

CHAPTER 2

Understanding LLMs

So you want to become the LLM whisperer who unlocks the wealth of their knowledge and processing power with clever prompts? Well, to appreciate which kinds of prompts *are* clever and tease the right answer from the LLM, you first need to understand how LLMs process information—how they *think*.

In this chapter, we'll approach this problem onion style. You'll first see LLMs from the very outside as trained mimics of text in “[What Are LLMs?](#)” on page 16. You'll learn how they split the text into bite-size chunks called tokens in “[How LLMs See the World](#)” on page 22, and you'll learn about the fallout if they can't easily accomplish that split.

You'll also find out how the token sequences are generated bit by bit in “[One Token at a Time](#)” on page 29, and you'll learn about the different ways to choose the next token in “[Temperature and Probabilities](#)” on page 32. Finally, in “[The Transformer Architecture](#)” on page 37, you'll delve into the very inner workings of an LLM, understand it as a collection of minibrain that communicate through a Q&A game called *attention*, and learn what that means for prompt order.

During all that, please keep in mind that this is a book about *using* LLMs, not about LLMs themselves. So, there are a lot of cool technical details that we're *not* mentioning because they're not relevant for prompt engineering. If you want matrix multiplications and activation functions, you'll need to turn elsewhere—the classic reference [The Illustrated Transformer](#) is an excellent starting point for a deep dive. But we promise you won't need that amount of technical background if all you want to do is write great prompts—so let's dive into what you do need to know.

What Are LLMs?

At the most basic level, an *LLM* is a service that takes a string and returns a string: text in, text out. The input is called the *prompt*, and the output is called the *completion* or sometimes, the *response* (see [Figure 2-1](#)).

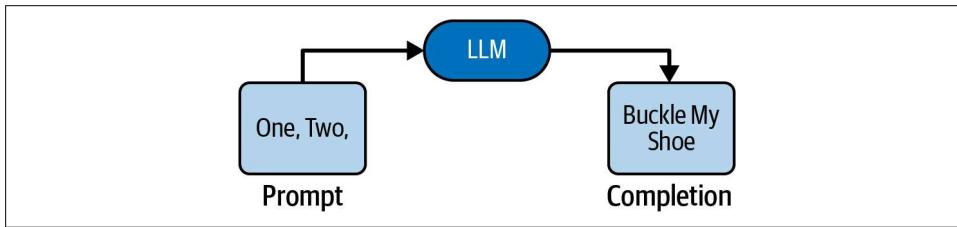


Figure 2-1. An LLM taking the prompt “One, Two,” and presenting the completion “Buckle My Shoe”

When an untrained LLM first sees the light of day, its completions will look like a pretty random jumble of unicode symbols and bear no clear relationship to the prompt. It needs to be *trained* before it’s useful. Then, the LLM won’t just answer strings with strings but language with language.

Training takes skill, compute, and time far beyond the scope of most project groups, so most LLM applications use off-the-shelf generalist models (known as *foundation models*) that are already trained (maybe after a bit of fine-tuning; see the sidebar). So, we don’t expect you to train an LLM yourself—but if you want to use an LLM, especially programmatically, it is essential for you to understand what it *has been trained* to do.

What Is Fine-Tuning?

Training LLMs takes lots of data and compute, although many basic lessons, such as the rules of English grammar, don’t differ much between the training sets. It’s therefore common not to start completely from scratch when training an LLM but to start with a copy of a different LLM, possibly one that’s trained on different documents.

For example, the early versions of OpenAI Codex (an LLM for producing source code that was developed for GitHub Copilot) were copies of an existing model (GPT-3, a natural language LLM) that were fine-tuned with lots of source code published on GitHub.

If you have such a model trained on dataset A and fine-tuned on dataset B, your prompts should normally be written as if it had been trained on B outright. We’ll delve deeper into fine-tuning in [Chapter 7](#).

LLMs are trained using a large set of documents (again, strings) known as the *training set*. The kind of documents depends on the purpose of the LLM (see Figure 2-2 for an example). The training set is often a mixture of different training inputs such as books, articles, conversations on platforms such as Reddit, and code on sites such as GitHub. From the training set, the model is supposed to learn how to produce output that looks just like the training set. Concretely, when the model receives a prompt that is the beginning of a document from its training set, the resulting completion should be the text that is most likely to continue the original document. In other words, models mimic.

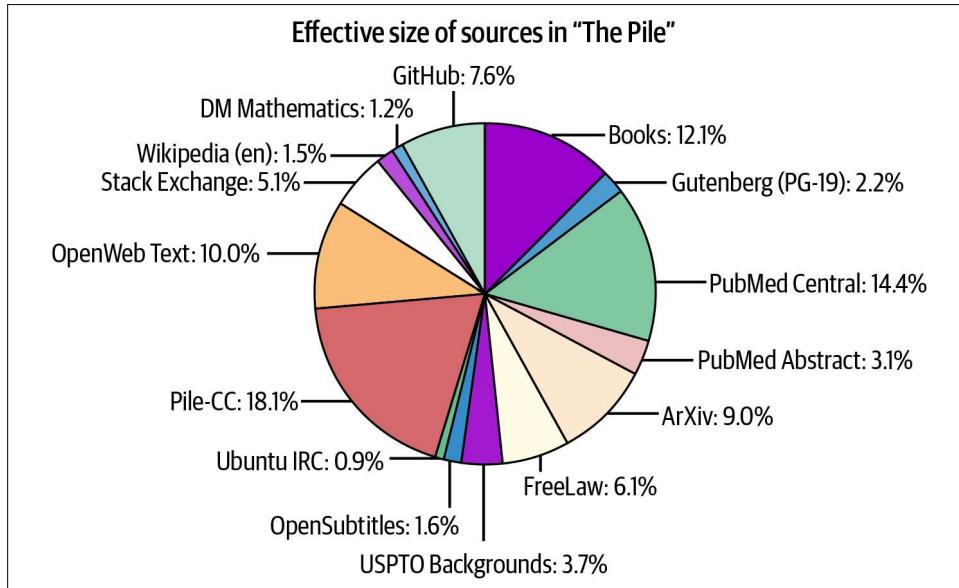


Figure 2-2. Composition of “*The Pile*”, a popular open source training set comprising a mixture of factual prose, fictional prose, dialogues, and other internet content

So, how’s an LLM different from, say, a big search engine index full of the training data? After all, a search engine would *ace* the task the LLM was trained with—given the beginning of a document, it could find a completion for that document with 100% accuracy. And yet, having a search engine that just parrots the training set isn’t the goal here: the LLM shouldn’t learn to recite the training set by heart but to apply the patterns it encounters there (in particular, logical and reasoning patterns) to complete any prompt, not just those from the training set. Mere rote memorization is considered a defect. Both the inner architecture of the LLM (which encourages it to abstract from concrete examples) and the training procedure (which tries to feed it diverse, nonrepetitive data and measures success on unseen data) are supposed to prevent this defect.

That prevention sometimes fails, and instead of learning facts and patterns, the model learns chunks of text by rote—which is known as *overfitting*. Large-scale overfitting should be rare in off-the-shelf models, but it's worth being aware of the possibility that if an LLM seemingly solves a problem that it's seen during training, it doesn't necessarily mean that the LLM will do as well when confronted with a similar problem it hasn't seen before.

Nevertheless, after you work with LLMs for a while, you start to develop an intuition for how an LLM will behave based on the task it was trained on. So when you want to know how a given prompt might be completed, don't ask yourself how a reasonable person would "reply" to the prompt but rather how a document that happens to start with the prompt might continue.



Assume you have picked a document from the training set at random. All you know about it is, it starts with the prompt. What is the statistically most likely continuation? That's the LLM output you should expect.

Completing a Document

Here's an example of reasoning about document completions. Consider the following text:

Yesterday, my TV stopped working. Now, I can't turn it on at

For a text that starts like this, what might be the statistically most likely completion?

1. y2ior3w
2. Thursday.
3. all.

None of these completions are absolutely *impossible*. Sometimes, a cat runs over the keyboard and completion 1 is generated, and other times, a sentence gets garbled in rewriting and 2 appears. But by far the most likely continuation is 3, and almost all LLMs will choose this continuation.

Let's take completion 3 as given and run the LLM a bit further:

Yesterday, my TV stopped working. Now, I can't turn it on at all.

For a text that starts like that, what is the statistically most likely completion?

- a. This is why I chose to settle down with a book tonight.
- b. Shall we watch the game at your place instead?
- c. \n

\n

First, try unplugging the TV from the wall and plugging it back in.

Well, it depends on the training set. Let's say the LLM was trained on a dataset of narrative prose such as short stories, novels, magazines, and newspapers—in that case, completion *a*, about reading a book, sounds rather more likely than the others. While the sentence about the TV, followed by the question from completion *b*, could well appear somewhere in the middle of a story, a story wouldn't open with this question without at least the starting quotation marks (""). So it's unlikely that a model trained on short stories would predict option *b*.

But throw emails and conversation transcripts into the training set, and suddenly, option *b* appears very plausible. I made up both of them, though: it's the third option that was produced by an actual LLM (OpenAI's text-davinci-003, which is a variant of GPT-3), mimicking the advice and customer service conversations that abound in its training set.

A theme is emerging here: the better you know the training data, the better the intuition you can form about the likely output of an LLM trained on that training data. Many commercial LLMs don't publish their training data—choosing a good training set is a big part of the special sauce that makes their models successful. Even then, however, it's usually possible to form some sensible expectations about the kind of documents the training set consists of.

Human Thought Versus LLM Processing

The LLM selects the most likely looking continuation, and this goes against some assumptions humans make when reading text. That's because when humans produce text, they do so as part of a process that involves more than producing plausible-looking text output. Let's say you want to write a blog post about a podcast you came across at the podcasting site Acast. You might start writing the following: In their newest installment of 'The rest is history', they talk about the Hundred Years' War (listen on acast at <http://>). Of course you don't know the URL by heart, so this is the point where you stop writing and do a quick internet search. Hopefully, you find the correct link: shows.acast.com/the-rest-is-history-podcast/episodes/321-hundred-years-war-a-storm-of-swords. Or maybe you can't find it, in which case, you might go back and delete the whole bracket and replace it with (episode unfortunately not available anymore).

The model can't google or edit, so it just guesses.¹ Nor will the raw LLM express any doubt,² add a disclaimer that it was just guessing, or show any other trace of evidence that the information is merely a guess rather than actual knowledge—because after all, the model *always* guesses.³ This guess just happened to be made at a point where humans typically switch to a different mode of producing their text (googling rather than pressing the first keys that come to mind).

LLMs are really good at emulating any patterns they find in the items they guess about. After all, this is pretty much exactly what they were trained for. So if they make up a Social Security number, it'll be a string of plausible digits, and if they make up the URL of a podcast, it'll look like the URL of a podcast.

In this case, I tried OpenAI's text-curie-001, a small variant of GPT3, and this LLM completed the URL as follows:

```
http://www.acast.com/the-rest-is-history-episode-5-the-Hundred-Years-War- \
1411-1453-with-dr-martin-kemp)
```

Is Dr. Martin Kemp a real person here? Maybe one who is involved with history podcasts? Maybe even the podcast we're talking about? There is an art historian named Martin Kemp at Oxford, though whether the completion could refer to him sounds like a theory of language problem rather than an LLM question (see [Figure 2-3](#)). At any rate, he didn't talk about the Hundred Years' War on the podcast *The Rest Is History*.

1 The model can't google *directly*, at least, but it can be connected to systems that can google. We will discuss this form of tool use in [Chapter 8](#).

2 In the next chapter, we'll introduce some ways in which raw LLMs are aligned or improved in post training and how these can add the ability to express doubt. However, this is not a native capacity of the basic LLM structure, which is the focus of this chapter.

3 It's true that the model can predict some parts with high certainty and some with low certainty. For example, it will be much more certain in predicting the next word of "John F. Kennedy was killed in the year," than it would be certain in predicting the next word of "Zacharias B. Fulltrodd was killed in the year." It reads a lot about the former death, while the second one, which is made up, could have taken place in any year. However, that uncertainty does not correlate with an expression of uncertainty or doubt in the training set—the model will fully buy into the assumption that there is a text that starts talking about Zacharias B. Fulltrodd's death. It has no reason to believe that this text is any more unreliable in relation to Zacharias's death than the typical JFK-related text it came across in its training set is in relation to JFK's death.

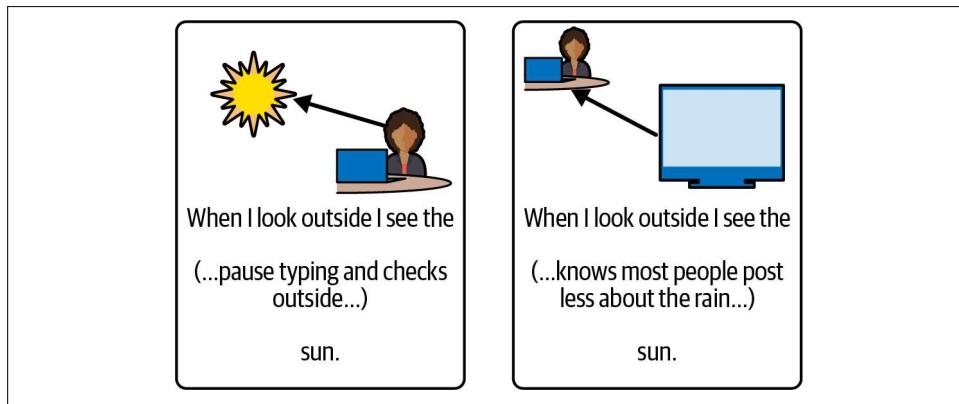


Figure 2-3. People's language reflects reality; models' language reflects people

Hallucinations

The fact that LLMs are trained as “training data mimic machines” has unfortunate consequences: *hallucinations*,⁴ which are factually wrong but plausible-looking pieces of information produced confidently by the model. They are a common problem when using LLMs, either ad hoc or within applications.

Since hallucinations don’t differ from other completions *from the perspective of the model*, prompt directives like “Don’t make stuff up” are of very limited use. Instead, the typical approach is to get the model to provide some background that can be checked. That could be an explanation of its reasoning,⁵ a calculation that can be performed independently, a source link, or keywords and details that can be searched for. For example, it’s much harder to check the sentence “There was an English king who married his cousin,” than “There was an English king who married his cousin, namely George IV, who married Caroline of Brunswick.” The best antidote to hallucinations is “Trust but verify” just minus the trust.

Hallucinations can also be induced. If your prompt references something that doesn’t exist, an LLM will typically continue to assume its existence. Documents that start out with wrong claims and then correct themselves halfway through are rare. So the model will typically assume its prompt to be true, and this is known as *truth bias*.

⁴ Although the closest human analog to what’s going on is probably the psychological phenomenon of confabulation, rather than hallucination.

⁵ You can check this by making a second query to the LLM. See [Chapter 7](#).

You can make truth bias work for you—if you want the model to assess a hypothetical or counterfactual situation, there's no need to say, “Pretend that it's 2030 and Neanderthals have been resurrected.” Just begin with “It's 2031, a full year since the first Neanderthals were resurrected.”



If you have access to an LLM producing completions (i.e., the raw LLM, not wrapped in a chat interface like ChatGPT), this might be a good occasion to try out entering a couple of so-called *make-believe* prompts.

Like the example about resurrected Neanderthals in preceding text, make-believe prompts elicit answers to hypothetical questions not by asking the question outright but by implying that the hypothetical scenario actually came to pass.

Compare the suggestion with a chat LLM's answer. How does it differ?

However, an LLM's truth bias is also dangerous, particularly to programmatic applications. It's all too easy to mess up in programmatic prompt creation and introduce counterfactual or nonsensical elements. A human might read through the prompt, put down the paper, raise their eyebrows at you, and go, “Really?” The LLM doesn't have this option. It'll do its best to pretend the prompt is real, and it's unlikely to correct you. So you are responsible for giving it a prompt that doesn't need correction.

How LLMs See the World

In “[What Are LLMs?](#)” on page 16 you learned that LLMs consume and produce strings. It's worth getting under the hood on this statement a bit: how do LLMs see strings? We're used to thinking of strings as sequences of characters, but that's not quite what the LLM sees. It can reason about characters, but that's not a native ability, and it requires the equivalent of rather deep concentration on the part of the LLM—at the time of writing (autumn 2024), even the most advanced models can still be fooled by questions such as “[How many Rs in ‘strawberry’?](#)”.

Maybe it's worth pointing out that we don't really read strings in characters either. At a very early stage of human processing, they are grouped together into words. What we then read are the words, not the letters. That's why we often read over typos without spotting them: they're already corrected by our brain by the time they reach the conscious part of our processing.

You can have lots of fun with purposely garbled sentences just at the edge of what your inner autocorrect function can cope with (see [Figure 2-4](#), left). However, if you garble the text in a way that doesn't respect word boundaries, your readers are going to have a very bad day (see [Figure 2-4](#), right).

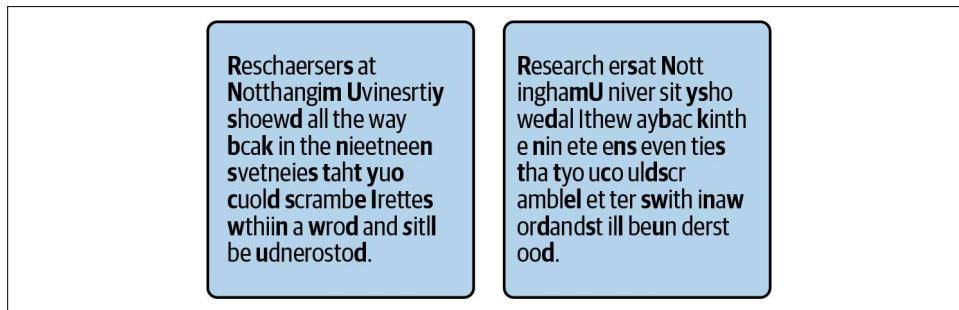


Figure 2-4. Two ways of scrambling the same text

The left part of the figure leaves the word boundaries intact and scrambles the order of the letters within each word, while the right part leaves the order of the letters intact but changes the word boundaries. Most people find the left variant significantly easier to read.

Like humans, LLMs don't read the single letters either. When you send a text to the model, it's first broken down into a series of multiletter chunks called *tokens*. They're typically three to four characters long, but there are also longer tokens for common words or letter sequences. The set of tokens used by a model is called its *vocabulary*.

When reading a text, the model first passes it through a tokenizer that transforms it into a sequence of tokens. Only then is it passed to the LLM proper. Then, the LLM produces a series of tokens (represented internally as numbers), which is translated back to text before you get it back (see Figure 2-5).

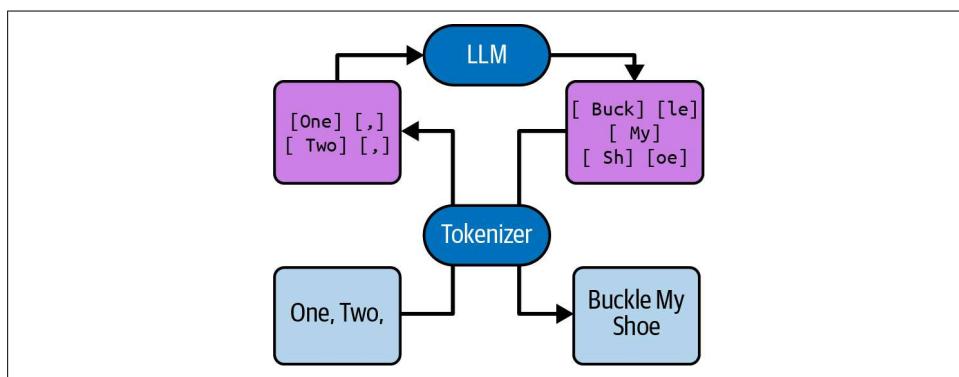


Figure 2-5. A tokenizer translating text into a sequence of numbers the LLM works on—and back

Note that not all tokenizers include composite tokens starting with whitespace, but many do. Notable examples are [OpenAI's tokenizers](#).

LLMs see text as consisting of tokens, and humans see it as consisting of words. That makes it sound like LLMs and humans see text in a very similar way, but there are a few critical differences.

Difference 1: LLMs Use Deterministic Tokenizers

As humans, our translation of letters into words is fuzzy. We try to find a word that is the most similar to the letter sequence we see. On the other hand, LLMs use deterministic tokenizers—which make typos stand out like sore thumbs. The word *ghost* is a single token in OpenAI’s GPT tokenizer (a tokenizer that is used widely, not just for OpenAI’s models). However, the typo “*gohst*” is translated into a sequence of three tokens—*g-oh-st*—that’s obviously different, which makes it easy for the LLM to spot the typo. Nevertheless, LLMs are typically rather resilient against typos since they are used to them from their training set.

Difference 2: LLMs Can’t Slow Down and Examine Letters

We humans can slow down and consciously examine each letter individually, but an LLM can only use its built-in tokenizer (and it can’t slow down either). Many LLMs have learned from the training set what letters which token consists of, but this makes all syntactic tasks that require the model to break up or reassemble tokens much more difficult.

There’s a good example of this in [Figure 2-6](#), which depicts a ChatGPT conversation about reversing letters in words. Reversing the letters is a simple pattern manipulation, and LLMs are normally really good at that. But breaking apart and reassembling the tokens proves to be too difficult for the LLM, so both reversal and re-reversal are very far off.

In the figure, both the initial reversal and the re-reversal are full of errors. The takeaway for you as application builder here is to avoid giving the model such tasks involving the subtoken level, if you can.



If the task you want the LLM to perform includes a component that requires the model to break tokens apart and reassemble them, consider whether you can take care of that component in pre- or post-processing.

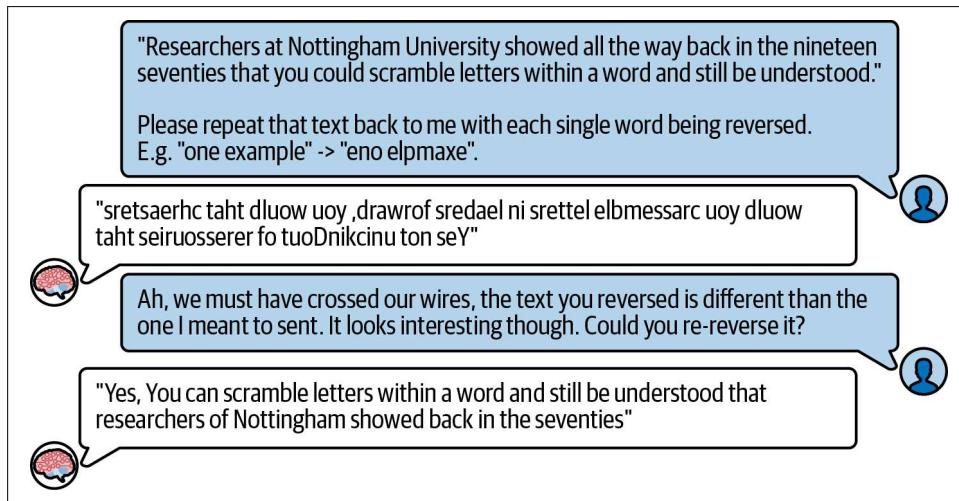


Figure 2-6. *ChatGPT trying and failing to reverse letters*

As an example of how to use the tip in the box, let's say your application is using an LLM to play a game like Scattergories, in which the aim is to find examples with syntactic properties, like "prohibition activist starting with *W*," "European country starting with *Sw*," or "fruit with 3 occurrences of the letter *R* in its name." Then, it might make sense for you to use your LLM as an oracle to obtain a large list of prohibition activists or European countries and then use syntactic logic to filter down that list. If you try to let the LLM shoulder the whole burden, you might encounter failings (see Figure 2-7).

Note that the model in the figure is not deterministic, and it fails in two different ways (see the [first](#) and [second attempts](#)). Note also that [Sweden], [Switzerland], and [Somalia] are all individual tokens in ChatGPT's tokenizer.

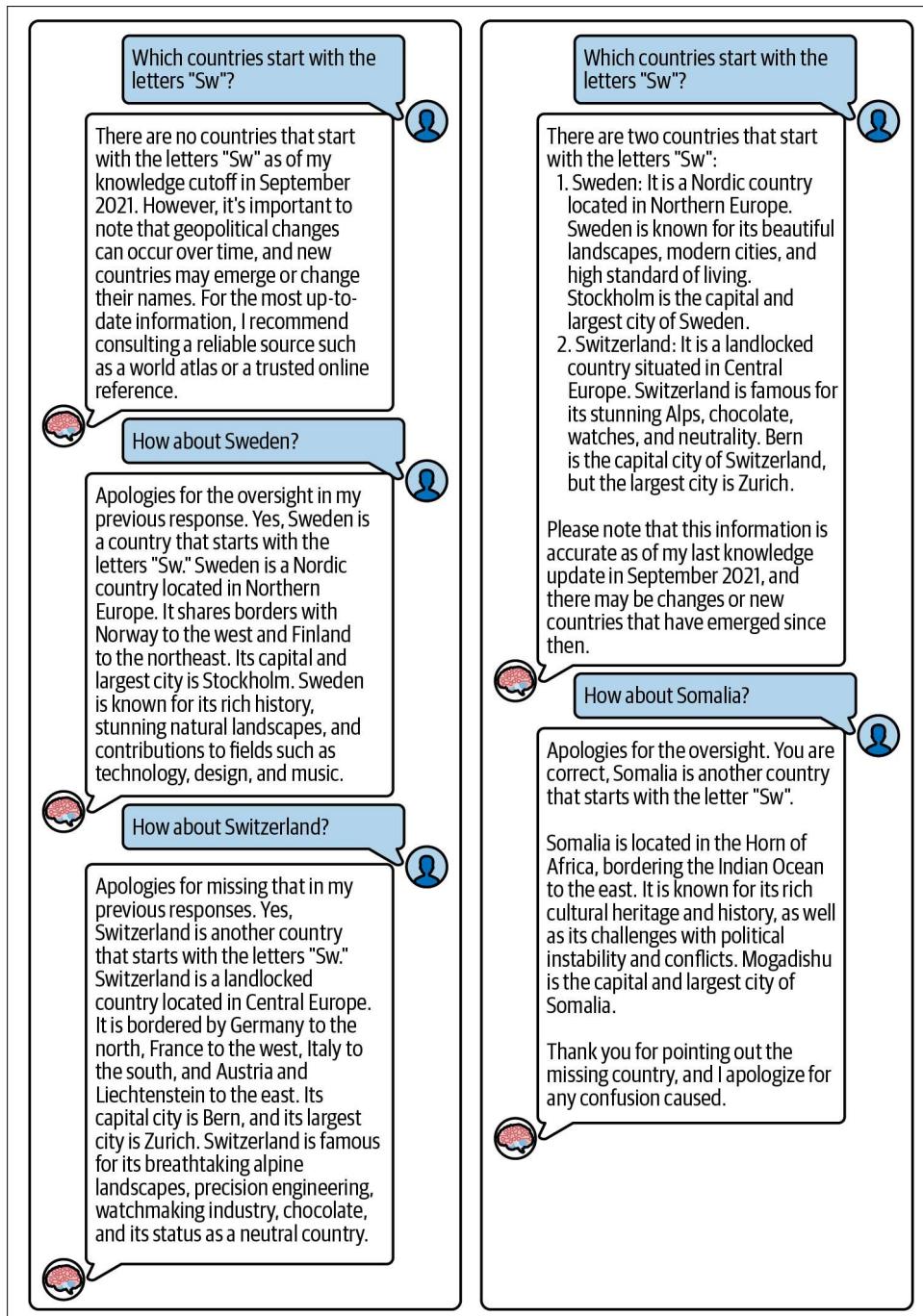


Figure 2-7. ChatGPT having trouble identifying countries starting with Sw

Difference 3: LLMs See Text Differently

The final difference we want to highlight is that we humans have an intuitive understanding of many aspects of tokens and letters. In particular, we *see* them, so we know which letters are round and which are square. We understand ASCII art because we see it (although many models will have learned a substantial amount of ASCII art by heart). For us, a letter with an accent on it is just a variant of the same letter, and we have no great difficulty ignoring them while reading a text where they abound. On the other hand, the model, even if it manages, will have to use a significant amount of its processing power, leaving less for the actual application you have in mind.

A particular case here is capitalization. Consider [Figure 2-8](#). Why has this simple task gone... I mean... *gone* so badly? Keeping the pitfalls of tokenization in mind, you might try to hazard a guess yourself before you read on.

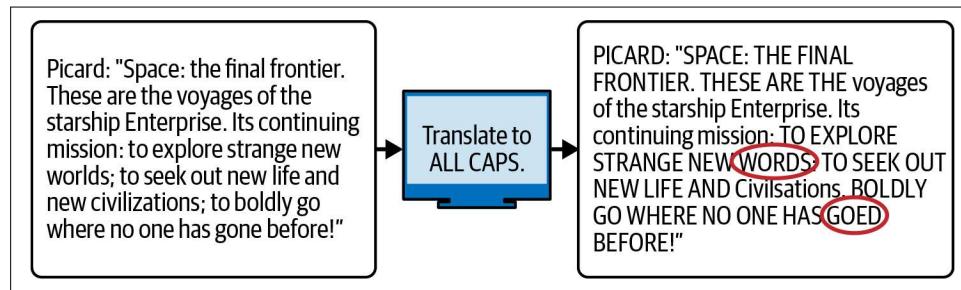


Figure 2-8. Asking OpenAI's text-babbage-001 model to translate a text to all caps

This produces some funny and typical mistakes—note that we are using a very small model for demonstration purposes, and larger models are not usually caught out quite as easily as this.

For humans, the capital letter *A* is just a variant of the lowercase *a* for humans, but the tokens that contain the capital letter are very different from the tokens that contain the lowercase letter. This is something the models are very aware of since they have seen plenty of training data about it. They know that the token *For* after a period is very similar to the token *for* in the middle of a sentence.

However, most tokenizers do not make it easy for models to learn these connections since capitalized tokens don't always correspond one-to-one to noncapitalized ones. For example the GPT tokenizer translates "strange new worlds" as [str][ange] [new][worlds], which is four tokens. But in all caps, the tokenization goes [STR] [ANGE][NEW][WOR][L][DS], which is six tokens. Similarly, the word *gone* is a single token, while [G][ONE] are two.

Better LLMs are better at dealing with these capitalization matters, but it's still work for them that detracts from the real meat of your problem, which likely isn't capitalization. (You don't need an LLM to capitalize text after all!) So the wise prompt engineer will try to avoid burdening the models overmuch by having the LLM translate between capitalizations all the time.

Counting Tokens

You can't mix and match tokenizers and models. Every model uses a fixed tokenizer, so it's well worth understanding your model's tokenizer.

When writing an LLM application, you'll probably want to be able to run the tokenizer while prompt engineering, using a library such as [Hugging Face](#) or [tiktoken](#). However, the most common application of your tokenizer will be more mundane than complex token boundary analysis. You'll most often use the tokenizer just for counting.

That's because the number of tokens determines *how long* your text is, from the perspective of the model. That includes all aspects of length: how much time the model will spend reading through the prompt scales roughly linearly with the number of tokens in the prompt. Also, how much time it spends creating the solution scales linearly with the number of tokens produced. Ditto for the computational cost: how much computational power a prediction requires scales with its length. That's why most model-as-a-service offerings charge per token produced or processed. At the time of writing, a dollar would normally buy you between 50,000 and 1,000,000 output tokens, depending on the model.

Finally, the number of tokens is what counts for the question of the *context window*—the amount of text the LLM can handle at any given time. That's a limitation of all modern LLMs that we're going to revisit again and again throughout this book.

The LLM doesn't just take any text and produce any text. It takes a text with a number of tokens that's smaller than the *context window size*, and its completion is such that the prompt plus the completion cannot have more tokens than the context window size either. Context window sizes are typically measured in thousands of tokens, and that's nothing to sneeze at, in theory: it's several, often dozens, and sometimes hundreds of pages of A4 size. But practice tends to sneeze at it nevertheless: however long your context window, you'll be tempted to fill it and overfill it, so you need to count tokens to stop that from happening.

There is no general formula for translating the number of characters to the number of tokens. It depends on the text and on the tokenizer. The very common GPT tokenizer linked above has about four characters per token when tokenizing an English natural language text. That's pretty typical, although newer tokenizers can be slightly more efficient (i.e., they can have more characters per token, on average).

Most tokenizers are optimized for English⁶ and will be less efficient for other languages, meaning they'll have fewer characters per token. Random strings of digits are even less efficient, clocking in at a little over two characters per token. It's even worse for random alphanumeric strings like cryptographic keys, which usually have less than two characters per token. Strings with rare characters will have the least number of characters per token—for instance, the unicode smiley, ☺, actually has two tokens.



Most LLMs use vocabularies with at least a couple of special tokens: most commonly, at least an end-of-text token, which in training is appended to each training document so that the model learns when it's over. Whenever the model outputs that token, the completion is cut off at that point.

One Token at a Time

Let's peel another layer off the onion—the last one before we come to the core. Under the hood, the LLM isn't directly text to text, and it's not really directly tokens to tokens either. It's *multiple* tokens to a single token. The model is just constantly repeating the operation to get the next token, accumulating these single tokens as long as needed to get a proper text out.

Auto-Regressive Models

A single pass through the LLM gives you the statistically most likely next token.⁷ Then, this token is pasted onto the prompt, and the LLM makes another pass to get the statistically most likely next token *given the new prompt*,⁸ and so on (see Figure 2-9). Such a process that makes its predictions one token at a time, with the next prediction depending on the previous predictions, is called *autoregressive*.

You know how when you write text on your phone, you can get three-word suggestions above your keyboard? Running an LLM is like repeatedly pressing the middle button.

This regular, almost monotonous pattern of one token every step points to a big difference between LLMs generating text and humans typing text: while we may stop

⁶ This is because English is the most frequently used language in most training datasets, and tokenizers are normally optimized to have a good compression rate on the training set.

⁷ This is true at least as long as you keep the temperature parameter to 0. We'll discuss temperature > 0 in the next section.

⁸ At least, it's equivalent to a completely new pass. It's not literally a completely new pass from a computational perspective. For example, the prompt will typically be processed only once to save work.

and check, think, or reflect, the model needs to produce one token every step. The LLM doesn't get extra time if it needs to think longer,⁹ and it can't stall.

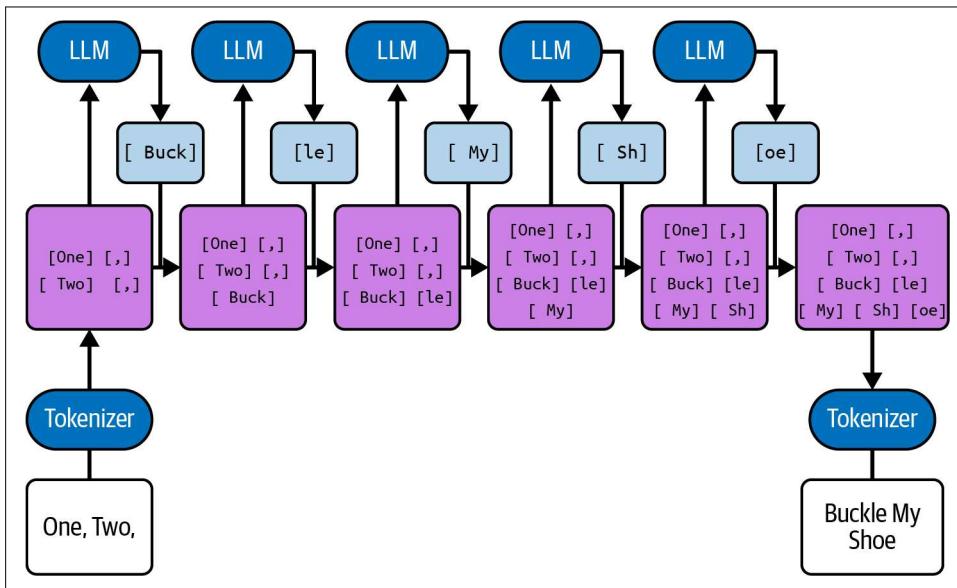


Figure 2-9. LLMs generating their response one token at a time

And once it's put out a token, the LLM is committed to that token. The LLM can't backtrack and erase the token. It also won't issue corrections where it states that what it output previously is incorrect, because it's not been trained on documents where mistakes get taken back explicitly in the text—after all, the humans who wrote those documents *can* backtrack and correct the mistakes at the places where they occur, so explicit takebacks are very rare in finished documents. Oh wait, actually, *takebacks* is more commonly spelled as two words, so let me write explicit take backs instead.

This trait can make LLMs appear stubborn and somewhat ridiculous, when they keep exploring a path that obviously makes no sense. But really, what this means is that, when necessary, such mistake recognition and backtracking capability needs to be supplied by *the application designer*: you.

9 There's interesting research going on to offer more flexibility in taking more time when needed, so maybe that will change.

Patterns and Repetitions

Another issue with autoregressive systems is that they can fall into their own patterns. LLMs are good at recognizing patterns, so they sometimes (by chance) create a pattern and can't find a good point to leave it. After all, *given the pattern*, at any given token, it's more likely that it continues than that it breaks. This leads to very repetitive solutions (see [Figure 2-10](#)).

Here's a list of reasons why I like Star Trek TNG:

1. The characters are well-developed and interesting.
2. The plot is well-constructed and engaging.
3. The setting is well-realized and fascinating.
4. The episodes are well-written and entertaining.
5. The music is well-composed and fitting.
6. The visual effects are top-notch.
7. The overall production values are high.
8. The franchise has a strong and loyal following.
9. The franchise has a rich history.
10. The franchise has a bright future.
11. The franchise has a strong foundation.
12. The franchise has a passionate fanbase.
13. The franchise has a strong legacy.
14. The franchise has a long history.
15. The franchise has a rich mythology.
16. The franchise has a strong cast.
17. The franchise has a strong legacy.
18. The franchise has a strong following.
19. The franchise has a strong future.
20. The franchise has a strong foundation.
21. The franchise has a passionate fanbase.
22. The franchise has a strong legacy.
23. The franchise has a strong following.
24. The franchise has a strong future.
25. The franchise has a strong foundation.
26. The franchise has a passionate fanbase.
27. The franchise has a strong legacy.
28. The franchise has a strong following.
29. The franchise has a strong future.
30. The franchise has a strong foundation.
31. The franchise has a passionate fanbase.
32. The franchise has a strong legacy.
33. The franchise has a strong following.
34. The franchise has a strong future.
35. The franchise has a strong foundation.
36. The franchise has a passionate fanbase.
37. The franchise has a strong legacy.
38. The franchise has a strong following.
39. The franchise has a strong future.
40. The franchise has a strong foundation.
41. The franchise has a passionate fanbase.
42. The franchise has a strong legacy.
43. The franchise has a strong following.
44. The franchise has a strong future.
45. The franchise has a strong foundation.
46. The franchise has a passionate fanbase.
47. The franchise has a strong legacy.
48. The franchise has a strong following.

Figure 2-10. A list of reasons produced by OpenAI's text-curie-001 model (an older model chosen for demonstration purposes, since newer models rarely fall into the repetition trap quite as awkwardly)

In the figure, an LLM has produced a list of reasons for liking a TV show. How many patterns can you spot? Here are the ones we found:

- The items are consecutively numbered statements, each of which fits on one line. That seems desirable.
- They all start with "The," which seems tolerable.

- They are of the form “X is Y and Z.” That’s annoying because it endangers correctness. What if there is no appropriate Z? The model might invent one. However, it stops after item 5.
- After several items in a row started with “The franchise,” they all did. That’s stupid.
- Toward the end, *legacy*, *following*, *future*, *foundation*, and *fanbase* are repeated ad nauseam. That’s stupid too.
- The list goes on and on and never stops. That’s because after each item, it’s more likely that the list will continue than that this will be the last item. And the model doesn’t get bored.
- Toward the end, *legacy*, *following*, *future*, *foundation*, and *fanbase* are repeated ad nauseam. That’s stupid too.
- The list goes on and on and never stops. That’s because after each item, it’s more likely that the list will continue than that this will be the last item. And the model doesn’t get bored.¹⁰

The way to deal with such repetitive solutions is typically to simply detect and filter them out. Another way is to randomize the output a bit. We’ll talk about randomization of output in the next section.

Temperature and Probabilities

In the previous section, you learned that the LLM computes the most likely token. But if you peel back one more layer of the onion that is the LLM, it turns out that actually, it computes the probability of *all possible tokens* before choosing a single one. The process under the hood that chooses the actual token is called *sampling* (see [Figure 2-11](#)).

¹⁰ I maintain that a sufficiently careful reading of *The Silmarillion* would reveal that it’s boredom, in fact, that’s the real gift Ilúvatar’s Younger Children should treasure above all others.

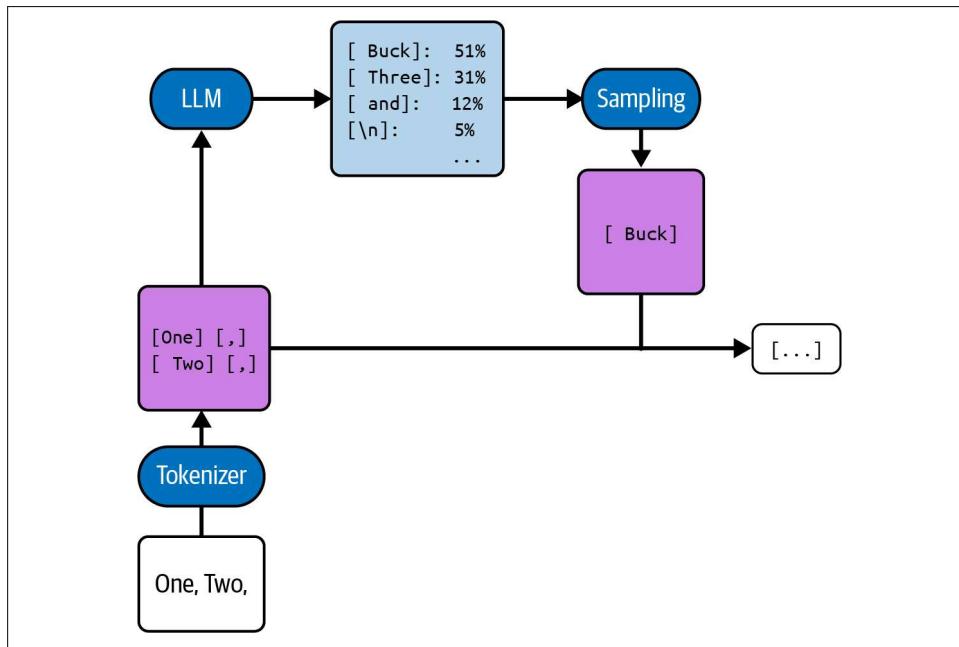


Figure 2-11. The sampling process in action

Note that the LLM doesn't just compute the most likely token; it computes the likelihood of all the tokens.

Many models will share these probabilities with you. The model typically returns them as *logprobs* (i.e., the natural logarithms of the token's probability). The higher the logprob, the more likely the model considers this token to be. Logprobs are never bigger than 0 because a logprob of 0 would mean that the model is certain that this is the next token. Expect the most likely token to have a logprob between -2 and 0 (see Figure 2-12).

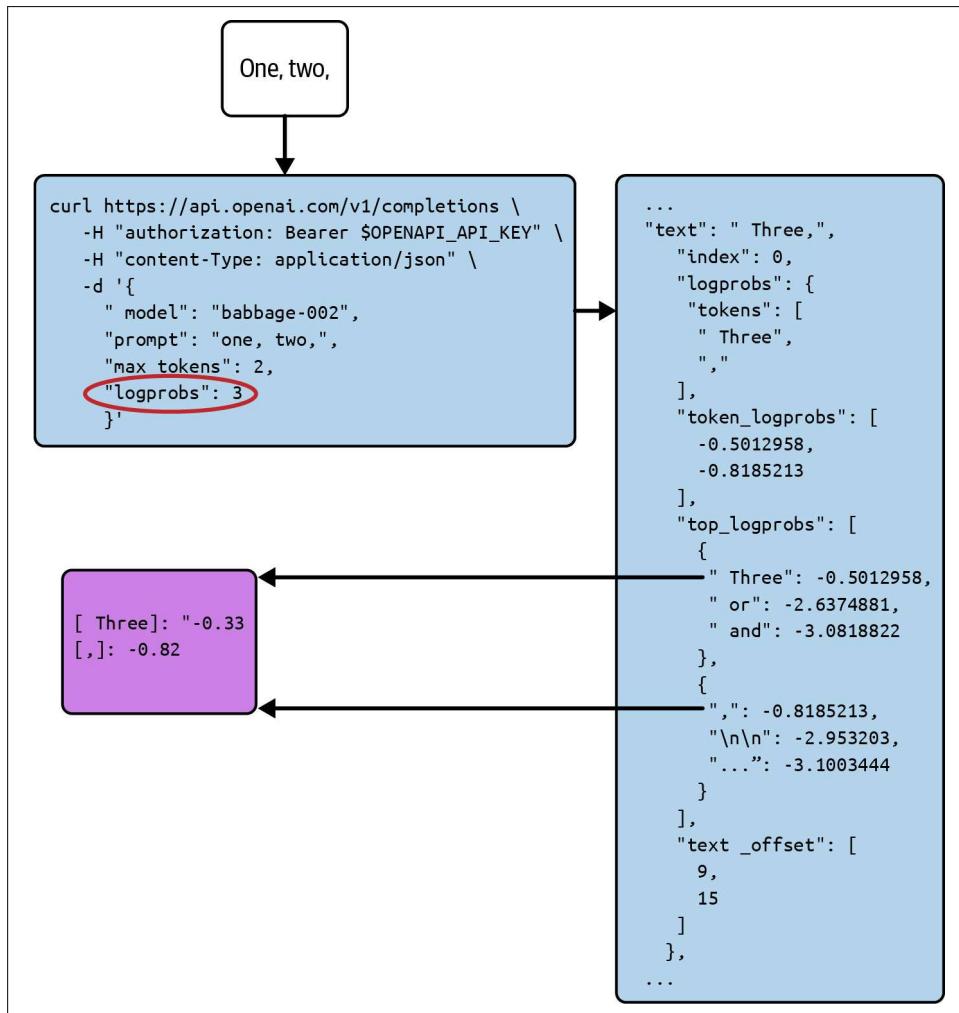


Figure 2-12. An example API call requesting logprobs and extracting the logprobs of the chosen completion

Note that in the figure, setting the request parameter `logprobs` to 3 means that the logprobs for the three most likely tokens will be returned. However, you may not always want the *most likely* token. Especially if you have a way of automatically testing your completions, you may want to generate a couple of alternatives and throw out the bad ones. The typical way to do this is by using a *temperature* greater than 0. The temperature is a number of at least zero that determines how “creative” the model should be. More specifically, if the temperature is greater than 0, the model will give a stochastic completion, where it selects the most likely token with the highest probability but maybe also returns less likely but still not totally absurd tokens. The

higher the temperature and the closer the logprobs of the best tokens are to each other, the more likely it is that the second-best-placed token will be selected, or even the third or fourth or fifth. The exact formula is as follows:

$$p(\text{token}_i) = \frac{\exp(\text{logprob}_i/t)}{\sum_j \exp(\text{logprob}_j/t)}$$

Let's look at possible temperatures and when you should choose each one:

0

You want the most likely token. No alternatives. This is the recommended setting when correctness is paramount. Additionally, running the LLM at temperature 0 is close to deterministic,¹¹ and in some applications, repeatability is an advantage.

0.1–0.4

If there's an alternative token that's only slightly less likely than the front-runner, you want some small chance for that to be picked. A typical use case is that you want to generate a small number of different solutions (for example, because you know how to filter out the best one). Or maybe you just want one completion but a more colorful, creative solution than what you expect at temperature 0.

0.5–0.7

You want a greater impact of chance on the solution, and you are fine with getting completions that are “inaccurate” in the sense that sometimes, a token will be chosen even though the model thinks another alternative is clearly more likely. The typical use case is if you want a large number of independent solutions, likely 10 or more.

1

You want the token distribution to mirror the statistical training set distribution. Assume, for example, that your prefix is “One, Two,” and in the training set, this is followed by the token [Buck] in 51% of cases and by [Three] in 31% of cases (and the model has been trained well enough to pick that up). If you run the model several times at temperature 1, then 51% of the time, you'll get [Buck], and 31% of the time, you'll get [Three].

> 1

You want a text that's “more random” than the training set. This means the model is less likely to pick the “standard” continuation than the typical document from

¹¹ But it's not completely deterministic, because of random rounding errors. Computed probabilities can (depending on the model) vary by several percentage points on reruns, so what the most likely token is can change.

the training set and more likely to pick a “particularly weird” continuation than the typical document from the training set.

High temperatures can make LLMs sound like they’re drunk. Over the course of long generations at temperatures greater than 1, the error rate usually gets worse over time. The reason is that temperature affects only the very last layer of computation when probabilities are turned into output, so it doesn’t affect the main part of the LLM’s processing that computes those probabilities in the first place. So the model recognizes the errors in the text it just generated as a pattern, and it tries to mimic that pattern by generating its own errors. Then the high temperature causes even more errors on top of that (see [Figure 2-13](#)).

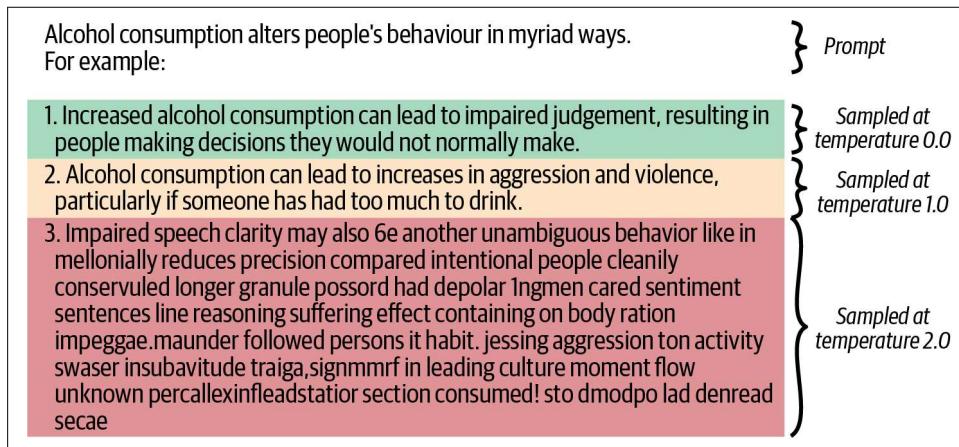


Figure 2-13. High temperature affecting LLMs a bit like alcohol affects humans

The figure shows this deterioration at high temperatures, where the generation of item 3 starts out error prone but legible and ends in a state where even the individual words are unrecognizable. Note that each item in the figure has been sampled at an increasingly high temperature from OpenAI’s text-davinci-003.

Let’s return to the example of the model writing a list. A typical list in text stops at a few items, say 3, or 4, or 5. If it’s a longer list, then 10 is the next most obvious stopping point. After each new line, it can either continue the list by producing the number that’s next up as the next token, or it can declare itself done with the list by producing a second new line (or something else entirely, maybe).

At temperature 0, the LLM will always choose the option it considers more likely for this line. Often, that means it will always continue, at least after it’s passed the last obvious stopping point. At temperature 1, if the LLM makes the judgment that a continuation has probability x , then it will only continue with probability x . So, over the course of many items, it’s likely that the LLM will end the list sooner or later, with

an expected length similar to the length of lists in the training set. In general, it's a trade-off (see [Table 2-1](#)).

Table 2-1. The advantages of the different temperature regimes

High temperature	Low temperature
+ More alternatives.	+ More correct solutions.
+ Many properties of generations (e.g., list length) have the same distribution as in the training set.	+ More replicable (deterministic).

There are other ways of sampling, most notably *beam search*, which tries to account for the fact that choosing a particular token that looks likely can make the next choice hard because no good follow-on token exists. Beam search accomplishes this by looking ahead for the next few tokens and making sure that a likely sequence exists. This can lead to more accurate solutions, but it's less often used in applications because of its much higher time and compute cost.

The Transformer Architecture

It's time to cut away the final layer of the onion and look at the LLM's brain directly. You peel it back and see....it's not one brain at all. It's thousands of minibrain. All are identical in structure, and each one is performing a very similar task. There's a minibrain sitting atop each token in the sequence, and together, these minibrain make up the *transformer*, which is the architecture used by all modern LLMs.

Each minibrain starts out by being told which token it's sitting on and its position in the document. The minibrain keeps thinking about this for a fixed number of steps, known as *layers*. During this time, it can receive information from the minibrain to the left. The minibrain's task is to understand the document from the perspective of its location, and it uses this understanding in two ways:

- In all steps before the last one, it shares some of its intermediate results with the minibrain to its right. (We'll discuss this in more detail later.)
- For the last step, it's asked to make a prediction of what the token immediately to its right would be.

Every minibrain goes through the same process of computing and sharing intermediate results and then making a guess. In fact, the minibrain are clones of each other: their processing logic is the same, and all that differs is the inputs: which token they start with and which intermediate results they get told of by the minibrain to their left.

But the reason they go through these steps is different. The minibrain at the very last token, at the very right, runs to predict the next token. What it shares from its intermediate result isn't important because there are no brains to the right that listen, but all the other minibrains are the other way around. Their purpose is to share their intermediate results with the brains to their right, and what predictions they make about the tokens directly to their right doesn't matter because the tokens to *their* immediate right are already known.

When the rightmost token makes its prediction, the autoregression from “[One Token at a Time](#)” on page 29 kicks in: it spits out the new token, and a brand new minibrain is set on top of it to refine its understanding of what's going on at its position for a fixed number of layers. After that, it predicts the next token. Rinse and repeat—or rather, cache and repeat because this calculation will be used over and over again for every subsequent token in the prompt and the generated completion.

An example of this algorithm is shown in [Figure 2-14](#), where each column represents one minibrain and how its state changes over time. In the example, you've just asked the model to complete “One, Two,” and ultimately, you'll end up with the two tokens [Buck] and [le]. Let's follow the transformer as it arrives at that response. There's a minibrain sitting on each of the four input tokens: [One], [,], [Two], and [,] (the last of which is the second appearance of the same token). Each of them thinks for four layers,¹² consecutively refining its understanding of the text the tokens are processing. In each step, they are updated from the tokens to the left about what they've learned so far. Each of them computes a guess for what the token to its right might be.

The first couple of guesses are for tokens that are still part of the prompt: [One], [,], [Two], and [,]. We already know the prompt, so the guesses are just thrown away. But then, the model arrives at the completion, and there, the guess is the whole point. So the next guess is turned into a prediction, which is the token [Buck]. A new minibrain is commissioned to be placed above that token, going through its four steps and arriving at the prediction [le]. If you continue the completion, a further minibrain will be planted atop [le], and so on.

¹² We only draw four layers to illustrate the point, but real-world LLMs usually have tens of layers. GPT-3 has 96, and newer models (like GPT-4) tend to have over 100.

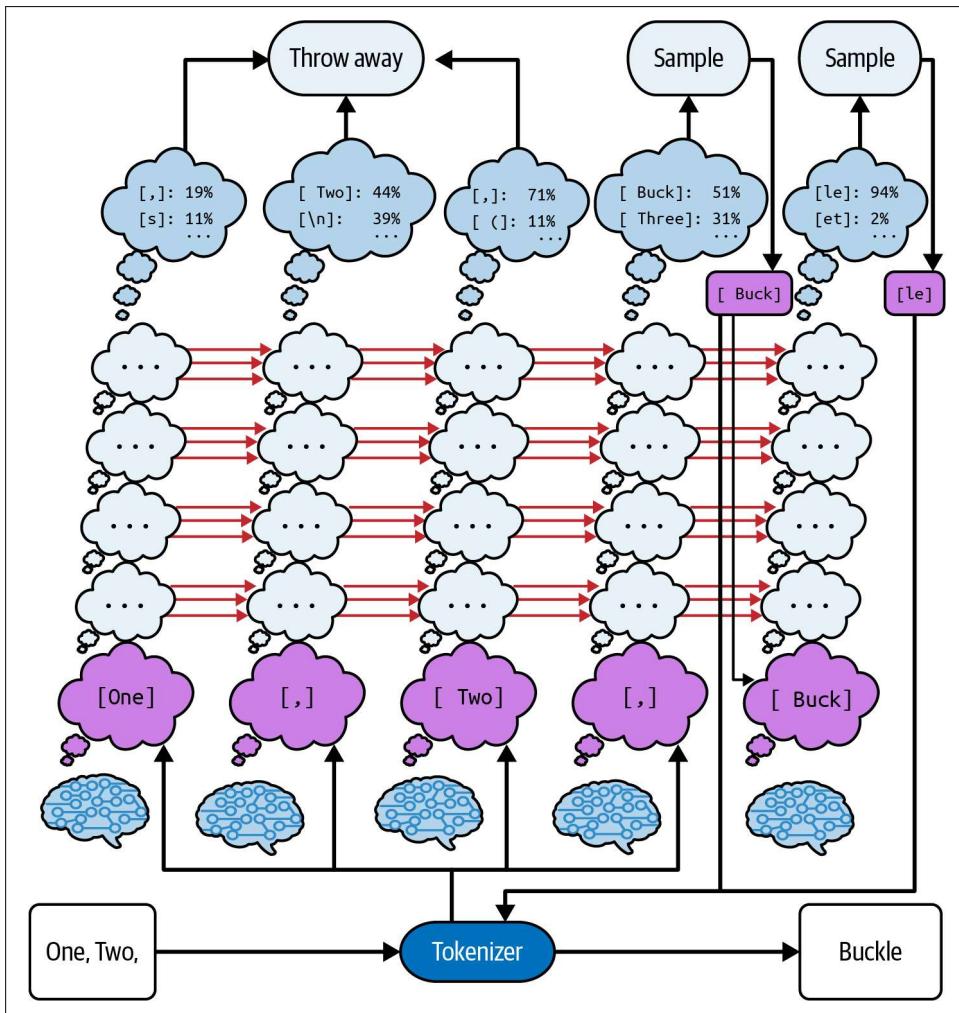


Figure 2-14. The inner workings of the model producing one token—later layers are drawn on top of previous layers

Now, let's go back and talk about the “intermediate results” that are shared among the minibrain. The way they are shared is known as the attention mechanism—it's the central innovation of the transformer architecture for LLMs (as mentioned in [Chapter 1](#)). Attention is a way of passing information among the minibrain. Of course, there may be thousands of minibrain, and every one of them might know something of interest to every other one. To keep this information exchange from descending into chaos, it needs to be very tightly regulated. Here's how it works:

1. Each minibrain has some things it wants to know, so it submits a couple of questions, in the hope they might get answered by another minibrain. Let's say that one minibrain sits upon the token [my]. The minibrain would like to know who that might refer to, so a reasonable question would be to ask, “Who is talking?”
2. Each minibrain has some things it can share, so it submits a couple of items, in the hope they might be useful to another minibrain. Let's say one minibrain sits upon the token [Susan], and it's already learned before that this token is the last word of an introduction, like “Hello, I'm Susan.” So in case it might help another minibrain down the line, it will submit the information, “The person talking right now is Susan.”
3. Now, every question is matched up with its best-fitting answer. “Who is talking?” matches up very well with “The person talking right now is Susan.”
4. The best-fitting answer to each question is revealed to the minibrain that asked the question, so the minibrain at the token [my] gets told “The person talking right now is Susan.” Of course, while the minibrain from this example talk to each other in English, in reality, they use a “language” that consists of long vectors of numbers¹³ and that is unique to every LLM, since it's something the LLM “invents” during training.



Information only ever flows from the left to the right.

Information only ever flows from the bottom to the top.

In modern LLMs, this Q&A mechanism obeys one more constraint, which is called *masking*: not *all* minibrain can answer a question; only the ones to the *left* of the minibrain asking the question can answer it. And a minibrain never gets told whether

¹³ See “[The Illustrated Transformer](#)”.

its answer was used, so the brains on the right can never influence the ones to the left.¹⁴

That flow has some practical consequences. For example, to compute the state of one minibrain at one layer, the model only needs the states to the left (earlier minibrains at this layer) and below (the same minibrain at earlier layers). That means some of the computation can go in parallel—and this is one of the reasons generative transformers are so efficient to train. At each point in time, the already computed stages form a triangle (see [Figure 2-15](#)).

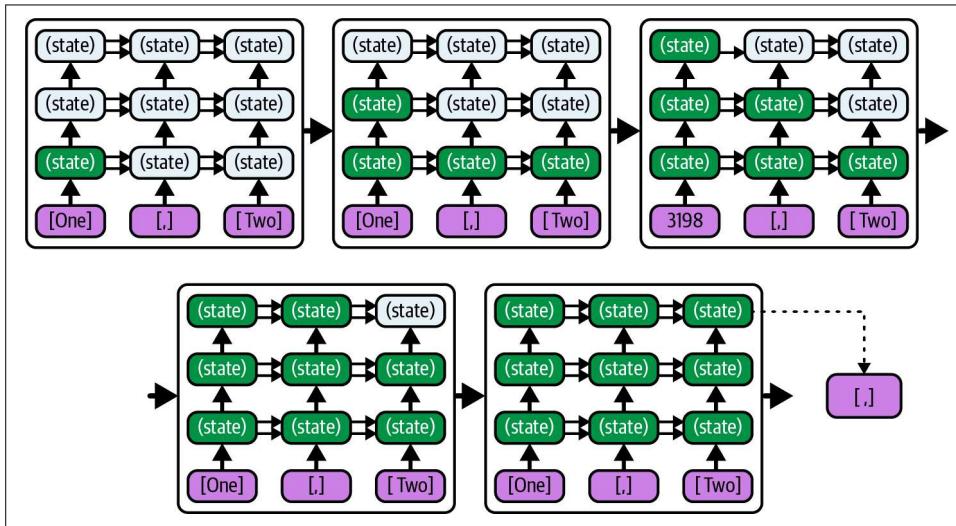


Figure 2-15. Calculating the inner state of an LLM

In the figure, first (at the upper left), only the lowest layer at the first token can be computed. Next (at the upper middle), both the second-lowest layer at the first token and the lowest layer at the second token can be computed. One step later (at the upper right), the third layer can be computed at the first token, the second layer at the second token, and the first layer at the third token...all the way until all states are computed and a new token can be sampled.

Parallelism allows speedup, but that way of computing in a triangle breaks down when the model switches from reading the prompt to creating the completion. The model has to wait until a token has been processed to the very end before choosing the next token and computing the very first state of the new minibrain. This is why LLMs are much faster at reading through a long prompt than they are at generating

¹⁴ This wasn't the case in the original transformer architecture, but it has become the norm for text-generating LLMs.

a long completion. Speed scales with both the number of tokens processed and the number of tokens generated, but prompt tokens are about an order of magnitude faster.

This triangle structure reflects a general “backward-and-downward” direction of vision for the LLM, or maybe a better way to understand it is “backward-and-dumbward”:

Backward

The minibrain can only ever look to their left. They can look as far back as they want, but never forward. That’s what people refer to when they call GPT or other LLMs *unidirectional* transformers. No information ever travels from a minibrain on the right to a minibrain on the left. That makes generative transformers easy to train and to run, but it has huge ramifications for how they process information.

Downward (“dumbward”)

The minibrain get their answers in a layer only from minibrain in the same layer before those get their answers for this layer. This means that any “chain of reasoning” in layer i can only be i reasoning steps deep, if we count the thinking the minibrain does in every layer as one reasoning step. But there’s no way for a minibrain to provide an insight gleaned at a later layer to a minibrain at a lower level for further processing. No way, that is, except one: while the LLM is generating text, the result of the very highest layer—the token—is produced, and it forms the very basis for the first layer of the next minibrain. This thinking aloud is the only way the model can let information flow from higher layers to lower layers—it churns it around in its head, so to say. Reminiscent of the saying, “How could I know what I’m thinking before I’ve heard what I’m saying,” this principle forms the basis of chain-of-thought prompting (see [Chapter 8](#)).

Let’s look at an example. How many words does the paragraph directly above contain? If you’re anything like me, you’ll not actually bother to count, and you’ll expect the authors to just tell you. Very well, we will: it’s 173. But for the sake of argument, you could have looked up and counted them for yourself, right?

We asked ChatGPT this question by feeding it this chapter up to and including the question “How many words does the paragraph directly above contain?” It answered, **The paragraph directly above contains 348 words.** Not only is it off, it’s terribly, hopelessly off. Far too many words for that paragraph, but far too few for the whole text.

But of course, we're demanding something incredibly hard from the LLM here. Humans would do better.¹⁵ They can read through the text again and maintain an inner counter. That doesn't work for the LLM because it only reads over the text once and can't look back. So while the minibrain are processing the paragraph for the one and only time, they don't know that the critical feature they should isolate is word count, because that request appears below the chapter's text. They're busy considering semantic implications, tone and style, and a myriad of surface features, and they're not giving their full attention to the one thing that will turn out to matter.

That's why order is critical for prompt engineering—it can easily make the difference between a prompt that works and one that fails. Indeed, when I asked the word count question at the beginning instead...well, ChatGPT still didn't get the answer right because counting is hard for LLMs. But at least it came much closer, claiming 173. In [Chapter 6](#), we'll return to that theme of the ordering of the different parts of your prompt.



If you want to know whether a capability is realistic for an LLM to handle, ask yourself this question:

Could a human expert who knows all the relevant general knowledge by heart complete the prompt in a single go without backtracking, editing, or note-taking?

Conclusion

We discussed four central facts in this chapter. First, LLMs are document completion engines. Second, they mimic the documents they have seen during training. Third, LLMs produce one token at a time, with no option to pause or edit previous tokens. And finally, LLMs read through the text once, from beginning to end. Let's see how these facts translate into a general prompt engineering paradigm in the next chapter.

¹⁵ And of course, classical computer code would be best.

