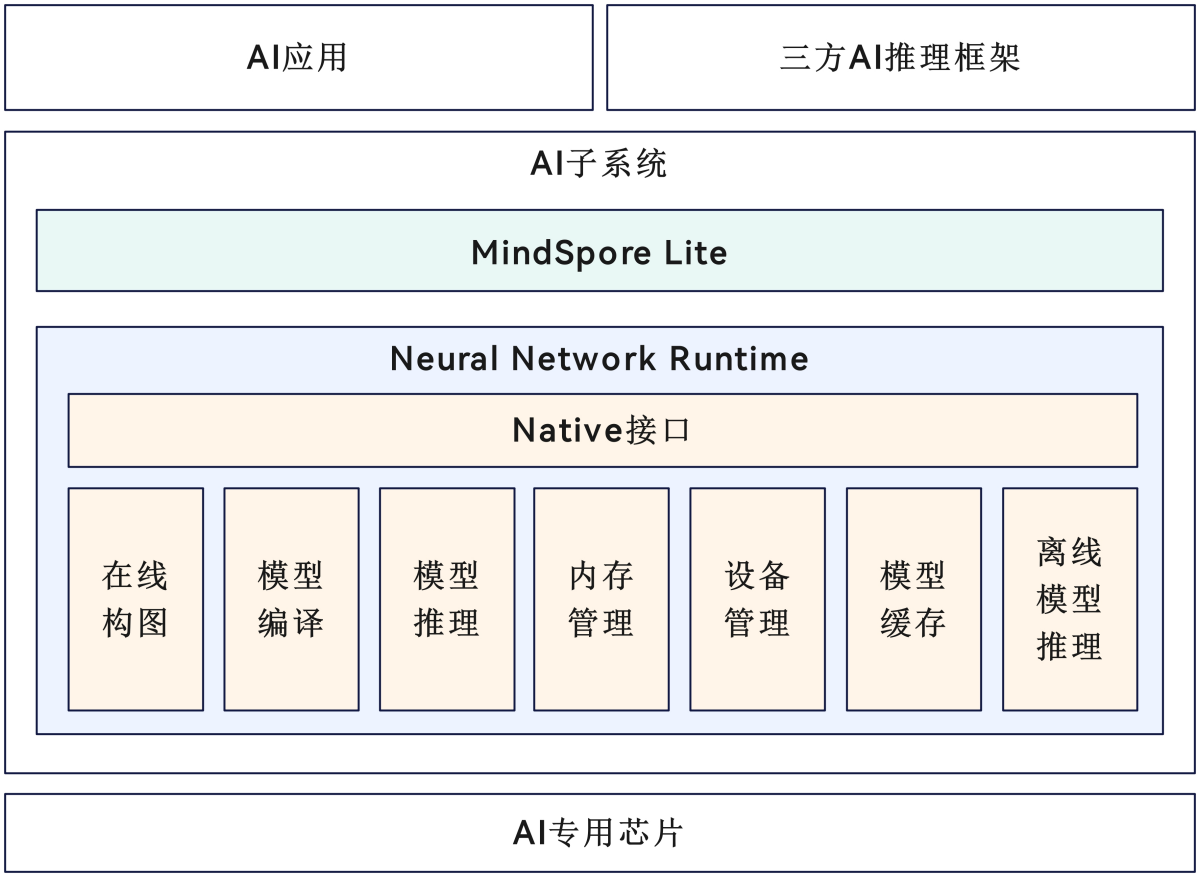
- `ai_engine`：提供统一的AI引擎框架，实现算法能力快速插件化集成
- `intelligent_voice_framework`：智能语音组件，包括智能语音服务框架和智能语音驱动，主要实现语音注
  册及语音唤醒相关功能
- `mindspore`：适用于端边云场景的新型开源深度学习训练/推理框架
- `neural_network_runtime`：NNRt, 神经网络运行时，是面向AI领域的跨芯片推理计算运行时，作为中间桥梁连通上层AI推理框架和底层加速芯片，实现AI模型的跨芯片推理计算

NNRt

> 参考链接：https://gitee.com/openharmony/ai_neural_network_runtime

框架

Neural Network Runtime架构图



Neural Network Runtime（NNRt, 神经网络运行时）是面向AI领域的跨芯片推理计算运行时，作为中间桥梁连通上层AI推理框架和底层加速芯片，实现AI模型的跨芯片推理计算。

如图所示，NNRt开放北向Native接口供AI推理框架接入，当前NNRt对接了系统内置的MindSpore Lite推理框架。同时NNRt开放南向HDI接口，供端侧AI加速芯片（如NPU、DSP等）接入OpenHarmony硬件生态。AI应用通过AI推理框架和NNRt能直接使用底层芯片加速推理计算。

功能

Neural Network Runtime与MindSpore Lite使用MindIR统一模型的中间表达，减少中间过程不必要的模型转换，使得模型传递更加高效。

通常，AI应用、AI推理引擎、Neural Network Runtime处在同一个进程下，芯片驱动运行在另一个进程下，两者之间需要借助进程间通信（IPC）传递模型和计算数据。Neural Network Runtime根据HDI接口实现了HDI客户端，相应的，芯片厂商需要根据HDI接口实现并开放HDI服务。

使用介绍

- **编译构建**

在OpenHarmony源码根目录下，调用以下指令，单独编译Neural Network Runtime。

```
./build.sh --product-name rk3568 --ccache --build-target
neural_network_runtime --jobs 4
```

> **说明：**
> --product-name：产品名称，例如Hi3516DV300、rk3568等。
> --ccache：编译时使用缓存功能。
> --build-target: 编译的部件名称。
> --jobs：编译的线程数，可加速编译。

- **对接AI推理框架**

Neural Network Runtime作为AI推理引擎和加速芯片的桥梁，为AI推理引擎提供精简的Native接口，满足推理引擎通过加速芯片执行端到端推理的需求。

由于Neural Network Runtime通过OpenHarmony Native API对外开放，需要通过OpenHarmony的Native开发套件编译Neural Network Runtime应用。在社区的每日构建中下载对应系统版本的ohos-sdk压缩包，从压缩包中提取对应平台的Native开发套件，解压后目录为 `~/oslab/ohos-sdk/linux/native` 。

> Native开发套件即 MindSpore 使用的交叉编译工具链。

- **开发步骤**

Neural Network Runtime的开发流程主要包含**模型构造**、**模型编译**和**推理执行**三个阶段。

1. 创建应用样例文件。

   首先，创建Neural Network Runtime应用样例的源文件。在项目目录下执行以下命令，创建 `nnrt_example/` 目录，并在目录下创建 `nnrt_example.cpp` 源文件。

```
mkdir ~/nnrt_example && cd ~/nnrt_example
touch nnrt_example.cpp
```

2. 导入Neural Network Runtime。

   在 `nnrt_example.cpp` 文件的开头添加以下代码，引入Neural Network Runtime。

```
#include <iostream>
#include <cstdarg>
#include "hilog/log.h"
#include "neural_network_runtime/neural_network_runtime.h"
```

3. 定义日志打印、设置输入数据、数据打印等辅助函数。
4. 构造模型：使用Neural Network Runtime的模型构造接口，构造目标模型。
5. 查询Neural Network Runtime已经对接的AI加速芯片。
6. 在指定的设备上编译模型。
7. 创建执行器。
8. 执行推理计算，并打印推理结果。
9. 构建端到端模型构造-编译-执行流程。

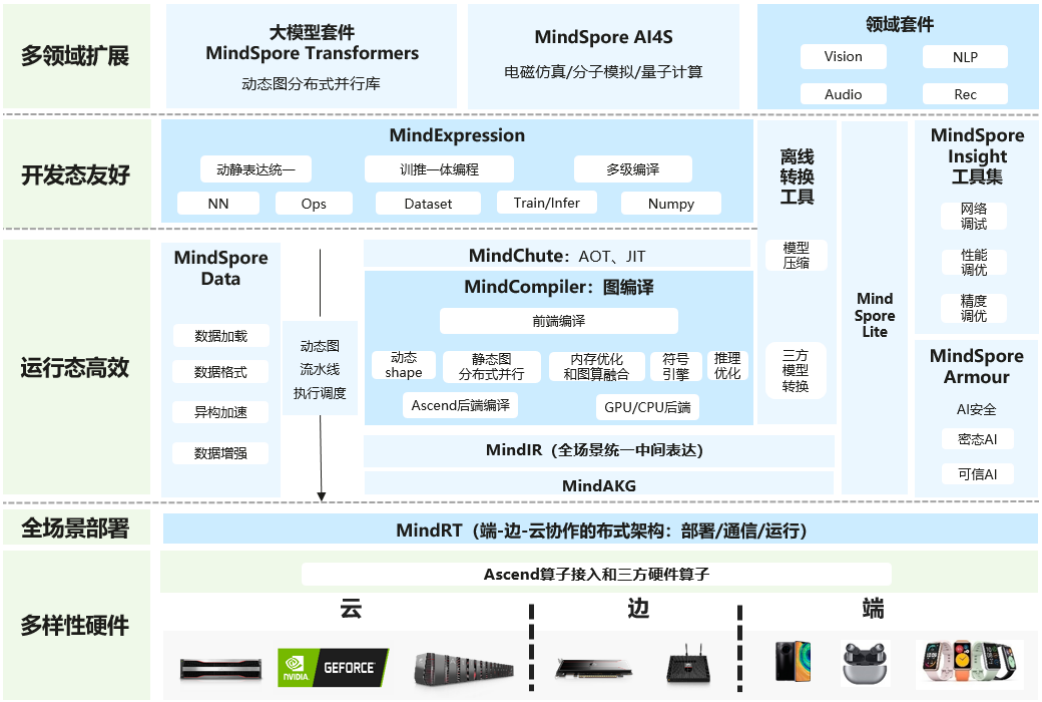
> 参考文档：`ai_neural_network_runtime-master/neural-network-runtime-guidelines.md`

MindSpore

> 参考链接：https://gitee.com/openharmony/ai_neural_network_runtime

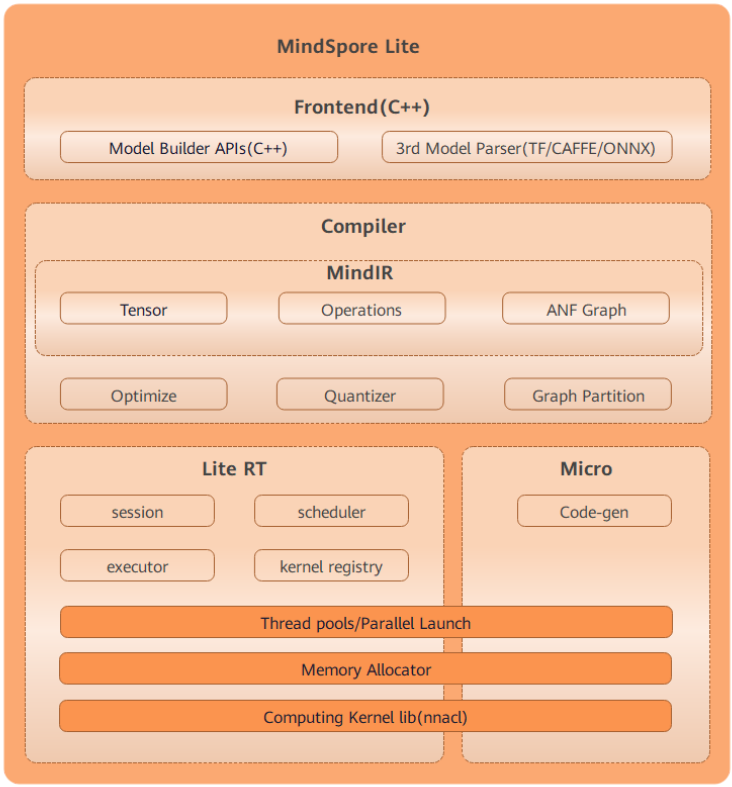
框架

昇思MindSpore是一个全场景**深度学习框架**，旨在实现易开发、高效执行、全场景统一部署三大目标。其中，易开发表现为API友好、调试难度低；高效执行包括计算效率、数据预处理效率和分布式训练效率；全场景则指框架同时支持云、边缘以及端侧场景。昇思MindSpore总体架构如下图所示：



OpenHarmony的标准系统内置了Mindspore Lite，和MindIR构图接口。

MindSpore Lite是MindSpore推出的端云协同的、轻量化、高性能AI推理框架，用于满足越来越多的端测AI应用需求。MindSpore Lite聚焦AI技术在端侧设备上的部署和运行，已经在华为HMS和智能终端的图像分类、目标识别、人脸识别、文字识别等应用中广泛使用，未来MindSpore Lite将与MindSpore AI社区一起，致力于丰富AI软硬件应用生态。



功能

1. 高效训练与推理

- 自动微分：MindSpore提供了自动微分功能，能够自动计算模型的梯度，方便实现反向传播和优化算法。
- 分布式训练：支持在多个设备和机器上进行模型的并行训练，通过原创的多副本、多流水交织等8种并行技术，实现更快的分布式训练，提高训练速度和性能。
- 动态计算图：支持动态计算图，可以根据不同的输入数据调整模型的计算图，提高计算效率和灵活性。

2. 丰富的数据处理能力

- 数据处理：MindSpore提供了丰富的数据处理功能，包括数据增强、数据管道等，方便用户进行数据预处理和增广操作。
- 模型量化：支持模型量化，通过压缩模型的参数和计算过程，减小模型的存储和计算开销，提高模型的运行效率。

3. 灵活的模型构建与部署

- 模型层：用户可以使用Python等编程语言定义和构建模型，同时可以利用MindSpore提供的高级API简化模型构建过程。
- 跨平台部署：支持将训练好的模型部署到多种设备上，包括手机、边缘设备和云端服务器等，实现端到端的AI部署。
- 模型部署工具：提供MindSpore Serving等工具，简化模型部署流程。

使用介绍

- **编译MindSpore Lite**

以rk3568为例，单独编译mindspore lite动态库：

```
./build.sh --product-name rk3568 --target-cpu arm -T mindspore_lib --ccache
```

编译出的so文件位于`out/rk3568/thirdparty/mindspore/`目录。

```
libmindir.z.so  libmindspore-lite.huawei.so
```

- **通过交叉编译在开发板使用Mindspore**

在社区的每日构建中下载对应系统版本的ohos-sdk压缩包，从压缩包中提取对应平台的Native开发套件，解压ohos.toolchain.cmake的位置为`ohos-sdk/linux/native/build/cmake/ohos.toolchain.cmake`。

通过脚本build.sh即可下编译实例代码并下载ms模型文件。
build.sh文件接收两个参数：

- 第二个参数是架构，目前支持arm32和arm64。
- 第一个参数是ohos.toolchain.cmake的路径。

使用方法如下：

```
bash build.sh 'PATH TO ohos.toolchain.cmake' arm32
```

> OHOS全量系统为arm32架构。

脚本运行成功之后，编译得到的`demo`可执行程序会存放在`build`目录下。`demo`接收一个参数，该参数为`.ms`模型文件的路径。

使用`hdc send file`将编译好二进制文件和模型文件都传到开发板的 /data 文件夹下。通过 hdc shell 连接开发板，并使用以下命令进行使用：

```
cd /data
chmod 755 ./demo
./demo ./mobilenetv2.ms
```

若看到类似如下输出，则证明模型运行成功：

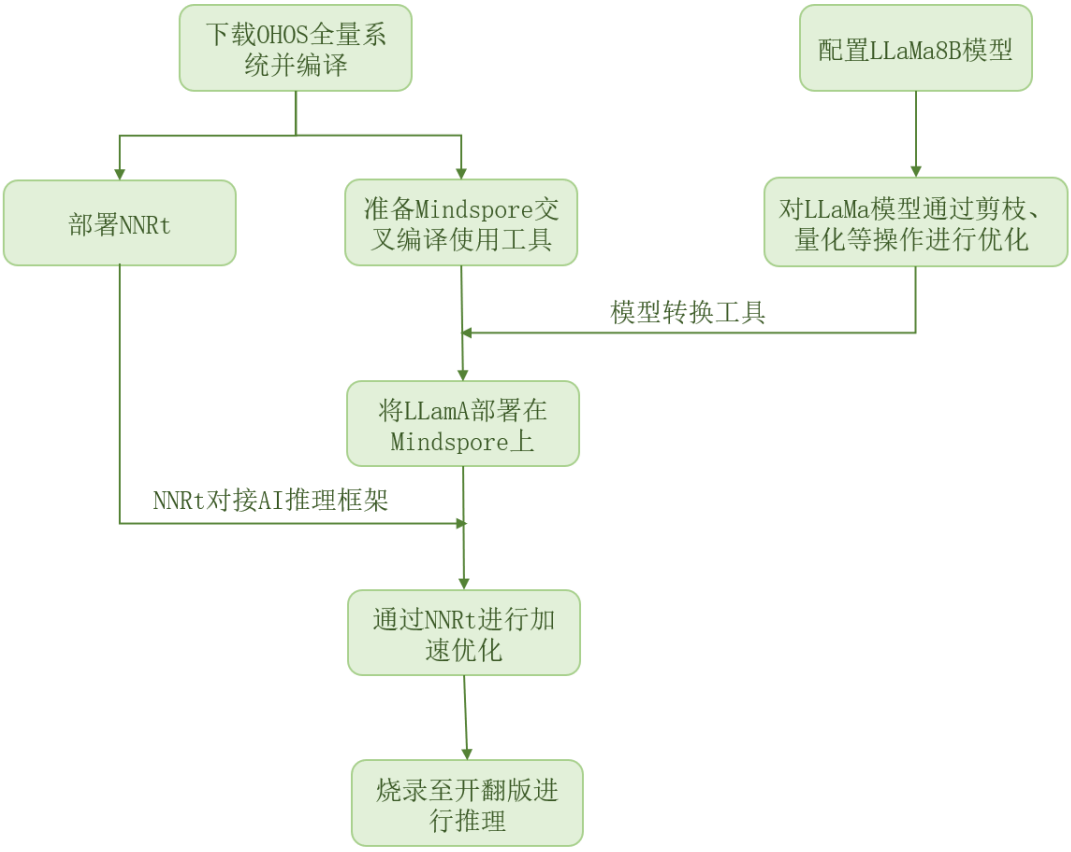
```
Tensor name: Softmax-65, tensor size is 4004 ,elements num: 1001.
output data is:
0.000018 0.000012 0.000026 0.000194 0.000156 0.001501 0.000240 0.000825
0.000016 0.000006 0.000007 0.000004 0.000004 0.000004 0.000015 0.000099
0.000011 0.000013 0.000005 0.000023 0.000004 0.000008 0.000003 0.000003
0.000008 0.000014 0.000012 0.000006 0.000019 0.000006 0.000018 0.000024
0.000010 0.000002 0.000028 0.000372 0.000010 0.000017 0.000008 0.000004
0.000007 0.000010 0.000007 0.000012 0.000005 0.000015 0.000007 0.000040
0.000004 0.000085 0.000023
```

> 参考文档：`third_party_mindspore-OpenHarmony-3.2-Release/mindspore/lite/examples/quick_start_c/Readme.md`

技术路线

> **实现需求**
>
> 将主流的开源LLM（即：LLaMA）通过模型压缩、内存优化等技术成功部署在OpenHarmony开发板上，并对终端推理性能进行优化（主要指标有推理时延、吞吐量以及准确率）；最后使用指定的Benchmark测试**准确率**，并计算吞吐量（平均每秒完成的请求数）和**平均时延**（每个请求的平均完成时间）。

部署方案



性能优化方案

1 Page attention

1.1 KV缓存

LLM 在生成过程中与内存限制作斗争。在生成的解码部分，为先前的令牌生成的所有注意力键和值都存储在 GPU 内存中以供重复使用。这称为 **KV 缓存**，对于大型模型和长序列，它可能会占用大量内存。

PagedAttention 尝试通过将 KV 缓存划分为可通过查找表访问的块来优化内存使用。因此，KV 缓存不需要存储在连续内存中，并且根据需要分配块。内存效率可以提高内存受限工作负载上的 GPU 利用率，因此可以支持更多推理批处理。

使用查找表来访问内存块也有助于跨多代共享 KV。这对于并行采样等技术很有帮助，因为在*并行采样*中，会为同一提示同时生成多个输出。在这种情况下，缓存的 KV 块可以在各代之间共享。

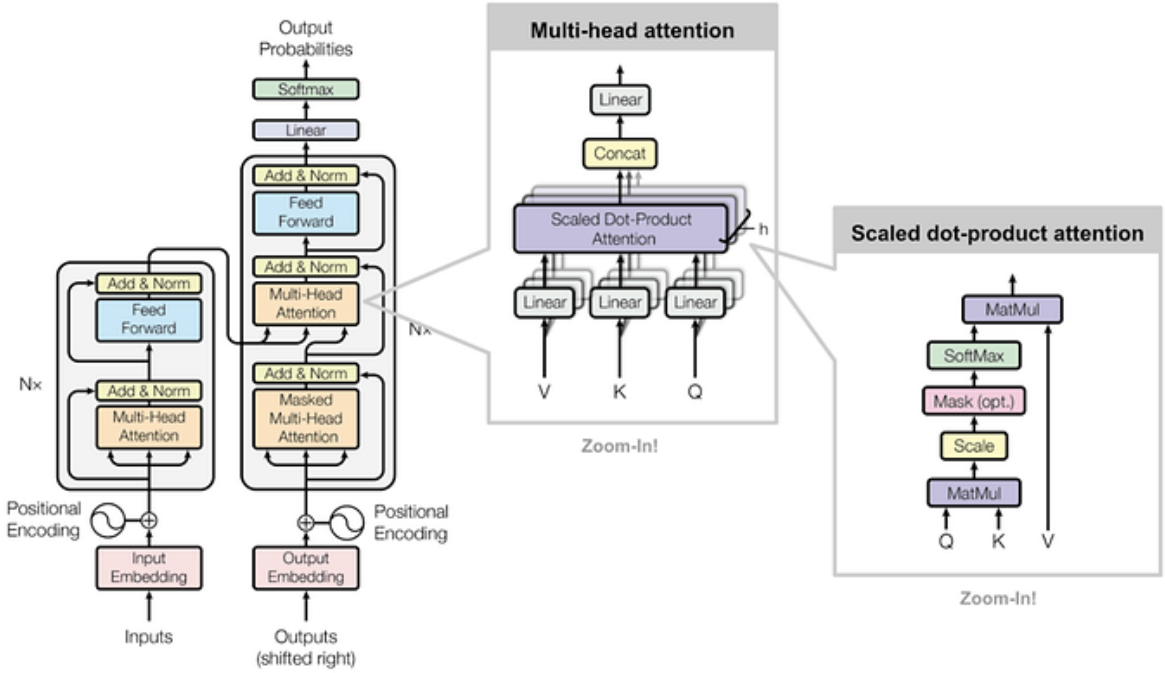
PagedAttention 将请求的 KV 缓存划分为块，每个块可以包含固定数量 token 的注意力键和值。在PagedAttention中，KV缓存的块不一定存储在连续的空间中。因此，我们可以像操作系统的虚拟内存一样以更灵活的方式管理KV缓存：

将块视为页面，将令牌视为字节，将请求视为进程。

这种设计通过使用相对较小的块并按需分配来减少内部碎片。此外，它*消除了外部碎片*，因为所有块都具有相同的大小。最后，它支持*以块为粒度*、跨与同一请求关联的不同序列甚至跨不同请求的*内存共享*。

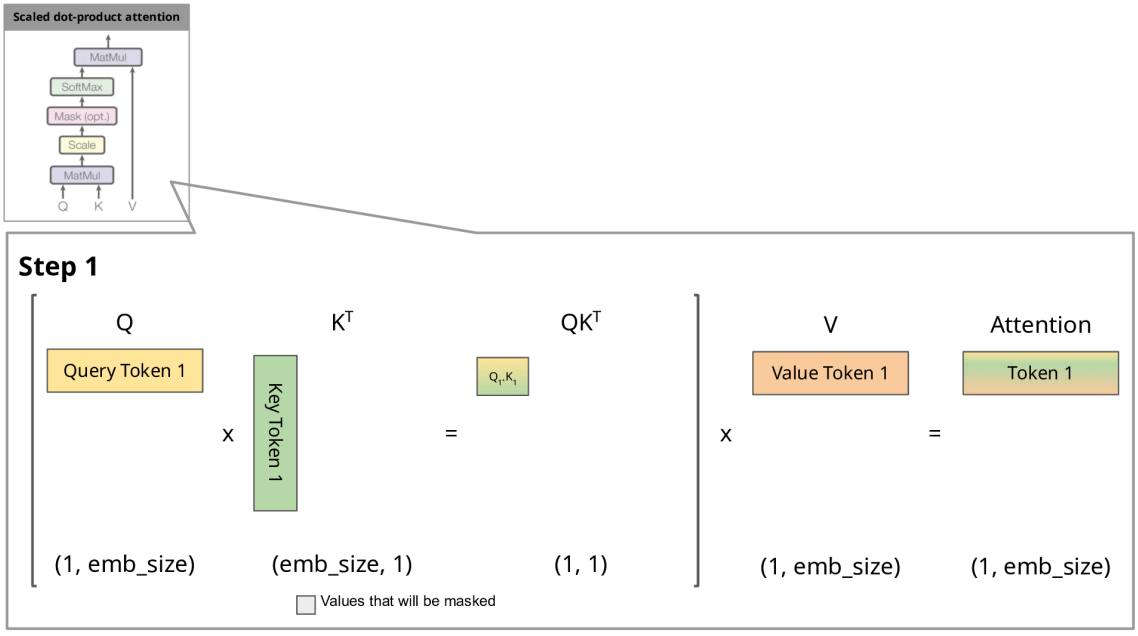
1.2 KV-cache

KV-cache主要应用于transformer中：

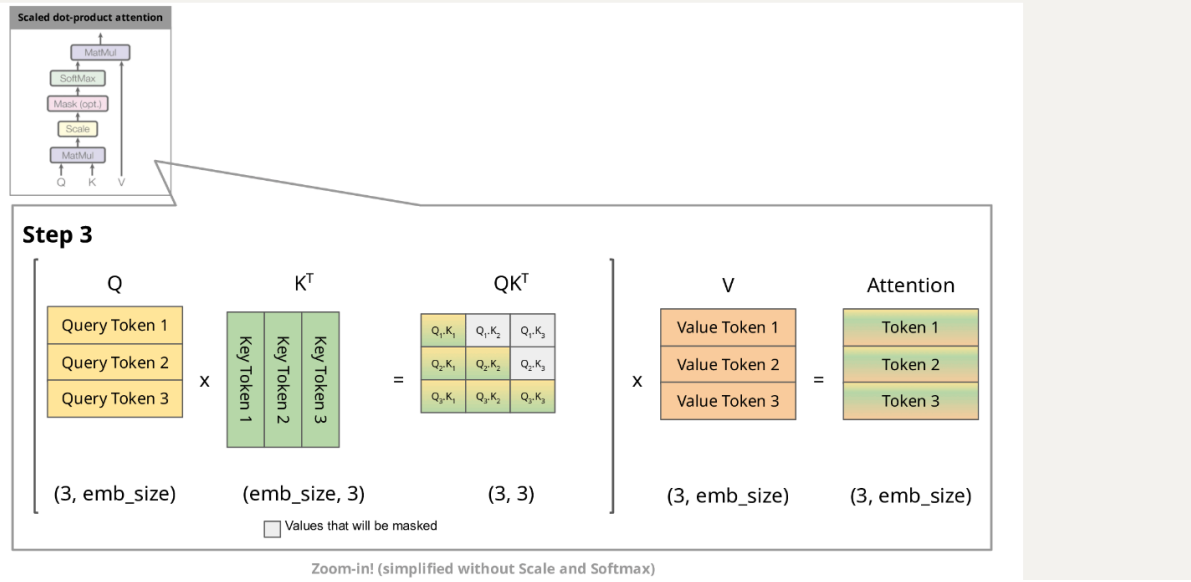


图解：*KV caching occurs during multiple token generation steps and only happens in the decoder.*

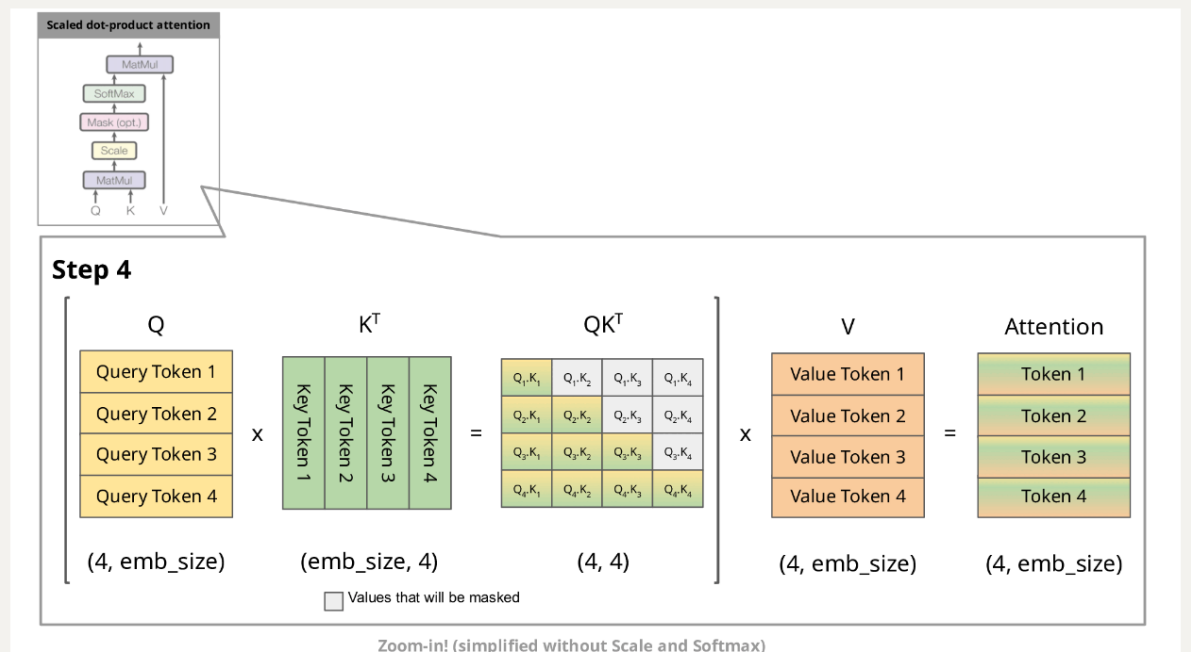
如果我们确保数据不会向后看，我们才能够前半段生成好的Key和Value是不会因为新生成的数据而改变的，



Zoom-in! (simplified without Scale and Softmax)



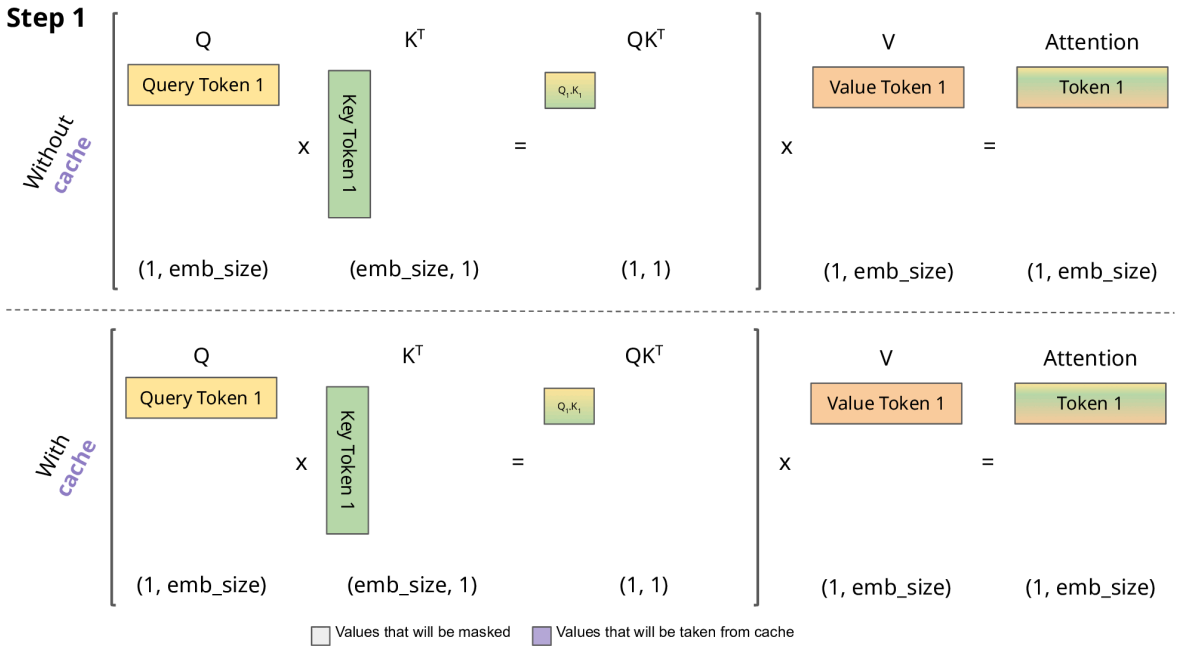
Zoom-in! (simplified without Scale and Softmax)

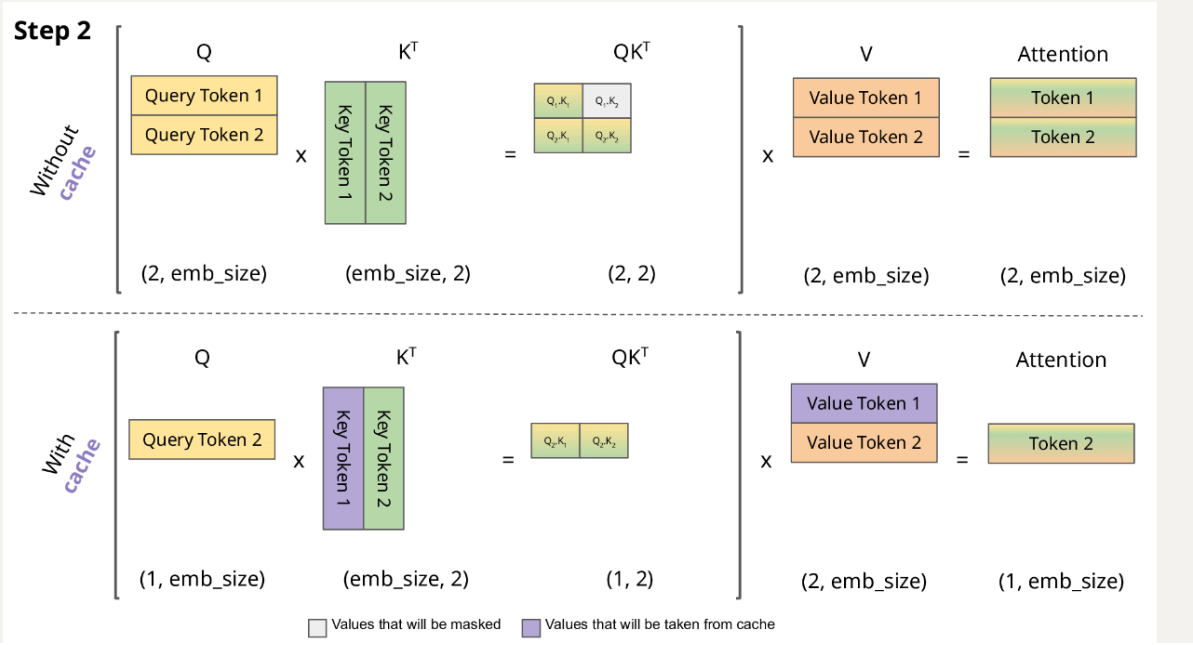


Zoom-in! (simplified without Scale and Softmax)

自从解码器是因果性的(即一个token的注意力只依赖于它之前的tokens),在每一个生成步骤中,我们都在重新计算同样的之前token的注意力,而实际上我们只想计算新token的注意力。

这就是KV (key-value) 缓存发挥作用的地方。通过缓存之前的Keys和Values,我们可以专注于只计算新token的注意力,而不需要重复计算之前token的注意力。





这个是很局部的东西，是在同一层attention当中进行的cache。

2 模型压缩

主要的优化方式包括：

- **量化**：将 float32 权重或激活转换为低位浮点或整数。更少的位数意味着更少的内存需求。此外，更少的比特可以指示更高的并行性和更快的推理速度。
- **剪枝**：致力于删除预先设计的模型中不重要的组件（例如神经元、层等），从而减少推理成本中的内存和计算成本。
- **知识蒸馏**：引入了一个预训练的大型模型作为教师，并将其知识转移到一个新的较小模型（称为学生模型）。然后，较小的模型将具有与教师几乎相同的能力，并且享有更少的内存和计算成本。

- **紧凑的架构设计**：以较低的成本设计新的算子来替换（通常近似）原始模型中的繁琐算子。对于 Transformer 模型，自注意力是主要目标，并且经常被其他算子取代。
- **动态网络**：以不同的方式对待每个推理样本。原始模型是一个超网，每个样本只选择超网的一个子结构进行推理。专家混合（MoE）是一种动态推理。（稀疏化，不100%调用）在

2.1 量化

量化方法

神经网络量化的基本概念和方法，包括量化误差、后训练量化和量化感知训练、静态/动态量化、模拟/整数量化、仅权重/权重+激活量化等。其中，量化感知训练可以通过模拟量化过程或使用额外参数来解决量化误差问题，而静态量化和动态量化则分别适用于权重和激活量化。此外，模拟量化可以减少神经网络的内存成本和数据移动时间，而整数量化则可以进一步加速低位操作。

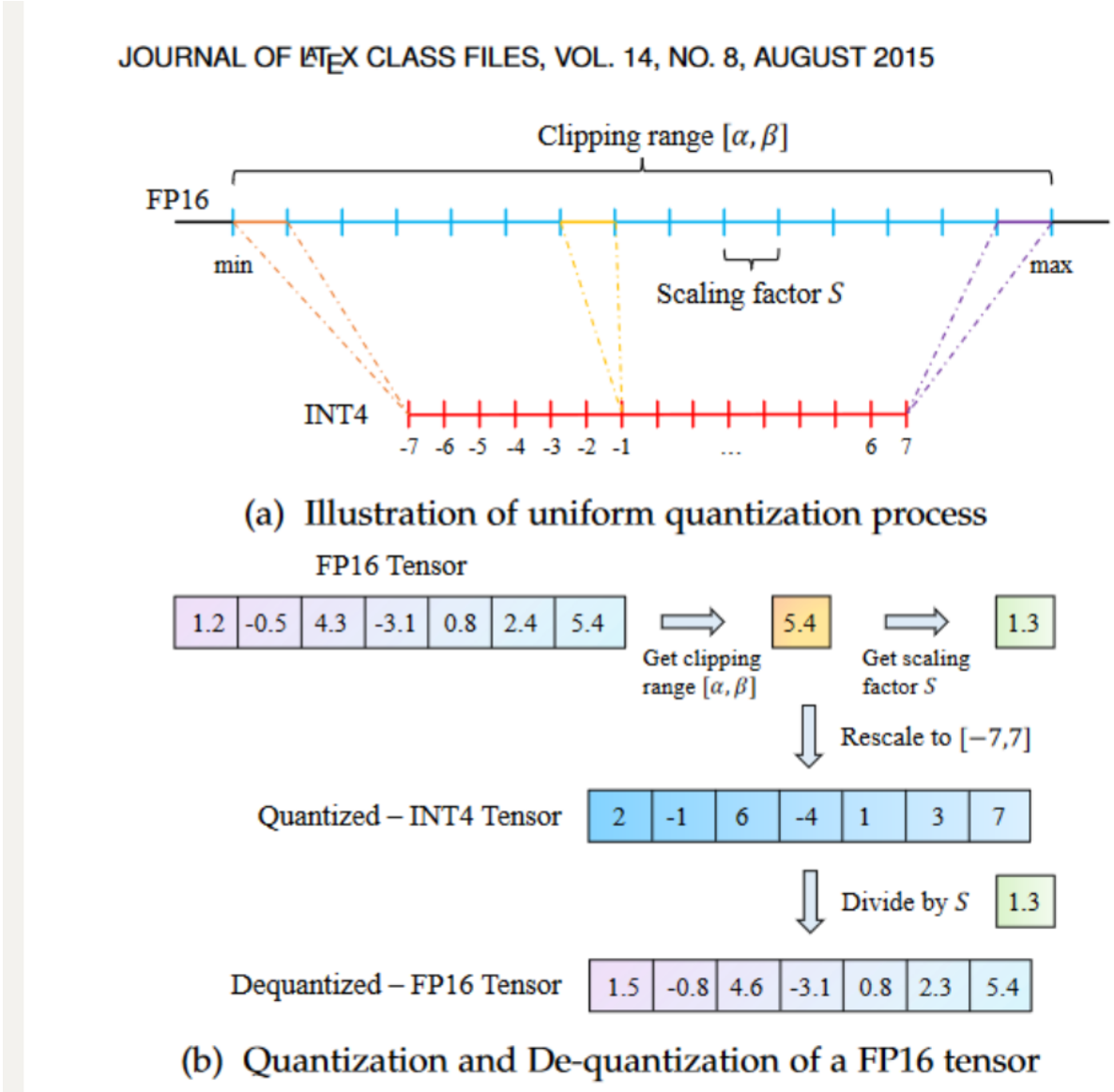


Fig. 2: (a) Uniform quantization separates a real-valued range into *uniform, finite* intervals and then maps real values within the same interval to the same integer. (b) An FP16 tensor is quantized into INT4 format and then dequantized back into FP16.

中等规模语言模型的量化方法主要采用QAT框架，而不是PTQ，因为重新训练这样的模型的成本相对可接受。重新训练带来的评估指标（如准确性）提升显著，特别是在极低比特设置下。

- **PTQ**（训练后量化）：无需重新训练的方法，量化模型可直接用于推理。

  一些针对BERT模型量化的方法，早期的工作主要是将权重量化为INT8，如Q8BERT。一些更复杂的方法可以将量化位宽降至低于8位，如Q-BERT、TernaryBERT和BinaryBERT。还有一些方法可以自动平衡模型性能降低和量化位宽之间的关系，如Differentiable Neural Architecture Search。这些方法都旨在减小模型的大小和计算量，以便在嵌入式设备上部署BERT模型。

- **QAT**（量化感知训练）：重新训练的方法，有助于恢复量化引入的误差。

  GOBO使用非均匀量化（即聚类）将符合高斯分布的大多数权重量化为3位，并将一些离群权重单独保存为FP32。I-BERT为特定的非线性函数（例如GeLU、Softmax、LayerNorm）设计了整数近似方法，以实现端到端的整数BERT推理，无需任何浮点计算。Dai等人使用更细的粒度来减少量化误差，将权重和激活量化为4位，并使用分组量化（例如，将16~64个连续权重作为一组）来确定每组的缩放因子。

此外，需要注意的是，*PTQ方法得到的量化参数通常可以作为QAT方法重新训练的良好初始化点*。

量化对象

- Weight-Only Quantization

> 本文介绍了几种常见的量化方法，其中最常见的是基于权重的量化方法。LUTGEMM使用二进制编码量化格式，将LLM的参数分解为二进制参数和一组缩放因子，以加速量化矩阵乘法。GPTQ提出了一种基于Optimal Brain Quantization的逐层量化方法，将LLM量化为3/4位。QuIP通过利用Hessian矩阵的LDL分解来优化权重，并使用随机正交矩阵的Kronecker积来确保权重和Hessian矩阵之间的不相关性，成功将LLM量化为2位。
>
> 本文介绍了一些减小LLMs量化误差的方法，其中包括存储对LLMs量化性能影响最大的敏感权重，并使用高精度存储；使用L2误差作为权重敏感度度量的SpQR方法；使用Hessian矩阵近似权重敏感度的SqueezeLLM方法等。这些方法都能有效地减小LLMs的量化误差，提高模型的速度和性能。

- Weight-Activation Quantization

> 这些研究关注于LLMs中的权重-激活量化。一些方法采用分组量化权重和逐令量化激活，将LLMs的精度降至INT8。由于LLMs中存在激活值的异常值，一些方法采用特殊处理来减少量化误差，如高精度存储异常特征维度、平滑激活值、通道重排和OVP量化等。还有一些方法采用浮点格式来处理异常值，并引入搜索框架来确定最佳指数偏差和最大量化值。OmniQuant则通过调整权重的极端值来处理激活值的异常值。

- KV Cache Quantization

> 近期研究致力于通过kv cache量化来减少LLMs的内存占用和加速推理。KVQuant提出了多种kv cache量化方法，如Per-Channel Key Quantization、PreRoPE Key Quantization和Non-Uniform kv cache quantization。KIVI发现key cache应该按通道量化，而value cache应该按标记量化，并成功将KV cache量化为2位。WKVQuant通过整合过去的量化来优化注意力计算，采用二维量化策略来有效管理key/value（KV）cache的分布，并利用交叉块重构正则化来优化参数，实现了权重和KV cache的量化，节省了内存，同时几乎达到了仅权重量化的性能水平。

潜在问题

BERT-like模型的量化方法在成功之后，对生成式语言模型（如GPT、BART）的量化尝试较少。这是因为：*在逐标记生成过程中，量化误差会累积，使得量化生成式语言模型成为一个更加复杂的问题。*

研究表明，将为BERT-like模型设计的量化方法直接应用于生成式语言模型受到了均匀的词嵌入和权重分布不均匀的阻碍。为了解决这些挑战，研究者提出了两种解决方案：基于标记级对比蒸馏和模块相关的动态缩放。DQ-BART采用了QAT框架和蒸馏训练目标，通过最小化量化和蒸馏低精度学生模型与全精度教师模型之间输出对数、注意力和隐藏状态的差异来蒸馏和量化一个序列到序列模型，即BART。

因此，尝试知识蒸馏可能是一个好的选择。

2.2 剪枝

神经网络压缩和加速的传统技术之一——剪枝，通过消除模型中的非必要权重或结构来压缩模型，同时保持网络性能接近原始状态。然而，对于大型语言模型（LLMs），剪枝的效果不如量化和蒸馏等其他压缩技术。

剪枝是一种减少模型大小或复杂度的强大技术，可分为非结构化剪枝、半结构化剪枝和结构化剪枝。结构化剪枝基于特定规则删除整个组件，而非结构化剪枝则剪枝单个参数，导致不规则稀疏结构。半结构化剪枝是介于结构化剪枝和非结构化剪枝之间的方法，能够同时实现细粒度剪枝和结构正则化。

Table 2: The performance of various representative LLM pruning methods.

Category†	Methods	LLM	Perplexity Difference (WikiText-2)‡	Compression Rate	Speed up
Unstructured Pruning	SparseGPT	OPT-175B	-0.14	50%	-
	Wanda	LLaMA-65B	1.01	50%	-
	SAMSP	LLaMA2-13B	0.63	50%	-
	DSnoT	LLaMA-65B	2.08e4	90%	-
Structured Pruning	LLM-Pruner	LLaMA-13B	3.6	20%	-
	Shortened LLaMA	LLaMA-7B	10.5	35%	-
	FLAP	LLaMA-65B	7.09	50%	-
	SliceGPT	LLaMA2-70B	1.73	30%	1.87×
Semi-Structured Pruning	E-Sparse	LLaMA-65B	2.13	2:4	1.53×
	SparseGPT	OPT-175B	0.39	2:4	2×
	Wanda	LLaMA-65B	2.69	2:4	1.24×

† : The results presented in the table are solely derived from the original papers.
‡ : (The perplexity of the pruned LLM) - (The perplexity of the origin LLM).

Unstructured Pruning

无结构修剪保留了修剪模型的性能，因此与无结构修剪相关的工作通常不需要重新训练来恢复性能。然而，无结构修剪使修剪后的模型不规则化，需要专门处理或软件优化以加速推断。SparseGPT引入了一种一次性修剪策略，将修剪视为广泛的稀疏回归问题，并使用近似稀疏回归求解器来解决。SparseGPT在最大的GPT模型上实现了显著的无结构稀疏性，即OPT-175B和BLOOM-176B，而困惑度几乎没有增加。

Structured Pruning

基于结构化剪枝的语言模型压缩方法，可分为基于损失、基于幅度和基于正则化。其中，基于损失的剪枝方法通过测量剪枝单元对损失或梯度信息的影响来评估其重要性。基于幅度的剪枝方法则通过测量权重大小来评估其重要性。基于正则化的剪枝方法则通过对权重进行正则化来实现剪枝。

- 基于大小的剪枝方法通过度量剪枝单元的大小来评估其重要性，并剪掉得分低于预定阈值的单元。
- 基于正则化的剪枝方法通过在损失函数中添加正则化项来诱导稀疏性，例如L0、L1和L2正则化。

- 结构化剪枝通常通过去除冗余参数来减小模型大小，但可能会降低模型性能。一种新方法是将知识蒸馏与结构化剪枝相结合，以帮助较小的模型保持性能并减小其大小。

3 其他方法

知识蒸馏是一种将大型复杂模型的知识转移到较小简单模型的技术。这些方法分为黑盒蒸馏和白盒蒸馏两种类型。黑盒蒸馏只能访问教师模型的输出，通常来自闭源LLMs；白盒蒸馏可以访问教师模型的参数或输出分布，通常来自开源LLMs。

白盒知识蒸馏能够帮助学生LM更深入地理解教师LLM的内部结构和知识表示，通常会导致更高水平的性能改进。MINILLM使用反向Kullback-Leibler散度目标进行蒸馏，防止学生模型高估教师分布的低概率区域，并提出了有效的优化方法。GKD探索了从自回归模型进行蒸馏的方法，通过使用自动生成的输出训练学生模型，并允许在学生无法完全复制教师分布时使用不同的损失函数。TED提出了一种面向任务的逐层蒸馏方法，设计了任务感知滤波器，以减少学生和教师模型之间的知识差距。白盒蒸馏通常涉及理解和利用LLM的内部结构，更深入地探索不同网络结构和层之间的交互可以提高白盒蒸馏的有效性。黑盒蒸馏可以从封闭源LLM中蒸馏知识到开源LLM，然后使用白盒蒸馏将知识转移给学生LLM。

低秩分解可以将大矩阵分解成小矩阵以节省空间和计算量。最近的研究尝试使用低秩分解来压缩LLM，并取得了显著的成功。LPLR通过随机低秩和低精度分解来压缩LLM的权重矩阵。ASVD通过分析每个层的权重矩阵的奇异值分布，为不同层分配最适合的压缩比率，从而确保在压缩过程中模型性能的最小损失。LASER通过选择性减少权重矩阵的高阶分量来改善Transformer模型的性能。

4 llama_cpp工具

完成了cousra课程，发现llama.cpp可以简化代码复杂程度，可以作为部署的一种方案。