# 自制大模型推理框架-第20次课程-LLama3.2模型的支持

## 背景介绍

在9月26日，Meta公司在Meta开发者大会上一共发布了Llama 3.2的4个型号：

- 110亿和900亿参数的多模态版本
- 10亿参数和30亿参数的轻量级纯文本模型

官方数据显示，与同等规模的“中小型”大模型相比，Llama 3.2 11B和90B表现出了**超越闭源模型**的性能。

尤其是在图像理解任务方面，Llama 3.2 11B超过了Claude 3 Haiku，而90B版本更是能跟GPT-4o-mini掰掰手腕了。

## 视频

llama3.2模型的支持.mp4

## 新闻摘要

Meta 公司推出了 Llama 3.2 版本，该版本新增了中等规模的视觉与文本模型（参数分别为 11B 和 90B）以及适用于移动设备的轻量级模型（参数分别为 1B 和 3B）。这两个轻量级模型支持长达 128,000 个标记的上下文，专为设备端处理（如摘要生成和指令执行）而设计，并且已经针对高通和联发科的硬件进行了优化。对于 1B 和 3B 参数的模型，Meta 是通过对 Llama 3.1 的 8B 参数模型进行一次性结构化剪枝来创建的，即通过移除网络部分、调整权重，构建出更小、更高效且能保持原有性能的模型。知识蒸馏是大网络向小网络传授知识的过程，小模型可借此提升性能。在 Llama 3.2 的 1B 和 3B 参数模型中，预训练阶段将 Llama 3.1 的 8B 和 70B 参数模型的逻辑值作为标记级目标融入其中，在剪枝后利用知识蒸馏恢复性能。

课程将重点介绍 1B 和 3B 参数模型，它们在多语言文本生成和工具调用方面有着出色的表现，有助于开发基于设备的智能应用并保障用户隐私。例如可以用于汇总近期消息、提取待办事项或者直接安排会议。同时结合我们已经参加课程同学的硬件情况，1B和3B的模型也的确最适合用于我们的教学。

Vision instruction-tuned benchmarks

| Modality | Benchmark | Llama 3.2 11B | Llama 3.2 90B | Claude 3 - Haiku | GPT-4o-mini |
|---|---|---|---|---|---|
| Image | College-level Problems and Mathematical Reasoning MMMU (val, 0-shot CoT, micro avg accuracy) | 50.7 | **60.3** | 50.2 | 59.4 |
| | MMMU-Pro, Standard (10 opts, test) | 33.0 | **45.2** | 27.3 | 42.3 |
| | MMMU-Pro, Vision (test) | 23.7 | 33.8 | 20.1 | **36.5** |
| | MathVista (testmini) | 51.5 | **57.3** | 46.4 | 56.7 |
| | Charts and Diagram Understanding ChartQA (test, 0-shot CoT relaxed accuracy)* | 83.4 | 85.5 | 81.7 | — |
| | AI2 Diagram (test)* | 91.1 | 92.3 | 86.7 | — |
| | DocVQA (test, ANLS)* | 88.4 | 90.1 | 88.8 | — |
| | General Visual Question Answering VQAv2 (test) | 75.2 | 78.1 | — | — |
| Text | General MMLU (0-shot, CoT) | 73.0 | **86.0** | 75.2 (5-shot) | 82.0 |
| | Math MATH (0-shot, CoT) | 51.9 | 68.0 | 38.9 | **70.2** |
| | Reasoning GPQA (0-shot, CoT) | 32.8 | **46.7** | 33.3 | 40.2 |
| | Multilingual MGSM (0-shot, CoT) | 68.9 | 86.9 | 75.1 | **87.0** |



## LLama2和LLama3模型的差异

> 这里的差异主要是我们在移植过程当中会遇到的，其他差异也有但是和我们关系不大，所以就没有展开。

### 上下文窗口的变长

LLama3.1 的上下文窗口从 8k 扩展到 128k，扩大了 16 倍，从而能够支持更长的文本处理。较大的上下文窗口使 LLama3.1 可处理更复杂的任务，例如长文本理解和多轮对话，还提高了模型的推理能力。这在我们课程的推理框架中会产生一定影响，**因为我们的 KV Cache 是依据模型的最长窗口长度（max seq len）来申请空间的，所以窗口长度增加会导致我们的 KV Cache 占用更多的显存空间。**

### 对输入编码的处理

> 参考资料：https://wmathor.com/index.php/archives/1517/

与 Llama 2 相比，**Llama 3 最大的变化在于采用了新的 Tokenizer，**其词汇表大小扩展至 128,256（之前版本为 32,000 个 Token）。这个更大的词汇库能够更高效地对输入和输出的文本进行编码，并且有可能提升模型的多语种处理能力。

不过，**这也会导致嵌入层权重矩阵尺寸增大**，进而使得小型模型的参数从 Llama 2 的 7B 增加到 Llama 3 的 8B，换句话说，因为词表的关系，会使得LLama3的模型在同等规模下占用显存更多。

同时，这两个模型分别采用了不同的分词器，Llama 2 使用 SentencePiece 进行标记化，而 Llama 3 已经过渡到使用 OpenAI 的 Tiktoken。以下是简单版本的BPE算法的实现流程：

1. 设定最大的子词（subwords）数量 ☒。
2. 将所有单词拆分为单个字符序列，并在末尾添加单词结束符 </w>，同时记录该单词出现的频次。例如，若 “low” 这个单词出现了 5 次，它将被表示为 {"l o w </w>": 5}。
3. 统计每个连续字节对（byte-pair）的出现频率，选取频率最高的字节对合并为新的子词（subword）。
4. 重复第 3 步，直至达到第 1 步设定的期望子词词表大小，或者下一个最高频的字节对出现频率为 1。

总的来说，我们会先从上到下的顺序遍历输入文本，找到出现频率最高的相邻字符或子词，然后进行合并。由于更新后的文本中出现的新的相邻字符或子词可能也会成为下一轮迭代中的候选，因此我们需要反复迭代，直到达到指定的词汇表大小为止。现在假设现在把所有单词都拆分成单个字符，并在末尾添加单词结束符：

```
{'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w e s t </w>': 6, 'w i d e s t </w>': 3}
```

以下流程都是在一个循环的多轮中进行的 ，

1. 分析以上被拆分的单词中，两个字符同时出现的概率，以pair的形式进行组织，pair的key是两个连续出现的字符，pair的value是这两个连续出现字符的频次，流程结束后得到的输出为：

```
{('l', 'o'): 7,
('o', 'w'): 7,
('w', '</w>'): 5,
('w', 'e'): 8,
('e', 'r'): 2,
('r', '</w>'): 2,
('n', 'e'): 6,
('e', 'w'): 6,
('e', 's'): 9,
('s', 't'): 9,
('t', '</w>'): 9,
('w', 'i'): 3,
('i', 'd'): 3,
('d', 'e'): 3})}
```

2. 我们选中以上组合中频次最高的一个，也就是('e', 's'): 9，并对vocab中同时出现e s的地方进行替换，例如在这一项中的：

```
'n e w e s t </w>': 6
```

应该被替换为n e w est</w> ，简单来说就是将e s替换为es. 把两个分开的字符进行合并。

```
{'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w es t </w>': 6, 'w i d es t
</w>': 3}
```

在下一轮的迭代中，同时顺序出现次数最多的是es和t，可以看到在n e w es t和w i d es t中分别出现了6次和3次，一共出现了9次，所以我们将es t进行合并。

```
{'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w est </w>': 6, 'w i d est </w>':
3}
```

经过多轮迭代之后，达到预设的 subwords 词表大小或下一个最高频的字节对出现频率为 1，对于原始的输入：

```
{'l o w </w>': 5, 'l o w e r </w>': 2, 'n e w es t </w>': 6, 'w i d es t
</w>': 3}
```

**经过10次迭代后，最后输出的单词组合为：**

```
{'low</w>': 5,
'low': 2,
'e': 2,
'r': 2,
'</w>': 2,
'newest</w>': 6,
'wid': 3,
'est</w>': 3}
```

这样我们就得到了更加合适的词表，这个词表可能会出现一些不是单词的组合，但是其本身有意义的一种形式。有同学可能会好奇，</w>是什么意思？停止符 </w> 的意义在于表示 subword 是词后缀。举例来说：st 不加 </w> 可以出现在词首，如 st ar；加了 </w> 表明该字词位于词尾，如 wide st</w>，二者意义截然不同。**随后我们还需要将输出的单词组合进行逆序处理，让出现频率最高的一项排在最前方，也就是：**

```
['newest</w>', 'low</w>', 'est</w>', 'wid', 'low', 'e', 'r', '</w>']
```

当我们获取词表之后，就能够对输入的单词进行编码了。**在之前的算法里，我们已经得到了子词（subword）词表，还应将该词表按照对应出现的个数从多到少进行排序。**在编码时针对每个单词，**遍历排好序的子词词表，查看是否存在标记（token）为当前单词的子字符串，如果存在，那么这个标记就是表示该单词的标记（tokens）之一。**因此有以下的编码过程：

```
# 给定单词序列
["the</w>", "highest</w>", "mountain</w>"]

# 排好序的subword表
# 长度 6         5           4        4         4       4           2
["errrr</w>", "tain</w>", "moun", "est</w>", "high", "the</w>", "a</w>"]

# 迭代结果
"the</w>" -> ["the</w>"]
"highest</w>" -> ["high", "est</w>"]
"mountain</w>" -> ["moun", "tain</w>"]


{'low</w>': 5, 'low': 2, 'e': 2, 'r': 2, '</w>': 2, 'newest</w>': 6, 'wid': 3, 'est</w>': 3}
'newest</w>': 6
'low</w>': 5

widlow --> [wid,low</w>]
```

我们从最长的 token 迭代到最短的 token，尝试将每个单词中的子字符串替换为 token。 最终我们会迭代所有 tokens，并将所有子字符串替换为 tokens。 如果仍然有子字符串没被替换但所有 token 都已迭代完毕，则将剩余的子词替换为特殊 token，如 <unk>。

## 课程项目中对bpe算法的接入

### 单词表打开部分

和spe算法一样，bpe类也继承于EncodeLayer，

```
class BpeEncodeLayer : public EncodeLayerBase {
public:
  explicit BpeEncodeLayer(std::string token_model_path, bool has_bos, bool has_eos);

```

```
    std::vector<int32_t> encode(const std::string& sentence) const override;

    std::string decode(int32_t token_id) const override;

    std::string decode(const std::vector<int32_t>& token_ids) const override;

    bool is_sentence_ending(int32_t token_id) const override;

    int32_t vocab_size() const override;

  private:
    int32_t bos_id_ = -1;
    int32_t eos_id_ = -1;
    int32_t stop_token1_ = -1;
    int32_t stop_token2_ = -1;
    int32_t num_token_ = 0;
    std::unique_ptr<tiktoken::tiktoken> tiktoken_;
};
```

我们bpe算法实现采用了qwen.cpp中的实现，见kuiper/include/base/tiktoken.h，编码过程中使用的单词表使用了HuggingFace中的，共两种类型的单词，一类是特殊单词，例如<|begin_of_text|>表示一段文本的开始，<|end_of_text|>表示一段文本的结束，以及其他的一些特殊单词单词。

```
BpeEncodeLayer::BpeEncodeLayer(std::string token_model_path, bool has_bos,
bool has_eos)
    : EncodeLayerBase(std::move(token_model_path), has_bos, has_eos) {
    ...
    ...
  }
```

Huggingface的单词表（https://huggingface.co/unsloth/Llama-3.2-1B-Instruct/blob/main/tokenizer.json）以Json格式来保存，所以我们要使用json库对它进行读取，采用的是https://github.com/nlohmann/json ，这个库以友好的用户接口见长。

```
{
  "version": "1.0",
  "truncation": null,
  "padding": null,
  "added_tokens": [
    {
      "id": 128000,
      "content": "<|begin_of_text|>",
```

```
      "single_word": false,
      "lstrip": false,
      "rstrip": false,
      "normalized": false,
      "special": true
    },
    ]
  }
```

我们在BpeEncodeLayer构造函数中对特殊单词的词表进行读取：

```
BpeEncodeLayer::BpeEncodeLayer(std::string token_model_path, bool has_bos,
bool has_eos)
    : EncodeLayerBase(std::move(token_model_path), has_bos, has_eos) {
  using json = nlohmann::json;
  std::ifstream f(token_model_path_);

  json data = json::parse(f);
  const auto& datas = data["added_tokens"];
  ankerl::unordered_dense::map<std::string, int> special_tokens;
  for (const auto& data1 : datas) {
    int id = data1["id"];
    std::string content = data1["content"];
    special_tokens.insert({content, id});
  }
```

首先，我们打开 tokenizer.json 文件，然后运用 json 库对该文件进行解析。接下来，定位到名为 add_tokens 的字典，读取这个字典中的键值对，并把它们放置于 special_tokens 这一 map 结构（这里的 map 是单词和序号的映射关系）之中。

在此过程中，我们还需要从这个映射关系里读出几个特殊单词对应的序号，以方便后续的使用，例如表示文本起始、结束的特殊单词，即 <|begin_of_text|>、<|end_of_text|> 和 <|eot_id|> 等单词。同时也需要对常规的单词进行读取：

```
ankerl::unordered_dense::map<std::string, int> encoder;
const auto& vocabs = data["model"]["vocab"];
const auto& vocab_items = vocabs.items();
for (const auto& v : vocab_items) {
  const auto& key = v.key();
  const int32_t id = v.value();
  encoder[key] = id;
}
```

需要读取的常规单词在 json 文件中的存储格式如下：我们能够利用 json 库中的索引“model”、“vocab” 来获取相应位置的键值对，然后再将其逐个放入 encoder 变量中（encoder 类似于一个 map 结构，其 key 为单词本身，value 为单词对应的编码）。

```
"model": {
    "type": "BPE",
    "dropout": null,
    "unk_token": null,
    "continuing_subword_prefix": null,
    "end_of_word_suffix": null,
    "fuse_unk": false,
    "byte_fallback": false,
    "ignore_merges": true,
    "vocab": {
      "!": 0,
      "\"": 1,
      "#": 2,
```

## Tokenizer的初始化、编码与解码

### 初始化

```
tiktoken_ =
    std::make_unique<tiktoken::tiktoken>(encoder, special_tokens, PAT_STR);
```

其中encoder就是刚才赋值的单词和编码id，special tokens就是特殊的单词（比如句尾、句首）和对应的编码id。

### 编码

```
std::vector<int32_t> BpeEncodeLayer::encode(const std::string& sentence) const
 {
  CHECK(this->tiktoken_ != nullptr);
  std::map<std::string, std::string> replacements;
  replacements[" "] = "Ġ";// hello sentence
  // hello
  // [space]sentence

  // Ġsentence
  std::string s = absl::StrReplaceAll(sentence, replacements);
  auto input_ids = this->tiktoken_->encode(s);
```

```
11
12    if (has_bos_) {
13      input_ids.insert(input_ids.begin(), bos_id_);
14    }
15    if (has_eos_) {
16      input_ids.push_back(eos_id_);
17    }
18    return input_ids;
19  }
```

**sentence** 是我们要进行编码的句子，例如 “hello, the”。不过，为什么会有第 3 - 4 行这么奇怪的一个字符呢？这是因为在 hugging face 的编码表中，“the” 被当作一个特殊单词进行编码，并且在编码时，“the” 第一个位置的空格会被编为Ġ，所以在此我们要将空格替换为Ġ以与词表相对应。下面的两个判断是关于是否需要在编码后的句子前后添加特殊的起始或结束单词。

### 解码

```
1   std::string BpeEncodeLayer::decode(const std::vector<int32_t>& token_ids)
    const {
2     CHECK(this->tiktoken_ != nullptr);
3     auto s = tiktoken_->decode(token_ids);
4     std::map<std::string, std::string> reverse_replacements;
5     reverse_replacements["Ġ"] = " ";
6     const std::string& sentence = absl::StrReplaceAll(s, reverse_replacements);
7     return sentence;
8   }
```

在进行解码时，同样因为词表的缘故，依据词表索引解码得到的句子会呈现出 “hello,Ġthe” 这种情况，所以我们需要将Ġ替换为空格。

## 对Qwen Bpe算法的分析

### 单个单词的编码

我们以一个输入的单词为例，假设这个单词为 helxlo，不难发现它并非常见单词，而是在常见单词 hello 的基础上添加了两个随机字符。Bpe 算法将通过最长匹配的方式把它分割成三个子单词（sub - word），具体如下：

“hel”：50222

“x”：87

“lo”：385

分割编码算法可参考以下内容，即 kuiper/include/base/tiktoken.h中的方法：

```
_byte_pair_merge(
        const std::string &piece,
        const ankerl::unordered_dense::map<std::string, int> &ranks,
        std::function<int (int, int)> func
)
```

这里有三个参数，第一个参数piece是单个单词，第二个参数ranks是词表，第三个参数是一个回调函数从词表中获取单词的编码值。

```
_byte_pair_merge(...){
    for (auto i = 0U; i < parts.size() - 2; ++i) {
            auto rank = get_rank(parts, i, 0);
            if (rank) {
                    assert(*rank != std::numeric_limits<int>::max());
                    parts[i].second = *rank;
            }
    }
```

在_byte_pair_merge 函数里，要将单词的输入字符两两合并。例如在单词 helxlo 中，把 he 当作一组，el 当作一组，lx 当作一组，……，依此类推，**然后到词表中查找对应的两两组合能否构成一个已知的单词**，如果可以的话，就在 parts 数组中记录下对应 subword 的编码，例如he的编码是383.

如果一个单词能够被分割为多个 subword，且目前这些子词是由两个字符组成的，**那我们为了让它们尽可能地变长，也就是探究某个长度为 2 个字符的子词能否变为 3 个字符的长度。**在 auto rank = get_rank (parts, i, 1); **这一操作中，我们以当前的子词为基础（以i作为起始位置），将其长度延长一个字符进行探查。**例如，对于 he 这个单词，我们探查 hel 是否同样为词表中所记录的单词，如果是，就对 parts 中对应位置的 rank 进行更新。

```
_byte_pair_merge(...){
    if (min_rank.first != std::numeric_limits<int>::max()) {
            auto i = min_rank.second;
            auto rank = get_rank(parts, i, 1);
            if (rank) {
                    parts[i].second = *rank;
            } else {
                    parts[i].second = std::numeric_limits<int>::max();
            }
            if (i > 0) {
                    auto rank = get_rank(parts, i - 1, 1);
                    if (rank) {
                            parts[i - 1].second = *rank;
```

```
14                      } else {
15                              parts[i - 1].second =
    std::numeric_limits<int>::max();
16                      }
17              }
18
19              parts.erase(parts.begin() + (i + 1));
20      } else {
21              break;
22      }
```

同理，在代码的第 11 行到第 14 行也是为了多探查一个字符，只不过这一次是将 subword 的起始位置往前移一个位置。例如，对于子词 lo，将起始位置提前一个位置的话，就是探查 xlo 是否为词表中的一个单词，如果是，同样要更新 parts 中的记录。

删除无意义的 sub - word 记录，即执行 parts.erase (parts.begin () + (i + 1))。对于下标起始为 i 的单词（i，i + 2），**若（i，i + 3）同样是一个 sub - word**，那么（i + 1，i + 3）这项记录是无效的，因为它已经被（i，i + 3）这个项目所涵盖，所以需要删除。（这里我是这么理解的），这里的 2 是第一轮探查的长度，随着迭代次数的增加，此处探查的长度也会增加。经过多轮迭代和子词的合并之后，parts当中仅留下以下几项：

```
“hel”
“x”
“lo”
```

随后我们再将这些子词查表得到对应的编码值，于是流程结束。

```
“hel”: 50222
“x”: 87
“lo”: 385

[50222,87,385]
```

有同学可能会问，如果是一个词表中已经有的单词会怎么样，那当然是直接查表返回而不会有cut subword的过程。例如对于hello这个单词，因为词表中有所以会直接返回：

```
"hello": 15339
```

## 整句话的编码流程

1. 输入字符串。
2. 运用正则表达式的方法把字符串切割成两个部分，其中一部分是用于表示起始或者结束的特殊单词，另一部分则是需要进行编码的常规单词，切割后的结果是常规单词构成的一个数组。
3. 针对常规数组中的每一项均进行正则表达式匹配，并且将匹配到的单词按照单个单词来进行编码。

例如，现在输入为<begin of text>hello,my name，在经过步骤 2 和 3 中利用正则表达式去除特殊单词之后并切出子句之后，我们将常规单词以数组形式返回，如下所示：

- hello
- ,
- my
- [space]name。

4. 接着，我们针对这里的每个常规单词进行编码，依据词表获取相应的编码值。

### 解码流程

解码流程要简单得多，只需通过查询词表将对应的编码值转换为相应的单词，组成单词数组并返回即可。

## 需要额外安装的库（只有Llama3需要）

需要额外一些库来支持json和正则表达式，其实我感觉有点麻烦，但是毕竟是从别人那边借鉴的，也没有太好的办法。首先是需要在项目根目录的CMake中打开支持LLama3的选项：

```
option(LLAMA3_SUPPORT OFF)
if (LLAMA3_SUPPORT)
  message(STATUS "LLAMA3 SUPPORT")
  add_definitions(-DLLAMA3_SUPPORT)
endif()
```

1. 修改CMake，打开支持LLama3的方式如下，在使用CMake的时候带LLAMA3_SUPPORT参数，也就是在build目录下执行cmake .. **-DLLAMA3_SUPPORT=ON**
2. 环境安装
    - Abseil https://github.com/abseil/abseil-cpp
    - re2 https://github.com/google/re2
    - Json https://github.com/nlohmann/json

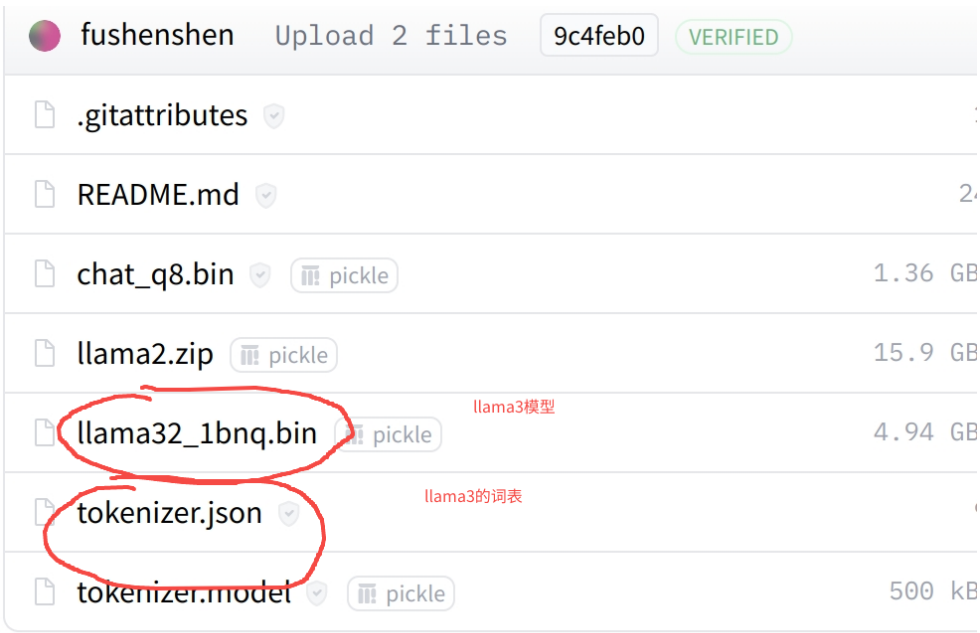
**安装的方式都是clone后，cmake，随后再使用make进行编译最后再make install执行安装**

3. 随后在main.cpp中选择对应的tokernizer类型，Bpe就是上文中的算法

```
model::LLama2Model model(base::TokenizerType::kEncodeBpe, tokenizer_path,
  checkpoint_path, false);
```

4. 下载LLama3对应的模型和tokenizer，随后再通过命令行进行传递。

https://huggingface.co/fushenshen/lession_model/tree/main




5. 执行推理

```
Generating...
hello, i am new to this forum. i am trying to find a way to get the "last" value of a variable. i have a variable that is set to a value when the program starts. i w
steps/s:147.732429

Process finished with exit code 0
```