AI-Controlled Robotic Surgical Assistance

2A-Internship Report

Author: Hussein RAMMAL from ENSTA Bretagne
Supervisors: Nabil ZEMITI, Lucas LAVENIR &
Philippe POIGNET

August 29, 2025

Contents

Abstract

In this report, I describe my experience at LIRMM during my internship.

The subject of my internship was controlling the Franka Emika Panda robot with an AI framework dedicated for surgical assistance, making the Franka, which is already designed for research in the surgical robotic assistance field, an AI-controlled surgical assistance robot. With the development of the system having to be based around a state-of-the-art framework that serves this purpose, and having to be achieved in simulation first then moved into real life, I chose to design my system around a framework that is capable of controlling a real Franka through visuo-motor skills learnt entirely within a simulative environment by imitation.

On top of this framework is a french-made LLM that has been recently introduced, is open-source, fine-tuned for instruction following and is the most efficient relative to its size.

Supporting the capturing of the most realistic demonstration for the learning of the visuo-motor skills, I have created a teleoperation mechanism for the real Franka that could be paired simultaneously with the teleoperation of the simulated Franka, independent from the position of the recorder of the demos relative to the real Franka, as the teleoperation of the real Franka is made to be global.

Introduction

After the Covid-19 pandemic, AI tools spread all over our devices from smart phones, cameras, TVs, laptops, computers up to military weaponry, transportation vehicles and surgical tools, becoming a big part of individuals daily lifes post the pandemic.

With that being said, surgical robotics researchers are collaborating with AI researchers to push the AI-controlled surgical assistance robotics up to a point where all medical practitioners would have robots that use LLMs so that the practitioner could communicate to the robot an order for the completion of a task from the set of tasks it has been trained to complete with high safety, accuracy and reliability, and the robot would go on to complete such a task under the his supervision along side his ability to completely take control and teleoperate the robot to his own will. The aim of such robots is to facilitate the progress in the medical field, and revolutionize the way surgeries get conducted for the better.

Having been an AI researcher at LIRMM during my internship, I was tasked to review the publications of the published state-of-the-art frameworks within this field, to provide guidance for my supervisors, which are experts in the surgical assistance robotics field, with a desire to launch projects that feature AI.

Also, I was tasked to design and implement such a project as far as my time at LIRMM would allow, moving from simulation to real implementation. In doing so, I had to work under critical constraints like me not having any background in AI going into this internship, not having AI experts at LIRMM that could follow along my research and decision-making, not having that could provide data for AI training, and not having a budget dedicated for my project.

As you further read this report, you would realize how I worked my way around these constraints in a way that forced me to think differently and use very important tools and technologies, making this internship a very valuable and fruitful internship.

Chapter 1

A LLM

A.1 Introducing LLMs

A large language model (LLM) is a language model trained with self-supervised machine learning on a vast amount of text, designed for natural language processing tasks, especially language generation.

The largest and most capable LLMs are generative pretrained transformers (GPTs), which are largely used in generative chatbots such as ChatGPT, Gemini or Claude. LLMs can be fine-tuned for specific tasks or guided by **prompt engineering**[1].These models acquire predictive power regarding syntax, semantics, and **ontologies**[2] inherent in human language corpora, but they also inherit inaccuracies and biases present in the data they are trained in.

A.2 Revealing the LLM Used

I used the v0.3 mistral:7b (the current latest), a 7-billion-parameter LLM developed by the French startup Mistral AI. It is fine-tuned for instruction following, which enhances its faithfulness to prompts. Having 7.3 billion parameters didn't prevent it from outperforming way bigger LLMs from way more developed companies like Llama 2 13B and Llama 1 34B created by Meta. More about how it stacks up against competitors could be found here.

A.3 Introducing Ollama

Ollama is a locally deployed AI model runner, designed to allow users to download and execute large language models (LLMs) directly on their personal computer, such as a MacBook or Windows machine. Unlike cloud-hosted LLM services, Ollama runs as a background application and provides

[1]Prompt engineering: is the process of structuring or crafting an instruction in order to produce better outputs from a generative artificial intelligence (AI) model.

[2]Ontology: is a way of showing the properties of a subject area and how they are related, by defining a set of terms and relational expressions that represent the entities in that subject area.

a straightforward command-line interface (CLI) and an Application Programming Interface (API) for interacting with various model families, including Mistral, Meta, and Google's Gemma. This local operation ensures that the processing of AI tasks occurs on the user's device.

A.4 Installing Mistral:7B Using Ollama

To install mistral:7b using Ollama, first install Ollama on your machine. Then run this terminal command which for the first time would install mistral:7b v0.3 (as for now) then run a **stateless**[3] interactive loop between you and it in your terminal. Later, this terminal would only run this interactive loop as the LLM would've already been locally installed.

```
ollama run mistral:7b
```

[3]LLMs are, by design, stateless. They don't inherently remember past interactions. If you ask a question, the model responds based on the prompt you give it, but it doesn't retain any memory of the conversation. One way to solve this is by including the entire conversation history in each new prompt. But this approach has limitations.

Chapter 2

B IL and RL

B.1 Introducing IL and RL

Reinforcement learning (RL) is one of the most interesting areas of machine learning, where an agent interacts with an environment by following a policy. In each state of the environment, it takes action based on the policy, and as a result, receives a reward and transitions to a new state. The goal of RL is to learn an optimal policy which maximizes the long-term cumulative rewards.

To achieve this, there are several RL algorithms and methods, which use the received rewards as the main approach to approximate the best policy. Generally, these methods perform really well. In some cases, though the teaching process is challenging. This can be especially true in an environment where the rewards are sparse (e.g. a game where we only receive a reward when the game is won or lost). To help with this issue, we can manually design rewards functions, which provide the agent with more frequent rewards. Also, in certain scenarios, there isn't any direct reward function (e.g. teaching a self-driving vehicle), thus, the manual approach is necessary. However, manually designing a reward function that satisfies the desired behaviour can be extremely complicated.

A feasible solution to this problem is imitation learning (IL). In IL instead of trying to learn from the sparse rewards or manually specifying a reward function, an expert (typically a human) provides us with a set of demonstrations. The agent then tries to learn the optimal policy by following, imitating the expert's decisions.

B.2 Revealing the Used Framework

I built my entire system around SuFIA-BC. An IL and RL framework created by NVIDIA, that creates visuo-motor policies learnt from the expert demonstrations provided and the reward functions specified within the high fidelity simulator called Isaac Sim, extended by the Isaac Lab platform (that houses the ORBIT-Surgical platform mentioned within the documentation).

Those policies, though learned within simulative environments, could also be used to control real robots if the actions of those policies were written

in terms of the low-level control functions of the real robot (in our case, in terms of the functions of libfranka, the library including all the low-level control functions of the Franka Emika Panda), knowing that the generated policies are already compatible with the internal controllers of the simulator.

B.3 Installing SuFIA-BC

You could find on NVIDIA's website the installation procedures of multiple versions of SuFIA-BC, yet I advise installing the version immediately preceding the latest one, the one that uses Isaac Sim 4.5 and Isaac Lab 2.2.0, as the latest one is still under development and has been released to the community to get its feedback (at the time of writing). While older versions have been deprecated.

Chapter 3

C The Project in Simulation

C.1 Design

I created a driver code that starts a communication loop between mistral:7b and the user (considered to be a medical practitioner). The result of this communication would be one of two.

The first is the agreeing of the user on a task fulfillment plan written in terms of the policies that where taught to the AI agents found in SuFIA-BC using its imitation and reinforcement learning pipelines. That would lead to the execution of this plan by SuFIA-BC sequentially, which results in controlling the simulated Franka.

The second is the termination of this communication on the user's behalf, which would bypass the rest of the functionalities in the driver.

C.2 Driver Code

The driver code houses the means of prompting mistral:7b, which has been locally installed, using what I call a system prompt, a prompt that specifies to the LLM how it should carry out its conversation with the user and defines the format it should follow when generating the plan.

The system prompt serves as an instruction manual. It gets sent to mistral:7b along with the input text of the user every time the user inputs text to the LLM, and that's to guarantee the **faithfulness**[4] of LLM to the prompt and to minimize the chance of it **hallucinating**[5].

[4]Faithfulness means, the LLM model should not change the meaning of the information which was given. A faithful answer stays true to the fact and does not add, remove or change the important details.

[5]An AI hallucination is essentially the model making stuff up. It outputs text that sounds plausible and confident but isn't based on truth or reliable data. It's the AI equivalent of "confidently wrong."

```python
def run_ollama_query(instruction):
    prompt = SYSTEM_PROMPT + "\n\nInstruction: " +
       instruction + "\nOutput only the plan steps."
    cmd = ["ollama", "run", "mistral:7b", prompt]
    result = subprocess.run(cmd, capture_output=True,
       text=True)
    return result.stdout.strip()
```

And since the output of the LLM is **stochastic**[6] yet follows the prompt, the above function returns the output of the LLM, stripped of whitespace, and the following function then takes this output as input, bypasses the LLM's chain-of-thought (the stage where the LLM explains the task to itself and reasons before producing the plan), and extracts the plan, whose format is specified in the prompt, as a list of policies to be executed in order from first to last.

```python
def extract_plan(text):
    lines = text.splitlines()
    policy_list = []
    pattern = r'^\d+\.(\w[\w-]*)'

    for line in lines:
        match = re.match(pattern, line.strip())
        if match:
            policy_list.append(match.group(1))

    return policy_list
```

The function **Y** is designed to take the name of a trained policy, an element from the policy list generated by **extract_plan**, and execute it in a new terminal. The shell instructions for the selected policy are contained in its corresponding command string, which navigates to the user's home directory, sets up and activates the Conda environment, changes to the IsaacLab project folder, runs the terminal command responsible for launching the environment where the SuFIA-BC AI agents were trained to control the Franka to perform the passed policy, and instructs them to carry out that policy exactly as learned during training.

[6]Stochastic: non-deterministic

```python
def Y(action):
    if action == "lift":
        command = (
            "cd /home/hrammal && source /home/hrammal/miniconda3/etc/profile.d/conda.sh "
            "&& conda activate env_isaaclab && cd /home/hrammal/Desktop/stage_HRAMMAL_2025/3-developments/IsaacLab "
            "&& ./isaaclab.sh -p scripts/reinforcement_learning/rsl_rl/play.py "
            "--task Isaac-Lift-Cube-Franka-v0 --video --video_length 100 --num_envs 1"
        )
        subprocess.run(command, shell=True, executable="/bin/bash")
    # The implementations for 'extend' and 'open-drawer' follow the same pattern as above
```

Finally, the function **interactive_session** integrates all components of the system. It continuously collects user instructions, submits them to the LLM via **run_ollama_query**, extracts a plan using **extract_plan**, and presents the proposed plan for verification. Once approved with F2, each action in the plan is executed sequentially through **Y(action)**. If clarification or a new instruction is required, pressing F3 restarts the session. This design ensures that only fully validated and approved policies are executed, preserving operational safety and precision in the surgical context.

C.3 Training the AI Agents

The AI agents were trained to control a Franka Emika Panda, enabling it to lift a specific cube from a designated position, open a particular drawer, and extend its arm in a precise manner. This training was performed using reinforcement learning, as NVIDIA had already provided suitable reward functions for these tasks. I opted for reinforcement learning instead of imitation learning to avoid the time-consuming process of recording demonstrations for each behavior.

Here, you can watch me training the AI agents for the drawer-opening policy. Notably, at **24 minutes and 22 seconds**, the video shows me executing this policy using the same command as in the **Y** function.

C.4 Results

A week and into July, I had finished developing the simulative part of the system and started focusing on its real-world implementation. I had promised myself that the last couple of days of my internship would be spent organizing files and capturing footage of the system in action.

When those days came, I realized that the warm-up process for both Isaac Sim and Isaac Lab had changed. Launching Isaac Sim extended with Isaac Lab, which is the simulator within SuFIA-BC, suddenly took much longer and became more resource-intensive. In this footage, you can see me starting it right after a system restart, and I only began recording once the application was ready and the simulator had finished initializing.

Since the system also had to run a 7B-parameter LLM locally at the same time, the computer started running out of memory. Because of this, I couldn't reproduce the same robustness it had before the update, when I could send multiline commands to the LLM and generate correct plans of up to six policies, all of which SuFIA-BC would execute smoothly in simulation.

Here, you can see me working with the system in its final state, once when it ran out of memory and once when it did not.

Chapter 4

D The Project in Real Time

D.1 Design

The design of the real-time system is the same as the simulative version.

What is intended here is to establish the branch connecting the simulation to the real-time implementation, which requires two main components:

1. Creating the means to teleoperate the real Franka so that, when capturing demonstrations, which is typically done entirely within Isaac Sim extended by Isaac Lab, we can do so while simultaneously observing what we are teaching the AI agents to perform on the real robot. Moreover, this approach allows us to simulate the real robot in the desired way and, at the same time, record demonstrations to create policies that are as realistic as possible.

2. Creating the mapping of those learned policies onto the low-level functions of the Franka (provided by the **libfranka** library), so that executing them directly controls the real Franka.

D.2 Development Progress

After finishing the simulation part of the system, I had about a week and a half left to work on the real part. I spoke with one of my supervisors about building the real teleoperation in that time, since the second component would take at least a month. I described to him the design for a teleoperation system that would let the demonstrator work without being physically near the robot while still recording demonstrations from anywhere in the world, and he liked the idea and encouraged me to go ahead with it.

D.3 The Teleoperation System

D.3.1 Introducing MQTT

MQTT (Message Queuing Telemetry Transport) is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code

footprint and minimal network bandwidth. MQTT today is used in a wide variety of industries, such as automotive, manufacturing, telecommunications, oil and gas, etc.

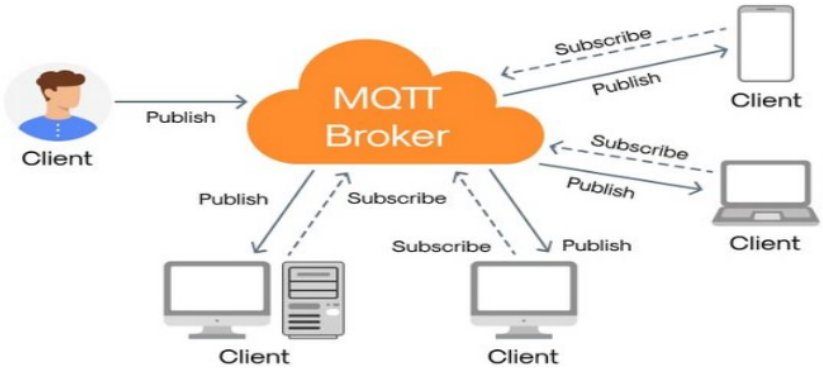

Figure 1: MQTT Client–Broker Model

The two main components of the MQTT protocol are the client and the broker. An MQTT client can be any device that runs an MQTT library and connects to an MQTT broker over a network. The publisher and subscriber labels refer to whether the client is publishing or subscribed to receive messages. The MQTT broker, on the other hand, is responsible for receiving all messages, filtering them, and sending them to subscribed clients. The broker also handles client authentication and authorization and holds all clients' session data with persistent sessions.

D.3.2 Design

Prior to explaining the design of the teleoperation system, it is important to understand how the real Franka can be controlled via an external PC.

To achieve this, the external PC must be connected to the Franka's controller using an Ethernet cable. Once the LAN connection is established, the user can open **Google Chrome** and enter the IP address of the Franka, which is usually indicated on a label on the controller. This will open the Franka interface, from which the user can unlock or lock the robot's joints, or even shut it down.

After unlocking the joints, a script can be run locally on the PC to connect to the robot and control it using the low-level functions provided by the **libfranka** library.

The teleoperation system is designed to control the real Franka using an **Xbox Series X** joystick. In this setup, the joystick is not connected to the PC linked to the robot via an Ethernet cable. Instead, it can be connected to any other PC in the world. This second PC captures the joystick commands as a ROS2 topic, converts them to JSON format, and sends them wirelessly as an MQTT topic to the PC connected to the robot. The robot-connected PC then receives these commands, converts them back to the original ROS2 topic, and maps them to the low-level control functions provided by **libfranka**, allowing the real Franka to be controlled remotely and in real time.

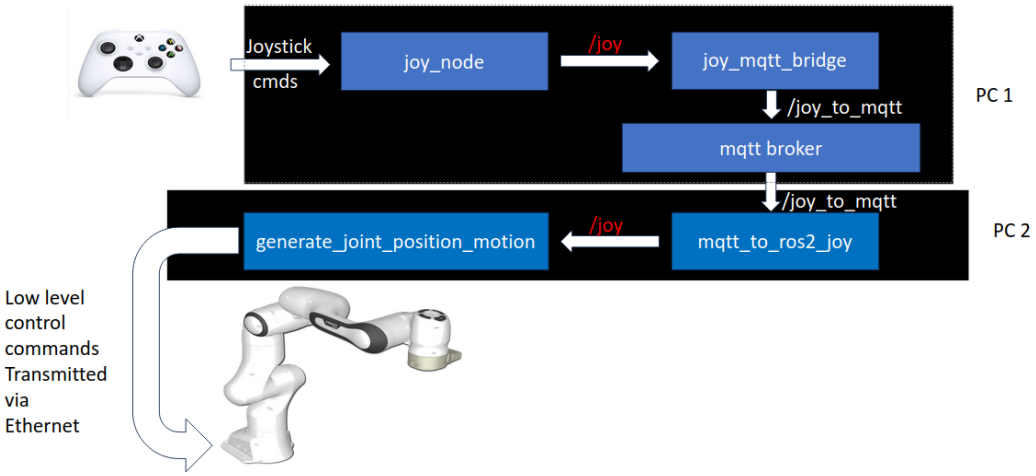

Figure 2: Teleoperation System Design

Both PCs must have ROS2 **humble** installed. PC 1 is the demonstrator's PC, while PC 2 is the one connected to the real robot.

The demonstrator runs the following command to start recording demonstrations:

```
./isaaclab.sh -p scripts/tools/record_demos.py --task x
   --teleop_device joystick --dataset_file ./datasets/
   dataset.hdf5 --num_demos y
```

Replace **y** with the number of demonstrations to record, and **x** with the name of the environment in which you want to demonstrate the task. To see the available environments, run:

```
./isaaclab.sh -p scripts/environments/list_envs.py
```

Now he's recording demos on SuFIA-BC for it to later use to teach its AI agents, he runs in a new terminal:

```
ros2 run joy joy_node
```

which enables PC 1 to capture joystick commands and publish them on the ROS 2 topic **/joy**.

In another terminal, the demonstrator runs the ROS 2 node named **joy_mqtt_bridge**, which subscribes to the **/joy** topic and transforms the messages to **JSON** format so it could be published to MQTT , and publishes it under to MQTT topic **/joy_to_mqtt** using the following function:

```
def joy_callback(self, msg):
    data = {
        "axes": list(msg.axes),
        "buttons": list(msg.buttons)
    }
    json_str = json.dumps(data)

    self.mqtt_client.publish("/joy_to_mqtt", json_str)
    self.get_logger().info("Published to MQTT: " +
       json_str)
```

Now on PC 2, we run the ROS 2 node **mqtt_to_ros2_joy** which subscribes to **/joy_to_mqtt**, transforms its message back into the format of the ROS 2 topic **/joy**, and publishes it as a ROS 2 topic called transforms its message back into the format of the ROS 2 topic **/joy** using:

```
def on_message(self, client, userdata, msg):
    try:
        json_str = msg.payload.decode('utf-8')
        data = json.loads(json_str)

        joy_msg = Joy()
        joy_msg.axes = data["axes"]
        joy_msg.buttons = data["buttons"]

        self.joy_publisher.publish(joy_msg)
        self.get_logger().info(f"Published Joy message
           from MQTT JSON: {json_str}")
```

Thanks to that, the **/joy** within PC 1 is now within PC 2.

In a new terminal on PC 2, we run the ROS2 node **generate_joint_position_motion**, which subscribes to the **/joy** topic and maps the received messages to the low-level control functions of **libfranka** to control the real robot.

Here, the axes and buttons from the Joy message are scaled and stored in the shared array **joy_deltas**, which represents joint increments for all 7 DOFs of the robot:

```
void joy_callback(const sensor_msgs::msg::Joy::SharedPtr
    msg) {
    std::lock_guard<std::mutex> lock(joy_mutex);
    joy_deltas[0] = msg->axes[0] * kScale;
    joy_deltas[1] = msg->axes[1] * kScale;
    joy_deltas[2] = msg->axes[3] * kScale;
    joy_deltas[3] = msg->axes[4] * kScale;
    joy_deltas[4] = msg->axes[6] * kScale;
    joy_deltas[5] = msg->axes[7] * kScale;
    joy_deltas[6] = (msg->buttons[4] - msg->buttons[5])
       * kScale;
}
```

kScale is a numeric value that determines how sensitive the robot is to these increments, and it is the same for all joints.

In the real-time control loop, the joystick deltas are applied to the current joint positions stored in `current_position`, producing updated desired joint positions. Finally, **franka::JointPositions(current_position)** sends these updated positions as low-level commands to the Franka robot, effectively transforming the joystick input into real 7-DOF joint motion:

```
std::array<double, 7> delta_input;
{
    std::lock_guard<std::mutex> lock(joy_mutex);
    delta_input = joy_deltas; // copy the latest
       joystick deltas
}

for (size_t i = 0; i < 7; ++i) {
    current_position[i] += delta_input[i]; // update
       desired joint positions
}

```

```
11 return franka::JointPositions(current_position); // send
       to robot
```

D.3.3 Result

The MQTT communication is secured using **TLS**[7]**encryption**[8], ensuring that all joystick data transmitted from ROS 2 to the MQTT broker is encrypted and protected from interception or tampering. Both clients, **joy_mqtt_bridge** and **mqtt_to_ros2_joy**, explicitly set **tls_insecure_set(False)**, enforcing strict certificate validation to prevent man-in-the-middle attacks. Client certificates (**client.crt** and **client.key**) along with the CA certificate (**ca.crt**) are used to authenticate both the broker and the client, ensuring that only authorized devices can publish or subscribe to the joystick topic. Additionally, by running the MQTT loop in a separate thread using **loop_start()**, the system maintains continuous, non-blocking communication, preventing delays or dropped messages that could otherwise result in unsafe robot commands.

You can see me using the teleoperation system in this video.

[7]Transport Layer Security (TLS) stands as a crucial security protocol widely employed to establish secure communication between clients and servers on the internet, facilitating interactions like those between web pages and browsers. Serving a pivotal role, TLS ensures the encryption of data in transit while verifying the identities of both parties involved, assuring exclusive access to the intended recipient.

[8]Encryption: Safeguarding data from unauthorized access, ensuring privacy.

Chapter 5

E Noteworthy Research

E.1 Reviewing the State of the Art

As I mentioned in the **Introduction**, my supervisors asked me to go through a large set of state-of-the-art frameworks for AI-controlled surgical assistance robots. This work took me about a month and a week and ended with me presenting the lab with a review of these frameworks. The review was useful for guiding future research and new projects. Alongside this, I also presented the design of my own system, which is based on **SuFIA-BC**, one of the frameworks I had studied.

E.2 Building GPT-2 from Scratch

With no background in AI, I jumped straight into the state-of-the-art review, and LLMs immediately stood out. To make the most of my internship, I dove into learning and developing with them while doing the review, making both research and development time more efficient. I got started by following one of the top-rated free courses on LLMs, available here.

I ended up creating **libllm**, a **PyTorch**[9] library for building, training, and fine-tuning GPT-2–style transformer models for **autoregressive**[10] text generation and classification tasks.

With it, I created a 124M-parameter GPT-2 fine-tuned to classify a user query (also assumed to be a medical practitioner) into one of two task categories: **pick-up-needle** or **pick-up-block**.

[9] PyTorch is a software-based open source deep learning framework used to build neural networks, combining the machine learning (ML) library of Torch with a Python-based high-level API.

[10] An autoregressive model is a category of machine learning models in which algorithms predict future data based on a series of their own past data.

E.3 libllm

First, it defines a class that stores the architecture's parameters, including embedding size, number of attention heads, number of transformer blocks, total vocabulary size, and the regularization coefficient. It then defines the token and positional embedding layers. Summing the outputs of these layers for each input token produces its vector embedding:

$$x = tok\_emb + pos\_emb$$

A dropout is then applied to this embedding for regularization:

$$x = Dropout(x)$$

And since the core of the GPT model is a stack of transformer blocks, it defines that with:

```
self.blocks = nn.Sequential(*[Block(config) for _ in
    range(config.n_layer)])
```

Each block consists of a layer normalization, a multi-head self-attention mechanism, a feedforward network, residual connections, and dropout. The composition of a block is illustrated as follows:

```
self.ln1 = nn.LayerNorm(config.n_embd)
self.sa = MultiHeadAttention(config)
self.ln2 = nn.LayerNorm(config.n_embd)
self.ffwd = FeedForward(config.n_embd)
```

Its forward pass is defined as follows:

$$x = x + SelfAttention(LayerNorm_1(x))$$

$$x = x + FeedForward(LayerNorm_2(x))$$

which improves numerical stability.

As for the multi-head self-attention mechanism, each head computes scaled dot-product attention independently. Outputs are concatenated and projected:

$$out = Proj\Big(Concat([h(x)\ \forall h \in heads])\Big)$$

The calculation of the scaled dot-product attention with causal masking is done with:

$$weights = \frac{QK^\top}{\sqrt{d_k}}, \quad weights = weights \odot tril, \quad out = weights \cdot V$$

This ensures tokens only attend to previous or current positions.

The feedforward network is implemented as a two-linear-layer perceptron with GELU activation and dropout:

$$FFN(x) = Dropout\Big(Linear_2(GELU(Linear_1(x)))\Big)$$

This expands the embedding dimension from **n_embd** to $4 \times$ **n_embd** in the hidden layer, and then projects it back to **n_embd**.

Finally, the output projection head maps output embeddings to vocabulary logits with:

$$logits = Linear(x)$$

I will stop at this point to avoid overcrowding the report with raw mathematics and data processing techniques. For those interested in more details, I recommend consulting the comprehensive tutorial that I followed..

E.3.1 Result

Here, you can see me using the GPT-2 model that I created.

This experience made me realize how challenging it is to build an LLM that is sufficiently intelligent for my system on my own. Consequently, I began searching for capable open-source LLMs and eventually found Ollama, installing Mistral:7B instead.

Conclusion

The design of the system, when compared to more advanced frameworks, is more foundational rather than highly sophisticated, as the latter tend to be far more meticulous. To bring our design closer to such frameworks, we could, for instance, integrate a vision-language model (VLM) to verify that the objects intended for manipulation are actually present and accessible to the robot.

Nevertheless, I believe that everything developed during this internship fulfilled its purpose and worked exactly as intended.

Given the time constraints, I was able to implement the core of the simulation part, but the system could be significantly improved with more time. Enhancements could include better management of the interactive loop between the user and the LLM, improved handling of conversation history, refined prompts, or even bypassing the need to run a local LLM by subscribing to an OpenAI API key, which would allow querying an online LLM with up to 1.8 trillion parameters at the time of writing.

On the other hand, the design of the teleoperation interface has reached its full potential using MQTT. Other networking methods might be faster, safer, or offer more configurability, but within the limits of MQTT, this design performs optimally.

For the remaining part of the real-time implementation, which is executing the learned policies on the real Franka robot, a function must be created to perform these policies using the robot's low-level control functions. If this were not feasible for some reason, it would require creating a separate imitation learning pipeline for the real robot, which in turn would necessitate adding extra sensors to the robot.

On a personal level, this internship really pushed me to think like an engineer, which was exactly the point. Honestly, the state-of-the-art review probably took much longer than necessary. Most of the frameworks I looked at were so advanced that studying them did not give me much practical insight, especially since their creators did not share useful resources such as code, datasets, or AI agents. With some guidance, I could have understood the designs quickly, built my system, and moved straight into development. I also did not have any ready-made resources to build on. Something like a link to Ollama or other open-source LLMs would have made things much easier. Perhaps that was the purpose of my stage: to identify what was missing and provide resources that could help future research. I am proud that I completed this internship and made valuable connections along the way.

References

[1] Large Language Model - Wikipedia

[2] Prompt Engineering - Wikipedia

[3] Ontology (Information Science) - Wikipedia

[4] Announcing Mistral 7B - Mistral.ai

[5] What is Ollama? - Medium

[6] Ollama Official Website

[7] Ollama Mistral 7B Library

[8] Vector Databases - Medium

[9] Imitation Learning Overview - Medium

[10] Sufia BC - Orbit Surgical

[11] ArXiv Paper

[12] Isaac Sim Documentation - NVIDIA

[13] Isaac Lab - NVIDIA Developer

[14] Isaac Lab Installation Guide

[15] LLM Hallucinations Explained - Medium

[16] MQTT Official Website

[17] MQTT Client-Broker Connection - HiveMQ Blog

[18] What is TLS Encryption? - SecureW2 Blog

[19] Autoregressive Model - TechTarget