CMSC 25700/35100 Winter 2026 Assignment 4 SFT and Sampling

Due: 11:59 PM CST, Friday, February 20, 2026
This assignment focuses on **Supervised Fine-Tuning (SFT)** and **advanced sampling techniques** to improve the performance of Large Language Models (LLMs) on reasoning tasks. Building on the prompting techniques explored in Assignment 3, you will now update model parameters by training on domain-specific data to improve task performance. Furthermore, moving beyond greedy decoding, you will also explore how generating multiple samples and selecting the best one based on a reward model [3] or through majority voting [4] can further improve models' reasoning performance. Here is a quick summary:

1. **Supervised Fine-Tuning for Math Reasoning:** You will fine-tune two Gemma 3 models on the GSM-Symbolic [2] math reasoning dataset. This involves implementing the data encoding and dataset preparation pipeline (coding), setting up the HuggingFace Trainer (coding), analyzing training arguments and configurations (written), and evaluating fine-tuned vs. base models on multiple test sets to study the effects of SFT on generalization (written).

2. **Inference-Time Scaling with Majority Voting and Best-of-N:** You will implement two inference-time scaling strategies–Majority Voting and Best-of-N (BoN) sampling–to improve reasoning accuracy beyond what SFT alone achieves. Using both the base and SFT'ed Gemma 3 1B models, you will generate multiple candidate responses per question and select the final answer via majority vote or a reward model (coding). You will then analyze how accuracy scales with the number of samples, compare the effectiveness of both strategies across base and SFT models, and examine failure modes using reasoning trace recovery metrics (written).

Submission Instructions. There are two types of questions in this assignment. Each question is annotated as either "written" or "coding" at the beginning of its specification.
For all the **written** questions, we have provided a latex-based report template. You should typeset and render your answer in the provided `\ifans` command or the `solutionblock` environment after each question. The final compiled **PDF** report should be submitted to **`Assignment 4 - Written`** on GradeScope.
For all the **coding** questions, you should build your work on top of the provided source code by completing all the `TODOs` according to the specification of each question. After that, you should execute `a4_sft_dist/test_a4_q1.py` and `a4_sft_dist/test_a4_q2.py` to generate `A4-Q1.json` and `A4-Q2.json` respectively, similar to the practice in Assignment 3.
You should then zip together `a4_sft_dist/` and `a4_sampling_dist/`, which contain both your completed source code and the newly generated .json files, into **a single .zip file**. This .zip file should be submitted to **`Assignment 4 - Code`** on GradeScope. NOTE that **the source code and data should be organized in the same file structure as they are originally distributed, and the json file should remain in its default location after it is generated.** The submitted .zip file should have the following structure:

```
A4.zip
├── a4_sft_dist/
│   ├── data/
│   ├── scripts/
│   ├── data_utils.py (TODOs completed)
│   ├── train.py (TODOs completed)
│   ├── training_arguments.py
│   ├── eval.py
│   ├── eval_utils.py
│   ├── modal_app.py
│   ├── download_modal_volumes.py
│   ├── test_a4_q1.py
│   └── A4-Q1.json (generated)
└── a4_sampling_dist/
    ├── data/
    ├── results/
    ├── scripts/
    ├── utils.py
    ├── generate_samples.py (TODOs completed)
    ├── evaluate.py (TODOs completed)
    ├── analyze_traces.py (TODOs completed)
    ├── modal_app.py
    ├── test_a4_q2.py
    └── A4-Q2.json (generated)
```

Important: Do not include any additional folder layer (e.g., `A4/`) between the `.zip` root and the two `_dist` folders. The two folders should be at the top level when the zip is extracted.

GPU Usage Instructions. For this assignment, you will use Modal to run training and evaluation on cloud GPUs. Follow the steps below to set up your Modal environment:

1. **Create a Modal account** at `https://modal.com` and redeem the $500 GPU credit provided for this course according to the code posted on Ed, so that you will have $530 in total.
2. **Set up the Modal CLI.** Before running any Modal jobs, execute `scripts/setup_modal.sh` from the `a4_sft_dist/` directory to authenticate the Modal CLI and create Modal Secrets for Weights & Biases and HuggingFace API tokens.
3. **Modal Volumes for persistent storage.** Throughout this assignment, all training checkpoints and evaluation results are saved on Modal Volumes, a persistent cloud storage that can be conveniently accessed using the Modal Web Dashboard. Contents stored on Modal Volumes will remain intact even after Modal instances are terminated, can be shared across different Modal images/runs, and can be easily downloaded to local machines.

Explore the official Modal documentation for more details about Modal Volumes and other Modal functionalities:

- `https://modal.com/docs/guide/images`
- `https://modal.com/docs/guide/gpu`
- `https://modal.com/docs/guide/resources`
- `https://modal.com/docs/guide/volumes`

- `https://modal.com/docs/guide/model-weights`
- `https://modal.com/pricing`

1. Supervised Fine-Tuning for Math Reasoning (60 points)

In the previous assignment, we explored how prompting techniques can steer the behavior of pre-trained LLMs *without* updating their parameters. In this question, we take the next step: **Supervised Fine-Tuning (SFT)**, where we update a pre-trained model's parameters on a domain-specific dataset so that it learns to produce better responses for a target task.

Specifically, you will fine-tune two Gemma 3 models on the GSM-Symbolic [2] math reasoning dataset (training split), and then evaluate the SFT'ed models against their base counterparts on multiple test sets to analyze the effects of SFT. You will implement the data processing pipeline for SFT, set up the HuggingFace `Trainer`, and run training and evaluation on cloud GPUs via Modal.

All coding efforts in this section use `a4_sft_dist/` as the root directory.

Task Description and Setup

Overview

You will fine-tune `gemma-3-1b-it` and `gemma-3-4b-it` on a training set of 4500 GSM-Symbolic examples using the HuggingFace `Trainer` API. After training, you will evaluate **four** models–the two base (pre-SFT) models and the two fine-tuned models–on **three** evaluation datasets (100 examples each) to study how SFT affects performance across different data distributions.

Clarification on Term Use: Throughout this handout, we refer to `gemma-3-1b-it` and `gemma-3-4b-it` as "base models," even though they have already undergone some closed-source post-training (e.g., instruction tuning and preference tuning) by Google internally. Here, "base" simply means the model *before* the SFT that you will perform in this assignment.

You need to set up the following accounts and services before completing your assignment:

1. You need to register a **HuggingFace** account, review and accept Google's usage license on the homepages of `gemma-3-1b-it` and `gemma-3-4b-it`, and create a User Access Token in order to download and use these models.
2. You need to register a **Wandb (Weights & Biases)** account, and create a Wandb API key following the documentations here. Wandb will be useful for experimental logging and visualization.
3. After completing the two steps above, you need to execute `scripts/setup_modal.sh` to authenticate the Modal CLI and create Modal Secrets for Wandb and HuggingFace API tokens.

Data Files

All data files are provided in the `data/` folder:

- `gsm_symbolic_train_4500_student.jsonl` — Training set (4500 examples from GSM-Symbolic)
- `gsm_symbolic_test_100_student.jsonl` — GSM-Symbolic test set (100 examples; the same test set as used in A3-Q3)
- `gsm8k_test_100_same_templates.jsonl` — A subset of GSM8K [1] test examples that share the same question templates as GSM-Symbolic, but are uniformly one sentence shorter in question length

- `gsm8k_test_100_non_overlapping.jsonl` — A subset of GSM8K test examples whose question templates do *not* overlap with GSM-Symbolic at all (also one sentence shorter)

The three test sets are designed to probe different aspects of generalization after SFT. Detailed descriptions of their similarities and differences are provided in the evaluation section below.

Codebase

The codebase reuses several utilities from A3-Q3, including `build_gsm_prompt` (for constructing chain-of-thought prompts) and answer extraction functions in `eval_utils.py`. The main files you will work with are:

- `data_utils.py` — Data processing pipeline for SFT (encoding and dataset preparation)
- `train.py` — Training script that loads the model, prepares data, and runs the HuggingFace `Trainer`
- `training_arguments.py` — Dataclass definitions for model, data, and training arguments
- `scripts/run_sft_train.sh` — Shell script that launches SFT training with `torchrun`
- `modal_app.py` — Modal application for running training and evaluation on cloud GPUs

Please carefully read the provided `README.md` file for a detailed walkthrough on how to get this codebase running on Modal. It usually takes less than 2 hours to complete all the training and evaluations for the whole section, using the default Modal configurations that we provide.

Coding: Data Processing for SFT (36 points)

(a) (24 points) **[Coding]** Complete the `encode_function` in `data_utils.py`. This function encodes a batch of question-answer pairs into tokenized training examples for SFT. For each (question, answer) pair, you need to build the prompt and completion texts, tokenize them, and create label masks so the model is only trained on the completion (assistant) tokens.

Follow the high-level workflow below for each example in the batch:

1. **Build the prompt and completion texts.** Several helper functions and constants are provided at the top of `data_utils.py` for constructing the prompt and completion strings. Read the docstrings and source code of these helpers carefully to understand how to use them. You will also need to apply the tokenizer's chat template (with appropriate arguments) and append relevant special tokens to form the complete training sequence.
2. **Tokenize.** Tokenize both the prompt portion and the full sequence (prompt + completion) separately, without adding special tokens (since the chat template already includes them). If a maximum sequence length is specified, ensure that tokenization respects this limit.
3. **Validate alignment.** Verify that the tokenized prompt is a prefix of the tokenized full sequence. If this does not hold, raise a `ValueError` — prefix alignment is essential for correct label masking.
4. **Construct the output.** For each example, produce three parallel lists:
   - `input_ids`: the token IDs of the full sequence (prompt + completion).
   - `labels`: a sequence of the same length as `input_ids`, where positions corresponding to the *prompt* are masked with -100 (so the loss ignores them), and positions corresponding to the *completion* contain the actual token IDs that the model should learn to predict.
   - `attention_mask`: indicates which positions are valid tokens (as opposed to padding).

Refer to the detailed docstring inside `encode_function` for additional guidance, including a concrete example of the expected prompt and completion texts. Below is a high-level illustration of how the prompt and completion are structured:

```
// prompt_text (context -- NOT trained on):
<bos><start_of_turn>user
Solve the following math problem step by step, and provide the final numeric answer.

Problem: What is 2 + 3?
<end_of_turn>
<start_of_turn>model
Answer: Let's think step by step.

// completion_text (trained to generate; pay attention to the single whitespace at the very
beginning):
 2 + 3 = 5.
Final answer: 5.<end_of_turn>
<eos>
```

Learn more about the differences among `input_ids`, `labels`, and `attention_mask` via the following Huggingface documentations:

- https://huggingface.co/docs/transformers/glossary
- https://huggingface.co/learn/llm-course/chapter3/2

(b) (12 points) **[Coding]** Complete the `load_and_prepare_train_dataset` function in `data_utils.py`. The loading, shuffling, and percentage-based subsampling of the raw dataset are already implemented. You need to complete the remaining two steps:

1. **Apply `encode_function`** to the raw dataset using `Dataset.map()` in batched mode. Note that `encode_function` requires additional arguments beyond what `Dataset.map()` passes by default — you will need to find a way to bind these extra arguments before passing the function as a callable. Check `training_arguments.py` → `DataArguments` to determine which attributes of `cfg` correspond to the keyword arguments of `Dataset.map()` (e.g., number of parallel workers, cache behavior). Also ensure that only the three output columns (`input_ids`, `labels`, `attention_mask`) remain in the resulting dataset.
2. **Set the dataset format** to PyTorch tensors for the columns: `input_ids`, `labels`, `attention_mask`.

You may find the following documentations about Huggingface `datasets` APIs useful:

- https://huggingface.co/docs/datasets/process
- https://huggingface.co/learn/llm-course/chapter3/2

(c) (0 points) **[Coding]** Set up the `DataCollatorForSeq2Seq` and `Trainer` in `train.py`. Complete the TODO section (around lines 109–116) by creating:

1. A `DataCollatorForSeq2Seq` from the `transformers` package. This collator dynamically pads batches of variable-length sequences during training. Refer to the HuggingFace documentation to determine the appropriate constructor arguments based on the variables already defined earlier in `train.py`.
2. A `Trainer` from the `transformers` package, initialized with the relevant components (model, training arguments, dataset, tokenizer, and data collator) that have been set up in the preceding code.

This part carries 0 points; its correctness is validated by successfully running the SFT training in a later step.

You may learn something useful about the Huggingface `transformers.Trainer` API from the following documentations:

- https://huggingface.co/docs/transformers/main_classes/trainer
- https://huggingface.co/learn/llm-course/chapter3/3
- https://huggingface.co/learn/llm-course/chapter7/6

Written: Training Arguments Analysis (10 points)

(a) (10 points) **[Written]** Read `scripts/run_sft_train.sh` and `train.py` carefully, and answer the following questions about the training configuration. You are encouraged to consult the relevant HuggingFace Transformers documentation (e.g., `TrainingArguments`, `Trainer`) and PyTorch distributed training documentation.

i. (3 points) What are the effects of setting `--bf16 True` and `--torch_dtype bfloat16` respectively, and what is their difference?

ii. (3 points) Currently, the script uses `torchrun --nproc_per_node 1 --nnodes 1`. What would happen if we instead set `--nproc_per_node 4 --nnodes 1`, assuming we have enough GPU resources to support this configuration?

iii. (4 points) What is the difference between `per_device_train_batch_size` and `effective_bsz`? What is the general formula for calculating `effective_bsz` from `per_device_train_batch_size`, `gradient_accumulation_steps`, `nproc_per_node`, and `nnodes`, and why?

Written: SFT Evaluation and Analysis (14 points)

After completing all the coding parts above, run **`modal run modal_app.py::launch_all`** to launch training and evaluation jobs on Modal GPUs. By default, this will SFT both `gemma-3-1b-it` and `gemma-3-4b-it` on the GSM-Symbolic training set (4500 examples), and then evaluate **four models** (both SFT'ed and base models of each architecture) on **three evaluation datasets** (100 examples each):

1. **gsm-symbolic**: The native GSM-Symbolic test set, the same test set as used in A3-Q3.
2. **gsm8k-same_templates**: A subset of GSM8K [1] test examples that share the same question templates as GSM-Symbolic. However, these questions are uniformly one sentence shorter in length compared to their GSM-Symbolic counterparts, thus having fewer mathematical conditions.
3. **gsm8k-non_overlapping**: A subset of GSM8K test examples whose question templates do *not* overlap with GSM-Symbolic at all. These questions are also one sentence shorter, as this is a systematic difference between GSM8K and the GSM-Symbolic split used in this assignment.

(a) (4 points) **[Written]** Report the accuracy of all four models on all three test datasets in Table 1. Use the percentage scale (i.e., numbers on a range from 0 to 100 without the percentage unit).

Model\Test Set	**gsm-symbolic**	**gsm8k-same_templates**	**gsm8k-non_overlapping**
`gemma-3-1b-it`			
`gemma-3-1b-it-SFT`			
`gemma-3-4b-it`			
`gemma-3-4b-it-SFT`			

Table 1: Accuracy of base and SFT'ed models on three evaluation datasets.

(b) (10 points) **[Written]** Based on your results in Table 1, answer the following analysis questions:

i. (3 points) For each test dataset (i.e., each column in the table), compare the SFT'ed models with their base model counterparts. What performance trends do you observe? Which dataset benefits the most from SFT, and which suffers the most performance degradation (if any)?

ii. (3 points) Why are there different performance trends on different datasets after SFT? Analyze by comparing each test dataset with the training dataset.

iii. (4 points) Download and paste the training loss curve for `gemma-3-1b-it` on the GSM-Symbolic training dataset, which should be automatically generated by Wandb (Weights & Biases). When this model is SFT'ed on other more practical GSM-style datasets (e.g., the MetaMathQA-GSM8K [5] split), the final training loss is observed to be much higher (> 0.2) than on the GSM-Symbolic training set. Why do you think this is the case? Hint: look into the GSM-Symbolic training dataset — any special pattern you can find? You are also encouraged to read the original GSM-Symbolic paper [2] for additional context.

2. Inference-Time Scaling with Majority Voting and Best-of-N (40 points)

While Supervised Fine-Tuning (SFT) improves a model's base performance on a task, we can further boost accuracy at inference time by generating multiple candidate responses and using a selection strategy to pick the final answer. This approach is a form of "inference-time scaling," where we trade off more computation at test time for better results.

In this question, you will explore two popular inference-time strategies:

- **Majority Voting (Self-Consistency):** Generating multiple samples and picking the numeric answer that appears most frequently.
- **Best-of-N (BoN):** Generating multiple samples and using a separate **Reward Model** to rank them, picking the one with the highest score.

All coding questions in this section use `a4_sampling_dist/` as the root directory.

Task Description and Setup

Overview

In this part, you will use both the base `gemma-3-1b-it` model and your fine-tuned `gemma-3-1b-it-SFT` model to generate $N = 8$ candidate responses for each question in the GSM-Symbolic test set. You will then evaluate the accuracy of both models when their final answers are selected via Majority Voting or Best-of-N sampling, and analyze how performance scales as the number of samples n increases.

Models

You will work with the following models:

- **Generators:** Both the base `gemma-3-1b-it` and your fine-tuned `gemma-3-1b-it-SFT` checkpoint.
- **Reward Model:** `Skywork-Reward-V2-Llama-3.1-8B`, used to rank the generated candidates.

Codebase

The main files for this section are:

- `generate_samples.py` — Script for generating $N = 8$ samples and scoring them using the Reward Model.
- `evaluate.py` — Script for computing Majority Voting and BoN accuracies from the scored samples.
- `utils.py` — Shared utilities for model loading, prompting, and answer extraction.
- `scripts/generate_samples.sh` — Shell script to launch the generation and scoring process.
- `scripts/get_results.sh` — Shell script to download the scored samples from Modal to your local machine.
- `scripts/evaluate.sh` — Shell script to run the local evaluation.
- `modal_app.py` — Modal application for running the remote sampling and scoring jobs.

Please carefully read the provided `README.md` file for a detailed walkthrough on how to get this codebase running on Modal. It usually takes less than 1 hour to complete all the sampling and evaluations for the whole section, using the default Modal configurations that we provide.

Reward Models

A **Reward Model (RM)** is a neural network trained to predict a scalar score representing the quality or "desirability" of a model-generated response relative to a given prompt. In the context of RLHF (Reinforcement Learning from Human Feedback), reward models are often trained on human preference data. The RM learns to assign higher scores to responses that are more accurate, helpful, or logically sound.

For a given question Q and multiple candidate answers $\{A_1, A_2, \ldots, A_n\}$, the reward model provides scores $S_i = RM(Q, A_i)$. In **Best-of-N** sampling, we select the answer A^* that maximizes this score:

$$A^* = \arg\max_{A_i} S_i$$

Coding: Implementing Sampling and Analysis (24 points)

(a) (8 points) **[Coding]** Complete the sampling and scoring logic in `generate_samples.py`. Specifically, you should:

1. In `generate_samples_batched`, implement the batched generation logic: build prompt texts with the chat template and answer prefix, tokenize them, and call `bundle.model.generate` with `do_sample=True`, `temperature=0.7`, and `top_p=0.95` to produce n samples.
2. In `score_samples_batched`, implement the reward model inference logic to compute scores for each generated sample.

(b) (8 points) **[Coding]** Complete the evaluation loop in `evaluate.py`. For each question and for each $n \in \{1, 2, 4, 8\}$, you should:

1. **Majority Voting:** Identify the most frequent numeric answer among the first n samples and compare it with the gold answer.
2. **Best-of-N (BoN):** Identify the sample with the highest reward score among the first n candidates, extract its numeric answer, and compare it with the gold answer.

(c) (8 points) **[Coding]** Complete the reasoning trace analysis logic in `analyze_traces.py`. You will implement the following modular functions and use them to calculate statistics:

- `check_majority_failure`: Returns True if at least one sample is correct but the majority prediction is not.
- `check_bon_failure`: Returns True if at least one sample is correct but the Best-of-N prediction is not.
- `get_unique_preds_count`: Returns the number of unique numeric answers in the predictions.
- **Distribution Visualization:** Implement the logic in `main` to calculate the unique answer distribution and plot it as a histogram (`unique_answers_dist.png`).

Written: Analysis of Inference-Time Scaling (16 points)

How to Run

The experiment is a three-step process:

1. **Generate and Score (Remote):** Run `modal run modal_app.py::launch_all` to generate $N = 8$ samples and score them using the reward model on Modal GPUs.
2. **Download Results (Local):** Run `bash scripts/get_results.sh` to pull the scored samples from the Modal cloud volume to your local `results/` directory.

3. **Evaluate Accuracy (Local):** Run `bash scripts/evaluate.sh` to compute Majority Voting and BoN accuracies locally for $n \in \{1, 2, 4, 8\}$.

(a) (2 points) **[Written]** Report your accuracies for both Majority Voting and Best-of-N, for both the base and SFT models, in Table 2.

Model & Strategy \ n	**n=1**	**n=2**	**n=4**	**n=8**
`gemma-3-1b-it` (Base) + Majority				
`gemma-3-1b-it` (Base) + BoN				
`gemma-3-1b-it-SFT` + Majority				
`gemma-3-1b-it-SFT` + BoN				

Table 2: Accuracy of Majority Voting and Best-of-N for base and SFT models.

(b) (2 points) **[Written]** Based on Table 2, compare the scaling behavior of Majority Voting and Best-of-N across both the base and SFT models. Does SFT change which sampling strategy is more effective?

(c) (6 points) **[Written]** Beyond simple accuracy, analyze the failure modes of Majority Voting and Best-of-N for **both the base and SFT models** using the metrics you implemented in Part (c) of the coding task.

Report your measured failure rates in Table 3. Furthermore, attach your generated histogram of unique responses per question (`unique_answers_dist.png`) for your analysis.

Model	**Metric**	**Majority Vote**	**Best-of-N**
`gemma-3-1b-it` (Base)	Failure to Pick Correct (given correct exists)		
`gemma-3-1b-it-SFT`	Failure to Pick Correct (given correct exists)		

Table 3: Reasoning trace analysis metrics for base and SFT models.

Based on your results, respond to the following claim: *"Inference-time scaling with Majority Voting is more robust than Best-of-N because it relies on the model's internal consistency rather than an external reward model's potential inaccuracies."*

Is this claim supported by your results? Use your measured failure rates to justify your answer.

(d) (6 points) **[Written]** Perform a manual failure analysis using the `results/failure_modes.txt` file generated by `analyze_traces.py` for the **base model** (`gemma-3-1b-it`).

1. **Majority Voting Failures:** Select 3 questions where Majority Voting failed despite a correct answer being present in the sample set (ideally questions with exactly 3 unique answers). For each:
   - State the question and the gold answer.
   - What was the majority answer?
   - Why might the majority answer be more plausible to the model than the correct one?
2. **Best-of-N (BoN) Failures:** Select 3 questions where the Reward Model picked an incorrect answer over a correct one. For each:
   - State the question and the gold answer.
   - What was the answer picked by the RM?
   - Compare the reasoning traces. Why do you think the Reward Model assigned a higher score to the incorrect trace?

References

[1] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[2] Mirzadeh, S. I., Alizadeh, K., Shahrokhi, H., Tuzel, O., Bengio, S., and Farajtabar, M. GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models. In *The Thirteenth International Conference on Learning Representations* (2025).

[3] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems 35* (2022), 27730–27744.

[4] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems 35* (2022), 24824–24837.

[5] Yu, L., Jiang, W., Shi, H., YU, J., Liu, Z., Zhang, Y., Kwok, J., Li, Z., Weller, A., and Liu, W. Metamath: Bootstrap your own mathematical questions for large language models. In *The Twelfth International Conference on Learning Representations* (2024).