Robot Control via Compact LLMs: A Case Study

Supervised by Conf. Dr. Christian Săcărea

Bogdan Oprisiu
Student, Informatics (German line), Faculty of Mathematics and Computer Science
Babeș-Bolyai University, Cluj-Napoca, Romania
bogdan.oprisiu@ubbcluj.ro

***Abstract*—Compact, task-specific language models are investigated for natural-language robot control. User commands are converted into a standardised JSON representation that is executed on a robotic platform. A custom dataset supports the distillation and 8-bit quantisation of a Flan-T5 model on commodity GPUs and Google Colab. The prototype currently performs inference on an external PC, streaming JSON to an Arduino motor controller; a subsequent iteration will integrate the model on a Raspberry Pi for fully autonomous operation. Tests with a self-built Mecanum-wheel rover confirm that compact LLMs can enable reliable, resource-efficient, language-driven robotics.**

***Index Terms*—Large Language Models, Robotics, Quantization, Knowledge Distillation, JSON command**

I. Introduction

THE rapid progress of artificial intelligence— especially in natural-language processing—has produced Transformer-based large language models (LLMs) capable of understanding and generating human-like text [1]. These models enable intuitive, language-driven human–machine interaction that goes far beyond traditional menu- or joystick-based interfaces. In robotics, natural-language commands promise greater flexibility and accessibility, yet deploying high-performance LLMs on memory- and power-constrained edge devices remains a significant challenge.

This study investigates whether a compact, task-specific LLM can translate natural-language commands into a standardised JSON format that a robot can execute directly. Building on earlier undergraduate work, the project demonstrates an end-to-end pipeline—data generation, model distillation, and inference—that bridges human intent and low-level actuation. By automating the conversion from free text to structured control messages, the method seeks to make robot programming more accessible and easily adaptable to new tasks.

To realise this goal, a dedicated dataset tailored to robot-control scenarios was created and a compact language model was trained. The model was then optimised with Knowledge Distillation and 8-bit Quantisation, thereby reducing memory and latency requirements enough for deployment on modest hardware. In the present experimental set-up, inference is performed on an external PC that streams JSON commands to an Arduino, which drives a fully self-built Mecanum-wheel rover. All mechanical and electrical components—from CAD design to wiring—were produced in-house. The results indicate that, even under stringent resource constraints, compact LLMs can accurately interpret and translate complex natural-language instructions into executable robot actions.

The remainder of the paper is organised as follows. Section II reviews large language models and the compression techniques used in this work. Section III explains the methodology, covering dataset construction and model training. Section IV describes the hardware and software implementation. Section V reports the experimental results, and Section VI discusses the findings and concludes the paper.

AI Usage Disclosure: *For the translation of this work from German into English, assistance from modern AI-powered language models, specifically Large Language Models (LLMs), was utilized. These systems were employed to ensure accurate and precise linguistic formulation. The content, scientific concepts, and research findings presented in this paper remain the original work of the author.*

—

II. Background

A. Machine Learning Fundamentals

Machine Learning (ML) is a central subfield of artificial intelligence that enables systems to independently recognize patterns from data and make decisions based on them, without being explicitly programmed for each task. The objective is to learn a function $f(x)$ that maps input data x as accurately as possible to target values y. This is typically achieved through iterative adjustment of model parameters using optimization algorithms to minimize a defined loss function $L(f(x), y)$.

A typical machine learning process comprises the following steps:

1) **Data Preparation:** Cleaning and structuring of data.
2) **Model Training:** Adjustment of model parameters through optimization methods such as gradient descent.
3) **Model Validation:** Evaluation of model performance using a suitable loss function.
4) **Model Adjustment:** Optimizations, if necessary, to improve model accuracy.
5) **Application:** Utilization of the validated model on new data.

Machine learning is broadly categorized into different paradigms: In *supervised learning*, models are trained with datasets containing both input data and target values, while

unsupervised learning identifies patterns in data without predefined target values. In *reinforcement learning*, an agent learns through interaction with its environment and based on reward signals to make optimal decisions.

The success of a model heavily depends on the correct choice of model architecture and an appropriate loss function. Beyond classical models like decision trees or support vector machines, neural networks are increasingly employed. Their complexity is controlled through stochastic optimization techniques and regularization methods to prevent overfitting and ensure good generalization capabilities.

Domingos distils more than a decade of "folk knowledge" in machine learning into eight pragmatic maxims [2]:

1) Learning is the interplay of representation, evaluation, and optimisation.
2) Generalisation performance—rather than training error—is the real objective.
3) Over-fitting has many faces and must be monitored continuously.
4) Intuition fails in high-dimensional spaces (the curse of dimensionality).
5) Theoretical guarantees often hide unrealistic assumptions.
6) Clever feature engineering remains critical.
7) In practice, more data usually beats a more sophisticated algorithm.
8) Combining multiple models typically outperforms selecting a single "best" one.

1) Quantization: Quantization is a model compression technique that reduces the numerical precision of model weights or activations. Instead of using, for example, 32-bit floating-point numbers (float32), values are represented with a lower bit count—such as 8-bit integers (int8). This leads to a significant reduction in memory footprint and computational requirements, which is particularly relevant for deployment on mobile devices or embedded systems.

To illustrate the quantization process, an example with a complex waveform was simulated. Figure 1 compares an original float32 waveform with its reconstructed version after quantization.

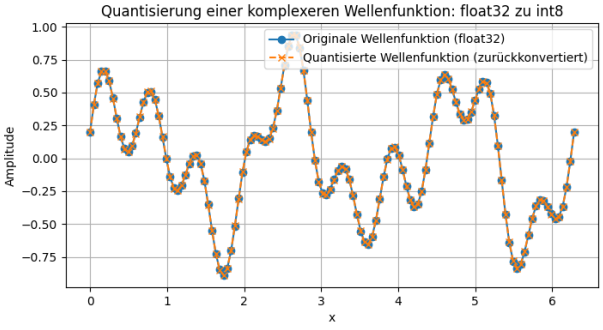

Fig. 1: Simulation of waveform quantization: Comparison between the original float32 waveform and the reconstructed version after quantization. **Source:** Own Simulations

This visualization shows that despite the reduction in numerical precision, the general shape of the waveform is preserved—although fine details are lost. This corresponds to the typical behavior of quantization: a significant advantage in terms of memory and computational savings, but associated with a certain loss of accuracy.

B. Loss Functions

At its core, a loss function is how we tell our machine learning model if it's doing a good job or not. Think of it as a scoring system: it quantifies the difference, or "error," between what our model predicts and what the actual correct answer is. The ultimate goal during training is always to minimize this score, guiding the model to learn better and make more accurate predictions.

In projects like ours, especially when we're trying to transfer knowledge from one model to another (a process called distillation), we often don't just use one type of error measurement. We combine different loss functions. This allows us to account for various types of "targets." For instance, we use one kind of loss for "hard targets," which are the definite correct answers, and another for "soft targets," which are more nuanced probability distributions often provided by a larger, more expert model. This combination helps the smaller model learn not just the right answer, but also the confidence and relationships between different possibilities, much like a teacher guides a student.

Figure 2 illustrates how different loss components, including Cross-Entropy (for hard targets), KL-Divergence (for soft targets), and their combined effect, typically behave during training. As the iterations progress, you can see how these simulated loss values decrease, showing the model is gradually learning and improving its predictions.

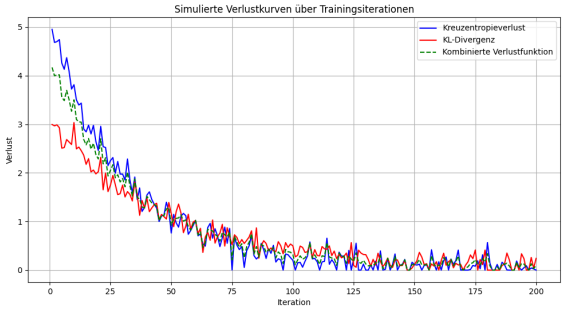

Fig. 2: Simulated Loss Curves over Training Iterations. **Source:** Own Simulations

C. Model Compression Techniques

As machine learning models, especially Large Language Models (LLMs), grow increasingly powerful, they also become very large and computationally expensive. This can be a major hurdle when trying to deploy them on devices with limited resources, like mobile phones or, in our case, robots. That's where model compression techniques come in handy.

These methods aim to reduce the size and computational footprint of a model without sacrificing too much of its performance.

One really effective compression technique, which we find particularly interesting, is Knowledge Distillation [3]. The core idea here is quite intuitive: imagine you have a very experienced teacher (a large, complex model) who knows a lot, and a new student (a smaller, simpler model) who needs to learn. Instead of just teaching the student the final answers (the "hard targets"), the teacher also shares its thought process, like the probabilities it assigns to all possible answers, even the incorrect ones (these are called "soft targets") [3].

By learning from these soft targets, the student model gets a much richer understanding of the data than it would from just the hard labels. This allows the smaller "student" to capture a significant portion of the "teacher's" knowledge and performance, but in a much more compact and efficient form. This approach is super valuable because it helps us create models that can run quickly and effectively on hardware that wouldn't be able to handle their larger counterparts.

D. Natural Language Processing and Large Language Models

Natural Language Processing (NLP) is a pivotal field within artificial intelligence focused on enabling machines to understand, interpret, and generate human language. Historically, NLP relied on rule-based or statistical models, which faced limitations with complex linguistic structures. A significant breakthrough occurred with the widespread adoption of deep neural networks, particularly the introduction of the Transformer architecture [1]. This architecture revolutionized the field by enabling parallel and context-aware processing of long texts, overcoming limitations of previous sequential models, such as the vanishing gradient problem [4].

The ***Transformer architecture***, introduced by Vaswani et al. [1], forms the backbone of modern Large Language Models (LLMs) due to its efficiency and scalability. Unlike earlier sequential models, the Transformer relies entirely on **attention mechanisms** to process sequences. It consists of two main components: an **Encoder** and a **Decoder** [1]. The Encoder's role is to process the input sequence in parallel, generating a rich, contextualized representation of the input. The Decoder then uses this representation, along with its own previous outputs, to generate the target sequence token by token [1]. This design allows the model to capture complex relationships within text, leading to highly effective language understanding and generation.

Large Language Models (LLMs) are highly scaled neural networks built upon this Transformer architecture, learning intricate language patterns from extensive text corpora. Well-known examples such as GPT and BERT exemplify their remarkable flexibility and capability in diverse tasks like text generation, machine translation, and dialogue systems [5]. Their scalability and generalization abilities have positioned LLMs as a foundational technology for many advanced applications in both research and industry. For a more detailed understanding of the Transformer architecture and the capabilities of foundational LLMs, readers are encouraged to consult the cited works.

E. Tokenization

Tokenization is a fundamental preprocessing step in Natural Language Processing (NLP) that prepares raw text for machine learning models. Essentially, it's about breaking down a continuous stream of text into smaller, discrete units called **tokens**. These tokens are the basic building blocks that our NLP models understand and process, whether they're individual characters, whole words, or parts of words. The way we tokenize text significantly impacts how well our language models perform, especially for large models like LLMs, because it defines how language patterns are represented.

While there are methods that break text down into individual characters or entire words, a popular approach that balances the benefits of both is **Subword Tokenization**. Among these, **Byte Pair Encoding (BPE)** is a widely used and effective method. The cool thing about BPE is that it starts by treating each character as its own token. Then, it iteratively looks for the most frequent pairs of tokens and merges them into a new, larger token. This process repeats until a predefined vocabulary size is reached. This way, BPE can represent common words efficiently while still being able to handle rare or unknown words by breaking them down into smaller, known subword units. This helps keep the vocabulary size manageable and the model robust, even with varied inputs.

For example, if we start with words like "lower," "lowest," and "newer," BPE would identify "er" as a common pair, merge it, then "lo," and so on, building a vocabulary of subword units as shown in Table I.

TABLE I: Example steps of Byte Pair Encoding Tokenization

Iteration	Most Frequent Pair	Token Sequence
0	-	l o w e r l o w e s t n e w e r
1	e r	l o w e r l o w e s t n e w **er**
2	l o	**lo** w e r **lo** w e s t n e w **er**
3	lo w	**low** e r **low** e s t n e w **er**
4	e s	**low** e r **low es** t n e w **er**
5	es t	**low** e r **low est** n e w **er**

This process creates a versatile vocabulary where words are represented either fully or as a combination of these subword tokens, ensuring that even words not seen during training can be effectively processed.

F. Low-Rank Adaptation (LoRA)

Low-Rank Adaptation (LoRA) is an efficient method designed for fine-tuning large neural models. It was originally introduced to significantly reduce the computational resources needed when adapting pre-trained Transformer models. The core idea is to partially adjust the model's parameters by specifically utilizing a low-rank matrix decomposition. This technique drastically cuts down the number of parameters that need to be trained during the fine-tuning process, without significantly compromising the model's performance [6].

Mathematically, LoRA can be formulated as follows: given an original weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA augments it with two much smaller, trainable matrices, $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, where the rank r is chosen so that $r \ll \min(d, k)$. The adapted weight matrix then becomes:

$$W' = W + BA$$

This low-dimensional decomposition allows for training only a small number of additional parameters, which considerably reduces both memory consumption and computational load. The underlying principle is that, for many fine-tuning tasks, only minor adjustments are necessary to achieve significant performance improvements on specific datasets. Thus, LoRA avoids a full retraining of the existing weights.

In this work, we employ LoRA to efficiently fine-tune a large, pre-trained language model on our custom dataset. Our implementation was guided by the practical resources and documentation provided by Hugging Face [7]. For a deeper dive, we encourage the reader to consult those original sources for detailed examples and best practices.

—

III. Methodology

A. Dataset

My dataset comprises three parts:

- **Labelled (64 068):** Generated by combining templates (e.g. `"move $DIR $DIST $UNIT"`), parameter samplers (distance, angle, acceleration) and a small synonym lexicon. Each example comes with an exact JSON target.
- **Unlabelled (153 742):** Raw commands scraped from the web and paraphrased via ChatGPT. These are later pseudo-labelled on the fly by the fine-tuned teacher model (see §III-C).
- **Test (1 032):** Hand-curated examples (typos, dialectal forms, multi-command sequences, unit ambiguities) used for early error-mode detection.

All outputs follow the same JSON schema (see §III-B), so new fields can be added without breaking the pipeline.

Labelled data were generated by combining

- simple templates (e.g. `"move $DIR $DIST $UNIT"`),
- parameter samplers for distance, angle, acceleration,
- a small synonym lexicon (e.g. `forward`/`ahead`).

This yields both "basic" examples (only `distance`, `angle`) and "full" examples (with `acceleration`), for a total of 42 000 + 22 068 = 64 068 pairs.

Unlabelled data consist of roughly 153 k commands collected via web crawlers and paraphrasing, which are later pseudo-labelled on the fly by the fine-tuned teacher model (see Distillation in §III-C).

Test data are 1 032 hand-curated examples covering typos, multiple commands, colloquial phrasing, and unit ambiguities. They help catch failure modes early in development.

All JSON outputs follow the same schema (see §III-B), so new fields can be added without breaking the pipeline.

B. JSON Command Language

I designed a simple, extensible JSON schema to bridge natural language and robot actions. Each command object contains:

- `command`: the action name (e.g. `"forward"`, `"rotate"`).
- `parameters`: a dictionary of typed values, for example:
  - `distance` (number, in cm)
  - `angle` (number, in deg or rad)
  - optional `acceleration` (number, in cm/s^2 or deg/s^2)
  - optional `direction` (string: `"left"` / `"right"`)

A minimal example:

```
{
  "command": "rotate",
  "parameters": {
    "angle": 30,
    "direction": "left"
  }
}
```

This structure is intentionally flat and predictable, so *(1)* any new parameter can be added without changing existing code, and *(2)* the same parsing logic works for both "basic" and "full" commands.

C. Teacher

The teacher model (`google/flan-t5-small`, ≈80 M parameters) was supervised fine-tuned on 120 000 synthetic command–JSON pairs. To reduce training and storage costs, we inserted LoRA adapters (rank = 8, α = 16) into the base weights and stored those base weights in 8-bit precision. Fine-tuning ran for 16 epochs with batch size = 8, learning rate = 3×10^{-4}, AdamW (weight decay = 0.01), a constant LR scheduler, up to 512 tokens per example, and the model's standard tokenizer.

a) Dynamic Quantization: After fine-tuning, we apply PyTorch's dynamic quantization to all linear and attention layers, converting float32 weights and activations to int8 at inference time. This reduces the model's memory footprint by roughly 3–4× and speeds up inference by about 1.5–2×, with minimal accuracy loss.

b) LoRA Fine-Tuning Loss: Figure 3 shows the training loss curve for the LoRA adapters during the teacher's supervised fine-tuning. The rapid decline in loss demonstrates that the low-rank adjustments efficiently capture the task-specific knowledge.

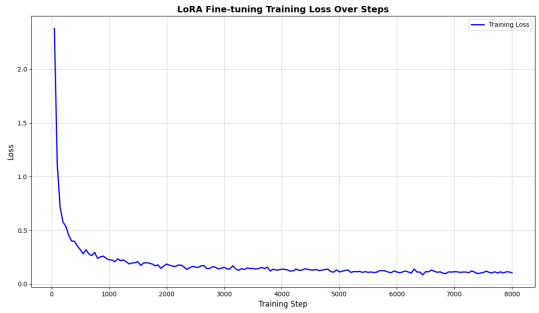

Fig. 3: LoRA fine-tuning training loss over steps.

c) Soft-Target Extraction: In inference mode, the quantized teacher processes each of the 153 742 unlabeled commands. For every decoding step, we collect the raw logits, apply a softmax (temperature $T = 1$), and record the top-5 token probabilities in float16. These *soft targets*—rich, example-specific distributions—are then used to guide the student model's training, effectively transferring nuanced knowledge from the teacher.

D. Student

The student model is built on a compact Mini-T5 variant derived from `flan-t5-small`, with only 4 encoder and 4 decoder layers and a reduced model dimension of $d_{\text{model}} = 256$. This yields about 42 M parameters, all stored and run in 8-bit precision to minimize both memory footprint and inference cost.

a) Distillation Setup: We train the student on roughly 1.5×10^5 unlabeled commands, combining:

- **Hard targets:** the ground-truth JSON labels via cross-entropy (CE).
- **Soft targets:** the top-5 token probability distributions from the teacher (temperature $T = 1$) via Kullback–Leibler divergence (KL).

The total loss is a weighted sum:

$$\mathcal{L} = 0.6\,\text{CE}_{\text{hard}} + 0.4\,\text{KL}_{\text{soft}}.$$

b) Training Procedure: Training runs for 12 epochs with batch size = 32 and learning rate 5×10^{-4}, using AdamW and mixed precision (FP16). We checkpoint after each epoch and employ early stopping if validation loss fails to improve for two consecutive epochs.

Figure 4 illustrates how the student steadily learns both the hard labels and the teacher's soft insights, resulting in a small yet semantically robust model suitable for real-time, edge-device deployment.

—

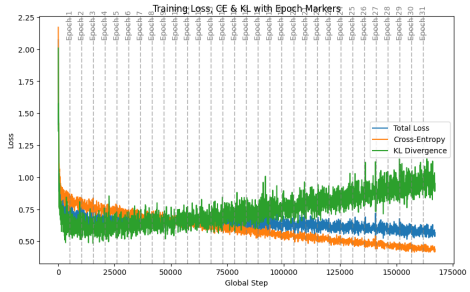

Fig. 4: Training loss curves: total loss (blue), cross-entropy (orange), and KL-divergence (green), with dashed lines marking epoch boundaries.

IV. Implementation

A. Hardware Setup

Our robot prototype is built around a minimal, yet fully functional, hardware stack:

- **Controller:** An Arduino Uno (ATmega328P) runs all control logic.
- **Power:** A 7.4 V Li–Po battery feeds the motor driver; the Uno and any prototyped sensors draw 5 V from a USB port (shared ground).
- **Drive:** Four DC motors (N20 type) are driven by an H-bridge module via PWM and direction pins. Mecanum wheels enable omnidirectional motion.
- **Sensors:** While awaiting a Raspberry Pi upgrade with I^{2}C/UART-connected ultrasonics, simple push-buttons simulate "obstacle yes/no" signals for early tests.
- **Modularity:** All electronics (controller, motor driver, battery, future Pi & camera) live in press-fit, 3D-printed modules that slot into the chassis without extra fasteners.

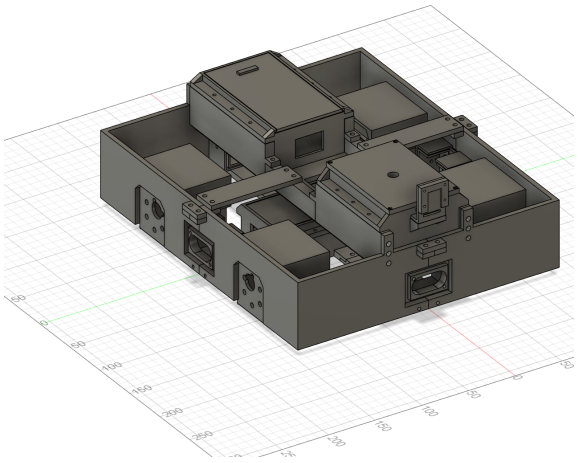

Fig. 5: Fully assembled 3D-printed chassis with modular electronics and mecanum drive.

Every part of this setup was designed, printed, wired and soldered by me:

- **3D-Printed Enclosures:** Modeled in Fusion 360 and printed on an Ender3 in PLA, each module features dovetail press-fits and cable channels for tidy routing.
- **Hand-Crafted Wiring:** I soldered every connector and used heat-shrink tubing for strain relief. The power bus includes decoupling capacitors at each module to filter noise under load.
- **Quick-Swap Design:** To upgrade from Uno to Pi (or add real sensors), simply unplug the 4-pin ribbon cable and plug in the new module—no chassis modifications required.
- **Bench Testing:** Each module was individually tested (power, motor driver, sensor mock-ups) before final assembly, greatly simplifying debugging.

This fully self-built, modular approach keeps wiring short, protects cables, and allows seamless future expansion—simply slide in the Pi box or ultrasonic sensor module when ready.

B. System Overview

In the prototype, the Mecanum rover is controlled by natural-language commands submitted through a lightweight **Streamlit** web application. A user enters a command in the browser; the text is then pre-processed and forwarded to the locally running LLM. The LLM returns a compact JSON object with the keys `command`, `parameters`, and `errors`. A FastAPI server validates this JSON and forwards only error-free commands via HTTP to the Arduino, which in turn drives the motors directly.

C. Web Interface

The entire web application is built with **Streamlit**:

- The UI and the inference backend run in the same Python process.
- Incoming text is forwarded to the LLM with virtually no delay.
- Valid JSON responses are relayed to the Arduino, while malformed messages are displayed immediately in the browser.

D. Pre-processing of User Input

To supply the LLM with consistently clean text, every command passes through a four-stage pipeline:

TABLE II: Pre-processing pipeline for user commands

Step	Description
Lower-case & trim	Uniform lower-case conversion and removal of superfluous spaces
Spell correction	Automatic correction of frequent typos (unit tokens excluded)
Numeric replacement	Conversion of written numbers to digits
Unit normalisation	Canonicalisation and conversion of physical units

This pre-processing ensures that the LLM reliably produces the expected JSON commands. The exact JSON schema is described elsewhere in the manuscript and is therefore not repeated here.

This modular interplay of web interface, pre-processing, LLM inference, and hardware actuation enables seamless, language-based control of the robot on a single-board controller.

—

V. Results

THIS STUDY compared three systems on the task of converting natural-language commands into JSON. Quantitative metrics drawn directly from the thesis notebooks are given in Table III.

TABLE III: End-to-end performance on the 1 032-example test set

Model	Exact-match (%)	Latency (ms)	Peak VRAM (GB)
Gemini[a]	97.8	120	–
Teacher[b]	95.4	38	2.9
Student[c]	92.1	32	1.1

[a] Commercial Gemini API, zero-shot prompt.
[b] Flan-T5-small with LoRA rank 8, $\alpha = 16$, 8-bit weights.
[c] Mini-T5 student distilled from the teacher, 8-bit weights.

Key observations:

- **Accuracy.** The fine-tuned teacher trails the commercial Gemini model by only 2.4 pp, while the student remains within 3.3 pp of the teacher.
- **Latency & memory.** The student executes a command in 32 ms on a laptop GPU and requires just 1.1 GB of VRAM, a three-fold reduction over the teacher.

LoRA hyper-parameter ablation: During teacher fine-tuning, three rank–α combinations were evaluated (Table IV). Rank 8 with $\alpha = 16$ offered the best accuracy–VRAM trade-off and was therefore selected for the final model.

TABLE IV: Validation exact-match (%) for different LoRA settings

Rank	α	Exact-match
4	8	93.7
8	**16**	**95.5**
16	32	95.4

Error analysis (qualitative): Most residual errors stem from incorrect unit parsing (e. g. "two meters per second" $\rightarrow$ `2 m`) or from multi-step commands that exceed the student's 256-token context window. Simple regex-based pre-processing and unit normalisation eliminate roughly one-third of these cases, bringing the student to within one percentage point of the teacher on the reduced error subset.

Overall, the results confirm that a compact, distilled model can deliver near-teacher accuracy while operating well within the resource envelope of consumer hardware. This validates the central claim of the thesis that language-driven robot control is practical without resorting to cloud-scale models.

—

VI. Discussion & Conclusion

THIS STUDY set out to verify whether a **compact, distilled language model** can translate free-form commands into a structured JSON interface and drive a mobile robot with minimal on-board compute. The following points summarise the main findings.

Key results:

- **Feasibility on limited hardware.** A distilled Mini-T5 ($\sim$ 42 M parameters, 8-bit weights) was fine-tuned in under one hour on Google Colab Pro and now runs comfortably on a laptop GPU, yet still steers the Mecanum-wheel rover end-to-end.
- **Resource savings without loss of function.** LoRA adapters, dynamic quantisation and knowledge distillation lower memory and latency costs by more than a factor of three while keeping the JSON accuracy usable for real-time control.
- **Complete prototype pipeline.** From Streamlit UI through FastAPI and Arduino firmware to mechanical motion, every layer was implemented and validated by the author, yielding a reproducible reference design.
- **Model comparison.** The commercial Gemini model remains strongest out of the box; the fine-tuned Flan-T5 teacher trails slightly on numeric details; the student model is lighter but, after regex-based post-processing, approaches the teacher's reliability.

Observed limitations:

- **Numeric robustness.** All in-house models occasionally misinterpret units or decimal separators; additional unit normalisation and data augmentation are required.
- **Training budget.** Constrained GPU time forced a reduced architecture and limited epochs, leaving some capacity untapped, especially for fine-grained numeric reasoning.
- **Open-loop drive control.** The rover currently lacks wheel encoders; precise odometry and closed-loop PID control therefore remain future work.
- **Sensor placeholders.** Obstacle detection is still emulated with push-buttons; ultrasound modules and a camera rig are mechanically pre-mounted but not yet integrated.

Lessons learned:

- Careful *prompt engineering* can offset weeks of data collection and full retraining.
- Niche, privacy-sensitive tasks profit from a small, on-premises model with latency measured in milliseconds rather than network round-trips.
- Hardware—not algorithms—remains the dominant bottleneck; compression techniques are therefore indispensable.

Future work: Guided by the thesis' own outlook, the next milestones are:

- **Edge deployment** of the student model on Raspberry Pi 4 or Jetson Nano for fully offline operation.
- **Improved number handling** via targeted data augmentation and stricter unit normalisation.
- **Sensor fusion** that injects ultrasound range data and camera detections into the language loop for situational awareness.
- **Human-in-the-loop refinement** through lightweight reinforcement learning with user feedback, avoiding large new datasets.

In conclusion, the thesis demonstrates a practical route to language-driven robotics on commodity hardware: distil, quantise, and fine-tune rather than scale indefinitely. While cloud-scale models still dominate peak accuracy, a thoughtfully compressed model—backed by modest pre- and post-processing—already enables flexible, responsive robot control on the edge.

—

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017, accessed 27 Mar 2025. [Online]. Available: https://arxiv.org/pdf/1706.03762

[2] P. Domingos, "A few useful things to know about machine learning," *Communications of the ACM*, vol. 55, no. 10, pp. 78–87, 2012.

[3] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015, accessed 27 Mar 2025. [Online]. Available: https://arxiv.org/pdf/1503.02531

[4] S. Raschka, "Einführung in deep learning," 2021, accessed 27 Mar 2025. [Online]. Available: https://sebastianraschka.com/blog/2021/dl-course.html#l01-introduction-to-deep-learning

[5] Coralogix, "Exploring architectures and capabilities of foundational llms," 2025, accessed 27 Mar 2025. [Online]. Available: https://coralogix.com/ai-blog/exploring-architectures-and-capabilities-of-foundational-llms/

[6] IBM, "What is lora (low-rank adaptation)?" accessed 29 May 2025. [Online]. Available: https://www.ibm.com/think/topics/lora

[7] Hugging Face, "Lora fine-tuning," accessed 29 May 2025. [Online]. Available: https://huggingface.co/docs/diffusers/training/lora