How to generate text: using different decoding methods for language generation with Transformers

Introduction

The field of open-ended language generation has advanced rapidly, fueled by the emergence of even more powerful transformer-based language models. These models, trained on vast quantities of text data, surpass predecessors like OpenAI's GPT-2. Cutting-edge language models like GPT-3 and its variations now lead the way. Additionally, continuous innovation in decoding methods further improves the quality of generated text.

Let's explore prominent decoding methods and how you can employ them using the `transformers` library for your language generation tasks.

Understanding Auto-regressive Language Generation

Auto-regressive language generation remains a fundamental technique (refresher). Briefly, it works on the principle that a word sequence's probability distribution can be broken down as follows:

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

where W_0 is the starting context. The sequence length, T, is dictated by when the model produces an EOS (end-of-sequence) token.

Decoding Strategies

Let's look at the prevalent decoding techniques available in the `transformers` library:

- **Greedy Search:** A straightforward approach; in each step, the word with the highest probability is selected.
- **Beam Search:** Keeps track of multiple top candidates throughout generation, offering more refined results.
- **Top-K Sampling:** Randomly picks from the top 'K' most likely words; introduces variety and unexpected text directions.
- **Top-p Sampling (Nucleus Sampling):** Selects from a subset of words whose cumulative probability accounts for a defined percentage 'p'; balances predictability and creativity.

Note: Decoding techniques have become significantly more sophisticated alongside model improvements.

Availability

You can use auto-regressive language generation with models like GPT-2, GPT-3 (through APIs), XLNet, OpenAI-GPT, CTRL, TransfoXL, XLM, BART, T5, and others in PyTorch and TensorFlow >= 2.0.

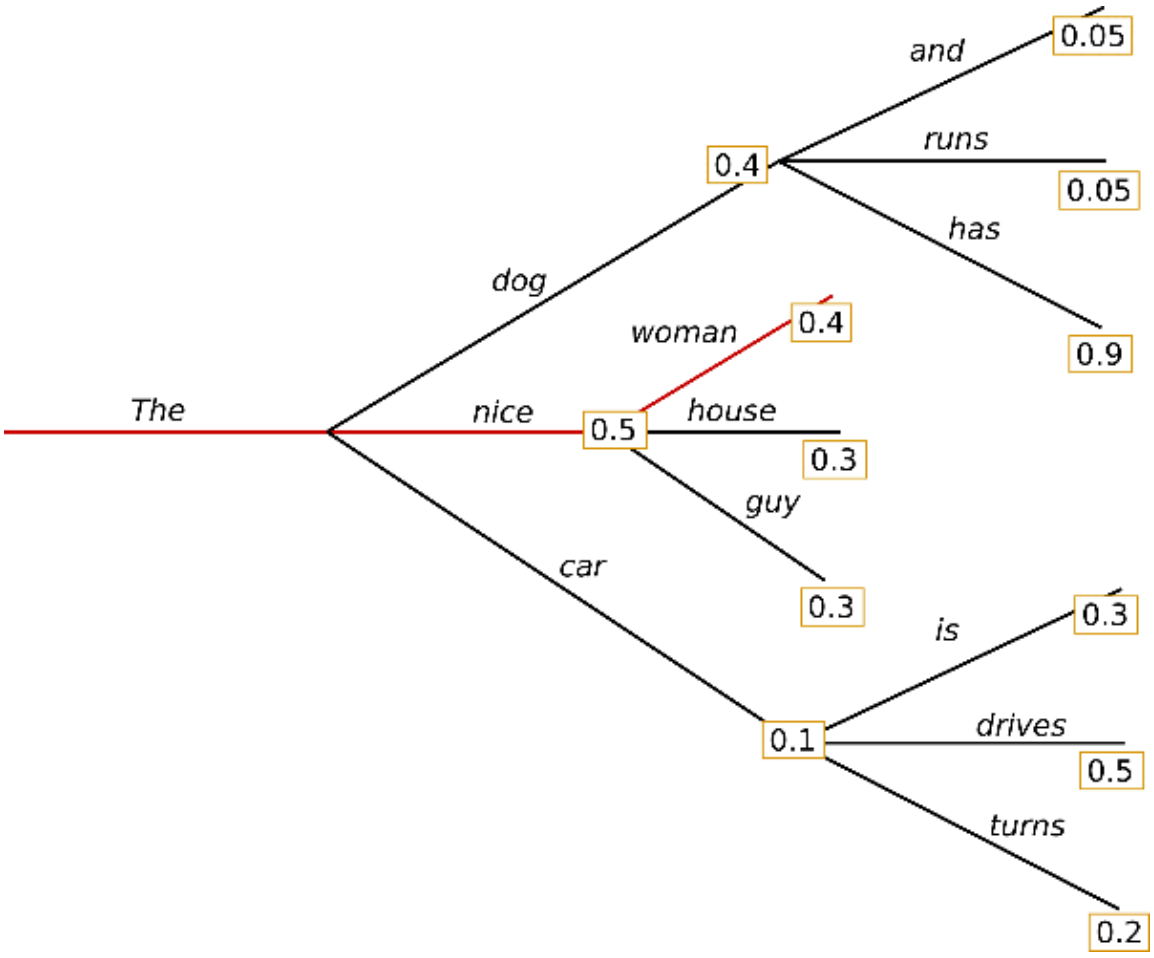
Loading the model and tokenizer

The transformers library makes it very easy to load and use existing models published to their site. For this example, we will load the `TFGP2LMHeadModel` and `GPT2Tokenizer` classes from the `transformers` library. The `TF` prefix indicates that we are using the TensorFlow version of the model. If you are using PyTorch, you would use the `GPT2LMHeadModel` and `GPT2Tokenizer` classes instead.

In [ ]:
```
import tensorflow as tf
from transformers import TFGPT2LMHeadModel, GPT2Tokenizer

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

# add the EOS token as PAD token to avoid warnings
model = TFGPT2LMHeadModel.from_pretrained("gpt2", pad_token_id=tokenizer.eos
```

Greedy Search

Greedy search simply selects the word with the highest probability as its next word: $w_t = argmax_w P(w|w_{1:t-1})$ at each timestep t. The following sketch shows greedy search.



Starting from the word "The", the algorithm greedily chooses the next word of highest probability "nice" and so on, so that the final generated word sequence is "The", "nice", "woman" having an overall probability of $0.5 \times 0.4 = 0.2$.

In the following we will generate word sequences using GPT2 on the context ("I", "enjoy", "walking", "with", "my", "cute", "dog"). Let's see how greedy search can be used in `transformers` as follows:

In [ ]:
```
# encode context the generation is conditioned on
input_ids = tokenizer.encode('I enjoy walking with my cute dog', return_tens

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

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

Alright! We have generated our first short text with GPT2 😊. The generated words following the context are reasonable, but the model quickly starts repeating itself! This is a very common problem in language generation in general and seems to be even more so in greedy and beam search.

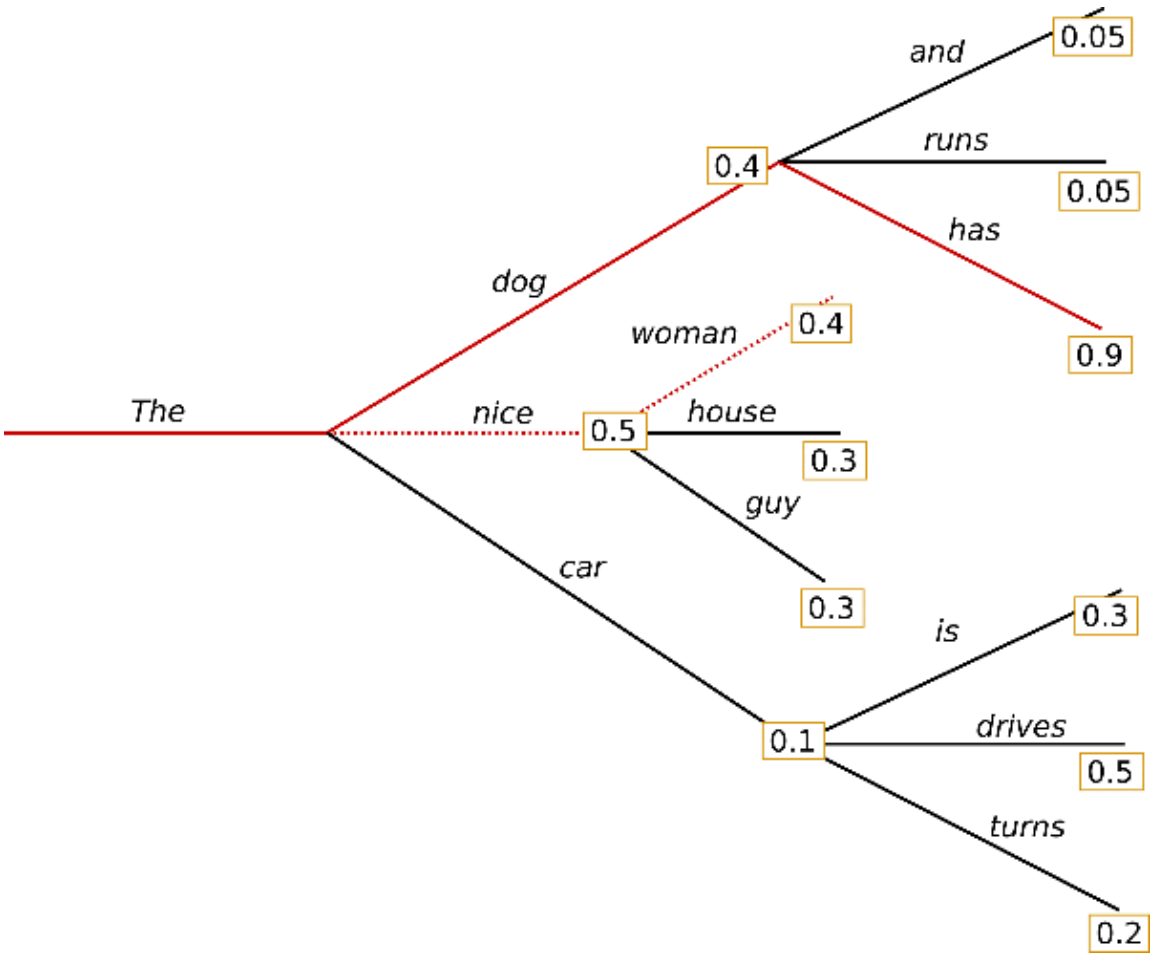
The major drawback of greedy search though is that it misses high probability words hidden behind a low probability word as can be seen in our sketch above:

The word "has" with its high conditional probability of 0.9 is hidden behind the word "dog", which has only the second-highest conditional probability, so that greedy search misses the word sequence "The", "dog", "has".

Thankfully, we have beam search to alleviate this problem!

Beam search

Beam search reduces the risk of missing hidden high probability word sequences by keeping the most likely `num_beams` of hypotheses at each time step and eventually choosing the hypothesis that has the overall highest probability. Let's illustrate with `num_beams=2`:



At time step 1, besides the most likely hypothesis "The", "nice", beam search also keeps track of the second most likely one "The", "dog". At time step 2, beam search finds that the word sequence "The", "dog", "has" has with 0.36 a higher probability than "The", "nice", "woman", which has 0.2. Great, it has found the most likely word sequence in our toy example!

Beam search will always find an output sequence with higher probability than greedy search, but is not guaranteed to find the most likely output.

Let's see how beam search can be used in `transformers`. We set `num_beams > 1` and `early_stopping=True` so that generation is finished when all beam hypotheses reached the EOS token.

In [ ]:
```
# 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))
```

While the result is arguably more fluent, the output still includes repetitions of the same word sequences.

A simple remedy is to introduce *n-grams* (*a.k.a* word sequences of n words) penalties as introduced by Paulus et al. (2017) and Klein et al. (2017). The most common *n-grams* penalty makes sure that no *n-gram* appears twice by manually setting the probability of next words that could create an already seen *n-gram* to 0.

Let's try it out by setting `no_repeat_ngram_size=2` so that no *2-gram* appears twice:

In [ ]:
```
# 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))
```

Nice, that looks much better! We can see that the repetition does not appear anymore. Nevertheless, *n-gram* penalties have to be used with care. An article generated about the city *New York* should not use a *2-gram* penalty or otherwise, the name of the city would only appear once in the whole text!

Another important feature about beam search is that we can compare the top beams after generation and choose the generated beam that fits our purpose

best.

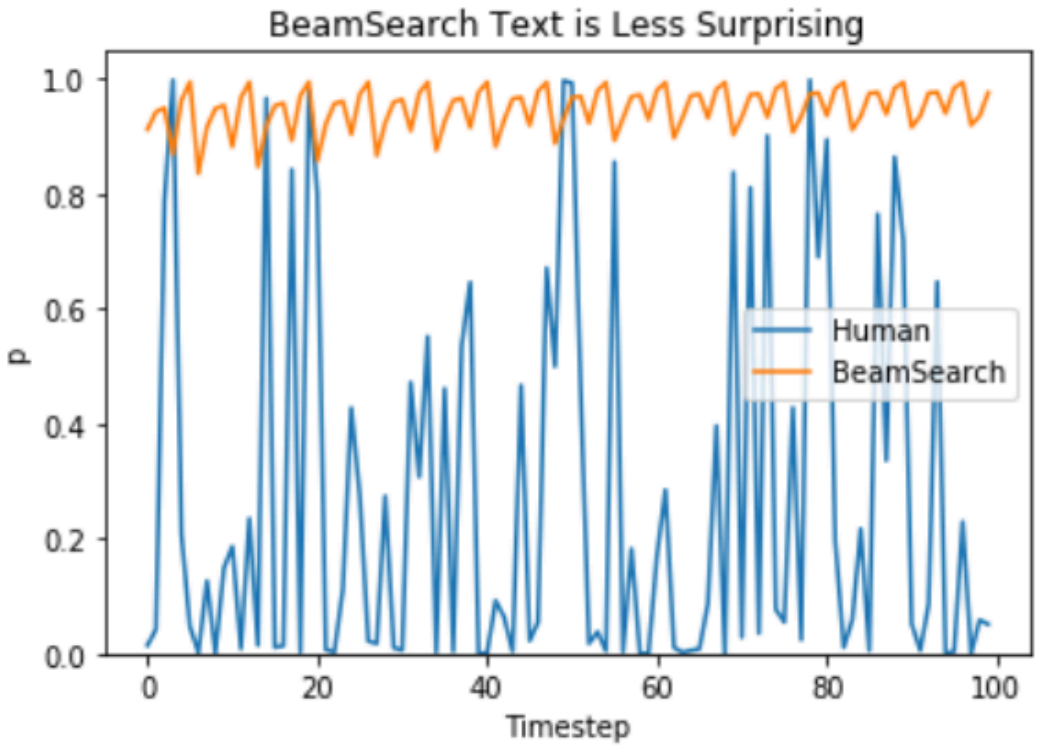
In `transformers`, we simply set the parameter `num_return_sequences` to the number of highest scoring beams that should be returned. Make sure though that `num_return_sequences <= num_beams`!

In [ ]:
```
# 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(f'{i}: {tokenizer.decode(beam_output, skip_special_tokens=True)}')
    print('---')
```

As can be seen, the five beam hypotheses are only marginally different to each other - which should not be too surprising when using only 5 beams.

In open-ended generation, a couple of reasons have recently been brought forward why beam search might not be the best possible option:

- Beam search can work very well in tasks where the length of the desired generation is more or less predictable as in machine translation or summarization. But this is not the case for open-ended generation where the desired output length can vary greatly, e.g. dialog and story generation.
- We have seen that beam search heavily suffers from repetitive generation. This is especially hard to control with *n-gram*- or other penalties in story generation since finding a good trade-off between forced "no-repetition" and repeating cycles of identical *n-grams* requires a lot of finetuning.
- High quality human language does not follow a distribution of high probability next words. In other words, as humans, we want generated text to surprise us and not to be boring/predictable.



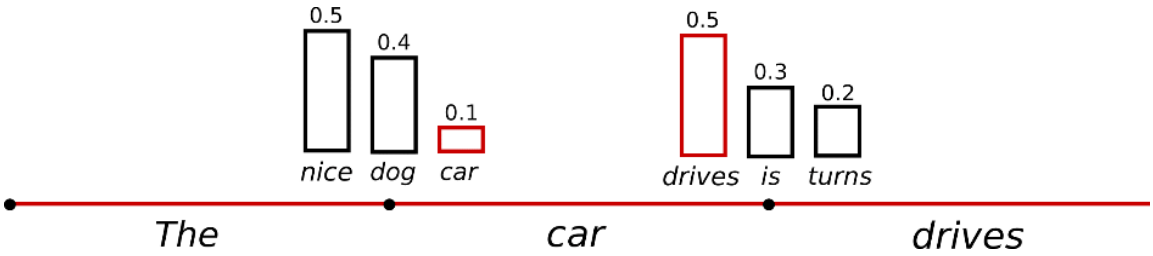
So let's stop being boring and introduce some randomness .

Sampling

In its most basic form, sampling means randomly picking the next word w_t according to its conditional probability distribution:

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

Taking the example from above, the following graphic visualizes language generation when sampling.



It becomes obvious that language generation using sampling is not *deterministic* anymore. The word "car" is sampled from the conditioned probability distribution $P(w|\text{"The"})$, followed by sampling "drives" from $P(w|\text{"The"}, \text{"car"})$.

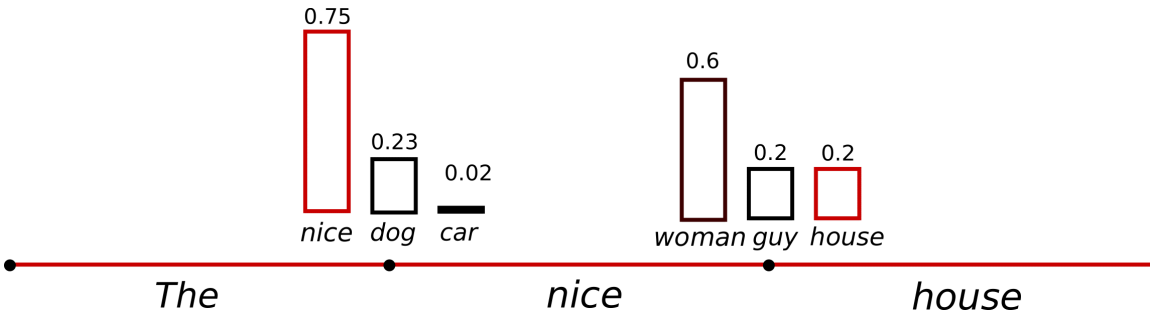
In `transformers`, we set `do_sample=True` and deactivate *Top-K* sampling (more on this later) via `top_k=0`. In the following, we will fix `random_seed=0` for illustration purposes. Feel free to change the `random_seed` to play around with the model.

In [ ]:
```
# 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))
```

Interesting! The text seems alright - but when taking a closer look, it is not very coherent. Much of the text is very weird, and and it doesn't sound like it was written by a human. That is the big problem when sampling word sequences: The models often generate incoherent gibberish.

A trick is to make the distribution $P(w|w_{1:t-1})$ sharper (increasing the likelihood of high probability words and decreasing the likelihood of low probability words) by lowering the so-called `temperature` of the softmax.

An illustration of applying temperature to our example from above could look as follows.



The conditional next word distribution of step $t = 1$ becomes much sharper leaving almost no chance for word "car" to be selected.

Let's see how we can cool down the distribution in the library by setting `temperature=0.7`:

In [ ]:
```
# 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
```

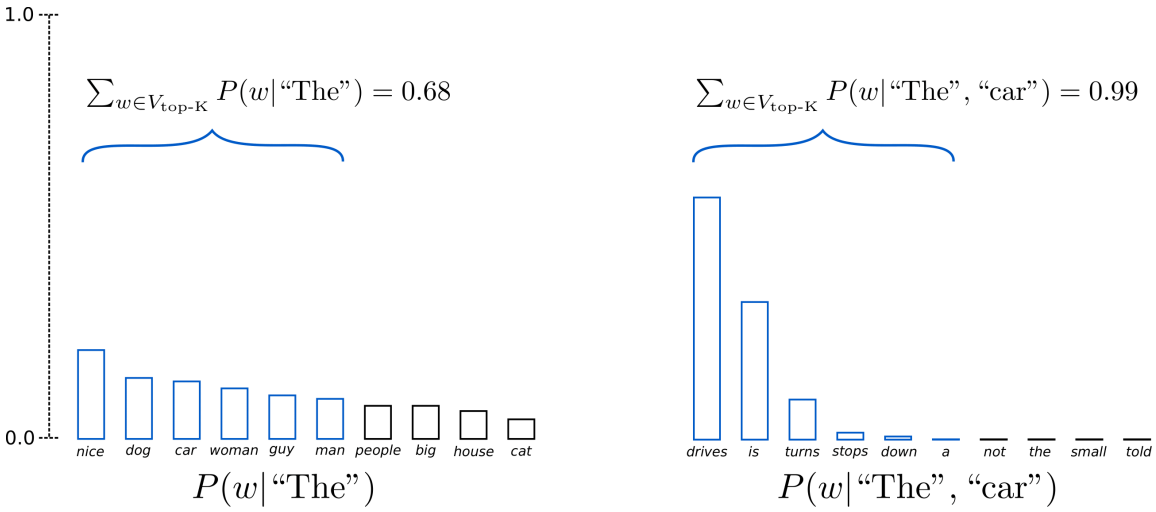
```
)

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

OK. There are less weird n-grams and the output is a bit more coherent now! While applying temperature can make a distribution less random, in its limit, when setting `temperature` $\to 0$, temperature scaled sampling becomes equal to greedy decoding and will suffer from the same problems as before.

Top-K Sampling

Fan et. al (2018) introduced a simple, but very powerful sampling scheme, called ***Top-K*** sampling. In *Top-K* sampling, the *K* most likely next words are filtered and the probability mass is redistributed among only those *K* next words. GPT2 adopted this sampling scheme, which was one of the reasons for its success in story generation.

We extend the range of words used for both sampling steps in the example above from 3 words to 10 words to better illustrate *Top-K* sampling.



Having set $K = 6$, in both sampling steps we limit our sampling pool to 6 words. While the 6 most likely words, defined as $V_{\text{top-K}}$ encompass only about two-thirds of the whole probability mass in the first step, it includes almost all of the probability mass in the second step. Nevertheless, we see that it successfully eliminates the rather weird candidates $\text{”not”}, \text{”the”}, \text{”small”}, \text{”told”}$ in the second sampling step.

Let's see how *Top-K* can be used in the library by setting `top_k=50`:

In [ ]:
```
# set seed to reproduce results. Feel free to change the seed though to get
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))
```

Not bad at all! The text is arguably the most *human-sounding* text so far.

One concern though with *Top-K* sampling is that it does not dynamically adapt the number of words that are filtered from the next word probability distribution $P(w|w_{1:t-1})$.

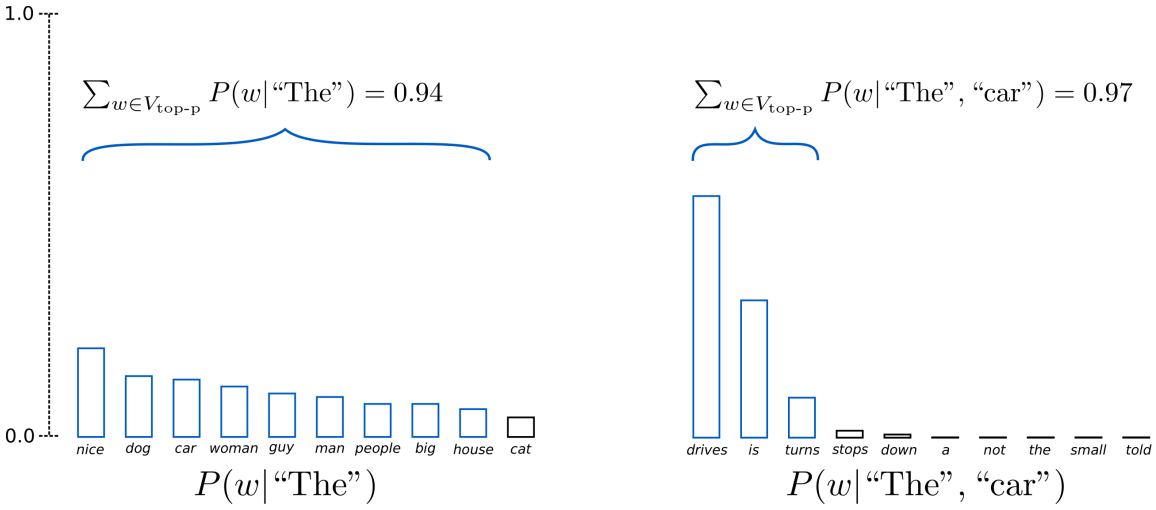
This can be problematic as some words might be sampled from a very sharp distribution (distribution on the right in the graph above), whereas others from a much more flat distribution (distribution on the left in the graph above).

In step $t = 1$, *Top-K* eliminates the possibility to sample $\text{”people”}, \text{”big”}, \text{”house”}, \text{”cat”}$, which seem like reasonable candidates. On the other hand, in step $t = 2$ the method includes the arguably ill-fitted words $\text{”down”}, \text{”a”}$ in the sample pool of words. Thus, limiting the sample pool to a fixed size *K* could endanger the model to produce gibberish for sharp distributions and limit the model's creativity for flat distribution.

This intuition led Ari Holtzman et al. (2019) to create ***Top-p***- or ***nucleus***-sampling.

Top-p (nucleus) sampling

Instead of sampling only from the most likely *K* words, in *Top-p* sampling chooses from the smallest possible set of words whose cumulative probability exceeds the probability *p*. The probability mass is then redistributed among this set of words. This way, the size of the set of words (*a.k.a* the number of words in the set) can dynamically increase and decrease according to the next word's probability distribution. Ok, that was very wordy, let's visualize.



Having set $p = 0.92$, *Top-p* sampling picks the *minimum* number of words to exceed together $p = 92\%$ of the probability mass, defined as $V_{\text{top-p}}$. In the first example, this included the 9 most likely words, whereas it only has to pick the top 3 words in the second example to exceed 92%. Quite simple actually! It can be seen that it keeps a wide range of words where the next word is arguably less predictable, *e.g.* $P(w|\text{”The”})$, and only a few words when the next word seems more predictable, *e.g.* $P(w|\text{”The”, ”car”})$.

Alright, time to check it out in `transformers`! We activate *Top-p* sampling by setting `0 < top_p < 1`:

In [ ]:
```
# 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))
```

Great, that sounds like it could have been written by a human. Well, maybe not quite yet.

While in theory, *Top-p* seems more elegant than *Top-K*, both methods work well in practice. *Top-p* can also be used in combination with *Top-K*, which can avoid very low ranked words while allowing for some dynamic selection.

Finally, to get multiple independently sampled outputs, we can *again* set the parameter `num_return_sequences > 1`:

In [ ]:
```
# 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
)

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

Exercise

Now that you've learned about different sampling methods, it's time to put your knowledge to the test. Try to generate the most realistic sounding text you can using the GPT-2 model. You can use the context "I enjoy walking with my cute dog" and any of the sampling methods you've learned about. Feel free to experiment with the parameters to see how they affect the output. How long can you make the text before it starts to sound incoherent? What happens if you change the temperature or the top-k or top-p values?

In [ ]:
```
# Your code here
```