

Chapter 18

Reinforcement Learning for Large Language Models

In Chapter 15, we formulated sequential decision making using Markov Decision Processes (MDPs), and in Chapter 17 we introduced policy gradient methods for optimizing a randomized policy. Large language models fit naturally into the same framework. A language model repeatedly observes a context and samples a next token; the token changes the context seen at the next step, and the resulting sequence can be assigned a reward.

In this chapter, we focus on settings where the reward can be **verified** by the environment. Examples include checking a mathematical answer, running unit tests on generated code, checking a formal proof, or testing whether an agent has reached a desired state. Thus, throughout this chapter, the reward is assumed to be computed by a program, simulator, or other task-specific verifier. We will not consider learned reward functions or preference-based objectives.

The main goal of this chapter is to make the mathematical objects explicit: What are the states and actions? What are the transition probabilities? What role does the language model play? And what objective are we optimizing? We will first answer these questions for ordinary language generation, and then extend the formulation to agents that can call tools and receive observations from an external environment.

18.1 Language generation as an MDP

Let x denote a prompt sampled from a distribution $\mathcal{D}$. The prompt can contain a user request together with any other fixed context supplied to the

model. The model generates tokens $a_0, a_1, \dots$ until generation terminates. We use $\mathcal{V}$ to denote the model vocabulary.

We can describe this process by a finite-horizon MDP. To make contact with the notation in Chapter 15, we write the MDP as

$$(\mathcal{S}, \mathcal{A}, \{P_{sa}\}, \gamma, R),$$

with a maximum generation length $T < \infty$. We now define each component.

18.1.1 States and actions

At token time t , define the state

$$s_t = (x, a_0, \dots, a_{t-1}).$$

Thus, the state is the complete context available to the model before it chooses the next token. The state space $\mathcal{S}$ is the set of all such valid contexts up to the horizon T .

The action space is the vocabulary,

$$\mathcal{A} = \mathcal{V},$$

and action $a_t \in \mathcal{V}$ is the next token emitted by the model. The language model itself is the randomized policy

$$\pi_\theta(a_t \mid s_t),$$

where θ denotes the model parameters. In other words, the LLM plays exactly the role played by π_θ in Chapter 17: it maps a state to a distribution over actions.

If $y = (a_0, \dots, a_{T_y-1})$ is a complete response, then its probability is

$$\pi_\theta(y \mid x) = \prod_{t=0}^{T_y-1} \pi_\theta(a_t \mid s_t),$$

and hence

$$\log \pi_\theta(y \mid x) = \sum_{t=0}^{T_y-1} \log \pi_\theta(a_t \mid s_t).$$

This factorization is the main reason that token-level generation fits the policy gradient framework so directly.

18.1.2 Transition probabilities

For ordinary generation, the transition dynamics are especially simple. Once the model chooses token a_t , the next state is obtained by appending that token to the context:

$$s_{t+1} = s_t \parallel a_t.$$

Thus, the transition is deterministic. If $s' = s \parallel a$, then

$$P_{sa}(s') = 1,$$

and $P_{sa}(\tilde{s}) = 0$ for every other successor state $\tilde{s}$.

This is different from many of the control problems considered in Chapters 15 and 16. There, the transition law was an important property of the physical system and could be stochastic or unknown. For ordinary language generation, appending a token is known exactly. The randomness comes from the policy π_θ , rather than from the state transition.

18.1.3 Rewards and the discount factor

Suppose that after generation terminates we can run a verifier on the complete response. Let

$$\text{ver}(x, y) \in \mathbb{R}$$

denote its score. A common case is a binary verifier,

$$\text{ver}(x, y) = \begin{cases} 1 & \text{if the response passes the verifier,} \\ 0 & \text{otherwise.} \end{cases}$$

For example, the verifier might compare a final mathematical answer against a known answer, or execute unit tests on generated code.

One simple reward function gives zero reward on every intermediate token and places the entire reward on the terminal transition:

$$R(s_t, a_t) = 0 \quad \text{for } t < T_y - 1,$$

and

$$R(s_{T_y-1}, a_{T_y-1}) = \text{ver}(x, y).$$

Nothing requires the reward to be binary. For example, the reward could be the fraction of tests passed, or a sum of several quantities that are all directly measurable by the environment.

Because the episode has a finite horizon, it is usually natural to set

$$\gamma = 1.$$

As in Chapter 16, there is no need for discounting merely to make the return finite. One can still choose $\gamma < 1$ when rewards received sooner should be worth more, but in agentic applications it is often clearer to represent costs such as tool usage or latency explicitly in the reward.

18.1.4 The objective function

Let τ denote the complete token trajectory generated from prompt x . The usual reinforcement learning objective is

$$\eta(\theta) = \mathbb{E}_{x \sim \mathcal{D}} \mathbb{E}_{\tau \sim P_\theta(\cdot|x)} \left[\sum_{t=0}^{T(\tau)-1} \gamma^t R(s_t, a_t) \right]. \quad (18.1)$$

Our goal is to find parameters θ that maximize $\eta(\theta)$.

In the terminal-reward setting with $\gamma = 1$, Equation (18.1) simplifies to

$$\eta(\theta) = \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_\theta(\cdot|x)} [\text{ver}(x, y)]. \quad (18.2)$$

If ver is binary, then $\eta(\theta)$ has a particularly simple interpretation: it is the probability that a response sampled from the policy passes the verifier, averaged over prompts.

Recall the policy gradient identity from Chapter 17. Since the transition probabilities for token appends do not depend on θ , we obtain

$$\nabla_\theta \eta(\theta) = \mathbb{E} \left[\sum_{t=0}^{T(\tau)-1} \nabla_\theta \log \pi_\theta(a_t | s_t) G_t \right], \quad (18.3)$$

where G_t denotes the return following action a_t . With a single terminal reward and $\gamma = 1$, $G_t = \text{ver}(x, y)$ for every model action in the trajectory. As before, a baseline $B(s_t)$ can be subtracted from G_t to reduce variance without changing the expected gradient.

18.2 Verifiable rewards

The formulation above does not require a “correct” token at each timestep. It only requires a way to score the behavior after it has been generated. This

makes reinforcement learning particularly natural for tasks where success can be checked automatically but the sequence of actions needed to obtain success is not known in advance.

We will call a reward **verifiable** when it is computed from observable properties of the trajectory by a fixed task-specific procedure. Some common examples are:

1. For a mathematics problem, parse the final answer and compare it with the known answer.
2. For a programming problem, execute the submitted program on a set of tests and use the number of tests passed as the reward.
3. For a formal reasoning problem, pass the generated object to a proof checker or another deterministic validator.
4. For an interactive task, check whether the environment is in a goal state when the episode terminates.

The key property is that the reward does not need to infer whether the model’s behavior “looks good.” It measures a property that the environment can check. For example, with K unit tests we might use

$$\text{ver}(\tau) = \frac{1}{K} \sum_{i=1}^K \mathbf{1}\{\text{test } i \text{ passes}\}. \quad (18.4)$$

This produces a reward in $[0, 1]$ and gives partial credit when some tests pass. If the task has only a final success/failure condition, Equation (18.4) can be replaced by a single indicator.

Verifiable rewards can also include costs that are exactly observed during the trajectory. For example, an agent that can call tools might use

$$R(\tau) = \text{ver}(\tau) - \lambda_{\text{call}} N_{\text{call}}(\tau) - \lambda_{\text{tok}} N_{\text{tok}}(\tau), \quad (18.5)$$

where N_{call} is the number of tool calls and N_{tok} is the number of model-generated tokens. These penalties are useful when success should be traded off against a measurable resource such as latency or compute.

A difficulty with verifiable rewards is that they are often sparse. A unit test suite may return a score only after the model has written an entire program; a theorem checker may return success only after a long proof has been constructed. This is a credit assignment problem: many token actions may have contributed to the final reward. Equation (18.3) nevertheless assigns the

terminal return to all preceding model actions. Baselines and other variance reduction methods can make this estimator more useful, but the underlying objective remains the verifier score.

When the environment exposes verifiable intermediate events, they can also be included as intermediate rewards. For example, a program that compiles may receive a small reward before the full unit test suite is run. The important point for this chapter is that these rewards are still produced by the task environment rather than by a learned evaluator.

18.3 Agents and tool calling

So far, the environment has done almost nothing: after each action, it simply appends the chosen token to the context. An LLM agent has a richer interaction with its environment. The model may generate a request to call a tool, the environment executes that request, and the result is inserted into the context before generation resumes.

We will consider the following simple interaction protocol. During an assistant generation, the model emits tokens until generation stops for one of two reasons:

`end_turn` or `tool_call`.

If generation stops with `end_turn`, we treat the current task episode as finished and run the verifier. If generation stops with `tool_call`, the model has also generated a tool name and its arguments. The environment executes the tool, appends a tool output to the context, and starts another model generation. Thus, a trajectory can alternate between model generation and environment observations many times before it ends.

Schematically, an interaction has the form

$$\begin{aligned} \text{prompt} \longrightarrow \text{LLM generation} \longrightarrow \text{tool_call} \longrightarrow \text{tool output} \\ \longrightarrow \text{LLM generation} \longrightarrow \cdots \longrightarrow \text{end_turn}. \end{aligned} \tag{18.6}$$

18.3.1 The agent MDP

The state must now contain all information that can affect the model’s next decision. We can again take the state to be the complete serialized history, including the original prompt, all model-generated tokens, all tool calls, and all tool outputs observed so far. In symbols,

$$s_t = (x, z_0, \dots, z_{t-1}),$$

where the z_i may have been generated either by the model or by the environment.

At model-controlled positions, the action is still a token sampled from $\pi_\theta(\cdot \mid s_t)$. The tokens that encode a tool name and its arguments are therefore actions of the policy and are trained in the same way as ordinary language tokens. One can equivalently group an entire structured tool request into a single “macro-action”; the token-level formulation is usually more convenient for connecting the agent back to the LLM policy.

The transition dynamics now have two cases. During ordinary model generation, choosing a token deterministically appends that token to the context. When a complete tool call c is emitted, the external environment produces an observation o according to a tool transition distribution

$$o \sim P_{\text{tool}}(\cdot \mid s, c). \quad (18.7)$$

The next model-visible state is then

$$s' = s \parallel c \parallel o. \quad (18.8)$$

A calculator might make Equation (18.7) deterministic. A search tool, a simulator, or a remote service may produce stochastic outputs. From the RL point of view, all of these are simply environment transition probabilities.

Notice the division of labor. The language model determines the distribution over *actions*. The tool or external system determines the distribution over the *observation* returned after the action. The observation becomes part of the next state, so it can strongly affect what the model does next, but it was not itself chosen by the policy.

18.4 Policy gradients with tool outputs

The distinction between model actions and environment observations is important when we compute policy gradients. Consider a complete trajectory containing both model-generated tokens and tool outputs. Let $\mathcal{M}(\tau)$ denote the set of positions generated by the model. Let the tool calls be $c_1, \dots, c_J$ and the corresponding tool observations be $o_1, \dots, o_J$.

The trajectory probability factors into policy terms and environment terms:

$$\begin{aligned} P_\theta(\tau \mid x) &= \prod_{t \in \mathcal{M}(\tau)} \pi_\theta(z_t \mid s_t) \\ &\quad \cdot \prod_{j=1}^J P_{\text{tool}}(o_j \mid s_j, c_j). \end{aligned} \quad (18.9)$$

The second product does not depend on θ . Therefore,

$$\nabla_{\theta} \log P_{\theta}(\tau \mid x) = \sum_{t \in \mathcal{M}(\tau)} \nabla_{\theta} \log \pi_{\theta}(z_t \mid s_t). \quad (18.10)$$

This is the same phenomenon we saw in the REINFORCE derivation in Chapter 17: transition probabilities disappear from the gradient because they are not parameterized by the policy.

To connect Equation (18.10) to the serialized sequence used to train an LLM, define an origin mask

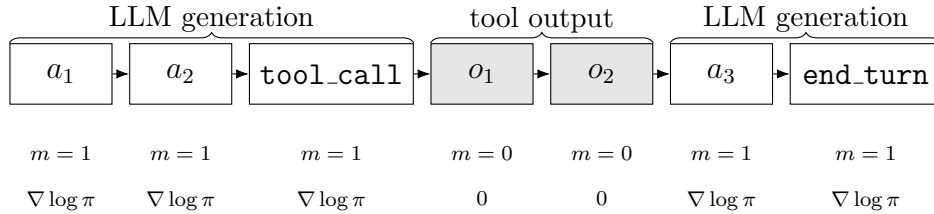
$$m_t = \begin{cases} 1 & \text{if position } t \text{ was generated by the LLM,} \\ 0 & \text{if position } t \text{ came from a tool or environment.} \end{cases} \quad (18.11)$$

Then a sample policy-gradient estimator can be written schematically as

$$\hat{g} = \sum_t m_t \nabla_{\theta} \log \pi_{\theta}(z_t \mid s_t) (G_t - B(s_t)). \quad (18.12)$$

At a tool-output position, $m_t = 0$, so there is no direct policy-gradient term. In an implementation, these positions are masked out of the policy loss. The tool-call tokens, including the tool arguments generated by the model, have $m_t = 1$ and do receive gradient.

The following time-unrolled picture illustrates the masking rule. The gray boxes are tokens supplied by the environment rather than sampled from the policy.



There is one subtle but important point. Masking tool-output positions does not mean that the model ignores tool outputs. The observations o_1, o_2 stay in the context, and the distribution of later model actions, such as a_3 , is conditioned on them. A useful tool output can therefore change later actions and ultimately change the verifier reward. Equation (18.12) says only that we do not assign a log-probability gradient to an observation that the policy did not generate.

If the reward is received only after `end_turn` and $\gamma = 1$, then every model-generated position shares the same terminal return. With a baseline, Equation (18.12) becomes

$$\hat{g} = \sum_{t:m_t=1} \nabla_{\theta} \log \pi_{\theta}(z_t \mid s_t) (\text{ver}(\tau) - B(s_t)). \quad (18.13)$$

Thus a successful episode can reinforce not only the final answer, but also the decision to call a tool, the choice of tool, and the arguments supplied to it. The environment’s returned bytes or tokens are observations and are never mistaken for policy actions.

18.5 Example: a coding agent

Consider a programming task. A prompt x specifies a function that the model must implement. The agent can call a tool that executes the current candidate program on public tests. When the model stops with `end_turn`, a hidden test suite evaluates the final submission.

The state contains the problem statement, the program text generated so far, and the results of any previous tool calls. Model-generated source-code tokens and tool-call arguments are actions. The output of the test runner is an environment observation. If there are K hidden tests, a verifiable terminal reward could be

$$\text{ver}(\tau) = \frac{1}{K} \sum_{i=1}^K \mathbf{1}\{\text{hidden test } i \text{ passes}\}. \quad (18.14)$$

Suppose the agent first writes a candidate function, calls the public test runner, sees a failing test, edits the program, and then ends the turn. The trajectory has the rough form

$$\begin{aligned} \text{prompt} &\longrightarrow \text{candidate code} \longrightarrow \text{tool_call}(\text{run tests}) \\ &\longrightarrow \text{test output} \longrightarrow \text{revised code} \longrightarrow \text{end_turn} \longrightarrow \text{ver}(\tau). \end{aligned} \quad (18.15)$$

The policy gradient includes terms for generating the candidate code, the tool call, and the revised code. It contains no policy term for the test output. Nevertheless, the test output changes the state from which the revised code is generated, so learning can favor policies that use observations effectively.

If tool use has a measurable cost, we could instead optimize

$$R(\tau) = \text{ver}(\tau) - \lambda N_{\text{call}}(\tau). \quad (18.16)$$

For small λ , the model is encouraged to use tools when they improve the chance of passing the hidden tests, but not to call them when they do not add enough value. Both pieces of the reward are directly observable: the hidden tests determine correctness and the environment counts the tool calls.

This example also shows why the agentic formulation is genuinely sequential. Before seeing the test output, the model does not know whether the current program will pass. The tool call changes what the agent observes, and the agent can condition later actions on that new information. Treating the whole response as a single one-shot action would hide this interaction structure, whereas the MDP makes it explicit.

18.6 Summary

The reinforcement learning formulation for an LLM can be summarized as follows. For ordinary token generation,

$$\begin{aligned}
 \text{state } s_t &= \text{the context seen by the model,} \\
 \text{action } a_t &= \text{the next model-generated token,} \\
 P_{s_t a_t} &= \text{append the chosen token to the context,} \\
 \pi_\theta(a_t \mid s_t) &= \text{the LLM,} \\
 R &= \text{a task-specific verifiable reward,} \\
 \gamma &= 1 \text{ for the finite-horizon setting considered here.}
 \end{aligned} \tag{18.17}$$

The objective is to maximize the expected return,

$$\max_{\theta} \eta(\theta) = \max_{\theta} \mathbb{E}_{\tau \sim P_{\theta}}[R(\tau)]. \tag{18.18}$$

For a tool-using agent, the state additionally contains environment observations. The model can stop a generation with either `end_turn` or `tool_call`. A tool call causes the environment to execute an external action and append a tool output before the model resumes. The model-generated parts of the trajectory contribute $\nabla \log \pi_{\theta}$ terms to the policy gradient; tool outputs do not. Tool outputs still matter because they become part of the state and affect all subsequent decisions.

Thus, the same MDP and policy-gradient formalism from the preceding chapters extends directly from language generation to agents. The main new feature is that an agent trajectory interleaves *policy actions* with *environment observations*. Once those two sources of tokens are kept separate, the

objective and the gradient follow from exactly the same factorization used in standard reinforcement learning.