大模型的解码策略是指在生成文本时，模型如何从可能的候选词中选择下一个词或 token。常见的解码策略包括贪心搜索（Greedy Search）、束搜索（Beam Search）、随机采样（Sampling）等。不同的解码策略会影响生成文本的质量、多样性和效率。

昇思学习打卡营·NLP特辑第四课

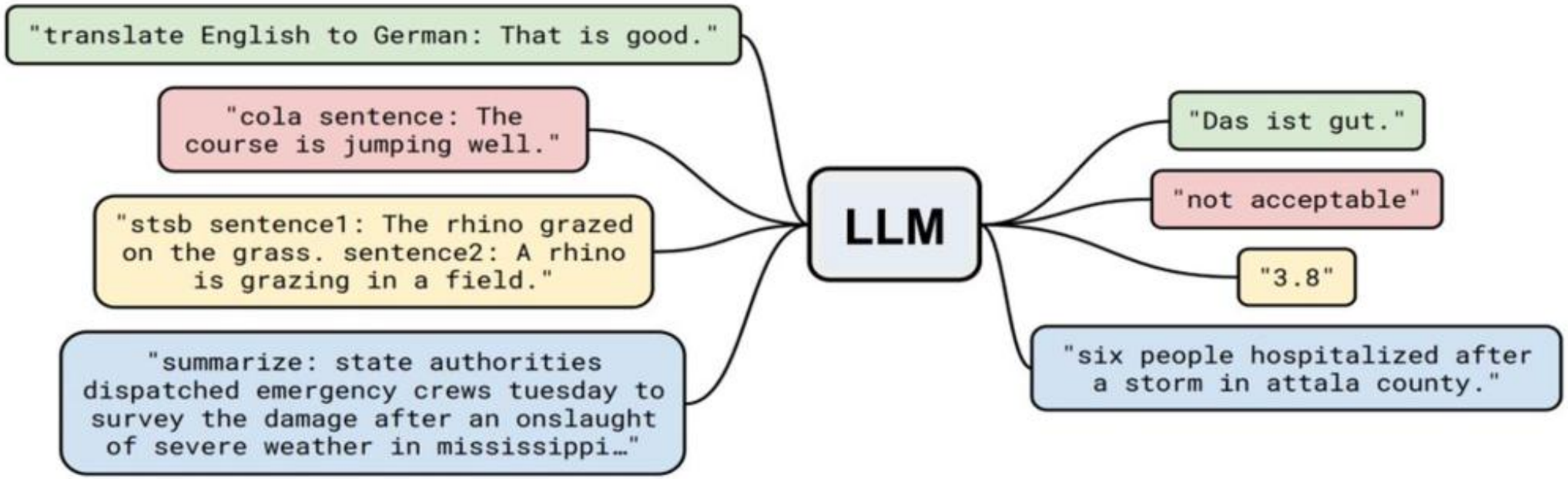
大模型Decoding+MindSpore NLP分布式推理详解

王洁怡

简介：大语言模型（LLM）在生成或推理时有不同的解码策略，解码策略的选择可能会影响推理过程的计算负载和资源分配，与并行推理结合使用可以提高效率。MindNLP0.4实现模型大量更新的同时，进一步支持多卡多进程并行推理，使得多种解码策略能够在合理的时间和硬件资源下实现。合理组合解码策略与分布式推理技术可以显著提高大语言模型的性能和输出质量，同时也能够适应不同的应用场景和资源条件。本次直播课将介绍三类经典的采样解码策略，利用MindNLP的分布式推理方法对其效果进行详细比较。

昇思 MindSpore

大语言模型有很强大泛化能力



大模型Decoding

- 思考：LLM模型文本生成过程：
  - 一段长句子作为输入，编码后也会得到多个token对应的embedding，那么哪个embedding用于预测下一个token?
  - 预测到下一个token的概率后，LLM模型是怎么生成完整回答的?
  - 为什么相同的输入会有不同的回答，这种随机性是如何实现的?

- 大语言模型的原始生成：

```
text = "say"
inputs = tokenizer(text, return_tensors="ms")
print(f"inputs:{inputs}")
```

```
inputs:{'input_ids': Tensor(shape=[1, 2], dtype=Int64, value=
[[128000,  37890]]), 'attention_mask': Tensor(shape=[1, 2], dtype=Int64, value=
[[1, 1]])}
```

大模型Decoding

- 大语言模型的原始生成：

```
text = "say"
inputs = tokenizer(text, return_tensors="ms")
print(f"inputs:{inputs}")
```

```
inputs:{'input_ids': Tensor(shape=[1, 2], dtype=Int64, value=
[[128000,  37890]]), 'attention_mask': Tensor(shape=[1, 2], dtype=Int64, value=
[[1, 1]])}
```

```
[13] input_ids = inputs["input_ids"].to(mindspore.int32)
```

```
[14] logits = model.forward(input_ids)
print("Logits Shape:", logits.logits.shape)
print(f"logits:{logits.logits}")
```

```
its Shape: (1, 2, 128256)
its:[[[ 4.8867188  6.0546875 10.78125   ... -3.6152344 -3.6152344 -3.6132812]
 4.3359375  4.4414062  3.6054688 ... -5.28125   -5.28125   -5.28125  ]]]
```

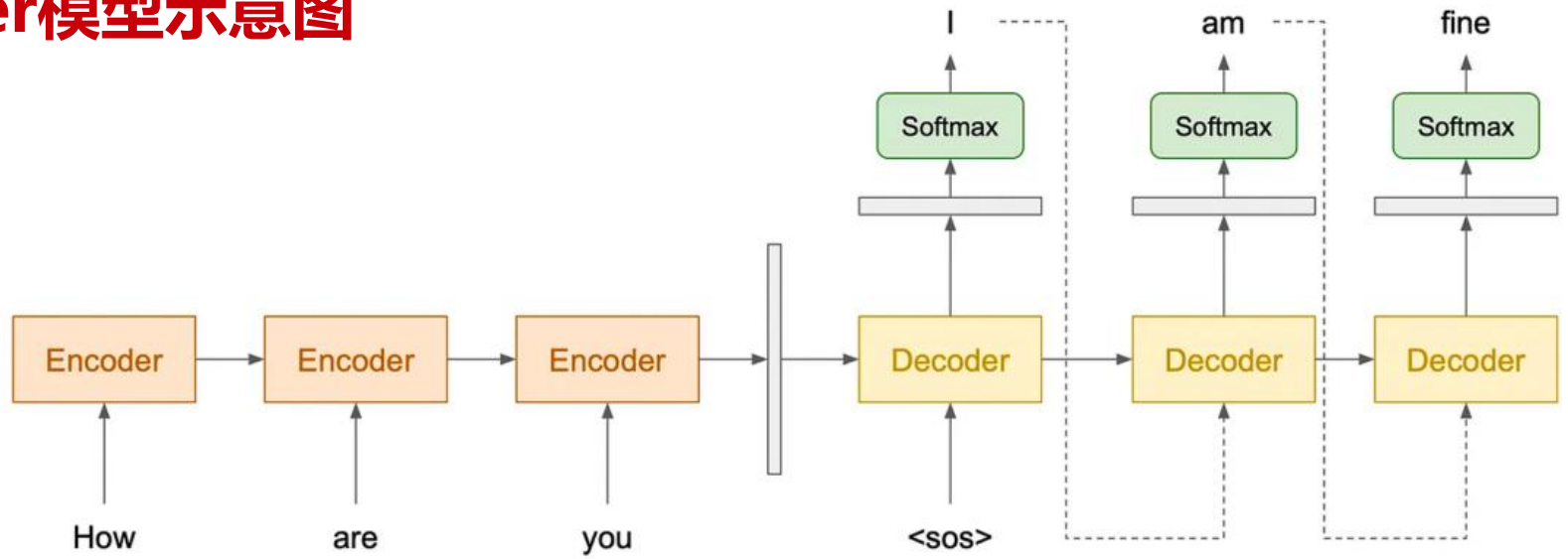
```
[17] next_token = mindspore.ops.argmax(logits.logits, dim=-1).reshape(-1)[1]
print(f"next_token:{next_token}")
```

```
next_token:24748
```

```
[18] next_word = tokenizer.decode(next_token)
print(f"next_word:{next_word}")
```

```
next_word: hello
```

Transformer模型示意图



- 要生成文本，通常是通过一次生成一个token来自回归方式完成的。如果没有适当的技术策略，生成的响应可能非常普通且没有智能味。

解码策略的softmax函数数学表示为：

$$P(x_i|x_{1:i-1}) = \frac{\exp(u_i)}{\sum_j \exp(u_j)}$$

△

Softmax 函数：x 是时间步进为 i 的token，u 是词汇表中每个token值的向量

这个公式描述了在解码过程中使用softmax函数来计算下一个token的概率分布。具体来说，公式如下：

$P(x_i|x_{1:i-1}) = \frac{\exp(u_i)}{\sum_j \exp(u_j)}$

公式解释

1. $P(x_i|x_{1:i-1})$:
   - 这表示在给定前 $i-1$ 个token的情况下，第 i 个token为 x_i 的概率。
   - $x_{1:i-1}$ 表示从第一个token到第 $i-1$ 个token的序列。
2. u_i:
   - 这是词汇表中每个token对应的未归一化得分（logits）。
   - u_i 是模型预测出的第 i 个token的原始得分。
3. $\exp(u_i)$:
   - 这是对 u_i 应用指数函数的结果，用于将负无穷到正无穷的实数映射到正区间内。
4. $\sum_j \exp(u_j)$:
   - 这是对所有可能token的指数得分求和，确保概率分布的总和为1。

昇思 MindSpore

回答之前的疑问

- 1. “say goodbye”作为输入，编码后也会得到四个token对应的embedding。那么哪个embedding用于预测下一个token?
- 答：根据上文可知，LLM模型在推理时，根据当前输入序列最后一个token对应的embedding预测下一个token。因为Transformer模型结构使用masked attention，每个token在编码时都只会和他前面的token交互。那么最后一个token自然就包括了当前序列的所有信息，因此用于预测下一个token是最合理的。

- 2. 预测到下一个token的概率后，LLM模型是怎么生成完整回答的？为什么相同的输入会有不同的回答？这种随机性是如何实现的?
- 答：LLM模型不是直接使用贪心解码策略(greedy decoding)，即选择概率最大的那个token作为输出。因为使用贪心解码策略，对于相同的输入序列LLM模型每次都会给出一样的回答，推理模式下所有参数都固定，不存在随机性。

- LLM模型回答的文本多样性就需要一些其他的生成策略。

- Huggingface库常用的一些超参数传递给模型，来执行不同的文本生成策略，达到模型文本多样性和随机性生成的目的。

LLM推理参数

- temperature参数用于控制生成文本的随机性和多样性，调整模型输出的logits张量的概率分布。

```
# temperature
import mindspore
from mindspore import Tensor
import numpy as np
import mindspore.ops as ops

logits = Tensor(np.array([[0.5, 1.2, -1.0, 0.1]]), mindspore.float32)

probs = ops.softmax(logits, axis=-1)
# temperature low 0.5
probs_low = ops.softmax(logits / 0.5, axis=-1)
# temperature high 2
probs_high = ops.softmax(logits / 2, axis=-1)

print(f"probs:{probs}")
print(f"probs_low:{probs_low}")
print(f"probs_high:{probs_high}")
```

```
probs:[[0.25593758 0.515395   0.05710739 0.1715601 ]]
probs_low:[[0.18004017 0.7300989  0.00896367 0.08089726]]
probs_high:[[0.26952982 0.382481   0.12731686 0.22067234]]
```

LLM推理参数

- top-k参数，用于生成下一个token时，限制LLM模型只能考虑前k个概率最高的token，可以降低模型生成无意义或重复的输出的概率，同时提高模型的生成速度和效率。当top_k为2时，只取概率值前2的参与采样。

```
原始logits张量:tensor([[3., 4., 2., 5.]])
经softmax计算后概率值分布:tensor([[0.0871, 0.2369, 0.0321, 0.6439]])
当top_k值为:2时，最终概率值分布为:tensor([[  -inf, 0.2369,   -inf, 0.6439]])
```

对概率进行排序，然后将位置第2之后的概率转换成0，再替换成-inf。

- top-p也是一个用于控制生成文本多样性的参数。这个参数的全名是"top probability"，通常用一个介于 0 到 1 之间的值来表示，在生成下一个token时，概率分布中选择的最高概率累积阈值。

```
原始logits张量:tensor([[3., 4., 2., 5.]])
经softmax计算后概率值分布:tensor([[0.0871, 0.2369, 0.0321, 0.6439]])
降序后概率值分布:tensor([[0.6439, 0.2369, 0.0871, 0.0321]])
降序后概率值对应的ID:tensor([[3, 1, 0, 2]])
降序后概率值的累积和分布:tensor([[0.6439, 0.8808, 0.9679, 1.0000]])
当top_p值为:0.9时，最终概率值分布为:tensor([[0.0871, 0.2369,   -inf, 0.6439]])
```

https://github.com/Glanvery/LLM-Travel/blob/main/generate_parameter.ipynb

解码策略

- 大语言模型（LLM）在生成或推理时有不同的解码策略，解码策略的选择可能会影响推理过程的计算负载和资源分配，与并行推理结合使用可以提高效率。

LM也可以看做生成模型，99%的LM属于自回归(auto-regression)生成模型，先来看一个概率分解的数学式子，假设词序列是$w_{1:T}$：

$$P(w_{1:T}|W_0) = \sum_{t=1}^{T} P(w_t|w_{1:t-1}, W_0), with\ w_{1:0} = \emptyset$$

W_0是prompt。T是LM生成的response 长度。

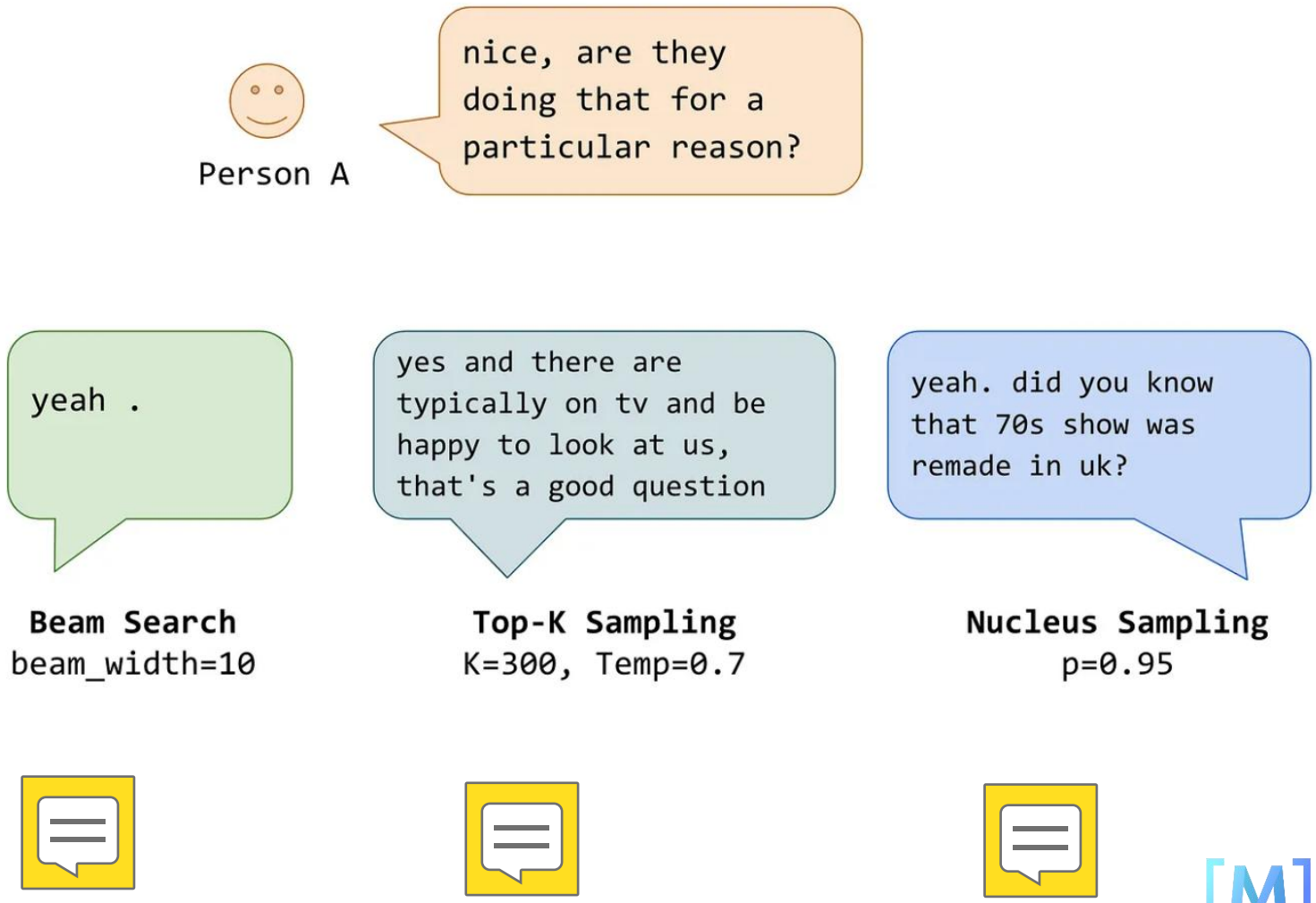
点击图片跳转详解

LM就是按照上面的分解式子，一个个的生成token (w_t).

- 目前解码方法主要分为三类：
  - Greedy Search (贪心)
  - Beam Search
  - Sampling (采样)

解码策略

- 目前解码方法主要分为三类：
  - Greedy Search (贪心)
  - Beam Search
  - Sampling (采样)



https://towardsdatascience.com/decoding-strategies-that-you-need-to-know-for-response-generation-ba95ee0faadc

LLM Decoding

- 在解码过程中的每个时间步，获取向量（保存从一步到另一步的信息）并将其与 softmax 函数一起应用以将其转换为每个单词的概率数组。
  - Greedy: Select the best probable token at a time | 贪心：一次选择最可能的标记
  - Beam Search: Select the best probable response | 波束搜索：选择最可能的响应
  - Random Sampling: Random based on probability | 随机抽样：基于概率的随机
  - Temperature: Shrink or enlarge probabilities | 温度：缩小或扩大概率
  - Top-K Sampling: Select top probable K tokens | Top-K 采样：选择最可能的 K 个 token
  - Nucleus Sampling: Dynamically choose the number of K (sort of) | 核采样：动态选择K的数量（类似）

贪心解码策略 (greedy decoding)

输入文本：The chicken crossed the

Token Embedding Layer Lookup：将 The, chicken, crossed, the 转换为对应的词向量。

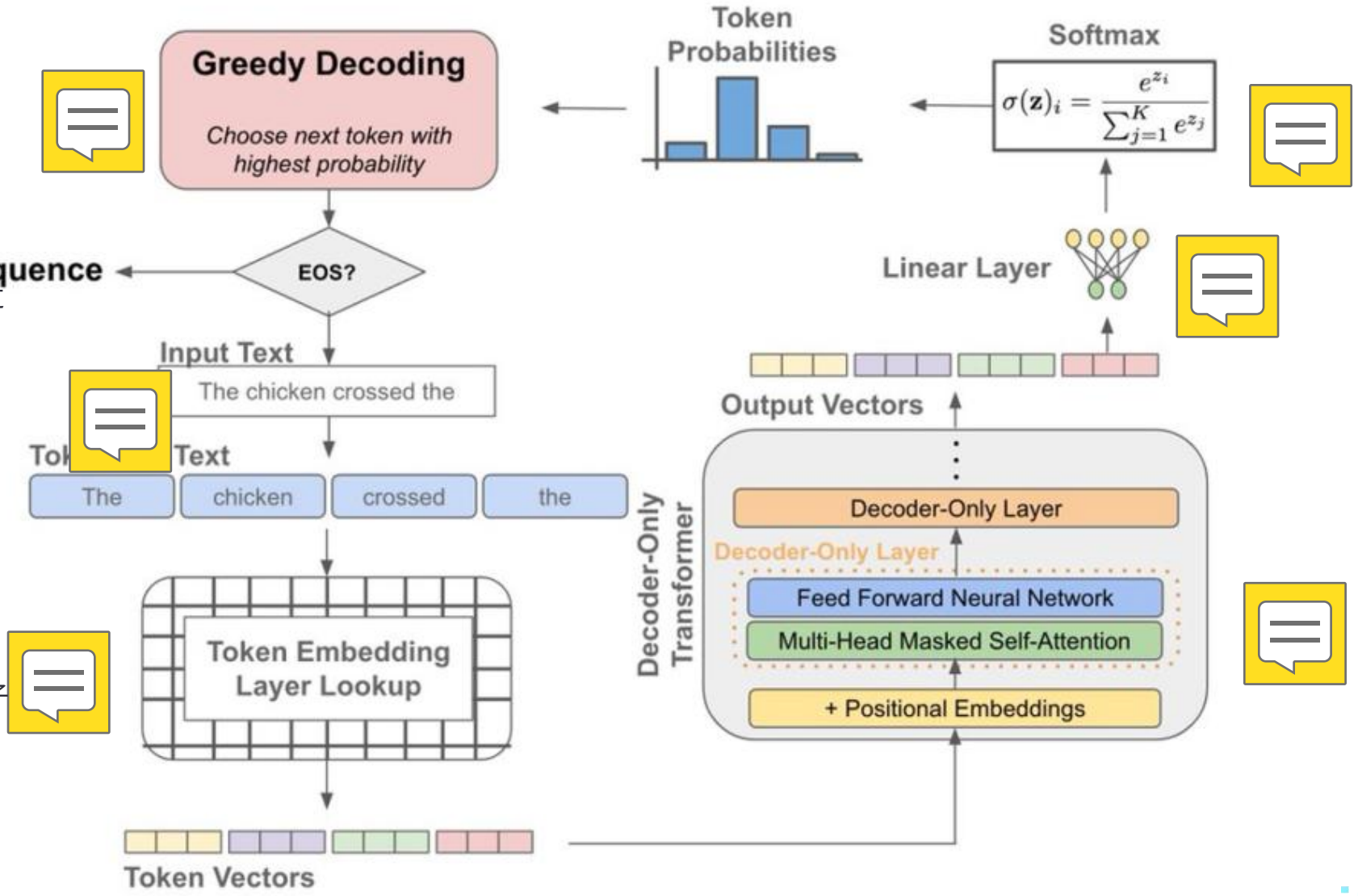
Decoder-Only Transformer：词向量通过多层解码器进行处理。输出向量：解码器输出的向量传递到线性层。

Softmax Layer：线性层输出的向量通过 Softmax 层，得到每个词的概率分布。

Greedy Decoding：选择概率最高的词作为下一个词。重复上述步骤直到生成完整的句子或达到 EOS 标记。

假设初始输入为 The chicken crossed the，经过处理后，模型预测下一个词的概率分布如下：road: 0.7street: 0.2way: 0.1 根据贪心解码策略，模型会选择概率最高的词 road 作为下一个词，生成的句子为 The chicken crossed the road。

贪心解码策略是一种简单且高效的解码方法，它在每一步都选择当前时间步上概率最高的词。



贪心解码策略 (greedy decoding)

model.generate函数详解跳转

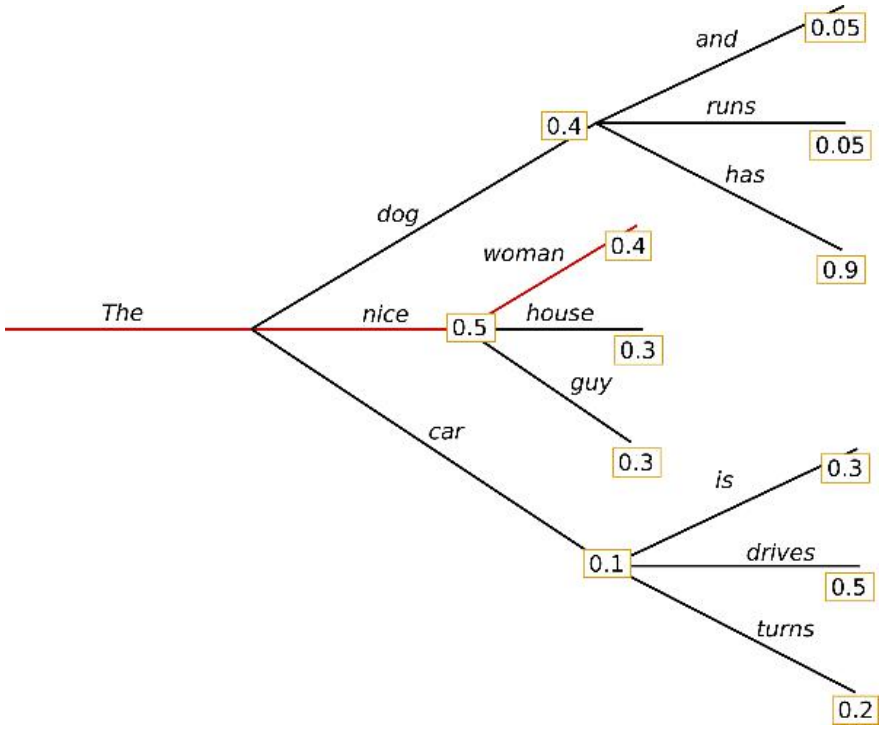
- 最经典最原始的贪心解码策略，在 model.generate() 中，当 num_beams 等于 1 且 do_sample 等于 False 时进入此模式，也可以直接使用 model.greedy_search()，这个解码策略很简单，就是在每一步中选择预测概率最高的 token 作为下一个 token，从而生成文本，和之前的 forword 是一样的，这种方法通常会导致生成的文本比较单一和局部最优。注意此策略不能使用 temperature，top_k，top_p 等改变 logits 的参数。

$$w_t = argmax_w \, P(w|w_{1:t-1})$$

- 红色线表示贪心算法的解码结果，

  生成序列 "The nice woman"

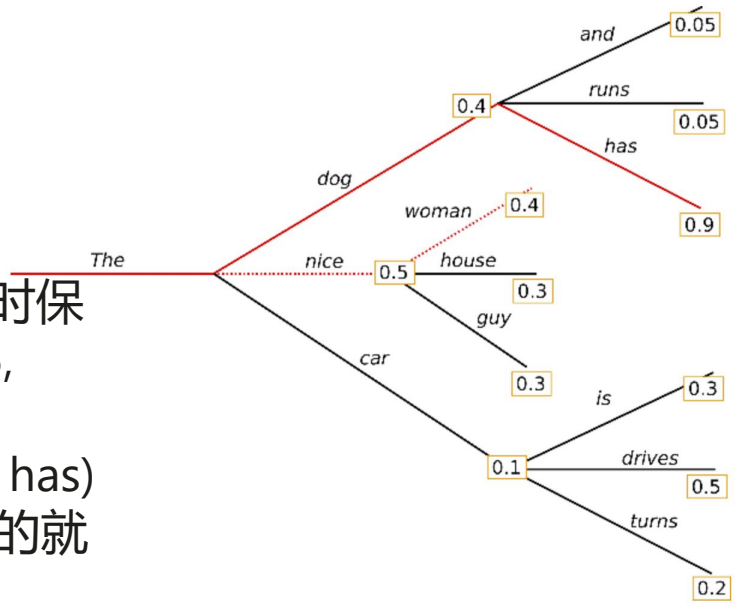
- 贪心算法自然有缺点，只考虑当前的概率最大，会忽略"has"这种大概率的词，因为"has"在小概率词"dog"后面，可以认为贪心算法没有考虑到“词组”的情况，毕竟只是单个词往外蹦。



昇思 MindSpore

束搜索策略（Beam Search）

- beam 在物理领域有光束的含义，简单理解为“多个”就行。他总是保留当前概率最大的num_beams个序列（注意不是词，考虑了词组、短语），假设num_beams=2。
  - num_beams参数是用于束搜索策略(beam search)，控制生成的候选句子的数量，该参数控制的是每个生成步要保留的生成结果的数量，用于在生成过程中增加多样性或生成多个可能的结果。
- 束搜索策略(beam search)本质上也是一个贪心解码策略 (greedy decoding)，所以无法保证一定可以得到最好的结果。步骤如下：
  1. 在每个生成步骤中，都保留着 top K(K 即 num_beams 参数的值) 个可能的候选单词。
  2. 这 K 个单词都做下一步 decoding，分别选出 top K，然后对这 K^2 个候选句子再保留前 K 个最可能的结果。
  3. 如果已经生成了结束符，则将其对应的结果保留下来。
  4. 重复上述过程直到生成所有的结果或达到最大长度。



"The"后面有三个词(token): "dog"(0.4), "nice" (0.5), "car" (0.1)。 此时保留概率最大的两个词，也就是"nice", "dog"。解析来从(dog, and)=0.4*0.05, (dog, runs)=0.4*0.05, (dog, has)=0.4*0.9, (nice, woman)=0.5*0.4, (nice, house)=0.5*0.3, (nice, guy)=0.5*0.3 中找两个概率最大的。答案就是 (dog, has)和(nice, woman). 接下来继续生成下面的token，如果此时停止，那么生成的就是概率最大的(The, dog, has).

束搜索策略（Beam Search）

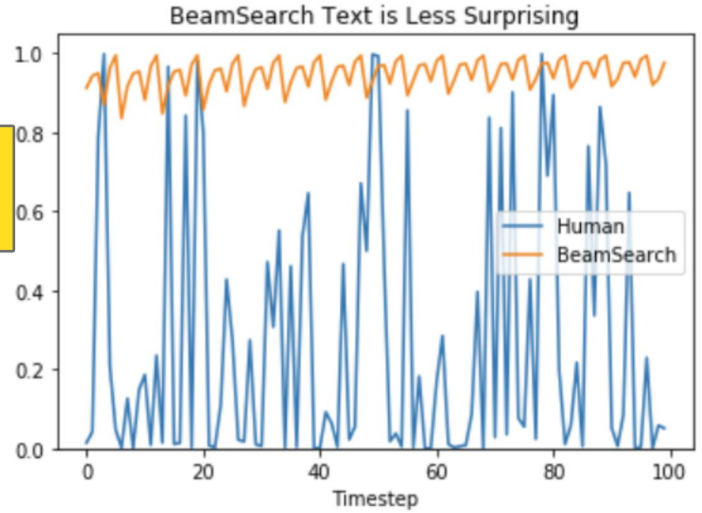
model.generate()讲解 跳转

```python
# activate beam search and early_stopping
# 设置num_beams和early_stopping来激活beam search
beam_output = model.generate(
    **model_inputs,
    max_new_tokens=40,
    num_beams=5,
    early_stopping=True
)

print("Output:\n" + 100 * '-')
print(tokenizer.decode(beam_output[0], skip_special_tokens=True))
```

```python
# set no_repeat_ngram_size to 2
# no 2-gram appears twice
beam_output = model.generate(
    **model_inputs,
    max_new_tokens=40,
    num_beams=5,
    no_repeat_ngram_size=2,
    early_stopping=True
)

print("Output:\n" + 100 * '-')
print(tokenizer.decode(beam_output[0], skip_special_tokens=True))
```

- 如果beam search生成的response也出现了复读机（重复）现象，"I'm not sure if I'll ever be able to walk with him again. I'm not sure"，可以通过设置不要出现重复词组(n-gram)来避免

- beam search也有缺点：
  - 适合于短文/短句生成（句子级机器翻译），不适合long response
  - beam search的复读机现象也很严重，而不重复n-gram的方法又太粗暴了
  - beam search本质上还是取top n的高概率，生成的response很难给人surprise



昇思 MindSpore

束搜索策略（Beam Search）

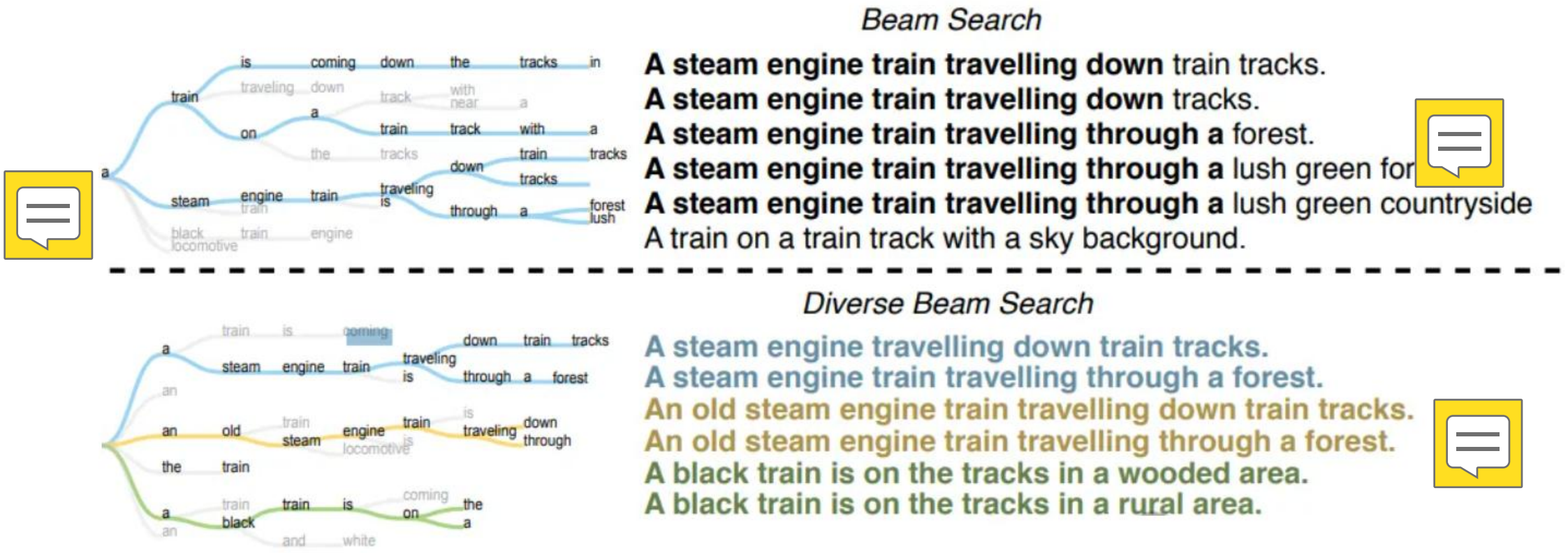
```
model_inputs = tokenizer('I enjoy walking with my cute dog', return_tensors="ms")
```

```
beam_outputs = model.generate(
    **model_inputs,
    max_new_tokens=40,
    num_beams=5,
    no_repeat_ngram_size=2,
    num_return_sequences=5,
    early_stopping=True
)

# now we have 3 output sequences
print("Output:\n" + 100 * '-')
for i, beam_output in enumerate(beam_outputs):
  print("{}: {}".format(i, tokenizer.decode(beam_output, skip_special_tokens=True)))
```

```
Output:
----------------------------------------------------------------------------------------------------
0: I enjoy walking with my cute dog, Luna, in the nearby woods. It's a great way to clear my mind and get some exercise. The woods a
re beautiful this time of year, with the leaves changing colors and the smell of
1: I enjoy walking with my cute dog, Luna, in the nearby woods. It's a great way to clear my mind and get some exercise. The woods a
re beautiful this time of year, with the leaves changing colors and the sound of
2: I enjoy walking with my cute dog, Luna, in the nearby woods. It's a great way to clear my mind and get some exercise. The woods a
re beautiful this time of year, with the leaves changing colors and the scent of
3: I enjoy walking with my cute dog, Luna, in the nearby woods. It's a great way to clear my mind and get some exercise. The woods a
re beautiful this time of year, with the leaves changing colors and the air filled
4: I enjoy walking with my cute dog, Luna, in the nearby woods. It's a great way to clear my mind and get some exercise. We like to
explore the trails and see what kind of interesting things we can find. Sometimes
```

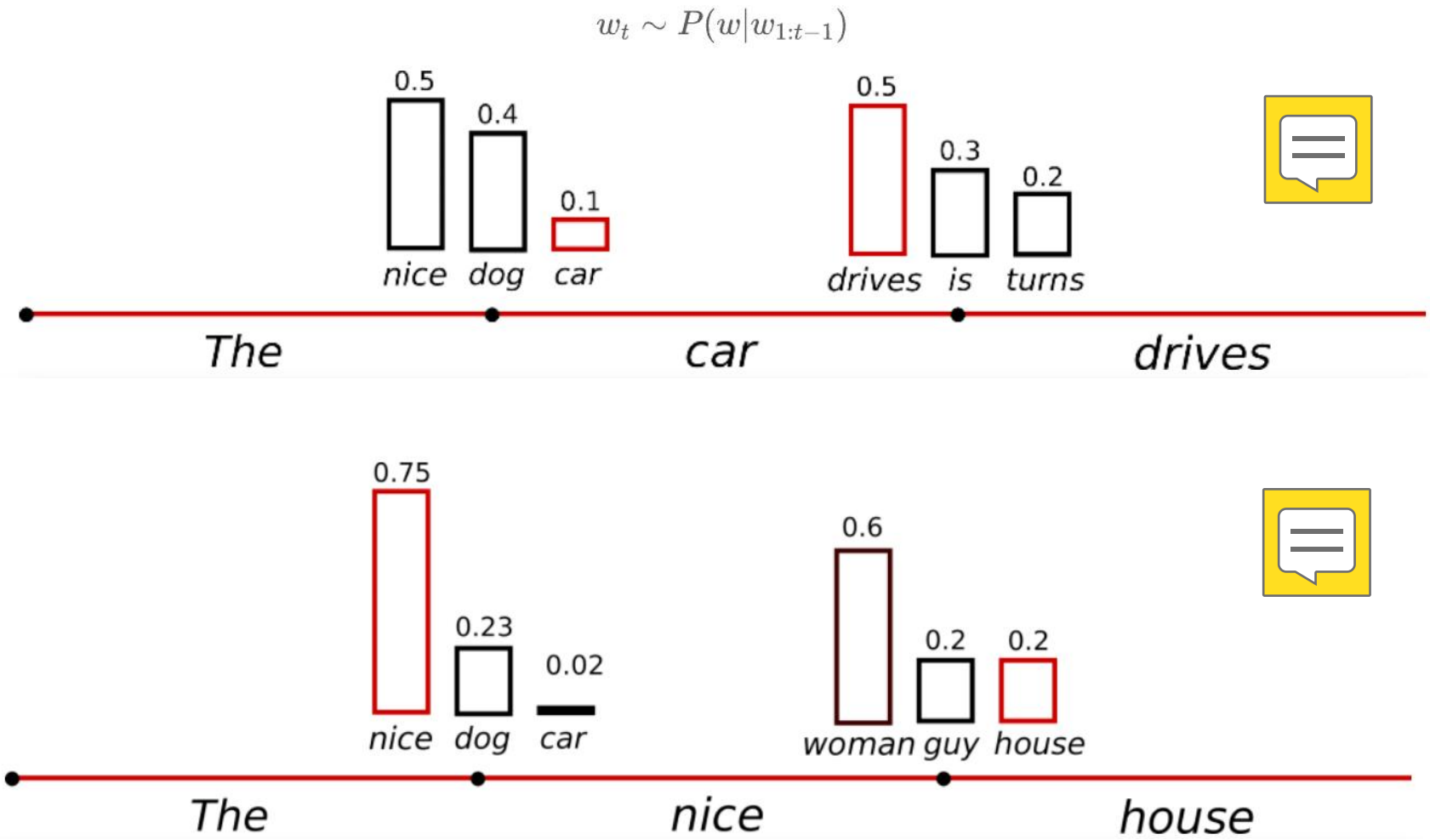
beam search改进-Diverse Beam Search (DBS)

- 束搜索(beam search)生成的结果还是会过于相似的，DBS做了一些改进，核心就是分组机制。
- 例如，当 num_beams=2，num_beam_groups=2，分成2个组，每个组里的beam可以相似，但组和组之间要有足够的多样性。



Sampling

最简单的采样方法，根据概率$P(w|w_{1:t-1})$来做离散采样咯



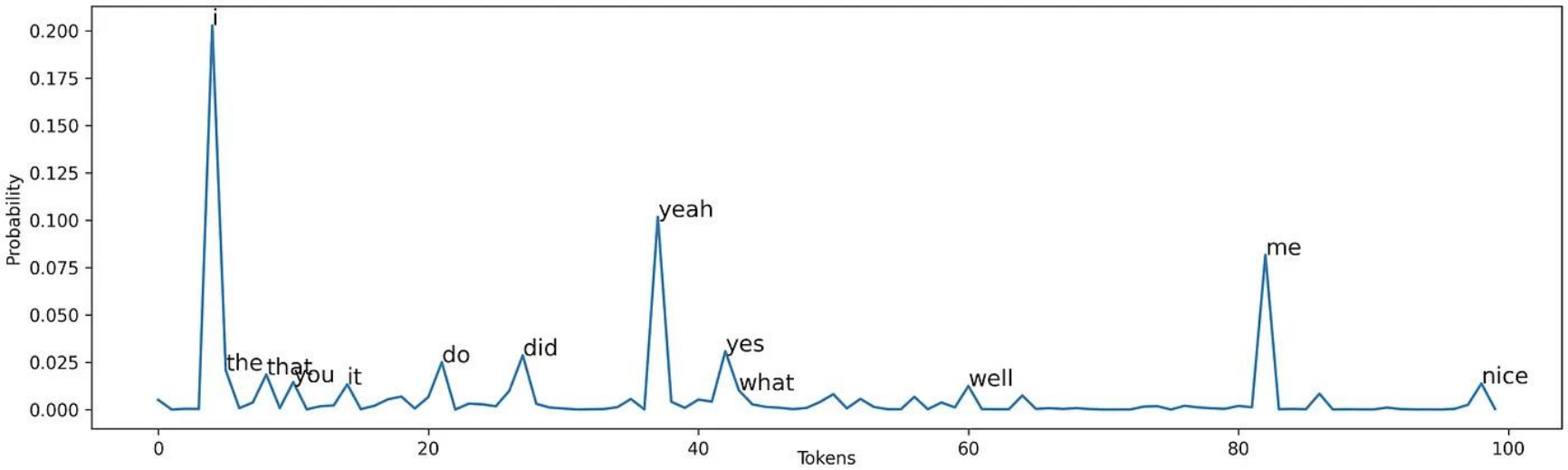
I enjoy walking with my cute dog for the rest of the day, but this had me staying in an unusual room and not going on nights out with friends (which will always be wondered for a mere minute or so at this point).

可以通过温度(temperate)参赛修改P，让原本高概率的概率更高，低概率的概率更低，这样就可以尽量采样出高概率的词，在随机性和surprise之间做trade-off。注意temperate越大越随机，如果temperate=0，就没有随机了，就是贪心算法。

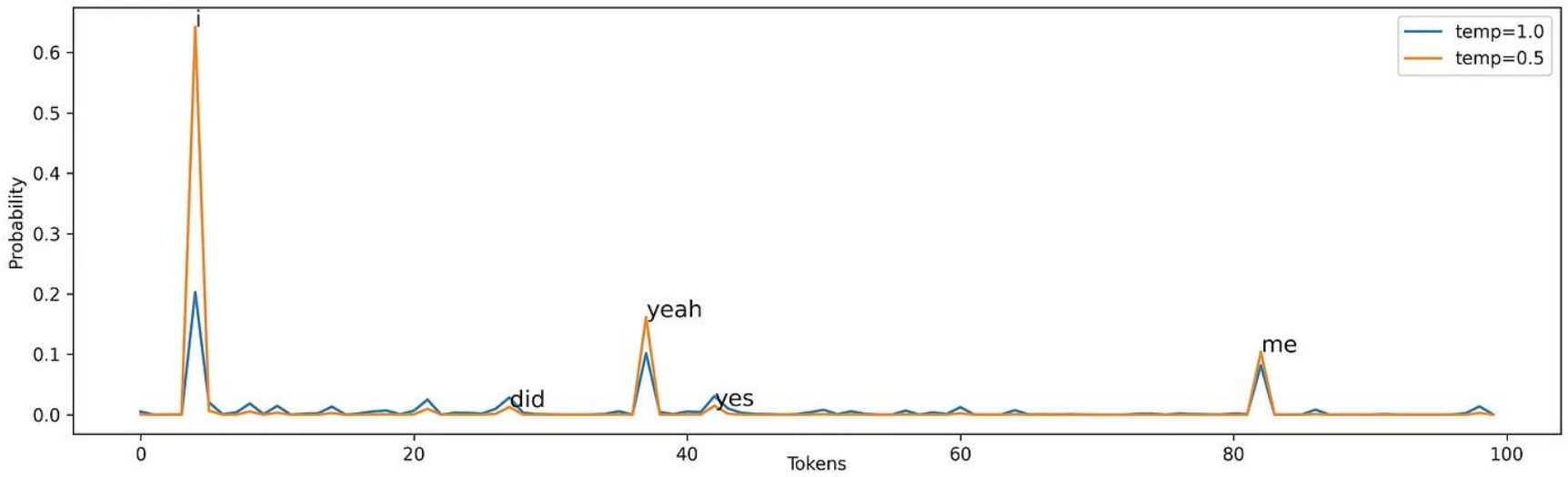
Random Sampling & Temperature Sampling

假设我们正在生成上下文“*I love watching movies*”的第一个标记，下图显示了第一个单词应该是什么的概率。

Random Sampling



Temperature Sampling



Temperature Sampling

```
messages = [
    {"role": "system", "content": "You are a psychological counsellor, who is good at emotional comfort."},
    {"role": "user", "content": "I don't sleep well for a long time."},
]

input_ids = tokenizer.apply_chat_template(
    messages,
    add_generation_prompt=True,
    return_tensors="ms"
)
```

[10]
```
outputs = model.generate(
    input_ids,
    max_new_tokens=20,
    eos_token_id=terminators,
    do_sample=True,
    temperature=0.6
)
response = outputs[0][input_ids.shape[-1]:]
print(tokenizer.decode(response, skip_special_tokens=True))
```

```
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your inpu
t's `attention_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:128009 for open-end generation.
Sweetheart, I'm so sorry to hear that you're struggling with sleep. It can be such
```

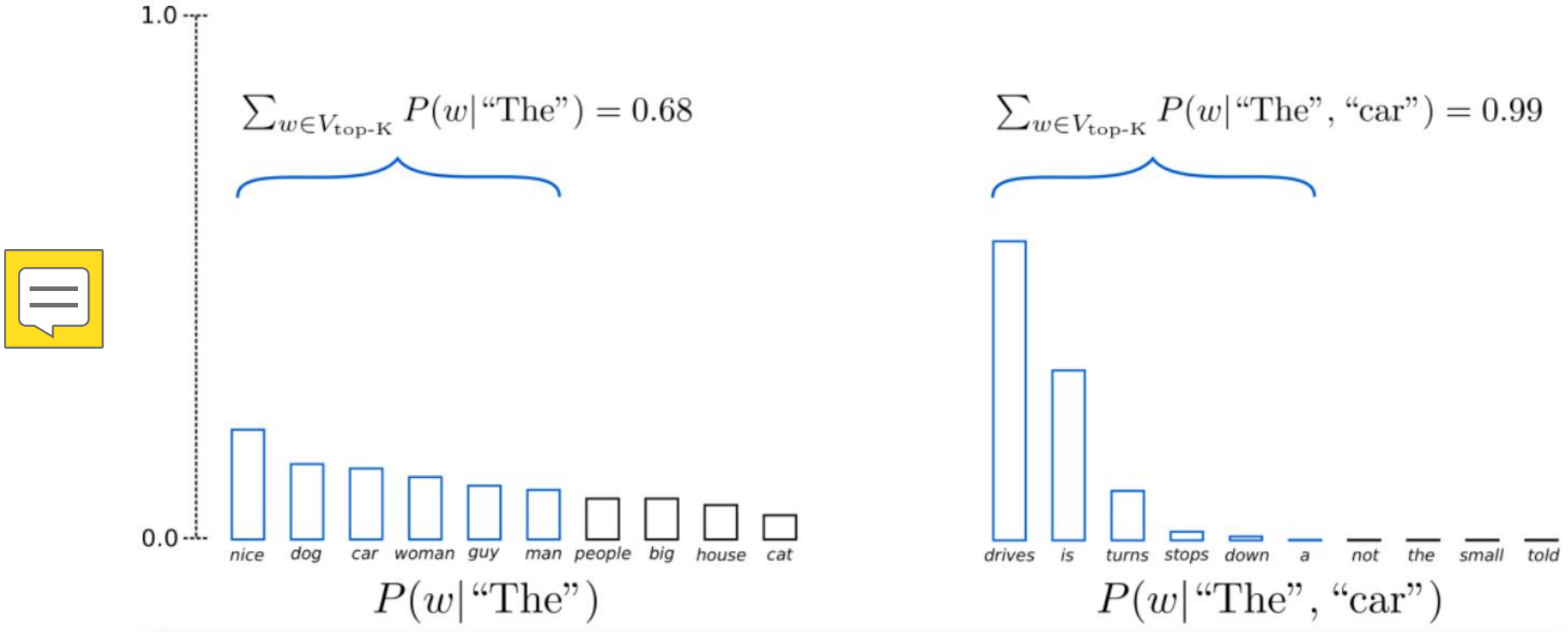
```
outputs = model.generate(
    input_ids,
    max_new_tokens=20,
    eos_token_id=terminators,
    do_sample=True,
    temperature=0.6
)
response = outputs[0][input_ids.shape[-1]:]
print(tokenizer.decode(response, skip_special_tokens=True))
```

```
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your inpu
t's `attention_mask` to obtain reliable results.
Setting `pad_token_id` to `eos_token_id`:128009 for open-end generation.
I'm so sorry to hear that you're struggling with sleep. It can be really frustrating and affect
```

设置 *temperature=0.6*: 更少的*weird n-grams*，同时输出更连贯了

top-K Sampling（GPT2）

- top-K做法很简单，从概率最大的K个token中采样，避免稀奇古怪的输出。

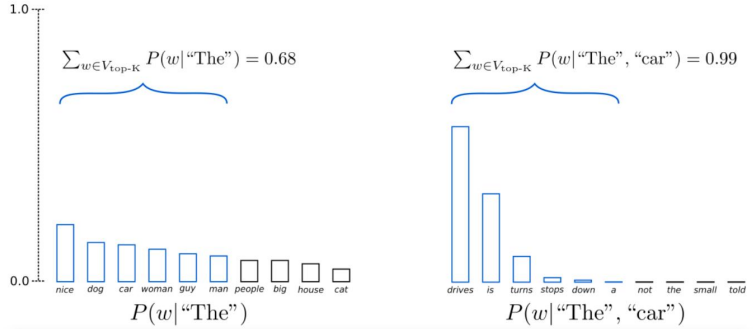


设置K= 6后，在两个采样步骤中，我们将采样池限制为6个单词。
成功地在第二采样步骤中消除了相当奇怪的候选者weird candidates（“not”，“the”，“small”，“told”）。

Top-K Sampling

```
model_inputs = tokenizer('I enjoy walking with my cute dog', return_tensors="ms")
```

- deactivate Top-K sampling




```
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog, reading, and spending time with my loved ones.
I'm a bit of a perfectionist, so I can get stressed when things don't go according to plan. But I'm working on being
```

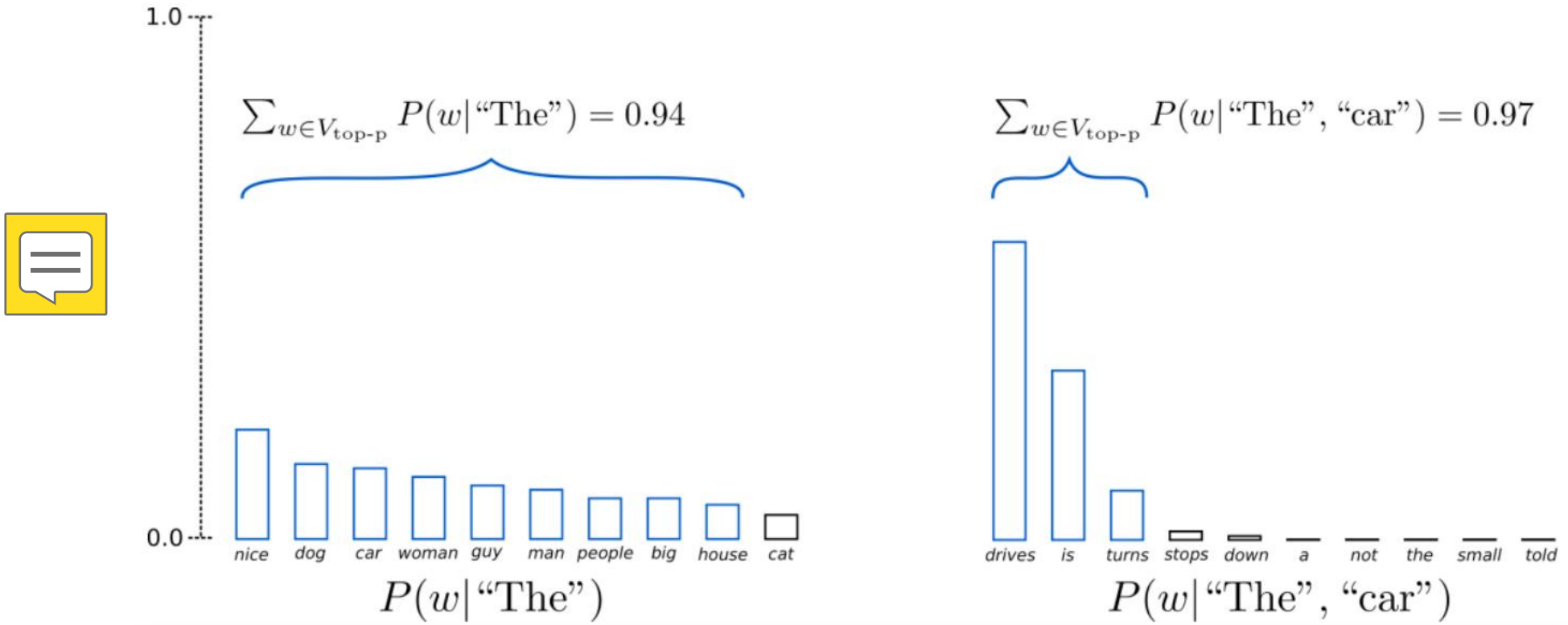
- top_k=50，得到目前为止最人性化human-sounding的文本:

```
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog, playing with my cat, and taking long baths with a good book. I'm a bit of a introvert and I prefer
quiet, low-key activities. I'm also a bit of a perfection
```

Top-K抽样的一个问题是，它不能动态地调整从下一个单词概率分布 $P(w|w_{1:t-1})$ 中过滤出来的单词的数量。这可能会产生问题，因为有些单词可能来自一个非常陡峭的分布，而其他单词可能来自一个更平坦的分布；在右上角的案例中，第一步的top-6是前68%，而第二步达到99%，囊括了几乎所有的单词。

Top-p (nucleus) Sampling

- 如果K不固定而是每一次自适应设置，是不是更合理呢？这就是Top-p (nucleus) sampling.想法也很简单，我们不从概率1（完整词表）中采样，而是从一个概率池子（可以联系二八定律，20%的词贡献了80%的概率）中采样，设置累计概率阈值，比如p=0.92, 至于p中包含了多少个词，是根据t情况而变化的，这就是自适应。



Top-p Sampling

- activate Top-p sampling by setting 0 < top_p < 1

```
# top_p
sample_output = model.generate(
    **model_inputs,
    max_new_tokens=40,
    do_sample=True,
    top_p=0.92,
    top_k=0
)

print("Output:\n" + 100 * '-')
print(tokenizer.decode(sample_output[0], skip_special_tokens=True))
```

```
Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog, reading, and trying out new recipes in the kitchen. I'm a bit of a foodie and love trying new rest
aurants and cuisines. I'm also a bit of a movie buff and enjoy
```

Top-P 的一般设置
常见设置:
通常，Top-P 的值在0.9到0.95之间。这个范围可以确保模型生成的文本既具有多样性又保持连贯性。
不同情景下的调整:
需要更多多样性时:
如果希望生成的文本更加多样化，可以将Top-P的值设置得稍低一些（如0.85或0.9）。
需要更高连贯性时:
如果希望生成的文本更加连贯和合理，可以将Top-P的值设置得稍高一些（如0.95或0.98）。

计算资源和效率的影响
计算资源:
Top-P采样相对于Top-K采样可能会稍微增加计算资源的需求，因为每次都需要重新计算累积概率。
然而，这种增加通常是微小的，尤其是在现代硬件上。
效率:
Top-P采样的效率与Top-K采样相当，因为它只是改变了选择词的方式，而不是增加了额外的计算步骤。
实际上，由于Top-P采样能够更好地适应不同的概率分布，它可能会提高生成文本的质量，从而间接提高整体效率。

- 对比Top_K Sampling效果

```
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog, playing with my cat, and taking long baths with a good book. I'm a bit of a introvert and I prefer
quiet, low-key activities. I'm also a bit of a perfection
```

Top-p Sampling

- For multiple independently sampled outputs, set the parameter num_return_sequences > 1

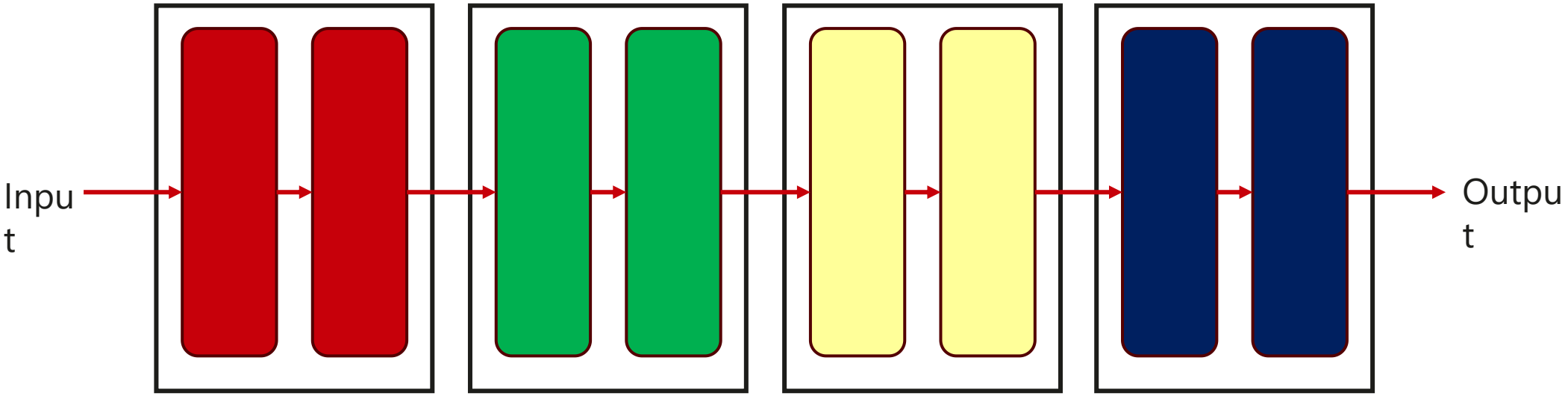
```
3] sample_outputs = model.generate(
    **model_inputs,
    max_new_tokens=40,
    do_sample=True,
    top_k=50,
    top_p=0.95,
    num_return_sequences=3,
)

print("Output:\n" + 100 * '-')
for i, sample_output in enumerate(sample_outputs):
  print("{}: {}".format(i, tokenizer.decode(sample_output, skip_special_tokens=True)))
```

```
Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.
Output:
----------------------------------------------------------------------------------------------------
0: I enjoy walking with my cute dog, playing video games, and learning new things.
I'm a bit of a perfectionist and can be quite critical of myself. I also have a hard time saying no to people and can get overwhelme
d
1: I enjoy walking with my cute dog, playing the guitar, and trying out new recipes in the kitchen.
I'm a bit of a perfectionist, but I'm also a big fan of taking risks and trying new things. I love
2: I enjoy walking with my cute dog, Max. He loves going on adventures and sniffing all the interesting smells. I also like reading
books and watching TV shows. I'm a bit of a foodie, so I love trying new
```

不同Decoding策略对比

- 1. **贪婪解码（Greedy Decoding）**：
- 当需要快速生成文本且对生成质量要求不是特别高时，贪婪解码是一个简单且计算效率高的选择。它选择具有最大logit值的token作为下一个输出，适用于需要快速响应的场景，如聊天机器人的初步响应生成。
- 2. **束搜索（Beam Search）**：
- 适用于需要精确控制输出质量的场景，如机器翻译或问答系统。束搜索通过考虑多个候选序列来生成文本，可以提高翻译的准确性和流畅性。
- 3. **抽样解码（Sampling Decoding）**：
- 适用于需要多样性输出的场景，如创意写作或开放性问题的回答。抽样解码从词汇表中根据概率分布选择token，可以通过调整参数如温度（Temperature）来控制随机性。
  - **Top-K和Top-P**：
- 适用于需要控制输出长度和提高生成质量的场景。Top-K和Top-P通过限制候选token的数量来提高生成的连贯性和减少重复，适用于需要高质量输出的任务。
  - **温度采样（Temperature Sampling）**：
- 适用于需要在生成过程中增加随机性的场景，如创意写作或探索性任务。温度参数可以调整输出的随机度，较低的温度值会使采样更接近确定性解码，而较高的温度值则增加随机性。

MindNLP并行推理——多进程多卡



1. 创建空模型
2. 计算每卡放置的层
3. 插入send/receive
   a) 重写forward， 重新构造一个切分后的模型
   b) 把不需要的层替换为nn.Identity
4. 加载每卡需要的权重到模型
5. 避免除了卡0外多余的计算

MindNLP并行推理——多进程多卡

```python
import mindspore
from mindspore.communication import init
from mindnlp.transformers import AutoTokenizer, Auto rCausalLM

model_id = "LLM-Research/Meta-Llama-3-8B-Instruct"

init()
tokenizer = AutoTokenizer.from_pretrained(model_id, mirror='modelscope')
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    ms_dtype=mindspore.float16,
    mirror='modelscope',
    device_map="auto"
)
```

import mindspore：导入 MindSpore 库，这是一个开源的机器学习框架。
from mindspore.communication import init：导入用于分布式计算的初始化函数。
from mindnlp.transformers import AutoTokenizer, AutoModelForCausalLM：导入 MindNLP 中的 AutoTokenizer 和 AutoModelForCausalLM 类，这两个类分别用于自动加载预训练的分词器和模型。

设置模型的标识符，这里指定的是一个名为 Meta-Llama-3-8B-Instruct 的预训练模型。

init()：初始化分布式计算环境。如果代码运行在分布式环境中，这一步是必要的。

model = AutoModelForCausalLM.from_pretrained(...)：使用 AutoModelForCausalLM 类从预训练模型中加载因果语言模型。
model_id：模型的标识符。
ms_dtype=mindspore.float16：设置模型的数据类型为 float16，这可以减少内存占用并加速计算。
mirror='modelscope'：指定模型镜像源为 modelscope。
device_map="auto"：自动选择设备映射策略。这意味着模型会根据可用的硬件资源自动分配到合适的设备上（如 GPU 或 CPU）。

```
mpirun -n 2 python run_llama3_distributed.py
```

分布式并行启动方式

- 目前GPU、Ascend和CPU分别支持多种启动方式。主要有msrun、动态组网、 mpirun和rank table四种方式：

  - msrun： msrun 是动态组网的封装，允许用户使用单命令行指令在各节点拉起分布式任务，安装MindSpore后即可使用。此方式不依赖第三方库以及配置文件，具有容灾恢复功能，安全性较好，支持三种硬件平台。建议用户优先使用此种启动方式。

  - 动态组网：动态组网需要用户手动拉起多进程以及导出环境变量，是 msrun 的具体实现，Parameter Server训练模式建议使用此方式，其余分布式场景建议使用 msrun 。

  - mpirun：此方式依赖开源库OpenMPI，启动命令简单，多机需要保证两两之间免密登录，推荐有OpenMPI使用经验的用户使用此种启动方式。

  - rank table：此方式需要在Ascend硬件平台使用，不依赖第三方库。手动配置rank_table文件后，就可以通过脚本启动并行程序，多机脚本一致，方便批量部署。

    rank_table 启动方式将在MindSpore 2.4版本废弃。

msrun启动

- msrun是动态组网启动方式的封装，用户可使用msrun以单个命令行指令的方式在各节点拉起多进程分布式任务，并且无需手动设置动态组网环境变量。msrun同时支持Ascend，GPU和CPU后端。与动态组网启动方式一样，msrun无需依赖第三方库以及配置文件。
- 参数定义（网页）

- 脚本msrun_single.sh使用msrun指令在当前节点拉起1个Scheduler进程以及8个Worker进程（无需设置master_addr，默认为127.0.0.1；单机无需设置node_rank）：
- 执行指令：bash msrun_single.sh

即可执行单机8卡分布式训练任务，日志文件会保存到./msrun_log目录下，结果保存在./msrun_log/worker_*.log中

```
EXEC_PATH=$(pwd)
if [ ! -d "${EXEC_PATH}/MNIST_Data" ]; then
    if [ ! -f "${EXEC_PATH}/MNIST_Data.zip" ]; then
        wget http://mindspore-website.obs.cn-north-4.myhuaweicloud.com/notebook/
datasets/MNIST_Data.zip
    fi
    unzip MNIST_Data.zip
fi
export DATA_PATH=${EXEC_PATH}/MNIST_Data/train/

rm -rf msrun_log
mkdir msrun_log
echo "start training"

msrun --worker_num=8 --local_worker_num=8 --master_port=8118 --log_dir=msrun_log
 --join=True --cluster_time_out=300 net.py
```

EXEC_PATH=$(pwd)：获取当前工作目录的路径，并将其赋值给变量 EXEC_PATH

if [! -d "${EXEC_PATH}/MNIST_Data"]; then：检查当前工作目录下是否存在名为 MNIST_Data 的目录。
如果不存在，则进一步检查是否存在名为 MNIST_Data.zip 的压缩文件。
如果压缩文件也不存在，则从指定URL下载该文件。
使用 unzip 解压下载的文件。

将 MNIST_Data/train/ 目录的路径赋值给环境变量 DATA_PATH。

清理日志目录并创建新的日志目录：
使用 rm -rf 清除旧的日志目录 msrun_log。
使用 mkdir 创建新的日志目录 msrun_log。
输出一条信息表示开始训练。

mpirun启动

openMPI官网

- OpenMPI（Open Message Passing Interface）是一个开源的、高性能的消息传递编程库，用于并行计算和分布式内存计算，它通过在不同进程之间传递消息来实现并行计算，适用于许多科学计算和机器学习任务。使用OpenMPI进行并行训练是一种通用的在计算集群或多核机器上利用并行计算资源来加速训练过程的方法。OpenMPI在分布式训练的场景中，起到在Host侧同步数据以及进程间组网的功能。

- mpirun启动命令如下，其中DEVICE_NUM是所在机器的GPU数量：

  mpirun -n DEVICE_NUM python net.py

- mpirun还可以配置以下参数，更多配置可以参考mpirun文档：

  --output-filename log_output：将所有进程的日志信息保存到log_output目录下，不同卡上的日志会按rank_id分别保存在log_output/1/路径下对应的文件中。

  --merge-stderr-to-stdout：合并stderr到stdout的输出信息中。

  --allow-run-as-root：如果通过root用户执行脚本，则需要加上此参数。

  -mca orte_abort_on_non_zero_status 0：当一个子进程异常退出时，OpenMPI会默认abort所有的子进程，如果不想自动abort子进程，可以加上此参数。

  -bind-to none：OpenMPI会默认给拉起的子进程指定可用的CPU核数，如果不想限制进程使用的核数，可以加上此参数。

实操

```
model_id = "/home/ma-user/work/models/llama/LLM-Research/Meta-Llama-3-8B-Instruct"
#models/llama/LLM-Research/Meta-Llama-3-8B-Instruct
init()
tokenizer = AutoTokenizer.from_pretrained(model_id) #, mirror='modelscope')
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    ms_dtype=mindspore.float16,
    # mirror='modelscope',
    device_map="auto"
)

messages = [
    {"role": "system", "content": "You are a pirate chatbot who always responds in pirate speak!"},
    {"role": "user", "content": "Who are you?"},
]

input_ids = tokenizer.apply_chat_template(
    messages,
    add_generation_prompt=True,
    return_tensors="ms"
)

terminators = [
    tokenizer.eos_token_id,
    tokenizer.convert_tokens_to_ids("<|eot_id|>")
]

outputs = model.generate(
    input_ids,
    max_new_tokens=100,
    eos_token_id=terminators,
    do_sample=True,
    top_p=0.9,
)
response = outputs[0][input_ids.shape[-1]:]
print(tokenizer.decode(response, skip_special_tokens=True))
```

的路径。

init(): 初始化分布式计算环境。如果代码运行在分布式环境中，这一步是必要的。
AutoTokenizer.from_pretrained(model_id): 使用AutoTokenizer 类从预训练模型中加载分词器。
mirror='modelscope': 指定了模型镜像源为 modelscope，但这里被注释掉了。

tokenizer.apply_chat_template(...): 使用分词器将对话消息转换为输入ID，并添加生成提示。
add_generation_prompt=True: 添加生成提示。
return_tensors="ms": 返回MindSpore张量。

terminators: 定义终止符ID列表。
tokenizer.eos_token_id: 结束符号的ID。
tokenizer.convert_tokens_to_ids("<|eot_id|>"): 将特殊符号 <|eot_id|> 转换为ID。

model.generate(...): 使用模型生成文本。
input_ids: 输入ID。
max_new_tokens=100: 最大生成的新token数。
eos_token_id=terminators: 终止符ID列表。
do_sample=True: 启用采样。
top_p=0.9: 使用Top-P采样策略，设置阈值为0.9。

命令行：msrun --worker_num=2 --local_worker_num=2 --master_port=8118 --join=True run_llama3_distributed.py

在两个计算单元的集群中运行分布式llama3模型推理，主要端口号定为8118

实操

```python
import mindspore
from mindspore.communication import init
from mindnlp.transformers import AutoTokenizer, AutoModelForCausalLM

model_id = "LLM-Research/Meta-Llama-3-8B-Instruct"

init()
tokenizer = AutoTokenizer.from_pretrained(model_id, mirror='modelscope')
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    ms_dtype=mindspore.float16,
    mirror='modelscope',
    device_map="auto"
)

messages = [
    {"role": "system", "content": "You are a pirate chatbot who always responds in pirate speak!"},
    {"role": "user", "content": "Who are you?"},
]

input_ids = tokenizer.apply_chat_template(
    messages,
    add_generation_prompt=True,
    return_tensors="ms"
)

terminators = [
    tokenizer.eos_token_id,
    tokenizer.convert_tokens_to_ids("<|eot_id|>")
]

outputs = model.generate(
    input_ids,
    max_new_tokens=100,
    eos_token_id=terminators,
    do_sample=True,
    temperature=0.6,
    top_p=0.9,
)
response = outputs[0][input_ids.shape[-1]:]
print(tokenizer.decode(response, skip_special_tokens=True))
```

init(): 初始化分布式计算环境。
tokenizer = AutoTokenizer.from_pretrained(model_id, mirror='modelscope'): 使用 AutoTokenizer 类从预训练模型中加载分词器，并指定镜像源为 modelscope。
model = AutoModelForCausalLM.from_pretrained(...): 使用 AutoModelForCausalLM 类从预训练模型中加载因果语言模型。
ms_dtype=mindspore.float16: 设置模型的数据类型为 float16，以减少内存占用并加速计算。
mirror='modelscope': 指定了模型镜像源为 modelscope。
device_map="auto": 自动选择设备映射策略，这意味着模型会根据可用的硬件资源自动分配到合适的设备上（如 GPU 或 CPU）。

messages: 定义对话消息列表。
第一条消息是系统消息，指示模型以海盗风格回应。
第二条消息是用户消息，询问“你是谁？”。

tokenizer.apply_chat_template(...): 使用分词器将对话消息转换为输入ID，并添加生成提示。
add_generation_prompt=True: 添加生成提示。
return_tensors="ms": 返回MindSpore张量。

terminators: 定义终止符ID列表。
tokenizer.eos_token_id: 结束符号的ID。
tokenizer.convert_tokens_to_ids("<|eot_id|>"): 将特殊符号 <|eot_id|> 转换为ID。

model.generate(...): 使用模型生成文本。
input_ids: 输入ID。
max_new_tokens=100: 最大生成的新token数。
eos_token_id=terminators: 终止符ID列表。
do_sample=True: 启用采样。
temperature=0.6: 设置温度参数，控制生成文本的随机性。
top_p=0.9: 使用Top-P采样策略，设置阈值为0.9。

http[illegible]ub.com/mindspore-lab/mindnlp/blob/master/llm/inference/llama3/run_llama3_distributed.py

命令行：mpirun -n 2 python run_llama3_distributed.py

mpirun 2台GPU 跑llama3分布式计算脚本

作业

- 基于一种新的LLM Decoding策略实现LLM文本生成的任务，比较不同解码策略对文本生成的随机性和多样性的影响