

HOMWORK 3

APPLYING AND ADAPTING LLMs *

10-423/623/723 GENERATIVE AI
<http://423.mlcourse.org>

OUT: October 5th, 2026
DUE: October 21st, 2026
TAs: Catherine, Dhruv, Jeffrey

Instructions

- **Collaboration Policy:** Please read the collaboration policy in the syllabus.
- **Late Submission Policy:** See the late submission policy in the syllabus.
- **Submitting your work:** You will use [PhiGrade](#) to submit answers to all questions and code.
 - **Written:** You will submit your completed homework as a PDF to [PhiGrade](#). Please use the provided template. Submissions can be handwritten, but must be clearly legible; otherwise, you will not be awarded marks. Alternatively, submissions can be written in $\text{\LaTeX}$. Each answer should be within the box provided. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader and there will be a **2% penalty** (e.g., if the homework is out of 100 points, 2 points will be deducted from your final score).
 - **Programming:** You will submit your code for programming questions by running the [PhiGrade](#) autograder on your own machine. We will examine your code by hand and may award marks for its submission.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

Question	Points
$\text{\LaTeX}$ Template Alignment	0
In-Context Learning	14
Parameter Efficient Fine-Tuning	10
Direct Preference Optimization	15
Programming: LoRA and On-Policy Distillation	23
Code Upload	0
Collaboration Questions	2
Total:	64

*Compiled on Friday 9th October, 2026 at 14:46

1 $\text{\LaTeX}$ Template Alignment (0 points)

1.1. (0 points) **Select one:** Did you use $\text{\LaTeX}$ for the entire written portion of this homework?

☐ Yes

☐ No

1.2. (0 points) **Select one:** I have ensured that my final submission is aligned with the original template given to me in the handout file and that I haven't deleted or resized any items or made any other modifications which will result in a misaligned template. I understand that incorrectly responding yes to this question will result in a penalty equivalent to 2% of the points on this assignment.

Note: Failing to answer this question will not exempt you from the 2% misalignment penalty.

☐ Yes

2 In-Context Learning (14 points)

- 2.1. (2 points) Describe the difference between in-context learning and few-shot chain-of-thought prompting.

- 2.2. (3 points) Write a prompt that might help facilitate in-context learning for the following question:

The cost of electricity per kilowatt-hour increases by 5 cents. Last month, a family used 150 kilowatt-hours at the old rate. This month, they used 100 kilowatt-hours at the new rate. Altogether, their electricity bills for these two months amount to \$45. How much was the old rate per kilowatt-hour?

- 2.3. (3 points) Modify your answer from the previous question to use one-shot chain-of-thought prompting (i.e. few-shot chain-of-thought prompting with only one example).

- 2.4. (3 points) Modify your answer from the previous question to use zero-shot chain-of-thought prompting.

- 2.5. (2 points) Describe an advantage and a disadvantage of zero-shot chain-of-thought prompting as compared to chain-of-thought prompting.

- 2.6. (1 point) Meta learning refers to a subfield of machine learning involving the process of learning how to learn. One use case of meta learning is determining adaptation rules that, given small amounts of data for new tasks, facilitate good performance. Describe a similarity and a difference between in-context learning and meta learning.

3 Parameter Efficient Fine-Tuning (10 points)

- 3.1. Suppose you are building a simple feed-forward neural network consisting of L layers. Each layer is fully connected with sigmoid activations, $\sigma(\cdot)$, applied elementwise. The D_0 input features are $\mathbf{z}_0$. The l th hidden layer $\mathbf{z}_l$ has D_l hidden units. The output layer $\mathbf{z}_L$ has only one unit $D_L = 1$. The model architecture is defined as follows for each, $\mathbf{z}_l \in \mathbb{R}^{D_l}$:

$$\mathbf{z}_l = \sigma(\mathbf{W}_l \mathbf{z}_{l-1} + \mathbf{b}_l), \quad \forall l \in \{1, \dots, L\}$$

where $\mathbf{W}_l$ and $\mathbf{b}_l$ are the parameters of the model.

In the questions below, assume that $L = 10$ and $D_l = 2^{L-l}$ for all $l \in \{0, \dots, L\}$.

- 3.1.a. (1 point) **Numerical answer:** How many hidden units are in this model? Your answer must be an integer.

- 3.1.b. (1 point) **Numerical answer:** How many parameters are in this model? Your answer must be an integer.

- 3.1.c. (1 point) **Numerical answer:** Now suppose we instead do parameter efficient fine-tuning in a style similar to BitFit ([Ben-Zaken et al., 2021](#)). Specifically, we leave the architecture unchanged and we fine-tune only the bias terms, $\mathbf{b}_l, \forall l \in \{1, \dots, L\}$, keeping all other parameters fixed. What percentage of the total parameters are fine-tuned in this setting? Your answer must be a percentage. (Report percent with two decimal places.)

- 3.1.d. (2 points) **Numerical answer:** Suppose we instead inject a bottleneck adapter module (of the variety introduced by [Houlsby et al. \(2019\)](#)) after each layer:

$$\begin{aligned} \mathbf{z}_l &= \sigma(\mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l), \quad \forall l \in \{1, \dots, L\} \\ \mathbf{a}_l &= \text{adapter}(\mathbf{z}_l, \mathbf{V}_l), \quad \forall l \in \{1, \dots, L\} \end{aligned}$$

where $\mathbf{a}_0 = \mathbf{z}_0$ and $\mathbf{a}_l \in \mathbb{R}^{D_l}$, $\mathbf{V}_l = [\mathbf{V}_{l,\text{down}}, \mathbf{v}_{l,\text{down}}, \mathbf{V}_{l,\text{up}}, \mathbf{v}_{l,\text{up}}]$ is a collection of all the adapter parameters for the l th adapter layer, and:

$$\text{adapter}(\mathbf{a}, \mathbf{V}) = \mathbf{a} + (\mathbf{V}_{\text{up}} \tilde{\sigma}(\mathbf{V}_{\text{down}} \mathbf{a} + \mathbf{v}_{\text{down}}) + \mathbf{v}_{\text{up}})$$

We follow the original paper and define $\tilde{\sigma}(\cdot) = \text{GELU}(\cdot)$ as the nonlinearity—GELU has no parameters. Assume the rank of the adapter is 4, i.e. its hidden layer after down-projection has 4 units. We tune only the adapter parameters. What percentage of the total parameters are fine-tuned in this setting? Your answer must be a percentage. (Report percent with two decimal places.)

Note: The total number of parameters you start with for the base architecture should be the same as the parameters in 3.1.b, do not carry over additional parameters from 3.1.c. From there, the total number of parameters will include the parameters from the base architecture and the parameters from the adapters.

- 3.2. In Prefix Tuning ([Li & Liang, 20](#)), we augment each attention head with a prefix, whose parameters are fine tuned while the rest of the model remains fixed. If $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ are the standard query, key, and value matrices in attention, the prefix tuning does the following attention computation:

$$\begin{aligned} \tilde{\mathbf{K}} &= [\mathbf{P}_K; \mathbf{K}] \\ \tilde{\mathbf{V}} &= [\mathbf{P}_V; \mathbf{V}] \\ \tilde{\mathbf{A}} &= \text{softmax}(\mathbf{Q}\tilde{\mathbf{K}}^T / \sqrt{d}) \\ \tilde{\mathbf{X}}' &= \tilde{\mathbf{A}}\tilde{\mathbf{V}} \end{aligned}$$

where $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{n \times d}$ and $\tilde{\mathbf{K}}, \tilde{\mathbf{V}} \in \mathbb{R}^{m \times d}$ with $m = n + p$ where p is the prefix length. $\mathbf{P}_K$ and $\mathbf{P}_V$ are the learned prefix parameters (typically encoded via a bottleneck network).

- 3.2.a. (2 points) Write down how $\tilde{A}_{i,j}$ the (i, j) th entry of the attention weights is computed in prefix tuning. Your answer must be in terms of the $\exp(\cdot)$ function. (You should not refer to $\text{softmax}(\cdot)$).

- 3.2.b. (3 points) **Proof:** Conceptually, the key/value vectors $\tilde{\mathbf{K}}_i$ and $\tilde{\mathbf{V}}_i$ can be divided into *prefix* tokens, $i \in \{1, \dots, p\}$, and *content* tokens, $i \in \{p+1, \dots, p+n\}$.

Show that Prefix Tuning does not change the *relative* attention weight between content tokens. That is, show that for any $i \in \{1, \dots, n\}$ and any $j, k \in \{p+1, \dots, p+n\}$ we have:

$$\frac{\tilde{A}_{i,j}}{\tilde{A}_{i,k}} = \frac{A_{i,(j-p)}}{A_{i,(k-p)}}$$

where A_{ij} are the attention weights of regular attention without prefix tuning.

4 Direct Preference Optimization (15 points)

- 4.1. **Background (PPO in RLHF).** After fitting a reward model $r(x, y)$ from preference data, RLHF typically fine-tunes a policy by maximizing a KL-regularized objective:

$$\max_{\pi} \mathbb{E}_{x \sim D, y \sim \pi(\cdot|x)} [r(x, y)] - \beta D_{\text{KL}}(\pi(\cdot|x) \parallel \pi_{\text{ref}}(\cdot|x)),$$

where π_{ref} is usually the SFT model and $\beta > 0$ controls how far π may drift from π_{ref} . In practice, Proximal Policy Optimization (PPO) is used to optimize this objective with policy-gradient updates and clipping.

In this question, you will step through the derivation of the direct preference optimization (DPO) objective function. Recall that in DPO, we assume there exists a latent reward function $r^*(x, y)$ that returns a real-valued score which represents how good some response y is given a prompt x . We further assume that this latent reward governs the probability that a human ranker prefers response y_w to another response y_l according to the formula

$$p(y_w \succ y_l \mid x) = \frac{\exp r^*(x, y_w)}{\exp r^*(x, y_w) + \exp r^*(x, y_l)} \quad (1)$$

where $y_w \succ y_l$ indicates that y_w is preferred over y_l .

Given a (parameterized) LLM that defines a conditional distribution over responses given a prompt, $\pi_{\phi}(y \mid x)$, the goal of DPO is to fine-tune the parameters, ϕ , such that the expected reward subject to a KL-divergence penalty is maximized:

$$\pi_{\phi}^*(y \mid x) = \operatorname{argmax}_{\pi_{\phi}(y|x)} \mathbb{E}_{\pi_{\phi}(y|x)} \left[r^*(x, y) - \beta \log \frac{\pi_{\phi}(y \mid x)}{\pi_{\text{ref}}(y \mid x)} \right] \quad (2)$$

for some reference LLM π_{ref} .

As you work on the following questions, you are encouraged to read the DPO paper ([Rafailov et al.](#)) all the way to the end.

4.1.a. (3 points) Show that the optimizer of the KL-constrained objective has the form

$$\pi_{\phi}^*(y | x) = \frac{1}{Z(x)} \pi_{\text{ref}}(y | x) \exp\left(\frac{1}{\beta} r^*(x, y)\right) \quad (3)$$

for a normalizing partition function $Z(x) = \sum_y \pi_{\text{ref}}(y|x) \exp(\frac{1}{\beta} r^*(x, y))$. Explain briefly why, in principle, having this closed-form solution means that we need not run a policy-gradient loop to obtain the optimal policy distribution.

Hint: for the proof, first manipulate the objective function in (2) to be of the form

$$\pi_{\phi}^*(y | x) = \underset{\pi_{\phi}(y|x)}{\operatorname{argmin}} KL(\pi_{\phi}(y | x) || p(y | x)) + C \quad (4)$$

for some distribution $p(y|x)$ and constant term(s), C , that don't depend on y . From there, argue what distribution $\pi_{\phi}(y | x)$ optimizes the objective in (4). You may reference Appendix A.1 of [the original DPO paper](#).

- 4.1.b. (2 points) Rearrange the expression from part 4.1.a to write the reward in terms of a policy ratio and a partition term:

$$r(x, y) = \beta \log \frac{\pi_\phi(y | x)}{\pi_{\text{ref}}(y | x)} + \beta \log Z(x). \quad (5)$$

State why $Z(x)$ depends only on x (and the reference) and not on y or the learned policy.

- 4.1.c. (1 point) Under the Bradley–Terry model, only *differences* in reward matter:

$$p(y_w \succ y_l | x) = \sigma(r(x, y_w) - r(x, y_l)).$$

Substituting the reparameterized reward from (5) yields a preference probability purely in terms of policy–reference log-ratios where $\log Z(x)$ cancels:

$$p(y_w \succ y_l | x) = \sigma \left(\beta \log \frac{\pi_\phi^*(y_w | x)}{\pi_{\text{ref}}(y_w | x)} - \beta \log \frac{\pi_\phi^*(y_l | x)}{\pi_{\text{ref}}(y_l | x)} \right) \quad (6)$$

Replace π_ϕ^* with a trainable π_ϕ and write the maximum-likelihood objective over preference pairs to obtain the DPO loss—a binary cross-entropy on the difference of log-ratios:

$$\mathcal{L}_{\text{DPO}}(\pi_\phi; \pi_{\text{ref}}) = -\mathbb{E}_{(x, y_w, y_l) \sim D} \left[\log \sigma \left(\beta \left[\log \frac{\pi_\phi(y_w | x)}{\pi_{\text{ref}}(y_w | x)} - \log \frac{\pi_\phi(y_l | x)}{\pi_{\text{ref}}(y_l | x)} \right] \right) \right].$$

Explain how this turns PPO’s RL objective into a supervised learning problem on preference pairs.

- 4.1.d. (2 points) (3) shows that, given a reward r , the KL-regularized optimum has a Gibbs form $\pi_\phi(y|x) \propto \pi_{\text{ref}}(y|x) \exp(r(x, y)/\beta)$. So why do PPO-based RLHF methods *not* simply use this closed form? Give two reasons.

- 4.1.e. (3 points) The probability in (6) can be maximized by minimizing the objective function

$$\ell_{DPO}(\phi) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} [\log \sigma(\delta(\phi))] . \quad (7)$$

where $\delta(\phi) = \beta \log \frac{\pi_\phi(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\phi(y_l|x)}{\pi_{\text{ref}}(y_l|x)}$.

Show that the gradient of ℓ_{DPO} , $\nabla_\phi \ell_{DPO}(\phi)$, is equal to

$$\nabla_\phi \ell_{DPO}(\phi) = \mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[\sigma(-\delta(\phi)) \left(-\beta \frac{\nabla_\phi \pi_\phi(y_w | x)}{\pi_\phi(y_w | x)} + \beta \frac{\nabla_\phi \pi_\phi(y_l | x)}{\pi_\phi(y_l | x)} \right) \right] \quad (8)$$

4.1.f. The gradient in (8) can be used to optimize the parameters of π_ϕ directly i.e., without having to train an intermediate reward model! The two terms inside the expectation of (8) can be interpreted as having different effects on the actual updates made to π_ϕ .

4.1.f.i. (2 points) What *direction* does the second term, $-\beta \frac{\nabla_\phi \pi_\phi(y_w|x)}{\pi_\phi(y_w|x)} + \beta \frac{\nabla_\phi \pi_\phi(y_l|x)}{\pi_\phi(y_l|x)}$, encourage the parameters to move in? Recall that gradient descent moves in the *opposite* direction of the gradient. **Hint:** think about the effect of updating ϕ in this direction on the likelihoods $\pi_\phi(y_w | x)$ and $\pi_\phi(y_l | x)$.

4.1.f.ii. (2 points) Under what conditions is the *magnitude* of the first term, $\sigma(-\delta(\phi))$ large i.e., ≥ 0.5 ? Frame your answer in terms of the likelihoods $\pi_\phi(y_w | x)$ and $\pi_\phi(y_l | x)$.

5 Programming: LoRA and On-Policy Distillation (23 points)

Introduction

For large pre-trained models, full fine-tuning, which retrain all model parameters, becomes less feasible due to the increased training time and memory requirements. In this section, you will explore, and build from scratch, a parameter efficient fine-tuning (PEFT) method, **Low Rank Adaptation (LoRA)**, and apply it to a pre-trained Qwen2.5 model.

You will use LoRA to teach a small model, [Qwen2.5-0.5B-Instruct](#) (the *student*), to call tools, and compare two ways of doing it on the same training set: supervised fine-tuning (SFT), which trains on the dataset's gold answers, and on-policy distillation (OPD), in which the student writes its own answers and a stronger model, [Hammer2.0-0.5b](#) (the *teacher*), grades every token of them.

Dataset

The dataset for this homework is the [xLAM Function Calling Dataset](#) from HuggingFace. It contains 60,000 function-calling examples, each with a `query` (the user's request), a JSON list of the `tools` on offer (each with a name, a description and its parameters, some of which have a default), and the gold answers (a JSON list of calls, each `{"name": ..., "arguments": {...}}`). Here is a real training row, where the model has to pick the right tool out of two:

```
query: What is the binary sum of 10010 and 11101?
tools: [{"name": "binary_addition",
        "description": "Adds two binary numbers and returns the result as a binary string.",
        "parameters": {
            "a": {"description": "The first binary number.", "type": "str"},
            "b": {"description": "The second binary number.", "type": "str"}
        }}, {"name": "create_histogram",
        "description": "Create a histogram based on provided data.",
        "parameters": {
            "data": {"description": "...", "type": "list", "default": 10},
            "bins": {"description": "...", "type": "int", "default": 10}
        }}]
answers: [{"name": "binary_addition", "arguments": {"a": "10010", "b": "11101"}}]
```

Answers can hold several calls (several tools, or one tool called repeatedly). The scorer treats an argument given at its documented default the same as one left out, so `bins=10` above is optional.

A model that only ever sees answerable requests learns to always call *something*. To teach it when *not* to, the xLAM rows are mixed with the 7,500 rows of the [xLAM Irrelevance Dataset](#), whose requests cannot be served by the tools on offer and whose correct answer is the empty list `[]` (abstaining). For example:

```
query: Could you tell me the city for ZIP 20001 and analyze building 9012 in static mode for floors 2 and 4?
tools: [{"name": "get_holidays_by_year",
        "description": "Retrieves the list of public holidays for a given year ...",
        ...}]
answers: []
```

Each row becomes one training sequence: a **prompt** built from `tools` and `query`, followed by a **completion** built from `answers` (see `prompts.py`). The student's prompt is a chat whose system message lists the tools and states the output contract (one fenced JSON array, omit arguments left at their defaults, `[]` if no tool applies), followed by the query as the user message. The completion is the gold answer, fenced, ending with the end-of-turn token:

```

<|im_start|>system
You are a helpful assistant that performs the appropriate function call from the given set
of functions ...
The tools available to you are:

[{"name": "binary_addition", ...}, {"name": "create_histogram", ...}]

Respond with a single JSON array of calls,
each {"name": <tool name>, "arguments": {"arg": <value>}}.
...
If no tool applies, respond with [].
Respond with the fenced JSON only, no other text.<|im_end|>
<|im_start|>user
What is the binary sum of 10010 and 11101?<|im_end|>
<|im_start|>assistant
```
[{"name": "binary_addition", "arguments": {"a": "10010", "b": "11101"}}]
```<|im_end|>

```

The teacher's prompt uses its own format, the one it was trained on: the same tools and query, in [BEGIN OF TASK INSTRUCTION] / [BEGIN OF AVAILABLE TOOLS] / [BEGIN OF QUERY] blocks. It is longer (441 tokens against the student's 312 for this row), but the completion is the same tokens either way.

All 67,500 rows are tokenized, and the rows are selected by token length (see `data.py`):

	training (train_rows)	evaluation (eval_rows)
rows	2,000, drawn at random	the 1,000 longest
length	at most 1,023 tokens (median 439)	1,136 to 2,824 tokens
tools offered per row	2.5 on average	5.1 on average
abstention rows	248 (12%)	35 (3.5%)

The two sets never overlap: every evaluation row is longer than any training row. Longer rows offer more tools, so the evaluation set is also harder: the model has to pick and fill the right calls from about five candidates instead of two or three.

This dataset will download automatically when you run `data.py`, `evaluate.py`, `train_sft.py` or `train_opd.py`. Run `python src/data.py` to see a real training row, its rendered prompt and its completion (the first run tokenizes the full dataset, which takes a few minutes on Colab). Note: the xLAM dataset is gated, so you must accept its terms on its HuggingFace page and log in with a HuggingFace token first.

Starter Code

The main structure of the files is organized as follows:

```

hw3/
  hw3.ipynb
  pyproject.toml
  uv.lock
  src/
    generate.py
    lora.py
    losses.py
    train_opd.py

```

```
train_sft.py
evaluate.py
model.py
data.py
prompts.py
metrics.py
config.py
engine/
test_finetune.py
test_inputs.npz
phigrade.yaml
pytest.ini
```

Here is what you will find in each file:

1. `hw3.ipynb`: Provides the commands to run each part of the assignment, in order, in Google Colab. All instructions are in this writeup.
2. `pyproject.toml` and `uv.lock`: A list of packages that need to be installed for this homework, including the `phigrade` client that you will need in order to submit. If you work locally, install them by running `uv sync` from the handout directory, activate the environment with `source .venv/bin/activate`, and log in to HuggingFace with `hf auth login`. To use the notebook locally, launch your notebook environment from that shell, select the `.venv` interpreter, and skip the cells marked *Colab only*.
3. `src/generate.py`: The script for generating text with your trained (or raw) Qwen2 model. The generation loop is provided. Implement `sample` in this. **SUBMITTED via PhiGrade**
4. `src/lora.py`: Implement LoRA in this. Some starter code is provided to guide you. Only implement LoRA in a linear layer, and use your implemented LoRA in the MLP layers. **SUBMITTED via PhiGrade**
5. `src/losses.py`: The two training losses. Implement the SFT loss and the reverse KL in this. **SUBMITTED via PhiGrade**
6. `src/train_opd.py`: The script for training with on-policy distillation. This file is long but your only requirement is to implement the marked blocks of the training loop. **SUBMITTED via PhiGrade**
7. `src/train_sft.py`: The script for training with supervised fine-tuning.
8. `src/evaluate.py`: The script for evaluating your trained (or raw) model on the evaluation set.
9. `src/model.py`: A working implementation of the Qwen2 transformer (i.e. with GQA and RoPE, as in HW1). You should be familiar with this file, since LoRA plugs into its attention and MLP layers.
10. `src/data.py`: A custom dataloader implemented for the xLAM dataset.
11. `src/prompts.py`: The prompt templates for the student and the teacher.
12. `src/metrics.py`: The scorer used by `evaluate.py`.
13. `src/config.py`: The model names and the model configuration.

14. `src/engine/`: Helper code (KV cache, decoding, mixed precision, optimizer, checkpointing, plotting) that makes this homework efficient enough to run on a T4. You do not need to read it.
15. `test_finetune.py` and `test_inputs.npz`: The unit tests that make up the autograder, and their inputs. Run them by executing `pytest test_finetune.py` from the handout directory. Doing so both scores your code and submits it.
16. `phigrade.yaml` and `pytest.ini`: Configuration for the autograder.

(We also included some IDE specific settings files `{.noai, .vscode, .zed}` to help you avoid AI use during Slot A.)

Flags

All the parameters of a run can be modified by passing flags to `train_sft.py`, `train_opd.py` and `evaluate.py`. Table 1, Table 2 and Table 3 contain a list of flags you may find useful while implementing HW3. You can change other parameters as well in a similar manner.

Configuration Parameter	Example Flag Usage
<code>out</code>	<code>--out=checkpoints/warm.pt</code>
<code>init_from</code>	<code>--init-from=checkpoints/warm.pt</code>
<code>rank</code>	<code>--rank=16</code>
<code>alpha</code>	<code>--alpha=32</code>
<code>epochs</code>	<code>--epochs=1</code>
<code>n</code>	<code>--n=2000</code>

Table 1: Useful flags for `train_sft.py`

Configuration Parameter	Example Flag Usage
<code>out</code>	<code>--out=checkpoints/opd.pt</code>
<code>init_from</code>	<code>--init-from=checkpoints/warm.pt</code>
<code>epochs</code>	<code>--epochs=5</code>
<code>n</code>	<code>--n=32</code>
<code>max_new_tokens</code>	<code>--max-new-tokens=192</code>
<code>temperature</code>	<code>--temperature=1.0</code>
<code>save_every_epoch</code>	<code>--save-every-epoch</code>

Table 2: Useful flags for `train_opd.py`

Configuration Parameter	Example Flag Usage
<code>checkpoint</code>	<code>--checkpoint=checkpoints/warm.pt</code>
<code>teacher</code>	<code>--teacher</code>
<code>n</code>	<code>--n=1000</code>
<code>show</code>	<code>--show=5</code>

Table 3: Useful flags for `evaluate.py`

There are more parameters available to modify (see `train_sft.py`, `train_opd.py` and `evaluate.py`), but we don't expect that you will need to modify more than the ones mentioned above.

Command Line

Colab provides a free T4 GPU for code execution, albeit with a time limitation that may result in slower training. In the event of GPU depletion on Colab, options include waiting for GPU recovery, switching Google accounts, purchasing additional GPU resources, switching to Kaggle, or switching to a cloud provider (such as GCP or AWS). The full pipeline takes about 2 hours on a T4.

The notebook `hw3.ipynb` runs the full pipeline in order; you can also copy its commands into a terminal. The pipeline has three training phases, each followed by an evaluation (`python src/evaluate.py --checkpoint ...`):

1. **Warm start:** one epoch of SFT on the 2,000 training prompts, which saves the LoRA adapter to `checkpoints/warm.pt`.

```
python src/train_sft.py --out checkpoints/warm.pt --epochs 1
```

2. **SFT control:** 5 more epochs of SFT on the same 2,000 examples, starting from the warm start. The `--init-from` flag *fuses* the warm start's adapter into the base weights and trains a **fresh** adapter on top.

```
python src/train_sft.py --init-from checkpoints/warm.pt \  
--out checkpoints/csft.pt --epochs 5
```

3. **OPD:** 5 epochs of on-policy distillation from the same starting point (the warm start fused in, a fresh adapter, the same 2,000 prompts). Run a short smoke test first (`--n 32 --epochs 1`). `--save-every-epoch` also keeps the adapter after each epoch.

```
python src/train_opd.py --init-from checkpoints/warm.pt \  
--out checkpoints/opd.pt --save-every-epoch --epochs 5
```

Each training run also saves its loss curve next to its checkpoint (e.g. `checkpoints/warm.png`), and each evaluation saves its scores (e.g. `checkpoints/warm.eval.json`).

Low-Rank Adaptation (LoRA) of LLMs

In this problem, you will implement Low-Rank Adaptation (LoRA), following the approach outlined in (Hu et al., 2021). Before you continue, we strongly recommend you to go through the paper and understand how LoRA works.

Models can continue to learn efficiently even when their parameters are projected onto a smaller subspace. Essentially, this means that the vast majority of the model's capabilities can be retained and modified through adjustments in a significantly reduced parameter space. This allows for us to inject trainable low-rank decomposition matrices into each layer of the Transformer architecture, greatly reducing the number of trainable parameters for downstream tasks.

For a pretrained weight matrix $W_0 \in \mathbb{R}^{d \times k}$, LoRA constrains its update through a low-rank decomposition, expressed as follows:

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

where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and the rank $r \ll \min(d, k)$. During the adaptation process, W_0 remains unchanged—frozen—to ensure the stability of the pre-trained knowledge, while A and B are updated, serving as the trainable parameters. Note that that we achieve this by setting `requires_grad = False` for all parameters except the matrices A and B .

We then apply both W_0 and the adjustment $\Delta W = BA$ to the same input x , with their outputs being summed coordinate-wise, resulting in the modified forward pass:

$$h = W_0x + \Delta Wx = W_0x + BAx$$

As depicted in Figure 1, our initial conditions for training involve setting A with a Kaiming Uniform initialization and B to zero, making $\Delta W = BA$ start from zero. To integrate these updates effectively, remember to scale ΔWx by α/r , with α acting as a constant relative to r . This approach simplifies the optimization process, akin to adjusting the learning rate in Adam, and eliminates the need for hyperparameter retuning as r varies. You can start with setting α to twice the initial value of r you explore, and experiment with different scaling factors(α/r) by adjusting α and r accordingly. Thus, the scaled LoRA forward pass you should implement is:

$$h = W_0x + \frac{\alpha}{r}\Delta Wx = W_0x + \frac{\alpha}{r}BAx$$

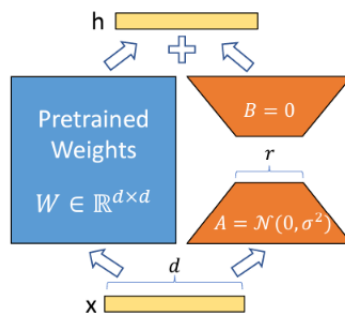


Figure 1: Low-Rank Adaptation (LoRA) applied to a Transformer model.

Supervised Fine-Tuning

As in Homework 1 with the Shakespeare dataset, when given a text, Qwen2.5 (or any LLM for that matter) generates more text to complete the given text. But for a task like tool calling, how do you get the model to generate the calls you want for the given context?

Enter Supervised Fine-Tuning. Supervised fine-tuning is a specialized form of fine-tuning in which a model is trained using prompt-completion pairs. It helps bridge the gap between the next-word prediction objective of LLMs and the our objective of having LLMs adhere to human instructions.

For the purpose of this homework, this is done by prepending each sample in the xLAM dataset with a prompt template that lists the tools and states the query, and then appending it with the gold answer (the completion). This modified text is what you will train the model with. This happens in the `student_prompt()` and `completion()` functions in `prompts.py`. Only the completion carries a loss: in a batch, `labels` holds the completion's token ids and `-100` everywhere else, including the prompt and the padding. The SFT loss is the mean next-token cross-entropy over the completion.

On-Policy Distillation

SFT on a fixed set of 2,000 answers stops improving within a few epochs; more epochs only memorize the answers. In on-policy distillation, the student instead samples its own completion for each prompt, and the teacher grades every token of it. At every position of a completion, the student and the teacher each give a distribution over the next token, p_s and p_t . OPD minimizes the *reverse* KL from the student to the teacher, averaged over positions:

$$\text{KL}(p_s \parallel p_t) = \sum_v p_s(v) (\log p_s(v) - \log p_t(v))$$

Because it is weighted by the student's own probabilities, it is *mode-seeking*: it pushes down whatever the student puts mass on that the teacher does not, instead of spreading the student over every token the teacher might say. OPD draws a fresh sample from the current student every epoch, so the data keeps changing as the student improves.

The teacher reads the student's completions behind *its own* prompt format (see `teacher_prompt()` in `prompts.py`). The two prompts differ in length, but the completion tokens are identical, so the positions line up one to one: only completion positions carry a loss.

Implementation

Note: In the original paper, LoRA has been implemented only in the attention layers, specifically for the query and value matrices. In this homework you will implement LoRA on the query, key, value and output matrices of the attention layers, and on the three linear projections of the MLP layers.

Sampling:

- In `generate.py`, implement `sample`, which returns one token id per row from the logits. A temperature of 0 means greedy decoding. Otherwise, convert the logits to `torch.float32`, divide by the temperature, apply softmax, and draw with `torch.multinomial`.
- All of `generate` is complete. However, you should read this loop to see how it uses your sampler.

The LoRA Linear Layer:

- In `lora.py`, implement these modifications in the `LoRALinear` class. This includes:
 - `__init__`: Initialize inherited `nn.Linear` class, LoRA parameters, and matrices A and B . Freeze the weight and the bias (if present). A is typically initialized with `kaiming_uniform_` and B is initialized with zeroes according to the paper. Both should be created in `torch.float32`, even when the base weight is in `torch.float16`: they are the only trained weights, and an fp16 update of $\sim 1e-7$ against a $\sim 1e-2$ weight would round to nothing.
 - `forward`: Implement the forward pass of the layer, including the application of LoRA modifications. Compute the update in `torch.float32` and cast it to the dtype of the base output before adding.
 - `to_linear`: Ensure that LoRA matrices are merged with the actual model weight matrices, returning a new `nn.Linear`.

LoRA for Transformer LMs:

- In `apply_lora`, set only LoRA matrices as trainable parameters for a model, and apply your above implemented `LoRALinear` layer to the MLP layers within your transformer model. The attention layers are provided as an example. This is marked with TODOs in `lora.py`.

Supervised Fine-Tuning Method

- All the methods in `data.py` and `prompts.py` are complete.
- Take note of what `completion_positions(hidden, labels)` in `losses.py` is doing. Position t predicts token $t + 1$, so the function drops the last hidden state and the first label, and then returns only the hidden states that predict a completion token, along with those tokens.
- In `losses.py`, implement `sft_loss`, which returns the mean next-token cross-entropy over the completion and the number of tokens. Unembed only the selected positions, and compute the cross-entropy in `torch.float32`.

On-Policy Distillation Method

- In `losses.py`, implement `reverse_kl`, which returns the reverse KL averaged over positions, given student and teacher logits for the same positions.
- **Details:** The teacher's vocabulary is 151,665 tokens; the student's is padded to 151,936 with ids that never occur. Compare only the shared ids. Note that `F.kl_div(input, target, log_target=True)` computes $KL(target \parallel input)$, so pay attention to which argument is which.

Training:

- Now that you have made your Qwen2 model lora friendly, modify `train_opd.py` to enable training with on-policy distillation. The student samples a completion for each prompt in the batch at temperature 1 with `generate` (with the student in `eval()` mode, then back to `train()`), the teacher reads those completions with no gradients, and the student reads the same completions with gradients, as indicated by the relevant TODOs.

Accuracy Evaluation Method

- In `evaluate.py` the provided code iterates through the evaluation set, constructs the prompt for each example, decodes greedily, and scores the response of the model against the gold answer.
- **Details:** A completion scores **exact match** only if it is one fenced JSON array whose calls match the gold answer. **Argument F1** measures how many of the arguments of the calls match. **Abstention** is the share of rows whose gold answer is `[]` where the model correctly answered `[]`, and **false abstain** is the share of answerable rows where it wrongly did.
- Garbage generations are accounted for: if the model predicts something that does not parse, it is not omitted when calculating accuracy and scores zero on everything.

Hints

1. While implementing your code, you may find it help to adjust the number of training prompts, e.g. `--n 32` is enough for a short smoke test, but you will need the full 2,000 prompts to see decent results from fine-tuning with LORA.
2. When trying different variations (across `r`, `alpha`, etc) it is recommended you use the `--out` flag so you can save the different models you create.
3. If you are facing errors, run `python src/generate.py` and `python src/lora.py` to isolate errors better. These self-tests check your sampler and your LoRA implementation on the real model before any training. The `generate.py` self-test decodes an answer greedily, which should say that Paris is the capital of France. The `lora.py` self-test applies LoRA to the real model, verifies that the logits are unchanged (which is exactly what the zero initialization of B guarantees), and reports the trainable parameter count.
4. There is no separate self-test for `sft_loss`. The first logged steps of the warm start should show a loss that starts somewhere around 0.1–0.5 and falls quickly (below 0.1 within a few dozen steps). A loss that is huge, NaN, or stuck indicates a bug.
5. For OPD, the smoke test (`--n 32 --epochs 1`, a few minutes) should show a reverse KL that is small and positive (roughly 0.02–0.06), with no assertion errors.
6. Keep the Colab tab open while jobs run, since Colab reclaims idle runtimes. The notebook can also keep your checkpoints on Google Drive.

LoRA Implementation and Training

Note: For all the empirical questions report results on the 1,000 evaluation rows. If not specified, return results using default parameters (i.e with $r = 16$, $\alpha = 32$, 2,000 training prompts, and learning_rate = $2e-4$ for SFT and $1e-4$ for OPD).

- 5.1. (2 points) What percentage of parameters are fine-tuned with when you set $r = 16$ and $\alpha = 32$?

Inference and Evaluation with LoRA

- 5.2. (1 point) What is the exact match of your student model without any fine-tuning? (Hint: you can run this directly using `python src/evaluate.py`) *Note: In this and all subsequent questions, be sure to report the “exact match” as a percentage of the 1,000 evaluation rows.* [Expected runtime on Colab T4: 7-10 minutes]

- 5.3. (4 points) What is the exact match and argument F1 with LoRA fine-tuning (one epoch of SFT) across $(r, \alpha) \in \{(16, 32), (8, 16), (16, 16)\}$? What is the exact match and argument F1 of the teacher i.e. without any training? (Hint: you can run this directly using `python src/evaluate.py --teacher`) In this question, the first two settings maintain a constant scaling factor of 2, i.e. $\alpha = 2r$.

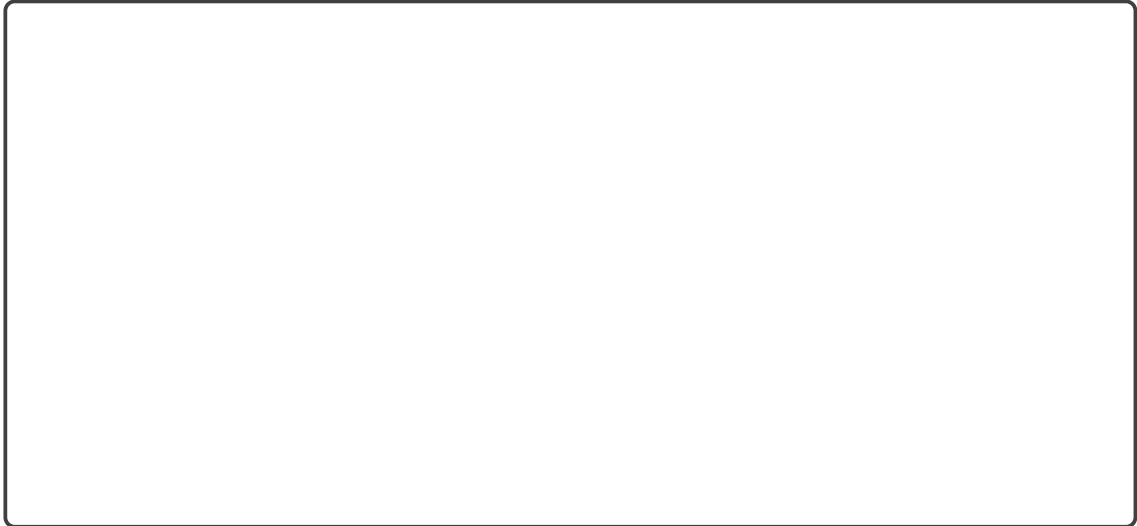
[Expected runtime on Colab T4: 15-25 minutes per experiment]

method	r	alpha	exact match	argument F1
LoRA	16	32		
LoRA	8	16		
LoRA	16	16		
Teacher	–	–		

- 5.4. (4 points) Plot training loss curves for LoRA experiments with

$$(r, \alpha) \in \{(16, 32), (8, 16), (16, 16)\}.$$

(Hint: the notebook saves this plot as `checkpoints/lora_losses.png`.)



- 5.5. (3 points) How does your fine-tuning with LoRA model's performance compare to the model without any fine-tuning? Also, how do the values of r and α affect performance? Briefly discuss (include comments on convergence analysis of training loss).



Comparing Training Methods

- 5.6. (2 points) Now, we are going to use the model trained in the last part with LoRA r 16 and α 32, as warm start for the next phase. Is there anything unexpected about the shape of the training loss when you continue SFT from the warm start for 5 more epochs (the SFT control)? If yes, explain what is unexpected. If no, describe why it appears typical. Do you think it is useful to continue SFT beyond this point or should we train with a different objective?

[Expected runtime on Colab T4: 15-20 minutes for the new setting]



- 5.7. (2 points) Report the exact match and plot loss curves for the SFT control and for OPD, each trained for 5 epochs from the warm start. (Hint: the scripts save these plots as `checkpoints/csft.png` and `checkpoints/opd.png`.) *Fill in the table with the exact match values.*

[Expected runtime on Colab T4: 60-80 minutes for the new setting]

method	r	alpha	exact match
Warm Start	16	32	
SFT Control	16	32	
OPD	16	32	

loss plots

- 5.8. (1 point) In your results from the previous question, how did training on the student's own samples with OPD, rather than on the gold answers with SFT, affect the performance of the model? Why might this be the case?

- 5.9. (1 point) Changing both the learning rate and α may be redundant. Why?

- 5.10. (1 point) Your `reverse_kl()` function in `losses.py` is either computing $\text{KL}(student||teacher)$ or $\text{KL}(teacher||student)$. Which one is it computing and why is that the correct $\text{KL}(\cdot)$ divergence for this setting?

- 5.11. (2 points) For the default setting $r = 16$, $\alpha = 32$, report the abstention and false abstain of the student without any fine-tuning, the warm start, the SFT control, OPD, and the teacher. (Note that you do *not* need to run any new experiments.) *Add these values to the following table:*

method	abstention	false abstain
Untrained Student		
Warm Start		
SFT Control		
OPD		
Teacher		

6 Code Upload (0 points)

You submit your code with the [PhiGrade](#) autograder, which you run from your own machine. There is no separate “submit” step and no file upload: running the autograder *is* how you submit.

One-time setup The `phigrade` client is included in `pyproject.toml`. Create an API key on your account page at <https://phigrade.mlcourse.org>, then run the command below **from the handout directory** (the one containing `phigrade.yaml`) and paste the key when prompted. Do not share your key or check it into version control.

```
phigrade auth login
```

Submitting From the handout directory, run:

```
pytest test_finetune.py
```

This scores your code and uploads `src/generate.py`, `src/lora.py`, `src/losses.py` and `src/train_opd.py`, the only files you submit for this homework. You will be asked to choose a submission slot: *Slot A* (for human work only) or *Slot B* (for human or AI work). Choose the slot that reflects how you actually completed the assignment; see the syllabus for what is permitted in each.

Note: Every run of `pytest test_finetune.py` is recorded as a submission, and your most recent run before the deadline is the one that counts.

6.1. (0 points) Did you upload your code to the appropriate programming slot on PhiGrade?
Hint: The correct answer is ‘yes’.

- ☐ Yes
- ☐ No

7 Collaboration Questions (2 points)

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found in the syllabus.

- 7.1. (1 point) Did you collaborate with anyone on this assignment? If so, list their name or Andrew ID and which problems you worked together on.

- 7.2. (1 point) Did you find or come across code that implements any part of this assignment? If so, include full details.