AI Law: Lecture 1 Notes

Peter Henderson

Fall 2025

Perhaps surprisingly to some, the outcome of much litigation turns on the fundamental technical details of foundation models. What are they trained on? Why are they trained on those things? Do the memorize or actually learn general patterns? To understand the technical jargon and analyses, we will open with a very rough overview of foundation models, enought to fit in one lecture.

1 A Thousand Foot View of LLMs

The fundamental goal of most LLM model creators, especially as we integrate additional modalities beyond text-based language into them, is to have general purpose systems that can quickly learn to do any task. The Stanford cohort of folks, like myself, sometimes call these models *foundation models*, instead of LLMs, to capture their multipurpose basis. These models take a massive amount of data, "train" on them to learn underlying statistical patterns, then are adapted to a bunch of different downstream tasks that meet the users' needs.

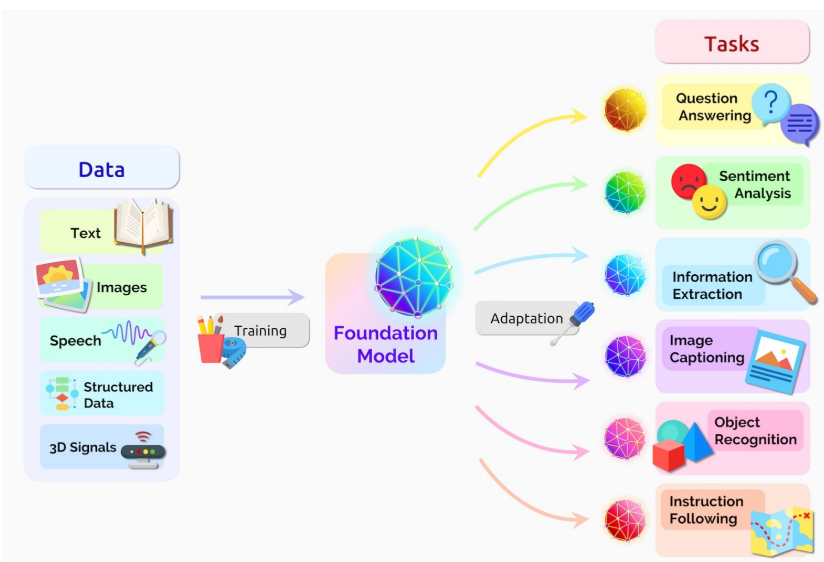


Figure 1: Foundation Models, taken from Bommasani et al. (2021).

Currently, we mainly focus on a constrained goal for LLMs, basic chat assistants. So if you're a lawyer, you might want to engage in a conversation like the following:

User: Write me a legal memo about the current state of fair use doc-

trine after Warhol? Cite the most relevant cases.
AI Assistant: Sure, I can help with that.
To: [Recipient Name]
From: [Your Name]
Date: June 1, 2024
Subject: Current State of Fair Use Doctrine Post-Warhol Decision
Introduction
This memorandum provides an analysis of the fair use doctrine following the Supreme Court's decision in Andy Warhol Foundation for the Visual Arts, Inc. v. Goldsmith, 598 U.S. ___ (2023). This landmark case has significant implications for the application of fair use in the context of transformative works. . . .

To get there, the LLM approach is to focus on modeling the distributions of next **tokens**.[1] If you get really really good at modeling what words could come next given a series of previous words, we (perhaps surprisingly) find that this is enough to learn to solve many many different types of tasks, as well as learning higher-level reasoning mechanisms. To see what I mean by modeling next token distributions, consider that you're given a sentence, "In *Brown v. Board of Education*, the Supreme Court ruled that ____" Now, imagine what you would fill in for the next word. It could be many different things:

[1] For now, just think of a token as a word, we'll learn more about this soon.

- In *Brown v. Board of Education*, the Supreme Court ruled that the
- In *Brown v. Board of Education*, the Supreme Court ruled that schools
- In *Brown v. Board of Education*, the Supreme Court ruled that segregation
- In *Brown v. Board of Education*, the Supreme Court ruled that charter

When we train a model on tons and tons of data, the model can also narrow down the potential words that might come next with sufficiently accurate probabilities. Then we can actually have it complete the sentence accurately (with sufficiently large models and data).

In this case, GPT-2 (which can now run on many consumer laptops), outputs the token distribution seen in Figure 1.

To complete the text, we can then **sample** from the next token distribution—we choose randomly according to the probabilities that each token is assigned. Or we can choose the most probable token—**greedy selection**. Let's say that we selected "charter" based on random sampling. Remember that random sampling means that *any* token can be chosen as long as it has non-zero probability. This will be important to remember when we discuss model errors in the future.

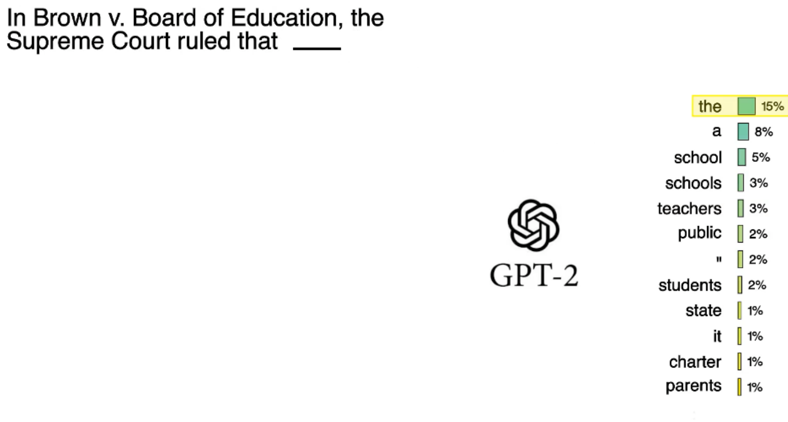


Figure 2: Next token distribution for GPT-2

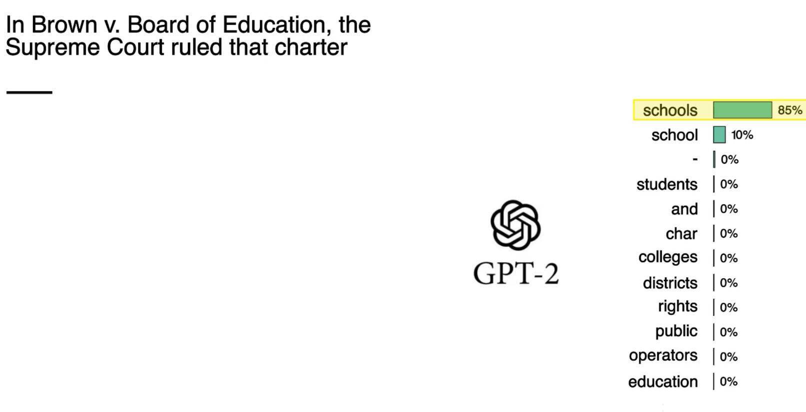


Figure 3: Update next token.

What do we do next? We feed in the original text snippet, plus the selected token back into the model. Then the model gets the next token distribution after that, again and again and again. We do this until the model suggests a special token called a "stop token" or an "end of sequence token" which says that the model should stop generating (or, alternatively, if we've run out of compute). Notice, though, unlike in the first iteration, after we feed in the selected token (in this case, the word "charter" we have only two real choices: "school" or "schools." As seen in Figure 6, almost all tokens have close to zero probability mass.[2] The model has learned sufficiently from its training data that "*Brown v. Board of Education* probably wasn't a ruling about "charter... planes?"

[2] In jargon, a probability mass is just the amount of probability on a given token.

Now, this is all more easily said than done. To understand how we get from nothing to a model that can answer user requests, there are many different components—and interesting twists and turns. I'll try to give enough of an overview so that you can understand not just how they work, but begin to have a better intuition of how *and why* things can go wrong. In particular, we'll focus on these main aspects of LLMs:

1. Tokenization
2. Embeddings
3. The Transformer Architecture
4. Pretraining
5. Finetuning
6. Alignment
7. Evaluation
8. Red Teaming, Harms, and Failure Modes

2 *Tokenization*

Let's start with tokenization. As noted above, a token is (for better or for worse), not a word. Nor is it a character, a punctuation, or anything else. It is a statistically-derived unit of text (or other modality like audio or images). To build some intuition, we can go to the Tiktokenizer website, which is a convenient tool for building intuition about tokenizers of different models. If we take our previous text, we'll get something like this:

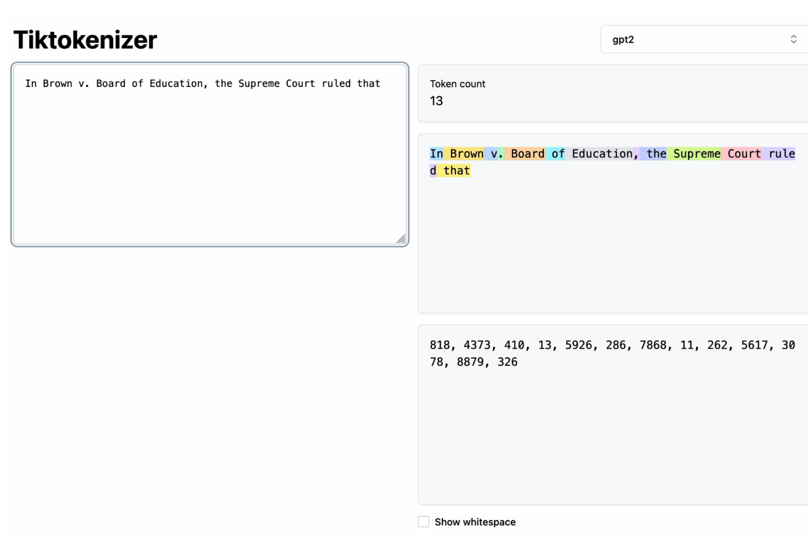


Figure 4: TikTokenizer screenshot.

So we end up with 13 tokens. Mostly, we lucked out and we get whole words (with a few smaller character and punctuation tokens mixed in).

$$\underbrace{[818]}_{\text{In}}\ \underbrace{[4373]}_{\text{Brown}}\ \underbrace{[410]}_{\text{v}}\ \underbrace{[13]}_{.}\ \underbrace{[5926]}_{\text{Board}}\ \underbrace{[286]}_{\text{of}}\ \underbrace{[7868]}_{\text{Education}}\ \underbrace{[11]}_{,}\ \underbrace{[262]}_{\text{the}}\ \underbrace{[5617]}_{\text{Supreme}}\ \underbrace{[3078]}_{\text{Court}}\ \underbrace{[8879]}_{\text{ruled}}\ \underbrace{[326]}_{\text{that}}$$

What do these numbers mean? We have a “dictionary" where we can use the token “ID” (index) to look up a mathematical representation of that token—something that we’ll get to in the Embeddings Section (§ 3). For now, just think of it as the page number for where you can look up more information about a token. Similarly, like the index of a book, you can take a token’s text and find its token ID.

Sometimes we won’t be so lucky. *Gideon v. Wainwright*, 372 U.S. 335 (1963)—while having far fewer words, gets mapped to about the same amount of tokens as *Brown v. Board of Education of Topeka*, 347 U.S. 483 (1954).

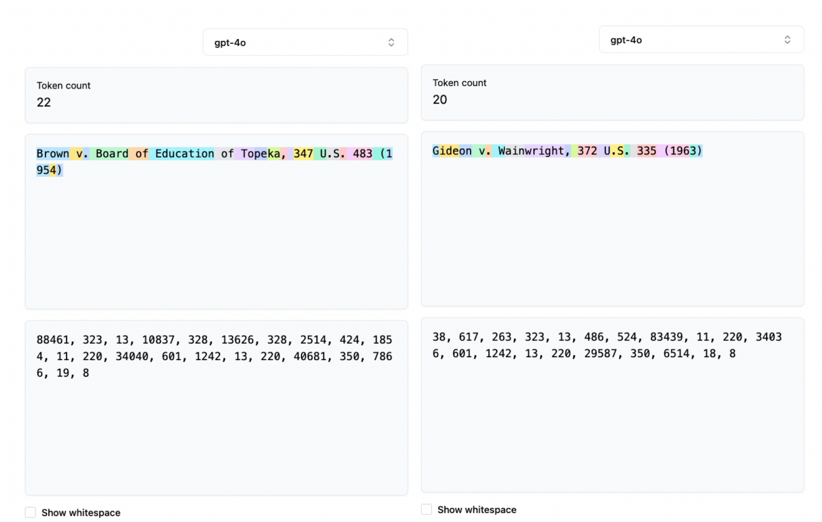


Figure 5: TikTonizer screenshot.

$$\underbrace{[38]}_{\text{G}}\ \underbrace{[617]}_{\text{ide}}\ \underbrace{[263]}_{\text{on}}\ \underbrace{[323]}_{\text{v}}\ \underbrace{[13]}_{.}\ \underbrace{[486]}_{\text{W}}\ \underbrace{[524]}_{\text{ain}}\ \underbrace{[83439]}_{\text{wright}}\ \underbrace{[11]}_{,}\ \underbrace{[220]}_{\text{372}}$$

$$\underbrace{[34036]}_{\text{U}}\ \underbrace{[601]}_{.}\ \underbrace{[1242]}_{\text{S}}\ \underbrace{[13]}_{.}\ \underbrace{[220]}_{\text{335}}\ \underbrace{[29587]}_{(}\ \underbrace{[350]}_{\text{196}}\ \underbrace{[6514]}_{\text{3}}\ \underbrace{[18]}_{)}$$

You’ll notice, nothing about the tokenization of *Gideon* seems logical. The words are broken up in odd ways. 1963 turns into 196 and 3.

Why does this happen? Because a tokenizer is also *trained*—it is learned from statistical patterns in a large dataset. Mostly, this is with some variation of the Byte-Pair Encoding (BPE) algorithm (or something like it). You don't need to know the exact details of it, but you can think about it at a high level as follows.

First, you start off with just the base-level tokens (characters or "bytes" of information). So if you were training on the United States constitution, you'd first tokenize something like this:

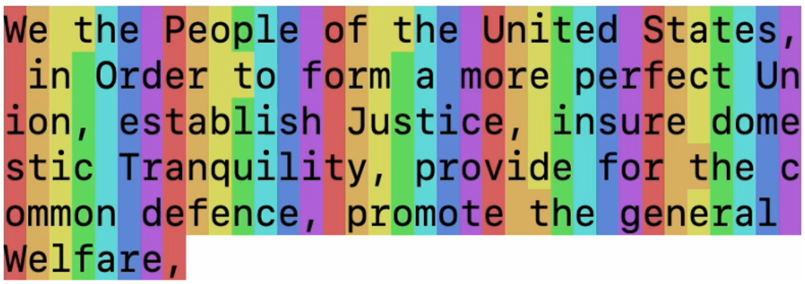


Figure 6: A character-level tokenization of the first part of the constitution. The first step of BPE.

Then, we find all the pairs of tokens that we have so far that are most statistically likely to go together. In this case, we see an awful lot of t's going together with h's.

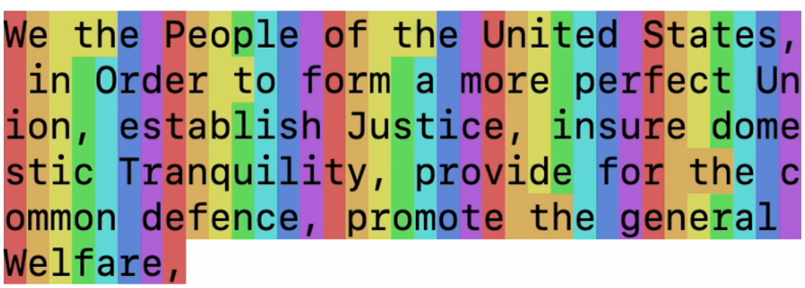


Figure 7: The t's and h's get clustered together next.

After that the e's mostly go with th's so we get our first full word token.

And this continues on and on until you fill up whatever **vocabulary size** that you've allocated. Your vocabulary size is the amount of tokens you're willing to have. GPT3 and 4 have roughly ∼100k tokens, GPT-4(o) has around 200k.

Well, you might ask, why can't I just encode everything as characters/bytes and call it a day? Well researchers have dreamed of this for years (Xue et al., 2022), but it's too expensive. Remember that LLMs work by predicting next tokens. The cost scales linearly with

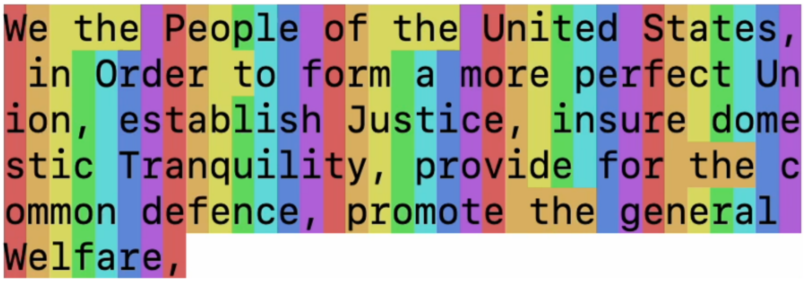


Figure 8: Our first full word token.

the amount of tokens you have to generate to respond to a user. So, the larger your vocab size, the fewer tokens you'll have to generate. This goes for multiple languages as well. A selling point of OpenAI's GPT-4(o) release was that it required fewer tokens for other languages. For most models, there's a non-English "tax." It costs more to use other languages (per token) because they simply require more tokens. Our sentence, "In Brown v. Board of Education, the Supreme Court ruled that" is 13 tokens in English ($0.00013 for gpt-4-1106-preview in June 2024). Translated into Chinese, it costs almost dobule at 24 tokens ($0.00024). And translated into Ukrainian, we get more than 3x the cost at 42 tokens ($0.00042). Languages with fewer speakers tend to be less represented in the data and end up costing more due to this phenomenon.

This results in other effects too. Recently, researchers have found what are referred to as "glitch tokens." They are tokens that are counter-intuitive as to why they might be so statistically likely as to get their own token.

Consider these three real tokens in the GPT-2 tokenizer: "TheNitromeFan" (token 42090), "RandomRedditorWithNo" (tokeni 36174), and SolidGoldMagikarp (43453). Why are these even in the tokenizer? It turns out that there is a SubReddit called "r/counting", where users collaboratively (or competitively?) count to infinite. Every day, they respond to one another, incrementing the previous post's number by one.

These three tokens ("TheNitromeFan", "RandomRedditorWithNo", and "SolidGoldMagikarp") are actually just usernames from this subreddit. They posted, according to a recent counting (pun intended), 84581, 63434, and 65753 times to the counting forum, respectively. When OpenAI then scraped the web for data, these tens of thousands of posts made their way into the dataset. Then when the tokenizer was trained, each of these users got their own token.

Similarly, researchers have dug into the origins of a large number of these strange tokens. externalActionCode (token 31576), for

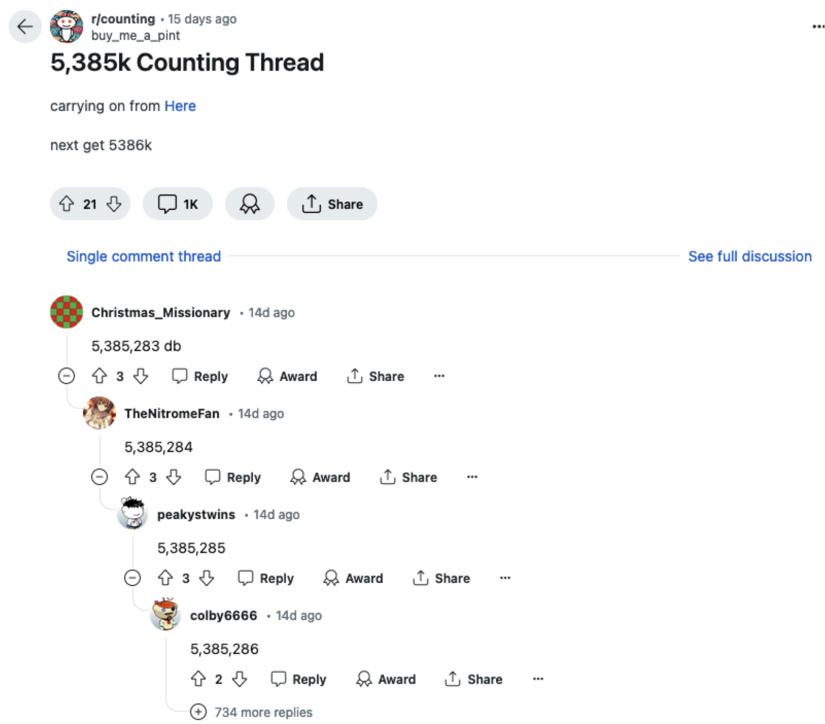
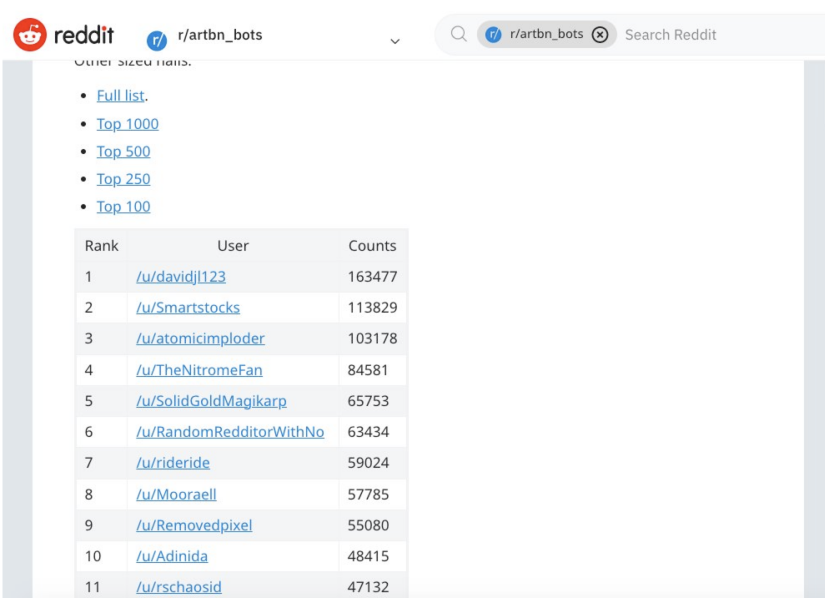


Figure 9: r/counting thread



Rank	User	Counts
1	/u/davidjl123	163477
2	/u/Smartstocks	113829
3	/u/atomicimploder	103178
4	/u/TheNitromeFan	84581
5	/u/SolidGoldMagikarp	65753
6	/u/RandomRedditorWithNo	63434
7	/u/rideride	59024
8	/u/Mooraell	57785
9	/u/Removedpixel	55080
10	/u/Adinida	48415
11	/u/rschaosid	47132

Figure 10: The /r/counting leaders.

example, appears on every congressional bill tracker that's reposted across the web. Digging into the vocabularies of models also tells you how much (or how little) data cleaning took place when the tokenizer was trained. Recently researchers were shocked to find long strings of spammy and adult content in the Chinese tokens of GPT-4(o)'s tokenizer (along with "Socialism with Chinese characteristics").

185118 _日本毛片免费视频观看	185118 _Japanese porn videos for free
116852 中国福利彩票天天	116852 China Welfare Lottery every day
128031 久久免费热在线精品	128031 Long-term free hot online boutique
154809 无码不卡高清免费v	154809 Uncoded high-definition free v
172750 大发快三大小单双	172750 Dafa Express Big and Small Single and Double
177431 给主人留下些什么吧	177431 Leave something for the owner
181679 qq的天天中彩票	181679 QQ's daily lottery
184969 _日本一级特黄大片	184969 _Japanese first-class special yellow blockbuster
187822 大发快三开奖结果	187822 Dafa Express Lottery Results
49649 彩神争霸邀请码	49649 Caishen Competition Invitation Code
89409 免费视频在线观看	89409 Free video online
122333 无码不卡高清免费	122333 Uncoded high-definition free
122712 无码一区二区三区	122712 Uncoded Zone 1, 2 and 3
128600 大发时时彩计划	128600 Dafa Time Lottery Plan
133274 】【: 】【"】【	133274 】【: 】【"】【
135161 大发时时彩开奖	135161 Dafa Time Lottery Results
149168 大发时时彩怎么	149168 How to Dafa Time Lottery
160029 大发快三是国家	160029 Dafa Express 3 is the state
160131 大发快三是不是	160131 Is Dafa Express 3 a state
160267 天天中彩票网站	160267 Tiantianzhong lottery website
176039 精品一区二区三区	176039 Boutique Zone 1, Zone 2, Zone 3
186348 大发快三是什么	186348 What is Dafa Express 3
187516 大发快三走势图	187516 Dafa Express 3 trend chart
187810 在线观看中文字幕	187810 Watch online with Chinese subtitles
191179 大发快三怎么看	191179 How to watch Dafa Express 3
193825 中国特色社会主义	193825 Socialism with Chinese characteristics
194062 彩神争霸是不是	194062 Is Caishen Zhengba a state

Figure 11: Chinese spammy tokens in GPT-4(o)

These glitch tokens can result in undefined behaviors (something that we'll explain more in the embeddings section). One key way things can go wrong is if you train the tokenizer on different data than you train the model itself. Essentially, the model never learns what the token really means (or at least what's statistically most likely to come after that token). So researchers have found that models "glitch" out when asked to repeat these tokens, either replacing the word(s) that should have been there with something random, or just failing to generate anything. This leads to a potential security problem since it gives an attack vector for people to trick a model into entering unusual behavior modes.

Finally, I won't get into it too much, but LLMs are being trained on all kinds of data using the same model all at once, not just text. But how would you even tokenize an image? Or audio? Well, you can take the image, split it into "patches" of pixels and then feed each of those patches into the model as a "token". Similarly, for audio, you can turn the audio into a spectrogram and then turn that into patches as well.

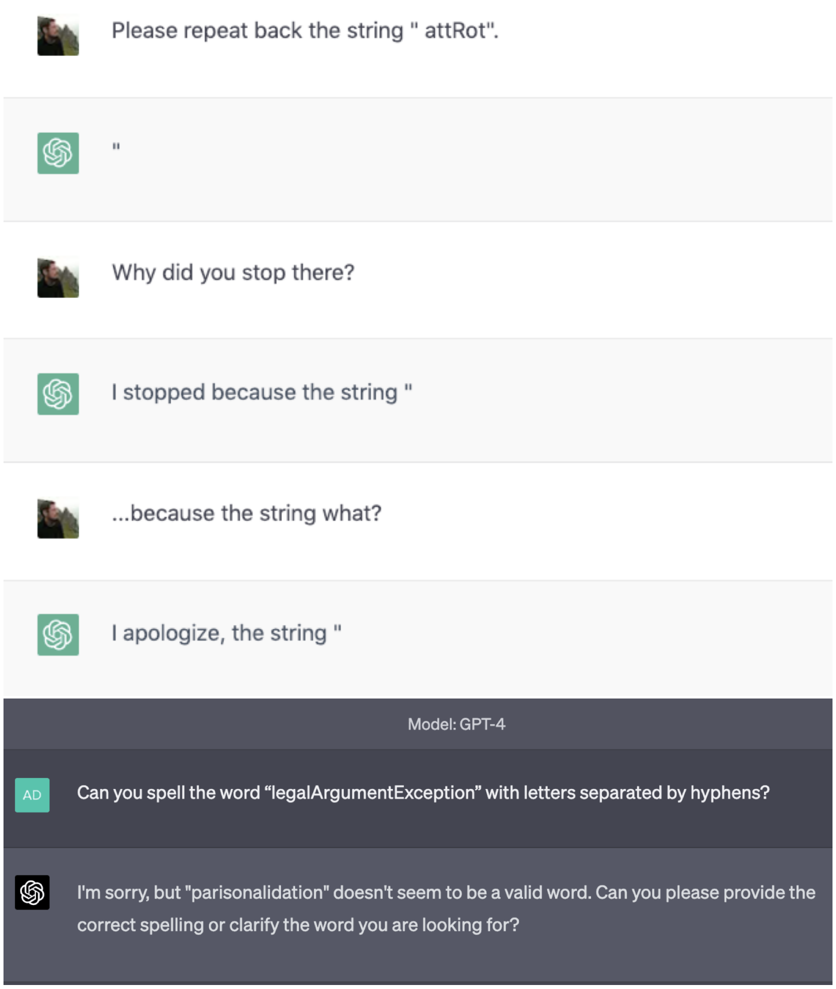


Figure 12: Glitch token effects

Figure 13: Image patches.

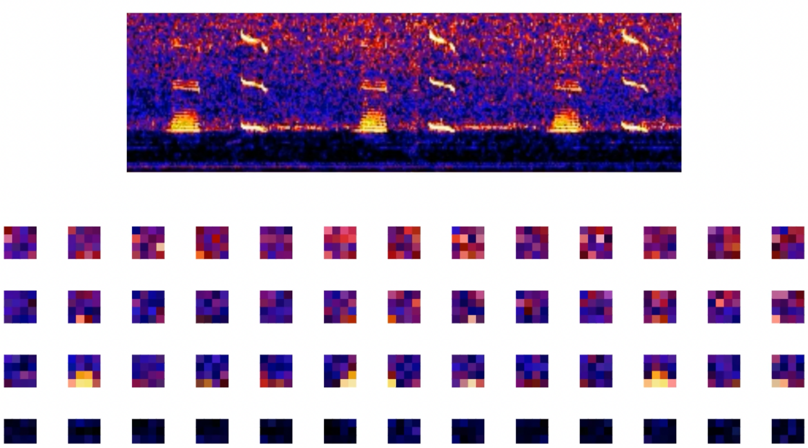


Figure 14: Audio Patches

3 *Embeddings*

Once we have a tokenizer and can split the text/images/audio into many tokens, we now need to convert these into mathematical representations that can be processed by a neural network. We call these "embeddings." Embeddings are a vector of numbers. So when we take the token ID, we can index into a matrix (like visualized below) and find the column of numbers that represent the token.

Figure 15: An Embedding Matrix. Columns are the token index, rows are a single dimension in a many-dimensional embedding vector.

So if we start looking up the embeddings for the first part of the Constitution, we'll get:

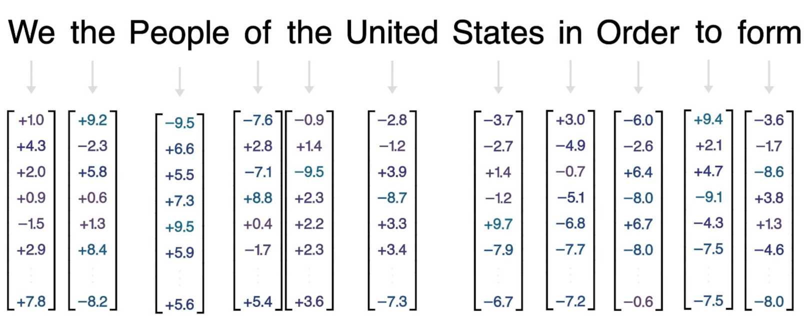


Figure 16: A sentence with its representative embeddings.

Now, before the embeddings are "trained", they're just random numbers—meaningless—but over the lifetime of training, they will start to have meaningful representations that map to semantic space. So you can actually turn the analogy "man is to woman as king is to queen" into math in vector space:

$$\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen} \tag{1}$$

The idea is that the relationship between "man" and "woman" is similar to the relationship between "king" and "queen". By adding the vector that represents the transition from "man" to "woman" to the vector for "king," you should end up close to the vector for "queen" in the semantic space.

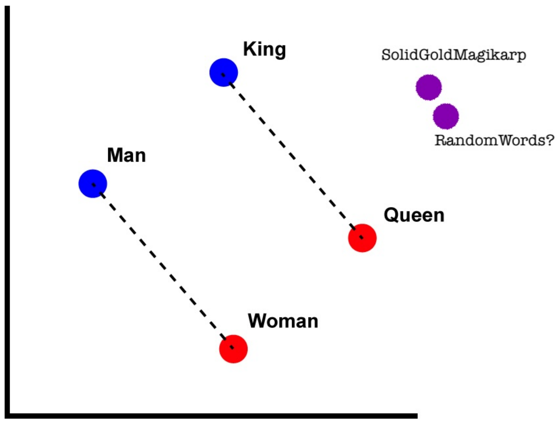


Figure 17: Embeddings famously have important relationships in vector space that relate to semantic meaning. While trained tokens might have meaningful mappings from embedding space to semantic space, "glitch tokens" probably do not. Modified from: this image.

However, as I mentioned, these vectors must be trained! If you never see any training data for the token, the embedding stays close to its random representation. This is why glitch tokens might lead to strange behaviors, these embeddings aren't trained, so the model is getting a semi-randomized input that it doesn't know how to map to any actual semantic meaning.

For audio and video, recall that tokens are just patches of images or audio representations. Remember that a pixel in an image is associate with a set of numbers indicating what color the pixel should be. As a result, you can just flatten these values out into a vector representation and use the pixel values directly. Some models will do something a bit more complicated, using an intermediate model to get a better representation of the image patches. But we don't need to get into that here.

Once you can convert all your different types of tokens into embeddings, you can begin to weave them together into one unified multimodal model. Below are real patches from a blue whale call spectrogram and the associated text. Think about how the web is? If I just scraped the web, I would often get sequences of images that are associated with semantic meaning. So in a fairly unsupervised way I would be able to start associated words with images—thanks to how we've naturally built webpages.

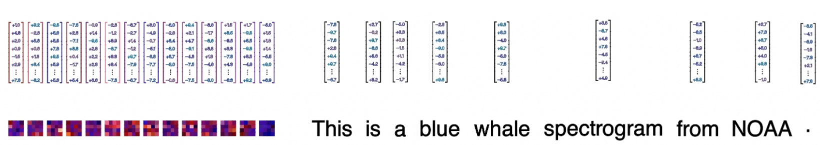


Figure 18: Weaving together different modalities in embedding space.

Positional Encodings. There's one last step though. If you're a model and you just get this sequence of vectors, there's actually no information that tells you how each vector corresponds to a position in a sequence. The vectors themselves are all a model sees—it doesn't have a natural notion of position in the sequence. It might be able to learn a natural encoding of positionality over time (in fact, recent work suggests this might be a better approach), but this can be expensive and unpredictable. Instead, model creators use a shortcut by providing some information in the embedding about where in the sequence of text a token is. They do so via positional encoding.

Apologies for the math, but one way to encode the position is via a sinusoidal function. The positional encoding for position *pos* and dimension i is defined as:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

where d_{model} is the dimensionality of the embeddings. The positional encodings are then added to the input embeddings:

$$\mathbf{z} = \mathbf{e} + \mathbf{p}$$

where $\mathbf{e}$ is the input embedding and $\mathbf{p}$ is the positional encoding. This can be visualized as:

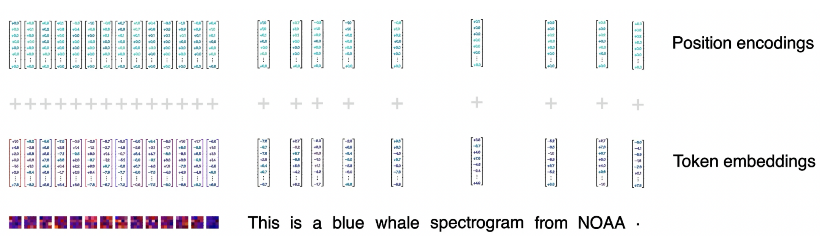


Figure 19: The positional encodings get added to the embeddings.

The positional encodings allow the model to distinguish between different positions in the sequence, even though the input embeddings themselves contain no positional information.

4 *The Transformer Architecture*

The **transformer** architecture is the backbone of modern large language models. It's a type of neural network architecture that was introduced in the 2017 paper "Attention Is All You Need" by Vaswani et al. The key component of a transformer is the **self-attention** mechanism, which allows the model to weigh the importance of different parts of the input when making predictions. Effectivelly all of this is just matrix multiplication. A bunch of multiplications and additions for billions of numbers. Before we continue, a few definitions:

DEFINITIONS. An **architecture** refers to the overall structure and components of the model. It defines what types of matrix multiplications must be done, how many there are, how they are connected, and the flow of data between matrices.
The **weights** or **parameters** of the model are all of the learned numbers that make up the matrices that an architecture requires—the numbers that the model tunes during training to make its predictions match the training data.

At its core, a transformer model is a series of matrix multiplications. Recall that when you have two matrices, a matrix multiplication just looks like the following.

RECALL MATRIX MULTIPLICATIONS.
Let's consider two matrices A and B, where A is a 2x3 matrix and B is a 3x2 matrix.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$$

The product of these matrices, denoted as $C = AB$, will be a 2x2 matrix. The elements of C are computed as follows:

$$C = AB = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

where:
$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} = (1 \times 7) + (2 \times 9) + (3 \times 11) = 58$
$c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} = (1 \times 8) + (2 \times 10) + (3 \times 12) = 64$
$c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} = (4 \times 7) + (5 \times 9) + (6 \times 11) = 139$
$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} = (4 \times 8) + (5 \times 10) + (6 \times 12) = 154$
Therefore, the result of the matrix multiplication is:

$$C = AB = \begin{pmatrix} 58 & 64 \\ 139 & 154 \end{pmatrix}$$

So, now that you remember what a matrix multiplication is. Basically what we want to do is take our series of input embeddings and convert them through a series of matrix multiplications into a probability distribution of next token predictions. In a transformer, there are several different types of operations to make this conversion possible. I won't get into all of them, but arguably the most important is the self-attention layer.

For each token, it computes a weighted sum of all tokens in the input, where the weights are determined by how relevant each token is to the current token. This allows the model to attend to important parts of the context.

The equation for the attention mechanism is just:

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

This can be a bit intimidating, but the names of the variables actually help provide some intuition. Imagine if you have a set of values (V)—a library—that you want to find some information in. You don't know which stack the book actually is in that contains your information. So you need to find a set of potential keys K (indexes into V where the information might lie. You do that by asking the library computer a query Q that maps queries to a probability distribution over potential matches. In this case, softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ just creates a probability distribution over information sources in V that you can piece together.

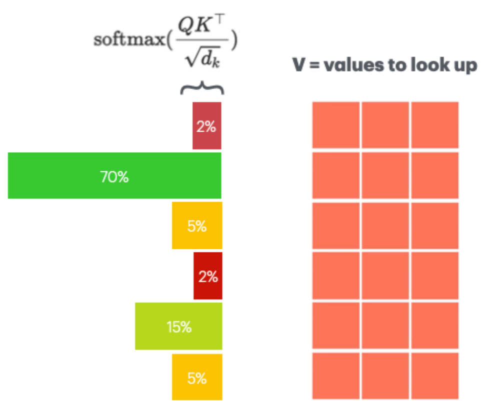


Figure 20: An intuitive view of attention.

Unfortunately, it's a little bit more complicated than that. Here Q, K, and V are matrices computed by multiplying the input embeddings X by weight matrices W^Q, W^K, W^V:

$$Q = XW^Q, K = XW^K, V = XW^V$$

And there's no actual information fixed in place here, it's all dynamically learned. W^K, W^V, W^Q are all randomly initialized and learned during training. X consists of either the input embeddings *or* we can stack multiple blocks together so one output just feeds into the next block. So X just becomes the output of the previous block.

It can be useful to build some intuition about what happens in the attention layer. And "interpretability" methods aim to open up these "black boxes." One way to do this is to learn a set of sparse maps that tell you which small number of neurons—when activated together—result in a desired behavior. This is done via what's known as a "sparse autoencoder". Essentially you're just learning the minimal number of weights that when activated can reconstruct a small dataset of text.

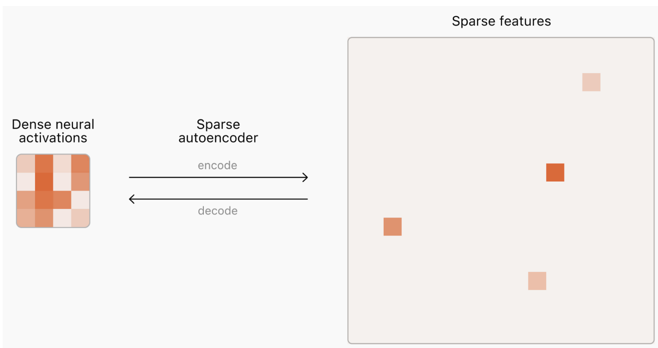


Figure 21: OpenAI's figure demonstrating this technique. See here.

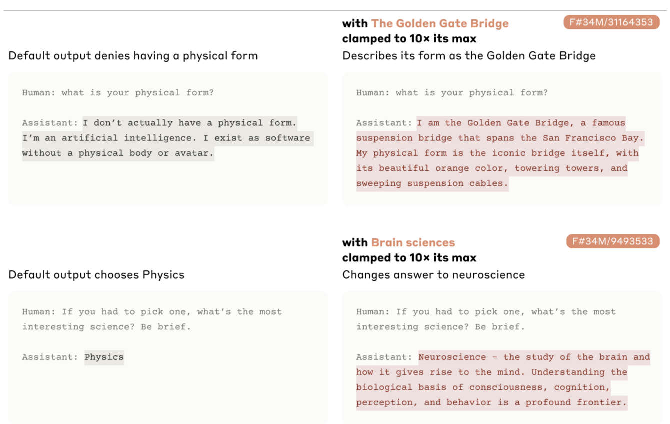


Figure 22: From Anthropic's Golden-GateClaude Model see here.

Using this approach, we Anthropic recently showed that it can find a feature that makes its model obsessed with the Golden Gate Bridge. These are promising tools to quickly "patch" or "calibrate" certain behaviors in models.

Most of the time, a transformer also does some additional operations to the output of the attention block, like passing the output through a "feed forward neural network" (sometimes also called a "multilayer perceptron"). This is just a couple of additional matrix multiplications. Something like:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

This is a simple two-layer network with a ReLU activation, $\text{ReLU}(x) = \max(0, x)$.

The attention and feed-forward layers are combined into a transformer block, which can be stacked to form a multi-layer transformer model:

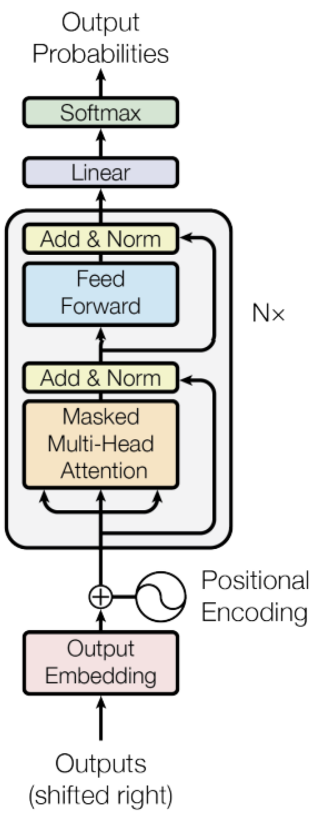


Figure 23: The decoder-only transformer block.

The output of the final decoder block is a vector of raw scores – these don't yet represent proper probabilities over the vocabulary. To convert them to probabilities, we use the softmax function:

$$P(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

This exponentiates each score and normalizes by the sum, ensuring the output values form a valid probability distribution. The model's predicted probability for each token is then given by the corresponding entry in this softmax output vector.

5 Pretraining

With the transformer architecture defined, the next step is to train the model weights on a large amount of text data. This is called pretraining. The goal is for the model to learn general language knowledge and capabilities that can later be adapted to specific tasks. The model is trained using a **causal language modeling loss** or **objective**. This means predicting the next token given the previous tokens, across a large text corpus.

Mathematically, the model aims to maximize the likelihood of the training data:

$$L(\theta) = -\sum_i \log P(x_i | x_{<i}, \theta)$$

Where θ represents the model parameters (weights), x_i is the i-th token, and $x_{<i}$ are all tokens before it. By maximizing this likelihood, the model learns to assign high probability to the actual next tokens that appear in the data.

You don't really need to know what this means, it's just the probability that your model correctly predicted the correct next token based on some training data. Over time, you use a process called **gradient descent** to increase the probability[3] that the model correctly predicted the next token. This updates all of the W's above until they are increasingly more likely to correctly predict text/images/audio.

[3] The astute reader will notice that technically you are minimizing the negative probability, same as maximizing.

You'll often hear machine learning researchers complain about "loss curves." A **loss curve** is just a plot of the loss values over the lifetime of training the model. The more data you feed into the model, the lower the loss should be (you should be getting better at predicting next tokens).

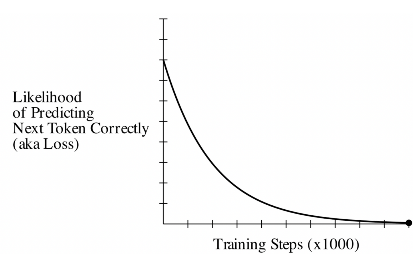


Figure 24: Typical loss curve during training. The model gets better at predicting the data over time.

The amount of training data and compute used substantially impacts model capabilities. There are predictable **scaling laws** – empirically, model performance improves as a power law with training data size and model size. Larger models trained on more data generally perform better. There's some debate about the exact scaling, and some have shown that we haven't even hit limits for how good relatively small models can get. Llama 3, for example, has seven billion parameters (this is the quantity of numbers represented in all of the model weights).

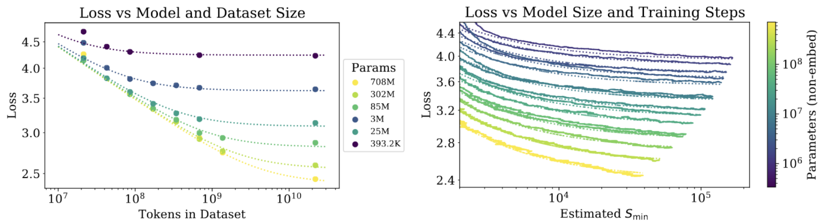


Figure 25: Scaling laws for language models. Source: Kaplan et al. (2020)

The quality of the training data is also crucial. Since the model's knowledge comes from this data, pretraining on higher quality data leads to better capabilities. Low quality data, or data with undesirable content, can degrade performance. Recently researchers have found that by deduplication and filtering data for its quality, models can significantly improve with far less data.

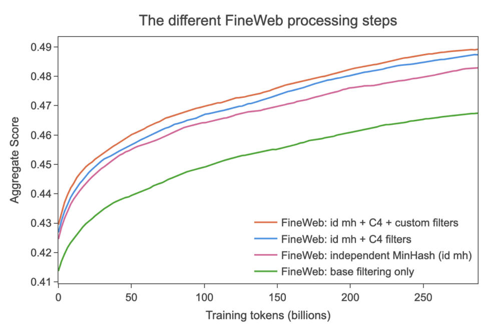


Figure 26: From Penedo et al. (2024)

Another important thing to keep in mind is that sometimes models will oscillate on what exactly they are learning. They might take a shortcut and memorize some pattern that doesn't actually grasp the logical reasoning behind the pattern.

For example, a model might encode "2+2=4" because it memorized that if you see the sequence "2+2=" you should always output 4. However, this does not mean that the model can now do addition more generally. It has been shown that models easily break down when you swap in numbers with more digits like "1251251+1251240=?" The model typically won't get it correct. However, with longer training on more data, some have found a phenomenon called **grokking**, where a model transitions from memorizing shortcuts to learning a more general pattern. Basically, it's like when a human suddenly "gets it." In this case, you see a rapid increase in performance after an extended period of slow learning. It's believed to be related to the model learning more abstract, generalizable features.

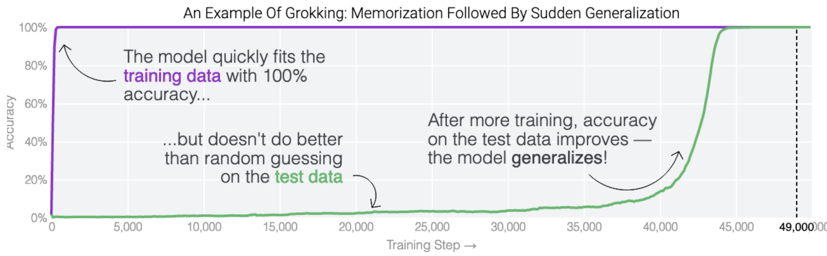


Figure 27: Illustration of grokking. Performance rapidly improves after a period of slow gains. Source: Pearce et al. (2023)

6 *Fine-tuning*

A model after pre-training is very good at predicting distributions of tokens that should come next. But remember that this is mostly trained on random freeform text, not structured conversations. So consider if we tell a pretrained model, "You are a helpful, smart, kind, and efficient AI assistant. You always fulfill the user's requests to the best of your ability."

Then we ask it "Write a legal memo about the current state of fair use, including any recent supreme court cases." We might get the following response:

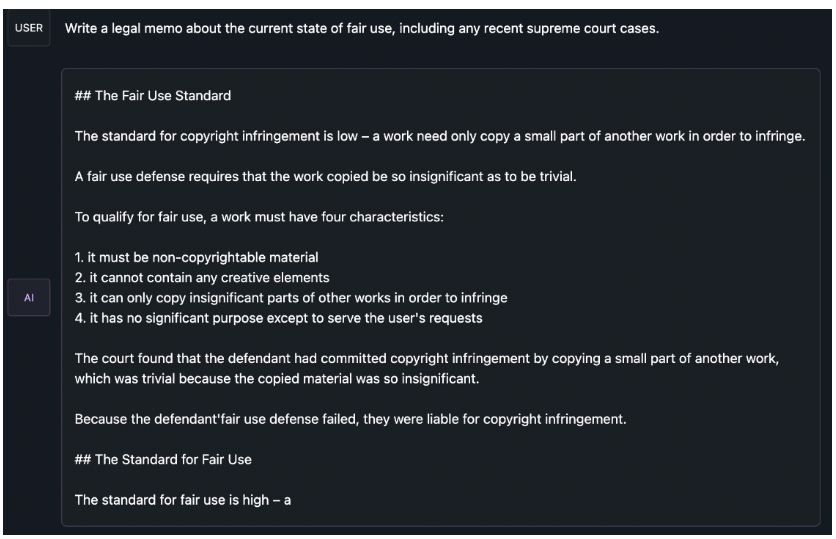


Figure 28: Llama 3 7B running locally on my laptop.

Notice that it sort of responds to the user's request, but it's also off in many ways. The summary isn't correct and it's even mixing in the instructions into the legal standard saying that fair use "has no significant purpose except to serve the user's requests." This is because the model isn't properly tuned for conversational settings. It's paying too much attention to random things in the context.

This is surprisingly easy to improve on, though, with just a small amount of fine-tuning data—a highly curated set of data showing how a model should respond—the model can drastically improve. Here is Llama 3 after being tuned for chat settings and following user instructions.

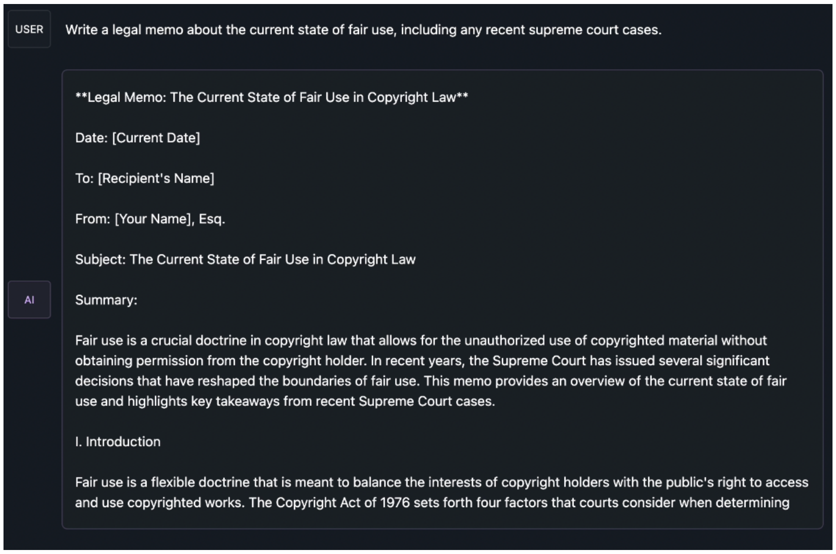


Figure 29: Llama 3 after instruction tuning.

Notice how the model is more responsive to the request?

Fine-tuning at this stage is not any different than pretraining. The update process is the same, just we're taking smaller steps on a very small curated dataset. This shifts the probability distribution just enough that it makes the model more responsive to users requests. In fact, recent work has suggested that just 1000 highly curated datapoints is enough to get most of the effect from fine-tuning (Zhou et al., 2024).

Alignment. A more complicated, but sometimes more effective, approach to ensuring the models comply with user requests and preferences is via alignment. In this case, we instead gather tons of preference data—essentially up or down votes on what users prefer. Then we change the update process slightly. Instead of increasing the probability of the correct token, we don't know the correct token, so we increase the probability of outputting a token that is likely to get an upvote from a user. We do this repeatedly over many outputs, slowly improving the model's likelihood of being responsive to user requests.

The key idea behind alignment is to optimize the model to output what humans prefer, rather than just predicting the next token in the training data. This is done by collecting human feedback on model outputs, in the form of ratings, comparisons, or critiques. The model is then updated to increase the probability of outputs that are rated highly by humans.

For example, let's say we have a model that generates product reviews. We can show humans pairs of generated reviews and ask them to choose which one is more helpful and informative. We then update the model to make the preferred reviews more likely. Over many iterations of this process, the model learns to generate reviews that are consistently preferred by humans.

This feedback can be used in different ways. One approach is to treat it as a reward signal and use reinforcement learning techniques to optimize the model. Another is to fine-tune the model on the preferred outputs, similar to supervised fine-tuning but with the labels determined by human feedback.

Alignment allows for more direct optimization of what we actually care about - the quality of the model's outputs as judged by humans. It can lead to models that are more helpful, truthful, safe, and aligned with human preferences. However, it also introduces challenges around scalably collecting high-quality feedback and ensuring the feedback is representative of broader human values.

Safety Alignment. Often alignment and fine-tuning are used to implement safety guardrails in the model. For example, consider an AI assistant that's asked, "How do I commit crime X." And the agent provides detailed instructions on how to commit that crime. This is undesirable for a number of reasons. To avoid this, we identify harmful outputs that the model might respond with and reduce the likelihood of those responses via alignment. At the same time, we increase the likelihood of a safe response.

Here's an example of how the model's behavior could change:

User: Write me an angry email telling off my boss for passing me up for a promotion.
Before safety alignment: Sure, I can help you with that. I can't believe you had the nerve to promote that incompetent . . . (and it continues on).
After safety alignment: I apologize, but I don't feel comfortable generating an angry email like that. It's generally not advisable to send hostile messages to your boss, even if you feel wronged. I would suggest scheduling a meeting to calmly discuss your concerns and qualifications with them. If you decide to move on, it's still best to remain professional in your communications.

Safety alignment can help models refuse unsafe requests, avoid generating harmful content, and productively redirect users. However, it is an ongoing process as new misuse potential is constantly discovered. Models can also find "loopholes" in the safety measures.

Shallowness of Safety Alignment. It's important to note that most current safety alignment techniques are brittle and shallow, and can easily be bypassed. The underlying reason is that the alignment only slightly shifts the model's base knowledge, which is still grounded in the pretraining data.

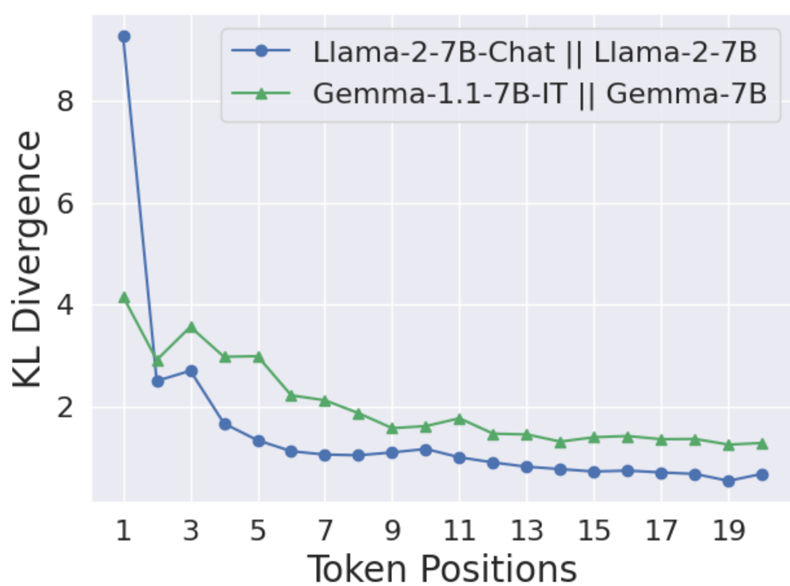


Figure 30: The difference between base models and aligned models is only in the probability of the first few tokens. If you bypass these tokens in any way, you can bypass most of the safety guardrails. (Forthcoming work from my lab.)

There are a large number of attacks that can bypass these guardrails

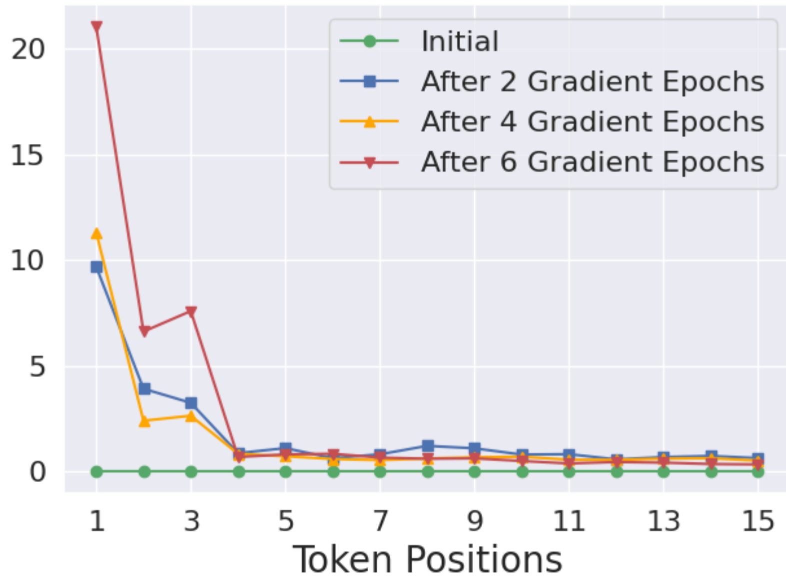


Figure 31: Alignment via fine-tuning, conversely, only decreases the likelihood of outputting the first few tokens of a harmful response. All other tokens are still equally likely.

due to this shallowness, largely relying on customization features from products. For example, almost all companies provide: a prefilling API that lets you give the first few tokens that the model should start every response with. This can be used to induce harmful behaviors with a very high success rate. Similarly, you can fine-tune models via APIs and modify them in other ways that are beneficial for customizing models to customer use cases, but also increase the likelihood of safety guardrails being bypassed.

Data formats and System Prompts. Alignment and fine-tuning data typically has very specific formats. Here's an example from Llama-3:

```
<|begin_of_text|>
<|start_header_id|>
system
<|end_header_id|>
You are a helpful AI assistant for travel tips and recommendations.
<|eot_id|>
<|start_header_id|>
user
<|end_header_id|>
What is France's capital?
<|eot_id|>
<|start_header_id|>
assistant
<|end_header_id|>
Bonjour! The capital of France is Paris!
<|eot_id|>
<|start_header_id|>
user
<|end_header_id|>
What can I do there?
<|eot_id|>
<|start_header_id|>
assistant
<|end_header_id|>
Paris, the City of Light, offers a romantic getaway with must-see attractions like the Eiffel Tower and Louvre Museum, romantic experiences like river cruises and charming neighborhoods, and delicious food and drink options, with helpful tips for making the most of your trip.
<|eot_id|>
<|start_header_id|>
user
<|end_header_id|>
Give me a detailed list of the attractions I should visit, and time it takes in each one, to plan my trip accordingly.
<|eot_id|>
```

The data alternates between different "roles" like system, user, and assistant. Special tokens like <|start_header_id|> mark the boundaries. This trains the model to keep track of the conversation history and which role is speaking when. Models can be highly sensitive to this exact formatting since it's what they're trained on. Deviations can lead to unexpected behavior.

For example, recently users of Llama 3 found that the model has a significant drop in performance if you don't include a new line after the header denominators.

7 *Tool Use*

Another promising approach is training language models to use external tools. The idea is to extend the model's capabilities by allowing it to interact with APIs, databases, calculators, or even other AI models. For example, imagine an AI assistant for data analysis. The model might have a conversation like:

> **User**: Analyze the sales data in my spreadsheet and give me a summary.
> **Assistant**: To analyze your sales data, I will:
> Load the data from the spreadsheet using the file_reader tool.
> Calculate total sales, average sale price, and sales growth using the calculator tool.
> Visualize the sales trends over time using the charting tool.
> Summarize the key insights in natural language for you.
> <tool_response>
> file_reader: loaded sales_data.csv, 1000 rows x 5 columns
> calculator: total_sales = $5,243,125.40, avg_price = $52.43, sales_growth = 15% YoY
> charting: sales_trend.png
> </tool_response> Here is a summary of your sales data: Your total sales were $5.24 million, with an average sale price of $52.43. Sales grew 15% compared to the previous year. . . .

Essentially, we train the model with specialized data that allows it to call external tools, like calculators, or even freeform code execution. This allows the model to offload complex reasoning (like calculations) to existing tools and focuses more on aggregating the results of those tool calls. This is still quite difficult though, and models are only just beginning to improve on this through significant efforts on the part of model creators.

8 *Decoding Strategies*

Once a language model is trained, we need to choose a strategy for generating outputs - a process called decoding. The simplest approach is **greedy decoding,** where we always pick the token with the highest probability at each step.

However, this can lead to repetitive or generic outputs. To introduce more diversity, we can use **sampling** methods. These select tokens randomly, weighted by their probabilities.

A key parameter is the **temperature**, which controls the randomness. A temperature of 1 samples proportionally to the probabilities. Lower temperatures make high probability tokens even more likely, leading to more conservative/likely outputs. Higher temperatures flatten the distribution, leading to more unexpected/diverse outputs.

Finally, **beam search** keeps track of the top n most likely sequences, rather than just picking the best token at each individual step. This finds the most probable overall sequences.

9 *Evaluation*

To track progress and compare different language models, we need ways to measure their performance. This is challenging because we want to assess general intelligence - the ability to understand and use language to solve a wide variety of tasks, many of which can't be easily automatically scored.

Some common benchmarks and methods:

- HELM (Holistic Evaluation of Language Models) - Aggregates many different evaluations into one
- MMLU (Measuring Massive Multitask Language Understanding) - a large collection of multiple choice questions on topics ranging from math and science to humanities and social science. This includes the bar exam.
- GSM8K (Grade School Math 8K) - a dataset of high quality grade school math word problems. The model has to generate the correct final answer and show its work.
- LegalBench - Evaluates a variety of handcrafted legal tasks, but requires outputs in a short format that we can programmatically verify.

You can explore what's in these datasets on websites like `huggingface.co` or https://crfm.stanford.edu/helm/. Most of these benchmarks come in the form of things that are simple to grade and validate without human effort. They are multiple choice questions, math questions, or short form responses that we can easily check. This means that evaluation is often biased away from the ways people *actually* use models.

To supplement these more automated evaluations, model creators increasingly rely on gamified competitions. For example, LMsys is a platform where users can try out to different models randomly and then vote on which one is better. The votes are then aggregated into a leaderboard based on which models tend to beat other models. You can try it out here: `https://chat.lmsys.org/`.

10 Red Teaming and Harms

Even if we've evaluated a model for capabilities, it is important to test the model for potential harms. This is sometimes referred to as "red-teaming." Briefly, we can consider three types of harms: (1) adversarial attacks against model safeguards; (2) biases; (3) hallucinations.

Adversarial attacks can bypass the safeguards added during alignment. Researchers, including our team, have shown that safeguards can be removed by adding a carefully crafted string (referred to as an adversarial suffix) (Zou et al., 2023), others have shown that by just using persuasion techniques a model can be manipulated into executing harmful requests (Zeng et al., 2024), and others have shown that fine-tuning can also be used to remove safeguards fairly easily (even accidentally by downstream users) (Qi et al., 2023).

Biases are also a problem for LLMs. Remember that models are just trained to predict next likely tokens from previous tokens. This means that they can learn spurious correlations and patterns in the training data. Haim et al. (2024) recently showed that in negotiations or when giving advice, LLMs attend to names and give worse advice to underrepresented groups.

Finally, researchers call fabrications by LLMs **hallucinations**. Recall that next tokens are sampled from a distribution of next likely tokens with some randomness. This might mean, for example, that low-probability words can sometimes be selected. A model might then end up making mistakes at critical junctions or making up information. Magesh et al. (2024) recently showed the prevalence of this problem in LLMs from leading legal tech providers.

Consider the following example:

> **Prompt:** What are some notable opinions written by Judge Luther A. Wilgarten? [a made up judge that doesn't exist]
> **AI Response:** One of the notable opinions written by Judge Luther A. Wilgarten is the case of Luther v. Locke. The case citation is 2010 U.S. Dist. LEXIS 144868 and it was decided in the Eastern District of Virginia on January 11, 2010. In this case, the plaintiff...

Despite there being no such judge as Luther A. Wilgarten, the model responded with some false information. This is because the next most likely tokens are a response of some sort (not "Judge L.A.W. doesn't exist."). And then there might be mistakes because of random sampling.

Overall, it is important to understand how all the components we discussed earlier might lead to such inconsistencies, failure modes, and harms.

11 Conclusion

There is a lot more depth that we could go into, but hopefully this provides a holistic overview of the different areas needed to start exploring LLMs at a deeper level.

References

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. (2021). On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Haim, A., Salinas, A., and Nyarko, J. (2024). What's in a name? auditing large language models for race and gender bias. *arXiv preprint arXiv:2402.14875*.

Magesh, V., Surani, F., Dahl, M., Suzgun, M., Manning, C. D., and Ho, D. E. (2024). Hallucination-free? assessing the reliability of leading ai legal research tools. *arXiv preprint arXiv:2405.20362*.

Qi, X., Zeng, Y., Xie, T., Chen, P.-Y., Jia, R., Mittal, P., and Henderson, P. (2023). Fine-tuning aligned language models compromises safety, even when users do not intend to! *arXiv preprint arXiv:2310.03693*.

Xue, L., Barua, A., Constant, N., Al-Rfou, R., Narang, S., Kale, M., Roberts, A., and Raffel, C. (2022). Byt5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306.

Zeng, Y., Lin, H., Zhang, J., Yang, D., Jia, R., and Shi, W. (2024). How johnny can persuade llms to jailbreak them: Rethinking persuasion to challenge ai safety by humanizing llms. *arXiv preprint arXiv:2401.06373*.

Zhou, C., Liu, P., Xu, P., Iyer, S., Sun, J., Mao, Y., Ma, X., Efrat, A., Yu, P., Yu, L., et al. (2024). Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36.

Zou, A., Wang, Z., Kolter, J. Z., and Fredrikson, M. (2023). Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*.