配置

genereation_config = {

"bos_token_id": lq_tokenizer.tokenizer.bos_token_id,

"do_sample": True,

"eos_token_id": [128009],

”pad_token_id“= lq_tokenizer.tokenizer.pad_token_id，

"repetition_penalty": 1.15,

"temperature": 1.0,

"top_p": 0.001,

"top_k": 5，

"max_new_tokens"：1024

}

1. `bos_token_id` 用于标记序列的开始。
2. `eos_token_id` 用于标记序列的结束。
3. `eot_token` 通常也是用于表示文本结束，但在某些上下文中可能有不同的语义。
4. `do_sample`：控制生成文本时是否使用采样。`True` 时生成多样化的文本，`False` 时生成确定性文本。
5. `pad_token_id`：用于指定填充标记的 ID，以确保序列的统一长度，尤其在处理变长输入时非常有用。
6. `repetition_penalty`：减少重复生成相同内容的概率，防止模型生成过于单一的文本。

   值：通常设置为大于 1 的浮动值。较高的值会增加对重复的惩罚，鼓励模型生成更多新颖的内容。

- `repetition_penalty`设置为1.2，意味着模型在生成文本时会轻微惩罚重复的词组，以增加生成文本的多样性。

  > - 如果repetition_penalty大于1，减少重复词的生成概率。
  > - 如果repetition_penalty等于1，保持原有生成策略。
  > - 如果repetition_penalty小于1，增加重复词的生成概率。

- repetition_penalty=1.5` 表示在生成文本时，如果模型尝试重复相同的单词或短语，会受到更大的惩罚（1.5已经算大了），避免这种重复现象。

7. `temperature`：控制生成文本的随机性，温度越高越随机，越低越确定。e^x，x越大softmax越大。

   https://blog.csdn.net/qq_35812205/article/details/131714617

   值:

- `temperature=1.0`：通常表示“正常”生成，较为平衡。
- `temperature<1.0`：生成的文本更加确定性，模型倾向于选择概率最高的词。
- `temperature>1.0`：增加了随机性，生成的文本更加多样化和创意，但可能会牺牲一些连贯性。

8. `top_p`：通过设置一个累积概率阈值，决定生成时可以选择的词汇范围，通常用来提高生成文本的多样性。

   值：`top_p` 介于 0 和 1 之间。比如：

- `top_p=0.9`：模型会从概率累积达到 90% 的词汇中随机选择生成下一个词。
- `top_p=1.0`：没有任何限制，相当于全选模型所有词汇。

9. `top_k`：限制每次生成时可选择的词汇数量，常用于确保文本的连贯性并控制生成的随机性。

   值：`top_k` 是一个整数，表示每次生成时模型从前 `k` 个概率最高的词汇中进行选择。

- 例如：`top_k=50`，表示每次生成时从概率排名前 50 的词汇中进行选择。

10. max_new_tokens：最大生成token数

其它参数（少见）

正常不设置用默认的即可

https://blog.csdn.net/BIT_666/article/details/131983746

https://blog.csdn.net/a1920993165/article/details/134691021

四个例子

1. `num_beams`（束宽）

作用：控制束搜索（Beam Search）中保留的候选序列的数量。

- 束搜索是一种在文本生成中用于寻找最优序列的启发式算法。它通过在每个时间步选择多个最可能的词，并且保留前`num_beams`个最优的候选路径来探索。
- `num_beams`的影响：束宽越大，模型会考虑更多的可能性，但计算开销和生成时间也会增加；束宽越小，生成过程更快，但生成的文本质量可能会稍差，因为模型探索的可能性较少。

例如：

- `num_beams = 1`：实际上等同于贪心搜索（Greedy Search），模型只保留最优的路径，计算效率最高，但生成的文本可能会不够多样。
- `num_beams = 5`：模型会在每个时间步选择5个候选路径，计算开销更大，但有可能生成更优质的文本。

2. `length_penalty`（长度惩罚）

作用：控制生成文本的长度。这个超参数用来对生成过程中生成的文本的长度进行惩罚，避免生成过短或过长的文本。

- `length_penalty`的影响：
  - 当`length_penalty` > 1时，模型会惩罚较长的序列，倾向于生成较短的文本。
  - 当`length_penalty` < 1时，模型会奖励较长的序列，倾向于生成更长的文本。
  - 当`length_penalty` = 1时，不进行任何惩罚，生成的文本长度由模型的概率分布自然决定。

例如：

- `length_penalty = 1.2`：生成较短的文本。
- `length_penalty = 0.8`：生成较长的文本。

3. `num_return_sequences`（返回序列数量）

作用：控制模型返回的生成结果的数量。

- 这个参数决定了每次生成时返回多少个候选序列。它通常与`num_beams`一起使用，可以生成多个不同的序列，并通过某种方式选择最佳的一个。
- `num_return_sequences`的影响：返回的序列越多，模型生成的多样性越大，但计算开销也会随之增加。通常，`num_return_sequences`的值应该小于或等于`num_beams`，因为如果生成的序列数大于束宽，可能会导致多次重复的结果。

例如：

- `num_return_sequences = 3`：每次生成时返回3个不同的候选序列。
- `num_return_sequences = 1`：每次只返回一个生成结果。

结合使用的效果

- `num_beams`：决定生成时搜索的范围。更多的束宽意味着模型会尝试更多的候选序列，通常可以得到更高质量的文本。
- `length_penalty`：影响生成文本的长度。较大的值会生成较短的文本，较小的值会生成较长的文本。
- `num_return_sequences`：控制返回多少个候选文本。结合`num_beams`，可以返回多个不同的生成结果，并进行选择。

举个例子：

假设我们使用以下参数：

- `num_beams = 5`：模型会保留5个候选路径。
- `length_penalty = 1.2`：生成较短的文本。
- `num_return_sequences = 3`：每次返回3个不同的候选生成序列。

此时，模型会：

1. 在生成过程中选择5条最优路径，并通过束搜索探索可能的文本。
2. 对生成的文本进行长度惩罚，鼓励生成较短的文本。
3. 最终返回3个生成结果，提供更多的选择。

总结：

- `num_beams`：控制搜索的广度，影响生成质量和计算开销。默认1
- `length_penalty`：控制文本生成的长度，避免文本过长或过短。默认1.0
- `num_return_sequences`：控制每次生成时返回多少个候选序列。默认1

4. **`no_repeat_ngram_size`是在文本生成模型中常用的一个超参数，用来防止生成过程中出现重复的 n-gram（n-元组）。（默认0）**

   示例 1：不设置`no_repeat_ngram_size`

   假设模型生成了以下文本：

   ```
   arduino


   复制代码
   AI is the future. The future is bright.
   ```

这里，“the future”重复出现了两次，模型没有避免重复。

示例 2：设置 `no_repeat_ngram_size = 2`

如果设置了 `no_repeat_ngram_size = 2`，模型生成的文本将变为：

```
arduino


复制代码
AI is the future. The future looks bright.
```

这里，`"the future"` 不会再重复出现，因为它是一个 2-gram，设置了 `no_repeat_ngram_size` 后，

生成词作用顺序：

`top_p=0.9, top_k=50`：这表示模型将从概率累积达到 90% 的词汇中选择，并且最多只考虑前 50 个概率最高的词汇，还可以配合temperature

注意：

`temperature` 调整原始的概率分布。

`top_k` 或 `top_p` 进一步限制候选词汇的范围。

`repetition_penalty` 最后施加惩罚，减少重复。

repetition_penalty例子：

引入`repetition_penalty`的情况：

现在我们假设设置了`repetition_penalty = 1.5`（惩罚因子为1.5）。

当生成“苹果”时，模型会按照以下步骤来调整其概率：

1. 假设当前词是“苹果”，而它已经在之前的句子中出现过一次，`repetition_penalty`会对这个词产生影响。
2. 计算调整后的概率，公式为：

   Pnew(苹果)=P(苹果)repetition_penaltyP_{\text{new}}(\text{苹果}) = \frac{P(\text{苹果})}{\text{repetition_penalty}}Pnew(苹果)=repetition_penaltyP(苹果)

   假设原始概率为0.4，`repetition_penalty = 1.5`，那么：

   Pnew(苹果)=0.41.5=0.267P_{\text{new}}(\text{苹果}) = \frac{0.4}{1.5} = 0.267Pnew(苹果)=1.50.4=0.267

   即，原本40%的概率现在下降到26.7%。
3. 模型会根据调整后的概率分布来选择下一个词。例如，如果“苹果”被选中之前的概率是0.4（40%），经过惩罚后，它的概率下降到了0.267（26.7%），从而减少了重复生成的可能性。

重新归一化

1. 调整后的概率：假设一个词的概率在应用`repetition_penalty`后发生了变化，如之前的例子中，`P(苹果)` 从 0.4 降到了 0.267。

2. 重新计算总和：假设当前所有其他词的调整后概率分别是 `P(词1)`，`P(词2)` 等，调整后所有词的概率总和为 `S'`。

   S'=Pnew(苹果)+Pnew(词1)+Pnew(词2)+...S' = P{\text{new}}(\text{苹果}) + P{\text{new}}(\text{词1}) + P_{\text{new}}(\text{词2}) + \dotsS'=Pnew(苹果)+Pnew(词1)+Pnew(词2)+...

3. 重新归一化：然后，所有词的概率会按比例缩放，使得调整后的所有词的概率和为1。具体地，调整后的每个词的概率为：

   Pfinal(wi)=Pnew(wi)S'P{\text{final}}(w_i) = \frac{P{\text{new}}(w_i)}{S'}Pfinal(wi)=S'Pnew(wi)

   这样，经过归一化后，总概率就会恢复为1。

示例

假设有5个候选词：`苹果`，`香蕉`，`橙子`，`草莓`，`葡萄`，并且它们的原始概率分别为：

- `P(苹果) = 0.4`
- `P(香蕉) = 0.2`
- `P(橙子) = 0.2`
- `P(草莓) = 0.1`
- `P(葡萄) = 0.1`

通过应用 `repetition_penalty = 1.5`，`P(苹果)` 被调整为0.267，其它词的概率也可能会发生调整。假设调整后的概率如下：

- `P_{\text{new}}(苹果) = 0.267`
- `P_{\text{new}}(香蕉) = 0.2`
- `P_{\text{new}}(橙子) = 0.2`
- `P_{\text{new}}(草莓) = 0.15`
- `P_{\text{new}}(葡萄) = 0.15`

那么，调整后的概率总和为：

S'=0.267+0.2+0.2+0.15+0.15=0.967S' = 0.267 + 0.2 + 0.2 + 0.15 + 0.15 = 0.967S'=0.267+0.2+0.2+0.15+0.15=0.967

为了使总概率和为1，所有词的概率需要被缩放，即每个概率都除以 `S'`。计算后的概率为：

- `P_{\text{final}}(苹果) = \frac{0.267}{0.967} ≈ 0.276`
- `P_{\text{final}}(香蕉) = \frac{0.2}{0.967} ≈ 0.207`
- `P_{\text{final}}(橙子) = \frac{0.2}{0.967} ≈ 0.207`
- `P_{\text{final}}(草莓) = \frac{0.15}{0.967} ≈ 0.155`
- `P_{\text{final}}(葡萄) = \frac{0.15}{0.967} ≈ 0.155`

现在，这些调整后的概率的总和是1。

length_penalty例子：

假设：

我们使用一个简单的语言模型进行文本生成，当前已经生成了 5 个 token，接下来要生成第 6 个 token。

- 当前已生成的文本长度是 5（即已经生成了 5 个 token）。
- 第 6 个 token 的原始概率是 0.1。

我们设置 `length_penalty = 2.0` 来计算它对生成概率的影响。

调整后的概率计算：

模型的生成概率 $P_{\text{adjusted}}(x)$ 会被调整为：

$$P_{\text{adjusted}}(x) = \frac{P(x)}{(\text{len}(x))^{\alpha}}$$

其中：

- $P(x)$ 是未调整的原始概率。
- $\text{len}(x)$ 是当前生成文本的长度（包括当前 token）。
- α 是 `length_penalty`，即 2.0。

计算过程：

1. 当前文本长度 $\text{len}(x) = 5$（已经生成了 5 个 token）。
2. 生成第 6 个 token 时，原始概率 $P(x) = 0.1$。
3. 由于 `length_penalty = 2.0`，我们计算调整后的概率：

$$P_{\text{adjusted}}(x) = \frac{0.1}{5^{2.0}} = \frac{0.1}{25} = 0.004$$

因此，经过 `length_penalty` 调整后，第 6 个 token 的概率从原始的 0.1 被惩罚到 0.004，显著降低了生成该 token 的概率。这意味着模型倾向于生成较短的文本，避免过长的生成。

另一个例子，`length_penalty = 0.5`：

如果我们将 `length_penalty` 设置为 0.5，计算过程会如下：

$$P_{\text{adjusted}}(x) = \frac{0.1}{5^{0.5}} = \frac{0.1}{\sqrt{5}} \approx \frac{0.1}{2.236} \approx 0.045$$

在这种情况下，由于 `length_penalty` 较小，调整后的概率较高（0.045），模型更倾向于生成更多的 token，鼓励生成较长的文本。