3

SUPERVISED FINE-TUNING: THE FOUNDATION TECHNIQUE

"If intelligence is a cake," Yann LeCun said during his famous presentation almost a decade ago at NIPS (as NeurIPS was then known), "the bulk of the cake is unsupervised learning, the icing on the cake is supervised learning, and the cherry on the cake is reinforcement learning." While the exact proportions have been hotly debated (indeed, most of the deliciousness of the current AI cake comes from the icing and the cherry), the sentiment remains true—supervised fine-tuning (SFT) remains the workhorse of post-training LLMs . It is the most widely used technique, the one with the broadest applicability, and often the only one necessary to achieve satisfactory results.

The apparent simplicity of SFT conceals substantial depth. The mechanics of optimization, the geometry of loss landscapes, and above all the art of data curation determine whether a fine-tuning effort succeeds or fails.

Practitioners who understand these foundations can diagnose problems that would mystify those who treat fine-tuning as a black box. Data quality matters more than model size, more than hyperparameter tuning, more than training compute. A thousand carefully curated examples will outperform 10,000 mediocre ones because the model learns from what you show it, and showing it the wrong thing—even if technically correct—teaches the wrong lesson.

This chapter develops SFT from first principles: the causal language modeling objective, gradient-based optimization, and the constraints of batch size and memory. We explore data curation strategies, including synthetic data generation, instruction backtranslation, and quality verification. We examine training dynamics: how to read loss curves, when to stop training, and how to verify that models learned what you intended rather than surface patterns. We address infrastructure requirements, distributed training approaches, and cost estimation. Finally, we develop criteria for when SFT suffices and when RL becomes necessary.

The Mechanics Under the Icing

The causal language modeling loss, gradient-based optimization, and the constraints of batch size and memory are foundations that are worth understanding precisely, because this knowledge enables intelligent debugging when training goes awry.

Learning by Predicting the Next Token

SFT is one of those concepts that appears almost trivial, yet its fundamental importance to post-training means we cannot gloss over its details. SFT not only does much of the heavy lifting in practice but also is foundational to later, more complex techniques—not in a methodological but a practical sense. Often enough, SFT in and of itself is sufficient, but the reverse is almost never true: Jumping to RL without SFT is guaranteed to end in tears and/or wasted GPU time.

The pre-trained model already "knows" language. It can generate fluent text, follow grammatical rules, and draw on vast knowledge encoded during pre-training. But it lacks the specific behavior you want: answering questions in a particular format, adopting a certain tone, or following domain-specific conventions. SFT bridges this gap by showing the model examples of correct behavior—pairs of inputs and desired outputs—and adjusting the model's weights so that, given those inputs, the model assigns higher probability to those outputs. The model learns by example, and the examples define what *correct* means (hence *supervised*). This is why data quality matters so profoundly: The model can learn only behaviors that the training data demonstrates—and it will learn whatever behaviors the data contains, whether you intended them or not.

The model processes tokens $x_1, x_2, \ldots, x_{t-1}$ and produces a probability distribution over the vocabulary for what x_t should be. Training adjusts

weights to make this distribution assign higher probability to the actual next token and lower probability to alternatives. Repeat for every position in every training sequence, and the model learns to generate text that resembles the training data.

This "teacher forcing" paradigm—always conditioning on ground truth tokens rather than the model's own predictions—enables efficient training through parallelization. Every position in a sequence can be processed simultaneously, since each position sees the true preceding tokens rather than waiting for the model to generate them. The efficiency gain is enormous, as what would otherwise require sequential generation becomes a single parallelized forward pass. The price of this efficiency is, of course, that we aren't really training in the same conditions as our model will be called upon to inhabit at the time of inference. In inference, a model *does* have to condition on its own imperfect outputs, which result in the compounding errors so familiar to users of autoregressive LLMs. The model has never practiced recovering from its own mistakes, because training never exposed it to them.

WARNING

Causal*, which has a somewhat specific meaning in language modeling, refers to being directionally limited. An event in the present cannot cause an event in the past. In the same vein, a token at a position cannot influence tokens that came before it. Therefore, in a causal model, a token can attend to only previous tokens and not future ones. Contrast this with masked language modeling, used, for example, in BERT-style models, where tokens can attend in whichever direction they wish. Do not, however, allow the language to lead you to believe that some sort of cause-and-effect relationship is involved here.*

As we know from supervised learning in general, we need a metric of just how far off we are from the desired results. The standard loss function for causal language modeling is cross-entropy, which measures how well the model's predicted probability distribution matches the actual next token. Cross-entropy earned this role because it directly optimizes what we care about: making the model assign high probability to correct continuations. Other loss functions do exist, but cross-entropy's combination of theoretical soundness (being the maximum likelihood estimator) and practical effectiveness has made it ubiquitous.

Understanding this loss mathematically matters because it helps you interpret the loss curve, your key indicator of the training process's overall health. Practitioners have a somewhat unhelpful tendency to consider the loss curve and ignore its values. This is good enough for the most part, because what we care about, first and foremost, is the loss curve going the right way (which at least for SFT is down). However, understanding what it means when "loss: 2.3" drops to "loss: 1.8," does confer a deeper understanding. For this, we must understand how cross-entropy loss is calculated. For a sequence of tokens $x_1, \ldots, x_T$, it takes the form

$$\mathcal{L}(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log p_\theta(x_t \mid x_1, \ldots, x_{t-1})$$

where p_θ denotes the model's predicted probability distribution over the vocabulary. The negative log probability has an intuitive interpretation: At a loss of 2.3, our model assigns a $-\log^{-1}(2.3) \approx 0.1$ probability to the correct token on average, while at a loss of 1.8, it assigns $-\log^{-1}(1.8) \approx 0.165$ probability. In short, that drop from 2.3 to 1.8 means the model has become about 65 percent more confident in its prediction of the correct token. Incidentally, these numbers connect directly to perplexity (the exponentiated loss), which we explore in Chapter 6.

For fine-tuning specifically, the loss is typically computed on only response tokens, not prompt tokens. The model should learn to generate appropriate responses, not to predict the prompt (which is provided at inference time). This "completion-only" loss focuses learning signal on the behavior we want to optimize, ignoring the prompt tokens that merely provide context.

Nudging the Model in the Right Direction

The loss function tells us how wrong the model is but not how to fix it. That is the optimizer's job. An *optimizer* takes the gradient of the loss with respect to each parameter—a vector pointing in the direction of steepest increase—and updates the parameters in the opposite direction, nudging the model toward lower loss. Repeat this process over thousands of batches, and the model gradually improves. The choice of optimizer determines how these updates are computed: how large each step ought to be, whether to treat all parameters equally, and how to handle the noisy gradients that arise from training on small batches rather than the full dataset.

The de facto default optimizer for language model fine-tuning is AdamW.[72] It's useful to understand what it does, because that understanding is essential to identifying those rare instances when other optimizers are a better choice. AdamW is a per parameter adaptive learning rate optimizer. It scales updates inversely with the historical magnitude of gradients. Parameters that have received large gradients get smaller updates, while parameters that have received small gradients get larger ones. This adaptivity helps navigate loss landscapes where different parameters operate at different scales, a common situation in deep networks with varied layer structures.

The *W* in AdamW denotes decoupled weight decay, a refinement that separates regularization from gradient-based updates. Standard L2 regularization adds a penalty proportional to weight magnitude to the loss function, which then affects gradients, while weight decay directly shrinks weights toward 0, independent of the loss gradient. The decoupling matters because it enables the regularization to function properly regardless of learning rate schedule and empirically produces better generalization for language models.

Optimization is conditioned by hyperparameters, the most important by far being the learning rate. The optimal value for fine-tuning differs dramatically from pre-training. Pre-training starts from random initialization and can use learning rates on the order of 10^{-4} to 10^{-3}. Fine-tuning, as the name suggests, starts from a carefully trained model that must not be disrupted,

typically requiring learning rates of 10^{-6} to 10^{-5}. Too high a learning rate during fine-tuning causes the model to "forget" pre-training, losing general capabilities while pursuing the fine-tuning objective. Too low a learning rate, on the other hand, means the model never adapts meaningfully to the fine-tuning data. Learning rate schedules—starting lower and warming up, or starting higher and decaying—can help navigate this sensitivity. My favorite, which happens to also be the most widely adopted, is cosine decay with warm-up, which eases down after an initial ramp (warm-up).

How do we pick the ideal values for our hyperparameters? Some, of course, are conditioned by our environment (we can push the batch size only so high before the GPU decides that it's too big of a bite to chew). For others, practitioners used to rely on guesswork and experience.

Fortunately, we now have hyperparameter optimization (HPO) frameworks like Optuna that use Bayesian optimization to efficiently explore the hyperparameter space, learning from each trial to focus subsequent trials on promising regions.[73] For learning rate specifically, Optuna can search across several orders of magnitude, evaluating each candidate on a shortened training run before committing to full training with the best-performing value. This approach extends naturally to other hyperparameters (weight decay, warm-up steps, batch size) and can optimize them jointly rather than independently.

The compute cost of hyperparameter search is real, but negligible compared to that of even a single failed training run. I have more than once seen teams waste weeks of GPU time on training runs premised on well-meant but incorrect hyperparameters—effort that a six-hour focused run of Optuna, with Hydra as the configuration manager, readily averted in subsequent runs.

VIBE CHECK: HYPERPARAMETER OPTIMIZATION WITH OPTUNA

Coding agents like Claude Code and Codex are good at much more than writing code—for example, coordinating complex tasks and workflows. While watching hyperparameter searches converge is fascinating the first few times, it gets old rather quickly. Agents, on the other hand, can tirelessly babysit HPO runs.

The `hpo-optuna` agentskill in the agentskills folder of the book's companion repository implements this functionality. Not only does it help set up and run Optuna studies, but it also monitors their progress, summarizes results, and suggests next steps. You can even ask it to generate visualizations of the search process. This offloads the tedium of HPO management to an agent, freeing you to focus on higher-level strategy and analysis.

The skill integrates with Weights & Biases as well as Trackio, an open source alternative developed by the Hugging Face team. Especially if you're training on Hugging Face's serverless Jobs interface, it has become popular to use ephemeral Hugging Face private repos as interim storage for experiment artifacts, then spin up a Hugging Face Space to visualize it with Trackio.

Learning rates have an interesting effect that is sometimes neglected. At times, a cascade of multiple SFT runs is necessary to achieve the desired result. This is typically the case when you need both a broader capability (such as answering MCQ-style questions) and a specific, narrow capability (for example, doing so in the biomedical domain). In such cases, it pays to consider adapting relative learning rates so that the narrower capability is learned at a lower learning rate, typically at least one order of magnitude less than the broader. This protects the broader capability from being inadvertently forgotten while the narrower is being learned. Or, to use the loss landscape geometry we explored in Chapter 2, reducing the learning rate helps ensure that the model remains in the basin of attraction of the broader capability while finding its optimum with respect to the narrower one.

Understanding Batch Size: More Than a Memory Constraint

Batch size occupies a peculiar position in the hyperparameter hierarchy. Those who read tutorials aimed mainly at hobbyists could be forgiven for thinking that batch size is primarily the result of memory constraints. This is true, as reducing batch size is one of the trade-offs available to users with limited VRAM. But there's more to batch size than memory economy.

Larger batch sizes produce more accurate gradient estimates by averaging over more examples, reducing the noise inherent in stochastic gradient descent. This stability can accelerate training by enabling larger learning rates and more confident updates. On the other hand, larger batch sizes do have a potential drawback beyond memory: They tend to find "sharper" minima in the loss landscape, which may generalize less accurately to held-out data than the "flatter" minima that smaller, noisier batches discover.[74]

Of course, regardless of resources, VRAM imposes hard constraints on batch size. Each example in a batch requires memory for activations stored during the forward pass (needed for gradient computation during the backward pass), and this activation memory often dominates total memory usage during training. A model that fits comfortably in memory with batch size 1 may exceed memory with batch size 8, even though the model weights themselves fit easily. The scaling is approximately linear: Doubling batch size approximately doubles activation memory. This is precisely why access to desktop high-VRAM prototyping tools like the NVIDIA DGX Spark is so valuable. They enable experimentation with larger batch sizes that would be infeasible on typical workstations.

An elegant work-around to the VRAM problem is *gradient accumulation*. Instead of processing a large batch in a single forward-backward pass, this technique processes multiple smaller batches, accumulating their gradients before performing a weight update. The mathematical result is identical to a single large batch, as the accumulated gradients equal the gradient of the combined examples. Effective batch sizes far larger than memory permits thus become achievable at the cost of proportionally more forward-backward passes per weight update. The training dynamics of batch size 32

via four steps of accumulation with batch size 8 closely match those of native batch size 32.

A common misconception holds that batch sizes have to be powers of two. They do not. The mathematics of gradient descent is indifferent to whether you train with batch size 32 or 37. GPU hardware historically achieved better utilization with power-of-two tensor dimensions, but modern hardware and frameworks handle arbitrary sizes efficiently. That said, there are practical reasons to prefer "round" numbers.

Evaluation and test set sizes that divide evenly by batch size ensure that every example is seen exactly once per evaluation pass, avoiding the complexity of partial batches. When using gradient accumulation, the per device batch size multiplied by the accumulation steps should yield your target effective batch size, so if you want an effective batch size of 32 with four accumulation steps, you need a per device batch size of 8. Distributed training adds another constraint: The global batch size equals the per device batch size times the accumulation steps times the number of devices, so all three must align. TRL and the Hugging Face `Trainer` will warn you if your dataset size is not divisible by the global batch size, since the final incomplete batch can cause subtle reproducibility issues. None of this requires powers of two specifically, but it does reward choosing batch sizes with convenient factors.

Using Sequence Packing for Efficiency

Most fine-tuning datasets contain examples of varying length. A customer service dataset might include brief clarification questions alongside detailed troubleshooting conversations. Without special handling, each training example occupies a fixed sequence length, with shorter examples padded to match the longest. This padding wastes compute: The model processes padding tokens that contribute nothing to learning.

Sequence packing addresses this inefficiency by concatenating multiple short examples into a single sequence, separated by end-of-sequence tokens. A batch that would contain eight examples with 50 percent average padding can instead pack those eight examples into four sequences with minimal padding, halving compute while processing the same data. The efficiency gains are substantial for datasets with high length variance, where short examples would otherwise waste considerable compute on padding.

The implementation requires attention masking to prevent examples from attending to one another within a packed sequence. Without proper masking, the model might learn spurious correlations between adjacent examples that happen to share a packed sequence. Modern training frameworks like TRL's `SFTTrainer` implement packing correctly, automatically handling the attention mask when packing is enabled.[75] The practitioner need only set a configuration flag.

Packing interacts with loss computation in ways that merit attention. When examples vary dramatically in length, the short examples contribute fewer tokens to the loss than long examples, even when both represent equally important training signal. Some implementations weight loss by example

rather than by token to address this imbalance, though the default token-level weighting works well for most use cases. The choice depends on whether you believe each example or each token should contribute equally to learning.

Garbage In, Garbage Out

If there is one message to take from this chapter, it is this: Data quality matters more than anything else. Neither model size, nor hyperparameter tuning, nor training compute have as much of an impact on the overall outcome as data quality. You cannot out-algorithm poor inputs.

Understanding That Less Is More (When It Is Better)

But what exactly is high-quality data? *Quality* in this context means more than mere correctness, though correctness is necessary. Quality means appropriateness: examples that demonstrate precisely the behavior you want, in the format you need, at the difficulty level that characterizes your deployment scenario.

A dataset of a thousand examples that perfectly capture your use case will outperform one with 10 thousand examples that are merely related. This is especially true for SFT, where the model can learn only differences that the training data more or less directly captures. As a consequence, the data curation process must have some level of intentionality. This is one aspect in which post-training large models is rather different from classical statistical sampling. We are not necessarily looking for representative samples of reality, but for something that is good enough for the model to learn that reality from.

Sampling therefore does not have to be as assiduously unbiased as is expected in more traditional ML contexts. What matters, however, is exposure. What a model is exposed to during training is what it learns, and the absence of meaningful exposure does deprive the model of that opportunity. It is not uncommon to need only a few examples of fairly distinctive content to convey its signal and meaning to the model during SFT, while more subtle information may require more exposure.

WHEN SCALE OBSCURES

Legendary teams start with a hundred solid examples, while novice teams start with a million and wonder why everything is confusing. The model learns from what you show it, and showing it the wrong thing—even if that thing is technically correct—teaches the wrong lesson.

This is, incidentally, how *slop* (low-quality, generic model output) happens. Models trained on oceans of mediocre content learn to produce mediocre content at scale, with all the hallmarks of quantity without quality: verbose, generic, indistinguishable from a thousand other outputs. When scale makes assessing data quality difficult, it goes from ally to enemy.

When a model keeps failing at the SFT stage or begins to exhibit unexpected and undesirable behaviors, it may be helpful to resort to a time-tested approach: ordering as much pizza as can be delivered safely, then manually sifting through the dataset to curate a small, super-high-quality subset, even if it is just a few hundred examples. This is sufficient to establish what good data looks like. And often enough, that's all the model needs to start learning the right behavior. It can then be boosted with additional data from the original set.

Consider a practical example. An organization wants to fine-tune a model for customer service responses. It has access to two datasets: 10,000 historical email exchanges between agents and customers, and a thousand carefully curated examples written by its best agents following its latest style guidelines. The historical data is "real" but contains inconsistent formatting, outdated information, and the full spectrum of agent quality. The curated data is "artificial" but precisely represents desired behavior.

The curated data will almost certainly produce better results, because the model learns what it sees, and the curated data shows what the organization wants. At this point, it is worth once again reiterating that this is not statistical sampling, where we want to know what reality is like. Data is not our guide; it is our tool to shape the model's behavior. We are perfectly free to be as opinionated as we desire in our selection of samples to achieve that objective.

THE BABY BASELINE METHODOLOGY

Before investing in sophisticated training runs, establish what the Hugging Face team calls *baby baselines*.[76] The process is simple: Train an absurdly simple model first, over a small set (maybe 100 examples), fine-tune for a single epoch, and evaluate. This is not meant to produce a good model. It is meant to verify that your pipeline works, your data is formatted correctly, and your evaluation harness produces sensible numbers.

The next step is *vibe testing*. Before running formal evaluations, look at outputs. Generate responses to 20 prompts you care about and read them. Does the model sound right? Does it follow the format you expect? This informal check often reveals problems that metrics miss: subtle format errors, tone mismatches, or capabilities that work but feel wrong. Only after vibe testing confirms that the model is directionally correct should you run expensive formal evaluations.

This methodology saves enormous amounts of wasted compute. I've seen teams run week-long training jobs before discovering their chat template was wrong. I've seen teams celebrate benchmark improvements while their model's actual outputs degraded in ways the benchmarks could not detect. Baby baselines and vibe testing aren't exactly the peak of sophistication and sound statistical methodology, but they do grant an early insight into the training process that, most importantly, reflects much of the subjective experience that SFT (and post-training in general) is meant to inculcate in the model.

Speaking the Model's Language

For the success of SFT, the format in which training data is presented matters as much as its content. Modern instruction-tuned models expect conversations structured in particular ways: special tokens that delimit speaker turns, system messages that establish context, specific patterns that signal the end of assistant responses. A model trained on data formatted one way will behave unpredictably when prompted with data formatted another way, even if the semantic content is identical. Format mismatch is one of the most common causes of fine-tuning failure and one of the easiest to prevent.

Chat templates provide the solution. A *chat template* is a specification of exactly how to convert a structured conversation—a list of turns with roles and content—into the token sequence the model expects. Different model families use different templates. Llama models expect certain control tokens, Mistral models expect others, and ChatML provides yet another format. By far the easiest way around this somewhat confusing landscape is using the `Tokenizers` class in Transformers, which is exposed throughout the Hugging Face ecosystem's packages. This class comes with a set of the most commonly used chat templates, which can be extended using Jinja, a fairly well-known templating syntax. Chat templates help with automatic conversion from structured data to properly formatted sequences.

Beyond the mechanical question of correct tokenization, format consistency provides semantic benefits. When the model sees consistent formatting during training, it learns not just what to say but when and how to say it. Consistent use of system messages teaches the model to attend to system instructions, and consistent response formatting teaches the model appropriate output structure. Variation in training format creates uncertainty about what the model should produce. The goal is to make deployment conditions identical to training conditions, and format consistency is the foundation of that identity.

CHAT TEMPLATE FORMATS ACROSS MODEL FAMILIES

Different model families expect different chat templates. The following examples are illustrative. Specific tokens and formatting evolve with model versions, so always verify against your model's tokenizer rather than copying these verbatim.

ChatML Format (Used by Many OpenAI-Style Models)

```
<|im_start|>system
You are a helpful assistant.<|im_end|>
<|im_start|>user
What is the capital of France?<|im_end|>
<|im_start|>assistant
The capital of France is Paris.<|im_end|>
```

Llama 3/4 Format

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

You are a helpful assistant.<|eot_id|><|start_header_id|>user<|end_header_id|>

What is the capital of France?<|eot_id|><|start_header_id|>assistant
<|end_header_id|>

The capital of France is Paris.<|eot_id|>
```

In practice, rely on the tokenizer's `apply_chat_template` method rather than constructing these formats manually. This ensures that you match the exact tokens the model was trained to expect.

Cleaning House Before Training

Duplicate examples in training data produce models that memorize specific phrasings rather than learning general patterns. If the same example appears 10 times in the dataset, the model receives 10 times the gradient signal for that exact phrasing compared to unique examples. The model learns that precise wording very well indeed and generalizes poorly beyond it. Even two or three copies of an example can produce measurable memorization effects, and datasets assembled from multiple sources often contain far more duplication.

Note that this point is somewhat distinct from the equally true assertion that models aren't necessarily sensitive to the relative weight of a type, class, or category of examples in the training data vis-a-vis others. Here, the issue is specifically with the multiplicity of identical or near-identical instances that encourage memorization, rather than relative overweight or underweight of a particular class of examples that would not.

Deduplication methods exist on a spectrum, between exact and computationally inexpensive on one end and semantically accurate but computationally typically much more intensive on the other. Exact deduplication catches only identical examples, yet from an SFT perspective, near-duplicates cause similar problems. Two customer service call transcripts that differ only in the customer's name teach the same lesson and should contribute almost the same gradient signal, but without deduplication they contribute doubly.

Fuzzy matching techniques identify near-duplicates despite superficial differences. MinHash efficiently estimates Jaccard similarity between text documents,[77] SimHash creates compact fingerprints that can be compared for similarity,[78] and embedding-based methods measure semantic similarity that catches paraphrases. Embedding-based methods are especially useful in domain-specific contexts, as these may be able to resolve identities that are not syntactically apparent (for example, *macrophage activation syndrome*, *cytokine release syndrome*, and *cytokine storm* denote vaguely the same

pathological entity, but only highly domain-specific embeddings would be able to make this association). A proper deduplication pipeline combines a cascade of these techniques for efficient yet comprehensive deduplication.

Contamination checking addresses a related but distinct concern: overlap between training data and evaluation data. If the model has seen evaluation examples during training, evaluation scores cease to measure generalization and instead measure memorization; in short, they cease to be meaningful. This contamination can be accidental (when training and evaluation data derive from similar sources) or can arise from data processing errors. Before training, the responsible practitioner verifies that no training examples appear in evaluation sets, and ideally checks for near-duplicates as well. The validation that seems to confirm success may be confirming only that the model memorized the test.

Transforming Raw Material into Training Data

The data that trains the model must come from somewhere, and that somewhere determines the ceiling on model quality. Three broad strategies exist for generating training data: *expert annotation* (domain specialists create examples), *trained annotator annotation* (workers without prior domain expertise are taught the task), and *synthetic generation* (models create examples that humans or other models then verify). Each strategy has its place, and most successful projects combine them.

Expert annotation produces the highest quality but at significant cost. A physician writing medical consultation examples brings clinical judgment that no annotation guideline can replicate, and a lawyer writing legal analysis brings professional expertise that takes years to develop. Yet experts are expensive, their time is limited, and annotation may not be the best use of their expertise. Expert annotation typically works best for creating gold-standard seed sets—a few hundred examples that establish the quality bar and calibrate subsequent annotation efforts. These seed examples then define what *good* looks like for trained annotators or synthetic generation.

Trained annotators offer a middle path: better economics than experts, better quality than purely synthetic generation. The key is selection and training. Annotators should demonstrate capacity to learn the task, even if they lack prior domain knowledge. Training should include calibration against expert examples, ongoing feedback as they produce annotations, and quality control mechanisms that catch errors before they propagate. The investment in annotator development compounds, and a skilled annotation team becomes an organizational asset that produces high-quality data across multiple projects.

BUILDING ANNOTATION CAPABILITY

The best annotation teams I've worked with were not hired but grown. Start with people who can write clearly and follow complex instructions, then invest heavily in their domain education. Weekly calibration sessions for discussing borderline cases builds shared judgment. A feedback loop that enables annotators to see how models trained on their data perform can create ownership and motivation. After six months, a well-developed annotation team can produce data that rivals expert annotation—at a fraction of the cost.

Most importantly, treat your annotators well. Not only are they producing the data that makes your models work, but they are also your first line of defense against data quality issues. Empower them to raise concerns, suggest improvements, and take pride in their work. More than once, I've seen enterprises bitterly regret alienating their annotators by treating them like low-skilled, disposable labor. Not only is this unethical and unfair, given their paramount importance to building quality models, it is also a very expensive mistake in the long run.

Generating Synthetic Training Data

Human data generation is, of course, subject to limitations of scale and cost. Synthetic generation is much less constrained and thus has emerged as an attractive option. A capable model can generate thousands of examples in the time a human produces one, but those examples may contain errors, biases, or subtle failures that a human would avoid. Verification therefore becomes essential for filtering synthetic examples to acceptable quality, accepting those that pass scrutiny and discarding those that don't. The efficiency of synthetic generation depends on the pass rate: If most generated examples are acceptable, synthetic generation is efficient, but if most are rejected, the approach may collapse to human input with extra steps.

The research literature on synthetic data has matured considerably, and we now have a well-characterized set of strategies to choose from depending on the circumstances. The decision of which approach to use depends on the data you already have, the capabilities you are trying to instill, and the amount of verification capacity you can deploy. Figure 3-1 summarizes my (somewhat opinionated) decision framework to guide the choice among synthetic data strategies.

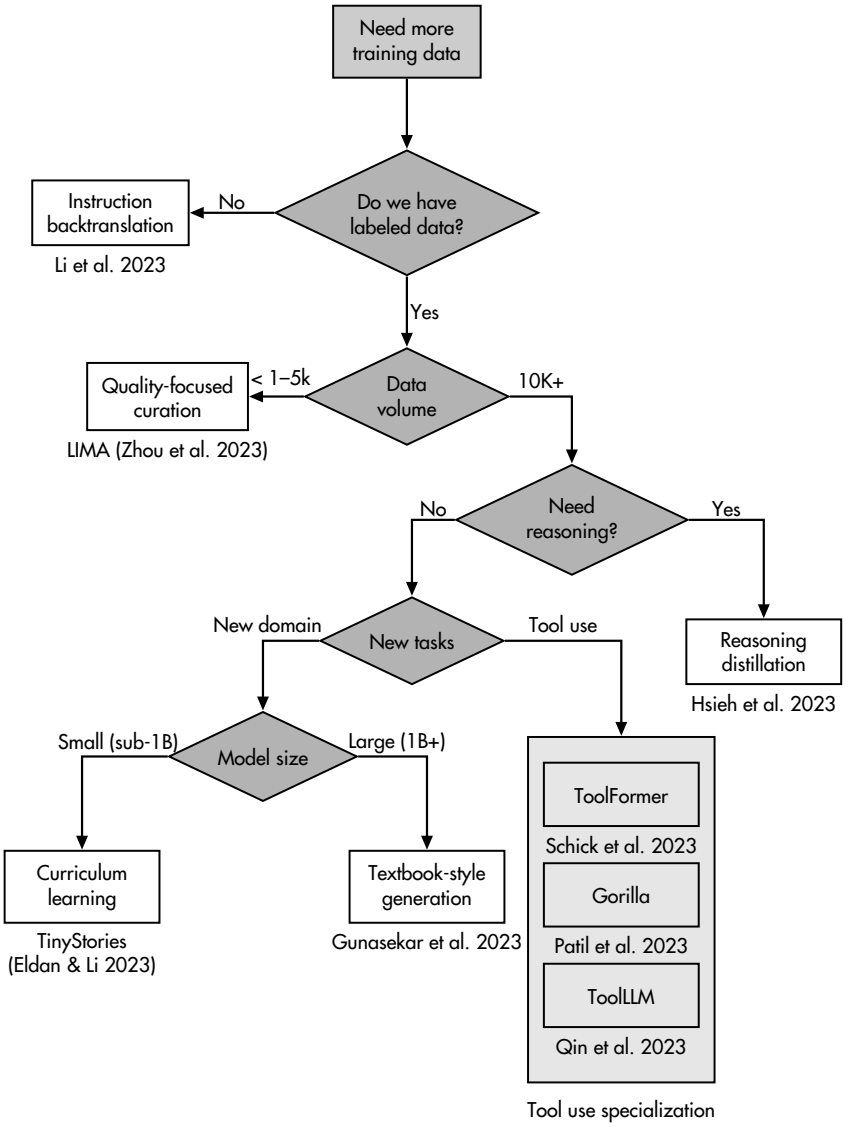

Figure 3-1: A decision flowchart for synthetic data strategies. The choice depends on existing data availability, target capabilities, and whether the task involves tool or API integration. Dashed arrows indicate progressive scaling of complexity within the tool-use family of approaches.

When You Have No Labeled Data

In the most constrained situation, you have raw documents or text but no instruction-response pairs. Instruction backtranslation addresses this by inverting the usual generation direction;[79] rather than generating responses to prompts, you generate prompts that would elicit existing text as a response.

Start with a seed model fine-tuned on a small number of examples, use it to generate instruction prompts for unlabeled documents, and then filter the resulting pairs for quality. The model that produced the backtranslations can itself be used for filtering, creating a self-improving loop. This approach produced Humpback, which at the time of publication outperformed other non-distilled models despite being trained on self-generated data.

When Quality Matters More Than Scale

The LIMA paper demonstrated a result that surprised many practitioners: A 65-billion-parameter model fine-tuned on only 1,000 carefully curated examples performed comparably to models trained on orders of magnitude more data.[80] The implication is that for many tasks, the quality ceiling is determined by the best examples in your dataset, not the quantity. If you have a limited annotation budget, spend it on diversity and quality rather than scale. Cover the range of behaviors you want the model to exhibit, ensure that each example is exemplary, and stop when diminishing returns set in. This quality-focused approach proves especially effective when you can identify what *excellent* looks like and can produce or select examples that achieve it.

Distilling Reasoning from Larger Models

When you have abundant data but want to transfer capabilities from a large model to a smaller one, distillation offers a powerful path. The standard approach fine-tunes the smaller model to match the larger model's outputs, but Cheng-Yu Hsieh et al. in "Distilling Step-by-Step" demonstrated that extracting and training on the reasoning process produces better results than training on outputs alone.[81] The larger model generates not just answers but explanations of how it reached those answers, and the smaller model learns from both.

This approach achieved remarkable efficiency. Models that would normally require orders of magnitude more data matched the performance of much larger teachers. The key insight is that the reasoning trace contains information that raw outputs do not, and smaller models can leverage this information to punch above their weight.

Teaching New Domains Through Curricula

When introducing models to entirely new domains, the structure of training data matters as much as its content. TinyStories demonstrated that even very small models can generate coherent text when trained on appropriately structured synthetic data: short stories using only vocabulary that young children understand.[82] The constrained vocabulary forced the data to exhibit clear narrative structure and logical progression, which the model learned to reproduce.

This curriculum approach, using synthetic data designed to teach concepts progressively from simple to complex, proves especially valuable for domain adaptation. The "Textbooks Are All You Need" paper extended this insight to code generation, showing that training on textbook-style

explanations produced dramatically better results than training on raw code.[83] The phi-1 model, trained on synthetic textbooks and exercises, achieved 50 percent on HumanEval with only 1.3 billion parameters, outperforming models many times its size that were trained on vastly more data.

NOTE *Research has found that synthetic data designed like a textbook (with progressive concept building, worked examples, and exercises with solutions) teaches more effectively than raw examples.[83] The structure itself conveys information that unstructured data lacks. When generating synthetic training data, consider not just what the data says but how it teaches. Explanations that build on previously established concepts, examples that illustrate principles before requiring their application, and exercises that test understanding all contribute to more efficient learning. This principle applies whether the domain is code, reasoning, or specialized knowledge.*

Integrating Tools and APIs

A distinct family of synthetic data strategies addresses tool use and API integration. Toolformer demonstrated that models can learn to use external tools through self-supervised training;[84] the models can generate candidate API calls, execute them, and keep only those that improve predictions. The model learns when to call tools, what arguments to provide, and how to incorporate results, all without explicit supervision. This approach requires only a handful of demonstrations per tool and scales to new tools with minimal additional annotation.

When the goal is integration with specific API ecosystems, Gorilla showed that retrieval-aware training dramatically improves accuracy.[85] By training on API documentation alongside usage examples and providing documentation at inference time, the model learns to generate correct API calls even for APIs not seen during training. This approach significantly reduced hallucinations compared to models that relied on memorized API knowledge.

For complex multistep tool use, ToolLLM introduced depth-first search-based decision trees (DFSDT) to navigate the space of possible API call sequences.[86] The resulting dataset covers over 16,000 APIs across 49 categories, with multistep chains that demonstrate how to compose tools for complex tasks. Models trained on this data learn not just individual tool use but the planning required to orchestrate multiple tools toward a goal.

The choice of synthetic data strategy depends on your starting point and your destination. Figure 3-1 provides a decision framework. If you lack labeled data entirely, instruction backtranslation bootstraps from raw documents. If you have limited data, quality-focused curation following the LIMA approach maximizes impact. If you have abundant data and want to transfer capabilities to smaller models, distillation captures reasoning that raw outputs miss.

WARNING *Models trained on synthetic data generated by other models risk amplifying whatever biases the generator model contains. If the generator produces subtly incorrect reasoning, stereotyped language, or factual errors, the student model learns these patterns as if they were correct. Verification becomes essential, and ideally includes checks specifically designed to catch the failure modes of the generator model. The efficiency of synthetic data makes this verification investment worthwhile, but skipping verification in pursuit of scale produces models that confidently reproduce their teacher's mistakes.*

For new domains, curriculum-style synthetic data teaches more efficiently than unstructured examples. And for tool integration, the progression from Toolformer through Gorilla to ToolLLM offers increasingly sophisticated approaches as your tool ecosystem grows. Chapter 11 discusses synthetic data more extensively.

From Theory to GPU Clusters

With data prepared and objectives understood, we turn to running training. This section traces the path from the basic training loop through monitoring, infrastructure planning, and distributed training. Understanding what happens during training—not just how to configure it—enables informed debugging when things go wrong.

Understanding the Training Loop

Every fine-tuning run, regardless of framework or scale, executes the same fundamental loop. Data flows through the model, loss quantifies error, gradients indicate how to improve, and parameters update accordingly. Understanding this loop demystifies training and grounds discussions of optimization, monitoring, and debugging.

The loop proceeds in steps (Figure 3-2). Each step processes a batch of examples through the forward pass, computing predictions and loss. The backward pass then computes gradients—the partial derivative of the loss with respect to each parameter, indicating how changes to that parameter would affect the loss. The optimizer uses these gradients to update parameters, nudging them in directions that should reduce loss. After the update, the next batch arrives, and the process repeats.

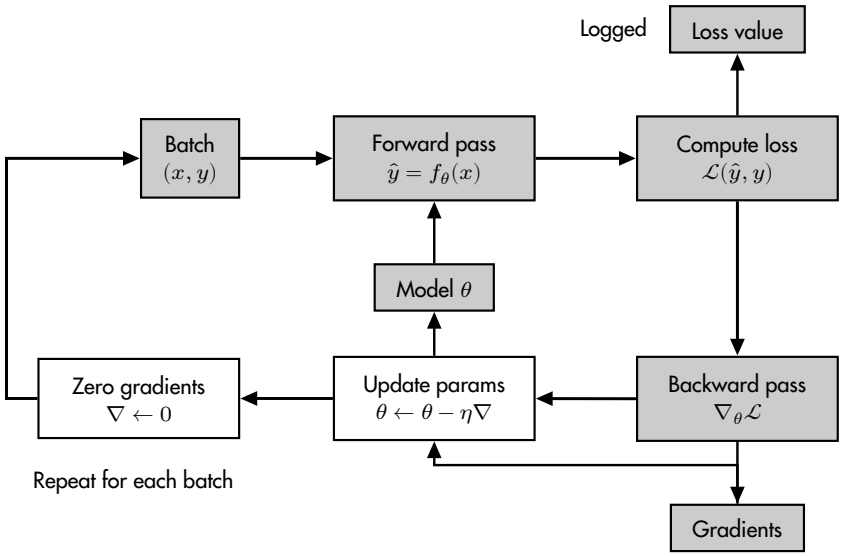

Figure 3-2: The fine-tuning training loop. Each batch flows through the forward pass to compute predictions and loss, then backward through the network to compute gradients. The optimizer updates parameters with the gradients, which are then cleared before the next iteration.

Listing 3-1 shows this fine-tuning training loop in practice.

```
for epoch in range(num_epochs):
    for batch in train_dataloader:
     ❶ outputs = model(**batch)
       loss = outputs.loss

     ❷ loss.backward()

     ❸ optimizer.step()
       scheduler.step()

     ❹ optimizer.zero_grad()
```

Listing 3-1: The essential training loop, stripped to fundamentals

In practice, the loop grows more complex:

- Gradient accumulation splits large effective batch sizes across multiple forward-backward passes before updating.
- Mixed-precision training maintains some computations in lower precision for speed while keeping critical values in higher precision for stability.
- Gradient clipping caps gradient magnitudes to prevent explosive updates.

- Checkpointing periodically saves state to enable recovery from failures.
- Logging records metrics for later analysis.

Yet, beneath these elaborations, the fundamental rhythm remains: forward ❶, backward ❷, update ❸, repeat ❹. Modern frameworks like Hugging Face's `Trainer` and TRL's `SFTTrainer` abstract this loop into configuration, handling the complexities automatically.

Listing 3-2 shows a typical SFT configuration.

```
from trl import SFTTrainer, SFTConfig

config = SFTConfig(
    output_dir="./sft-output",

 ❶ per_device_train_batch_size=2,
    gradient_accumulation_steps=8,

 ❷ learning_rate=2e-5,
 ❸ lr_scheduler_type="cosine",
    warmup_ratio=0.1,

 ❹ num_train_epochs=3,
    max_seq_length=2048,

    logging_steps=10,
    eval_strategy="steps",
    eval_steps=100,

    save_strategy="steps",
    save_steps=500,
    save_total_limit=3,

    bf16=True,
    gradient_checkpointing=True,
)

trainer = SFTTrainer(
    model=model,
    args=config,
    train_dataset=train_data,
    eval_dataset=eval_data,
    tokenizer=tokenizer,
)

trainer.train()
```

Listing 3-2: SFT configuration with TRL's `SFTTrainer`

This configuration specifies batch size ❶, learning rate ❷, schedule ❸, and duration ❹, leaving the framework to manage the loop mechanics.

Performing the Hardware Reality Check

The infrastructure required for SFT scales with model size, but the scaling is gentler than many practitioners expect. A single GPU with 80GB of memory (an A100 or H100) can fine-tune 7-billion-parameter models with gradient checkpointing and careful batch sizing. The NVIDIA DGX Spark's generous unified memory can handle much larger models. Parameter-efficient fine-tuning methods like LoRA further reduce requirements, enabling the same model to be fine-tuned on consumer hardware with 24GB of memory.[87] (We explore these methods in depth in Chapter 7.) The crucial middle does not require the infrastructure of a foundation lab but does require thoughtful resource planning.

Memory during training comprises four components: model parameters, gradients (the same size as the parameters), optimizer states (typically twice the size of the parameters for Adam), and activations (variable, depending on batch size and sequence length). The rule of thumb is that full fine-tuning in `float16` or `bfloat16` requires approximately 16 bytes per parameter: 2 for parameters, 2 for gradients, 8 for optimizer states, and variable overhead for activations. A 7-billion-parameter model thus requires roughly 112GB for training overhead before accounting for activations, explaining why 80GB GPUs are marginal and why parameter-efficient methods are so popular.

The choice between `float16` and `bfloat16` merits brief attention (bit precision is discussed in more detail in Chapter 7). Both use 16 bits, but they allocate those bits differently. The `float16` option dedicates more bits to precision (10 mantissa bits) and fewer to range (5 exponent bits), while `bfloat16` reverses this trade-off (7 mantissa bits, 8 exponent bits). The practical consequence is that `bfloat16` matches `float32`'s dynamic range (it can represent the same extremely large and small values), while `float16` cannot. During training, gradients and activations occasionally spike to values that overflow `float16`'s range, causing `NaN` losses and failed training runs. The `bfloat16` datatype handles these spikes gracefully, making training more stable at the cost of slightly reduced precision that rarely matters in practice. When available (A100, H100, all Blackwell-class GPUs and recent consumer GPUs), `bfloat16` is the better default for fine-tuning. Older hardware that supports only `float16` requires more careful gradient scaling and may need gradient clipping to avoid instability.

Picking Your Weapons

The Hugging Face ecosystem has become the de facto standard for fine-tuning work, and this dominance reflects genuine merit.[88] Its Transformers library provides unified interfaces to most popular model architectures, the Datasets library handles efficient data loading and preprocessing, the

Parameter-Efficient Fine-Tuning (PEFT) library implements parameter-efficient methods like LoRA and QLoRA, the `Trainer` class abstracts common training patterns into a configuration-driven interface. Documentation is extensive, community support is active, and solutions to most problems can be found through search. Starting elsewhere requires justification.

Higher-level frameworks like Axolotl and LLaMA-Factory abstract even further, reducing fine-tuning to configuration files rather than code. These frameworks are valuable for rapid experimentation, as changing from full fine-tuning to LoRA becomes a configuration change rather than a code rewrite. The cost is flexibility. When your requirements diverge from what the framework anticipated, you must either work around the framework's assumptions or descend into its implementation. For standard fine-tuning tasks, this limitation rarely binds, but for novel requirements, it can become prohibitive.

Custom training loops represent the opposite trade-off: maximum flexibility at the cost of development time and expertise. Writing a training loop from scratch requires understanding gradient computation, memory management, distributed training, checkpoint handling, and dozens of other details that frameworks handle automatically. The effort is justified when frameworks prove inadequate, performance optimization requires low-level control, or research pushes beyond established patterns. For most enterprise fine-tuning projects, custom loops are unnecessary. Start with Hugging Face `Trainer` and graduate to custom loops only when `Trainer`'s limitations become binding. Even then, consider whether modifying `Trainer` through callbacks and subclassing might suffice.

HIGHER-LEVEL ABSTRACTIONS: UNSLOTH AND AXOLOTL

Two tools have emerged as particularly valuable for practitioners who want more than the basic `Trainer` but less than a custom loop.[6, 7] Understanding how they relate to the broader ecosystem helps in choosing when to use them.

The Hugging Face stack comprises several layers. At the bottom, Accelerate handles device placement, mixed precision, and distributed training—the infrastructure that makes PyTorch code run efficiently across hardware configurations.[8][9] Above it, TRL's `SFTTrainer` provides training loops specifically designed for language model fine-tuning, with built-in support for chat templates, packing, and completion-only loss.[7][5] Unsloth and Axolotl sit alongside or above these layers, not replacing them but enhancing them.

Unsloth

Unsloth focuses on training speed and memory efficiency through custom Triton kernels and careful memory management. The results are striking: training goes roughly twice as fast with 70 percent less VRAM than standard implementations.

(continued)

A model that requires an A100 with vanilla training might fit on a consumer RTX 4090 with Unsloth.

Beyond raw efficiency, Unsloth provides an exceptional ecosystem for getting started. Its collection of Colab and Kaggle notebooks covers most popular model families and use cases, offering working examples that run on free-tier hardware.

Unsloth also supports loading models in 4-bit or 8-bit precision for inference and fine-tuning, making large models accessible on consumer hardware. The trade-off is that Unsloth supports a narrower range of models and configurations than the general-purpose frameworks, though support expands rapidly.

Axolotl

Axolotl prioritizes configuration-driven experimentation. Training configurations live in YAML files, making it trivial to switch between full fine-tuning, LoRA, and QLoRA, or to experiment with hyperparameters. Under the hood, Axolotl uses TRL's `Trainer` and Accelerate for the actual training; it is orchestration rather than reimplementation. This approach suits teams that run many experiments with varying configurations, as code changes become configuration changes.

Axolotl integrates well with Weights & Biases for experiment tracking and supports most popular model architectures. Axolotl's powerful feature of storing configurations as YAML files meshes very well with pluggable configuration managers like Hydra, which has emerged as the de facto standard for managing complex experiment configurations in ML research.[90]

Crucially, these tools are not mutually exclusive. Axolotl can use Unsloth as its backend for memory-efficient training. You might use TRL's `SFTTrainer` directly for simple cases, Axolotl for complex experimentation, and Unsloth when memory or speed constraints bind. Many practitioners maintain familiarity with all three, choosing the right tool for each situation. The common thread is that all build on the same foundations—Transformers for model architectures, Accelerate for distributed training, TRL for training loops—so skills transfer across tools.

Monitoring Training Health

Launching a training run without monitoring is like driving blindfolded; you might reach the destination, but you won't know until you crash or arrive. Monitoring provides real-time visibility into training health, enabling intervention before problems become catastrophic as well as insight into dynamics that inform future runs.

The minimal monitoring setup logs loss at regular intervals. Training loss should decrease, though not monotonically, as noise from stochastic batching causes fluctuation. Validation loss provides the crucial check: If validation loss rises while training loss falls, the model is memorizing rather than generalizing. The gap between training and validation loss indicates overfitting severity. These signals alone enable basic diagnosis, and any training framework provides them by default.

Richer monitoring records additional signals. Learning-rate evolution matters because schedules vary the rate across training, and verifying that the

schedule behaves as expected catches configuration errors. Gradient norms indicate training stability: Exploding gradients (norm suddenly grows large) signal numerical instability, while vanishing gradients (norm approaches 0) suggest that the model has stopped learning. Memory utilization reveals whether the configuration approaches hardware limits, warning of potential out-of-memory failures before they occur.

NOTE *Every training run should track the following at minimum:*

Training loss *This should decrease with noise. A failure to decrease indicates fundamental problems.*
Validation loss *This should track training loss. Divergence signals overfitting.*
Learning rate *Use to verify that the schedule executes as configured.*
Gradient norm *Sudden spikes warn of instability before* `NaN` *losses appear.*
Throughput *Tokens per second indicates efficiency. Sudden drops suggest problems.*

Experiment-tracking platforms like Weights & Biases and MLflow elevate monitoring from observation to analysis. These platforms automatically log metrics, visualize trends in real time, compare runs across experiments, and preserve history for later analysis. The investment in setup pays dividends when debugging a failing run requires comparing it to successful predecessors, or when optimizing hyperparameters requires understanding how previous choices performed.

Integration with Hugging Face `Trainer` requires minimal configuration. Setting `report_to="wandb"` in training arguments enables automatic logging of all standard metrics. Custom metrics can be logged through callbacks, enabling domain-specific monitoring like generation-quality samples at intervals during training. The overhead is negligible (a few API calls per logging step, as Listing 3-3 illustrates), while the insight gained is substantial.

```
from trl import SFTConfig

config = SFTConfig(
    output_dir="./sft-output",
 ❶ report_to="wandb",
 ❷ logging_steps=10,
    . . .
)
```

Listing 3-3: Enabling basic Weights & Biases monitoring

By default, setting Weights & Biases logging ❶ will log training and validation loss, the learning rate schedule, gradient norms, and some system metrics—in this case, every 10 steps ❷.

It may often prove helpful to also sample some generations for a qualitative sense or vibe check of generated quality (Listing 3-4).

```
from transformers.integrations import WandbCallback
import torch
import wandb

class SamplingCallback(WandbCallback):
    def __init__(self, trainer, test_dataset, num_samples=3,
                 max_new_tokens=128):
        super().__init__()
        self.trainer = trainer
        self.sample_ds = test_dataset.select(range(num_samples))
        self.max_new_tokens = max_new_tokens

 ❶ def on_evaluate(self, args, state, control, **kwargs):
        super().on_evaluate(args, state, control, **kwargs)
        rows = []
        for item in self.sample_ds:
            prompt = item["prompt"]
            inputs = self.trainer.tokenizer(
                prompt, return_tensors="pt"
            ).to(self.trainer.model.device)
          ❷ with torch.inference_mode():
                outputs = self.trainer.model.generate(
                    **inputs, max_new_tokens=self.max_new_tokens
                )
            response = self.trainer.tokenizer.decode(
              ❸ outputs[0][inputs["input_ids"].shape[1]:],
                skip_special_tokens=True
            )
            rows.append([prompt, response])
      ❹ wandb.log({"sample_generations": wandb.Table(
            columns=["prompt", "generation"], data=rows
        )})

trainer = SFTTrainer(model, args, train_dataset,. . .)
❺ wandb_cb = SamplingCallback(trainer, eval_dataset, num_samples=5)
❻ trainer.add_callback(wandb_cb)
trainer.train()
```

Listing 3-4: Vibe checking through custom callbacks

The `on_evaluate` hook ❶ fires after each evaluation phase and runs the generation in `torch.inference_mode` ❷, which disables gradient computation for efficiency (`inference_mode` is the modern version of the `torch.no_grad` context manager). We slice the output ❸ to extract only the newly generated tokens, excluding the input prompt, and log it as a W&B `Table` ❹. Note that the canonical way is to instantiate the trainer, then instantiate the callback ❺, and finally add the latter to the former ❻. This returns a convenient way to see side-by-side qualitative comparisons across training steps. Watching

these generations evolve over training provides intuition that loss curves alone cannot. You can see the model's tone shift and its formatting improve, and you can occasionally catch regressions that metrics miss.

Scaling to Multiple GPUs

When models exceed single-GPU memory or training speed becomes a bottleneck, distributed training across multiple GPUs extends reach. The landscape of distributed training strategies has matured rapidly, with frameworks abstracting complexity that once required distributed systems expertise.

Data parallelism replicates the model on each GPU, with different GPUs processing different batches. After each forward-backward pass, gradients synchronize across GPUs through all-reduce, ensuring that all replicas update identically. The effective batch size scales with GPU count; four GPUs each processing batch size 4 yield an effective batch size of 16. PyTorch's `DistributedDataParallel` (DDP) implements this pattern efficiently and remains the default choice when models fit in single-GPU memory.[91]

Model parallelism distributes model layers across GPUs when models exceed single-GPU memory. Different GPUs hold different parts of the model, with activations switching between GPUs during forward and backward passes. The approach enables the training of arbitrarily large models but introduces communication overhead as activations transfer between GPUs. Pipeline parallelism, a variant, reduces this overhead by overlapping computation across micro-batches.

Fully sharded approaches combine the benefits of data and model parallelism. DeepSpeed ZeRO partitions parameters, gradients, and optimizer states across GPUs, gathering them through communication when needed;[92] PyTorch's Fully Sharded Data Parallel (FSDP) does the same. This sharding dramatically reduces per-GPU memory requirements (each GPU stores only a fraction of the total state) while maintaining the simplicity of a data-parallel programming model. A 7-billion-parameter model that requires 80GB for full fine-tuning might fit on four GPUs with 24GB each using FSDP, with each GPU holding roughly one-quarter of the state.

The choice between DeepSpeed and FSDP depends on your requirements and ecosystem. FSDP integrates natively with PyTorch, requiring no additional dependencies, and works well for straightforward sharding scenarios. DeepSpeed offers more configuration options, including CPU and NVMe offloading for extreme memory constraints, and provides additional features like automatic mixed precision and gradient compression.

For fine-tuning workloads within the Hugging Face ecosystem, both integrate through the Accelerate library with minimal configuration changes. An example of a ready-to-go FSDP Accelerate configuration is set out in Listing 3-5.

accelerate_config.yaml

```
compute_environment: LOCAL_MACHINE
distributed_type: FSDP
fsdp_config:
  fsdp_auto_wrap_policy: TRANSFORMER_BASED_WRAP
```

```
  fsdp_backward_prefetch: BACKWARD_PRE
  fsdp_sharding_strategy: FULL_SHARD
  fsdp_state_dict_type: SHARDED_STATE_DICT
mixed_precision: bf16
num_processes: 4
```

Listing 3-5: Enabling FSDP through an Accelerate configuration

Recent benchmarks show that with proper configuration, both approaches achieve similar throughput for typical fine-tuning workloads. FSDP excels in simplicity when staying within PyTorch's native capabilities suffices. DeepSpeed excels when memory pressure requires advanced offloading or when its additional optimizations (like ZeRO-Infinity for NVMe offloading) become necessary. For most enterprise fine-tuning scenarios involving models up to 70 billion parameters on modest GPU clusters, either approach works well.

WARNING *Distributed training introduces failure modes that do not exist in single-GPU training. Deadlocks occur when GPUs wait on communication that never arrives. Gradient synchronization bugs produce different model states on different GPUs, causing training to silently diverge. Out-of-memory errors on one GPU crash the entire run. Debugging these requires understanding distributed systems concepts that many ML practitioners lack. For this reason, you would want to start with single-GPU training until you're confident the configuration works, then scale to distributed. The time saved debugging distributed failures exceeds the time spent on sequential initial development.*

Estimating and Managing Costs

Accurate cost estimation distinguishes feasible projects from aspirational ones and enables the ROI analysis that justifies investment to stakeholders. Costs fall into three major categories: compute, storage, and human effort. Of these, human effort typically dominates for projects with serious quality requirements, though compute receives disproportionate attention because it is easier to measure.

Compute Cost

This scales approximately linearly with model parameters, dataset tokens, and number of epochs. Estimating training time is not a precise science, and accurately guessing is impossible without reflecting on the training logic, the code, and any accelerators. A rough estimate for fine-tuning a 7-billion-parameter model on 100,000 examples for three epochs might be 10 to 20 GPU-hours on an 80GB A100, costing $20 to $50 at typical cloud rates. Larger models, larger datasets, or more epochs scale proportionally. Parameter-efficient methods substantially reduce compute requirements, often by an order of magnitude, shifting the cost structure decisively toward data curation rather than training. Spot instances or preemptible virtual machines can reduce cloud compute costs by up to 70 percent compared to on-demand pricing, albeit with the risk of interruptions that require checkpointing and

recovery strategies. Tools like SkyPilot reduce the engineering overhead through auto-checkpointing and relatively seamless recovery.[93]

Storage

Costs are usually minor relative to compute but can accumulate unexpectedly. Model checkpoints can require tens of gigabytes each, and saving checkpoints every few hundred steps across multiple experimental runs produces terabytes of data. Cloud storage is cheap per gigabyte but expensive when terabytes accumulate unnoticed. A disciplined checkpoint-rotation policy and regular cleanup of failed experiments keeps storage manageable.

Human Effort

This aspect dominates the total cost for projects where data quality matters. A data curator producing 100 carefully crafted examples per day at typical professional rates generates data far more slowly than GPU-hours accumulate. The cost of expert annotation for domain-specific tasks can exceed compute costs by an order of magnitude. This reality argues for investing heavily in data quality tools, verification pipelines, and annotator training, as anything that improves data quality per unit of human effort has an outsized impact on total project economics.

ALWAYS BE COSTING

I have always been in awe of my colleagues at various enterprise clients who could give me an almost dollar-accurate estimate for the cost of various cloud services and infrastructure needs—and they would be right almost every time. As I have come to learn, this is not magic but experience.

What set these star estimators apart from their less proficient kin was not tenure but a habit of always checking the cost of jobs. We may be tempted to just be glad that the job finished with exit code 0, but architects who become proficient at costing almost always spend time retrospectively examining the bill. Was it more than expected? Could we have used a different service? Maybe a different region with slightly more advantageous pricing, or a more disciplined policy around retention? Was the equipment used to the fullest? It may be rather embarrassing to admit having paid for a B200 GPU to run a quick LoRA on a 3B class model, but it also is the kind of mistake you would not make twice. If in doubt, since costs scale fairly predictably, micro-batch tests can be run very quickly to compare vendors, architectures, and speeds—especially if you are following good experiment management practices!

Reading Training Signals

In most of "traditional" ML with gradient-based methods, the gradient norm and the training curve tell most of the story. Not only is this generally not the case in post-training, but many of the indicators can behave rather counterintuitively to a mind used to traditional ML metrics. This section eases us into the habit of metrics-driven post-training with our still familiar case of SFT.

What Loss Curves Are Trying to Tell You

The loss curve is to the ML practitioner what the vital signs chart is to the clinician: the primary indicator of whether things are proceeding as they should. Learning to read loss curves, to recognize healthy patterns and diagnose pathological ones, is among the most valuable skills a practitioner can develop.

It is helpful to recall the geometric intuition of the fine-tuning process that we developed in Chapter 2. The model traverses a high-dimensional loss landscape, descending toward minima that represent better fits to the training data. The loss curve traces this descent over time, revealing how effectively the model learns, and the gradient magnitude represents the steepness of the curve—or in more geometric terms, the distance from an idealized minimum in parameter space together with the model's current approach velocity. This intuition is helpful to explain both vanishing gradients (the model is going nowhere) and exploding gradients (going too fast for its own good). Table 3-1 summarizes common loss curve patterns and their interpretations.

Table 3-1: Loss Curve Diagnostics: Patterns and Their Causes

	Pattern	Likely cause	Remedy
	Loss does not decrease	Learning rate too low; data-loading bug; format mismatch	Increase learning rate (LR) by 10×; verify data pipeline; check chat template
	Wild oscillations	Learning rate too high	Reduce LR by 10×; add warm-up
	Sudden spike	Numerical instability; gradient explosion	Enable gradient clipping; reduce LR; check for anomalous examples
	Gradual rise	Catastrophic forgetting	Reduce LR; add regularization; check data distribution
	Train drops, validation rises	Overfitting	Early stopping; increase dropout; add data
	Train ≈ 0	Memorization	Reduce epochs; add regularization; increase diversity
	Both plateau early	Underfitting; capacity limit	Increase LR; train longer; use larger model
	Validation very noisy	Validation set too small; distribution mismatch	Increase val set size; check val distribution

Just as for conventional ML training runs, a healthy fine-tuning loss curve shows rapid initial decline as the model adapts to the fine-tuning distribution, followed by gradual flattening as the model approaches its capacity to fit the training data. The training loss and validation loss should track each other closely, with training loss slightly lower. As training progresses, the gap between them may widen slightly, but they should continue moving in the same direction. When training loss continues to fall but validation loss rises, overfitting has begun, and the model is memorizing training examples rather

than learning generalizable patterns. This divergence signals that training should stop or that regularization should increase.

Other patterns indicate other problems. A loss that fails to decrease suggests a learning rate that is too low, a data-loading bug, or a format mismatch that prevents the model from engaging with the task. A loss that oscillates wildly suggests a learning rate that is too high, causing the model to bounce around the loss landscape rather than descend into a minimum. A loss that decreases but then suddenly spikes may indicate numerical instability, often caused by learning rate or gradient issues. Each pattern has characteristic causes, and learning to map patterns to causes enables rapid diagnosis. The practitioner who stares at the loss curve understands nothing, while the practitioner who reads the loss curve understands the training.

WARNING *A training loss that falls to near zero is not a success signal; it is a warning. In fine-tuning, extremely low loss usually indicates memorization rather than generalization. The model has learned to reproduce training examples verbatim rather than learning patterns that transfer. Watch the validation loss. If training loss plummets while validation loss stagnates or rises, the model is memorizing. Stop training, add regularization, or acquire more-diverse data. A healthy loss curve shows training and validation declining together, with a modest gap.*

The Art of Holding Data Back

Validation loss is meaningful only if the validation set represents deployment conditions. A validation set drawn from the same distribution as training data measures the model's ability to fit that distribution, not its ability to generalize to real-world inputs. The validation set should look like what the model will actually encounter: the same distribution of task difficulty, the same input format, the same edge cases that deployment will present. Constructing such a validation set requires understanding the deployment context deeply enough to sample from it accurately.

The temporal separation between training and validation data matters in many contexts. If training data comes from one time period and deployment will see data from a later period, the validation set should reflect the later period's characteristics. Seasonal patterns, vocabulary drift, and topic evolution all create gaps between training and deployment distributions. A validation set that captures these gaps provides honest assessment of deployment performance, while one that doesn't may overestimate success.

Evaluation frequency presents practical trade-offs. Evaluating after every gradient step catches problems immediately but adds computational overhead and may produce noisy signals. Evaluating only at the end of training is efficient but provides no opportunity to detect and correct overfitting mid-run. Common practice evaluates every few hundred or few thousand steps, frequently enough to catch trends but infrequently enough to be efficient. The evaluation schedule should be fixed before training begins, as adjusting it based on observed results invites the same selection bias problems that fixed validation sets exist to prevent.

Verification Beyond Loss Metrics

Loss curves confirm that the model is learning *something*, but not *what*. A model might achieve excellent validation loss while producing outputs that fail in ways the loss cannot capture: subtle format errors, tone mismatches, or capabilities that technically work but feel wrong. Verification bridges this gap between optimization metrics and actual fitness for purpose.

The simplest verification is generation testing: Present the trained model with representative prompts and examine its outputs. This is the vibe testing mentioned in the baby baseline methodology, but applied systematically. Select 20 to 50 prompts that span your use case, generate responses, and read them carefully. Does the model follow expected formats? Does it maintain appropriate tone? Does it handle edge cases sensibly? Human inspection catches problems that metrics miss, because humans evaluate the gestalt of a response rather than averaging over tokens.

Comparative verification further strengthens confidence. Generate responses from both the fine-tuned model and the base model (or a previous checkpoint) for the same prompts. Side-by-side comparison reveals whether fine-tuning improved behavior or merely changed it. A model that correctly formats responses but loses the base model's factual grounding has not improved, even if its loss decreased. Comparative testing exposes these trade-offs.

Memorization testing checks whether the model generalized or merely memorized. Paraphrase a few training examples and observe the responses. A model that generalized will produce appropriate but distinct responses, while a model that memorized will reproduce training responses verbatim or fail entirely on the paraphrased version. Finding memorization early enables intervention: more diverse data, stronger regularization, or fewer training epochs.

VIBES AREN'T VERIFICATION

Looking at a few rollouts and declaring that the model "looks good" is sometimes tempting. The problem is that we are often training for things that are hard to define and may even be a matter of reasoned disagreement. When does a model trained to use a "professional tone" achieve that? When does a model trained to avoid harmful content actually avoid it?

Relying on vague impressions invites bias and overconfidence. Unfortunately, many well-meaning tutorials fall into this trap. Structured testing to determine whether the model has indeed attained its training objectives is relatively rare. This is a problem when having to reason about what's next—more training, progressing to RL, deployment, or calling it a day and pursuing another approach altogether.

These impressions ought to be quantitative definitions, and in the ROI-focused world of enterprise AI, they usually are. "The rollouts looked fine" is not going to meet the bar for most stakeholders. Therefore, approaching verification scientifically is crucial: Form a hypothesis, clarify your expectations, define how you will measure, then gather evidence. Table 3-2 sets out the key metrics for SFT validation.

For tasks with objective correctness criteria, automated verification complements human inspection. If fine-tuning taught the model to produce valid JSON, validate the JSON. If it taught structured extraction, verify extracted fields against ground truth. Automated checks scale where human inspection cannot, though they catch only the failures you anticipate. The combination of automated checks for known failure modes and human inspection for unknown ones provides reasonable coverage. Objective correctness, of course, is rather relative. Methodologies like LLM-as-a-judge or even simple embedding distance metrics can provide some objectivity. For instance, a small embedding model can often be a good detector of subject pertinence, indicating that the model is responding into the correct conceptual space.

Table 3-2: Quick-Reference Metrics for SFT Verification

Metric	Tests for	Implementation	SFT interpretation
Perplexity	Model confidence on held-out data	TorchMetrics, HF Evaluate	Lower is better; `val` increase signals overfitting
Exact match	Perfect response correctness	Manual or regex	Target 80%–95% for structured tasks
ROUGE-L	Structural similarity to references	HF Evaluate, TorchMetrics	F1 > 0.5 suggests format adherence
BERTScore	Semantic equivalence	*bert-score* package	F1 > 0.85 indicates good alignment
Format validity	Structural compliance (JSON, schema)	*jsonschema*, custom validators	Target 95%–100% for production
Embedding similarity	Topical relevance	Sentence-Transformers	Cosine > 0.7 suggests on-topic responses

Verification proceeds from cheap to expensive (the verification hierarchy), stopping when confidence is sufficient:

Loss curves Training and validation trending correctly? (Seconds to check)

Spot generation Do a handful of outputs look right? (Minutes)

Systematic generation Do 50 diverse prompts produce good outputs? (Hours)

Comparative testing Is the fine-tuned model better than the base? (Hours)

Formal evaluation Benchmarks, human evaluation, A/B testing (Days to weeks)

Most fine-tuning failures are caught by levels 1 to 3. Reserve formal evaluation for models that pass informal verification.

No single metric is going to give you all the information. The notebook accompanying this chapter (*notebooks/chapter-03-sft*) includes a multistage verification pathway, which exemplifies how quantitative and qualitative metrics can work in tandem to help identify training issues.

Saving Your Work (and Your Sanity)

Checkpointing saves model state at intervals during training, enabling recovery from failures and selection of the best model across training. Frequent checkpointing is also a good auditability habit that allows deep offline introspection into the progress of the training run. A training run that crashes without checkpoints loses all progress, while one with checkpoints loses only the progress since the last checkpoint.

The best checkpoint is often not the final one. Overfitting causes validation performance to peak before training ends, meaning the optimal model exists at an intermediate point. Without checkpoints at that point, you cannot recover the previous model. A common pattern saves checkpoints every n steps while also maintaining the best checkpoint seen so far according to validation metrics. At the end of training, you have both the final model and the best-performing one and can choose whichever serves your purposes.

Checkpoint storage, however, does impose costs that must be managed. A 7-billion-parameter model checkpoint occupies roughly 14GB in `float16` precision, and a 70-billion-parameter checkpoint occupies 140GB. Saving every 100 steps across a training run of 10,000 steps produces 100 checkpoints, which may total more than a terabyte for large models.

Rotating checkpoints (keeping only the most recent k and deleting older ones) manages storage while preserving recovery capability. The trade-off is that you cannot return to arbitrarily early states, only to recent ones. In TRL, the `Trainer` class supports automatic checkpoint rotation through the `save_total_limit` parameter, simplifying this management; as a general rule, keeping the last 3 to 5 checkpoints suffices for most recovery needs. One point to keep in mind is that if you rely on early stopping based on validation performance, you may want to set the number of checkpoints to keep to be at least as large as your patience parameter.

When SFT Is Enough

Knowing when to stop at SFT and when to move on to RL is an essential skill. RL and other advanced approaches demand significant additional infrastructure and expertise. For a remarkable range of tasks, SFT proves optimal. For others, it is but one step toward the desired result. Knowing when to stop, whether to move on, and if so when, saves both resources and complexity.

Finding the SFT Sweet Spot

The cases where SFT on its own proves optimal share a common characteristic: The desired behavior can be demonstrated through examples. Recognizing these cases early saves the complexity and cost of techniques that provide no additional benefit.

A canonical success case is format adaptation—teaching a model to produce JSON, Markdown, or other structured outputs. You can write examples of correct formatting, the model can learn from them, and you are done. Style transfer operates similarly: Show the model examples of the desired writing style, and it learns to produce that style. Domain vocabulary adoption, structured output generation, template following, citation formatting—any task where you can produce perfect examples that cover the relevant variation is a candidate for SFT-only training.

The common thread is that you can demonstrate correct behavior, not just recognize it. The distinction is important. SFT requires demonstrations of correct behavior, while RL requires only behavior evaluations. Tasks that fall into the latter category, where quality is easy to assess but hard to generate, are where RL becomes necessary.

NOTE *Ask yourself: "Can I write the perfect response for this input?" If you answer yes for most inputs in your domain, SFT will likely succeed. If you can say only, "I would recognize a good response when I see one," you need RL or human feedback. Format tasks (JSON, structured output) pass the demonstration test easily. Creativity tasks (engaging writing, persuasive argument) often fail it. This single question predicts which technique to use better than any technical consideration. "I know it when I see it" (the Lemon test) is usually a sign that some sort of preference modeling is needed.*

Recognizing Diminishing Returns

Performance gains from additional SFT data or more runs follow a predictable pattern: dramatic improvement from the first examples, progressively smaller gains as the dataset grows.[94] The relationship is approximately logarithmic. A dataset of 100 examples might produce substantial improvement. Growing to 1,000 produces more, and growing to 10,000 produces still more, but the performance increments shrink. Eventually, you reach a regime where doubling the dataset produces gains barely distinguishable from noise.

Recognizing when you have reached this plateau is crucial for resource allocation. More data takes time and money to collect, and if additional data yields negligible improvement, that time and money is wasted. Plotting performance against dataset size (on a log scale) reveals where you are on the curve. If the curve is flattening, more data will not help much. If the curve still rises steeply, more data is the highest-value investment available.

A plateau can mean two things. If validation performance is satisfactory, the plateau means success; the model has learned what it needs to learn, and you are done. If validation performance is unsatisfactory, the plateau means

that SFT has reached its limits for this task. More data will not help because the model has already learned what the data can teach. The gap between current performance and desired performance cannot be closed by more examples of the same kind. Either the data needs to change in character (better quality, different distribution), or the approach needs to change (RL, different model architecture). Recognizing a performance plateau in the unsatisfactory region is a signal to reconsider or evolve strategy, not to collect more data.

Graduating to Reinforcement Learning

As noted at the beginning of this chapter, SFT is a natural starting point. It is not always a destination, however. In fact, the standard post-training pipeline as practiced by frontier labs or sophisticated applications proceeds from SFT to RL. This ordering is not arbitrary convention but reflects the different capabilities each stage develops. SFT establishes the foundations: correct formatting, appropriate vocabulary, task understanding, domain knowledge. RL then refines quality within those foundations, optimizing for aspects of response quality that can be recognized but not easily demonstrated.

This synergistic relationship between SFT and RL is not accidental. In a recent paper, Matsutani et al. analyzed the reasoning trajectories of models trained with SFT versus RL and found a striking asymmetry: SFT expands the space of correct reasoning paths but also preserves incorrect ones, while RL compresses incorrect paths but tends to reduce correct ones as well.[95] This explains why the two-stage approach works so effectively. First, SFT broadens the model's repertoire of valid solutions, then RL prunes away the errors without needing to discover correct paths from scratch. This also underlines the evident utility of multiple cascades of SFT followed by RL, which is now the standard approach for large-scale post-training.

Summary

The transition from SFT to RL should occur when SFT has taught what it can teach but a quality gap remains. The symptoms are easy to spot, albeit somewhat less easy to define. The model produces correctly formatted, topically appropriate responses that nonetheless lack the quality you care about, such as persuasiveness, clinical appropriateness, elegance, reason, insight, or simply that ineffable sense of a response that truly serves the user. You can recognize this quality when you see it, but you cannot easily produce examples that demonstrate it. At this point, RL becomes necessary, because a reward model can capture this recognition, and RL can then optimize toward it. The combination closes the gap that SFT alone cannot, and the next chapter is therefore devoted to the myriad flavors of RL.