

2 Preliminary

2.1 Fundamentals of LLMs

LLMs have become the foundation of modern natural language processing. Their success largely follows a simple but powerful paradigm: first learn general language capabilities from large-scale text corpora, and then adapt these capabilities to specific tasks through prompting or supervised fine-tuning. At inference time, an LLM generates an output by predicting tokens sequentially according to the given input and previously generated tokens. This paradigm has substantially changed how NLP systems are developed, replacing many task-specific models with a unified foundation model that can support a wide range of language understanding and generation tasks.

Understanding these basic mechanisms is important before introducing RL for LLMs. Many RL concepts can be naturally connected to the standard generation process of an LLM. The input and previously generated tokens define the current context, the next-token distribution determines the model’s possible actions, and the complete output forms a sequence of decisions that can be evaluated by a reward signal. In this subsection, we briefly review the fundamentals of LLMs and introduce the key concepts and notations used throughout this paper. We focus only on the concepts necessary for understanding the subsequent discussion of RL, rather than providing a comprehensive review of LLM techniques and their recent developments. Interested readers are referred to [Xiao & Zhu \(2025\)](#) for a more systematic introduction to LLMs.

In this paper, we mainly focus on generative LLMs based on decoder-only Transformers ([Vaswani et al., 2017](#)). Let $\mathcal{V}$ denote the vocabulary of tokens. A token is the basic unit processed by an LLM. It can be a word, a subword, a punctuation mark, or another text fragment produced by the tokenizer. Before a raw text is input into an LLM, it is first converted into a sequence of tokens from $\mathcal{V}$. Therefore, throughout this paper, all inputs and outputs of an LLM are represented as token sequences. Given a token sequence $\mathbf{z} = z_1 \dots z_N$, a language model parameterized by θ estimates the probability of the sequence by factorizing it from left to right:

$$\Pr_{\theta}(\mathbf{z}) = \prod_{i=1}^N \Pr_{\theta}(z_i | \mathbf{z}_{<i}) \quad (1)$$

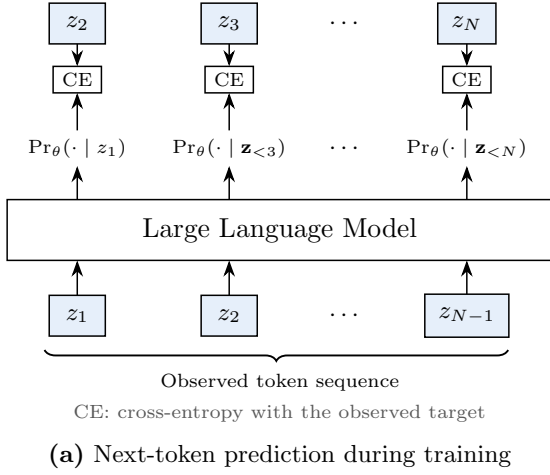
$$\log \Pr_{\theta}(\mathbf{z}) = \sum_{i=1}^N \log \Pr_{\theta}(z_i | \mathbf{z}_{<i}) \quad (2)$$

where $\mathbf{z}_{<i} = z_1 \dots z_{i-1}$ denotes the tokens before z_i . A decoder-only Transformer implements this conditional distribution with causal self-attention, so that each position can only depend on previous tokens. The model then produces a probability distribution over $\mathcal{V}$, denoted by $\Pr_{\theta}(\cdot | \mathbf{z}_{<i})$. This next-token prediction view is the central interface through which LLMs are trained, prompted, fine-tuned, and later optimized by RL. [Figure 1](#) illustrates this process together with prompt-based autoregressive generation. During training, the predicted distributions are matched to the observed target tokens. During inference, each selected token is appended to the context and used to condition the next prediction.

2.1.1 Pre-training

Pre-training is the first stage of building most LLMs. The goal is to train the model on large-scale unlabeled text so that it can acquire general knowledge about languages and the world. In generative LLMs, this is usually achieved through self-supervised next-token prediction: the model observes the preceding tokens and learns to predict the next token. Although no human-annotated labels are required, each token in the text can naturally serve as a supervision signal for predicting itself from its left context.

Minimizing the next-token prediction loss:
 $\mathcal{L}_{\text{LM}} = -\sum_{i=1}^N \log \Pr_{\theta}(z_i | \mathbf{z}_{<i})$



Generating successive tokens by an inference algorithm (e.g., sampling): $y_t \sim \Pr_{\theta}(\cdot | \mathbf{x}, \mathbf{y}_{<t})$

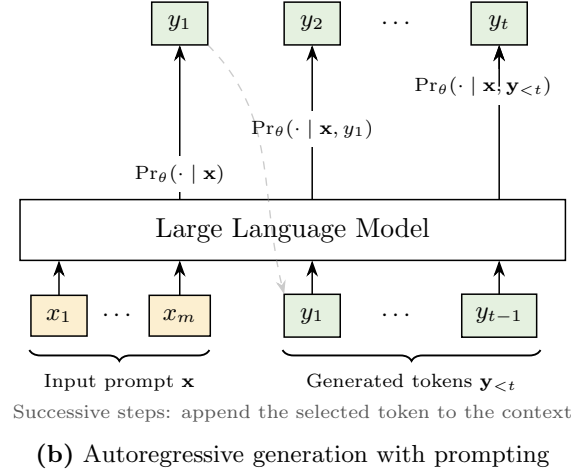


Figure 1: Language model training and autoregressive generation. (a) In next-token prediction, an LLM predicts a distribution over the next token from the preceding context; the predicted distribution is compared with the observed target token using cross-entropy. (b) In autoregressive generation, an LLM selects the next token from the distribution conditioned on the prompt and previously generated tokens. The selected token is appended to the context and then conditions the next prediction step.

Let $\mathcal{S}_{\text{pre}}$ denote the pre-training corpus, where each sample is a token sequence $\mathbf{z} = z_1 \dots z_N$. The pre-training objective is to maximize the log-likelihood of all tokens in the corpus:

$$\hat{\theta} = \arg \max_{\theta} \sum_{\mathbf{z} \in \mathcal{S}_{\text{pre}}} \sum_{i=1}^N \log \Pr_{\theta}(z_i | \mathbf{z}_{<i}) \quad (3)$$

where $\hat{\theta}$ denotes the optimized pre-trained parameters. Equivalently, this objective can be viewed as minimizing the cross-entropy loss between the observed next token and the model-predicted distribution.

After pre-training, the LLM learns to model natural language and acquires rich linguistic and factual knowledge from large-scale text. Given a preceding context, it can predict plausible continuations and generate fluent text. For example, given the prefix “The capital of France is”, a pre-trained LLM may assign a high probability to the next token “Paris”. Similarly, given a longer context such as “To solve this equation, we first move all terms to one side”, the model can continue the text with a linguistically and semantically plausible sequence. These examples reflect the core capability learned during pre-training: predicting likely continuations from the preceding context.

However, pre-training primarily teaches the model to perform language modeling rather than to explicitly solve user-specified tasks. In particular, the model is not directly trained to interpret instructions or produce outputs in a desired format. As a result, simply providing a task description to a pre-trained LLM does not necessarily lead to the intended behavior. This limitation motivates us to use additional adaptation approaches to better equip pre-trained LLMs for downstream tasks.

2.1.2 Prompting

Prompting is a simple and lightweight way to adapt an LLM to different tasks without updating its parameters. A *prompt* is the input text used to guide the model toward a desired task or output. It may

contain an instruction, user-provided content, output requirements, or demonstrations. For example, if we want an LLM to act as a homework assistant and answer a student’s question, we can provide the following prompt:

You are a homework assistant helping students improve their problem-solving skills.
Give me three tips to improve my accuracy in solving math problems.
Please make the response clear, practical, and easy to follow.

In this example, the prompt specifies both the task and the desired response style. The LLM then generates the answer by continuing the input sequence. Prompts can also be constructed from templates. A *prompt template* contains placeholders that can be replaced with task-specific information. For example, we can use the following template for a homework assistant:

You are a homework assistant helping students improve their problem-solving skills.
Give me three tips to improve my accuracy in solving `{*subject*}` problems.
Please make the response clear, practical, and easy to follow.

If we set `{*subject*}` to “math”, the template becomes the prompt used above. In this way, users can construct different prompts by changing the placeholder while keeping the main task description unchanged.

Another important concept related to prompting is in-context learning. When prompting an LLM, we can add demonstrations to the context and let the model infer the desired input-output pattern from these examples. For instance, we can show the model how to answer similar student questions before asking it to respond to a new one:

Input: Give me two tips to improve my English writing.
Output: 1. Read well-written essays regularly.
2. Revise your sentences to make them clear.
Input: Give me three tips to improve my accuracy in solving math problems.
Output: _____

When a single demonstration is provided, this setting is commonly called *one-shot* prompting. When several demonstrations are included, it is referred to as *few-shot* prompting. In contrast, prompting without any demonstration is usually called *zero-shot* prompting.

In-context learning can also be combined with *chain-of-thought* (CoT) prompting (Wei et al., 2022b). The basic idea is to encourage the model to generate intermediate reasoning steps before producing the final answer. This can be achieved either by explicitly asking the model to reason step by step or by providing demonstrations that contain both reasoning traces and answers. For example:

Input: A student solves 8 math problems and gets 6 correct. What is the accuracy? Please reason step by step.

Output: The student gets 6 out of 8 problems correct. The accuracy is therefore $6/8 = 0.75$, or 75%.

Input: A student solves 20 math problems and gets 17 correct. What is the accuracy? Please reason step by step.

Output: _____

Prompting adapts an LLM by changing the context used for generation. Therefore, prompt quality can strongly affect model performance. Even for the same task, we can write the prompt in many different ways. For example, “Give me two tips to improve my English writing.” can also be written as “What are two simple ways to improve my English writing skills?”. The two prompts express nearly the same intent, but they may lead to noticeably different outputs. In practice, we often refine prompts through repeated trial and error for a given LLM. More advanced methods automate this process and search for better prompts using techniques such as RL (Deng et al., 2022) or evolutionary algorithms (Guo et al., 2024).

2.1.3 Supervised Fine-Tuning

Although pre-trained LLMs possess a vast amount of general knowledge, they are typically limited to tasks related to language modeling. Our ultimate goal is to enable them to perform various specific tasks, such as summarization, question answering, and machine translation. One straightforward method to achieve this goal is supervised fine-tuning (SFT), in which the pre-trained LLM is further trained on a dataset comprising task-specific inputs (i.e., instruction + user input)¹ paired with their expected outputs (Ouyang et al., 2022; Wei et al., 2022a). Table 1 gives several examples of SFT data for different tasks.

Specifically, let $\mathbf{x} = x_1 \dots x_m$ be an input and $\mathbf{y} = y_1 \dots y_T$ be the corresponding output. In SFT, we aim to maximize the probability of the output $\mathbf{y}$ given the input $\mathbf{x}$. Consider an LLM with pre-trained parameters $\hat{\theta}$. The fine-tuning objective can then be formulated as:

$$\tilde{\theta} = \arg \max_{\tilde{\theta}^+} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{S}} \log \Pr_{\tilde{\theta}^+}(\mathbf{y}|\mathbf{x}) \quad (4)$$

where $\Pr(\cdot)$ denotes the probability distribution, $\mathcal{S}$ denotes the labeled data, $\tilde{\theta}$ denotes the parameters optimized via SFT, and $\tilde{\theta}^+$ represents an adjustment to $\hat{\theta}$. Here, we will omit the superscript + and use θ to represent $\tilde{\theta}^+$ to keep the notation uncluttered. However, the reader should remember that the fine-tuning starts from the pre-trained parameters rather than randomly initialized ones.

The objective function $\log \Pr_{\theta}(\mathbf{y}|\mathbf{x})$ is computed by summing the log-probabilities of the tokens in $\mathbf{y}$, conditional on the input $\mathbf{x}$ and all the previous output tokens $\mathbf{y}_{<t}$:

$$\log \Pr_{\theta}(\mathbf{y}|\mathbf{x}) = \sum_{t=1}^T \log \Pr_{\theta}(y_t|\mathbf{x}, \mathbf{y}_{<t}) \quad (5)$$

This formulation is equivalent to minimizing the cross-entropy loss over the output segment. In practice, the input tokens in $\mathbf{x}$ are used as the context, while the loss is usually computed only on the output tokens

¹In this paper, the *instruction* represents the description of a specific task, e.g., “Summarize the following article”; the *user input* represents the extra content added to the instruction, e.g., “Article: In recent years, solar energy has seen a significant increase in the amount of energy available to the public. Solar energy has seen unprecedented growth, becoming the fastest-growing ...”

x (Instruction + User Input)	y (Output)
Summarization: Summarize the following article. Article: Solar energy has experienced rapid growth in recent years and has become one of the fastest-growing sources of renewable energy.	Solar energy has grown rapidly in recent years and has become a major renewable energy source.
Question Answering: Answer the following question. Question: What is the capital of France?	Paris.
Classification: Classify the following message as spam or not spam. Message: Congratulations! You have won a \$500 gift card. Click here to claim it.	Spam.
Machine Translation: Translate the following sentence from English to Chinese. Sentence: RL is widely used to train large language models.	强化学习被广泛用于训练大语言模型。

Table 1: Examples of SFT data for different downstream tasks.

in $\mathbf{y}$. In this way, SFT teaches the model not only what knowledge to express, but also how to respond in a desired instruction-following format. In the following section, we refer to the LLM after SFT as the “SFT LLM” for short.

2.1.4 Inference

Inference is the process of applying a trained LLM to generate outputs for new inputs. Given an input $\mathbf{x}$, the model first predicts a next-token distribution $\Pr_\theta(\cdot|\mathbf{x})$ and selects a token y_1 according to a decoding strategy. The selected token is then appended to the context, after which the model predicts the next token according to $\Pr_\theta(\cdot|\mathbf{x}, y_1)$. This process is repeated until a stopping criterion is reached, such as generating an end-of-sequence token or reaching the maximum generation length. We refer to this sequential generation process as *autoregressive generation*.

Different decoding strategies determine how the next token is selected from the model-predicted distribution. Commonly used strategies include:

- **Greedy Decoding.** Greedy decoding is one of the simplest decoding strategies. At each generation step, it selects the token with the highest probability under the model. Consider the probability of the generated sequence up to timestep t :

$$\log \Pr_\theta(\mathbf{y}_{\leq t}|\mathbf{x}) = \log \Pr_\theta(\mathbf{y}_{< t}|\mathbf{x}) + \log \Pr_\theta(y_t|\mathbf{x}, \mathbf{y}_{< t}) \quad (6)$$

where $\mathbf{y}_{< t} = y_1 \dots y_{t-1}$ denotes the previously generated tokens. Since the first term is fixed once $\mathbf{y}_{< t}$ has been generated, greedy decoding selects

$$y_t^{\text{greedy}} = \arg \max_{w \in \mathcal{V}} \Pr_\theta(w|\mathbf{x}, \mathbf{y}_{< t}) \quad (7)$$

The selected token is appended to the current prefix, and the same procedure is repeated at the next timestep. Greedy decoding is deterministic and computationally efficient, but locally selecting the most probable token does not necessarily produce the most probable complete sequence.

- **Beam Search.** Beam search extends greedy decoding by maintaining multiple candidate sequences during generation. Instead of keeping only the single best prefix at each timestep, it retains the top B candidates, where B is referred to as the *beam size*. Let $\mathcal{B}_{t-1}$ denote the set of candidate

prefixes retained at timestep $t - 1$. Each prefix $\mathbf{y}_{<t}$ is extended with possible next tokens, and the resulting candidates are ranked according to their accumulated sequence scores:

$$\begin{aligned} s(\mathbf{y}_{\leq t}) &= \log \Pr_{\theta}(\mathbf{y}_{\leq t} | \mathbf{x}) \\ &= s(\mathbf{y}_{<t}) + \log \Pr_{\theta}(y_t | \mathbf{x}, \mathbf{y}_{<t}) \end{aligned} \quad (8)$$

The beam for the next timestep is then constructed by retaining the B highest-scoring candidates:

$$\mathcal{B}_t = \text{TopB} \{(\mathbf{y}_{<t}, w) : \mathbf{y}_{<t} \in \mathcal{B}_{t-1}, w \in \mathcal{V}\} \quad (9)$$

By exploring multiple prefixes simultaneously, beam search can avoid some locally optimal decisions made by greedy decoding and often produces higher-probability sequences. However, it requires more computation and memory as the beam size increases.

- **Sampling.** Sampling is widely used in modern LLM inference, particularly when diverse outputs are desired. Instead of always selecting the most probable token or maintaining a fixed set of candidate sequences, the next token is sampled from the model-predicted distribution:

$$y_t \sim \Pr_{\theta}(\cdot | \mathbf{x}, \mathbf{y}_{<t}) \quad (10)$$

This makes generation stochastic, allowing the same input to produce different outputs. In practice, the sampling distribution is often adjusted using a temperature parameter T . Given the model logit l_w for token w , the temperature-adjusted probability is

$$\Pr_{\theta}^{(T)}(w | \mathbf{x}, \mathbf{y}_{<t}) = \frac{\exp(l_w/T)}{\sum_{v \in \mathcal{V}} \exp(l_v/T)} \quad (11)$$

A smaller temperature sharpens the distribution and favors high-probability tokens, while a larger temperature produces a flatter distribution and increases diversity. Sampling is also commonly combined with *top-k* or *top-p* filtering. *Top-k* sampling restricts the candidate set to the k tokens with the highest probabilities (Fan et al., 2018), whereas *top-p* sampling retains the smallest set of tokens whose cumulative probability reaches a threshold p (Holtzman et al., 2019). The next token is then sampled from the renormalized distribution over the retained candidates. These techniques help reduce the probability of selecting unlikely tokens while preserving diversity.

Regardless of the decoding strategy, LLM generation can be viewed as a sequence of token-level decisions. This view naturally connects LLMs with RL. At timestep t , the current context $(\mathbf{x}, \mathbf{y}_{<t})$ can be treated as the state, the next token y_t as the action, and the next-token distribution $\Pr_{\theta}(\cdot | \mathbf{x}, \mathbf{y}_{<t})$ as the policy. The complete output $\mathbf{y}$ then corresponds to a trajectory whose quality can be evaluated by a reward signal. In particular, sampling closely resembles the stochastic policy sampling process used in RL. Based on this connection, the following subsection reviews the fundamentals of RL, and Section ?? further shows how RL can be used to optimize an SFT LLM.

2.2 Fundamentals of RL

RL, supported by a well-established theoretical framework, has been widely applied across a broad range of domains. In particular, with the rapid advancement of LLMs, RL has become a standard approach in the post-training stage. Before using LLM training examples to explain RL algorithms, we first provide a brief overview of deep reinforcement learning in this subsection. We then introduce the general formulation of RL, along with key terminologies and notations that are essential for understanding RL.

2.2.1 Markov Decision Process

The RL formulation for LLMs is illustrated in Figure 2. An RL system primarily consists of two components: an agent and an environment. At each timestep t , the agent observes a state s_t from the environment and selects an action a_t according to a policy π , which is typically parameterized by a neural network. After

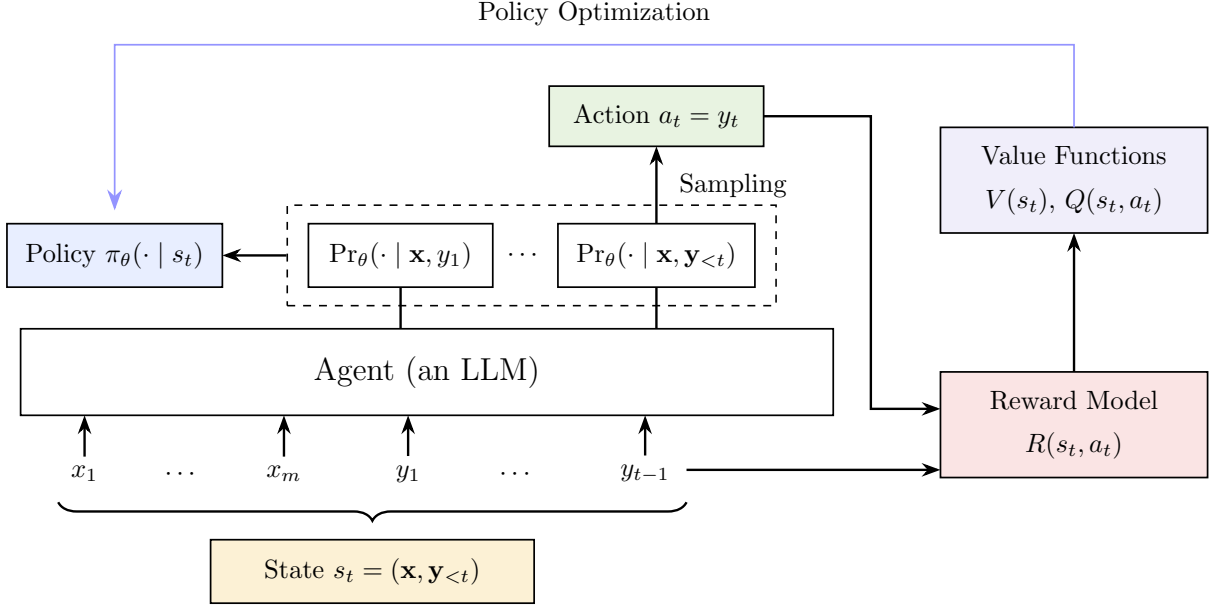


Figure 2: A schematic illustration of RL for LLMs.

executing the action, the environment transitions to a new state s_{t+1} and returns a reward r_t corresponding to the taken action.

This interaction process is commonly modeled as a Markov Decision Process (MDP), where the agent interacts with the environment over discrete timesteps (Sutton & Barto, 2018). A sequence of states and actions forms a trajectory, denoted as $\tau = (s_0, a_0, s_1, a_1, \dots, s_{H-1}, a_{H-1})$, where H represents the trajectory length. Each trajectory accumulates rewards from the environment.

The objective of RL is to learn an optimal policy π^* that maximizes the expected cumulative reward over trajectories, which can be formulated as:

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t=0}^{H-1} r_t \mid \pi \right] \quad (12)$$

In practice, the agent improves its policy through repeated interactions with the environment by observing states and receiving feedback in the form of rewards. Each such interaction sequence constitutes a trajectory τ . The process of generating one or more trajectories under a given policy is commonly referred to as *sampling* in the RL literature.

2.2.2 Key Elements

The MDP formulation provides a compact way to describe RL, but its terminology can feel distant from the language modeling setting at first glance. To make the connection more explicit, Table 2 summarizes the core elements of RL and describes how each one can be interpreted in LLM training. This tabular view is intended to serve as a bridge between the standard RL notation introduced above and the token-level generation process discussed earlier.

With these correspondences in place, we can describe RL algorithms using familiar language-modeling objects such as contexts, tokens, policies, and reward signals. In contrast to conventional RL literature, which often presents algorithms through robotics or control examples, *we adopt an NLP perspective to introduce RL in the following section.*

Element	Interpretation
Agent	The learner or decision-maker in RL. In the context of LLMs, the agent corresponds to the language model itself, which generates tokens sequentially and updates its behavior based on feedback signals.
Environment	Everything external to the agent with which it interacts. Unlike traditional RL settings that involve physical or simulated environments, the environment in LLM-based RL is typically abstract, consisting of the training framework that provides feedback (e.g., reward models, human annotations, or evaluation metrics) for generated outputs.
State (s)	A state represents the current situation of the environment. For language modeling, the state at timestep t can be defined as the sequence of observed tokens up to that point, i.e., the context used to predict the next token. Formally, the state can be represented as $s_t = (\mathbf{x}, \mathbf{y}_{<t})$, where $\mathbf{x}$ denotes the input prompt and $\mathbf{y}_{<t}$ denotes the previously generated tokens.
Action (a)	An action corresponds to a decision made by the agent. In LLMs, actions are naturally defined as selecting the next token from the vocabulary, i.e., $a = y_t$.
Reward (r)	The reward provides feedback from the environment to evaluate the quality of an action. In general, the reward function can be defined as $r(s, a, s')$, representing the feedback received when the agent transitions from state s to s' by taking action a . At timestep t , this can be written as $r_t = r(s_t, a_t, s_{t+1})$. In deterministic settings, where the next state is uniquely determined by (s_t, a_t) , the reward can be simplified as $r(s_t, a_t)$.
Policy (π)	The policy defines the agent’s behavior, i.e., the probability of taking an action given a state. For LLMs, the policy corresponds to the conditional probability distribution over the next token given the context: $\pi_{\theta}(a_t \mid s_t) = \Pr_{\theta}(y_t \mid \mathbf{x}, \mathbf{y}_{<t})$ where $a_t = y_t$ and $s_t = (\mathbf{x}, \mathbf{y}_{<t})$. Under this formulation, an LLM can be naturally interpreted as a parameterized policy.
Value Function (V and Q)	The value function estimates the expected cumulative reward when following a policy. The state-value function $V(s)$ measures the expected discounted return starting from state s: $V(s) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, \pi \right]$ where $\gamma \in [0, 1]$ is the discount factor. The action-value function $Q(s, a)$ further conditions on the initial action: $Q(s, a) = \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right]$

Table 2: Key elements of RL and their interpretations

References

- Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 conference on empirical methods in natural language processing*, pp. 3369–3391, 2022.
- Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 889–898, 2018.
- Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024.
- Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html.
- Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (2nd ed.)*. The MIT Press, 2018.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>.
- Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022a. URL <https://openreview.net/forum?id=gEZrGCozdQR>.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022b.
- Tong Xiao and Jingbo Zhu. Foundations of large language models, 2025. URL <https://arxiv.org/abs/2501.09223>.