1 Post-training techniques

About this chapter

In Volume 1 we focused on models trained from scratch or via fine-tuning. In this chapter we consider instead techniques for **post-training** large language models (LLMs), or more in general large multimodal models (LMMs), making them useful for a large variety of tasks with minimal (or no) fine-tuning. We cover a broad selection of material including preference alignment, instruction tuning, and tool usage, with a specific focus on reinforcement learning.

While differentiable models trained via gradient descent exist since many decades [LBBH98], in the last 15 years we saw a significant shift from training *specialized* neural networks on *individual* tasks, to having extremely large *generalist* models (e.g., GPT-5) able to perform well on a variety of tasks with minimal human supervision and with no (or very little) fine-tuning.

As we discussed in Volume 1, the first component of these models is always a **pre-training** phase, where the model is trained with some unsupervised loss (e.g., next-token prediction) on huge unlabeled datasets. Once these models reach a given scale, they already show the ability to solve tasks via few-shot prompting [BMR+20], although they are sub-optimal at following instructions or performing well when no examples are given.

To this end, researchers and companies have developed a variety of **post-training** techniques to turn basic pre-trained models into the generalist models we use daily: aligned with human preferences, able to reason for longer periods on a given problem, and capable of using external tools or follow complex structured prompts. The techniques and the history behind some of these algorithms are the main focus of this chapter.

The chapter is composed of two broad parts. Since these techniques evolved very quickly over a span of just a few years, in the first part we provide a short historical overview of some important breakthroughs for contextualization. We focus the overview on papers that have popularized specific techniques (which generally coincide with the first instance where the idea has been shown to scale to large models and datasets), omitting many historical details for the sake of brevity and leaving them to more specialized material or surveys [Lam25, LLG+25].

In the second part we focus on the use of reinforcement learning (RL) for fine-tuning LLM models, which has emerged as one of the most essential post-training strategies. Readers interested only in the technical content can skip directly to the second part.

1.1 From pre- to post-training

Fine-tuning: the "ImageNet moment" (2012)

In Volume 1 we considered a simple scenario, where the parameters of a model are initialized at random and optimized on a labeled dataset. If we assume that the size of a model and the data used to train it are roughly proportional, it follows that this recipe works if either (a) the model is kept small, or (b) we have a very large labeled dataset.

Alternatively, we can take another trained model as initialization, and **fine-tune** its parameters on the new task (where, in general, we need to re-initialize at least the last layer, called the **head** of the model, while the pre-trained part is referred to as the **backbone**). Informally, while we still need to run gradient descent on the model, if the pre-training and fine-tuning datasets are similar, the model will be initialized closer to a good solution and will require less data to achieve good performance.

In computer vision, this strategy was popularized in 2012 with the AlexNet [KSH12] model pre-trained on the ImageNet dataset. As we saw in Volume 1, Chapter 9, scaling convolutional models to work better on ImageNet led to several innovations in the design of the models (e.g., residual connections), and to the first **pre-trained** models having hundreds of layers. However, from the point of view of training, there is comparatively little difference (scaling issues aside) between training a model from scratch and fine-tuning it, see Figure F.1.1.[1]

[1] There might be *qualitative* differences, e.g., there is agreement that fine-tuning a large model results in low-rank updates which can

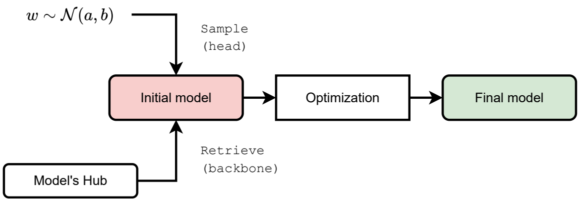

Figure F.1.1: *In fine-tuning, the initial weights of the backbone are initialized from an external source instead of being sampled randomly.*

Self-supervised learning (2014)

While a million images for training seems a lot, accuracy on the ImageNet dataset in the period 2012-2016 eventually saturated with scale (i.e., number of layers) to human-level capabilities. Thus, larger datasets were needed. Since performance correlates in a power-law (in the *order of magnitude*) with the size of the dataset [SSSG17], significant improvements required datasets in the order of 10^7 images [RBBNZM21], then 10^8 [SSSG17], and so on. As a result, labeling once again became the biggest bottleneck in the process.

An alternative approach stems from the observation that *unlabeled* data (i.e., images) are abundant on the web and can be easily scraped (in the limit of what is legal) in the order of billions. The idea of **self-supervised learning** (SSL) is to pre-train a model on this unlabeled dataset by creating what are known as **pretext** tasks, i.e., auxiliary tasks that can be created automatically from the unlabeled data, and which are used only for the purpose of pre-training the model. Examples in the computer vision field abound, including solving jigsaw puzzles [MM20],

be exploited for more efficient training [HSW+22].

reconstructing patches [HCX+22], or aligning video frames with respect to time.

SSL was especially impactful for natural language processing (NLP), given the size and diversity of text available online (e.g., books, code, chats, ...). The two most successful approaches emerged shortly after the introduction of transformers, relying on either masked token reconstruction (popularized by the BERT model [DCLT19]) or next-token prediction (i.e., autoregressive generation, popularized by the first GPT model [RNS+18]).

In 2021, a publication by the Stanford Institute for Human-Centered Artificial Intelligence proposed the name **foundation models** [B+21] to describe extremely large models that are pre-trained on one or more proxy tasks in order to be fine-tuned on a large number of downstream tasks. This terminology remains popular, although it is sometimes used out of context by referring to generalist models such as GPT-5, which are usable without fine-tuning, as discussed next.

In-context learning (2020)

Fine-tuning a "foundation model" requires hundreds (or thousands) of labeled examples, and these have to be collected for each individual task we need to solve. Before the advent of LLMs, this problem was explored broadly in machine learning under the name **few-shot learning** [WYKN20], which refers to a scenario where only a few (e.g., 1-5) examples are available for a fine-tuning. However, models able to perform few-shot learning had to be optimized with specific algorithms, such as model-agnostic meta-learning [WYKN20]. Surprisingly, in

2020 OpenAI showed that a scaled-up version of GPT, called GPT-3, was able to perform few-shot learning on many NLP tasks with no need for additional fine-tuning, by simply providing the examples in the prompt to the model [BMR+20].

In the extreme case of having no access to labeled examples, some tasks could still be solved by an appropriate task description in the prompt (what is now called **instruction prompting**). Few-shot learning done by prompting was one of the first examples of **in-context learning**, i.e., the ability of a very large model to “learn” without gradient-based fine-tuning (context being a different name for the prompt of the model).[2]

Instruction tuning (2021)

Few-shot learning (whether in-context or not) still requires the availability of a small set of high-quality labeled data, and performance is also dependent on the choice (and possibly the ordering) of these examples in the prompt. In addition, the performance of GPT-3 rapidly deteriorated in the zero-shot scenario [BMR+20]. However, this does not mean necessarily that the model is incapable of solving these tasks: more trivially, it can be that the next-token prediction loss does not immediately endow a model with the ability to follow instructions and solve a large set of tasks without supervision.

In 2021 a paper from Google [WBZ+21] popularized the idea of **instruction tuning**, i.e., fine-tuning a pre-trained

[2] The discovery of in-context learning has prompted (pun intended) a massive literature on exploring how it works [VONR+23] and how it emerges at scale [WWT+23]. We will consider **induction heads** [OEN+22] as a possible mechanism in the chapter on explainability.

```python
tokenizer = AutoTokenizer.from_pretrained("Qwen3-8B")
chat = [
  {"role": "system",
        "content": "You are a travel assistant."},
  {"role": "assistant",
        "content": "Hello, how can I help you?"},
  {"role": "user",
        "content": "What should I visit in Paris?"},
]
tokenizer.apply_chat_template(chat, ...)

# [Out]:
# system: You are a travel assistant.
# assistant: Hello, how should I help you?
# user: What should I visit in Paris?
# assistant:
```

Box C.1.1: *Example of a* ***chat template*** *as implemented in the Hugging Face Transformers library, forcing the user to adhere to the specific chat format used during instruction tuning of the model (we are not showing special tokens in the generated prompt). A final* `assistant` *tag is added at the end to continue generating the appropriate response.*

LLM on a large set of text datasets, where the input contains a description of the task to be solved (the *instructions*). The resulting FLAN ("Finetuned LAnguage Net") was able to efficiently perform zero-shot learning on new tasks, provided these tasks were given with a similar instruction template. Also in this case, the improvement from instruction tuning reduces (or vanishes) when considering smaller models, hinting that scale is necessary for the approach. As a result, from 2022 onward many releases of LLMs included a "base" model (pre-training only) and an "instruct" version (the base model fine-tuned by some form of instruction tuning).

Instruction fine-tuning is a "pure" supervised fine-tuning and does not require any new ideas from the optimization point of view - it is enough to define a proper template for the prompt (e.g., Harmony), structure a dataset in this format, and perform training on it. Most instruction tuning datasets also contain multi-turn examples to enable the model to follow instructions structured over several rounds of responses. See Box C.1.1 for an example.

Preference alignment (2022)

In modern terminology, instruction tuning can be classified as both a **post-training** algorithm (an optimization step that follows the pre-training stage) and as an **alignment** technique. *Aligned* models are models that have been fine-tuned to ensure their outputs are consistent with specific human preferences, such as (in the case of instruction tuning) "being good at following instructions", without altering their performance or their knowledge. Collecting supervised data for alignment, however, is not always easy as in instruction tuning: consider, for example, the tasks of making models that are safer, more creative, or "gentler" towards the users.

In 2022 OpenAI released InstructGPT [OWJ+22], which popularized a different type of alignment technique known as **reinforcement learning from human feedback** (RLHF). In this case, after an initial instruction tuning phase (rebranded as **supervised fine-tuning**, SFT, to distinguish it from the main RLHF phase), the model is asked to generate multiple responses (**completions**) for a set of prespecified prompts. These responses are then collected and ranked by human labelers according to a set of guidelines describing the desired preferences (e.g., "prefer more concise responses").

A reward model is trained on these rankings (starting from a small pre-trained model), and the reward model is then used to optimize the fine-tuned model via reinforcement learning (specifically, **proximal policy optimization**, PPO [SWD+17]). This process avoids the need to collect explicit human-generated completions for the prompts, generalizing directly from the rankings of the labelers. Hence, this is also known as learning from *implicit* intentions.

Reinforcement learning in this phase is needed since the reward is computed only at the end of the generation and (more importantly) we do not have access to a fully-supervised completion, only to the preference data. This three-step process (instruction tuning, reward collection, reward optimization) was extremely successful, and it was one of the main drivers behind the first public release of ChatGPT.[3] It also asserted itself as the leading paradigm for several generations of open and closed LLMs, with multiple variations proposed in the literature (e.g., **direct policy optimization**, DPO [RSM+23], Constitutional AI [BKK+22], ...).

Tool use and function calling (2023)

One limitation of LLMs (irrespective of whether they are aligned or not) is their inability to *act* on the outside world. For example, even a simple query such as "*Who is the current president of France?*" can fail unless the model is given access to an external repository of data. One possibility to this end is enhancing the LLMs with access to **tools** [SDYD+23], i.e., allowing the LLM to execute calls towards external APIs (e.g., Wikipedia, calculators, weather systems) during the generation. For example, the following completion can represent a call to a clock

[3] https://openai.com/index/chatgpt/

function:

```
"The current time in Rome is <API>CLOCK(Rome)</API>"
```

where `<API>` and `</API>` are special tokens delimiting the function call. At this point, an external context manager executes the relevant call `CLOCK(Rome)`, takes the result, and substitutes it into the generation before proceeding:

```
"The current time in Rome is 3 PM"
```

LLMs must be trained to use these tools in a proper way. ToolFormer [SDYD+23] was one of the first papers to show that tool usage could be scaled in a self-supervised way with only a limited set of labeled examples. Since then, most proprietary and several open-source LLMs have been augmented with function calling capabilities, including the possibility for the user to specify new tools to be added dynamically. Currently, the Model Context Protocol (MCP) is considered the leading standard to provide access to a predefined set of tools in a flexible way.

When using tools, the way to interface with the tools as well as their description must eventually fit into the prompt of the model, together with the original system prompt, detailed instructions for the current task, the previous chat history, additional documents retrieved from memory, and so on, giving rise to extremely sophisticated, model-dependent prompts that must be optimized (see Figure F.1.2), possibly in an automatic way [KSM+24].

[4] https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

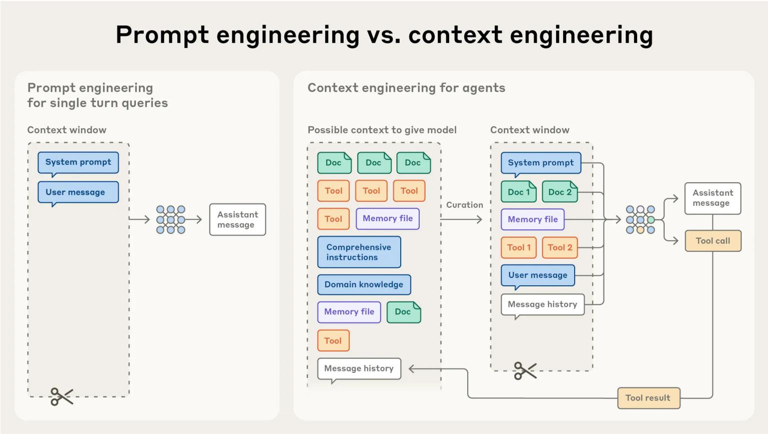

Figure F.1.2: *Anthropic has recently popularized the name **context engineering** to denote practices to optimize these highly structured, information-heavy prompts. Figure reproduced from a blog post.*[4]

Reasoning models (2024)

Another issue of most labeled datasets (including those for instruction tuning) is that they provide the model directly with the solution, which may be suboptimal for domains where the problem can be suitably decomposed into smaller subproblems (e.g., math or coding tasks). Multiple papers consistently showed that letting the model generate these intermediate steps before the answer was beneficial, by either fine-tuning [LYDB17], few-shot examples (also known as **chain-of-thought** [WWS+22]), or even by tuning the prompt to elicit a more step-by-step answer [KGR+22].

In 2024, OpenAI released a preview of its o1 model, one of the first generalist models to exhibit realistic "reasoning traces" for hard problems. Importantly, the material accompanying the release showcased examples of scaling laws for **test-time** compute, i.e., proof that the model performance scaled in a power-law with the compute allocated during the inference phase, due to the possibility of generating longer reasoning traces.

A similar behavior was replicated in 2025 by DeepSeek in their DeepSeek-R1 model [GYZ+25]. The core idea is to consider **verifiable** tasks, i.e., tasks whose solution is difficult, but where evaluating a solution is simple (e.g., in coding, a solution can be evaluated by running it against a known test suite of examples). A pre-trained model can be fine-tuned via RL on a combination of task accuracy (using the verifier as reward) and formatting accuracy, where the model is encouraged to output answers of the form:

```
<THINK>...</THINK><ANSWER>...</ANSWER>
```

This is known as **reinforcement learning from verifiable rewards** (RLVR). Different types of fine-tuning can also consider indirect supervision on the reasoning traces (**process reward model** [LKB+23]) or multiple loops of supervised fine-tuning and RLVR to improve the readability of the answers [GYZ+25]. Importantly, the improvement in accuracy given by the reasoning step can be shown to be beneficial also outside the verifiable tasks, and smaller models distilled from the larger models are superior to their equivalent “instructed” variants.

Agents (2025)

All the techniques described up to now can provide an LLM with a huge set of capabilities: reasoning over extremely long timeframes, acting on the external world via tools, (partially) planning and decomposing difficult problems, and improving their performance over tasks that are only partially specified through a reward function or via human preferences. As a result, the term **agent** has become popular to describe models (or collections of interacting models) provided with a task and left to act autonomously to solve it. Since this is an extremely new "revamping", the definition is still broad, going simply from LLMs augmented with function calling to models that act in a loop for long periods of time (e.g., hours). Due to this, agent-specific ideas are not covered in the rest of the chapter and are left for the future.

> **Scaling up reasoning datasets**
>
> Both preference alignment and verifiable datasets require the presence of a (human-labeled) gold truth, which limits scaling as this data is slowly being exhausted. Recent work (that we are not able to cover here) explores the use of *unsupervised* rewards (e.g., internal coherence maximization [WAS+25]) or **reinforcement learning from AI feedback** [LPM+23] to label datasets automatically from pre-trained models.

1.2 Reward modeling

Do not miss

The rest of the chapter is devoted to an overview of two specific classes of post-training algorithms, preference alignment and reasoning, with a focus on reward modeling and RL algorithms to optimize rewards.

1.2.1 What is a reward?

Rewards are the main difference between classical SFT, which requires fully labeled completions to work, and RL techniques, that can work on more indirect signals such as binary 0/1 verifiers. To understand the distinction, consider a generic prompt x, which is sent to an LLM model π that outputs a distribution $\pi(y|x)$ over the answers.[5] In particular, we focus on the case where the LLM is a standard autoregressive model, in which case the distribution factorizes as (see Volume 1, Chapter 8.4):

$$\pi(y|x) = \prod_t \overbrace{\pi(y_t|x, y_{<t})}^{\text{Distribution for token } t} \tag{E.1.1}$$

Each output $\pi(y|x, y_{<t})$ is a distribution over possible values of the t-th token (whose size depends on the specific tokenization process being used). In order to obtain an actual output, the distribution (E.1.1) is *sampled* to generate a specific token sequence:

$$y \sim \pi(y|x) \tag{E.1.2}$$

This is also known as **decoding** the output. There are many existing decoding strategies, such as greedy decoding, beam search, or top-p sampling [HBD+19]. Importantly, the sampling process in (E.1.2) is in general non-differentiable, meaning that we cannot trivially compute $\frac{\partial y}{\partial x}$ to perform gradient descent.[6]

To overcome this issue, in SFT we have access to a *complete* reference output y^{ref} (the indices corresponding to the correct tokens), and we can compute a token-level loss such as the cross-entropy loss (see Volume 1, Chapters 3 and 4):

$$l(y, y^{\text{ref}}) = \sum_t \log \pi(y_{y_t^{\text{ref}}}|x, y_{<t})$$

[5]We use π to be consistent with RL terminology, where the LLM will define a **policy** function.

[6]Differentiable decoding strategies exist [CHS19], but they are harder to implement and generally uncommon in practice.

Since we are optimizing the model's probabilities, everything is differentiable and we can proceed to gradient descent. In RL, instead, we have only access to a *sequence-level* reward function $r(x, y)$, that we want to maximize on average for all prompts and completions.

Definition D.1.1 (Reward maximization)

We define the ***reward maximization*** *problem for model* $\pi(y|x)$ *and reward* $r(x, y)$ *as:*

$$\pi^* = \arg\max_{\pi} \left\{ \mathbb{E}_x \mathbb{E}_{y\sim\pi(y|x)} \left[r(x, y) \right] \right\} \tag{E.1.3}$$

Since the decoding process is non-differentiable (and sometimes the reward itself is non-differentiable, i.e., when unit-testing a code program), we require a new set of tools to optimize the models, which are the focus of the next section (see also Figure F.1.3). In the rest of this section we consider instead specific designs of the reward model by considering the two most common cases: trained rewards for preference data, and rule-based rewards for reasoning.

1.2.2 Reward models for preference data

The first step in training a reward function for preference data is to collect a set of triplets (x, y_1, y_2) describing the users' preferences, where x is a generic prompt and y_1, y_2 two different completions from the same model (or from different models, or variants of the same model), where we assume that y_1 is the preferred outcome. This collection can be done manually, crowdsourced, or obtained automatically via another LLM.[7] This is an interesting topic by itself (e.g., in regard to potential biases due to the human evaluations) - I suggest [Lam25, Chapter 6] for an entry point in the literature and

[7] https://huggingface.co/learn/cookbook/en/llm_judge

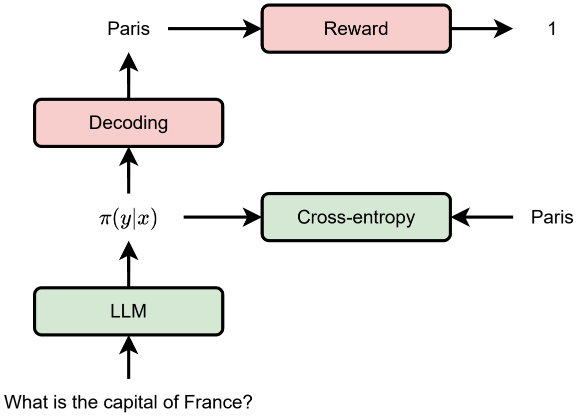

Figure F.1.3: *Difference between optimizing a token-level loss (e.g., cross-entropy) and a reward function. Possibly non-differentiable elements are shown in red, differentiable elements are shown in green instead.*

[LPM+25] for a discussion on how to evaluate the collected data.

The main task we want to solve is to train a model to predict the probability that $y_1 > y_2$ (user prefers completion y_1 to y_2) for two generic completions outside of our preference dataset. Note that this would be trivial if we had access to *absolute* preference values from the users (e.g., on a scale to 1 to 5). In this case we could directly train a **reward model** (RM) $r(x, y)$ to predict the preference values, and then fine-tune the model to maximize the preferences assigned to its completions by the RM.[8]

Given a triplet (x, y_1, y_2), the RM processes the concatenated strings $x \| y_1$, $x \| y_2$ and outputs two real values $r(x, y_1)$, $r(x, y_2)$. One issue for building a loss function is that we only know that $r(x, y_1) > r(x, y_2)$, but we do not know the absolute rewards

[8]Practically, to build $r(\bullet, \bullet)$ we can start from a pre-trained LLM, and replace the token prediction head with a randomly initialized regression head on the EOS tokens.

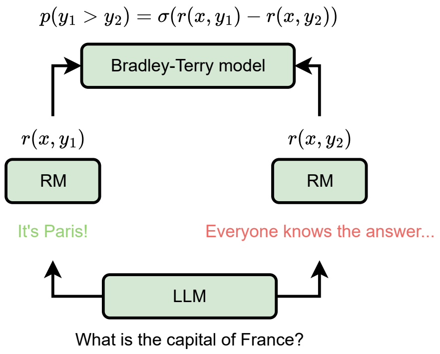

Figure F.1.4: *The (trainable) RM predicts an absolute score for each completion, optimized from preference (paired) data.*

assigned by the labelers. Lucky for us, this is a known problem in statistics and social sciences: for example, the Bradley-Terry model[9] models the preferences as a simple softmax normalization (more sophisticated preference models are also possible).

Definition D.1.2 (Bradley-Terry model)

Given scores $r(x, y_1)$ and $r(x, y_2)$ for options y_1 and y_2, the ***Bradley-Terry model*** *assumes that the user prefers y_1 to y_2 with probability:*

$$p(y_1 > y_2) = \frac{\exp(r(x, y_1))}{\exp(r(x, y_1)) + \exp(r(x, y_2))} \tag{E.1.4}$$

We can train the RM by maximizing (E.1.4) in expectation. In particular, manipulating a bit the expression [Lam25] we obtain a common form for reward modeling:

$$\min \left\{ -\log \left[\sigma \big(r(x, y_1) - r(x, y_2) \big) \right] \right\} \tag{E.1.5}$$

where σ is the sigmoid function, and the minimization is done

[9] https://en.wikipedia.org/wiki/Bradley%E2%80%93Terry_model

w.r.t. (at least) the parameters of the new head. Interestingly, given the form of the Bradley-Terry model, the maximization problem is fundamentally a binary classification problem where the probability of y_1 being better than y_2 is given by $\sigma\big(r(x, y_1) - r(x, y_2)\big)$. Due to the nature of rewards, training RMs has a tendency to overfit, leading to a preference for small base models as a starting point, and a small (even one) number of epochs for training.[10] See Figure F.1.4 for an overview.

Rejection sampling

With a trained RM, a simple baseline for preference alignment (requiring only standard supervised learning) is known as **rejection sampling**. In this case, we first prompt an LLM to provide multiple completions on a set of "interesting" prompts. Second, we use the reward model to score the completions, keeping only the best ones. We finally use the obtained dataset to perform SFT on the base model. Multiple rounds of rejection sampling may be enough to improve the model's alignment depending on the starting quality, although the results tend to be inferior to the use of RL or DPO.

Francesco Verdini: I like this box but I believe I think it would be beneficial to have some intuition on why this is/might be the case, ideally with a few references. The first thing I asked myself when I read about RLHF was "why can't i simply do SFT on the good answers? What's the problem with that?"

[10] Recovering the optimal reward is fundamentally an ill-posed problem: for example, any affine scaling of an optimal reward model produces the same preference data. Thus, directly optimizing the trained RM can be unstable and lead to phenomena of "reward hacking" on the part of the model. For this reason, it is common to normalize reward data over a mini-batch [Lam25] - see also the discussion of the mean-reward baseline in Section 1.3.3.

1.2.3 Verifiable rewards

As we discussed in the introduction, "reasoning" is the idea of allowing an LLM to allocate intermediate tokens (the "reasoning trace") before its final output, in order to force it to decompose the answer, plan, or backtrack over incorrect ideas. From a more general point of view, reasoning can be understood as allowing the model to allocate more compute during the inference time to improve its answer: in this case, we can distinguish between *explicit* reasoning (where the compute is used to generate the reasoning traces) and *implicit* reasoning, where the additional computation is allocated to the model's internals (e.g., running a block of layers multiple times [SDL+25]). We focus on the former case here.[11]

In the most basic setup, recall that we want to modify the model's outputs to include the reasoning trace:

```
<THINK>...</THINK><ANSWER>...</ANSWER>
```

where <THINK>, </THINK>, etc., are properly defined additional tokens (or reused tokens from the original vocabulary). To achieve this, we first select one or more benchmarks where the answer is easily verifiable via one or more rules (**rule-based rewards**), for example:

1. For math benchmarks, the model is forced to output the answer in a predefined format, which is checked against the known result to provide a binary reward.[12]

2. For code benchmarks, the model outputs a code program

[11] A broader term for allocating larger amount of compute to the inference process is **test-time scaling**. This can also be achieved by acting at the decoding level, e.g., by adding "waiting tokens" during the generation [MYS+25], or even by repeatedly sampling multiple outputs via a search procedure [LLG+25].

[12] This is not as trivial as it sounds, e.g.: https://github.com/huggingface/Math-Verify.

that can be run against a known suite of tests, and the fraction of passed tests provides a reward.

In practice, any domain where rules can be defined can be used as reward, greatly expanding the potential for fine-tuning LLMs in specific domains [SYS+25]. In addition, we generally add a (binary) formatting penalty to force the model to provide outputs in the required format.

As we mentioned, DeepSeek-R1 has popularized the use of GRPO [GYZ+25] as RL algorithm (the original model also used additional safety alignment rewards, as well as multiple training runs to improve the readability of the answers [GYZ+25]). RL with verifiable rewards (RLVR) was shown to be successful also with a single example [WYZ+25].

In general, rule-based rewards are not differentiable as they require a complex environment to run, and each evaluation tend to be more costly than the RM forward pass, requiring methods that target sample efficiency above all. This is compounded by the use of the pass@k metric for evaluation [Che21], where k completions are sampled from the model, and a problem is considered solved if at least one of these k samples is correct.

1.3 Reinforcement learning algorithms for post-training

We now proceed to overview some common RL approaches to solve the reward maximization task in (E.1.3), where the reward can be either a trained RM for preference modeling or a series of rules to be satisfied. As we will see, most algorithms can be classified as (a) relaxing the gradient of (E.1.3) (e.g., REINFORCE, Section 1.3.2), or (b) replacing the loss with a differentiable surrogate (e.g., PPO, Section 1.3.7). We begin by introducing some common RL notation and terminology,[13] and how it relates to LLMs.

1.3.1 Interlude: terminology

Do not miss

Once again we consider a generic model taking an input now denoted as s (the reason for the change in notation will be explained shortly) and providing a (potentially stochastic) output $y \sim \pi(y|s)$. In the context of RL, the input represents a (partial) **state** of the environment, while the output is an **action** to be taken, such as a move in an ongoing game. Because the actions determine the behavior of the model, in the context of RL we also refer to π as a **policy**. In many scenarios, actions form a discrete space, and the output of a model is used to parameterize a categorical distribution over this space.

Differently from supervised learning, taking an action modifies the state of the environment according to some transition function $\Theta(s_{t+1}|s_t, y_t)$, which can be stochastic or deterministic. If we start from a given state s_0, take an action, re-evaluate the current state, and iterate the process we obtain what is known as (depending on the circumstances) a **path**, **trajectory**, **episode**, or **rollout**:[14]

$$\tau = (s_0, y_0, s_1, y_1, \dots) \tag{E.1.6}$$

[13] https://spinningup.openai.com/en/latest/spinningup/rl_intro.html

[14] A pedantic footnote in case you are interested in why we have so many synonyms for the same concept: *path* and *trajectory* are fundamentally equivalent and originate from control theory; *episode* is used in RL to distinguish scenarios that admit finite trajectories that have a start and end points (e.g., a chess match) from potentially infinite trajectories; while *rollout* is used to denote a trajectory that is generated according to a policy as opposed to, e.g., a path collected from real-world data.

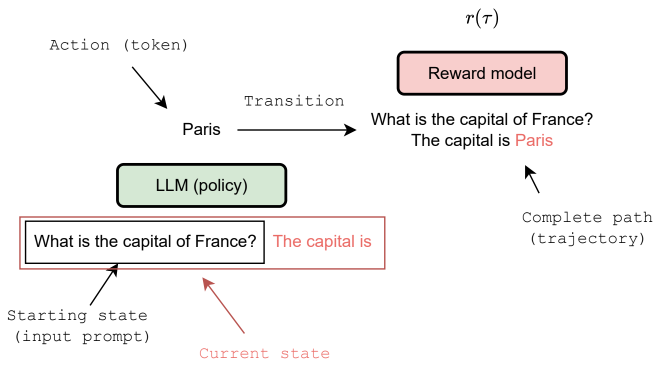

Figure F.1.5: *The LLM parameterizes a policy taking an action by selecting the next token. Once the sequence is completed, the complete path (completion) is given a reward.*

where $y_t \sim \pi(y_t|s_t)$. Every time an action is taken, the model can receive a reward $r_t = r(s_t, y_t)$, which may be sparse (e.g., in most games an agent receives a reward depending on the outcome only at the end of the game). The goal is to train the model in order to maximize this expected reward:

$$\max \mathbb{E}_\tau \left[\sum_t \gamma^t r(s_t, y_t) \right] \tag{E.1.7}$$

where $\gamma \in [0, 1]$ is called the **discount factor**, while the discounted sum of rewards over the trajectory is called the discounted **return** of the policy.[15]

Classical RL algorithms are framed in what is known as *generalized policy iteration* [SB18, Mur24], alternating between finding optimal functions to evaluate states (e.g., **value**

[15]Mathematically, the discount factor ensures that the weight of actions taken far into the future drops exponentially, which has a number of benefits (e.g., the possibility of working with *infinite* trajectories).

functions) and optimal policies that exploit this knowledge. They can also be categorized depending on whether, e.g., they can rely on simulated data, or whether they require single-step transitions (e.g., **temporal difference learning** [SB18]) or complete rollouts, and so on.

When post-training LLMs, we find that this very general setting is vastly simplified (see Figure F.1.5 for a summary):

- The initial state s_0 for a trajectory is a prompt x taken from a dataset (e.g., coding tasks).
- The LLM defines a policy $\pi(y_t|s_t)$ over possible tokens (i.e., the action space corresponds to the vocabulary).
- After a token is sampled, the state evolves according to a deterministic transition function where s_{t+1} is the concatenation of s_t and the new token y_t (a **prefix** in LLM terminology). Since the transition is deterministic, we can also consider the complete decoding $\pi(y|x)$ as a single action, or even the decoding $\pi(y|s_t)$ of the full completion conditioned on an intermediate state.
- Once the EOS token is reached, the full completion receives a reward that is either a trained RM or a combination of verifiers (as in reasoning). We can denote it generally as $r(x,y)$ as before or $r(\tau)$ to denote dependence on a rollout.[16]

Since we only receive a single reward at the end of the sequence, the expected return simplifies and we return to the simpler objective in (E.1.3), which we restate for simplicity with our new notation:

[16]Since we only have a single reward at the end, this is essentially a **contextual bandit problem** with a very large action space (the full sequence). We use RL terminology (policy, advantage, value, TD, etc.) for convenience and to connect with the literature, but the underlying scenario is much simpler than general RL.

$$\max \mathbb{E}_x \, \mathbb{E}_{\tau \sim \pi(y|x)} [r(\tau)]$$

Summary of notation

At the risk of being pedantic, a quick summary of symbols to keep in mind in the following (the specific meaning should always be clear from context):

- $\pi(y_t|s_t)$ denotes a probability distribution over *a single token* y_t starting from an *incomplete* completion s_t.
- $\pi(y|x)$ is a probability distribution over the full completion (with $s_0 = x$).
- Generalizing, $\pi(y|s_t)$ is a probability distribution over the *full* completion, but conditioned on the partial prefix.

All these terms are parameterized by the same LLM as in (E.1.1).

The first expectation can be approximated by taking a batch over prompts $x_1, \ldots, x_m$ sampled from a dataset:

$$\max \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}_{\tau \sim \pi(y|x_i)} [r(\tau)]$$

In the following we focus on the inner expectation term which, once again, is non-differentiable given the decoding process of the LLM. Because of the absence of intermediate rewards and discounting, most RL methods for LLMs use **policy optimization** algorithms, directly optimizing the previous reward on the trained policy, although sometimes having access to value functions can be useful, as we detail later.

What is a "model"?

Possible point of confusion: in the RL field, the term "model" is used to refer to a *world* model, i.e., an explicit representation of the underlying state transition and reward function. By contrast, we mostly use "model" to refer to an *LLM* model, hence to what would be known as a policy in the RL literature. Because we do not train models in the RL sense, all the algorithms below are confusingly known as **model-free**.

1.3.2 Vanilla policy optimization

We start with a relatively simple algorithm known as **vanilla policy optimization**. While not common in practice, it allows us to introduce several interesting aspects that will be useful in the following. In order to optimize (E.1.3) directly we would need access to the **policy gradient** $g = \nabla_\theta \mathbb{E}_\tau [r(\tau)]$, where θ are the trainable parameters of the policy.[17] This in turn would allows us to apply a gradient ascent procedure or some variant, e.g., Adam:

$$\theta = \theta + \alpha g \tag{E.1.8}$$

where α is the learning rate. We can obtain a simple estimator for the policy gradient by exploiting the so-called **log-derivative trick**. For a positive function $f(x) > 0$ we have:

[17]While outside the scope of the chapter, fine-tuning the entire LLM may exceed the computational resources of many labs and companies, especially in the case of large base models. A common alternative is to fine-tune only smaller low-rank updates of selected layers, which significantly reduces the required memory and was shown to work well for RL [SL25]. This is known as **low-rank adaptation** (LoRA), and it is part of a broader set of techniques known as **parameter efficient fine-tuning** [LDR23].

$$\nabla_x f(x) = f(x)\nabla_x \log f(x) \tag{E.1.9}$$

(E.1.9) can be obtained immediately by rearranging the terms in the derivative of $\log f(x)$: $\nabla f(x) = \frac{1}{f(x)}\nabla f(x)$. To see why the log-derivative trick can be useful, we first write down explicitly the expectation inside the reward maximization objective:

$$\mathbb{E}_{y\sim\pi(y|x)}[r(x,y)] = \sum_y r(x,y)\pi(y|x)$$

where the sum is now over all possible completions (of which there are, of course, a combinatorially large number). To perform gradient ascent, we need the gradient of this term, which by the product rule of derivatives decomposes into two separate terms involving, respectively, the derivative $\nabla r(x,y)$ of the reward function and the derivative $\nabla\pi(y|x)$ of the model:[18]

$$\nabla\left[\sum_y r(x,y)\pi(y|x)\right] = \sum_y \left[\nabla r(x,y)\right]\pi(y|x) + \sum_y r(x,y)\left[\nabla\pi(y|x)\right]$$

Since in our scenario the reward model is fixed (no trainable parameters), the red gradient is zero, while the green gradient can be computed easily via reverse-mode automatic differentiation over the model's outputs. Note a very simple interpretation: the policy gradient is a combination of gradients from all possible completions, which are weighted proportionally to their obtained reward. However, explicitly computing all terms in the sum is unfeasible.

To solve this, we apply the log-derivative trick to the second term

[18] Recall that $\nabla\pi(y|x) = \nabla\prod_t \pi(y_t|s_t)$, and (for the following) $\nabla\log\pi(y|x) = \nabla\sum_t \log\pi(y_t|s_t)$.

to obtain a slightly different formulation:

$$\sum_y r(x,y)\, \overbrace{\nabla \pi(y|x)}^{\text{Log-derivative trick: } = \pi(y|x)\nabla \log \pi(y|x)} = \sum_y r(x,y)\left[\nabla \log \pi(y|x)\right] \pi(y|x)$$

The presence of $\pi(y|x)$ on the right side turns it into a new expectation, known as the REINFORCE [Wil92] estimator of the policy gradient.

Definition D.1.3 (REINFORCE [Wil92])

Important

The REINFORCE estimator of the policy gradient is defined as:

$$\nabla\, \mathbb{E}_{y\sim\pi(y|x)}\left[r(x,y)\right] = \mathbb{E}_{y\sim\pi(y|x)}\left[r(x,y)\nabla \log \pi(y|x)\right] \tag{E.1.10}$$

Being an expectation, the right term can be estimated via **Monte Carlo approximation**: first, we draw several completions $y^1, \ldots, y^k$ from the model using the same input prompt x. Then, we compute an approximation of the expectation by averaging the k terms:

$$\nabla\, \mathbb{E}_{y\sim\pi(y|x)}\left[r(x,y)\right] \approx \frac{1}{k}\sum_i r(x,y^i)\nabla \log \pi(y^i|x) \tag{E.1.11}$$

This requires only gradients with respect to the model itself, with the rewards amplifying or decreasing the effect of specific samples (see Figure F.1.6 for a visualization). The resulting policy optimization algorithm is known as **vanilla policy optimization**. Since Monte Carlo samples are drawn from the policy we are optimizing, this is also called an **on-policy** algorithm in the RL literature.

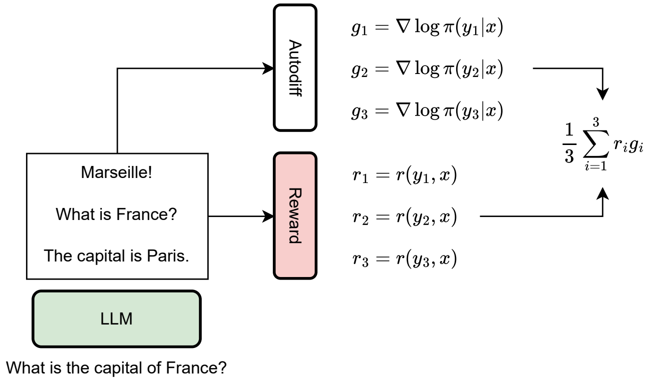

Figure F.1.6: *An example of the REINFORCE estimator applied with $k = 3$ samples.*

1.3.3 Baseline and advantage functions

Looking at the expression (E.1.10) for the policy gradient estimator, we can see that the reward signal $r(x, y)$ is only used as a *global* scalar that multiplies the cumulative gradients for a completion. This has multiple drawbacks: for example, different prompts can have different complexity, but their reward would be identical, meaning their gradients will be upscaled or downscaled by the same factor $r(x, y)$. In addition, not all actions (tokens) contribute equally to the final reward (this is also known as the **credit assignment** problem in RL), different actions at separate time steps can have confounding effects on each other, and so on. As a result, the vanilla REINFORCE estimation can suffer from high variance and the resulting model can be very slow to train.

One possible solution is to recognize that (the proof is on the box on the next page):

$$\mathbb{E}_{y\sim\pi(y|x)}\left[\nabla \log \pi(y|x)\right] = 0 \tag{E.1.12}$$

As an immediate result of this lemma, we can consider a generalized estimator for the policy gradient where we add an additional term $b(x)$ (highlighted in red), which can be any function that does not depend on y:

$$\nabla \mathbb{E}_{y\sim\pi(y|x)}[r(x,y)] \\ = \mathbb{E}_{y\sim\pi(y|x)}[(r(x,y) - b(x))\nabla \log \pi(y|x)] \tag{E.1.13}$$

Proof of (E.1.12)

Expanding the expectation, similarly to how we proceeded for the REINFORCE estimator:

$$\begin{aligned} &\sum_y \pi(y|x)\nabla \log \pi(y|x) && \text{Definition of } \mathbb{E} \\ &= \sum_y \pi(y|x)\frac{\nabla \pi(y|x)}{\pi(y|x)} && \text{Derivative of log} \\ &= \nabla \sum_y \pi(y|x) && \text{Linearity of } \nabla \\ &= \nabla\, 1 = 0 && \pi \text{ is a categorical distribution} \end{aligned}$$

Due to the linearity of the expectation, adding the new term adds a term $-b(x)\cdot\mathbb{E}_{y\sim\pi(y|x)}[\nabla \log \pi(y|x)]$ to the gradient, which by the previous lemma is equal to 0, keeping the estimator unbiased.

Definition D.1.4 (Baseline and advantage functions)

We call $b(x)$ a ***baseline function****, and we define the* ***advantage*** *of action y in state x as:*[a]

$$A(x,y) = r(x,y) - b(x) \tag{E.1.14}$$

[a]Properly, in RL the advantage is defined as the difference between a state-action function and a value function (introduced in Section 1.3.4). In our setup, sampling a completion from π and computing the reward is equivalent to what is known as a **Monte**

> **Carlo estimate** of the state-action function, making our definition equivalent (see [SB18] for more details). Many of our definitions (e.g., value functions) are similarly simplified by this "bandit-style" setup.

The choice of $b(x)$ introduces a degree of freedom that can help in reducing variance, by turning the original problem of maximizing the reward $r(x, y)$ into the problem of maximizing the *advantage* $A(x, y)$.

Important

> **Definition D.1.5 (Policy gradient estimators)**
>
> *Given any baseline function $b(x)$, the following is an unbiased policy gradient estimator:*
>
> $$\nabla\, \mathbb{E}_{y\sim\pi(y|x)}\left[r(x,y)\right] = \mathbb{E}_{y\sim\pi(y|x)}\left[A(x,y)\nabla \log \pi(y|x)\right]$$
>
> *where the advantage $A(x, y)$ is defined as in (E.1.14).*

In particular, one option is to try to reduce the variance by measuring the reward only in relation to the expected reward that could be achieved starting from a given prompt (e.g., simple prompts should contribute less reward). If we have access to several completions $y^1, \ldots, y^k$ and corresponding rewards $r^i = r(x, y^i)$ from the same prompt, we can take their average reward as baseline:

$$b(x) = \text{mean}(r^1, \ldots, r^k) \tag{E.1.15}$$

We will call this the **mean-reward** baseline. Note that if the samples are shared from the mini-batch, this comes at the cost of a bias in the computation of the estimator since the baseline is not fully independent of the action. However, for a fixed dataset the baseline values can be easily pre-computed and stored in memory before running the optimization algorithm for some iterations.

1.3.4 Actor-critic methods

The mean-reward baseline incentives high rewards for "difficult" prompt, i.e., prompts having a low baseline value. However, it says nothing about the specific sequence y which is used to obtain the reward. For example, a code program that works but has a lot of unnecessary functions (*bloat*) would score identically to an equivalent clean piece of code.

To solve this, suppose we define a baseline function $b(s_t)$ for each possible intermediate state: ideally, we would like this baseline to score higher for states that bring us closer to a solution with high rewards. Formally, this is known as a **value function** in RL.

> **Definition D.1.6 (Value function)**
>
> *The **value function** for state s_t is defined as:*[a]
>
> $$V(s_t) = \mathbb{E}_{y \sim \pi(y|s_t)}\left[r(x, y)\right]$$
>
> [a] Properly, in RL this would be denoted by $V^{\pi}(s_t)$ to show dependence on a specific policy. However, we omit the superscript for readability.

As an example, the value of "*What is the capital of France? Par*" would be higher than the value for *"What is the capital of France? Mar"*. As a result, learning to predict "*Par*" as first token should be scored more than learning to predict "*is*" once "*Par*" is given, as it brings us closer to the optimal reward.

If we know the value function, we can decompose the policy gradient estimator on a per-token basis and use the value function as per-token baseline:

$$\mathbb{E}_{y \sim \pi(y|x)}\left[A(x) \nabla \log \pi(y|x)\right] = \mathbb{E}_{y \sim \pi(y|x)}\left[\sum_t \underbrace{A(s_t)}_{\text{Token-wise advantage}} \nabla \log \underbrace{\pi(y_t|s_t)}_{\text{Token-wise probability}}\right] \tag{E.1.16}$$

where we defined a per-token baseline as:

$$A(s_t) = r(x, y) - V(s_t) \tag{E.1.17}$$

The mean-reward baseline, however, is unfeasible in this context because it requires sampling several completions *from each intermediate state*. An alternative approach is to learn the optimal value function, by attaching a separate head to the LLM to predict a scalar value for each token. In RL, methods that learn simultaneously a policy function and a value function are known as **actor-critic** methods, where the actor is the policy and the critic is the value function. Similarly to the mean-reward baseline, we want the value function to provide an estimate of the *expected* reward across possible completions:

$$\mathbb{E}_{y \sim \pi(y|s_t)}[V(s_t) - r(x, y)] = 0 \tag{E.1.18}$$

As always, we can approximate this expectation via a Monte Carlo average. For example, if we re-use the same completion(s) y we are using for training we obtain a simple squared loss:

$$L_{\text{critic}}(x, y) = \sum_t (V(s_t) - \texttt{sg}(r(x, y)))^2 \tag{E.1.19}$$

where use the *stop-gradient* operation $\texttt{sg}$ to highlight that we use $r(x, y)$ as a fixed target over which we do not back-propagate. We can train the complete actor-critic model by alternating one step of gradient descent on the critic loss (E.1.19) and one step of gradient ascent on the reward maximization objective.[19]

[19]Note that in (E.1.19) we are training all value functions $V(s_0), V(s_1), \ldots, V(s_n)$ to regress on the same final reward. Thus, we can also interpret the value function as a generic reward predictor conditioned on the prefixes.

1.3.5 Generalized advantage estimation

> This is a slightly more technical subsection, which introduces a significant amount of RL ideas to discuss a common form of the baseline function. It can be safely skipped on first reading.

We already mentioned that in RL terminology, (E.1.17) is known as a **Monte Carlo** advantage, since we are pushing the value function towards the final reward obtained over one or more completions. At the other extreme, RL theory has developed a rich theory of **temporal difference** (TD) methods, that use an advantage function defined over a limited number of steps, leading to a rich variety of value function and advantage function estimators. To introduce this theory, in this section we briefly move back to the general RL problem from Section 1.3.1, before discussing how the resulting ideas simplify in the LLM scenario.

The fundamental quantity around which one-step methods revolve is known as the **temporal difference** TD(0) error [SB18]:[20]

$$\delta_t = \underbrace{r_t + \gamma V(s_{t+1})}_{\text{Single-step estimate of the return}} - V(s_t) \tag{E.1.20}$$

TD methods can be useful in general RL scenarios since they provide a target for the value function over each time step, and they do not necessarily require full rollouts (full episodes) to work. In fact, it is easy to see that MC and TD(0) are two extremes of a broad set of possible implementations. Suppose we define a discounted sum of TD terms over l time steps:

[20] Technical sidenote: TD methods are generally introduced as *value function* estimators, but we focus here on *advantage* estimators as it is more common in LLM literature.

$$A^l(s_t) = \sum_{t'=0}^{l-1} \gamma^{t'} \delta_{t+t'} \tag{E.1.21}$$

All these sums provide unbiased estimators of the true advantage [SML+15], with decreasing variance as l increases. For $l = 1$ the estimator reduces to the TD(0) error, while for l equal to the sequence length we obtain the MC advantage:

$$A^\infty(s_t) = \sum_{l \geq 0} \gamma^l r_{t+l} - V(s_t) \tag{E.1.22}$$

If we consider an exponential average of the terms $A^1(s_t), A^2(s_t), \dots$ we arrive at the very popular **generalized advantage estimator** [SML+15].

Definition D.1.7 (Generalized advantage estimator)

Given a smoothing hyper-parameter $\lambda \in [0,1]$, we define the GAE advantage [SML+15] for state s_t as:

$$A^\lambda(s_t) = \sum_{l \geq 0} (\gamma\lambda)^l \delta_{t+l} \tag{E.1.23}$$

where λ trade-offs the unbiasedness of the estimator for reduced variance [SML+15].

Extending the formulation in (E.1.17), we can define a target for the value function as:

$$V^\dagger(s_t) = A^\lambda(s_t) + V(s_t) \tag{E.1.24}$$

and regress towards this target:

$$L_{\text{critic}}(x,y) = \sum_t \left[V(s_t) - \mathrm{sg}(V^\dagger(s_t)) \right]^2 \tag{E.1.25}$$

When training LLMs, only the final reward is non-zero, and in general $\gamma = 1$. As a result, all TD terms will converge towards the final reward (the MC advantage): $\delta_t \approx r(x,y) - V(s_t)$, and the GAE estimator becomes a smoothed sum of multiple estimations of the same reward:

$$A^\lambda(s_t) \approx \sum_{l \geq 0} \lambda^l \left(r(x,y) - V(s_{t+l}) \right) \tag{E.1.26}$$

Thus, we keep a similar interpretation as before of the value function being used as a reward predictor, except that we train it using a smoothed average over the sequence, which can significantly improve stability and reduce variance.

1.3.6 Off-policy optimization

Do not miss

All methods we described up to now are known as on-policy methods, since they use rollouts from a policy to update the parameters of the same policy. As we have already mentioned, doing this is expensive, especially if we need to compute multiple completions for the same prompt to perform Monte Carlo estimation of the reward or of the value function. In addition, they always incur in some variance due to the need of estimating the gradient itself.

In principle, one solution for the former problem would be to reuse data (e.g., completions) across several iterations of gradient ascent, i.e.:

1. At fixed intervals, copy the current policy. Denote the copy as $\pi^{\text{old}}(y|x)$.

2. Sample a complete dataset from $\pi^{\text{old}}(y|x)$ and pre-compute all necessary values, including rewards, value functions, and advantages if needed.

3. In-between intervals, use the "stale" data from point (2) to perform training.

The length of the interval trade-offs computation with possibly large differences between the current and the frozen policy. However, naively switching the old policy for the new policy in our loss changes the objective:

$$\mathbb{E}_{y\sim\pi(y|x)}[r(x,y)] \neq \mathbb{E}_{y\sim\pi^{\text{old}}(y|x)}[r(x,y)]$$

To solve this, we can exploit a simple trick known as importance sampling.[21]

Definition D.1.8 (Importance sampling)

For some generic $f(x)$ and distributions $p(x)$, $q(x)$, we can alternate between expectations in one distribution and expectations in the other distribution as:

$$\mathbb{E}_{p(x)}[f(x)] = \int_x \frac{q(x)}{q(x)} f(x)p(x)dx = \mathbb{E}_{q(x)}\left[\frac{q(x)}{p(x)}f(x)\right]$$

where $\rho(x) = \frac{q(x)}{p(x)}$ is known as the ***importance ratio****.*

Thus, to be consistent our new objective requires us to maximize the reward *scaled by the importance ratio*:[22]

[21] https://en.wikipedia.org/wiki/Importance_sampling

[22] The importance ratio requires the two distributions to have equal support (i.e., π^{old} should not assign 0 probability to a predicted token).

$$\begin{aligned} \mathbb{E}_{y\sim\pi(y|x)}\left[r(x,y)\right] = \\ \mathbb{E}_{y\sim\pi^{\text{old}}(y|x)}\left[\frac{\pi(y|x)}{\pi^{\text{old}}(y|x)}r(x,y)\right] = \\ \mathbb{E}_{y\sim\pi^{\text{old}}(y|x)}\left[\rho(x,y)r(x,y)\right] \end{aligned} \tag{E.1.27}$$

This is known as an **off-policy** method, since the policy that is used to generate the data is different from the policy we are optimizing. Since the reward $r(x,y)$ is now a fixed quantity (as it depends on the old, non-trainable policy) this "surrogate" objective is immediately differentiable and can be trained with standard gradient ascent, with no need for gradient estimations.

For better results, we can also combine the off-policy objective with an advantage function.

Important

Definition D.1.9 (Off-policy reward maximization)

Given an advantage function $A^\lambda(s_t)$, the complete off-policy reward maximization is given by (see Figure F.1.7 for a visualization):

$$\pi^* = \arg\max_{\pi} \ \mathbb{E}_{y\sim\pi^{old}(y|x)}\left[\sum_t \rho(s_t)A^\lambda(s_t)\right] \tag{E.1.28}$$

where we have defined a per-token importance ratio:

$$\rho(x,y) = \frac{\prod_t \pi(y_t|s_t)}{\prod_t \pi^{old}(y_t|s_t)} = \prod_t \rho(s_t)$$

This is not an issue if fine-tuning an LLM with a fixed tokenizer, while it should be taken into consideration for more complex scenarios.

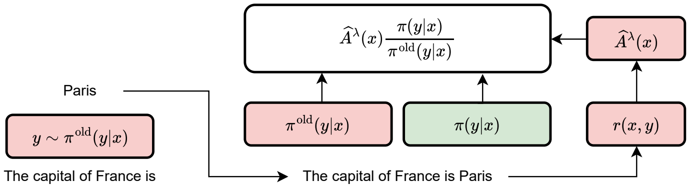

Figure F.1.7: *Off-policy reward maximization, where we show in red the non-differentiable components (sampled from the old policy) and in green the differentiable components.*

> *If the old policy is updated after every iteration, this reverts to the on-policy estimator from before (as $\rho(s_t) = 1$ everywhere).*

If we look at the gradient of (E.1.28), since $A^{\lambda}(s_t)$ and $\pi^{\text{old}}(y|x)$ are constant we have:

$$\nabla\left[\sum_t \rho(s_t)A^{\lambda}(s_t)\right] = \sum_t A^{\lambda}(s_t)\frac{1}{\pi^{\text{old}}(y_t|s_t)}\nabla\pi(y_t|s_t) \quad \text{(E.1.29)}$$

and once again by the log-derivative trick we can replace the gradients $\nabla\pi(y|x) = \pi(y|x)\nabla\log\pi(y|x)$ and obtain:

$$\nabla\left[\sum_t \rho_t(s_t)A^{\lambda}(s_t)\right] = \sum_t A^{\lambda}(s_t)\rho(s_t)\nabla\log\pi(y_t|s_t) \quad \text{(E.1.30)}$$

which is similar to the REINFORCE estimator, albeit with an additional $\rho(s_t)$ term. Conceptually, however, they are very different: in the on-policy case, we sample completions from the current policy, and estimate the gradient via the REINFORCE estimator; in the off-policy case, we compute the loss on a sample of our (off-policy) dataset, and we use (E.1.30) as the *true* gradient of the new loss.

1.3.7 PPO, GRPO, DPO, ...

In the previous sections we introduced the main building blocks used in reinforcement learning for post-training LLMs: policy gradients, advantage functions, off-policy optimization via importance sampling, and value-based baselines. Most practical algorithms used today are - in practice - carefully engineered combinations of these ideas, designed to improve stability, sample efficiency, and robustness at scale. In this section we briefly review some of them, while skipping some implementative details. Given the fast pace of the field, this overview should be seen as a snapshot rather than an exhaustive taxonomy.

Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [SWD+17], popularized in the LLM context by InstructGPT [OWJ+22], can be understood as a **trust-region variant** of off-policy optimization. It starts from the off-policy surrogate objective introduced in Section 1.3.6, but introduces an explicit mechanism to prevent large, potentially destabilizing updates to the policy. Concretely, PPO replaces the importance-weighted reward (or advantage) with a clipped version that avoids too large updates:

$$\begin{aligned}\max \ \mathbb{E}_{y\sim\pi^{\text{old}(y|x)}}\big[\ &\min\big(\rho(x,y)r(x,y),\\ &\text{clip}(\rho(x,y),1-\varepsilon,1+\varepsilon)\,r(x,y)\big)\big]\end{aligned}$$

If an action (completion) has positive reward, PPO allows its probability to increase, but only up to a bounded factor $1+\varepsilon$. Conversely, if the reward is negative, the probability decrease is similarly bounded by $1-\varepsilon$. This prevents the policy from drifting too far from the reference policy π^{old} in a single update, which is particularly important when rewards are noisy or learned (as in RLHF).[23]

Group Relative Policy Optimization (GRPO)

Group Relative Policy Optimization (GRPO) is a simplification of PPO originally proposed for DeepSeekMath [SWZ+24] and later popularized in DeepSeek-R1 [GYZ+25]. The core idea is to eliminate the learned value function and instead define advantages *relatively*, by comparing a completion only to other completions generated for the same prompt.

The original GRPO algorithm is very similar to PPO with a mean-reward baseline (E.1.15), except that rewards are also normalized by their standard deviation:

$$A(x,y) = \frac{r(x,y) - \text{mean}(r_1,\dots,r_n)}{\sqrt{\text{std}(r_1,\dots,r_n) + \varepsilon}}$$

where $r_1,\dots,r_n$ are randomly sampled completions from the (off) policy. However, this formulation can favor easy prompts with small standard deviations - for this reason, Dr. GRPO [LCL+25] is a common variant reverting to the classical mean-reward baseline.

KL regularization

It is also common to augment the reward maximization loss with additional regularizers to prevent overfitting, avoid decreases in performance, or improve the diversity of the answers. For completeness, we shortly overview two popular ones here (knowledge distillation and entropy regularization).

[23] In practice, the reference policy is refreshed frequently (every few gradient steps). Also, because of the clipping, PPO produces a biased loss that is no longer an unbiased estimate of the true policy gradient as in (E.1.30). As a result, PPO is better interpreted as a **soft trust-region** method [SLA+15] approximating an on-policy algorithm, rather than a fully off-policy one.

First, it is common to prefer models that do not deviate too much from the base models, in order to guarantee the same performance. To achieve this, we can apply a **knowledge distillation** (KD) loss to keep the instructed model, $\pi(y|x)$, close to the original base model, denoted by $\pi^{\text{base}}(y|x)$. The reward loss together with the KD term is given by:

$$\mathbb{E}_{x,y}\left[r(x,y)\right] - \beta \mathbb{E}_{x,y}\left[\log \frac{\pi(y|x)}{\pi^{\text{base}}(y|x)}\right]$$

where $\beta > 0$ is a scaling factor and the second expectation is taken w.r.t. the teacher (the base model). To understand the KD term, simply note that:

$$\mathbb{E}_y\left[\log \frac{\pi(y|x)}{\pi^{\text{base}}(y|x)}\right] = -\sum_t \pi^{\text{base}}(y_t|x, y_{<t}) \log \pi(y_t|x, y_{<t})$$

which is a per-token cross-entropy loss, where the target distribution is given by the base model, forcing the two to stay close. When using the same data to approximate the two distributions, the KL term can also be subsumed inside the reward function (**reward shaping**) [ZLY+25], modifying the reward to impose a preference for staying close to the reference distribution at every decoding step. Recent works have experimented with several (stabler) estimators of the KD divergence [SWZ+24], as well as no KD regularization at all [LCL+25].

Second, especially for reasoning tasks, we would like the model to explore as many solutions as possible in the space of possible completions, and avoid collapse early in training. To this end, we can add an entropy regularizer over the tokens:

$$\mathbb{E}_{x,y}\left[r(x,y)\right] - \alpha \mathbb{E}_{x,y}\left[\sum_t \pi(y_t|x, y_{<t}) \log \pi(y_t|x, y_{<t})\right]$$

The entropy term is maximized whenever $\pi(y_t|x, y_{<t})$ is a uniform distribution, and it is scaled by another (small) hyper-parameter α.

Direct Preference Optimization

A final important mention is DPO [RSM+23]. This is a very special case, since DPO (a) can only be applied to preference data, and (b) it works by pure supervised learning, bypassing RL altogether. However, it is now extremely popular for preference alignment. The core insight of DPO is that in preference alignment we know the functional form of our reward, e.g., the Bradley-Terry preference model in (E.1.4). [RSM+23] showed that the KL-regularized policy optimization problem in this case admits a solution satisfying all optimality constraints:

$$p(y_1 > y_2|x) = \frac{1}{1 + \exp\left(\beta \log \frac{\pi(y_2|x)}{\pi^{\text{base}}(y_2|x)} - \beta \log \frac{\pi(y_1|x)}{\pi^{\text{base}}(y_1|x)}\right)}$$

where β is the regularization factor for the KL term. Thus, we can avoid the two step process of fitting a RM model to use in an RL loop, and directly perform maximum likelihood on the policy, whose loss can be written as:

$$L(x, y_1, y_2) = -\log \sigma \left(\beta \log \frac{\pi(y_2|x)}{\pi^{\text{base}}(y_2|x)} - \beta \log \frac{\pi(y_1|x)}{\pi^{\text{base}}(y_1|x)}\right)$$

At the time of writing, DPO and its variants are the state-of-the-art approaches to align models on preference data.

Acknowledgments

I thank Marco Montagna for feedback on the draft. Should you have suggestions or improvements, feel free to contact me on Twitter or by email.[24]

[24] https://www.sscardapane.it

Bibliography

[B+21] R. Bommasani et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021. 5

[BKK+22] Y. Bai, S. Kadavath, S. Kundu, A. Askell, J. Kernion, A. Jones, A. Chen, A. Goldie, A. Mirhoseini, C. McKinnon, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022. 9

[BMR+20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. 2, 6

[Che21] M. Chen. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 20

[CHS19] R. Collobert, A. Hannun, and G. Synnaeve. A fully differentiable beam search decoder. In *International Conference on Machine Learning*, pages 1341–1350. PMLR, 2019. 14

[DCLT19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, pages 4171–4186, 2019. 5

[GYZ+25] D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, R. Xu, R. Zhang, S. Ma, X. Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025. 12, 20, 40

[HBD+19] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019. 14

[HCX+22] K. He, X. Chen, S. Xie, Y. Li, P. Dollár, and R. Girshick. Masked autoencoders are scalable vision learners. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 16000–16009, 2022. 5

[HSW+22] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022. 4

[KGR+22] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022. 11

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *NeurIPS*, 2012. 3

[KSM+24] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, et al. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024. 10

[Lam25] N. Lambert. Reinforcement learning from human feedback. *arXiv preprint arXiv:2504.12501*, 2025. 2, 15, 17, 18

[LBBH98] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 1

[LCL+25] Z. Liu, C. Chen, W. Li, P. Qi, T. Pang, C. Du, W. S. Lee, and M. Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025. 40, 41

[LDR23] V. Lialin, V. Deshpande, and A. Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.15647*, 2023. 25

[LKB+23] H. Lightman, V. Kosaraju, Y. Burda, H. Edwards, B. Baker, T. Lee, J. Leike, J. Schulman, I. Sutskever, and K. Cobbe. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*, 2023. 12

[LLG+25] H. Lai, X. Liu, J. Gao, J. Cheng, Z. Qi, Y. Xu, S. Yao, D. Zhang, J. Du, Z. Hou, et al. A survey of post-training scaling in large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2771–2791, 2025. 2, 19

[LPM+23] H. Lee, S. Phatale, H. Mansoor, T. Mesnard, J. Ferret, K. Lu, C. Bishop, E. Hall, V. Carbune, A. Rastogi, et al. Rlaif vs. rlhf: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023. 13

[LPM+25] N. Lambert, V. Pyatkin, J. Morrison, L. J. V. Miranda, B. Y. Lin, K. Chandu, N. Dziri, S. Kumar, T. Zick, Y. Choi, et al. Rewardbench: Evaluating reward models for language modeling. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1755–1797, 2025. 16

[LYDB17] W. Ling, D. Yogatama, C. Dyer, and P. Blunsom. Program induction by rationale generation: Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*, 2017. 11

[MM20] I. Misra and L. v. d. Maaten. Self-supervised learning of pretext-invariant representations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6707–6717, 2020. 4

[Mur24] K. Murphy. Reinforcement learning: an overview. *arXiv preprint arXiv:2412.05265*, 2024. 22

[MYS+25] N. Muennighoff, Z. Yang, W. Shi, X. L. Li, L. Fei-Fei, H. Hajishirzi, L. Zettlemoyer, P. Liang, E. Candès, and T. B. Hashimoto. s1: Simple test-time scaling. In

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 20286–20332, 2025. 19

[OEN+22] C. Olsson, N. Elhage, N. Nanda, N. Joseph, N. DasSarma, T. Henighan, B. Mann, A. Askell, Y. Bai, A. Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022. 6

[OWJ+22] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022. 8, 39

[RBBNZM21] T. Ridnik, E. Ben-Baruch, A. Noy, and L. Zelnik-Manor. Imagenet-21k pretraining for the masses. *arXiv preprint arXiv:2104.10972*, 2021. 4

[RNS+18] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al. Improving language understanding by generative pre-training. 2018. 5

[RSM+23] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023. 9, 42

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. 22, 23, 30, 33

[SDL+25] N. Saunshi, N. Dikkala, Z. Li, S. Kumar, and S. J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. *arXiv preprint arXiv:2502.17416*, 2025. 19

[SDYD+23] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023. 9, 10

[SL25] J. Schulman and T. M. Lab. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. https://thinkingmachines.ai/blog/lora/. 25

[SLA+15] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015. 40

[SML+15] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015. 34

[SSSG17] C. Sun, A. Shrivastava, S. Singh, and A. Gupta. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE international conference on computer vision*, pages 843–852, 2017. 4

[SWD+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 9, 39

[SWZ+24] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. 40, 41

[SYS+25] Y. Su, D. Yu, L. Song, J. Li, H. Mi, Z. Tu, M. Zhang, and D. Yu. Crossing the reward bridge: Expanding rl with verifiable rewards across diverse domains. *arXiv preprint arXiv:2503.23829*, 2025. 20

[VONR+23] J. Von Oswald, E. Niklasson, E. Randazzo, J. Sacramento, A. Mordvintsev, A. Zhmoginov, and M. Vladymyrov. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*, pages 35151–35174. PMLR, 2023. 6

[WAS+25] J. Wen, Z. Ankner, A. Somani, P. Hase, S. Marks, J. Goldman-Wetzler, L. Petrini, H. Sleight, C. Burns, H. He, et al. Unsupervised elicitation of language models. *arXiv preprint arXiv:2506.10139*, 2025. 13

[WBZ+21] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021. 6

[Wil92] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992. 27

[WWS+22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. 11

[WWT+23] J. Wei, J. Wei, Y. Tay, D. Tran, A. Webson, Y. Lu, X. Chen, H. Liu, D. Huang, D. Zhou, et al. Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846*, 2023. 6

[WYKN20] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys*, 53(3):1–34, 2020. 5

[WYZ+25] Y. Wang, Q. Yang, Z. Zeng, L. Ren, L. Liu, B. Peng, H. Cheng, X. He, K. Wang, J. Gao, et al. Reinforcement learning for reasoning in large language models with one training example. *arXiv preprint arXiv:2504.20571*, 2025. 20

[ZLY+25] Y. Zhang, Y. Liu, H. Yuan, Y. Yuan, Q. Gu, and A. C.-C. Yao. On the design of kl-regularized policy gradient algorithms for llm reasoning. *arXiv preprint arXiv:2505.17508*, 2025. 41