

of LLMs heavily rely on data selection and mixture, and one can boost the proportions of specific data sources to enhance certain model abilities [64, 227]. For example, the mathematical reasoning and coding abilities can be specially enhanced by training with more mathematical texts and code data, respectively. Furthermore, experimental results on the LAMBADA dataset [252] show that increasing the proportion of books data can improve the model capacity in capturing long-term dependencies from text, and increasing the proportion of the C4 dataset [82] leads to performance improvement on the C4 validation dataset [64]. Generally, it is important to identify more implicit relations between data sources and model abilities. To enhance specific skills such as mathematics and coding in LLMs, or to develop specialized LLMs, a practical way is to employ a multi-stage training approach, *e.g.*, general and skill-specific data can be scheduled at two consecutive stages. This approach of training LLMs on varying sources or proportions of data across multiple stages is also known as “data curriculum”, which will be introduced below.

Data Curriculum. After preparing the data mixture, it is important to schedule the order that specific data is presented to LLMs for pre-training. It has been shown that, in some cases, to learn a certain skill, learning in a skill-set sequence (*e.g.*, basic skills → target skill) outperforms direct learning from a corpus focused solely on the target skill [253, 254]. Following the idea of curriculum learning [255], *data curriculum* has been proposed and widely used in model pre-training [253, 254, 256, 257]. It aims to organize different parts of pre-training data for LLMs in a specific order, *e.g.*, starting with easy/general examples and progressively introducing more challenging/specialized ones. More generally, it can broadly refer to the adaptive adjustment of data proportions for different sources during pre-training. Existing work about data curriculum mainly focuses on continual pre-training, such as specialized coding LLMs (*e.g.*, CodeLLaMA [254]) or long context LLMs (*e.g.*, LongLLaMA [257]). However, it still lacks of more detailed report about data curriculum for general-purpose LLMs (*e.g.*, LLaMA) in the literature. 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. Next, we take three common abilities as examples to introduce how the concept of data curriculum²³ applies in continual pre-training.

- **Coding.** To improve the coding ability of LLMs, CodeLLaMA [254] is developed based on LLaMA 2 [99] (2T general tokens → 500B code-heavy tokens), aiming to improve the code generation ability and retain natural language understanding skills. CodeLLaMA also provides a version that is further specialized to a certain programming language, namely CodeLLaMA-Python (2T general tokens → 500B code-heavy tokens → 100B Python-heavy tokens).

- **Mathematics.** Llemma [258] is proposed to enhance the mathematical capacities of general-purpose LLMs. It

²³ We utilize the symbol “→” to represent the data order in data curriculum. For example, “2T webpage tokens → 500B code tokens” means that the LLM is firstly trained with 2T webpage tokens and subsequently with 500B code data tokens.

is developed based on CodeLLaMA. Although CodeLLaMA [254] mainly focuses on the coding ability, experiments have shown that it performs better than its base model LLaMA 2 on mathematics benchmarks [258]. Based on CodeLLaMA, Llemma is continually trained on mixtures of scientific papers, web data containing mathematical text and code (2T general tokens → 500B code-heavy tokens → 50~200B math-heavy tokens). Note that the pre-training data of Llemma also contains 5% general domain data as a form of regularization.

- **Long context.** Long context modeling is an important ability for LLMs, and many studies have explored extending the context windows of LLMs via continually training [254, 257]. With modifications on position embeddings (*i.e.*, position interpolation) of RoPE-based LLMs [57, 99, 259], CodeLLaMA further extends the context window of LLaMA 2 (2.5T tokens with 4K context window → 20B tokens with 16K context window). LongLLaMA [257] also achieves longer context window with the help of external memory and a unique training objective (1T tokens with 2K context window → 10B tokens with 8K context window).

4.1.4 Summary of Data Preparation

In this part, we summarize the general procedure and key points to prepare pre-training data for LLMs, which are detailed in the following three aspects.

- **Data collection.** It is suggested to include diverse data sources in the pre-training data. Although Falcon [171] shows that webpages alone can be employed to train powerful LLMs, a more typical approach is to also incorporate diverse high-quality text like code, books, scientific papers, *etc.* If a LLM is specialized with a certain skill, the proportion of corresponding data source should be increased accordingly. For example, Gopher [64] and Chinchilla [34] are trained with approximately 40% of data from books. PaLM [44] and LaMDA [68] use approximately 50% conversational data.

- **Data cleaning.** After data collection, it is crucial to clean the raw corpus to enhance its quality as possible. First, deduplication is commonly used in existing work [99, 171, 248]. Second, low-quality text, toxic content, and data with privacy concerns should be removed at different granularities (*e.g.*, document, passage or sentence). In practice, both heuristic and classifier-based methods can be employed for quality and toxicity filtering (*e.g.*, CCNet [260], fastText [261], and Data-Juicer [262]). Third, with the cleaned data, one can further unify or specify the format for pre-training data, and perform the tokenization by training the tokenizer on the filtered and deduplicated corpus with libraries like SentencePiece [245].

- **Data scheduling.** With the preprocessed data, the next step is to determine the data mixture and the specific order of data for pre-training LLMs. To determine both settings, a practical way is to first train several small language models with multiple candidate plans and then select a good plan among them [59]. Overall, it is more difficult to find a suitable data curriculum. In practice, one can monitor the performance of intermediate model checkpoints on specific evaluation benchmarks, and dynamically tune the data mixture and distribution during pre-training. In this process, it is also useful to explore the potential relations between data

sources and model abilities to instruct the design of data curriculum.

4.2 Architecture

In this section, we review the architecture design of LLMs, *i.e.*, mainstream architecture, pre-training objective, and detailed configuration. Table 5 presents the model cards of several representative LLMs with public details.

4.2.1 Typical Architectures

Due to the excellent parallelizability and capacity, the Transformer architecture [22] has become the de facto backbone to develop various LLMs, making it possible to scale language models to hundreds or thousands of billions of parameters. In general, the mainstream architectures of existing LLMs can be roughly categorized into three major types, namely encoder-decoder, causal decoder, and prefix decoder, as shown in Figure 9.

Encoder-decoder Architecture. The vanilla Transformer model is built on the encoder-decoder architecture [22], which 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 for generating its latent representations, while the decoder performs cross-attention on these representations and autoregressively generates the target sequence. Encoder-decoder PLMs (*e.g.*, T5 [82] and BART [24]) have shown effectiveness on a variety of NLP tasks. So far, there are only a small number of LLMs that are built based on the encoder-decoder architecture, *e.g.*, Flan-T5 [69]. We leave a detailed discussion about the architecture selection in Section 4.2.5.

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. As representative language models of this architecture, the GPT-series models [26, 55, 122] are developed based on the causal-decoder architecture. In particular, GPT-3 [55] has successfully demonstrated the effectiveness of this architecture, also showing an amazing in-context learning capability of LLMs. Interestingly, GPT-1 [122] and GPT-2 [26] do not exhibit such superior abilities as those in GPT-3, and it seems that scaling plays an important role in increasing the model capacity of this model architecture. So far, the causal decoders have been widely adopted as the architecture of LLMs by various existing LLMs, such as OPT [90], BLOOM [78], and Gopher [64]. Note that both the causal decoder and prefix decoder discussed next belong to decoder-only architectures. When mentioning “decoder-only architecture”, it mainly refers to the causal decoder architecture in existing literature, unless specified.

Prefix Decoder Architecture. The prefix decoder architecture (*a.k.a.*, non-causal decoder [263]) revises the masking mechanism of causal decoders, to enable performing bidirectional attention over the prefix tokens [264] and unidirectional attention only on generated tokens. In this way,

like the encoder-decoder architecture, the prefix decoders can bidirectionally encode the prefix sequence and autoregressively predict the output tokens one by one, where the same parameters are shared during encoding and decoding. Instead of pre-training from scratch, a practical suggestion is to continually train causal decoders and then convert them into prefix decoders for accelerating convergence [29], *e.g.*, U-PaLM [118] is derived from PaLM [56]. Existing representative LLMs based on prefix decoders include GLM-130B [93] and U-PaLM [118].

Mixture-of-Experts. For the above three types of architectures, we can further extend them via the mixture-of-experts (MoE) scaling, in which a subset of neural network weights for each input are sparsely activated, *e.g.*, Switch Transformer [25] and GLaM [112]. The major merit is that MoE is a flexible way to scale up the model parameter while maintaining a constant computational cost [25]. It has been shown that substantial performance improvement can be observed by increasing either the number of experts or the total parameter size [265]. Despite the merits, training large MoE models may suffer from instability issues due to the complex, hard-switching nature of the routing operation. To enhance the training stability of MoE-based language models, techniques such as selectively using high-precision tensors in the routing module or initializing the model with a smaller range have been introduced [25]. More recently, there is widespread speculation that GPT-4 has been developed based on the MoE architecture, but without official verification.

Emergent Architectures. The conventional Transformer architecture typically suffers from quadratic computational complexity with respect to sequence length, resulting in a high processing cost for dealing with long inputs. To improve efficiency, recent studies aim to devise new architectures for language modeling, most based on parameterized state space models (SSM) [266], which can be viewed as a combination of RNN and CNN. On the one hand, SSM can generate outputs recursively like RNN, meaning that they only need to refer to the single previous state during decoding. It makes the decoding process more efficient as it eliminates the need to revisit all previous states as in conventional Transformers. On the other hand, these models have the capability to encode an entire sequence in parallel like Transformers via convolution computation. Thus, they can benefit from the parallelism of GPUs with techniques such as Parallel Scan [267, 268], FFT [269, 270], and Chunkwise Recurrent [271]. Despite the high computation efficiency of SSMs, their performance still lags behind Transformer. Thus, several variants of SSM have been proposed, including Mamba [272], RetNet [271], RWKV [273], and Hyena [269].

- **Mamba.** Mamba [272] aims to selectively filter out or remember information during state update. It replaces the original fixed parameters of SSM layers with functions of the input, selectively filtering out information of the previous state and the current input depending on the current input. Compared with traditional SSMs, Mamba has demonstrated improved text modeling capacities.
- **RWKV.** RWKV [273] combines the advantages of Trans-

TABLE 5: Model cards of several selected LLMs with public configuration details. Here, PE denotes position embedding, #L denotes the number of layers, #H denotes the number of attention heads, d_{model} denotes the size of hidden states, and MCL denotes the maximum context length during training.

Model	Category	Size	Normalization	PE	Activation	Bias	#L	#H	d_{model}	MCL
GPT3 [55]	Causal decoder	175B	Pre LayerNorm	Learned	GeLU	✓	96	96	12288	2048
PanGU- α [84]	Causal decoder	207B	Pre LayerNorm	Learned	GeLU	✓	64	128	16384	1024
OPT [90]	Causal decoder	175B	Pre LayerNorm	Learned	ReLU	✓	96	96	12288	2048
PaLM [56]	Causal decoder	540B	Pre LayerNorm	RoPE	SwiGLU	✗	118	48	18432	2048
BLOOM [78]	Causal decoder	176B	Pre LayerNorm	ALiBi	GeLU	✓	70	112	14336	2048
MT-NLG [113]	Causal decoder	530B	-	-	-	-	105	128	20480	2048
Gopher [64]	Causal decoder	280B	Pre RMSNorm	Relative	-	-	80	128	16384	2048
Chinchilla [34]	Causal decoder	70B	Pre RMSNorm	Relative	-	-	80	64	8192	-
Galactica [35]	Causal decoder	120B	Pre LayerNorm	Learned	GeLU	✗	96	80	10240	2048
LaMDA [68]	Causal decoder	137B	-	Relative	GeLU	-	64	128	8192	-
Jurassic-1 [107]	Causal decoder	178B	Pre LayerNorm	Learned	GeLU	✓	76	96	13824	2048
LLaMA [57]	Causal decoder	65B	Pre RMSNorm	RoPE	SwiGLU	✗	80	64	8192	2048
LLaMA 2 [99]	Causal decoder	70B	Pre RMSNorm	RoPE	SwiGLU	✗	80	64	8192	4096
Falcon [171]	Causal decoder	40B	Pre LayerNorm	RoPE	GeLU	✗	60	64	8192	2048
GLM-130B [93]	Prefix decoder	130B	Post DeepNorm	RoPE	GeGLU	✓	70	96	12288	2048
T5 [82]	Encoder-decoder	11B	Pre RMSNorm	Relative	ReLU	✗	24	128	1024	512

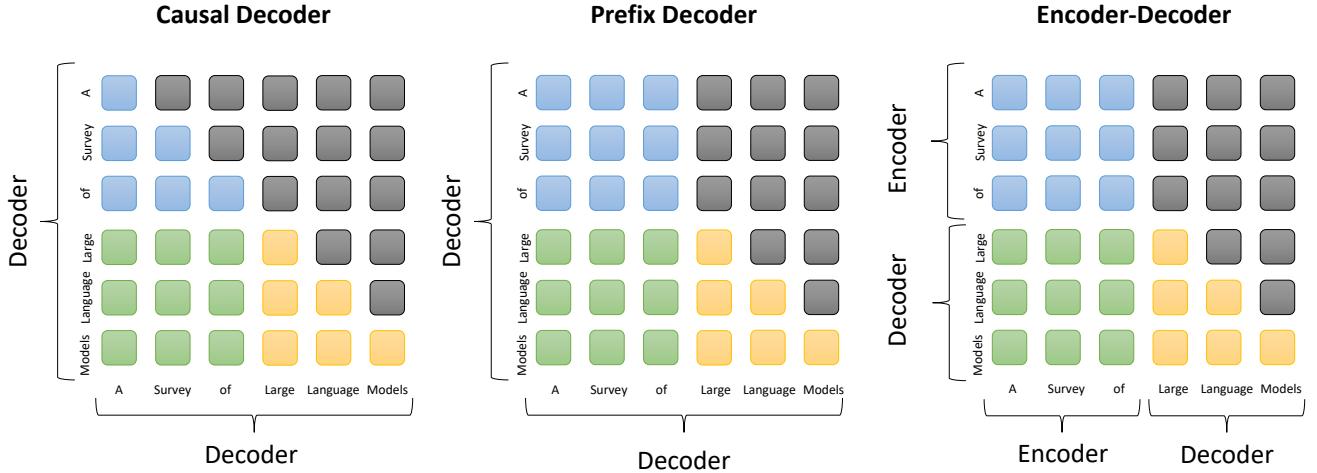


Fig. 9: A comparison of the attention patterns in three mainstream architectures. Here, the blue, green, yellow and grey rounded rectangles indicate the attention between prefix tokens, attention between prefix and target tokens, attention between target tokens, and masked attention respectively.

TABLE 6: Comparison of parallelism and complexity of different models. T represents sequence length, H represents the dimension of the input representation, N represents the dimension after compression in SSMs, and M represents the number of layers in each Hyena module.

Model	Decoding Complexity	Training Complexity
Transformer	$O(H(T + H))$	$O(TH(T + H))$
SSM	$O(H(N^2 + H))$	$O(TH(\log T + N^2 + H))$
Mamba	$O(H(N^2 + H))$	$O(TH(N^2 + H))$
RWKV	$O(H^2)$	$O(TH^2)$
RetNet	$O(H^2)$	$O(TH^2)$
Hyena	$O(MH(T + H))$	$O(TMH(\log T + H))$

former and RNN. It employs time-mixing modules, *i.e.*, RNN with gating, and channel-mixing modules that are special feedforward neural networks [273]. Within these modules, token shift, a linear combination of the current and previous token, is used instead of the token representation

as the input.

- *RetNet*. RetNet [271] proposes multi-scale retention (MSR) to replace the attention module in Transformer. Similar to linear attention, in the MSR module, the input is first mapped into query, key, and value, and the product of key and value is employed to update the state. Then, the query is used to project the state into the output. Similar to traditional SSMs, RetNet keeps the parallel and recurrent computation capacity at the same time.

- *Hyena*. Hyena employs long convolution to replace the attention module. In the long convolution module, the filters based on relative positions are used to aggregate information at different positions into the middle representations, and gating functions are employed to further project intermediate representations into the final output. However, due to the long convolution, Hyena can not infer like RNN and must explicitly access all previous states.

TABLE 7: Detailed formulations for the network configurations. Here, Sublayer denotes a FFN or a self-attention module in a Transformer layer, d denotes the size of hidden states, \mathbf{p}_i denotes position embedding at position i , A_{ij} denotes the attention score between a query and a key, r_{i-j} denotes a learnable scalar based on the offset between the query and the key, and $\mathbf{R}_{\Theta,t}$ denotes a rotary matrix with rotation degree $t \cdot \Theta$.

Configuration	Method	Equation
Normalization position	Post Norm [22]	$\text{Norm}(\mathbf{x} + \text{Sublayer}(\mathbf{x}))$
	Pre Norm [26]	$\mathbf{x} + \text{Sublayer}(\text{Norm}(\mathbf{x}))$
	Sandwich Norm [274]	$\mathbf{x} + \text{Norm}(\text{Sublayer}(\text{Norm}(\mathbf{x})))$
Normalization method	LayerNorm [275]	$\frac{\mathbf{x} - \mu}{\sigma} \cdot \gamma + \beta, \quad \mu = \frac{1}{d} \sum_{i=1}^d x_i, \quad \sigma = \sqrt{\frac{1}{d} \sum_{i=1}^d (x_i - \mu)^2}$
	RMSNorm [276]	$\frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} \cdot \gamma, \quad \text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2}$
	DeepNorm [277]	$\text{LayerNorm}(\alpha \cdot \mathbf{x} + \text{Sublayer}(\mathbf{x}))$
Activation function	ReLU [278]	$\text{ReLU}(\mathbf{x}) = \max(\mathbf{x}, \mathbf{0})$
	GeLU [279]	$\text{GeLU}(\mathbf{x}) = 0.5\mathbf{x} \otimes [1 + \text{erf}(\mathbf{x}/\sqrt{2})], \quad \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
	Swish [280]	$\text{Swish}(\mathbf{x}) = \mathbf{x} \otimes \text{sigmoid}(\mathbf{x})$
	SwiGLU [281]	$\text{SwiGLU}(\mathbf{x}_1, \mathbf{x}_2) = \text{Swish}(\mathbf{x}_1) \otimes \mathbf{x}_2$
	GeGLU [281]	$\text{GeGLU}(\mathbf{x}_1, \mathbf{x}_2) = \text{GeLU}(\mathbf{x}_1) \otimes \mathbf{x}_2$
Position embedding	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 [282]	$A_{ij} = \mathbf{W}_q \mathbf{x}_i \mathbf{R}_{\Theta,i-j} \mathbf{x}_j^T \mathbf{W}_k^T = (\mathbf{W}_q \mathbf{x}_i \mathbf{R}_{\Theta,i}) (\mathbf{W}_k \mathbf{x}_j \mathbf{R}_{\Theta,j})^T$
	ALiBi [283]	$A_{ij} = \mathbf{W}_q \mathbf{x}_i \mathbf{x}_j^T \mathbf{W}_k^T - m(i-j)$

4.2.2 Detailed Configuration

Since the launch of Transformer [22], various improvements have been proposed to enhance its training stability, performance, and computational efficiency. In this part, we will discuss the corresponding configurations for four major parts of the Transformer, including normalization, position embeddings, activation functions, and attention and bias. To make this survey more self-contained, we present the detailed formulations for these configurations in Table 7.

Normalization Methods. Training instability is a challenging issue for pre-training LLMs. To alleviate this issue, normalization is a widely adopted strategy to stabilize the training of neural networks. In the vanilla Transformer [22], LayerNorm [275] is employed. Recently, several advanced normalization techniques have been proposed as alternatives to LayerNorm, *e.g.*, RMSNorm, and DeepNorm.

- **LayerNorm.** In the early research, BatchNorm [284] is a commonly used normalization method. However, it is difficult to deal with sequence data of variable lengths and small-batch data. Thus, LayerNorm [275] is introduced to conduct layerwise normalization. Specifically, the mean and variance over all activations per layer are calculated to re-center and re-scale the activations.

- **RMSNorm.** To improve the training speed of LayerNorm (LN), RMSNorm [276] is proposed by re-scaling the activations with only the root mean square (RMS) of the summed activations, instead of the mean and variance. Related research has demonstrated its superiority in training speed and performance on Transformer [285]. Representative models that adopt RMSNorm include Gopher [64] and Chinchilla [34].

- **DeepNorm.** DeepNorm is proposed by Microsoft [277] to stabilize the training of deep Transformers. With DeepNorm as residual connections, Transformers can be scaled up to 1,000 layers [277], which has shown the advantages of stability and good performance. It has been adopted by GLM-130B [93].

Normalization Position. In addition to the normalization method, normalization position also plays a crucial role in the LLMs. There are generally three choices for the normalization position, *i.e.*, post-LN, pre-LN, and sandwich-LN.

- **Post-LN.** Post-LN is used in the vanilla Transformer [22], which is 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 [286]. Thus, post-LN is rarely employed in existing LLMs except combined with other strategies (*e.g.*, combining post-LN with pre-LN in GLM-130B [93]).

- **Pre-LN.** Different from post-LN, pre-LN [287] is applied before each sub-layer, and an additional LN is placed before the final prediction. Compared with post-LN, the Transformers with pre-LN are more stable in training. However, it performs worse than the variants with post-LN [288]. Despite the decreasing performance, most LLMs still adopt pre-LN due to the training stability. However, one exception is that pre-LN has been found unstable in GLM when training models more than 100B parameters [93].

- **Sandwich-LN.** Based on pre-LN, Sandwich-LN [274] 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 [93].

Activation Functions. To obtain good performance, activation functions also need to be properly set in feed-forward networks. In existing LLMs, GeLU activations [289] are widely used. Specially, in the latest LLMs (*e.g.*, PaLM and LaMDA), variants of GLU activation [281, 290] have also been utilized, especially the SwiGLU and GeGLU variants, which often achieve better performance in practice [285]. However, compared with GeLU, they require extra parameters (about 50%) in the feed-forward networks [291].

Position Embeddings. Since the self-attention modules in

Transformer are permutation equivariant, position embeddings (PE) are employed to inject absolute or relative position information for modeling sequences.

- *Absolute position embedding.* In the vanilla Transformer [22], absolute position embeddings are employed. At the bottoms of the encoder and the decoder, the absolute positional embeddings are added to the input embeddings. There are two variants of absolute position embeddings proposed in the vanilla Transformer [22], *i.e.*, sinusoidal and learned position embeddings, where the latter is commonly used in existing pre-trained language models.

- *Relative position embedding.* Unlike absolute position embeddings, relative positional embeddings are generated according to the offsets between keys and queries [292]. A popular variant of relative PE was introduced in Transformer-XL [293, 294]. The calculation of attention scores between keys and queries has been modified to introduce learnable embeddings corresponding to relative positions. T5 [82] further simplified relative positional embeddings, which was subsequently adopted by Gopher [64]. 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. Compared with the absolute PE, Transformers with relative position embedding can generalize to sequences longer than those sequences for training, *i.e.*, extrapolation [283].

- *Rotary position embedding.* Rotary position embedding (RoPE) [282] 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 (Table 7). 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 $i \in \{1, \dots, d/2\}$, the pair of involved elements will rotate based on the rotation angle $t \cdot \theta_i$, where t denotes the position index and θ_i is the basis in the dimension. Following sinusoidal position embeddings [22], RoPE defines the *basis* θ_i as an exponentiation of the *base* b (set to 10000 by default):

$$\Theta = \{\theta_i = b^{-2(i-1)/d} | i \in \{1, 2, \dots, d/2\}\}. \quad (4)$$

Furthermore, a recent study [295] defines the distance required to rotate one cycle (2π) for each dimension as wavelength:

$$\lambda_i = 2\pi b^{2(i-1)/d} = 2\pi/\theta_i. \quad (5)$$

Due to the excellent performance and the long-term decay property, RoPE is widely adopted in the latest LLMs, *e.g.*, PaLM [56] and LLaMA [57]. Based on RoPE, xPos [296] further improves the translation invariance and length extrapolation of Transformer. At each dimension of the rotation angle vector, xPos adds a special exponential decay that is smaller when the basis is larger. It can alleviate the unstable phenomenon during training as the distance increases.

- *ALiBi.* ALiBi [283] is proposed to improve the extrapolation of Transformer. Similar to relative position embedding, it biases attention scores with a penalty based on the distances between keys and queries. Different from the relative positional embedding methods like T5 [82], the penalty scores in ALiBi are pre-defined without any trainable parameters. Empirical results in [283] have shown that ALiBi

has a better extrapolation performance on sequences that are longer than those for training than several popular position embedding methods such as sinusoidal PE [22], RoPE [282], and T5 bias [82]. In addition, it has been shown that ALiBi can also improve training stability in BLOOM [78].

Attention. Attention mechanism is a critical component of Transformer. It allows the tokens across the sequence to interact with each other and compute the representations of the input and output sequence.

- *Full attention.* In the vanilla Transformer [22], the attention mechanism is conducted in a pairwise way, considering the relations between all token pairs in a sequence. It adopts scaled dot-product attention, in which the hidden states are mapped into queries, keys, and values. Additionally, Transformer uses multi-head attention instead of single 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. Therefore, various efficient Transformer variants are proposed to reduce the computational complexity of the attention mechanism [297, 298]. For instance, locally banded sparse attention (*i.e.*, Factorized Attention [299] has been adopted in GPT-3 [55]. Instead of the whole sequence, each query can only attend to a subset of tokens based on the positions.

- *Multi-query/grouped-query attention.* Multi-query attention refers to the attention variant where different heads share the same linear transformation matrices on the keys and values [300]. It achieves higher inference speed with only a minor sacrifice in model quality. Representative models with multi-query attention include PaLM [56] and StarCoder [98]. To make a trade-off between multi-query attention and multi-head attention, grouped-query attention (GQA) [301] has been explored. In GQA, heads are assigned into different groups, and those heads that belong to the same group will share the same transformation matrices. Specially, GQA has been adopted and empirically tested in the recently released LLaMA 2 model [99].

- *FlashAttention.* Different from most existing approximate attention methods that trade-off model quality to improve the computing efficiency, FlashAttention [302] proposes to optimize the speed and memory consumption of attention modules on GPUs from an IO-aware perspective. There exist different levels of memory on modern GPUs, *e.g.*, SRAM with a fast IO and HBM with a relatively slow IO. FlashAttention organizes the input into blocks and introduces necessary recomputation, both to make better use of the fast memory SRAM. Implemented as a fused kernel in CUDA, FlashAttention has been integrated into PyTorch [211], DeepSpeed [74], and Megatron-LM [75]. The updated version FlashAttention-2 [303] further optimizes the work partitioning of GPU thread blocks and warps, leading to around $2\times$ speedup when compared to the original FlashAttention.

- *PagedAttention.* It has been observed when LLM are deployed on servers, GPU memory is largely occupied by cached attention key and value tensors (called *KV cache*). The major reason is that the input lengths are often varied,

leading to fragmentation and over-reservation issues. Inspired by the classic paging technique in operating systems, PagedAttention has been proposed to improve the memory efficiency and throughput of deployed LLMs [304]. In detail, 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 the GPU utilization and enables efficient memory sharing in parallel sampling.

To put all these discussions together, we summarize the suggestions from existing literature for detailed configuration. For stronger generalization and training stability, it is suggested to choose the pre RMSNorm for layer normalization, and SwiGLU or GeGLU as the activation function. In addition, LN may not be used immediately after embedding layers, which is likely to incur performance degradation. As for position embeddings, RoPE or ALiBi is a better choice since it performs better on long sequences.

4.2.3 Pre-training Tasks

Pre-training plays a key role that encodes general knowledge from large-scale corpus into the massive model parameters. For training LLMs, there are two commonly used pre-training tasks, namely language modeling and denoising autoencoding.

Language Modeling. The language modeling task (LM) is the most commonly used objective to pre-train decoder-only LLMs, e.g., GPT3 [55] and PaLM [56]. Given a sequence of tokens $\mathbf{x} = \{x_1, \dots, x_n\}$, the LM task aims to autoregressively predict the target tokens x_i based on the preceding tokens $x_{<i}$ in a sequence. A general training objective is to maximize the following likelihood:

$$\mathcal{L}_{LM}(\mathbf{x}) = \sum_{i=1}^n \log P(x_i|\mathbf{x}_{<i}). \quad (6)$$

Since most language tasks can be cast as the prediction problem based on the input, these decoder-only LLMs might be potentially advantageous to implicitly learn how to accomplish these tasks in a unified LM way. Some studies have also revealed that decoder-only LLMs can be naturally transferred to certain tasks by autoregressively predicting the next tokens [26, 55], without fine-tuning. An important variant of LM is the *prefix language modeling* task, which is designed for pre-training models with the prefix decoder architecture. The tokens within a randomly selected prefix would not be used in computing the loss of prefix language modeling. With the same amount of tokens seen during pre-training, prefix language modeling performs slightly worse than language modeling, since fewer tokens in the sequence are involved for model pre-training [29].

Denoising Autoencoding. In addition to conventional LM, the denoising autoencoding task (DAE) has also been widely used to pre-train language models [24, 82]. The inputs $\mathbf{x}_{\setminus \tilde{\mathbf{x}}}$ for DAE task are corrupted text with randomly replaced spans. Then, the language models are trained to recover the replaced tokens $\tilde{\mathbf{x}}$. Formally, the training objective of DAE is denoted as follows:

$$\mathcal{L}_{DAE}(\mathbf{x}) = \log P(\tilde{\mathbf{x}}|\mathbf{x}_{\setminus \tilde{\mathbf{x}}}). \quad (7)$$

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.

However, the DAE task seems to be more complicated in implementation than LM task. As a result, it has not been widely used to pre-train large language models. Existing LLMs that take DAE as pre-training objectives include T5 [82] and GLM-130B [93]. These models are mainly trained to recover the replaced spans in an autoregressive way.

Mixture-of-Denoisers. Mixture-of-Denoisers (MoD) [89], also known as UL2 loss, was introduced as a unified objective for pre-training language models. MoD regards both LM and DAE objectives as different types of denoising tasks, namely S-denoiser (LM), R-denoiser (DAE, short span and low corruption), and X-denoiser (DAE, long span or high corruption). Among the three denoising tasks, S-denoiser is similar to the conventional LM objective (Equation (6)), while R-denoiser and X-denoiser are similar to DAE objectives (Equation (7)) but differ from each other in the lengths of spans and ratio of corrupted text. For input sentences started with different special tokens (*i.e.*, { [R], [S], [X] }), the model will be optimized using the corresponding denoisers. MoD has been applied in the latest PaLM 2 model [120].

4.2.4 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.

Background. We start the discussion with the prevalent decoder-only architecture, and introduce the auto-regressive decoding mechanism. Since such LLMs are pre-trained based on the language modeling task (Equation 6), a basic decoding method is *greedy search* that predicts the most likely token at each step based on the previously generated tokens, formally modeled as:

$$x_i = \arg \max_x P(x|\mathbf{x}_{<i}), \quad (8)$$

where x_i is the token with the highest probability at i -th step of generation conditioned on the context $\mathbf{x}_{<i}$. For instance in Figure 10, when predicting the next token of the sentence “*I am sleepy. I start a pot of*”, greedy search selects the token “coffee” which has the highest probability at the current step. Greedy search can achieve satisfactory results in text generation tasks (*e.g.*, machine translation and text summarization), in which the output is highly dependent on the input [305]. However, in terms of open-ended generation tasks (*e.g.*, story generation and dialog),

greedy search sometimes tends to generate awkward and repetitive sentences [306].

As another alternative decoding strategy, sampling-based methods are proposed to randomly select the next token based on the probability distribution to enhance the randomness and diversity during generation:

$$x_i \sim P(x|\mathbf{x}_{<i}). \quad (9)$$

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.

Not limited to the decoder-only architecture, these two decoding methods can be generally applied to encoder-decoder models and prefix decoder models in a similar way.

Improvement for Greedy Search. 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. Next, we introduce several improvement strategies to alleviate this issue.

- *Beam search.* Beam search [307] 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. Typically, the beam size is configured within the range of 3 to 6. However, opting for a larger beam size might result in a decline in performance [308].

- *Length penalty.* Since beam search favours shorter sentences, imposing length penalty (*a.k.a.*, length normalization) is a commonly used technique [309] to overcome this issue, which normalizes the sentence probability according to the sentence length (divided by an exponential power α of the length).

Besides, some researchers [310] propose to penalize the generation of previously generated tokens or n -grams to alleviate the issue of repetitive generation. In addition, diverse beam search [311] can be leveraged to produce a set of diverse outputs based on the same input.

Improvement for Random Sampling. 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. To improve the generation quality, several strategies have been proposed for mitigating or preventing the selection of words with exceedingly low probabilities.

- *Temperature sampling.* To modulate the randomness of sampling, a practical method is to adjust the temperature coefficient of the softmax function for computing the probability of the j -th token over the vocabulary:

$$P(x_j|\mathbf{x}_{<i}) = \frac{\exp(l_j/t)}{\sum_{j'} \exp(l_{j'}/t)}, \quad (10)$$

where $l_{j'}$ is the logits of each word and t is the temperature coefficient. Reducing the temperature t increases the chance of selecting words with high probabilities while decreases the chances of selecting words with low probabilities. When t is set to 1, it becomes the default random sampling; when t is approaching 0, it is equivalent to greedy search. In addition, when t goes to infinity, it degenerates to uniform sampling.

- *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 [312]. For example in Figure 10, top-5 sampling will sample from the words “coffee”, “water”, “tea”, “rice”, and “chai” from their re-scaled probabilities.

- *Top- p sampling.* Since top- k sampling does not consider the overall possibility distribution, a constant value of k may be 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 [306]. In practice, the smallest set can be constructed by gradually adding tokens from the vocabulary sorted in descending order of generative probability, until their cumulative value exceeds p .

Recently, researchers have also explored other sampling strategies for LLMs. For instance, η -sampling [313] further improves top- p sampling by introducing a dynamic threshold based on the probability distribution. Furthermore, *contrastive search* [314] and *typical sampling* [315] can be utilized to improve the generation coherence during decoding. Since it has been found that large models tend to assign higher probability to important tokens compared to small models, *contrastive decoding* [316] utilizes a larger LM (*e.g.*, OPT-13B) and a smaller LM (*e.g.*, OPT-125M) to measure their log-likelihood differences. Subsequently, tokens are sampled based on the delta value of the probability distribution, thereby amplifying the impact of important tokens. Based on this contrastive idea, DoLa [317] further extends this approach to contrasting the logits across different layers of a single LLM, as higher layers tend to assign more weight to important tokens.

Practical Settings. In practice, existing libraries (*e.g.*, Transformers [201]) and public APIs of LLMs (*e.g.*, OpenAI) have supported various decoding strategies to serve different scenarios of text generation. Next, we present the decoding settings of several representative LLMs:

- T5 [82] utilizes greedy search as the default setting and applies beam search (beam size of 4) with a length penalty of 0.6 for translation and summarization tasks.

- GPT-3 [55] employs beam search with a beam size of 4 and a length penalty of 0.6 for all generation tasks.

- Alpaca [146] utilizes sampling-based strategies with top- k ($k = 50$), top- p ($p = 0.9$), and temperature of 0.7 for open-ended generation.

- LLaMA [57] applies diverse decoding strategies tailored to specific tasks. For instance, it employs the greedy search for question answering tasks while utilizes a sampling strategy with the temperature settings of 0.1 (pass@1) and 0.8 (pass@100) for code generation.

- OpenAI API supports several basic decoding strategies, including greedy search (by setting `temperature` to 0), beam search (with the setting `best_of`), temperature sampling (with the setting `temperature`), nucleus sampling (with the setting `top_p`). It also introduce parameters `presence_penalty` and `frequency_penalty` to control the repetition degree of generation. According to the OpenAI’s document, their APIs would produce different outputs even if the input and the hyper-parameters are the same. Setting `temperature` to 0 can yield more deterministic

outputs, albeit with a slight chance of variability.

4.2.5 Summary and Discussion

The choice of architecture and pre-training tasks may incur different inductive biases for LLMs, which would lead to different model capacities. In this part, we discuss one open issue about the architecture choice for LLMs.

Why does Predicting the Next Word Works?

The essence of decoder-only architecture is to *accurately predict the next word* for reconstructing the pre-training data. Till now, there has been no formal study that theoretically demonstrates its advantage over other architectures. An interesting explanation was from Ilya Sutskever during the interview held by Jensen Huang^a. The original transcript from the interview was copied below^b:

Say you read a detective novel. It's like complicated plot, a storyline, different characters, lots of events, mysteries like clues, it's unclear. Then, let's say that at the last page of the book, the detective has gathered all the clues, gathered all the people and saying, "okay, I'm going to reveal the identity of whoever committed the crime and that person's name is". Predict that word.
...
Now, there are many different words. But predicting those words better and better, the understanding of the text keeps on increasing. GPT-4 predicts the next word better.

^a <https://www.nvidia.com/en-us/on-demand/session/gtcspring23-S52092/>

^b <https://lifearchitect.ai/ilya/>

Architecture Choice. In earlier literature of pre-trained language models, there are lots of discussions on the effects of different architectures [29, 89]. However, most LLMs are developed based on the causal decoder architecture, and there still lacks a theoretical analysis on its advantage over the other alternatives. Next, we briefly summarize existing discussions on this issue.

- By pre-training with the LM objective, it seems that causal decoder architecture can achieve a superior zero-shot and few-shot generalization capacity. Existing research has shown that without multi-task fine-tuning, the causal decoder has better zero-shot performance than other architectures [29]. The success of GPT-3 [55] has demonstrated that the large causal decoder model can be a good few-shot learner. In addition, instruction tuning and alignment tuning discussed in Section 5 have been proven to further enhance the capability of large causal decoder models [66, 67, 69].

- Scaling law has been widely observed in causal decoders. By scaling the model size, the dataset size, and

the total computation, the performance of causal decoders can be substantially improved [30, 55]. Thus, it has become an important strategy to increase the model capacity of the causal decoder via scaling. However, more detailed investigation on encoder-decoder models is still lacking, and more efforts are needed to investigate the performance of encoder-decoder models at a large scale.

More research efforts about the discussions on architectures and pre-training objectives are in need to analyze how the choices of the architecture and pre-training tasks affect the capacity of LLMs, especially for encoder-decoder architectures. Despite the effectiveness of decoder-only architecture, it is also suggested to make more diverse exploration on architecture design. Besides the major architecture, the detailed configuration of LLM is also worth attention, which has been discussed in Section 4.2.2.

4.3 Model Training

In this part, we review the important settings, techniques, or tricks for training LLMs.

4.3.1 Optimization Setting

For parameter optimization of LLMs, we present the commonly used settings for batch training, learning rate, optimizer, and training stability.

Batch Training. For language model pre-training, existing work generally sets the batch size to a large number (e.g., 2,048 examples or 4M tokens) to improve the training stability and throughput. For LLMs such as GPT-3 and PaLM, they have introduced a new strategy that dynamically increases the batch size during training, ultimately reaching a million scale. Specifically, the batch size of GPT-3 is gradually increasing from 32K to 3.2M tokens. Empirical results have demonstrated that the dynamic schedule of batch size can effectively stabilize the training process of LLMs [56].

Learning Rate. Existing LLMs usually adopt a similar learning rate schedule with the warm-up and decay strategies during pre-training. Specifically, in the initial 0.1% to 0.5% of the training steps, a linear warm-up schedule is employed for gradually increasing the learning rate to the maximum value that ranges from approximately 5×10^{-5} to 1×10^{-4} (e.g., 6×10^{-5} for GPT-3). Then, a cosine decay strategy is adopted in the subsequent steps, gradually reducing the learning rate to approximately 10% of its maximum value, until the convergence of the training loss.

Optimizer. The Adam optimizer [318] and AdamW optimizer [319] are widely utilized for training LLMs (e.g., GPT-3), which are based on adaptive estimates of lower-order moments for first-order gradient-based optimization. Commonly, its hyper-parameters are set as follows: $\beta_1 = 0.9$, $\beta_2 = 0.95$ and $\epsilon = 10^{-8}$. Meanwhile, the Adafactor optimizer [320] has also been utilized in training LLMs (e.g., PaLM and T5), which is a variant of the Adam optimizer specially designed for conserving GPU memory during training. The hyper-parameters of the Adafactor optimizer are set as: $\beta_1 = 0.9$ and $\beta_2 = 1.0 - k^{-0.8}$, where k denotes the number of training steps.

TABLE 8: Detailed optimization settings of several existing LLMs.

Model	Batch Size (#tokens)	Learning Rate	Warmup	Decay Method	Optimizer	Precision Type	Weight Decay	Grad Clip	Dropout
GPT3 (175B)	32K→3.2M	6×10^{-5}	yes	cosine decay to 10%	Adam	FP16	0.1	1.0	-
PanGu- α (200B)	-	2×10^{-5}	-	-	Adam	-	0.1	-	-
OPT (175B)	2M	1.2×10^{-4}	yes	manual decay	AdamW	FP16	0.1	-	0.1
PaLM (540B)	1M→4M	1×10^{-2}	no	inverse square root	Adafactor	BF16	lr^2	1.0	0.1
BLOOM (176B)	4M	6×10^{-5}	yes	cosine decay to 10%	Adam	BF16	0.1	1.0	0.0
MT-NLG (530B)	64 K→3.75M	5×10^{-5}	yes	cosine decay to 10%	Adam	BF16	0.1	1.0	-
Gopher (280B)	3M→6M	4×10^{-5}	yes	cosine decay to 10%	Adam	BF16	-	1.0	-
Chinchilla (70B)	1.5M→3M	1×10^{-4}	yes	cosine decay to 10%	AdamW	BF16	-	-	-
Galactica (120B)	2M	7×10^{-6}	yes	linear decay to 10%	AdamW	-	0.1	1.0	0.1
LaMDA (137B)	256K	-	-	-	-	BF16	-	-	-
Jurassic-1 (178B)	32 K→3.2M	6×10^{-5}	yes	-	-	-	-	-	-
LLaMA (65B)	4M	1.5×10^{-4}	yes	cosine decay to 10%	AdamW	-	0.1	1.0	-
LLaMA 2 (70B)	4M	1.5×10^{-4}	yes	cosine decay to 10%	AdamW	-	0.1	1.0	-
Falcon (40B)	2M	1.85×10^{-4}	yes	cosine decay to 10%	AdamW	BF16	0.1	-	-
GLM (130B)	0.4M→8.25M	8×10^{-5}	yes	cosine decay to 10%	AdamW	FP16	0.1	1.0	0.1
T5 (11B)	64K	1×10^{-2}	no	inverse square root	AdaFactor	-	-	-	0.1
ERNIE 3.0 Titan (260B)	-	1×10^{-4}	-	-	Adam	FP16	0.1	1.0	-
PanGu- Σ (1.085T)	0.5M	2×10^{-5}	yes	-	Adam	FP16	-	-	-

Stabilizing the Training. During the pre-training of LLMs, it often suffers from the training instability issue, which may cause the model collapse. To address this issue, weight decay and gradient clipping have been widely utilized, where existing studies [55, 78, 90, 93, 113] commonly set the threshold of gradient clipping to 1.0 and weight decay rate to 0.1. However, with the scaling of LLMs, the training loss spike is also more likely to occur, leading to unstable training. To mitigate this problem, PaLM [56] and OPT [90] use a simple strategy that restarts the training process from an earlier checkpoint before the occurrence of the spike and skips over the data that may have caused the problem. Further, GLM [93] finds that the abnormal gradients of the embedding layer usually lead to spikes, and proposes to shrink the embedding layer gradients to alleviate it.

4.3.2 Scalable Training Techniques

As the model and data sizes increase, it has become challenging to efficiently train LLMs under a limited computational resource. Especially, two primary technical issues are required to be resolved, *i.e.*, increasing training throughput and loading larger models into GPU memory. In this part, we review several widely used approaches in existing work to address the above two challenges, namely 3D parallelism [75, 321, 322] and mixed precision training [323], and also give general suggestions about how to utilize them for training.

3D Parallelism. 3D parallelism is actually a combination of three commonly used parallel training techniques, namely data parallelism, pipeline parallelism [321, 322], and tensor parallelism [75]²⁴. We next introduce the three parallel training techniques.

• *Data parallelism.* Data parallelism 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

24. Model parallelism is a more broader term that includes tensor parallelism and pipeline parallelism in some work [75].

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. In this way, as the calculations of gradients are independently performed on different GPUs, the data parallelism mechanism is highly scalable, enabling the way that increases the number of GPUs to improve training throughput. Furthermore, this technique is simple in implementation, and most of existing popular deep learning libraries have already implemented data parallelism, such as TensorFlow and PyTorch.

- *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, leading to the unnecessary cost of *bubbles overhead* [321]. To reduce these bubbles in pipeline parallelism, GPipe [321] and PipeDream [322] propose the techniques of padding multiple batches of data and asynchronous gradient update to improve the pipeline efficiency.

- *Tensor parallelism.* Tensor parallelism is also a commonly used technique that 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, A_1 and A_2 , by column, which can be expressed as $Y = [XA_1, XA_2]$. By placing matrices A_1 and A_2 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. Currently, tensor parallelism has been supported in several open-source libraries, *e.g.*, Megatron-LM [75], and can be extended to higher-dimensional tensors. Also, Colossal-AI has implemented tensor parallelism for higher-dimensional

tensors [324–326] and proposed sequence parallelism [327] especially for sequence data, which can further decompose the attention operation of the Transformer model.

Mixed Precision Training. In previous PLMs (*e.g.*, BERT [23]), 32-bit floating-point numbers, also known as FP32, have been predominantly used for pre-training. In recent years, to pre-train extremely large language models, some studies [323] have started to utilize 16-bit floating-point numbers (FP16), which reduces memory usage and communication overhead. Additionally, as popular NVIDIA GPUs (*e.g.*, A100) have twice the amount of FP16 computation units as FP32, the computational efficiency of FP16 can be further improved. However, existing work has found that FP16 may lead to the loss of computational accuracy [64, 78], 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. For pre-training, BF16 generally performs better than FP16 on representation accuracy [78].

Overall Training Suggestion. In practice, the above training techniques, especially 3D parallelism, are often jointly used to improve the training throughput and large model loading. For instance, researchers have incorporated 8-way data parallelism, 4-way tensor parallelism, and 12-way pipeline parallelism, enabling the training of BLOOM [78] on 384 A100 GPUs. Currently, open-source libraries like DeepSpeed [74], Colossal-AI [203], and Alpa [328] can well support the three parallel training methods. To reduce the memory redundancy, ZeRO, FSDP, and activation recomputation techniques [77, 329] can be also employed for training LLMs, which have already been integrated into DeepSpeed, PyTorch, and Megatron-LM. In addition, the mixed precision training technique such as BF16 can be also leveraged to improve the training efficiency and reduce GPU memory usage, while it requires necessary support on hardware (*e.g.*, A100 GPU). Because training large models is a time-intensive process, it would be useful to forecast the model performance and detect abnormal issues at an early stage. For this purpose, GPT-4 [46] has recently introduced a new mechanism called *predictable scaling* built on a deep learning stack, enabling the performance prediction of large models with a much smaller model, which might be quite useful for developing LLMs. In practice, one can further leverage the supporting training techniques of mainstream deep learning frameworks. For instance, PyTorch supports the data parallel training algorithm FSDP [330] (*i.e.*, fully sharded data parallel), which allows for partial offloading of training computations to CPUs if desired.

5 POST-TRAINING OF LLMs

After pre-training, LLMs can acquire the general abilities for solving various tasks. However, an increasing number of studies have shown that LLM’s abilities can be further adapted according to specific goals. In this section, we introduce two major approaches to adapting pre-trained LLMs, namely instruction tuning and alignment tuning. The former approach mainly aims to enhance (or unlock) the abilities of LLMs, while the latter approach aims to align the

behaviors of LLMs with human values or preferences. Further, we will also discuss efficient tuning and quantization for model adaptation in resource-limited settings. In what follows, we will introduce the four parts in detail.

5.1 Instruction Tuning

In essence, instruction tuning is the approach to fine-tuning pre-trained LLMs on a collection of formatted instances in the form of natural language [67], which is highly related to supervised fine-tuning [66] and multi-task prompted training [28]. In order to perform instruction tuning, we first need to 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 [28, 67, 69], even in a multilingual setting [94].

A recent survey [331] presents a systematic overview of the research on instruction tuning. In comparison to that, we mainly focus on the effect of instruction tuning on LLMs and provide detailed guidelines or strategies for instance collection and tuning. In addition, we also discuss the use of instruction tuning for satisfying the real needs of users, which has been widely applied in existing LLMs, *e.g.*, InstructGPT [66] and GPT-4 [46].

5.1.1 Formatted Instance Construction

Generally, 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). As important public resources, existing studies have released a large number of labeled data formatted in natural language (see the list of available resources in Table 3) as introduced in Section 3.3.1. Next, we introduce four major methods for constructing formatted instances (see an illustration in Figure 11) and then discuss several key factors for instance construction.

Formatting NLP Task Datasets. Before instruction tuning was proposed, several early studies [181, 332, 333] collected the instances from a diverse range of traditional NLP tasks (*e.g.*, text summarization, text classification, and translation) to create supervised multi-task training datasets. As a major source of instruction tuning instances, it is convenient to format these multi-task training datasets with natural language task descriptions. Specifically, recent work [28, 66, 67, 88] augments the labeled datasets with human-written task descriptions, which instructs LLMs to understand the tasks by explaining the task goal. For example, in Figure 11(a), a task description “*Please answer this question*” is added for each example in the question-answering task. After instruction tuning, LLMs can generalize well to other unseen tasks by following their task descriptions [28, 67, 69]. In particular, it has been shown that instructions are the crucial factor in task generalization ability for LLMs [67]: by fine-tuning the model on labeled datasets with the task descriptions removed, it results in a dramatic drop in model performance. To better generate labeled instances for instruction tuning, a crowd-sourcing platform, PromptSource [180] has been proposed to effectively create, share, and verify the task descriptions for different datasets. To enrich the training

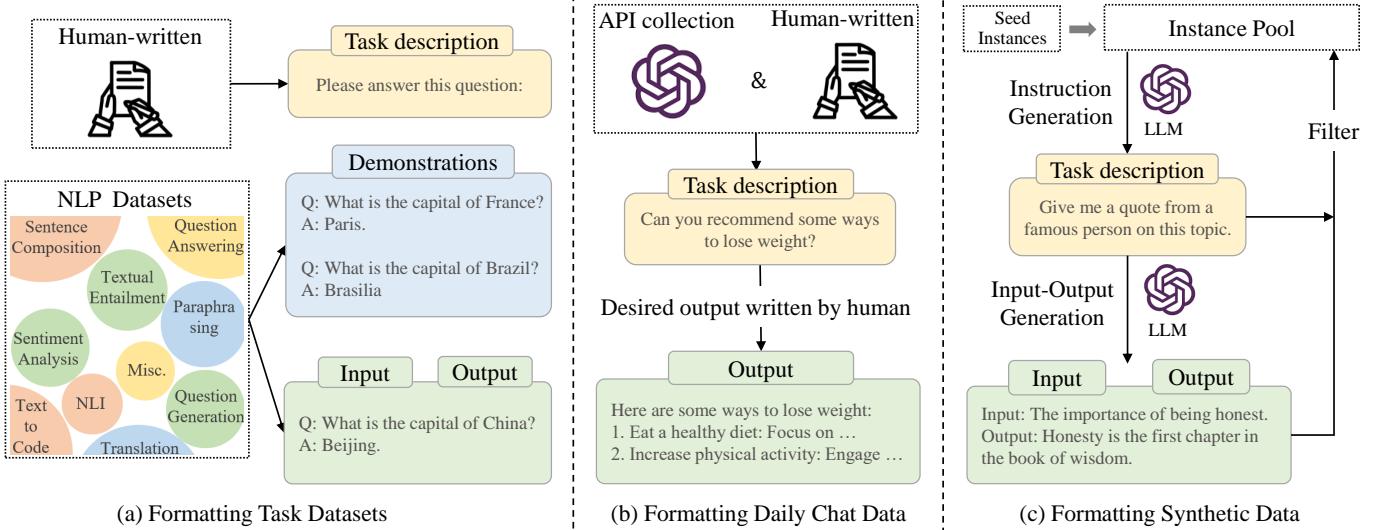


Fig. 11: An illustration of instance formatting and three different methods for constructing the instruction-formatted instances.

instances, several studies [28, 181, 334] also try to invert the input-output pairs of existing instances with specially designed task descriptions for instruction tuning. For instance, given a question-answer pair, we can create a new instance by predicting the answer-conditioned question (*e.g.*, “*Please generate a question based on the answer:*”).

Formatting Daily Chat Data. Despite that a large number of training instances have been formatted with instructions, they mainly come from public NLP datasets, either lacking instruction diversity or mismatching with real human needs [66]. To overcome this issue, InstructGPT [66] 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 open-ended generation, open question answering, brainstorming, and 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. Note that InstructGPT also employs these real-world tasks formatted in natural language for alignment tuning (discussed in Section 5.2). Further, GPT-4 [46] has designed potentially high-risk instructions and guided the model to reject these instructions through supervised fine-tuning for safety concerns. Considering the absence of high-quality public chat data, several studies have also collected users’ chat requests as input data, and then utilized ChatGPT or GPT-4 to generate responses as output data. A notable example of such a dataset is the conversational data from ShareGPT [153]. Additionally, Dolly [185] and OpenAssistant [186] have further released their conversation data, which has been carefully labeled by human annotators to attain a high level of quality.

Formatting Synthetic Data. To reduce the burden of human annotation or manual collection, several semi-automated approaches [147] have been proposed for constructing in-

stances by feeding existing instances into LLMs to synthesize diverse task descriptions and instances. As illustrated in Figure 11(c), 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 a LLM to generate new instructions and corresponding input-output pairs. After the quality and diversity filtering, newly generated instances would be added into the task pool. Hence, the synthetic method is an effective and economical way to generate large-scale instruction data for LLMs. However, the instances generated by the Self-Instruct method might be simplistic or lack the diversity. To improve the quality of synthetic instructions, WizardLM [335] introduces Evol-Instruct by proposing in-depth and in-breadth evolving to enrich the complexity and diversity of the instances. Furthermore, Self-Align [336] establishes multiple human-aligned principles to filter the synthesized instances. It then employs these instances to train a LLM in order to yield more aligned instances. To enhance the quality of the instance output, researchers directly adopt human-written texts as the output and synthesize corresponding instructions using ICL examples [337].

Key Factors for Instruction Dataset Construction. The quality of instruction instances has an important impact on the performance of the model. Here, we discuss some essential factors for instance construction.

- *Scaling the instructions.* It has been widely shown that scaling the number of tasks can largely enhance the generalization ability of LLMs [28, 67, 88]. With the increasing of the task number, the model performance initially shows a continuous growth pattern, while the gain becomes negligible when it reaches a certain level [69, 88]. A plausible speculation is that a certain number of representative tasks can provide relatively sufficient knowledge and adding more tasks may not bring additional gains [69]. Also, it is beneficial to enhance the diversity of the task descriptions in several aspects, such as length, structure, and creativity [28]. As for the number of instances per task, it has been found

that a small number of instances can usually saturate the generalization performance of the model to perform a specific task [67, 69]. Specially, several recent work [338, 339] has explored the effect of fine-tuning with a small amount of high-quality instruction data (*e.g.*, one or a few thousand instances), showing very promising results on the evaluation tasks. In contrast, another line of studies continue to explore the scaling effect of instruction data [340, 341]. For example, Orca [340] scales up the synthesized instances to 5 million with step-by-step explanations, and it achieves superior performance across a wide range of tasks.

- *Formatting design.* As an important factor, the design of natural language format also highly impacts the generalization performance of LLMs [88]. Typically, we can add task descriptions and optional demonstrations to the input-output pairs of existing datasets, where the task description is the most key part for LLMs to understand the task [88]. Further, it can lead to substantial improvements by using an appropriate number of exemplars as demonstrations [69], which also alleviates the model sensitivity to instruction engineering [67, 69]. However, incorporating other components (*e.g.*, things to avoid, reasons, and suggestions) into instructions may have a negligible or even adverse effect on the performance of LLMs [88, 179]. Recently, to elicit the step-by-step reasoning ability of LLMs, some work [69] proposes to include chain-of-thought (CoT) examples for some reasoning datasets, such as arithmetic reasoning. It has been shown that fine-tuning LLMs with both CoT and non-CoT examples can lead to a good performance across various reasoning tasks, including those that require multi-hop reasoning ability (*e.g.*, commonsense question answering and arithmetic reasoning) as well as those without the need for such a reasoning way (*e.g.*, sentiment analysis and extractive question answering) [69, 95].

- *Instruction quality improvement.* Data quality is very important for the performance of instruction tuning, and a surge of work has been proposed to further improve the quality of existing instruction datasets. Typically, these methods mostly rely on carefully designed prompts, to guide LLMs to refine or rewrite the given instruction. WizardLM [335] aims to complexify and diversify the Alpaca dataset [187] by devising prompts to widen and deepen the required knowledge of given instructions. It also crafts the filter strategy to remove the low-quality instructions. To further provide fine-grained knowledge guidance, recent work also involves the knowledge taxonomy into the input prompt, *e.g.*, knowledge key points [342] and the human-AI conversation topic taxonomy [343]. To guarantee the instruction quality, early methods mainly employ close-source API or powerful open-source LLMs, which would take a huge cost for large-scale instructions synthesis. Considering this issue, recent studies widely explore the potential of relatively small models for data synthesis. For instance, JiuZhang3.0 [344] fine-tunes a 7B language model to synthesize questions by distilling the knowledge from GPT-4, and then utilizes it to synthesize massive high-quality instructions based on pre-training corpus. Such a way can achieve better performance on mathematical reasoning tasks than baseline methods, with only 20% data synthesis cost.

- *Instruction selection.* As a surge of instruction datasets are proposed, it is non-trivial to select the high-quality

ones from them to construct the training dataset. Generally, existing work either leverages quality estimation metrics or employs LLMs as the judge model to rank all the instruction instances, and then selects those with relatively higher scores. Concretely, for metrics, perplexity and other heuristic measurements (*e.g.*, length) [345] have been widely used in practice, *e.g.*, we can consider removing high-perplexity or very short instructions, which might correspond to low-quality ones. To better estimate the effect of an instruction for the LLM capability, more complex metrics (*e.g.*, IFD [346]) have also been proposed, which are computed by combining multiple simple metrics. Additionally, diversity-aware sampling methods have been introduced to ensure the overall coverage of representative instruction data [347]. Besides, when downstream task data is available, cross-instance gradient similarity can be employed to measure the value of training instances for the target task. LESS [348] computes gradients for both downstream validation and training instruction data, to evaluate the contribution of instruction data based on extensions of influence function [349].

To summarize, diversity and quality of instructions are important factors to consider when scaling the number of instances [338]. As the capacities of LLMs improve, data synthesis methods have become the mainstream approach for generating large amount of instruction data. Following this trend, there are increasingly more automatically generated instruction datasets available, and selection and refining methods are key to effectively use these datasets. To help readers understand how different factors affect instruction tuning, we conduct an empirical study by experimenting with multiple specially constructed instruction datasets in Section 5.1.4.

5.1.2 Instruction Tuning Strategies

Unlike pre-training, instruction tuning is often more efficient since only a moderate number of instances are used for training. Since instruction tuning can be considered as a supervised training process, its optimization is different from pre-training in several aspects [69], such as the training objective (*i.e.*, sequence-to-sequence loss) and optimization configuration (*e.g.*, smaller batch size and learning rate), which require special attention in practice. In addition to these optimization configurations, there are also four important aspects to consider for instruction tuning:

Balancing the Data Distribution. Since instruction tuning involves a mixture of different tasks, it is important to balance the proportion of different tasks during fine-tuning. A widely used method is the *examples-proportional mixing* strategy [82], *i.e.*, combining all the datasets and sampling each instance equally from the mixed datasets. Furthermore, increasing the sampling ratio of high-quality collections (*e.g.*, FLAN [67] and P3 [180]) can generally lead to performance improvement according to recent findings [69, 95]. Further, it is common to set a *maximum cap* to control the maximum number of examples that a dataset can contain during instruction tuning [82], which is set to prevent larger datasets from overwhelming the entire distribution [82, 95]. In practice, the maximum cap is typically set to several thousands or tens of thousands

TABLE 9: Basic statistics of the required number of GPUs, tuning time, batch size (denoted as BS) per device (full tuning and LoRA tuning), and inference rate (the number of generated tokens per second). Our experiments are conducted based on two Linux servers having 8 A800-80G SXM4 GPUs with 6 NVSwitch and 8 3090-24G GPUs, respectively. The major difference between A800 and A100 lies in the NVLink interconnect speed. Thus, our estimations about training and inference efficiency would be slightly improved for A100, while the rest memory consumption would remain the same. For full tuning experiments, we use data parallel training, ZeRO Stage 3, BF16, and gradient checkpointing. Additionally, the LoRA tuning can be executed on one 80G GPU utilizing INT8 quantization with the rank setting set to 16. All the experiments are conducted with Alpaca-52K dataset by training LLaMA models three epochs. The max sequence length for both training settings is set to 512. The inference experiments are performed with the batch size set to 1.

Models	A800 Full Tuning			A800 LoRA Tuning			A800 Inference (16-bit)		3090 Inference (16-bit)		3090 Inference (8-bit)	
	#GPU	BS	Time	#GPU	BS	Time	#GPU	#Token/s	#GPU	#Token/s	#GPU	#Token/s
LLaMA (7B)	2	8	3.0h	1	80	3.5h	1	36.6	1	24.3	1	7.5
LLaMA (13B)	4	8	3.1h	1	48	5.1h	1	26.8	2	9.9	1	4.5
LLaMA (30B)	8	4	6.1h	1	24	14.3h	1	17.7	4	3.8	2	2.6
LLaMA (65B)	16	2	11.2h	1	4	60.6h	2	8.8	8	2.0	4	1.5

according to different datasets [67, 69]. Recently, it has been empirically found that existing instruction datasets (Table 3) mainly focus on enhancing LLMs’ capabilities in certain aspects, and a single dataset alone cannot lead to a comprehensive enhancement in model capacity [350]. Therefore, it is often suggested to use a mixture of existing instruction datasets to achieve a balanced improvement in different capacities, including NLP task data (*e.g.*, FLAN v2 [351]), chat data (*e.g.*, ShareGPT [153]), and synthetic data (*e.g.*, GPT4-Alpaca [352]).

Combining Instruction Tuning and Pre-Training. To make the tuning process more effective and stable, OPT-IML [95] 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 (*pre-training* 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 [82]. Specifically, GLM-130B [93] and Galactica [35] integrate instruction-formatted datasets as a small proportion of the pre-training corpora to pre-train LLMs, which potentially achieves the advantages of pre-training and instruction tuning at the same time.

Multi-stage Instruction Tuning. For instruction tuning, there are two kinds of important instruction data, namely task-formatted instructions and daily chat instructions. Generally, the former has a significantly larger volume than the latter. It is important to balance the training with the two kinds of instruction data. In addition to carefully mixing different instruction data, we can also adopt a multi-stage instruction tuning strategy [341], where LLMs are first fine-tuned 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 an amount of task-formatted instructions at the second stage. Actually, such a multi-stage tuning strategy can be also applied to other settings for instruction tuning. For example, we can schedule different fine-tuning stages with progressively increased levels on difficulty and complexity, and gradually improve the capacities of LLMs to follow complex instructions.

Other Practical Tricks. In practice, there are also several

useful strategies and tricks that are helpful to improve the fine-tuning performance of LLMs. We list several representative ones as follows:

- *Efficient training for multi-turn chat data.* Given a multi-turn chat example (the conversation between a user and chatbot), a straightforward fine-tuning way is to split it into multiple context-response pairs for training; a LLM is fine-tuned to generate the response based on the corresponding context for all splits (*i.e.*, at each utterance from the user). In such a fine-tuning way, it is apparent that there exist overlapping utterances in the split examples from a conversation. To save the training cost, Vicuna [152] has adopted an efficient way that feeds the whole conversation into the LLM, but relies on a loss mask that only computes the loss on the responses of the chatbot for training. It can significantly reduce the compute costs derived from the overlapped utterances.

- *Establishing self-identification for LLM.* To deploy LLMs for real-world applications, it is necessary to establish its identity and make LLMs aware of these 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.

In addition to the above practical strategies and tricks, existing work has also used other tricks, *e.g.*, concatenating multiple examples into a single sequence to approach the max length [353].

5.1.3 The Effect of Instruction Tuning

In this part, we discuss the effect of instruction tuning on LLMs in three major aspects.

Performance Improvement. Despite being tuned on a moderate number of instances, instruction tuning has become an important way to improve or unlock the abilities of LLMs [69]. Recent studies have experimented with language models in multiple scales (ranging from 77M to 540B), showing that the models of different scales can all benefit from instruction tuning [69, 334], yielding improved perfor-

mance as the parameter scale increases [94]. Further, smaller models with instruction tuning can even perform better than larger models without fine-tuning [28, 69]. Besides the model scale, instruction tuning demonstrates consistent improvements in various model architectures, pre-training objectives, and model adaptation methods [69]. In practice, instruction tuning offers a general approach to enhancing the abilities of existing language models [69] (including small-sized PLMs). Also, it is much less costly than pre-training, since the amount of instruction data required by LLMs is significantly smaller than pre-training data.

Task Generalization. Instruction tuning encourages the model to understand natural language instructions for task completion. It endows LLMs with the ability (often considered as an emergent ability) to follow human instructions [31] to perform specific tasks without demonstrations, even on unseen tasks [69]. A large number of studies have confirmed the effectiveness of instruction tuning to achieve superior performance on both seen and unseen tasks [95, 334]. Also, instruction tuning has been shown to be useful in alleviating several weaknesses of LLMs (*e.g.*, repetitive generation or complementing the input without accomplishing a certain task) [66, 69], leading to a superior capacity to solve real-world tasks for LLMs. Furthermore, LLMs trained with instruction tuning can generalize to related tasks across languages. For example, BLOOMZ-P3 [94] is fine-tuned based on BLOOM [78] using English-only task collection P3 [180]. Interestingly, BLOOMZ-P3 can achieve a more than 50% improvement in multilingual sentence completion tasks compared to BLOOM, which shows that instruction tuning can help LLMs acquire general task skills from English-only datasets and transfer such skills into other languages [94]. In addition, it has been found that using English-only instructions can produce satisfactory results on multilingual tasks [94], which helps reduce the effort of instruction engineering for a specific language.

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, and finance (See Section 8 for a detailed discussion of LLMs in different applications). Instruction tuning is an effective approach to adapting existing general LLMs to be domain-specific experts. For instance, researchers propose to fine-tune Flan-PaLM [69] using medical datasets to create Med-PaLM [354], a medical knowledge assistant that achieves performance levels comparable to those of expert clinicians. Furthermore, a recent study [355] fine-tunes FLAN-T5 to support e-commerce recommender systems with natural language instructions, showing strong performance in a variety of recommendation tasks. There are also several open-sourced medical models instruction-tuned based on LLaMA [57], such as BenTsao [356]. Also, researchers explore instruction tuning on law [357], finance [358], and arithmetic computation [359].

5.1.4 Empirical Analysis for Instruction Tuning

Fine-tuning LLMs with different instruction sets tend to lead to model variants with varied performance on downstream tasks. In this section, we will explore the effect of different

types of instructions in fine-tuning LLMs (*i.e.*, LLaMA (7B) and LLaMA (13B)²⁵), as well as examine the usefulness of several instruction improvement strategies.

Instruction Datasets. According to the discussion in Section 5.1.1, we mainly consider three common kinds of instructions as follows:

- *Task-specific instructions.* For the first type of instructions, we adopt the most commonly-used multi-task instruction dataset, *FLAN-T5* [69], which contains 1,836 tasks and over 15M instructions by combining four data mixtures from prior work.

• *Daily chat instructions.* This type of instructions are conversations posed by users about daily life, which are more closely related to real-life scenarios. We adopt the ShareGPT instruction set, consisting of 63K real-user instructions. It has been used as the core instructions for Vicuna.

- *Synthetic instructions.* In addition to reusing existing instructions, we can also automatically synthesize massive instructions using LLMs. We adopt the popular synthetic instruction dataset Self-Instruct-52K [147], consisting of 52K instructions paired with about 82K instance inputs and outputs. These generated instructions have a similar data distribution as the human-written seed tasks (*e.g.*, grammar checking, brainstorming).

As the original FLAN-T5 dataset is very large (*i.e.*, over 15M), we randomly sample 80,000 instructions from it for conducting a fair comparison with other instruction datasets (*i.e.*, ShareGPT and Self-Instruct-52K) at a similar scale. In our experiments, we test on each individual instruction set to explore their own effects and also examine their combinatorial effects on model performance.

Improvement Strategies. Although real-world instructions from human users are more suitable for fine-tuning LLMs, it is difficult to collect them at a large scale. As alternatives to human-generated instructions, most existing research mainly adopts synthetic instructions generated by LLMs. However, there are some potential problems with synthetic instructions, such as poor topic diversity and uneven instruction difficulty (either too simple or too difficult). Thus, it is necessary to improve the quality of the synthetic instructions. Next, we summarize four major improvement strategies widely used in existing work as follows:

- *Enhancing the instruction complexity.* As discussed in existing work [335], 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, we follow WizardLM [335] by gradually increasing the complexity levels, *e.g.*, adding constraints, increasing reasoning steps, and complicating the input. We leverage the publicly released WizardLM-70K instructions [335] as the complexity-enhanced instruction dataset, which has been generated via the above enhancement approach based on the Self-Instruct-52K dataset [335].

- *Increasing the topic diversity.* In addition to the complexity, improving the topic diversity of the instruction dataset

25. Due to the limit of computational resources, we cannot conduct large-scale experiments on larger LLaMA variants right now, which would be scheduled in a future version.

TABLE 10: Results of instruction-tuning experiments (all in a single-turn conversation) based on the LLaMA (7B) and LLaMA (13B) model under the chat and QA setting. We employ four instruction improvement strategies on the Self-Instruct-52K dataset, *i.e.*, enhancing the complexity (*w/ complexity*), increasing the diversity (*w/ diversity*), balancing the difficulty (*w/ difficulty*), and scaling the instruction number (*w/ scaling*). *Since we select the LLaMA (7B)/(13B) model fine-tuned on Self-Instruct-52K as the baseline, we omit the win rate of the fine-tuned model with Self-Instruct-52K against itself.

Models	Dataset Mixtures	Instruction Numbers	Lexical Diversity	Chat		QA	
				AlpacaFarm	MMLU	BBH3k	BBH3k
LLaMA (7B)	① FLAN-T5	80,000	48.48	23.77	38.58	32.79	
	② ShareGPT	63,184	77.31	81.30	38.11	27.71	
	③ Self-Instruct-52K	82,439	25.92	/*	37.52	29.81	
	② + ③	145,623	48.22	71.36	41.26	28.36	
	① + ② + ③	225,623	48.28	70.00	43.69	29.69	
	③ Self-Instruct-52K	82,439	25.92	/*	37.52	29.81	
	w/ complexity	70,000	70.43	76.96	39.73	33.25	
	w/ diversity	70,000	75.59	81.55	38.01	30.03	
	w/ difficulty	70,000	73.48	79.15	32.55	31.25	
	w/ scaling	220,000	57.78	51.13	33.81	26.63	
LLaMA (13B)	① FLAN-T5	80,000	48.48	22.12	34.12	34.05	
	② ShareGPT	63,184	77.31	77.13	47.49	33.82	
	③ Self-Instruct-52K	82,439	25.92	/*	36.73	25.43	
	② + ③	145,623	48.22	72.85	41.16	29.49	
	① + ② + ③	225,623	48.28	69.49	43.50	31.16	
	③ Self-Instruct-52K	82,439	25.92	/*	36.73	25.43	
	w/ complexity	70,000	70.43	77.94	46.89	35.75	
	w/ diversity	70,000	75.59	78.92	44.97	36.40	
	w/ difficulty	70,000	73.48	80.45	43.15	34.59	
	w/ scaling	220,000	57.78	58.12	38.07	27.28	

can help elicit different abilities of LLMs on diverse tasks in real world [336]. However, it is difficult to directly control the self-instruct process for generating diverse instructions. Following YuLan-Chat [341], we employ ChatGPT to rewrite the instructions from Self-Instruct-52K dataset for adapting them into 293 topics via specific prompts. Finally, we obtain 70K instructions as the diversity-increased dataset.

- *Scaling the instruction number.* In addition to the above aspects, the number of instructions is also an important factor that may affect the model performance. Specially, using more instructions can extend the task knowledge and improve the ability of instruction following for LLMs [69]. To examine this strategy, we sample new instructions from the synthesized instruction set released from the MOSS project [360], as they are also synthesized using the same self-instruct method [147]. We mix them with the Self-Instruct-52K dataset to compose a larger one containing 220K 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. To explore the potential effects, we leverage the perplexity score of LLMs to estimate the difficulty of instructions and remove too easy or too hard instructions. To generate the same scale of instructions for fair comparison, we adopt a LLaMA (7B) model to compute the perplexity for the 220K instructions from the large instruction dataset, and then keep 70K instructions of moderate perplexity scores as the difficulty-balanced dataset.

Experimental Setup. To conduct the experiments on the effect of instruction data, we leverage these new instruction

datasets for tuning LLaMA, a popular LLM backbone that has been widely used for instruction-tuning. We use the code from YuLan-Chat [341] for our experiments, and train LLaMA 7B and 13B on a server of 8 A800-80G GPUs. All the hyper-parameters settings remain the same as Stanford Alpaca. To better evaluate the instruction following ability of fine-tuned models, we consider two settings, namely *Chat setting* and *QA setting*. The chat setting mainly utilizes user instructions and queries from daily chat, whereas the QA setting mainly employs question answering examples from existing NLP datasets. The evaluation on the chat setting is conducted based on the AlpacaFarm evaluation set [361]. Instead of using a full pairwise comparison, we select the LLaMA 7B and 13B models fine-tuned on Self-Instruct-52K as the reference baselines, and then compare them with other fine-tuned LLaMA 7B and 13B models using different instructions, respectively. Since our focus is to examine the usefulness of different strategies to generate the instructions, the model fine-tuned on Self-Instruct-52K can serve as a good reference. Following AlpacaFarm [361], for each comparison, we employ ChatGPT to automatically annotate which response from two compared models each time is the best for the user query, and report the win rate (%) as the evaluation metric. For the QA setting, we select two benchmarks, MMLU [362] and BBH [363], and evaluate the accuracy based on their default settings by using heuristic rules to parse the answers from these LLMs.

For both instruction tuning and evaluation, we adopt the following prompt: “The following is a conversation between a human and an AI assistant. The AI assistant gives helpful, detailed, and polite answers to the user’s questions.\n

`[|Human|]:{input}\n[|AI|]:". To reproduce our results, we release the code and data at the link: https://github.com/RUCAIBox/LLMSurvey/tree/main/Experiments.`

Results and Analysis. The results using different instruction datasets based on 7B and 13B LLaMA are in Table 10. Next, we summarize and analyze our findings in detail.

- *Task-formatted instructions are more proper for the QA setting, but may not be useful for the chat setting.* By comparing the performance of instruction tuning using FLAN-T5 with that of ShareGPT and Self-Instruct-52K, we can observe that FLAN-T5 mostly achieves a better performance on QA benchmarks while underperforms ShareGPT on the chat setting. The reason is that FLAN-T5 is composed of a mixture of instructions and examples from existing NLP tasks, *e.g.*, translation and reading comprehension. As a result, LLaMA fine-tuned with FLAN-T5 performs better on QA tasks, but poorly on user queries. In contrast, ShareGPT consists of real-world human-ChatGPT conversations, which is able to better elicit LLaMA to follow user instructions in daily life, while may not be suitable for accomplishing the QA tasks.

- *A mixture of different kinds of instructions are helpful to improve the comprehensive abilities of LLMs.* After mixing the three kinds of instructions for fine-tuning, we can see that the derived LLaMA variant (with FLAN-T5, ShareGPT and Self-Instruct-52K) performs well in both task settings. In MMLU, the performance of LLaMA (7B) can surpass the ones using individual instruction set by a large margin, *i.e.*, 43.69 vs. 38.58 (FLAN-T5). It shows that mixing multiple sources of instruction datasets is helpful to improve the performance of instruction-tuned LLMs, which scales the instruction number as well as increases the diversity.

- *Enhancing the complexity and diversity of instructions leads to an improved model performance.* By increasing the complexity and diversity of the Self-Instruct-52K dataset respectively, the chat and QA performance of LLaMA can be consistently improved, *e.g.*, from 37.52 to 39.73 in MMLU for LLaMA (7B). It demonstrates that both strategies are useful to improve the instruction following ability of LLMs. Further, we can see that improving the complexity yields a larger performance improvement on QA tasks. The reason is that the QA tasks mostly consist of difficult questions for evaluating LLMs, which can be better solved by LLMs that have learned complex instructions at the fine-tuning stage.

- *Simply increasing the number of instructions may not be that useful, and balancing the difficulty is not always helpful.* As the results shown in Table 10, balancing the difficulty and increasing the number of fine-tuning instructions are not very helpful in our experiments. Especially for scaling the instruction number, it even hurts the performance, *e.g.*, a decrease from 29.81 to 26.63 in BBH for LLaMA (7B). It shows that simply scaling the number of synthesized instructions without quality control may not be effective to improve the performance. Furthermore, fine-tuning with the instructions of moderate difficulty also performs well in the chat setting, while slightly decreasing the performance in the QA setting. A possible reason is that we filter complex and hard instructions with large perplexity scores, hurting the model performance in answering complex questions.

- *A larger model scale leads to a better instruction following performance.* By comparing the performance of LLaMA (7B)

and LLaMA (13B) models fine-tuned with the same set of instruction data, we can see that LLaMA (13B) mostly achieves a better performance. It indicates that scaling the model size is helpful for improving the instruction following capability. Besides, we can see that the QA performance has been improved a lot, *e.g.*, from 38.11 to 47.49 in MMLU. It is likely because that the larger models generally have better knowledge utilization and reasoning capability [33, 55], which can accurately answer more complex questions.

Instruction Tuning Suggestions

To conduct instruction tuning on LLMs, one can prepare the computational resources according to the basic statistics about the required number of GPUs and tuning time in Table 9. After setting up the development environment, we recommend beginners to follow the code of Alpaca repository [187] for instruction tuning. Subsequently, one should select the base model and construct the instruction datasets as we discuss in this section. When computational resources for training are constrained, users can utilize LoRA for parameter-efficient tuning (see Section 5.3). As for inference, users can further use quantization methods to deploy LLMs on fewer or smaller GPUs (see Section 5.3).

5.2 Alignment Tuning

This part first presents the background of alignment with its definition and criteria, then focuses on the collection of human feedback data for aligning LLMs, and finally discusses the key technique of reinforcement learning from human feedback (RLHF) for alignment tuning.

5.2.1 Background and Criteria for Alignment

Background. LLMs have shown remarkable capabilities in a wide range of NLP tasks [55, 56, 67, 90]. However, these models may sometimes exhibit unintended behaviors, *e.g.*, fabricating false information, pursuing inaccurate objectives, and producing harmful, misleading, and biased expressions [66, 364]. For LLMs, the language modeling objective pre-trains the model parameters by word prediction while lacking the consideration of human values or preferences. To avert these unexpected behaviors, human alignment has been proposed to make LLMs act in line with human expectations [66, 365]. However, unlike the original pre-training and adaptation tuning (*e.g.*, instruction tuning), 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 [366].

Alignment Criteria. Recently, there is increasing attention on developing multifarious criteria to regulate the behaviors of LLMs. Here, we take three representative alignment criteria (*i.e.*, helpful, honest, and harmless) as examples for discussion, which have been widely adopted in existing literature [66, 366]. In addition, there are other alignment

criteria for LLMs from different perspectives including behavior, intent, incentive, and inner aspects [364], which are essentially similar (or at least with similar alignment techniques) to the above three criteria. It is also feasible to modify the three criteria according to specific needs, *e.g.*, substituting honesty with correctness [116]. Next, we give brief explanations about the 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 in a concise and efficient manner 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 [366]. Realizing the alignment of helpful behavior is challenging for LLMs since it is difficult to precisely define and measure the intention of users [364].

- *Honesty*. At a basic level, a 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, in order 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”). According to the discussion in [366], honesty is a more objective criterion compared to helpfulness and harmlessness, hence honesty alignment could potentially be developed with less reliance on human efforts.

- *Harmlessness*. To be harmless, it requires that 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. Nonetheless, *what behaviors* are deemed harmful and *to what extent* vary amongst individuals or societies [366] highly depend 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.

As we can see, these criteria are quite subjective, and are developed based on human cognition. Thus, it is difficult to directly formulate them as optimization objectives for LLMs. In existing work, there are many ways to fulfill these criteria when aligning LLMs. A promising technique is *red teaming* [367], which involves using manual or automated means to probe LLMs in an adversarial way to generate harmful outputs and then updates LLMs to prevent such outputs.

5.2.2 Collecting Human Feedback

During the pre-training stage, LLMs are trained using the language modeling objective on a large-scale corpus. However, it cannot take into account the subjective and qualitative evaluations of LLM outputs by humans (called *human feedback* in this survey). High-quality human feedback is extremely important for aligning LLMs with human preferences and values. In this part, we discuss how to select a team of human labelers for feedback data collection.

Human Labeler Selection. In existing work, the dominant method for generating human feedback data is human annotation [66, 116, 365]. This highlights the critical role of selecting appropriate human labelers. To provide high-quality feedback, human labelers are supposed to have a qualified level of education and excellent proficiency in English. For example, Sparrow [116] requires human labelers to be UK-based native English speakers who have obtained at least an undergraduate-level educational qualification. Even then, several studies [365] have found that there still 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 [66] further conducts a screening process to filter labelers by assessing the agreement between human labelers and researchers. Specifically, researchers first label a small amount of data and then measure the agreement between themselves and human labelers. The labelers with the highest agreement will be selected to proceed with the subsequent annotation work. In some other work [368], “super raters” are used to ensure the high quality of human feedback. 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 to collaborate with the researchers in the subsequent study. When human labelers annotate the output of LLMs, it is helpful to specify detailed instructions and provide instant guidance for human labelers, which can further regulate the annotation of labelers.

Human Feedback Collection. In existing work, there are mainly three kinds of approaches to collecting feedback and preference data from human labelers.

- *Ranking-based approach*. In early work [365], human labelers often evaluate model-generated outputs in a coarse-grained manner (*i.e.*, only selecting the best) without taking into account more fine-grained alignment criteria. Nonetheless, different labelers may hold diverse opinions on the selection of the best candidate output, and this method disregards the unselected samples, which may lead to inaccurate or incomplete human feedback. To address this issue, subsequent studies [116] introduce the Elo rating system 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 [81], covering the alignment criteria as well as additional constraints for LLMs. Specially, in WebGPT [81], to assist the model in filtering and utilizing relevant information from retrieved documents, human labelers are required to answer questions with multiple options about whether the retrieved documents are useful for answering the given input.

- *Rule-based approach*. Many studies also develop rule-based methods to provide more detailed human feedback. As a typical case, Sparrow [116] 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

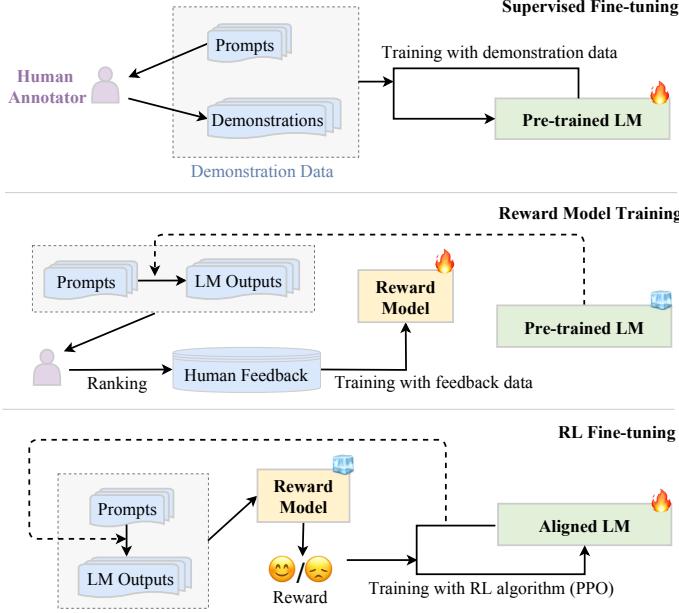


Fig. 12: The workflow of the RLHF algorithm.

alignment criteria of being helpful, correct, and harmless. In this way, two kinds of human feedback data can be obtained: (1) the response preference feedback is obtained by comparing the quality of model-generated output in pairs, and (2) the rule violation feedback is obtained by collecting the assessment from human labelers (*i.e.*, a score indicating to what extent the generated output has violated the rules). Furthermore, GPT-4 [46] utilizes a set of zero-shot classifiers (based on GPT-4 itself) as rule-based reward models, which can automatically determine whether the model-generated outputs violate a set of human-written rules.

In the following, we focus on a well-known technique, reinforcement learning from human feedback (RLHF), which has been widely used in the recent powerful LLMs such as ChatGPT. As discussed below, the alignment criteria introduced in Section 5.2.1 can be fulfilled by learning from human feedback on the responses of LLMs to users' queries.

5.2.3 Reinforcement Learning from Human Feedback

To align LLMs with human values, reinforcement learning from human feedback (RLHF) [79, 365] has been proposed to fine-tune LLMs with the collected human feedback data, which is useful to improve the alignment criteria (*e.g.*, helpfulness, honesty, and harmlessness). RLHF employs reinforcement learning (RL) algorithms (*e.g.*, Proximal Policy Optimization (PPO) [128]) to adapt LLMs to human feedback by learning a reward model. Such an approach incorporates humans in the training loop for developing well-aligned LLMs, as exemplified by InstructGPT [66].

RLHF System. The RLHF system mainly comprises three key components: a pre-trained LM to be aligned, a reward model learning from human feedback, and a RL algorithm training the LM. Specifically, the *pre-trained LM* is typically a generative model that is initialized with existing pre-trained LM parameters. For example, OpenAI uses 175B GPT-3 for its first popular RLHF model, InstructGPT [66],

and DeepMind uses the 280 billion parameter model Gopher [64] for its GopherCite model [368]. Further, the *reward model* (RM) provides (learned) guidance signals that reflect human preferences for the text generated by the LM, usually in the form of 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. Existing work typically employs reward models having a parameter scale different from that of the aligned LM [66, 368]. For example, OpenAI uses 6B GPT-3 and DeepMind uses 7B Gopher as the reward model, respectively. Finally, to optimize the pre-trained LM using the signal from the reward model, a specific *RL algorithm* is designed for large-scale model tuning. Specifically, Proximal Policy Optimization (PPO) [128] is a widely used RL algorithm for alignment in existing work [66, 116, 368].

Key Steps for RLHF. Figure 12 illustrates the overall three-step process of RLHF [66] as introduced below.

- *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 LM. These prompts and outputs can be written by human labelers for some specific tasks while ensuring the diversity of tasks. For example, InstructGPT [66] asks human labelers to compose prompts (*e.g.*, “List five ideas for how to regain enthusiasm for my career”) and desired outputs for several generative tasks such as open QA, brainstorming, chatting, and rewriting. Note that the first step is optional in specific settings or scenarios.

- *Reward model training.* The second step is to train the RM using human feedback data. Specifically, we 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. We then invite human labelers to annotate the preference for these pairs. The annotation process can be conducted in multiple forms, and a common approach is to annotate by ranking the generated candidate texts, which can reduce the inconsistency among annotators. Then, the RM is trained to predict the human-preferred output. In InstructGPT, labelers rank model-generated outputs from best to worst, and the RM (*i.e.*, 6B GPT-3) is trained to predict the ranking. Note that, in recent work [369], the annotation of preference on response pairs has been conducted by an AI agent (usually an aligned LLM) instead of humans, which is called “*reinforcement learning from AI feedback (RLAIF)*”. LLMs trained with typical RLHF algorithms tend to generate harmless responses with less helpfulness, which is called *evasion problem* [369]. To guarantee both the harmlessness and helpfulness, RLAIF generates the AI feedback based on pre-set alignment principles in instructions [369, 370], which can also reduce the efforts of human annotation.

- *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 of it is the vocabulary, the state is the currently generated token sequence, and the reward is provided by the RM. To avoid eviating significantly from the initial (before tuning) LM, a penalty term is commonly incorporated into the reward function. For example, InstructGPT optimizes the

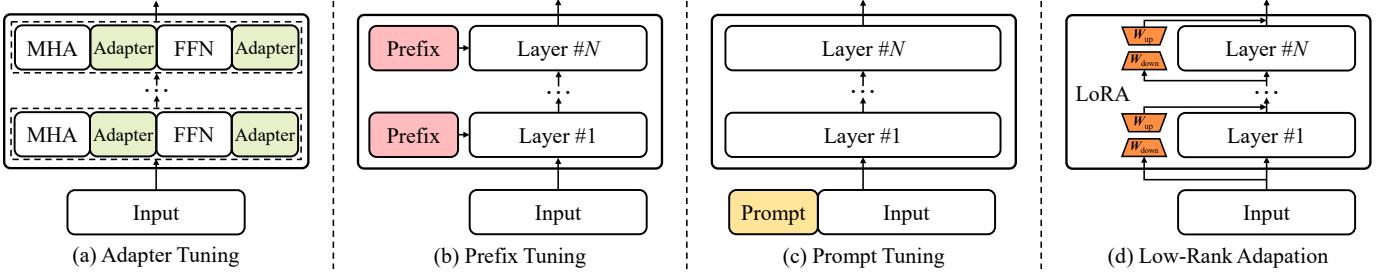


Fig. 13: An illustration of four different parameter-efficient fine-tuning methods. MHA and FFN denote the multi-head attention and feed-forward networks in the Transformer layer, respectively.

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. It is noted that the second and final steps can be iterated in multiple turns for better aligning LLMs. Due to the instability of the RL algorithm, recent work [371] replaces the RL tuning with another supervised fine-tuning by reusing the best ranked samples with higher rewards.

Practical Strategies for RLHF. Although RLHF is promising to effectively improve the alignment of LLMs with humans, it is practically challenging for researchers to successfully implement it. In this part, we focus on discussing several useful strategies and tricks for improving the effectiveness and efficiency of RLHF. Concretely, we focus on the effective training of reward models, efficient and effective RL training, respectively.

- *Effective reward model training.* Despite that InstructGPT used a small reward model (6B GPT model), increasing work [99] has shown it is often more effective to use a large reward model (*e.g.*, equal or greater than the original model size), since large reward models generally perform better in judging the quality of the LLM generated outputs. In LLaMa 2 [99], pretrained chat model checkpoints are used to initialize the reward model, they argue that such an approach can effectively reduce the information mismatch between the model to be aligned and the reward model by sharing the same pre-training knowledge. Whereas, it is common to encounter the overfitting problem when training large-scale reward models. As a simple yet effective solution, existing work [372, 373] has introduced the LM loss on the preferred response of the input prompt from the human-annotated alignment dataset as a regularizer, which alleviates the overfitting of the reward model on the binary classification task. In addition, as there are multiple criteria for alignment (*e.g.*, helpfulness and honesty), it is often difficult to train a single reward model that can satisfy all the alignment criteria. Therefore, it is useful to train multiple reward models that focus on different alignment criteria [99], and compute the final reward based on the produced ones from them via special combination strategies (*e.g.*, mean pooling and weighted sum). Such a way enables more flexible rules or standards on multiple criteria, *e.g.*, relaxing the requirement on helpfulness while posing more strict limits on harmfulness.

- *Effective RL training.* As the RL training process tends to be unstable and hyper-parameter sensitive, it is suggested that the language model should be well supervised fine-

tuned before RL training, so as to reaching a good model capacity. A commonly-used way is to fine-tune the LLM on its best outputs of the prompts (referred to as *rejection sampling* or *best-of-N*) from the alignment dataset until convergence before RL. Given a prompt, the LLM would first produce N outputs via the sampling algorithm, and then the best candidate from the model will be selected by the reward model for learning. After fine-tuning the LLM on the best samples until convergence, the RL process will be performed to further improve the performance. LLaMA 2 [99] has successively trained five versions of RLHF models, where the LLM has been progressively improved with the improvement of the reward models. In this way, the collected prompts and annotations of human preference data can better reflect the issues of the current model checkpoint, thus making special tuning to address these issues. In addition, LLaMA 2 also adds samples from prior iterations into the subsequent ones, to alleviate the possible capacity regression issue during iterative optimization.

- *Efficient RL training.* As the RL training requires to iterate the inference process of both the LLM and reward models, it would greatly increase the total memory and computation cost, especially for larger reward models and LLMs. As a practical trick, we can deploy the reward model on a separate server, and invoke the corresponding API to work with the LLM on its own server. In addition, as RLHF requires the LLM to generate multiple candidate outputs, instead of calling the sample decoding procedure for multiple times, it is more efficient to utilize the beam search decoding algorithm²⁶. It only needs to perform one-pass decoding for response generation, meanwhile such a strategy can also enhance the diversity of the generated candidate responses.

Process-Supervised RLHF. In existing literature of RLHF [374], the supervision approach for RL training generally takes two major forms, either using outcome-supervision signals or process-supervision signals. The outcome-supervised RLHF employs a quantitative score to assess the quality of the whole text generated by LLMs. In contrast, process-supervised RLHF offers an evaluation of each individual component (*e.g.*, sentence, word, or reasoning step) within the generated content, which leverage fine-grained supervision signals to guide the training, helping

26. https://huggingface.co/docs/transformers/v4.31.0/en/main_classes/text_generation#transformers.GenerationMixin.group_beam_search

LLMs refine the undesired generation contents [374, 375]. In what follows, we discuss two key aspects of process-supervised RLHF.

- *Obtaining Fine-grained Supervision Signals.* Compared with outcome rewards, it is more difficult to obtain fine-grained supervision signals. OpenAI has released a fine-grained annotation dataset named PRM800k [375] consisting of 12K process-annotated mathematical problems (*i.e.*, MATH dataset [376]) and 75K solutions generated by LLMs of these problems, where each reasoning step of mathematical problems is labeled as *positive*, *negative* or *neutral* in PRM800k. Considering the cost and efficiency of the human annotation process, several methods aim to automatically annotate the correctness of intermediate reasoning steps, *e.g.*, using powerful LLMs to directly replace human annotators [377] or Monte Carlo tree search [378]. After obtaining fine-grained supervision signals, existing work typically leverages them to train process-supervised reward models (PRM) [375, 379], which can produce step-level rewards (*e.g.*, sentence based or token based rewards) during the RLHF procedure. Furthermore, rather than leveraging the discriminative model to produce the rewards, RLMEC [380] utilizes a generative reward model trained on rewriting tasks with the minimum editing constraint, to provide token-level rewards. In addition, for the downstream tasks where fine-grained supervision signals are difficult to collect, outcome-supervision signals can also be utilized to perform process-supervised RLHF [381].

- *Utilizing the PRMs.* To effectively leverage process-supervision signals from PRMs, existing work mainly utilizes these fine-grained signals to evaluate individual parts within the LLM responses and then guides LLMs to adjust their generation behaviors to maximize the received reward of the response. Concretely, expert iteration [382, 383], an effective RL algorithm, has been utilized to improve the base policy via learning from expert policy [374]. Typically, expert iteration contains two main stages: policy improvement and distillation [374]. In the policy improvement stage, expert policy processes the systematic search procedure to produce the samples under the guidance of PRMs. Subsequently, during the distillation stage, the samples generated by expert policy in the first stage are utilized to improve the base policy through supervised fine-tuning. In addition to expert iteration, PRMs can also be utilized to re-rank the candidates of the final answers generated by LLMs [375] or to select better intermediate reasoning steps during step by step reasoning [379, 384].

5.2.4 Alignment without RLHF

Although RLHF has achieved great success in aligning the behaviors of LLMs with human values and preferences, it also suffers from notable limitations. First, RLHF needs to train multiple LMs including the model being aligned, the reward model, and the reference model at the same time, which is tedious in algorithmic procedure and memory-consuming in practice. Besides, the commonly-used PPO algorithm in RLHF is rather complex and often sensitive to hyper-parameters. As an alternative, increasing studies explore to directly optimize LLMs to adhere to human preferences, using supervised fine-tuning without reinforcement learning [338].

Overview. The basic idea of non-RL alignment approaches is to directly fine-tune LLMs with *supervised learning* on high-quality *alignment dataset*. It basically assumes that response feedback or golden rules to avert unsafe behaviors have been injected or included in the specially curated alignment dataset, so that LLMs can directly learn aligned behaviors from these demonstration data via suitable fine-tuning strategies. Thus, to implement this approach, two key issues are the construction of alignment dataset and the design of fine-tuning loss. For the first issue, the alignment dataset can be automatically constructed by an aligned LLMs according to human-written safety principles [336] or refining existing examples using edits operations [385]. In addition, we can also reuse existing reward models to select high-rated responses from existing human feedback data [371]. For the second issue, non-RL alignment approaches mainly fine-tune LLMs in a supervised learning way (the same as the original instruction tuning loss) on a high-quality alignment dataset, meanwhile auxiliary learning objectives can be used to enhance the alignment performance, *e.g.*, ranking responses or contrasting instruction-response pairs.

Alignment Data Collection. The construction of alignment data is important to effectively align the behaviors of LLMs with human preferences. To collect high-quality alignment data, some work tries to reuse existing reward models to select high-rated responses, and others explore to leverage powerful LLMs (*e.g.*, ChatGPT) or build a simulated environment to generate synthetic alignment examples. Next, we will discuss these three lines of research.

- *Reward model based approaches.* The reward model in RLHF has been trained to measure the alignment degree on the responses of LLMs. It is straightforward to leverage existing reward models to select high-quality responses as alignment data for subsequent fine-tuning. Based on this idea, RAFT [371] adopts reward models trained on human preference data to rank the responses of LLMs and collect those with higher rewards for supervised fine-tuning. In addition, the reward model can be also used to score model responses and assign them to different quality groups. Quark [386] sorts the responses of LLMs into different quantiles based on the reward scores. Each quantile is attached with a special reward token to represent the reward level of the quantile. Conditioned on the highest-reward tokens, LLMs are subsequently prompted to generate high-quality responses. Given an initial answer and the corresponding human feedback, ILF [387] first adopts LLMs to generate refined answers, then utilizes the reward model to select the answer that best matches the feedback for further training. As valuable resources for aligning LLMs, several reward models have been released, including DeBERTa-base/large/xxlarge from OpenAssistant²⁷, Moss-7B from Fudan²⁸, and Flan-T5-xl from Stanford²⁹.

- *LLM based generative approaches.* Reward models help to select aligned data from model responses. However, training reward models itself necessitates substantial high-quality human-labeled data, which is typically expensive and in short supply. In addition, although existing reward

27. <https://huggingface.co/OpenAssistant>

28. <https://github.com/OpenLMLab/MOSS-RLHF>

29. <https://huggingface.co/stanfordnlp/SteamSHP-flan-t5-xl>