LangSpace: Spatial Reasoning with Natural Language Queries using 3D Scene Graphs and Fine-Tuned LLMs

Stanford CS224N Custom Final Project

Laszlo Szilagyi
SCPD
Stanford University
laszlosz@stanford.edu

Abstract

This project explores spatial reasoning with large language models (LLMs) over 3D scene graphs to enable high-level task planning from natural language instructions. Previous works, such as SayPlan [1] and SayCan [2], have demonstrated the potential of LLMs in robotic planning, but rely on frontier models like ChatGPT-4, which are computationally expensive and require internet access. In this work, I investigate fine-tuning open-source LLMs (Llama3 1/3/8B [3]) for task planning over structured scene representations. I build my own 3D scene graph task planning system to assess alternative inference strategies, such as single-shot and iterative multi-prompting and to generate training data for SFT and PPO fine tuning. The goal is to determine whether a fine-tuned, locally deployable model can achieve planning performance comparable to or exceeding that of general-purpose frontier LLMs while improving efficiency. I will present experimental results evaluating the impact of representation choices, prompting techniques, and fine-tuning configurations on plan quality and execution feasibility.

1 Key Information

- TA mentor: Johnny Chang
- External collaborators: No
- External mentor: Christopher Agia
- Sharing project: No

2 Introduction

Task planning in robotics refers to the process of generating a sequence of high-level actions that enable a robot to achieve a specific goal in a structured environment. It requires reasoning over possible actions, object affordances, and environmental constraints. Traditional approaches have included rule-based symbolic planning with formalisms such as Planning Domain Definition Language (PDDL) and heuristic search-based approaches. Large language models (LLMs) have demonstrated impressive results in developing generalist planning agents for diverse tasks with natural language as input. A recent advancement is to leverage LLMs in robotics task planning [1][4][2]. In these cases, we ground the LLM with a representation from the real-world environment. While some approaches augment LLMs with image and video modalities, 3D scene graphs are showing promising results as a 3D representation in LLM spatial reasoning.

As works, like [1] demonstrated, LLMs grounded with a 3D scene graphs can generate high level step-by-step task plans from natural language input to instruct robotic agents executing tasks in real world environments. However, methods, like SayPlan relies on an online generic frontier LLM (e.g., ChatGPT-4) rather than a more efficient, locally deployable, and smaller (fine-tuned) model. These

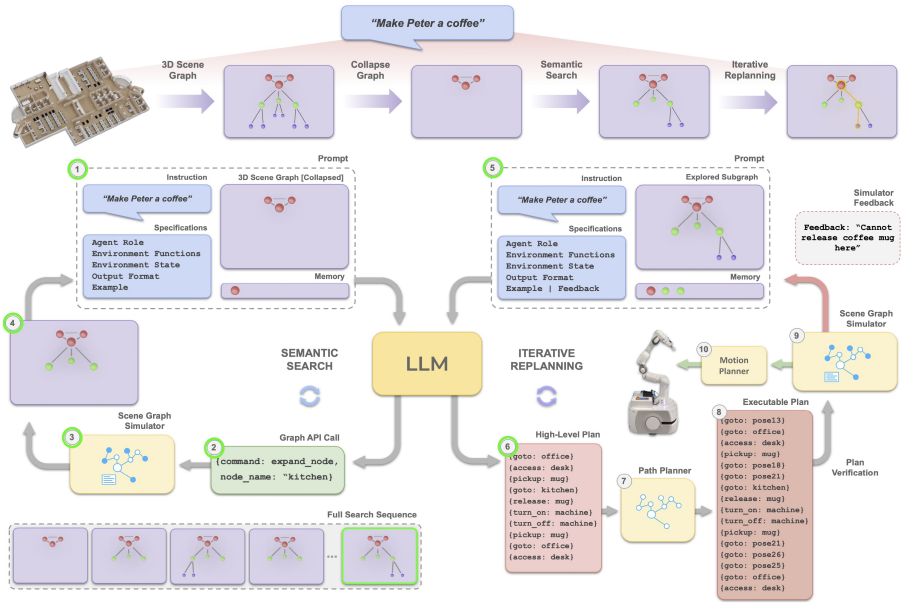

Figure 1: Overview of the SayPlan [1] system architecture including the Semantic Search stage (left) and Iterative Replanning (tight) phases. This course project will focus on the subset of the steps marked with green (step 1-6).

implementations don't have task-specific fine-tuning or Retrieval-Augmented Generation (RAG) explored. Also, there is no real-time constraint on planning performance, making it unclear how well they scale in embedded robotics applications.

In this project I investigate the possibility of fine tuning smaller, LLMs, like 1B, 3B and 8B Llama for task planning. The eventual goal to achieve a fully offline, locally deployable embedded language planning system. Such an approach is not only relevant for robotics, but for use cases like augmented reality assistance too.

To execute my experiments I build my own 3D scene graph robotics task planning system, leveraging the Gemini 1.5 API as a baseline and for training data generation. I experiment with a two phase fine tuning configuration using the combination of Supervised Fine-Tuning (SFT) and Proximal Policy Optimization (PPO). While my fine tuning efforts do not conclude as expected, I outline a plan to improve the siytem in a next iteration, including representation and prompting improvements, data generation and redefining the problem specification.

3 Related Work

SayCan [2] is one of the first notable works to leverage LLMs for robotic planning, grounding language-based reasoning in real-world feasibility through vision-based affordance functions. It integrates image/video inputs to assess action feasibility, ensuring that robots execute tasks that are both semantically meaningful and physically viable. By combining LLM reasoning with vision-driven affordance scores, SayCan enhances task success rates, enabling interpretable, long-horizon robotic planning in real-world settings. SayPlan has some severe limitations though. It relies on a pre-trained affordance model that scores the feasibility of actions based on objects the robot has already perceived. This means, that the robot cannot plan ahead for objects that are out of view and It lacks spatial memory—if an object is behind a table, it won't reason about looking around the table first. It lacks true spatial and 3D scene understanding.

SayPlan [1] improves on the spatial understanding limitations with a method to ground LLMs in large-scale robotics task planning via 3D scene graphs (3DSGs). 3DSGs encode spatial, semantic, and relational information about an environment, making them a suitable intermediate representation for robot reasoning. As described on Figure 1, SayPlan operates across two stages to ensure scalability: (left) Given a collapsed 3D scene graph and a natural language task instruction, semantic search

is conducted by the LLM to identify a suitable subgraph that contains the required items to solve the task; (right) The explored subgraph is then used by the LLM to generate a high-level task plan, where a classical path planner completes the navigational component of the plan; finally, the plan goes through an iterative re-planning process with feedback from a scene graph simulator until an executable plan is identified. As mentioned in the introduction, SayPlan is an online frontier LLM-based implementation, which I have intended to address in this work. It also recognizes the need for fine tuning [5].

While ConceptGraphs [4] is not a planning algorithm, it represents the state of the art in online scene graph construction and has been a key motivation for this project (note, that SayPlan does not implement a scene graph mapping backend). ConceptGraphs builds an open-vocabulary 3D scene graph from a sequence of posed RGB-D images by leveraging generic instance segmentation models to segment RGB image regions, extract semantic feature vectors, and project them into a 3D point cloud. These regions are incrementally fused from multiple views, forming 3D objects with associated vision and language descriptors. Large vision-language models then caption these objects and infer inter-object relationships, generating the edges of a 3D scene graph. This structured representation enables rich scene understanding and can be easily translated into text, making it valuable for LLM-based task planning. In the next phase of this project, I plan to experiment further with ConceptGraphs as a language-embedded scene graph backend, integrating it more closely with SayPlan.

4 Approach

The environments used for task planning are represented as 3D scene graphs \todo{figure: }, which provide a hierarchical breakdown of a building, including floors, rooms, objects, transition paths, and affordances (e.g., open, close, switch on). These graphs can also encode the environment's state and update dynamically as tasks are executed. One aspect of the experiments is how to best represent these 3D scene graphs for language models.

The Planning Domain Definition Language (PDDL) is a widely used representation of scenes and affordances in robotics planning. However, the examples I could get my hands on (via the Taskography API [6]) proved to be too verbose, exceeding 30,000 tokens even for small scenes. For this reason, I relied on a JSON representation I have implemented based on the SayPlan paper, which reduced the representation size to around 3,000 tokens (the paper does not share code).

This representation includes the following information:

- The list of objects that the agent can grab and move around, for example, teddy bear, apple, bottle.
- The list of assets that the agent can interact with to store objects. There are openable assets, like fridges and microwaves; switchable assets, like sinks and toilets; and passive assets, like dining tables, chairs, and beds.
- The list of rooms.
- The agent itself.
- The list of connections between rooms, objects, assets, and the agent.

I have experimented with two variants of this representation: one with generic node IDs and one with natural language node IDs. The latter turned out to be more effective. An example scene graph is attached to the submitted code.

To interact with the environment, the agent can perform the following actions on the graph:

- `goto(Pose)`: Move the agent to an adjacent room. Respecting room adjacencies defined in the scene graph is mandatory.
- `access(Asset)`: Provide access to the set of affordances associated with an asset node and its connected objects.
- `pickup(Object)`: Pick up an accessible object from the accessed node.
- `release(Object)`: Release a grasped object at an asset node.

- `turn_on(Object) / turn_off(Object)`: Toggle an object at the agent's node, if accessible and has an affordance.
- `open(Asset) / close(Asset)`: Open or close an asset at the agent's node, affecting object accessibility.

I have implemented the graph class, which is able to load scenes from the Gibson [7] dataset. It can carry out the actions listed above, be visualized, and integrate the scene graph simulator with a language model to simulate the behavior of the plans step by step. Figure 2 demonstrates the representation layers of the Gibson dataset. See the extracted scene graph in Figure 3.

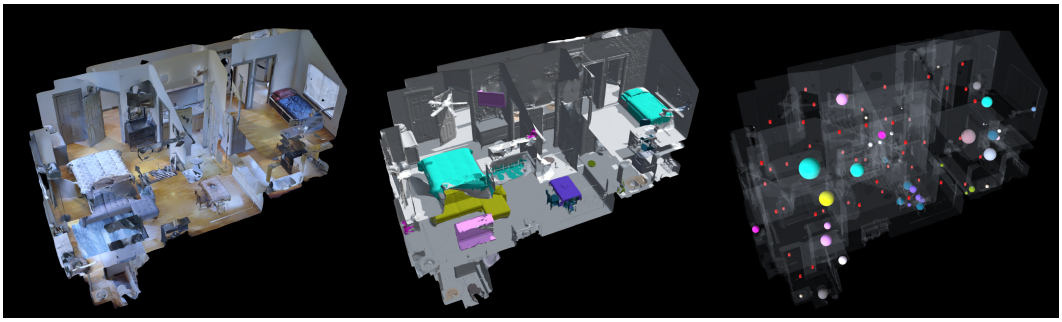

Figure 2: The representation layers of the Gibson [7] dataset on the Benovelance scene. Pointcloud (left), semantic segmentation (center) and scene graph (right). I am extracting the scene graph.

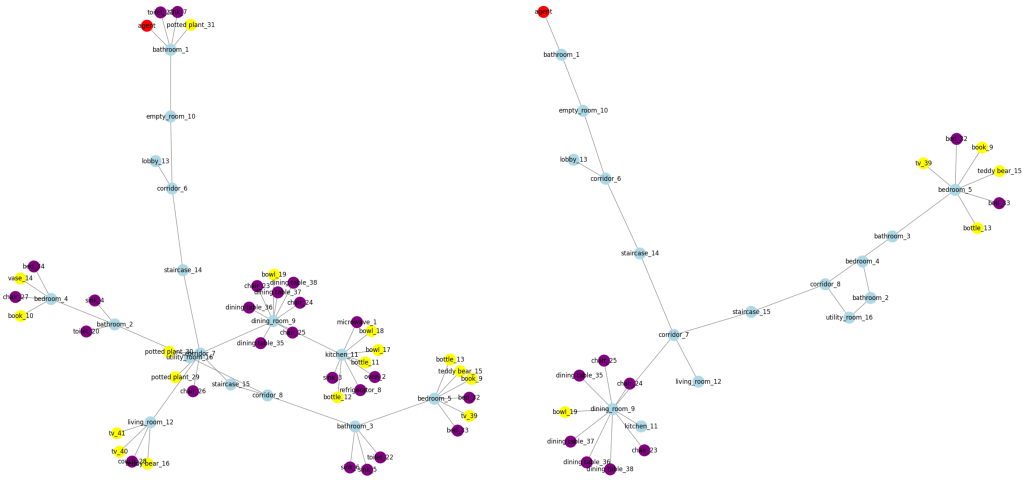

Figure 3: A full graph (left) and a pruned graph (right) of the Gibson Benovelance scene.

4.1 Plan generation with LLMs

A plan is a series of actions that the agent executes to accomplish a task defined in natural language. My goal is to assess how well LLMs can perform these tasks. In general, the prompt to the LLM consists of a system prompt describing the rules, the scene graph, and the natural language description of the instruction. The LLM should return a JSON-formatted string with the list of steps to execute, which can be passed to a downstream subsystem for execution.

A typical natural language description looks like the following:

"Please go to bedroom_5, pick up tv_39, and place it on couch_28 in living_room_12."

A typical plan looks like the following:

['goto(empty_room_10)', 'goto(corridor_6)', 'goto(staircase_14)', 'goto(corridor_7)', 'goto(dining_room_9)', 'goto(kitchen_11)', 'goto(dining_room_9)', 'goto(corridor_7)', 'goto(staircase_15)', 'goto(corridor_8)', 'goto(bathroom_3)', 'goto(bedroom_5)', 'pickup(tv_39)', 'goto(bathroom_3)', 'goto(corridor_8)', 'goto(staircase_15)', 'goto(corridor_7)', 'goto(living_room_12)', 'access(couch_28)', 'release(tv_39)']

The resulting plan can be parsed and simulated by the scene graph instance step by step. If validation steps (see later) are provided (e.g., `tv_39` on `couch_28`), the plans can be fully validated.

Plan generation can be carried out in multiple configurations. I evaluate two key design choices: whether to pass the full scene graph to the model or prune it using a semantic search pre-processing phase, and whether the model generates the plan in a single pass (single-shot) or iteratively, refining and validating each step. These choices are particularly important for large scene graphs and models with constrained token limits. Figure 3 in the appendix demonstrates the effect of graph pruning.

The baseline configuration takes the scene graph alongside the instruction and system prompt as-is and produces the plan in one shot. However, such prompts can be significantly large for scenes with many rooms, assets, and objects, sometimes exceeding 10,000 tokens, which is unfeasible for some language models and can slow down processing for others. The graph pruning mechanism I have implemented uses a language model to identify the rooms relevant to the task description and includes objects and assets only in those rooms. This can reduce scene graph sizes (token count) by more than 60 percent.

For iterative re-planning, I needed a mechanism to validate a plan and a way to provide meaningful feedback to the model to improve re-planning performance. Plan validation is performed in two stages. First, plan feasibility is checked: the `execute_plan` method executes the plan step by step, and if an unfeasible step is found, it returns a description of the problem (e.g., `room_10` and `room_14` are not adjacent! Follow room transition rules.). The second validation step checks whether the generated plan achieves the desired final state. For this, I generate a validation step for each task description, and the validator's `check_objects_classes` method returns a description if an invalid execution is detected (e.g., `bowl_10` is not on `chair_11`, replan!). Invalid plans and validation messages are appended to the prompt, and a new inference is initiated.

As a summary, the evaluated design choices are:

- Single-shot, full graph
- Single-shot, pruned graph
- Iterative, full graph
- Iterative, pruned graph

For completeness, step-by-step planning should also be mentioned. This is technically possible with the implemented scene graph simulator but has not been investigated. It requires a more sophisticated prompting mechanism with external function calls and a refined training data generation strategy. However, note that step-by-step planning typically involves 10+ LLM inferences, which can be time-consuming.

4.2 Fine tuning

For fine-tuning, I experimented with Llama 3.2 1B, 3B, and Llama 3.1 8B. The fine-tuning follows a two-phase approach. In the initial **Supervised Fine-Tuning (SFT)** phase, the goal is to establish a strong foundational understanding of the planning task using a dataset of input-output pairs (graph + instruction $\rightarrow$ plan pairs). I reused Torchtune's multi-GPU LoRA recipe with configurations specific to the Llama model variants. These configurations apply LoRA modifications to the model's **query, value, and output projection layers**, while also incorporating LoRA into the MLP layers but not the output layer. I prepare the dataset in the OpenAI chat format.

To refine the model's behavior, I use Proximal Policy Optimization (PPO). PPO enables the model to optimize for specific objectives, such as plan executability, plan validity, and plan length, through an iterative reinforcement process. I leverage HuggingFace PEFT (Parameter-Efficient Fine-Tuning) for LoRA and HuggingFace TRL (Transformers Reinforcement Learning) for PPO. I load the SFT fine-tuned models and PPO-specific training data through a custom data loader I implemented. I have

also developed a task-specific reward function leveraging the plan execution and validation methods discussed earlier.

For SFT, the training data follows the ChatGPT chat format. The system prompt, the scene graph, the dedicated room connectivity graph, and the task description are provided as input, while a valid plan serves as the ground truth output. The data is stored in a JSONL file, which is natively supported by the Torchtune framework.

```
"messages": [
    {"role": "system", "content": system_prompt},
    {"role": "user", "content": f"The room adjacencies to strictly follow:\
    {room_graph}
    The 3D Scene Graph in JSON format: {pruned_scene_graph}
    The task description: {task_description}"},
    {"role": "assistant", "content": f"{plan}"}
]
```

An important aspect of PPO training data is the need for online validation of the generated data. For this reason, the training data must be accompanied by a proper reward function. In the planner case, I score plans based on their executability and validity: executable and valid (+1), executable but not valid (+0.5), or neither (-1). I implemented the reward function by pairing natural language task descriptions with the validation steps described earlier. During inference, the reward function, alongside the generated plans, also receives the validation steps. Inside the reward function, I instantiate the scene graph from the input, test the plan's executability and validity, and score it accordingly.

The output of the scene graph data loader is as follows:

```
query = f"{system_prompt} The room adjacencies to strictly follow: \
{room_graph}
The 3D Scene Graph in JSON format: {scene_graph} \
The task description: {task_description}"
return {
    "input_ids": self.tokenizer(query, return_tensors="pt", \
    truncation=True, max_length=512)["input_ids"],
    "scene_graph": scene_graph,
    "verifying_step": verifying_step
}
```

4.3 Baseline

As a baseline, I use the *Single-shot, full graph* generation configuration. See results in the results section.

5 Experiments

5.1 Training Data Generation

To produce the training data, I follow the process described in Figure 4.

In the Gibson [7] dataset, I have 35 tiny and 105 medium building scans, which include point clouds, segmentation masks, and a graph describing relationships and other attributes (e.g., material). To extract the scene graph, I instantiate my LSSceneGraph class with the help of Taskography's [6] Building class. The Building class implements the parsing of the Gibson files, which I then leverage to initialize my class representation. Once instantiated, the LSSceneGraph instance can generate the scene graph in JSON format for the given Gibson scene.

To produce the task descriptions, I use the Gemini API. I provide the JSON scene graph and prompt the model to generate 50 natural language task descriptions for each task, along with the corresponding evaluation steps. I consider two main categories of tasks: "concrete" and "open." Concrete tasks instruct the agent to perform an action with a specific object, e.g., "Please, bring book_11 from

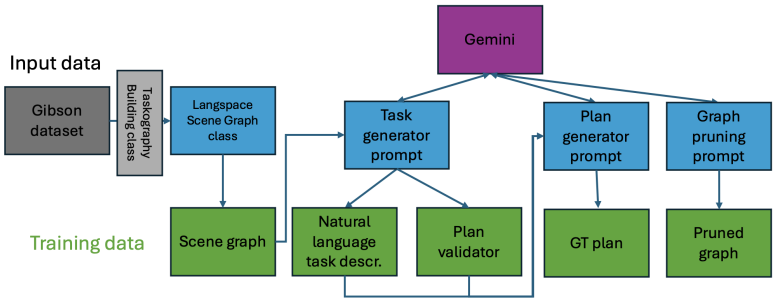

Figure 4: The data generation flow leveraging the Gibson [7] dataset and the Gemini 1.5 API.

bedroom_4 to table_2 in kitchen_1." Open tasks, on the other hand, request an action involving any object from a specific class, e.g., "Please, bring a book to table_2 in kitchen_1." Accordingly, the validation step checks for a specific object in the designated location/asset in the first case and for an object class in the second case. Tasks can also be designed to involve multiple objects and classes, generating multiple validation steps per description. For training constrained LLMs, I also generate pruned graphs using the previously described mechanism.

To complete the training data generation, I generate ground truth plans through actual planning execution using the Gemini API. I feed in the system prompt, the scene graph, the room transition graph, and the textual descriptions. Then, I validate each generated plan using the task validator steps. This allows me to evaluate the online LLM performance and extract valid plans as ground truth for fine-tuning. As will be discussed in the results, the rate of valid plans has been relatively low. In total, 3,141 ground truth task-plan pairs have been generated. These assets are saved per scene and later transformed into the SFT and PPO formats described earlier.

5.2 Evaluation Method

The evaluation metrics and their measurement are as follows:

- **Plan executability:** Can the generated plan be executed end-to-end without breaking room transition and affordance rules? Measured using the success rate of the scene graph plan execution method.
- **Plan validity:** If the generated plan is executable, does it result in the desired state (e.g., objects placed in the correct rooms or assets)? Measured using the success rate of the scene graph plan validator method alongside the plan validation steps.
- **Plan parsability:** Is the generated plan parseable? This question is particularly relevant for the fine-tuning steps. Measured using the success rate of the `json.loads()` method.
- **Plan generation timeout rate:** Does the generation time out (e.g., not finishing within 20s)? Measured by counting the timeout rate of plan generation.
- **Mean plan generation time:** The average time taken to generate a plan. Measured using the `time` method.
- **Mean plan length:** The average number of steps in the generated plans. Measured using the `len()` method after parsing the plan into a Python list.

5.3 Experimental Details

The model used as an online frontier model reference and for task generation was Gemini 1.5. Initially, I started with ChatGPT-4o but encountered throughput quota issues, leading me to switch to the Google Cloud product instead. For local deployment and fine-tuning, I used the Meta Llama 3.2 1B Instruct, Llama 3.2 3B Instruct, and Llama 3.1 8B Instruct models. For deployment, I leveraged the LangChain [8] framework, which simplified switching between online and offline models. For fine-tuning, I used a Google Cloud VM with 8 A100 GPUs to accelerate training iterations.

As mentioned earlier, for the **Supervised Fine-Tuning (SFT)** phase, I reused Torchtune's [9] multi-GPU LoRA recipe with configurations specific to the Llama model variants. These configurations

apply LoRA modifications to the model's **query, value, and output projection layers**, while also incorporating LoRA into the MLP layers but not the output layer. This configuration employs **AdamW** as the optimizer with a **learning rate of 3e-4**, **weight decay of 0.01** for regularization, and **fused operations enabled** for improved GPU efficiency. To stabilize training, it applies a **cosine learning rate decay with 100 warmup steps**, gradually increasing the learning rate before following a smooth decay curve. The **LoRA rank is set to 8**, balancing accuracy and memory usage, and training is conducted using **bf16 precision**. I trained on 3,141 training instances for 10 epochs across all three LLMs. The configuration file is included in the code submission.

I could not conclude my PPO experiments. I experimented with HuggingFace PEFT (Parameter-Efficient Fine-Tuning) for LoRA with HuggingFace TRL (Transformers Reinforcement Learning) and also attempted to create my own PPO configuration using Torchtune. I implemented custom data loaders and reward functions for each framework. However, I am still troubleshooting and have not yet managed to get them working properly. I aim to have at least one framework running by the poster session and will share my findings then. Nonetheless, I am including the code and configuration drafts I experimented with, including my custom data loaders and reward functions, in the code submission.

5.4 Results

Gemini 1.5	Valid (%)	Executable (%)	Parseable (%)	Timeout (10s, %)	Mean time (s)	Mean length (steps)
Single, Full	50.88%	4.11%	41.47%	2.14%	1.06	14.95
Single, Pruned	38.53%	0.88%	53.82%	8.23%	0.95	13.67
Iterative, Full	62.94%	4.41%	32.06%	2.05%	3.87	15.76
Iterative, Pruned	55.29%	1.18%	40.29%	4.71%	10.45	15.07

Table 1: Results with the Gemini 1.5 experiments. Iterative, full-graph plan generation yields the best performance in terms of validity, though execution success remains low across all configurations.

SFT	Valid	Executable	Parseable	Timeout	Mean time	Mean length
Llama 3.2 1B	0%	0%	0%	n/a	n/a	n/a
Llama 3.2 3B	0%	0%	0%	n/a	n/a	n/a
Llama 3.1 8B	0%	0%	0%	n/a	n/a	n/a

Table 2: Results after SFT. At this point the generated results are not even parseable.

The training loss after 10 training epocs:

- Llama 3.2 1B: 0.0070 (stagnation after 4 epochs)
- Llama 3.2 3B: 0.0064 (stagnation after 4 epochs)
- Llama 3.8 8B: 0.0030 (stagnation after 10 epochs)

SFT	Valid	Executable	Parseable	Timeout	Mean time	Mean length
Llama 3.2 1B	0%	0%	0%	n/a	n/a	n/a
Llama 3.2 3B	0%	0%	0%	n/a	n/a	n/a
Llama 3.1 8B	0%	0%	0%	n/a	n/a	n/a

Table 3: Results after PPO in progress. The training script is still under development

In summary the Llama 1B and 3B models inference returns no usable results, while the Llama 8B model variant produces some promising results, however, more training data is needed.

5.5 Analysis

The task, at least with the current promt and representation, proved to be challenging even for a frontier online LLM, like Gemini 1.5. Even iterative replanning suffered, which was surprising.

As I've learned the performance is highly sensitive to the prompt and the graph representation. Analyzing the errors showed that the largest portion of errors came from bad room transitions, this

was partially addressed by adding the room transitions sub-graph to the prompt. The second largest group was ignoring affordances (like opening the door of the fridge). The prompt sensitivity suggests that improving the graph representation (e.g. separating room transition links from per-room object and asset relations with better naming conventions) and providing more in-context learning examples (e.g. on room transitions and affordance handling) has potentials to further improve performance.

Unfortunately, fine tuning the smaller LLMs has not concluded. Based on the mixed experience with Gemini, it is safe to assume, that the task in its current form is too complex for the smaller models: The prompts are too long, the representation is suboptimal and the complexity of the task planning is too large. The Lllama 8B model was showing some promising results, need to iterate with more data and focus on smaller scenes only.

5.6 Conclusion

This project set out to evaluate whether large language models—especially fine-tuned, locally deployable ones—can effectively generate high-level task plans from natural language instructions over 3D scene graphs. The findings so far suggest that this is a non-trivial problem, even for state-of-the-art frontier models like Gemini 1.5.

That said, there are several promising directions for further exploration:

- **Prompt and Graph Representation Optimization:** Performance is highly sensitive to prompt construction and scene graph representation. Improving naming conventions, separating object relationships from room transitions, and including more targeted in-context examples—especially for affordance usage and movement—are likely to yield gains.
- **Training Data Expansion:** The 8B model exhibited signs of overfitting during supervised fine-tuning (SFT), suggesting the need for a significantly larger and more diverse training set. Expanding to multi-object, open-ended, and more varied tasks could easily increase dataset size by $5\times$ or more.
- **Graph Embeddings for Compression:** Representing the scene graph as an embedding or using other compact encodings could help reduce prompt length and make the task more tractable for smaller models.
- **Step-by-Step Planning:** Although supported by the system, step-wise planning—where the model generates and validates one action at a time—was not tested due to time constraints. This approach may improve reliability but comes at the cost of increased inference time due to multiple LLM calls.
- **Hierarchical Planning:** Another unexplored option is decomposing the planning problem into hierarchical sub-problems. For example, one model could plan inter-room navigation, while another handles intra-room object manipulation. This might allow specialization and reduce complexity per model.

Looking ahead, it's worth considering whether an end-to-end LLM-based approach is ideal for this type of spatial planning. A hybrid system—where the LLM interprets natural language tasks and outputs structured sub-goals, which are then solved using classical graph-based planners—may be a more efficient and scalable architecture. Such a system would combine the interpretability and generalization capabilities of LLMs with the precision and reliability of symbolic planners.

Although the project made less progress on fine-tuning than originally intended, the infrastructure built and the insights gained have laid a solid foundation. I plan to continue refining the approach along the directions outlined above.

References

[1] Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. Sayplan: Grounding large language models using 3d scene graphs for scalable task planning. In *7th Annual Conference on Robot Learning*, 2023.

[2] Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel

Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can and not as i say: Grounding language in robotic affordances. In *arXiv preprint arXiv:2204.01691*, 2022.

[3] Yutong Wu, Di Huang, Wenxuan Shi, Wei Wang, Lingzhe Gao, Shihao Liu, Ziyuan Nan, Kaizhao Yuan, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Yewen Pu, Dawei Yin, Xing Hu, and Yunji Chen. Inversecoder: Unleashing the power of instruction-tuned code llms with inverse-instruct, 2024.

[4] Qiao Gu, Alihusein Kuwajerwala, Sacha Morin, Krishna Murthy Jatavallabhula, Bipasha Sen, Aditya Agarwal, Corban Rivera, William Paul, Kirsty Ellis, Rama Chellappa, Chuang Gan, Celso Miguel de Melo, Joshua B. Tenenbaum, Antonio Torralba, Florian Shkurti, and Liam Paull. Conceptgraphs: Open-vocabulary 3d scene graphs for perception and planning. *arXiv*, 2023.

[5] Jiawei Zhang. Graph-toolformer: To empower llms with graph reasoning ability via prompt augmented by chatgpt, 2023.

[6] Christopher Agia, Krishna Murthy Jatavallabhula, Mohamed Khodeir, Ondrej Miksik, Vibhav Vineet, Mustafa Mukadam, Liam Paull, and Florian Shkurti. Taskography: Evaluating robot task planning over large 3d scene graphs. In *Conference on Robot Learning*, pages 46–58. PMLR, 2022.

[7] Fei Xia, Amir R. Zamir, Zhiyang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson Env: real-world perception for embodied agents. In *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*. IEEE, 2018.

[8] Langchain homepage. `https://www.langchain.com/`.

[9] torchtune GitHub page. `https://github.com/pytorch/torchtune`.