Ho Chi Minh city - University of Science
Faculty of Information Technology

Introduction to Machine Learning

LECTURER

- Bùi Tiến Lên

PREPARED BY

- 22127032 - Nguyễn Thiên Bảo
- 22127220 - Nguyễn Anh Kiệt
- 22127275 - Trần Anh Minh
- 22127280 - Đoàn Đặng Phương Nam
- 22127353 - Cao Minh Quang

Ho Chi Minh city, 2024

Contents

1) Introduction 3
1.1) Problem Statement 3
1.2) Environments and Resources 3
2) Dataset 5
2.1) Orca Math Word Problems Dataset 5
2.2) Vietnamese Intermediate Reality Math Problems Dataset 5
2.3) Integration of Datasets 6
3) Original Model 6
4) Our Fine-tuned Model 7
4.1) Theoretical Basis 7
4.1.1) What is Fine-tuning? 7
4.1.2) From Fine-tuning To PEFT 7
4.1.3) QLoRA 8
4.2) Implementation 9
4.2.1) Dataset 9
4.2.2) Model 9
4.2.3) Code Sample of Fine-tuning 10
4.3) Our fine-tuning strategy 11
5) Web Application for the Model 13
5.1) Project structure 13
5.1.1) Client side 13
5.1.2) Server side and Database 13
5.1.3) Chat bot service 14
5.2) User instructions 14
5.2.1) Client side 14
5.2.2) Server side 14
5.2.3) Chat bot service 14
5.3) The final product: 15
6) Statistics and Evaluation 19
6.1) What is ROUGE Score? 19
6.2) Experiments and Evaluation 19
7) Resource and Demo 21
8) References 21

1) Introduction

1.1) Problem Statement

Mathematics education plays a crucial role in the cognitive development of students from elementary to secondary school. However, providing personalized assistance to every student in understanding mathematical concepts and solving problems is a persistent challenge. Teachers often face constraints in time and resources, while students may struggle with gaps in understanding that hinder their academic progress.

To address this challenge, we propose MathLLM, a tailored large language model framework designed specifically to assist with elementary to secondary mathematics. Unlike general-purpose LLMs, which may lack precision in mathematical reasoning, MathLLM is fine-tuned to understand the curriculum and problem-solving approaches suitable for younger learners. It provides step-by-step explanations, clarifies concepts, and generates accurate answers in an accessible and interactive manner.

The framework employs efficient techniques such as 4-bit quantization for resource optimization and Low-Rank Adaptation (LoRA) for cost-effective fine-tuning. MathLLM ensures scalability and flexibility, allowing educators and developers to adapt the model for specific grade levels or curricula. Furthermore, its comprehensive pipeline for model training, fine-tuning, and deployment makes it an ideal solution for integrating AI into mathematics education.

By bridging the gap between artificial intelligence and accessible learning, MathLLM aims to democratize personalized math education, empowering students to build confidence and proficiency in mathematics from elementary to secondary levels.

1.2) Environments and Resources

For training and testing the model as well as the web application, we will use the Kaggle environment since it provides us with GPU T4 x2 with 16GB each, which is efficient for the training and testing process.

For saving essential resources such as pretrained and fine-tuned models, datasets, tokenizer,... we will use theHugging Face environment since it allows us to create repositories for pushing and storing our resources. We create two repositories:

- **Group01-ML-Project**: This is our private repository that saves our resources.

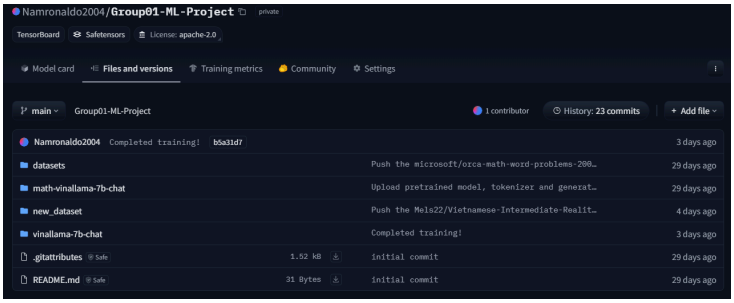


Figure 2: Our private repository on HuggingFace

- **math-vinallama-7b-chat**: This is our public repository utilized to deploy necessary files publicly that third-party applications can use for their personal objectives.

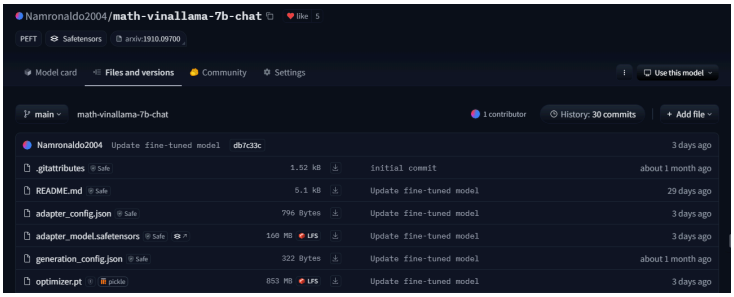


Figure 3: Our public repository on HuggingFace

For the modules and functions that we used:

- **torch**: A deep learning framework used for creating, training, and evaluating machine learning models.
- **torch.utils.data.Dataset**: A base class for handling datasets in PyTorch.
- **transformers**: Provides tools for working with state-of-the-art NLP models like BERT, GPT, etc. Here we will use the following class:
  - **AutoModelForCausalLM**: Loads causal language models for text generation tasks.
  - **AutoTokenizer**: Loads tokenizers compatible with the specified model.
  - **BitsAndBytesConfig**: Configures 4-bit quantization to optimize memory usage during training and inference.
  - **GenerationConfig**: Sets parameters (e.g., temperature, top-p) for generating responses from the model.
  - **TrainingArguments** and **Trainer** (referenced indirectly): Handle training configurations and execution.
- **datasets**: A Hugging Face library for working with datasets in machine learning workflows. Specific methods and classes used:
  - **load_dataset**: For loading datasets from various sources.
  - **Dataset**: A dataset object for creating and modifying data.
- **googletrans**: A Python library for language translation.
  - **Translator**: Used for translating text during preprocessing or evaluation.
- **peft**: A library for parameter-efficient fine-tuning.
  - **LoraConfig**: Configures LoRA (Low-Rank Adaptation) for fine-tuning large models efficiently.
  - **PeftModel**: Handles models fine-tuned with parameter-efficient techniques.

- **get_peft_model**: Applies PEFT techniques to an existing model.
- **prepare_model_for_kbit_training**: Prepares models for training with k-bit quantization.

- **huggingface_hub**: A library for managing and interacting with resources on the Hugging Face Hub.
  - **HfFolder**: Provides utility functions for managing authentication tokens locally.
  - **Repository**: Enables interaction with Hugging Face model repositories for cloning, pushing, and updating.
  - **HfApi**: Offers programmatic access to Hugging Face Hub features, such as searching for models and datasets.
- **os**: Provides functions for interacting with the operating system (e.g., file and directory management).
- **shutil**: Used for high-level file operations such as copying and moving files.

2) Dataset

2.1) Orca Math Word Problems Dataset

The #Orca Math Word Problems dataset, developed by Microsoft, is a comprehensive resource containing 200,000 English-language records of straightforward math problems. This dataset is designed to train models in solving basic mathematical queries expressed in natural language. This dataset serves as a foundation for enhancing mathematical reasoning in AI models, thanks to its structured format and diverse problem statements.

For our project, the dataset will undergo a translation process to adapt its content to the Vietnamese language. This ensures that the fine-tune model is capable of understanding and effectively responding to math queries in Vietnamese.

2.2) Vietnamese Intermediate Reality Math Problems Dataset

In addition to utilizing the *Orca Math Word Problems* dataset, our team has compiled a supplementary dataset tailored specifically to the Vietnamese educational context. The #Vietnamese Intermediate Reality Math Problems dataset comprises 500 instances of more complex mathematical problems drawn from real-world scenarios typically encountered by secondary school students. These problems require advanced reasoning and are presented in plain text for accessibility and ease of processing.

This dataset plays a critical role in fine-tuning the model to follow the Chain-of-Thought (CoT) methodology. Training with this dataset equips the model to generate structured, step-by-step solutions for more challenging math problems. This enhances the model's capability to tackle read-world scenarios and deliver logical, comprehensive explanations.

2.3) Integration of Datasets

The integration of the *Orca Math Word Problems* and *Vietnamese Intermediate Reality Math Problems* datasets serves a dual purpose. The fine-tuned translated *Orca Math Word Problems* dataset establishes a foundational understanding of basic mathematical concepts in Vietnamese. Meanwhile, the subsequent fine-tuning with the *Vietnamese Intermediate Reality Math Problems* dataset enhances the model's ability to tackle more complex, real-world mathematical challenges and produce step-by-step solutions.

This sequential fine-tuning approach ensures the model develops a comprehensive skill set, ranging from solving simple problems to addressing nuanced, real-life scenarios. It bridges the gap between theoretical and practical math applications, resulting in a versatile tool to provide accurate, thoughtful solutions in a way that's relevant to everyday use.

3) Original Model

In our project on utilizing LLMs for a web application, we selected **VinaLLaMa-7b-chat**, a chat variant of the foundation model **VinaLLaMa**. According to the official paper, this model exhibits the following features: it is based on the **LLaMA-2** architecture, pre-trained with over 800 billion tokens, optimized for Vietnamese while also supporting English, and leverages a diverse range of public datasets alongside high-quality synthetic data. Its performance is reported to be closely competitive with **ChatGPT-3.5-Turbo**.

We conducted a survey on a range of Vietnamese LLMs and found a statistical table as follows.

Model	**arc_vi**	**hellaswag_vi**	**mmlu_vi**	**truthfulqa_vi**	**average**
URA-LLaMA-13B	0.3752	0.4830	0.3973	0.4574	0.4282
BLOOMZ-7B	0.3205	0.4930	0.3975	0.4523	0.4158
PhoGPT-7B5-Instruct	0.2470	0.2578	0.2413	0.4759	0.3055
SeaLLM-7B-chat	0.3607	0.5112	0.3339	0.4948	0.4252
Vietcuna-7b-v3	0.3419	0.4939	0.3354	0.4807	0.4130
VinaLLaMA-2.7B-chat	0.3273	0.4814	0.3051	0.4972	0.4028
VinaLLaMA-7B-chat	**0.4239**	**0.5407**	**0.3932**	**0.5251**	**0.4707**
VBD-LLaMA2-7B-50b	0.3222	0.5195	0.2964	0.4614	0.3999

Table 1: VLSP Benchmark Scores of Supervised Fine-tuning Models

We chose **VinaLLaMa-7b-chat** for our project due to several compelling reasons.

- First, as seen in the table above, which presents VLSP Benchmark Scores of Supervised Fine-tuning Models, **VinaLLaMa-7b-chat** demonstrates the highest accuracy among all models listed. According to the VLSP-LLM 2023 benchmark, this evaluation suite is tailored specifically for Vietnamese and parallels the OpenLLM Leaderboard by HuggingFace. It encompasses four adapted assessments—ARC Challenge, HellaSwag, MMLU, and TruthfulQA—providing a comprehensive evaluation of language models' ability to understand and generate Vietnamese text across various complexities. VinaLLaMa-7b-chat's superior accuracy highlights its effectiveness in handling Vietnamese language tasks.
- However, based on data from VMLU Leaderboard, the model's performance on VMLU datasets, particularly in the ***STEM domain***, is relatively lower. This dataset includes questions related to mathematics as well as other STEM fields. Given this limitation, our group aims to address this weakness by fine-tuning the model. This fine-tuning focuses on math-specific datasets, enhancing the model's capabilities in **mathematical reasoning and problem-solving**.

4) Our Fine-tuned Model

4.1) Theoretical Basis

4.1.1) What is Fine-tuning?

Fine-tuning is a foundational process in machine learning that involves adapting a pre-trained model to a specific task or domain. Pre-trained models are typically developed on vast datasets, enabling them to learn general patterns, representations, and features. Fine-tuning capitalizes on this generalized knowledge by training the model further on a smaller, task-specific dataset. This method significantly reduces the computational costs and data requirements compared to training a model from scratch.

The process of fine-tuning adjusts the weights of the pre-trained model to optimize its performance for the new task. By retaining the core capabilities learned during pre-training, the model can quickly adapt to specialized tasks such as sentiment analysis, object detection, or medical image classification. Fine-tuning has become a cornerstone in applications like natural language processing (e.g., GPT, BERT), computer vision (e.g., ResNet, EfficientNet), and speech recognition systems.

However, fine-tuning large-scale models presents challenges, including high memory usage and computational demands. These challenges have led to the development of more efficient approaches, such as Parameter-Efficient Fine-Tuning (PEFT).

4.1.2) From Fine-tuning To PEFT

Parameter-Efficient Fine-Tuning (PEFT) is a set of advanced techniques designed to address the resource-intensive nature of fine-tuning large models. Instead of updating all

the parameters of a pre-trained model, PEFT focuses on modifying only a small subset of parameters or incorporating additional lightweight modules. This selective adjustment reduces computational costs and memory usage while preserving the pre-trained model's knowledge.

PEFT is particularly beneficial in scenarios where deploying or fine-tuning full-scale models is impractical due to hardware or cost constraints. Some of the key techniques used in PEFT include:

- **Adapter Tuning**

Adapters are lightweight modules added to the model's architecture, typically between the layers of a pre-trained model. During fine-tuning, only these adapter layers are updated, while the rest of the model remains frozen. This modular design allows for task-specific adaptations without modifying the entire model.

- **Low-Rank Adaptation (LoRA)**

LoRA is a technique that injects low-rank matrices into the model's weight space. These matrices capture task-specific information, reducing the number of parameters that need to be fine-tuned. LoRA is highly efficient and well-suited for adapting large language models to new tasks.

- **BitFit**

BitFit stands for "Bias Fine-Tuning," a method where only the bias parameters of a model are updated during fine-tuning. This approach minimizes the number of trainable parameters, reducing computational costs while maintaining good performance.

4.1.3) QLoRA

QLoRA (Quantized Low-Rank Adaptation) is an advanced method that combines the principles of quantization and low-rank adaptation for fine-tuning large-scale models. It builds on the foundation of PEFT by introducing two critical optimizations:

- **Quantization**

QLoRA quantizes the model's parameters to lower precision, such as 8-bit or 4-bit, to reduce memory usage and computational requirements. Quantization is especially beneficial for large models, as it significantly reduces the hardware resources needed for storage and inference without a substantial loss in model accuracy.

- **Low-Rank Adaptation**

In addition to quantization, QLoRA incorporates low-rank adaptation layers that enable task-specific fine-tuning. These layers are designed to capture the task-specific knowledge required for adaptation while keeping the original pre-trained model largely intact.

The combination of these techniques allows QLoRA to achieve high efficiency and accuracy. Its hybrid approach enables organizations to deploy and fine-tune large-scale

models in resource-constrained environments, such as edge devices or smaller cloud infrastructures. QLoRA is particularly effective for tasks that require adapting state-of-the-art language models to specialized domains, such as healthcare, finance, or legal text processing.

In summary, QLoRA exemplifies the evolution of fine-tuning methods, providing a scalable solution for adapting large-scale models without the high resource demands typically associated with traditional fine-tuning approaches.

4.2) Implementation

4.2.1) Dataset

Before conducting fine-tuning, **defining methods or classes to collect datasets** is necessary. In our implementation, we define two classes: the first one is **Microsoft-MathDataset** for the **Orca Math** dataset, the remaining one is **MyMathDataset** for our custom dataset.

Each of the dataset class will inherit the base class **Dataset** supported by the **torch** library, so it must have 3 main methods:

- `__init__`: This method is used to load data through the `load_dataset` function. The data will then be split into two parts: one for the questions or problems to be solved, and the other for the answers or solutions to the problems. This method also allows users to specify the portion of the dataset to use for training through two variables: `start_sample_index` and `num_get_samples`. Finally, the dataset will be saved to our repository on **HuggingFace** for easy access during subsequent training sessions.
- `__len__`: This method determines the number of questions in the extracted sample.
- `__getitem__`: This method retrieves a pair of question and answer, then passes them to the `generate_and_tokenize_prompt` function. This function transforms the pair into a prompt formatted according to the template provided on the **HuggingFace** page for the LLM and tokenizes the prompt into numerical tokens before passing it into the training process.

4.2.2) Model

To facilitate fine-tuning, we also define a class called **MathLLM**, which includes the following key methods:

- `get_model`: This method is used to fetch the model to be trained. The model can either be the original LLM or a previously fine-tuned model that will undergo further fine-tuning. This method is also used to retrieve the tokenizer and configuration parameters for text generation, such as top_p, temperature, etc. Finally, like the dataset, the fine-tuned LLM information will also be saved to the team's **HuggingFace** repository if it hasn't been saved yet.

- A highlight of this method is that our team employs **QLoRA (Quantization combined with Low-rank Adaptation)**, which is a **PEFT** technique. The primary goals of using these techniques are to enable faster model training, reduce GPU usage, and achieve additional benefits such as mitigating the **"catastrophic forgetting"** problem during LLM fine-tuning and avoiding model **overfitting**.

- `fine_tune_LLM`: This method requires the user to input data for fine-tuning and specify additional hyperparameters (if any). For training, we utilize several classes from HuggingFace's transformers library, namely **TrainingArguments** and **Trainer**. In **TrainingArguments**, besides defining hyperparameters, we specify parameters such as `save_total_limit = 1`, meaning only the information from the most recent fine-tuning session is saved. This is because we've already prepared a backup folder on our repository. Other examples include settings like the optimizer, `lr_scheduler_type`, or `warmup_ratio`, which help regulate the learning rate during each weight update, enabling the model to learn more effectively. Fine-tuning details will be saved to the repository after the process is completed.

- `generate_answer`: This method is used for testing by inputting a query and letting the LLM generate a response.

- `deploy_fine_tuned_model`: This method is used to transfer the necessary files from our private repository to our public repository, supporting other applications that may use our model for their purposes.

4.2.3) Code Sample of Fine-tuning

After implementing all essential classes of datasets and model, we will conduct fine-tuning through the below code.

```
# Initialize the model class
math_llm = MathLLM()

# Get model
math_llm.get_model(model_name = MODEL_NAME)

# Get data and fine-tune model on this data
data = MyMathDataset(start_sample_index = 0, tokenizer = math_llm.tokenizer)
math_llm.fine_tune_LLM(train_data = data, num_epoch = 10, batch_size = 100,
learning_rate = 2e-4)

# Test the model after fine-tuning
response = math_llm.generate_answer(query = "Hãy cho tôi biết kết quả của phép
tính 45 * 6 là bao nhiêu?")
print(response)

# Deploy the model on the public repository.
math_llm.deploy_fine_tune_model()
```

While running the above code, the pretrained or fine-tuned model will be obtained, and then, to officially fine-tune LLM, we must pass the API Token provided by Wandb.

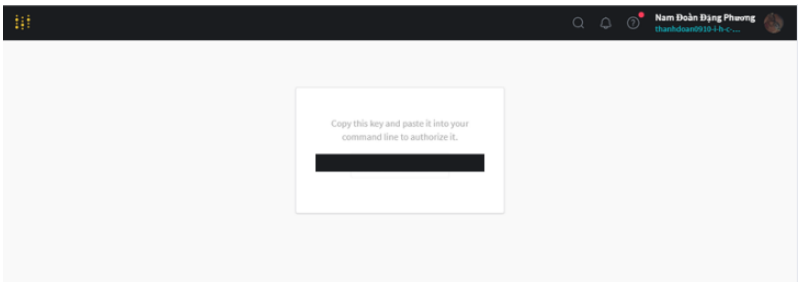


Figure 4: Get API access token from Wandb

After passed successfully, the process of fine-tuning will be shown through a statistical table of steps and corresponding losses

[40/40 2:23:11, Epoch 8/10]

Step	Training Loss
1	1.922500
2	1.937000
3	1.585000
4	1.241400
5	1.109000
6	0.761800
7	0.673200
8	0.646000
9	0.614700
10	0.538400

Figure 5: Fine-tuning process

Finally, after finishing the fine-tuning process, the fine-tuned model will be saved to our private repository and essential files will be deployed to our public one.

4.3) Our fine-tuning strategy

As mentioned on the **Dataset** section, we fine-tuned on two datasets, one of them is the **Orca Math Dataset** supported by Microsoft. But the number of samples in this dataset is 200.000, an enormous number compared to our current hardware and software resources to train and test. So, to fine-tune effectively, we conducted **Continual Learning**. The idea of this strategy is that we will split the original dataset into smaller

parts, then we will train on each part, and to avoid **catastrophic forgetting**, we can implement as the below example:

Attempt 1: Fine-tune on part A in 4 epochs
Attempt 2: Fine-tune on part B in 4 epochs
Attempt 3: Fine-tune on part A + B in 1 - 2 epoch(s)

In conducting **Continual Learning**, we can experiment by modifying **learning rate**, for instance, in the ***attempt 3***, we can maintain the number of epochs and decrease the learning rate. There is no theoretical basis for implementing **Continual Learning** in the most effective way; therefore, we will need to experiment to find the optimal solution.

Until now, we fine-tuned on **the first 20.000 samples** of this dataset beside our created dataset. Our process of fine-tuning on the **Orca Math** dataset is as follows:

Attempt 1: Fine-tune on the first 2500 samples in 5 epochs, lr = 2.10^{-4}
Attempt 2: Fine-tune on the first 5000 samples in 1 epochs, lr = 2.10^{-4}
Attempt 3: Fine-tune on the first 5000 samples in 2 epochs, lr = 10^{-4}
Attempt 4: Fine-tune on the first 5000 samples in 1 epochs, lr = 2.10^{-4}
Attempt 5: Fine-tune on 5000 samples from 2500 to 7500 in 3 epochs, lr = 2.10^{-4}
Attempt 6: Fine-tune on the first 7500 samples in 2 epochs, lr = 2.10^{-4}
Attempt 7: Fine-tune on 7500 samples from 2500 to 10000 in 2 epochs, lr = 2.10^{-4}
Attempt 8: Fine-tune on 5000 samples from 5000 to 10000 in 3 epochs, lr = 2.10^{-4}
Attempt 9: Fine-tune on the first 10000 samples in 1 epochs, lr = 2.10^{-4}
Attempt 10: Fine-tune on 2500 samples from 10000 to 12500 in 5 epochs, lr = 2.10^{-4}
Attempt 11: Fine-tune on 2500 samples from 12500 to 15000 in 5 epochs, lr = 2.10^{-4}
Attempt 12: Fine-tune on 2500 samples from 15000 to 17500 in 5 epochs, lr = 2.10^{-4}
Attempt 13: Fine-tune on 2500 samples from 17500 to 20000 in 5 epochs, lr = 2.10^{-4}
Attempt 14: Fine-tune on 5000 samples from 10000 to 15000 in 2 epochs, lr = 2.10^{-4}
Attempt 15: Fine-tune on 5000 samples from 15000 to 20000 in 2 epochs, lr = 2.10^{-4}

The fine-tuning result of each attempt will saved on our **Wandb** account, in this report, we will show an result image as representation, remaining images will be stored in our Google Drive.

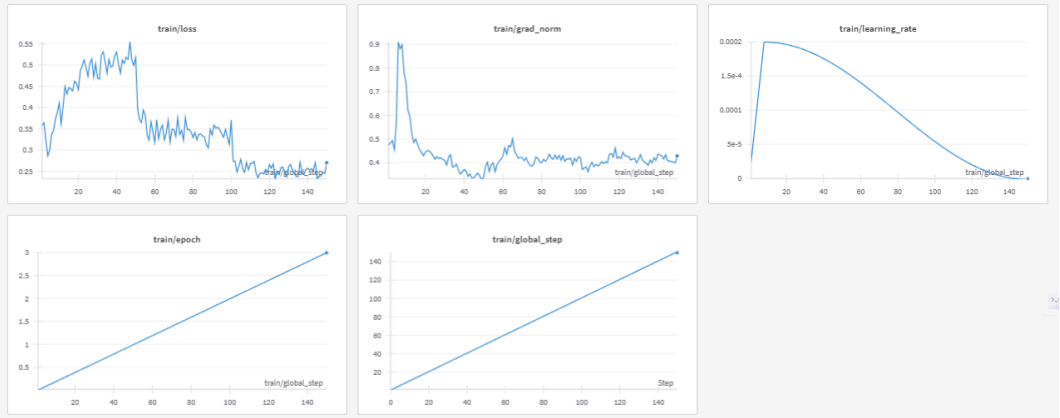


Figure 6: Fine-tuning result

5) Web Application for the Model

To applicate the model, we have constructed a web application which allows users to ask and get answers about mathematical problems. Furthermore, we also have developed an account system that allows user to save and share their chat history.

5.1) Project structure

The application consists of fours components: client side, server side, database, and chat bot service. The interaction flow is illustrated in the diagram below:

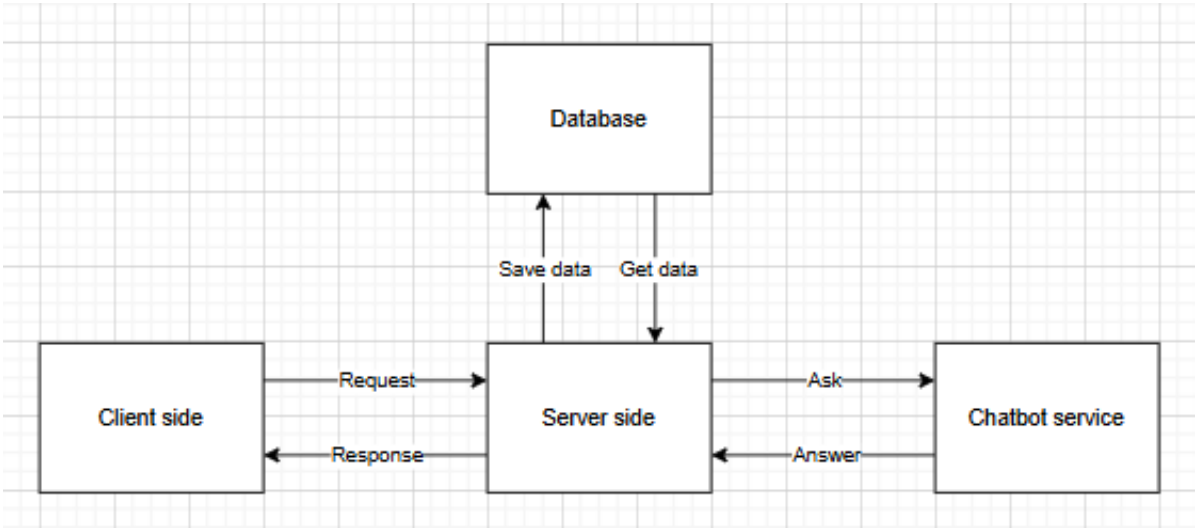


Figure 7: Interaction flow

5.1.1) Client side

The client side take responsibility for handling user events and rendering user interfaces.

We have implemented this component using Vue.js (a JavaScript framework for building user interfaces). The project structure consists of the following folders:

- **public**: Contains public resources.
- **src**: Contains the source code, which includes the following sub-folders:
  1. **views** and **components**: Contain function for UI rendering and user event handling.
  2. **router**: Contains the logic for routing and handling different pages on the web.
  3. **utils**: Stores helper functions to be used among the app.

5.1.2) Server side and Database

The server side is responsible for processing requests from the client, handling data, interacting with the database and the chat service. The database used is PostgreSQL which stores and manages all the application's data.

We have implemented this components using Express.js, with the project structure as below:

- **src**: Contains the source code, which includes the following sub-folders:
  1. **controllers**: Stores functions for handling different requests from Client side, each function correspond with an API route.

2. **routers**: Contains API route declarations.
3. **utils**: Stores helper functions to be used among the app.

- **prisma**: In this project, we have used Prisma (an ORM tool for Node.js and TypeScript) to working with database, this folder contains database configurations and data models

5.1.3) Chat bot service

The chat bot service is responsible for handling the chat logic. It exposes an API that accepts a question and chat history, processes the request, and responses the answer.

We have implemented this components using Flask for constructing API, Hugging Face API for loading and using the model.

5.2) User instructions

Here are the instructions for installing and running the application:

5.2.1) Client side

1. Go to client directory: `cd client`
2. Install packages: `npm install`
3. Create a **.env** file:

```
VITE_SERVER_URL="http://localhost:8000"
```

4. Run the client: `npm run dev`
5. The client then should be running at `http://localhost:5173`

5.2.2) Server side

1. Go to client directory: `cd server`
2. Install packages: `npm install`
3. Create a **.env** file:

```
PORT=8000
CLIENT_URL="http://localhost:5173"
CHAT_URL="http://localhost:5000"
DATABASE_URL="<your_postgresql_connect_url>"
JWT_KEY="<any_string>"
```

4. Run the server: `npm run dev`
5. The server then should be running at `http://localhost:8000`

5.2.3) Chat bot service

For devices with CUDA supported:

1. Go to client directory: `cd service`
2. Create python virtual environment: `python -m venv .venv`

3. Activate the virtual environment (for Window OS): `.venv\Scripts\activate`
4. Install packages: `pip install -r requirements.txt`
5. Run the service: `flask run --debug`
6. The service then should be running at `http://localhost:5000`

For devices without CUDA supported (Recommended running chatbot service with this method):

1. Create a new notebook in Kaggle on any browser.
2. Import the `chatbot.ipynb` file in `service` folder to the current notebook.
3. Set the accelerator to `GPU T4 x2` and turn on the `Internet` section.
4. Choose `Run all` and wait for the `NGROK` to generate the link to the service.
5. Copy the link and paste to the `.env` file in `Server` folder like this:

```
PORT=8000
CLIENT_URL="http://localhost:5173"
CHAT_URL="<NGROK generated link>"
DATABASE_URL="<your_postgresql_connect_url>"
JWT_KEY="<any_string>"
```

6. Run the server and client as usual.

5.3) The final product:

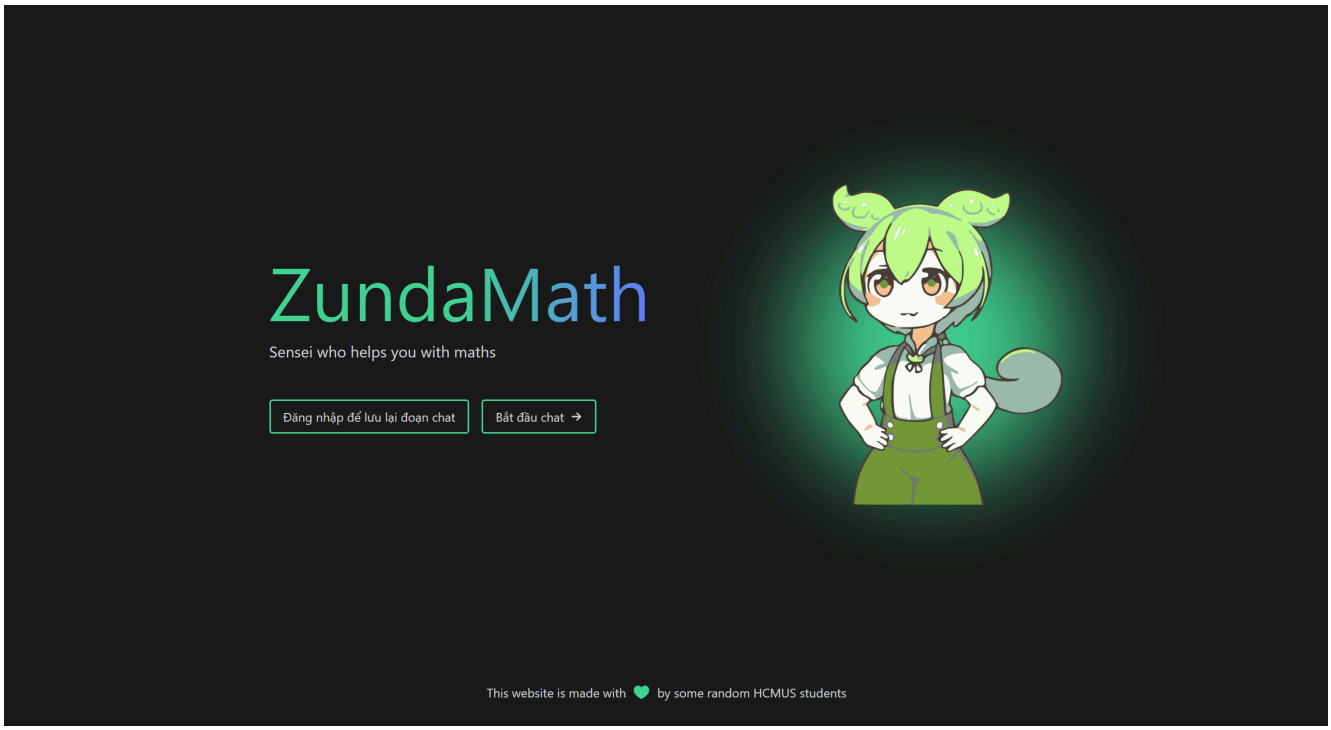


Figure 8: This is the landing page of our web application.

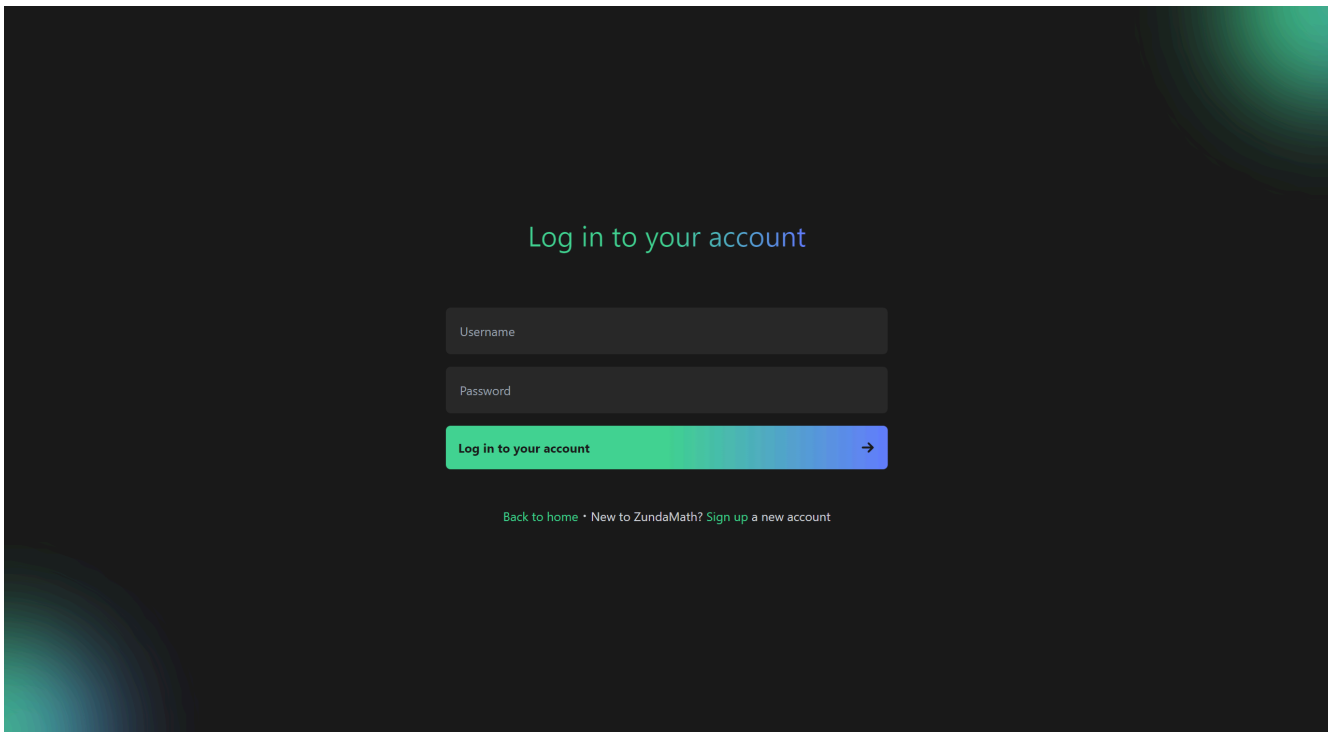


Figure 9: This is the login page.

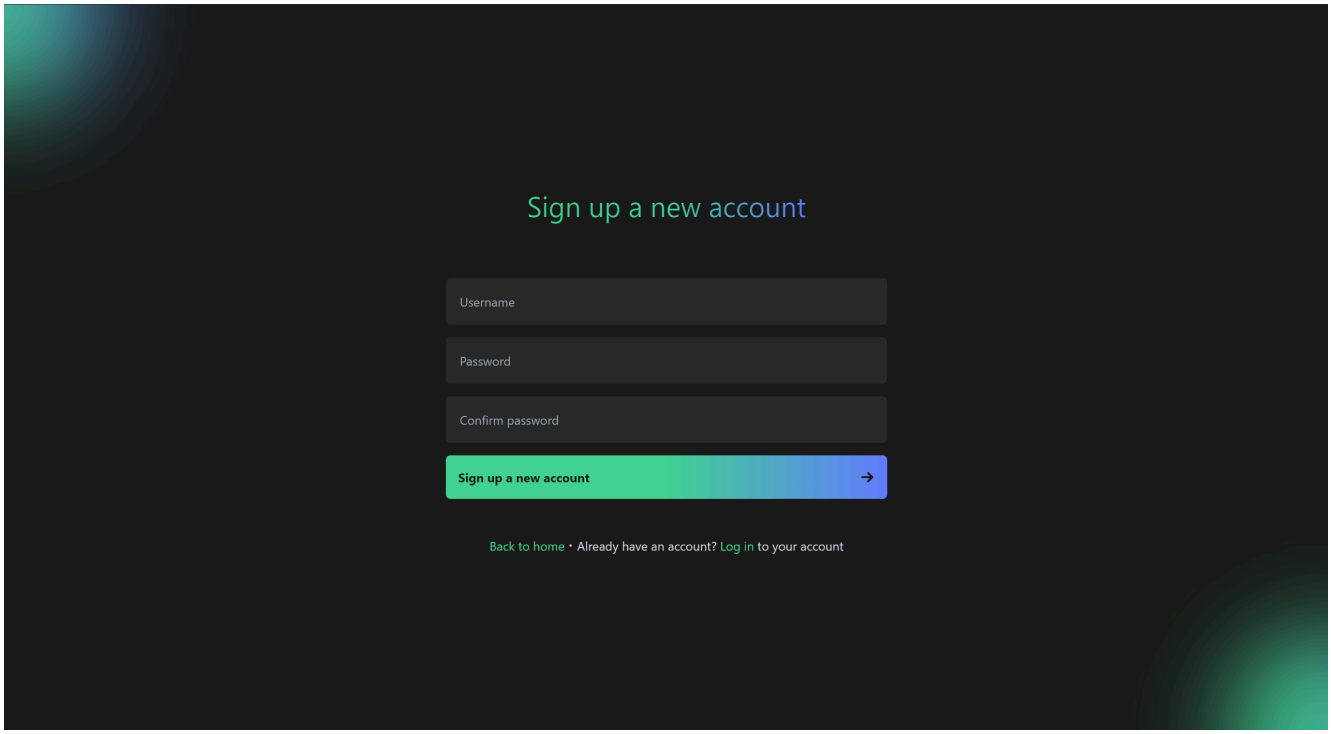


Figure 10: This is the signup page.

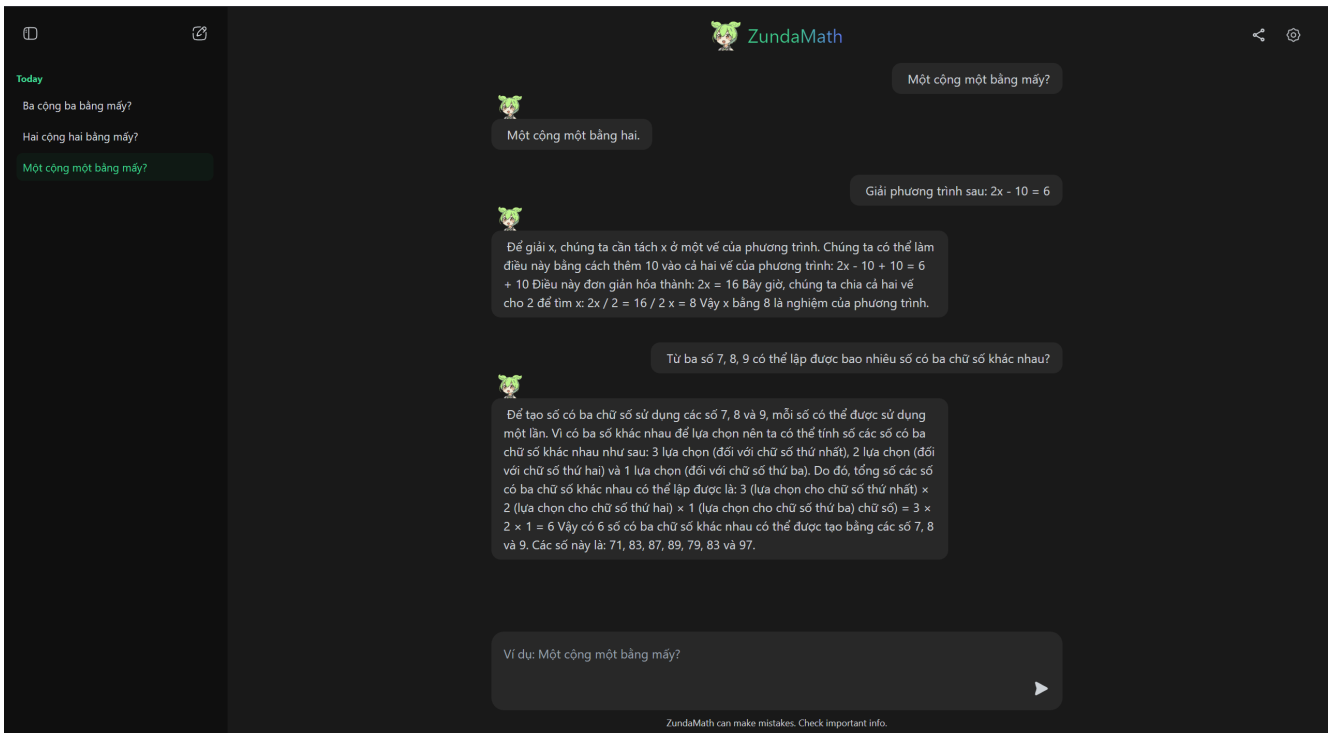
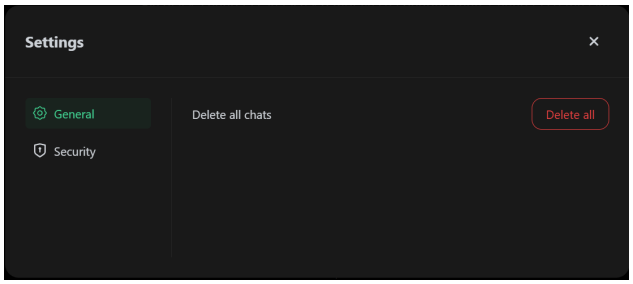


Figure 11: Chatting with the bot after login.
The conversation will be stored and can be re-opened from the right of the page.



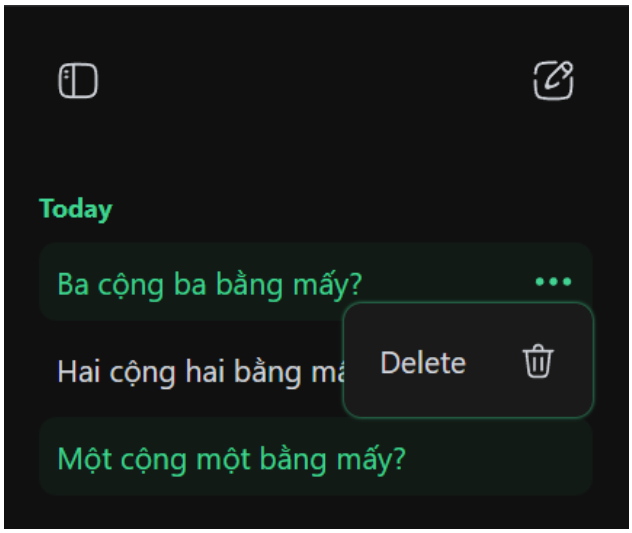


Figure 12: You can delete your chat log one by one or delete all.

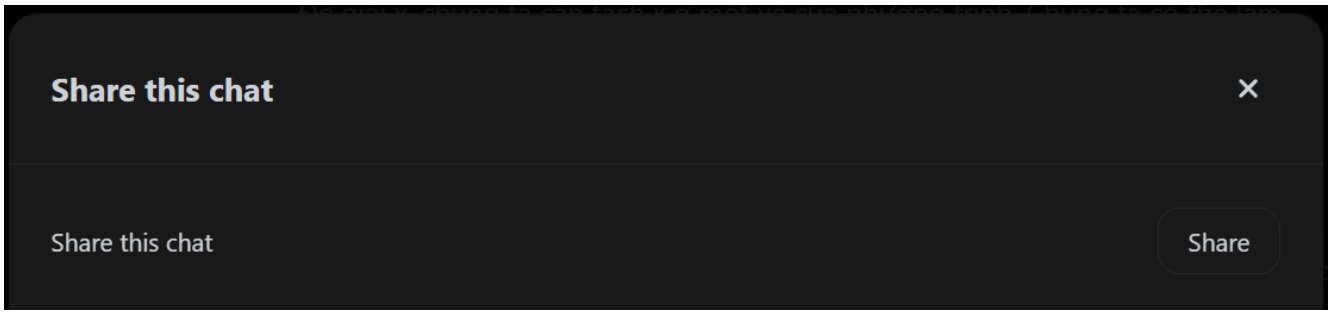


Figure 13: You can also share your chat log with other people.

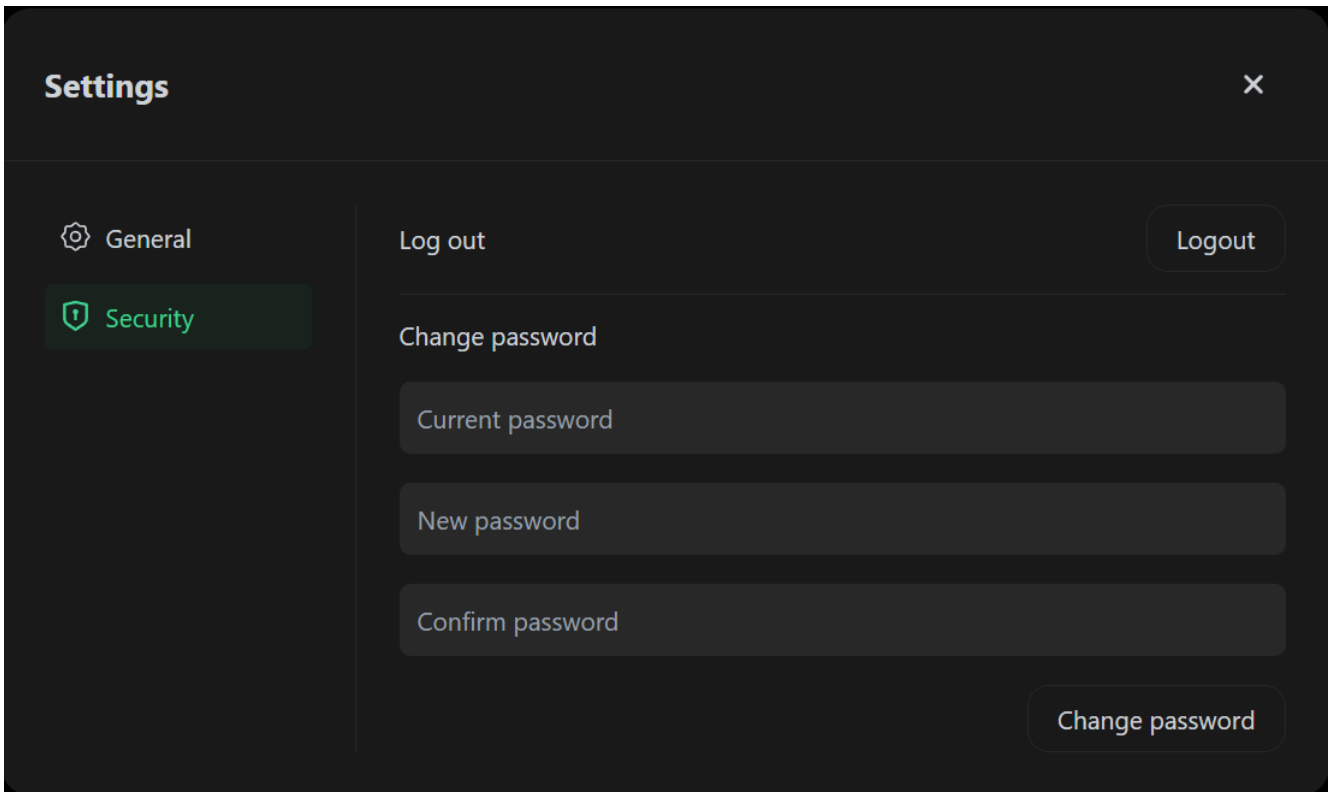


Figure 14: Here is where you logout from your current account or change your account's current password.

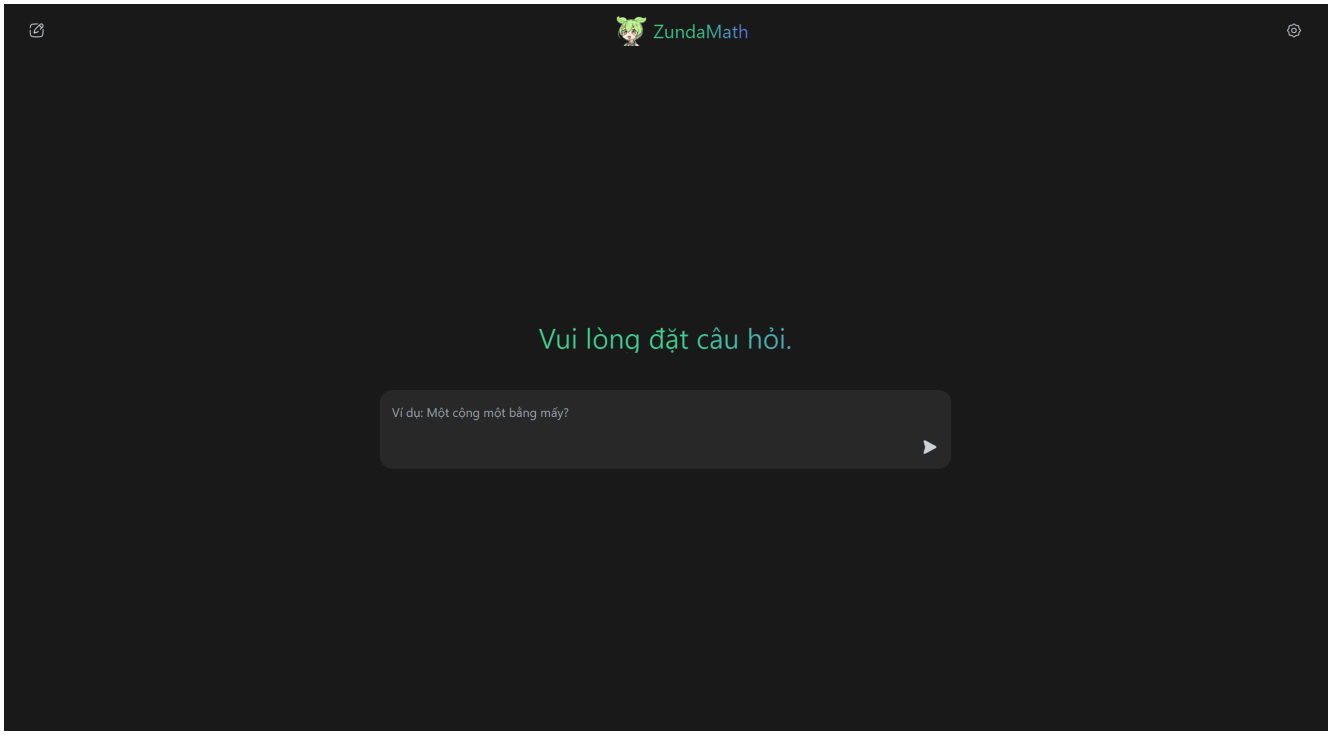


Figure 15: If you want a quick chat without login, you just need to click "Bắt đầu chat". However, your conversation will not be saved.

6) Statistics and Evaluation

Since this model is generative model, not classification or regression model, we cannot use the common metrics such as accuracy, F1-score, MSE, SSE, … Therefore, we might think about another metric for this problem to evaluate our model. After doing some research, we will use ROUGE Score as our main metric for justifying and evaluating our model.

6.1) What is ROUGE Score?

The ROUGE (Recall-Oriented Understudy for Gisting Evaluation) Score is a set of metrics widely used to evaluate the quality of automatically generated text, particularly in natural language processing (NLP) tasks like summarization, machine translation, or text generation. It measures how well the generated text overlaps with one or more reference texts (considered ground truth).

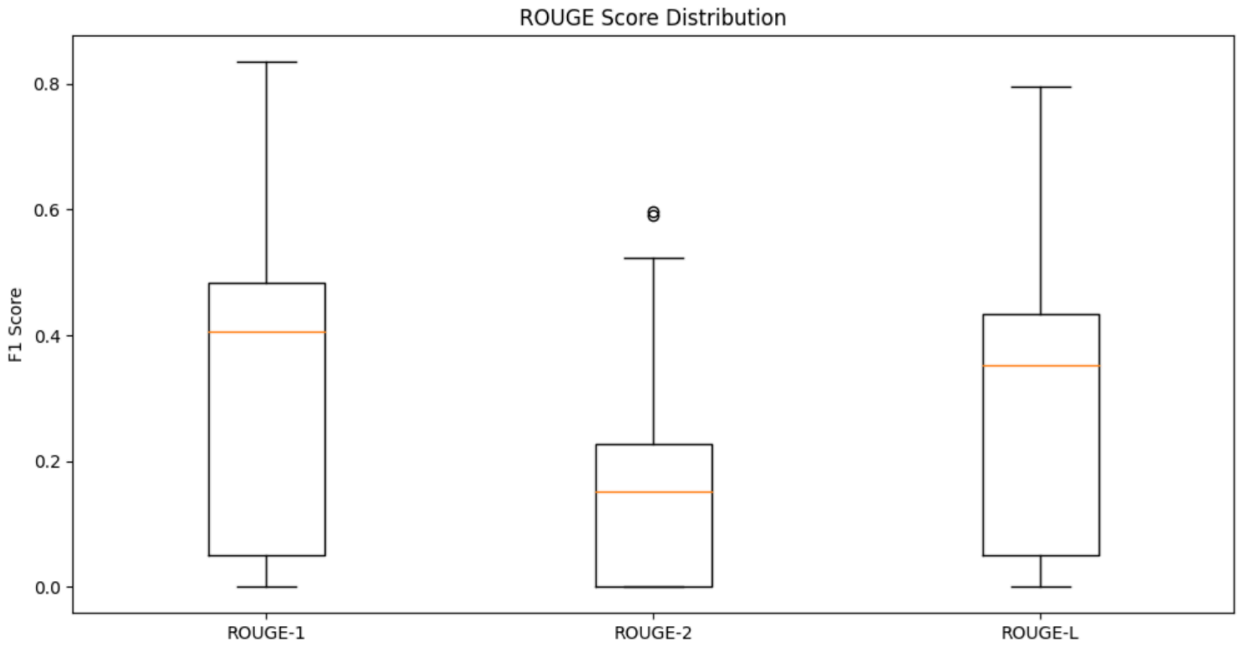
Key Variants of ROUGE:

- ROUGE-N: Measures the overlap of n-grams (sequences of n words) between the generated text and the reference text. For example: ROUGE-1 evaluates unigrams (single words), ROUGE-2 evaluates bigrams (two-word sequences).
- ROUGE-L: Measures the longest common subsequence (LCS) between the generated and reference texts. It captures sentence-level fluency and structure.
- ROUGE-W: A weighted version of ROUGE-L, giving more weight to longer contiguous matches.
- ROUGE-S: Evaluates skip-bigrams, which are pairs of words in the same order but not necessarily adjacent.
- ROUGE-SU: Extends ROUGE-S by including unigrams to provide a more comprehensive view.

Here, I will mainly use ROUGE-1, ROUGE-2 and ROUGE-L to measure our model. Although it is not a good metric for a Math LLM as this metric cannot decide whether the final answer is the correct answer, we still use it as we want to see if our model can clear and concise reasoning during solving the problem compared to the given answer from the test dataset. And since our model is based on elementary and secondary school math problem level, the answer is somewhat straight-forward so there is no need for checking if there is other way to solve the problem.

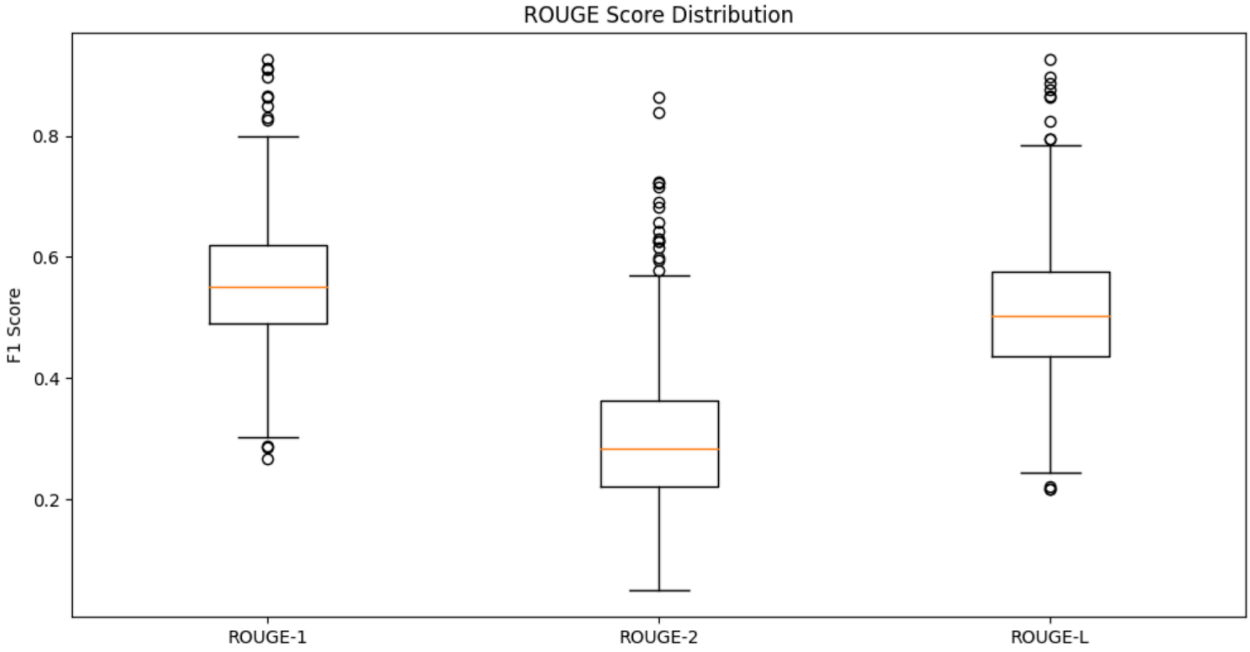
6.2) Experiments and Evaluation

For evaluating our model, we will use 1000 samples that has not been trained yet, then have it extracted and preprocessed from the #Orca Math Word Problems dataset to test on both the original model and our model. Here is the result that we got:




```
Average ROUGE-1: 0.3336277738464415
Average ROUGE-2: 0.14667775549937062
Average ROUGE-L: 0.2995828122122967
```

Figure 16: ROUGE Score on the original model




```
Average ROUGE-1: 0.5570828761313785
Average ROUGE-2: 0.30201642571343085
Average ROUGE-L: 0.5114550544613491
```

Figure 17: ROUGE Score on the our model

1. **Average ROUGE Scores**

ROUGE Type	**Original Model**	**Our Model**	**Improvement By (%)**
ROUGE-1	0.3336	0.5578	67.21%
ROUGE-2	0.1467	0.3020	105.86%
ROUGE-L	0.2995	0.5114	70.75%

Our model significantly outperforms the original model across all metrics (ROUGE-1, ROUGE-2, and ROUGE-L). This suggests that the second model is better at capturing lexical and structural overlap with reference answers.

2. **Score Distributions**

- Original Model:
  - Wider distribution for ROUGE-1 and ROUGE-L, with scores ranging from near 0 to above 0.6.
  - ROUGE-2 shows a tighter range, but scores are generally lower, with a few outliers.
- Our Model:
  - Higher median scores across all metrics.
  - ROUGE-1 has a narrower spread with fewer low scores, indicating better consistency.
  - ROUGE-2 and ROUGE-L also show a tighter and higher score range compared to the first model.

The second model demonstrates better consistency and performance across the board, with higher medians and narrower distributions.

Conclusion:

- Original Model: Likely generates less precise or complete outputs compared to the references, with more variability in performance.
- Our Model: Likely produces outputs that align more closely (lexically and structurally) with references, which might suggest better reasoning or explanation clarity.

7) Resource and Demo

- *Google Drive for storing source code, dataset and report*
- *MathLLM Demo Video*

8) References

- Microsoft. (2024). *Orca-Math: Unlocking the potential of SLMs in Grade School Math*
- Nguyen, Q., Pham, H., & Dao, D. (2023). *VinaLLaMA: LLaMA-based Vietnamese Foundation Model*
- VinBigData (2023). *VBD-LLaMA2-Chat - a Conversationally-tuned LLaMA2 for Vietnamese*