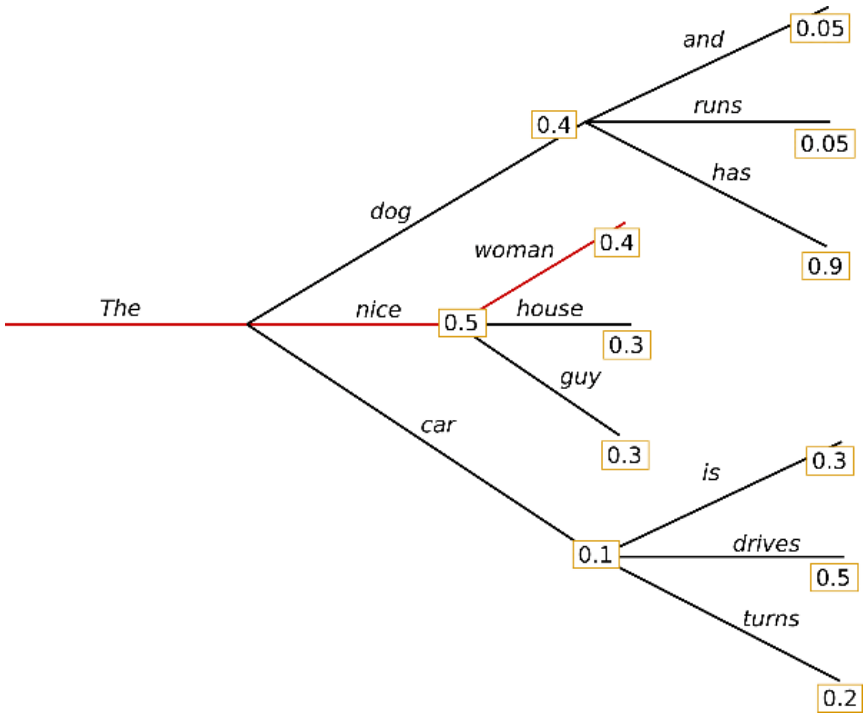
大模型生成参数配置

目前最常用的解码方法，主要有 **贪心搜索** (Greedy search)、**波束搜索** (Beam search)、**Top-K 采样** (Top-K sampling) 以及 **Top-p 采样** (Top-p sampling)。

1. 贪心搜索

贪心搜索在每个时间步 t 都简单地选择概率最高的词作为当前输出词: $w_t = argmax_w P(w|w_{1:t-1})$ ，如下图所示。



从单词 “The” 开始，算法在第一步贪心地选择条件概率最高的词 “nice” 作为输出，依此往后。最终生成的单词序列为 (“The” , “nice” , “woman”)，其联合概率为 0.5 * 0.4 = 0.2。

下面，我们输入文本序列 (“I”, “enjoy”, “walking”, “with”, “my”, “cute”, “dog”) 给 GPT2 模型，让模型生成下文。我们以此为例看看如何在 `transformers` 中使用贪心搜索（不进行任何设置）：

```
# encode context the generation is conditioned on
input_ids = tokenizer.encode('I enjoy walking with my cute dog', return_tensors='tf')

# generate text until the output length (which includes the context length) reaches 50
greedy_output = model.generate(input_ids, max_length=50)

```

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

```
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to walk with my dog. I'm not sure if I'll ever be able to walk with my dog.
I'm not sure if I'll
```

通过简单配置，我们使用 GPT2 生成了第一个短文本。根据上文生成的单词是合理的，但模型很快开始输出重复的文本，这在语言生成中是一个非常普遍的问题，在贪心搜索和波束搜索中非常常见。

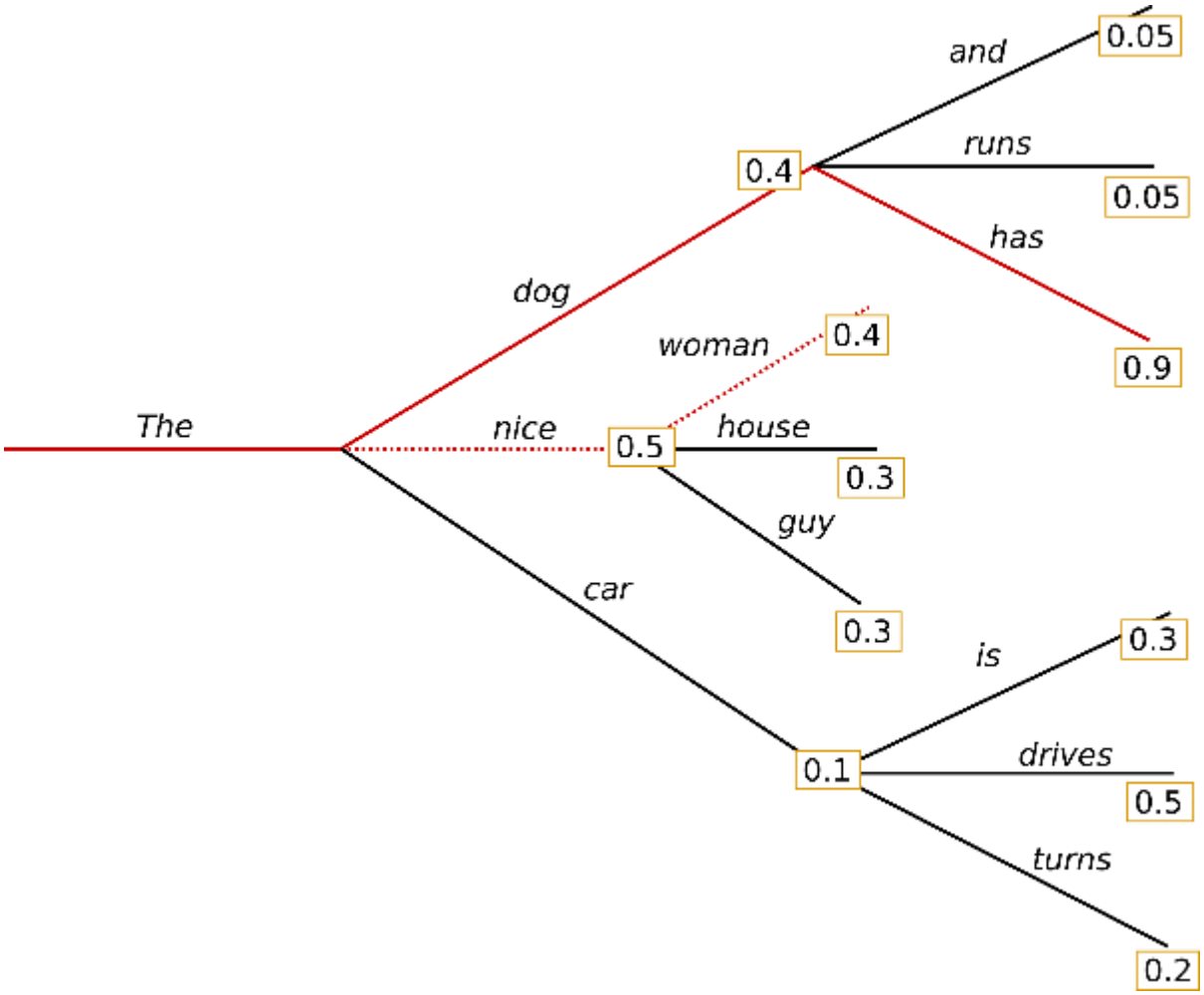
贪心搜索的主要缺点是它错过了隐藏在低概率词后面的高概率词，如上图所示:

条件概率为 0.9 的单词 “has” 隐藏在单词 “dog” 后面，而 “dog” 因为在 `t=1` 时条件概率值只排第二所以未被选择，因此贪心搜索会错过序列 “The”, “dog”, “has” 。

幸好我们可以用波束搜索来缓解这个问题!

2. 波束搜索

波束搜索通过在每个时间步保留最可能的 `num_beams` 个词，并从中最终选择出概率最高的序列来降低丢失潜在的高概率序列的风险。以 `num_beams=2` 为例:



在时间步 1，除了最有可能的假设 (“The” , “nice”)，波束搜索还跟踪第二可能的假设 (“The” , “dog”)。在时间步 2，波束搜索发现序列 (“The”, “dog”, “has”) 概率为 0.36 ，比 (“The”, “nice”, “woman”) 的 0.2 更高。太棒了，在我们的例子中它已经找到了最有可能的序列!

波束搜索一般都会找到比贪心搜索概率更高的输出序列，但仍不保证找到全局最优解。

让我们看看如何在 `transformers` 中使用波束搜索。我们设置 `num_beams > 1` 和 `early_stopping=True` 以便在所有波束达到 EOS 时直接结束生成。

```
# activate beam search and early_stopping
beam_output = model.generate(
    input_ids,
    max_length=50,
    num_beams=5,
    early_stopping=True
)

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

```
Output:
```

```
2 --------------------------------------------------------------------------------
  --------------------
3 I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to
  walk with him again.
4 I'm not sure if I'll ever be able to walk with him again. I'm not sure if I'll
```

虽然结果比贪心搜索更流畅，但输出中仍然包含重复。

针对模型生成过程中的重复问题，一个常用措施是引入 ***n-grams* (即连续 n 个词的词序列) 惩罚**。最常见的 *n-grams* 惩罚是确保每个 *n-gram* 都只出现一次，方法是如果看到当前候选词与其上文所组成的 *n-gram* 已经出现过了，就将该候选词的概率设置为 0。

我们可以通过设置 `no_repeat_ngram_size=2` 来试试，这样任意 *2-gram* 不会出现两次:

```python
# set no_repeat_ngram_size to 2
beam_output = model.generate(
    input_ids,
    max_length=50,
    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))
```

```
Output:
--------------------------------------------------------------------------------
--------------------
I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to
walk with him again.
I've been thinking about this for a while now, and I think it's time for me to
take a break
```

通过以上的配置，生成的文本已经没有重复了。但是，***n-gram* 惩罚使用时必须谨慎**，如一篇关于 *纽约* 这个城市的文章就不应使用 *2-gram* 惩罚，否则，城市名称在整个文本中将只出现一次。

波束搜索的另一个重要特性是我们能够同时获取概率最高的几个波束，并选择最符合我们要求的波束作为最终生成文本。

在 `transformers` 中，我们只需将参数 `num_return_sequences` 设置为需返回的概率最高的波束的数量，注意 `num_return_sequences <= num_beams` 。

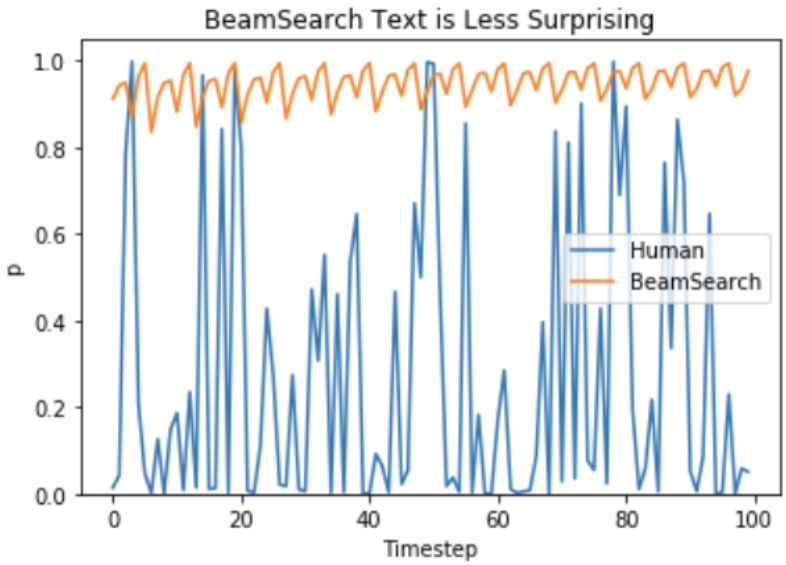
```python
# set return_num_sequences > 1
beam_outputs = model.generate(
    input_ids,
    max_length=50,
    num_beams=5,
    no_repeat_ngram_size=2,
    num_return_sequences=5,
    early_stopping=True
)

# 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, but I'm not sure if I'll ever be able to walk with him again.
I've been thinking about this for a while now, and I think it's time for me to take a break

1: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to walk with him again.
I've been thinking about this for a while now, and I think it's time for me to get back to

2: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to walk with her again.
I've been thinking about this for a while now, and I think it's time for me to take a break

3: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to walk with her again.
I've been thinking about this for a while now, and I think it's time for me to get back to

4: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to walk with him again.
I've been thinking about this for a while now, and I think it's time for me to take a step
```

上面生成的五个波束彼此之间仅有少量差别 —— 这是因为设置的波束数量较少。

开放域文本生成的研究人员最近提出了几个理由来说明对该领域而言波束搜索可能不是最佳方案:

- 在机器翻译或摘要等任务中，因为所需生成的长度或多或少都是可预测的，所以波束搜索效果比较好。但**开放域文本生成**情况有所不同，**其输出文本长度可能会有很大差异**，如对话和故事生成的输出文本长度就有很大不同。
- **波束搜索已被证明存在重复生成的问题**。在故事生成这样的场景中，很难用 *n-gram* 或其他惩罚来控制，因为需要大量的测试才能在“不重复”和最大可重复 *n-grams* 之间找到一个好的折衷。
- **高质量的人类语言并不遵循最大概率法则**。换句话说，作为人类，我们希望生成的文本能让我们感到惊喜，而可预测的文本使人感觉无聊。如下图所示，从图中可以看出**人类文本带来的惊喜度比波束搜索好不少**。



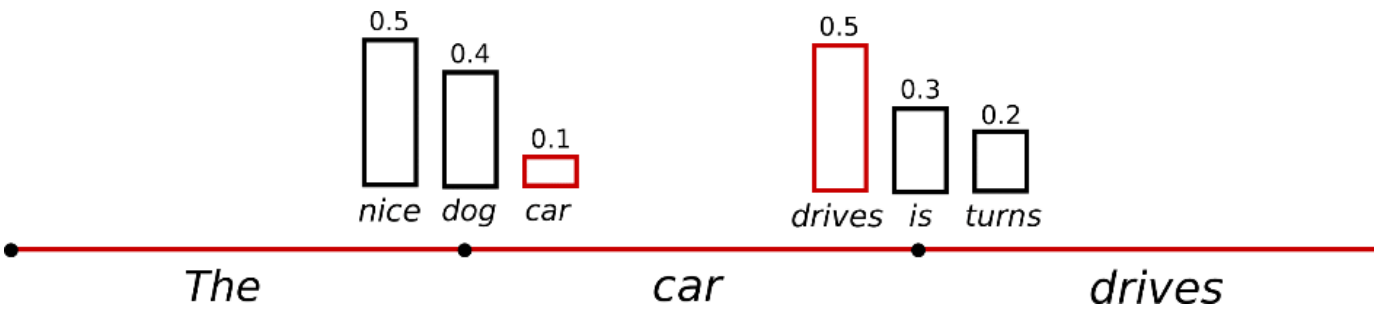
因此，**如果希望生成的文本更具有创意，可以引入一些随机性**。

3. 采样

在其最基本的形式中，**采样意味着根据当前条件概率分布随机选择输出词** wt:

$$w_t \sim P(w|w_{1:t-1})$$

继续使用上文中的例子，下图可视化了使用采样生成文本的过程。



很明显，使用**采样方法时文本生成本身不再是确定性的**。单词 “car” 从条件概率分布 $P(w|\text{“The”})$ 中采样而得，而 “drives” 则采样自 $P(w|\text{“The”}, \text{“car”})$。

在 `transformers` 中，我们设置 `do_sample=True` 并通过设置 `top_k=0` 停用 *Top-K* 采样(稍后详细介绍)。在下文中，为便于复现，我们会固定 `random_seed=0`，但你可以在自己的模型中随意更改 `random_seed`。

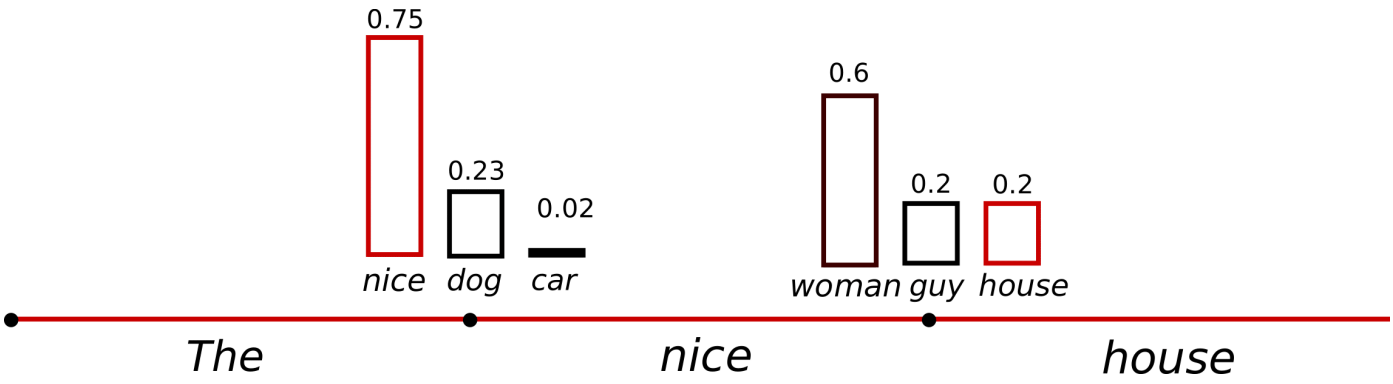
```
# set seed to reproduce results. Feel free to change the seed though to get different results
tf.random.set_seed(0)

# activate sampling and deactivate top_k by setting top_k sampling to 0
sample_output = model.generate(
    input_ids,
    do_sample=True,
    max_length=50,
    top_k=0
)

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

```
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog. He just gave me a whole new hand sense."
But it seems that the dogs have learned a lot from teasing at the local batte harness once they take on the outside.
"I take
```

上面生成的句子不是很连贯。*3-grams 的词组 new hand sense* 和 *local batte harness* 的短语非常奇怪，看起来不像是人写的。这就是对单词序列进行采样时的大问题：模型通常会产生不连贯的乱码。

缓解这一问题的一个技巧是通过降低 Softmax 的 “temperature” 使分布 $P(w|w_{1:t-1})$ 更陡峭。而降低 “temperature”，**本质上是增加高概率单词的似然并降低低概率单词的似然**。

将 temperature 应用到于我们的例子中后，结果如下图所示。



$t = 1$ 时刻单词的条件分布变得更加陡峭，几乎没有机会选择单词“car”了。

让我们看看如何通过设置 `temperature=0.7` 来冷却生成过程:

```
# set seed to reproduce results. Feel free to change the seed though to get different results
tf.random.set_seed(0)

# use temperature to decrease the sensitivity to low probability candidates
sample_output = model.generate(
    input_ids,
    do_sample=True,
    max_length=50,
    top_k=0,
    temperature=0.7
)

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

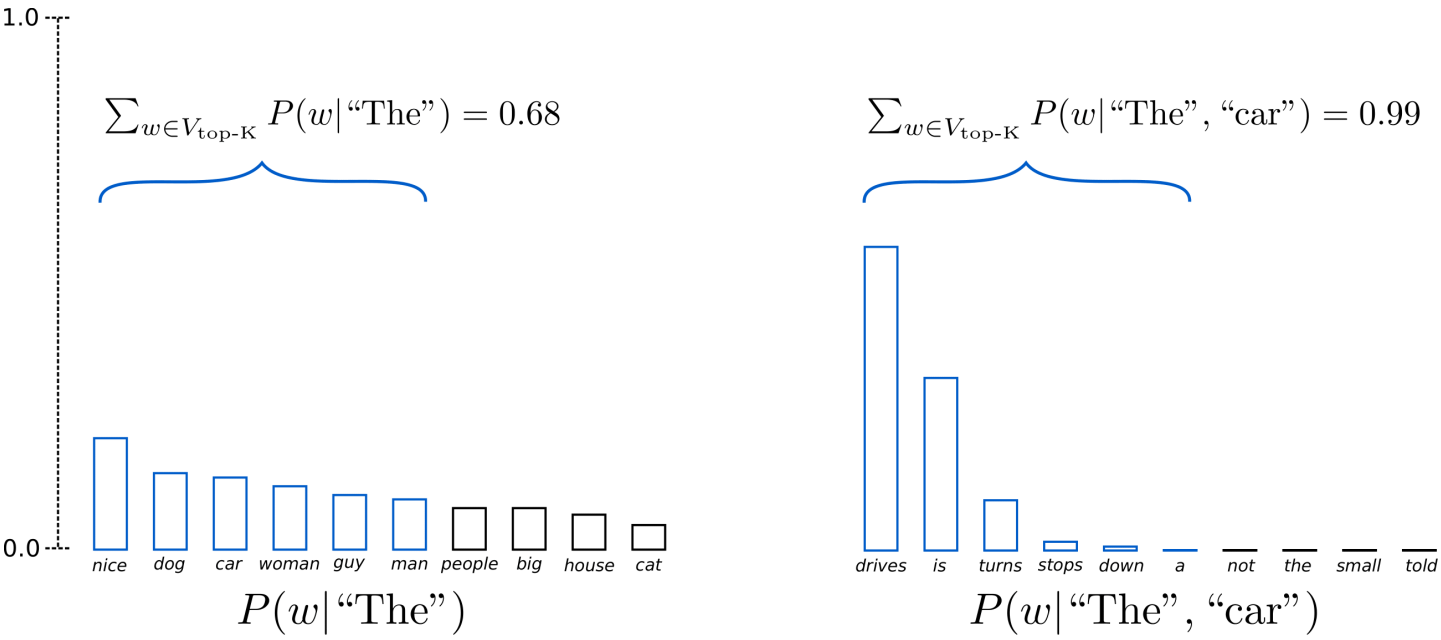
```
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog, but I don't like to be at home too much. I also find it a bit weird when I'm out shopping. I am always away from my house a lot, but I do have a few friends
```

调整后，不连贯的 n-gram 变少了，现在输出更连贯了，虽然温度可以使分布的随机性降低，但极限条件下，**当“温度”设置为 0 时，温度缩放采样就退化成贪心解码了，因此会遇到与贪心解码相同的问题**。

4. Top-K 采样

Top-K 采样是一种强大的采样方式。在 *Top-K* 采样中，**概率最大的 *K* 个词会被选出，然后这 *K* 个词的概率会被重新归一化，最后就在这重新被归一化概率后的 *K* 个词中采样**。 GPT2 采用了这种采样方案，这也是它在故事生成这样的任务上取得成功的原因之一。

我们将上文例子中的候选单词数从 3 个单词扩展到 10 个单词，以更好地说明 *Top-K* 采样。



设 K = 6 ，即我们将在两个采样步的采样池大小限制为 6 个单词。我们定义 6 个最有可能的词的集合为 $V_{\text{top-k}}$ 。在第一步中， $V_{\text{top-k}}$ 仅占总概率的大约三分之二，但在第二步，它几乎占了全部的概率。同时，我们可以看到在第二步该方法成功地消除了那些奇怪的候选词 $(\text{“not”}, \text{“the”}, \text{“small”}, \text{“told”})$ 。

我们以设置 `top_k=50` 为例看下如何在 `transformers` 库中使用 *Top-K*:

```
# set seed to reproduce results. Feel free to change the seed though to get different results
tf.random.set_seed(0)

# set top_k to 50
sample_output = model.generate(
    input_ids,
    do_sample=True,
    max_length=50,
    top_k=50
)

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

```
Output:
```

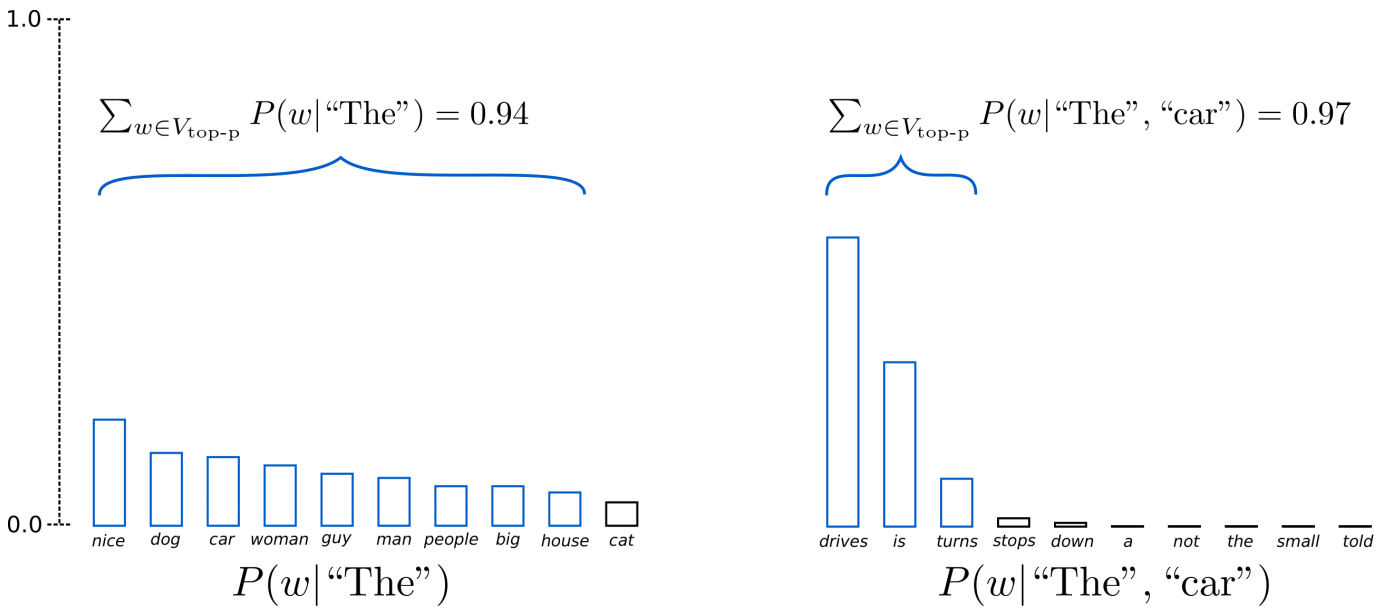
```
2 ------------------------------------------------------------------------------
  --------------------
3 I enjoy walking with my cute dog. It's so good to have an environment where
  your dog is available to share with you and we'll be taking care of you.
4 We hope you'll find this story interesting!
```

相当不错！该文本可以说是迄今为止生成的最 "像人" 的文本。现在还有一个问题，*Top-K* 采样不会动态调整从需要概率分布 $P(w|w_{1:t-1})$ 中选出的单词数。这可能会有问题，因为**某些分布可能是非常尖锐 (上图中右侧的分布)，而另一些可能更平坦 (上图中左侧的分布)，所以对不同的分布使用同一个绝对数 *K* 可能并不普适**。

在 $t = 1$ 时，*Top-K* 将 $(\text{“people”}, \text{“big”}, \text{“house”}, \text{“cat”})$ 排出了采样池，而这些词似乎是合理的候选词。另一方面，在 t=2 时，该方法却又把不太合适的 $(\text{“down”}, \text{“a”})$ 纳入了采样池。因此，**将采样池限制为固定大小 *K* 可能会在分布比较尖锐的时候产生胡言乱语，而在分布比较平坦的时候限制模型的创造力**。这一发现促使 Top-p 采样的出现。

5. Top-p（核）采样

在 *Top-p* 中，采样不只是在最有可能的 *K* 个单词中进行，而是**在累积概率超过概率 *p* 的最小单词集中进行**。然后在这组词中重新分配概率质量。这样，词集的大小 (*又名* 集合中的词数) 可以根据下一个词的概率分布动态增加和减少。



假设 p=0.92，*Top-p* 采样对单词概率进行降序排列并累加，然后选择概率和首次超过 $p = 92$ 的单词集作为采样池，定义为 $V_{\text{top-p}}$ 。在 $t = 1$ 时 $V_{\text{top-p}}$ 有 9 个词，而在 $t = 2$ 时它只需要选择前 3 个词就超过了 92%。其实很简单吧！可以看出，在单词比较不可预测时，它保留了更多的候选词，如 $P(w|\text{“The”})$，而当单词似乎更容易预测时，只保留了几个候选词，*如* $P(w|\text{“The”}, \text{“car”})$。

可以通过设置 `0 < top_p < 1` 来激活 *Top-p* 采样:

```
# set seed to reproduce results. Feel free to change the seed though to get different results
tf.random.set_seed(0)

# deactivate top_k sampling and sample only from 92% most likely words
sample_output = model.generate(
    input_ids,
    do_sample=True,
    max_length=50,
    top_p=0.92,
    top_k=0
)

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

```
Output:
----------------------------------------------------------------------------------------------------
I enjoy walking with my cute dog. He will never be the same. I watch him play.

Guys, my dog needs a name. Especially if he is found with wings.

```

虽然从理论上讲， *Top-p* 似乎比 *Top-K* 更优雅，但这两种方法在实践中都很有效。 *Top-p* 也可以与 *Top-K* 结合使用，这样可以避免排名非常低的词，同时允许进行一些动态选择。

最后，如果想要获得多个独立采样的输出，我们可以 *再次* 设置参数 `num_return_sequences > 1`：

```
# set seed to reproduce results. Feel free to change the seed though to get different results
tf.random.set_seed(0)

# set top_k = 50 and set top_p = 0.95 and num_return_sequences = 3
sample_outputs = model.generate(
    input_ids,
    do_sample=True,
    max_length=50,
    top_k=50,
    top_p=0.95,
    num_return_sequences=3
)
```

```
13
14 print("Output:\n" + 100 * '-')
15 for i, sample_output in enumerate(sample_outputs):
16   print("{}: {}".format(i, tokenizer.decode(sample_output,
   skip_special_tokens=True)))
```

```
1 Output:
2 ----------------------------------------------------------------------------------------------------
3 0: I enjoy walking with my cute dog. It's so good to have the chance to walk with a dog. But I have this problem with the dog and how he's always looking at us and always trying to make me see that I can do something
4 1: I enjoy walking with my cute dog, she loves taking trips to different places on the planet, even in the desert! The world isn't big enough for us to travel by the bus with our beloved pup, but that's where I find my love
5 2: I enjoy walking with my cute dog and playing with our kids," said David J. Smith, director of the Humane Society of the US.
6 "So as a result, I've got more work in my time," he said.
```

以上就是 Transformer generate 方法中的核心配置参数，generate 方法还有几个常用的参数：

- min_length 用于强制模型在达到 min_length 之前不生成 EOS。这在摘要场景中使用得比较多，但如果用户想要更长的文本输出，也会很有用。
- **repetition_penalty 可用于对生成重复的单词这一行为进行惩罚**，它可以非常有效地防止重复，但似乎对模型和用户场景非常敏感。
- attention_mask 可用于屏蔽填充符。
- pad_token_id、bos_token_id、eos_token_id: 如果模型默认没有这些 token，用户可以手动选择其他 token id 来表示它们。

6. 总结

在开放域语言生成场景中，作为最新的解码方法， *top-p* 和 *top-K* 采样于传统的贪心和波束搜索相比，似乎能产生更流畅的文本。但，最近有更多的证据表明**贪心和波束搜索的明显缺陷—生成重复的单词序列，是由模型 (特别是模型的训练方式) 引起的，而不是解码方法**， 在一些研究中发现，*top-K* 和 *top-p* 采样也会产生重复的单词序列。

一些论文的研究表明，根据人类评估，**在调整训练目标后，波束搜索相比 *Top-p* 采样能产生更流畅的文本**。

开放域语言生成是一个快速发展的研究领域，而且通常情况下这里没有放之四海而皆准的方法，因此必须了解哪种方法最适合自己的特定场景。

7. 实际业务中的应用

在不同场景的大模型应用中，需要根据实际测试结果，确认合理的generate配置，不同的大模型，相同大模型在不同数据集训练的结果，可能需要配置不同的generate参数。

```
output_texts = model.generate(
    input_ids=input_ids,
    attention_mask=attention_mask,
    num_beams=5,
    do_sample=False,
    min_new_tokens=1,
    max_new_tokens=512,
    no_repeat_ngram_size=2,
    num_return_sequences=3,
    temperature=0.15,
    early_stopping=True
)
```

```
output_texts = model.generate(
    input_ids=input_ids,
    attention_mask=attention_mask,
    num_beams=4,
    do_sample=False,
    min_new_tokens=50,
    max_new_tokens=512,
    no_repeat_ngram_size=2,
    num_return_sequences=3,
    temperature=0.3,
    early_stopping=True,
)
```