Weights & Biases

Current Best Practices for Training LLMs from Scratch

Authors: Rebecca Li, Andrea Parker, Justin Tenuto

Table of Contents

Introduction

Although we're only a few years removed from the transformer breakthrough, LLMs have already grown massively in performance, cost, and promise. At W&B, we've been fortunate to see more teams try to build LLMs than anyone else. But many of the critical details and key decision points are often passed down by word of mouth.

The goal of this white paper is to distill the best practices for training your own LLM for scratch. We'll cover everything from scaling and hardware to dataset selection and model training, letting you know which tradeoffs to consider and flagging some potential pitfalls along the way. This is meant to be a fairly exhaustive look at the key steps and considerations you'll make when training an LLM from scratch.

The first question you should ask yourself is whether training one from scratch is right for your organization. As such, we'll start there:

BUILD VS. BUY PRE-TRAINED LLM MODELS

Before starting LLM pre-training, the first question you need to ask is whether you should pre-train an LLM by yourself or use an existing one. There are three basic approaches:

- **Option 1: Use the API of a commercial LLM**, e.g. GPT-3 (OpenAI, 2020), Cohere APIs, AI21 J-1
- **Option 2: Use an existing open-sourced LLM**, e.g. GPT-J (EleutherAI, 2021), GPT-NeoX (EleutherAI, 2022), Galactica (Meta AI), UL2 (Google, 2022), OPT (Meta AI, 2022), BLOOM (BigScience, 2022), Megatron-LM (NVIDIA, 2021), CodeGen (Salesforce, 2022)
- **Option 3: Pre-train an LLM by yourself or with consultants:** You can either manage your own training or hire LLM consultants & platforms. For example, Mosaic ML provides training services focusing on LLMs.

That said, there are a lot of details to consider when making your choice. Here are the pros, cons, and applicable scenarios for each option:

Option 1 **Use the API of a commercial LLM**	Option 2 **Use an existing open-sourced LLM**	Option 3 **Pre-train an LLM by yourself or with consultants**
Pros		
• Requires the least LLM training technical skills. • Minimum upfront training / exploration cost, given main cost incurs at inference time. • The least data-demanding option. Only a few examples (or no examples) are needed for models to perform inference. • Can leverage the best-performing LLMs in the market and build a superior experience. • Reduce time-to-market of your apps and de-risk your project with a working LLM model.	• A good way to leverage what LLMs have learned from a vast amount of internet data and build on top of it without paying for the IP at inference. • Compared to option one, you are less dependent on the future direction of LLM service providers and thus have more control regarding roadmap & backwards compatibility. • Compared to option three, you have a much faster time-to-value given you are not building LLMs from scratch, also leading to less data, training time, training budget needed.	• Compared to options one and two, you have the most control of your LLM's performance and future direction, giving you lots of flexibility to innovate on techniques and/or customize to your downstream tasks. • Gain full control of training datasets used for the pre-training, which directly impacts model quality, bias, and toxicity issues. In comparison, those issues are less controllable in option one or two. • Training your own LLM also gives you a deep moat: superior LLM performance either across horizontal use cases or tailored to your vertical, allowing you to build a sustaining advantage especially if you create a positive data/feedback loop with LLM deployments.

Option 1 **Use the API of a commercial LLM**	Option 2 **Use an existing open-sourced LLM**	Option 3 **Pre-train an LLM by yourself or with consultants**
Cons		
• Commercial LLM services can get expensive with a high volume of fine-tuning or inference tasks. It comes down to LLM total-cost-of-ownership (TCO) amortized to each inference. • Many industries / use cases forbid the use of commercial LLM services as sensitive data / PII data cannot be seen by the service for compliance (healthcare use cases, for example). • If building external apps, you'll need to find other moats and de-risk your business if you're highly reliant on external LLM service technology. • Less flexible downstream: doesn't support edge inference, limited ability to customize the model (fine-tuning gets expensive), limited ability for ongoing model improvements.	• Not as demanding as building your own, but still requires lots of domain expert skills to train, fine-tune, and host an open-sourced LLM. LLM reproducibility is still a significant issue so the amount of time and work needed cannot be underestimated. • Slower time-to-market and less agile if you are building downstream apps, due to a more vertical tech stack. • Open-sourced models typically lag performance compared to commercial models by months/ years. If your competitor leverages commercial models, they have an advantage on LLM tech and you'll need to find other competitive advantages.	• Very expensive endeavor with high risks. Need cross-domain knowledge spanning from NLP/ML, subject matter expertise, software and hardware expertise. If not done well, you could end up in a situation where you've spent thousands or even millions of dollars with a suboptimal model. Mistakes, especially late into training stages, are hard to fix / unwind. • Less efficient than option two. Option two leverages existing LLMs, learning from an entire internet's worth of data and can provide a solid starting point. With option 3, you start from scratch and need lots of high-quality / diverse datasets for your models to gain generalized capabilities.
When to consider each option		
• Best if you either have less technical teams but want to leverage LLM techniques to build downstream apps, or you want to leverage the best-in-class LLMs for performance reasons (outsourcing the LLM tech). • Good if you have very limited training datasets and want to leverage an LLM's capability to do zero/few-shot learning. • Good for prototyping apps and exploring what is possible with LLMs.	• Between options two and three, if you aren't trying to change the model architecture, it is almost always better to either directly take an existing pre-trained LLM and fine-tune it or take the weights of an existing pre-trained LLM as a starting point and continue pre-training. The reason is because a good pre-trained LLM like GPT-NeoX has already seen a vast amount of data and thus has learned general capabilities from the data. You can leverage that learning especially if your training dataset is not huge or diverse. • Another typical scenario is that you operate in a regulatory environment or have user / sensitive data that cannot be fed to commercial LLM services. Or you need edge deployment of the model for latency or locational reasons.	• Best if you need to change model architecture or training dataset from existing pre-trained LLMs. For example, if you want to use a different tokenizer, change the vocabulary size, or change the number of hidden dimensions, attention heads, or layers. • Typically, in this case the LLM is a core part of your business strategy & technological moat. You are taking on some or a lot of innovations in LLM training, and have a large investment appetite to train and maintain expensive models on an ongoing basis. • Typically, you have or will have lots of proprietary data associated with your LLM to create a continuous model improvement loop for sustainable competitive advantage.

It is also worth mentioning that if you only have a very targeted set of use cases and don't need the general-purpose capabilities or generative capabilities from LLMs, you might want to consider training or fine-tuning a much smaller transformer or other much simpler deep learning models. That could result in much less complexity, less training time, and less ongoing costs.

THE SCALING LAWS

Before you dive into training, it's important to cover how LLMs scale. Understanding scaling lets you effectively balance the size and complexity of your model and the size of the data you'll use to train it.

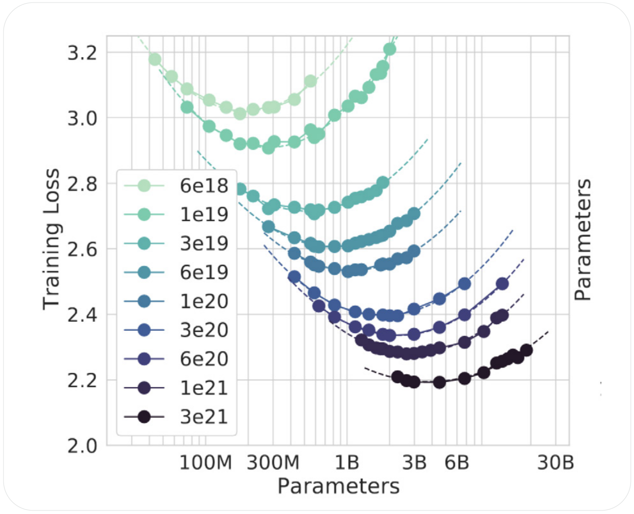
Some relevant history here: OpenAI originally introduced "the LLM scaling laws" in 2020. They suggested that increasing model size was more important than scaling data size. This held for about two years before DeepMind suggested almost the polar opposite: that previous models were significantly undertrained and that increasing your foundational training datasets actually leads to better performance.

That changed in 2022. Specifically, DeepMind put forward an alternative approach in their Training Compute-Optimal Large Language Models paper. They found that current LLMs are actually significantly undertrained. Put simply: these large models weren't trained on nearly enough data.

Deepmind showcased this with a model called Chinchilla, which is a fourth the size of the Gopher model above but trained on 4.6x more data. At that reduced size but with far more training data, Chinchilla outperformed Gopher and other LLMs.

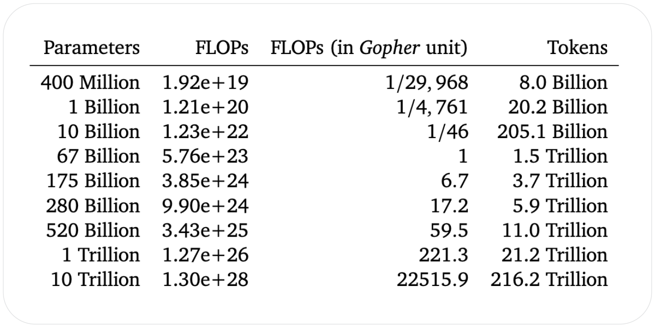
DeepMind claims that **the model size and the number of training tokens* should instead increase at roughly the same rate to achieve optimal performance.** If you get a 10x increase in compute, you should make your model 3.1x times bigger and the data you train over 3.1x bigger; if you get a 100x increase in compute, you should make your model 10x bigger and your data 10x bigger.

*Note: Tokenization in NLP is an essential step of separating a piece of text into smaller units called tokens. Tokens can be either words, characters, or subwords. The number of training tokens is the size of training data in token form after tokenization. We will dive into detailed tokenization methods a little later.



To the left of the minima on each curve, models are too small -- a larger model trained on less data would be an improvement. To the right of the minima on each curve, models are too large -- a smaller model trained on more data would be an improvement. The best models are at the minima.

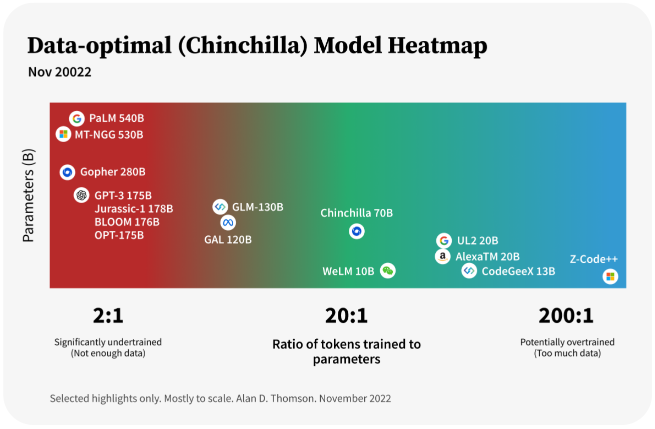
DeepMind provides the following chart showing how much training data and compute you'd need to optimally train models of various sizes.



Parameters	FLOPs	FLOPs (in *Gopher* unit)	Tokens
400 Million	1.92e+19	1/29,968	8.0 Billion
1 Billion	1.21e+20	1/4,761	20.2 Billion
10 Billion	1.23e+22	1/46	205.1 Billion
67 Billion	5.76e+23	1	1.5 Trillion
175 Billion	3.85e+24	6.7	3.7 Trillion
280 Billion	9.90e+24	17.2	5.9 Trillion
520 Billion	3.43e+25	59.5	11.0 Trillion
1 Trillion	1.27e+26	221.3	21.2 Trillion
10 Trillion	1.30e+28	22515.9	216.2 Trillion

Estimated optimal training FLOPs and training tokens for various model sizes, Training Compute-Optimal Large Language Models

That said, most existing LLMs are still undertrained:



Data/compute-optimal (Chinchilla) heatmap, Chinchilla data-optimal scaling laws: In plain English

In summary, the current best practices in choosing the size of your LLM models are largely based on two rules:

- Decide on your dataset and find the Chinchilla-optimal model size based on data size (or close to Chinchilla-optimal within the boundary of your data collection limitation)
- Determine the data and model size combination that's best for your model, based on your training compute budget and inference latency requirements

HARDWARE

It should come as no surprise that pre-training LLMs is a hardware-intensive effort. The following examples of current models are a good guide here:

- **PaLM (540B, Google)**: 6144 TPU v4 chips used in total, made of two TPU v4 Pods connected over data center network (DCN) using a combination of model and data parallelism
- **OPT (175B, Meta AI)**: 992 80GB A100 GPUs, utilizing fully shared data parallelism with Megatron-LM tensor parallelism
- **GPT-NeoX (20B, EleutherAI):** 96 40GB A100 GPUs in total
- **Megatron-Turing NLG (530B, NVIDIA & MSFT)**: 560 DGX A100 nodes, each cluster node has 8 NVIDIA 80-GB A100 GPUs

Training LLMs is challenging from an infrastructure perspective for two big reasons. For starters, it is simply no longer possible to fit all the model parameters in the memory of even the largest GPU (e.g. NVIDIA 80GB-A100), so you'll need some parallel architecture here. The other challenge is that a large number of compute operations can result in unrealistically long training times if you aren't concurrently optimizing your algorithms, software, and hardware stack (e.g. training GPT-3 with 175B parameters would require about 288 years with a single V100 NVIDIA GPU).

Memory vs. Compute Efficiency

To achieve the full potential of thousands of distributed GPUs, it is crucial to design parallelism into your architecture to balance memory and compute efficiency.

Memory efficiency

Training a LLM requires terabytes of aggregate memory for model weights, gradients, and optimizer states - far beyond what is available on a single GPU. One typical mitigation strategy is gradient accumulation, in which the full training batch is split into micro-batches that are processed in sequence with their resulting gradients accumulated before updating the model weights. That means your training batch size can scale without increasing the peak resident activation memory.

Compute efficiency

While large GPU clusters can have thousands of high-throughput GPUs, achieving high compute efficiency at this scale is challenging. A large batch size can be an effective way to increase compute efficiency, because it increases the arithmetic intensity of a GPU kernel and helps amortize the time spent stalled on communication and synchronization. However, using too large of a batch size can have negative effects on the model quality.

While parallelization is paramount, there are many different ways to do it. We'll get into the most common in our next section.

Techniques for Parallelization

Parallelization refers to **splitting up tasks and distributing them across multiple processors or devices, such as GPUs, so that they can be completed simultaneously.** This allows for more efficient use of compute resources and faster completion times compared to running on a single processor or device. Parallelized training across multiple GPUs is an effective way to reduce the overall time needed for the training process.

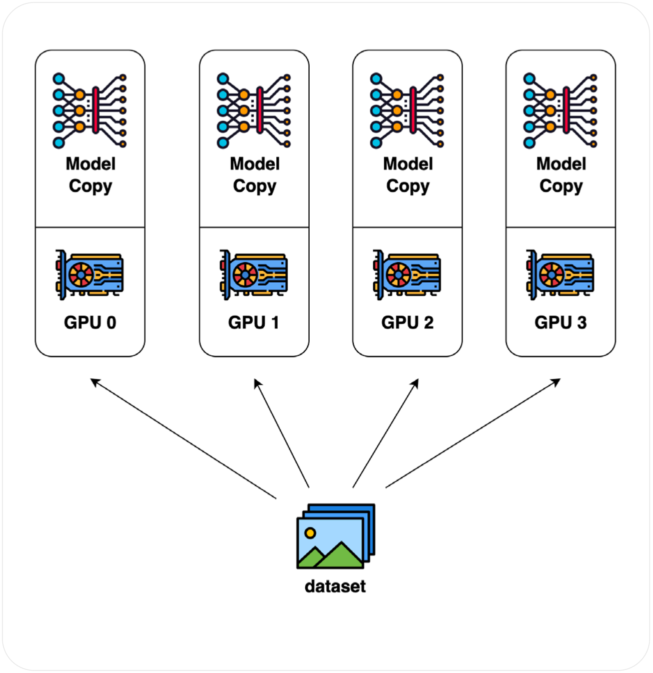
There are several different strategies that can be used to parallelize training, including gradient accumulation, micro-batching, data parallelization, tensor parallelization and pipeline parallelization, and more. Typical LLM pre-training employs a combination of these methods. Let's define each:

Data Parallelism

Data parallelism is the best and most common approach for dealing with large datasets that cannot fit into a single machine in a deep learning workflow.

More specifically, data parallelism divides the training data into multiple shards (partitions) and distributes them to various nodes. Each node first works with its local data to train its sub-model, and then communicates with the other nodes to combine their results at certain intervals in order to obtain the global model. The parameter updates for data parallelism can be either asynchronous or synchronous.

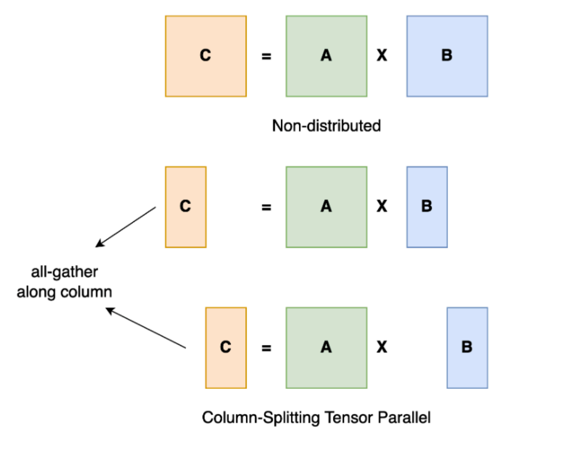
The advantage of this method is that it increases compute efficiency and that it is relatively easy to implement. The biggest downside is that during the backward pass you have to pass the whole gradient to all other GPUs. It also replicates the model and optimizer across all workers which is rather memory inefficient.



Tensor Parallelism

Tensor parallelism divides large matrix multiplications into smaller submatrix calculations which are then executed simultaneously using multiple GPUs.

This allows for faster training times due to its asynchronous nature and the ability to reduce communication overhead between nodes. The benefit of this method is that it is memory-efficient. The downside, however, is that it introduces additional communication of activations in each forward & backward propagation, and therefore requires high communication bandwidth to be efficient.

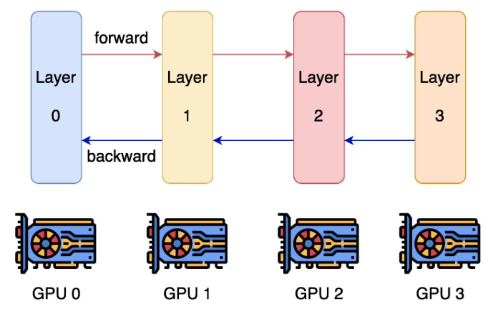


Tensor parallel illustration

Pipeline parallelism and model parallelism

Pipeline parallelism improves both the memory and compute efficiency of deep learning training by partitioning the layers of a model into stages that can be processed in parallel.

This helps with overall throughput speeds significantly while adding the smallest communication overhead. You can think of pipeline parallelism as "inter-layer parallelism" (where tensor parallelism can be thought of as "intra-layer parallelism"). Similar to pipeline parallelism, model parallelism is when you split the model among GPUs and use the same data for each model; so each GPU works on a part of the model rather than a part of the data. The downside of pipeline and model parallelism is that it cannot scale infinitely given that the degree of pipeline parallelism is bounded by the depth of the model.



Pipeline parallel illustration

As mentioned at the start of this section, it's not uncommon for teams to leverage a combination of parallelism techniques during training. For example, PaLM (Google Brain, 2022) and OPT (Meta AI, 2022) both used a combination of tensor model parallelism and data parallelism.

NVIDIA approached things a little differently in the Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM paper. They proposed a PTD-P technique that combines pipeline, tensor, and data parallelism to achieve state-of-the-art computational performance (52% of peak device throughput) on 1000s of GPUs.

Specifically, PTD-P leverages a combination of pipeline parallelism across multi-GPU servers, tensor parallelism within a multi-GPU server, and data parallelism to practically train models with a trillion parameters. The method also employs graceful scaling in an optimized cluster environment with high-bandwidth links between GPUs on the same server and across servers.

Using these techniques to train LLMs requires not only the highest-performing GPUs to be efficient, but also needs high-bandwidth networking for optimal communication––InfiniBand is often used to move data between nodes.

But this of course comes with a cost. Leveraging thousands of high-performing GPUs and high-bandwidth networks to train LLMs is infrastructure-intensive. For example, a back-of-the-envelope calculation estimated that the cost of the PaLM model (540B, Google) might be as high as $23MM (see detailed analysis).

To implement distributed deep learning training systems, software toolkits such as Distributed TensorFlow, Torch Distributed, Horovod, and libraries such as DeepSeed and Megatron are often needed. There is implementation complexity here so it requires system expertise if you're going to be successful.

In addition, the following techniques and strategies are commonly employed to achieve parallelism:

Gradient accumulation

Gradient accumulation involves adding up gradients from multiple batches before performing one weight update step on all accumulated gradients at once.

This approach reduces communication overhead between GPUs by allowing them to work independently on their own local batch of data until they have synchronized with each other again, after accumulating enough gradients for a single optimization step.

Asynchronous stochastic gradient descent optimization

Asynchronous stochastic gradient descent optimization methods can also be employed when performing model optimization over multiple GPUs.

This method uses small subsets (microbatches) of data from each node instead of loading all data at once, which helps reduce memory requirements while still allowing for fast convergence rates due to its asynchronous nature. It works like this:

- First, we fetch the most up-to-date parameters of the model needed to process the current mini-batch from the parameter servers.
- We then compute gradients of the loss with respect to these parameter
- Finally, these gradients are sent back to the parameter servers, which then updates the model accordingly.

Micro-batching

Micro-batching combines small mini-batches into larger ones so that more batches can be processed in less time and with fewer synchronization points between devices during backpropagation operations. It has become increasingly popular for training very large models across many GPUs due to its ability to reduce memory consumption and improve scalability. Overall, micro-batching is an effective way to leverage distributed deep learning techniques when dealing with very large datasets or models that require significant amounts of processing power.

Now that we've gone through scaling, hardware, and some techniques for parallelizing your training runs, let's look at what your LLM will actually learn from: data.

DATASET COLLECTION

Bad data leads to bad models. But careful processing of high-quality, high-volume, diverse datasets directly contributes to model performance in downstream tasks as well as model convergence.

Dataset diversity is especially important for LLMs. That's because diversity improves the cross-domain knowledge of the model, as well as its downstream generalization capability. Training on diverse examples effectively broadens the ability of your LLM to perform well on myriad nuanced tasks.

A typical training dataset is comprised of textual data from diverse sources, such as crawled public data, online publication or book repositories, code data from GitHub, Wikipedia, news, social media conversations, etc.

For example, consider The Pile. The Pile is a popular text corpus created by EleutherAI for large-scale language modeling. It contains data from 22 data sources, coarsely broken down into five broad categories:

- Academic Writing: PubMed Abstracts and PubMed Central, arXiv, FreeLaw, USPTO Backgrounds, PhilPapers, NIH Exporter
- Online or Scraped Resources: CommonCrawl, OpenWebText2, Stack Exchange, Wikipedia
- Prose: BookCorpus2, Bibliotik, Project Gutenberg
- Dialog: YouTube subtitles, Ubuntu IRC, OpenSubtitles, Hacker News, Europarl
- Miscellaneous: GitHub, the DeepMind Mathematics dataset, Enron emails

Note that The Pile dataset is one of the very few large-scale text datasets that is free for the public. For most of the existing models like GPT-3, PaLM, and Galactica, their training and evaluation datasets are not publicly available. Given the large scale effort it takes to compile and pre-process these datasets for LLM training, most companies have kept them in-house to maintain competitive advantage. That makes datasets like The Pile and a few datasets from AllenAI extremely valuable for public large-scale NLP research purposes.

Another thing worth mentioning is that, during dataset collection, general data can be collected by non-experts but data for specific domains normally needs to be collected or consulted by subject matter experts (SMEs), e.g. doctors, physicists, lawyers, etc. SMEs can flag thematic or conceptual gaps that NLP engineers might miss. NLP engineers should also be heavily involved at this stage given their knowledge of how a LLM "learns to represent data" and thus their abilities to flag any data oddities or gaps in the data that SMEs might miss.

Once you've identified the dataset(s) you'll be using, you'll want to prepare that data for your model. Let's get into that now:

DATASET PRE-PROCESSING

In this section, we'll cover both data adjustments (like deduplication and cleaning) and the pros and cons of various tokenization strategies. Let's start with the former:

Dataset Handling

To ensure training data is high-quality and diverse, several pre-processing techniques can be used before the pre-training steps:

Data sampling:

Certain data components can be up-sampled to obtain a more balanced data distribution. Some research down-samples lower-quality dataset such as unfiltered web crawl data. Other research up-samples data of a specific set of domains depending on the model objectives.

There are also advanced methods to filter high-quality data such as using a trained classifier model applied to the dataset. For example, the model Galactica by Meta AI is built purposefully for science, specifically storing, combining and reasoning about scientific knowledge.

Due to its goals, the pre-training dataset is composed of high-quality data mainly from science resources such as papers, textbooks, lecture notes, encyclopedias. The dataset is also highly curated, for example, with task-specific datasets to facilitate composition of this knowledge into new task contexts.

Data cleaning

Normally, data cleaning and reformatting efforts are applied before training. Examples include removing boilerplate text and removing HTML code or markup. In addition, for some projects, fixing misspellings, handling cross-domain homographs, and/or removing biased / harmful speech are performed to improve model performance. For other projects, these techniques are not used under the idea that models should see the fair representation of the real world and learn to deal with misspellings and toxicity as a part of the model capabilities.

Non-standard textual components handling

In some cases, it is important to convert non-standard textual components into texts, e.g. converting emoji into their text equivalent: ❄️ becomes "snowflake."This conversion can be done programmatically, of course.

Data deduplication

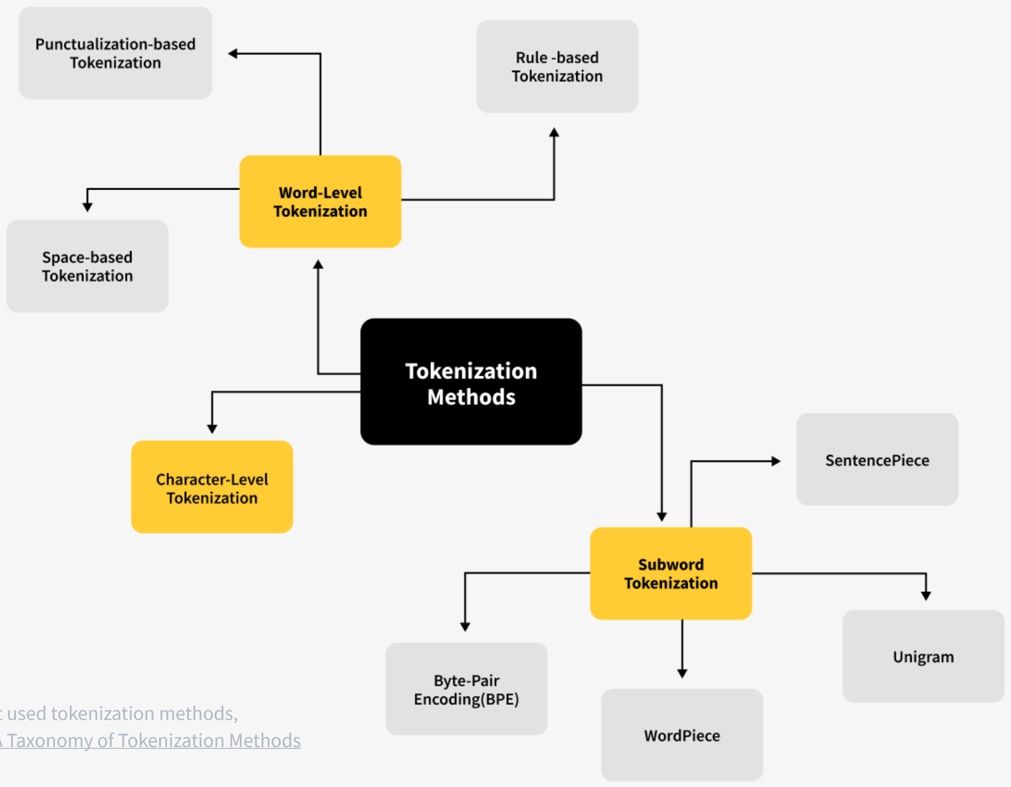
Some researchers see significant benefits from deduplicating training data. Fuzzy deduplication methods such as locality sensitive hashing (LSH) are commonly used here. See Deduplicating Training Data Makes Language Models Better paper to understand details regarding deduplication.

Downstream task data removal

Data leakage happens when the data you are using to train happens to have the information you are trying to predict. Downstream task data removal methods (such as n-grams) are needed to remove training data also present in the evaluation dataset.

Tokenization

Tokenization is the process of **encoding a string of text into transformer-readable token ID integers.** Most state-of-the-art LLMs use subword-based tokenizers like byte-pair encoding (BPE) as opposed to word-based approaches. We'll present the strengths and weaknesses of various techniques below, with special attention to subword strategies as they're currently the most popular versus their counterparts.



Summary of the most used tokenization methods,
Two minutes NLP — A Taxonomy of Tokenization Methods

Tokenization Methods	Word-based tokenization	Character-based tokenization	Subword-based tokenization
Example Tokenizers	Space tokenization (split sentences by space); rule-based tokenization (e.g. Moses, spaCy)	Character tokenization (simply tokenize on every character)	Byte-Pair Encoding (BPE); WordPiece; SentencePiece; Unigram (tokenizing by parts of a word vs. the entirety of a word; see table above)
Considerations	• **Downside:** Generates a very large vocabulary leading to a huge embedding matrix as the input and output layer; large number of out-of-vocabulary (OOV) tokens; and different meanings of very similar words • Transformer models normally have a vocabulary of less than 50,000 words, especially if they are trained only on a single language	• Lead to much smaller vocabulary; no OOV (out of vocabulary) tokens since every word can be assembled from individual characters • **Downside:** Generates very long sequences and less meaningful individual tokens, making it harder for the model to learn meaningful input representations. However, if character-based tokenization is used on non-English language, a single character could be quite information rich (like “mountain” in Mandarin).	• Subword-based tokenization methods follow the principle that frequently used words should not be split into smaller subwords, but rare words should be decomposed into meaningful subwords • **Benefit:** Solves the downsides faced by word-based tokenization and character-based tokenization and achieves both reasonable vocabulary size with meaningful learned context-independent representations.

In other words, the choice of tokenization technique depends on the specific task and language being analyzed.

- Word-based tokenization is simple and efficient but can be limited in its ability to handle complex languages.
- Character-based tokenization can be useful for languages without distinct word boundaries.
- Subword-based tokenization, including BPE, wordpiece tokenization, sentencepiece tokenization, and unigram tokenization, is particularly useful for handling complex morphology and out-of-vocabulary words.

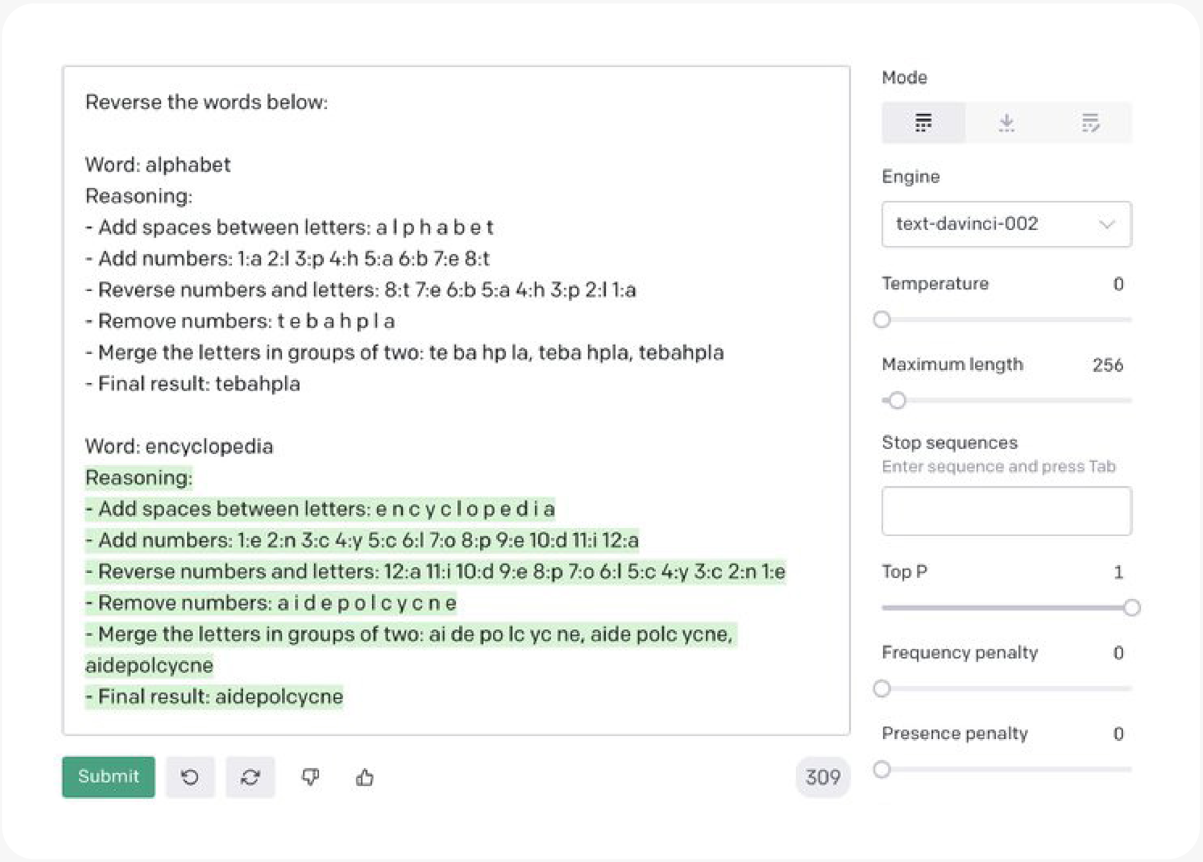
Let’s look at those subword-based methods in a little more detail:

Subword-based Tokenization Methods	Byte-Pair Encoding (BPE)	WordPiece	Unigram	SentencePiece
Description	One of the most popular subword tokenization algorithms. The Byte-Pair-Encoding works by starting with characters, while merging those that are the most frequently seen together, thus creating new tokens. It then works iteratively to build new tokens out of the most frequent pairs it sees in a corpus. BPE is able to build words it has never seen by using multiple subword tokens, and thus requires smaller vocabularies, with less chances of having “unk” (unknown) tokens.	Very similar to BPE. The difference is that WordPiece does not choose the highest frequency symbol pair, but the one that maximizes the likelihood of the training data once added to the vocabulary (evaluates what it loses by merging two symbols to ensure it’s worth it)	In contrast to BPE / WordPiece, Unigram initializes its base vocabulary to a large number of symbols and progressively trims down each symbol to obtain a smaller vocabulary. It is often used together with SentencePiece.	The left 3 tokenizers assume input text uses spaces to separate words, and therefore are not usually applicable to languages that don’t use spaces to separate words (e.g. Chinese). SentencePiece treats the input as a raw input stream, thus including the space in the set of characters to use. It then uses the BPE / Unigram algorithm to construct the appropriate vocabulary.
Considerations	BPE is particularly useful for handling rare and out-of-vocabulary words since it can generate subwords for new words based on the most common character sequences. Downside: BPE can result in subwords that do not correspond to linguistically meaningful units.	WordPiece can be particularly useful for languages where the meaning of a word can depend on the context in which it appears.	Unigram tokenization is particularly useful for languages with complex morphology and can generate subwords that correspond to linguistically meaningful units. However, unigram tokenization can struggle with rare and out-of-vocabulary words.	SentencePiece can be particularly useful for languages where the meaning of a word can depend on the context in which it appears.
Transformers using this tokenization method	GPT-2, GPT3, Roberta	BERT, DistilBERT, Electra	Unigram is not used directly for any of the transformer models. Instead, it’s used together with SentencePiece.	ALBERT, XLNet, Marian, T5

Note: there is also recent work proposing a token-free model (ByT5) that learns from raw bytes. The benefit is that these models can process text in any language out of the box, that they work better with corpora with a large quantity of OOV (out-of-vocabulary) words/tokens and are more robust to noise, and that they minimize technical debt by removing complex and error-prone text preprocessing pipelines. The downside is that they are in general less accurate than tokenization-based models. More research in this area is needed to determine how promising this direction is.

After tokenization, we usually want to consider a few extra steps, namely padding and truncation. Padding is a strategy to ensure tensors are rectangular by adding a special padding token to shorter sentences so all inputs have a uniform tensor shape. In contrast, truncation shortens the length of sequences for the ones that are too long for a model to handle.

Note: there is an inherent limitation for subword tokenizer-based LLMs that is directly related to the tokenizer technique. Due to that subword-based tokenizer's token granularity is between word and character, LLMs don't see letters and words as humans do. Instead, it sees "tokens," which are chunks of characters. An example due to this inherent limitation is that they have trouble merging characters.



Reverse the words below:

Word: alphabet
Reasoning:
- Add spaces between letters: a l p h a b e t
- Add numbers: 1:a 2:l 3:p 4:h 5:a 6:b 7:e 8:t
- Reverse numbers and letters: 8:t 7:e 6:b 5:a 4:h 3:p 2:l 1:a
- Remove numbers: t e b a h p l a
- Merge the letters in groups of two: te ba hp la, teba hpla, tebahpla
- Final result: tebahpla

Word: encyclopedia
Reasoning:
- Add spaces between letters: e n c y c l o p e d i a
- Add numbers: 1:e 2:n 3:c 4:y 5:c 6:l 7:o 8:p 9:e 10:d 11:i 12:a
- Reverse numbers and letters: 12:a 11:i 10:d 9:e 8:p 7:o 6:l 5:c 4:y 3:c 2:n 1:e
- Remove numbers: a i d e p o l c y c n e
- Merge the letters in groups of two: ai de po lc yc ne, aide polc ycne, aidepolcycne
- Final result: aidepolcycne

Mode
Engine: text-davinci-002
Temperature: 0
Maximum length: 256
Stop sequences: Enter sequence and press Tab
Top P: 1
Frequency penalty: 0
Presence penalty: 0

Example from Peter Welinder

PRE-TRAINING STEPS

Training a multi-billion parameter LLM is usually a highly experimental process with lots of trial and error. Normally, the team would start with a much smaller model size, make sure it's promising, and scale up to more and more parameters. Keep in mind that as you scale, there will be issues that require addressing which simply won't be present when training on smaller data sizes.

Let's look at some common pre-training steps, starting with architecture.

Model Architecture

To reduce risk of training instabilities, practitioners often start with the model architecture and hyperparameters of a popular predecessor such as GPT-2 and GPT-3 and make informed adjustments along the way to improve training efficiency, scale the size of the models (in both depth and width), and improve performance. Two examples:

GPT-NeoX-20B (20B, EleutherAI) originally took GPT-3's architecture and made these changes:

- Rotary embedding used for the first 25% of embedding vector dimensions instead of learned positional embeddings, to balance performance and computational efficiency.
- Parallel attention combined with feed forward layers instead of running them in series, primarily for computing efficiency purposes.
- While GPT-3 uses alternating dense and sparse layers, GPT-NeoX exclusively uses dense layers to reduce implementation complexity.

OPT-175B (175B, Meta AI) also built on GPT-3 and adjusted:

- Batch size for increased computational efficiency.
- Learning rate schedule. Specifically, it follows a linear learning rate (LR) schedule, warming up from 0 to the maximum learning rate over the first 2000 steps in OPT-175B, or over 375M tokens in the smaller baselines, then decaying down to 10% of the maximum LR over 300B tokens. A number of mid-flight changes to LR were also required.
- Token amount. OPT-175B, despite the same model size as GPT-3 (175B) was trained on a much smaller dataset of 180B tokens (as compared to the 300B tokens by GPT-3).

Experiments and Hyperparameter Search

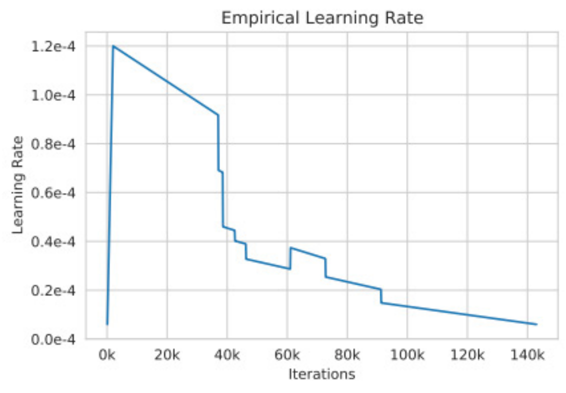
As we mentioned above, typical pre-training involves lots of experiments to find the optimal setup for model performance. Experiments can involve any or all of the following: weight initialization, positional embeddings, optimizer, activation, learning rate, weight decay, loss function, sequence length, number of layers, number of attention heads, number of parameters, dense vs. sparse layers, batch size, and dropout.

A combination of manual trial and error of those hyperparameter combinations and automatic hyperparameter optimization (HPO) are typically performed to find the optimal set of configurations to achieve optimal performance. Typical hyperparameters to perform automatic search on: learning rate, batch size, dropout, etc.

Hyperparameter search is an expensive process and is often too costly to perform at full scale for multi-billion parameter models. It's common to choose hyperparameters based on a mixture of experiments at smaller scales and by interpolating parameters based on previously published work instead of from scratch.

In addition, there are some hyperparameters that need to be adjusted even during a training epoch to balance learning efficiency and training convergence. Some examples:

- Learning rate: can increase linearly during the early stages, then decay towards the end.
- Batch size: it's not uncommon to start with smaller batch sizes and gradually ramp up to larger ones.



You'll want to do a lot of this early in your pre-training process. This is largely because you'll be dealing with smaller amounts of data, letting you perform more experiments early versus when they'll be far more costly down the line.

Before we continue, it's worth being clear about a reality here: you will likely run into issues when training LLMs. After all: these are big projects and like anything sufficiently large and complicated, things can go wrong..

Hardware Failure

During the course of training, a significant number of hardware failures can occur in your compute clusters, which will require manual or automatic restarts. In manual restarts, a training run is paused, and a series of diagnostics tests are conducted to detect problematic nodes. Flagged nodes should then be cordoned off before you resume training from the last saved checkpoint.

Training Instability

Training stability is also a fundamental challenge. While training the model, you may notice that hyperparameters such as learning rate and weight initialization directly affect model stability. For example, when loss diverges, lowering the learning rate and restarting from an earlier checkpoint might allow the job to recover and continue training.

Additionally, the bigger the model is, the more difficult it is to avoid loss spikes during training. These spikes are likely to occur at highly irregular intervals, sometimes late into training.

There hasn't been a lot of systematic analysis of principled strategy to mitigate spikes. Here are some best practices we have seen from the industry to effectively get models to converge:

- **Batch size:** in general, using the biggest batch size that your GPU allows you to use is the best policy here.
- **Batch Normalization:** Normalizing the activations within a mini-batch can speed up convergence and improve model performance.
- **Learning Rate Scheduling:** A high learning rate can cause the loss to oscillate or diverge, leading to loss spikes. By scheduling the learning rate to decrease over time, you can gradually decrease the magnitude of updates to the model's parameters and improve stability. Common schedules include step decay, where the learning rate is decreased by a fixed amount after a fixed number of steps, and exponential decay, where the learning rate is decreased by a fixed factor each step. Note that it is not really possible to know ahead of time what LR to use, but you can use different LR schedules to see how your model responds. See more details here.
- **Weight Initialization:** Properly initializing the weights can help the model converge faster and improve performance. For example, it is common to use small Gaussian noise or, in the case of Transformers, the T-Fixup initialization. Techniques that can be used for weight initialization include random initialization, layer-wise initialization, and initialization using pretrained weights.
- **Model training starting point:** Using a pretrained model that is trained on related tasks as a starting point can help the model converge faster and improve performance.
- **Regularization:** Regularization techniques, such as dropout, weight decay, and L1/L2 regularization, can help the model converge better by reducing overfitting and improving generalization.
- **Data Augmentation:** Augmenting the training data by applying transformations can help the model generalize better and reduce overfitting.
- **Hot-swapping during training:** Hot-swapping of optimizers or activation functions are sometimes used during LLM training to fix issues as they appear during the process. It sometimes requires a team on it almost 24/7 trying various heuristics to train further.
- **Other simple strategies mitigating the instability issue when encountered:** Restart training from a previous checkpoint; skip some data batches that were seen during the spike (the intuition is that pikes occur due to the combination of specific data batches with a particular model parameter state).

Note: most of the above model convergence best practices not only apply to transformer training, but also apply in a broader deep learning context across architectures and use cases.

Finally, after your LLM training is completed, it is very important to ensure that your model training environment is saved and retained in that final state. That way, if you need to re-do anything or replicate something in the future you can because you have the training state preserved.

A team could also try some ablation studies. This allows you to see how pulling parts of the model out might impact performance. Ablation studies can allow you to massively reduce the size of your model, while still retaining most of a model's predictive power.

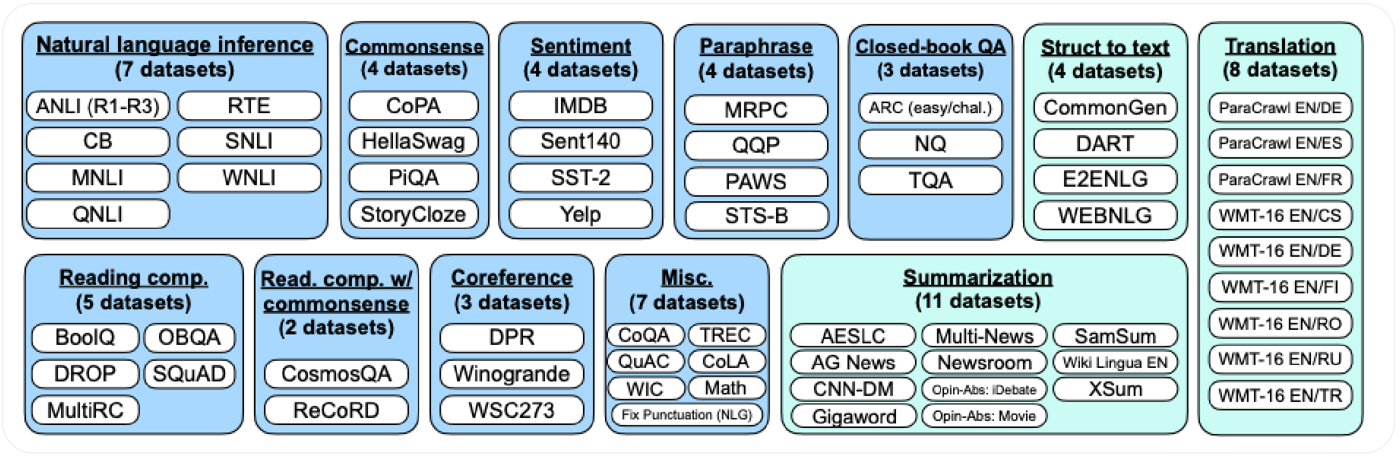
MODEL EVALUATION

Typically, pre-trained models are evaluated on a diverse language model datasets to assess their ability to perform logical reasoning, translation, natural language inference, question answering, and more.

Machine learning practitioners coalesced around a variety of standard evaluation benchmarks. A few popular examples include:

- Open-Domain Question Answering tasks: TriviaQA, Natural Questions, Web Questions
- Cloze and Completion tasks: LAMBADA, HellaSwag, StoryCloze
- Winograd-style tasks: Winograd, WinoGrande
- Common Sense Reasoning: PIQA, ARC, OpenBookQA
- In-context Reading Comprehension: DROP, CoQA , QuAC, SQuADv2, RACE, SuperGLUE
- Natural Language Inference (NLI): SNLI, QNLI
- Reasoning tasks: Arithmetic reasoning tasks
- Code tasks: HumanEval, MBPP (text-to-code); TransCoder (code-to-code)
- Translation tasks: Translation BLEU score on WMT language pairs
- BIG-bench: A collaborative benchmark aimed at producing challenging tasks for large language models, including 200+ tasks covering diverse textual tasks and programmatic tasks.
- LM Evaluation Harness: A library for standardized evaluation of autoregressive LLMs across 200+ tasks released by EleutherAI. It has gained popularity because of its systematic framework approach and robustness.

Here is a summary of typical language tasks (NLU tasks in blue; NLG tasks in teal):



Datasets and task clusters in NLP, Finetuned Language Models are Zero-Shot Learners

Another evaluation step is n-shot learning. It's a task-agnostic dimension, and refers to the number of supervised samples (demonstrations) you provide to the model right before asking it to perform a given task. N-shots are normally provided via a technique called prompting. You'll often see n-shot bundled into the following three categories:

- Zero-shot: refers to evaluation on any tasks without providing any supervised samples to the model at inference time.
- One-shot: similar to few-shot but with n=1, evaluation where one supervised sample is provided to the model at inference time.
- Few-shot: refers to evaluation where a few supervised samples are provided to the model at inference time (e.g. 5 samples provided -> 5-shot).

An example of few-shot learning:

Task: sentiment analysis

Prompt:

Tweet: “I hate it when my phone battery dies.”

Sentiment: Negative

Tweet: “My day has been👍 ”

Sentiment: Positive

Tweet: “This is the link to the article”

Sentiment: Neutral

Tweet: “This new music video was incredible”

Sentiment:

Answer:______

Evaluation typically involves both looking at benchmarking metrics of the above tasks and more manual evaluation by feeding the model with prompts and looking at completions for human assessment. Typically, both NLP Engineers and subject matter experts (SMEs) are involved in the evaluation process and assess the model performance from different angles:

NLP engineers: people with a background in NLP, computational linguistics, prompt engineering, etc., who can probe and assess the model’s semantic and syntactic shortcomings and come up with model failure classes for continuous improvement. A failure class example would be “the LLM does not handle arithmetic with either integers (1, 2, 3, etc.) nor their spelled-out forms: one, two, three.”

Subject matter experts (SMEs): In contrast to the NLP engineers, the SMEs are asked to probe specific classes of LLM output, fixing errors where necessary, and “talking aloud” while doing so. The SMEs are required to explain in a step-by-step fashion the reasoning and logic behind their correct answer versus the incorrect machine-produced answer.

BIAS AND TOXICITY

There are potential risks associated with large-scale, general purpose language models trained on web text. Which is to say: humans have biases, those biases make their way into data, and models that learn from that data can inherit those biases. In addition to perpetuating or exacerbating social stereotypes, you want to ensure your LLM doesn’t memorize and reveal private information.

It’s essential to analyze and document such potential undesirable associations and risks through transparency artifacts such as model cards.

Similar to performance benchmarks, a set of community-developed bias and toxicity benchmarks are available for assessing the potential harm of LLM models. Typical benchmarks include:

Hate speech detection: The ETHOS dataset can help measure the ability of LLM models to identify whether or not certain English statements are racist or sexist

Social bias detection: CrowSPairs is a benchmark aiming to measure intrasentence level biases in 9 categories: gender, religion, race/color, sexual orientation, age, nationality, disability, physical appearance, and socioeconomic status; StereoSet benchmark measure stereotypical bias across 4

Toxic language response: The RealToxicityPrompts dataset helps evaluate if and how models use toxic language.

Dialog safety evaluations: The SaferDialogues benchmark measures how unsafe a model’s response is, stratified across four levels: safe, realistic, unsafe, and adversarial.

To date, most analysis on existing pre-trained models indicate that internet-trained models have internet-scale biases. In addition, pre-trained models generally have a high propensity to generate toxic language, even when provided with a relatively innocuous prompt, and adversarial prompts are trivial to find.

Bias and Toxicity Mitigation

So how do we fix this? Here are a few ways to mitigate biases during and after the pre-training process:

Training set filtering: Here, you want to analyze the elements of your training dataset that show evidence of bias and simply remove them from the training data

Training set modification: This technique doesn’t filter your training data but instead modifies it to reduce bias. This could involve changing certain gendered words (from policeman to policewoman or police officer, for example) to help mitigate bias.

Additionally, you can mitigate bias after pre-training as well:

Prompt engineering: The inputs to the model for each query are modified to steer the model away from bias (more on this later).

Fine-tuning: Take a trained model and retrain it to unlearn biased tendencies.

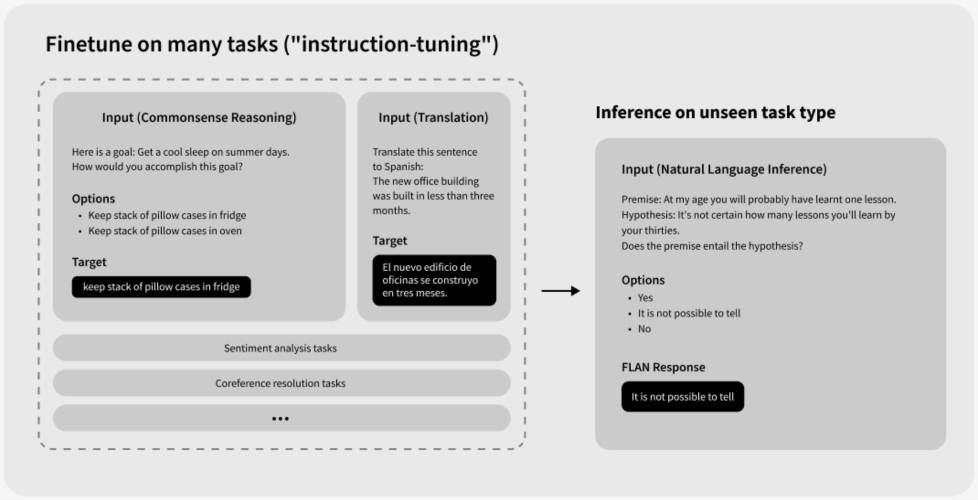
Output steering: Adding a filtering step to the inference procedure to re-weigh output values and steer the output away from biased responses.

INSTRUCTION TUNING

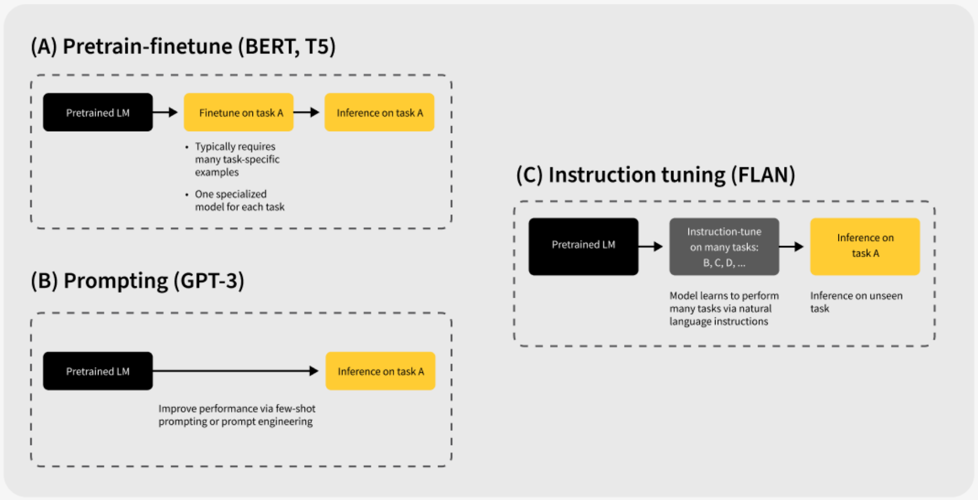
At this point, let's assume we have a pre-trained, general-purpose LLM. If we did our job well, our model can already be used for domain-specific tasks without tuning for few-shot learning and zero-shot learning scenarios. That said, zero-shot learning is in general much worse than its few-shot counterpart in plenty of tasks like reading comprehension, question answering, and natural language inference. One potential reason is that, without few-shot examples, it's harder for models to perform well on prompts that are not similar to the format of the pretraining data.

To solve this issue, we can use instruction tuning. Instruction tuning is a state-of-the-art fine-tuning technique that fine-tunes pre-trained LLMs on a collection of tasks phrased as instructions. It enables pre-trained LLMs to respond better to instructions and reduces the need for few-shot examples at prompting stage (i.e. drastically improves zero-shot performance).

Instruction tuning has gained huge popularity in 2022, given that the technique considerably improves model performance without hurting its ability to generalize. Typically, a pre-trained LLM is tuned on a set of language tasks and evaluated on its ability to perform another set of language tasks unseen at tuning time, proving its generalizability and zero-shot capability. See illustration below:



Finetune on many tasks ("instruction-tuning")
Input (Commonsense Reasoning)
Here is a goal: Get a cool sleep on summer days. How would you accomplish this goal?
Options
Keep stack of pillow cases in fridge
Keep stack of pillow cases in oven
Target
keep stack of pillow cases in fridge
Input (Translation)
Translate this sentence to Spanish: The new office building was built in less than three months.
Target
El nuevo edificio de oficinas se construyo en tres meses.
Sentiment analysis tasks
Coreference resolution tasks
...
Inference on unseen task type
Input (Natural Language Inference)
Premise: At my age you will probably have learnt one lesson.
Hypothesis: It's not certain how many lessons you'll learn by your thirties.
Does the premise entail the hypothesis?
Options
Yes
It is not possible to tell
No
FLAN Response
It is not possible to tell

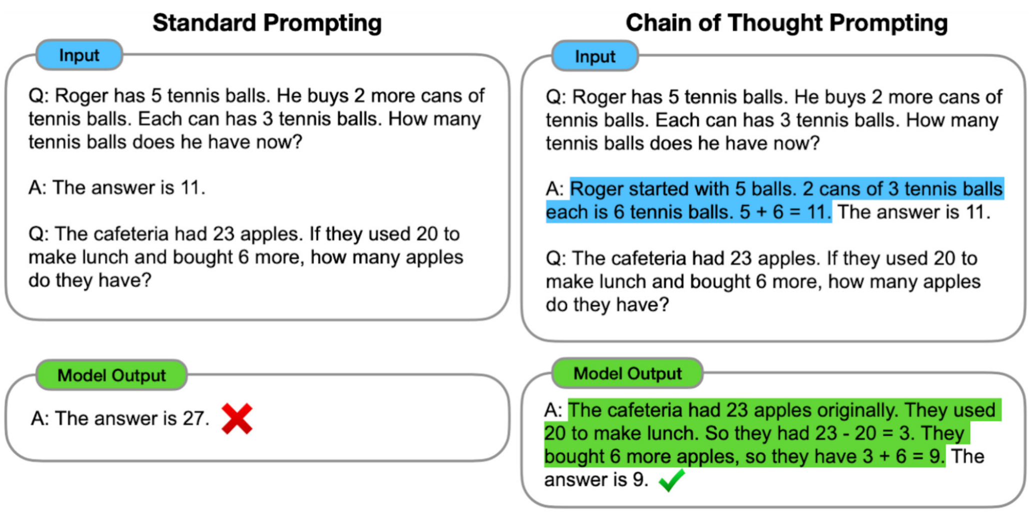


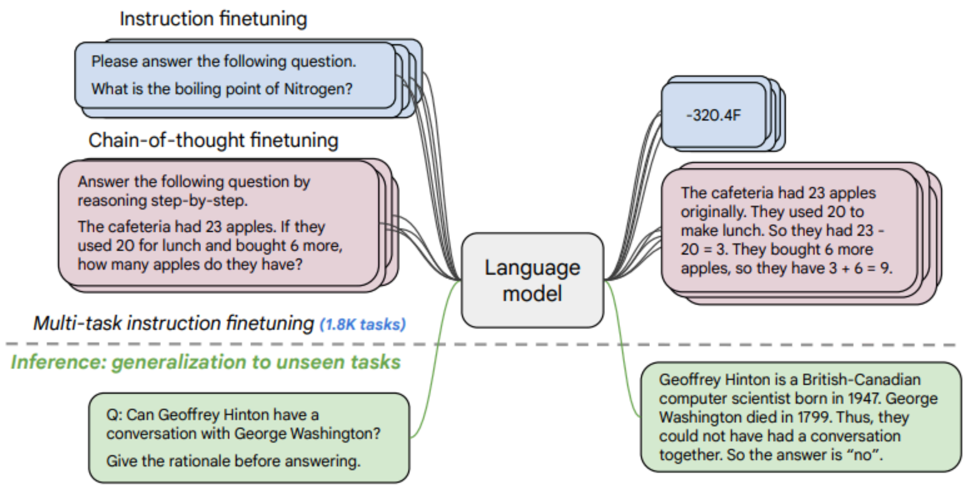
(A) Pretrain-finetune (BERT, T5)
Pretrained LM
Finetune on task A
Inference on task A
Typically requires many task-specific examples
One specialized model for each task
(B) Prompting (GPT-3)
Pretrained LM
Inference on task A
Improve performance via few-shot prompting or prompt engineering
(C) Instruction tuning (FLAN)
Pretrained LM
Instruction-tune on many tasks: B, C, D, ...
Inference on task A
Model learns to perform many tasks via natural language instructions
Inference on unseen task

Comparing instruction tuning with pretrain–finetune and prompting, Finetuned Language Models are Zero-Shot Learners

A few things to keep in mind about instruction tuning:

- Instruction tuning tunes full model parameters as opposed to freezing a part of them in parameter-efficient fine tuning. That means it doesn't bring with it the cost benefits that come with parameter-efficient fine tuning. However, given that instruction tuning produces much more generalizable models compared to parameter-efficient fine tuning, instruction-tuned models can still serve as a general-purpose model serving multiple downstream tasks. It often comes down to whether you have the instruction dataset available and training budget to perform instruction tuning.
- Instruction tuning is universally effective on tasks naturally verbalized as instructions (e.g., NLI, QA, translation), but it is a little trickier for tasks like reasoning. To improve for these tasks, you'll want to include chain-of-thought examples during tuning.


Standard Prompting
Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The answer is 27.
Chain of Thought Prompting
Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.


Instruction finetuning
Please answer the following question.
What is the boiling point of Nitrogen?
Chain-of-thought finetuning
Answer the following question by reasoning step-by-step.
The cafeteria had 23 apples. If they used 20 for lunch and bought 6 more, how many apples do they have?
Language model
-320.4F
The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9.
Multi-task instruction finetuning (1.8K tasks)
Inference: generalization to unseen tasks
Q: Can Geoffrey Hinton have a conversation with George Washington?
Give the rationale before answering.
Geoffrey Hinton is a British-Canadian computer scientist born in 1947. George Washington died in 1799. Thus, they could not have had a conversation together. So the answer is "no".

Instruction tuning both with and without exemplars (i.e., zero-shot and few-shot) and with and without chain-of-thought, enabling generalization across a range of evaluation scenarios from Scaling Instruction-Finetuned Language Models

REINFORCEMENT LEARNING THROUGH HUMAN FEEDBACK (RLHF)

RLHF is an extension of instruction tuning, with more steps added after the instruction tuning step to further incorporate human feedback.

As discussed above, pre-trained LLMs often express unintended behaviors such as making up facts, generating biased or toxic responses, or simply not following user instructions. This is because the objective for many recent large LMs – i.e. predicting the next token on a webpage from the internet – is rather different from the objective "follow the user's instructions safely."

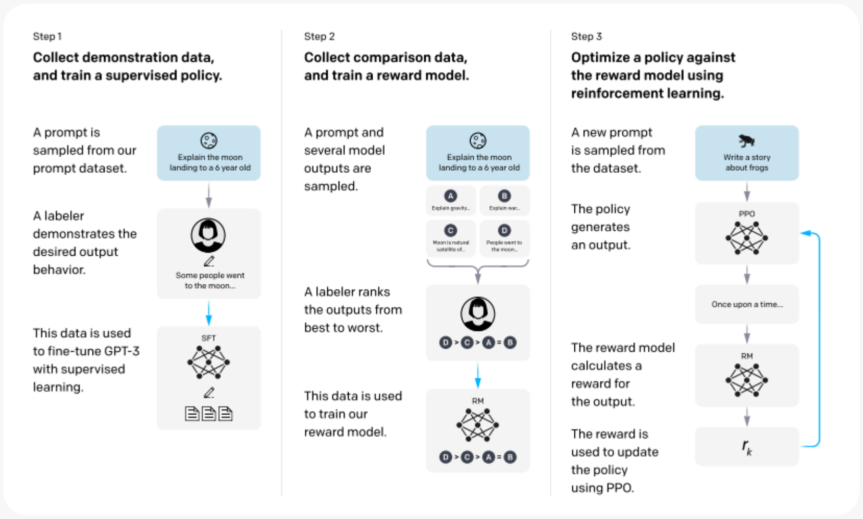
RLHF behaves how its name suggests it would. Here, we incorporate human feedback about a model's outputs given certain prompts. Those opinions about whether the quality of the outputs are then used as additional data points to improve the model's overall performance.

OpenAI has had some recent success here with InstructGPT. It's essentially a pre-trained model GPT-3, fine tuned with RLHF. In fact, their recent ChatGPT model also leverages RLHF on a more advanced GPT model series (referred to as GPT-3.5).

More granularly, RLHF generally works like this:

- Step 1: Instruction tuning – just collect a dataset of labeler demonstrations of the desired model behavior, and use them to fine-tune the pre-trained LLM using supervised learning.
- Step 2: Collect a dataset of comparisons between model outputs, where labelers indicate which output they prefer for a given input. Then, train a reward model to predict the human-preferred output.
- Step 3: Take the trained reward model and optimize a policy against the reward mode using reinforcement learning.

Steps 2 and 3 can be iterated continuously. More comparison data is collected on the current best policy, which is used to train a new reward model and then a new policy. See below for RLHF process demonstration.


Step 1
Collect demonstration data, and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain the moon landing to a 6 year old
A labeler demonstrates the desired output behavior.
Some people went to the moon...
This data is used to fine-tune GPT-3 with supervised learning.
SFT
Step 2
Collect comparison data, and train a reward model.
A prompt and several model outputs are sampled.
Explain the moon landing to a 6 year old
A Explain gravity...
B Explain war...
C Moon is natural satellite of...
D People went to the moon...
A labeler ranks the outputs from best to worst.
D > C > A = B
This data is used to train our reward model.
RM
D > C > A = B
Step 3
Optimize a policy against the reward model using reinforcement learning.
A new prompt is sampled from the dataset.
Write a story about frogs
The policy generates an output.
PPO
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

A diagram illustrating the three steps of our method: (1) supervised fine-tuning (SFT), (2) reward model (RM) training, and (3) reinforcement learning via proximal policy optimization (PPO) on this reward model, Training language models to follow instructions with human feedback

To date, RLHF has shown very promising results with InstructGPT and ChatGPT, bringing improvements in truthfulness and reductions in toxic output generation while having minimal performance regressions compared to the pre-trained GPT.

Note that the RLHF procedure does come with the cost of slightly lower model performance in some downstream tasks - referred to as the alignment tax. Companies like Scale AI, Labelbox, Surge, and Label Studio offer RLHF as a service so you don't have to handle this yourself if you're interested in going down this path. But research has shown promising results using RLHF techniques to minimize the alignment cost to increase its adoption, so it's absolutely worth considering.

Conclusion

Whether it's OpenAI, Cohere, or open-source projects like EleutherAI, cutting-edge large language models are built on Weight & Biases. Our platform enables collaboration across teams performing the complex, expensive work required to train and push these models to production, logging key metrics, versioning datasets, enabling knowledge sharing, sweeping through hyperparameters, and a whole lot more. LLM training is complex and nuanced and having a shared source of truth throughout the lifecycle of a model is vital for avoiding common pitfalls and understanding performance every step of the way.

We'd like to also send a hearty thanks to OpenAI, Deepmind, Meta, and Google Brain. We referenced their research and breakthroughs frequently in this white paper and their contributions to the space are already invaluable.

If you're interested in learning more about how W&B can help, please reach out and we'll schedule some time. And if you have any feedback we'd love to hear those too.

References

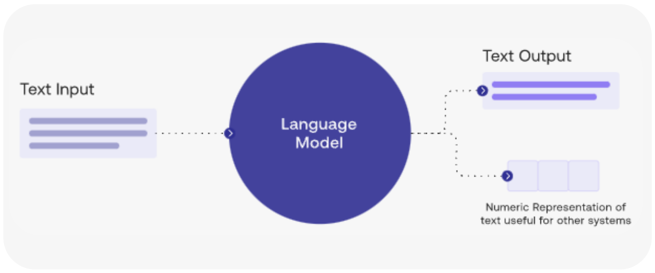
- What Language Model Architecture and pre-training Objective Work Best for Zero-Shot Generalization?
- GPT 3 Paper - Language Models are Few-Shot Learners
- GPT-NeoX-20B: An Open-Source Autoregressive Language Model
- OPT: Open Pre-trained Transformer Language Models
- PaLM: Scaling Language Modeling with Pathways
- Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM
- Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques and Tools
- New Scaling Laws for Large Language Models by DeepMind
- New Scaling Laws for Large Language Models
- Understanding the Difficulty of Training Transformers
- How To Build an Efficient NLP Model
- Emergent Abilities of Large Language Models
- Beyond the Imitation Game benchmark (BIG-bench)
- Talking About Large Language Models
- Galactica: A Large Language Model for Science
- State of AI Report 2022
- Finetuned Language Models are Zero-Shot Learners
- Scaling Instruction-Fine Tuned Language Models
- Training language models to follow instructions with human feedback

Appendix

LLM OVERVIEW

Large pre-trained transformer language models, or simply large language models (LLM), are a recent breakthrough in machine learning that have vastly extended our capabilities in natural language processing (NLP).

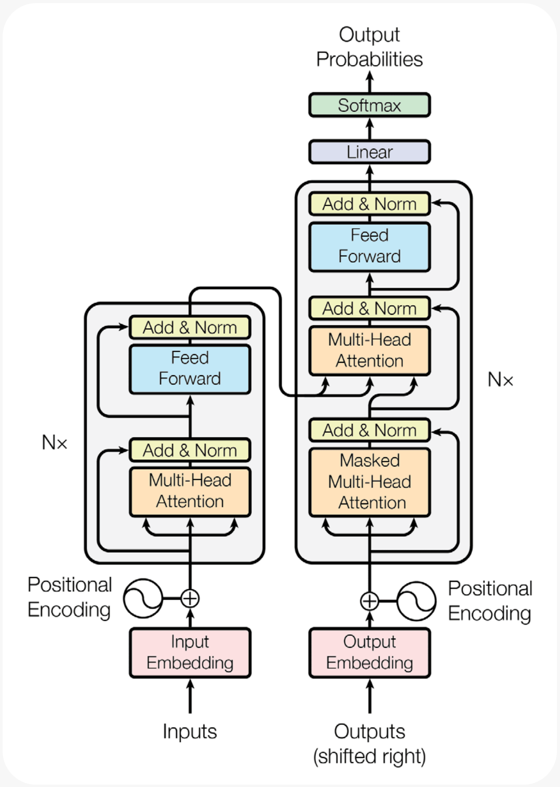
Based on transformer architectures, with as many as hundreds of billions of parameters, and trained on hundreds of terabytes of textual data, recent LLMs such as GPT-3 (OpenAI, 2020), GPT-NeoX (EleutherAI, 2022), PaLM (Google Brain,2022), OPT (Meta AI, 2022), and Macaw (Allen Institute) have demonstrated significant improvements in the ability to perform a wide range of NLP tasks. Here's a brief introduction to the model architecture at play here:



Large language models are computer programs that open new possibilities of text understanding and generation in software systems, CohereAI Large Language Models

TRANSFORMER MODEL ARCHITECTURE

Modern LLMs are based on the transformer architecture. The main architectural unit is a transformer block, which consists of (at a minimum) multi-headed self attention, layer normalization, a dense two-layer feedforward network, and residual connection. A transformer stack is a sequence of such blocks. The below graph shows a typical transformer architecture with an encoder-decoder structure:



The transformer model architecture. Source: Attention Is All You Need

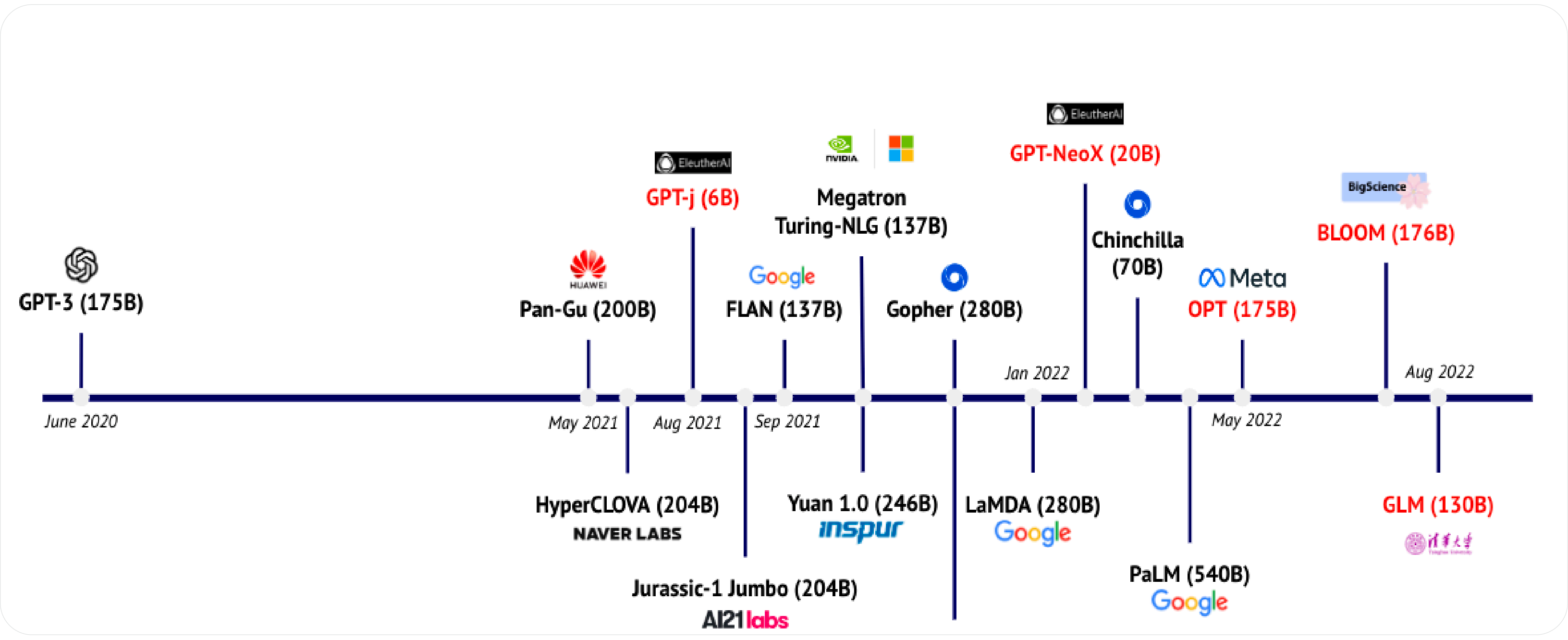
Since the advent of transformers, many architectural variants have been proposed. These can vary by architecture (e.g. decoder-only models, encoder-decoder models), by pre-training objectives (e.g. full language modeling, prefix language modeling, masked language modeling), and other factors.

While the original transformer included a separate encoder that processes input text and a decoder that generates target text (encoder-decoder models), the most popular LLMs like GPT-3, OPT, PaLM, GPT-NeoX are causal decoder-only models trained to autoregressively predict a text sequence. In contrast with this trend, there is some research showing that encoder-decoder models outperform decoder-only LLMs for transfer learning (i.e. where a pre-trained model is finetuned on a single downstream task). For detailed architecture types and comparison, see What Language Model Architecture and pre-training Objective Work Best for Zero-Shot Generalization.

Here are a few of the most popular pre-training architectures:

- **Encoder-decoder models:** As originally proposed, the transformer consists of two stacks: an encoder and a decoder. The encoder is fed the sequence of input tokens and outputs a sequence of vectors of the same length as the input. Then, the decoder autoregressively predicts the target sequence, token by token, conditioned on the output of the encoder. Representative models of this type include T5 and BART.
- **Causal decoder-only models:** these are decoder-only models trained to autoregressively predict a text sequence. “Casual” means that the model is just concerned with the left context (next-step-prediction). Representative examples of this type include GPT-3, GPT-J, GPT-NeoX, OPT, etc.
- **Non-causal decoder-only models:** to allow decoder-only models to build richer non-causal representations of the input text, the attention mask has been modified so that the region of the input sequence corresponding to conditioning information has a non-causal mask (i.e. not restricted to past tokens). Representative PLM models include: UniLM 1-2, ERNIE-M.
- **Masked language models:** these are normally encoder-only models pre-trained with a masked language modeling objective, which predict masked text pieces based on surrounding context. Representative MLM models include BERT and ERNIE.

The graph below shows recent pre-trained LLMs:



Community-driven open sourcing of GPT et al., State of AI Report 2022

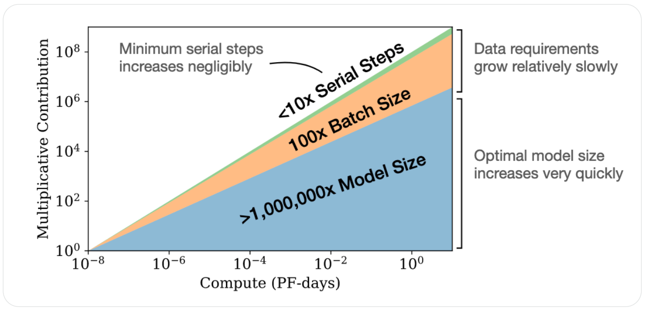
THE ORIGINAL LLM SCALING LAWS

The LLM Scaling Laws (first introduced by OpenAI) tries to answer questions like “Given a certain quantity of compute, how large of a model should I train in order to get the best possible performance?”

The answer is essentially a trade-off between model size and data size. For example, for models at GPT-3 scale, the trade-off is somewhere between:

- (a) training a 20-billion parameter model on 40% of an archive of the Internet, or
- (b) training a 200-billion parameter model on 4% of an archive of the Internet

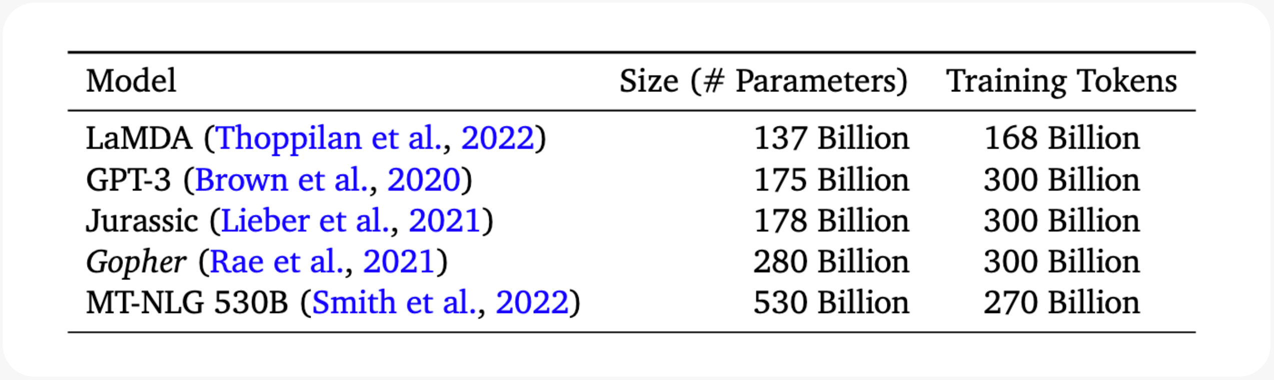
In 2020, OpenAI published the Scaling Laws for Neural Language Models. The paper suggests that increasing model size is more important than increasing data size for compute-optimal training. If you get ten times more compute, you should increase your model size by about five times and double your data size. Another 10x in compute, and model size is twenty-five times bigger and data size is only 4x bigger.



Allocate more compute between increasing model size and training with more data, Scaling Laws for Neural Language Models

Many researchers took this philosophy to heart and focused on how to engineer larger and larger models rather than training comparatively smaller models with more data.

The chart below outlines this trend, showing models built and trained during 2020-2022:



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

Figure x: Current LLMs and their sizes, Training Compute-Optimal Large Language Models