The Chinese University of Hong Kong, Shenzhen
School of Data Science

DDA5001 Final Project — Large Language Models
Part III - Test-Time Scaling and Reasoning

Due: 23:59, Dec 21, 2025.

We guess that all of you, or at least almost all of you, are using GPT nowadays to do a lot of things (e.g., homework, coding, even for completing this project). In this project (three parts), we will study some basic working principles of these big AI models.

Contents

1 Introduction

Large Language Models (LLMs) are a transformative development in artificial intelligence and machine learning, with broad impact across applications such as text generation, translation, summarization, code synthesis, retrieval-augmented question answering, and tool use. Trained on large corpora of text (and increasingly multimodal data), these models can produce human-like outputs that are often coherent, informative, and contextually nuanced. Much of their capability arises from large-scale pre-training on next-token prediction using the Transformer architecture, hence the name "Generative Pre-trained Transformer" (GPT). During pre-training, models learn to predict the next token given previous context (see Section 2.3), implicitly acquiring linguistic structure, world knowledge, and useful reasoning patterns.

One of the most well-known LLMs is ChatGPT (with GPT-3.5, GPT-4, GPT-4o, GPT-5, o1, and o3 as its cores), developed by OpenAI, has demonstrated remarkable capabilities in understanding and generating text. It is widely utilized. There are also other representative models such as Clause by Anthropic, Gemini by Google, Llama by Meta, Qwen by Alibaba, to name a few. Nowadays, LLMs represent a significant advancement in the field of artificial intelligence, providing a convincing approach towards artificial general intelligence (AGI).

In this project, we aim to study several important elements of LLMs, including the modeling, training environment, simple pre-training, finetuning on math dataset, and reasoning model.

1. Part I (this part): Understand language modeling and install the training codes. Then, implement a simple (means small dataset) pre-training task for a simplified (means small size) yet very classic GPT model from scratch. In this part, you do not need to write codes. We will provide all the codes. Your task is to get familiar with LLM modeling and the training environment.

2. Part II (next part): Given a well pre-trained Qwen3 model (we will provide), perform fine-tuning on math datasets. You will need to write codes in this part. The aim is to see that fine-tuning can be important for improving downstream abilities such as math problem solving ability.

3. Part III (last part): Study reasoning models, that is, LLMs with strong mathematical reasoning ability. You will need to write codes in this part. We will see how sampling with different settings (as one method of test-time scaling) can improve the reasoning ability. We will use Qwen3 models in this part.

2 Basics of Large Language Models (LLMs)

2.1 Prompts

A prompt is the model's steering wheel: a short specification that tells the model what to do, with what context, and in what style. Unlike a generic "input", a well-crafted prompt guides behavior, small phrasing changes can shift tone, reasoning depth, or accuracy.

Typical ingredients:

- Task: What is needed (question, math, summarize, translate, classify, plan).
- Context: The evidence to use (passage, table, code, query).
- Constraints/style: Format, length, tone, schema.

Common forms (brief examples):

- Instruction style:

  ```
  Summarize the paragraph below in 2 bullets, each <12 words.
  ```

- Few-shot (learn by example) style:

  ```
  Q: 2+2?
  A: 4
  Q: 3+5?
  A:
  ```

- Chain-of-thought (reason stepwise) style:

  ```
  Think step by step before giving the final answer.
  ```

- Structured output style:

  ```
  Return JSON: {"verdict": "...", "evidence": ["...", "..."]}.
  ```

- System + user (chat setup) style:

  ```
  [System] You are a precise, terse tutor.
  [User] Explain dropout in one sentence.
  ```

Why it matters: Prompts guide correctness, reduce hallucinations, and make LLMs useful without retraining. Even a simple sentence is a prompt, e.g.,

```
Shenzhen is a great city.
```

2.2 Tokenization

Before a model can process text, it must convert characters of the input prompt into discrete units called *tokens*. This conversion, *tokenization*, is handled by a *tokenizer*, which maps text to token IDs and back (via a vocabulary). Modern LLMs use subword tokenization (e.g., BPE, WordPiece, Unigram) to balance coverage and efficiency: Common words stay intact; rare words are split into meaningful pieces.

Why it matters:

- **Efficiency**: Fewer tokens per sentence means faster, cheaper inference.

- **Robustness**: Subwords handle misspellings, new words, and multilingual text.
- **Limits**: Context windows are measured in tokens, not characters.

Example (illustrative only; actual splits/IDs depend on the tokenizer):

```
Input: Shenzhen is a great city.
Subwords: Sh en zhen is a great city .
Token IDs: 50, 831, 46732, 318, 257, 1049, 1748, 13
```

Note: Different tokenizers (GPT-2 BPE, Llama SentencePiece, etc.) produce different splits and IDs.

Since in this part we will pre-train a GPT model, we use GPT-2's tokenizer. It consists of 50257 tokens in total.

2.3 LLM Modeling and Training Formulation

2.3.1 Language Modeling

In almost all of the cases, LLM is an *auto-regressive* model (a.k.a. causal model), for predicting the next token. Given an input prompt $\boldsymbol{x}$, many language tasks are to predict what should be the next tokens $\boldsymbol{y}$, by choosing $\boldsymbol{y}$ that has the largest conditional probability

$$\mathbb{P}[\boldsymbol{y}|\boldsymbol{x}]. \tag{1}$$

This is is exactly the same as how we perform classification in logistic regression.

In LLMs, it models the conditional probability in (1) using an *auto-regressive Transformer*. Mathematically, it has the model

$$\mathbb{P}_{\boldsymbol{W}}[\boldsymbol{y}|\boldsymbol{x}] = \prod_{j=1}^{m} \mathbb{P}_{\boldsymbol{W}}[y_j|\boldsymbol{x}, \boldsymbol{y}_{1:j-1}]. \tag{2}$$

Here, "$\mathbb{P}_{\boldsymbol{W}}$" is the transformer, and thus $\boldsymbol{W} \in \mathbb{R}^d$ encompasses all trainable parameters of the transformer, including the query, key, value, and output attention matrices, as well as the gate, up, and down projection matrices of each transformer layer. y_j is the j-th token in the output $\boldsymbol{y}$.

Hence, LLMs generate text in the following manner:

(1) Receiving your input prompt $\boldsymbol{x}$ (which may be a question or a story you would like the model to do the continuation) to the LLMs.

(2) The LLMs will sample a y_1 from its vocabulary according to the distribution $\mathbb{P}_{\boldsymbol{W}}[y_1|x]$.

(3) Append the newly generated token y_1 to the input, giving $[x, y_1]$. Repeat the aforementioned steps by inputting $[x, y_1]$ into the model, until the maxmimum length / other stopping criterion is reached, which gives the generation $\boldsymbol{y}$.

2.3.2 Learning Problem Formulation of LLMs

The learning problem of LLMs is simply formulated from the maximum likelihood estimation (MLE) principle. That is, given a set of training data pairs $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$, the MLE learning problem of LLMs can be formulated as minimizing the negative log-likelihood of the generation probability:

$$\widehat{\boldsymbol{W}} \leftarrow \operatorname*{argmin}_{\boldsymbol{W}} \ \mathcal{L}(\boldsymbol{W}) = \sum_{i \in \mathcal{D}} \sum_{j=1}^{m} - \log \left(\mathbb{P}_{\boldsymbol{W}}[y_{i,j} | \boldsymbol{x}_i, \boldsymbol{y}_{i,1:j-1}] \right). \tag{3}$$

Both pre-training and finetuning are using this learning problem formulation. The major difference between them lies in that they use different dataset $\mathcal{D}$.

3 Part I: Pre-train a GPT Model on Shakespeare Dataset

Part I is an **individual** project, where you are asked to pre-train a GPT model to generate text that mimics the style of Shakespeare. Because the task is very simple, we only need a small pre-training dataset. You do not need to write any code in this part, while you only need to learn the training environment and install it by yourself.

3.1 Dataset

We will use the Shakespeare dataset, which consists of 1,003,854 tokens (i.e., $n = 1,003,854$). We provide this dataset in project package. The following is a quick preview of the dataset:

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.
```

You can see that this is not normal English, but a style of special writing, i.e., Shakespeare.

3.2 Code structure

How to get our code and data: Our code and data for project Part I is available at https://github.com/CUHKSZ-Course/DDA5001-25Fall.

Our code is based on the NanoGPT[1]. Here are the main files that you need to care about:

```
README.md  # project description and quick start
p1/  # project part 1 directory
  src/  # main source code folder
    train.py  # main file that implements the training logic
    model.py  # definitions for the GPT model
    sample.py  # code for model inference
    data/  # datasets and corresponding download scripts
```

If you do not have a powerful GPU (with at least 16GB RAM), you may have to use the free Tesla-P100 GPU offered by Kaggle platform. The following files are related to Kaggle setup:

```
Kaggle_training.md  # detailed instruction on using Kaggle GPU
p1/  # project part 1 directory
  main.ipynb  # notebook executed on Kaggle platform
  dataset-metadata-template.json  # template for creating dataset-metadata.json
  kernel-metadata-template.json  # template for creating kernel-metadata.json
```

3.3 Task

The objective of Part I is to setup the computational environment for running the experiment. To install the libraries, run

```
pip install torch numpy transformers datasets tiktoken wandb tqdm
```

Then, change `student_id` in `train.py` and `sample.py` to your own student id, e.g. 224040001. Run the following command to train a GPT model:

```
python train.py config/train_shakespeare_char.py
```

This will train a GPT model with the configuration specified in `config/train_shakespeare_char.py`. You should see the evolve of training loss and validation loss in the terminal output. Make sure that you have a GPU with enough memory when running the experiment, which can be either the Kaggle GPU or your own GPU. Do not attempt to use CPU since it is very slow. The checkpoint will be saved in the "`out`" directory automatically when the training is finished.

After you finishing pre-training, use the following command to generate text:

```
python sample.py --out_dir=out
```

This will automatically generate a Shakespeare-style text, without the need to provide a specific input prompt $\boldsymbol{x}$.

[1] https://github.com/karpathy/nanoGPT

3.4 Submission

1. A PDF report for Part I **up to one-page**. The page limit do not contain references pages and appendix. You can have two more pages for references and appendix. In this PDF, it should include
    - Plot the training loss and validation loss over 5000 steps in the same figure. Report the validation loss at iteration 5000.
    - Display the generated text for model trained at iteration 5000.
    - The difficulties (as well as how you solve them) and observations you had during completing project Part I.
2. Submit your checkpoint (i.e. the ckpt file) on Blackboard.
3. Since Part I is an **individual** work, everyone of you need to go through the process of installing this environment, running it, and submitting the report.

Note that there are **no** standard answers to the project. Different random seed may result in different performance, and different students may have different difficulties and observations. Once your results are reasonable, you can get the corresponding grades.

In terms of grades, Part I worth 30pts out of the 100pts of the whole project.

4 Part II: Math Finetuning on Qwen Model

As we have learned from the project part I, optimizing the auto-regressive loss enables LLM to effectively mimic the style of a given corpus and capture the underlying language structures. The procedure of training a model from scratch on next token prediction loss is termed *"pre-train"*. In reality, the pre-training dataset is of significantly larger size than the Shakespear dataset we used, which is usually in the tokens scale of billions or even trillions, enabling the model to learn from wide knowledge domains. For example, the WebText dataset that GPT-2 model was pre-trained on contains over 10 billion tokens.

However, pre-training language model on massive text may not directly turn them into powerful agents. To make the pre-trained model more useful and helpful, one needs to further *post-train* the pre-trained model to acquire the capability for downstream tasks such as question-answering, knowledge retrieval, task planning, math solving, etc. *Finetuning* is one of the important post-training methods.

Math finetuning. This finetuning method is simply to stimulate / improve base model's ability for solving challenging mathematical problems. Nowadays, mathematical ability of an LLM is an important evaluation criterion. We will see more about LLM reasoning (test-time scaling) for solving challenging mathematical problems in part III.

4.1 Dataset: Math500

We will use the Math500 dataset, it contains 12000 training and 500 test data. We have applied filtering to filter some too long answers, and hence we have less than 12000 training data in total.

We have split the training set to two parts: 90% of them are remained as training, while 10% are regarded as validation set.

- Example from Math500:

  `Problem: What is the smallest positive perfect cube that can be written as the sum of three consecutive integers?`

  `Solution: The sum of three consecutive integers takes the form $(k-1)+(k)+(k+1)=3k$ and hence is a multiple of 3. Conversely, if a number` n `is a multiple of 3, then` $n/3-1$, $n/3$, `and` $n/3+1$ `are three consecutive integers that sum to give` n. `Therefore, a number is a sum of three consecutive integers if and only if it is a multiple of 3. The smallest positive perfect cube that is a multiple of 3 is` $3^3 =$ `\boxed{27}.`

4.2 Tasks

We will finetune the Qwen3-0.6B-Base model (disable its thinking mode) on the training set of Math500. Then, we will use different optimizers (you can call packages directly) to compare their performance. Finally, we will evaluate the finetuned model on the test set of Math500, in comparison to the base model.

4.2.1 Task 1: Tokenization

This part contains two sub-parts:

- Apply chat-template, which is to extract problem and solution parts in the dataset to be "User" and "Assistant", respectively.

  ```
  ### User: Put the problem description here, and add one more instruction sentence at the end: "Please reason step by step, and put your final answer within \boxed{}."
  ```

  ```
  ### Assistant: Solution to the problem.
  ```

- Apply tokenization to the training dataset after applying chat-template.

In summary, we first transfer the training set to the template that the Qwen3 model can recognize, and then apply the corresponding tokenizer provided by Qwen3.

Task details. Finish the instruction tuning pipeline by following the steps below:

- Finish `# TODO` in `prepare.py` to finish the `apply_chat_template` function. Specifically, fill in the `XXX` positions we have removed in this function.

- Fill in the `XXX` in "system prompt", which is to add the following instruction sentence after stating the problem / question:

  ```
  "\nPlease reason step by step, and put your final answer within \\boxed{}."
  ```

  This instruction sentence turns out to be useful for solving mathematical problems.

- Applying masking to the prompt, which means we do not calculate loss in terms of prompt, as loss is calculated for the labels. Specifically, fill in the `XXX` in `label[:XXX] = [-100] * XXX # Mask prompt`.
- Once finished, you should be able to train the model by running `python prepare.py` listed in the `main.ipynb`. This will generated the prepared training and validation dataset for finetuning.

4.2.2 Task 2: Training under Different Optimizers

After preparing the training and validation dataset above, we will explore different training optimizers, especially about their hyper-parameter setting, and then apply them to finetune Qwen3-0.6B-Base model on the Math500 training set.

Optimizers we consider include:

1. **SGD**. The optimizer we studied in lecture.
2. **Adam**. The optimizer we studied in lecture.
3. **Low-rank adaptation (LoRA).** LoRA is a memory-efficient optimizer that can handle cases where Adam will cause out-of-memory (GPU RAM) issue. LoRA freezes the pre-trained weight and only update the factorized low-rank matrix:

   $$\boldsymbol{W} = \boldsymbol{W}_0 + \boldsymbol{B}\boldsymbol{A}.$$

   Here, $\boldsymbol{W}_0 \in \mathbb{R}^{m \times n}$ is the frozen pre-trained weight, $\boldsymbol{B} \in \mathbb{R}^{m \times r}$ and $\boldsymbol{A} \in \mathbb{R}^{r \times n}$ are the low-rank matrices to be trained. r is usually much smaller than $\min\{m, n\}$. Hence, the trainable parameter is $(m + n)r$, which is much less than mn.

   In LoRA, we still need to specify an inner optimizer for optimizing $\boldsymbol{A}, \boldsymbol{B}$, which we use Adam by default.

All the above optimizers are available in torch (SGD and Adam) and hugging face PEFT packages (LoRA). You can directly call them from torch and hugging face libraries.

Task details.

- **Call optimizers.** Finish `XXX` in `# TODO` to call all the above optimizers. Specifically, you need to call the AdamW (decoupled weight decay in Adam), SGD, and LoRA (to all possible modules).
- **Optimizer hyperparameters.** Try different hyperparameters including different learning rate for all optimizers, number of epochs (do not exceed 3 due to overfitting), and LoRA rank r in LoRA.
- Once you have finished all above steps, you should be able to run the finetuning by `python finetune.py` listed in the `main.ipynb`. Then, you will observe the training and validation dynamic in the log file. You also need to write a small code for plotting the figures of training and validation loss along with training steps.

4.2.3 Task 3: Evaluate on Math500 Test Split

Evaluate both the base model and finetuned model on the provided Math500 Test split.

Task details.

- Fill in XXX in `rollout.py`. These XXX are the same as the one you filled in the `prepare.py`.
- Then, you will be able to run `rollout.py` and `evaluate.py` listed in the `main.ipynb`. You will get the scores on the 500 test data points.

4.2.4 Code and Dataset

Our code and dataset for part II is still available at `https://github.com/CUHKSZ-Course/DDA5001-25Fall` inside folder p2.

4.3 Submission

1. A PDF report for part II **up to 4 pages (less than 4 pages is not an indicator for grade deduction; we will focus on your contents rather than report length)**. The page limit do not contain references pages and appendix. In this PDF, it should include
    - Figures contain the training and validation losses for all the optimizers with different hyperparameters.
        - Try different hyperparameters (learning rates, number of epochs, and LoRA rank) determined by yourself. Any are fine once they work and you observe some trends to find good hyperparameters.
        - We suggest to plot training loss and validation loss in different figures.

      You need to write a small code to plot figures.
    - A table which summarizes the maximum allocated memory and training time for Adam, SGD, and LoRA. You need to write a small code to collect these values.
    - A table summarize the Math500 test score of the base and finetuned models using the found good hyperparameters.
    - The difficulties (as well as how you solve them) and observations you had during completing project part II.
2. Your full code, **without** the model checkpoint.

Responsibilities of each group member: Please clearly indicate the responsibilities and contributions of each group member in the footnote of the first page. A group member with zero contribution cannot earn the score.

Note that there are **no** standard answers to the project. Different random seed may result in different performance, and different students may have different difficulties and observations. Once your results are reasonable, you can get the corresponding grades.

Part II worth 40pts out of the 100pts of the whole project.

5 Part III: Test-time Scaling on Qwen Model (new content)

In part II, we have learned that finetuning model in a small-scale dataset can improve its capability in solving mathematical problems. Apart from finetuning, sampling strategy (e.g. temperature, pass@k, etc.) also plays an critical role in affecting the model's generation quality. In part III, we will explore test-time scaling phenomenon to understand reasoning ability of LLMs.

5.1 Dataset and Model

Datasets. We evaluate our methods on three math problem-solving benchmarks: Math500,[2] AMC23,[3] and AIME25.[4] Math500 contains 500 problems drawn from the MATH benchmark, while AMC23 and AIME25 contain 40 AMC 2023 and 30 AIME 2025 problems, respectively, each paired with a short numerical or algebraic answer. A preview of the question format is shown below:

- `Convert the point` $(0, 3)$ `in rectangular coordinates to polar coordinates.`
- `How many positive perfect squares less than` 2023 `are divisible by` 5`?`
- `Find the sum of all integer bases` $b > 9$ `for which` 17_b `is a divisor of` 97_b`.`

You are suggested to look at some of questions (especially AIME25) to see how hard they are.

Models. Since Part III does not involve additional training, we directly evaluate strong math-specialist language models. In particular, we use the math-specific base model Qwen2.5-Math-1.5B[5] and its GRPO-based instruction-tuned variant Qwen2.5-Math-1.5B-Instruct.[6]

5.2 Test-time Scaling

In Part II, we mainly improved the model itself by finetuning. In this part, we keep the model frozen and study *how we use it* at inference time. This idea is called *test-time scaling*: by adjusting the decoding strategy and drawing multiple samples, we can often obtain better answers without any extra training.

Decoding strategies: temperature and top-p. An LLM defines a probability distribution over the next token at each step. To turn this into an actual sequence, we must choose a sampling rule. Two very common controls are *temperature* and *top-p*.

For a token x_t at step t with logits ℓ_t, temperature scaling with parameter $\tau > 0$ replaces the distribution by

$$p_\tau(x_t \mid x_{<t}) = \mathrm{softmax}\big(\ell_t/\tau\big).$$

[2] `https://huggingface.co/datasets/math-ai/math500`
[3] `https://huggingface.co/datasets/math-ai/amc23`
[4] `https://huggingface.co/datasets/math-ai/aime25`
[5] `https://huggingface.co/Qwen/Qwen2.5-Math-1.5B`
[6] `https://huggingface.co/Qwen/Qwen2.5-Math-1.5B-Instruct`

When $\tau = 1$, we recover the original model distribution. A smaller temperature (e.g. $\tau = 0.2$) makes the distribution sharper and the generation more deterministic, while a larger temperature (e.g. $\tau > 1$) makes the distribution flatter and the outputs more random and diverse.

Top-p (nucleus) sampling truncates the distribution: we sort tokens by probability, keep the smallest prefix whose total probability is at least p (for example $p = 0.9$), renormalize over this prefix, and then sample. This avoids extremely low-probability tokens while still allowing variety in the output.

From one sample to many samples. In normal ChatGPT-style usage, we give a prompt and only see *one* generated answer. The accuracy in this setting is called *pass@1*, and it is still an important metric for everyday interactions.

Modern inference engines (such as vLLM) can, however, generate *multiple* answers for the same prompt very efficiently using batching and KV caching. Let T_k be the wall-clock time to generate k answers for one prompt. In practice, efficient batching often gives

$$T_{16} \ll 16 \cdot T_1,$$

so drawing 16 candidate solutions is much cheaper than running 16 completely separate chats.

Once we have several answers, we can combine them. A simple example is *majority vote*: for each prompt, we generate k responses, extract the final numeric answer (e.g. the content inside `\boxed{}`) from each, and then choose the most frequent one as the final prediction. More generally, we can imagine any rule that looks at a small set of candidates and picks the best one.

Pass@k: measuring the benefit of multiple tries. If we are allowed to look at more than one answer, it is natural to ask:

How often does the model get it right in at least one of the k tries?

This is exactly what the *pass@k* metric measures. It is especially useful for math and code benchmarks, where we can automatically check whether an answer is correct.

Unbiased estimation of pass@k: suppose for a fixed problem we generate n samples, among which c are correct. Imagine we randomly pick k samples *without* replacement from these n candidates. The probability that *none* of the selected samples is correct is

$$\frac{\binom{n-c}{k}}{\binom{n}{k}}, \qquad n \geq k.$$

Therefore, the probability that at least one of the k chosen samples is correct, i.e. the pass@k value for this problem, is

$$\text{pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}, \qquad n \geq k. \tag{4}$$

In practice, we usually evaluate pass@k for a set of values such as $k \in \{1, 2, 4, 8, 16\}$ (or more generally $k = 2^i$). Here:

- pass@1 corresponds to the "single answer" chat setting that users experience in daily life;

- pass@k with larger k shows how much additional performance we can gain by using modern infrastructure (batching, KV cache) to generate and combine multiple samples at test time.

In this part of the project, you will vary sampling hyperparameters (such as temperature, top-p, and the number of samples k), and compare pass@1 with pass@k on our math benchmarks to see how test-time scaling changes the model's effective reasoning ability.

5.3 Tasks

In this part, we keep the model fixed and study how test-time scaling (sampling multiple responses per problem) affects math reasoning performance. You will use the provided scripts `inference.py` (generation) and `evaluate.py` (evaluation). Some key lines are masked with `# TODO` for you to complete.

Setup. We consider three benchmarks: Math500, AMC23, and AIME25 (see the previous subsection), and two models:

- Base model: `Qwen/Qwen2.5-Math-1.5B`.
- GRPO-tuned reasoning model: `Qwen/Qwen2.5-Math-1.5B-Instruct`.

For each dataset, you should always generate a fixed number of rollouts per problem:

- Math500: 16.
- AMC23 and AIME25: 64.

Smaller pass@k values (e.g. $k = 1, 2, 4, 8$) will then be computed from these rollouts using the unbiased pass@k formula in (4).

5.3.1 Task 1: Run generation with different temperatures

To keep the workload reasonable, we fix top-p (e.g. $p = 0.9$) and only vary the temperature.

Task details.

- For each dataset and each model (base and GRPO-tuned), choose about three temperature values, for example
$$\tau \in \{0.6,\ 1.0,\ 1.2\}.$$
Here $\tau = 1.0$ corresponds to the original model distribution, $\tau = 0.6$ is more deterministic, and $\tau = 1.2$ is more random.
- For Math500, an example command is:

```
python inference.py \
  --model "Qwen/Qwen2.5-Math-1.5B-Instruct" \
  --dataset "math" \
  --dp-size 2 \
  --batch-size 16 \
  --rollout-n 16 \
  --temperature 1.0 \
  --top-p 0.9 \
  --output_file outputs/math500_instruct.jsonl
```

You can change the sampling temperature using the above `--temperature` input.

For AMC23 and AIME25, use the same pattern but with `--dataset "amc"` / `"aime"` and `--rollout-n 64`.

- Each group can split the runs across members (e.g. different students use their own Kaggle accounts) to save time.

5.3.2 Task 2: Implement pass@k and Majority Vote

In `evaluate.py` we group multiple rollouts by problem ID and call a verifier to obtain a score for each response. Recall that if a problem has n samples in total and c of them are correct, the unbiased pass@k is given by (4). In addition, we can aggregate the n predictions by *majority vote*: choose the most frequent extracted answer as the final prediction and check whether it is correct.

Task details.

- Complete the `# TODO` parts in `evaluate.py` to:
  - Count n (number of rollouts) and c (number of correct rollouts) for each problem.
  - Implement the pass@k formula (4) in Python (e.g. using `math.comb`), and compute pass@k for $k \in \{1, 2, 4, \dots\}$ up to the total number of rollouts.
  - Implement majority vote over the extracted predictions (break ties in a fixed way) and compute the overall maj@1 accuracy across problems.
- Run `evaluate.py` on at least one generated JSONL file and check that pass@k for several k and maj@1 are printed correctly.

5.3.3 Task 3: Analyze and compare models

Once you have both the generation results (Task 1) and the evaluation metrics (Task 2), you can compare the base and GRPO-tuned models under test-time scaling.

Task details.

- For each benchmark (Math500, AMC23, AIME25) and each temperature setting, run `evaluate.py` on the JSONL files you generated in Task 1, for both models (base and GRPO-tuned).
- From the printed metrics, collect pass@k (for several k, including $k = 1$) and maj@1 for:
  - different temperatures, and

  - different models (base vs GRPO-tuned).

- Use *tables or plots* to summarize the main trends you observe. You are free to choose the exact format, as long as your findings are clear.

5.3.4 Code

The code for part III is available at `https://github.com/CUHKSZ-Course/DDA5001-25Fall` inside folder p3. If you are using Kaggle, please ensure you have carefully read the instructions in the Part 3 section of the repository README.

5.4 Submission

1. A PDF report for Part III (you may use up to **4 pages** for the main content; additional pages for references and appendix are allowed). In this PDF, you should include:
   - Tables or plots that summarizes pass@k (for several k, including $k = 1$) and maj@1 under different temperatures and different models (base vs GRPO-tuned), on the benchmarks (Math500, AMC23, or AIME25).
   - A brief description of your observations, for example: how temperature and the number of rollouts affect performance, and how the GRPO-tuned model compares with the base model.
   - The difficulties (as well as how you solved them) and any additional observations you had during completing Part III.
2. Your full code for Part III (including the completed `inference.py`, `evaluate.py`, any plotting scripts) and your evaluated benchmarks JSONL files.

Responsibilities of each group member: Please clearly indicate the responsibilities and contributions of each group member in the footnote of the first page. A group member with zero contribution cannot earn the score.

Note that there are **no** standard answers to the project. Different random seed may result in different performance, and different students may have different difficulties and observations. Once your results are reasonable, you can get the corresponding grades.

Part III worth 30pts out of the 100pts of the whole project.

6 Project Tutorials

- We will have 3 tutorial (at 6:30pm - 7:30pm of Nov 4, Nov 18, Dec 2) to help you finish the project.

- These three tutorials will be **online** with zoom link: `https://cuhk-edu-cn.zoom.us/j/99410791830?pwd=bGR2NuHazCaaOFOPKhqYPf6ulb2bAq.1`. We will also provide **recordings** of these three tutorials.

- **Note:** We highly recommend that you attend (or go through the recording) every tutorial before completing each part. We will basically teach the main steps on how to complete the project.