Generative AI for Software Development: Embedded Systems

Authors: El Alout Ismail, Haliloua Othmane, Lahmouz Zakaria, Karmaoui Oussama, El Habti Ouassel

3SN- HPC & BIG DATA

March 2024

Contents

Abstract

Large language models are known for their capacity to achieve a high accuracy in many generation tasks. This paper introduces a project leveraging generative AI to streamline the development of embedded systems software. We present the dataset building starting from the data collection, the transformations applied on this data and the final structure of the data used in fine-tuning. In addition, we present parameter efficient fine-tuning as a method that allows to optimize the fine-tuning of LLMs. Finally, we deploy our fine-tuned model as an interactive web-application with Streamlit.

Key words: Large language model, PEFT, LORA, fine-tuning, transformers.

1 Introduction:

The rise of generative AI has brought about significant progress in the IT realm, given its wide knowledge in different domains and the speed of its responses, this revolutionary technology ceases not to be developed, as it is flexible enough to provide room for tweaking and enhancing its performance in a specific domain. Therefore, we intend to use this technology to serve IRIT's interests in developing a generative AI application to help with the software development of embedded systems, for this subject relies on limited resources while imposing specialized functionality. This report delves into the project details and the steps taken to implement it, highlighting its cornerstones. Despite it not being complete, our functioning product serves as the primordial building block to continue constructing a self-maintained application fostering the AI services provided, which can be alleviated through automation and human supervision.

Our motivation:
Traditionally C coding with embedded systems has always proved to be labor-intensive and specific expertise, but we desire to help researchers go over these hurdles that slow down the processes of development. By harnessing generative AI methodologies, we aim to empower software engineers with tools that not only expedite development but also enhance the overall quality and performance of embedded systems software.

Our objectives:
Through this project we aim for:

- ***Fast development processes:*** Provide the user with fast, accurate, and well-organized written C codes in response to their coding requests.
- ***Adaptability:*** Designing an AI model capable of adapting to diverse embedded systems contexts, ensuring versatility and applicability across different domains.

- ***Easy access for users:*** Introducing a user-friendly interface to facilitate seamless interaction with AI-powered tools, fostering accessibility and usability for software engineers.

2 Methodology:

2.1 Project management:

Before setting foot to any coding steps, we initialized our project with an extensive bibliographic study about generative AI, ways to host them as a service, and the requirements for their setup (Data, hardware...). This study took slightly over a month under the supervision of our project manager.
As soon as an overall understanding of the direction to which we wanted to go was formed, we adopted a modular approach for development, designing a functional chain composing three distinct parts: Data, the Model, and The Interface. This approach allowed us to develop each component independently, ensuring parallel progress, until a later synchronization when every component is in place. To make sure that this approach would work, we insisted on the seamless functionality of each component and its interfacing with its subsequent step.
Hence, we benefited greatly, and we optimized efficiency, flexibility, and scalability, throughout the project lifecycle.
However, this required a meticulous follow up of all the updates, the new issues, and the new explorable horizons for our project, which was ensured by our use of Trello, where we set a workspace that served as a central hub for task management and collaboration. Trello provided us with a flexible and visual platform to organize our workflow, track progress, and allocate responsibilities effectively.

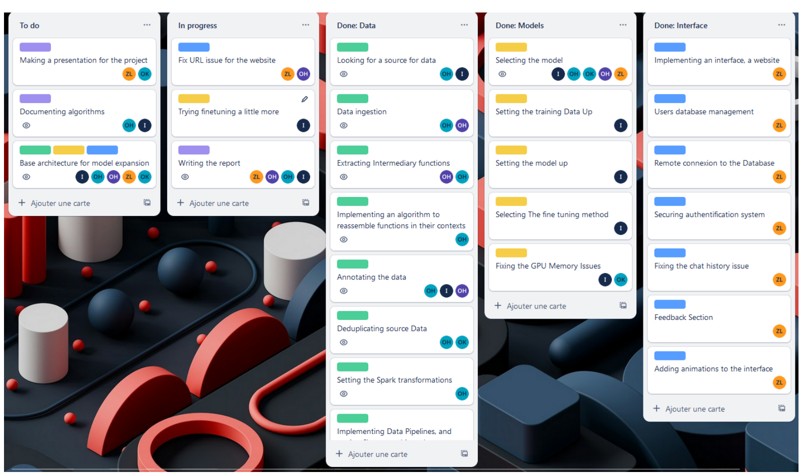


Figure 1: An illustration of what an API does in general.

2.2 Developement Modules:

2.2.1 Data:

It is about looking for a source of data, a way to interact with it and automate the data retrieval, and then setting the policy for data ingestion, so it could be passed to data pipelines where we set the transformations and actions necessary on data so it can be fed to the model.

2.2.2 Model:

And it is about selecting the appropriate model for our task, the finetuning method and the hardware requirements for it, as well as iterating the process to ameliorate the results and quality.

2.2.3 Interface:

And it is about setting an interface to host our software, the only changeable thing about it is the AI model, for progress in development a traditional GPT-3 model was used to put the whole chat and its history in place, the website styling, the user database, the access URL, and authentification mechanisms.

2.3 Developement tools:

2.3.1 Data:

- Python: Helped us automate the data retrieval process, the implementation of our data pipelines, and the data transformations.
- C++: Helped us accomplish the extraction of the intermediary functions from each parent code.
- Schikit-learn: Helped us implement a clustering algorithm, to put codes of the same structure in a cluster for deduplication.
- Spark: It was the mainstream accomodation for big data processing.
- Python APIs: Helped us interact with external application: GitHib, OpenAI.

2.3.2 Model:

- Python's Pytorch API: Helped us implement the finetuning algorithms.
- CUDA: A hardware driver to pilot our use of GPU for model training.

2.4 Interface:

- Python's Streamlit API: Building the web application.
- Postgre: For our web application's users database setup and management.

3 Dataset building:

As we are working to build a ChatGPT-like application which the user asks to provide him with a C coding solution for a given embedded systems context, the dataset envisioned is a two-column table: Prompt and table, the latter being the answer of the former. The continuity of this section explains how raw data were gathered to be later processed and treated through pipelines so that they would take the required form to be fed to our model's architecture.

3.1 Data scrapping:

3.1.1 C code files:

Of course, the first question that pops in mind for such a quest is, where can we get the data from ?
First of all, thankfully we have access to platforms for open source code like ***GitHub***. It is only logical to say that we can't rationally go through every bit of embedded systems C code and manually download it; Hence, the challenge is to automate the downloading process, this can be done through the
GitHub API (Application Programming Interface).

- The GitHub API provides endpoints for various functionalities, including searching for repositories, retrieving information about repositories, and downloading repository contents. With the GitHub API, we can programmatically search for repositories that contain embedded systems C code, retrieve metadata about these repositories, and download the code files.

- To start, we authenticated our requests to the GitHub API using personal access tokens. Once authenticated, we could make requests to the appropriate endpoints to search for repositories based on specific criteria such as programming language, and keywords, in our case: C language, Embedded Systems, or topics: Digital signal processing, Embedded Hardware, Real-time operational systems... all of which are subjects in the field of embedded systems.

- Once we identified relevant repositories, we can retrieve information about them, such as the repository name, owner, description, and other metadata. We can then select repositories that meet our criteria and proceed to download the code files contained within them.

- Downloading code files from GitHub repositories can be done using the Git version control system or by directly downloading the repository as a zip file. using the former, we could automate this process by making requests to the GitHub API to download the C files directly.

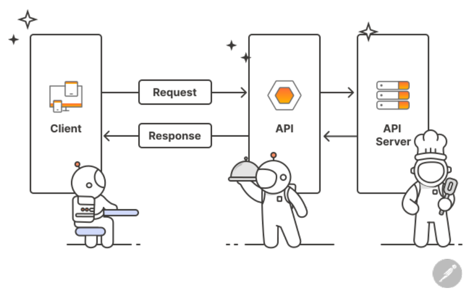


Figure 2: An illustration of what an API does in general.

In summary, data scraping for embedded systems C code can be achieved by leveraging the GitHub API to search for repositories, retrieve metadata, and download code files. With automation, we could efficiently collect thousands of C code files for embedded systems. Consequently,

3.1.2 Prompts:

Knowing the corresponding prompts for each C code retrieved requires expertise and an elaborate understanding of the overall domain in question which we lack. On the other hand, we couldn't find a person to supervise the construction of these prompts, so we turned to ChatGpt for this task. Yet again, we can not manually proceed with requests for thousands of C code files, therefore, we used the OpenAI API with the GPT 3.5 engine to generate the appropriate prompt for each C code, which would be further developed in the next sections.

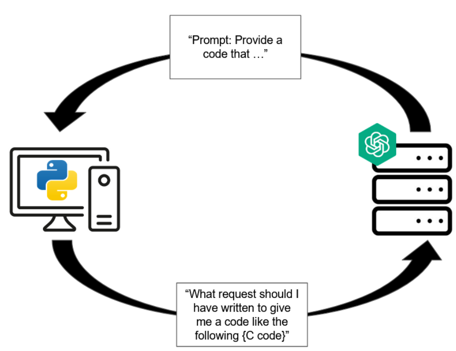


Figure 3: The flow of data via OpenAI's API.

3.2 Deriving more insights from data: Segmentation of Source Files into Functions and Comments

The creation of a robust dataset is pivotal for the fine-tuning of a Language Model aimed at generating high-quality embedded C code. This section delves into the methodology employed for extracting functions and their corresponding comments from a corpus of C source files for the supervised part. It highlights the inherent challenges, including those presented by custom-type definitions, and the rationale for utilizing Clang's LLVM framework. This derivation is essential because the user may be tasked with a specific use-case rather than a global one.

3.2.1 Challenges in Extracting Functions and Comments

Extracting functions and their respective comments from C code poses significant challenges, primarily due to the complexity and variability of the C language syntax. Traditional approaches using regular expressions or simple Python scripts fall short due to:

- The diversity of comment styles (single-line vs. multi-line) and their placements in relation to function definitions.
- Variations in function declaration and definition syntax, complicated further by custom type definitions, inline, static, and extern qualifiers, among others.
- The presence of preprocessor directives that can alter the interpretation of the code in a manner that is opaque to simple parsing strategies.
- The additional complexity introduced by custom type definitions, which can significantly vary in syntax and complicate the accurate identification of function signatures.

These challenges necessitate a more sophisticated analysis tool capable of understanding the syntax and semantics of C code, including the nuances introduced by custom types.

3.2.2 Adopting Clang's LLVM for Code Analysis

To address these challenges, we leveraged Clang's LLVM-based tools, benefiting from their advanced code analysis capabilities. Clang's LLVM framework offers:

- A rich Abstract Syntax Tree (AST) representation that accurately reflects the structure and semantics of the source code, enabling precise extraction of functions and comments, while effectively handling custom type definitions.
- Robust handling of the C language's complexity, including conditional compilation, macros, and the intricacies introduced by custom types, ensuring comprehensive analysis.

- The ability to hook into the compilation process, capturing comments and associating them with the corresponding functions through custom AST visitors and comment handlers.

The integration of Clang's LLVM tools into our dataset construction process allowed for the automated and accurate segmentation of source files into functions and their associated comments, forming the basis for the supervised part of training our LLM for enhanced performance in embedded C code generation.

3.2.3 Implementation Overview

Our implementation involves two main components: *main.cpp* and *CommentCapturer.cpp*. `main.cpp` orchestrates the traversal of directories containing C source files, invoking *ASTConsumer* on each file to extract functions and comments, taking into account the complexities posed by custom type definitions. *CommentCapturer.cpp*, on the other hand, defines the logic for parsing the AST, capturing comments, and associating them with their respective functions. This modular approach facilitates efficient processing and scalability of our dataset construction efforts, despite the added complexity of custom types.

The sophisticated dataset construction methodology employing Clang's LLVM framework significantly enhances the LLM's training process for embedded C code generation. By overcoming the limitations of simpler parsing methods and addressing the challenges posed by custom type definitions, we ensure the creation of a high-quality dataset that accurately reflects the nuances of embedded C programming. This paves the way for more intelligent and capable generative AI models in the domain of embedded systems development.

4 File System Architecture and Data Pipelines:

4.1 Overview:

This data-centric project, requires efficient management and processing of data. Data pipelines offer a structured method to handle the flow of data from the source to our downstream application *(Model Training, cf section 5 and 6)*, enabling seamless transformation. We shall delve into the design and the implementation of data pipelines within the context of our project: Data Ingestion, Data Preprocessing, Data Transformation, Data output.

4.2 Data Ingestion:

And refers to the process of collecting and importing data from various sources into a storage system. In our case, the source is the GitHub API, and the storage system is the local file system. That initially contains the folder ***from_source***, which is derived into many other folders, each one referring to the code files of a topic in the embedded systems realm (eg: DIgital signal processing, Embedded Networking ...). the parent folder is named ***data***

4.3 Data Preprocessing:

Data preprocessing is the initial stage of data preparation where raw data is cleaned, and organized to make it suitable to what comes ahead. Preprocessing took two steps:

1. Extracting intermediary functions from the parent codes using LLVM (seen above). This brought about a change in the topics folders. Each C file has become coupled with a folder that contains text files, each one containing the body of a function written in the parent code, named ***(c_file_name)_ext***.

2. Discarding the parent codes that already have a "twin" (code that has a similar structure) within the initial dataset using a hierarchical clustering over the contents of the parent files: First of all every code is converted to TF-IDF (Term Frequency-Inverse Document Frequency) vectors that are a numerical representation of text data that capture the importance of words in documents relative to a collection of documents, and then, It applies the agglomerative hierarchical clustering algorithm to the TF-IDF vectors. This algorithm recursively merges the closest pairs of clusters until a stopping criterion is met. In this case, the stopping criterion is defined by the distance_threshold parameter, which specifies the maximum distance between clusters for merging.
   After choosing just one element from each cluster we save the corresponding file names in a Spark RDD, that being for each topic, which would be saved in a folder with the appropriate topic name in the folder **raw**.

PS: We could have chosen the files representing each cluster before executing the extraction, but we found some problems automating that with C++ due to our limited expertise in the latter.

4.4 Data Transformation:

Data transformation is the process of converting data from its original format into a more structured, clean, and meaningful format suitable for analysis, visualization, or other downstream tasks. This process involves various operations such as filtering, aggregating, merging, and reshaping data to meet the specific requirements of an analysis or application. In the context of our project, we follow two steps using Pyspark for our big data transformations:

1. ***Hashing:*** after reading the files referred in the previously saved RDD, we attribute to every one of them a unique numerical key, therefore constructing a PairRDD, we stack the corresponding intermediary functions

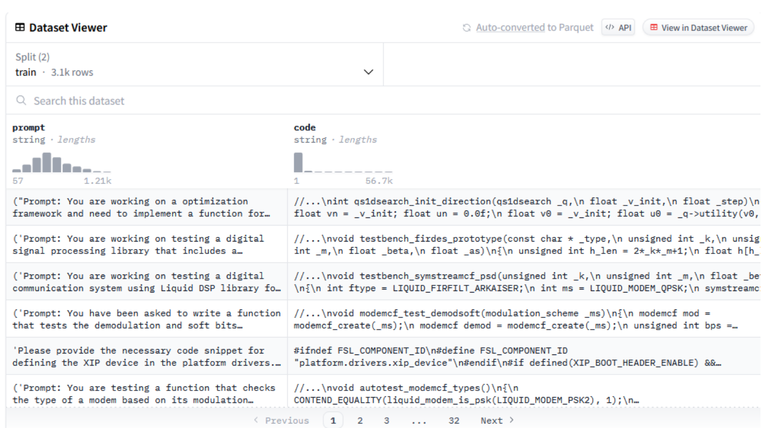
in a list to which we attribute the same numerical, getting with that another PairRDD doing that will help us complete the upcoming assembling transformation. In a folder named ***cleaned*** we create for each topic folder a folder called ***Hashing*** that contains the two PairRDDs. The hashing is useful because it saves us the trouble of using the direct names of the files which could create confusion, should some files have the same name.

2. ***Assembling:*** The body of intermediary functions isn't enough to fully define them, they could use other customized types, other intermediary functions, or global variables initially declared in the parent folder, hence, for each intermediary function we go back to the parent to extract this information that will later be added right before the body of the said function. We stack the parent codes alongside the reassembled codes, we clean the comments, extra white spaces, and empty lines for each code and we save them in an RDD, under the name ***assembled*** in the file system next to ***hashing***, and that for each topic. This is achieved by applying this transformation to the PairRDD containing the numerical keys and the list of intermediary functions' bodies.

3. ***Annotation:*** For each topic we read all the codes in an RDD to which we apply a transformation that generates for each code a tuple (prompt, code), for each topic we save the latter in a folder named ***Annotated***. The prompts are generated by feeding the code content to the ChatGpt API.

4.5 Data Output:

After annotating every assembled code and every parent code in the form of stacked tuples in text files, we created a two-column table (code, prompt) that was saved in a CSV format and stored in a Huggingface repository. This cleaned, structured, transformed and augmented dataset is what will be fed to our model finetuning algorithms.



4.6 An illustration of the Data Pipelines and the file system:

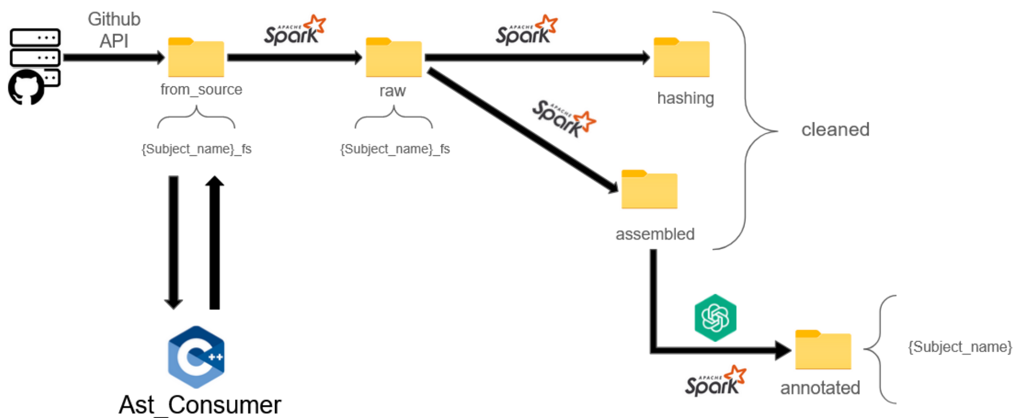


Figure 4: An illustration of the pipelines.

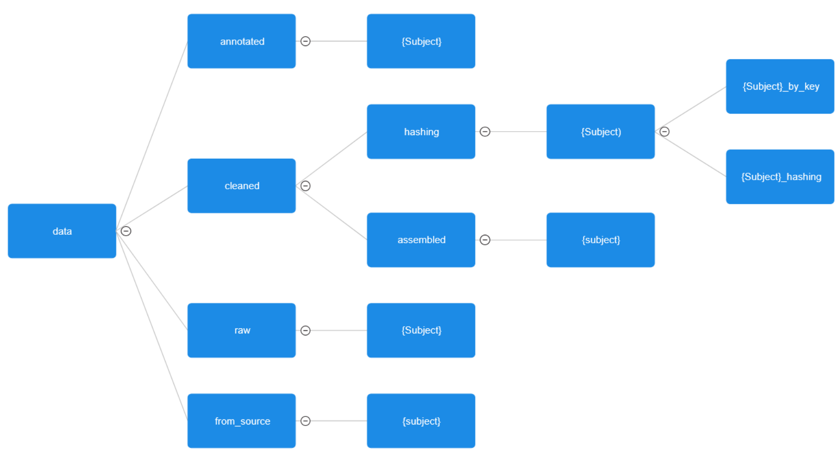


Figure 5: An illustration of the file system hierarchy.

4.7 Challenges Faced:

The task of implementing the data pipelines in coherence with the chosen file system architecture, presented manier complexities than it seems:

- ***Data ingestion:***
  - Despite the automation of the downloading process, the more complex the queries given to the API were, the fewer the numbers of repositories, hence, of the c codes downloadable, not to mention the disparity between the instances of data for every embedded systems topic that we could get our hands on, which would later cause bias, or even worse, false results.
  - The modular approach of searching for c codes by topic may have been useful in terms of organizing the search, but our unfamiliarity with the domain limited the scope of our processes, and expertise was needed for us to get more topics.
  - As we downloaded all the files we could get to download all the interface codes *(.h)* that would be coupled with other *.c* codes for they were needed later for extraction.
- ***Data preprocessing:***
  - For extraction, the soundest of strategies would be to integrate the deduplication process as a filter in the downloading stream, but we couldn't do it for it would have cost us a great deal of time in terms of its execution given the limited performance of our standard computers.
  - The extraction required the reading of the .h files that were missing on many occasions, which resulted the extraction in failure (no code extracted).
  - We couldn't find however hard we tried to get the equivalent software of the Clang LLVM extractor in C++ with Python, in case we could achieve it, we could have put the extractor as a transformation within spark that could enrich our data Python data pipelines for more programming flexibility.
- ***Data Transformation:***
  - The use of the open AI API for data annotation was very restrictive in terms of the limit number of requests per minute that it could accept, as we manipulated heavy data that traveled within these requests, we exceeded the limits on many occasions which caused the loss of our results. To overcome this we divided the data into chunks, which was laborious.

5 Models

5.1 The decoder-only architecture (GPT-2 architecture)

In order to understand the architecture of the models cited in the next paragraphs, we will first take a look at GPT-2's architecture. GPT-2 is built upon the Transformer architecture (Figure 6). However, GPT-2 only utilizes the decoder part (decoder-only) as it is primarily used for auto-regressive language modeling tasks where the output only depends on the previous tokens such as translation, question - answer, text completion... The decoder-only models take as input only the target sequence and generates the output based only on that input: First, Tokenization breaks down the input into discrete elements (words or characters which we call tokens). Next, each token is represented as a vector of real numbers in the embedding space. This representation help the model in capturing the semantic and syntactic information about the tokens and their relationships with other tokens. Also, positional encoding is used to provide information about the position of the tokens in the input sequence, which is very crucial for generation tasks and language understanding. The decoder-only architecture consists of multiple identical layers stacked on top of each others. Within each layer, multi-head self-attention mechanism allows the model to attend to relevant parts of the input by calculating attention scores between queries and keys which are used then to weigh the values [4]. Masking in these layers is employed to prevent the model from attending to future tokens in the sequence. This means that the model generates the output tokens based only on the tokens that precede them in the input sequence. The last layer calculates a probability, for each token in the vocabulary of tokens, for each position in the output sequence.

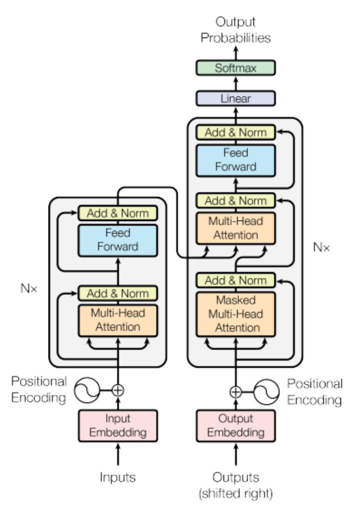


Figure 6: The Transformer architecture

PS: In the multi-head self attention mechanism, keys are representations of the input sequence used to determine the similarity of each element in the sequence to other elements. Values are representations of the input sequence that serve to construct the output sequence and Queries are representations of the elements in the output sequence for which the attention weights are computed. Each attention layer has its own set of keys, values and queries.

5.2 Model choice for fine-tuning

This is an overview of some potential models that we can choose for fine-tuning on our specific task. Our model choice is given and justified in the Fine-tuning section.

5.2.1 StarCoderBase

StarCoderBase is a 15.5B parameter code LLM trained on more than 80 programming languages built upon the GPT-2 model with multi-query attention (see [5]). Multi-query attention is similar to multi-head attention (see 4.1), the only difference is that all heads (attention layers) use a shared set of keys and values. The evaluation results show that StarCoderBase outperforms all open code LLMs.

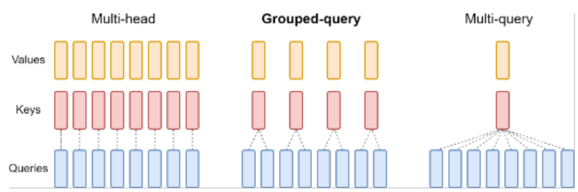


Figure 7: The difference between multi-head and multi-query attention

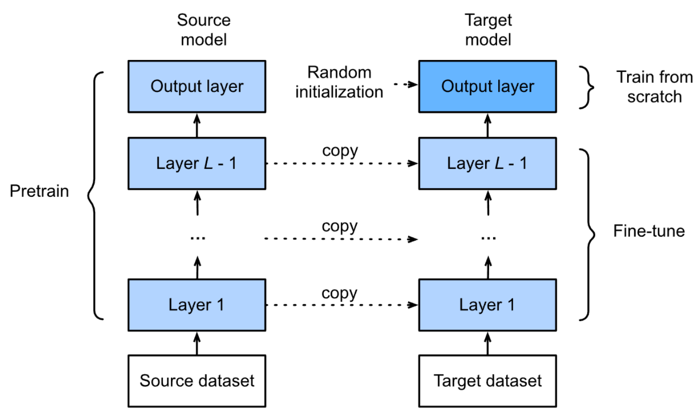
5.2.2 Llama-2

Llama-2 (released by meta) [6] is an auto regressive language family of models based on the transformer decoder architecture. Meta has released different versions of Llama-2 with different parameter sizes: 7B, 13B and 70B. Llama-2 models could be used to in several natural language processing tasks. It is true that it was not released mainly for code generation task. However, the evaluation results show that the Llama-2 models can answer programming questions and generate code sequences in many programming languages.

6 Fine-tuning

6.1 Introduction

Fine-tuning involves customizing a pre-trained extensive language model for a particular task. This is accomplished by training the model on a dataset which is specific to the task at hand, enhancing its performance in that area while retaining its overall knowledge base. Thus, after fine-tuning, our model should be capable of answering questions on our chosen task (code generation in embedded C).



6.2 Parameter efficient fine-tuning with LORA

The full model fine-tuning consists on training each layer of the model on the target dataset. However, this is very expensive computationally: if we consider a full fine-tuning with Adam Optimizer and a half-precision model, we will need to allocate 16 bytes (without counting the needed allocations for the intermediate hidden states) per parameter (see [8]). This means that for a simple model of 7B parameters (such as Llama-2), we will need to allocate 112GB.

Parameter efficient fine-tuning (PEFT) seek to address this challenge by reducing the number of trainable parameters while maintaining the same performance as a full fine-tuning. This is achieved by freezing some layers and only adjusting the weights of a smaller set of the last layers. Also, PEFT is an approach that uses a small dataset compared to datasets that are used in a full fine-tuning, which results in a smaller training time. This also helps in decreasing computational and storage costs and addresses the challenge of catastrophic forgetting (this is the case when a model forgets what it has learned in the pre-training phase) as only the weights of a small set of the layers are modified.

6.2.1 PEFT Adapters

Adapters represent a unique form of sub-module that can be integrated into pre-trained models to adjust their hidden representations. Thus, by incorporating adapters following the feed forward and the multi-head attention layers (Figure 8), it becomes feasible to refine only the parameters within the adapters while leaving the rest of the model unchanged. The integration of adapters can be implemented directly by embedding them into each transformer layer and situating a classifier atop the pre-trained model. Through the updating of the adapter and the classifier parameters, we can enhance the pre-trained model's performance for our specific task.

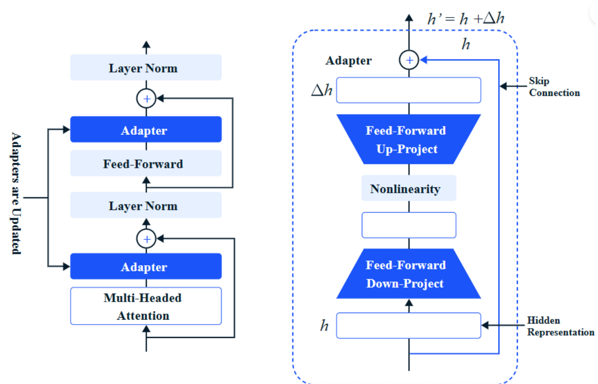


Figure 8: the model architecture with PEFT adapters

6.2.2 LORA adapter

Lora serves as trainable sub-module, compatible with the transformer architecture. It operates by preserving the weights of the pre-trained model and incorporating trainable rank decomposition matrices into each layer of the transformer structure: this process consists in decomposing a matrix into the product of matrices of lower rank (see Figure 9).

$$\begin{bmatrix} 5 & 1 & -1 & 3 & 4 \\ 15 & 3 & -3 & 9 & 12 \\ 35 & 7 & -7 & 21 & 28 \\ -20 & -4 & 4 & -12 & -16 \\ 10 & 2 & -2 & 6 & 8 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix} \times \begin{bmatrix} ? & ? & ? & ? & ? \end{bmatrix}$$

Figure 9: Matrix decomposition

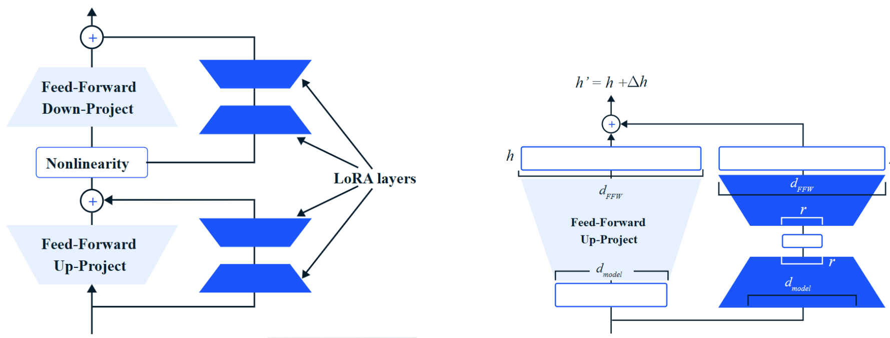


Figure 10: Insertion of LORA layers in the transformer architecture.

As illustrated in the Figure 10, LORA is integrated alongside the modules within the pre-trained model, aligned with the feed-forward layers. Each feed-forward layer comprises two projection layers separated by a non linear layer (see [9], [12]). The LORA layers are placed adjacent to both projection layers. In the right illustration (Figure 10), we can see that the LORA layer, which contains two feed-forward layers, takes the same input as the Feed-Forward Up-Project and then maps it into an r-dimensional vector with $r \ll d_{model}$. Then, this vector is mapped by the second feed-forward layer (of LORA layer) into a d_{FFW} -dimensional vector. This vector is then summed with the hidden representation of the model h in order to obtain the modified hidden representation h'.

As a conclusion, the LORA method allows to significantly reduce the number of trainable parameters, by using low rank decomposition matrices. Hence, this helps in lowering the computational complexity and memory requirements.

6.2.3 QLORA

QLORA is the quantized version of LORA. It aims to reduce memory usage while retaining model effectiveness. To delve more into the details of QLORA, the main difference from LORA is that it uses another numerical values representation: quantized 4-bit (instead of 8-bit used by LORA), when loading the pre-trained weights [10]. Weights are represented in a 4-bit quantization system, this means that weights undergo a process of reducing the precision of numerical values. But, they are dequantized (which is the opposite process, converting quantized values back to high precision form) during forward pass in order to maintain the accuracy. The main idea of QLORA allows to reduce memory access. Thus, it allows a faster computations during the fine-tuning.

6.3 Results of our fine-tuning

6.3.1 First attempt: 5 training epochs

The table below shows our training parameters settings for the fine-tuning: We were able to fine-tune our model only for 5 epochs (see 6.4 for more details about the reasons). Also, we chose such a batch size because a higher value requires more GPU VRAM (see 6.4).
The Lora_alpha parameter is used when adding back the weight changes into the base model weights, we multiply the weights by $\frac{Lora\_alpha}{r}$.
The Lora_dropout allows to mitigate overfitting of the model to the target dataset.It indicates the chance that a trainable parameter could be set to 0 in a particular training batch.

PS: Adaptive learning rate is used, it is initialized with 10^{-4}, but this value is dynamically adjusted during the fine-tuning based on the model's performance in this process.

Parameter	Value
Training epochs	5
Batch size	1
Gradient accumulation steps	2
Learning rate	initialized with 10^{-4}
Weight decay (for regularization)	0.001
Lora_alpha	16
Lora_dropout	0.1
r (the rank used in matrix decomposition of LoRA	64

Table 1: Fine-tuning settings

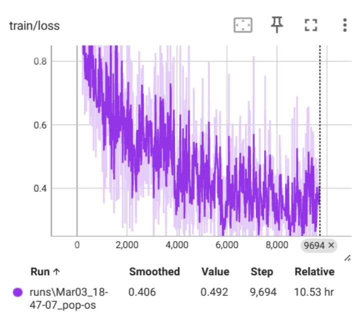


Figure 11: Evolution of the loss during the fine-tuning.

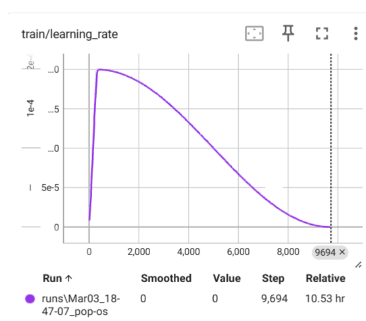


Figure 12: Evolution of the learning rate during the fine-tuning.

The figure 11 shows the evolution of the loss during the fine-tuning. The overall trend shows that the loss is decreasing over time, which means that the model is learning. The fluctuations in this case are normal as we chose a very small batch size (we could not test a higher batch size because of the limitations regarding the GPU VRAM that we used).

The figure 12 shows the evolution of the learning rate during the fine-tuning process, the initial spike allows the model to quickly adjust the weights based on the pre-trained knowledge. Then, the learning rate decreases gradually as it encounters our task-specific data, which allows the model to stabilize the training and prevent forgetting.

These results show that our model is effectively learning from our task-specific dataset. The loss reaches 0.49 after 5 epochs of fine-tuning (see 6.4 for more details about the reasons behind this number of training epochs). However, we want to precise that this is not a sufficient number of training epochs to achieve a high accuracy, especially with a 7B parameters model.
We can retain from the results of this fine-tuning that a higher accuracy can be achieved with more training epochs.

PS: Some examples of the answers outputted by the model after 5 epochs of fine-tuning are given in the Appendix section.

6.3.2 Second attempt: 15 epochs

As the results of the first fine-tuning attempt were not as accurate as the expectations, we tried to pursue the fine-tuning for another 15 epochs going from the fine-tuned model (5 epochs from the first attempt) as our new pre-trained model. This means that after this attempt, we will have 7B Llama-2 fine-tuned for 20 epochs on our task. We used the same settings as in the first attempt.

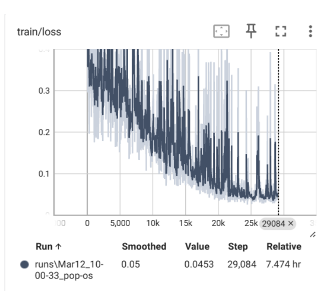


Figure 13: Evolution of the loss during the fine-tuning.

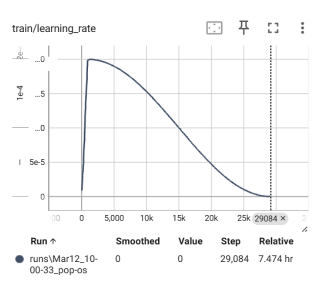


Figure 14: Evolution of the learning rate during the fine-tuning.

This second attempt lasted approximately 7 hours and 30 minutes. We were able to reach a training loss of 0.0453 (Figure 13). The decreasing rate of the training loss (Figure 13) indicates that the model is learning from our dataset. For the learning rate, as in the first attempt: the spike in the beginning allows the model to quickly adjust the weights based on the learned information from pre-training (the first attempt in this case) and then the gradual decrease allows to have a stable learning.

The generated content (see Appendix) after this second attempt improved a lot compared to the first attempt. Hence, we can conclude from this results that it is possible to achieve a better performance if we increase the number of training epochs. However, due to our hardware issues (precisely GPU), this is the best fine-tuned version of the Llama-2 7B model that we were able to achieve.

6.4 Challenges faced during fine-tuning

We faced many problems during fine-tuning, mainly related the GPU usage. As GPU is crucial for our fine-tuning task, we tried to use Google Colab which allows to have access to approximately 12 hours of GPU usage per day.
However, The free Google Colab GPU is a Tesla T4 16GB and our finetuning reaches its peak approximately at 22.8GB of the GPU VRAM. Thus, the only left solution was to pay for Colab pro in order to have access to a Tesla A100 GPU with 40GB.

However, this was not a perfect solution for our fine-tuning issues as even with Colab pro subscription, the number of compute units is limited (100 per month) and a usage of the Tesla A100 GPU consumes approximately 14 compute units per hour. We can also add that we faced several times runtime disconnection issues during our fine-tuning execution (due to servers overload), which basically means losing some compute units without obtaining any results.

In addition, we noted that our specific task (embedded C programming) is complex and requires more fine-tuning time. This can be explained by the big length of our dataset instances; as we tried to compare, in terms of execution time, our fine-tuning with the fine-tuning of the same model on another task with a dataset that was 6 times bigger than our dataset, and we found that the second task needed less time even with a bigger dataset.

7 Model deployment:

7.1 Streamlit:

We deploy the model in a web server using Streamlit cloud which is a platform provided by Streamlit that allows deploying, manage and share the application with the world for free. Streamlit is an open-source Python library that grants the creation of interactive web applications for data science and machine learning projects.
With Streamlit, we can write Python scripts that are automatically turned into web apps. It provides a simple and intuitive way to create user interfaces enabling data visualization, creating dashboards, and interacting with machine learning models without having to write HTML or JavaScript. Streamlit provides a variety of widgets for user interaction, including sliders, buttons, and text inputs.

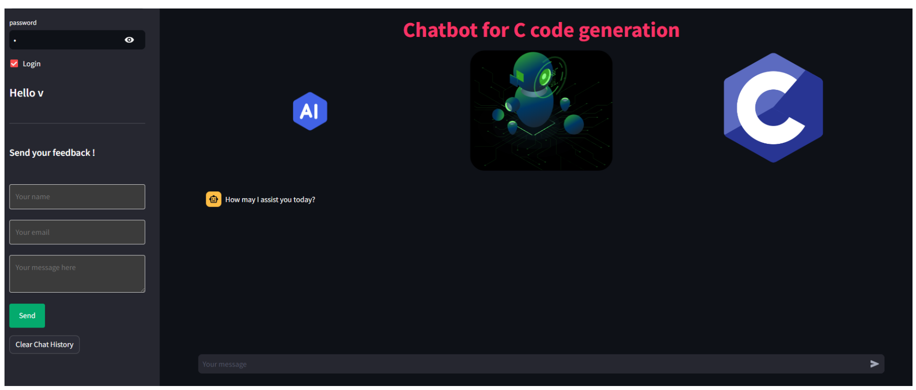


Figure 15: Our Web application interface.

When users first access the application, they are invited to either login or sign up for an account. The login/Signup process includes standard fields: username, email, and password. Upon successful authentication, users are directed to the main page of the application. The chatbot conversation is the central feature of the application, located prominently on the main page. Users can interact with the chatbot by typing messages in the input field and receiving responses in real time. A Lottie animation is also displayed on the main page to provide an engaging visual element, enhance the user experience, and add an element of delight to the interface. In the sidebar of the main page, there is a form for users to provide feedback. The feedback form includes fields for the user's name, email address, and a text area for users to write their comments. Upon submission, the feedback is sent to the appropriate team for review and action.

7.2 Features of the web application:

7.2.1 Database Management:

For users to access the chatbot interface, they must first sign up and log in. This necessitates the creation of a PostgreSQL database, either by connecting to an existing one or by creating a new database if none exists. Additionally, we must establish a cursor object to execute SQL queries, ensuring changes are committed and the connection is closed appropriately. During the sign-up process, we validate that the chosen username is not already present in the database. Upon successful verification, we collect the user's username and hash their password before storing this information securely in the database. Subsequently, during the login process, we verify the provided credentials against the stored data. Recognizing the importance of password security, we opt to employ the SHA-256 cryptography hashing algorithm to safeguard user passwords, thereby enhancing the overall security of the authentication system.

7.2.2 Chat history:

In our application, we store the conversation between the user and the assistant in a dictionary format. Each entry in the dictionary corresponds to a turn in the conversation, with the key indicating the role (either "user" or "assistant") and the value representing either the prompt provided by the user or the text generated by the model.
To manage this data across different parts of our Streamlit application and ensure its persistence throughout the user's session, we leverage Streamlit's 'session_state' attribute. This attribute serves as a dictionary-like object that enables us to store and retrieve data seamlessly. The data stored within 'session_state' remains accessible as long as the Streamlit session is active, allowing users to interact with various components of the application or navigate between different pages without losing context or progress in the conversation. This ensures a smooth and uninterrupted user experience, enhancing the overall usability and functionality of our chatbot interface.

7.2.3 Feedback of the application:

We decide to add the app review feature to improve the user experience functionality and overall performance by collecting feedback from users, analyzing their suggestions, criticism, and preferences, and then incorporating this information into iterative improvements of the application. Thus, we can identify recurring issues or complaints raised by users. These pain points might include bugs, usability issues, missing features, or performance issues. In addition, a user review can offer valuable feedback on the application's UI/UX design, including layout, navigation, visual elements, and overall user flow.
Reinforcement learning can be a powerful tool in this process, particularly for enhancing user engagement, customization, and decision-making within the application. For example, reinforcement learning techniques can be applied to

build sophisticated recommendation systems that personalize the user experience based on individual preferences and past interactions. By continuously learning from user feedback and behavior, these systems can adapt and improve their recommendations over time, leading to higher user engagement and satisfaction.

7.3 Challenges faced:

During the deployment of the interface, we encountered a series of challenges. Firstly, our efforts were impeded by the PostgreSQL firewall, which prevented us from establishing a connection to the database. To address this, we modified the 'listen_addresses' variable in the *'postgres.conf'* file to allow connections from any source, thus enabling access to the data. However, this adjustment revealed another obstacle: the free PostgreSQL account limits concurrent connections to five, leading to the frequent occurrence of the "too many connections" error, particularly during simultaneous access attempts. Consequently, we found it imperative to upgrade to a premium account to accommodate more concurrent connections. Furthermore, our attempts to deploy the model on Streamlit Cloud and Render were hindered by its substantial size (approximately 50 GB) and the requirement of a GPU with a minimum of 15 GB VRAM. This necessitated an upgrade to our Render account, incurring a monthly cost of approximately £400. Additionally, the code generation process experienced significant delays, with responses taking up to two minutes to generate. This latency stemmed from the loading of checkpoint shards within the model each time a user requested assistance, further highlighting the need for optimization measures.

8 What's the next step ?

8.1 Reinforcement Learning:

Reinforcement learning plays a crucial role in enhancing the effectiveness of chatbot applications, offering a dynamic approach to learning and decision-making. Unlike traditional rule-based or supervised learning methods, reinforcement learning enables chatbots to learn from interactions with users in real-time, continuously improving their performance and adaptability. This iterative learning process allows chatbots to understand user preferences, intents, and context, enabling them to deliver more personalized and relevant responses. By understanding reinforcement learning algorithms, chatbots can dynamically adjust their behavior based on feedback received from users, optimizing their conversational abilities over time.
Moreover, reinforcement learning enables chatbots to navigate complex and uncertain environments, allowing them to explore different strategies and learn from trial and error. This adaptive learning approach empowers chatbots to handle a wide range of user queries and scenarios, even in situations where predefined rules or training data may be insufficient. Additionally, reinforcement learning facilitates the discovery of optimal conversational strategies, enabling chatbots to achieve desired outcomes while minimizing errors and misunderstandings.
Furthermore, reinforcement learning enables chatbots to engage in long-term planning, enhancing their ability to help users to accomplish their objectives. By incorporating reward signals into the learning process, chatbots can learn to prioritize actions that lead to positive rewards, such as resolving user inquiries efficiently or providing helpful information. This results-driven approach enables chatbots to continuously refine their decision-making processes and optimize their performance in real-world applications.

This can be done by adding two networks:

- Actor-Critic network : This network push the fine-tuned LLM to generate a number of samples for a same prompt(input), then filter them by removing the samples that didn't pass a collection of syntax checking and unit tests and passing them to the repair network. The samples which passed the tests will receive a critic scoring that highlights their relevance.

- Repair network : This network evaluate the failed outputs taken by the agent (the actor-critic network) and try to refine them and correct potential on them based on desired outcomes and ethical guidelines.

Here is an instructive image taken the research paper [14] that shows the RL integration

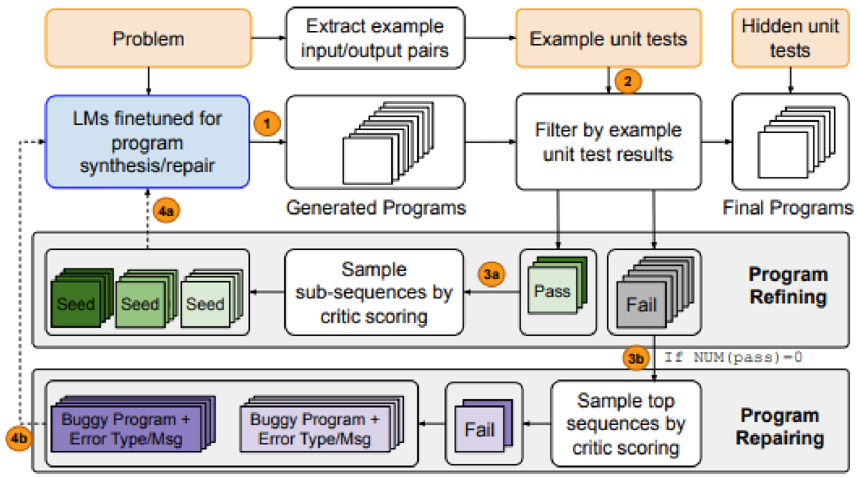


Figure 16: Critic sampling method.

8.2 Deploying the application in the cloud:

We suggest deploying our application in the cloud because it provides several advantages and revolutionizes the way businesses interact with their customers and streamline their operations.In fact,Cloud deployment provides scalability, allowing the chatbot to handle varying levels of user engagement without compromising performance. By leveraging cloud infrastructure, businesses can easily adjust computing resources based on demand, ensuring optimal responses even during peak usage periods. Moreover, deploying a chatbot in the cloud eliminates the need for on-premises hardware,and thus reducing the costs of hardware maintenance. This flexibility enables businesses to allocate resources more efficiently and focus on core competencies rather than infrastructure management.
Furthermore, cloud-based deployment facilitates the access to the application, enabling users to interact with the chatbot from any device with an internet connection. Whether it's through a web browser or a mobile application.Additionally, cloud deployment enables rapid deployment and updates, allowing businesses to roll out new features and improvements seamlessly. With cloud-based deployment models such as Platform as a Service (PaaS) or Containers as a Service (CaaS).
Security is another critical aspect of deploying a chatbot in the cloud. Leading cloud providers offer robust security features, including encryption, access controls, and threat detection mechanisms, to protect sensitive data and ensure

compliance with regulatory requirements. By leveraging cloud-based security solutions, businesses can mitigate risks associated with data breaches and cyber threats, enhancing trust and confidence among users.

When considering cloud providers for deploying chatbots, several options stand out. Amazon Web Services (AWS) offers several cloud services, including AI and machine learning tools like Amazon Lex, which can be used to build conversational interfaces efficiently. AWS provides robust security features, global scalability, and a pay-as-you-go pricing model, making it a popular choice for businesses of all sizes. Similarly, Microsoft Azure offers a wide range of AI services, including Azure Bot Service, which simplifies the development, deployment, and management of chatbots. Azure's integration with other Microsoft products and services, such as Microsoft Teams and Office 365, enhances collaboration and productivity. Google Cloud Platform (GCP) provides powerful AI and machine learning capabilities through services like Dialogflow, enabling businesses to create sophisticated chatbot experiences. GCP's global network infrastructure and advanced analytics tools make it a good environment for deploying chatbots that require real-time insights and scalability.

9 Conclusion:

Ultimately, this project has been a candid attempt from us to alleviate the standards of innovation in specialized laboratories, by refining the technology of large language models, creating an environment for them to be hosted in a user-friendly manner to provide access with no restriction. This product -though unfinished- is the harvest of a meticulous process of data engineering, model fine-tuning, and interface implementation, each playing a crucial role for that matter.

This project finds its landing on a robust data-driven policy, that is initiated by determining the protocols to structure the knowledge that is fetched code, followed by the operations that would help us process it, and the methods to derive further insights from it, setting the stage up for the final annotated format that gives data meaning regarding the fine-tuning process.

The latter is the heart of our endeavor, We tried to leverage state-of-the-art techniques, to ensure embedded systems C code is provided upon asking for the users of our application. This process involved trending and newly discovered methods that reduce computation cost, increase precision and performance, and provide fertile ground for expansion, reversibly and flexibly.

However, the added value of this model must reach its users, hence the importance of an interface, to this end we developed a website that streamlined the utilization of our model, making it accessible to developers and engineers working in embedded systems without an extensive background in natural language processing.

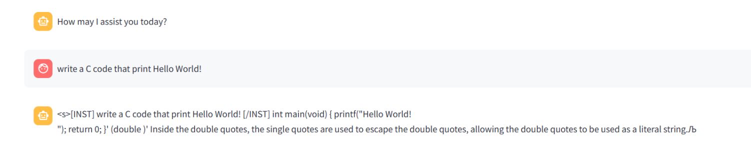
Through our collective efforts in data engineering, model fine-tuning, interface implementation, not only have we produced an embedded systems-focused AI, but a foundation for further expansion thanks to the architecture by which data is structured, the peculiar fine-tuning technique employed, and the flexible interface put in place.

To sum up, we recognize the difficulties we overcame, the information we gained, and the accomplishments we made as we look back on our journey. Our project serves as an example of the ability of creativity, tenacity, and teamwork to push the envelope of software development in general.

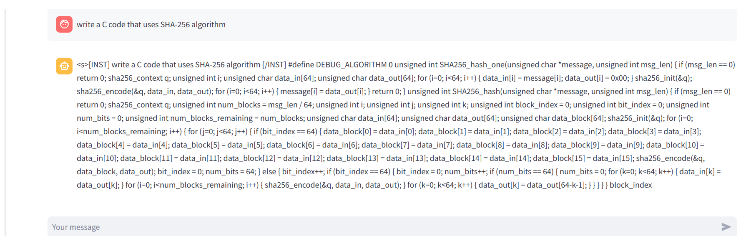
10 Appendix

10.1 Generated content after 5 training epochs

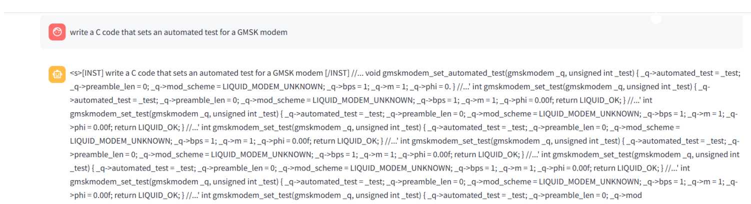
- First code snippet



- Second code snippet



- Third code snippet



10.2 Generated content after 20 training epochs

How may I assist you today?

write a C code that programs FSA key

<s>[INST] write a C code that programs FSA key [/INST] Programming an FSA (Finite State Automaton) key in C involves creating a finite state machine that recognizes the language of the FSA. Here is an example of how you could do this in C:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

// Structure to represent an FSA state
struct fsa_state {
    uint8_t state; // State number (0..n-1)
    uint8_t input; // Input symbol (0..255)
};

// Structure to represent an FSA
struct fsa {
    uint8_t n_states; // Number of states
    struct fsa_state states[]; // Array of state structures
};

// Function to create an FSA from a given FA
```

Your message

```
// Function to create an FSA from a given FA
struct fsa *fsa_from_fa(const fa_t fa) {
    // Initialize the FSA structure
    struct fsa *fsa = malloc(sizeof(struct fsa));
    fsa->n_states = fa->n_fa_states;
    fsa->states = malloc(sizeof(struct fsa_state) * fsa->n_states);

    // Populate the FSA states from the FA states
    for (uint8_t i = 0; i < fa->n_fa_states; i++) {
        struct fsa_state *state = &fsa->states[i];
        state->state = fa->fa_states[i];
        state->input = fa->fa_inputs[i];
    }

    return fsa;
}

// Function to recognize a string using an FSA
uint8_t fsa_recognize(const struct fsa *fsa, const uint8_t *input, uint8_t len) {
    uint8_t current = 0; // Current state
    uint8_t prev = 0; // Previous state

    // Initialize the recognition result to false
    uint8_t recognized = 0;

    // Loop over the input symbols
    for (uint8_t i = 0; i < len; i++) {
        // If the current state is the final state, set the recognition result
```

Your message

```
// Function to recognize a string using an FSA
uint8_t fsa_recognize(const struct fsa *fsa, const uint8_t *input, uint8_t len) {
    uint8_t current = 0; // Current state
    uint8_t prev = 0; // Previous state

    // Initialize the recognition result to false
    uint8_t recognized = 0;

    // Loop over the input symbols
    for (uint8_t i = 0; i < len; i++) {
        // If the current state is the final state, set the recognition result
        if (current == fsa->n_states - 1) recognized = 1;

        // Update the current state based on the input symbol and the previous state
        switch (input[i]) {
            case fsa->states[current].input: // Transition to the next state
                current = fsa->states[current].state;
                break;
            default: // Non-transitional input symbol
                recognized = 0; // Recognition failure
                break;
        }
        prev = current;
        current = fsa->states[current].state;
    }

    return recognized; // 1 if the string was recognized, 0 otherwise
}
```

Your message

```
int main() {
    // Create an FSA from a given FA using the fsa_from_fa function
    struct fsa *fsa = fsa_from_fa(fa_t{
        0, // Initial state
        2, // Number of input symbols
        {0, 1, 1}, // Transition table
        {'A', 'B', 'C'} // Input symbols
    });

    // Create an array of input symbols to test the FSA recognition
    uint8_t input[FA_INPUT_SIZE] = {0, 1, 1, 1, 0};

    // Recognize the input using the FSA
    uint8_t recognized = fsa_recognize(fsa, input, FA_INPUT_SIZE);
    printf("Recognized: %u
", recognized);

    // Clean up the FSA structure
    free(fsa->states);
    free(fsa);

    return 0;
}
```

This code defines a `struct fsa` to represent an FSA, with fields for the number of states and an array of `struct fsa_state` structures to represent the states. It also defines a `fsa_from_fa` function to create an FSA from a given FA, and a `fsa_recognize` function to recognize a string using the FSA.

Your message

11 Bibliography

[1] https://www.postman.com/what-is-an-api/

[2] Getting Started with Streamlit for Data Science: Create and deploy Streamlit web applications from scratch in Python:Tyler Richards

[3] Web Application Development with Streamlit: Develop and Deploy Secure and Scalable Web Applications to the Cloud Using a Pure Python framework: Mohammad Khorasani

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin: "Attention Is All You Need" https://arxiv.org/abs/1706.03762

[5] "StarCoder: may the source be with you!" https://arxiv.org/abs/2305.06161

[6] "Llama 2: Open Foundation and Fine-Tuned Chat Models" https://arxiv.org/abs/2307.09288

[7] "Fine-tuning in Deep Learning" https://ai.plainenglish.io/fine-tuning-in-deep-learning-909666d4c151

[8] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, Colin Raffel: "Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning "https://arxiv.org/abs/2205.05638

[9] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen: "LoRA: Low-Rank Adaptation of Large Language Models" https://arxiv.org/abs/2106.09685

[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, Luke Zettlemoyer: "QLoRA: Efficient Finetuning of Quantized LLMs" https://arxiv.org/abs/2305.14314

[11] https://www.leewayhertz.com/parameter-efficient-fine-tuning/

[12] https://www.entrypointai.com/blog/lora-fine-tuning/

[13] https://production-media.paperswithcode.com/methods/b0e96a0e-e3ad-4832-be7d-2fdce5f21352.png

[14] https://arxiv.org/pdf/2207.01780.pdf