Fundamentals and Adaptation of Large Language Model

An End-to-End LLM Bootcamp Presentation

OpenACC
More Science, Less Programming

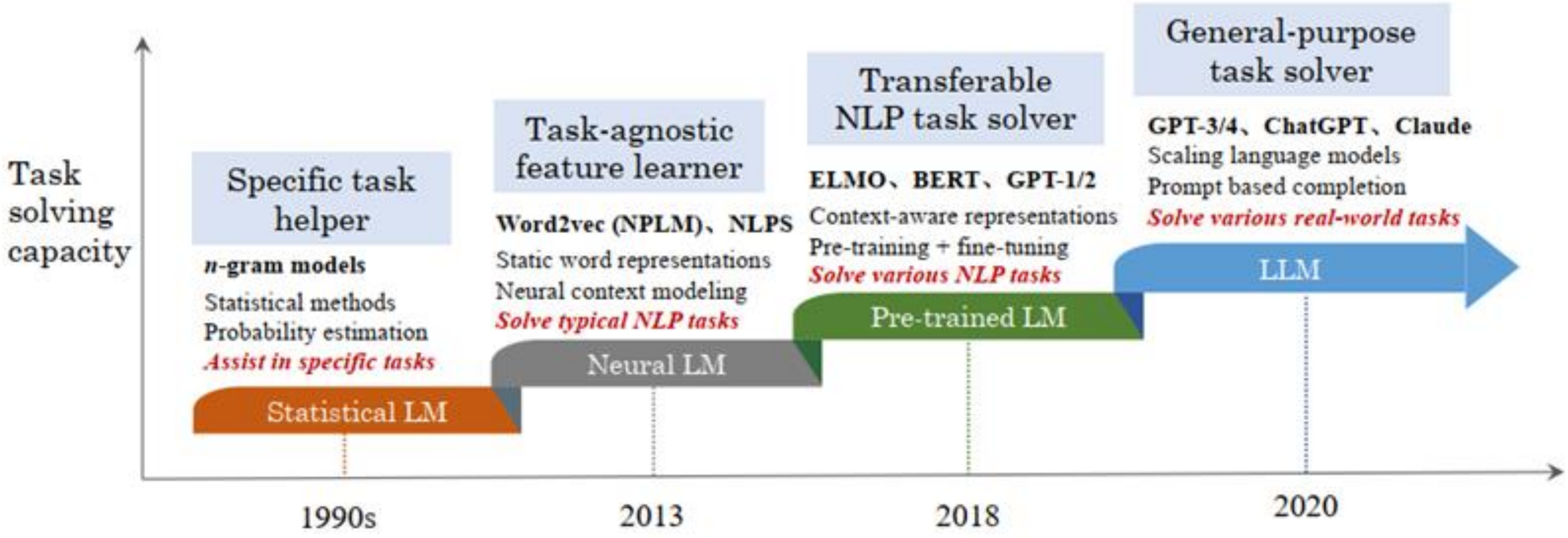
Agenda

- Evolution of LLM
- Background for LLMs
- Resources for developing LLMs
- Pretraining LLM
- Model Training
- Transformer Architecture
- Attention Mechanism
- Transformer Configurations & Position Embedding
- Long Context Modelling
- Decoding Strategy
- Adaptation of LLM
- PEFT

Evolution Of LLM

Background

- **Language modeling** (LM) is one of the major approaches to advancing the language intelligence of machines.
- LM aims to model the generative likelihood of word sequences to predict the probabilities of future (or missing) tokens.
- Its domain is categorized into four major development stages from the perspective of task solving capacity:



Background for LLMs

Scaling LLMs

Large Language Models (LLMs)

Refer to Transformer language models that contain hundreds of billions (or more) of parameters, which are trained on massive text data, such as GPT-3, PaLM, Galactica, LLaMA etc.

Formulation of Scaling Laws for LLMs

- Existing LLMs adopt similar Transformer architectures and pre-training objectives (e.g., language modeling) as small language models.
- However, LLMs significantly extend the **model size, data size**, and **total compute** (orders of magnification).

Scaling laws for Transformer language models

- **KM scaling law** by Kaplan et al. 2020 (the OpenAI team): *Scaling Laws for Neural Language Models [https://arxiv.org/abs/2001.08361]*
- **Chinchilla scaling law** by Hoffmann et al. 2022 (the Google DeepMind team): *Training Compute-Optimal Large Language Models [https://arxiv.org/abs/2203.15556]*

Emergent Abilities of LLMs

The abilities that are not present in small models but arise in large models”, which is one of the most prominent features that distinguish LLMs from previous PLMs.

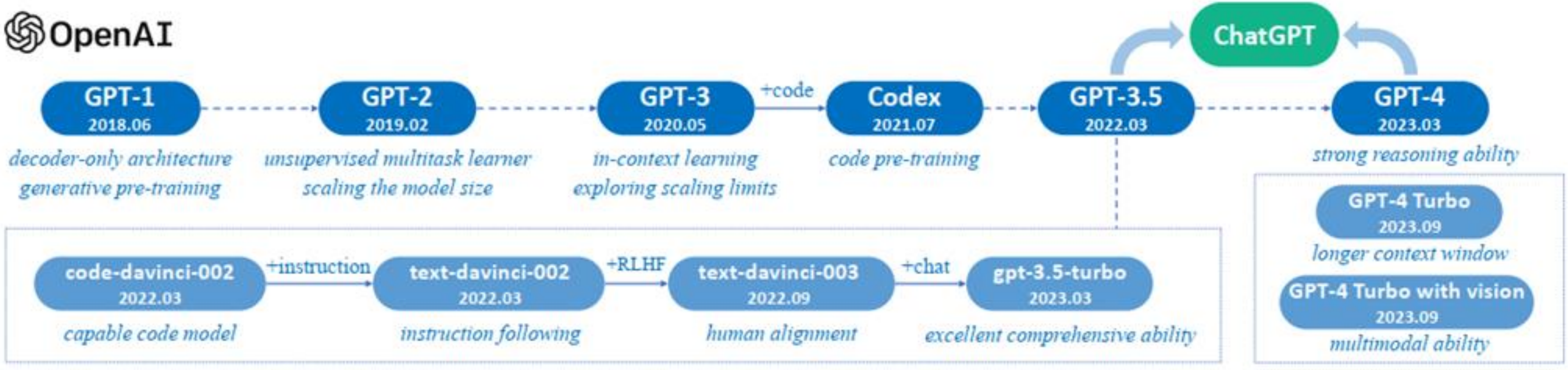
Three typical emergent abilities for LLMs

- **In-context learning** : Provided with several task demonstrations, that can generate the expected output for the test instances by completing the word sequence of input text, without requiring additional training or gradient update.

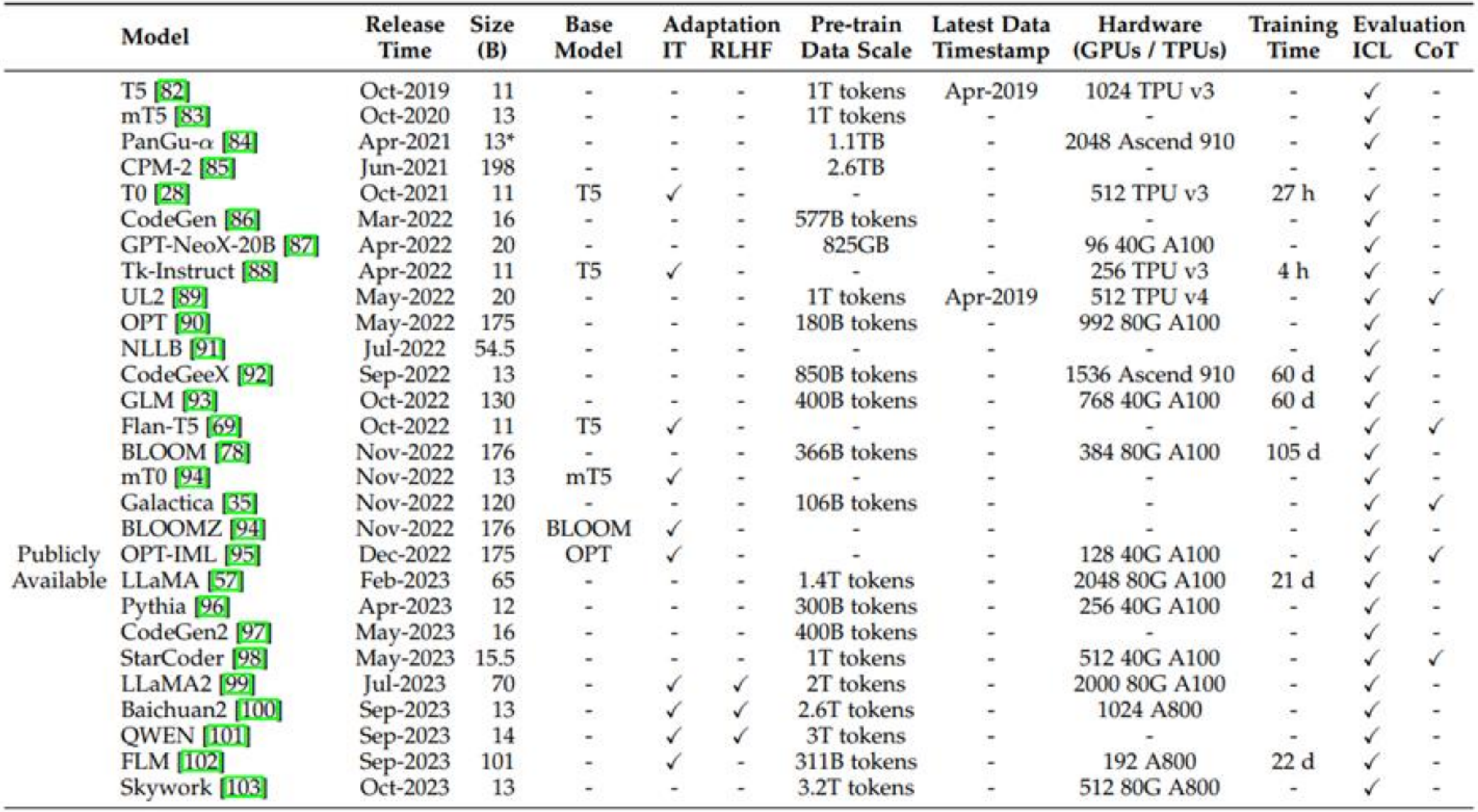
- **Instruction following** : With instruction tuning, LLMs are enabled to follow the task instructions for new tasks without using explicit examples, thus having an improved generalization ability.

- **Step-by-step reasoning:** with the chain-of-thought (CoT) prompting strategy, LLMs can solve e.g., mathematical word problems tasks by utilizing the prompting mechanism that involves intermediate reasoning steps for deriving the final answer.

A brief Illustration for GPT-series Model



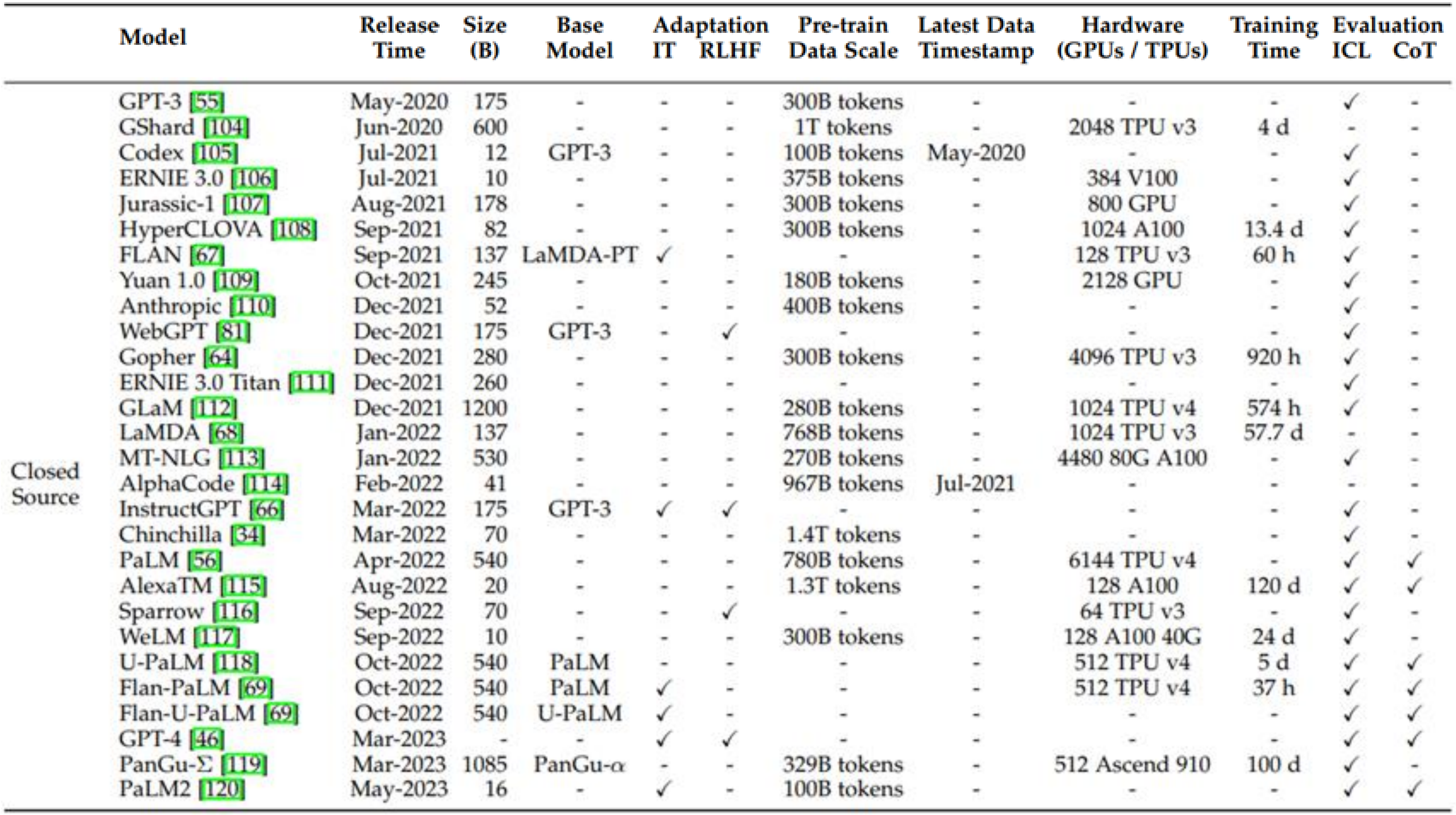
Publicly Available LLMs



	Model	Release Time	Size (B)	Base Model	Adaptation		Pre-train Data Scale	Latest Data Timestamp	Hardware (GPUs / TPUs)	Training Time	Evaluation	
					IT	RLHF					ICL	CoT
Publicly Available	T5 [82]	Oct-2019	11	-	-	-	1T tokens	Apr-2019	1024 TPU v3	-	✓	-
	mT5 [83]	Oct-2020	13	-	-	-	1T tokens	-	-	-	✓	-
	PanGu-α [84]	Apr-2021	13*	-	-	-	1.1TB	-	2048 Ascend 910	-	✓	-
	CPM-2 [85]	Jun-2021	198	-	-	-	2.6TB	-	-	-	-	-
	T0 [28]	Oct-2021	11	T5	✓	-	-	-	512 TPU v3	27 h	✓	-
	CodeGen [86]	Mar-2022	16	-	-	-	577B tokens	-	-	-	✓	-
	GPT-NeoX-20B [87]	Apr-2022	20	-	-	-	825GB	-	96 40G A100	-	✓	-
	Tk-Instruct [88]	Apr-2022	11	T5	✓	-	-	-	256 TPU v3	4 h	✓	-
	UL2 [89]	May-2022	20	-	-	-	1T tokens	Apr-2019	512 TPU v4	-	✓	✓
	OPT [90]	May-2022	175	-	-	-	180B tokens	-	992 80G A100	-	✓	-
	NLLB [91]	Jul-2022	54.5	-	-	-	-	-	-	-	✓	-
	CodeGeeX [92]	Sep-2022	13	-	-	-	850B tokens	-	1536 Ascend 910	60 d	✓	-
	GLM [93]	Oct-2022	130	-	-	-	400B tokens	-	768 40G A100	60 d	✓	-
	Flan-T5 [69]	Oct-2022	11	T5	✓	-	-	-	-	-	✓	✓
	BLOOM [78]	Nov-2022	176	-	-	-	366B tokens	-	384 80G A100	105 d	✓	-
	mT0 [94]	Nov-2022	13	mT5	✓	-	-	-	-	-	✓	-
	Galactica [35]	Nov-2022	120	-	-	-	106B tokens	-	-	-	✓	✓
	BLOOMZ [94]	Nov-2022	176	BLOOM	✓	-	-	-	-	-	✓	-
	OPT-IML [95]	Dec-2022	175	OPT	✓	-	-	-	128 40G A100	-	✓	✓
	LLaMA [57]	Feb-2023	65	-	-	-	1.4T tokens	-	2048 80G A100	21 d	✓	-
	Pythia [96]	Apr-2023	12	-	-	-	300B tokens	-	256 40G A100	-	✓	-
	CodeGen2 [97]	May-2023	16	-	-	-	400B tokens	-	-	-	✓	-
	StarCoder [98]	May-2023	15.5	-	-	-	1T tokens	-	512 40G A100	-	✓	✓
	LLaMA2 [99]	Jul-2023	70	-	✓	✓	2T tokens	-	2000 80G A100	-	✓	-
	Baichuan2 [100]	Sep-2023	13	-	✓	✓	2.6T tokens	-	1024 A800	-	✓	-
	QWEN [101]	Sep-2023	14	-	✓	✓	3T tokens	-	-	-	✓	-
	FLM [102]	Sep-2023	101	-	✓	-	311B tokens	-	192 A800	22 d	✓	-
	Skywork [103]	Oct-2023	13	-	-	-	3.2T tokens	-	512 80G A800	-	✓	-

Source: A Survey of Large Language Models arXiv:2303.18223 [cs.CL]

Closed Source LLMs



	Model	Release Time	Size (B)	Base Model	Adaptation		Pre-train Data Scale	Latest Data Timestamp	Hardware (GPUs / TPUs)	Training Time	Evaluation	
					IT	RLHF					ICL	CoT
Closed Source	GPT-3 [55]	May-2020	175	-	-	-	300B tokens	-	-	-	✓	-
	GShard [104]	Jun-2020	600	-	-	-	1T tokens	-	2048 TPU v3	4 d	-	-
	Codex [105]	Jul-2021	12	GPT-3	-	-	100B tokens	May-2020	-	-	✓	-
	ERNIE 3.0 [106]	Jul-2021	10	-	-	-	375B tokens	-	384 V100	-	✓	-
	Jurassic-1 [107]	Aug-2021	178	-	-	-	300B tokens	-	800 GPU	-	✓	-
	HyperCLOVA [108]	Sep-2021	82	-	-	-	300B tokens	-	1024 A100	13.4 d	✓	-
	FLAN [67]	Sep-2021	137	LaMDA-PT	✓	-	-	-	128 TPU v3	60 h	✓	-
	Yuan 1.0 [109]	Oct-2021	245	-	-	-	180B tokens	-	2128 GPU	-	✓	-
	Anthropic [110]	Dec-2021	52	-	-	-	400B tokens	-	-	-	✓	-
	WebGPT [81]	Dec-2021	175	GPT-3	-	✓	-	-	-	-	✓	-
	Gopher [64]	Dec-2021	280	-	-	-	300B tokens	-	4096 TPU v3	920 h	✓	-
	ERNIE 3.0 Titan [111]	Dec-2021	260	-	-	-	-	-	-	-	✓	-
	GLaM [112]	Dec-2021	1200	-	-	-	280B tokens	-	1024 TPU v4	574 h	✓	-
	LaMDA [68]	Jan-2022	137	-	-	-	768B tokens	-	1024 TPU v3	57.7 d	-	-
	MT-NLG [113]	Jan-2022	530	-	-	-	270B tokens	-	4480 80G A100	-	✓	-
	AlphaCode [114]	Feb-2022	41	-	-	-	967B tokens	Jul-2021	-	-	-	-
	InstructGPT [66]	Mar-2022	175	GPT-3	✓	✓	-	-	-	-	✓	-
	Chinchilla [34]	Mar-2022	70	-	-	-	1.4T tokens	-	-	-	✓	-
	PaLM [56]	Apr-2022	540	-	-	-	780B tokens	-	6144 TPU v4	-	✓	✓
	AlexaTM [115]	Aug-2022	20	-	-	-	1.3T tokens	-	128 A100	120 d	✓	✓
	Sparrow [116]	Sep-2022	70	-	-	✓	-	-	64 TPU v3	-	✓	-
	WeLM [117]	Sep-2022	10	-	-	-	300B tokens	-	128 A100 40G	24 d	✓	-
	U-PaLM [118]	Oct-2022	540	PaLM	-	-	-	-	512 TPU v4	5 d	✓	✓
	Flan-PaLM [69]	Oct-2022	540	PaLM	✓	-	-	-	512 TPU v4	37 h	✓	✓
	Flan-U-PaLM [69]	Oct-2022	540	U-PaLM	✓	-	-	-	-	-	✓	✓
	GPT-4 [46]	Mar-2023	-	-	✓	✓	-	-	-	-	✓	✓
	PanGu-Σ [119]	Mar-2023	1085	PanGu-α	-	-	329B tokens	-	512 Ascend 910	100 d	✓	-
	PaLM2 [120]	May-2023	16	-	✓	-	100B tokens	-	-	-	✓	✓

Source: A Survey of Large Language Models arXiv:2303.18223 [cs.CL]

Resources for Developing LLMs

Data Source for Pre-training and Fine-tuning

Commonly Used Corpora for Pre-training

Corpora	Size	Source
BookCorpus	5GB	**Books**
Gutenberg	-	Books
C4	800GB	**CommonCrawl**
CC-Stories-R	31GB	CommonCrawl
CC-NEWS	78GB	CommonCrawl
REALNEWs	120GB	CommonCrawl
OpenWebText	38GB	**Reddit links**
Pushift.io	2TB	Reddit links
Wikipedia	21GB	**Wikipedia**
BigQuery	-	**Codes**
the Pile	800GB	**Others**
ROOTS	1.6TB	Others

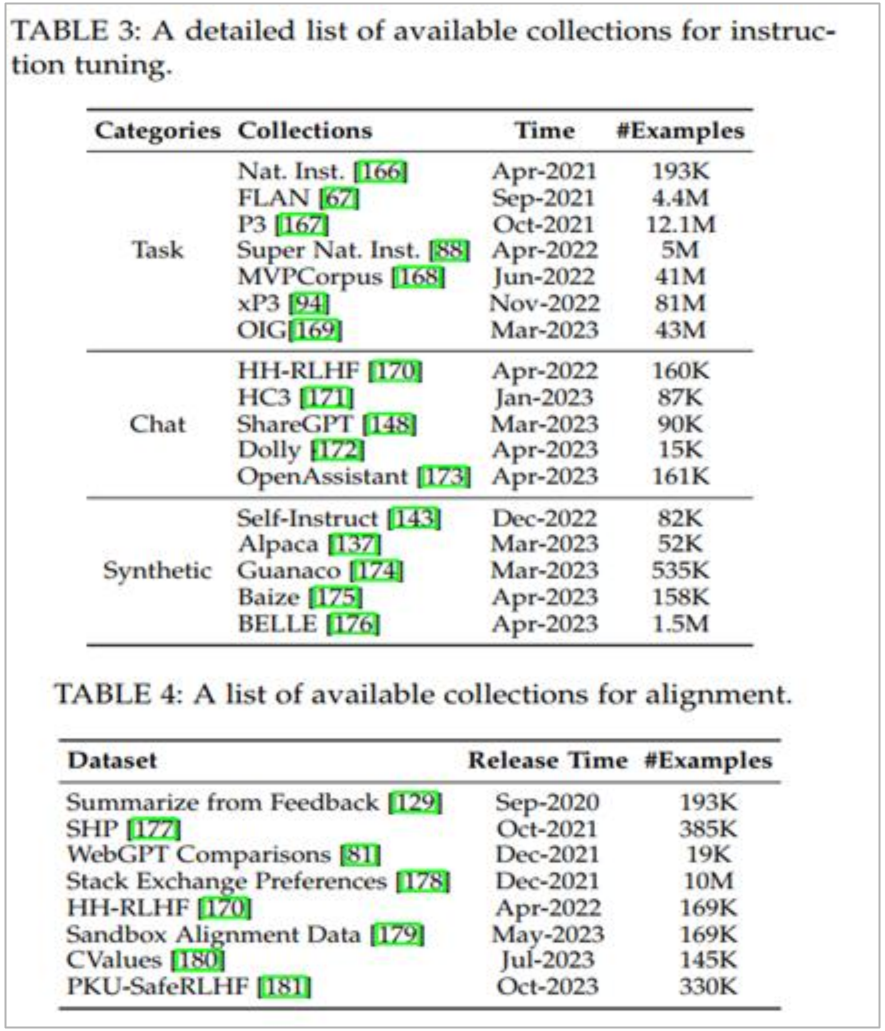


TABLE 3: A detailed list of available collections for instruction tuning.

Categories	Collections	Time	#Examples
Task	Nat. Inst. [166]	Apr-2021	193K
	FLAN [67]	Sep-2021	4.4M
	P3 [167]	Oct-2021	12.1M
	Super Nat. Inst. [88]	Apr-2022	5M
	MVPCorpus [168]	Jun-2022	41M
	xP3 [94]	Nov-2022	81M
	OIG[169]	Mar-2023	43M
Chat	HH-RLHF [170]	Apr-2022	160K
	HC3 [171]	Jan-2023	87K
	ShareGPT [148]	Mar-2023	90K
	Dolly [172]	Apr-2023	15K
	OpenAssistant [173]	Apr-2023	161K
Synthetic	Self-Instruct [143]	Dec-2022	82K
	Alpaca [137]	Mar-2023	52K
	Guanaco [174]	Mar-2023	535K
	Baize [175]	Apr-2023	158K
	BELLE [176]	Apr-2023	1.5M

TABLE 4: A list of available collections for alignment.

Dataset	Release Time	#Examples
Summarize from Feedback [129]	Sep-2020	193K
SHP [177]	Oct-2021	385K
WebGPT Comparisons [81]	Dec-2021	19K
Stack Exchange Preferences [178]	Dec-2021	10M
HH-RLHF [170]	Apr-2022	169K
Sandbox Alignment Data [179]	May-2023	169K
CValues [180]	Jul-2023	145K
PKU-SafeRLHF [181]	Oct-2023	330K

Fine-tuning

- Instruction tuning
- Alignment tuning

Library Resource

Transformers

is an open-source Python library for building models using the Transformer architecture, which is developed and maintained by Hugging Face

DeepSpeed

compatible with PyTorch and developed by Microsoft. It provides the support of various optimization techniques for distributed training, such as memory optimization (**ZeRO technique, gradient checkpointing**), and **pipeline parallelism**.

Megatron-LM

is a deep learning library developed by NVIDIA for training large-scale language models. It also provides rich optimization techniques for distributed training, including **mode**l and **data parallelism**, **mixedprecision** training, and **FlashAttention**.

JAX

is a Python library for **high-performance machine learning algorithms** developed by Google, allowing users to easily perform computations on arrays with hardware acceleration (e.g., GPU or TPU)

Colossal-AI

Colossal-AI: is a deep learning library developed by HPC-AI Tech for training large-scale AI models

vLLM

is a fast, memory efficient, and easy-to-use library for LLM inference and serving. To enable fast inference, it is specially optimized with high serving throughput, effective attention memory management using **PagedAttention**, **continuous batching**, and **optimized CUDA kernels**.

DeepSpeed-MII, FastMoE, BMTrain , DeepSpeed-Chat,

etc.

Pretraining LLM

Data Collection and Preparation

- Compared with small-scale language models, LLMs have a stronger demand for high-quality data for model pretraining, and their model capacities largely rely on the pretraining corpus and how it has been preprocessed.

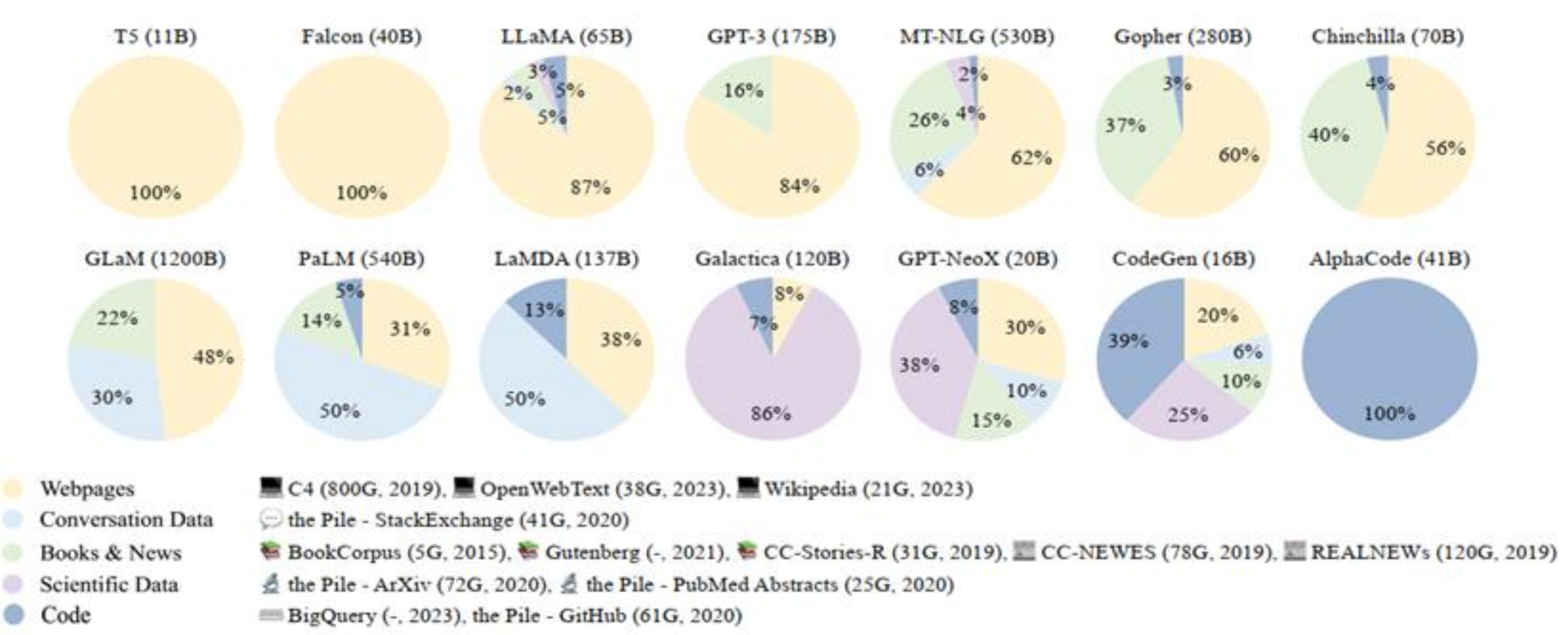


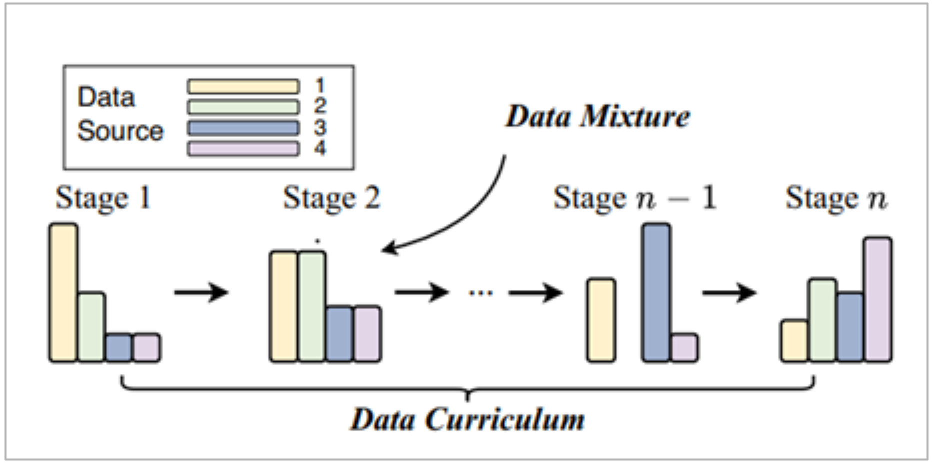
Fig. 6: Ratios of various data sources in the pre-training data for existing LLMs.

Data Scheduling

- After data preprocessing, it is essential to design suitable strategies to schedule multi-source data for pretraining a capable LLM.

Two key aspects for data scheduling:

- **the proportion** of each data source (data mixture) and,
- **the order** in which each data source is scheduled for training (data curriculum).



To determine data curriculum, a practical approach is to monitor the development of key abilities of LLMs based on specially constructed evaluation benchmarks, and then adaptively adjust the data mixture during pre-training.

Text Data Classification

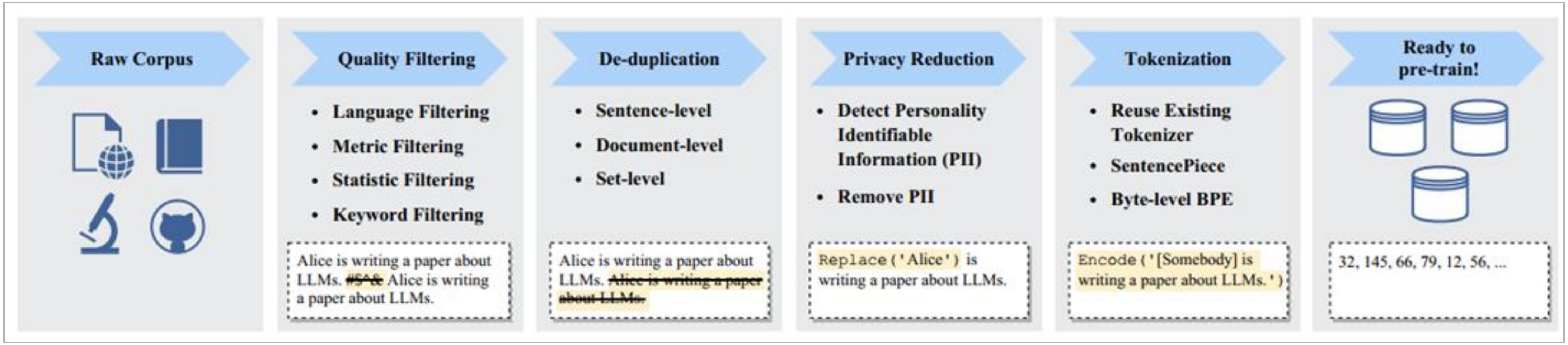
General Text Data: general-purpose pre-training data, such as webpages, books, and conversational text, which provides rich text sources on a variety of topics.

Webpages	Conversation text	Book
Enables LLMs to **gain diverse linguistic knowledge** and enhance their generalization capabilities	Conversation data can **enhance the conversational competence** of LLMs and potentially	Compared to other corpus, books provide an important source of formal long texts, which are potentially beneficial for LLMs **to learn linguistic knowledge**, **model long-term dependency**, and **generate narrative and coherent texts**.

Specialized Text Data: Specialized datasets are useful to improve the specific capabilities of LLMs on downstream tasks

Multilingual text	Scientific text	Code
In addition to the text in the target language, integrating a multilingual corpus can **enhance the multilingual abilities of language understanding and generation**	The exploration of science by humans has been witnessed by the increasing growth of scientific publications. In order to **enhance the understanding of scientific knowledge for LLMs,** it is useful to incorporate a scientific corpus for model pre-training	It remains challenging for these PLMs to generate high quality and accurate programs. Training LLMs on a **vast code corpus can lead to a substantial improvement in the quality of the synthesized programs**.

Data Preprocessing



Raw Corpus

Quality Filtering

- Language Filtering
- Metric Filtering
- Statistic Filtering
- Keyword Filtering

Alice is writing a paper about LLMs. ~~#$^&~~ Alice is writing a paper about LLMs.

De-duplication

- Sentence-level
- Document-level
- Set-level

Alice is writing a paper about LLMs. ~~Alice is writing a paper about LLMs.~~

Privacy Reduction

- Detect Personality Identifiable Information (PII)
- Remove PII

`Replace('Alice')` is writing a paper about LLMs.

Tokenization

- Reuse Existing Tokenizer
- SentencePiece
- Byte-level BPE

`Encode('[Somebody] is writing a paper about LLMs.')`

Ready to pre-train!

32, 145, 66, 79, 12, 56, ...

An illustration of a typical data preprocessing pipeline for pre-training LLM

Tokenization Methods

- LLMs use tokenization as the essential preprocessing step. It aims to segment raw text into sequences of individual tokens, which are subsequently used as the inputs of LLMs
- Tokens can be **characters**, **sub-words**, **symbols**, or **words**, depending on the size and type of the model. Some of the commonly used tokenization schemes in LLMs are:

Byte-Pair Encoding (BPE) tokenization.

- BPE was originally proposed as a general data compression algorithm and then adapted to NLP for tokenization.
- It starts with a set of **basic symbols** and iteratively combine frequent pairs of two consecutive tokens in the corpus as new tokens **(called merge**).
- For each merge, the selection criterion is based on the **co-occurrence frequency of two contiguous tokens**: the top frequent pair would be selected.
- The merge process continues until it reaches the predefined size.

WordPiece tokenization.

- WordPiece was a Google internal subword tokenization algorithm.
- similar idea with BPE by iteratively merging consecutive tokens,
- To conduct the merge, it first **trains a language model** and employs it to **score all possible pairs**.
- Then, at each merge, it selects the pair that leads to the most increase in the likelihood of training data.

Unigram tokenization.

- Unlike BPE and WordPiece, Unigram tokenization starts with a sufficiently large set of **possible substrings** or **subtokens** for a corpus, and iteratively **removes the tokens** in the current vocabulary until the **expected vocabulary size is reached**

Model Training

Scalable Training Techniques

3D parallelism

Data parallelism

- It is one of the most fundamental approaches to improving the training throughput.
- It replicates the model parameters and optimizer states across multiple GPUs and then distributes the whole training corpus into these GPUs.
- In this way, each GPU only needs to process the assigned data for it and performs the forward and backward propagation to obtain the gradients.
- The computed gradients on different GPUs will be further aggregated to obtain the gradients of the entire batch for updating the models in all GPUs.

Pipeline parallelism

- Pipeline parallelism aims to distribute the different layers of a LLM into multiple GPUs.
- Especially, in the case of a Transformer model, pipeline parallelism loads consecutive layers onto the same GPU, to reduce the cost of transmitting the computed hidden states or gradients between GPUs.
- However, a naive implementation of pipeline parallelism may result in a lower GPU utilization rate as each GPU has to wait for the previous one to complete the computation.

Scalable Training Techniques

Tensor parallelism

- It aims to decompose the LLM for multi-GPU loading.
- Unlike pipeline parallelism, tensor parallelism focuses on decomposing the tensors (the parameter matrices) of LLMs.
- For a matrix multiplication operation **Y = XA** in the LLM, the parameter matrix A can be split into two submatrices, **A1 and A2**, by column, which can be expressed as **Y = [XA1, XA2]**.
- By placing matrices **A1** and **A2** on different GPUs, the matrix multiplication operation would be invoked at two GPUs in parallel, and the final result can be obtained by combining the outputs from the two GPUs through across-GPU communication.

Scalable Training Techniques

ZeRO

- Proposed by the DeepSpeed library, **focuses on the issue of memory redundancy in data parallelism**.
- Data parallelism requires each GPU to store the same copy of a LLM, including model parameters, model gradients, and optimizer parameters.
- Whereas, not all of the above data is necessary to be retained on each GPU, which would cause a memory redundancy problem.
- To resolve it, **the ZeRO technique aims to retain only a fraction of data on each GPU, while the rest data can be retrieved from other GPUs when required.**
- Specifically, ZeRO provides three solutions, depending on how the three parts of the data are stored, namely **optimizer state partitioning, gradient partitioning**, and **parameter partitioning**.
- Implemented by PyTorch as **Fully Sharded Data Parallel (FSDP)**

Mixed Precision Training

- To pre-train extremely large language models, some studies have started to utilize **16-bit floating-point numbers (FP16)**,
- However, existing work has found that FP16 may lead to the loss of computational accuracy, which affects the final model performance.
- To alleviate it, an alternative called **Brain Floating Point (BF16**) has been used for training, which allocates more exponent bits and fewer significant bits than FP16.

TRANSFORMER ARCHITECTURE

Categories Of LLM Architecture

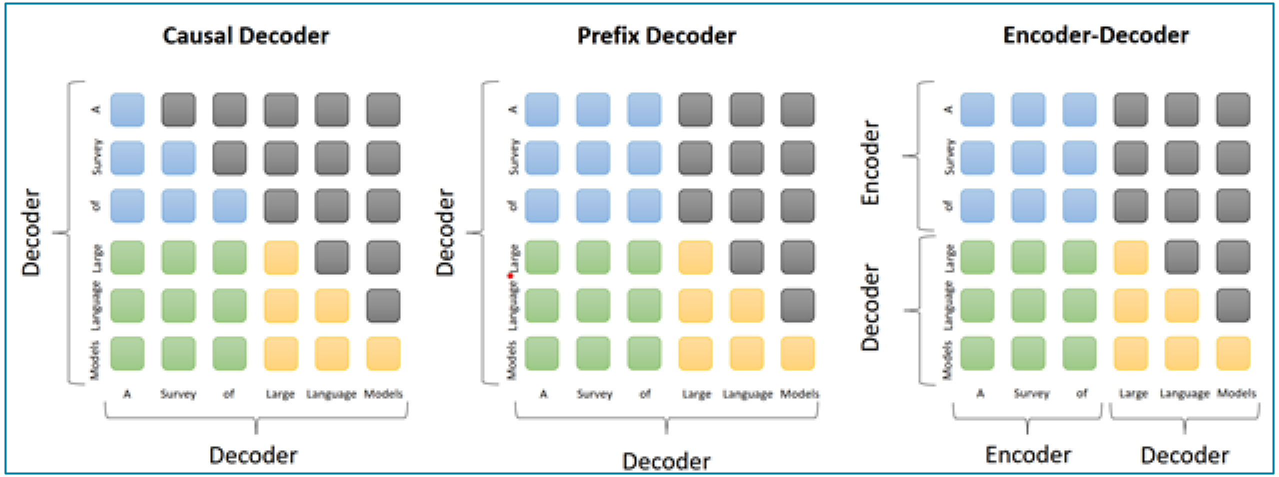
- Due to its excellent parallelizability and capacity, **the Transformer architecture has become the de facto backbone for developing various LLMs**, making it possible to scale language models to hundreds or thousands of billions of parameters.
- The mainstream architectures of existing LLMs can be roughly categorized into three major types:

Causal Decoder Architecture

The causal decoder architecture **incorporates the unidirectional attention mask** to guarantee that each **input token can only attend to the past tokens and itself**. The input and output tokens are processed in the same fashion through the decoder (e.g., GPT-series models)

Prefix Decoder Architecture

The prefix decoder architecture (a.k.a., non-causal decoder) **revises the masking mechanism of causal decoders to enable performing bidirectional attention over the prefix tokens** and unidirectional attention only on generated tokens. (e.g., GLM-130B and U-PaLM)



The blue, green, yellow, and grey rounded rectangles indicate the attention between prefix tokens, attention between prefix and target tokens, attention between target tokens, and masked attention, respectively.

Encoder-decoder Architecture

The encoder-decoder architecture **consists of two stacks of Transformer blocks as the encoder and decoder**, respectively. **The encoder adopts stacked multi-head self-attention layers to encode the input sequence and generate its latent representations.** At the same time, **the decoder performs cross-attention** on these representations and autoregressively generates the target sequence. (e.g., T5 and BART)

Vanilla Architecture Transformer

Based on the **Attention Is All You Need** paper by **Vaswani et al., 2017.**

Encoder

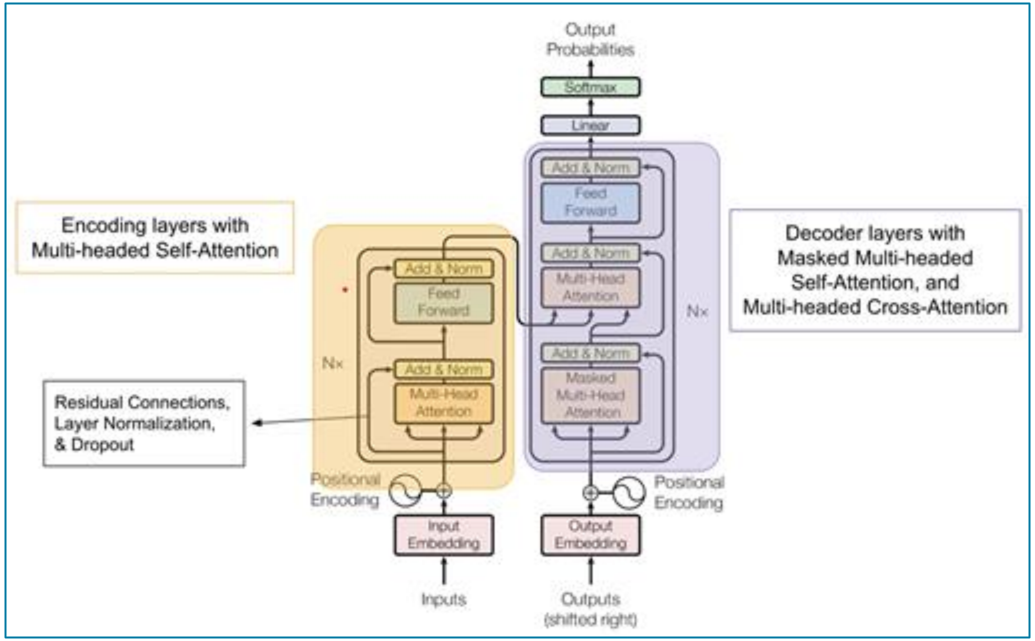
The encoder comprises a stack of 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise, fully connected feed-forward network.

Decoder

The decoder is also composed of a stack of 6 identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack.

Multi-Head Attention

It consists of several attention layers running in parallel.

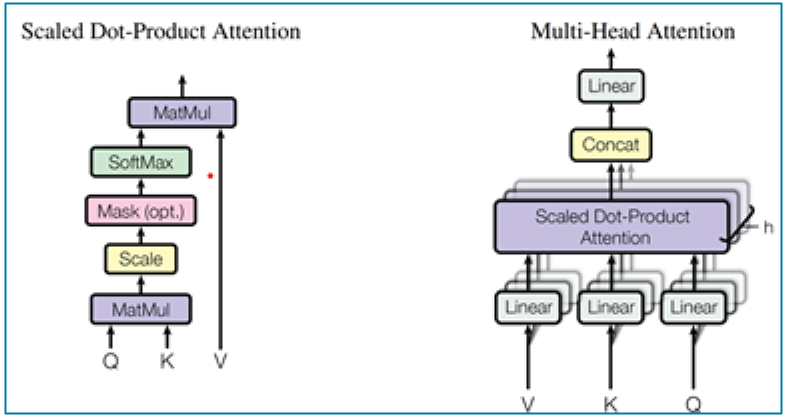


Encoding layers with Multi-headed Self-Attention

Decoder layers with Masked Multi-headed Self-Attention, and Multi-headed Cross-Attention

Residual Connections, Layer Normalization, & Dropout

Source: https://aiml.com/explain-the-transformer-architecture/



Source: arXiv:1706.03762 [cs.CL]

Scaled Dot-Product Attention

The input consists of **queries** and **keys** of dimension d_k and **values** of dimension d_v. The dot products of the query is computed with all keys divided by $\sqrt{d_k}$. And apply a SoftMax function to obtain the weights of the values.

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

Position-wise Feed-Forward Networks

This consists of two linear transformations with a ReLU activation in between.

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

ATTENTION MECHANISM

Attention Visualization

The Attention mechanism allows the tokens across the sequence to interact with each other and compute the representations of the input and output sequence.

An attention function maps a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors.

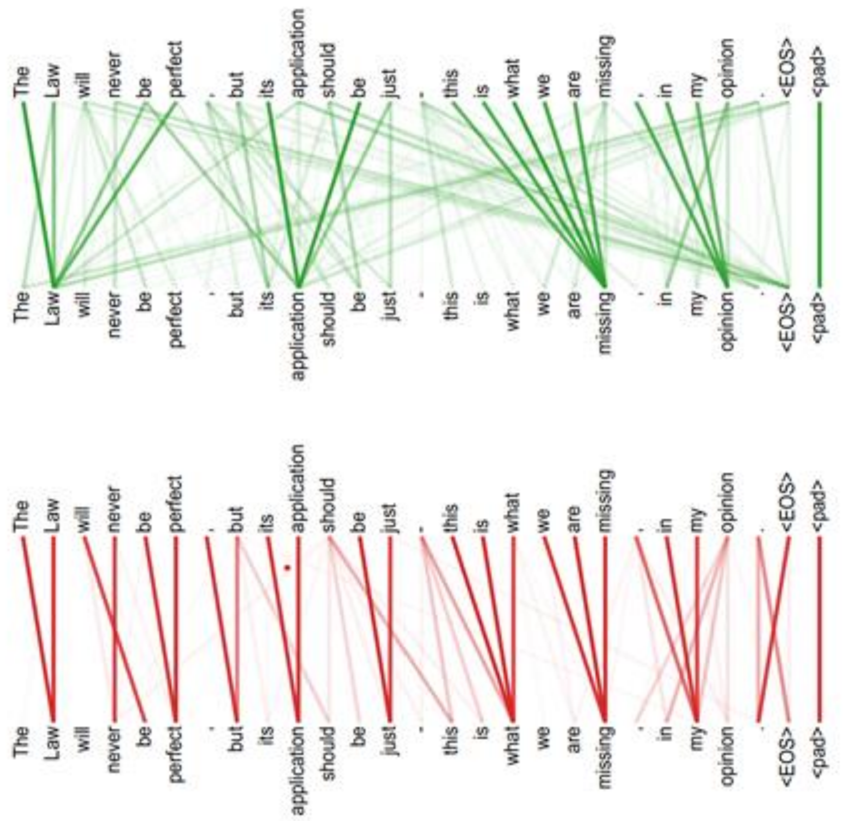


Figure 5: Many of the attention heads exhibit behaviour that seems related to the structure of the sentence. We give two such examples above, from two different heads from the encoder self-attention at layer 5 of 6. The heads clearly learned to perform different tasks.

Source: arXiv:1706.03762 [cs.CL]

Self Attention Illustration (1/ 7)

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

Sample Sentence: "a b c D"

1. Turning words into tokens:

$$\begin{aligned} a &= [a_1 \quad a_2 \quad a_3] \\ b &= [b_1 \quad b_2 \quad b_3] \\ c &= [c_1 \quad c_2 \quad c_3] \\ D &= [D_1 \quad D_2 \quad D_3] \end{aligned}$$

2. Our sequence consists of four tokens, each a vector of three values. Now, let's turn these tokens into a

$$X = \begin{array}{c} \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \\ D_1 & D_2 & D_3 \end{bmatrix} \begin{matrix} a \\ b \\ c \\ D \end{matrix} \end{array}$$

X, a 4×3 matrix made of the vectors, a, b, c, and D

3. (i) Generate the keys, queries, and values using weighted matrices.

(ii) Transform our sentence into a 4×2 matrix, so each weight matrices will be of shape 3×2.

$$QW = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \\ Q_{31} & Q_{32} \end{bmatrix}$$

Weights to transform the X matrix into the Q (Query) matrix

$$Q = X \bullet QW = \begin{matrix} a \\ b \\ c \\ D \end{matrix} \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \\ D_1 & D_2 & D_3 \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \\ Q_{31} & Q_{32} \end{bmatrix}$$

$$Q = \begin{bmatrix} a_1Q_{11} + a_2Q_{21} + a_3Q_{31} & a_1Q_{12} + a_2Q_{22} + a_3Q_{32} \\ b_1Q_{11} + b_2Q_{21} + b_3Q_{31} & b_1Q_{12} + b_2Q_{22} + b_3Q_{32} \\ \cdots & \cdots \\ D_1Q_{11} + D_2Q_{21} + D_3Q_{31} & D_1Q_{12} + D_2Q_{22} + D_3Q_{32} \end{bmatrix} = \begin{bmatrix} a_1^Q & a_2^Q \\ b_1^Q & b_2^Q \\ c_1^Q & c_2^Q \\ D_1^Q & D_2^Q \end{bmatrix} = \begin{bmatrix} a^Q \\ b^Q \\ c^Q \\ D^Q \end{bmatrix}$$

The query matrix computed from the X matrix and the query weights

Each vector is a linear combination of itself and some weights.

Source: https://gmongaras.medium.com/how- ition-masks-work-72ed9382510f

Self Attention Illustration (2/7)

4. This transformation is the same for the keys and values resulting in the following matrices.

$$Q = \begin{bmatrix} a_1^Q & a_2^Q \\ b_1^Q & b_2^Q \\ c_1^Q & c_2^Q \\ D_1^Q & D_2^Q \end{bmatrix} = \begin{bmatrix} a^Q \\ b^Q \\ c^Q \\ D^Q \end{bmatrix} \qquad K = \begin{bmatrix} a_1^K & a_2^K \\ b_1^K & b_2^K \\ c_1^K & c_2^K \\ D_1^K & D_2^K \end{bmatrix} = \begin{bmatrix} a^K \\ b^K \\ c^K \\ D^K \end{bmatrix} \qquad V = \begin{bmatrix} a_1^V & a_2^V \\ b_1^V & b_2^V \\ c_1^V & c_2^V \\ D_1^V & D_2^V \end{bmatrix} = \begin{bmatrix} a^V \\ b^V \\ c^V \\ D^V \end{bmatrix}$$

According to **Vaswani et al., 2017** the scalar d_k is used because they suspected that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions with extremely small gradients. So, d_k is just a scalar helping the gradients, which we can neglect to simplify this illustration.

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

$$\boxed{Attention(Q,K,V) = softmax(QK^T)V}$$

Source: https://gmongaras.medium.com/how-do-self-attention-masks-work-72ed9382510f

Self Attention Illustration (3/7)

5. Compute the **QK^T** matrix as follows.

$$QK^T = \begin{bmatrix} a_1^Q & a_2^Q \\ b_1^Q & b_2^Q \\ c_1^Q & c_2^Q \\ D_1^Q & D_2^Q \end{bmatrix} \begin{bmatrix} a_1^K & b_1^K & c_1^K & D_1^K \\ a_2^K & b_2^K & c_2^K & D_2^K \end{bmatrix} = \begin{bmatrix} a_1^Q a_1^K + a_2^Q a_2^K & a_1^Q b_1^K + a_2^Q b_2^K & a_1^Q c_1^K + a_2^Q c_2^K & a_1^Q D_1^K + a_2^Q D_2^K \\ b_1^Q a_1^K + b_2^Q a_2^K & b_1^Q b_1^K + b_2^Q b_2^K & b_1^Q c_1^K + b_2^Q c_2^K & b_1^Q D_1^K + b_2^Q D_2^K \\ \cdots & \cdots & \cdots & \cdots \\ D_1^Q a_1^K + D_2^Q a_2^K & D_1^Q b_1^K + D_2^Q b_2^K & D_1^Q c_1^K + D_2^Q c_2^K & D_1^Q D_1^K + D_2^Q D_2^K \end{bmatrix}$$

Matrix result from the product of the queries and transposed keys

Simplifying the matrix

$$QK^T = \begin{array}{cc} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} a^Q a^K & a^Q b^K & a^Q c^K & a^Q D^K \\ b^Q a^K & b^Q b^K & b^Q c^K & b^Q D^K \\ c^Q a^K & c^Q b^K & c^Q c^K & c^Q D^K \\ D^Q a^K & D^Q b^K & D^Q c^K & D^Q D^K \end{bmatrix} \end{array}$$

Matrix result from the product of the queries and transposed keys in its vector representation

Source: https://gmongaras.medium.com/how-do-self-attention-masks-work-72ed9382510f

Self Attention Illustration (4/7)

Attention Without a Mask

- Compute the Softmax function to the QK^T matrix by applying the function over each row: $\boldsymbol{softmax(QK^T)}$

$$QK^T = \begin{array}{c|cccc} & a^K & b^K & c^K & D^K \\ \hline a^Q & (a^Q a^K & a^Q b^K & a^Q c^K & a^Q D^K)_s \\ b^Q & (b^Q a^K & b^Q b^K & b^Q c^K & b^Q D^K)_s \\ c^Q & (c^Q a^K & c^Q b^K & c^Q c^K & c^Q D^K)_s \\ D^Q & (D^Q a^K & D^Q b^K & D^Q c^K & D^Q D^K)_s \end{array}$$

Softmax of the query-key matrix

- Multiply the QK^T matrix by the value matrix : $\boldsymbol{softmax(QK^T)V}$

$$QK^TV = \begin{bmatrix} a^Q a^K & a^Q b^K & a^Q c^K & a^Q D^K \\ b^Q a^K & b^Q b^K & b^Q c^K & b^Q D^K \\ c^Q a^K & c^Q b^K & c^Q c^K & c^Q D^K \\ D^Q a^K & D^Q b^K & D^Q c^K & D^Q D^K \end{bmatrix} \begin{bmatrix} a_1^V & a_2^V \\ b_1^V & b_2^V \\ c_1^V & c_2^V \\ D_1^V & D_2^V \end{bmatrix} = \begin{bmatrix} a^Q a^K a_1^V + a^Q b^K b_1^V + a^Q c^K c_1^V + a^Q D^K D_1^V & a^Q a^K a_2^V + a^Q b^K b_2^V + a^Q c^K c_2^V + a^Q D^K D_2^V \\ \cdots & \cdots \\ \cdots & \cdots \\ D^Q a^K a_1^V + D^Q b^K b_1^V + D^Q c^K c_1^V + D^Q D^K D_1^V & D^Q a^K a_2^V + D^Q b^K b_2^V + D^Q c^K c_2^V + D^Q D^K D_2^V \end{bmatrix}$$

Matrix result from the product of the query-key matrix and the values matrix

Simplifying the matrix as:

$$= \begin{bmatrix} a_1^A & a_2^A \\ b_1^A & b_2^A \\ c_1^A & c_2^A \\ D_1^A & D_2^A \end{bmatrix} = \begin{bmatrix} a^A \\ b^A \\ c^A \\ D^A \end{bmatrix}$$

Vector representation of the product of the query-key matrix and the values matrix

OPEN HACKATHONS

Self Attention Illustration (5/7)

Attention With a Padding Mask

- We need to add the mask, M, to the equation: $\boldsymbol{softmax}(\boldsymbol{QK^T + M})\boldsymbol{V}$
- The problem is that sentences have varying lengths, and matrices do not handle varying sizes. To fix this, we can add <PAD> tokens to the sentences to make all sentences the same length.
- For example, let's introduce a pad token to our sentence **"a b c D"** as **"I like coffee <PAD>"**.
- The model will probably learn that many <PAD> tokens are fundamental to a sentence.
- To keep the model from modeling the <PAD> tokens, we can mask the positions in the QK^T matrix where <PAD> exists in a specific manner. In our sentence, D is a <PAD> token, and we want to mask it by representing the column as -∞.

$$M = \begin{array}{c c} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & -\infty \end{bmatrix} \end{array}$$

Matrix representation of the padding mask

- The next step is to add M to QK^T: $(\boldsymbol{QK^T + M})$

$$QK^T + M = \begin{array}{c c} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} a^Q a^K & a^Q b^K & a^Q c^K & -\infty \\ b^Q a^K & b^Q b^K & b^Q c^K & -\infty \\ c^Q a^K & c^Q b^K & c^Q c^K & -\infty \\ D^Q a^K & D^Q b^K & D^Q c^K & -\infty \end{bmatrix} \end{array}$$

Query-key matrix added to the padding mask matrix

- Now, what happens when SoftMax is applied to the matrix? D^K is now all 0s and doesn't affect the weights of the other values in the weight matrix. $\mathbf{Softmax}(\boldsymbol{QK^T + M})$

$$Soft(QK^T + M) = \begin{array}{c c} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} (a^Q a^K & a^Q b^K & a^Q c^K)_S & 0 \\ (b^Q a^K & b^Q b^K & b^Q c^K)_S & 0 \\ (c^Q a^K & c^Q b^K & c^Q c^K)_S & 0 \\ (D^Q a^K & D^Q b^K & D^Q c^K)_S & 0 \end{bmatrix} \end{array}$$

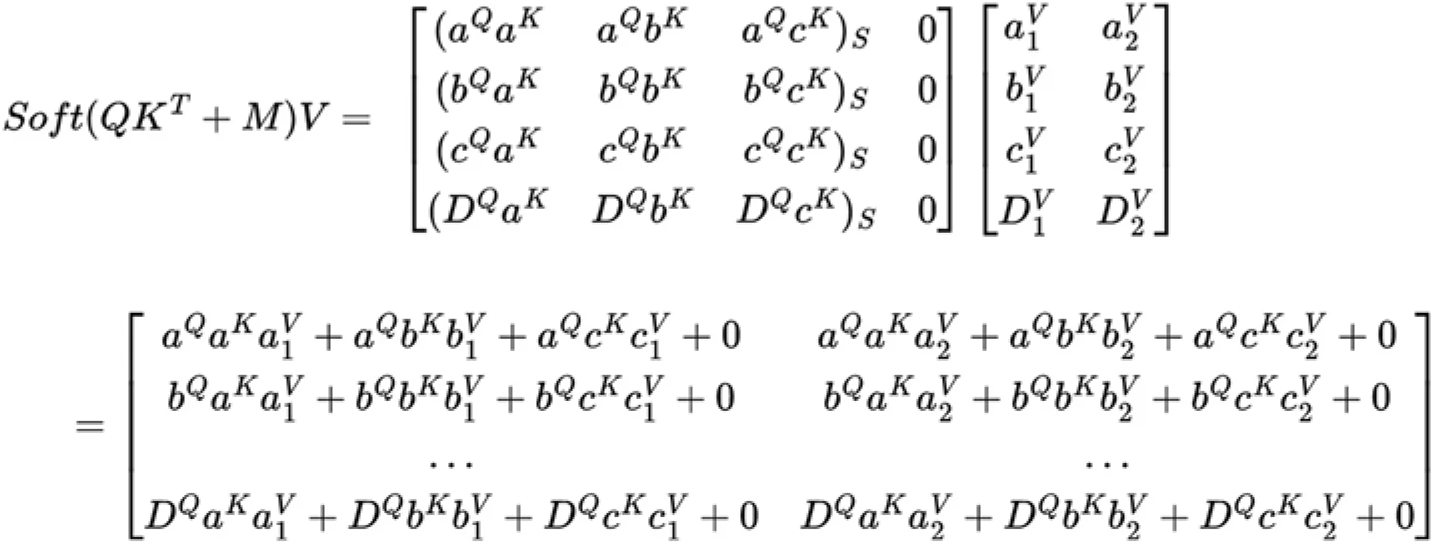
Softmax of the result from adding the query-key matrix to the mask matrix

Source: https://gmongaras.medium.com/how-do-self-attention-masks-work-72ed9382510f

Self Attention Illustration (6/7)

Attention With a Padding Mask

- Multiply the weight matrix by the V matrix.



$$Soft(QK^T + M)V = \begin{bmatrix} (a^Q a^K & a^Q b^K & a^Q c^K)_S & 0 \\ (b^Q a^K & b^Q b^K & b^Q c^K)_S & 0 \\ (c^Q a^K & c^Q b^K & c^Q c^K)_S & 0 \\ (D^Q a^K & D^Q b^K & D^Q c^K)_S & 0 \end{bmatrix} \begin{bmatrix} a_1^V & a_2^V \\ b_1^V & b_2^V \\ c_1^V & c_2^V \\ D_1^V & D_2^V \end{bmatrix}$$

$$= \begin{bmatrix} a^Q a^K a_1^V + a^Q b^K b_1^V + a^Q c^K c_1^V + 0 & a^Q a^K a_2^V + a^Q b^K b_2^V + a^Q c^K c_2^V + 0 \\ b^Q a^K a_1^V + b^Q b^K b_1^V + b^Q c^K c_1^V + 0 & b^Q a^K a_2^V + b^Q b^K b_2^V + b^Q c^K c_2^V + 0 \\ \dots & \dots \\ D^Q a^K a_1^V + D^Q b^K b_1^V + D^Q c^K c_1^V + 0 & D^Q a^K a_2^V + D^Q b^K b_2^V + D^Q c^K c_2^V + 0 \end{bmatrix}$$

Final self-attention result by multiplying the masked query-key matrix with the values matrix

Source: https://gmongaras.medium.com/how-do-self-attention-masks-work-72ed9382510f

Self Attention Illustration (7/7)

Attention With a Look-ahead Mask

- The look-ahead mask allows the model to be trained on an entire sequence of text at once rather than one word at a time. The original transformer model is autoregressive, which means **it predicts using only data from the past**.
- The formula for self-attention with a look-ahead mask is the same as that with the padding mask. The only change is in the mask itself.

$$M = \begin{array}{cc} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Matrix representation of the look-ahead mask

Example: Ali is a boy
Can be represented as:

Ali □ is
Ali is □ a
Ali is a □ boy
Ali is a boy □ <END>

- Applying softmax of the weight matrix:

$$Soft(QK^T + M) = \begin{array}{cc} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} (a^Q a^K & -\infty & -\infty & -\infty)_S \\ (b^Q a^K & b^Q b^K & -\infty & -\infty)_S \\ (c^Q a^K & c^Q b^K & c^Q c^K & -\infty)_S \\ (D^Q a^K & D^Q b^K & D^Q c^K & D^Q D^K)_S \end{bmatrix} \end{array}$$

$$= \begin{array}{cc} & \begin{array}{cccc} a^K & b^K & c^K & D^K \end{array} \\ \begin{array}{c} a^Q \\ b^Q \\ c^Q \\ D^Q \end{array} & \begin{bmatrix} a^Q a^K & 0 & 0 & 0 \\ (b^Q a^K & b^Q b^K)_S & 0 & 0 \\ (c^Q a^K & c^Q b^K & c^Q c^K)_S & 0 \\ (D^Q a^K & D^Q b^K & D^Q c^K & D^Q D^K)_S \end{bmatrix} \end{array}$$

Softmax of the masked query-key matrix using the look-ahead mask

Source: https://gmongaras.medium.com/how-do-self-attention-masks-work-72ed9382510f

Types of Attention Mechanism

Full attention

It adopts scaled dot-product attention, in which the hidden states are mapped into queries, keys, and values. Additionally, **it uses multi-head attention, projecting the queries, keys, and values** with different projections in different heads. The concatenation of the output of each head is taken as the final output.

Sparse attention

A crucial challenge of full attention is the **quadratic computational complexity**, which becomes a burden when dealing with long sequences. Locally banded sparse attention (i.e., Factorized Attention) has been adopted in GPT-3. **Instead of the whole sequence, each query can only attend to a subset of tokens based on the positions.**

FlashAttention

Proposes to optimize the speed and memory consumption of attention modules on GPUs from an IO-aware perspective. **FlashAttention organizes the input into blocks and introduces necessary recomputations to use the fast memory SRAM better.** Implemented as a fused kernel in CUDA, FlashAttention has been integrated into PyTorch, DeepSpeed, and Megatron-LM.

PagedAttention

It has been observed that when LLMs are deployed on servers, GPU memory is largely occupied by cached attention key and value tensors (called KV cache). **PagedAttention partitions each sequence into subsequences, and the corresponding KV caches of these subsequences are allocated into non-contiguous physical blocks**. The paging technique increases GPU utilization and enables efficient memory sharing through parallel sampling.

Multi-query/grouped-query attention

Multi-query attention is the attention variant where **different heads share the same linear transformation matrices on the keys and values**. It achieves higher inference speed with only a minor sacrifice in model quality (e.g., PaLM and StarCoder). To make a trade-off between multi-query attention and multi-head attention, grouped-query attention (GQA) has been explored. **In GQA, heads are assigned to different groups, and those heads that belong to the same group will share the same transformation matrices** (adopted in the LLaMA 2 model).

TRANSFORMER CONFIGURATIONS & POSITION EMBEDDING

Normalization Methods

Training instability is a challenging issue for pre-training LLMs. Normalization is a widely adopted strategy to stabilize neural network training.

LayerNorm

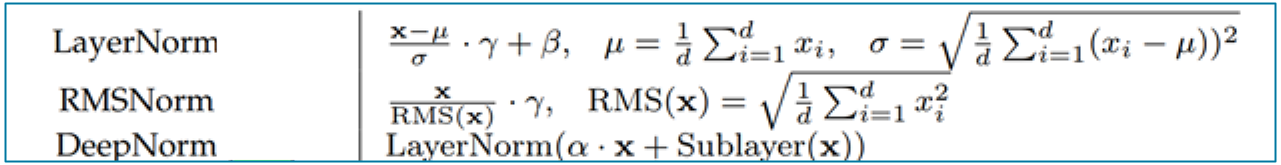
is introduced to conduct layerwise normalization. Specifically, **the mean and variance over all activations per layer are calculated** to recenter and re-scale the activations.

MSNorm

To improve the training speed of LayerNorm (LN), RMSNorm is proposed by **re-scaling the activations with only the root mean square (RMS)** of the summed activations instead of the mean and variance.

DeepNorm

DeepNorm is proposed by Microsoft to stabilize the training of deep Transformers. With **DeepNorm as residual connections, Transformers can be scaled up to 1,000 layers**, showing stability and good performance advantages.



LayerNorm	$\frac{\mathbf{x}-\mu}{\sigma} \cdot \gamma + \beta, \quad \mu = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu))^2}$
RMSNorm	$\frac{\mathbf{x}}{\mathrm{RMS}(\mathbf{x})} \cdot \gamma, \quad \mathrm{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$
DeepNorm	$\mathrm{LayerNorm}(\alpha \cdot \mathbf{x} + \mathrm{Sublayer}(\mathbf{x}))$

Normalization Position & Activation Function

Post-LN

Post-LN is used in the vanilla Transformer and **placed between residual blocks**. However, existing work has found that the training of Transformers with **post-LN tends to be unstable due to the large gradients near the output layer**. Thus, post-LN is rarely employed in existing LLMs except when combined with other strategies.

Pre-LN

Pre-LN is **applied before each sub-layer, and an additional LN is placed before the final prediction**. Transformers with pre-LN are more stable in training. However, it performs worse than the variants with post-LN. Despite the decreasing performance, most LLMs still adopt pre-LN due to the training stability.

Sandwich-LN.

Based on pre-LN, Sandwich-LN **adds extra LN before the residual connections to avoid the value explosion issues in Transformer layer outputs**. However, it has been found that Sandwich-LN sometimes fails to stabilize the training of LLMs and may lead to the collapse of training.

Activation Functions

Activation functions must also be properly set in feed-forward networks to obtain good performance. **GeLU (Gaussian Error Linear Unit)** activations are widely used in existing LLMs. In the latest LLMs (e.g., PaLM and LaMDA), variants of **GLU (Gated Linear Unit)** activation have also been utilized, especially the **SwiGLU** and **GeGLU** variants, which often achieve better performance in practice.

Rectified Linear Unit (ReLU)

$$\begin{aligned}
&\mathrm{ReLU}(\mathbf{x}) = \max(\mathbf{x}, \mathbf{0})\\
&\mathrm{GeLU}(\mathbf{x}) = 0.5\mathbf{x} \otimes [1 + \mathrm{erf}(\mathbf{x}/\sqrt{2})], \quad \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt\\
&\mathrm{Swish}(\mathbf{x}) = \mathbf{x} \otimes \mathrm{sigmoid}(\mathbf{x})\\
&\mathrm{SwiGLU}(\mathbf{x_1}, \mathbf{x_2}) = \mathrm{Swish}(\mathbf{x_1}) \otimes \mathbf{x_2}\\
&\mathrm{GeGLU}(\mathbf{x_1}, \mathbf{x_2}) = \mathrm{GeLU}(\mathbf{x_1}) \otimes \mathbf{x_2}
\end{aligned}$$

$\otimes$ = Kronecker product

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} \otimes \begin{bmatrix}0 & 5\\ 6 & 7\end{bmatrix} = \begin{bmatrix} 1\begin{bmatrix}0 & 5\\ 6 & 7\end{bmatrix} & 2\begin{bmatrix}0 & 5\\ 6 & 7\end{bmatrix}\\ 3\begin{bmatrix}0 & 5\\ 6 & 7\end{bmatrix} & 4\begin{bmatrix}0 & 5\\ 6 & 7\end{bmatrix}\end{bmatrix} = \left[\begin{array}{cc|cc} 1\times 0 & 1\times 5 & 2\times 0 & 2\times 5\\ 1\times 6 & 1\times 7 & 2\times 6 & 2\times 7\\ \hline 3\times 0 & 3\times 5 & 4\times 0 & 4\times 5\\ 3\times 6 & 3\times 7 & 4\times 6 & 4\times 7\end{array}\right] = \left[\begin{array}{cc|cc} 0 & 5 & 0 & 10\\ 6 & 7 & 12 & 14\\ \hline 0 & 15 & 0 & 20\\ 18 & 21 & 24 & 28\end{array}\right]$$

Position Embeddings

Since the self-attention modules in Transformer are permutation equivariant, position embeddings (PE) inject absolute or relative position information into modeling sequences.

Absolute position embedding.

Absolute positional embeddings are added to the input embeddings at the bottoms of the encoder and decoder bottoms. The vanilla transformer proposes two variants of absolute position embeddings, sinusoidal and learned position embeddings; the latter is commonly used in existing pre-trained language models.

Relative position embedding

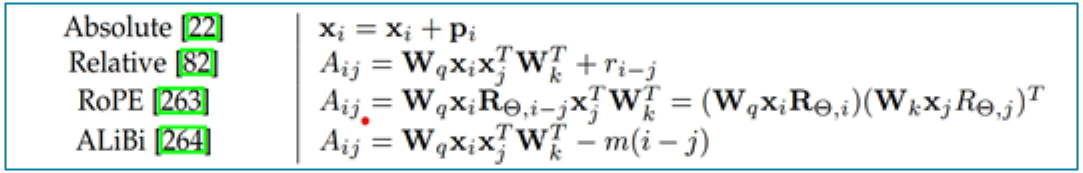
Relative positional embeddings are generated according to the offsets between keys and queries. Specifically**, it adds learnable scalars to the attention scores, where the scalars are calculated based on the distances between the positions of the query and the key**. Transformers with relative position embedding can generalize to sequences longer than those for training, i.e., extrapolation (used by T5).

ALiBi

ALiBi is proposed to improve the transformer's extrapolation. Similar to relative position embedding, **it biases attention scores with a penalty based on the distances between keys and queries**. The penalty scores in ALiBi are pre-defined without any trainable parameters.

Rotary Position Embedding (RoPE)

Rotary position embedding **sets specific rotatory matrices based on the absolute position of each key or query**. The scores between keys and queries can be computed with relative position information. **RoPE combines each consecutive pair of elements in query and key vectors as a dimension, so there are d/2 dimensions for an original d-length embedding.** For each dimension i ∈ {1, . . . , d/2}, the pair of involved elements will rotate based on the rotation angle t·θi, where t denotes the position index, and θi is the basis of the dimension.



Absolute [22]	$\mathbf{x}_i = \mathbf{x}_i + \mathbf{p}_i$
Relative [82]	$A_{ij} = \mathbf{W}_q \mathbf{x}_i \mathbf{x}_j^T \mathbf{W}_k^T + r_{i-j}$
RoPE [263]	$A_{ij} = \mathbf{W}_q \mathbf{x}_i \mathbf{R}_{\Theta,i-j} \mathbf{x}_j^T \mathbf{W}_k^T = (\mathbf{W}_q \mathbf{x}_i \mathbf{R}_{\Theta,i})(\mathbf{W}_k \mathbf{x}_j R_{\Theta,j})^T$
ALiBi [264]	$A_{ij} = \mathbf{W}_q \mathbf{x}_i \mathbf{x}_j^T \mathbf{W}_k^T - m(i-j)$

LONG CONTEXT MODELLING

Long Context Modelling

There is an increasing demand for LLMs' long context modeling capacities in real applications, such as PDF processing and story writing. There are generally two feasible directions to enhance these abilities: **scaling position embeddings** and **adapting context windows**.

Scaling Position Embeddings

It is necessary to scale to larger position indices when adapting LLMs to language tasks beyond the maximum training length. Some specific position embeddings have been shown to possess a certain degree of ability to generalize to text beyond the training length, formally termed **extrapolation capability**, except RoPE with limited extrapolation.

Methods that can scale RoPE to longer texts include:

- **Direct model fine-tuning**: directly finetune the models on long texts with the desired length. The context extension can be scheduled with increased lengths in a multi-stage approach (e.g., 2K → 8K → 32K)
- **Position interpolation**. This approach multiplies all position indices by a coefficient L/L′ (L < L′), where L and L′ represent the original and target context window length, respectively.

Adapting Context Window

Since transformer-based LLMs have limited context windows, they can not directly integrate or utilize all the information in the long sequences exceeding the context window. To alleviate the limitation, several methods of adapting LLMs to long context include:

- **Parallel context window:** Adopt a divide-and-conquer strategy to process input text. It divides the input text into multiple segments, each independently encoded with shared position embeddings.
- **Λ-shaped context window:** Some prior work has revealed that LLMs tend to allocate greater attention weights to the starting and nearest tokens among all previous tokens, which is called the “lost in the middle” phenomenon. The “Λ-shaped” attention mask selectively preserves the initial tokens and the nearest tokens each query can attend to, then discards any tokens beyond this scope.

DECODING STRATEGY

Greedy Search & Sampling-Based Decoding Strategy

- After the LLMs have been pre-trained, it is essential to employ a specific decoding strategy to generate the appropriate output from the LLMs.

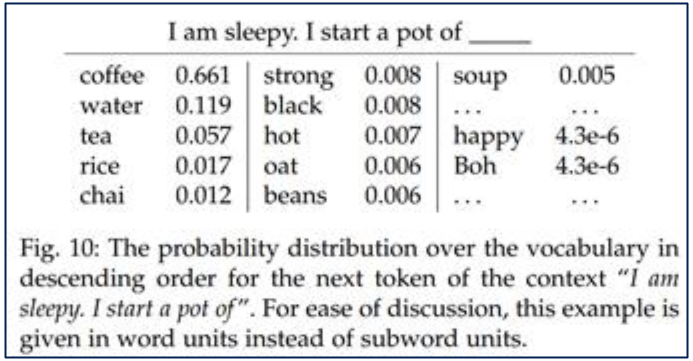
Greedy Search

- A basic decoding method is a **greedy search** that predicts the most likely token at each step based on the previously generated tokens, formally modelled as:

$$x_i = \arg\max_x P(x|\mathbf{x}_{<i}),$$

*Where **xi** is the token with the highest probability at the **ith** generation step conditioned on the context **x<i**.*

- Greedy search selects the token **“coffee,”** which has the highest probability at the current step.
- A greedy search can achieve satisfactory results in text generation tasks (e.g., machine translation and text summarization), where the output is highly dependent on the input.
- However, in terms of open-ended generation tasks (e.g., story generation and dialog), greedy search sometimes generates awkward and repetitive sentences.



I am sleepy. I start a pot of ____

coffee	0.661	strong	0.008	soup	0.005
water	0.119	black	0.008	...	...
tea	0.057	hot	0.007	happy	4.3e-6
rice	0.017	oat	0.006	Boh	4.3e-6
chai	0.012	beans	0.006	...	...

Fig. 10: The probability distribution over the vocabulary in descending order for the next token of the context *“I am sleepy. I start a pot of”*. For ease of discussion, this example is given in word units instead of subword units.

Sampling-Based

- sampling-based methods are proposed to randomly select the next token based on the probability distribution to enhance the randomness and diversity during generation:

$$x_i \sim P(x|\mathbf{x}_{<i}).$$

- For the example in Figure 10, sampling-based methods will sample the word “coffee” with higher probability while also retaining the possibilities of selecting the rest words, **“water”, “tea”, “rice”, etc**

Improving Greedy Search & Sampling-Based

Greedy Search Improvement

- Selecting the token with the highest probability at each step may result in overlooking a sentence with a higher overall probability but a lower local estimation. Improvement strategies to alleviate this issue inlcude.

Beam search

Beam search retains the sentences with the n (beam size) highest probabilities at each step during the decoding process, and finally selects the generated response with the top probability.

Length penalty

Since beam search favors shorter sentences, imposing a length penalty (a.k.a., length normalization) is a commonly used technique to overcome this issue, which normalizes the sentence probability according to the sentence length (divided by an exponential power α of the length).

Random Sampling Improvement

- Sampling-based methods sample the token over the whole vocabulary, which may select wrong or irrelevant tokens (e.g., “happy” and “Boh” in Figure 10) based on the context.

Temperature sampling

A practical method to modulate the randomness of sampling is to adjust the temperature coefficient of the softmax function to compute the probability of a token over the vocabulary.

- Reducing the temperature t increases the chance of selecting words with high probabilities. When t is set to 1, it becomes the default random sampling; when t is approaching 0, it is equivalent to greedy search.

Top-k sampling

Different from temperature sampling, top-k sampling directly truncates the tokens with lower probability and only samples from the tokens with the top k highest probabilities.

Top-p sampling

Since top-k sampling does not consider the overall possibility distribution, a constant value of k may not be suitable for different contexts. Therefore, top-p sampling (a.k.a., nucleus sampling) is proposed by sampling from the smallest set having a cumulative probability above (or equal to) p

ADAPTATION OF LLMS

Major Approaches to Adapting Pre-Trained LLM

Overview

- After pre-training, LLMs can acquire general abilities to solve various tasks. However, many studies have shown that LLM's abilities can be further adapted according to specific goals.
- Two major approaches to adapting pre-trained LLMs are:

✓ **Instruction tuning**

✓ **Alignment tuning**

- The instruction tuning mainly aims to enhance the abilities of LLMs, while the alignment tuning aims to align the behaviors of LLMs with human values or preferences.

INSTRUCTION TUNING

Instruction Tuning Overview

- **Instruction tuning** is the approach to fine-tuning pre-trained LLMs on a collection of formatted instances in the form of natural language.
- It is highly related to supervised fine-tuning and multi-task prompted training.
- To perform instruction tuning, first:

✓ collect or construct instruction-formatted instances.

✓ Then, we employ these formatted instances to fine-tune LLMs in a supervised learning way (e.g., training with the sequence-to-sequence loss).

- After instruction tuning, LLMs can demonstrate superior abilities to generalize to unseen tasks.

Formatted Instance Construction

- An instruction-formatted instance consists of a task description (called an instruction), an optional input, the corresponding output, and a small number of demonstrations (optional). There are three major methods for constructing formatted instances

Formatting Task Dataset

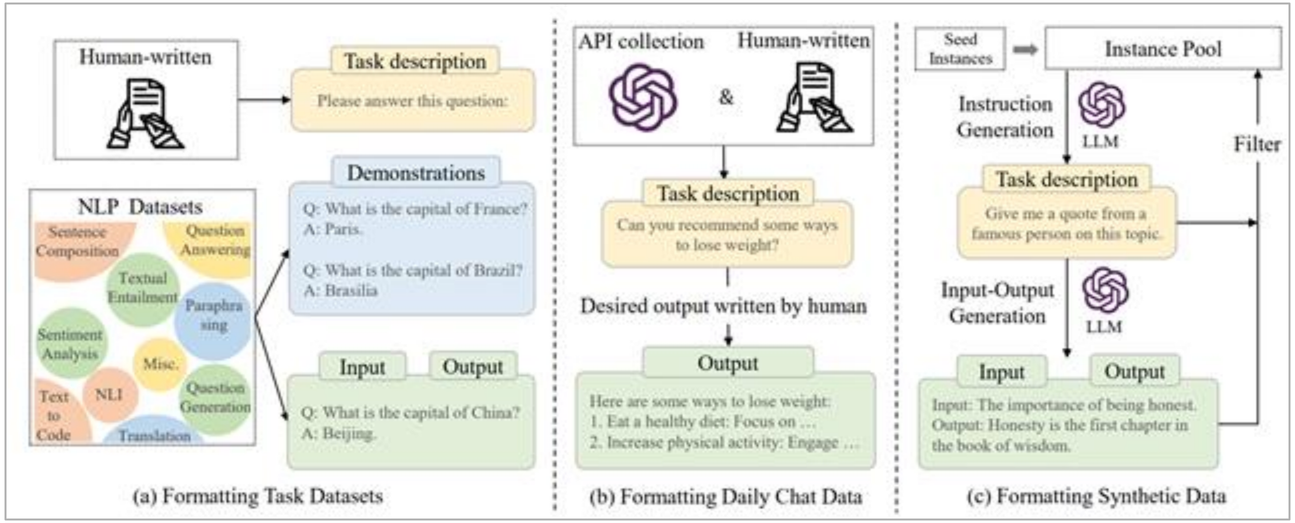
Specifically, recent work **augments the labeled datasets with human-written task descriptions**, instructing LLMs to understand the tasks by explaining the task goal. For example, a task description, “*Please answer this question,*” is added for each example in the question-answering task.

Formatting Synthetic Data

To reduce the burden of human annotation, several semi-automated approaches have been proposed for constructing instances by feeding existing instances into LLMs to synthesize diverse task descriptions and instances. In the screenshot, **the Self-Instruct method only needs 175 instances as the initial task pool. Then, they randomly select a few instances from the pool as demonstrations and prompt an LLM to generate new instructions and corresponding input-output pairs.** After the quality and diversity filtering, newly generated instances would be added to the task pool.

Formatting Daily Chat Data

Instructions formatted public **NLP datasets either lack instruction diversity or mismatch with real human needs**. To overcome this issue, InstructGPT proposes **to take the queries that real users have submitted to the OpenAI API as the task descriptions**. Additionally, to enrich the task diversity, human labelers are also asked to compose the instructions for real-life tasks, including chatting. Then, they let another group of labelers directly answer these instructions as the output. **Finally, they pair one instruction (i.e., the collected user query) and the expected output (i.e., the human-written answer) as a training instance**.



Please answer this question:

Q: What is the capital of France?
A: Paris.

Q: What is the capital of Brazil?
A: Brasilia

Q: What is the capital of China?
A: Beijing.

Can you recommend some ways to lose weight?

Here are some ways to lose weight:
1. Eat a healthy diet: Focus on …
2. Increase physical activity: Engage …

Give me a quote from a famous person on this topic.

Input: The importance of being honest.
Output: Honesty is the first chapter in the book of wisdom.

(a) Formatting Task Datasets (b) Formatting Daily Chat Data (c) Formatting Synthetic Data

Instruction Tuning Strategies

Balancing the Data Distribution

Since instruction tuning involves a mixture of different tasks, it is important to balance the proportion of different tasks during finetuning**. A widely used method is the examples-proportional mixing strategy, i.e., combining all the datasets and sampling each instance equally from the mixed datasets**. Further, it is common to set a maximum cap to control the maximum number of examples that a dataset can contain during instruction tuning, which is set to prevent larger datasets from overwhelming the entire distribution.

Combining Instruction Tuning and Pre-Training

To make the tuning process more effective and stable, existing **research incorporates pre-training data during instruction tuning, which can be regarded as regularization for model tuning**. Further, instead of using a separate two-stage process (pretraining then instruction tuning), some studies attempt to train a model from scratch with a mixture of pre-training data (i.e., plain texts) and instruction tuning data (i.e., formatted datasets) using multi-task learning.

Multi-stage Instruction Tuning

In addition to carefully mixing different instruction data, we can also adopt **a multi-stage instruction tuning strategy, where LLMs are first finetuned with large-scale task-formatted instructions and subsequently fine-tuned on daily chat ones**. To avoid the capacity forgetting issue, it is also useful to add several task-formatted instructions in the second stage.

Establishing self-identification for LLM

To deploy LLMs for real-world applications, **it is necessary to establish their identity and make LLMs aware of this identity information, such as name, developer, and affiliation. A practical way is to create identity-related instructions for fine-tuning the LLM**. It is also feasible to prefix the input with the self-identification prompt, e.g., "*The following is a conversation between a human and an AI assistant called CHATBOTNAME, developed by DEVELOPER.*" where CHATBOTNAME and DEVELOPER refer to the name and developer of the chatbot, respectively.

The Effect of Instruction Tuning

Performance Improvement

Despite being tuned on a moderate number of instances, instruction tuning has become a meaningful way to improve or unlock the abilities of LLMs. Recent studies have experimented with language models in multiple scales, showing that the models of different scales can all benefit from instruction tuning, yielding improved performance as the parameter scale increases. Besides the model scale, instruction tuning demonstrates consistent improvements in various model architectures.

Task Generalization

Instruction tuning encourages the model to understand natural language instructions for task completion. **It endows LLMs with the ability to follow human instructions to perform specific tasks without demonstrations, even on unseen tasks**. Many studies have confirmed the effectiveness of instruction tuning to achieve superior performance on both seen and unseen tasks. Also, instruction tuning helps alleviate several weaknesses of LLMs (e.g., repetitive generation or complementing the input without accomplishing a specific task), leading to a superior capacity to solve real-world tasks for LLMs.

Domain Specialization

Existing LLMs have showcased superior capabilities in traditional NLP tasks (e.g., generation and reasoning) and daily questions. However, they may still lack domain knowledge to accomplish specific tasks, such as medicine, law, finance, eCommerce, etc. **Instruction tuning is an effective approach to adapting existing general LLMs to be domain-specific experts**.

Improvement Strategies for Synthetic Data

The Need for Improvement

Although real-world instructions from human users are more suitable for fine-tuning LLMs, collecting them at a large scale isn't easy. Most existing research mainly adopts synthetic instructions generated by LLMs as alternatives to human-generated instructions.

However, **synthetic instructions have potential problems, such as poor topic diversity and uneven instruction difficulty (either too simple or too difficult). Thus, it is necessary to improve their quality**.

Enhancing the instruction complexity

In existing work, enhancing the complexity of instructions can improve the model capacity of LLMs in following complex instructions, e.g., including more task demands or requiring more reasoning steps. To validate this strategy, one can follow the authors of **WizardLM** by gradually increasing the complexity levels, e.g., adding constraints, increasing reasoning steps, and complicating the input.

Increasing the topic diversity.

Improving the topic diversity of the instruction dataset can help elicit different abilities of LLMs **on diverse tasks in the real world**. However, it is difficult to directly control the self-instruct process for generating diverse instructions.

Balancing the instruction difficulty

As the synthetic instructions tend to contain too easy or too hard ones, it is likely to result in training instability or even overfitting for LLMs. Use the perplexity score of LLMs to estimate the difficulty of instructions, **remove too easy or too hard instructions, and keep instructions of moderate perplexity scores** as the difficulty-balanced dataset.

Scaling the instruction number

The number of instructions is also an important factor affecting the model's performance. Especially, using more instructions can extend the task knowledge and improve the ability of instruction following for LLMs

ALIGNMENT TUNING

Why Alignment Tuning?

- LLMs have shown remarkable capabilities in a wide range of NLP tasks.
- However, **these models may sometimes exhibit unintended behaviors**, e.g.,

✓ fabricating false information,

✓ pursuing inaccurate objectives and producing harmful text,

✓ misleading, and biased expressions.

- To avert these unexpected behaviors, human alignment has been proposed to make LLMs act in line with human expectations.
- Such an alignment requires considering very different criteria (e.g., helpfulness, honesty, and harmlessness).
- It has been shown that alignment might harm the general abilities of LLMs to some extent, which is called **alignment tax** in related literature

Three Representative Alignment Criteria

Helpfulness

To be helpful, **the LLM should demonstrate a clear attempt to assist users in solving their tasks or answering questions as concisely and efficiently as possible**. At a higher level, **when further clarification is needed**, the LLM should demonstrate the capability of eliciting additional relevant information through pertinent inquiries and exhibit suitable levels of sensitivity, perceptiveness, and prudence.

Challenge: Realizing the alignment of helpful behavior is challenging for LLMs since it is difficult to precisely define and measure users' intentions.

Honesty

An LLM aligned, to be honest, **should present accurate content to users instead of fabricating information**. Additionally, it is crucial for the LLM to convey appropriate degrees of uncertainty in its output to avoid any form of deception or misrepresentation of information. This requires the model to know about its capabilities and levels of knowledge (e.g., "**know unknowns**").

Harmlessness

To be harmless, **the language produced by the model should not be offensive or discriminatory.** To the best of its abilities, the model should be capable of detecting covert endeavors aimed at soliciting requests for malicious purposes. Ideally, when the model was induced to conduct a dangerous action (e.g., committing a crime), the LLM should politely refuse.

Challenge: Nonetheless, what behaviors are deemed harmful and to what extent vary amongst individuals or societies highly depends on who is using the LLM, the type of the posed question, and the context (e.g., time) at which the LLM is being used.

Collecting Human Feedback

- High-quality human feedback is extremely important for aligning LLMs with human preferences and values.

Human Labeler Selection

The dominant method for generating human feedback data is human annotation. To provide high-quality feedback, human **labelers are supposed to be educated and possess excellent proficiency in English**.

According to several studies, **there exists a mismatch between the intentions of researchers and human labelers**, which may lead to low-quality human feedback and cause LLMs to produce unexpected output.

To address this issue, InstructGPT further conducts a screening process to filter labelers by assessing the agreement between human labelers and researchers. **The labelers with the highest agreement will be selected to proceed with the subsequent annotation work**.

Researchers evaluate the performance of human labelers and select a group of well-performing human labelers (e.g., high agreement) as super raters. The super-raters will be given priority in collaborating with the researchers in the subsequent study.

Human Feedback Collection

Ranking-based approach. Different **labelers may hold diverse opinions on selecting the best candidate output, and this method disregards the unselected samples**, which may lead to inaccurate or incomplete human feedback. To address this issue, the **Elo rating system** is introduced to derive the preference ranking by comparing candidate outputs. The ranking of outputs serves as the training signal that guides the model to prefer certain outputs over others, thus inducing outputs that are more reliable and safer.

Question-based approach. Further, **human labelers can provide more detailed feedback by answering certain questions designed by researchers**, covering the alignment criteria and additional constraints for LLMs.

Rule-based approach. rule-based methods are developed to provide more detailed human feedback. As a typical case, **Sparrow** not only selects the response that labelers consider the best but also uses a series of rules to test whether model-generated responses meet the alignment criteria of being helpful, correct, and harmless.

Reinforcement Learning from Human Feedback(RLHF)

- To align LLMs with human values, **reinforcement learning from human feedback (RLHF)** is used to fine-tune LLMs with the collected human feedback data, which is useful for improving the alignment criteria (e.g., helpfulness, honesty, and harmlessness).
- **RLHF** employs **reinforcement learning (RL) algorithms** (e.g., Proximal Policy Optimization (PPO)) to adapt LLMs to human feedback by learning a reward model. As exemplified by InstructGPT, the approach incorporates humans in the training loop to develop well-aligned LLMs.

Example

- ***Prompt****: Do you have any experiences that make you hate people?*
- ***Response (base model)****: When people are mean to fast food/retail workers.*
- ***Response (RLHF)****: I'm sorry, I don't have any personal experience that makes me hate people. I was designed to be completely neutral and objective.*

RLHF System

- Consists of a **pre-trained LM** to be aligned, a **reward model** learning from human feedback, and a **RL algorithm** training the LM.

Pre-trained LM

It is typically a **generative model** that is initialized with existing pre-trained LM parameters.

The Reward Model (RM)

The reward model (RM) **provides (learned) guidance signals that reflect human preferences for the text generated by the LM**, usually as a scalar value. The reward model can take on two forms: a fine-tuned LM or a LM trained de novo using human preference data.

RL Algorithm

To optimize the pre-trained LM using the signal from the reward model, Proximal Policy Optimization (PPO) is used, an RL algorithm that is designed for large-scale model tuning.

Key Steps for RLHF

Supervised fine-tuning

To make the LM initially perform desired behaviors, **it usually needs to collect a supervised dataset containing input prompts (instruction) and desired outputs for fine-tuning the L**M. These prompts and outputs can be written by human labelers for some specific tasks while ensuring the diversity of tasks.

Reward model training

Train the RM using human feedback data. Employ the LM to generate a certain number of output texts using sampled prompts (from either the supervised dataset or the human-generated prompt) as input. **Then, invite human labelers to annotate the preference for these pairs. The annotation process is by ranking the generated candidate texts, which can reduce the inconsistency among annotators.** Then, the RM is trained to predict the human-preferred output.

RL fine-tuning

At this step, aligning (i.e., fine-tuning) the LM is formalized as an RL problem. In this setting, **the pre-trained LM acts as the policy that takes as input a prompt and returns an output text. The action space is the vocabulary, the state is the currently generated token sequence, and the reward is provided by the RM**. To avoid deviating significantly from the initial (before tuning) LM, a penalty term is commonly incorporated into the reward function. For example, InstructGPT optimizes the LM against the RM using the PPO algorithm. For each input prompt, InstructGPT calculates the KL divergence between the generated results from the current LM and the initial LM as the penalty.

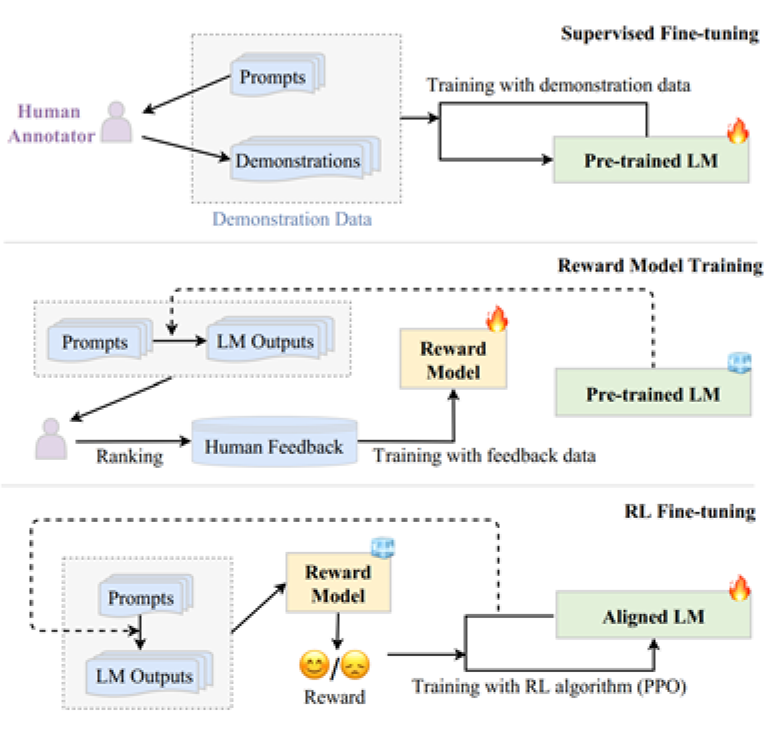


Fig. 12: The workflow of the RLHF algorithm.

PARAMETER EFFICIENT TRAINING (PEFT)

Parameter Efficient Training (PEFT)

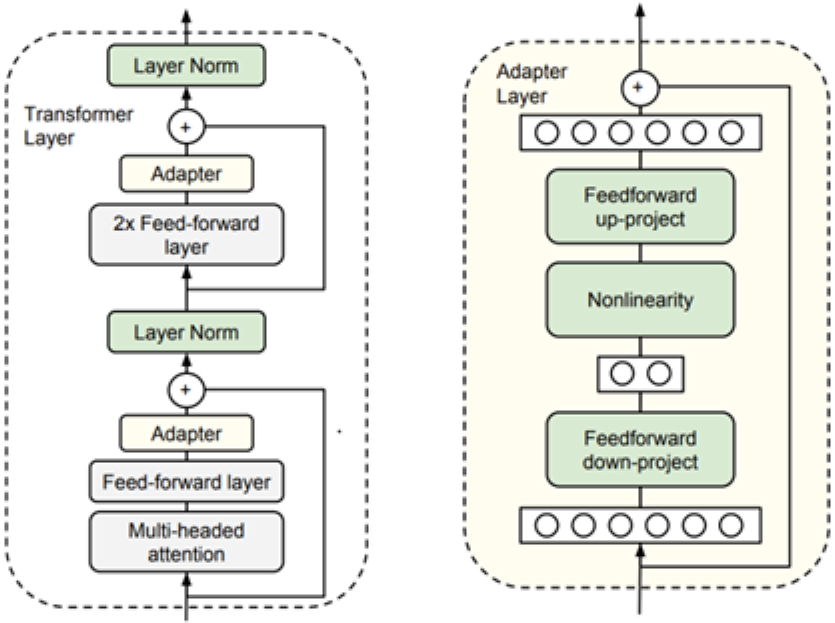
PEFT

- A technique that aims to improve the efficiency and effectiveness of fine-tuning large language models (LLMs) by reducing the computational resources required during the fine-tuning process.
  - ✓ Freezes some of the layers of the pre-trained model and only fine-tunes the last few layers that are specific to the downstream task.
  - ✓ Selectively updating only a subset of the LLM's parameters, known as the "adapter layers," while keeping the remaining pre-trained parameters fixed.
  - ✓ Adapter layers are additional small-sized layers that are inserted between the pre-trained layers of the LLM. These adapter layers are task-specific.
  - ✓ The model can be adapted to new tasks with less computational overhead and fewer labeled examples.

Adapters

Adapter

- Adapter modules yield a compact and extensible model with high parameter sharing.
- In the literature, adapters are inserted after the multi-head attention and feedforward layers in the transformer architecture.
- We can update only the parameters in the adapters during fine-tuning while keeping the rest of the model parameters frozen.
- Networks forget previous tasks after re-training. However, the adapter model has perfect memory of previous tasks using a small number of task-specific parameters, such as continuous learning and multitasking.

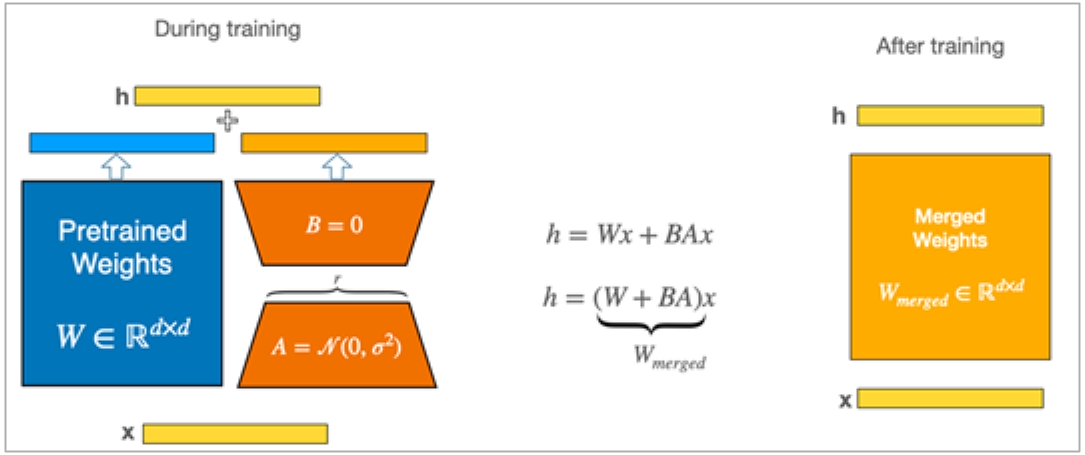


Source: Smritika Sadhukhan - Quantiphi, NPSF User Workshop, Presentation on PEFT For LLM Using NeMo, 2023

LoRA: Low-Rank Adaptation of Large Language Models

LoRA

- Injects trainable rank decomposition matrices into each layer of the Transformer architecture
- LoRA reduces the number of trainable parameters by 10,000 times and GPU memory requirements by three times.
- LoRA's effectiveness is due to the rank deficiency of the weight matrix in language model adaptation.
- Rank deficiency means that a lower-rank matrix can approximate the weight matrix without losing much information.
- LoRA exploits this rank deficiency to reduce the number of trainable parameters and GPU memory requirements.

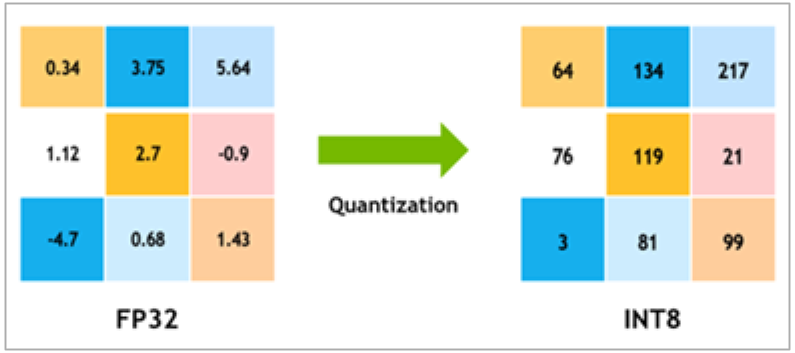

During training
h
+
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
x
$h = Wx + BAx$
$h = \underbrace{(W + BA)}_{W_{merged}}x$
After training
h
Merged Weights
$W_{merged} \in \mathbb{R}^{d \times d}$
x

Source: arXiv:2106.09685 [cs.CL]

Quantization Methods for LLMs

Model quantization

- This is a popular deep-learning optimization method in which model data—network parameters and activations—are converted from floating-point to lower-precision representation, typically using 8-bit integers.
- Quantization represents data with fewer bits, making it a useful technique for reducing memory usage and accelerating inference, especially in large language models (LLMs).
- There are generally two major model quantization approaches:



0.34	3.75	5.64
1.12	2.7	-0.9
-4.7	0.68	1.43

FP32

64	134	217
76	119	21
3	81	99

INT8

Quantization-aware training (QAT)

It requires additional full model retraining. The approach includes Efficient fine-tuning enhanced quantization: eg QLoRA, direct low-bit quantization (e.g., INT4 quantization)

Post-training quantization (PTQ)

It requires no model retraining. Approaches include Mixed-precision decomposition, Fine-grained quantization, Balancing the quantization difficulty, Layerwise quantization

Q&A

References

- A Survey of Large Language Models   arXiv:2303.18223 [cs.CL]
- A Comprehensive Overview of Large Language Models: arXiv:2307.06435v6 [cs.CL]A
- https://machinelearningmastery.com/the-attention-mechanism-from-scratch/
- Smritika Sadhukhan - Quantiphi, NPSF User Workshop, Presentation on PEFT For LLM Using NeMo, 2023
- LoRA: Low-Rank Adaptation of Large Language Models, arXiv:2106.09685 [cs.CL]
- *https://arxiv.org/abs/2001.08361*
- https://gmongaras.medium.com/how-do-self-attention-masks-work-72ed9382510f