White Paper

Weights & Biases

Best Practices for Fine-Tuning and Prompt Engineering LLMs

A practical guide about making LLMs work for you

contact@wandb.ai

www.wandb.ai

Fine-tuning vs. prompting

Given their general purpose nature, LLMs can already be used for some domain-specific tasks without tuning (think. zero-shot learning, for example). However, to achieve higher accuracy in a specific domain, we'll usually want to perform either fine-tuning or prompt engineering to adapt a general-purpose LLM to downstream tasks.

This whitepaper covers both topics in depth to help you choose the best approach for your current use case. We'll start with a high level overview of both before digging deeper into the research. Let's get started.

Popular fine-tuning methods

Fine-tuning is the process of adapting a pre-trained model to a specific task or domain, such as performing classifications or answering questions.

For LLMs, you might think about tuning them on your product's documentation or fine-tuning an LLM on an internal corpus of customer service messages to help triage the most important requests. During the process, it's common to update either full or partial model parameters through gradient descent. In many cases, this requires much less data to fine-tune the model than to pre-train it (under one percent). Do note that if you're adapting to a domain where there likely isn't much similar data in the original LLM training set (something like a biomedical use case, for example, as patient information and outcomes aren't readily available for LLM training), you may require significantly more data to achieve the performance you're looking for.

Later in this piece, you'll get a full understanding of each flavor of fine-tuning but to start, here are the most common approaches:

Full fine-tuning (or simply "fine-tuning")

All model parameters are updated. Relevant subtypes of fine-tuning include:

- Transfer learning: Update the layers of a pre-trained model to adapt to a new task.
- Knowledge Distillation: Use a new typically smaller model called a "student" to learn representations from a larger "teacher" for a new task.

Parameter-efficient fine-tuning

Only a small fraction of model parameters are updated while other parameters are frozen. Adapter-tuning: Inserts additional task-specific layers between the layers of pre-trained LLMs and only tunes parameters in the adapters.

- LoRA: Adaptors are low rank approximations of the original weight matrices (further reading)
- Prefix-tuning: Prepends task-specific vectors to the model and only tunes parameters in the prefix.

Instruction-tuning

Instruction-tuning: Fine-tuning using supervised samples, specifically a collection of tasks phrased as instructions. All model parameters are updated during instruction-tuning. It substantially improves zero-shot performance on unseen tasks and is considered one of the main breakthroughs in NLP in 2022.

Reinforcement learning through human feedback (RLHF)

Essentially an extension of instruction-tuning, with more steps added after the instruction-tuning step. The focus of the subsequent steps is to ensure models are aligned with human preferences (truthfulness, toxicity reduction, etc.). We covered this extensively in our previous LLM whitepaper. It's worth noting that research around RLHF, though promising, is in its early stages and should be considered experimental in nature as of publication.

Direct preference optimization (DPO)

Essentially an extension of instruction-tuning, with more steps added after the instruction-tuning step. The focus of the subsequent steps is to ensure models are aligned with human preferences (truthfulness, toxicity reduction, etc.). We covered this extensively in our previous LLM whitepaper. It's worth noting that research around RLHF, though promising, is in its early stages and should be considered experimental in nature as of publication.

Prompting

Prompting (a.k.a. in-context learning) involves prepending instructions or providing a few examples to the task input and feeding those to the pre-trained LLM.

Here, instead of fine-tuning the model with hundreds (or thousands) of input texts, the model operates by receiving an instruction along with a handful of task-specific examples (typically fewer than ten). These examples (essentially demonstrating the task we want performed), provide the model with a blueprint for handling such tasks, enabling it to quickly adapt and perform effectively on the new task. In this process, the **pre-trained model weights are frozen**. Instead, the prompt is engineered and optimized to elicit desired output from LLMs. In other words, here, you're tuning via instruction and natural language versus changing the underlying model parameters.

Typical types of prompt optimization

Prompt engineering: The process of designing a prompt template that results in the most effective performance on the downstream task. Prompt engineering can be done either manually or through algorithm searches (no parameters tuned).

Prompt-tuning: Prompt-tuning introduces additional parameters in prompts and optimizes those parameters with supervised samples. In comparison to prompt engineering, which optimizes the prompt in discrete space, prompt-tuning encodes the text prompt and optimizes it in the continuous space

How to choose between fine-tuning and prompting

Each technique is best suited for different scenarios which we'll cover below. Do keep in mind they are not mutually exclusive and could be used in combination. For example, you might first tune a pre-trained LLM via instructions to improve the model's general performance, then use prompting at inference time to further improve performance of the specific downstream task.

Still, there a few key factors to consider in choosing between these techniques, ranging from how you'll actually leverage your LLM to cost considerations:

Downstream task type

For more structured tasks where there is an objective true answer to a question (e.g. classification or translation), fine-tuning on smaller models generally works better than prompting at lower costs.
For creative, generative tasks with a wide range of desired output space (copywriting, code completion, chatGPT-style generation), prompting works much better.

Generalization vs. specialization

Fine-tuning conditions the LLM better than prompting does. Thus, fine-tuning generally improves specialization at the cost of hurting generalization.
If you have only a few specialized tasks, fine-tuning might give you the highest accuracy for each task and/or achieve desired accuracy with a smaller sized model (which reduces cost of inference)
If you have a high number of downstream tasks, or want to use the LLM for future tasks, parameter-efficient fine-tuning and prompt engineering allow you to condition the model for specialization without hurting pre-trained LLM's generalizability.

Scalability vs. simplicity

If you want to **build LLMs and offer them as a service** to a wide range of internal or external customers, taking a large LLM and parameter-efficient fine-tuning or prompt engineering it for each downstream task is more **scalable**.

If you are a single business with a contained set of use cases, taking a smaller language model and fine-tuning it to each downstream task is simpler and cheaper.

Task-specific supervised sample size

If you have **lots of task-specific supervised samples**, fine-tuning is feasible and would give you high task-specific accuracy. If it is hard to collect task-specific samples in advance, or the **input distribution of a task is wide**, fine-tuning becomes infeasible and prompt engineering gives you the best performance for zero-shot or few-short learning. (e.g. for fake news detection use case, it is impossible to collect data for events that haven't happened yet)

Memory and storage cost of inference

To r**educe inference cost for a high number downstream tasks**, you can either (1) use a pre-trained LLM and prefix- or prompt-tune it for each task or (2) use a smaller language model and fully fine-tune one for each task (though this can be cost-prohibitive as it's expensive to adapt full model parameters to all downstream tasks). **Prompt engineering**, although more intuitive, can be expensive, because it involves processing prompts containing one or more training examples for each prediction.

Summary

All told, fine-tuning is **superior for specialized tasks** given a fixed LLM size, since the whole network specializes to solve one problem only. However, if the pre-trained LLM is large enough and the prompt is good enough to "turn on" the right weights in the large network, the performance can be comparable. In a sense, **you can think of prompting as a way to help a generalist network act as a specialist.**

A more pragmatic approach to using LLMs involves breaking tasks into components, prototyping, and evaluating both the overall system (extrinsic) and its components (intrinsic). It often proves to be more efficient and maintainable and provides a balanced way of leveraging the power of LLMs without the high operational costs and maintenance complexity.

For a startup with GenAI ambitions, the general best practices are to following the sequence:

To start off, use commercial LLM APIs (e.g. OpenAI GPT4) out of the box, and productionize your first LLM-application with prompt engineering only. The focus here is to design optimal user experience and build a sustainable and effective data feedback loop (data flywheel).

Next, use the data collected to continuously improve your prompts, your LLM-based system, and your user experience to find the right product-market-fit. Keep in mind you may want to repeat this step if you experience drift or to train your LLM on new, novel data.

Post-product-market fit, you can start thinking of making additional improvements by changing the LLM—maybe switching out commercial APIs and replacing them with your self-hosted, self-trained (or tuned) model to achieve higher accuracy and efficiency.

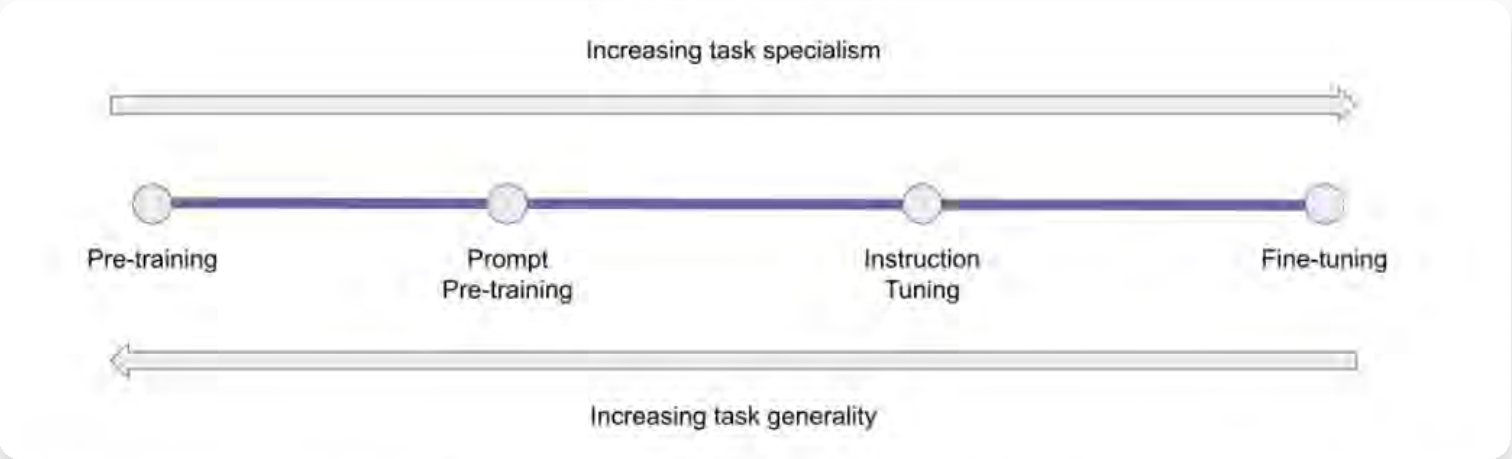

Figure 1: Prompt Pre-training, Galactica: A Large Language Model for Science

Fine-tuning LLMs

While we briefly touched on fine-tuning strategies in our introduction, we'd like to dig deeper and discuss the benefits and limitations of each. We'll start with full fine-tuning (something that is often referred to, simply, as "fine-tuning.")

Full Fine-Tuning
Full fine-tuning is the process of taking a pre-trained model and tuning it to optimize for a specific task. During the fine-tuning process, pre-trained model parameters are updated through gradient descent.

While the premise of fine-tuning might seem straightforward, its execution can be broken down into three core approaches, each with its unique set of procedures and outcomes.

Feature-based approach

In the feature-based method, we employ a pre-trained LLM and apply it to a target dataset, primarily interested in generating output embeddings for the training set. These output embeddings can be utilized as input features to train a classification model, such as a logistic regression model, a random forest, or XGBoost. Although typically employed for embedding-focused models like BERT, we can also extract embeddings from generative GPT-style models.

Fine-tuning I: Updating the output layers

Closely related to the feature-based approach is the practice of fine-tuning the output layers only. Here, we retain the parameters of the pre-trained LLM in a frozen state and only train the newly added output layers. This could be a classification layer for example.

Fine-tuning II: Updating all layers

The third approach updates all layers in the model. This method is more expensive due to the increased number of parameters involved but typically results in superior modeling performance.

For instance, while a BERT base model contains roughly 110 million parameters, the final layer of a BERT base model for binary classification contains a scant 1,500. However, the last two layers of a BERT base model account for 60,000 parameters – around 0.6% of the total model size. Although the performance varies based on the similarity between the target task and the dataset on which the model was pre-trained, fine-tuning all layers almost always enhances the model's performance.

Advantages of full fine-tuning

Fully fine-tuned LLMs might achieve the highest accuracy for the downstream tasks if there are sufficient supervised samples available in the fine tuning process.

Disadvantages of full fine-tuning

- This technique produces a model that is specialized for a single task with an entirely new set of parameter values, and thus can get expensive if there are lots of downstream tasks that all require their own models to be trained. You can compare this to using parameter-efficient fine tuning on the same LLM model size (with an example use case of personalization).
- Fine-tuned models may overfit or not learn stably on smaller datasets, meaning a new large dataset is needed for every new task.
- In comparison to in-context learning or prompt-based methods, fine-tuned models may lack out-of-distribution generalization capabilities.

Parameter-efficient fine-tuning

Parameter-efficient fine-tuning is a lightweight alternative to fine-tuning, which freezes most of the pre-trained parameters and augments the model with small, trainable modules. There are generally two approaches here, though each approach comes with more sub-variations. We'll omit those for this particular whitepaper.

Adapter-tuning

Adapter-tuning inserts additional task-specific layers between the layers of pre-trained language models, called adapters. During tuning, only parameters of the adapters are tuned while the pre-trained model parameters are kept frozen. Low-Rank Adaptation (or LoRA) is a recently popularized fine-tuning method that minimizes the number of tuning parameters, optimizes for tuning memory efficiency, mitigates catastrophic forgetting, while not introducing additional inference latency. LoRA keeps the model pretrained weights frozen while adding pairs of rank-decomposition weight matrices (called update matrices) and only training those newly added weights. See here for more details about the LoRA technique.

Prefix-tuning

Prefix-tuning is another light-weight alternative to fine-tuning. It prepends a sequence of continuous task-specific vectors to the model, which is called a prefix. The prefix consists entirely of free parameters which do not correspond to real tokens. During tuning, only prefix parameters are tuned while the pre-trained model parameters are kept frozen. This method is inspired by prompting (more on this later).

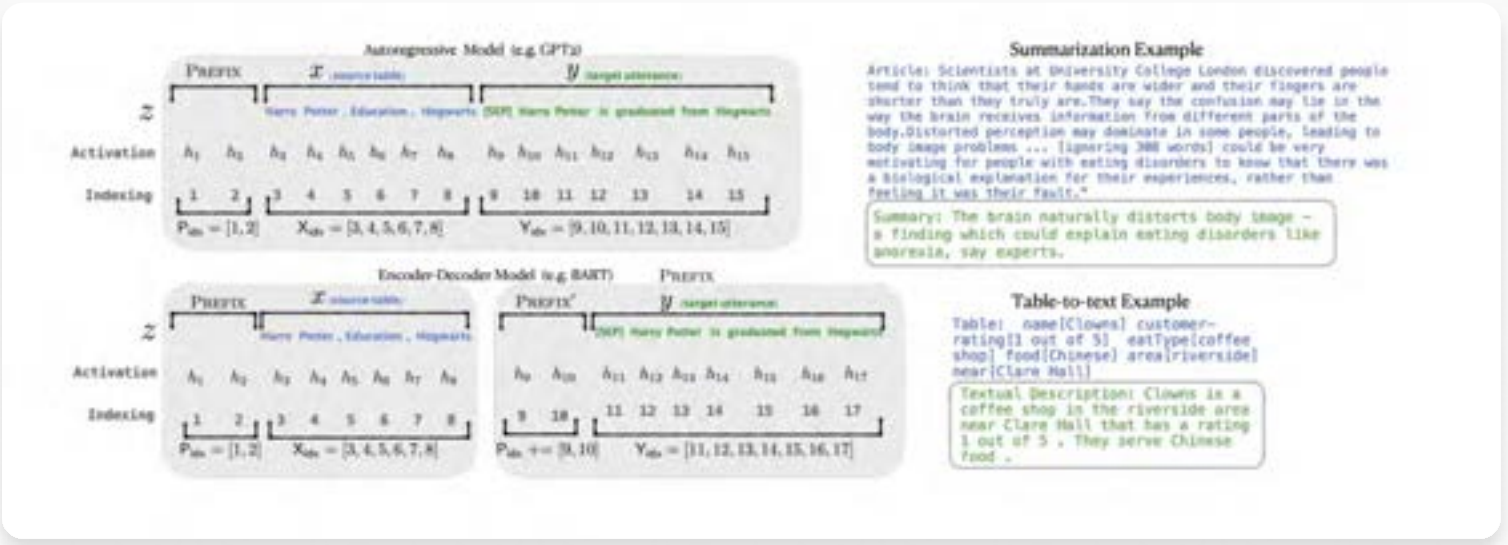

Figure 2: An annotated example of prefix-tuning using an autoregressive LM (top) and an encoder-decoder model (bottom), Prefix-Tuning: Optimizing Continuous Prompts for Generation

Advantages of parameter-efficient fine-tuning

- Cheaper than full fine-tuning given the same pre-trained LLM size. Parameter-efficient fine-tuning touches small, single digit percentages of the model's full parameters while achieving comparable results with full fine-tuning, resulting in 10-1000x reduction in task-specific parameters.
- Parameter-efficient fine-tuning—especially adapter-tuning and prefix-tuning—can obtain comparable performance to full fine-tuning in high data settings (i.e. the number of supervised samples), outperforms fine-tuning in low-data settings, and extrapolates better to examples with topics unseen during training.
- Parameter-efficient fine-tuning yields better accuracy over prompt engineering. Also, it is more expressive than discrete prompt engineering which requires matching the embedding of a real word.
- Prefix-tuning specifically allows for mixed-task batches during inference (queries of different tasks to be in the same batch), and are therefore computationally efficient with GPU sharing (N.B.: adapter-tuning inference cannot be batched).
- Some research shows that prefix-tuning requires considerably fewer parameters compared to adapter-tuning while maintaining comparable performance. The intuition is that prefix-tuning keeps the pre-trained LM intact as much as possible, and therefore exploits the language model more than adapter-tuning.

Disadvantages of parameter-efficient fine-tuning

- Might perform slightly worse than full fine-tuning in some tasks such as summarization in high data settings.
- Needs more supervised samples than prompt engineering.

Prompt-tuning

Prompt-tuning is very similar to prefix-tuning and can be thought of as a simplification to prefix-tuning. We freeze the pre-trained model parameters and only allow an additional set of tunable tokens per downstream task to be prepended to the input text. This “soft prompt” can condense the signal from an often larger supervised sample, allowing the method to outperform few-shot prompts and close the quality gap with model tuning. At the same time, since a single pre-trained model can be recycled for all downstream tasks, this approach scales better when there are a wide range and high volume of tasks to serve.

Given its similarity to prefix-tuning, prompt-tuning has similar advantages and disadvantages with prefix-tuning (refer to the previous section).

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 in a supervised fashion. 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 gained huge popularity in 2022, given that the technique considerably improves model performance without hurting its generalizability. 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.

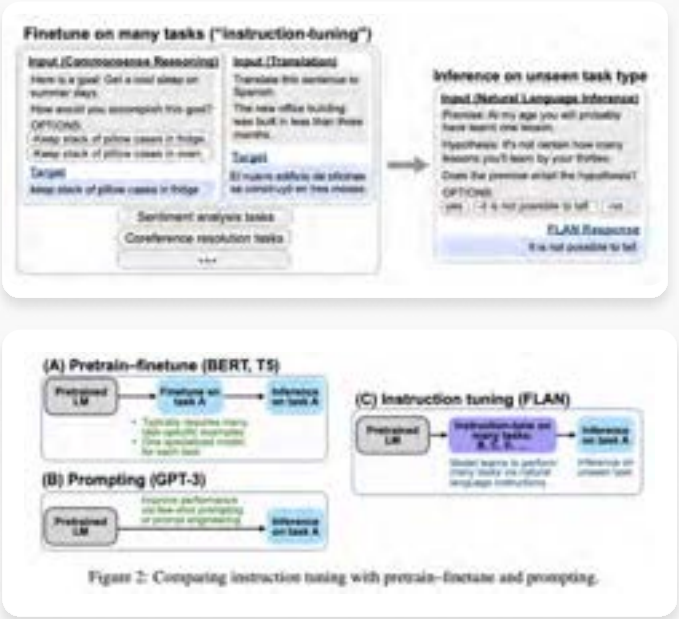

Figure 3: Comparing instruction-tuning with pretrain–fine-tune and prompting, Fine-tuned Language Models are Zero-Shot Learners

Advantages of instruction-tuning:

- As a breakthrough in LLM in 2022, instruction-tuning is the state-of-the-art technique for improving LLMs general performance and helpfulness.
- Instruction-tuning especially improves model performance in zero-shot learning scenarios (i.e. with unseen tasks), and thus instruction-tuned LLMs have much stronger generalizability than typical fine-tuned LLMs, which are built more for task-specific use cases.

Considerations for instruction-tuning:

- This method tunes full model parameters as opposed to freezing a part of them in parameter-efficient fine-tuning. That means it doesn't bring the cost benefit that comes 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. Essentially, it 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, struct-to-text), but it is a little more tricky for tasks directly formulated as language modeling (e.g. reasoning). For instruction-tuning to also perform better in reasoning tasks than traditional fine-tuning, chain-of-thought examples need to be included in the instruction-tuning supervised sample (chain-of-thought examples are demonstrations of how derive the output using step-by-step reasoning, more on this in the later Prompting with LLMs section).

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.

Pre-trained LLMs often express unintended behaviors such as making up facts (a.k.a. hallucinations), generating biased or toxic answers, or simply not following user instructions. This is because the language modeling objective used for many recent LLMs— predicting the next token on a webpage from the internet—is different from the objective "follow the user's instructions helpfully and safely."

Reinforcement learning through human feedback is a fine-tuning technique that helps language models act in accordance with both explicit user intentions (such as following instructions and staying truthful) desired behavior (not exhibiting biased, toxic, or otherwise harmful traits).

The best demonstration of this technique is by OpenAI. They took their pre-trained GPT-3 model, fine-tuned it using RLHF, and derived a more aligned model they called InstructGPT. In addition, ChatGPT is also derived using RLHF on a more advanced GPT model series (referred to as GPT-3.5).

The high-level steps of RLHF:

01

Instruction-tuning. As noted above, here, we collect a dataset of desired model behavior and use those to fine-tune the pre-trained LLM with supervised learning.

02

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.

03

Take the trained reward model and optimize a policy against the reward model 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. The following diagram shows the steps involved in aligning a pre-trained model with human preferences using RLHF.

Wrapping up RLHF

The following diagram shows the steps involved in aligning a pretrained model with human preferences using RLHF.

Figure 4: A diagram illustrating the steps involved in creating the GPT assistant. It highlights the different stages from pre-training to RLHF along with the type of algorithm and the model used. Source: State of GPT, Andrej Karparthy

To date, RLHF has shown very promising results with InstructGPT and ChatGPT, with their 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. But research has shown promising results of using techniques to minimize the alignment cost to increase its adoption.

LLM prompting

As mentioned above, there are plenty of scenarios where prompting may be better or more cost effective for your needs. Broadly, prompting can be implemented in two main ways:

- Take an open-source or self-trained LLM, design or tune prompts, then host inference yourself
- Take a commercial LLM API service (such as GPT-4) and design prompts in their playground UI. (Note: commercial API services largely don't support more sophisticated techniques such as prompt-tuning).

Prompt engineering

In prompt engineering, prompts are typically composed of a task description, one or more instructions, and/or several canonical examples. Typically, an instruction-tuned pre-trained model is used in this stage. The model parameters are frozen and this "freezing" of pre-trained models is highly beneficial, especially as model size continues to increase. Rather than requiring a separate copy of the model for each downstream task, a single generalist model can simultaneously serve many different tasks.

The intuition of prompting is that having a proper context can steer the LLM without changing its parameters. For example, if we want the LLM to generate a word (e.g., Bonaparte), we can prepend its common collocations as context (e.g., Napoleon), and the LLM will assign much higher probability to the desired word.

Manual prompt design vs. automated prompt design

Manual prompt design: the most natural way to create prompts is to manually create intuitive templates based on human introspection. The image below shows a manual prompt design UI for email generation with two demonstrations (two-shot learning). Popular manual prompt design tools are commercial tools such as OpenAI, Anthropic and Cohere, or some open source tools such as Promptsource (by BigScience).

Figure 5: CohereAI Playground screenshot, CohereAI Playground Overview

Automated prompt design

Automated prompt design tries to address the issue that manually designing and experimenting with prompts is more of an art than science. Automated prompt design involves automatically searching for prompt templates described in a discrete space, usually mapped to natural language phrases.

Popular techniques include prompt mining (scraping a large text corpus, e.g. Wikipedia, for strings containing input x and output y, and finding either the middle words or dependency paths between the inputs and outputs), prompt paraphrasing (taking in an existing seed prompt and paraphrasing it into a set of other candidate prompts, then selecting the one that achieves the highest training accuracy on the target task), and gradient-based search (applying a gradient-based search over actual tokens to find short sequences that can trigger the underlying pre-trained LM to generate the desired target prediction). More details can be found in the Pre-train, Prompt, and Predict paper.

Advantages of prompt engineering

- Few-shot prompting leads to a major reduction in the need for task-specific data. In the case where collecting a large number of supervised samples is infeasible, prompt engineering can work with a lot fewer examples.
- It's also applicable to zero-shot settings (no supervised samples provided).
- It's efficient in the sense that no parameter update is needed.
- No catastrophic forgetting, as LLM parameters remain fixed.
- It generally does better in out-of-distribution generalization compared to most other tuning methods.
- Prompt engineering generates human-interpretable prompt templates, and thus is more intuitive where human efforts/SMEs are involved for iteration.

Disadvantages of prompt engineering

- Because prompts are the only method that provide the task specification, heavy engineering is sometimes needed to achieve high accuracy. However, it is extremely hard to find robust, consistent, and interpretable rules for prompt selection.
- Providing many supervised samples can be slow at inference time, especially if you're not using one of the commercial LLM APIs, and thus cannot easily benefit from large task-specific samples if available.
- Typically produces inferior performance compared to fine-tuning especially if large task-specific samples are available, depending on the base LLM you are prompting.
- Processing all prompted input-target pairs every time the model makes an inference could incurs significant compute costs for high # inference scenario.
- LLMs can only condition on a bounded-length context (e.g., 8k tokens for GPT-4), prompting cannot fully exploit training sets longer than the context window.

Chain-of-thought prompting

Chain-of-thought prompting is a prompt composition technique that includes a series of intermediate natural language reasoning steps as demonstrations in the prompt, to unlock LLM's ability to perform reasoning tasks. Specifically, a chain-of-thought prompt consists of triples: {input, chain of thought, output}. The prompt endows LLM with the ability to generate a similar chain of thought reasoning that leads to the final answer for a problem. See below as an example:

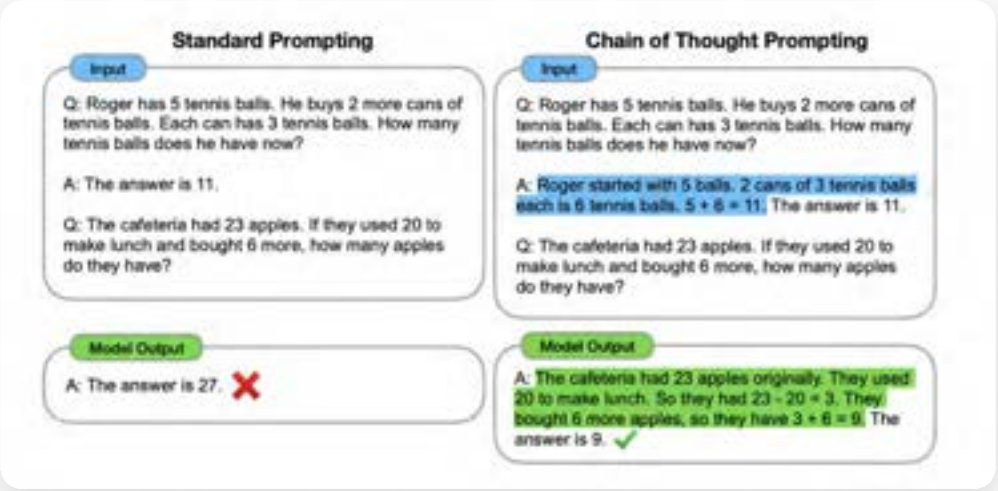

Figure 6: Chain-of-thought prompting enables large language models to tackle complex arithmetic, commonsense, and symbolic reasoning tasks., Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

The intuition is that a chain of thought provides an interpretable window into the behavior of the model, suggesting how it might have arrived at a particular answer and providing opportunities to debug where the reasoning path went wrong. Chain-of-thought reasoning can be used for tasks such as math word problems, common sense reasoning, and symbolic manipulation. Research has shown that chain-of-thought reasoning can be readily elicited in sufficiently large pre-trained LLMs (billions parameters) but work less well for smaller LLMs.

Chain-of-thought prompting varieties

In addition, research has shown that even Zero-shot-CoT (chain-of-thought), whereby simply adding the sentence "Let's think step by step" without any demonstration of examples, can improve model performance by a considerable degree. See below for a Zero-shot-CoT example:

Figure 7: Examples zero-shot CoT with GPT-3, Large Language Models are Zero-Shot Reasoners

Since the advent of Chain-of-Thought, several newer and more sophisticated techniques have been invented to improve the reasoning capability of LLMs, such as Tree of Thoughts, ReAct (Reasoning and Acting), and Reflexion. These all aim to elicit more sophisticated reasoning abilities from LLMs to solve tasks that involve complex reasoning and/or multiple steps, thus expanding the problem space potentially covered by LLM-based systems.

Prompt chaining

Prompt Chaining is a prompt decomposition technique that breaks down an overarching task into a series of highly targeted sub-tasks along a DAG (directed acyclic graph), mapping each to a distinct LLM step, and using the output from one step as an input to the next.

Research has shown that prompt chaining may help complete more complex tasks in a more transparent and controllable way. In addition, some research has depicted a promising new way of software programming using LLM prompt chaining. The approach would enhance software's capability by replacing deterministic code blocks with machine learning model chains.

There are tools such as W&B Prompts, an interactive interface to visually inspect their prompt chaining, to help prompt designers with the LLM chain authoring process. Essentially, Prompts allows users to interrogate the steps their LLM takes to deliver an output, allowing for granular understanding of the process. Among other advantages, prompt chaining makes debugging LLMs much easier.

Figure 8: Example of the W&B Prompts tool

LangChain

There are also sophisticated LLM pipeline building frameworks such as LangChain. It assists the building and maintaining of a pipeline composed of prompted LLM calls (can involve multiple LLM calls to different LLM APIs), external tools (web search, a calculator, a dictionary, etc.), and LLM agents that can route the input traffic to the most appropriate actions for automated planning and problem solving. Designing LLM-based pipelines and systems is an effective way to improve the overall performance of the system, given parts of the pipelines are designed to mitigate the unreliable and non-deterministic output by LLMs and to empower LLMs with externally available tools that are specialized at deterministic tasks (e.g. a calculator). We've written extensively about how W&B and LangChain work together and, if you'd like to take a look, we recommend starting here.

With the ever-improving performance of the latest commercial LLMs such as GPT families, the ecosystem's efforts have shifted from model building to building everything else that is not the model - essentially constructing the aforementioned LLM-based systems to optimize the performance to a certain task.

Conclusion

While we should still expect to see new foundational LLMs in the coming years, it's a safe bet that most organizations will focus less on training their own GPT variants and more on fine-tuning or prompt engineering existing models. Research in this particular area is incredibly fast-paced and we anticipate continued introduction of novel techniques in the coming months and years.

At Weights & Biases, we've been fortunate to serve as the LLMOps tool for companies like OpenAI, Cohere, and countless other leaders in the space. If you'd like to see how we can help you train, fine-tune, or prompt engineer models at your organization, please feel free to email us at contact@wandb.com. We'd love to hear from you.

Additionally, we're constantly publishing new LLM tutorials, experiments, and courses. You can find those first two on our blog in our dedicated LLM section and our free, interactive courses here. If you enjoyed this whitepaper, we'd specifically recommend the course on building LLM powered apps.

Appendix

	Fine-tuning Resource
01	Conditioning, Prompts, and Fine-Tuning
02	Prefix-Tuning: Optimizing Continuous Prompts for Generation
03	Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning
04	Finetuned Language Models are Zero-Shot Learners
05	Scaling Instruction-Fine Tuned Language Models
06	Training language models to follow instructions with human feedback
07	Raise a Child in Large Language Model: Towards Effective and Generalizable Fine-tuning

	Prompting Resource
01	Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods ...
02	The Power of Scale for Parameter-Efficient Prompt Tuning
03	Chain-of-Thought Prompting Elicits Reasoning in Large Language Models
04	Large Language Models are Zero-Shot Reasoners
05	PromptChainer: Chaining Large Language Model Prompts through Visual Programming