## **Pretrained GPT-2 Model**

**Introduction:**

In recent years, there has been an increasing interest in open-ended language generation thanks to the rise of large transformer-based language models trained on millions of webpages, such as OpenAI's famous GPT2 model. The results on conditioned open-ended language generation are impressive, e.g. GPT2 on unicorns, XLNet, Controlled language with CTRL. Besides the improved transformer architecture and massive unsupervised training data, better decoding methods have also played an important role.

All of the following functionalities can be used for auto-regressive language generation (here a refresher). In short, auto-regressive language generation is based on the assumption that the probability distribution of a word sequence can be decomposed into the product of conditional next word distributions:

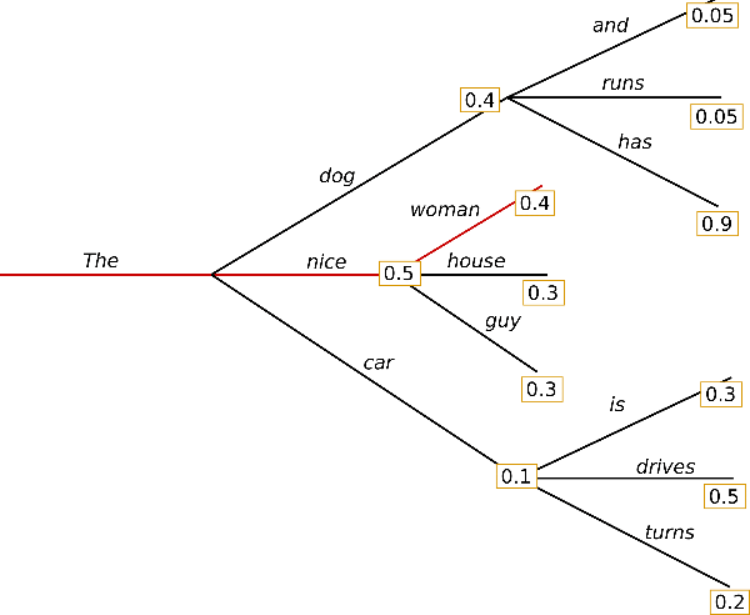
P(w1:T|W0)=∏t=1TP(wt|w1:t−1,W0) ,with w1:0=∅, and W0 being the initial context word sequence. The length T of the word sequence is usually determined on-the-fly and corresponds to the timestep t=T the EOS token is generated from P(wt|w1:t−1,W0) .

Auto-regressive language generation is now available for GPT2, XLNet, OpenAi-GPT, CTRL, TransfoXL, XLM, Bart, T5 in both PyTorch and Tensorflow >= 2.0!

We will go through most prominent decoding methods, mainly Greedy search, Beam search, Top-K sampling and Top-p sampling.

1. **Greedy Search approach:**

Greedy search simply selects the word with the highest probability as its next word: 𝑤𝑡=𝑎𝑟𝑔𝑚𝑎𝑥𝑤𝑃(𝑤|𝑤1: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 0.5×0.4=0.2.

In the following we will generate word sequences using GPT2 on the context ("I", "enjoy", "walking", "with", "my", "cute", "dog")

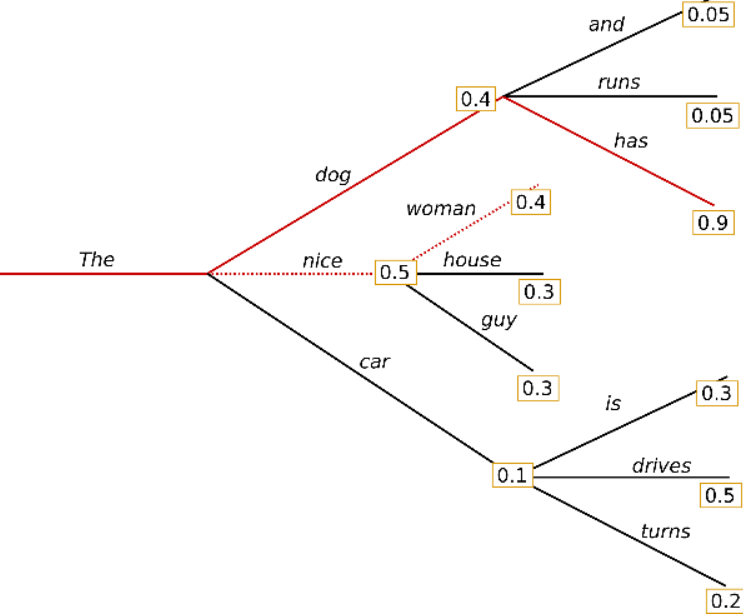
**The major drawback of greedy search though is that it misses high probability words hidden behind a low probability word as can be seen in our sketch above:**

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".

**We can solve by using Beam Approach.**

1. **Beam Approach:**

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", "woman", 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.**

**How can we stop repetitions to an extent?**

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. 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.

**Feedback of the beam approach:**

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. The authors show this nicely by plotting the probability, a model would give to human text vs. what beam search does.

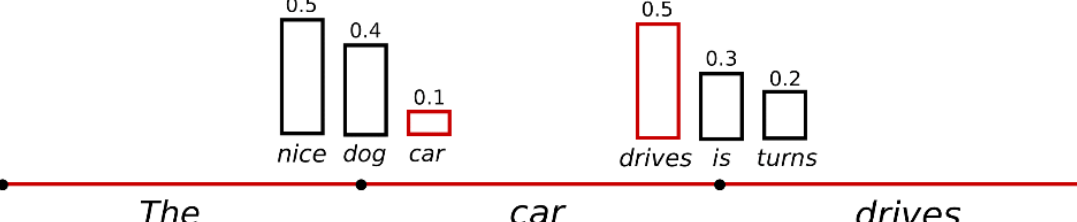
**Now we can introduce some randomness and optimize the model**

1. **Sampling Approach:**

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

𝑤𝑡∼𝑃(𝑤|𝑤1: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|"The"), followed by sampling "drives" from 𝑃(𝑤|"The","car").

**The text seems alright, but when taking a closer look, it is not very coherent.** A trick is to make the distribution 𝑃(𝑤|𝑤1:𝑡−1) sharper (increasing the likelihood of high probability words and decreasing the likelihood of low probability words) by lowering the temperature

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

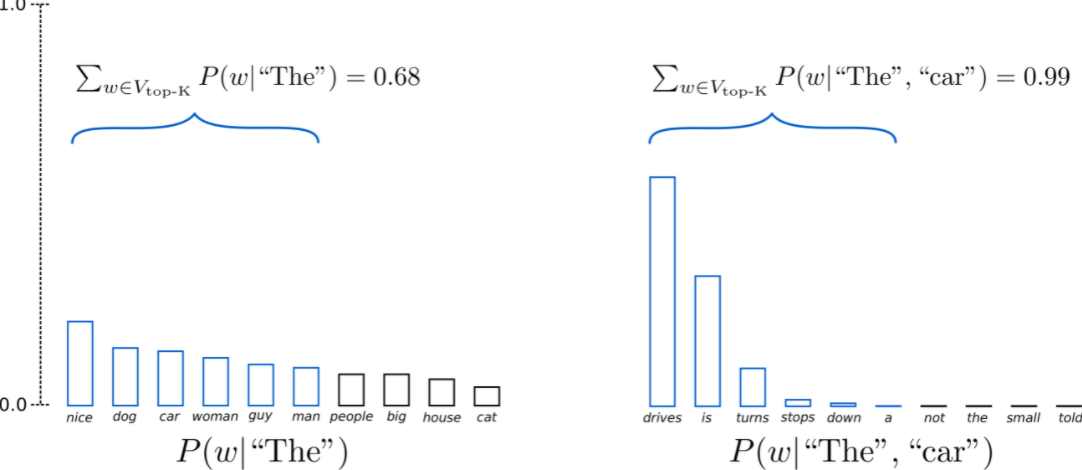


The conditional next word distribution of step 𝑡=1 becomes much sharper leaving almost no chance for word "car" to be selected.

Still we can optimize by selecting important words.

1. **Top K Sampling Approach:**

This is 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 𝑉top-K encompass only ca. 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 "not", "the", "small", "told" in the second sampling step.

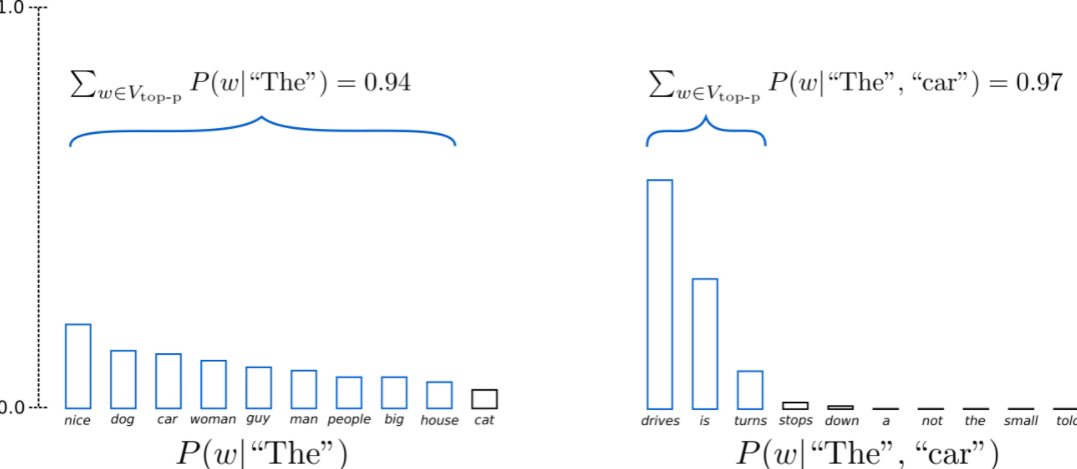
**Feedback of this model:**

It's average. 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|𝑤1: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 "people", "big", "house", "cat", which seem like reasonable candidates. On the other hand, in step 𝑡=2 the method includes the arguably ill-fitted words "down", "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 to create Top-p- or nucleus-sampling.

1. **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.



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 𝑉top-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|"The"), and only a few words when the next word seems more predictable, e.g. 𝑃(𝑤|"The", "car").

In theory, Top-p seems more elegant than Top-K, both methods work well in practice, In order for perfection we can use both Top-p and Top-k at a time to choose the at most important words.

**Conclusion:**

As top-p and top-K sampling seem to produce more fluent text than traditional greedy - and beam search on open-ended language generation. Recently, there has been more evidence though that the apparent flaws of greedy and beam search - mainly generating repetitive word sequences - are caused by the model (especially the way the model is trained), rather than the decoding method.

## **Pretrained Pipelines**

**Introduction:**

The pipelines are a great and easy way to use models for inference. These pipelines are objects that abstract most of the complex code from the library, offering a simple API dedicated to several tasks, including Named Entity Recognition, Masked Language Modeling, Sentiment Analysis, Feature Extraction and Question Answering. The pipeline abstraction is a wrapper around all the other available pipelines. It is instantiated as any other pipeline but requires an additional argument which is the task.

Pipelines are made of:

* A [tokenizer](https://huggingface.co/transformers/main_classes/tokenizer.html) in charge of mapping raw textual input to token.
* A [model](https://huggingface.co/transformers/main_classes/model.html) to make predictions from the inputs.
* Some (optional) post processing for enhancing model’s output.

**Text Generation Pipeline:**

Language generation pipeline using any **ModelWithLMHead**. This pipeline predicts the words that will follow a specified text prompt. This language generation pipeline can currently be loaded from [**pipeline()**](https://huggingface.co/transformers/main_classes/pipelines.html#transformers.pipeline) using the following task identifier: **"text-generation"**.

The models that this pipeline can use are models that have been trained with an autoregressive language modeling objective, which includes the unidirectional models in the library (e.g. gpt2). See the list of available models on [huggingface.co/models](https://huggingface.co/models?filter=causal-lm).

**Parameters**

* **model** ([**PreTrainedModel**](https://huggingface.co/transformers/main_classes/model.html#transformers.PreTrainedModel) or [**TFPreTrainedModel**](https://huggingface.co/transformers/main_classes/model.html#transformers.TFPreTrainedModel)) – The model that will be used by the pipeline to make predictions. This needs to be a model inheriting from [**PreTrainedModel**](https://huggingface.co/transformers/main_classes/model.html#transformers.PreTrainedModel) for PyTorch and [**TFPreTrainedModel**](https://huggingface.co/transformers/main_classes/model.html#transformers.TFPreTrainedModel) for TensorFlow.
* **tokenizer** ([**PreTrainedTokenizer**](https://huggingface.co/transformers/main_classes/tokenizer.html#transformers.PreTrainedTokenizer)) – The tokenizer that will be used by the pipeline to encode data for the model. This object inherits from [**PreTrainedTokenizer**](https://huggingface.co/transformers/main_classes/tokenizer.html#transformers.PreTrainedTokenizer).
* **modelcard** (**str** or **ModelCard**, *optional*) – Model card attributed to the model for this pipeline.
* **framework** (**str**, *optional*) – The framework to use, either **"pt"** for PyTorch or **"tf"** for TensorFlow. The specified framework must be installed.

If no framework is specified, will default to the one currently installed. If no framework is specified and both frameworks are installed, will default to the framework of the **model**, or to PyTorch if no model is provided.

* **task** (**str**, defaults to **""**) – A task-identifier for the pipeline.
* **args\_parser** ([**ArgumentHandler**](https://huggingface.co/transformers/internal/pipelines_utils.html#transformers.pipelines.ArgumentHandler), *optional*) – Reference to the object in charge of parsing supplied pipeline parameters.
* **device** (**int**, *optional*, defaults to -1) – Device ordinal for CPU/GPU supports. Setting this to -1 will leverage CPU, a positive will run the model on the associated CUDA device id.
* **binary\_output** (**bool**, *optional*, defaults to **False**) – Flag indicating if the output the pipeline should happen in a binary format (i.e., pickle) or as raw text.

**Returns**

Each result comes as a dictionary with the following keys:

* **generated\_text** (**str**, present when **return\_text=True**) – The generated text.
* **generated\_token\_ids** (**torch.Tensor** or **tf.Tensor**, present when **return\_tensors=True**) – The token ids of the generated text.

**Return type**

* A list or a list of list of **dict.**