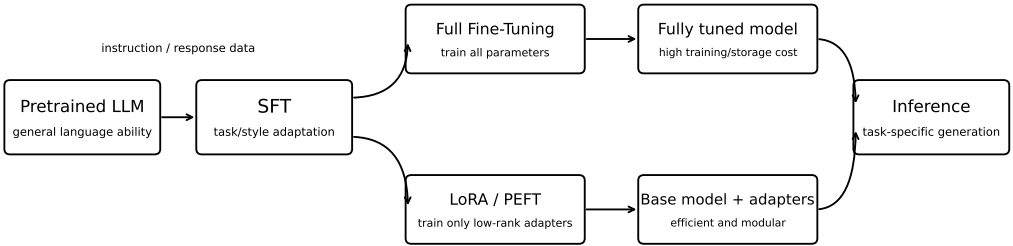
Selected Topics
Introduction to LoRA

Weiwen Wang

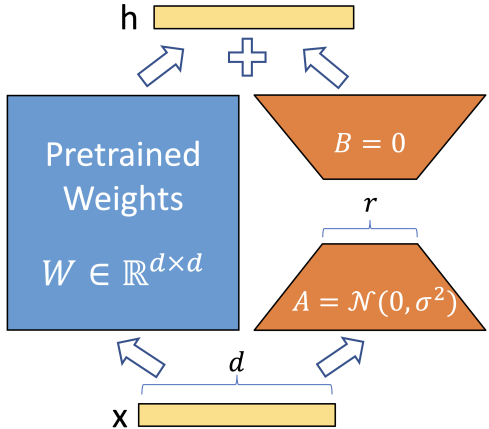
2026 年 4 月 16 日

Overview of SFT and LoRA



- **SFT: Supervised-Fine Tuning**
- **LoRA: Low-Rank Adaptation**

A Schematic of LoRA



Low-rank approximation of parameters[Hu et al., 2022].

Adaptation Before LoRA

- Adding external modules.
- Adapting some parameters.

Adaptation Before LoRA

- Adding external modules.
- Adapting some parameters.

Disadvantages

- Increase inference latency
- Reduce model's quality compared to fine-tuning.

Hypothesis LoRA

The changes in parameters of a pretrained model during fine-tuning have low-rank structure.

$$W_{FT} = W_{PE} + \Delta, \quad \Delta = BA.$$

- W_{PE}: weights in a pretrained model
- W_{FT}: weights after fine-tuning
- Δ: changes in weights.

Advantages of LoRA

- **Storage and task-switching efficiency**: a shared pretrained model with modularized task specific weights.
- **Lightweight fine-tuning:** only small low-rank matrices are optimized.

General Setting

For a task say natural language to SQL and given a training dataset of context-target pairs: $\mathcal{Z} = \{(x_i, y_i)\}_{i=1}^{N}$ where both x_i and y_i are sequences of tokens.

- **Full fine-tuning**: The model initialized by pre-trained weights Φ_0 is updated to $\Phi_0 + \Delta\Phi$ by maximize the conditional language modeling objective

$$\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(P_{\Phi}(y_t|x, y_{<t})\right)$$

General Setting

For a task say natural language to SQL and given a training dataset of context-target pairs: $\mathcal{Z} = \{(x_i, y_i)\}_{i=1}^{N}$ where both x_i and y_i are sequences of tokens.

- **Full fine-tuning**: The model initialized by pre-trained weights Φ_0 is updated to $\Phi_0 + \Delta\Phi$ by maximize the conditional language modeling objective

$$\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(P_\Phi(y_t|x, y_{<t})\right)$$

  - **For each downstream task, a different set of parameters $\Delta\Phi$ with the same size of Φ_0 should be learned. (GPT-3 $|\Phi_0| \approx 175B$)**

General Setting

- **Low-rank adaptation**: Suppose the task-specific parameter increment $\Delta\Phi = \Delta\Phi(\Theta)$ is encoded by a much smaller-size set of parameters Θ with $|\Theta| \ll |\Phi_0|$. Finding Φ becomes optimizing over Θ:

$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(P_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right)$$

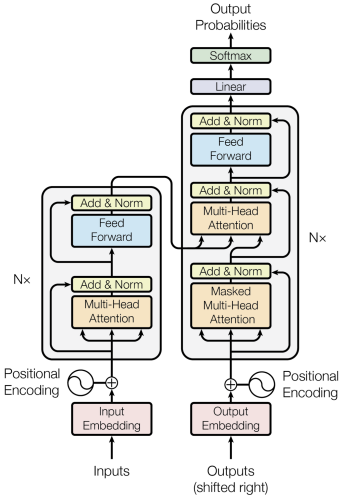
General Setting

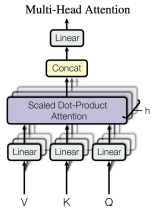
- **Low-rank adaptation**: Suppose the task-specific parameter increment $\Delta\Phi = \Delta\Phi(\Theta)$ is encoded by a much smaller-size set of parameters Θ with $|\Theta| \ll |\Phi_0|$. Finding Φ becomes optimizing over Θ:

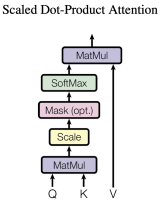
$$\max_{\Theta} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log\left(P_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right)$$

  - **The size of parameters should be updated can be significantly reduced, from $|\Phi_0|$ to $|\Theta|$. (claimed 0.01 % of $|\Phi_0|$ in [Hu et al., 2022])**

Transformer[Vaswani et al., 2017]





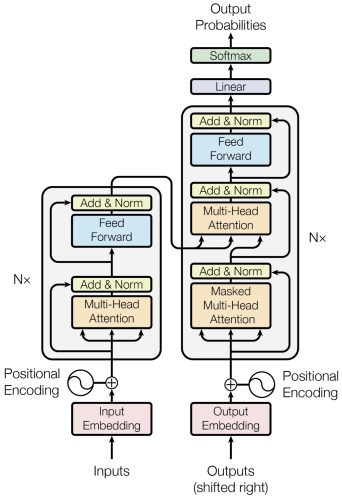


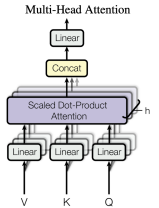
For a pretrained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, let its updated version after fine-tuning as the decomposition

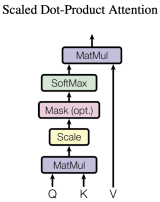
$$W + \Delta W = W_0 + BA,$$

where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$.

Transformer[Vaswani et al., 2017]







For a pretrained weight matrix $W_0 \in \mathbb{R}^{d\times k}$, let its updated version after fine-tuning as the decomposition

$$W + \Delta W = W_0 + BA,$$

where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$.

For the layer input x, its forward pass becomes

$$h = W_0 x + \Delta W x = W_0 x + BAx,$$

where A is initialized by Gaussian variables and B is initialized by zero.

Low-Rank Adaptation

Task Switching

$$\{(W_0 + BA) - BA\} + B'A'$$

- W_0: weight matrix of a pretrained model
- BA: incremental change of weight matrix of a task
- $B'A'$: incremental change of weight matrix of another task

No additional inference latency is introduced.

A Running Example

Suppose we have a four-layer MLP as

$$x \to L_1(W_1) \to L_2(W_2) \to L_3(W_3) \to L_4(W_4) \to \ell,$$

i.e. $W_4\{W_3\{W_2(W_1 x)\}\} \to \ell$, where we have omitted biases and activations for simplicity.

A Running Example

Suppose we have a four-layer MLP as

$$x \to L_1(W_1) \to L_2(W_2) \to L_3(W_3) \to L_4(W_4) \to \ell,$$

i.e. $W_4\{W_3\{W_2(W_1 x)\}\} \to \ell$, where we have omitted biases and activations for simplicity.

In this running example, LoRA is applied to L_1 and L_3. Then the output of L_1 and L_3 can be written as

$$h_1 = (W_1 + B_1 A_1)x \quad \text{and} \quad h_3 = (W_3 + B_3 A_3)h_2$$

respectively, where $A_\cdot$ and $B_\cdot$ are learnable parameters.

A Running Example(con't)

To update $A_{\cdot}$ and $B_{\cdot}$, we should compute $\frac{\partial \ell}{\partial A_{\cdot}}$ and $\frac{\partial \ell}{\partial B_{\cdot}}$.
By the chaining rule

$$\frac{\partial \ell}{\partial A_3} = \left(\frac{\partial h_3}{\partial A_3}\right)^T W_4^T \frac{\partial \ell}{\partial h_4} \quad \frac{\partial \ell}{\partial B_3} = \left(\frac{\partial h_3}{\partial B_3}\right)^T W_4^T \frac{\partial \ell}{\partial h_4}$$

$$\frac{\partial \ell}{\partial A_1} = \left(\frac{\partial h_1}{\partial A_1}\right)^T W_2^T (W_3 + B_3 A_3)^T W_4^T \frac{\partial \ell}{\partial h_4}$$

$$\frac{\partial \ell}{\partial B_1} = \left(\frac{\partial h_1}{\partial B_1}\right)^T W_2^T (W_3 + B_3 A_3)^T W_4^T \frac{\partial \ell}{\partial h_4}$$

A Running Example(con't)

To update $A_{\cdot}$ and $B_{\cdot}$, we should compute $\frac{\partial \ell}{\partial A_{\cdot}}$ and $\frac{\partial \ell}{\partial B_{\cdot}}$.
By the chaining rule

$$\frac{\partial \ell}{\partial A_3} = \left(\frac{\partial h_3}{\partial A_3}\right)^T W_4^T \frac{\partial \ell}{\partial h_4} \quad \frac{\partial \ell}{\partial B_3} = \left(\frac{\partial h_3}{\partial B_3}\right)^T W_4^T \frac{\partial \ell}{\partial h_4}$$

$$\frac{\partial \ell}{\partial A_1} = \left(\frac{\partial h_1}{\partial A_1}\right)^T W_2^T (W_3 + B_3 A_3)^T W_4^T \frac{\partial \ell}{\partial h_4}$$

$$\frac{\partial \ell}{\partial B_1} = \left(\frac{\partial h_1}{\partial B_1}\right)^T W_2^T (W_3 + B_3 A_3)^T W_4^T \frac{\partial \ell}{\partial h_4}$$

- **$\frac{\partial h_3}{\partial W_3}$ is replaced by $\frac{\partial h_3}{\partial A_3}$ and $\frac{\partial h_3}{\partial B_3}$.**
- **$\frac{\partial h_1}{\partial W_1}$ is replaced by $\frac{\partial h_1}{\partial A_1}$ and $\frac{\partial h_1}{\partial B_1}$**
- **The size of gradient is reduced from $d \times d$ to $d \times r + r \times d$ with additional overhead of matrix sum.**

Application of LoRA to Transformer

The weight matrices (i.e. W_q, W_k, W_v) in $d_{model} \times d_{model}$ are adapted by LoRA in [Hu et al., 2022].

Benefits

- VRAM(Video RAM) is reduced by 2/3 if $r \ll d_{model}$.
- On GPT-3 175B, the VRAM consumption during training is reduced from 1.2TB to 350GB.
- It costs less in task switching.
- It speeds up training compared to full fine-tuning as it does not need to calculate gradients of majority of the parameters.

Experiments

Baselines

- **Fine-Tuning(FT)**: The model is initialized to the pretrained weights and biases, and all model parameters take gradient updates. Or the gradient updates are conducted in only some of the layers while the rest is frozen.
- **BitFit**: Only biases are updated by the gradients and the rest is frozen.
- **Prefix-Embedding Tuning (PreEmbed)**: It inserts special tokens that have trainable word embeddings among the input tokens.
- **Prefix-Layer Tuning(PreLayer)**: It learns the activations of special tokens after every Transformer layer.
- **Adapter Tuning**: It inserts adapter layers between modules of Transformer.

Model&Method	# Trainable Parameters	WikiSQL Acc. (%)	MNLI-m Acc. (%)	SAMSum R1/R2/RL
GPT-3 (FT)	175,255.8M	**73.8**	89.5	52.0/28.0/44.5
GPT-3 (BitFit)	14.2M	71.3	91.0	51.3/27.4/43.5
GPT-3 (PreEmbed)	3.2M	63.1	88.6	48.3/24.2/40.5
GPT-3 (PreLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 ($\text{Adapter}^{\text{H}}$)	7.1M	71.9	89.8	53.0/28.9/44.8
GPT-3 ($\text{Adapter}^{\text{H}}$)	40.1M	73.2	**91.5**	53.2/29.0/45.1
GPT-3 (LoRA)	4.7M	73.4	**91.7**	**53.8/29.8/45.9**
GPT-3 (LoRA)	37.7M	**74.0**	**91.6**	53.4/29.2/45.1

Table 4: Performance of different adaptation methods on GPT-3 175B. We report the logical form validation accuracy on WikiSQL, validation accuracy on MultiNLI-matched, and Rouge-1/2/L on SAMSum. LoRA performs better than prior approaches, including full fine-tuning. The results on WikiSQL have a fluctuation around $\pm 0.5\%$, MNLI-m around $\pm 0.1\%$, and SAMSum around $\pm 0.2/\pm 0.2/\pm 0.1$ for the three metrics.

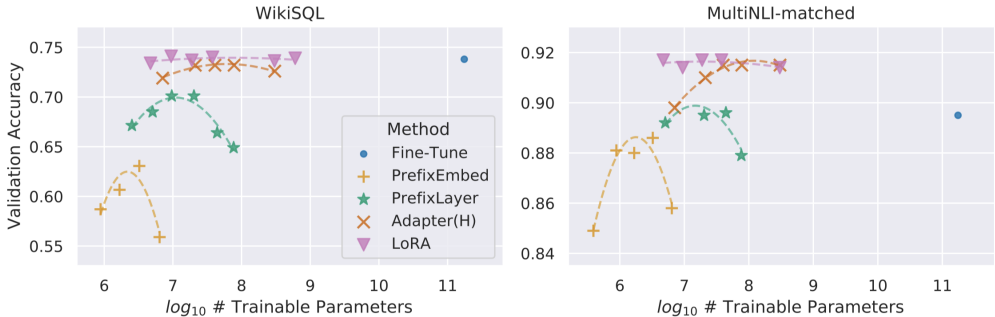


Figure 2: GPT-3 175B validation accuracy vs. number of trainable parameters of several adaptation methods on WikiSQL and MNLI-matched. LoRA exhibits better scalability and task performance. See Section F.2 for more details on the plotted data points.

- **Not all methods benefit from increasing trainable parameters.**

Understanding The Low-Rank Updates

- **Which weight matrices in transformer should we apply LoRA to?**
- **What is the optimal rank r for LoRA?**
- **Hoes does the adaptation matrix ΔW compare to W?**

Which Weight Matrices?

The authors set a parameter budget of 18M (∼35MB in FP16[1]) on GPT-3 175B, which corresponds to $r = 8$ if they adapt one type of attention weights or $r = 4$ if they adapt two types, for all 96 layers.

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL (±0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI (±0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

Table 5: Validation accuracy on WikiSQL and MultiNLI after applying LoRA to different types of attention weights in GPT-3, given the same number of trainable parameters. Adapting both W_q and W_v gives the best performance overall. We find the standard deviation across random seeds to be consistent for a given dataset, which we report in the first column.

[1] FP16: 16-bit floating point (or half precision)

What is The Optimal Rank?

	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL($\pm 0.5\%$)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

Table 6: Validation accuracy on WikiSQL and MultiNLI with different rank r. To our surprise, a rank as small as one suffices for adapting both W_q and W_v on these datasets while training W_q alone needs a larger r. We conduct a similar experiment on GPT-2 in Section H.2.

- **A very small r is sufficient for adapting $\{W_q, W_v\}$.**
- **Increasing r does not cover a more meaningful subspace.**

What is The Optimal Rank?(con't)

Given $A_{r=8}$ and $A_{r=64}$ the learned adaptation matrices with rank $r = 8$ and 64, respectively, using the same pretrained model, let $U_{A_{r=8}}$ and $U_{A_{r=64}}$ be their respective right-singular unitary matrices in SVD. The overlaps between the top i singular vectors $(1 \leq i \leq 8)$ of $U_{A_{r=8}}$ and the top j singular vectors $(1 \leq j \leq 64)$ of $U_{A_{r=64}}$ is measured by a normalized subspace similarity

$$\phi(A_{r=8}, A_{r=64}; i, j) = \frac{\| U^{iT}_{A_{r=8}} U^{j}_{A_{r=64}} \|_F^2}{\min(i, j)} \in [0, 1],$$

where $U^k_{:}$ represents the columns of $U_{\cdot}$ corresponding to the top k singular vectors.

What is The Optimal Rank?(con't)

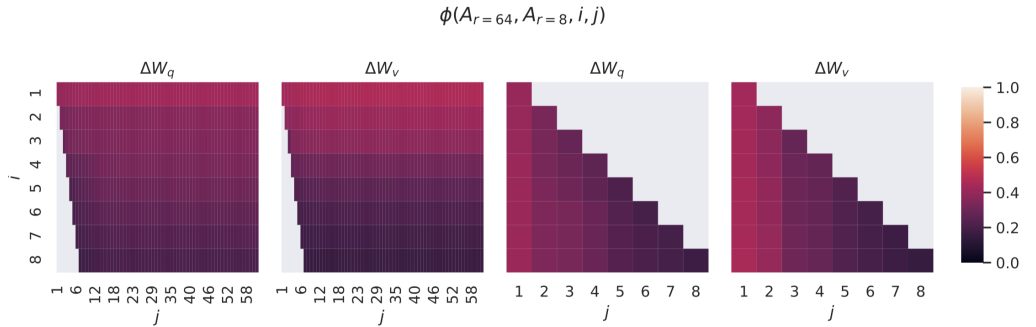


Figure 3: Subspace similarity between column vectors of $A_{r=8}$ and $A_{r=64}$ for both ΔW_q and ΔW_v. The third and the fourth figures zoom in on the lower-left triangle in the first two figures. The top directions in $r = 8$ are included in $r = 64$, and vice versa.

- **The top singular-vector directions of $A_{r=8}$ and $A_{r=64}$ are most useful. Hence, the adaptation matrix can have very low rank.**

What is The Optimal Rank?(con't)

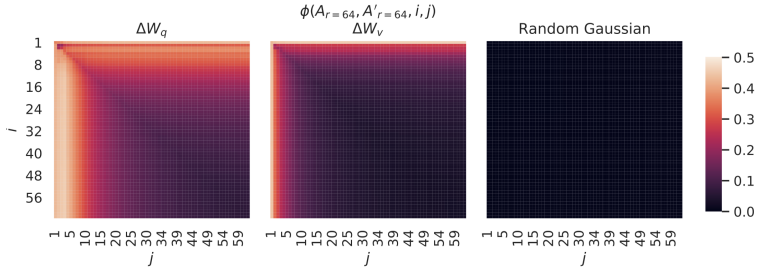


Figure 4: **Left and Middle:** Normalized subspace similarity between the column vectors of $A_{r=64}$ from two random seeds, for both ΔW_q and ΔW_v in the 48-th layer. **Right:** the same heat-map between the column vectors of two random Gaussian matrices. See Section H.1 for other layers.

- **The top singular-vector directions have higher similarities between two adaptation matrices learned from the same settings with different seeds. It indicates a low intrinsic rank within the adaptation matrix A.**

How Does ΔW Compare to W?

Let U/V be the right/left singular-vector matrix of ΔW and compare $\|U^T W V\|_F$ and $\|W\|$. The right/left singular-vector matrix of W or a random matrix is also used for comparison.

	$r = 4$			$r = 64$		
	ΔW_q	W_q	Random	ΔW_q	W_q	Random
$\|\|U^\top W_q V^\top\|\|_F =$	0.32	21.67	0.02	1.90	37.71	0.33
$\|\|W_q\|\|_F = 61.95$	$\|\|\Delta W_q\|\|_F = 6.91$			$\|\|\Delta W_q\|\|_F = 3.57$		

Table 7: The Frobenius norm of $U^\top W_q V^\top$ where U and V are the left/right top r singular vector directions of either (1) ΔW_q, (2) W_q, or (3) a random matrix. The weight matrices are taken from the 48th layer of GPT-3.

- **ΔW has a strong correlation with W compared to a random matrix.**

How Does ΔW Compare to W?

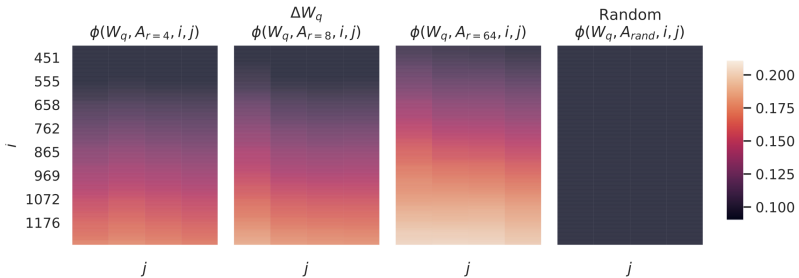


Figure 8: Normalized subspace similarity between the singular directions of W_q and those of ΔW_q with varying r and a random baseline. ΔW_q amplifies directions that are important but not emphasized in W. ΔW with a larger r tends to pick up more directions that are already emphasized in W.

- **The low-rank adaptation matrix potentially amplifies the important features for specific downstream tasks that were learned but not emphasized in the general pretrained model.**

Extensive Works I

- **QLoRA —Quantization-aware efficiency**[Dettmers et al., 2023]
  QLoRA makes LoRA practical for very large language models by backpropagating through a frozen 4-bit quantized backbone and introducing memory-saving techniques that preserve strong finetuning performance.
- **AdaLoRA —Adaptive rank allocation**[Zhang et al., 2023]
  AdaLoRA improves LoRA by parameterizing updates in an SVD-like form and dynamically reallocating the rank budget across layers and matrices according to their importance.
- **LoRA+ —Optimization and learning-rate design**[Hayou et al., 2024]
  LoRA+ improves LoRA by assigning different learning rates to the two adapter matrices, leading to more efficient feature learning, faster finetuning, and often better accuracy at the same parameter cost.

Extensive Works II

- **DoRA —Weight decomposition**[Liu et al., 2024]
  DoRA narrows the gap between LoRA and full fine-tuning by decomposing each pretrained weight into magnitude and direction, and applying LoRA only to the directional update.
- **PiSSA —SVD-based initialization**[Meng et al., 2024]
  PiSSA improves LoRA initialization by using the principal singular values and singular vectors of the pretrained weight matrix, which accelerates convergence and often improves downstream performance.
- **MoRA —Breaking the low-rank bottleneck**[Jiang et al., 2024]
  MoRA addresses LoRA's low-rank bottleneck by using a square trainable matrix together with dimension-transform operators to realize higher-rank updates under a similar parameter budget.

Running LoRA End-to-End: The Shortest Practical Path (credit to chatGPT)

- Goal: fine-tune a small instruction model with LoRA/QLoRA and make it work on a narrow task.
- Recommended path:
  - model: `Qwen2.5-1.5B-Instruct`
  - data: a small public Chinese dataset + a small custom dataset
  - method: QLoRA
  - stack: `Transformers + TRL + PEFT + bitsandbytes + datasets`
- Principle: do not optimize everything at once; first make the full training loop run successfully.

Why This Path Is the Easiest

- Small instruction models are easier to train, debug, and evaluate.
- QLoRA reduces memory usage and makes single-GPU training feasible.
- `SFTTrainer` removes much of the boilerplate.
- PEFT makes saving and loading LoRA adapters straightforward.
- A narrow task makes success easy to observe.

Recommended first project:

- Chinese academic sentence polishing
- Chinese email rewriting
- lab-style Chinese question answering
- structured response generation

Minimal Hardware and Model Choices

- If you only want to *run through the pipeline*, choose a small instruct model.
- Suggested starting points:
  - `Qwen2.5-0.5B-Instruct`
  - `Qwen2.5-1.5B-Instruct`
- Typical beginner setup:
  - a local NVIDIA GPU, or
  - Google Colab / Kaggle notebook
- Use QLoRA first; do not start with full fine-tuning.

A Concrete Small Chinese Dataset

- Recommended teaching dataset:
  - `Hello-SimpleAI/HC3-Chinese`
- Why this dataset is suitable for a first LoRA demo:
  - it is public and easy to access on Hugging Face
  - it is relatively small and classroom-friendly
  - it contains Chinese question-answer pairs
  - it is easy to convert into chat-style SFT data
- Practical teaching choice:
  - use only the `open_qa` subset
  - randomly sample 2,000 examples
  - use the question as the user input
  - use the first human answer as the target output

Why Use the Human Answers Only?

- HC3-Chinese contains both human answers and ChatGPT answers.
- For a first LoRA exercise, using only the human answers makes the target cleaner.
- This also makes the supervision easier to explain:
  - input: a Chinese question
  - target: a human-written Chinese answer
- Pedagogically, this avoids introducing extra noise in the first experiment.

A Good First Dataset Strategy

- Do **not** start with a huge and messy dataset.
- Use:
  - a small public instruction/chat dataset
  - plus 100–300 task-specific examples written by yourself
- Keep the task narrow and the outputs stylistically consistent.
- Split into:
  - training set
  - small test set
- For a first run, even a few hundred clean examples are enough to verify the pipeline.

Dataset Format: Use messages

```
{"messages": [
  {"role": "user",
   "content": "请把这句话改得更学术一些：这个方法效果很好。"},
  {"role": "assistant",
   "content": "该方法表现出较强的有效性。"}
]}

{"messages": [
  {"role": "user",
   "content": "请把这句话改得更礼貌一些：你发得太晚了。"},
  {"role": "assistant",
   "content": "方便的话，下次能稍微早一点发给我吗？谢谢。"}
]}
```

- Store each sample as one JSON line in `train.jsonl` and `test.jsonl`.
- Keep prompt style and answer style consistent across samples.

Environment Setup

```
pip install -U torch transformers trl peft \
    datasets bitsandbytes accelerate sentencepiece
```

What each package does:

- `transformers`: model loading and generation
- `trl`: supervised fine-tuning with `SFTTrainer`
- `peft`: LoRA / QLoRA adapters
- `bitsandbytes`: 4-bit quantization
- `datasets`: loading JSON/CSV/Parquet datasets
- `accelerate`: training infrastructure

How to Convert HC3-Chinese into LoRA Training Data I

```
from datasets import load_dataset
import json

ds = load_dataset("Hello-SimpleAI/HC3-Chinese", split="train")
ds = ds.filter(lambda x: x["source"] == "open_qa"
                         and len(x["human_answers"]) > 0)

ds = ds.shuffle(seed=42).select(range(2000))

def convert(example):
    return {
        "messages": [
            {"role": "user", "content": example["question"].strip()},
            {"role": "assistant",
             "content": example["human_answers"][0].strip()}
        ]
    }

converted = [convert(x) for x in ds]

with open("train.jsonl", "w", encoding="utf-8") as f:
```

How to Convert HC3-Chinese into LoRA Training Data II

```
    for x in converted[:1800]:
        f.write(json.dumps(x, ensure_ascii=False) + "\n")

with open("test.jsonl", "w", encoding="utf-8") as f:
    for x in converted[1800:]:
        f.write(json.dumps(x, ensure_ascii=False) + "\n")
```

How to Convert HC3-Chinese into LoRA Training Data III

What this script does:

- loads the Chinese HC3 dataset
- keeps only the `open_qa` subset
- drops examples without human answers
- randomly samples 2,000 examples
- converts each example into the `messages` format
- writes 1,800 training examples and 200 test examples

Minimal Training Script I

```
import torch
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
from peft import LoraConfig
from trl import SFTTrainer, SFTConfig

model_id = "Qwen/Qwen2.5-1.5B-Instruct"

use_bf16 = torch.cuda.is_available() and torch.cuda.is_bf16_supported()
compute_dtype = torch.bfloat16 if use_bf16 else torch.float16

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=compute_dtype,
)

tokenizer = AutoTokenizer.from_pretrained(model_id, use_fast=True)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
```

Minimal Training Script II

```
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    device_map="auto",
)

dataset = load_dataset(
    "json",
    data_files={"train": "train.jsonl", "test": "test.jsonl"},
)
```

Minimal Training Script III

```
peft_config = LoraConfig(
    r=16,
    lora_alpha=32,
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules="all-linear",
)

args = SFTConfig(
    output_dir="lora_out",
    max_length=1024,
    per_device_train_batch_size=1,
    gradient_accumulation_steps=8,
    num_train_epochs=3,
    learning_rate=1e-4,
    logging_steps=10,
    save_strategy="epoch",
    eval_strategy="epoch",
    warmup_ratio=0.03,
    lr_scheduler_type="cosine",
```

Minimal Training Script IV

```
    bf16=use_bf16,
    fp16=not use_bf16,
    report_to="none",
)

trainer = SFTTrainer(
    model=model,
    args=args,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    processing_class=tokenizer,
    peft_config=peft_config,
)

trainer.train()
trainer.model.save_pretrained("adapter_out")
tokenizer.save_pretrained("adapter_out")
```

What This Script Actually Does

- Loads a small instruction model.
- Quantizes the frozen base model to 4-bit.
- Attaches trainable LoRA adapters to linear layers.
- Fine-tunes only the adapters on your task dataset.
- Saves a lightweight adapter instead of a full new model.

This is the key beginner idea:

$$\text{large frozen base model} + \text{small trainable LoRA adapter}.$$

Minimal Inference Script I

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
from peft import PeftModel

base_model_id = "Qwen/Qwen2.5-1.5B-Instruct"
adapter_path = "adapter_out"

tokenizer = AutoTokenizer.from_pretrained(base_model_id, use_fast=True)
base_model = AutoModelForCausalLM.from_pretrained(
    base_model_id,
    device_map="auto",
    torch_dtype=torch.float16,
)

model = PeftModel.from_pretrained(base_model, adapter_path)
model.eval()

messages = [
    {"role": "user",
     "content": "请把这句话改得更学术一些：这个方法效果很好。"}
]
```

Minimal Inference Script II

```
inputs = tokenizer.apply_chat_template(
    messages,
    tokenize=True,
    add_generation_prompt=True,
    return_tensors="pt",
).to(model.device)

with torch.no_grad():
    outputs = model.generate(inputs, max_new_tokens=128, do_sample=False)

print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

How to Evaluate the First LoRA Project

- Do not rely on training loss alone.
- Compare:
  - the base model output
  - the LoRA-adapted model output
- Use a small held-out test set.
- Ask simple questions:
  - Is the output more consistent?
  - Is the style closer to the target style?
  - Is the format more reliable?
  - Is the task performance visibly better?

Common Beginner Mistakes

- Using a dataset that is too large, too noisy, or too diverse.
- Starting with a model that is too large for the hardware.
- Mixing many tasks in the first experiment.
- Looking only at loss and not at actual generations.
- Using inconsistent prompt or response formats.
- Trying RLHF, DPO, or multi-stage pipelines before basic SFT works.

A Practical Teaching Recipe

1. Choose a small instruction model.
2. Prepare a narrow dataset in `messages` format.
3. Install `transformers + trl + peft + bitsandbytes`.
4. Run QLoRA training with `SFTTrainer`.
5. Save the adapter.
6. Compare the adapter model with the base model on a small test set.

Pedagogical message: the first goal is not state-of-the-art performance; the first goal is to make the entire LoRA workflow run successfully from data to evaluation.

Take-Home Message

- The easiest path is:

  $$\text{small instruct model} + \text{clean narrow dataset} + \text{QLoRA} + \text{simple evaluation}.$$

- LoRA is most useful when:
  - the task is specific,
  - the data is consistent,
  - and hardware is limited.
- Once the first project works, you can move on to:
  - better datasets
  - hyperparameter tuning
  - larger models
  - improved LoRA variants

References I

- Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2022). Lora: Low-rank adaptation of large language models. ICLR, 1(2), 3.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.
- Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized LLMs. Advances in neural information processing systems, 36.
- Zhang, Q., Chen, M., Bukharin, A., He, P., Cheng, Y., Chen, W., & Zhao, T. (2023). Adaptive budget allocation for parameter-efficient fine-tuning. In The eleventh international conference on learning representations.
- Hayou, S., Ghosh, N., & Yu, B. (2024). LoRA+: Efficient low rank adaptation of large models. In Proceedings of the 41st International Conference on Machine Learning (pp. 17783-17806). PMLR.
- Liu, S.-Y., Wang, C.-Y., Yin, H., Molchanov, P., Wang, Y.-C. F., Cheng, K.-T., & Chen, M.-H. (2024). DoRA: Weight-decomposed low-rank adaptation. In Proceedings of the 41st International Conference on Machine Learning (pp. 32100-32121). PMLR.

References II

Meng, F., Wang, Z., & Zhang, M. (2024). PiSSA: Principal singular values and singular vectors adaptation of large language models. Advances in neural information processing systems, 37.

Jiang, T., Huang, S., Luo, S., Zhang, Z., Huang, H., Wei, F., Deng, W., Sun, F., Zhang, Q., Wang, D., & Zhuang, F. (2024). MoRA: High-rank updating for parameter-efficient fine-tuning. arXiv preprint arXiv:2405.12130.