

LLM Tuning and Utilization

COMP6211J

Binhang Yuan

Fine-Tuning

How LLM Are Usually Deployed?

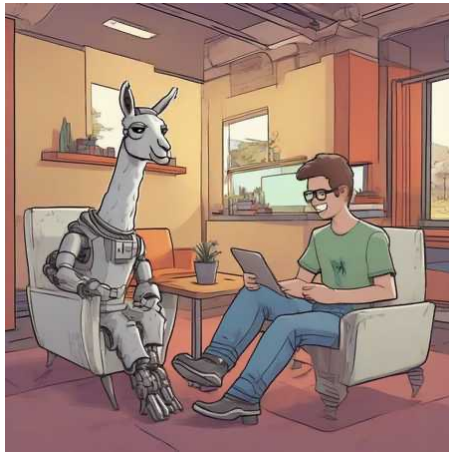
- **Pre-training** is the initial phase of training an LLM, during which it learns from a large, diverse dataset, often consisting of trillions of tokens.
 - The goal is to develop a broad understanding of language, context, and various types of knowledge for the model.
 - Pre-training is usually computationally intensive (thousands of GPUs for weeks) and requires huge amounts of data (trillions of tokens).
- **Fine-tuning** is where you take an already pre-trained model and further train it on a more specific dataset.
 - This dataset is typically smaller and focused on a particular domain or task.
 - The purpose of fine-tuning is to adapt the model to perform better in specific scenarios or on tasks that were not well covered during pre-training.
 - The new knowledge added during fine-tuning enhances the model's performance in specific contexts rather than broadly expanding its general knowledge.

How LLM Are Usually Deployed?



Stage 1: Pretraining

1. Prepare 10TB of text as the training corpus.
2. Use a cluster of thousands of GPUs to train a neural network with the corpus from scratch.
3. Obtain the *base model*.



Stage 2: Fine-tuning

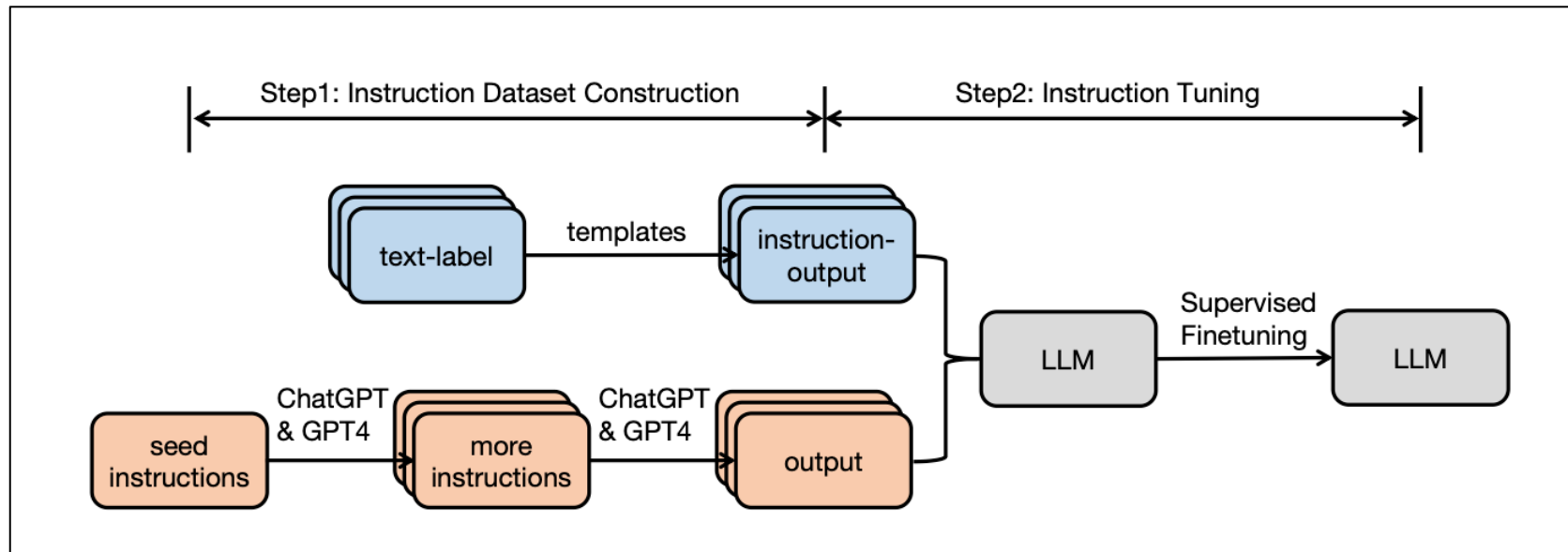
1. Prepare 100K high-quality text as the task-specific training dataset.
2. Use a small-scale GPU cluster to fine-tune the base model with the training dataset.
3. Obtain the *domain-specific model*.
4. Run the domain-specific model for evaluation.
5. Repeat until we are happy with the results.

Instruction Tuning

- Fine-tune the LLM using *(Instruction, output)* pairs:
 - *Instruction*: denotes the human instruction for the model;
 - *Output*: denotes the desired output that follows the instruction.
- Finetuning an LLM on the instruction dataset bridges the gap between the next-word prediction objective of LLMs and the users' objective of instruction following;
- Instruction tuning allows for a more controllable and predictable model behaviour compared to standard LLMs.
 - The instructions serve to constrain the model's outputs to align with the desired response characteristics or domain knowledge, providing a channel for humans to intervene with the model's behaviours.
- Instruction tuning can help LLMs rapidly adapt to a specific domain without extensive retraining or architectural changes.

Instruction Tuning

- Two methods to construct the instruction datasets:
 - Data integration from annotated natural language datasets: (instruction, output) pairs are collected from existing annotated natural language datasets by using templates to transform text-label pairs to (instruction, output) pairs.
 - Generate outputs using more advanced LLMs: employ LLMs such as GPT-3.5-Turbo or GPT4 to gather the desired outputs given the instructions instead of manually collecting the outputs.



Parameter Efficient Fine-Tuning

Parameter Efficient Fine-Tuning

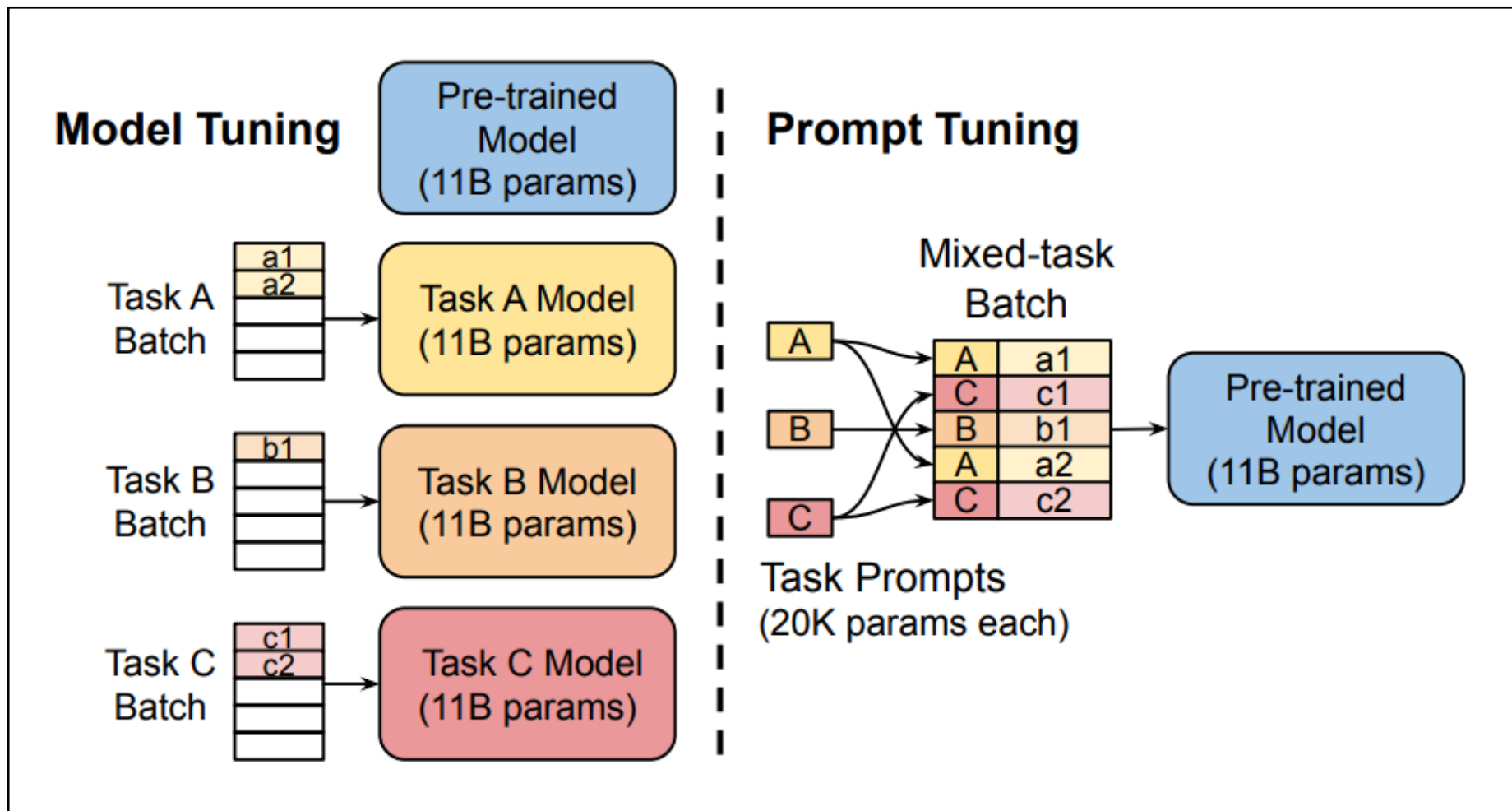
- Parameter efficient fine-tuning (PEFT): Rather than finetuning the entire model, we finetune only small amounts of weights.
- Goal: achieve performance on a downstream task that is comparable to fine-tuning all parameters.
- Some approaches:
 - Soft prompts: include some learnable tensors combined with the original input embeddings that can be optimized for a dataset.
 - Prompt tuning;
 - Prefix tuning;
 - Frozen layer/Subset fine-tuning: pick a subset of the parameters, fine-tune only those layers, and freeze the rest of the layers.
 - Adapters: add additional layers that have few parameters and tune only the parameters of those layers, keeping all others fixed.
 - Low-rank adaption (LoRA): learn a low rank approximation of the weight matrices.

Prompt Tuning

- In hard prompts, prompts are added to the input as a series of tokens:
 - The model parameters are fixed;
 - The prompt embedding are also fixed by the model parameters.
 - Prompt & original input tokens are mapped to embedding by fixed model parameters:

$$\{p_1, p_2, \dots, p_{L_p}, x_1, x_2, \dots, x_L\} \rightarrow X_{emb} \in \mathbb{R}^{(L_p+L) \times D}$$
 - L_p is the prompt sequence length; L is the input sequence length; D is the model dim.
- The key idea behind prompt tuning is that prompt tokens have their own parameters that are updated independently.
 - Keep the pretrained model's parameters frozen, and only update the gradients of the prompt token embeddings.
 - Concatenate the learned prompt parameters to the original input: $\{x_1, x_2, \dots, x_L\} \rightarrow P_{emb} + X_{emb}, P_{emb} \in \mathbb{R}^{P \times D}, X_{emb} \in \mathbb{R}^{L \times D};$
 - P is the prompt task name encoded sequence length; L is the input sequence length; D is the model dim.
 - $P_{emb} \in \mathbb{R}^{P \times D}$ is trained and have different values for different tasks.

Prompt Tuning



Only train and store a significantly smaller set of task-specific prompt parameters.

<https://arxiv.org/pdf/2104.08691.pdf>

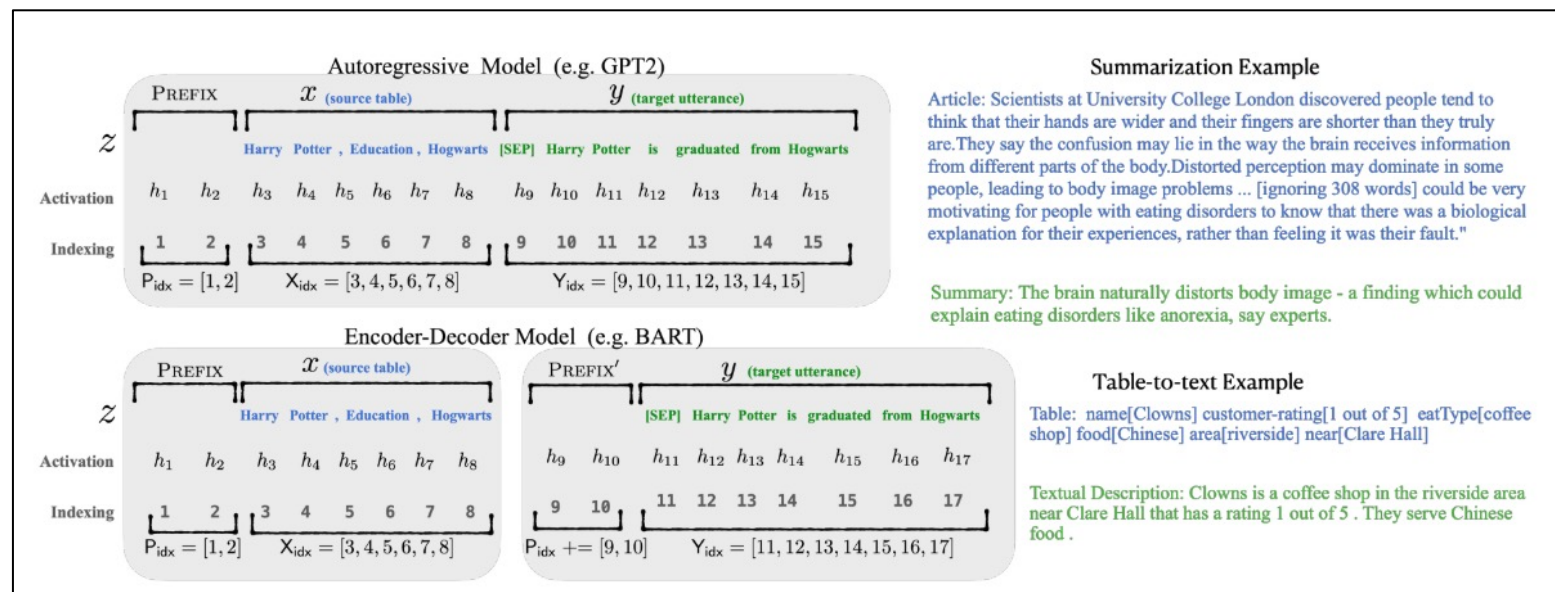
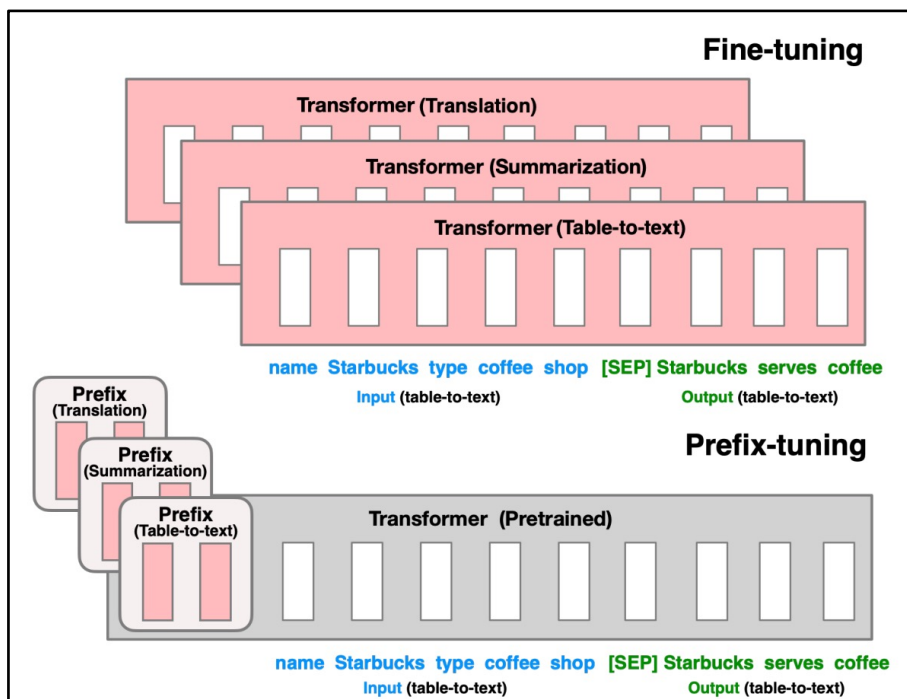
Prefix Tuning

- Prefix tuning was designed for natural language generation (NLG) tasks on GPT models.
- Prefix tuning also prepends a sequence of task-specific vectors to the input that can be trained and updated while keeping the rest of the pretrained model's parameters frozen.
- The main difference is that the prefix parameters are inserted in *all* of the model layers, whereas prompt tuning only adds the prompt parameters to the model input embeddings.
- The prefix parameters are also optimized by a separate feed-forward network (FFN) instead of training directly on the soft prompts because it causes instability and hurts performance.
- The FFN is discarded after updating the soft prompts.

Prefix Tuning



RELAXED
SYSTEM LAB



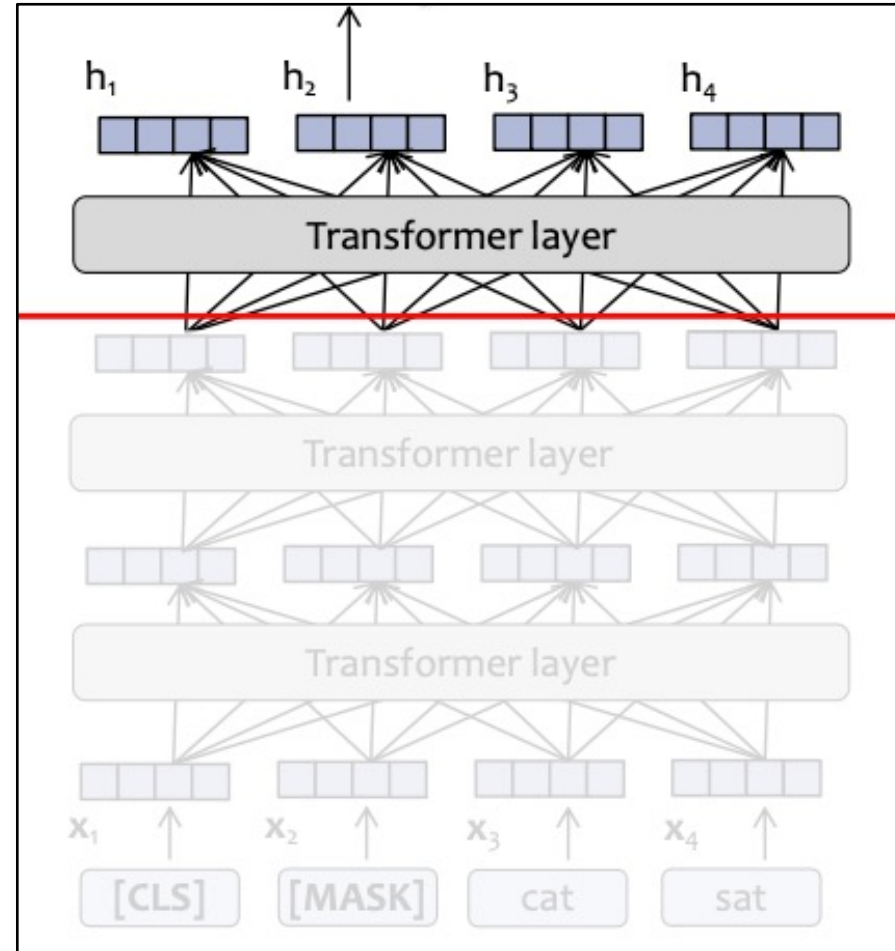
Optimize the prefix parameters for each task.
<https://aclanthology.org/2021.acl-long.353.pdf>

Subset Fine-Tuning

- Some interpretations from NLP research:
 - Earlier layers of the transformer tend to capture linguistic phenomena and basic language understanding;
 - Later layers are where the task-specific learning happens.
- We should be able to learn new tasks by freezing the earlier layers and tuning the later ones.
- This can be a simple baseline for PEFT.

Subset Fine-Tuning

- Keep all parameters fixed except for the top K layers.
- Gradients only need to flow through top K layers instead of all of layers.
- Reduce computation load.
- Reduce memory usage.

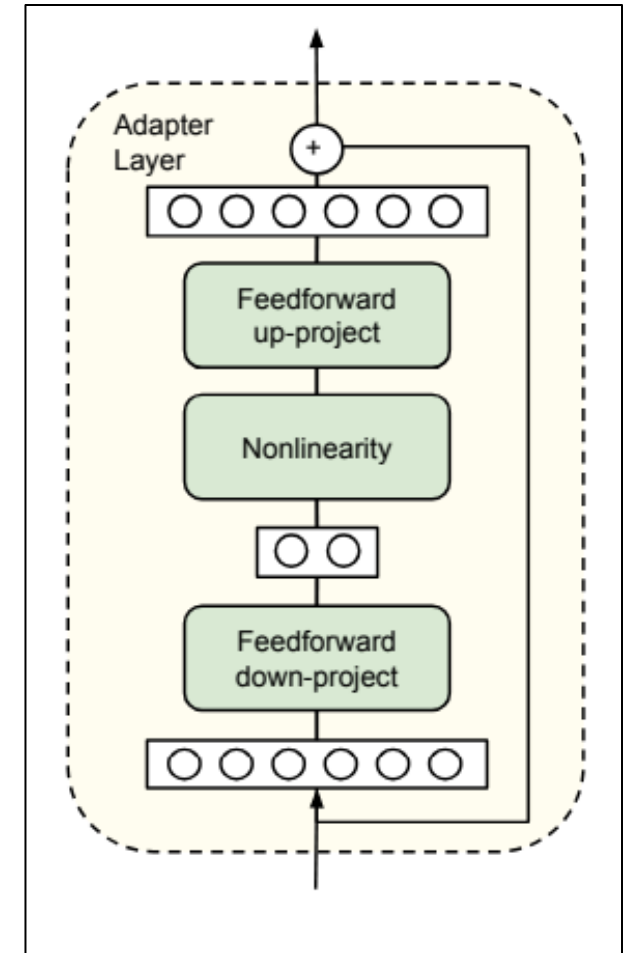


The gradient does not backpropagate to lower layers.

Adapter

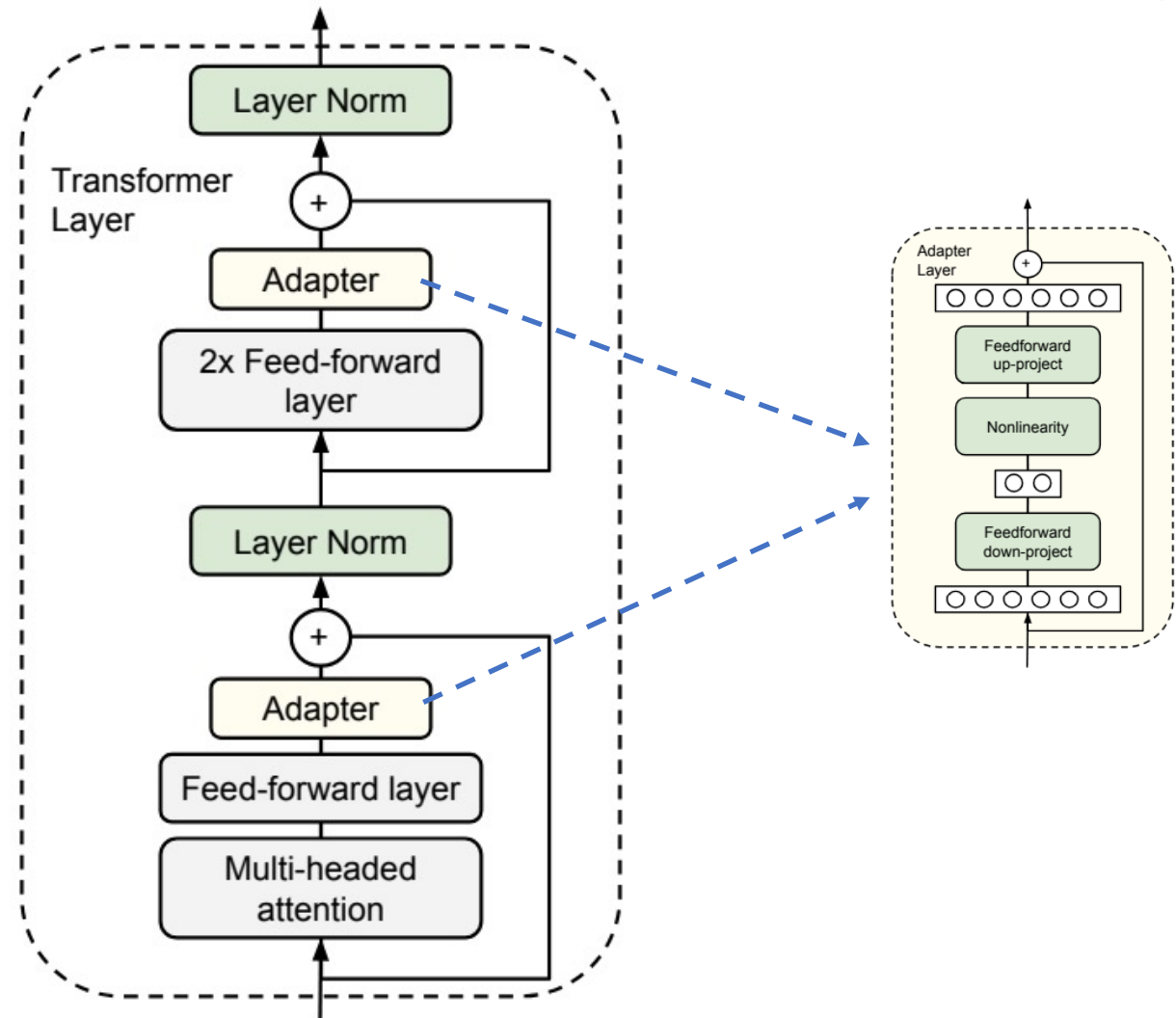


- Adapters are new modules are added between layers of a pre-trained network.
 - The original model weights are fixed;
 - Just the adapter modules are tuned.
- An adapter layer is simply a feed- forward neural network with one hidden layer, and a residual connection.
- Suppose the original LLM has a model dimension of D :
 - For the down-project weight matrix: $W_A \in \mathbb{R}^{D \times R}$;
 - For the up-project weight matrix: $W_B \in \mathbb{R}^{R \times D}$;
 - We have $R \ll D$.



Adapter

- Given $R \ll D$, in practice the adapter layers contain only 0.5% – 8% of the total parameters.
- When added to a deep neural network (e.g. transformer) all the other parameters of the pretrained model are kept fixed, and only the adapter layer parameters are fine-tuned.



Low-Rank Adaptation (LoRA)

Low-Rank Adaption

- Central idea:
 - “How can we re-parameterize the model into something more efficient to train?”
- Finetuning has a low intrinsic dimension, that is, the minimum number of parameters needed to be modified to reach satisfactory performance is not very large.
- This means we can re-parameterize a subset of the original model parameters with low-dimensional proxy parameters, and just optimize the proxy.

Intrinsic Dimension

- An objective function's intrinsic dimension measures the minimum number of parameters needed to reach a satisfactory solution to the objective.
- Can also be thought of as the lowest dimensional subspace in which one can optimize the original objective function to within a certain level of approximation error.
- Details in this paper: <https://arxiv.org/abs/2012.13255>
 - Suppose we have model parameters $\theta^{(D)} \in \mathbb{R}^D$, D is the number of parameters;
 - Instead of optimizing $\theta^{(D)}$, we could optimize a smaller set of parameters $\theta^{(d)} \in \mathbb{R}^d$, where $d \ll D$.
 - This done through clever factorization:
 - $\theta^{(D)} = \theta_0^{(D)} + P(\theta^{(d)})$; where $P: \mathbb{R}^d \rightarrow \mathbb{R}^D$
 - P is typically a linear projection: $\theta^{(D)} = \theta_0^{(D)} + \theta^{(d)}M$.
 - Intuitively, we do an arbitrary random projection onto a much smaller space; usually, a linear projection, we then solve the optimization problem in that smaller subspace. If we reach a satisfactory solution, we say the dimensionality of that subspace is the intrinsic dimension.

Low-Rank Adaption

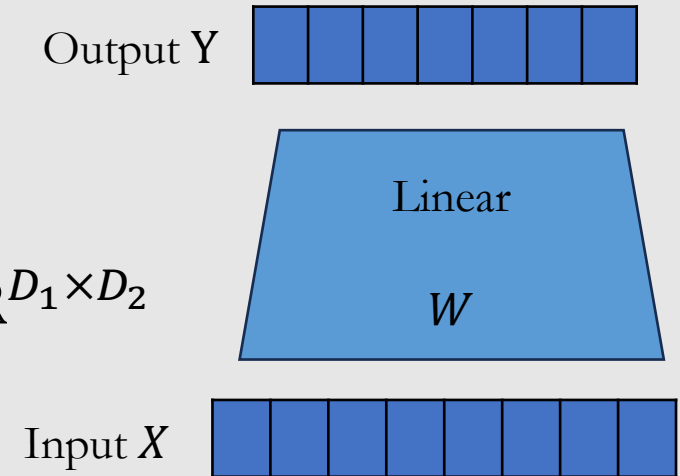
- Full paper: <https://arxiv.org/pdf/2106.09685>.
- Intuition: It's not just the model weights that are low rank, updates to the model weights are also low-rank.
- LoRA freezes the pre-trained model weights and injects trainable rank decomposition matrices into some or all layers.

LoRA Key Idea

- Keep the original pre-trained parameters W fixed during fine-tuning;
- Learn an additive modification to those parameters ΔW ;
- Define ΔW as a low-rank decomposition: $\Delta W = AB$.

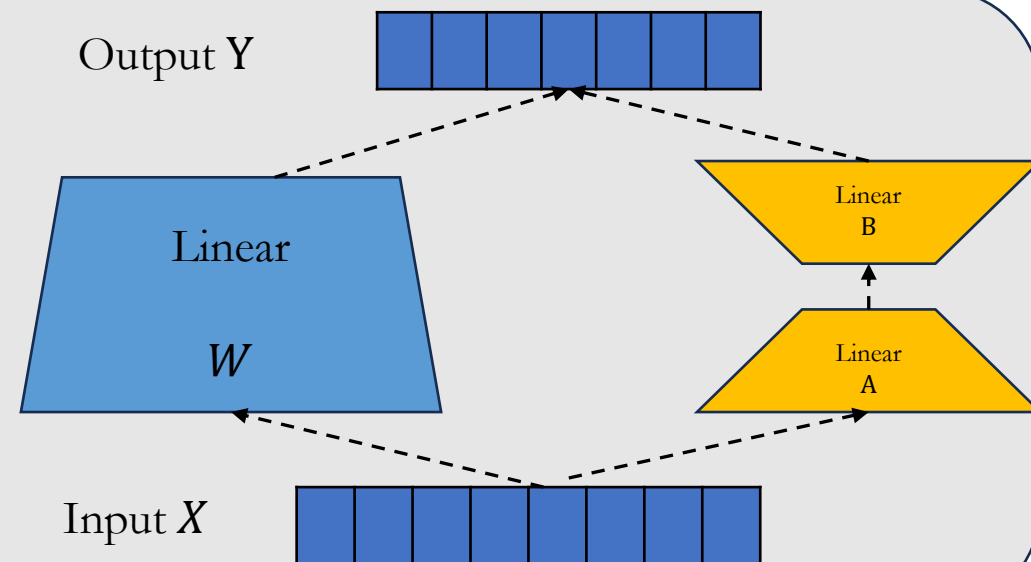
Standard Linear Layer

- $Y = XW$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, W \in \mathbb{R}^{D_1 \times D_2}$



LoRA Linear Layer

- $Y = XW + XAB = X(W + AB)$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, W \in \mathbb{R}^{D_1 \times D_2}$
- $A \in \mathbb{R}^{D_1 \times R}, B \in \mathbb{R}^{R \times D_2}$

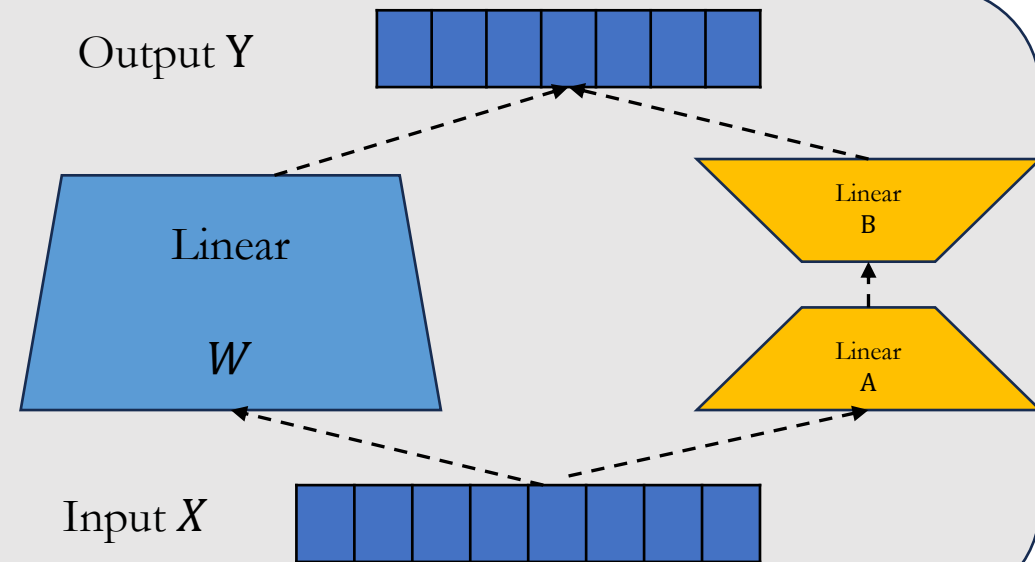


LoRA Initialization

- We initialize the trainable parameters:
 - $A_{ij} \sim \mathcal{N}(0, \sigma^2)$ $B_{ij} = 0$
- This ensures that, at the start of fine-tuning, the parameters have their pre-trained values:
 - $\Delta W = AB = 0$

LoRA Linear Layer

- $Y = XW + XAB = X(W + AB)$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, W \in \mathbb{R}^{D_1 \times D_2}$
- $A \in \mathbb{R}^{D_1 \times R}, B \in \mathbb{R}^{R \times D_2}$

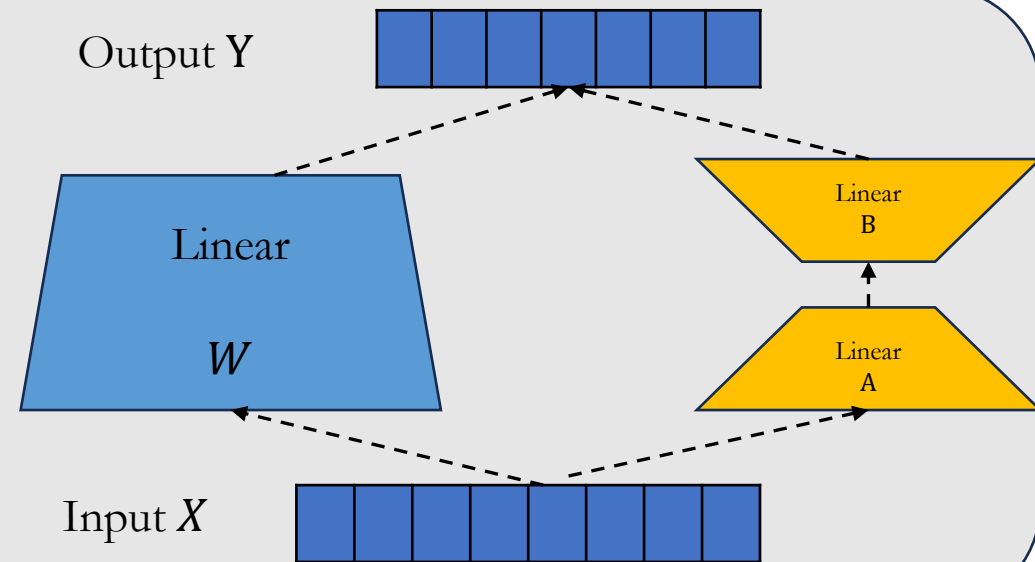


LoRA Hot Swapping Parameters

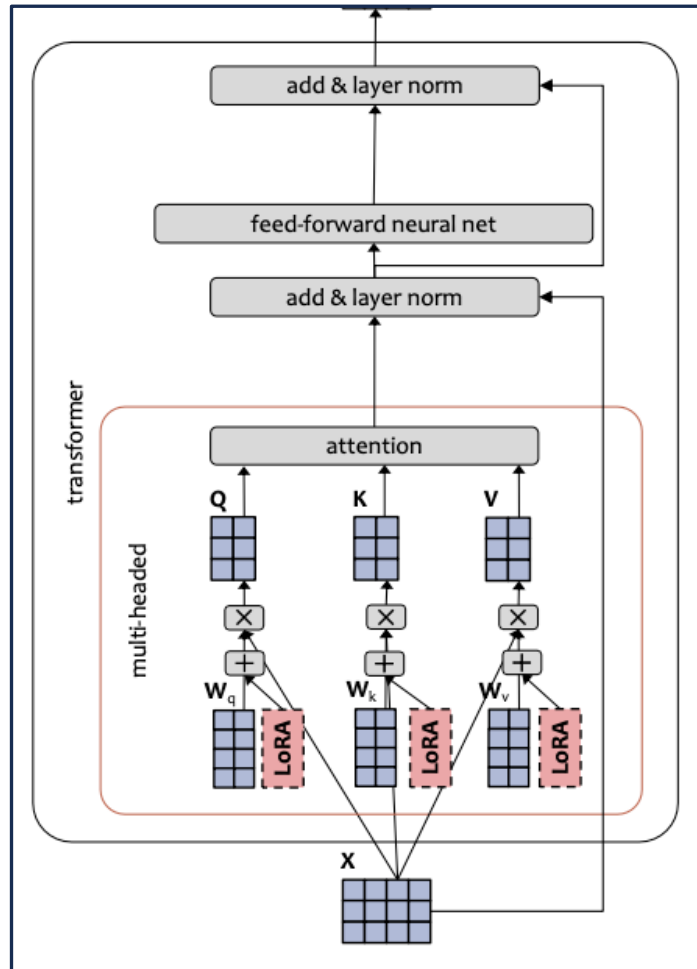
- W and AB the same dimension, so we can swap the LoRA parameters in and out of a Standard Linear Layer.
- To include LoRA:
 - $W' \leftarrow W + AB$
- To remove LoRA:
 - $W \leftarrow W' - AB$

LoRA Linear Layer

- $Y = XW + XAB = X(W + AB)$
- $X \in \mathbb{R}^{D_1}, Y \in \mathbb{R}^{D_2}, W \in \mathbb{R}^{D_1 \times D_2}$
- $A \in \mathbb{R}^{D_1 \times R}, B \in \mathbb{R}^{R \times D_2}$



LoRA for Transformers



- LoRA linear layers could replace every linear layer in the Transformer layer;
- But the original paper only applies LoRA to the attention weights:
 - $Q = \text{LoRALinear}(X, W_q, A_q, B_q)$
 - $K = \text{LoRALinear}(X, W_k, A_k, B_k)$
 - $V = \text{LoRALinear}(X, W_v, A_v, B_v)$
- Some further research found that most efficient to include LoRA only on the query and key linear layers.

LoRA Results

- Some empirical takeaways:
 - Applied to GPT-3, LoRA achieves performance almost as good as full parameter fine-tuning, but with far fewer parameters.
 - On some tasks, it even outperforms full fine-tuning.
 - For some datasets, a rank of $R = 1$ is sufficient.
 - LoRA performs well when the dataset is large or small.

Prompt Engineering

What Are Prompts?

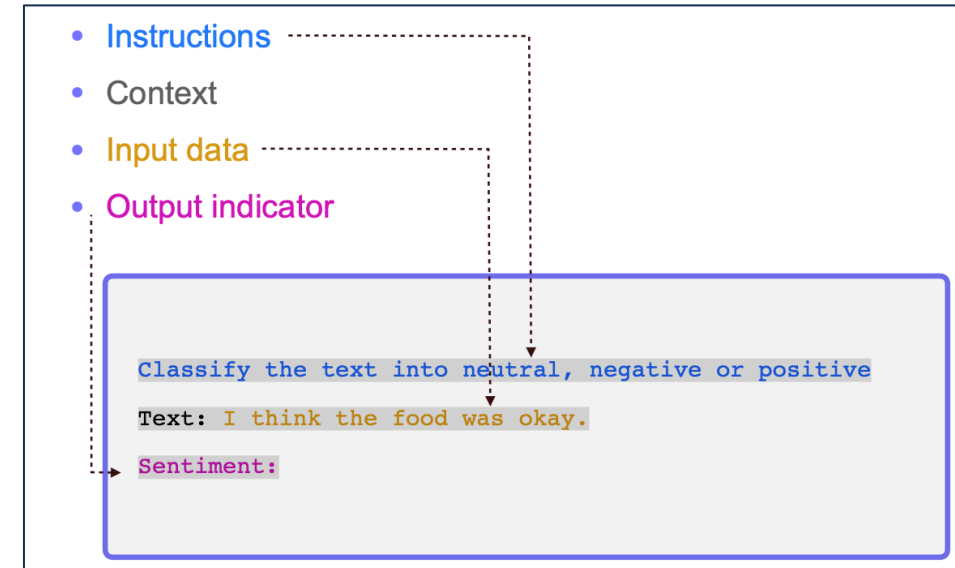
- **Prompts** involve instructions and context passed to a language model to accomplish a desired task.
- **Prompt engineering** is the practice of developing and optimizing prompts to efficiently use large language models (LLMs) for a variety of applications.
- Prompt engineering is a useful skill for AI engineers and researchers to improve and efficiently use language models.

Why Prompt Engineering?

- Prompt engineering is a relatively new discipline for developing and optimizing prompts to efficiently use language models (LMs) for a wide variety of applications and research topics.
- Prompt engineering skills help to better understand the capabilities and limitations of large language models (LLMs).
- Researchers use prompt engineering to improve the capacity of LLMs on a wide range of common and complex tasks such as question answering and arithmetic reasoning.
- Developers use prompt engineering to design robust and effective prompting techniques that interface with LLMs and other tools.

Elements of a Prompt

- A prompt is composed with the following components:
 - **Instruction:** a specific task or instruction you want the model to perform;
 - **Context:** external information or additional context that can steer the model to better responses;
 - **Input data:** the input or question that we are interested to find a response for;
 - **Output indicator:** the type or format of the output.
- You do not need all the four elements for a prompt and the format depends on the task at hand.



Rules of Thumb from OpenAI

Rule 1

- “Use the latest model.”
 - For best results, we generally recommend using the latest, most capable models. Newer models tend to be easier to prompt engineer.

GPTs

Discover and create custom versions of ChatGPT that combine instructions, extra knowledge, and any combination of skills.

Top Picks

DALL·E

Writing

Productivity

Research & Analysis


Programming

Education


Lifestyle

Featured


Curated top picks from this week




Instant Website
[Multipage]
Generates Functional Multipage Websites [in BETA]. Our mission is to simplify the creation of a...
By Max & Kirill Dubovitsky



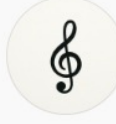
AskYourPDF Research Assistant
Free Chat Unlimited PDFs, Access 400M+ Papers (PubMed, Nature, Arxiv, etc), Analyse PDF (Unlimite...
By askyourpdf.com




Diagrams & Data:
Research, Analyze,...
Complex Visualizations (Diagram & Charts), Data Analysis & Reseach. For Coders: Visualize Databases,...
By Max & Kirill Dubovitsky



ChatPRD
An on-demand Chief Product Officer that drafts and improves your PRDs, while coaching you to...
By Claire V Lawless



Music Teacher
Regular ChatGPT isn't great at music theory and relative scales, so I trained Music Teacher to be an...
By gryphonedm.com



UX Design Mentor
I provide specific UX or Product Design feedback.
By community builder

Rule 2

- “Put instructions at the beginning of the prompt and use ### or """ to separate the instruction and context”

- Less effective :

Summarize the text below as a bullet point list of the most important points.

{text input here}

- Better :

Summarize the text below as a bullet point list of the most important points.

Text: """

{text input here}

"""

Rule 3

- “Be specific, descriptive and as detailed as possible about the desired context, outcome, length, format, style, etc”

- Less effective :

Write a poem about OpenAI.

- Better :

Write a short inspiring poem about OpenAI, focusing on the recent DALL-E product launch (DALL-E is a text to image ML model) in the style of a {famous poet}

Rule 4

- “Articulate the desired output format through examples”
 - Show, and tell - the models respond better when shown specific format requirements. This also makes it easier to programmatically parse out multiple outputs reliably.

- Less effective :

Extract the entities mentioned in the text below. Extract the following 4 entity types: company names, people names, specific topics and themes.

Text: {text}

- Better :

Extract the important entities mentioned in the text below. First extract all company names, then extract all people names, then extract specific topics which fit the content and finally extract general overarching themes

Desired format:

Company names:

<comma_separated_list_of_company_names>

People names: - | -

Specific topics: - | -

General themes: - | -

Text: {text}

Rule 5


- “Start with zero-shot, then few-shot, neither of them worked, then fine-tune.”

-  Zero-shot:

Extract keywords from the below text.

Text: {text}

Keywords:

-  Fine-tune: to be discussed later in the lecture.

-  Few-shot - provide a couple of examples:

Extract keywords from the corresponding texts below.

Text 1: Stripe provides APIs that web developers can use to integrate payment processing into their websites and mobile applications.

Keywords 1: Stripe, payment processing, APIs, web developers, websites, mobile applications

##

Text 2: OpenAI has trained cutting-edge language models that are very good at understanding and generating text. Our API provides access to these models and can be used to solve virtually any task that involves processing language.

Keywords 2: OpenAI, language models, text processing, API.

##

Text 3: {text}

Keywords 3:

Rule 6

- “Reduce “fluffy” and imprecise descriptions.”


- Less effective :

The description for this product should be fairly short, a few sentences only, and not too much more.

- Better :

Use a 3 to 5 sentence paragraph to describe this product.

Rule 7

- “Instead of just saying what not to do, say what to do instead.”
- Less effective :

The following is a conversation between an Agent and a Customer. DO NOT ASK USERNAME OR PASSWORD. DO NOT REPEAT.

Customer: I can't log in to my account.

Agent:

- Better :

The following is a conversation between an Agent and a Customer. The agent will attempt to diagnose the problem and suggest a solution, whilst refraining from asking any questions related to PII. Instead of asking for PII, such as username or password, refer the user to the help article www.samplewebsite.com/help/faq

Customer: I can't log in to my account.

Agent:

Rule 8

- “Code Generation Specific - Use “leading words” to nudge the model toward a particular pattern.”
 - In this code example below, adding “import” hints to the model that it should start writing in Python. (Similarly “SELECT” is a good hint for the start of a SQL statement.)

- Less effective :

```
# Write a simple python function that  
# 1. Ask me for a number in mile  
# 2. It converts miles to kilometers
```

- Better :


```
# Write a simple python function that  
# 1. Ask me for a number in mile  
# 2. It converts miles to kilometers  
  
import
```

Advanced Prompt Engineering Techniques

Baseline: Zero-Shot Prompting

- Large-scale training makes LLMs capable of performing some tasks in a "zero-shot" manner.
- Zero-shot prompting means that the prompt used to interact with the model won't contain examples or demonstrations.
- The zero-shot prompt directly instructs the model to perform a task without any additional examples to steer it.

Few-Shot Prompting

- *Few-shot prompting* allows us to provide exemplars in prompts to steer the model towards better performance.
 - Few-shot prompting can be used as a technique to enable in-context learning where we provide demonstrations in the prompt to steer the model to better performance.
 - The demonstrations serve as conditioning for subsequent examples where we would like the model to generate a response.
- Few-Shot Prompting Input:
- Output :

Great product, 10/10: positive
Didn't work very well: negative
Super helpful, worth it: positive
It doesnt work!:

negative

Chain-of-Thought Prompting

- *Chain-of-thought (CoT)* prompting enables complex reasoning capabilities through intermediate reasoning steps.
 - This is very useful for tasks that require reasoning;
 - It can be combined with both few-shot prompting and zero-shot prompting.

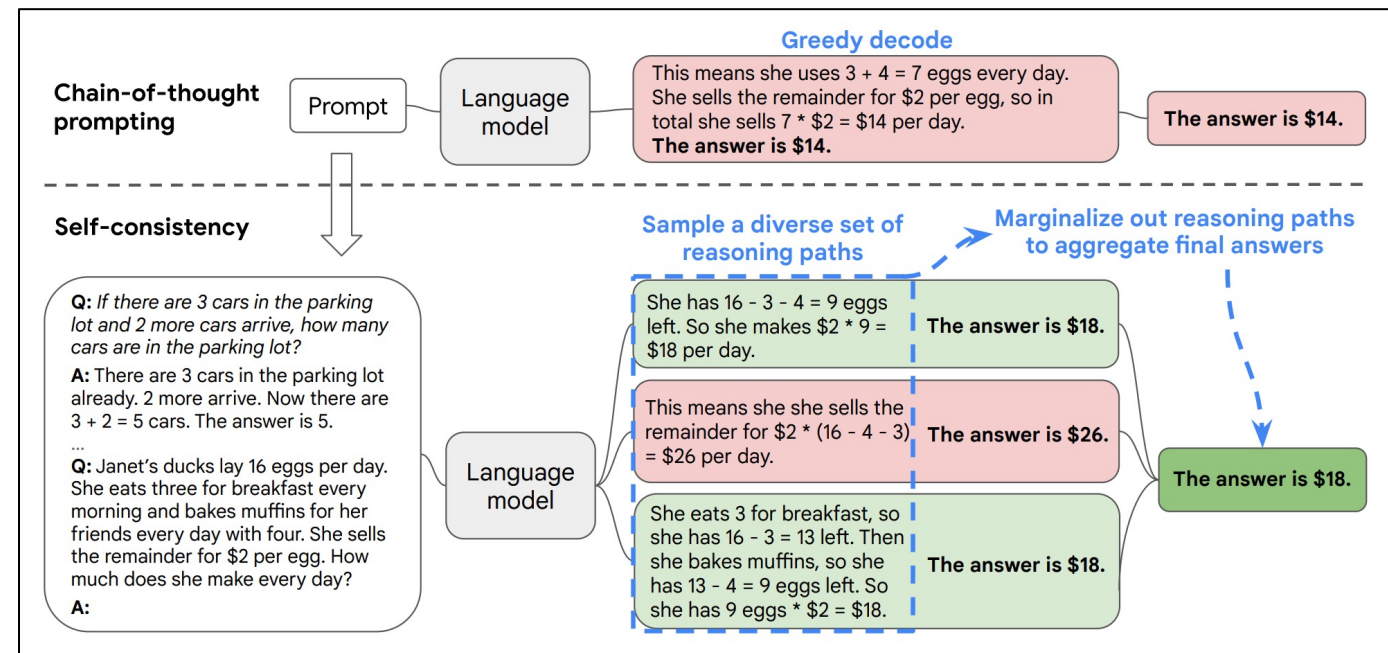
| (a) Few-shot | (b) Few-shot-CoT |
|--|--|
| <p>Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?</p> <p>A: The answer is 11.</p> <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A:</p> <p>(Output) The answer is 8. ✗</p> | <p>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?</p> <p>A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. $5 + 6 = 11$. The answer is 11.</p> <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A:</p> <p>(Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are $16 / 2 = 8$ golf balls. Half of the golf balls are blue. So there are $8 / 2 = 4$ blue golf balls. The answer is 4. ✓</p> |
| (c) Zero-shot | (d) Zero-shot-CoT (Ours) |
| <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A: The answer (arabic numerals) is</p> <p>(Output) 8 ✗</p> | <p>Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?</p> <p>A: Let's think step by step.</p> <p>(Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls. ✓</p> |

Self-Consistency

- *Self-consistency* aims to improve the naive greedy decoding used in chain-of-thought prompting.
 - The idea is to sample multiple, diverse reasoning paths through few-shot CoT, and use the generations to select the most consistent answer.

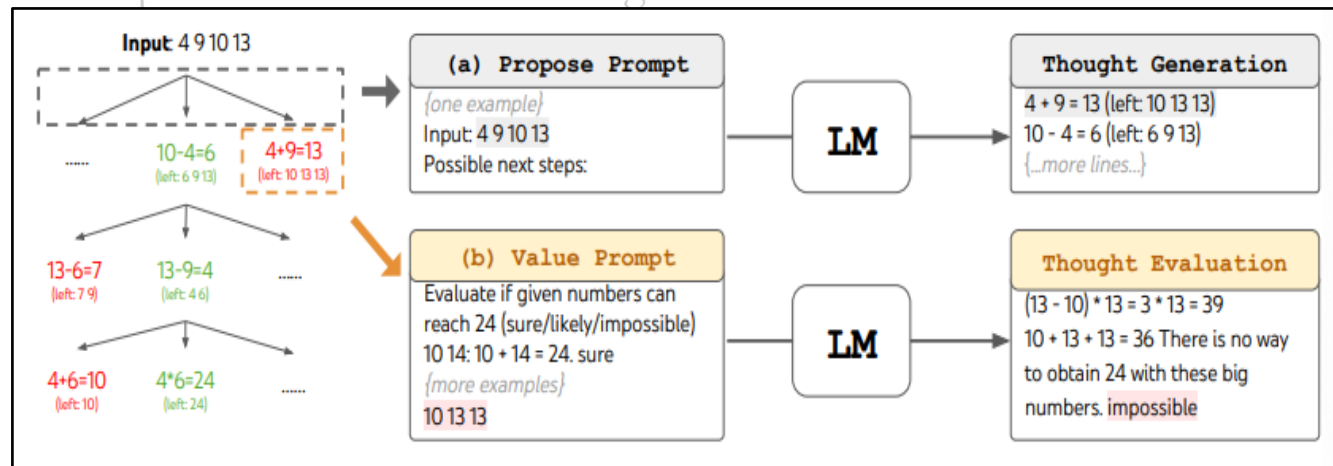
Three steps:

1. Prompt a language model using chain-of-thought (CoT) prompting.
2. Replace the “greedy decode” in CoT prompting by sampling from the language model’s decoder to generate a diverse set of reasoning paths.
3. Marginalize the reasoning paths and aggregate by choosing the most consistent answer in the final answer set.

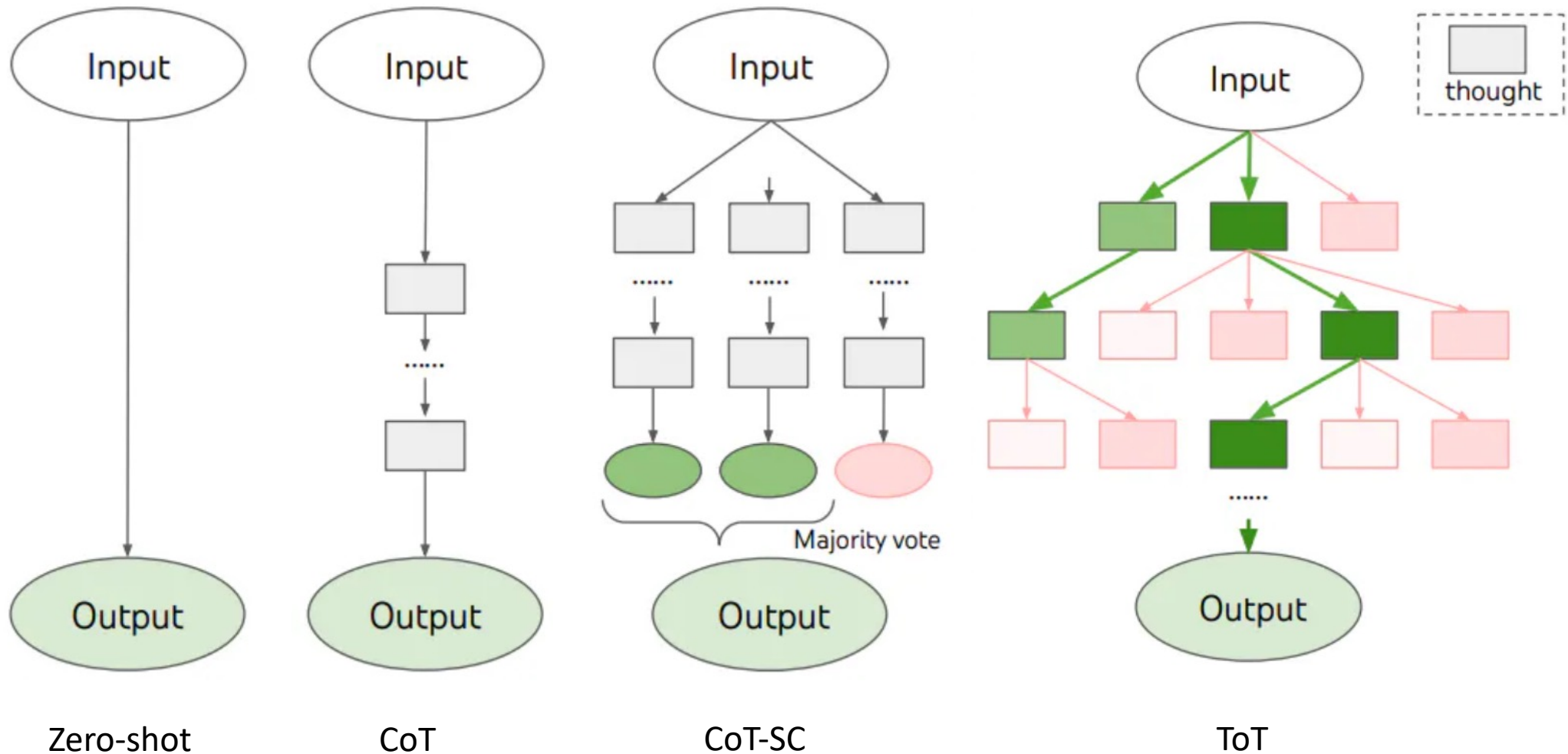


Tree of Thoughts (ToT)

- *Tree of Thoughts (ToT)* maintains a tree of thoughts where thoughts represent coherent language sequences that serve as intermediate steps to solve a problem.
- The LM's ability to generate and evaluate thoughts is then combined with search algorithms (e.g., breadth-first search and depth-first search) to enable systematic exploration of thoughts with lookahead and backtracking.
- Example: Game of 24:
 - To perform BFS in ToT for the Game of 24 task, the LLM is prompted to evaluate each thought candidate as sure/maybe/impossible w.r.t reaching 24.
 - The aim is to promote correct partial solutions that can be verified within a few lookahead trials, eliminate impossible partial solutions based on too big/small commonsense, and keep the rest maybe.
 - Values are sampled 3 times for each thought.



Some Comparison and Summary



Fine-Tuning v.s. Prompt Engineering

- Suppose we have:
 - A dataset $D = \{(x_i, y_i)\}_{i=1}^N$ and N is rather small.
 - A pre-trained LLM.
- How to fit it to your task?

- Option A: Fine-tuning:
 - Fine-tune the LLM on the training data using:
 - A standard training objective;
 - SGD to update (part of) the LLM's parameters.
 - Advantages:
 - Fits into the standard ML recipe;
 - Still works if N becomes relatively large.
 - Disadvantages:
 - Backward pass is computationally expensive in terms of FLOPs and memory footprint;
 - You have to have full access of the pre-trained LLM.

- Option B: Prompt engineering (in-context learning):
 - Feed training examples to the LLM as a prompt:
 - Allow the LLM to infer patterns in the training examples during inference;
 - Take the output of the LLM following the prompt as its prediction.
 - Advantages:
 - No backpropagation required and only one pass through the training data;
 - Does not require model weights, only API access.
 - Disadvantages:
 - The prompt may be very long.

Fine-Tuning v.s. Prompt Engineering

- Why would we ever bother with fine-tuning if it's so inefficient?
 - Because, even for very large LMs, fine-tuning often beats in-context learning.
 - In a fair comparison of fine-tuning (FT) and in-context learning (ICL), we find that FT outperforms ICL for most model sizes.

| | | FT | | | | | | | | | FT | | | | | | |
|-----|------|-------|-------|-------|-------|------|------|-------|-----|------|-------|-------|-------|-------|------|------|------|
| | | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | | | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B |
| ICL | 125M | -0.00 | 0.01 | 0.02 | 0.03 | 0.12 | 0.14 | 0.09 | ICL | 125M | -0.00 | 0.00 | 0.02 | 0.01 | 0.10 | 0.11 | 0.07 |
| | 350M | -0.00 | 0.01 | 0.02 | 0.03 | 0.12 | 0.14 | 0.09 | | 350M | -0.00 | 0.00 | 0.02 | 0.01 | 0.10 | 0.11 | 0.07 |
| | 1.3B | -0.00 | 0.01 | 0.02 | 0.03 | 0.12 | 0.14 | 0.09 | | 1.3B | -0.01 | -0.00 | 0.01 | 0.01 | 0.10 | 0.11 | 0.07 |
| | 2.7B | -0.00 | 0.01 | 0.02 | 0.03 | 0.12 | 0.14 | 0.09 | | 2.7B | -0.01 | -0.00 | 0.01 | 0.01 | 0.09 | 0.10 | 0.07 |
| | 6.7B | -0.00 | 0.01 | 0.02 | 0.03 | 0.12 | 0.14 | 0.09 | | 6.7B | -0.01 | -0.01 | 0.01 | 0.00 | 0.09 | 0.10 | 0.06 |
| | 13B | -0.04 | -0.02 | -0.01 | -0.00 | 0.09 | 0.11 | 0.05 | | 13B | -0.03 | -0.03 | -0.02 | -0.02 | 0.07 | 0.08 | 0.04 |
| | 30B | -0.11 | -0.09 | -0.08 | -0.08 | 0.02 | 0.03 | -0.02 | | 30B | -0.07 | -0.07 | -0.05 | -0.06 | 0.03 | 0.04 | 0.00 |

(a) RTE

(b) MNLI

Table 1: Difference between average **out-of-domain performance** of ICL and FT on RTE (a) and MNLI (b) across model sizes. We use 16 examples and 10 random seeds for both approaches. For ICL, we use the `gpt-3` pattern. For FT, we use pattern-based fine-tuning (PBFT) and select checkpoints according to in-domain performance. We perform a Welch's t-test and color cells according to whether: **ICL performs significantly better than FT**, **FT performs significantly better than ICL**. For cells without color, there is no significant difference.

References

- https://www.youtube.com/watch?v=zjkBMFhNj_g
- <https://arxiv.org/abs/2308.10792>
- https://www.andrew.cmu.edu/course/11-667/lectures/W4L2_PETM.pptx.pdf
- <https://www.cs.cmu.edu/~mgormley/courses/10423//slides/lecture11-peft-ink.pdf>
- <https://arxiv.org/abs/2012.13255>
- <https://arxiv.org/pdf/2106.09685>
- <https://www.promptingguide.ai/>
- <https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api>
- <https://arxiv.org/pdf/2203.11171.pdf>
- <https://arxiv.org/pdf/2305.10601.pdf>
- https://huggingface.co/docs/peft/en/conceptual_guides/prompting