A guide to language model sampling in AllenNLP

How Stochastic Beam Search can add ~creativity~ to your generated text

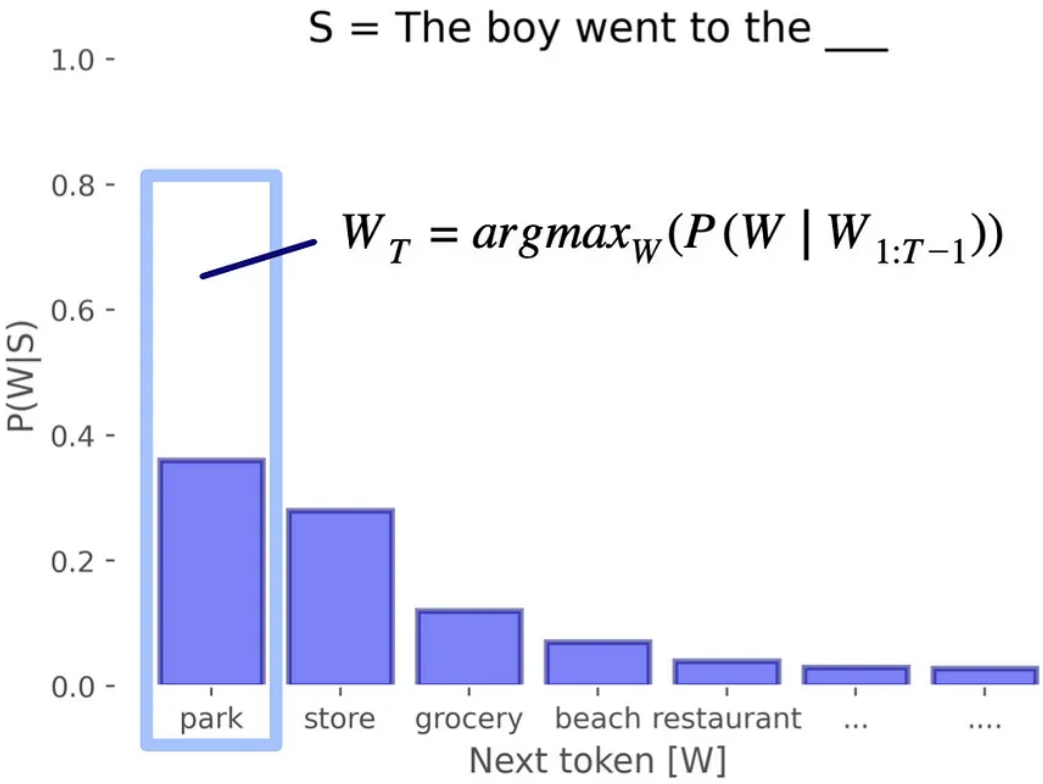
Jackson Stokes · Follow
Published in AI2 Blog
10 min read · Nov 17, 2020

With the increasing power of transformer-based language models, we see computer-generated text becoming more coherent. These language models give text generation algorithms their "smarts" by providing an understanding of how likely each word is to follow an existing sequence of words. However, the process of selecting subsequent words still holds some tricky nuances that can be the difference between generated text which sounds *human*, or more like computer

gibberish. In this article, we're going to go over a few of the different ways we can select subsequent words, including Stochastic Beam Search, and how they can add some ~creativity~ to your generated text in AllenNLP.

Generating without sampling (greedy search, beam search)

Before we introduce the concept of sampling, let's briefly cover the alternative: selecting the most probable sequence or next token. This can be done greedily by selecting the highest probability word at each step; for example, say I want to find the next word following the sequence "The boy went to the ". We will take the word "park" because it has the highest probability of following:

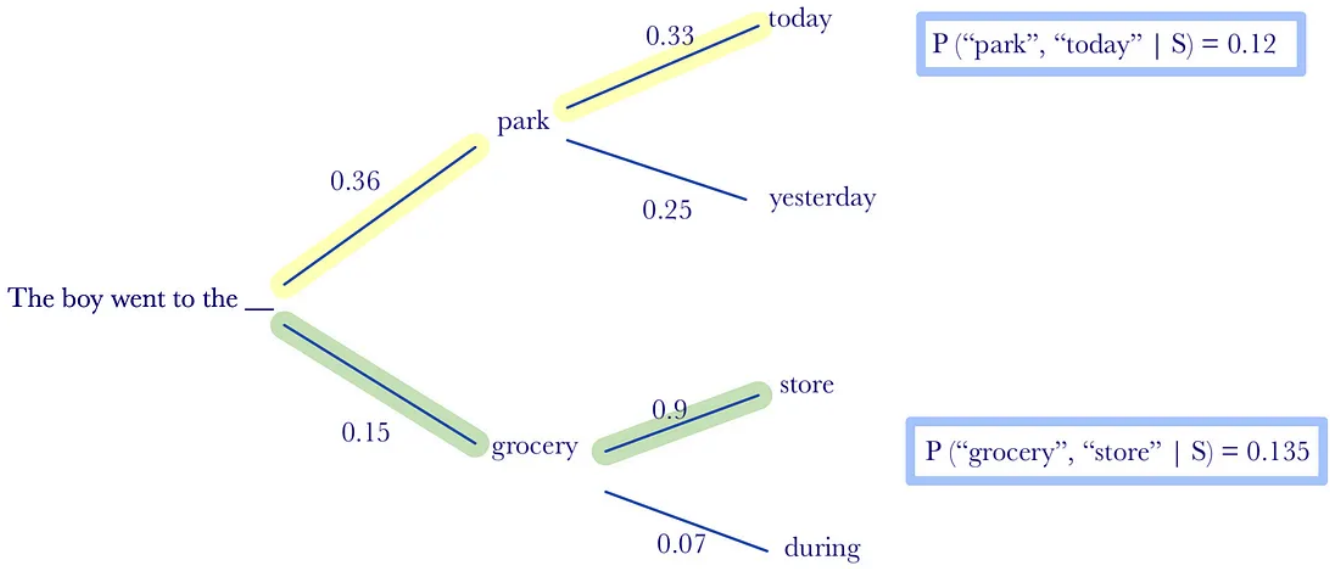


Here, we denote that the next word will have the greatest conditional probability given the previous words. Notation: W_i is the ith word in the sequence. T is the length of our sequence S including the new word.

However, sometimes the best sequences start off with words that don't have the highest probability. For example, what if the best sequence starts with the word "store" or "grocery"?

To address this, we use a technique called beam search. It works by looking at several (say, five) candidate sequences called beams, and considers several (again, five) next-best words for each beam. Then, it compares all of these (five beams * five words = 25 candidates) and selects the top five candidates by probability. These become the beams in the next step, now one word longer than before. This has the

advantage of giving the best sequences a few different possible beginnings. In the example below, note that during greedy search, we would select the words “park today”. However, a high probability word “store” is hidden behind a lower probability initial word “grocery”, and the words “grocery store” actually make up the best sequence.



This has the advantage of exposing high-probability sequences that might be hidden behind lower probability initial words. If we used the greedy search above, we never would have looked past the initial low-probability word, missing out on the most probable full sequence.

Now, lets use beam search within AllenNLP to see these generations in action:

```python
from allennlp_models.pretrained import load_predictor
predictor = load_predictor(
    "lm-next-token-lm-gpt2",
    overrides={
        "model.beam_search_generator": {
            "type": "transformer",
            "namespace": "gpt2",
            "beam_search": {
                "end_index": 50256,
                "max_steps": 25,
                "beam_size": 1,
            },
        }
    },
)
print("Deterministic beam search:")
text = "I went into town on Saturday morning because..."
print(text)
for tokens in predictor.predict(text)["top_tokens"]:
    string = "".join(tokens).replace("Ġ", " ").replace("<|endoftext|>", "").strip()
    print(" ->", string)
```

sample_beam_search.py hosted with ❤️ by GitHub view raw

```
Deterministic beam search:
I went into town on Saturday morning because...
-> I was going to go to the gym and I was going to go to the gym and
I was going to go to the ..
```

Beam search is great at finding high probability sequences, but both greedy search and beam search often run into problems with repetition, decoding the same high-probability sequence over and over again.

Human conversations (hopefully) don't consist of the exact same sentence repeated over and over again, so in order to make our generated text sound a bit more human, we need to help it break out of the pattern of selecting only the most probable word each time. There are a few ways to do this, such as rules to alternate selected tokens, or prevent repeated *n-grams* (i.e. making sure each string of 5 words can only be included once), but in this article we're going to focus on introducing randomness through sampling.

Text generation with sampling

Sampling, in this context, refers to randomly selecting the next token based on the probability distribution over the entire vocabulary given by the model. This means that every token with a non-zero probability has a chance of being selected. For example, the probability distribution of the next token for "*the boy went to the*" might be:

In this hypothetical example, we see that while the most likely subsequent word is "*park*", with P("*park*"| *S*) =0.36, it's still more probable that the selected word will be something else, 1-P("*park*"| *S*) = 0.64. This adds diversity to our samples, at the cost of reducing the total probability of our sequence when a low-probability word is chosen.

What if we want to make our sampler more likely to choose the high probability words, and less likely to choose the low probability words? We can make the PMF *sharper* by lowering the `temperature` parameter, which reduces the entropy of the resulting Softmax output:

Within low entropy distributions, sampling with replacement will produce many duplicates, and leveraging deterministic beam search will result in low variability.

Let's go back to the original non-sharpened PMF above. Even though this distribution has two high probability words, the cumulative probability of the thousands of other possible words sums up to ~0.35. Even though each of these words is low probability (and most wouldn't make for a great sequence) the effect of the long tail means that there is still a significant chance that a rather low probability word could be chosen, which results in some strange outputs. For example, with a basic multinomial sampler, we have:

```python
from allennlp_models.pretrained import load_predictor
predictor = load_predictor(
    "lm-next-token-lm-gpt2",
    overrides={
        "model.beam_search_generator": {
            "type": "transformer",
            "namespace": "gpt2",
            "beam_search": {
                "sampler": {
                    "type": "multinomial",
                },
                "end_index": 50256,
                "max_steps": 25,
                "beam_size": 1,
            },
        }
    },
)
print("Multinomial Sampling:")
text = "I went into town on Saturday morning because..."
print(text)
for tokens in predictor.predict(text)["top_tokens"]:
    string = "".join(tokens).replace("Ġ", " ").replace("<|endoftext|>", "").strip()
    print(" ->", string)
```

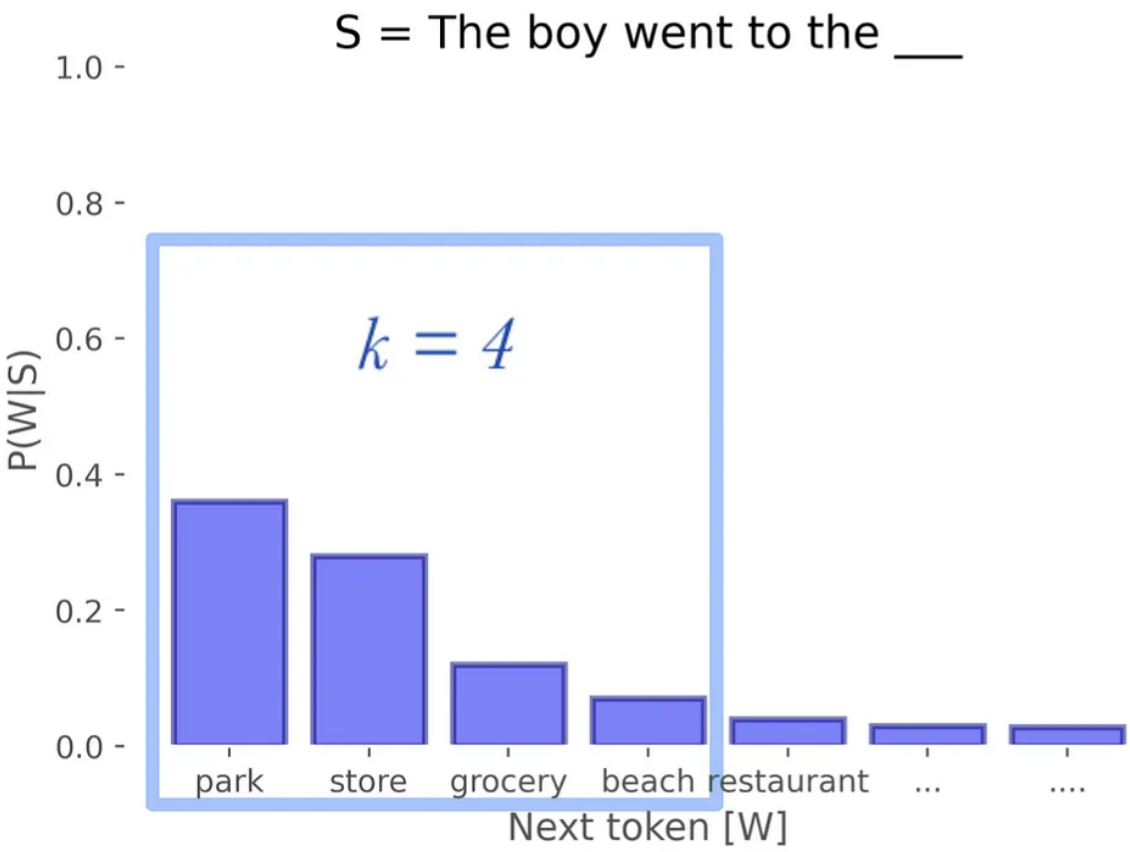
multinomial_sample.py hosted with ❤ by GitHub view raw

```
Multinomial Sampling:
I went into town on Saturday morning because...
-> I have to wear suits and collared in the South Bay. This was
shocking!" "This is our city. First of all, I'm strange in the name
of Santa, Howard Daniel, and
```

In the above example, the word “collared” is definitely out of place, and it was likely in the long-tail of low probability words. To remove the chance that one of these very low probability words would be chosen, we can employ one of two different sampling techniques: Top K or Top P (also known as Nucleus) Sampling.

Top K sampling

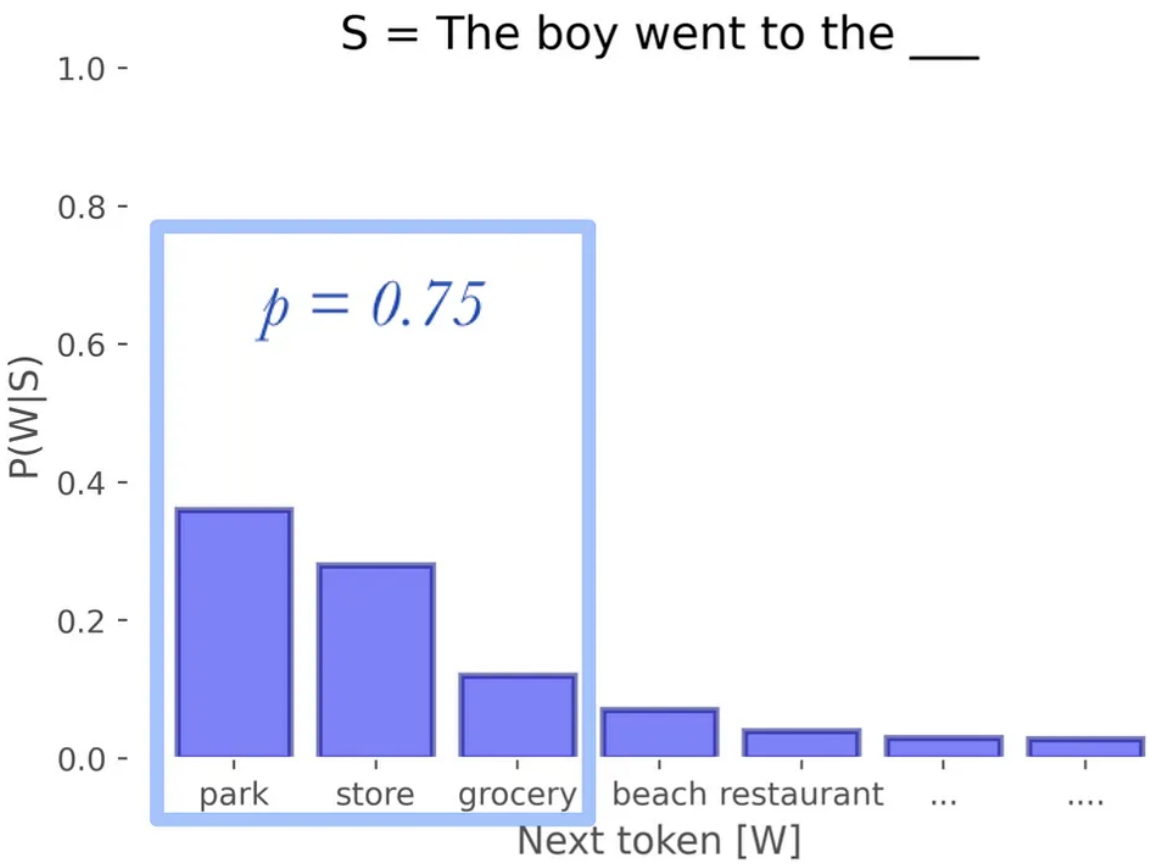
In Top K sampling, the idea is to only sample from the *k* most probable tokens. For example, suppose we set *k=4*. This means we redistribute the PMF over the four most probable tokens, and sample from those in the blue box:



This has the advantage of removing many the lowest probability words. However, picking a good value of *k* can be difficult as the distribution or words is different for each step. This means that the same value of *k* can allow for strange words to be included in one step, while also excluding reasonable words in a different step. In order to get around this, we can leverage a different sampling technique: Top P sampling.

Top P sampling

Another way to exclude very low probability tokens is to include the most probable tokens that make up the "nucleus" of the PMF, such that the sum of the most probable tokens just reaches *p*. For example, suppose we set *p = 0.75*; with Top P sampling, we would include the most probable next tokens until the cumulative probability until the sum reaches 0.75, at which point we stop including tokens.



Here, we consider park, store, and grocery, because those three satisfy our probability threshold of 0.75

This has the advantage of being flexible as the distribution changes, allowing the size of the filtered words to expand and contract when it makes sense.

Combining beam search with sampling

Sampling attempts to make generated text more interesting by adding lower-probability words. Beam search makes generated text more consistent by maximizing the full sequence probability. Ideally, we'd be able to combine the creativity of sampling with the stability of beam search, however directly combining them produces some issues.

First, recall that we trim down candidate beams based on their sequence probability. When we sample an interesting word (with lower probability), the overall probability of that candidate sequence is reduced. When the next beams are selected from candidates, it's unlikely that this beam would be included because the other beams that chose high probability words in that step would most likely have a higher sequence probability. This has the effect of compounding the effect of low probability: a low-probability word has to be chosen first when sampling from the language model, then that now lower probability beam must be chosen again and again at each subsequent step.

This means that our final beams have little to no variation early in the sequence, and often only include variation near the end of the sequence, because the effect of slightly low probability has fewer steps to compound. To demonstrate this, let's look at some generations using Top P sampling:

```python
from allennlp_models.pretrained import load_predictor
predictor = load_predictor(
    "lm-next-token-lm-gpt2",
    overrides={
        "model.beam_search_generator": {
            "type": "transformer",
            "namespace": "gpt2",
            "beam_search": {
                "sampler": {
                    "type": "top-p",
                    "p": 0.9,
                },
                "end_index": 50256,
                "max_steps": 15,
                "beam_size": 4,
            },
        }
    },
)
print("Top P sampling:")
text = "I went into town on Saturday morning because..."
print(text)
for tokens in predictor.predict(text)["top_tokens"]:
    string = "".join(tokens).replace("Ġ", " ").replace("<|endoftext|>", "").strip()
    print(" ->", string)
```

top_p_sample.py hosted with ❤ by GitHub view raw

```
Top P sampling:
I went into town on Saturday morning because…
 -> I didn't know what to do. I didn't know what to do. I was
 -> I didn't know what to do. I didn't know if I was going to be
 -> I didn't know what to do. I didn't know what to do with my life
 -> I didn't know what to do. I didn't know what to do, and I
```

If we want the best of both worlds, with the interesting, creative text of low-probability sequences, combined with the quality control of beam search, we need a

way to commit to the interesting words that we sampled in earlier steps. In the next section, we explore a way to do just that.

Stochastic Beam Search

Stochastic Beam Search is a method of sampling sequences of length *n* from their true probability distribution. It sounds similar to what's covered above, however sampling is done using a different approach. Rather than sample from a distribution based on its PMF, it instead adds randomly generated noise to the true sequence log-probabilities, creating perturbed log-probabilities. It then selects the sequences with the highest perturbed probabilities. The important thing here is that our process of selecting the best sequences is the same as in beam-search, just taking the top-*k* *perturbed* log-probabilities instead of true log probabilities. The result? The added noise is carried forward, which effectively removes the compounding effect of low probability words, making our final sequences much more acceptable of creative words.

Now that we have a have a high-level understanding of what's going on, let's look at Stochastic Beam Search in a bit more detail. This process uses the 'Gumbel max trick' at each step, in which we add Gumbel-distributed noise to the normalized log probabilities of each token, conditioned on the perturbed probability of the parent sequence, then select the candidate sequences with the highest perturbed log probability.

Huh?

Let's break down what this means. *Gumbel-distributed noise* refers to random samples from the Gumbel Distribution, which is traditionally used to model the maxima of a set of samples. By taking the top-*k* sequences from the Gumbel max trick, we effectively sample *k* instances *without replacement* from a categorical distribution. Again, this is done by simply perturbing the log-probabilities of each sequence by adding independent Gumbel distributed noise, and taking the categories with the highest perturbed log-probabilities.

This process of taking the top-*k* perturbed log probabilities is useful for us because it allows for diverse samples to be chosen (if, for example, a large amount of noise is added to a slightly lower probability word, propelling it into the top-*k*) while removing the issue with diverse words having a low chance of being re-selected each step (because the high perturbed probability is carried forward to future steps.) Finally, by

taking the highest perturbed probability beams at each step, we preserve the quality of the generated language.

We can see the impact this makes in our generated beams. We have significantly more diverse samples, but we still retain consistent control of language.

```python
from allennlp_models.pretrained import load_predictor
predictor = load_predictor(
    "lm-next-token-lm-gpt2",
    overrides={
        "model.beam_search_generator": {
            "type": "transformer",
            "namespace": "gpt2",
            "beam_search": {
                "sampler": {
                    "type": "gumbel",
                },
                "end_index": 50256,
                "max_steps": 18,
                "beam_size": 4,
            },
        }
    },
)
text = "I went into town on Saturday morning because..."
print(text)
for tokens in predictor.predict(text)["top_tokens"]:
    string = "".join(tokens).replace("Ġ", " ").replace("<|endoftext|>", "").strip()
    print(" ->", string)
```

gumbel_sampling.py hosted with ❤ by GitHub view raw

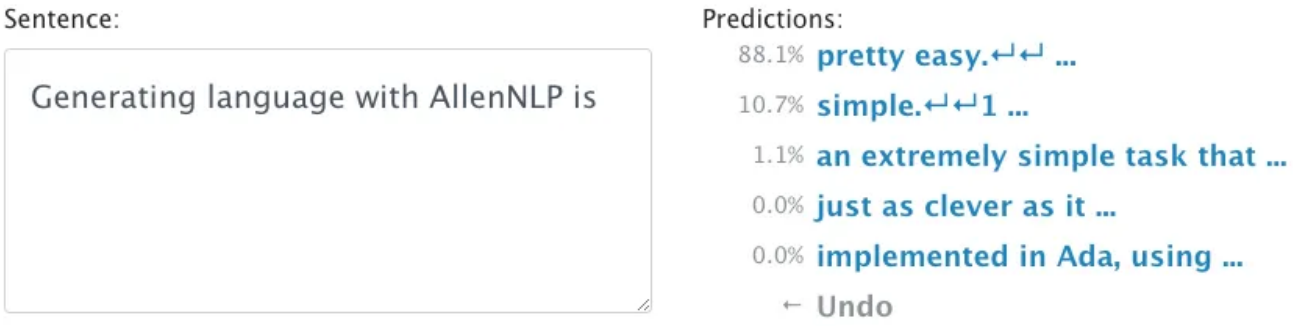
```
I went into town on Saturday morning because...
-> I was eager to buy a new B&O because I thought it would be a nice
-> I was tired, which is kind of why I went to the quarters and
having a laugh
-> I intended to instead attend one of Wednesday's protests to
campaign for retirement rights for workers,
-> her San Gabriel Valley speech was first class. Rather than having
your eyes and ears soaked with
```

Wow! These sequences contain a lot more variation than we saw with traditional sampling techniques, and as you play around with beam sizes larger than four, you can see increased reliability in the control of language. This example uses GPT-2 for

its language model, but these generations can be even further improved by using more sophisticated language models. To gain a more detailed understanding of Stochastic Beam Search, we recommend reading the paper which presents it.

Try it out!

Stochastic beam search (and other sampling techniques) are available in the `BeamSearch` class within AllenNLP! To get started right away, you can also play around with stochastic beam search paired with the GPT-2 language model in the AllenNLP language modeling demo. It's an easy and enjoyable way to play around with language generation without writing a single line of code. But don't let us tell you, let stochastic beam search tell you:



References

Stochastic Beams and Where to Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without...

The well-known Gumbel-Max trick for sampling from a categorical distribution can be extended to sample k elements...

arxiv.org

The Gumbel Trick

Until I read the recent paper at ICML 2017, I hadn't heard of the Gumbel trick. There is surprisingly little online...

irenechen.net

Graduate Descent

Goal: Sampling from a discrete distribution parametrized by unnormalized log-probabilities: $$ \pi_k = \frac{1}{z}...

timvieira.github.io

The Gumbel trick

Quantities of the form \(\displaystyle \log \Big(\sum_{i=1}^n \exp(x_i) \Big)\) for \(x \in \mathbb{R}^n\), often...

francisbach.com

Follow @allen_ai on Twitter and subscribe to the AI2 Newsletter to say current on news and research coming out of AI2.

NLP | Artificial Intelligence | Machine Learning | Allennlp

Follow

Written by Jackson Stokes

8 Followers · Writer for AI2 Blog

Computer Science & Math student at the University of Washington

More from Jackson Stokes and AI2 Blog

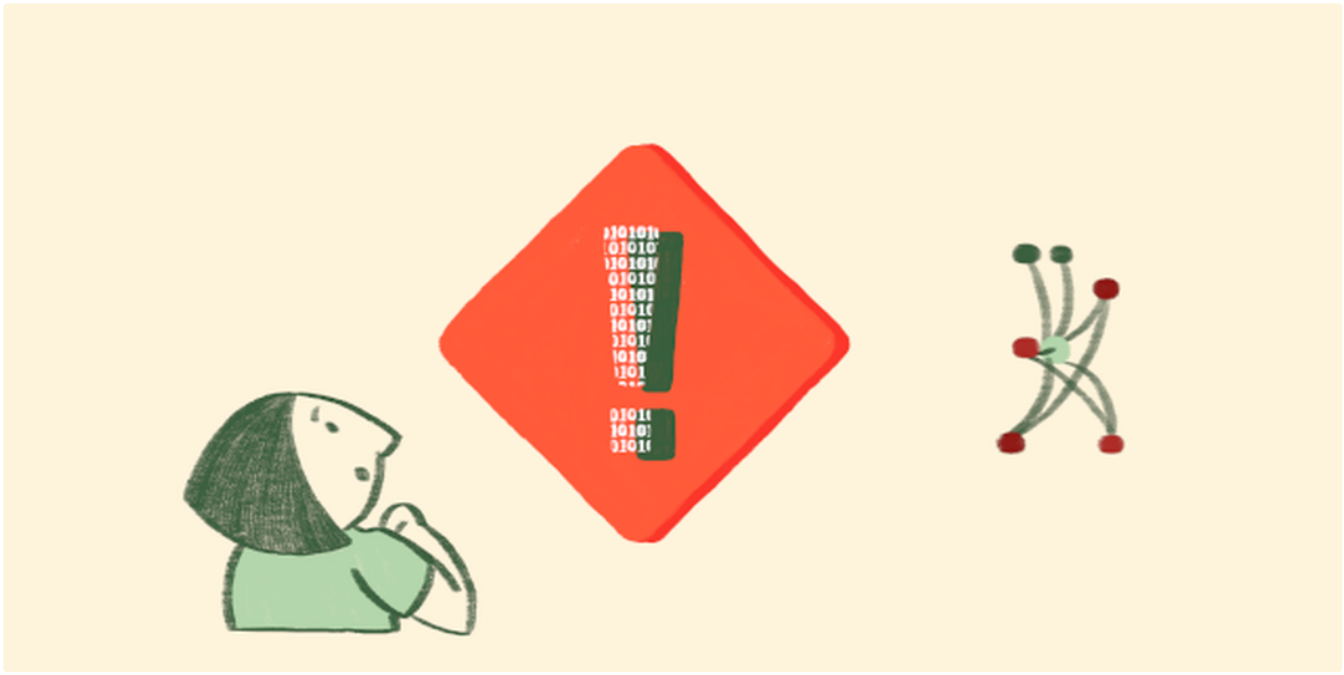
Jackson Stokes

How to price your Airbnb

A data-driven approach to finding the best price for your unique listing

9 min read · Dec 13, 2021

2



Maria Antoniak in AI2 Blog

Using Large Language Models With Care

How to be mindful of current risks when using chatbots and writing assistants

11 min read · Jun 20

59

Bill Yuchen Lin in AI2 Blog

SwiftSage: Building AI Agents for Complex Interactive Tasks via Fast and Slow Thinking with LLMs

SwiftSage, a novel AI agent inspired by fast-and-slow thinking, designed to optimize LLMs for planning for complex interactive tasks.

5 min read · Jun 21

13

AI2 in AI2 Blog

Allen Institute for Artificial Intelligence (AI2) Announces New CEO

AI Researcher, Executive, and Forbes Top 5 AI Entrepreneur Ali Farhadi to Lead Institute's Next Chapter

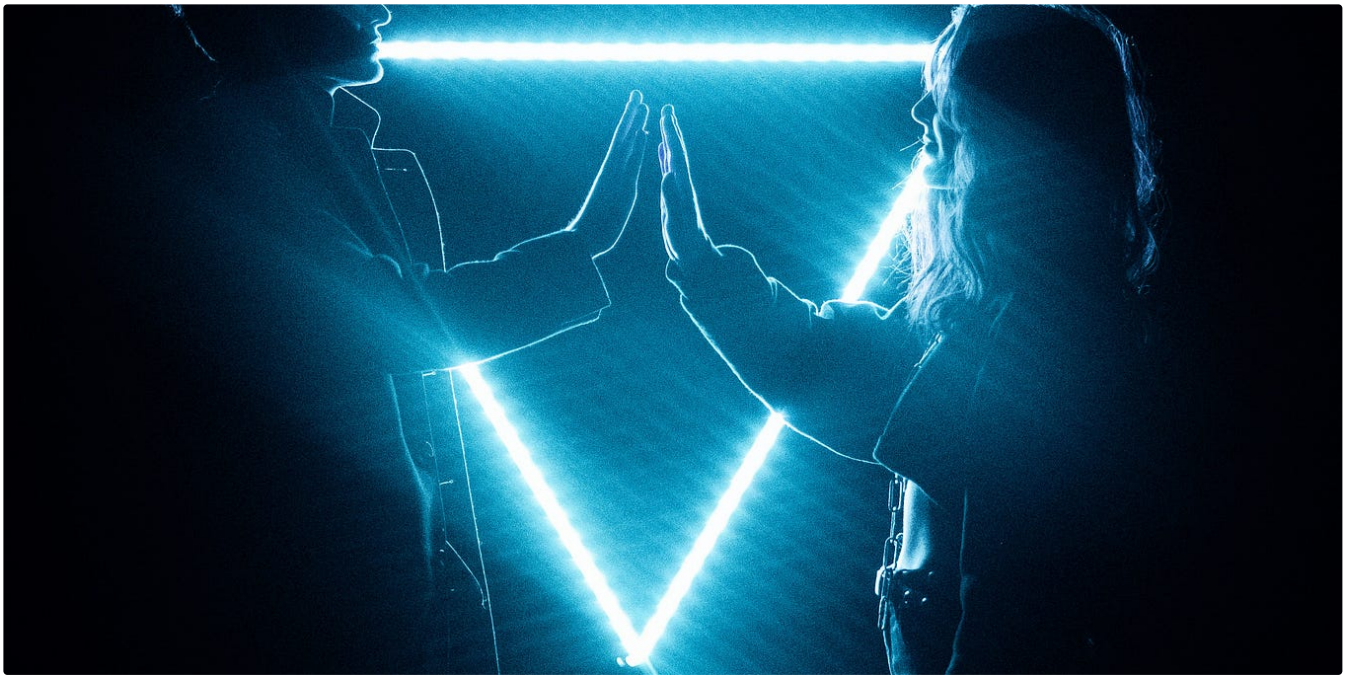
3 min read · Jun 20

2

See all from Jackson Stokes

See all from AI2 Blog

Recommended from Medium



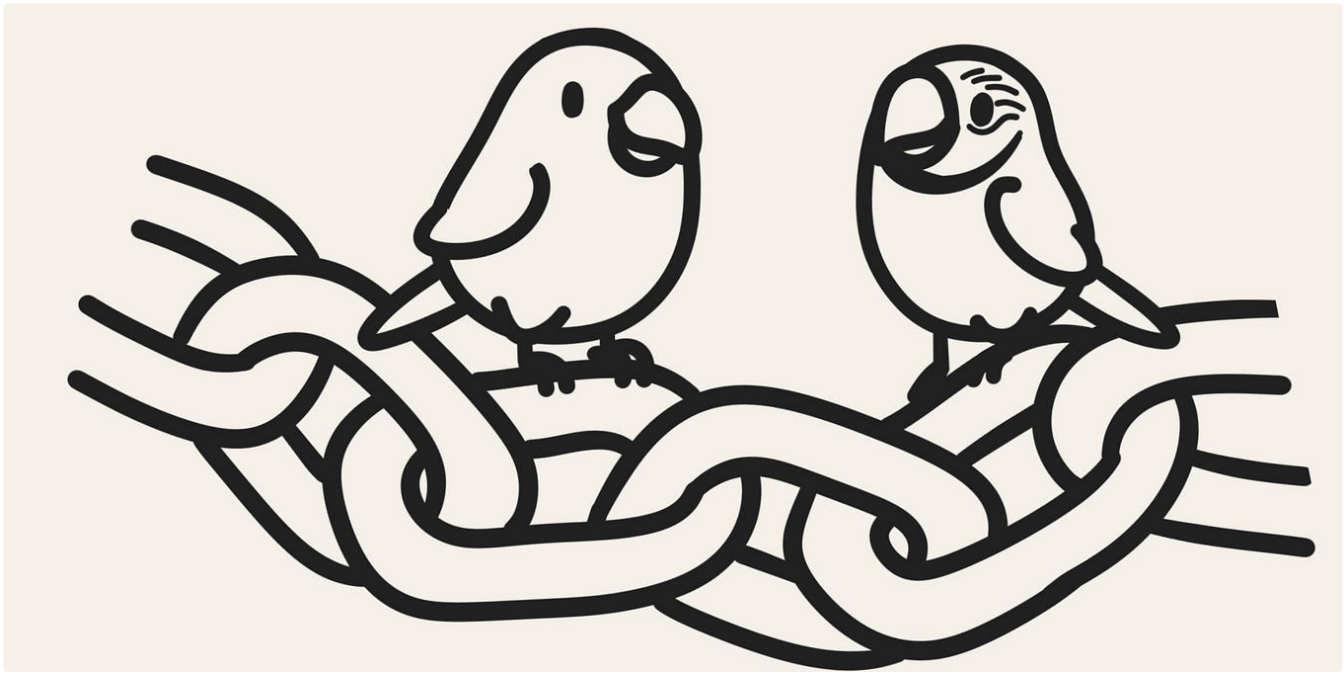
Wouter van Heeswijk, PhD in Towards Data Science

Proximal Policy Optimization (PPO) Explained

The journey from REINFORCE to the go-to algorithm in continuous control

· 13 min read · Nov 29, 2022

178 2



Leonie Monigatti in Towards Data Science

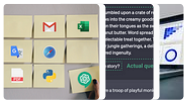
Getting Started with LangChain: A Beginner's Guide to Building LLM-Powered Applications

A LangChain tutorial to build anything with large language models in Python

✨ · 12 min read · Apr 25

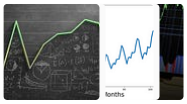
3.3K 20

Lists



Natural Language Processing

379 stories · 36 saves



Predictive Modeling w/ Python

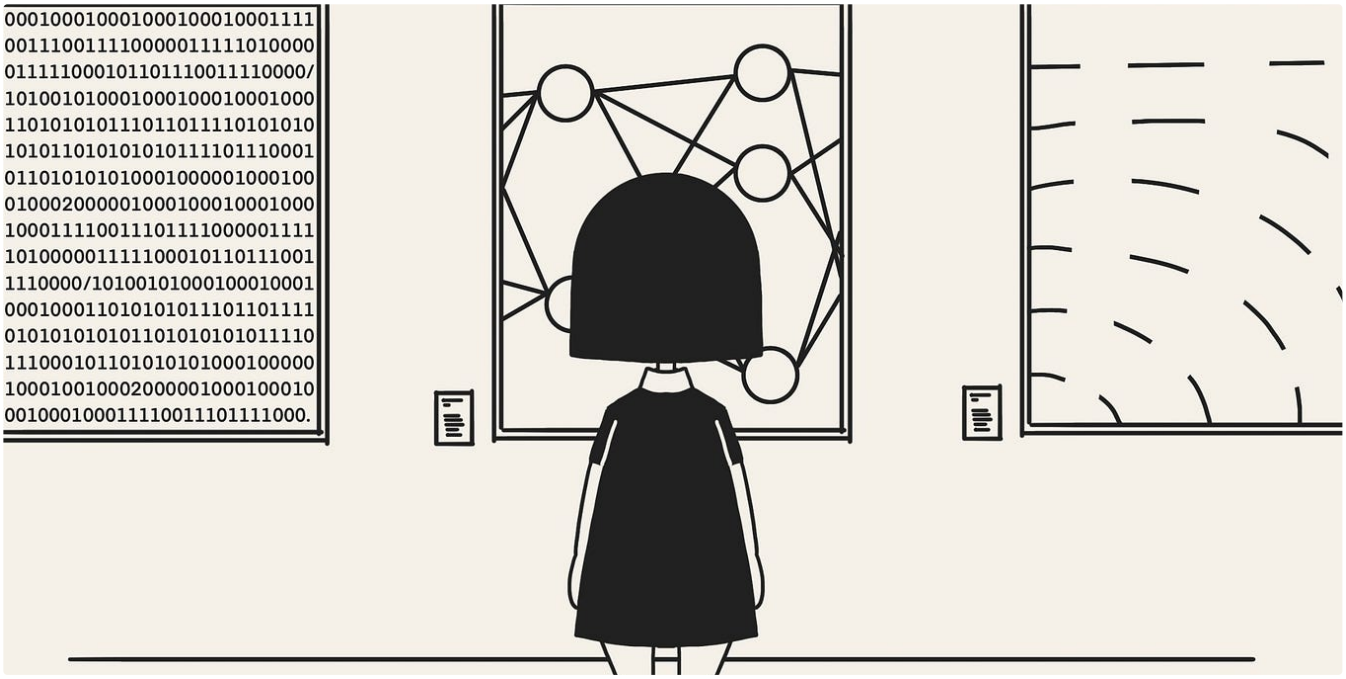
18 stories · 88 saves

AI Regulation

6 stories · 22 saves

Practical Guides to Machine Learning

10 stories · 101 saves



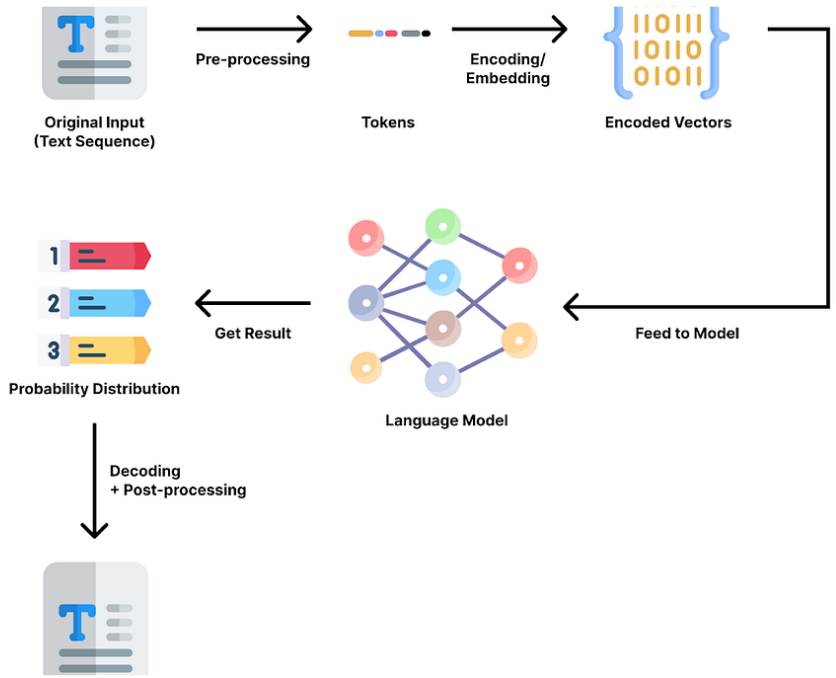
Leonie Monigatti in Towards Data Science

10 Exciting Project Ideas Using Large Language Models (LLMs) for Your Portfolio

Learn how to build apps and showcase your skills with large language models (LLMs). Get started today!

✦ · 11 min read · May 15

1.7K 9



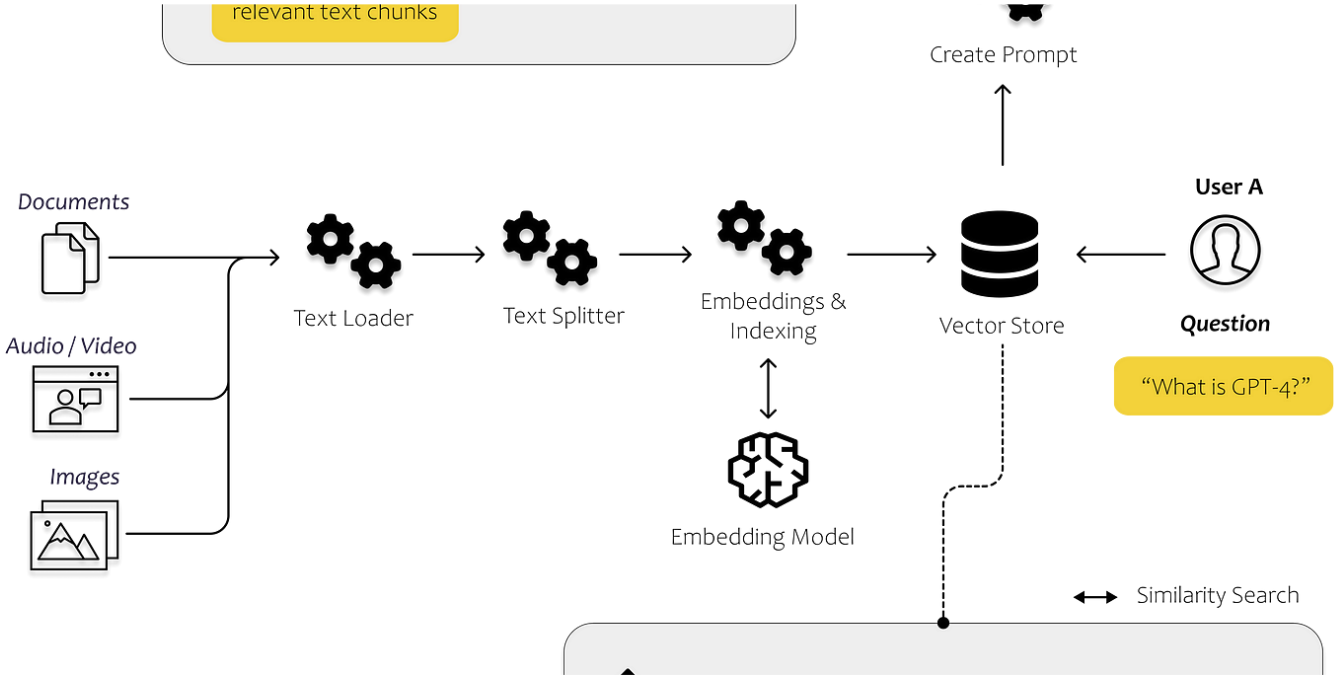
Guodong (Troy) Zhao in Bootcamp

How ChatGPT really works, explained for non-technical people

The transformer and GPT model explained for non-technical people with examples

✦ · 12 min read · Feb 21

1.3K 18



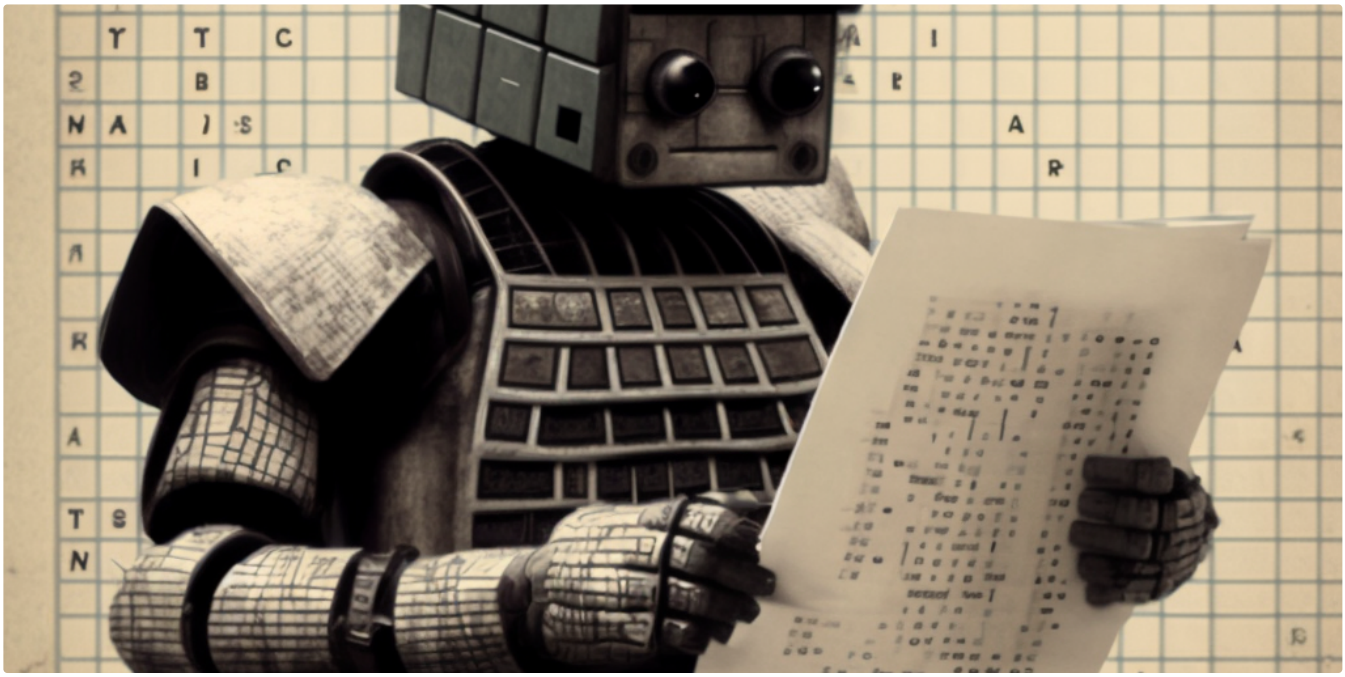
Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

· 25 min read · Jun 22

1.6K 16



The Jasper Whisperer in The Generator

The Dummy Guide to ‘Perplexity’ and ‘Burstiness’ in AI-generated content

Understanding Language Models: A Simplified Guide

✨ · 6 min read · Feb 17

271 5