

Improving LLM Performance Through Fine-tuning

Now that you've built a chatbot with memory and connected it to real-world data via (RAG), and gave it the power to interact with tools via agents. So far, we've relied on prompts and external tools to guide the model's behavior — but what happens when the model itself lacks the foundational skills to solve a problem?

Imagine your AI assistant can access all the world's textbooks, but still can't solve a basic math word problem. That's the limitation we face: no amount of prompting or retrieval can compensate for weak reasoning.

This is where fine-tuning comes in — not as a replacement for RAG or prompt engineering, but as a powerful complement. Fine-tuning allows us to "teach" the model how to think, reason, and decompose complex tasks, making it truly intelligent rather than just well-informed.

The story so far...

Users love **TaskFriend**, your AI productivity assistant! It remembers what they need to do, and helps them plan ahead with real-world data.

But recently, users have noticed inconsistencies. When asked to calculate deadlines, estimate workloads, or solve simple math problems, TaskFriend often gives up mid-reasoning or produces incorrect results. It knows about formulas, but struggles to *apply* them logically.

This isn't a prompt issue — it's a capability gap. Just like a student who reads a textbook but doesn't understand the math, **TaskFriend** can retrieve information but can't effectively reason through it.

We need a way to upgrade its "thinking engine" — and that's exactly what fine-tuning enables.

Goals

- Understand the principles of model fine-tuning, including loss functions, gradient descent, and training dynamics.
- Differentiate between pre-training and fine-tuning, and explain why fine-tuning is the most practical path to customization.
- Apply Parameter-Efficient Fine-Tuning (PEFT) using LoRA (Low-Rank Adaptation) to reduce memory and cost.
- Diagnose training states — underfitting, overfitting, and convergence — using training and validation loss.
- Iteratively improve a model by tuning hyperparameters like learning rate, LoRA rank, and number of epochs.
- Evaluate fine-tuned models on a benchmark task and measure performance gains.
- Deploy fine-tuned models using either dynamic LoRA loading or full parameter matrix fusion.

Setting Up the Environment

Model fine-tuning requires significant hardware resources, particularly GPU acceleration. To complete the labs in this course, we recommend that you create a GPU-accelerated instance in **Platform for AI's Data Science Workshop (PAI-DSW)**.

If you do not have access to GPU-accelerated environment, or the GPU you have has less than 30GB of VRAM, strongly advise against running this course locally, as you may run out of memory for some of the more memory intensive labs.

We'll continue to use PAI-DSW, but before that, we need to switch to a different instance - one that's **equipped with a GPU**.

If you're already using a GPU-accelerated PAI-DSW instance, continue using it.

If not, see: [00 Setting Up the Environment](#)

Once that's done, let's download our model from the [HuggingFace model library](#):

```
from huggingface_hub import snapshot_download

# Specify the repository ID of the model on Hugging Face Hub
repo_id = "Qwen/Qwen2.5-1.5B-Instruct" # Example: BERT base uncased model

# Specify the local directory where you want to save the model
local_dir = "./model/qwen2_5-1_5b-instruct"

# Download the model
snapshot_download(repo_id=repo_id, local_dir=local_dir)
```

Your Task: Enhancing Mathematical Reasoning in Large Language Models

Improving mathematical problem-solving capabilities has long been a key focus in the development of large language models. For your intelligent assistant to be effective, it must possess basic computational and logical reasoning skills.

To facilitate efficient model fine-tuning, we select a small-parameter open-source model—**Qwen2.5-1.5B-Instruct**—as the base model. This choice balances performance, resource efficiency, and fine-tuning feasibility.

Step 1: Load the Base Model

Begin by downloading and loading the model into memory:

```
%env TF_ENABLE_ONEDNN_OPTS=0

import torch
from swift.llm import (
    get_model_tokenizer, get_template, ModelType, get_default_template_type
)

# Get model information
model_type = ModelType.qwen2_5_1_5b_instruct
template_type = get_default_template_type(model_type)

# Configure local path of model
model_id_or_path = "./model/qwen2_5-1_5b-instruct"

# Initialize the model
kwargs = {}
model, tokenizer = get_model_tokenizer(model_type, torch.float32,
    model_id_or_path=model_id_or_path, model_kwargs={'device_map': 'cpu'}, **kwargs)
model.generation_config.max_new_tokens = 128
template = get_template(template_type, tokenizer, default_system='')
print("✅ Model initialization complete!")
```

Step 2: Observing Model Limitations

We're going to test the model's performance with a simple mathematical problem.

Farmer John has a right-angled triangular field. If the longest edge of the field is 15 meters, and each square meter yields 10kg of carrots, what is the total weight of carrots harvested from the entire field?

Tip: The answer is 540kg!

```
from swift.llm import inference
from IPython.display import Latex, display

math_question = """
    Farmer John has a right-angled triangular field.
    If the longest edge of the field is 15 meters,
    and each square meter yields 10kg of carrots,
    what is the total weight of carrots harvested from the entire field?
    """

query = math_question
response, _ = inference(model, template, query)
print(query)
print("The answer we're looking for is: 540kg\n")
print("-" * 25 + "Model response" + "-" * 25)
display(Latex(response))
print("-" * 23 + "Model response end" + "-" * 23)
```

It's evident that the model struggles to solve even basic mathematical problems accurately. While it is aware of the formula for calculating the area of a triangle, it fails to effectively apply this knowledge to compute the correct weight of the harvested radishes.

Using Retrieval-Augmented Generation (RAG) yields similar limitations. As previously discussed, RAG is analogous to an "open-book exam"—it enhances access to information but does not inherently improve reasoning. In practice, just as students cannot rely on reference materials to compensate for weak mathematical skills, an AI model cannot overcome deficiencies in logical or numerical reasoning through retrieval alone.

The core of mathematical proficiency lies in **logical deduction and step-by-step computation**, not merely in recalling formulas. Therefore, to directly enhance your question-answering assistant's performance on simple math tasks, **model fine-tuning** is essential. By training the model on a curated dataset of mathematical word problems and their reasoning steps, you can strengthen its internal reasoning capabilities.

Note: While fine-tuning improves logical reasoning, **computational accuracy** (e.g., arithmetic precision) can be further ensured by integrating a **calculator plugin** or external tool. This allows the model to offload exact calculations, combining robust reasoning with error-free computation.

In summary, the optimal strategy is:

- Use **fine-tuning** to improve reasoning and problem decomposition.
- Equip the model with a **calculator tool** to handle arithmetic reliably.

This hybrid approach ensures both intelligent understanding and precise results.

The Principles of Model Fine-Tuning (Heavy Math & Theory Warning!)

How models learn

Machine learning: Discovering patterns from data

In traditional programming, you typically encode explicit rules into functions. For example, a simple linear model might be defined as:

$$f(x) = ax$$

Here, a is a known, fixed parameter (also called a weight). This function represents a basic algorithmic model that maps an input x to a predicted output y .

However, in real-world scenarios, the underlying rules (parameters) are often unknown. What you *do* have is observational data — input-output pairs that reflect some hidden pattern.

The goal of **machine learning** is to use this data (the training set) to *learn* the optimal values of the parameters — a process known as **model training**. Instead of manually defining rules, the model infers them from examples.

Measuring performance: Loss function & cost function

To find the optimal parameters, you need a way to measure how well a given set of parameters performs. Let's consider how we're going to evaluate the parameter a in the model $f(x) = ax$.

Loss function: Error on a single example

For each training sample (x_i, y_i) , you can compare the model's prediction $f(x_i)$ with the true value y_i . The difference between them is the **loss** for that sample:

$$L(y_i, f(x_i)) = y_i - ax_i$$

However, using raw differences can lead to positive and negative errors canceling each other out when summed. To avoid this, we use the **squared error**:

$$L(y_i, f(x_i)) = (y_i - ax_i)^2$$

Squaring the error ensures all values are positive and amplifies larger mistakes, making it easier to identify poor parameter choices.

In practice, different models may use different forms of loss functions (e.g., cross-entropy for classification, Huber loss for robustness).

Cost function: Average error across all data

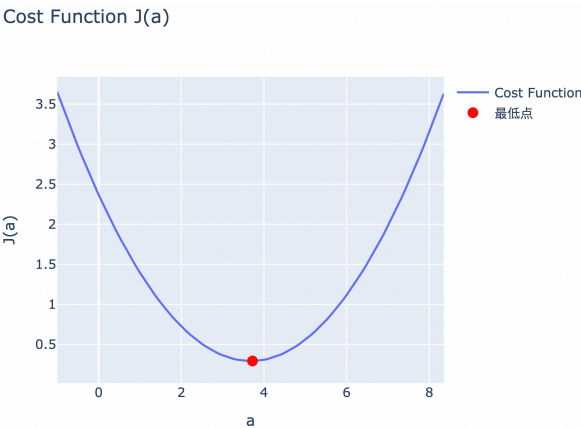
To evaluate the model’s performance across the entire dataset, we compute the average loss over all \$ m \$ samples. This overall measure is called the **Cost Function**, or **Mean Squared Error (MSE)**:

$$J(a) = \frac{1}{m} \sum_{i=1}^m (y_i - ax_i)^2$$

The cost function quantifies how well the model fits the training data as a whole.

In practice, the terms *loss function* and *cost function* are often used interchangeably. In code and engineering contexts, "loss" commonly refers to the average loss across a batch — effectively serving as the cost function.

Minimizing the cost function is equivalent to finding the optimal parameter \$ a \$ that best explains the data. Graphically, this corresponds to finding the **lowest point** on the cost surface.



Improving with each step: Gradient descent

Gradient descent is the most widely used algorithm to find the optimal parameters for a given function. It is an optimization algorithm used to minimize a function by iteratively moving in the direction of steepest descent, which is the negative of the gradient.

Imagine a U-shaped curve: can you spot the **minima**?

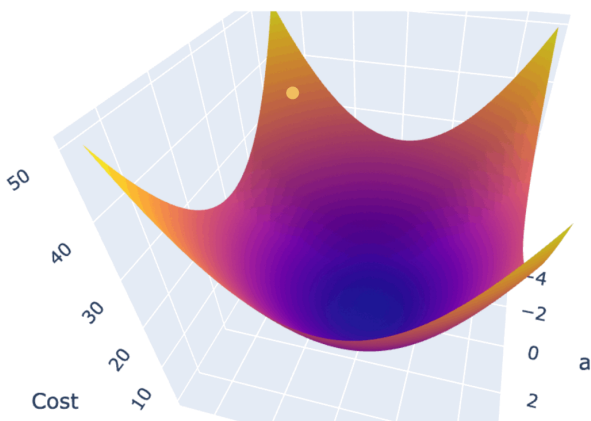
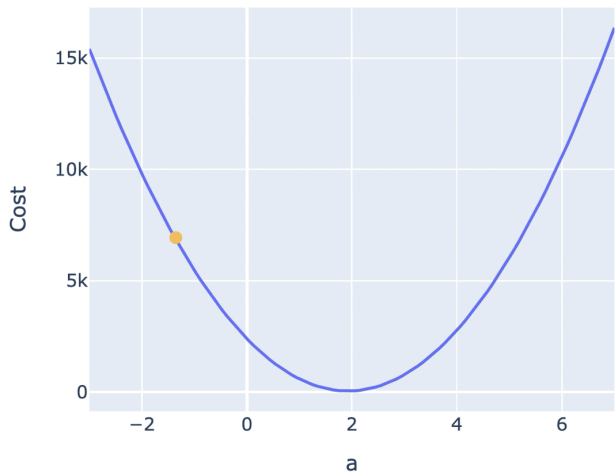
Note: The minima is the *lowest* point in the curve.

Sounds simple, no?

However, machine learning models, it's impossible to visualize and visually identify the minima, as each model is made up of thousands or millions of parameters, making the cost function a high-dimensional surface that cannot be visualized or solved analytically.

The **gradient descent** algorithm works as follows:

- 1. Start with a random initial guess for the parameters.
- 2. Iteratively adjust the parameters in small steps to reduce the cost function.
- 3. Continue until convergence near the minimum.



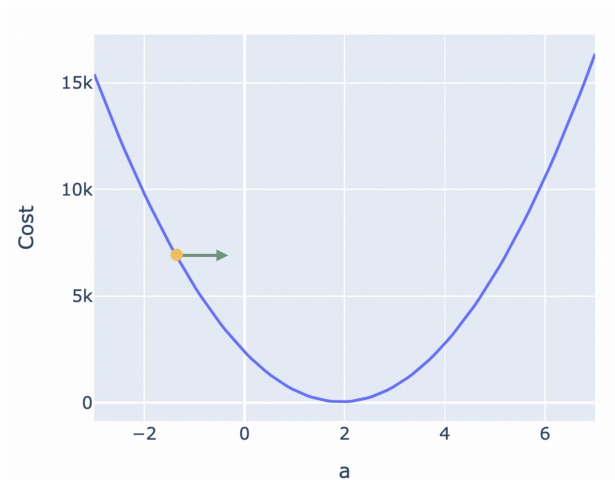
To automate this process, gradient descent must determine two key aspects:

- **Direction:** In which direction should parameters be adjusted?

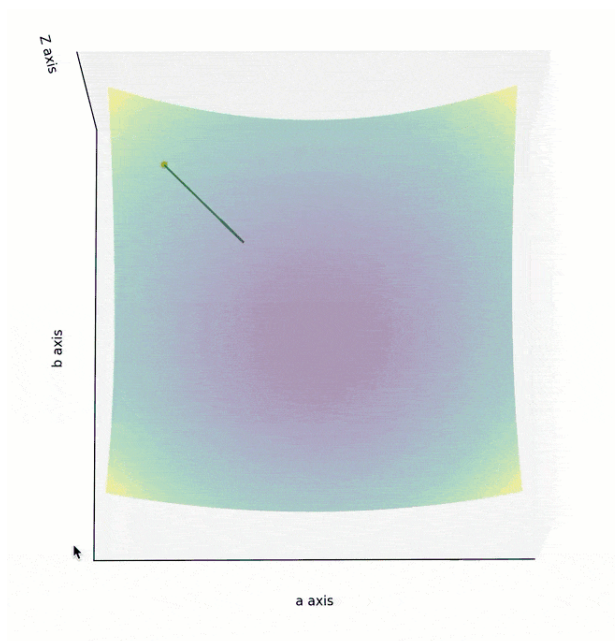
- **Step size:** How large should each step be?

Running down that curve: Descent direction

Back to the U-shaped cost curve. The minima of the curve is at the bottom., the minimum lies at the flattest point. The optimal direction is toward decreasing slope — i.e., **opposite the gradient**.



As we move to higher dimensions (e.g., a 3D surface), the **gradient** is a vector pointing in the direction of steepest ascent. Therefore, moving in the **opposite direction of the gradient** leads to the fastest decrease.

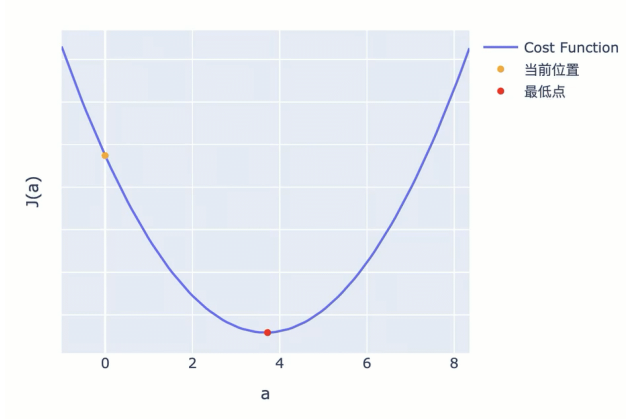


- For a 2D curve $f(a)$, the gradient is simply the derivative (slope).
- For a 3D surface $f(a,b)$, the gradient is a vector of partial derivatives: $\nabla f = \left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b} \right)$, indicating the rate of change along each axis.

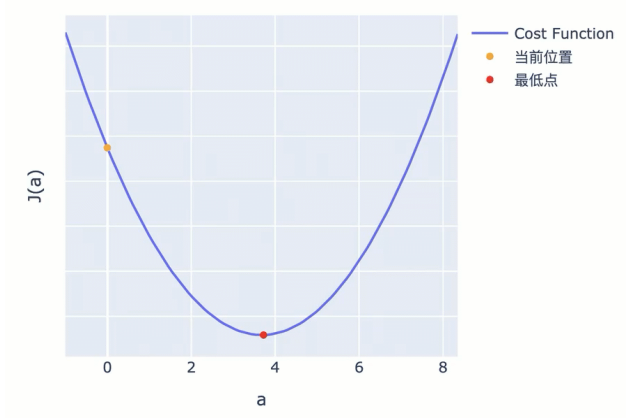
Learning speed: Step size

Now that we know which direction we're going, the next challenge is determining the **step size**.

If we don't choose a proper step size, the worst case scenarios is that our algorithm will "jump" across the minima, and never reach it.



A better approach is to scale the step size by the **magnitude of the gradient** — smaller gradients lead to smaller steps, enabling finer adjustments near the minimum.



However, if the cost function is very steep, even gradient-scaled steps may overshoot the minimum. To control this, we introduce a scaling factor called the **Learning Rate (α)**:

$$\text{Step} = \alpha \cdot \nabla J(\theta)$$

The learning rate critically impacts training:

- **Too high:** May overshoot the minima, causing divergence.
- **Too low:** Converges slowly, consuming more time and resources.
- **Just right:** Efficiently reaches the optimal solution.



Modern training frameworks often use **adaptive learning rate strategies**, such as linear decay or cosine annealing. Tools like Alibaba Cloud PAI's **AutoML** can automatically tune the learning rate for optimal performance.

Key training hyperparameters

Batch size: How many examples at once?

Each iteration of computing the gradient and updating parameters is called a **training step**.

Instead of using one sample (stochastic gradient descent) or the entire dataset (batch gradient descent), most training uses **mini-batches** — subsets of n samples.

- **Larger batch size:** Faster training, more stable gradients, but higher memory usage.
- **Smaller batch size:** Noisier updates, but can improve generalization.

Choosing the right batch size involves trade-offs between computational efficiency, convergence speed, and model performance.

Eval Steps: How often to check progress?

Due to large dataset sizes, evaluation on the validation set is not performed after every full epoch. Instead, it's done periodically — every **eval_steps** training steps — to monitor progress and prevent overfitting.

Epochs: how many times to review the material?

One **epoch** means one complete pass through the entire training dataset.

Since a single epoch may not suffice to converge, training typically runs for multiple epochs. However:

- Too few epochs → underfitting.
- Too many epochs → overfitting and wasted resources.

A common solution is **Early Stopping**:

- Train for a large number of epochs (or indefinitely).
- Monitor validation performance.
- Stop when performance plateaus or degrades.

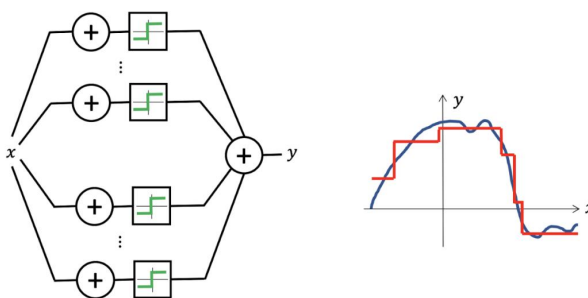
Alternative strategies include dynamic learning rate scheduling based on validation loss trends.

Neural Networks: The AI's Brain

The challenge

In complex tasks like text generation, inputs and outputs are high-dimensional, and the relationship between them is not obvious. How can we model such complexity?

Mathematicians have shown that **neural networks** are **universal function approximators** — they can approximate any continuous function given sufficient capacity.



A single layer is defined as:

$$Y = \sum (W \cdot X)$$

Where:

- X : Input (multi-dimensional)
- W : Weight matrix (learnable parameters)
- σ : Activation function (introduces non-linearity)
- Y : Output

A k -layer network stacks these transformations:

$$Y = \sigma(W_k \cdot \sigma(W_2 \cdot \sigma(W_1 \cdot X)))$$

Activation functions

Activation functions determine whether and how strongly a neuron responds. The most common is **ReLU**:

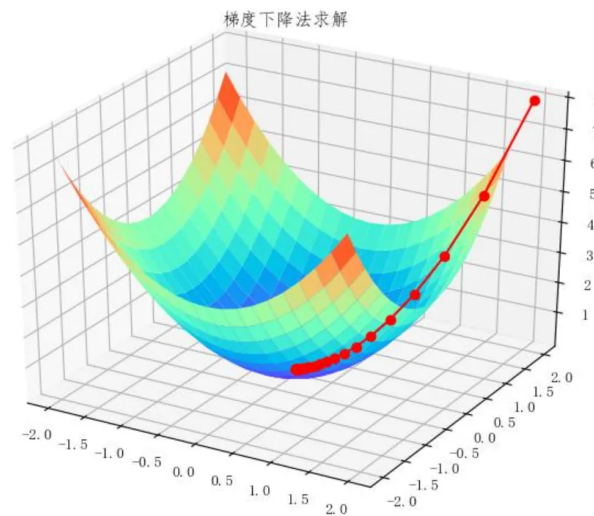
$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

It introduces non-linearity, enabling the network to model complex patterns.

Assume $X \in \mathbb{R}^{3 \times 2}$, $W \in \mathbb{R}^{2 \times 3}$:

$$\sigma(W_{2 \times 3} \cdot X_{3 \times 2}) = \sigma \left(\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix} \cdot \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \\ x_{3,1} & x_{3,2} \end{bmatrix} \right)$$

$$= \begin{bmatrix} \max(0, \sum_{k=1}^3 w_{1,k} x_{k,1}) & \max(0, \sum_{k=1}^3 w_{1,k} x_{k,2}) \\ \max(0, \sum_{k=1}^3 w_{2,k} x_{k,1}) & \max(0, \sum_{k=1}^3 w_{2,k} x_{k,2}) \end{bmatrix} = Y_{2 \times 2}$$



Fortunately, **gradient descent works effectively even in high-dimensional, non-linear spaces.**

Efficient Fine-Tuning Techniques

Pre-training vs. fine-tuning

From earlier chapters, you've learned that model training is essentially a search for the optimal set of parameters. The model you downloaded at the beginning — **qwen2.5-1.5b-instruct** — already contains a set of **pre-trained parameters**, learned from vast amounts of general data.

Fine-tuning builds on this foundation. Instead of training from scratch, you further adjust the model's parameters using task-specific data — for example, math word problems — so it performs better on your target use case.

But just how expensive would it be to train such a model from scratch? Let's estimate the time and hardware requirements for training a 1.5B-parameter model.

Hardware and memory requirements

To fine-tune large language models, you typically need high-end GPUs such as **T4, A10, V100, A100, or H100**:

- **T4, A10, V100**: More accessible and cost-effective in China.
- **A100/H100**: Higher performance but significantly more expensive.

Even a 1.5B-parameter model demands substantial memory:

- Each parameter in full precision (FP32) takes **4 bytes**.
- Total parameter memory:
 $1.5 \times 10^9 \times 4 / 2^{30} \approx 5.59 \text{ GB}$

However, during training, additional memory is needed for gradients, optimizer states, and activations. In practice, you need **7–8×** the parameter size:

$$5.59 \text{ GB} \times 8 \approx 45 \text{ GB}$$

This exceeds the VRAM capacity of most consumer GPUs — and even your current lab environment.

Estimating training time and cost

Suppose you rent out an **8xV100 GPU instance** on Alibaba Cloud:

- Total VRAM: 256 GB (sufficient for training)
- Cost: ~US\$ 35 per hour
- Processing speed: ~190 tokens/sec/GPU

Let’s assume your task requires high accuracy, so you prepare a dataset equivalent to **5,000 copies of the entire *Lord of the Rings* trilogy** (455,000 characters each), totaling about **2.28 billion tokens**.

$$\text{Time (days)} = \frac{2.28 \times 10^9 \times 8 \times 190 \times 3600 \times 24}{\approx 17 \text{ days}}$$

That’s 17 days! Now, factoring in the cost:

$$\text{Cost} = \left(\frac{2.28 \times 10^9 \times 8 \times 190 \times 3600}{\right) \times 35 \approx \$14,500$$

💡 For comparison:
Meta’s **LLaMA-65B** was pretrained on **1.4 trillion tokens**, using **2,048 A100 GPUs**, running for **21 days**. At \$3.93/hour per A100, the estimated cost was **\$4 million**.

This illustrates a key point:

The **scale of your training data** directly determines the **total compute required**, which drives both **time** and **cost**.

While better hardware reduces training time (and sometimes total cost), the most impactful levers are:

- **Data quality:** High-quality, relevant examples lead to faster convergence.
- **Efficient training methods:** Reducing parameter updates can drastically cut costs.

In practice, collecting large labeled datasets is expensive — especially for niche domains like medical diagnosis or legal text analysis.

To address this, modern AI uses a **two-stage approach**:

Stage	Goal	Data Type	Learning Method
Pretraining	Learn general language understanding	Massive unlabeled text (web, books, Wikipedia)	Self-supervised learning
Fine-tuning	Adapt to specific tasks	Small labeled dataset (e.g., math problems, sentiment labels)	Supervised learning

Key differences:

Feature	Pretraining	Fine-tuning
Goal	Learn general patterns and representations	Adapt to a specific downstream task
Data Size	Massive (terabytes of text)	Small (thousands to tens of thousands of samples)
Data Labeling	Unlabeled (self-supervised)	Labeled (supervised)
Training Method	Self-supervised (e.g., predict masked words)	Supervised (e.g., input-output pairs)
Parameter Updates	All parameters updated	All or partial parameters updated
Use Case	Build foundational models (e.g., Qwen, GPT, DeepSeek)	Customize models for specific applications

This two-stage strategy allows you to:

1. **Reuse powerful pretrained models** (saving millions in compute).
2. **Fine-tune with minimal labeled data** (saving time and annotation costs).

The path to fast, cost-effective model customization

You can build a powerful domain-specific model efficiently by following these steps:

1. **Start with a strong pretrained model** like Qwen2.5, DeepSeek, or Llama.
2. **Collect a small, high-quality labeled dataset** (e.g., 5,000 math word problems with solutions).
3. **Fine-tune the model** on this data — much faster and cheaper than training from scratch.

But even fine-tuning can be expensive if you update all parameters.

So: **Can we reduce GPU memory usage and training cost further?**

Yes — through **Efficient Fine-Tuning**.

Full fine-tuning vs. parameter-efficient fine-tuning (PEFT)

The main factor affecting memory and cost is the **number of parameters being updated**.

We can divide fine-tuning into two categories:

- **Full Fine-Tuning**
 - Updates **all parameters** of the model.
 - Achieves strong performance but requires high VRAM and compute.
 - Even with pretrained models, this remains costly for large models.
- **Parameter-Efficient Fine-Tuning (PEFT)**
 - Only updates a **small subset** of parameters.
 - Drastically reduces memory, compute, and cost.
 - Performance is close to full fine-tuning.

Popular PEFT methods include:

- **Adapter Tuning:** Inserts small neural modules into layers.
- **Prompt Tuning:** Learns soft prompts instead of modifying model weights.
- **LoRA (Low-Rank Adaptation):** Updates weight changes via low-rank matrices — often only **0.1%–1%** of original parameters.

 **LoRA has become the go-to method** for efficient fine-tuning under resource constraints.

In the next section, we'll dive into **how LoRA works** and why it enables high-performance fine-tuning with minimal overhead.

LoRA: Low-Rank Adaptation

LoRA (Low-Rank Adaptation) has become the most widely adopted method for efficient large model fine-tuning. It works across different model architectures and drastically reduces the number of trainable parameters — without sacrificing performance.

How LoRA Works

Instead of updating all the original model weights during fine-tuning, LoRA introduces a smart mathematical trick: it **decomposes weight changes into small, low-rank matrices**.

The core idea is captured in this equation:

$$W_{\text{new}} = W_{\text{frozen}} + \Delta W = W_{\text{frozen}} + A \cdot B$$

Where:

- W_{frozen} : Original pretrained weight matrix (not updated).
- $\Delta W = A \cdot B$: Learnable low-rank update.
- $A \in \mathbb{R}^{d \times r}$, $B \in \mathbb{R}^{r \times d}$: Two small matrices to train.
- $r \ll d$: The "rank" is much smaller than the full dimension.

This means only A and B are trained — not the entire W . As a result, VRAM usage and compute costs drop significantly.

Understanding low-rank decomposition

To understand why this works, let's look at a simple example.

Suppose we have a neural network layer:

- Input $X \in \mathbb{R}^4$

- Output $Y \in \mathbb{R}^5$
- Weight matrix $W \in \mathbb{R}^{5 \times 4} \rightarrow$ contains 20 parameters

The transformation is: $Y = \sigma(W \cdot X)$

Now, during fine-tuning, we don't retrain that W . Instead, we learn a small change ΔW , which we assume carries only **limited new information** — i.e., it has **low intrinsic rank**.

For example, consider this 5×4 matrix:

$$\Delta W = \begin{bmatrix} 1 & 0 & 2 & -1 \\ 2 & 0 & 4 & -2 \\ 3 & 0 & 6 & -3 \\ 4 & 0 & 8 & -4 \\ 5 & 0 & 10 & -5 \end{bmatrix}$$

Even though it looks big, each row is just a multiple of the first. So its **effective information** can be captured by one vector. This matrix has **rank 1**.

We can decompose it into two smaller matrices:

$$\Delta W = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \begin{bmatrix} 1 & 0 & 2 & -1 \end{bmatrix} = A \cdot B$$

Now, instead of training 20 values, we only train $5 + 4 = 9$ values — a huge saving.

✓

In real models, researchers find that **fine-tuning updates are inherently low-rank** — most knowledge is already in the pretrained model; only minor adjustments are needed.

Unlocking the efficiency behind LoRA

Let's apply this to a real-world scenario using the `qwen2.5-1.5b-instruct` model.

Assume:

- Layer dimension $d = 1024$
- LoRA rank $r = 8$ (much smaller: $r \ll d$)

Method	Trainable Parameters	Count	Savings
Full Fine-Tuning	Entire $W_{d \times d}$	$1024 \times 1024 = 1,048,576$	0%
LoRA Fine-Tuning	$A_{d \times r} + B_{r \times d}$	$1024 \times 8 + 8 \times 1024 = 16,384$	98.4%

✓ By training only **1.6% of the parameters**, LoRA achieves performance close to full fine-tuning — while reducing memory, cost, and time.

Inference: merging for speed

After training, LoRA weights can be **merged** back into the original model:

$$W_{\text{merged}} = W_{\text{original}} + A \cdot B$$

This merging can happen:

- **Before deployment:** Create a standalone optimized model.
- **Dynamically:** Apply LoRA on-the-fly for multi-task scenarios.

Once merged, there's **no inference overhead** — the model runs just like a fully fine-tuned version.



Choosing the right rank

The rank r controls the capacity of the adapter:

- **Small r** \rightarrow fewer parameters, faster training, less overfitting.
- **Large r** \rightarrow more expressive, but higher memory use and risk of overfitting.

Practical guidelines

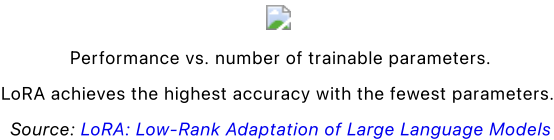
Training Data Size	Recommended Rank r	Rationale
--------------------	----------------------	-----------

Training Data Size	Recommended Rank r	Rationale
Small (1k–10k samples)	$r \leq 16$	Prevents overfitting; avoids memorizing data instead of learning patterns
Medium (10k–100k)	$r = 16 \text{ to } 32$	Balances capacity and efficiency
Large (>100k samples)	$r \geq 32$	Can capture more complex task-specific patterns

💡 Start with $r = 8$ or $r = 16$, then experiment based on validation performance.

LoRA performance

The original LoRA paper evaluated several parameter-efficient methods across two benchmark datasets. The results show a clear winner:



Key insights

- **More trainable parameters \neq better performance.** Some methods plateau quickly.
- **LoRA scales well:** It continues to improve as more data or higher rank is used.
- **Best cost-performance trade-off:** LoRA delivers strong results with minimal overhead.

✅ In practice, LoRA is the go-to method for fine-tuning large language models under resource constraints — whether you're working on math reasoning, code generation, or domain-specific Q&A.

By combining **pretrained knowledge** with **targeted, low-cost adaptation**, LoRA enables fast, affordable, and effective model customization — making advanced AI accessible even with limited hardware.

From Paper to Action: Fine-tuning our model

Model training states and evaluation metrics

Training a machine learning model is remarkably similar to how a student prepares for an exam. Just as students study with practice problems, take mock tests, and finally sit for the real exam, models go through a structured learning process using three distinct datasets — each serving a specific purpose.

These datasets help generate key performance indicators that reveal the model's current state and guide the training process.

The Three "Exams" for Your Model

Dataset	Purpose	Analogy	Key Metric
Training Set	Used to teach the model by adjusting its parameters	Course workbook with answers	Training Loss
Validation Set	Used to evaluate progress during training	Mock exam	Validation Loss
Test Set	Used only once — after training — to assess final performance	Final exam (taken only once)	Accuracy / Test Score


Let's dive into each:

1. Training Set → Training Loss
 - This is the "workbook" the model studies repeatedly.
 - During training, the model learns from input-output pairs and computes its **training loss** — a measure of how far its predictions are from the correct answers.
 - A decreasing training loss means the model is improving on the data it has seen.

🔄 The model uses this feedback (via gradient descent) to update its internal parameters.

2. Validation Set → Validation Loss

- Think of this as a **mock exam** — unseen during training.
- Periodically, the model is tested on this data to estimate how well it generalizes to new examples.
- The resulting **validation loss** helps you detect overfitting or underfitting early.




 You should never use the validation set to update the model — only to monitor progress.

3. Test Set → Final Performance
- This is the **real exam**, taken only once at the end of training.
 - It provides an unbiased evaluation of the model’s overall ability.
 - Once evaluated, you shouldn’t go back and retrain based on test results — otherwise, it’s no longer a fair assessment.

 Best practice: Keep the test set completely untouched until the very end.

Interpreting Training Dynamics: Three Key States

By comparing **training loss** and **validation loss**, you can identify the model’s learning status and decide what to do next.

Training Loss	Evaluation Loss	State	Interpretation	What to Do
Not decreasing or increasing	Not improving	 Training Failed	Model isn’t learning — like a student who doesn’t understand the material.	Check learning rate, data quality, or model architecture.
Decreasing	Decreasing	 Underfitting	Model is still learning — both on workbook and mock exams — but not yet good enough.	Let training continue; consider increasing training epochs.
Decreasing	Increasing	 Overfitting	Model is memorizing the workbook instead of understanding concepts — fails on new problems.	Stop training early, apply regularization, or add more diverse training data.

Understanding Overfitting: The "Memorization Trap"


When a model **overfits**, it becomes overly specialized to the training data — like a student who memorizes answers without understanding the underlying logic.

For example:

- It may solve every problem in the "practice workbook" perfectly.
- But when faced with a slightly different question on the "mock exam", it fails.

Solutions to prevent overfitting:

- Use **early stopping**: Halt training when validation loss stops improving.
- Increase **data diversity**: Add more varied examples so the model can’t just memorize.
- Apply **regularization techniques**: Such as dropout or weight decay.
- Use **larger batch sizes** or **data augmentation** to reduce noise in learning.

 Pro Tip: A small gap between training and validation loss is normal. But if validation loss starts rising while training loss keeps falling — it’s time to stop.

Summary

To ensure successful fine-tuning:

1. **Monitor both training and validation loss** throughout the process.
2. **Use the validation set as your guide** — not the training set.
3. **Reserve the test set for the final evaluation only.**
4. **React appropriately** to the three key states: failure, underfitting, and overfitting.

Just like a good teacher watches a student’s progress across practice, mock exams, and finals, you must guide your model with the right metrics at the right time.

This disciplined approach ensures your model doesn’t just "memorize answers" — but truly **learns to reason**.

Fine-tuning In Action

The baseline test

Before we start fine-tuning our model, let's give it a quick math test:

```
import json
from IPython.display import Markdown

sum, score = 0, 0
for line in open("./resources/benchmark_exam.jsonl"):

    math_question = json.loads(line)
    query = math_question["messages"][1]["content"]

    response, _ = inference(model, template, query)

    ans = math_question["messages"][2]["content"]
    pos = ans.find("ans")
    end_pos = ans[pos:].find('}}')
    ans = ans[pos - 2: end_pos + pos + 2]

    print("=" * 50)
    print(query.split("#Math Problem#\n")[1])
    print("The answer we're looking for is: " + ans + "\n")
    print("-" * 25 + "Model response" + "-" * 25)
    display(Latex(response))
    print("-" * 23 + "Model response end" + "-" * 23)

    if ans in response or ans[6:-2] in response:
        score += 1
        print("Model is correct!\n")
    else: print("Model is wrong :(\n")
    sum += 1

display(Markdown("Benchmark score: *" + str(int(100*score/sum)) + "*""))
```

The base model frequently gives up mid-reasoning during problem-solving, failing to reach correct answers. This behavior not only confirms that the task exceeds its current reasoning capacity, but also reveals a fundamental limitation of prompt engineering: **no matter how well-crafted the prompt, it cannot compensate for a lack of underlying capability.**

In other words, prompt engineering is like giving a student better instructions on how to take an exam — but if the student hasn't learned the material, clearer instructions won't lead to better results.

To truly improve performance on tasks like mathematical reasoning, **model fine-tuning is essential**. It enhances the model's internal reasoning ability by training it on relevant examples, effectively "teaching" it how to think through problems step by step.

Only through fine-tuning can the model develop the logical and computational reasoning skills needed to reliably solve complex tasks — going beyond pattern matching or shallow recall to perform genuine problem-solving.

Fine-tuning with **ms-swift**

In this section, we use the **MS-Swift** (<https://github.com/modelscope/ms-swift/tree/main>) (ModelScope Scalable lightweight Infrastructure for Fine-Tuning) framework — an open-source training framework developed by Alibaba's ModelScope (MoE) team. MS-Swift supports training, inference, evaluation, and deployment for over **350 large language models (LLMs)** and **90+ multimodal large models (MLLMs)**, including pretraining, fine-tuning, and alignment tasks.

One of the key advantages of MS-Swift is its **user-friendly design**. During training, the framework automatically:

- Computes the **evaluation loss** on the validation set at regular intervals.
- Saves a **checkpoint** of the model at each evaluation step.
- Retains the **best model checkpoint** — the one with the lowest validation loss — after training completes.



ms-swift outputs both last and best checkpoints when applicable

Experiment Setup

In the following experiments, we will focus on tuning three key hyperparameters:

- **Learning Rate** (`learning_rate`): Controls the step size during optimization.
- **LoRA Rank** (`lora_rank`): Determines the capacity of the low-rank adaptation.
- **Number of Training Epochs** (`num_train_epochs`): How many times the model sees the full dataset.

We will also experiment with different datasets to demonstrate how data quality and size impact fine-tuning performance.

Other parameters (e.g., `batch_size`, `max_length`) are adjusted primarily to **reduce training time** and fit within GPU memory constraints. You don't need to tune them deeply for this exercise.

Experiment 1 (≈1 minute)

For your first run, we recommend using the following configuration. This baseline uses a small dataset of **100 math problems with solutions**, generated by [Qwen](#), to enable quick iteration and comparison in later experiments.



Parameters	
Parameter	Value
Learning Rate (<code>learning_rate</code>)	0.1
LoRA Rank (<code>lora_rank</code>)	4
Training Epochs (<code>num_train_epochs</code>)	1
Dataset Path	"/resources/training_dataset_100.jsonl"
Constraints	
• <code>batch_size</code> ≤ 16	(Limited by VRAM)
• <code>max_length</code> ≤ 512	(Max token length per sample)
• <code>lora_rank</code> ≤ 64	(Higher ranks use more memory)
• <code>eval_steps</code> ≤ 20	(Frequent evaluation for visibility)

Start the experiment!

The MS-Swift framework uses **LoRA fine-tuning by default**, so you don't need to explicitly specify the method.

Additionally, the framework includes **automatic learning rate scheduling** — it gradually reduces the learning rate during training to prevent overshooting the optimal solution. This helps stabilize convergence, especially with relatively high initial learning rates like 0.1.

```
%env TF_ENABLE_ONEDNN_OPTS=0
%env CUDA_VISIBLE_DEVICES=0
%env LOG_LEVEL=INFO
!swift sft \
--learning_rate '0.1' \
--lora_rank 4 \
--num_train_epochs 1 \
--dataset './resources/training_dataset_100.jsonl' \
--per_device_train_batch_size 8 \
--eval_steps 1 \
--max_length 512 \
--model_type 'qwen2' \
--model './model/qwen2_5-1_5b-instruct'
```

Training loss	Evaluation loss
	
Observations	Both training and evaluation losses both trends upwards
Training	Failed

Observations	Both training and evaluation losses both trends upwards
Analysis	<div>The learning rate is too high, causing the solver to overshoot the minima, failing convergence.</div>
Adjustment strategy :	Reduce the learning rate significantly to 0.00005.

Experiment 2 (≈5 minutes)

Since the first experiment overshoot the convergence value, we'll drastically lower it this time around:

Parameters

Parameter	Original value	New value
Learning Rate (learning_rate)	0.1	0.00005
LoRA Rank (lora_rank)	4	
Training Epochs (num_train_epochs)	50	
Dataset Path	"./resources/training_dataset_100.jsonl"	
Batch Size (batch_size)	8	
Evaluation Steps (eval_steps)	1	

```
%env TF_ENABLE_ONEDNN_OPTS=0
%env CUDA_VISIBLE_DEVICES=0
%env LOG_LEVEL=INFO
!swift sft \
--learning_rate '0.00005' \
--lora_rank 4 \
--num_train_epochs 1 \
--dataset './resources/training_dataset_100.jsonl' \
--per_device_train_batch_size 8 \
--eval_steps 1 \
--max_length 512 \
--model_type 'qwen2' \
--model './model/qwen2_5-1_5b-instruct'
```

Training loss	Evaluation loss
Observations	Both training and evaluation losses trend downwards
Training	Underfitting
Analysis	Underfitting is a very common issue during training. It indicates that the model has not yet learned the underlying patterns sufficiently. With the current parameters, simply training for more epochs may lead to successful convergence. Alternatively, hyperparameter adjustments can accelerate the training process.
Adjustment strategy :	1. Train the model longer: increase the number of epochs to 50. 2. Increase the batch_size to the maximum value of 16 to speed up training.

Experiment 3 (≈10 minutes)



Since the first experiment overshoot the convergence value, we'll drastically lower it this time around:

Parameters

Parameter	Original value	New value
Learning Rate (learning_rate)	0.00005	
LoRA Rank (lora_rank)	4	
Training Epochs (num_train_epochs)	1	50

Parameter	Original value	New value
Dataset Path	"/resources/training_dataset_100.jsonl"	
Batch Size (batch_size)	8	16
Evaluation Steps (eval_steps)	1	20

```
%env TF_ENABLE_ONEDNN_OPTS=0
%env CUDA_VISIBLE_DEVICES=0
%env LOG_LEVEL=INFO
!swift sft \
--learning_rate '0.00005' \
--lora_rank 4 \
--num_train_epochs 50 \
--dataset './resources/training_dataset_100.jsonl' \
--per_device_train_batch_size 16 \
--eval_steps 20 \
--max_length 512 \
--model_type 'qwen2' \
--model './model/qwen2_5-1_5b-instruct'
```

Training loss	Evaluation loss
	
Observations	Training loss trends downwards, and evaluation loss trends downwards at first but then shoots back up
Training	Overfitting
Analysis	Overfitting is a very common issue during training. It indicates that the model is "memorizing the answers" rather than learning the underlying patterns in the dataset. We can mitigate this by reducing the number of training epochs or increasing the amount of training data, helping the model generalize instead of simply memorizing.
Adjustment strategy :	1. Reduce the number of training epochs to 3. 2. Increase the training sample to 500 sets. Dataset location: ./resources/training_dataset_500.jsonl 3. With the increased data volume, increase the LoRA rank to 16.



Experiment 4 (~5 minutes)

Since we're overfitting - we're going to do three things together:

- Reduce training epochs
- Increase training samples
- Increase LoRA rank

Parameters		
Parameter	Original value	New value
Learning Rate (learning_rate)	0.00005	
LoRA Rank (lora_rank)	4	8
Training Epochs (num_train_epochs)	50	3
Dataset Path	"/resources/training_dataset_100.jsonl"	"/resources/training_dataset_500.jsonl"
Batch Size (batch_size)	16	
Evaluation Steps (eval_steps)	20	

```
%env TF_ENABLE_ONEDNN_OPTS=0
%env CUDA_VISIBLE_DEVICES=0
%env LOG_LEVEL=INFO
!swift sft \
--learning_rate '0.00005' \
--lora_rank 8 \
--num_train_epochs 3 \
--dataset './resources/training_dataset_500.jsonl' \
--per_device_train_batch_size 16 \
--eval_steps 20 \
--max_length 512 \
--model_type 'qwen2' \
--model './model/qwen2_5-1_5b-instruct'
```

Training loss	Evaluation loss
	
Observations	Both training and evaluation losses are plateauing at the bottom
Training	Underfitting
Analysis	We might be close to successful convergence!
Adjustment strategy :	Increase the number of training epochs to 50.



Experiment 5 (≈20 minutes)

We seemed to have reached convergence, but we need to be sure. So we're going to increase the training to see what happens:

Parameters		
Parameter	Original value	New value
Learning Rate (learning_rate)	0.00005	
LoRA Rank (lora_rank)	8	
Training Epochs (num_train_epochs)	3	50
Dataset Path	"./resources/training_dataset_100.jsonl"	"./resources/training_dataset_1000.jsonl"
Batch Size (batch_size)	16	
Evaluation Steps (eval_steps)	20	

```
%env TF_ENABLE_ONEDNN_OPTS=0
%env CUDA_VISIBLE_DEVICES=0
%env LOG_LEVEL=INFO
!swift sft \
--learning_rate '0.00005' \
--lora_rank 8 \
--num_train_epochs 50 \
--dataset './resources/training_dataset_500.jsonl' \
--per_device_train_batch_size 16 \
--eval_steps 20 \
--max_length 512 \
--model_type 'qwen2' \
--model './model/qwen2_5-1_5b-instruct'
```

Training loss	Evaluation loss
---------------	-----------------

Training loss	Evaluation loss
	
Observations Both training and evaluation losses show no signs of further downward trends	
Training	Success!

Since the graphs obtained from this experiment are closely similar to that in **Experiment 4**, it supports the theory that we've reached convergence!

Retaking the baseline test

When we've successfully fine-tuned our LLM in `ms-swift`, it generates a few checkpoint files.

We'll usually take the latest checkpoint file as our final LoRA. You can find the checkpoint files in the `/output/` folder:



Finding the latest LoRA checkpoint

Let's validate our newly tuned LLM:

```
from swift.tuners import Swift

# Replace ckpt_dir with the directory pointing to your checkpoint
--ckpt_dir 'output/qwen2_5-1_5b-instruct/<replace_with_checkpoint_directory>' \ # ← Replace with
checkpoint directory

# Load the model
ft_model = Swift.from_pretrained(model, ckpt_dir, inference_mode=True)
```

```
import json
from IPython.display import Markdown

sum, score = 0, 0
for line in open("./resources/benchmark_exam.jsonl"):

    math_question = json.loads(line)
    query = math_question["messages"][1]["content"]

    response, _ = inference(model, template, query)

    ans = math_question["messages"][2]["content"]
    pos = ans.find("ans")
    end_pos = ans[pos:].find('}}')
    ans = ans[pos - 2: end_pos + pos + 2]

    print("=" * 50)
    print(query.split("#Math Problem#\n")[1])
    print("The answer we're looking for is: " + ans + "\n")
    print("-" * 25 + "Model response" + "-" * 25)
    display(Latex(response))
    print("-" * 23 + "Model response end" + "-" * 23)

    if ans in response or ans[6:-2] in response:
        score += 1
        print("Model is correct!\n")
    else: print("Model is wrong :(\n")
    sum += 1

display(Markdown("Benchmark score: *" + str(int(100*score/sum)) + "*""))
```

Model merging

Now that we've fine-tuned our model, we can start using it. There are two ways to do this:

- **Load the LoRA when calling the base model**

The LoRA matrices generated during fine-tuning are typically very small — often around 20MB in size. This compact footprint makes them ideal for incremental updates and efficient distribution, a key advantage widely leveraged in real-world engineering workflows. However, it's important to note that you must ensure compatibility between the adapter and the original base model. Specifically, the same base model used during training must be specified at inference time to correctly apply the fine-tuned parameters. We demonstrated this approach in the previous section by loading the adapter weights using the `ckpt_dir` parameter.

- **Merging the LoRA weights with base model**

Alternatively, you can permanently integrate the fine-tuned low-rank parameters into the original base model, resulting in a single, standalone model with updated weights. This fused model no longer depends on external adapter files and can be deployed independently, just like any fully trained model.

This section provides a brief look at the second method: model merging. In short, the weight matrix in the LoRA is merged with that in the base model.

The `swift export` makes merging models a breeze. Simply provide the path to your fine-tuned checkpoint—preferably the `best_model_checkpoint` identified during training—and the tool will generate a fully merged model ready for deployment.

Pro tip:

If `ms-swift` outputs a `best_model_checkpoint`, use that for optimal experience.

```
%env LOG_LEVEL=INFO
!swift export \
--ckpt_dir 'output/qwen2_5-1_5b-instruct/<replace_with_checkpoint_directory>' \ # ← Replace with
checkpoint directory
--merge_lora true
```



Merged model save location

You can see from the logs that the model has been merged successfully and the merged model has been saved to a new folder:

```
"/output/qwen2_5-1_5b-instruct/v##-yyymmdd-hhmmss/checkpoint-####-merged"
```

What's Next?

Quiz yourself!

► **1. What is the primary benefit of using LoRA (Low-Rank Adaptation) for fine-tuning large language models?**

- A) It increases the model's parameter count significantly
- B) It allows full retraining of all model weights with lower memory usage
- C) It reduces trainable parameters by up to 98%, cutting memory and cost while maintaining performance
- D) It eliminates the need for any training data

View answer →

✅ **Correct answer:** C) It reduces trainable parameters by up to 98%, cutting memory and cost while maintaining performance

📖 **Explanation:**

LoRA freezes the original model weights and introduces low-rank matrices (A and B) to represent weight updates. This means only a small fraction of parameters are trained—dramatically reducing VRAM usage and training costs—while achieving performance close to full fine-tuning.

► **2. What does it mean if training loss is decreasing but validation loss is increasing during fine-tuning?**

- A) The model is underfitting
- B) The model is generalizing well
- C) The model is overfitting
- D) The learning rate is too low

View answer →

✅ **Correct answer:** C) The model is overfitting

📖 **Explanation:**

This pattern indicates that the model is memorizing the training data rather than learning generalizable patterns. It performs better on seen data (lower training loss) but worse on unseen validation data (higher validation loss). This is the "memorization trap" — like a student who memorizes answers but fails on new questions.

► **3. Which of the following best describes the role of the validation set during fine-tuning?**

- A) Used to train the model parameters
- B) Used to evaluate final performance after training
- C) Used to monitor progress and prevent overfitting
- D) Used to increase batch size

View answer →

✅ **Correct answer:** C) Used to monitor progress and prevent overfitting

📖 **Explanation:**

The validation set acts like a "mock exam" — it provides an unbiased evaluation of model performance on unseen data during training. By tracking validation loss, you can detect overfitting early and apply techniques like early stopping.

Takeaways

- **Why fine-tuning matters**
 - **prompt engineering has limits** — no matter how well-crafted your prompts are, they cannot compensate for a lack of internal reasoning ability.
 - **Fine-tuning teaches the model how to think**, not just what to say. It improves logical, mathematical, and domain-specific reasoning.
 - **The hybrid strategy wins:** Combine fine-tuning (for reasoning) with tools/plugins (for accuracy) — e.g., teach the model to solve problems step-by-step, then use a calculator for arithmetic.
- **Pre-training vs. Fine-tuning**
 - **Pre-training:** Learns general language understanding from massive unlabeled data (e.g., books, web text).
 - **Fine-tuning:** Adapts a pretrained model to a specific task using a small labeled dataset (e.g., math problems, customer support queries).
 - You don't need to train from scratch — reuse powerful open models like Qwen, Llama, or DeepSeek.
- **Parameter-Efficient Fine-Tuning (PEFT)**
 - Full fine-tuning updates all parameters — **high cost, high VRAM**.
 - PEFT methods (like LoRA) update only a small subset — **low cost, minimal overhead**.
 - **LoRA is the gold standard:**
 - Freezes original weights.
 - Adds trainable low-rank matrices $\Delta W = A \cdot B$.
 - Reduces trainable parameters from millions to thousands.
 - Achieves near-full performance with ~1–2% of the cost.

