简单大模型推理系统

欢迎各位同学。本课程中，各位将用Rust语言分阶段实现一个简单的大模型推理程序。

本课程分为两个阶段：作业阶段，各位将实现大模型的几个关键算子，Feed-Forward神经网络，以及大模型的参数加载；项目阶段，各位将实现大模型最为核心的Self-Attention结构，完成大模型的文本生成功能。之后，可以选择继续实现AI对话功能，搭建一个小型的聊天机器人服务。

- 本项目支持Llama、Mistral及其同结构的Transformer模型，所使用的数据类型为FP32，使用CPU进行推理。当然，欢迎各位同学在此基础上进行拓展。
- 本项目使用safetensors模型格式，初始代码只支持单个文件的模型。
- 本项目自带两个微型的语言模型，分别用于文本生成和AI对话（模型来自于Hugginface上的raincandy-u/TinyStories-656K和Felladrin/Minueza-32M-UltraChat）。对话模型比较大，需要到github页面的release里下载。

一、作业阶段

作业说明

- 你的代码需要通过全部已有的测试才能晋级下一阶段（项目包含github on-push自动检测）。
- 请在指定文件中和位置添加你的代码，不要修改其他文件和函数、文件名称和项目结构。作业阶段不需要额外的第三方依赖。
- 请不要修改已有的测试代码。开发过程中如果有需要，你可以添加自己的测例。
- 调试代码时，你可以打印张量的数据，你也可以使用pytorch中的函数辅助调试。

1. 算子：SiLU函数（10分）

请在 `src/operators.rs` 中实现SiLU算子，其公式为：

$$y = silu(x) \times y$$

其中

$$silu(x) = sigmoid(x) \times x$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

注意：

- ‘y‘ 既是输入，也存储最终输出。
- 该算子是element-wise操作而非向量点乘，即单次运算只涉及输入和输出张量中对应的元素。
- 你可以默认输入输出长度相同，不用考虑广播的情况。
- 用 `src/operators.rs` 中的测例检验你的实现是否正确。

2. 算子：RMS Normalization（20分）

请在 `src/operators.rs` 中实现RMS Normalization，其公式为：

$$y_i = \frac{w \times x_i}{\sqrt{\frac{1}{n}\sum_j x_{ij}^2 + \epsilon}}$$

注意：

- 你可以只考虑对最后一维进行计算的情况。即张量 $X(..., n)$ 和 $Y(..., n)$ 都是由若干个长度为 n 的向量 x_i, y_i 组成的，每次求和都在向量内进行。参数 ‘w‘ 是个一维向量，与各个向量长度相同，且进行element-wise乘法。
- 用 `src/operators.rs` 中的测例检验你的实现是否正确。

3. 算子：矩阵乘（30分）

想必前两个算子的实现中你已经充分热身，那么重量级的来了。请在 `src/operators.rs` 中实现矩阵乘（Transpose B）算子，其公式为：

$$C = \alpha AB^T + \beta C$$

你有充足的理由质疑为什么这个矩阵乘算子要长成这个样子，以及为什么不用线代课上学的 $C = AB$ 这样更简洁的形式。

首先，我们为什么要对B矩阵进行转置？这其实涉及到机器学习中线性（linear）层的定义习惯，‘$y = xW^T + b$‘ 。矩阵乘算子中的 ‘B‘ 矩阵常常是权重矩阵，它的每一行是一个与输入‘x‘中每一行向量等长的权重向量，而行的数量则对应了特征数。我们使用的模型的参数，也是按照这个方法存储的。

其次，为什么要加 ‘C‘ 矩阵？大家如果查阅BLAS中矩阵乘的标准定义就会发现它也有这一项操作，而我们这里其实实现的就是BLAS矩阵乘的一个简化版本。将矩阵乘结果加到原本的矩阵上在实际中是应用很广泛的。比如线性层的bias，尽管这次我们实现的Llama模型并没有使用bias。之后实现全连接网络时，你就会发现有了这一项，我们可以和矩阵乘一起实现残差连接（residual connection）的功能。如果你只想计算 ‘$C = AB^T$‘ 那么你可以在传参时将 ‘β‘ 参数设置为0，将 ‘α‘ 参数设置为1。

你可以默认输入输出都是二维矩阵，即 ‘A‘ 形状为 ‘$m \times k$‘，‘B‘ 形状为 ‘$n \times k$‘，‘C‘ 形状为 ‘$m \times n$‘，可以不用考虑广播的情况。到了项目部分，你有可能需要（非必须）实现支持广播的矩阵乘，比如

‘$(b, h, m, k) \cdot (b, 1, k, n)$‘ 这种情况，你可以去pytorch官方文档看到关于broadcast的规则。

你可以用 `src/operators.rs` 中的测例检验你的实现是否正确。

4. 模型结构：Feed-Forward神经网络（20分）

请在 `src/models.rs` 中实现Feed-Forward神经网络（mlp函数），计算过程如下：

```
hidden = rms_norm(residual)
gate = hidden @ gate_weight.T
up = hidden @ up_weight.T
hidden = gate * sigmoid(gate) * up ## silu
hidden = hidden @ down_weight.T
residual = hidden + residual
```

如果你正确地实现了之前地几个算子，那么这个函数的实现应该是相当简单的。需要注意的是，上一层的输出存储于residual这个临时张量中，这就是用到了我们之前提到的残差连接的概念，最终我们实现的神经网络的输出也要加上前一层的residual并存储于residual中，以便于下一层的计算。hidden_states则用于存储过程中的计算结果。你可以用 `src/model.rs` 中的测例检验你的实现是否正确。

5. Llama模型参数加载（20分）

请结合课上所讲的模型结构，根据代码种的定义在 `src/params.rs` 以及 `src/model.rs` 中补全大模型参数加载代码。项目已经为你做好了safetensors以及json文件的读取功能，你需要将参数原始数据以代码中的形式存于正确的位置，并赋予模型对象正确的config属性。safetensors里带有各张量的名称，应该足够你判断出张量代表的是哪个参数。

以下是大模型config中一些比较重要的属性的含义：

```
{
  "bos_token_id": 1, # 起始符token id
  "eos_token_id": 2, # 结束符token id
  "hidden_size": 128, # 隐藏层大小，即各层输出的最后一维
  "intermediate_size": 384, # Feed-Forward神经网络的中间层大小
  "max_position_embeddings": 512, # 最大序列长度
  "num_attention_heads": 8, # Self-Attention的Q头数
  "num_hidden_layers": 2, # 隐藏层数
  "num_key_value_heads": 4, # Self-Attention的K和V头数
  "rms_norm_eps": 1e-6, # RMS Normalization的epsilon参数
  "rope_theta": 10000.0, # RoPE的theta参数
  "tie_word_embeddings": true, # 起始和结束embedding参数矩阵是否共享同一份数据
  "torch_dtype": "float32", # 模型数据类型
  "vocab_size": 2048 # 词表大小
}
```

注意：

- safetensors里存储的是原始数据，你需要以FP32的形式读取出来，创建出项目所使用的张量。
- safetensors包含张量的形状，你无需对原始张量做任何变形。
- 当"tie_word_embeddings"属性被打开时，模型最开始以及最后的embedding矩阵数据相同，safetensors会只存储一份数据，我们测试用的story模型就是这样。作业阶段你可以只关心story模型，但是后续项目中你需要处理两个矩阵不同的情况。
- 你可以用 `src/model.rs` 中的测例检验你的实现是否正确。

二、项目阶段

1. 模型结构：Self-Attention

恭喜你，来到了本项目最为核心的部分。在开始写代码前，建议你对着课上讲的大模型结构图把每一次计算所涉及的张量形状都推导一遍，尤其是对于“多头”的理解。项目已经帮你实现了kvcache的部分和RoPE等一些算子，写这些代码其实对于大模型的学习很有帮助，但是为了不让项目过于新手不友好而省略了。

在输入经过三个矩阵乘后，我们分别得到了Q、K、V三个张量，其中Q的形状为 (seq_len, q_head×dim)，而K、V在连接完kvcache后的形状为 (total_seq_len, k_head×dim)，其中seq_len是输入序列的长度，可以大于1，total_seq_len是输入序列和kvcache的总长度 。你应该还记得课上的内容，在Q和K进行矩阵乘后，我们希望对于seq_len中的每个token每个独立的“头”都得到一个 (seq_len, total_seq_len) 的权重矩阵。这里就出现了两个问题:

第一，Q的头数和KV的头数并不一定相等，而是满足倍数关系，一般Q头数是KV头数的整数倍；假如Q的头数是32而KV头数是8，那么每4个连续的Q头用一个KV头对应。

第二，我们需要将 (seq_len, dim) 和 (dim, total_seq_len) 的两个矩阵做矩阵乘才能得到我们想要的形状，而现在的QK都不满足这个条件；你有几种不同的选择处理这个情况，一是对矩阵进行reshape和转置（意味着拷贝），再用一个支持广播（因为你需要对“头”进行正确对应）的矩阵乘进行计算，二是将这些矩阵视为多个向量，并按照正确的对应关系手动进行索引和向量乘法，这里我推荐使用更容易理解的后一种方法。

同样的，在对权重矩阵进行完softmax后和V进行矩阵乘时也会遇到这个情况。

对于每个头，完整的Self-Attention层的计算过程如下；

```
x = rms_norm(residual)
Q = RoPE(x @ Q_weight.T)
K = RoPE(x @ K_weight.T)
V = x @ V_weight.T
K = cat(K_cache, K)
V = cat(V_cache, V)
### 以下是你需要实现的部分
score = Q @ K.T / sqrt(dim)
attn = softmax(score)
x = attn @ V
x = x @ O_weight.T
residual = x + residual
```

Self-Attention的调试是很困难的。这里推荐大家使用pytorch来辅助调试。各位可以用transformers库（使用llama模型代码）来加载模型并运行，逐层检查中间张量结果。

2. 功能：文本生成

请在 `src/model.rs` 中补充forward函数的空白部分，实现generate函数。注意在foward函数的准备阶段，我们定义了几个计算用的临时张量，这是为了在多层计算中不重复分配内存，这些临时张量会作为算子函数调用的参数，你可以根据自己的需要更改这一部分（你其实可以用比这更小的空间）。

文本生成所需的采样的算子已为你写好。你需要初始化一个会被复用的kvcache，并写一个多轮推理的循环，每一轮的输出作为下一轮的输入。你需要根据用户传的最大生成token数以及是否出现结束符来判断是否停止推理，并返回完整推理结果。

所使用的模型在 `models/story` 中。 `src/main.rs` 已经为你写好了tokenizer的编码和解码，代码完成后，可以直接执行main函数。

3. （可选）功能：AI对话

仿照文本生成的功能，写一个实现AI对话的chat函数，之后你可以搭建一个支持用户输入的命令行应用。你需要在多轮对话中，保存和管理用户的kvcache。

你可以使用 `models/chat` 中的对话模型。其对话模板如下：

```
"{% for message in messages %}{{'<|im_start|>' + message['role'] + '\n' + message['content'] +
```

这种模板语言叫做Jinja2，在本项目中你可以不用实现任意模板的render功能，直接在代码中内置这个模板。你可以忽略system角色功能。下面是一个首轮输入的例子：

```
<|im_start|>system
{system_message}<|im_end|>
<|im_start|>user
{user_message}<|im_end|>
<|im_start|>assistant
```

后续每轮输入也都应该使用该模板。如果你忘记了如何使用模板生成正确的输入，请回顾课堂上讲到的内容，提示：我们的模型的基础功能是故事续写。

如果你完成了项目，请向导师展示你的成果吧！其实这个项目还有很多可以拓展的地方，比如其他数据类型的支持、多会话的支持、GPU加速等等，欢迎你继续探索。