WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Domain-Specific LLM Fine-Tuning Using LoRA

Shantanu Pande[1,*], Akshat Meshram[2], Ashish Singh[3], Prof. Sangeeta Alagi[4]

[1,2,3,4]Department of Artificial Intelligence & Machine Learning, ISBM College of Engineering, Nande, Pune, Maharashtra, India

ABSTRACT :

Large language models answer general questions well but often fall short when a task needs a specific domain, such as customer-support replies or short Python programs. Fine-tuning all the weights of a model with billions of parameters is too expensive for a single free GPU. This paper describes a low-cost system that adapts one base model, Qwen2.5-3B-Instruct, to two domains with low-rank adaptation (LoRA) applied to a 4-bit quantized copy of the model (QLoRA). One adapter is trained on 2,000 customer-support examples and the other on 2,000 Python instruction–solution pairs. Only 0.96% of the parameters are trained, each adapter is about 114 MB, and the two runs took 26 and 14 minutes on a Tesla T4 GPU. Validation loss ended at 0.659 for the support adapter and 0.582 for the Python adapter. A small chat program attaches both adapters to the same base model and switches between them at run time. These are preliminary results from a project in progress, and evaluation on held-out test data is left as future work.

Keywords: LoRA, QLoRA, parameter-efficient fine-tuning, large language models, domain adaptation, customer support, code generation

1. Introduction

Large language models (LLMs) built on the transformer architecture (Vaswani et al., 2017) are now used for tasks that range from answering customer queries to writing code. After pre-training on very large text collections they are usually tuned to follow instructions (Wei et al., 2022; Ouyang et al., 2022), and the result is a model that can talk about almost anything. That breadth has a price. When a task belongs to a narrow domain, a general model often gives answers that are right in spirit but wrong in detail: a support reply that never mentions the actual steps of a process, or a code answer that is mostly explanation and little program. Training the model further on examples from the domain is the usual remedy (Gururangan et al., 2020).

The difficulty is hardware. Updating every weight of a model with three billion parameters needs memory for the weights, the gradients and the optimizer state. A common estimate for mixed-precision training with the Adam optimizer is about 16 bytes per parameter (Rajbhandari et al., 2020), which comes to roughly 50 GB for a model of this size. The free GPU we had access to has 15 GB. Students and small teams are left with two options: pay for larger machines, or skip fine-tuning altogether.

Parameter-efficient fine-tuning removes most of that cost by freezing the original weights and training only a small number of new ones (Houlsby et al., 2019; Lialin et al., 2023). Low-rank adaptation, or LoRA (Hu et al., 2022), is the most widely used of these methods, and QLoRA (Dettmers et al., 2023) goes one step further by keeping the frozen model in 4-bit precision while it is being trained. Together they make it possible to fine-tune a model of this size on a single entry-level GPU.

This paper describes a system that applies that idea to two domains at once. We start from one base model, Qwen2.5-3B-Instruct, and train two separate LoRA adapters: one for customer-support questions and one for Python programming tasks. A small chat program loads both adapters on the same base model and lets the user switch between them, or switch them off, while it is running. The project is still in progress. What we report here is the design of the system, the training procedure, and the first measurements taken during training, and we say plainly which parts have not been tested yet. The contributions are:

- a complete fine-tuning pipeline that runs on a free 15 GB GPU;
- a design in which one frozen base model serves several domains through swappable adapters;
- preliminary training results and resource measurements for two adapters.

Section 2 reviews related work. Section 3 describes the proposed system, Section 4 the data and training method, and Section 5 the preliminary results. Section 6 discusses the limits of what we have done so far, and Section 7 concludes with the next steps.

2. Related work

2.1. Parameter-efficient fine-tuning

Adapter modules were one of the first practical ways to avoid full fine-tuning. Houlsby et al. (2019) inserted small bottleneck layers into each transformer block and trained only those, and reached accuracy close to full fine-tuning on a set of language-understanding tasks. LoRA (Hu et al., 2022) takes a different route. Instead of adding layers, it writes the change to a weight matrix as the product of two thin matrices, so the number of trainable values is small, and because the update can be merged into the original weight there is no added delay at inference. QLoRA (Dettmers et al., 2023) combines LoRA

with a 4-bit NormalFloat (NF4) representation of the frozen weights, double quantization of the quantization constants, and paged optimizers, and showed that a 65-billion-parameter model can be fine-tuned on a single 48 GB GPU. Lialin et al. (2023) survey these and related methods.

2.2. Domain adaptation and instruction tuning

Gururangan et al. (2020) showed that continuing to train a language model on text from the target domain helps even when the model is already strong. For chat-style applications the more direct route is instruction tuning, where the model is trained on tasks written in natural language (Wei et al., 2022; Ouyang et al., 2022). That is the kind of training we use: each example is a question paired with the answer we want the model to give.

2.3. Models and tools

Qwen2.5 is a family of open-weight models with sizes from 0.5 to 72 billion parameters (Qwen Team, 2024). We chose the 3-billion-parameter instruction-tuned version because it fits on our GPU once quantized and already understands the chat format. The implementation uses the Hugging Face Transformers library (Wolf et al., 2020), PEFT for the adapters (Mangrulkar et al., 2022) and TRL for the supervised fine-tuning loop (von Werra et al., 2020).

3. Proposed system

3.1. Overview

Figure 1 shows the stages of the system. Instruction–response pairs from a public dataset are first converted into a short chat with one user message and one assistant message. The text is tokenized with the model's own tokenizer and cut at 512 tokens. The base model is loaded in 4-bit precision and the LoRA layers are attached. A supervised fine-tuning loop then trains only those layers and checks the loss on a validation set at the end of each epoch. When training finishes only the adapter weights are saved; nothing is written back to the base model.

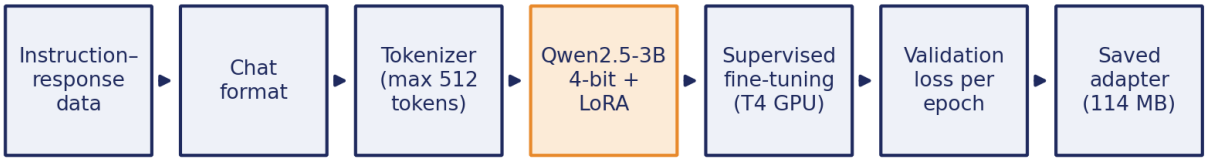


Fig. 1 - Stages of the fine-tuning pipeline.

3.2. One base model, several adapters

The main design decision is to keep the base model fixed and treat every domain as a separate adapter. An adapter is a file of about 114 MB, while the base model takes several gigabytes, so adding a domain means training and storing one more small file. The adapters do not interfere with each other, because none of them changes the base weights: the support adapter can be retrained or replaced without touching the Python one. Figure 2 shows the arrangement.

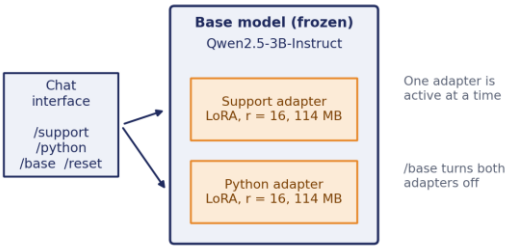


Fig. 2 - One frozen base model with two swappable LoRA adapters.

3.3. Chat interface

For demonstration we wrote a command-line chat loop in the same notebook. The base model is loaded in 16-bit precision and both saved adapters are attached to it under the names "support" and "python". Typing /support or /python selects an adapter and clears the conversation, /base switches the adapters off so that the unmodified model answers, and /reset clears the history. Replies are sampled with a temperature of 0.5 and top-p of 0.9, up to 250 new tokens. Because the base-model mode is just the same model with the adapters disabled, comparing the tuned and untuned answer to one prompt does not need a second copy of the model in memory.

4. Methodology

4.1. Datasets

We used two public instruction datasets from the Hugging Face Hub (Table 1). For customer support we took the Bitext customer-support chatbot dataset (Bitext, n.d.), where every row holds a customer message (instruction) and an agent answer (response). For Python we took a collection of about 18 thousand programming tasks in the Alpaca format (iamtarun, n.d.), where each row has an instruction, an optional input and the solution code in the output field. From each dataset we shuffled the rows with a fixed seed (42) and kept the first 2,000. Two thousand examples is a small sample, but it keeps each training run short enough for a free GPU session. The support data was divided 80/10/10 by a random split with the same seed; the Python data was divided in order after shuffling.

Table 1 - Datasets and splits used for the two adapters.

Dataset	Domain	Fields used	Rows used	Train / Val / Test
Bitext customer support	Customer support	instruction, response	2,000	1,600 / 200 / 200
Python code instructions 18k	Python programming	instruction, input, output	2,000	1,600 / 200 / 200

4.2. Data preparation

Each row was converted to the chat format that Qwen2.5 expects. The customer message or the programming instruction becomes the user turn, and the agent answer or the code becomes the assistant turn. For the Python data, when the input field is not empty it is appended to the instruction after a blank line. The model's chat template then turns the two turns into a single training string, and sequences longer than 512 tokens are truncated.

4.3. Quantized base model

The base model was loaded with 4-bit NF4 quantization and double quantization, with computation in 16-bit floating point (Dettmers et al., 2023). It was then passed through the PEFT helper for k-bit training, which prepares the quantized model for gradient-based training, for example by enabling gradient checkpointing and keeping some small layers in higher precision.

4.4. LoRA configuration

LoRA keeps a pretrained weight matrix W_0 of size $d \times k$ frozen and learns its change as the product of two small matrices, B of size $d \times r$ and A of size $r \times k$, where the rank r is much smaller than d and k. For an input x the layer computes

$$h = W_0 x + \frac{\alpha}{r} BAx \tag{1}$$

where α is a scaling constant (Hu et al., 2022). We used $r = 16$, $\alpha = 32$ and a dropout of 0.05 on the adapter input, and attached adapters to all seven linear projections of every transformer block: the query, key, value and output projections of the attention layers, and the gate, up and down projections of the feed-forward layers. With this setting PEFT reports 29,933,568 trainable parameters out of 3,115,872,256, which is 0.96% of the model. Both adapters use the same configuration, so they differ only in the data they were trained on.

4.5. Training setup

Training used the supervised fine-tuning trainer from TRL. The loss is the usual next-token cross-entropy over a sequence of N tokens,

$$L = -\frac{1}{N} \sum_{t=1}^{N} \log p_\theta(y_t \mid y_{<t}, x) \tag{2}$$

where θ stands for the adapter weights, the only parameters that change. Table 2 lists the settings. The per-device batch size was 4 with gradient accumulation over 4 steps, which gives an effective batch of 16, and the learning rate started at 2×10^{-4} with a cosine schedule. The optimizer was the paged 8-bit AdamW from the bitsandbytes library, and gradient checkpointing was on. One practical detail: the trainable adapter weights have to be kept in 32-bit precision, because the gradient scaler used in 16-bit training refuses to unscale gradients of half-precision parameters, so we cast them to float32 right after the trainer is created. The support adapter was trained for two epochs and the Python adapter for one. All runs used a free Tesla T4 GPU with 15 GB of memory.

Table 2 - Training configuration used for both adapters.

Setting	Value
Base model	Qwen2.5-3B-Instruct (4-bit NF4)
LoRA rank / alpha / dropout	16 / 32 / 0.05
Target modules	q, k, v, o, gate, up, down projections
Learning rate (schedule)	2×10^{-4} (cosine)
Batch size (per device × accumulation)	4 × 4 = 16
Epochs (support / Python)	2 / 1
Maximum sequence length	512 tokens
Optimizer	Paged AdamW, 8-bit
Precision	fp16 compute, adapter weights in fp32
Seed (data shuffling and splitting)	42
GPU	Tesla T4, 15 GB

4.6. Evaluation

After every epoch the trainer measures the loss on the 200 validation examples and also reports the mean token accuracy, the share of tokens for which the model's top prediction matches the reference. Validation loss is our main number. As a sanity check we also ran the base model and the tuned model on five hand-written prompts per domain and read the answers side by side. The support prompts were everyday cases such as a damaged order, a password reset and a double charge; the Python prompts were short tasks such as reversing a string, testing whether a number is prime and writing a bank-account class. The two 200-example test sets were held back and have not been used yet.

5. Preliminary results

5.1. Training and validation loss

Table 3 gives the values logged at the end of each epoch, and Fig. 3 plots the two losses. The training loss in the table is the last value logged during that epoch.

Table 3 - Loss and token accuracy at the end of each epoch.

Adapter	Epoch	Training loss	Validation loss	Token accuracy
Support	1	0.644	0.651	0.799
Support	2	0.452	0.659	0.800
Python	1	0.640	0.582	0.838

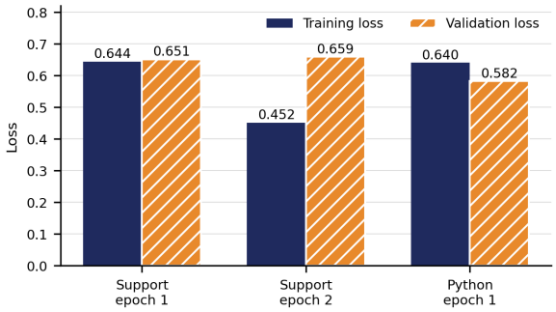


Fig. 3 - Training and validation loss at the end of each epoch.

For the support adapter the training loss fell from 0.644 after the first epoch to 0.452 after the second, but the validation loss stayed about where it was, moving from 0.651 to 0.659, and the token accuracy hardly changed (0.799 to 0.800). In the second epoch the model fitted the training examples better without getting better on unseen ones, which is an early sign of overfitting. The rise of 0.008 is small and we have a single run, so we do not read much into it, but it suggests that a third epoch would not help with 1,600 examples. The Python adapter, trained for one epoch, ended with a validation loss of 0.582 and a token accuracy of 0.838. The two adapters should not be ranked by these numbers. The losses are measured on different texts, and code, with its fixed syntax, is probably easier to predict token by token than free-form customer replies.

5.2. Resource use

Table 4 summarises the cost of each run. The support run took 200 optimizer steps and about 26 minutes; the Python run took 100 steps and about 14 minutes. In both cases only 29.9 million of 3.12 billion parameters were trained, and each saved adapter is 114.3 MB. The two adapters together take about 228 MB, a few percent of the roughly 6 GB needed for the base model in 16-bit precision.

Table 4 - Cost of training the two adapters on a Tesla T4.

Measure	Support adapter	Python adapter
Epochs	2	1
Optimizer steps	200	100
Training time	26 min 8 s	14 min 20 s
Final validation loss	0.659	0.582
Trainable parameters	29,933,568 (0.96%)	29,933,568 (0.96%)
Adapter file size	114.3 MB	114.3 MB

5.3. Side-by-side checks

The side-by-side prompts described in Section 4.6 were used to confirm that the adapters load correctly and can be switched on and off in the chat program. We have not scored these answers and draw no conclusion about answer quality from them.

6. Discussion and limitations

The results so far show that the pipeline trains without problems and at low cost. They do not yet show that the adapters give better answers than the base model, and several limits should be kept in mind.

- Each adapter comes from a single run with one random seed, so we cannot say how much the numbers vary between runs.
- Only validation loss and token accuracy have been measured. A lower loss shows that the model fits the data, not that its answers are better. We have no task-level metric yet, such as the share of generated functions that pass unit tests or a human rating of support replies.
- The two adapters were not trained under identical conditions (two epochs against one), so they should not be compared with each other.
- The training sets are small, 1,600 examples each. Both datasets are public collections and not logs from a real deployment, so the results may differ in practice.
- The adapters were trained against the 4-bit base model, but the chat program attaches them to a 16-bit copy. This is common practice and the difference is usually small, but we have not measured it for our adapters.

7. Conclusion and future work

We described a low-cost way to adapt one 3-billion-parameter model to two domains with LoRA adapters on a 4-bit base model, and we built a chat program that switches between the adapters at run time. The first measurements show that training fits on a free GPU: 0.96% of the parameters are trained, each adapter is 114 MB, and a run takes between 14 and 26 minutes. Because the evaluation is still limited, we regard this paper as a progress report on the system and not as a claim that the adapters beat the base model. The next steps are:

- score the held-out test sets: test loss and perplexity for the base model and each adapter, text-similarity scores against the reference answers for support, and the pass rate on unit-tested tasks for Python;
- run controlled comparisons with equal epochs, several seeds, and ranks of 8, 16 and 32;
- train on larger parts of both datasets and check whether the overfitting seen in the second support epoch goes away;
- choose the adapter automatically from the user's message instead of by manual command;
- package the chat program as a small web application.

Acknowledgements

The authors thank Prof. Sangeeta Alagi for her guidance throughout this project, and the Department of Artificial Intelligence & Machine Learning at ISBM College of Engineering, Nande, Pune, for the opportunity to carry out this work. All experiments were run on the free GPU tier of Google Colab.

REFERENCES :

1. Bitext. (n.d.). *Bitext customer support LLM chatbot training dataset* [Data set]. Hugging Face. https://huggingface.co/datasets/bitext/Bitext-customer-support-llm-chatbot-training-dataset
2. Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems, 36*.
3. Gururangan, S., Marasović, A., Swayamdipta, S., Lo, K., Beltagy, I., Downey, D., & Smith, N. A. (2020). Don't stop pretraining: Adapt language models to domains and tasks. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 8342–8360.
4. Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., & Gelly, S. (2019). Parameter-efficient transfer learning for NLP. *Proceedings of the 36th International Conference on Machine Learning*, 2790–2799.
5. Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2022). LoRA: Low-rank adaptation of large language models. *International Conference on Learning Representations*.

6. iamtarun. (n.d.). *Python code instructions 18k Alpaca* [Data set]. Hugging Face. https://huggingface.co/datasets/iamtarun/python_code_instructions_18k_alpaca
7. Lialin, V., Deshpande, V., & Rumshisky, A. (2023). *Scaling down to scale up: A guide to parameter-efficient fine-tuning*. arXiv:2303.15647.
8. Mangrulkar, S., Gugger, S., Debut, L., Belkada, Y., Paul, S., & Bossan, B. (2022). *PEFT: State-of-the-art parameter-efficient fine-tuning methods*. https://github.com/huggingface/peft
9. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems, 35*, 27730–27744.
10. Qwen Team. (2024). *Qwen2.5 technical report*. arXiv:2412.15115.
11. Rajbhandari, S., Rasley, J., Ruwase, O., & He, Y. (2020). ZeRO: Memory optimizations toward training trillion parameter models. *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–16.
12. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems, 30*, 5998–6008.
13. von Werra, L., Belkada, Y., Tunstall, L., Beeching, E., Thrush, T., Lambert, N., & Huang, S. (2020). *TRL: Transformer reinforcement learning*. https://github.com/huggingface/trl
14. Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2022). Finetuned language models are zero-shot learners. *International Conference on Learning Representations*.
15. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., et al. (2020). Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45.