PoC
On
Generative AI for Embedded Platform Code

Submitted by

KARTIK S

Under the guidance of

Nitin Dhingra

07/08/2023 – 17/11/2023

Table of Contents

1. INTRODUCTION 3
   A. Generative AI: 3
   B. Transformers: 3
   C. Large Language Models: 4
   D. Hugging Face: 4
2. APPROACH TO THE PROBLEM STATEMENT 5
   A. Model selection: 5
   B. Environment Used: 6
   C. Framework/Libraries used: 7
   D. Dataset Creation: 8
   E. Training/Fine-Tuning: 9
   F. Inferencing The Model: 12
   G. Output From the Model after Finetuning: 16
3. Challenging Areas: 21
4. Steps Ahead 23
5. Conclusion 24

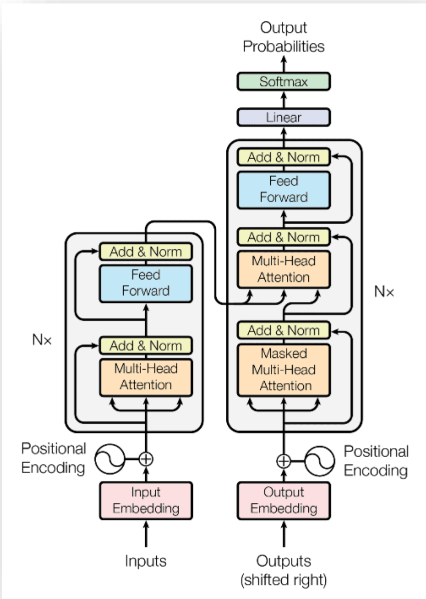
1. INTRODUCTION

A. Generative AI:

- Generative AI is a type of artificial intelligence technology that can produce various types of content, including text, imagery, audio.
- The recent buzz around generative AI has been driven by the simplicity for creating high-quality text, graphics, and videos in a matter of seconds.

B. Transformers:

- The recent advances that have played a critical part in generative AI going mainstream are transformers and the breakthrough of large language models that they enabled.
- Transformers are a type of machine learning models that made it possible for researchers to train large models easily. New models could thus be trained on billions of pages of text, resulting in answers with more depth.
- In addition, transformers unlocked a new notion called attention that enabled models to track the connections between words across pages, chapters, and books rather than just in individual sentences and not just words Transformers could also use their ability to track connections to analyze code.
- Like most neural networks, transformer models are basically large encoder/decoder blocks that process data.



- the task of the encoder, on the left half of the Transformer architecture, is to map an input sequence to a sequence of continuous representations, which is then fed into a decoder.
- The decoder, on the right half of the architecture, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence. With these tools, transformers can see the same patterns humans see.

C. <u>Large Language Models:</u>

- The rapid advances in large language models (LLMs) i.e., models with billions or even trillions of parameters have opened a new era in which generative AI models can do a variety of generation tasks.
- Large language models are a type of generative AI that are trained on text and produce textual content. It is a deep learning model that can perform a variety of natural language processing tasks.
- These models mostly use transformer models and are trained using massive datasets hence, large. This enables them to recognize, translate, predict, or generate text.
- In addition to teaching human languages to large language models, they can also be trained to perform tasks like writing software code.
- Large language models must be pre-trained and then fine-tuned to get the desired output from them for a specific downstream task.

D. <u>Hugging Face:</u>

- Hugging Face is a machine learning and data science platform and community that helps users build, deploy and train machine learning models.
- It provides the infrastructure to demo, run and deploy artificial intelligence in live applications. We can also browse through models and data sets that other people have uploaded.
- Hugging Face is known for its Transformers python library, which simplifies the process of downloading and training ML models. The library gives developers an efficient way to include one of the ML models hosted on Hugging Face in their workflow and create ML pipelines.
- The platform is important because of its open-source nature and deployment tools. It allows users to share resources, models, and research and to reduce model training time, resource consumption.

2. APPROACH TO THE PROBLEM STATEMENT

The objective of this PoC was to create a generative AI chat model that can convert natural language to code i.e., understand natural language as input and generate the required embedded platform code as the output. To achieve this, we had to finetune a pre-trained model on our dataset, as making a model from scratch was a very tedious task, we proceeded in the following steps:

A. Model selection:

- We wanted a pre-trained AI model that was already trained with large data, and that could understand English language easily. The other requirement was the model should be able to generate some basic c/c++ code by itself.
- We had a look at some of the models like Falcon 7-b, wizardcoder, and Llama2-7b.
- Falcon 7-b model was good at text generation tasks, but was not trained on any basic programming language and hence was not able to generate any code.
- Wizardcoder on the other hand was specifically designed for coding purposes, but the size of the base model itself was around 100gb, which made it difficult to use, as it was a very resource intensive model and was also not easy to access.
- Llama 2 on the other hand was good at text generation tasks as well as it was able to generate some basic code from the pre-trained knowledge it already had.
  - ✓ Llama 2 was also fine-tuned on a code dataset to create Codellama, an AI model that can generate code.
  - ✓ The integration of Llama 2 with hugging face made it easy to access and use.
  - ✓ Hugging face also provided frameworks and libraries to inference, train and deploy fine-tuned LLM’s
  - ✓ Llama 2 is also open source and can be used commercially.

Model	Size	Code	Commonsense Reasoning	World Knowledge	Reading Comprehension	Math	MMLU	BBH	AGI Eval
MPT	7B	20.5	57.4	41.0	57.5	4.9	26.8	31.0	23.5
	30B	28.9	64.9	50.0	64.7	9.1	46.9	38.0	33.8
Falcon	7B	5.6	56.1	42.8	36.0	4.6	26.2	28.0	21.2
	40B	15.2	69.2	56.7	65.7	12.6	55.4	37.1	37.0
LLAMA 1	7B	14.1	60.8	46.2	58.5	6.95	35.1	30.3	23.9
	13B	18.9	66.1	52.6	62.3	10.9	46.9	37.0	33.9
	33B	26.0	70.0	58.4	67.6	21.4	57.8	39.8	41.7
	65B	30.7	70.7	60.5	68.6	30.8	63.4	43.5	47.6
LLAMA 2	7B	16.8	63.9	48.9	61.3	14.6	45.3	32.6	29.3
	13B	24.5	66.9	55.4	65.8	28.7	54.8	39.4	39.1
	34B	27.8	69.9	58.7	68.0	24.2	62.6	44.1	43.4
	70B	**37.5**	**71.9**	**63.6**	**69.4**	**35.2**	**68.9**	**51.2**	**54.2**

Table 3: Overall performance on grouped academic benchmarks compared to open-source base models.

Considering the above factors, we finalized on LLma-2-7b model as our base model that we will be fine tuning further to achieve the result.

B. <u>Environment Used:</u>

- DGX is a line of servers and workstations built by NVIDIA, which can run large, demanding machine learning and deep learning workloads on GPUs.
- DGX provides a massive amount of computing power between 1-5 PetaFLOPS in one DGX system.
- It also provides advanced technology for interlinking GPUs and enabling massive parallelization across thousands of GPU cores.
- Beyond the powerful hardware they provide, DGX systems come out of the box with an optimized operating system and a complete pre-integrated environment for running deep learning projects.
- The configuration of the DGX Server used:

GPUs	8x NVIDIA Tesla V100
GPU Memory	Total of 256 GB
CPU	Dual 20 Core Intel Xeon E5-2698 v4 2.2GHz
CUDA Cores	40,960
Tensor Cores	5,120
Power	3500W
System RAM	512 GB 2133 MHz DDR4 RDIMM
Networking	Dual 10 GB Ethernet
Operating System	Canonical Ubuntu or Red Hat Enterprise Linux (RHEL)

C. Framework/Libraries used:

- **Python**, the entire code for this programming language was written in python programming language.
- **Hugging face hub**, it provides Tools that enable users to build, train and deploy ML models based on open-source code and technologies.
    - ❖ They contain different pre-trained models that we can use vary easily.
    - ❖ The also contain large amounts of open-source dataset for different down-stream tasks that can be used.
    - ❖ We can also upload our fine-tuned model to the hub and use it easily through the different tools they provide.
- **Transformers**, it is an open-source framework for deep learning created by Hugging Face. It provides APIs and tools to download state-of-the-art pre-trained models and further tune them to maximize performance.
- **PyTorch**, it is an open-source machine learning library used for developing and training neural network based deep learning models.
- **Datasets**, it is a lightweight library providing efficient data pre-processing for public datasets as well as your own local datasets in CSV, JSON, text, PNG, JPEG, WAV, MP3, Parquet, etc.
- **PEFT**, or Parameter-Efficient Fine-Tuning (PEFT), is a library for efficiently adapting pre-trained language models (PLMs) to various downstream applications without fine-tuning all the model's parameters.
- **TRL** is a full stack library where we provide a set of tools to train transformer language model, the specific trained used from this library for this PoC is Supervised Fine-tuning (SFT) trainer. The library is integrated with transformers.
- **Streamlit**, is a open-source Python library, which enables developers to build attractive user interfaces in no time.

D. Dataset Creation:

- The next main task was to create the dataset for the specific downstream task that we were going to fine tune the LLM for i.e generating embedded platform code.
- For this, dataset was created in a .jsonl format with "prompt" as one key in the dictionary and "completion" as the other key in the dictionary. Each row in a dictionary were in the form of prompt and completion pairs.
- The prompt describes the respective completion in the dictionary.
- E.g. {prompt: "python code to print the string Hello", completion: "print('hello')"}, this is considered to be 1 row in the dataset, like this, multiple rows of prompt and completion pairs were created to fine tune the model.
- For embedded code we broke down the main code into small functions, and we created 1 row for each function in the dataset with clear explanation of that piece of code in the "prompt" field.
- We created 125 rows of data for getting the output for this project which had all the necessary details of code for creating a basic embedded platform code.
- The dataset needs to be then formatted for the model to understand, so we used the below code to format the dataset, after this code is run and a new .jsonl file is created which is the final dataset that will be given to the model for fine-tuning.
- Code for formatting dataset(dataformat.py):

```
import jsonlines

# jsonl input and output paths
jsonl_input_paths = ["finaldataset.jsonl"]
jsonl_output_path = "f2i.jsonl"
def format_instruction(sample):
    return f"""### Instruction:
You are a coding assistant that will write a Solution to resolve the following Task:

### Task:
{sample['prompt']}

### Solution:
{sample['completion']}
""".strip()

with open(jsonl_output_path, "w") as output_file:
    json_writer = jsonlines.Writer(output_file)

    for jsonl_input_path in jsonl_input_paths:
        with open(jsonl_input_path, "r") as input_file:
            json_reader = jsonlines.Reader(input_file)

            for entry in json_reader:
                try:
                    # Combine prompt and completion into the desired format
                    formatted_text = format_instruction(entry)

                    # Write the formatted text as a JSONL line
                    json_writer.write({"text": formatted_text})
                except Exception as e:
                    print(f"Error parsing JSON in {jsonl_input_path}:", entry, e)
```

E. Training/Fine-Tuning:

Once the dataset is created, we must fine-tune the LLM on this dataset.

- Fine-tuning is taking a pre-trained model and training at least one internal model parameter.
- Parameters are the values/variables that a model learns during training to make predictions or classifications on new data. Parameters are usually represented as weights and biases in neural networks, and they control how the input data is transformed into output predictions.
- For our project the most suitable approach is to use supervised learning. This involves providing the model with a dataset, where each data point is a pair of prompt and completion. The model learns to map the input to the output by minimizing a loss function.
- Parameter-efficient Fine-tuning (PEFT) is a supervised learning technique used in (NLP) to improve the performance of pre-trained language models on specific downstream tasks. It involves reusing the pre-trained model's parameters and fine-tuning them on a smaller dataset, which saves computational resources and time compared to training the entire model from scratch.
- PEFT achieves this efficiency by freezing some of the layers of the pre-trained model and only fine-tuning the last few layers that are specific to the downstream task.
- This way, the model can be adapted to new tasks with less computational overhead and fewer labelled examples.
- We have used LoRA method under PEFT library for our fine-tuning process.
- LoRA is an innovative technique designed to efficiently fine-tune pre-trained language models by injecting trainable low-rank matrices into each layer of the Transformer architecture. LoRA aims to reduce the number of trainable parameters and the computational burden while maintaining or improving the model's performance on downstream tasks.
- With the help of LoRA the models can be trained very quickly as only some parameters in the model are updated, Lora Config is used to set the lora parameters for finetuning

- Code for Fine-Tuning(train.py):

```
import torch
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    HfArgumentParser,
    TrainingArguments,
    pipeline,
)
from peft import LoraConfig, PeftModel
from trl import SFTTrainer
from datasets import load_dataset


# The model that you want to train from the Hugging Face hub
model_name = "meta-llama/Llama-2-7b-hf"

# The instruction dataset to use
dataset_name = "final2i.jsonl"

# Fine-tuned model name
new_model = "7b"

# Output directory where the model predictions and checkpoints will be stored
output_dir = "./results"

# Number of training epochs
num_train_epochs = 8

# Load base model
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    # use the gpu
    device_map= "auto"
)

# don't use the cache
model.config.use_cache = False

# Load the tokenizer from the model (llama2)
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True, use_fast=False)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"

# Load the dataset
dataset = load_dataset("json", data_files="final2i.jsonl")

# Load LoRA configuration
peft_config = LoraConfig(
    lora_alpha=64,
```

```
    lora_dropout=0.1,
    r=16,
    bias="none",
    task_type="CAUSAL_LM",
)

# Set training parameters
training_arguments = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=num_train_epochs,      # uses the number of epochs earlier
    per_device_train_batch_size=4,          # 4 seems reasonable
    gradient_accumulation_steps=2,          # 2 is fine, as we're a small batch
    optim="paged_adamw_32bit",              # default optimizer
    save_steps=0,                           # we're not gonna save
    logging_steps=10,                       # same value as used by Meta
    learning_rate=2e-4,                     # standard learning rate
    weight_decay=0.001,                     # standard weight decay 0.001
    fp16=False,                             # set to true for A100
    bf16=False,                             # set to true for A100
    max_grad_norm=0.3,                      # standard setting
    max_steps=-1,                           # needs to be -1, otherwise overrides epochs
    warmup_ratio=0.03,                      # standard warmup ratio
    group_by_length=True,                   # speeds up the training
    lr_scheduler_type="cosine",           # constant seems better than cosine # TODO: why not
constant?
    report_to="tensorboard"
)

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=peft_config,                # use our lora peft config
    dataset_text_field="text",
    max_seq_length=None,                    # no max sequence length
    tokenizer=tokenizer,                    # use the llama tokenizer
    args=training_arguments,                # use the training arguments
    packing=False,                          # don't need packing
)

# Train model
trainer.train()

# Save trained model
trainer.model.save_pretrained(new_model)

# Empty VRAM
del model
del trainer
del tokenizer

import gc
```

```python
gc.collect()

# Reload model in Float16 and merge it with LoRA weights
base_model = AutoModelForCausalLM.from_pretrained(
    model_name,
    low_cpu_mem_usage=True,
    return_dict=True,
    torch_dtype=torch.float16,
    device_map="auto"
)
model = PeftModel.from_pretrained(base_model, new_model)
model = model.merge_and_unload()

# Reload tokenizer to save it
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"


model.save_pretrained("7b1")
tokenizer.save_pretrained("7b1")
```

F. Inferencing The Model:

- For inferencing the finetuned model we used transformers.pipleine() function which automatically tokenizes the input, generates the output, decodes the output.
- We can set various parameters in the pipeline function for generating response from the AI model:
    - **Temperature**: temperature is a hyperparameter that regulates the randomness, or creativity, of the AI's responses. **We have set it to 0.5 for this project**. The higher it is the more creative and random the answer is.
    - **do_sample**: if set to **True**, this parameter enables decoding strategies such as multinomial sampling, beam-search multinomial sampling, Top-K sampling and Top-p sampling.
    - **top_k**: Introduces random sampling for generated tokens by randomly selecting the next token from the k most likely options. **We have used top_k value of 10 for this project.**
    - **top_p**: It is also known as nucleus sampling, is another hyperparameter that controls the randomness of language model output. **We have set it to 0.5 for this project**.

- ❖ **Repetition Penalty**: It is a technique that penalizes or reduces the probability of generating tokens that have recently appeared in the generated text. It encourages the model to generate more diverse and non-repetitive output. **We have set it to 1.1 for this project.**
- ❖ **Max_length**: If set to a number, will limit the total sequence returned so that it has a maximum length. **We have set it to 2048**, since a higher value was taking more time for inference and was also generating repetitive answers.

- A TCP/IP client server connection was used to perform inferencing in the DGX server and display the output in the client side. The client and the server were on the same LAN. The input prompt was sent from the client to the server and the output was then returned from the server to the client and then displayed on the client side.
- A Simple UI also was created using streamlit library of python using the same client server application.
- Server Side Code(server.py):

```
import socket
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

model_name = "/home/dockuser1/users/Interns/kartik-final/7b1"

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto"
)

tokenizer = AutoTokenizer.from_pretrained(model_name)
model.use_cache = True
model.eval()

def format_instruction(sample):
    return f"""### Instruction:
You are a coding assistant that will write a Solution to resolve the following Task:

### Task:
{sample}

### Solution:
""".strip()

def chatbot_response(query):
```

```python
        final = format_instruction(query)
        pipe = pipeline(
            task="text-generation",
            model=model,
            tokenizer=tokenizer,
            torch_dtype=torch.float16,
            repetition_penalty=1.1
        )
        result = pipe(
            f"{final}",
            do_sample=True,
            top_p=0.5,
            temperature=0.5,
            top_k=10,
            num_return_sequences=1,
            eos_token_id=tokenizer.eos_token_id,
            max_length=2048
        )
        return result[0]['generated_text']

    def main():
        host = '0.0.0.0'  # Listen on all available network interfaces
        port = 12345  # Choose a port for your server

        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind((host, port))
        server.listen(1)  # Listen for a single client connection

        print(f"Server is listening on {host}:{port}")

        while True:
            client_socket, client_address = server.accept()
            print(f"Accepted connection from {client_address}")

            while True:
                data = client_socket.recv(1024).decode('utf-8')
                if not data:
                    break  # Connection closed by the client

                response = chatbot_response(data)

                client_socket.send(response.encode('utf-8'))

            client_socket.close()

    if __name__ == '__main__':
        main()
```

- Client side code(client.py):

```python
# chatbot_ui.py
import streamlit as st
import socket

def main():
    st.title("Chatbot UI")

    # Set the host and port to match the server's configuration
    host = '192.168.5.37'  # Use loopback IP for local testing
    port = 12345  # Use the same port as your server

    # Create a socket client to communicate with the server
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((host, port))

    st.subheader("Enter your inference prompt:")
    message = st.text_input("Prompt")
    if st.button("Send"):
        # Send the user's query to the server
        client.send(message.encode('utf-8'))

        # Receive and display the server's response
        response = client.recv(1024).decode('utf-8')
        st.subheader("Server's Response:")
        st.write(response)

    # Close the client socket when done
    client.close()

if __name__ == '__main__':
    main()
```

G. Output From the Model after Finetuning:

- **Prompt(User Input):** Implement a Embedded code snippet that defines the applet handle message function.

  Output From the model:

```
### Solution:
//Defintion of applet handle message function
MMI_RESULT_E  SampleAppletHandleEvent(

                                      IAPPLET_T*          iapplet_ptr,
                                      MMI_MESSAGE_ID_E    msg_id,
                                      DPARAM              param
                                      );
{
    MMI_RESULT_E      result      = MMI_RESULT_FALSE;
    SAMPLE_APPLET_T* applet_ptr = (SAMPLE_APPLET_T*)iapplet_ptr;
    LOCAL uint32 test_num = 0;
    switch (msg_id)
    {
    case MSG_APPLET_START:
        {
            uint32 start_param = 0;
            MMI_WINDOW_TABLE_CREATE_T create = {0};
            start_param = *(uint32*)param;
            create.applet_handle = applet_ptr->caf_applet.app_handle;
            create.win_table_ptr = MMI_HELLO_SPRD_WIN_TAB;

            applet_ptr->test_num = test_num++;

            MMK_CreateWinTable( &create );
        }
        break;

    case MSG_APPLET_STOP:
        break;

    default:
        break;
    }

    return result;
}

### End of solution
```

- **Prompt(User Input):** develop a code to define Window Table for the window screen to display string in Embedded .
  Output From the model:

```
### Solution:
//The macro WINDOW_TABLE defines the WINDOW data information TABLE
WINDOW_TABLE( MMI_HELLO_SPRD_WIN_TAB ) =
{
    WIN_ID(MMI_HELLO_SPRD_WIN_ID),
    WIN_FUNC((uint32)HandleHelloSprdWinMsg),
    WIN_TITLE(TXT_HELLO_TITLE),
    WIN_SOFTKEY(TXT_COMMON_OK, TXT_NULL, STXT_RETURN),
    END_WIN
};

### End of solution
```

- **Prompt(User Input):** Write Embedded code to create a window function that includes a list widget.
  Output From the model:

```
### Solution:
//window creation for list widget
PUBLIC MMI_HANDLE_T MMIHELLO_CreateListWin(void)
{
    return MMK_CreateWin((uint32*)MMI_TEST_LIST_WIN_TAB, PNULL);
}

//End of file
```

- **Prompt(User Input):** Write code in Embedded to define the handle message function for the button widget.
  Output From the model:

```
### Solution:
//definition of button widget handle message function
LOCAL MMI_RESULT_E HandleTestButtonWinMsg(
                                           MMI_WIN_ID_T     win_id,
                                           MMI_MESSAGE_ID_E msg_id,
                                           DPARAM           param
                                           )
{
    MMI_RESULT_E        recode = MMI_RESULT_TRUE;
    GUI_LCD_DEV_INFO    lcd_dev_info = {GUI_MAIN_LCD_ID, GUI_BLOCK_MAIN};
    GUI_RECT_T          button_rect1 = {0};
    GUI_RECT_T          button_rect2 = {0};
    GUI_RECT_T          rect = {0, 0, 239, 319};

    button_rect1.left = 0;
```

```
        button_rect1.top = MMI_TITLE_HEIGHT + 60;
        button_rect1.right = 120;
        button_rect1.bottom = MMI_TITLE_HEIGHT + 60 + 23;

        button_rect2.left = 0;
        button_rect2.top = MMI_TITLE_HEIGHT + 60 + 23 + 5;
        button_rect2.right = 120;
        button_rect2.bottom = MMI_TITLE_HEIGHT + 60 + 23 + 5 + 23;

        switch(msg_id)
        {
        case MSG_OPEN_WINDOW:
            GUIBUTTON_SetRect(MMI_TEST_BUTTON1_CTRL_ID,&button_rect1);
            GUIBUTTON_SetRect(MMI_TEST_BUTTON2_CTRL_ID,&button_rect2);
            {
                GUI_RECT_T lab_rect = {0};

                lab_rect.left = 0;
                lab_rect.top = MMI_TITLE_HEIGHT;
                lab_rect.right = 239;
                lab_rect.bottom = MMI_TITLE_HEIGHT + 25;

                GUILABEL_SetRect(MMI_TEST_BUTTON_LABEL1_CTRL_ID, &lab_rect, FALSE);
            }
            {
                GUI_RECT_T lab_rect = {0};

                lab_rect.left = 0;
                lab_rect.top = MMI_TITLE_HEIGHT + 30;
                lab_rect.right = 239;
                lab_rect.bottom = MMI_TITLE_HEIGHT + 55;

                GUILABEL_SetRect(MMI_TEST_BUTTON_LABEL2_CTRL_ID, &lab_rect, FALSE);
            }

            GUIBUTTON_SetCallBackFunc(MMI_TEST_BUTTON1_CTRL_ID, LeftButtonCallBack);
            GUIBUTTON_SetCallBackFunc(MMI_TEST_BUTTON2_CTRL_ID, RightButtonCallBack);
            {
                wchar             w_text[] = L"Button1leftrightober";
                uint16 len = MMIAPICOM_Wstrlen(w_text);

                GUIBUTTON_SetText(MMI_TEST_BUTTON1_CTRL_ID,w_text,len);
            }
            break;

        case MSG_GET_FOCUS:
            break;

        case MSG_LOSE_FOCUS:
            break;

        case MSG_FULL_PAINT:
```

```
        GUIRES_DisplayImg(PNULL, &rect, &rect, win_id, IMAGE_COMMON_BG,
&lcd_dev_info);
        GUIRES_DrawLabel(
            PNULL,
            (const GUI_POINTER_T*)MMI_TEST_BUTTON_LABEL1_CTRL_ID,
            (const GUI_RECT_T*)&rect,
            PNULL
            );
        GUIRES_DrawRect(PNULL, &rect, PNULL);
        GUIRES_DrawRect(PNULL, &rect, PNULL);
        break;

    case MSG_CTL_OK:
    case MSG_CTL_CANCEL:
        MMK_ShowWin(win_id, FALSE);
        break;
    case MSG_CLOSE_WINDOW:
        break;
    default:
        recode = MMI_RESULT_FALSE;
        break;
    }

    return recode;
}

### End of solution
```

- **Prompt(User Input):** rite Embedded code to define a function that displays information on the LCD screen.
  Output From the model:

```
### Solution:
//display information function
LOCAL void DrawTestTips(GUI_LCD_DEV_INFO *lcd_info_ptr, GUI_POINT_T position)
{
    GUI_FONT_T font = SONG_FONT_16;
    GUI_COLOR_T color = MMI_BLACK_COLOR;
    GUI_POINT_T pos = position;
    uint16 delta_y = GUI_GetFontHeight(font, UNICODE_HANZI) + 2;
    LOCAL wchar str_down[] = L"Down - List";
    LOCAL wchar str_1[] = L"1 - Textbox";
    LOCAL wchar str_2[] = L"2 - Button";
    wchar *str[] = {str_down,str_1, str_2};
    int32 i = 0;

    GUISTR_STYLE_T      text_style      = {0};
    GUISTR_STATE_T      state =         GUISTR_STATE_WORDBREAK;
    MMI_STRING_T text_str = {0};
```

```
    if(PNULL == lcd_info_ptr)
    {
        return;
    }

    for(i=0; i<sizeof(str)/sizeof(str[0]); i++)
    {
        text_str.wstr_len = MMIAPICOM_Wstrlen(str[i]);
        text_str.wstr_ptr = str[i];
        text_style.font = font;
        text_style.font_color = color;

        GUISTR_DrawTextToLCDSingleLine(
            (const GUI_LCD_DEV_INFO *)lcd_info_ptr,
            (const GUI_POINT_T      *)&pos,
            (const MMI_STRING_T     *)&text_str,
            (GUISTR_STYLE_T         *)&text_style,
            state
            );

        pos.y += delta_y;
    }

    return;
}

### End of solution
```

- As we can see from the above output samples, after training the base model on our code dataset, the model was able to generate the Embedded platform code for the prompts asked.

3.CHALLENGING AREAS

- The inference time for generation of 4096 token output was very long. This may be due to not utilizing all the resources to the maximum limit, there can me multiple reasons for this. More research for improving the accuracy and speed for inferencing is required.
- Since the model had only context length of 4096 tokens, there cannot be rows in the dataset that are exceeding this token limit, as the model would truncate that row in between itself.
- Considering the above reason, we had to break the code into small parts, and then entered in the dataset. During the inferencing of the finetuned model the model was not able to stitch all the pieces of code to generate the complete code, in shot establishing relationship between different rows in the dataset was a challenging task. This could be due to **overfitting**.
- Overfitting is a condition where the model becomes overly specialized on the training data and performs poorly on unseen data. This risk is particularly pronounced when the task-specific dataset is small or not representative of the broader context
- When ever the answer generated by the model was over the max_length set during the inference the model output was randomly truncated after the max_length no of tokens are generated.
- Explaining the complete structure of the Embedded code in the dataset was a challenging task as it was difficult to format this information to train the model.
- Since we were working on a generative AI model, that can generate piece of code and not do a database search using NLP, dataset creation was a challenging task, as the dataset needed to be very precise so that the model can generate anything asked related to Embedded platform code, explaining the structure, syntax, semantics of the Embedded platform code in the dataset was very challenging. Inadequate or noisy data can negatively impact the model's performance and reliability.
- The model sometimes was generating random answers upto the max_length token after the required answer was generated. Stopping the generation after the required answer is generated was a challenge.

- Selecting appropriate hyperparameters for fine-tuning can be intricate and time-consuming. Poor choices may result in slow convergence, overfitting, or suboptimal performance.
- During fine-tuning for a specific task, the model may forget previously acquired general knowledge.

4. STEPS AHEAD

- A different pre-trained base model could be used like the latest mistral-7b etc. Also, a model with less parameters could be experimented.
- For better generation of code from the AI Model, the Dataset should be updated and created in a more precise manner where there is basic explanation or description of each and every thing in the Embedded platform code.
- The Dataset length needs to be increased, and the rows need to be shorter than the current one.
- Methods for establishing relationships between rows of the dataset need to be researched on.
- Multiple training parameters need to be changed and experimented upon, which is a time-consuming task.
- The inferencing parameters on the trained model also needs to be experimented upon.
- Different training methods also needs to be explored; in the current project we have used PEFT-LoRA.
- Frameworks like DeepSpeed can be used to accelerate fine-tuning process.
- Data Parallel can be used to optimize inferencing speed.

5. CONCLUSION

This PoC has demonstrated the feasibility of using generative AI to create Embedded platform code. By leveraging the power of large language models and fine-tuning techniques, we have successfully generated code from natural language descriptions. The results suggest that generative AI has the potential to generate code for Embedded platform and significantly reduce the development time for software applications written in Embedded platform code.

Our PoC involved the following key steps:

1. Dataset Creation: We created a dataset with extension .jsonl , each containing a natural language description of a programming task in the custom language and its corresponding code.
2. Model Fine-tuning: We fine-tuned a base LLM using PEFT and LoRA techniques
3. Model Evaluation: We assessed the performance of the fine-tuned model by taking user prompt, demonstrating its ability to generate a piece of Embedded platform code.

The findings of our PoC are promising and suggest that generative AI has the potential to revolutionize the development of Embedded platform applications. However, further research is necessary to address the following challenges:

1. Code Quality: Ensuring the quality of generated code, particularly in terms of functionality.
2. Domain Adaptation: Adapting generative AI models to different programming languages and their unique syntax and semantics.

As we move forward, further optimizations and refinements can be explored to enhance the efficiency and robustness of the generative AI model on generating Embedded platform code. This PoC serves as a foundation for future developments.