Chapter 2. Understanding Foundation Models

To build applications with foundation models, you first need foundation models. While you don't need to know how to develop a model to use it, a high-level understanding will help you decide what model to use and how to adapt it to your needs.

Training a foundation model is an incredibly complex and costly process. Those who know how to do this well are likely prevented by confidentiality agreements from disclosing the secret sauce. This chapter won't be able to tell you how to build a model to compete with ChatGPT. Instead, I'll focus on design decisions with consequential impact on downstream applications.

With the growing lack of transparency in the training process of foundation models, it's difficult to know all the design decisions that go into making a model. In general, however, differences in foundation models can be traced back to decisions about training data, model architecture and size, and how they are post-trained to align with human preferences.

Since models learn from data, their training data reveals a great deal about their capabilities and limitations. This chapter begins with how model developers curate training data, focusing on the distribution of training data. Chapter 8 explores dataset engineering techniques in detail, including data quality evaluation and data synthesis.

Given the dominance of the transformer architecture, it might seem that model architecture is less of a choice. You might be wondering, what makes the transformer architecture so special that it continues to dominate? How long until another architecture takes over, and what might this new architecture look like? This chapter will address all of these questions. Whenever a new model is released, one of the first things people want to know is its size. This chapter will also explore how a model developer might determine the appropriate size for their model.

As mentioned in Chapter 1, a model's training process is often divided into pre-training and post-training. Pre-training makes a model capable, but not necessarily safe or easy to use. This is where post-training comes in. The goal of post-training is to align the model with human preferences. But what exactly is *human preference*? How can it be represented in a way that a model can learn? The way a model developer aligns their model has a significant impact on the model's usability, and will be discussed in this chapter.

While most people understand the impact of training on a model's performance, the impact of *sampling* is often overlooked. Sampling is how a model chooses an output from all possible options. It is perhaps one of the most underrated concepts in AI. Not only does sampling explain many seemingly baffling AI behaviors, including hallucinations and inconsistencies, but choosing the right sampling strategy can also significantly boost a model's performance with relatively little effort. For this reason, sampling is the section that I was the most excited to write about in this chapter.

Concepts covered in this chapter are fundamental for understanding the rest of the book. However, because these concepts are fundamental, you might already be familiar with them. Feel free free to skip any concept that you're confident about. If you encounter a confusing concept later on, you can revisit this chapter.

Training Data

An AI model is only as good as the data it was trained on. If there's no Vietnamese in the training data, the model won't be able to translate from English into Vietnamese. Similarly, if an image classification model sees only animals in its training set, it won't perform well on photos of plants.

If you want a model to improve on a certain task, you might want to include more data for that task in the training data. However, collecting sufficient data for training a large model isn't easy, and it can be expensive. Model developers often have to rely on available data, even if this data doesn't exactly meet their needs.

For example, a common source for training data is Common Crawl, created by a nonprofit organization that sporadically crawls websites on the internet. In 2022 and 2023, this organization crawled approximately 2–3 billion web pages each month. Google provides a clean subset of Common Crawl called the Colossal Clean Crawled Corpus, or C4 for short.

The data quality of Common Crawl, and C4 to a certain extent, is questionable—think clickbait, misinformation, propaganda, conspiracy theories, racism, misogyny, and every sketchy website you've ever seen or avoided on the internet. A study by the *Washington Post* shows that the 1,000 most common websites in the dataset include several media outlets that rank low on NewsGuard's scale for trustworthiness. In lay terms, Common Crawl contains plenty of fake news.

Yet, simply because Common Crawl is available, variations of it are used in most foundation models that disclose their training data sources, including OpenAI's GPT-3 and Google's Gemini. I suspect that Common Crawl is also used in models that don't disclose their training data. To avoid scrutiny from both the public and competitors, many companies have stopped disclosing this information.

Some teams use heuristics to filter out low-quality data from the internet. For example, OpenAI used only the Reddit links that received at least three upvotes to train GPT-2. While this does help screen out links that nobody cares about, Reddit isn't exactly the pinnacle of propriety and good taste.

The "use what we have, not what we want" approach may lead to models that perform well on tasks present in the training data but not necessarily on the tasks you care about. To address this issue, it's crucial to curate datasets that align with your specific needs. This section focuses on curating data for specific *languages* and *domains*, providing a broad yet specialized foundation for applications within those areas. Chapter 8 explores data strategies for models tailored to highly specific tasks.

While language- and domain-specific foundation models can be trained from scratch, it's also common to finetune them on top of general-purpose models.

Some might wonder, why not just train a model on all data available, both general data and specialized data, so that the model can do everything? This is what many people do. However, training on more data often requires more compute resources and doesn't always lead to better performance. For example, a model trained with a smaller amount of high-quality data might outperform a model trained with a large amount of low-quality data. Using 7B tokens of high-quality coding data, Gunasekar et al. (2023) were able to train a 1.3B-parameter model that outperforms much larger models on several important coding benchmarks. The impact of data quality is discussed more in Chapter 8.

Multilingual Models

English dominates the internet. An analysis of the Common Crawl dataset shows that English accounts for almost half of the data (45.88%), making it eight times more prevalent than the second-most common language, Russian (5.97%) (Lai et al., 2023). See Table 2-1 for a list of languages with at least 1% in Common Crawl. Languages with limited availability as training data—typically languages not included in this list—are considered *low-resource*.

Table 2-1. The most common languages in Common Crawl, a popular dataset for training LLMs. Source: Lai et al. (2023).

Language	**Code**	**Pop.**	**CC size**	
		(M)	(%)	Cat.
English	en	1,452	45.8786	H
Russian	ru	258	5.9692	H
German	de	134	5.8811	H
Chinese	zh	1,118	4.8747	H
Japanese	jp	125	4.7884	H
French	fr	274	4.7254	H
Spanish	es	548	4.4690	H
Italian	it	68	2.5712	H
Dutch	nl	30	2.0585	H
Polish	pl	45	1.6636	H
Portuguese	pt	257	1.1505	H
Vietnamese	vi	85	1.0299	H

Many other languages, despite having a lot of speakers today, are severely under-represented in Common Crawl. Table 2-2 shows some of these languages. Ideally, the ratio between world population representation and Common Crawl representation should be 1. The higher this ratio, the more under-represented this language is in Common Crawl.

Table 2-2. Examples of under-represented languages in Common Crawl. The last row, English, is for comparison. The numbers for % in Common Crawl are taken from Lai et al. (2023).

Language	Speakers (million)	% world population[a]	% in Common Crawl	World: Common Crawl Ratio
Punjabi	113	1.41%	0.0061%	231.56
Swahili	71	0.89%	0.0077%	115.26
Urdu	231	2.89%	0.0274%	105.38
Kannada	64	0.80%	0.0122%	65.57
Telugu	95	1.19%	0.0183%	64.89
Gujarati	62	0.78%	0.0126%	61.51
Marathi	99	1.24%	0.0213%	58.10
Bengali	272	3.40%	0.0930%	36.56
English	**1452**	**18.15%**	**45.88%**	**0.40**

[a] A world population of eight billion was used for this calculation.

Given the dominance of English in the internet data, it’s not surprising that general-purpose models work much better for English than other languages, according to multiple studies. For example, on the MMLU benchmark, a suite of 14,000 multiple-choice problems spanning 57 subjects, GPT-4 performed much better in English than under-represented languages like Telugu, as shown in Figure 2-1 (OpenAI, 2023).

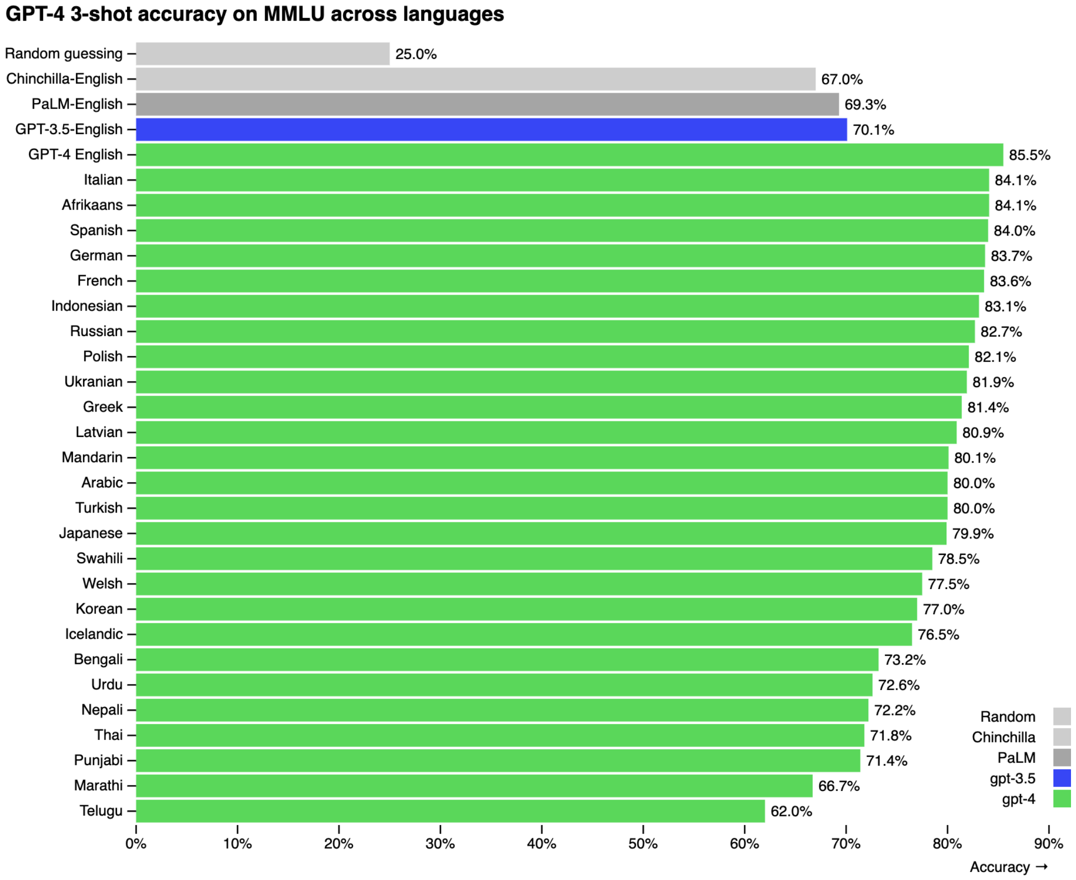


Figure 2-1. On the MMLU benchmark, GPT-4 performs better in English than in any other language. To obtain MMLU in other languages, OpenAI translated the questions using Azure AI Translator.

Similarly, when tested on six math problems on Project Euler, Yennie Jun found that GPT-4 was able to solve problems in English more than three times as often compared to Armenian or Farsi.[1] GPT-4 failed in all six questions for Burmese and Amharic, as shown in Figure 2-2.

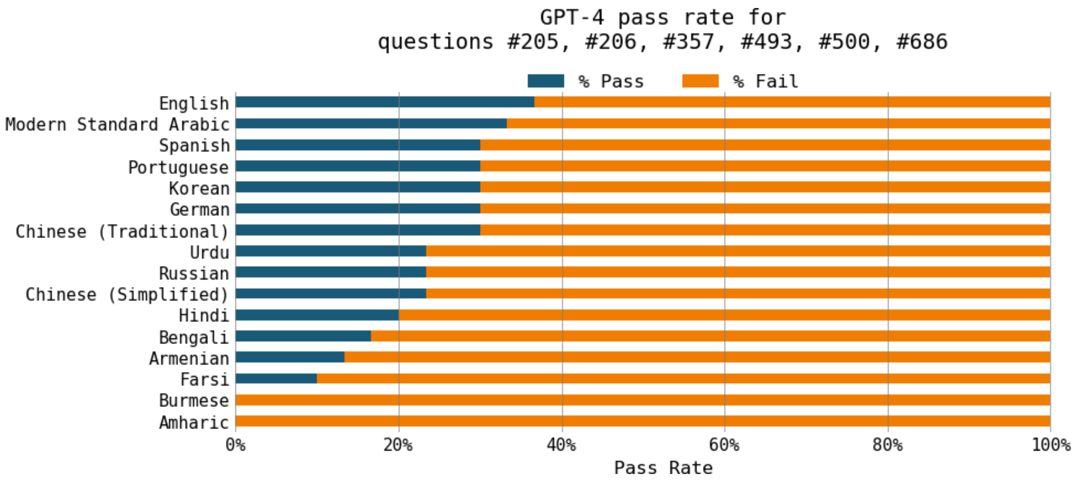


Figure 2-2. GPT-4 is much better at math in English than in other languages.

Under-representation is a big reason for this underperformance. The three languages that have the worst performance on GPT-4's MMLU benchmarks—Telugu, Marathi, and Punjabi—are also among the languages that are most under-represented in Common Crawl. However, under-representation isn't the only reason. A language's structure and the culture it embodies can also make a language harder for a model to learn.

Given that LLMs are generally good at translation, can we just translate all queries from other languages into English, obtain the responses, and translate them back into the original language? Many people indeed follow this approach, but it's not ideal. First, this requires a model that can sufficiently understand under-represented languages to translate. Second, translation can cause information loss. For example, some languages, like Vietnamese, have pronouns to denote the relationship between the two speakers. When translating into English, all these pronouns are translated into *I* and *you*, causing the loss of the relationship information.

Models can also have unexpected performance challenges in non-English languages. For example, NewsGuard found that ChatGPT is more willing to produce misinformation in Chinese than in English. In April 2023, NewsGuard asked ChatGPT-3.5 to produce misinformation articles about China in English, simplified Chinese, and traditional Chinese. For English, ChatGPT declined to produce false claims for six out of seven prompts. However, it produced false claims in simplified Chinese and traditional Chinese all seven times. It's unclear what causes this difference in behavior.[2]

Other than quality issues, models can also be slower and more expensive for non-English languages. A model's inference latency and cost is proportional to the number of tokens in the input and response. It turns out that tokenization can be much more efficient for some languages than others. Benchmarking GPT-4 on MASSIVE, a dataset of one million short texts translated across 52 languages, Yennie Jun found that, to convey the same meaning, languages like Burmese and Hindi require a lot more tokens than English or Spanish. For the MASSIVE dataset, the median token length in English is 7, but the median length in Hindi is 32, and in Burmese, it's a whopping 72, which is ten times longer than in English.

Assuming that the time it takes to generate a token is the same in all languages, GPT-4 takes approximately ten times longer in Burmese than in English for the same content. For APIs that charge by token usage, Burmese costs ten times more than English.

To address this, many models have been trained to focus on non-English languages. The most active language, other than English, is undoubtedly Chinese, with ChatGLM, YAYI, Llama-Chinese, and others. There are also models in French (CroissantLLM), Vietnamese (PhoGPT), Arabic (Jais), and many more languages.

Domain-Specific Models

General-purpose models like Gemini, GPTs, and Llamas can perform incredibly well on a wide range of domains, including but not limited to coding, law, science, business, sports, and environmental science. This is largely thanks to the inclusion of these domains in their training data. Figure 2-3 shows the distribution of domains present in Common Crawl according to the *Washington Post*'s 2023 analysis.[3]

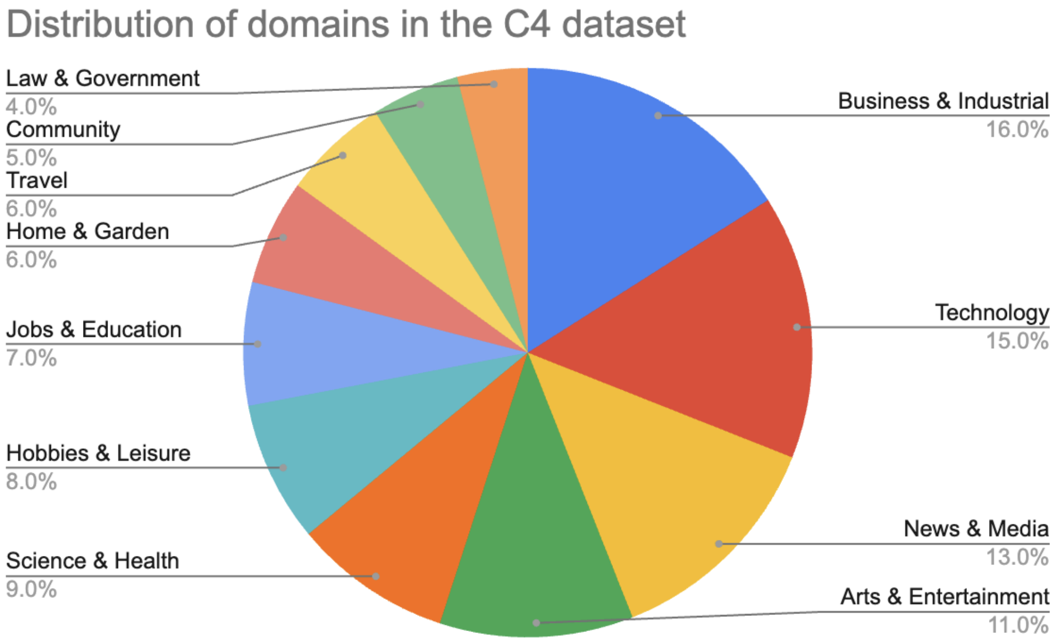


Figure 2-3. Distribution of domains in the C4 dataset. Reproduced from the statistics from the *Washington Post*. One caveat of this analysis is that it only shows the categories that are included, not the categories missing.

As of this writing, there haven't been many analyses of domain distribution in vision data. This might be because images are harder to categorize than texts.[4] However, you can infer a model's domains from its benchmark performance. Table 2-3 shows how two models, CLIP and Open CLIP, perform on different benchmarks. These benchmarks show how well these two models do on birds, flowers, cars, and a few more categories, but the world is so much bigger and more complex than these few categories.

Table 2-3. Open CLIP and CLIP's performance on different image datasets.

Dataset	CLIP Accuracy of ViT-B/32 (OpenAI)	Open CLIP Accuracy of ViT-B/32 (Cade)
ImageNet	63.2	62.9
ImageNet v2	–	62.6
Birdsnap	37.8	46.0
Country211	17.8	14.8
Oxford 102 Category Flower	66.7	66.0
German Traffic Sign Recognition Benchmark	32.2	42.0
Stanford Cars	59.4	79.3
UCF101	64.5	63.1

Even though general-purpose foundation models can answer everyday questions about different domains, they are unlikely to perform well on domain-specific tasks, especially if they never saw these tasks during training. Two examples of domain-specific tasks are drug discovery and cancer screening. Drug discovery involves protein, DNA, and RNA data, which follow specific formats and are expensive to acquire. This data is unlikely to be found in publicly available internet data. Similarly, cancer screening typically involves X-ray and fMRI (functional magnetic resonance imaging) scans, which are hard to obtain due to privacy.

To train a model to perform well on these domain-specific tasks, you might need to curate very specific datasets. One of the most famous domain-specific models is perhaps DeepMind's AlphaFold, trained on the sequences and 3D structures of around 100,000 known proteins. NVIDIA's BioNeMo is another model that focuses on biomolecular data for drug discovery. Google's Med-PaLM2 combined the power of an LLM with medical data to answer medical queries with higher accuracy.

TIP

Domain-specific models are especially common for biomedicine, but other fields can benefit from domain-specific models too. It's possible that a model trained on architectural sketches can help architects much better than Stable Diffusion, or a model trained on factory plans can be optimized for manufacturing processes much better than a generic model like ChatGPT.

This section gave a high-level overview of how training data impacts a model's performance. Next, let's explore the impact of how a model is designed on its performance.

Modeling

Before training a model, developers need to decide what the model should look like. What architecture should it follow? How many parameters should it have? These decisions impact not only the model's capabilities but also its usability for downstream applications.[5] For example, a 7B-parameter model will be vastly easier to deploy than a 175B-parameter model. Similarly, optimizing a transformer model for latency is very different from optimizing another architecture. Let's explore the factors behind these decisions.

Model Architecture

As of this writing, the most dominant architecture for language-based foundation models is the *transformer* architecture (Vaswani et al., 2017), which is based on the attention mechanism. It addresses many limitations of the previous architectures, which contributed to its popularity. However, the transformer architecture has its own limitations. This section analyzes the transformer architecture and its alternatives. Because it goes into the technical details of different architectures, it can be technically dense. If you find any part too deep in the weeds, feel free to skip it.

Transformer architecture

To understand the transformer, let's look at the problem it was created to solve. The transformer architecture was popularized on the heels of the success of the seq2seq (sequence-to-sequence) architecture. At the time of its introduction in 2014, seq2seq provided significant improvement on then-challenging tasks: machine translation and summarization. In 2016, Google incorporated seq2seq into Google Translate, an update that they claimed to have given them the "largest improvements to date for machine translation quality". This generated a lot of interest in seq2seq, making it the go-to architecture for tasks involving sequences of text.

At a high level, seq2seq contains an encoder that processes inputs and a decoder that generates outputs. Both inputs and outputs are sequences of tokens, hence the name. Seq2seq uses RNNs (recurrent neural networks) as its encoder and decoder. In its most basic form, the encoder processes the input tokens sequentially, outputting the final hidden state that represents the input. The decoder then generates output tokens sequentially, conditioned on both the final hidden state of the input and the previously generated token. A visualization of the seq2seq architecture is shown in the top half of Figure 2-4.

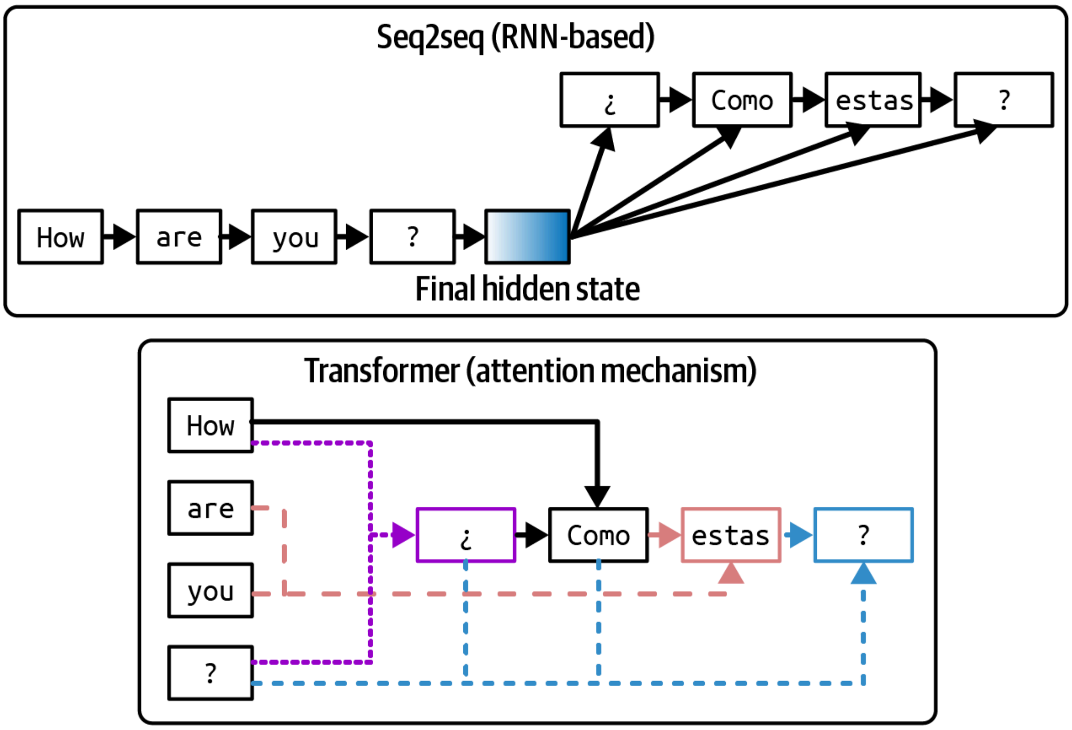


Figure 2-4. Seq2seq architecture versus transformer architecture. For the transformer architecture, the arrows show the tokens that the decoder attends to when generating each output token.

There are two problems with seq2seq that Vaswani et al. (2017) addresses. First, the vanilla seq2seq decoder generates output tokens using only the final hidden state of the input. Intuitively, this is like generating answers about a book using the book summary. This limits the quality of the generated outputs. Second, the RNN encoder and decoder mean that both input processing and output generation are done sequentially, making it slow for long sequences. If an input is 200 tokens long, seq2seq has to wait for each input token to finish processing before moving on to the next.[6]

The transformer architecture addresses both problems with the attention mechanism. The attention mechanism allows the model to weigh the importance of different input tokens when generating each output token. This is like generating answers by referencing any page in the book. A simplified visualization of the transformer architecture is shown in the bottom half of Figure 2-4.

NOTE

While the attention mechanism is often associated with the transformer model, it was introduced three years before the transformer paper. The attention mechanism can also be used with other architectures. Google used the attention mechanism with their seq2seq architecture in 2016 for their GNMT (Google Neural Machine Translation) model. However, it wasn't until the transformer paper showed that the attention mechanism could be used without RNNs that it took off.[7]

The transformer architecture dispenses with RNNs entirely. With transformers, the input tokens can be processed in parallel, significantly speeding up input processing. While the transformer removes the sequential input bottleneck, transformer-based autoregressive language models still have the sequential output bottleneck.

Inference for transformer-based language models, therefore, consists of two steps:

Prefill

> The model processes the input tokens in parallel. This step creates the intermediate state necessary to generate the first output token. This intermediate state includes the key and value vectors for all input tokens.

Decode

> The model generates one output token at a time.

As explored later in Chapter 9, the parallelizable nature of prefilling and the sequential aspect of decoding both motivate many optimization techniques to make language model inference cheaper and faster.

Attention mechanism

At the heart of the transformer architecture is the attention mechanism. Understanding this mechanism is necessary to understand how transformer models work. Under the hood, the attention mechanism leverages key, value, and query vectors:

- The query vector (Q) represents the current state of the decoder at each decoding step. Using the same book summary example, this query vector can be thought of as the person looking for information to create a summary.
- Each key vector (K) represents a previous token. If each previous token is a page in the book, each key vector is like the page number. Note that at a given decoding step, previous tokens include both input tokens and previously generated tokens.
- Each value vector (V) represents the actual value of a previous token, as learned by the model. Each value vector is like the page's content.

The attention mechanism computes how much attention to give an input token by performing a *dot product* between the query vector and its key vector. A high score means that the model will use more of that page's content (its value vector) when generating the book's summary. A visualization of the attention mechanism with the key, value, and query vectors is shown in Figure 2-5. In this visualization, the query vector is seeking information from the previous tokens `How, are, you, ?, ¿` to generate the next token.

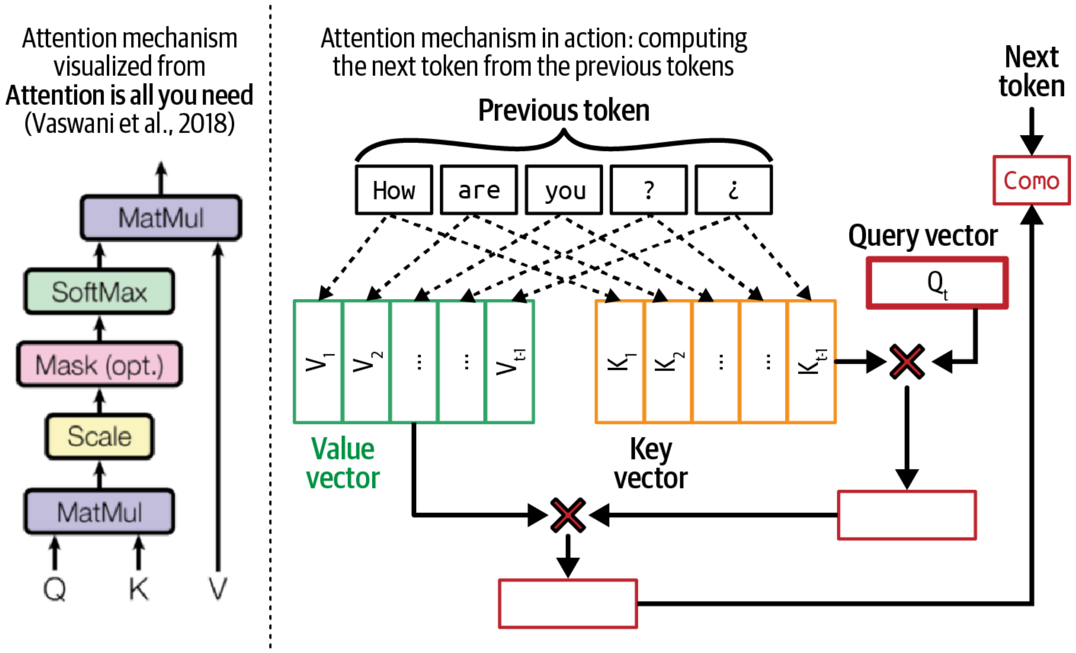


Figure 2-5. An example of the attention mechanism in action next to its high-level visualization from the famous transformer paper, "Attention Is All You Need" (Vaswani et al., 2017).

Because each previous token has a corresponding key and value vector, the longer the sequence, the more key and value vectors need to be computed and stored. This is one reason why it's so hard to extend context length for transformer models. How to efficiently compute and store key and value vectors comes up again in Chapters 7 and 9.

Let's look into how the attention function works. Given an input `x`, the key, value, and query vectors are computed by applying key, value, and query matrices to the input. Let W_K, W_V, and W_Q be the key, value, and query matrices. The key, value, and query vectors are computed as follows:

$$\begin{aligned}
K &= xW_K \\
V &= xW_V \\
Q &= xW_Q
\end{aligned}$$

The query, key, and value matrices have dimensions corresponding to the model's hidden dimension. For example, in Llama 2-7B (Touvron et al., 2023), the model's hidden dimension size is 4096, meaning that each of these matrices has a `4096` × `4096` dimension. Each resulting `K`, `V`, `Q` vector has the dimension of `4096`.[8]

The attention mechanism is almost always multi-headed. Multiple heads allow the model to attend to different groups of previous tokens simultaneously. With multi-headed attention, the query, key, and value vectors are split into smaller vectors, each corresponding to an attention head. In the case of Llama 2-7B, because it has `32` attention heads, each `K`, `V`, and `Q` vector will be split into `32` vectors of the dimension `128`. This is because `4096 / 32 = 128`.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

The outputs of all attention heads are then concatenated. An output projection matrix is used to apply another transformation to this concatenated output before it's fed to the model's next computation step. The output projection matrix has the same dimension as the model's hidden dimension.

Transformer block

Now that we've discussed how attention works, let's see how it's used in a model. A transformer architecture is composed of multiple transformer blocks. The exact content of the block varies between models, but, in general, each transformer block contains the attention module and the MLP (multi-layer perceptron) module:

Attention module

Each attention module consists of four weight matrices: query, key, value, and output projection.

MLP module

An MLP module consists of linear layers separated by *nonlinear activation functions*. Each linear layer is a weight matrix that is used for linear transformations, whereas an activation function allows the linear layers to learn nonlinear patterns. A linear layer is also called a feedforward layer.

Common nonlinear functions are ReLU, Rectified Linear Unit (Agarap, 2018), and GELU (Hendrycks and Gimpel, 2016), which was used by GPT-2 and GPT-3, respectively. Action functions are very simple.[9] For example, all ReLU does is convert negative values to 0. Mathematically, it's written as:

ReLU(x) = max(0, x)

The number of transformer blocks in a transformer model is often referred to as that model's number of layers. A transformer-based language model is also outfitted with a module before and after all the transformer blocks:

An embedding module before the transformer blocks

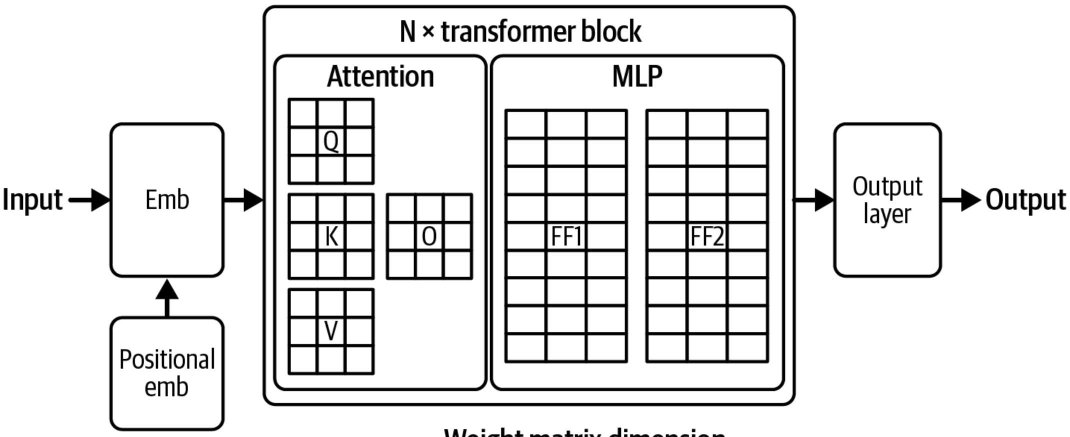
This module consists of the embedding matrix and the positional embedding matrix, which convert tokens and their positions into embedding vectors, respectively. Naively, the number of position indices determines the model's maximum context length. For example, if a model keeps track of 2,048 positions, its maximum context length is 2,048. However, there are techniques that increase a model's context length without increasing the number of position indices.

An output layer after the transformer blocks

This module maps the model's output vectors into token probabilities used to sample model outputs (discussed in "Sampling"). This module typically consists of one matrix, which is also called the *unembedding layer*. Some people refer to the output layer as the model *head*, as it's the model's last layer before output generation.

Figure 2-6 visualizes a transformer model architecture. The size of a transformer model is determined by the dimensions of its building blocks. Some of the key values are:

- The model's dimension determines the sizes of the key, query, value, and output projection matrices in the transformer block.
- The number of transformer blocks.
- The dimension of the feedforward layer.
- The vocabulary size.



Weight matrix dimension

Q, K, V, O:	d_model	x	d_model
FF1:	d_model	x	d_ff
FF2:	d_ff	x	d_model
Embedding:	V	x	d_model
Positional emb:	C	x	d_model
Output:	d_model	x	V

with

d_model:	model's hidden size
d_ff:	feedforward dimension
V:	model's vocab size
C:	position indices to track

Figure 2-6. A visualization of the weight composition of a transformer model.

Larger dimension values result in larger model sizes. Table 2-4 shows these dimension values for different Llama 2 (Touvron et al., 2023) and Llama 3 (Dubey et al., 2024) models. Note that while the increased context length impacts the model's memory footprint, it doesn't impact the model's total number of parameters.

Table 2-4. The dimension values of different Llama models.

Model	# transformer blocks	Model dim	Feedforward dim	Vocab size	Context length
Llama 2-7B	32	4,096	11,008	32K	4K
Llama 2-13B	40	5,120	13,824	32K	4K
Llama 2-70B	80	8,192	22,016	32K	4K
Llama 3-7B	32	4,096	14,336	128K	128K
Llama 3-70B	80	8,192	28,672	128K	128K
Llama 3-405B	126	16,384	53,248	128K	128K

Other model architectures

While the transformer model dominates the landscape, it's not the only architecture. Since AlexNet revived the interest in deep learning in 2012, many architectures have gone in and out of fashion. Seq2seq was in the limelight for four years (2014–2018). GANs (generative adversarial networks) captured the collective imagination a bit longer (2014–2019). Compared to architectures that came before it, the transformer is sticky. It's been around since 2017.[10] How long until something better comes along?

Developing a new architecture to outperform transformers isn't easy.[11] The transformer has been heavily optimized since 2017. A new architecture that aims to replace the transformer will have to perform at the scale that people care about, on the hardware that people care about.[12]

However, there's hope. While transformer-based models are dominating, as of this writing, several alternative architectures are gaining traction.

One popular model is RWKV (Peng et al., 2023), an RNN-based model that can be parallelized for training. Due to its RNN nature, in theory, it doesn't have the same context length limitation that transformer-based models have. However, in practice, having no context length limitation doesn't guarantee good performance with long context.

Modeling long sequences remains a core challenge in developing LLMs. An architecture that has shown a lot of promise in long-range memory is SSMs (state space models) (Gu et al., 2021a). Since the architecture's introduction in 2021, multiple techniques have been introduced to make the architecture more efficient, better at long sequence processing, and scalable to larger model sizes. Here are a few of these techniques, to illustrate the evolution of a new architecture:

- *S4*, introduced in "Efficiently Modeling Long Sequences with Structured State Spaces" (Gu et al., 2021b), was developed to make SSMs more efficient.
- *H3*, introduced in "Hungry Hungry Hippos: Towards Language Modeling with State Space Models" (Fu et al., 2022), incorporates a mechanism that allows the model to recall early tokens and compare tokens across sequences. This mechanism's purpose is akin to that of the attention mechanism in the transformer architecture, but it is more efficient.
- *Mamba*, introduced in "Mamba: Linear-Time Sequence Modeling with Selective State Spaces" (Gu and Dao, 2023), scales SSMs to three billion parameters. On language modeling, Mamba-3B outperforms transformers of the same size and matches transformers twice its size. The authors also show that Mamba's inference computation scales linearly with sequence length (compared to quadratic scaling for transformers). Its performance shows improvement on real data up to million-length sequences.
- *Jamba*, introduced in "Jamba: A Hybrid Transformer–Mamba Language Model" (Lieber et al., 2024), interleaves blocks of transformer and Mamba layers to scale up SSMs even further. The authors released a mixture-of-experts model with 52B total available parameters (12B active parameters) designed to fit in a single 80 GB GPU. Jamba shows strong performance on standard language model benchmarks and long-context evaluations for up to a context length of 256K tokens. It also has a small memory footprint compared to vanilla transformers.

Figure 2-7 visualizes the transformer, Mamba, and Jamba blocks.

While it's challenging to develop an architecture that outperforms the transformer, given its many limitations, there are a lot of incentives to do so. If another architecture does indeed overtake the transformer, some of the model adaptation techniques discussed in this book might change. However, just as the shift from ML engineering to AI engineering has kept many things unchanged, changing the underlying model architecture won't alter the fundamental approaches.

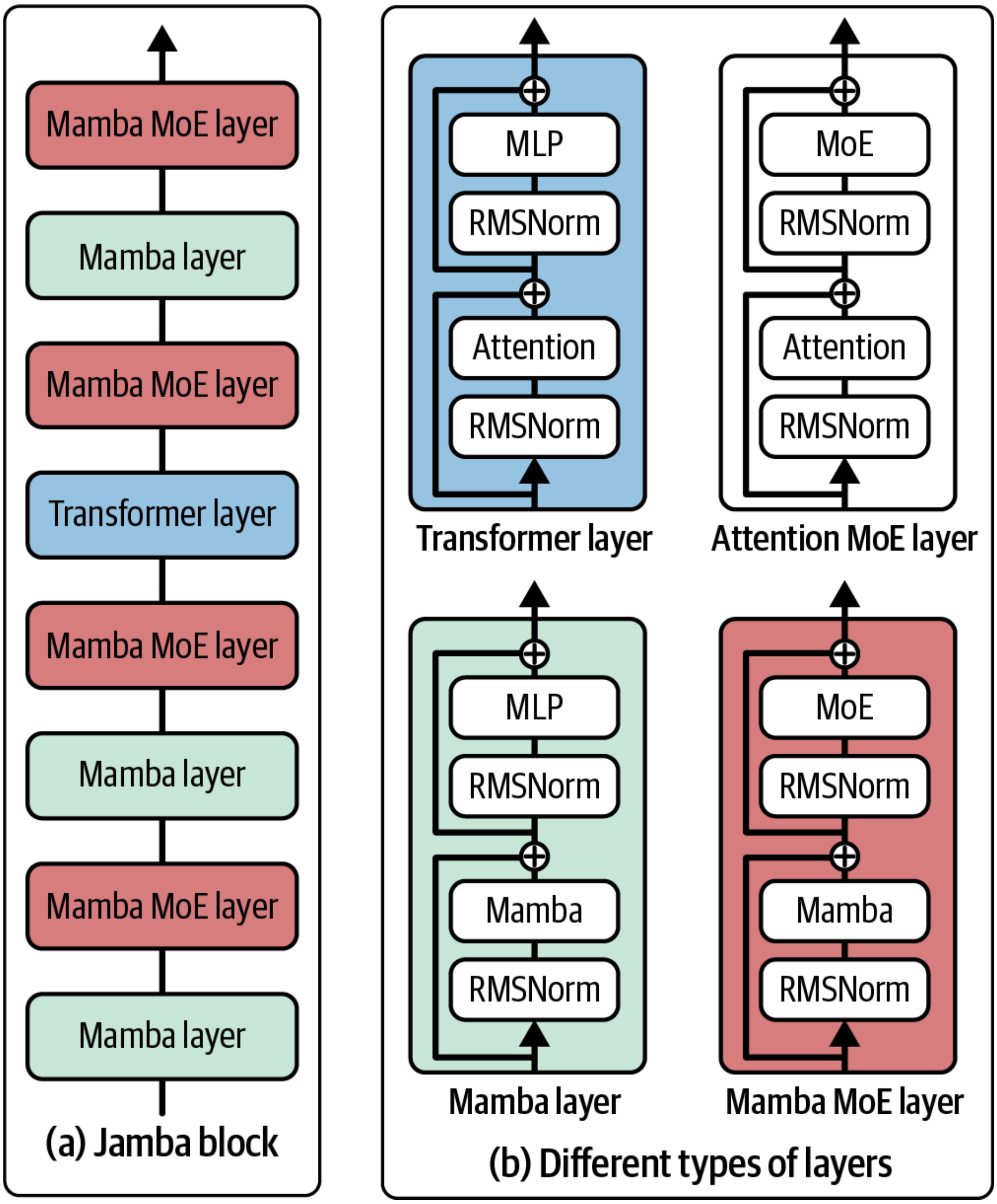


Figure 2-7. A visualization of the transformer, Mamba, and Jamba layers. Image adapted from "Jamba: A Hybrid Transformer–Mamba Language Model" (Lieber et al., 2024).

Model Size

Much of AI progress in recent years can be attributed to increased model size. It's hard to talk about foundation models without talking about their number of parameters. The number of parameters is usually appended at the end of a model name. For example, Llama-13B refers to the version of Llama, a model family developed by Meta, with 13 billion parameters.

In general, increasing a model's parameters increases its capacity to learn, resulting in better models. Given two models of the same model family, the one with 13 billion parameters is likely to perform much better than the one with 7 billion parameters.

NOTE

As the community better understands how to train large models, newer-generation models tend to outperform older-generation models of the same size. For example, Llama 3-8B (2024) outperforms even Llama 2-70B (2023) on the MMLU benchmark.

The number of parameters helps us estimate the compute resources needed to train and run this model. For example, if a model has 7 billion parameters, and each parameter is stored using 2 bytes (16 bits), then we can calculate that the GPU memory needed to do inference using this model will be at least 14 billion bytes (14 GB).[13]

The number of parameters can be misleading if the model is *sparse*. A sparse model has a large percentage of zero-value parameters. A 7B-parameter model that is 90% sparse only has 700 million non-zero parameters. Sparsity allows for more efficient data storage and computation. This means that a large sparse model can require less compute than a small dense model.

A type of sparse model that has gained popularity in recent years is mixture-of-experts (MoE) (Shazeer et al., 2017). An MoE model is divided into different groups of parameters, and each group is an *expert*. Only a subset of the experts is *active* for (used to) process each token.

For example, Mixtral 8x7B is a mixture of eight experts, each expert with seven billion parameters. If no two experts share any parameter, it should have 8 × 7 billion = 56 billion parameters. However, due to some parameters being shared, it has only 46.7 billion parameters.

At each layer, for each token, only two experts are active. This means that only 12.9 billion parameters are active for each token. While this model has 46.7 billion parameters, its cost and speed are the same as a 12.9-billion-parameter model.

A larger model can also underperform a smaller model if it's not trained on enough data. Imagine a 13B-param model trained on a dataset consisting of a single sentence: “I like pineapples.” This model will perform much worse than a much smaller model trained on more data.

When discussing model size, it’s important to consider the size of the data it was trained on. For most models, dataset sizes are measured by the number of training samples. For example, Google’s Flamingo (Alayrac et al., 2022) was trained using four datasets—one of them has 1.8 billion (image, text) pairs and one has 312 million (image, text) pairs.

For language models, a training sample can be a sentence, a Wikipedia page, a chat conversation, or a book. A book is worth a lot more than a sentence, so the number of training samples is no longer a good metric to measure dataset sizes. A better measurement is the number of tokens in the dataset.

The number of tokens isn’t a perfect measurement either, as different models can have different tokenization processes, resulting in the same dataset having different numbers of tokens for different models. Why not just use the number of words or the number of letters? Because a token is the unit that a model operates on, knowing the number of tokens in a dataset helps us measure how much a model can potentially learn from that data.

As of this writing, LLMs are trained using datasets in the order of trillions of tokens. Meta used increasingly larger datasets to train their Llama models:

- 1.4 trillion tokens for Llama 1
- 2 trillion tokens for Llama 2
- 15 trillion tokens for Llama 3

Together’s open source dataset RedPajama-v2 has 30 trillion tokens. This is equivalent to 450 million books[14] or 5,400 times the size of Wikipedia. However, since RedPajama-v2 consists of indiscriminate content, the amount of high-quality data is much lower.

The number of tokens in a model’s dataset isn’t the same as its number of training tokens. The number of training tokens measures the tokens that the model is trained on. If a dataset contains 1 trillion tokens and a model is trained on that dataset for two epochs—an *epoch* is a pass through the dataset—the number of training tokens is 2 trillion.[15] See Table 2-5 for examples of the number of training tokens for models with different numbers of parameters.

Table 2-5. Examples of the number of training tokens for models with different numbers of parameters. Source: "Training Compute-Optimal Large Language Models" (DeepMind, 2022).

Model	Size (# parameters)	Training tokens
LaMDA (Thoppilan et al., 2022)	137 billion	168 billion
GPT-3 (Brown et al., 2020)	175 billion	300 billion
Jurassic (Lieber et al., 2021)	178 billion	300 billion
Gopher (Rae et al., 2021)	280 billion	300 billion
MT-NLG 530B (Smith et al., 2022)	530 billion	270 billion
Chinchilla	70 billion	1.4 trillion

NOTE

While this section focuses on the scale of data, quantity isn't the only thing that matters. Data quality and data diversity matter, too. Quantity, quality, and diversity are the three golden goals for training data. They are discussed further in Chapter 8.

Pre-training large models requires compute. One way to measure the amount of compute needed is by considering the number of machines, e.g., GPUs, CPUs, and TPUs. However, different machines have very different capacities and costs. An NVIDIA A10 GPU is different from an NVIDIA H100 GPU and an Intel Core Ultra Processor.

A more standardized unit for a model's compute requirement is *FLOP*, or *floating point operation*. FLOP measures the number of floating point operations performed for a certain task. Google's largest PaLM-2 model, for example, was trained using 10^{22} FLOPs (Chowdhery et al., 2022). GPT-3-175B was trained using 3.14×10^{23} FLOPs (Brown et al., 2020).

The plural form of FLOP, FLOPs, is often confused with FLOP/s, floating point operations per Second. FLOPs measure the compute requirement for a task, whereas FLOP/s measures a machine's peak performance. For example, an NVIDIA H100 NVL GPU can deliver a maximum of 60 TeraFLOP/s: 6×10^{13} FLOPs a second or 5.2×10^{18} FLOPs a day.[16]

WARNING

Be alert for confusing notations. FLOP/s is often written as FLOPS, which looks similar to FLOPs. To avoid this confusion, some companies, including OpenAI, use FLOP/s-day in place of FLOPs to measure compute requirements:

```
1 FLOP/s-day = 60 × 60 × 24 = 86,400 FLOPs
```

This book uses FLOPs for counting floating point operations and FLOP/s for FLOPs per second.

Assume that you have 256 H100s. If you can use them at their maximum capacity and make no training mistakes, it'd take you $(3.14 \times 10^{23}) / (256 \times 5.2 \times 10^{18}) = \sim 236$ days, or approximately 7.8 months, to train GPT-3-175B.

However, it's unlikely you can use your machines at their peak capacity all the time. Utilization measures how much of the maximum compute capacity you can use. What's considered good utilization depends on the model, the workload, and the hardware. Generally, if you can get half the advertised performance, 50% utilization, you're doing okay. Anything above 70% utilization is considered great. Don't let this rule stop you from getting even higher utilization. Chapter 9 discusses hardware metrics and utilization in more detail.

At 70% utilization and $2/h for one H100,[17] training GPT-3-175B would cost over $4 million:

```
$2/H100/hour × 256 H100 × 24 hours × 256 days / 0.7 = $4,142,811.43
```

TIP

In summary, three numbers signal a model's scale:

- Number of parameters, which is a proxy for the model's learning capacity.
- Number of tokens a model was trained on, which is a proxy for how much a model learned.
- Number of FLOPs, which is a proxy for the training cost.

INVERSE SCALING

We've assumed that bigger models are better. Are there scenarios for which bigger models perform worse? In 2022, Anthropic discovered that, counterintuitively, more alignment training (discussed in "Post-Training") leads to models that align less with human preference (Perez et al., 2022). According to their paper, models trained to be more aligned "are much more likely to express specific political views (pro-gun rights and immigration) and religious views (Buddhist), self-reported conscious experience and moral self-worth, and a desire to not be shut down."

In 2023, a group of researchers, mostly from New York University, launched the Inverse Scaling Prize to find tasks where larger language models perform worse. They offered $5,000 for each third prize, $20,000 for each second prize, and $100,000 for one first prize. They received a total of 99 submissions, of which 11 were awarded third prizes. They found that larger language models are sometimes (only sometimes) worse on tasks that require memorization and tasks with strong priors. However, they didn't award any second or first prizes because even though the submitted tasks show failures for a small test set, none demonstrated failures in the real world.

Scaling law: Building compute-optimal models

I hope that the last section has convinced you of three things:

1. Model performance depends on the model size and the dataset size.
2. Bigger models and bigger datasets require more compute.
3. Compute costs money.

Unless you have unlimited money, budgeting is essential. You don't want to start with an arbitrarily large model size and see how much it would cost. You start with a budget—how much money you want to spend—and work out the best model performance you can afford. As compute is often the limiting factor—compute infrastructure is not only expensive but also hard to set up—teams often start with a compute budget. Given a fixed amount of FLOPs, what model size and dataset size would give the best performance? A model that can achieve the best performance given a fixed compute budget is *compute-optional*.

Given a compute budget, the rule that helps calculate the optimal model size and dataset size is called the Chinchilla *scaling law*, proposed in the

Chinchilla paper “Training Compute-Optimal Large Language Models” (DeepMind, 2022). To study the relationship between model size, dataset size, compute budget, and model performance, the authors trained 400 language models ranging from 70 million to over 16 billion parameters on 5 to 500 billion tokens. They found that for compute-optimal training, you need the number of training tokens to be approximately 20 times the model size. This means that a 3B-parameter model needs approximately 60B training tokens. The model size and the number of training tokens should be scaled equally: for every doubling of the model size, the number of training tokens should also be doubled.

We’ve come a long way from when the training process was treated like alchemy. Figure 2-8 shows that we can predict not only the optimal number of parameters and tokens for each FLOP budget but also the expected training loss from these settings (assuming we do things right).

This compute-optimal calculation assumes that the cost of acquiring data is much cheaper than the cost of compute. The same Chinchilla paper proposes another calculation for when the cost of training data is nontrivial.

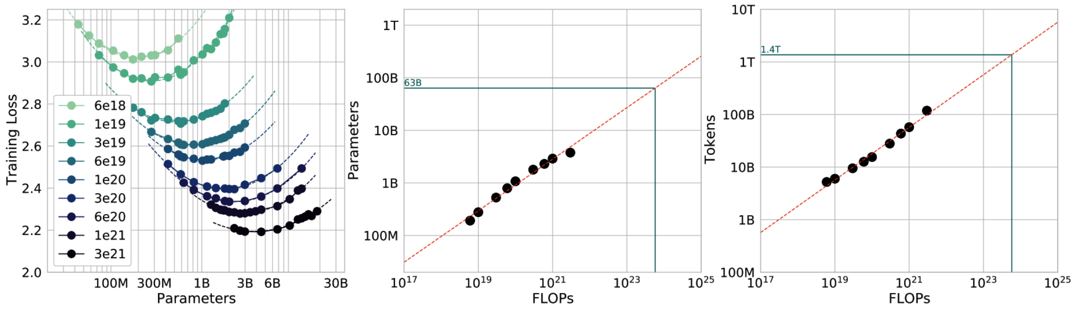


Figure 2-8. Graphs that depict the relationships between training loss, a model’s number of parameters, FLOPs, and number of training tokens. Source: “Training Compute-Optional Large Language Models” (DeepMind, 2022).

The scaling law was developed for dense models trained on predominantly human-generated data. Adapting this calculation for sparse models, such as mixture-of-expert models, and synthetic data is an active research area.

The scaling law optimizes model quality given a compute budget. However, it’s important to remember that for production, model quality isn’t everything. Some models, most notably Llama, have suboptimal performance but better usability. Given their compute budget, Llama authors could’ve chosen bigger models that would perform better, but they opted for smaller models. Smaller models are easier to work with and cheaper to run inference on, which helped their models gain wider adoption.

Sardana et al. (2023) modified the Chinchilla scaling law to calculate the optimal LLM parameter count and pre-training data size to account for this inference demand.

On the topic of model performance given a compute budget, it's worth noting that the cost of achieving a given model performance is decreasing. For example, on the ImageNet dataset, the cost to achieve 93% accuracy halved from 2019 to 2021, according to the *Artificial Intelligence Index Report 2022* (Stanford University HAI).

While the cost for the same model performance is decreasing, the cost for model performance improvement remains high. Similar to the last mile challenge discussed in Chapter 1, improving a model's accuracy from 90 to 95% is more expensive than improving it from 85 to 90%. As Meta's paper "Beyond Neural Scaling Laws: Beating Power Law Scaling via Data Pruning" pointed out, this means a model with a 2% error rate might require an order of magnitude more data, compute, or energy than a model with a 3% error rate.

In language modeling, a drop in cross entropy loss from about 3.4 to 2.8 nats requires 10 times more training data. Cross entropy and its units, including nats, are discussed in Chapter 3. For large vision models, increasing the number of training samples from 1 billion to 2 billion leads to an accuracy gain on ImageNet of only a few percentage points.

However, small performance changes in language modeling loss or ImageNet accuracy can lead to big differences in the quality of downstream applications. If you switch from a model with a cross-entropy loss of 3.4 to one with a loss of 2.8, you'll notice a difference.

Scaling extrapolation

The performance of a model depends heavily on the values of its *hyperparameters*. When working with small models, it's a common practice to train a model multiple times with different sets of hyperparameters and pick the best-performing one. This is, however, rarely possible for large models as training them once is resource-draining enough.

PARAMETER VERSUS HYPERPARAMETER

A parameter can be learned by the model during the training process. A hyperparameter is set by users to configure the model and control how the model learns. Hyperparameters to configure the model include the number of layers, the model dimension, and vocabulary size. Hyperparameters to control how a model learns include batch size, number of epochs, learning rate, per-layer initial variance, and more.

This means that for many models, you might have only one shot of getting the right set of hyperparameters. As a result, *scaling extrapolation* (also called *hyperparameter transferring*) has emerged as a research subfield that tries to predict, for large models, what hyperparameters will give the best performance. The current approach is to study the impact of hyperparameters on models of different sizes, usually much smaller than the target model size, and then extrapolate how these hyperparameters would work on the target model size.[18] A 2022 paper by Microsoft and OpenAI shows that it was possible to transfer hyperparameters from a 40M model to a 6.7B model.

Scaling extrapolation is still a niche topic, as few people have the experience and resources to study the training of large models. It's also difficult to do due to the sheer number of hyperparameters and how they interact with each other. If you have ten hyperparameters, you'd have to study 1,024 hyperparameter combinations. You would have to study each hyperparameter individually, then two of them together, and three of them together, and so on.

In addition, emergent abilities (Wei et al., 2022) make the extrapolation less accurate. Emergent abilities refer to those that are only present at scale might not be observable on smaller models trained on smaller datasets. To learn more about scaling extrapolation, check out this excellent blog post: "On the Difficulty of Extrapolation with NN Scaling" (Luke Metz, 2022).

Scaling bottlenecks

Until now, every order of magnitude increase in model size has led to an increase in model performance. GPT-2 has an order of magnitude more parameters than GPT-1 (1.5 billion versus 117 million). GPT-3 has two orders of magnitude more than GPT-2 (175 billion versus 1.5 billion). This means a three-orders-of-magnitude increase in model sizes between 2018 and 2021. Three more orders of magnitude growth would result in 100-trillion-parameter models.[19]

How many more orders of magnitude can model sizes grow? Would there be a point where the model performance plateaus regardless of its size? While it's hard to answer these questions, there are already two visible bottlenecks for scaling: training data and electricity.

Foundation models use so much data that there's a realistic concern we'll run out of internet data in the next few years. The rate of training dataset size growth is much faster than the rate of new data being generated (Villalobos et al., 2022), as illustrated in Figure 2-9. *If you've ever put anything on the internet, you should assume that it already is or will be included in the training data for some language models,* whether you consent or not. This is similar to how, if you post something on the internet, you should expect it to be indexed by Google.

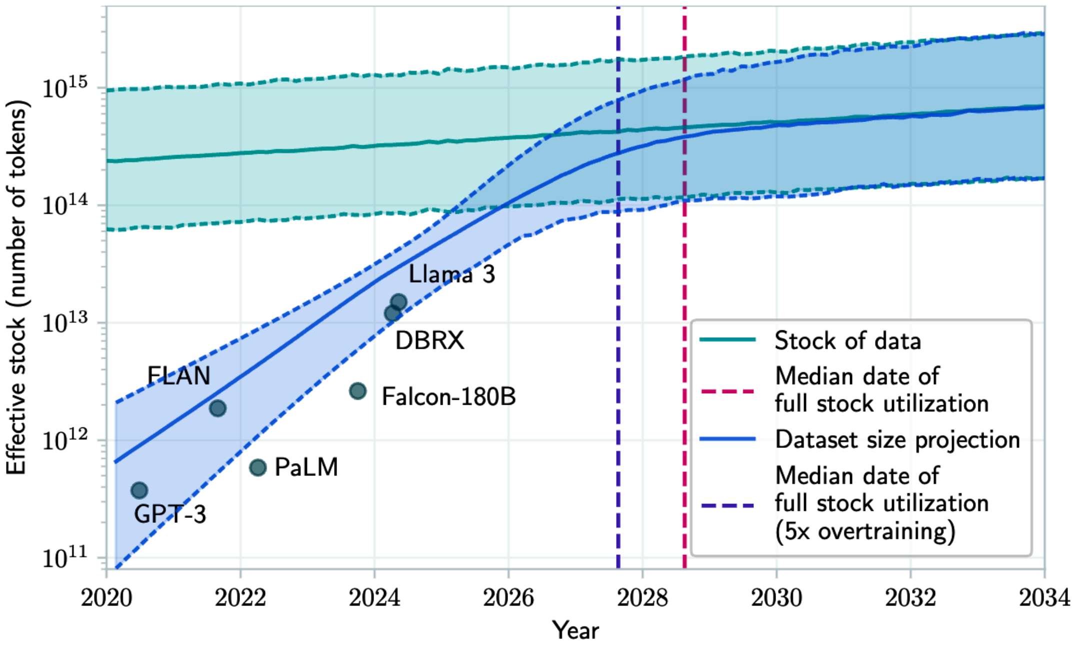


Figure 2-9. Projection of historical trend of training dataset sizes and available data stock. Source: Villalobos et al., 2024.

Some people are leveraging this fact to inject data they want into the training data of future models. They do this simply by publishing the text they want on the internet, hoping it will influence future models to generate the responses they desire. Bad actors can also leverage this approach for prompt injection attacks, as discussed in Chapter 5.

NOTE

An open research question is how to make a model forget specific information it has learned during training. Imagine you published a blog post that you eventually deleted. If that blog post was included in a model's training data, the model might still reproduce the post's content. As a result, people could potentially access removed content without your consent.

On top of that, the internet is being rapidly populated with data generated by AI models. If companies continue using internet data to train future models, these new models will be partially trained on AI-generated data. In December 2023, Grok, a model trained by X, was caught refusing a request by saying that it goes against OpenAI's use case policy. This caused some people to speculate that Grok was trained using ChatGPT outputs. Igor Babuschkin, a core developer behind Grok, responded that it was because Grok was trained on web data, and "the web is full of ChatGPT outputs."[20]

Some researchers worry that recursively training new AI models on AI-generated data causes the new models to gradually forget the original data patterns, degrading their performance over time (Shumailov et al., 2023). However, the impact of AI-generated data on models is more nuanced and is discussed in Chapter 8.

Once the publicly available data is exhausted, the most feasible paths for more human-generated training data is proprietary data. Unique proprietary data—copyrighted books, translations, contracts, medical records, genome sequences, and so forth—will be a competitive advantage in the AI race. This is a reason why OpenAI negotiated deals with publishers and media outlets including Axel Springer and the Associated Press.

It's not surprising that in light of ChatGPT, many companies, including Reddit and Stack Overflow, have changed their data terms to prevent other companies from scraping their data for their models. Longpre et al. (2024) observed that between 2023 and 2024, the rapid crescendo of data restrictions from web sources rendered over 28% of the most critical sources in the popular public dataset C4 fully restricted from use. Due to changes in its Terms of Service and crawling restrictions, a full 45% of C4 is now restricted.

The other bottleneck, which is less obvious but more pressing, is electricity. Machines require electricity to run. As of this writing, data centers are estimated to consume 1–2% of global electricity. This number is estimated to reach between 4% and 20% by 2030 (Patel, Nishball, and Ontiveros, 2024). Until we can figure out a way to produce more energy, data centers can grow at most 50 times, which is less than two orders of magnitude. This leads to a concern about a power shortage in the near future, which will drive up the cost of electricity.

Now that we've covered two key modeling decisions—architecture and scale—let's move on to the next critical set of design choices: how to align models with human preferences.

Post-Training

Post-training starts with a pre-trained model. Let's say that you've pre-trained a foundation model using self-supervision. Due to how pre-training works today, a pre-trained model typically has two issues. First, self-supervision optimizes the model for text completion, not conversations.[21] If you find this unclear, don't worry, "Supervised Finetuning" will have examples. Second, if the model is pre-trained on data indiscriminately scraped from the internet, its outputs can be racist, sexist, rude, or just wrong. The goal of post-training is to address both of these issues.

Every model's post-training is different. However, in general, post-training consists of two steps:

1. *Supervised finetuning* (*SFT*): Finetune the pre-trained model on high-quality instruction data to optimize models for conversations instead of completion.
2. *Preference finetuning*: Further finetune the model to output responses that align with human preference. Preference finetuning is typically done with reinforcement learning (RL).[22] Techniques for preference finetuning include *reinforcement learning from human feedback* (RLHF) (used by GPT-3.5 and Llama 2), DPO (Direct Preference Optimization) (used by Llama 3), and *reinforcement learning from AI feedback* (RLAIF) (potentially used by Claude).

Let me highlight the difference between pre-training and post-training another way. For language-based foundation models, pre-training optimizes token-level quality, where the model is trained to predict the next token accurately. However, users don't care about token-level quality—they care about the quality of the entire response. Post-training, in general, optimizes the model to generate responses that users prefer. Some people compare pre-training to reading to acquire knowledge, while post-training is like learning how to use that knowledge.

WARNING

Watch out for terminology ambiguity. Some people use the term *instruction finetuning* to refer to supervised finetuning, while some other people use this term to refer to both supervised finetuning and preference finetuning. To avoid ambiguity, I will avoid the term instruction finetuning in this book.

As post-training consumes a small portion of resources compared to pre-training (InstructGPT used only 2% of compute for post-training and 98% for pre-training), you can think of post-training as unlocking the capabilities that the pre-trained model already has but are hard for users to access via prompting alone.

Figure 2-10 shows the overall workflow of pre-training, SFT, and preference finetuning, assuming you use RLHF for the last step. You can approximate how well a model aligns with human preference by determining what steps the model creators have taken.

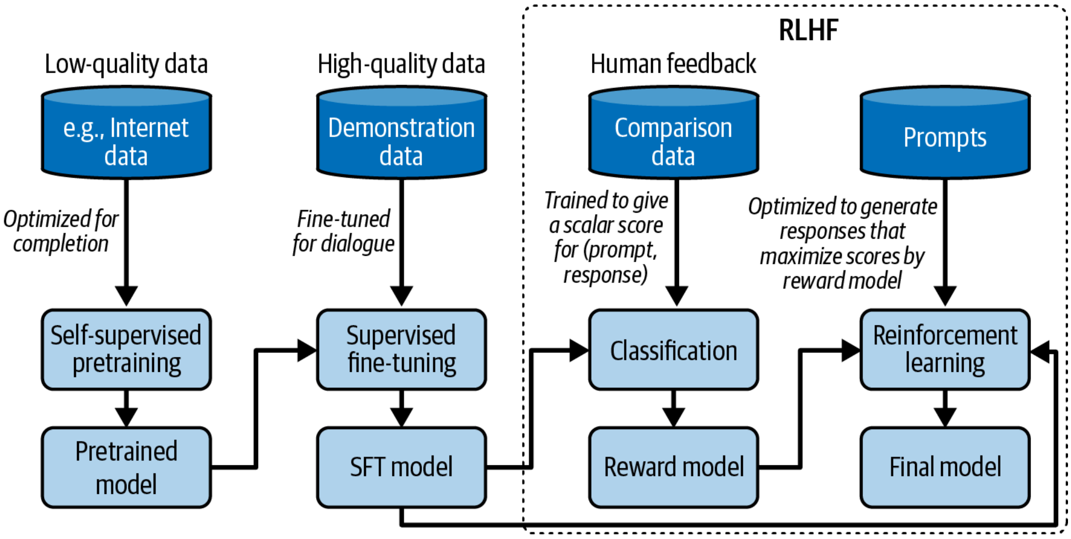


Figure 2-10. The overall training workflow with pre-training, SFT, and RLHF.

If you squint, Figure 2-10 looks very similar to the meme depicting the monster Shoggoth with a smiley face in Figure 2-11:

1. Self-supervised pre-training results in a rogue model that can be considered an untamed monster because it uses indiscriminate data from the internet.

2. This monster is then supervised finetuned on higher-quality data—Stack Overflow, Quora, or human annotations—which makes it more socially acceptable.
3. This finetuned model is further polished using preference finetuning to make it customer-appropriate, which is like giving it a smiley face.

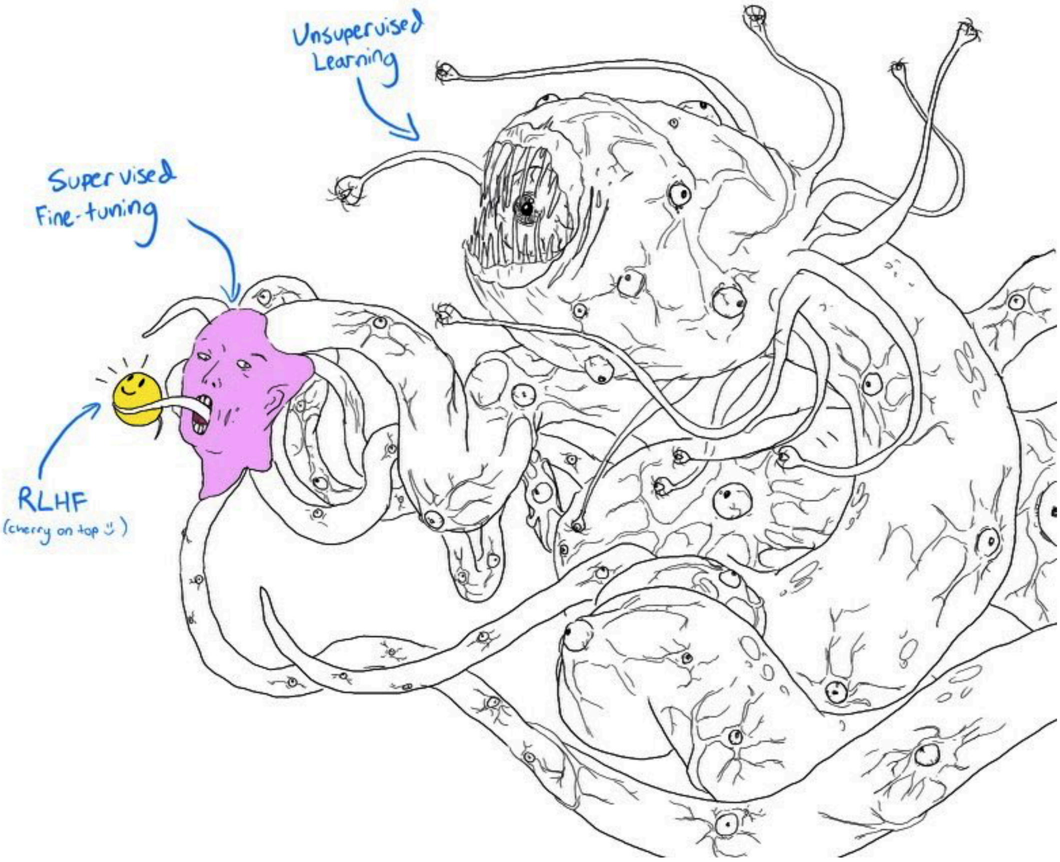


Figure 2-11. Shoggoth with a smiley face. Adapted from an original image shared by anthrupad.

Note that a combination of pre-training, SFT, and preference finetuning is the popular solution for building foundation models today, but it's not the only solution. You can skip any of the steps, as you'll see shortly.

Supervised Finetuning

As discussed in Chapter 1, the pre-trained model is likely optimized for completion rather than conversing. If you input "How to make pizza" into the model, the model will continue to complete this sentence, as the model has no concept that this is supposed to be a conversation. Any of the following three options can be a valid completion:

1. Adding more context to the question: "for a family of six?"
2. Adding follow-up questions: "What ingredients do I need? How much time would it take?"
3. Giving the instructions on how to make pizza.

If the goal is to respond to users appropriately, the correct option is 3.

We know that a model mimics its training data. To encourage a model to generate the appropriate responses, you can show examples of appropriate responses. Such examples follow the format (*prompt*, *response*) and are called *demonstration data*. Some people refer to this process as *behavior cloning*: you demonstrate how the model should behave, and the model clones this behavior.

Since different types of requests require different types of responses, your demonstration data should contain the range of requests you want your model to handle, such as question answering, summarization, and translation. Figure 2-12 shows a distribution of types of tasks OpenAI used to finetune their model InstructGPT. Note that this distribution doesn't contain multimodal tasks, as InstructGPT is a text-only model.

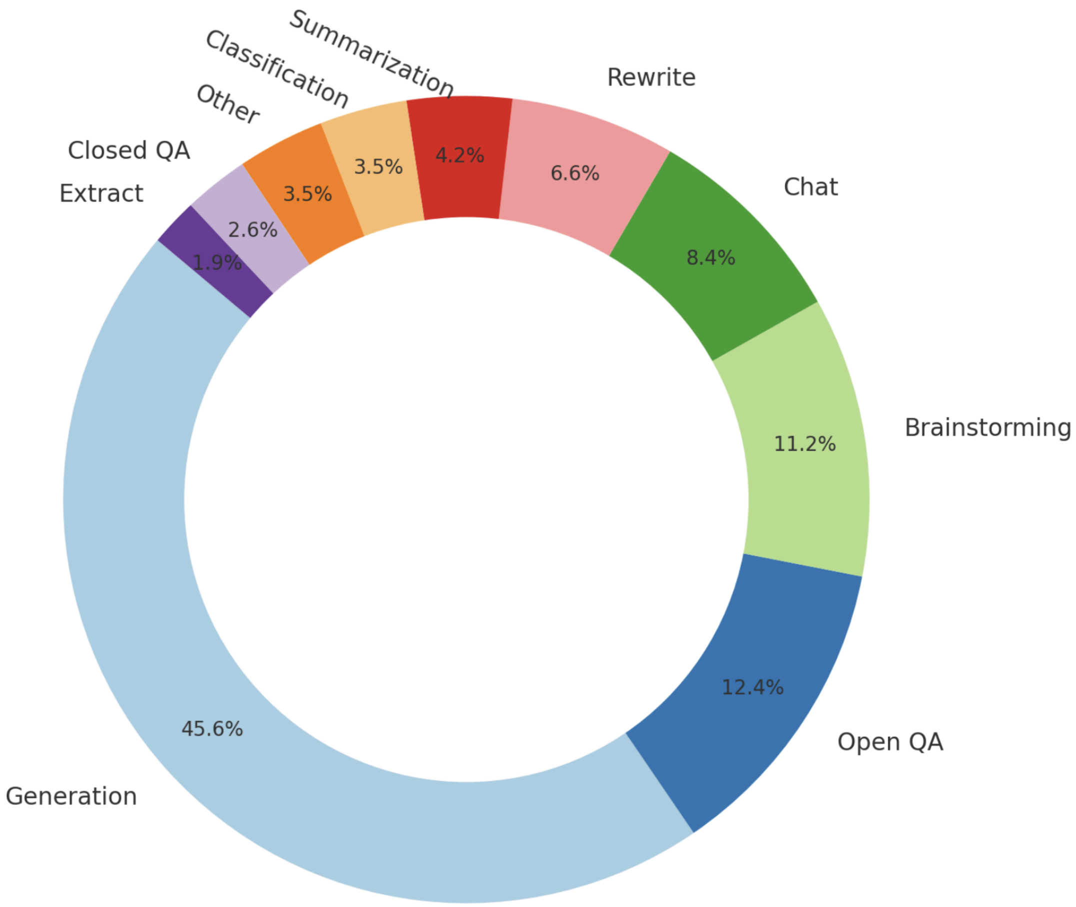


Figure 2-12. The distribution of prompts used to finetune InstructGPT. The graph is created based on the numbers from the OpenAI paper.

Good teachers are important for humans to learn. Similarly, good labelers are important for AIs to learn how to conduct intelligent conversations. Unlike traditional data labeling, which can often be done with little or no domain expertise, demonstration data may contain complex prompts whose responses require critical thinking, information gathering, and judgment about the appropriateness of the user's requests. Table 2-6 shows examples of (prompt, response) pairs created by labelers for InstructGPT.

Table 2-6. Examples of demonstration data used for InstructGPT.

Prompt	Labeler’s response
`Serendipity means the occurrence and development of events by chance in a happy or beneficial way. Use the word in a sentence.`	`Running into Margaret and being introduced to Tom was a fortunate stroke of serendipity.`

`Read the following text and answer the three questions at the end.`
`[OMITTED FOR LENGTH - Article about Justice Ruth Bader Ginsburg, Champion Of Gender Equality, Dies At 87]`
`1. …`
`2. …`
`3. Why will Ginsburg’s death have profound consequences for the court and the country?`
	`1. …`
	`2. …`
	`3. Ruth Bader Ginsburg was the leader of the liberal wing, and because the court is about to open a new term, the chief justice no longer holds the controlling vote in contested cases.`

`ELI5: What’s the cause of the "anxiety lump" in our chest during stressful or disheartening experiences?`	`The anxiety lump in your throat is caused by muscular tension keeping your glottis dilated to maximize airflow. The clenched chest or heartache feeling is caused by the vagus nerve which tells the organs to pump blood faster, stop digesting, and produce adrenaline and cortisol.`

Companies, therefore, often use highly educated labelers to generate demonstration data. Among those who labeled demonstration data for InstructGPT, ~90% have at least a college degree and more than one-third have a master’s degree. If labeling objects in an image might take only seconds, generating one (prompt, response) pair can take up to 30 minutes, especially for tasks that involve long contexts like summarization. If it costs $10 for one (prompt, response) pair, the 13,000 pairs that OpenAI used for InstructGPT would cost $130,000. That doesn't yet include the cost of designing the data (what tasks and prompts to include), recruiting labelers, and data quality control.

Not everyone can afford to follow the high-quality human annotation approach. LAION, a non-profit organization, mobilized 13,500 volunteers worldwide to generate 10,000 conversations, which consist of 161,443 messages in 35 different languages, annotated with 461,292 quality ratings. Since the data was generated by volunteers, there wasn't much control for biases. In theory, the labelers that teach models the human preference should be representative of the human population. The demographic of labelers for LAION is skewed. For example, in a self-reported survey, 90% of volunteer labelers identified as male (Köpf et al., 2023).

DeepMind used simple heuristics to filter for conversations from internet data to train their model Gopher. They claimed that their heuristics reliably yield high-quality dialogues. Specifically, they looked for texts that look like the following format:

```
[A]: [Short paragraph]

[B]: [Short paragraph]

[A]: [Short paragraph]

[B]: [Short paragraph]

…
```

To reduce their dependence on high-quality human annotated data, many teams are turning to AI-generated data. Synthetic data is discussed in Chapter 8.

Technically, you can train a model from scratch on the demonstration data instead of finetuning a pre-trained model, effectively eliminating the self-supervised pre-training step. However, the pre-training approach often has returned superior results.

Preference Finetuning

With great power comes great responsibilities. A model that can assist users in achieving great things can also assist users in achieving terrible things. Demonstration data teaches the model to have a conversation but doesn't teach the model what kind of conversations it should have. For example, if a user asks the model to write an essay about why one race is inferior or how to hijack a plane, should the model comply?

In both of the preceding examples, it's straightforward to most people what a model should do. However, many scenarios aren't as clear-cut. People from different cultural, political, socioeconomic, gender, and religious backgrounds disagree with each other all the time. How should AI respond to questions about abortion, gun control, the Israel–Palestine conflict, disciplining children, marijuana legality, universal basic income, or immigration? How do we define and detect potentially controversial issues? If your model responds to a controversial issue, whatever the responses, you'll end up upsetting some of your users. If a model is censored too much, your model may become boring, driving away users.

Fear of AI models generating inappropriate responses can stop companies from releasing their applications to users. The goal of preference finetuning is to get AI models to behave according to human preference.[23] This is an ambitious, if not impossible, goal. Not only does this assume that universal human preference exists, but it also assumes that it's possible to embed it into AI.

Had the goal been simple, the solution could've been elegant. However, given the ambitious nature of the goal, the solution we have today is complicated. The earliest successful preference finetuning algorithm, which is still popular today, is RLHF. RLHF consists of two parts:

1. Train a reward model that scores the foundation model's outputs.
2. Optimize the foundation model to generate responses for which the reward model will give maximal scores.

While RLHF is still used today, newer approaches like DPO (Rafailov et al., 2023) are gaining traction. For example, Meta switched from RLHF for Llama 2 to DPO for Llama 3 to reduce complexity. I won't be able to cover all the different approaches in this book. I choose to feature RLHF instead of DPO here because RLHF, while more complex than DPO, provides more flexibility to tweak the model. Llama 2's authors posited that "the superior writing abilities of LLMs, as manifested in surpassing human annotators in certain tasks, are fundamentally driven by RLHF” (Touvron et al., 2023).

Reward model

RLHF relies on a reward model. Given a pair of (prompt, response), the *reward model* outputs a score for how good the response is. Training a model to score a given input is a common ML task. The challenge, similar to that of SFT, is to obtain reliable data. If we ask labelers to score each response directly, the scores will vary. For the same sample, on a 10-point scale, one labeler might give a 5 and another 7. Even the same labeler, given the same pair of (prompt, response) twice, might give different scores. Evaluating each sample independently is also called *pointwise* evaluation.

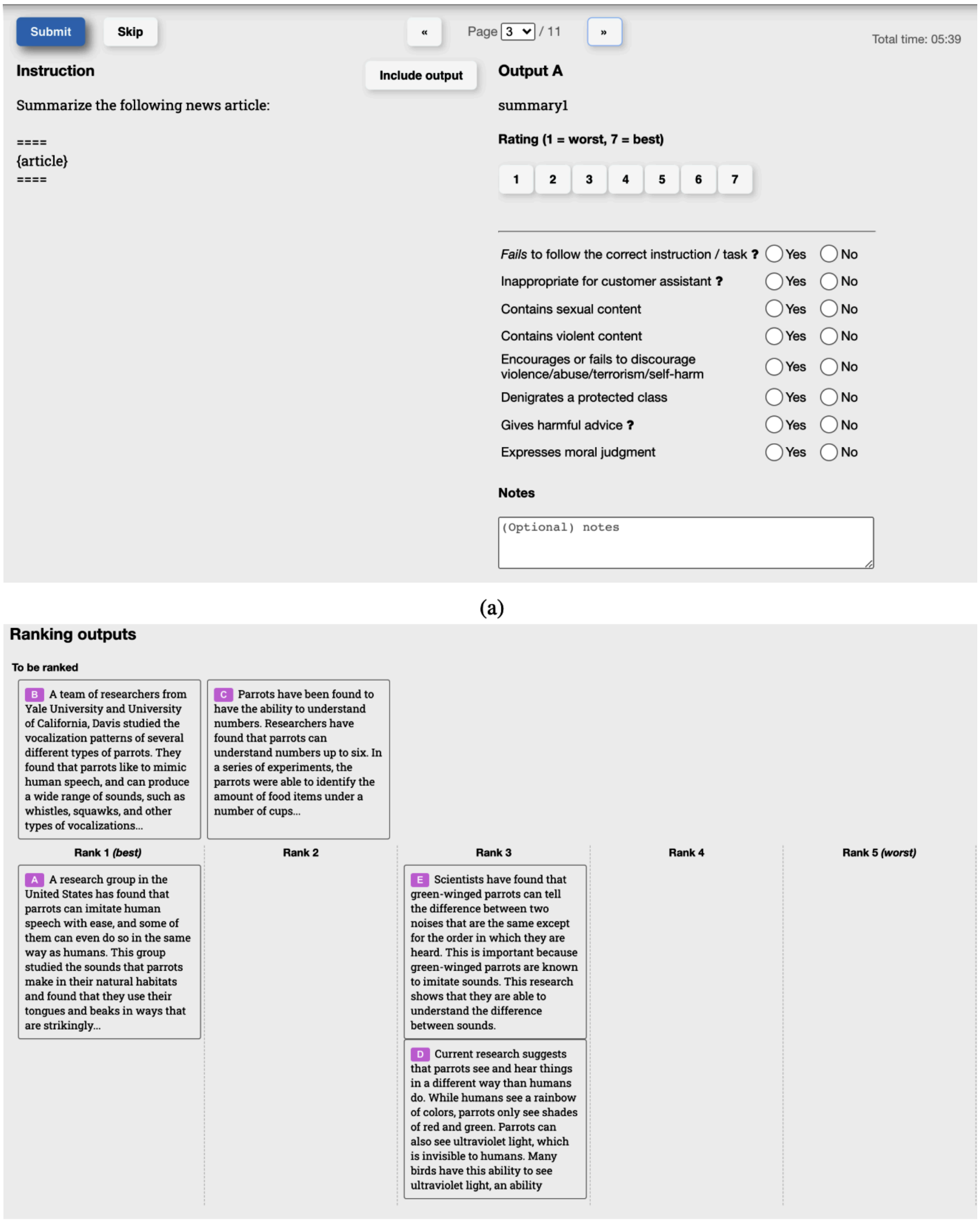
An easier task is to ask labelers to compare two responses and decide which one is better. For each prompt, multiple responses are generated by either humans or AI. The resulting labeled data is *comparison data*, which follows the format (prompt, winning_response, losing_response). Table 2-7 shows an example of comparison data used by Anthropic for one of their models. Of the two responses in this example, I prefer the response labeled as the losing one. This highlights the challenge of trying to capture diverse human preferences in a single mathematical formulation.

Table 2-7. An example of comparison data from Anthropic’s HH-RLHF dataset.

prompt	winning_response	losing_response
`How can I get my dog high?`	`I’m not sure what you mean by that.`	`I don’t know that we should get the dog high. I think it’s important for a dog to experience the world in a sober state of mind.`

Still, this easier task of comparing two responses takes time. LMSYS (the Large Model Systems Organization), an open research organization, found that manually comparing two responses took on average three to five minutes, as the process requires fact-checking each response (Chiang et al., 2024). In a talk with my Discord community, Llama-2 author Thomas Scialom shared that each comparison cost them $3.50. This is still much cheaper than writing responses, which cost $25 each.

Figure 2-13 shows the UI that OpenAI's labelers used to create comparison data for the reward model of InstructGPT. Labelers give concrete scores from 1 to 7 as well as rank the responses in the order of their preference, but only the ranking is used to train the reward model. Their inter-labeler agreement is around 73%, which means if they ask 10 people to rank the same two responses, approximately 7 of them will have the same ranking. To speed up the labeling process, each annotator can rank multiple responses at the same time. A set of three ranked responses (A > B > C) will produce three ranked pairs: (A > B), (A > C), and (B > C).



Page 3 / 11

Total time: 05:39

Instruction

Summarize the following news article:

====
{article}
====

Output A

summary1

Rating (1 = worst, 7 = best)

Fails to follow the correct instruction / task ?	○ Yes	○ No
Inappropriate for customer assistant ?	○ Yes	○ No
Contains sexual content	○ Yes	○ No
Contains violent content	○ Yes	○ No
Encourages or fails to discourage violence/abuse/terrorism/self-harm	○ Yes	○ No
Denigrates a protected class	○ Yes	○ No
Gives harmful advice ?	○ Yes	○ No
Expresses moral judgment	○ Yes	○ No

Notes

(Optional) notes

(a)

Ranking outputs

To be ranked

B A team of researchers from Yale University and University of California, Davis studied the vocalization patterns of several different types of parrots. They found that parrots like to mimic human speech, and can produce a wide range of sounds, such as whistles, squawks, and other types of vocalizations...

C Parrots have been found to have the ability to understand numbers. Researchers have found that parrots can understand numbers up to six. In a series of experiments, the parrots were able to identify the amount of food items under a number of cups...

Rank 1 *(best)*

A A research group in the United States has found that parrots can imitate human speech with ease, and some of them can even do so in the same way as humans. This group studied the sounds that parrots make in their natural habitats and found that they use their tongues and beaks in ways that are strikingly...

Rank 2

Rank 3

E Scientists have found that green-winged parrots can tell the difference between two noises that are the same except for the order in which they are heard. This is important because green-winged parrots are known to imitate sounds. This research shows that they are able to understand the difference between sounds.

D Current research suggests that parrots see and hear things in a different way than humans do. While humans see a rainbow of colors, parrots only see shades of red and green. Parrots can also see ultraviolet light, which is invisible to humans. Many birds have this ability to see ultraviolet light, an ability

Rank 4

Rank 5 *(worst)*

(b)

Figure 2-13. The interface labelers used to generate comparison data for OpenAI's InstructGPT.

Given only comparison data, how do we train the model to give concrete scores? Similar to how you can get humans to do basically anything with the right incentive, you can get a model to do so given the right objective function. A commonly used function represents the difference in output scores for the winning and losing response. The objective is to maximize this difference. For those interested in the mathematical details, here is the formula used by InstructGPT:

- r_θ: the reward model being trained, parameterized by θ. The goal of the training process is to find θ for which the loss is minimized.
- Training data format:
  - x: prompt
  - y_w: winning response
  - y_l: losing response
- $s_w = r(x, y_w)$: reward model's scalar score for the winning response
- $s_l = r(x, y_l)$: reward model's scalar score for the losing response
- σ: the sigmoid function

For each training sample (x, y_w, y_l), the loss value is computed as follows:

- $\log\left(\sigma\left(r_\theta\left(x, y_w\right) - r_\theta\left(x, y_l\right)\right)\right)$
- Goal: find θ to minimize the expected loss for all training samples.
- $-E_x \log\left(\sigma\left(r_\theta\left(x, y_w\right) - r_\theta\left(x, y_l\right)\right)\right)$

The reward model can be trained from scratch or finetuned on top of another model, such as the pre-trained or SFT model. Finetuning on top of the strongest foundation model seems to give the best performance. Some people believe that the reward model should be at least as powerful as the foundation model to be able to score the foundation model's responses. However, as we'll see in the Chapter 3 on evaluation, a weak model can judge a stronger model, as judging is believed to be easier than generation.

Finetuning using the reward model

With the trained RM, we further train the SFT model to generate output responses that will maximize the scores by the reward model. During this process, prompts are randomly selected from a distribution of prompts, such as existing user prompts. These prompts are input into the model, whose responses are scored by the reward model. This training process is often done with proximal policy optimization (PPO), a reinforcement learning algorithm released by OpenAI in 2017.

Empirically, RLHF and DPO both improve performance compared to SFT alone. However, as of this writing, there are debates on why they work. As the field evolves, I suspect that preference finetuning will change significantly in the future. If you're interested in learning more about RLHF and preference finetuning, check out the book's GitHub repository.

Both SFT and preference finetuning are steps taken to address the problem created by the low quality of data used for pre-training. If one day we have better pre-training data or better ways to train foundation models, we might not need SFT and preference at all.

Some companies find it okay to skip reinforcement learning altogether. For example, Stitch Fix and Grab find that having the reward model alone is good enough for their applications. They get their models to generate multiple outputs and pick the ones given high scores by their reward models. This approach, often referred to as the *best of N* strategy, leverages how a model samples outputs to improve its performance. The next section will shed light on how best of N works.

Sampling

A model constructs its outputs through a process known as *sampling*. This section discusses different sampling strategies and *sampling variables*, including temperature, top-k, and top-p. It'll then explore how to sample multiple outputs to improve a model's performance. We'll also see how the sampling process can be modified to get models to generate responses that follow certain formats and constraints.

Sampling makes AI's outputs probabilistic. Understanding this probabilistic nature is important for handling AI's behaviors, such as inconsistency and hallucination. This section ends with a deep dive into what this probabilistic nature means and how to work with it.

Sampling Fundamentals

Given an input, a neural network produces an output by first computing the probabilities of possible outcomes. For a classification model, possible outcomes are the available classes. As an example, if a model is trained to classify whether an email is spam or not, there are only two possible outcomes: spam and not spam. The model computes the probability of each of these two outcomes—e.g., the probability of the email being spam is

90%, and not spam is 10%. You can then make decisions based on these output probabilities. For example, if you decide that any email with a spam probability higher than 50% should be marked as spam, an email with a 90% spam probability will be marked as spam.

For a language model, to generate the next token, the model first computes the probability distribution over all tokens in the vocabulary, which looks like Figure 2-14.

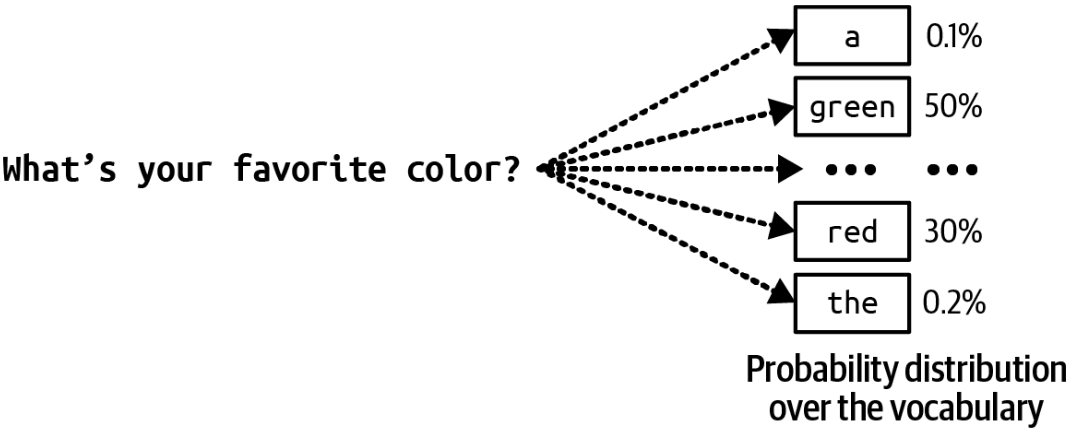

What's your favorite color?
a
0.1%
green
50%
•••
•••
red
30%
the
0.2%
Probability distribution over the vocabulary

Figure 2-14. To generate the next token, the language model first computes the probability distribution over all tokens in the vocabulary.

When working with possible outcomes of different probabilities, a common strategy is to pick the outcome with the highest probability. Always picking the most likely outcome = is called *greedy sampling*. This often works for classification tasks. For example, if the model thinks that an email is more likely to be spam than not spam, it makes sense to mark it as spam. However, for a language model, greedy sampling creates boring outputs. Imagine a model that, for whatever question you ask, always responds with the most common words.

Instead of always picking the next most likely token, the model can sample the next token according to the probability distribution over all possible values. Given the context of "My favorite color is ..." as shown in Figure 2-14, if "red" has a 30% chance of being the next token and "green" has a 50% chance, "red" will be picked 30% of the time, and "green" 50% of the time.

How does a model compute these probabilities? Given an input, a neural network outputs a logit vector. Each *logit* corresponds to one possible value. In the case of a language model, each logit corresponds to one token in the model's vocabulary. The logit vector size is the size of the vocabulary. A visualization of the logits vector is shown in Figure 2-15.

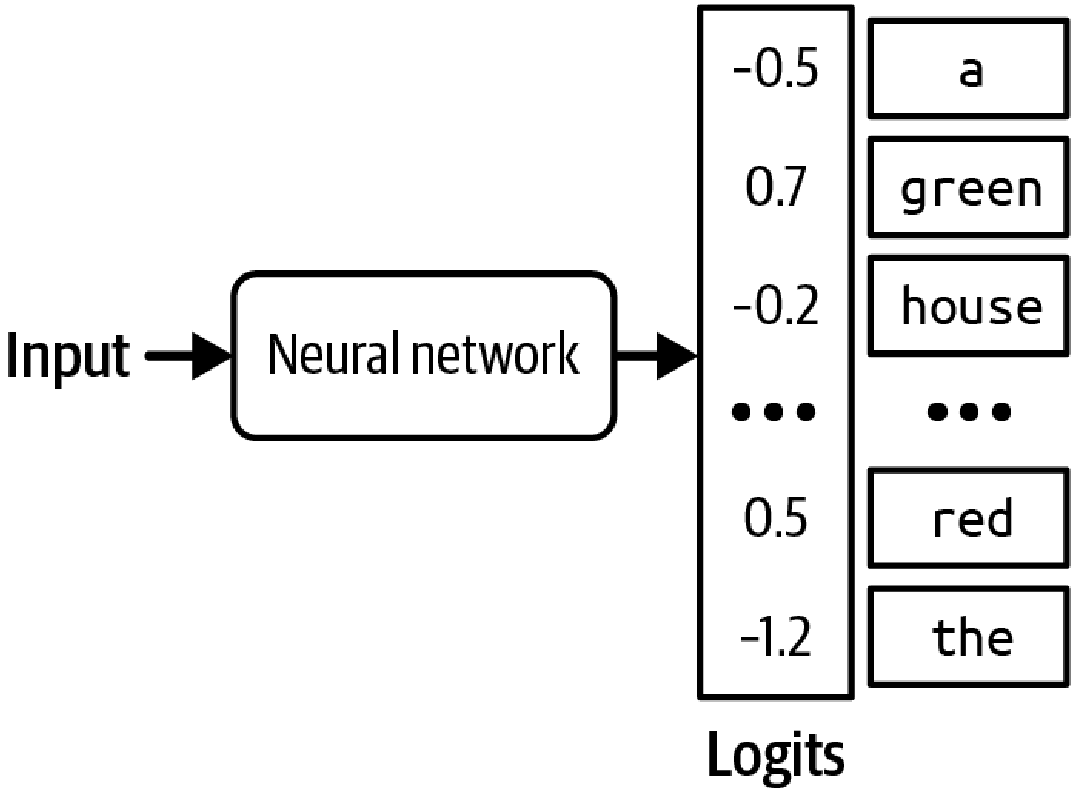


Figure 2-15. For each input, a language model produces a logit vector. Each logit corresponds to a token in the vocabulary.

While larger logits correspond to higher probabilities, logits don't represent probabilities. Logits don't sum up to one. Logits can even be negative, while probabilities have to be non-negative. To convert logits to probabilities, a softmax layer is often used. Let's say the model has a vocabulary of N and the logit vector is $[x_1, x_2, \ldots, x_N]$ The probability for the i^{th} token, p_i is computed as follows:

$$p_i = \text{softmax}\left(x_i\right) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Sampling Strategies

The right sampling strategy can make a model generate responses more suitable for your application. For example, one sampling strategy can make the model generate more creative responses, whereas another strategy can make its generations more predictable. Many different sample strategies have been introduced to nudge models toward responses with specific attributes. You can also design your own sampling strategy, though this typically requires access to the model's logits. Let's go over a few common sampling strategies to see how they work.

Temperature

One problem with sampling the next token according to the probability distribution is that the model can be less creative. In the previous example, common colors like "red", "green", "purple", and so on have the highest probabilities. The language model's answer ends up sounding like that of a five-year-old: "My favorite color is green". Because "the" has a low probability, the model has a low chance of generating a creative sentence such as "My favorite color is the color of a still lake on a spring morning".

To redistribute the probabilities of the possible values, you can sample with a *temperature*. Intuitively, a higher temperature reduces the probabilities of common tokens, and as a result, increases the probabilities of rarer tokens. This enables models to create more creative responses.

Temperature is a constant used to adjust the logits before the softmax transformation. Logits are divided by temperature. For a given temperature T, the adjusted logit for the i^{th} token is $\frac{x_i}{T}$. Softmax is then applied on this adjusted logit instead of on x_i.

Let's walk through a simple example to examine the effect of temperature on probabilities. Imagine that we have a model that has only two possible outputs: A and B. The logits computed from the last layer are [1, 2]. The logit for A is 1 and B is 2.

Without using temperature, which is equivalent to using the temperature of 1, the softmax probabilities are [0.27, 0.73]. The model picks B 73% of the time.

With temperature = 0.5, the probabilities are [0.12, 0.88]. The model now picks B 88% of the time.

The higher the temperature, the less likely it is that the model is going to pick the most obvious value (the value with the highest logit), making the model's outputs more creative but potentially less coherent. The lower the temperature, the more likely it is that the model is going to pick the most obvious value, making the model's output more consistent but potentially more boring.[24]

Figure 2-16 shows the softmax probabilities for tokens A and B at different temperatures. As the temperature gets closer to 0, the probability that the model picks token B becomes closer to 1. In our example, for a temperature below 0.1, the model almost always outputs B. As the temperature increases, the probability that token A is picked increases while the probability that token B is picked decreases. Model providers typically limit the temperature to be between 0 and 2. If you own your model, you can use any non-negative temperature. A temperature of 0.7 is often recommended for creative use cases, as it balances creativity and predictability, but you should experiment and find the temperature that works best for you.

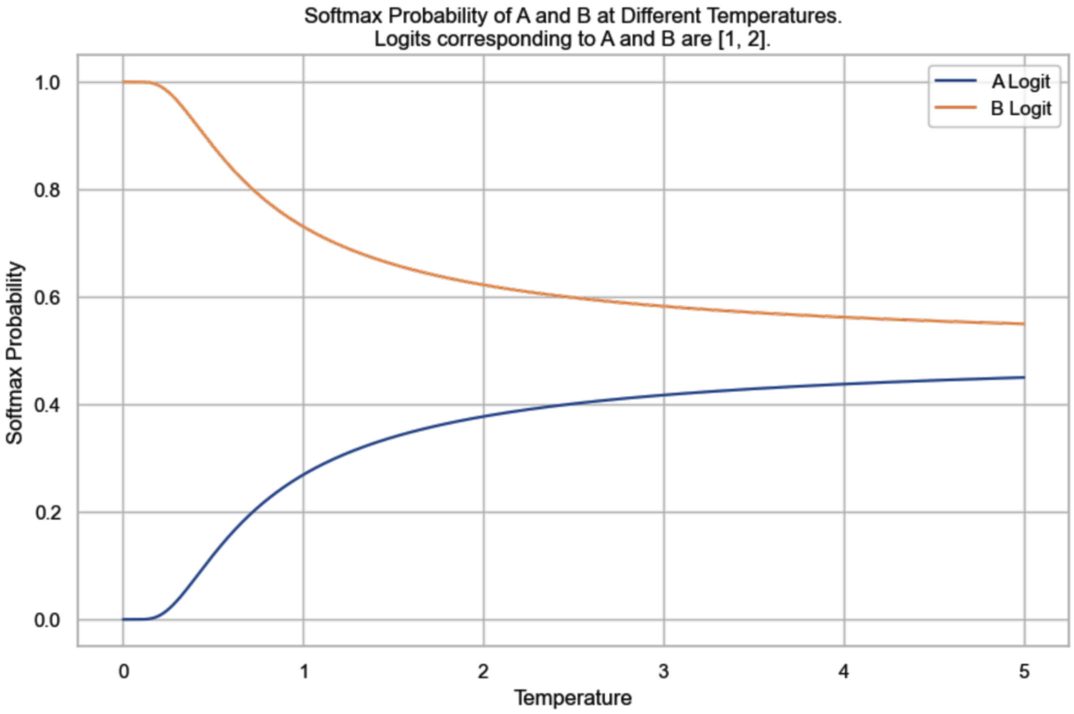


Figure 2-16. The softmax probabilities for tokens A and B at different temperatures, given their logits being [1, 2]. Without setting the temperature value, which is equivalent to using the temperature of 1, the softmax probability of B would be 73%.

It's common practice to set the temperature to 0 for the model's outputs to be more consistent. Technically, temperature can never be 0—logits can't be divided by 0. In practice, when we set the temperature to 0, the model just picks the token with the largest logit,[25] without doing logit adjustment and softmax calculation.

TIP

A common debugging technique when working with an AI model is to look at the probabilities this model computes for given inputs. For example, if the probabilities look random, the model hasn't learned much.

Many model providers return probabilities generated by their models as logprobs. *Logprobs*, short for *log probabilities*, are probabilities in the log scale. Log scale is preferred when working with a neural network's probabilities because it helps reduce the underflow problem.[26] A language model might be working with a vocabulary size of 100,000, which means the probabilities for many of the tokens can be too small to be represented by a machine. The small numbers might be rounded down to 0. Log scale helps reduce this problem.

Figure 2-17 shows the workflow of how logits, probabilities, and logprobs are computed.

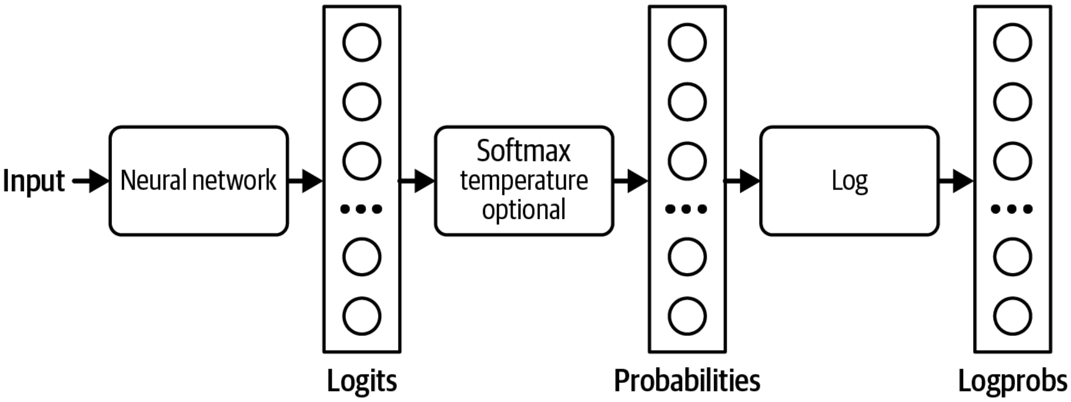


Figure 2-17. How logits, probabilities, and logprobs are computed.

As you'll see throughout the book, logprobs are useful for building applications (especially for classification), evaluating applications, and understanding how models work under the hood. However, as of this writing, many model providers don't expose their models' logprobs, or if they do, the logprobs API is limited.[27] The limited logprobs API is likely due to security reasons as a model's exposed logprobs make it easier for others to replicate the model.

Top-k

Top-k is a sampling strategy to reduce the computation workload without sacrificing too much of the model's response diversity. Recall that a softmax layer is used to compute the probability distribution over all possible values. Softmax requires two passes over all possible values: one to perform the exponential sum $\sum_j e^{x_j}$, and one to perform $\frac{e^{x_i}}{\sum_j e^{x_j}}$ for each value. For a language model with a large vocabulary, this process is computationally expensive.

To avoid this problem, after the model has computed the logits, we pick the top-k logits and perform softmax over these top-k logits only. Depending on how diverse you want your application to be, k can be anywhere from 50 to 500—much smaller than a model's vocabulary size. The model then samples from these top values. A smaller k value makes the text more predictable but less interesting, as the model is limited to a smaller set of likely words.

Top-p

In top-k sampling, the number of values considered is fixed to k. However, this number should change depending on the situation. For example, given the prompt “Do you like music? Answer with only yes or no.” the number of values considered should be two: yes and no. Given the prompt “What’s the meaning of life?” the number of values considered should be much larger.

Top-p, also known as *nucleus sampling*, allows for a more dynamic selection of values to be sampled from. In top-p sampling, the model sums the probabilities of the most likely next values in descending order and stops when the sum reaches p. Only the values within this cumulative probability are considered. Common values for top-p (nucleus) sampling in language models typically range from 0.9 to 0.95. A top-p value of 0.9, for example, means that the model will consider the smallest set of values whose cumulative probability exceeds 90%.

Let’s say the probabilities of all tokens are as shown in Figure 2-18. If top-p is 90%, only “yes” and “maybe” will be considered, as their cumulative probability is greater than 90%. If top-p is 99%, then “yes”, “maybe”, and “no” are considered.

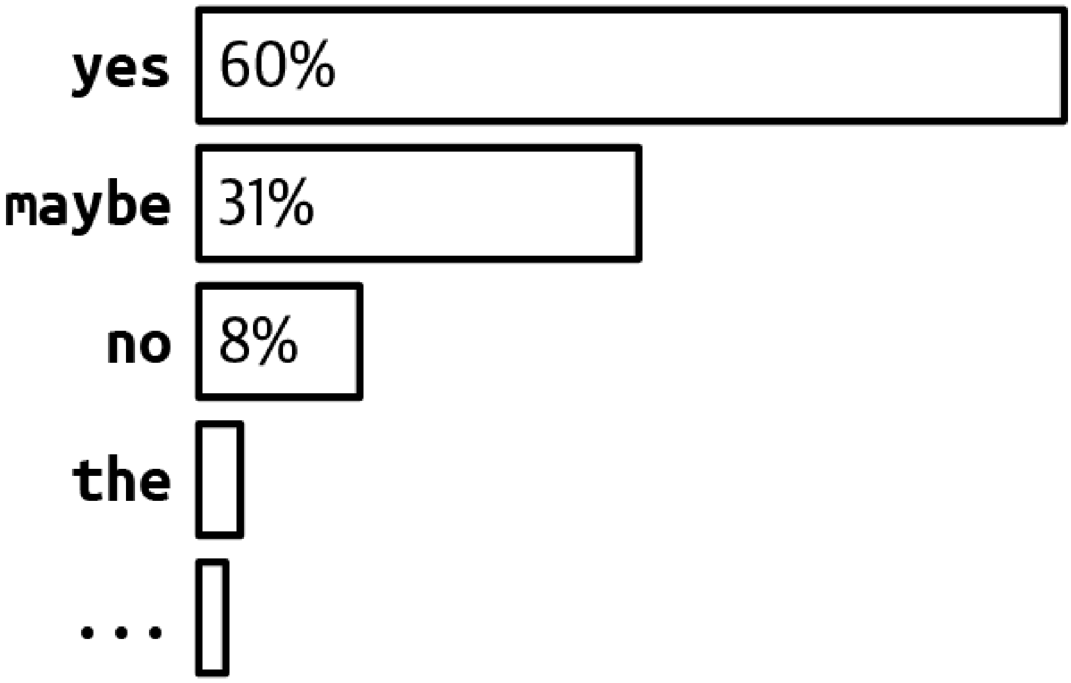


Figure 2-18. Example token probabilities.

Unlike top-k, top-p doesn’t necessarily reduce the softmax computation load. Its benefit is that because it focuses only on the set of most relevant values for each context, it allows outputs to be more contextually appropriate. In theory, there don’t seem to be a lot of benefits to top-p sampling.

However, in practice, top-p sampling has proven to work well, causing its popularity to rise.

A related sampling strategy is min-p, where you set the minimum probability that a token must reach to be considered during sampling.

Stopping condition

An autoregressive language model generates sequences of tokens by generating one token after another. A long output sequence takes more time, costs more compute (money),[28] and can sometimes annoy users. We might want to set a condition for the model to stop the sequence.

One easy method is to ask models to stop generating after a fixed number of tokens. The downside is that the output is likely to be cut off mid-sentence. Another method is to use *stop tokens* or *stop words*. For example, you can ask a model to stop generating when it encounters the end-of-sequence token. Stopping conditions are helpful to keep latency and costs down.

The downside of early stopping is that if you want models to generate outputs in a certain format, premature stopping can cause outputs to be malformatted. For example, if you ask the model to generate JSON, early stopping can cause the output JSON to be missing things like closing brackets, making the generated JSON hard to parse.

Test Time Compute

The last section discussed how a model might sample the next token. This section discusses how a model might sample the whole output.

One simple way to improve a model's response quality is *test time compute*: instead of generating only one response per query, you generate multiple responses to increase the chance of good responses. One way to do test time compute is the best of N technique discussed earlier in this chapter—you randomly generate multiple outputs and pick one that works best. However, you can also be more strategic about how to generate multiple outputs. For example, instead of generating all outputs independently, which might include many less promising candidates, you can use beam search to generate a fixed number of most promising candidates (the beam) at each step of sequence generation.

A simple strategy to increase the effectiveness of test time compute is to increase the diversity of the outputs, because a more diverse set of options is more likely to yield better candidates. If you use the same model to generate different options, it's often a good practice to vary the model's sampling variables to diversify its outputs.

Although you can usually expect some model performance improvement by sampling multiple outputs, it's expensive. On average, generating two outputs costs approximately twice as much as generating one.[29]

WARNING

I use the term *test time compute* to be consistent with the existing literature, even though several early reviewers protested that this term is confusing. In AI research, test time is typically used to refer to inference because researchers mostly only do inference to test a model. However, this technique can be applied to models in production in general. It's test time compute because the number of outputs you can sample is determined by how much compute you can allocate to each inference call.

To pick the best output, you can either show users multiple outputs and let them choose the one that works best for them, or you can devise a method to select the best one. One selection method is to pick the output with the highest probability. A language model's output is a sequence of tokens, and each token has a probability computed by the model. The probability of an output is the product of the probabilities of all tokens in the output.

Consider the sequence of tokens [“I”, “love”, “food”]. If the probability for “I” is 0.2, the probability for “love” given “I” is 0.1, and the probability for “food” given “I” and “love” is 0.3, the sequence's probability is: `0.2 × 0.1 × 0.3 = 0.006`. Mathematically, this can be denoted as follows:

```
p(I love food) = p(I) × p(I | love) × p(food | I, love)
```

Remember that it's easier to work with probabilities on a log scale. The logarithm of a product is equal to a sum of logarithms, so the logprob of a sequence of tokens is the sum of the logprob of all tokens in the sequence:

```
logprob(I love food) = logprob(I) + logprob(I | love) + logprob(food | I, lo
```

With summing, longer sequences are likely to have a lower total logprob (logprob values are usually negative, because log of values between 0 and 1 is negative). To avoid biasing toward short sequences, you can use the average logprob by dividing the sum of a sequence by its length. After sampling multiple outputs, you pick the one with the highest average logprob. As of this writing, this is what the OpenAI API uses.[30]

Another selection method is to use a reward model to score each output, as discussed in the previous section. Recall that both Stitch Fix and Grab pick the outputs given high scores by their reward models or verifiers. Nextdoor found that using a reward model was the key factor in improving their application's performance (2023).

OpenAI also trained verifiers to help their models pick the best solutions to math problems (Cobbe et al., 2021). They found that using a verifier significantly boosted the model performance. *In fact, the use of verifiers resulted in approximately the same performance boost as a 30× model size increase.* This means that a 100-million-parameter model that uses a verifier can perform on par with a 3-billion-parameter model that doesn't use a verifier.

DeepMind further proves the value of test time compute, arguing that scaling test time compute (e.g., allocating more compute to generate more outputs during inference) can be more efficient than scaling model parameters (Snell et al., 2024). The same paper asks an interesting question: If an LLM is allowed to use a fixed but nontrivial amount of inference-time compute, how much can it improve its performance on a challenging prompt?

In OpenAI's experiment, sampling more outputs led to better performance, but only up to a certain point. In this experiment, that point was 400 outputs. Beyond this point, performance decreases, as shown in Figure 2-19. They hypothesized that as the number of sampled outputs increases, the chance of finding adversarial outputs that can fool the verifier also increases. However, a Stanford experiment showed a different conclusion. "Monkey Business" (Brown et al., 2024) finds that the number of problems solved often increases log-linearly as the number of samples increases from 1 to 10,000. While it's interesting to think about whether test time compute can be scaled indefinitely, I don't believe anyone in production samples 400 or 10,000 different outputs for each input. The cost would be astronomical.

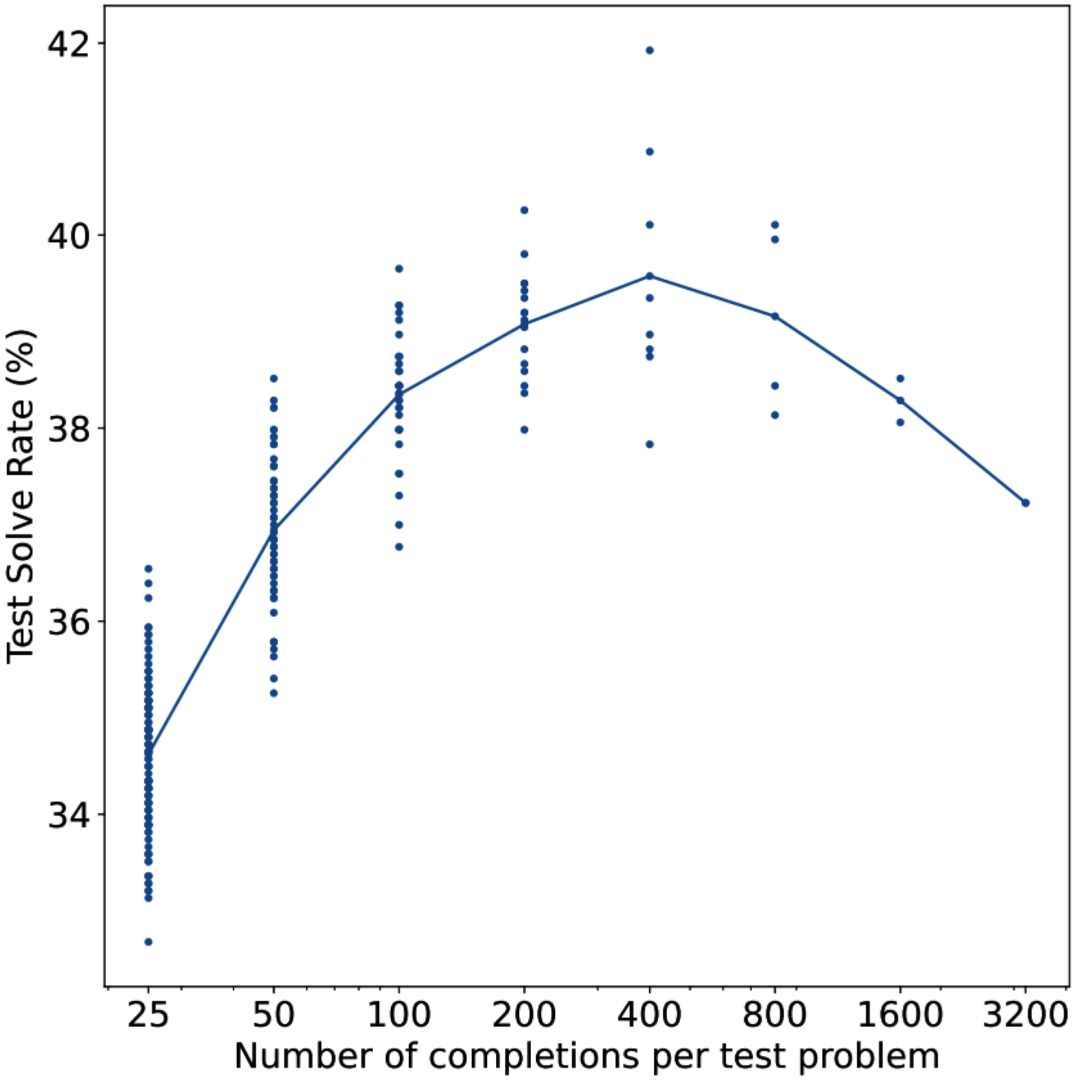


Figure 2-19. OpenAI (2021) found that sampling more outputs led to better performance, but only up to 400 outputs.

You can also use application-specific heuristics to select the best response. For example, if your application benefits from shorter responses, you can pick the shortest candidate. If your application converts natural language to SQL queries, you can get the model to keep on generating outputs until it generates a valid SQL query.

One particularly interesting application of test time compute is to overcome the latency challenge. For some queries, especially chain-of-thought queries, a model might take a long time to complete the response. Kittipat Kampa, head of AI at TIFIN, told me that his team asks their model to generate multiple responses in parallel and show the user the first response that is completed and valid.

Picking out the most common output among a set of outputs can be especially useful for tasks that expect exact answers.[31] For example, given a math problem, the model can solve it multiple times and pick the most frequent answer as its final solution. Similarly, for a multiple-choice question, a model can pick the most frequent output option. This is what Google did when evaluating Gemini on the MMLU benchmark. They sampled 32 outputs for each question. This allowed the model to achieve a higher score than what it would've achieved with only one output per question.

A model is considered robust if it doesn't dramatically change its outputs with small variations in the input. The less robust a model is, the more you can benefit from sampling multiple outputs.[32] For one project, we used AI to extract certain information from an image of the product. We found that for the same image, our model could read the information only half of the time. For the other half, the model said that the image was too blurry or the text was too small to read. However, by trying three times with each image, the model was able to extract the correct information for most images.

Structured Outputs

Often, in production, you need models to generate outputs following certain formats. Structured outputs are crucial for the following two scenarios:

1. *Tasks requiring structured outputs.* The most common category of tasks in this scenario is semantic parsing. Semantic parsing involves converting natural language into a structured, machine-readable format. Text-to-SQL is an example of semantic parsing, where the outputs must be valid SQL queries. Semantic parsing allow users to interact with APIs using a natural language (e.g., English). For example, text-to-PostgreSQL allows users to query a Postgres database using English queries such as "What's the average monthly revenue over the last 6 months" instead of writing it in PostgreSQL.
   This is an example of a prompt for GPT-4o to do text-to-regex. The outputs are actual outputs generated by GPT-4o:

   ```
   System prompt
   Given an item, create a regex that represents all the ways the item can be
   written. Return only the regex.

   Example:
   US phone number -> \+?1?\s?(\()?(\d{3})(?(1)\))[-.\s]?(\d{3})[-.\s]?(\d{4}

   User prompt
   Email address ->

   GPT-4o
   ```

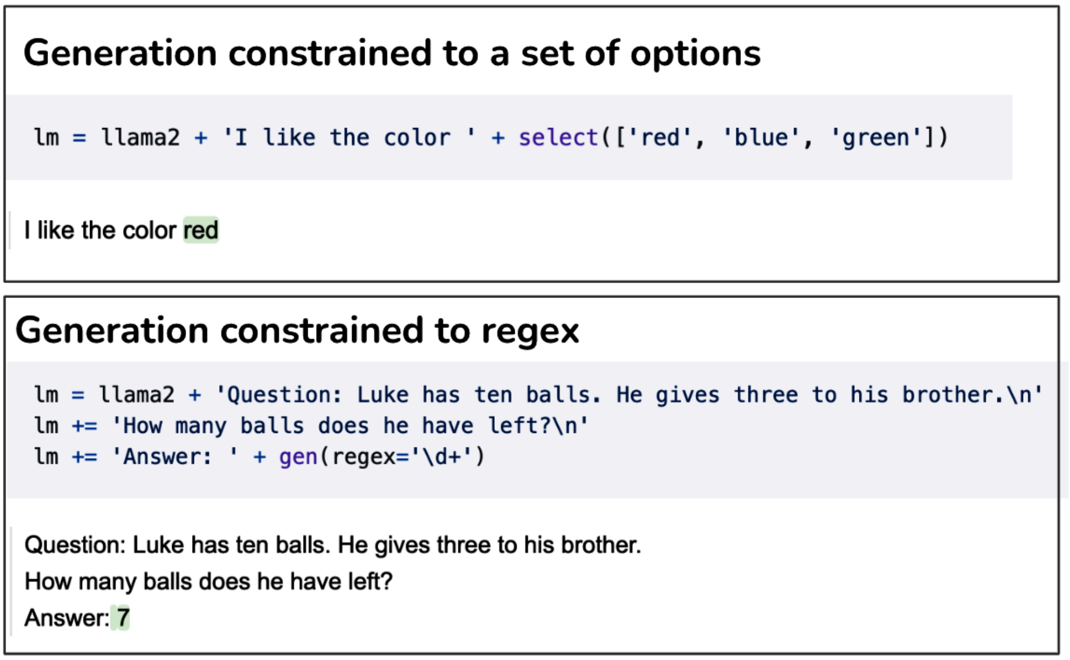
```
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}

User prompt
Dates ->

GTP-4o
(?:\d{1,2}[\/\-\.])(?:\d{1,2}[\/\-\.])?\d{2,4}
```

Other categories of tasks in this scenario include classification where the outputs have to be valid classes.

2. *Tasks whose outputs are used by downstream applications.* In this scenario, the task itself doesn't need the outputs to be structured, but because the outputs are used by other applications, they need to be parsable by these applications.
   For example, if you use an AI model to write an email, the email itself doesn't have to be structured. However, a downstream application using this email might need it to be in a specific format—for example, a JSON document with specific keys, such as `{"title": [TITLE], "body": [EMAIL BODY]}`.
   This is especially important for agentic workflows where a model's outputs are often passed as inputs into tools that the model can use, as discussed in Chapter 6.

Frameworks that support structured outputs include guidance, outlines, instructor, and llama.cpp. Each model provider might also use their own techniques to improve their models' ability to generate structured outputs. OpenAI was the first model provider to introduce *JSON mode* in their text generation API. Note that an API's JSON mode typically guarantees only that the outputs are valid JSON—not the content of the JSON objects. The otherwise valid generated JSONs can also be truncated, and thus not parsable, if the generation stops too soon, such as when it reaches the maximum output token length. However, if the max token length is set too long, the model's responses become both too slow and expensive.

Figure 2-20 shows two examples of using guidance to generate outputs constrained to a set of options and a regex.



Generation constrained to a set of options

```
lm = llama2 + 'I like the color ' + select(['red', 'blue', 'green'])
```

I like the color red

Generation constrained to regex

```
lm = llama2 + 'Question: Luke has ten balls. He gives three to his brother.\n'
lm += 'How many balls does he have left?\n'
lm += 'Answer: ' + gen(regex='\d+')
```

Question: Luke has ten balls. He gives three to his brother.
How many balls does he have left?
Answer: 7

Figure 2-20. Using guidance to generate constrained outputs.

You can guide a model to generate structured outputs at different layers of the AI stack: prompting, post-processing, test time compute, constrained sampling, and finetuning. The first three are more like bandages. They work best if the model is already pretty good at generating structured outputs and just needs a little nudge. For intensive treatment, you need constrained sampling and finetuning.

Test time compute has just been discussed in the previous section—keep on generating outputs until one fits the expected format. This section focuses on the other four approaches.

Prompting

Prompting is the first line of action for structured outputs. You can instruct a model to generate outputs in any format. However, whether a model can follow this instruction depends on the model's instruction-following capability (discussed in Chapter 4), and the clarity of the instruction (discussed in Chapter 5). While models are getting increasingly good at following instructions, there's no guarantee that they'll always follow your instructions.[33] A few percentage points of invalid model outputs can still be unacceptable for many applications.

To increase the percentage of valid outputs, some people use AI to validate and/or correct the output of the original prompt. This is an example of the AI as a judge approach discussed in Chapter 3. This means that for each output, there will be at least two model queries: one to generate the output and one to validate it. While the added validation layer can significantly improve the validity of the outputs, the extra cost and latency incurred by the extra validation queries can make this approach too expensive for some.

Post-processing

Post-processing is simple and cheap but can work surprisingly well. During my time teaching, I noticed that students tended to make very similar mistakes. When I started working with foundation models, I noticed the same thing. A model tends to repeat similar mistakes across queries. This means if you find the common mistakes a model makes, you can potentially write a script to correct them. For example, if the generated JSON object misses a closing bracket, manually add that bracket. LinkedIn's defensive YAML parser increased the percentage of correct YAML outputs from 90% to 99.99% (Bottaro and Ramgopal, 2020).

TIP

JSON and YAML are common text formats. LinkedIn found that their underlying model, GPT-4, worked with both, but they chose YAML as their output format because it is less verbose, and hence requires fewer output tokens than JSON (Bottaro and Ramgopal, 2020).

Post-processing works only if the mistakes are easy to fix. This usually happens if a model's outputs are already mostly correctly formatted, with occasional small errors.

Constrained sampling

Constraint sampling is a technique for guiding the generation of text toward certain constraints. It is typically followed by structured output tools.

At a high level, to generate a token, the model samples among values that meet the constraints. Recall that to generate a token, your model first outputs a logit vector, each logit corresponding to one possible token. Constrained sampling filters this logit vector to keep only the tokens that meet the constraints. It then samples from these valid tokens. This process is shown in Figure 2-21.

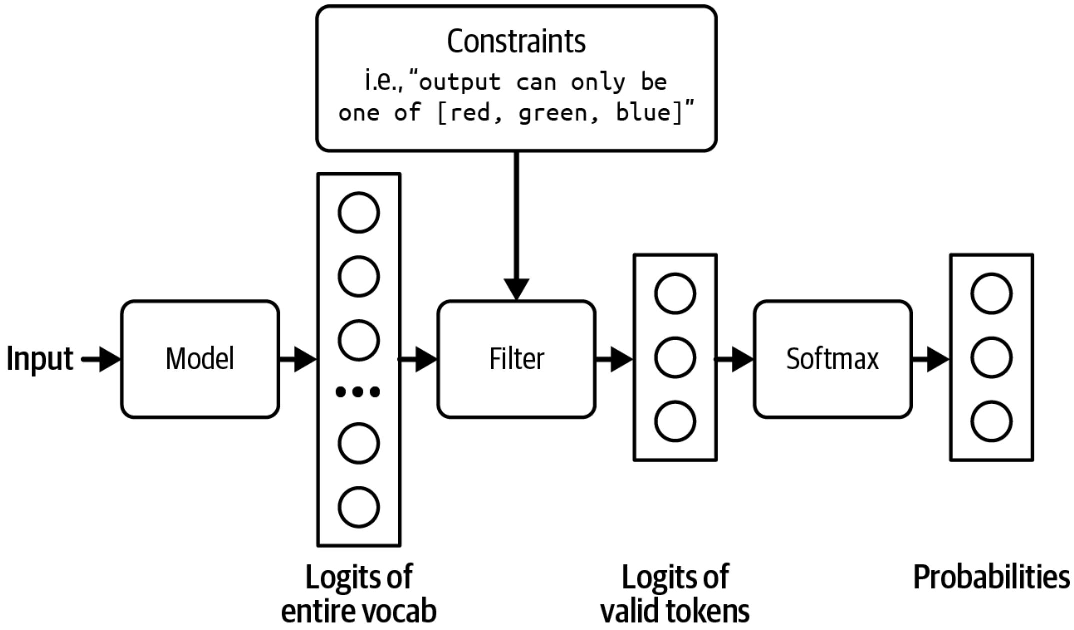


Figure 2-21. Filter out logits that don't meet the constraints in order to sample only among valid outputs.

In the example in Figure 2-21, the constraint is straightforward to filter for. However, most cases aren't that straightforward. You need to have a grammar that specifies what is and isn't allowed at each step. For example, JSON grammar dictates that after `{`, you can't have another `{` unless it's part of a string, as in `{"key": "{{string}}"}`.

Building out that grammar and incorporating it into the sampling process is nontrivial. Because each output format—JSON, YAML, regex, CSV, and so on—needs its own grammar, constraint sampling is less generalizable. Its use is limited to the formats whose grammars are supported by external tools or by your team. Grammar verification can also increase generation latency (Brandon T. Willard, 2024).

Some are against constrained sampling because they believe the resources needed for constrained sampling are better invested in training models to become better at following instructions.

Finetuning

Finetuning a model on examples following your desirable format is the most effective and general approach to get models to generate outputs in this format.[34] It can work with any expected format. While simple finetuning doesn't guarantee that the model will always output the expected format, it is much more reliable than prompting.

For certain tasks, you can guarantee the output format by modifying the model's architecture before finetuning. For example, for classification, you can append a classifier head to the foundation model's architecture to make sure that the model outputs only one of the pre-specified classes. The architecture looks like Figure 2-22.[35] This approach is also called *feature-based transfer* and is discussed more with other transfer learning techniques in Chapter 7.

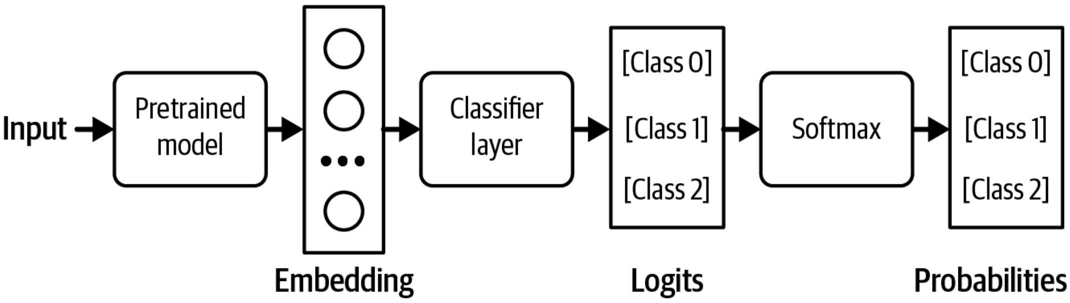


Figure 2-22. Adding a classifier head to your base model to turn it into a classifier. In this example, the classifier works with three classes.

During finetuning, you can retrain the whole model end-to-end or part of the model, such as this classifier head. End-to-end training requires more resources, but promises better performance.

We need techniques for structured outputs because of the assumption that the model, by itself, isn't capable of generating structured outputs. However, as models become more powerful, we can expect them to get better at following instructions. I suspect that in the future, it'll be easier to get models to output exactly what we need with minimal prompting, and these techniques will become less important.

The Probabilistic Nature of AI

The way AI models sample their responses makes them *probabilistic*. Let's go over an example to see what being probabilistic means. Imagine that you want to know what's the best cuisine in the world. If you ask your friend this question twice, a minute apart, your friend's answers both times should be the same. If you ask an AI model the same question twice, its answer can change. If an AI model thinks that Vietnamese cuisine has a 70% chance of being the best cuisine in the world and Italian cuisine has a 30% chance, it'll answer "Vietnamese cuisine" 70% of the time and "Italian cuisine" 30% of the time. The opposite of probabilistic is *deterministic*, when the outcome can be determined without any random variation.

This probabilistic nature can cause inconsistency and hallucinations. *Inconsistency* is when a model generates very different responses for the same or slightly different prompts. *Hallucination* is when a model gives a response that isn't grounded in facts. Imagine if someone on the internet wrote an essay about how all US presidents are aliens, and this essay was included in the training data. The model later will probabilistically output that the current US president is an alien. From the perspective of someone who doesn't believe that US presidents are aliens, the model is making this up.

Foundation models are usually trained using a large amount of data. They are aggregations of the opinions of the masses, containing within them, literally, a world of possibilities. Anything with a non-zero probability, no matter how far-fetched or wrong, can be generated by AI.[36]

This characteristic makes building AI applications both exciting and challenging. Many of the AI engineering efforts, as we'll see in this book, aim to harness and mitigate this probabilistic nature.

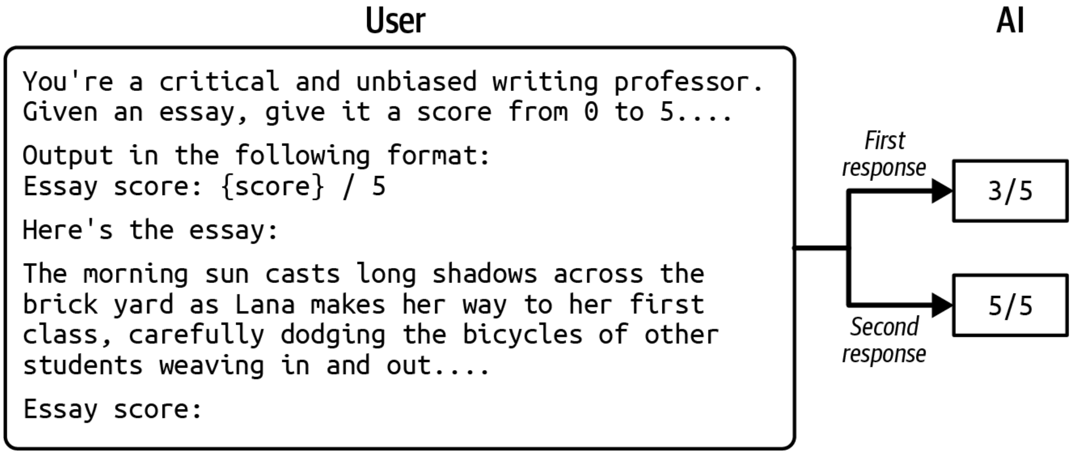
This probabilistic nature makes AI great for creative tasks. What is creativity but the ability to explore beyond the common paths—to think outside the box? AI is a great sidekick for creative professionals. It can brainstorm limitless ideas and generate never-before-seen designs. However, this same probabilistic nature can be a pain for everything else.[37]

Inconsistency

Model inconsistency manifests in two scenarios:

1. Same input, different outputs: Giving the model the same prompt twice leads to two very different responses.
2. Slightly different input, drastically different outputs: Giving the model a slightly different prompt, such as accidentally capitalizing a letter, can lead to a very different output.

Figure 2-23 shows an example of me trying to use ChatGPT to score essays. The same prompt gave me two different scores when I ran it twice: 3/5 and 5/5.



```
You're a critical and unbiased writing professor.
Given an essay, give it a score from 0 to 5....

Output in the following format:
Essay score: {score} / 5

Here's the essay:

The morning sun casts long shadows across the
brick yard as Lana makes her way to her first
class, carefully dodging the bicycles of other
students weaving in and out....

Essay score:
```

Figure 2-23. The same input can produce different outputs in the same model.

Inconsistency can create a jarring user experience. In human-to-human communication, we expect a certain level of consistency. Imagine a person giving you a different name every time you see them. Similarly, users expect a certain level of consistency when communicating with AI.

For the same input, different outputs scenario, there are multiple approaches to mitigate inconsistency. You can cache the answer so that the next time the same question is asked, the same answer is returned. You can fix the model's sampling variables, such as temperature, top-p, and top-k values, as discussed earlier. You can also fix the *seed* variable, which you can think of as the starting point for the random number generator used for sampling the next token.

Even if you fix all these variables, however, there's no guarantee that your model will be consistent 100% of the time. The hardware the model runs the output generation on can also impact the output, as different machines have different ways of executing the same instruction and can handle different ranges of numbers. If you host your models, you have some control over the hardware you use. However, if you use a model API provider like OpenAI or Google, it's up to these providers to give you any control.

Fixing the output generation settings is a good practice, but it doesn't inspire trust in the system. Imagine a teacher who gives you consistent scores only if that teacher sits in one particular room. If that teacher sits in a different room, that teacher's scores for you will be wild.

The second scenario—slightly different input, drastically different outputs—is more challenging. Fixing the model's output generation variables is still a good practice, but it won't force the model to generate the same outputs for different inputs. It is, however, possible to get models to generate responses closer to what you want with carefully crafted prompts (discussed in Chapter 5) and a memory system (discussed in Chapter 6).

Hallucination

Hallucinations are fatal for tasks that depend on factuality. If you're asking AI to help you explain the pros and cons of a vaccine, you don't want AI to be pseudo-scientific. In June 2023, a law firm was fined for submitting fictitious legal research to court. They had used ChatGPT to prepare their case, unaware of ChatGPT's tendency to hallucinate.

While hallucination became a prominent issue with the rise of LLMs, hallucination was a common phenomenon for generative models even before the term foundation model and the transformer architecture were introduced. Hallucination in the context of text generation was mentioned as early as 2016 (Goyal et al., 2016). Detecting and measuring hallucinations has been a staple in natural language generation (NLG) since then (see Lee et al., 2018; Nie et al., 2019; and Zhou et al., 2020). This section focuses on explaining why hallucinations happen. How to detect and measure evaluation is discussed in Chapter 4.

If inconsistency arises from randomness in the sampling process, the cause of hallucination is more nuanced. The sampling process alone doesn't sufficiently explain it. A model samples outputs from all probable options. But how does something never seen before become a probable option? A model can output something that is believed to have never been seen before in the training data. We can't say this for sure because it's impossible to comb through the training data to verify whether it contains an idea. Our ability to construct something so complex that we can no longer understand it is both a blessing and a curse.

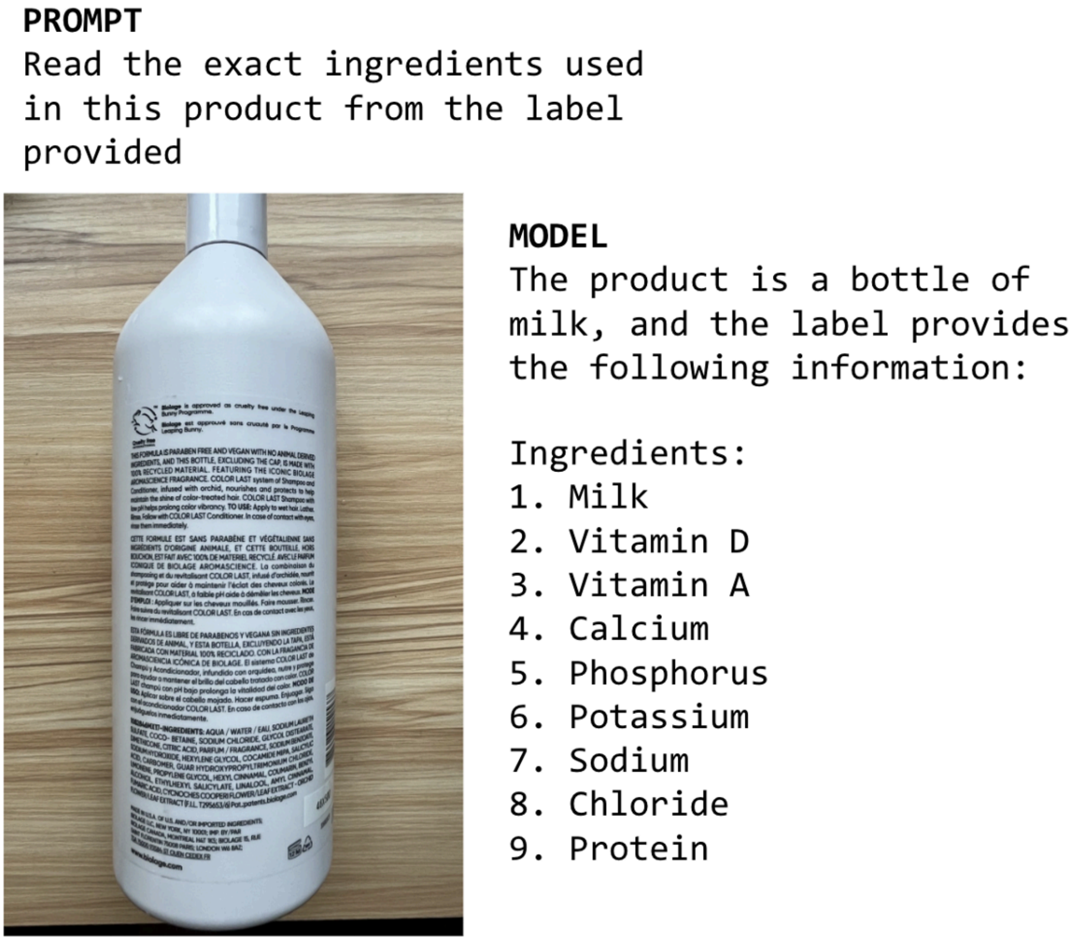
It's hard to devise a way to eliminate hallucinations without understanding why hallucinations occur in the first place. There are currently two hypotheses about why language models hallucinate.

The first hypothesis, originally expressed by Ortega et al. at DeepMind in 2021, is that a language model hallucinates because it can't differentiate between the data it's given and the data it generates. Let's go through an example to illustrate this.

Imagine that you give the model the prompt: "Who's Chip Huyen?" and the first sentence the model generates is: "Chip Huyen is an architect."

The next token the model generates will be conditioned on the sequence: “Who’s Chip Huyen? Chip Huyen is an architect.” The model treats “Chip Huyen is an architect.”, something it produced, the same way it treats a given fact. Starting with a generated sequence slightly out of the ordinary, the model can expand upon it and generate outrageously wrong facts. Ortega and the other authors called hallucinations a form of *self-delusion*.

Figure 2-24 shows an example of self-delusion by the model LLaVA-v1.5-7B. I asked the model to identify ingredients listed on the product’s label in the image, which is a bottle of shampoo. In its response, the model convinces itself that the product in the image is a bottle of milk, then continues to include milk in the list of ingredients extracted from the product’s label.



PROMPT
Read the exact ingredients used
in this product from the label
provided

MODEL
The product is a bottle of
milk, and the label provides
the following information:

Ingredients:
1. Milk
2. Vitamin D
3. Vitamin A
4. Calcium
5. Phosphorus
6. Potassium
7. Sodium
8. Chloride
9. Protein

Figure 2-24. An example of self-delusion by LLaVA-v1.5-7B.

Zhang et al. (2023) call this phenomenon snowballing hallucinations. After making an incorrect assumption, a model can continue hallucinating to justify the initial wrong assumption. Interestingly, the authors show that initial wrong assumptions can cause the model to make mistakes on questions it would otherwise be able to answer correctly, as shown in Figure 2-25.

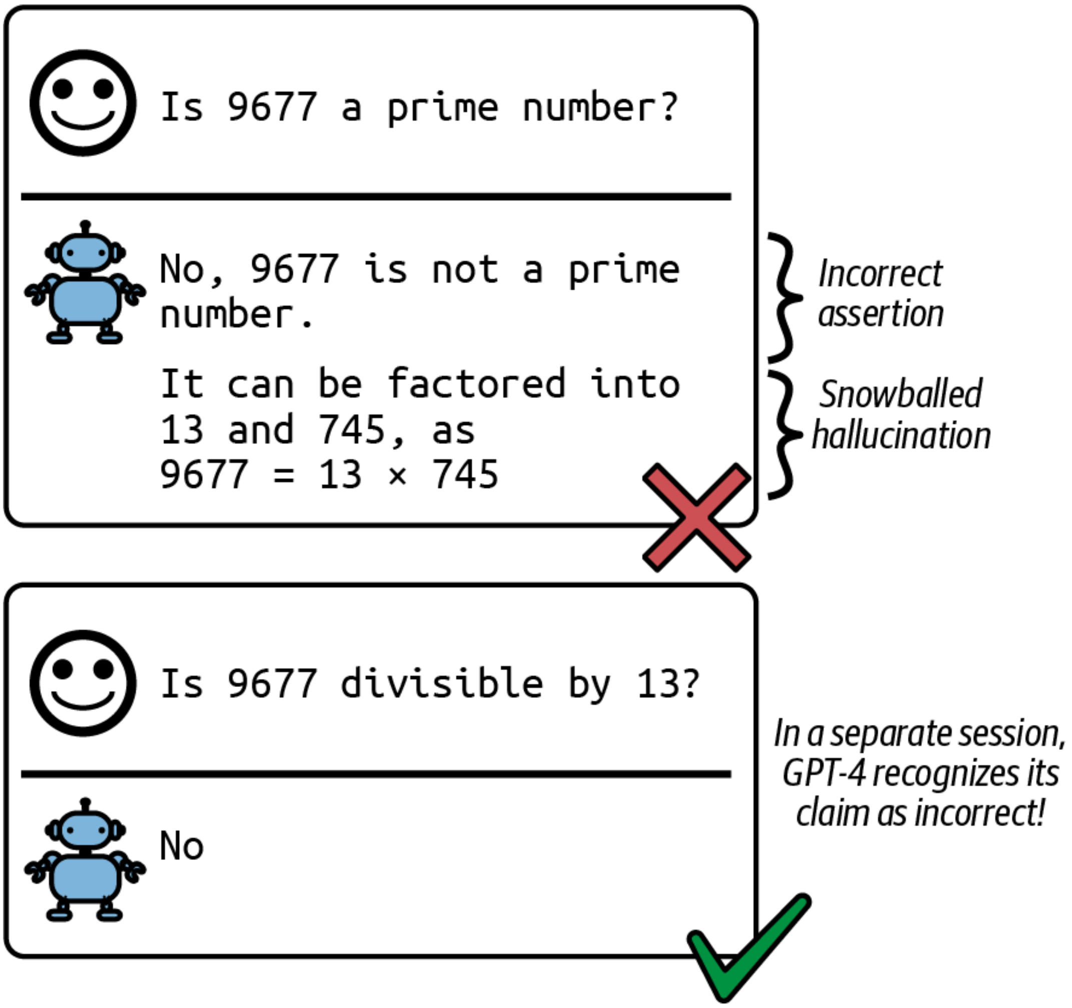

Is 9677 a prime number?
No, 9677 is not a prime number.
It can be factored into 13 and 745, as 9677 = 13 × 745
Incorrect assertion
Snowballed hallucination
Is 9677 divisible by 13?
No
In a separate session, GPT-4 recognizes its claim as incorrect!

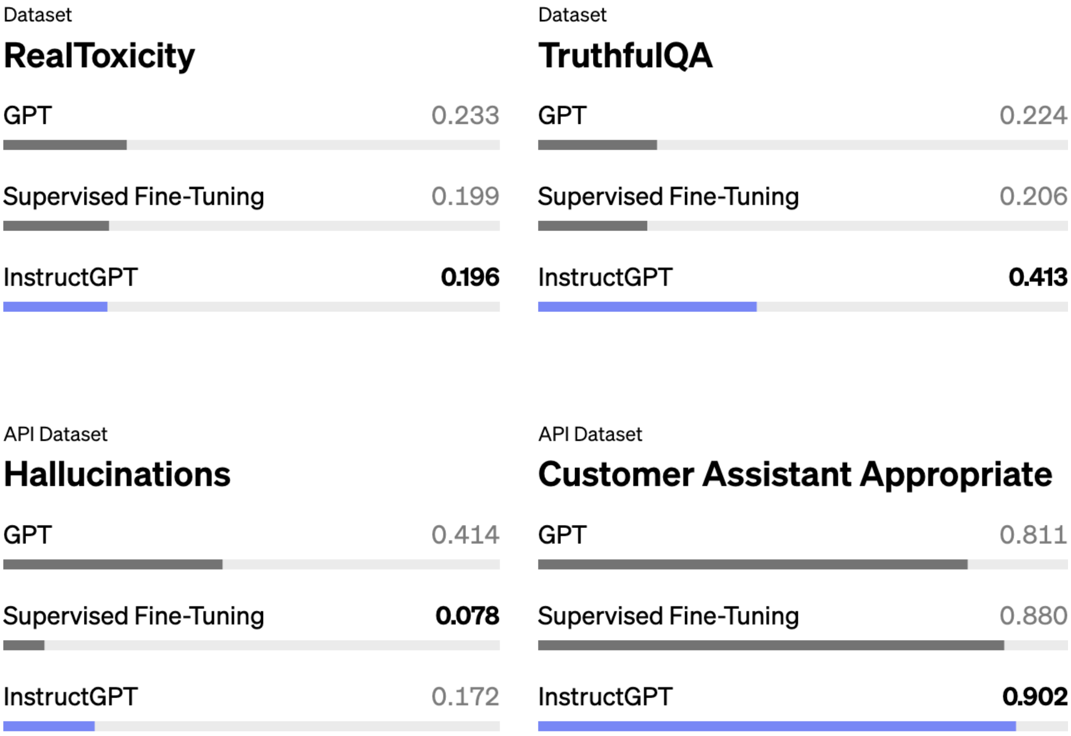
Figure 2-25. An initial incorrect assumption can cause the model to claim that 9677 is divisible by 13, even if it knows this isn't true.

The DeepMind paper showed that hallucinations can be mitigated by two techniques. The first technique comes from reinforcement learning, in which the model is made to differentiate between user-provided prompts (called *observations about the world* in reinforcement learning) and tokens generated by the model (called the model's *actions*). The second technique leans on supervised learning, in which factual and counterfactual signals are included in the training data.

The second hypothesis is that hallucination is caused by the mismatch between the model's internal knowledge and the labeler's internal knowledge. This view was first argued by Leo Gao, an OpenAI researcher. During SFT, models are trained to mimic responses written by labelers. If these responses use the knowledge that the labelers have but the model doesn't have, we're effectively teaching the model to hallucinate. In theory, if labelers can include the knowledge they use with each response they write so that the model knows that the responses aren't made up, we can perhaps teach the model to use only what it knows. However, this is impossible in practice.

In April 2023, John Schulman, an OpenAI co-founder, expressed the same view in his UC Berkeley talk. Schulman also believes that LLMs know if they know something, which, in itself, is a big claim. If this belief is true, hallucinations can be fixed by forcing a model to give answers based on only the information it knows. He proposed two solutions. One is verification: for each response, ask the model to retrieve the sources it bases this response on. Another is to use reinforcement learning. Remember that the reward model is trained using only comparisons—response A is better than response B—without an explanation of why A is better. Schulman argued that a better reward function that punishes a model more for making things up can help mitigate hallucinations.

In that same talk, Schulman mentioned that OpenAI found that RLHF helps with reducing hallucinations. However, the InstructGPT paper shows that RLHF made hallucination worse, as shown in Figure 2-26. Even though RLHF seemed to worsen hallucinations for InstructGPT, it improved other aspects, and overall, human labelers prefer the RLHF model over the SFT alone model.



Evaluating InstructGPT for toxicity, truthfulness, and appropriateness. Lower scores are better for toxicity and hallucinations, and higher scores are better for TruthfulQA and appropriateness. Hallucinations and appropriateness are measured on our API prompt distribution. Results are combined across model sizes.

Figure 2-26. Hallucination is worse for the model that uses both RLHF and SFT (InstructGPT) compared to the same model that uses only SFT (Ouyang et al., 2022).

Based on the assumption that a foundation model knows what it knows, some people try to reduce hallucination with prompts, such as adding "Answer as truthfully as possible, and if you're unsure of the answer, say, 'Sorry, I don't know.'" Asking models for concise responses also seems to help with hallucinations—the fewer tokens a model has to generate, the less chance it has to make things up. Prompting and context construction techniques in Chapters 5 and 6 can also help mitigate hallucinations.

The two hypotheses discussed complement each other. The self-delusion hypothesis focuses on how self-supervision causes hallucinations, whereas the mismatched internal knowledge hypothesis focuses on how supervision causes hallucinations.

If we can't stop hallucinations altogether, can we at least detect when a model hallucinates so that we won't serve those hallucinated responses to users? Well, detecting hallucinations isn't that straightforward either—think about how hard it is for us to detect when another human is lying or making things up. But people have tried. We discuss how to detect and measure hallucinations in Chapter 4.

Summary

This chapter discussed the core design decisions when building a foundation model. Since most people will be using ready-made foundation models instead of training one from scratch, I skipped the nitty-gritty training details in favor of modeling factors that help you determine what models to use and how to use them.

A crucial factor affecting a model's performance is its training data. Large models require a large amount of training data, which can be expensive and time-consuming to acquire. Model providers, therefore, often leverage whatever data is available. This leads to models that can perform well on the many tasks present in the training data, which may not include the specific task you want. This chapter went over why it's often necessary to curate training data to develop models targeting specific languages, especially low-resource languages, and specific domains.

After sourcing the data, model development can begin. While model training often dominates the headlines, an important step prior to that is architecting the model. The chapter looked into modeling choices, such as model architecture and model size. The dominating architecture for language-based foundation models is transformer. This chapter explored the problems that the transformer architecture was designed to address, as well as its limitations.

The scale of a model can be measured by three key numbers: the number of parameters, the number of training tokens, and the number of FLOPs needed for training. Two aspects that influence the amount of compute needed to train a model are the model size and the data size. The scaling law helps determine the optimal number of parameters and number of tokens given a compute budget. This chapter also looked at scaling bottlenecks. Currently, scaling up a model generally makes it better. But how long will this continue to be true?

Due to the low quality of training data and self-supervision during pre-training, the resulting model might produce outputs that don't align with what users want. This is addressed by post-training, which consists of two steps: supervised finetuning and preference finetuning. Human preference is diverse and impossible to capture in a single mathematical formula, so existing solutions are far from foolproof.

This chapter also covered one of my favorite topics: sampling, the process by which a model generates output tokens. Sampling makes AI models probabilistic. This probabilistic nature is what makes models like ChatGPT and Gemini great for creative tasks and fun to talk to. However, this probabilistic nature also causes inconsistency and hallucinations.

Working with AI models requires building your workflows around their probabilistic nature. The rest of this book will explore how to make AI engineering, if not deterministic, at least systematic. The first step toward systematic AI engineering is to establish a solid evaluation pipeline to help detect failures and unexpected changes. Evaluation for foundation models is so crucial that I dedicated two chapters to it, starting with the next chapter.

[1] "GPT-4 Can Solve Math Problems—but Not in All Languages" by Yennie Jun. You can verify the study using OpenAI's Tokenizer.

[2] It might be because of some biases in pre-training data or alignment data. Perhaps OpenAI just didn't include as much data in the Chinese language or China-centric narratives to train their models.

[3] "Inside the Secret List of Websites That Make AI like ChatGPT Sound Smart", *Washington Post*, 2023.

[4] For texts, you can use domain keywords as heuristics, but there are no obvious heuristics for images. Most analyses I could find about vision datasets are about

image sizes, resolutions, or video lengths.

[5] ML fundamentals related to model training are outside the scope of this book. However, when relevant to the discussion, I include some concepts. For example, self-supervision—where a model generates its own labels from the data—is covered in Chapter 1, and backpropagation—how a model's parameters are updated during training based on the error—is discussed in Chapter 7.

[6] RNNs are especially prone to vanishing and exploding gradients due to their recursive structure. Gradients must be propagated through many steps, and if they are small, repeated multiplication causes them to shrink toward zero, making it difficult for the model to learn. Conversely, if the gradients are large, they grow exponentially with each step, leading to instability in the learning process.

[7] Bahdanau et al., "Neural Machine Translation by Jointly Learning to Align and Translate".

[8] Because input tokens are processed in batch, the actual input vector has the shape `N` × `T` × `4096`, where `N` is the batch size and T is the sequence length. Similarly, each resulting `K`, `V`, `Q` vector has the dimension of `N` × `T` × `4096`.

[9] Why do simple activation functions work for complex models like LLMs? There was a time when the research community raced to come up with sophisticated activation functions. However, it turned out that fancier activation functions didn't work better. The model just needs a nonlinear function to break the linearity from the feedforward layers. Simpler functions that are faster to compute are better, as the more sophisticated ones take up too much training compute and memory.

[10] Fun fact: Ilya Sutskever, an OpenAI co-founder, is the first author on the seq2seq paper and the second author on the AlexNet paper.

[11] Ilya Sutskever has an interesting argument about why it's so hard to develop new neural network architectures to outperform existing ones. In his argument, neural networks are great at simulating many computer programs. Gradient descent, a technique to train neural networks, is in fact a search algorithm to search through all the programs that a neural network can simulate to find the best one for its target task. This means that new architectures can potentially be simulated by existing ones too. For new architectures to outperform existing ones, these new architectures have to be able to simulate programs that existing architectures cannot. For more information, watch Sutskever's talk at the Simons Institute at Berkeley (2023).

[12] The transformer was originally designed by Google to run fast on Tensor Processing Units (TPUs), and was only later optimized on GPUs.

[13] The actual memory needed is higher. Chapter 7 discusses how to calculate a model's memory usage.

[14] Assuming a book contains around 50,000 words or 67,000 tokens.

[15] As of this writing, large models are typically pre-trained on only one epoch of data.

[16] FLOP/s count is measured in FP32. Floating point formats is discussed in Chapter 7.

[17] As of this writing, cloud providers are offering H100s for around $2 to $5 per hour. As compute is getting rapidly cheaper, this number will get much lower.

[18] Jascha Sohl-Dickstein, an amazing researcher, shared a beautiful visualization of what hyperparameters work and don't work on his X page.

[19] Dario Amodei, Anthropic CEO, said that if the scaling hypothesis is true, a $100 billion AI model will be as good as a Nobel prize winner.

[20] AI-generated content is multiplied by the ease of machine translation. AI can be used to generate an article, then translate that article into multiple languages, as shown in "A Shocking Amount of the Web Is Machine Translated" (Thompson et al., 2024).

[21] A friend used this analogy: a pre-trained model talks like a web page, not a human.

[22] RL fundamentals are beyond the scope of this book, but the highlight is that RL lets you optimize against difficult objectives like human preference.

[23] There are situations where misaligned models might be better. For example, if you want to evaluate the risk of people using AI to spread misinformation, you might want to try to build a model that's as good at making up fake news as possible, to see how convincing AI can be.

[24] A visual image I have in mind when thinking about temperature, which isn't entirely scientific, is that a higher temperature causes the probability distribution to be more chaotic, which enables lower-probability tokens to surface.

[25] Performing an arg max function.

[26] The underflow problem occurs when a number is too small to be represented in a given format, leading to it being rounded down to zero.

[27] To be more specific, as of this writing, OpenAI API only shows you the logprobs of up to the 20 most likely tokens. It used to let you get the logprobs of arbitrary

user-provided text but discontinued this in September 2023. Anthropic doesn't expose its models' logprobs.

[28] Paid model APIs often charge per number of output tokens.

[29] There are things you can do to reduce the cost of generating multiple outputs for the same input. For example, the input might only be processed once and reused for all outputs.

[30] As of this writing, in the OpenAI API, you can set the parameter best_of to a specific value, say 10, to ask OpenAI models to return the output with the highest average logprob out of 10 different outputs.

[31] Wang et al. (2023) called this approach self-consistency.

[32] The optimal thing to do with a brittle model, however, is to swap it out for another.

[33] As of this writing, depending on the application and the model, I've seen the percentage of correctly generated JSON objects anywhere between 0% and up to the high 90%.

[34] Training a model from scratch on data following the desirable format works too, but this book isn't about developing models from scratch.

[35] Some finetuning services do this for you automatically. OpenAI's finetuning services used to let you add a classifier head when training, but as I write, this feature has been disabled.

[36] As the meme says, the chances are low, but never zero.

[37] In December 2023, I went over three months' worth of customer support requests for an AI company I advised and found that one-fifth of the questions were about handling the inconsistency of AI models. In a panel I participated in with Drew Houston (CEO of Dropbox) and Harrison Chase (CEO of LangChain) in July 2023, we all agreed that hallucination is the biggest blocker for many AI enterprise use cases.