Integrating Large Language Models for Enhanced Trajectory Generation in Mobile Robotics

ANAVART PANDYA

Introduction

In recent years, the realm of large language models (LLMs) has significantly expanded, impacting various fields, including robotics. This report explores the innovative integration of LLM concepts and other type of newly developed Neural Networks (like Liquid Neural Networks) into the domain of robotics, particularly for trajectory generation in mobile robots. The burgeoning interest in LLMs has led to novel implementations, prompting a shift in how robotic movements and decision-making processes are conceptualized and executed.

This study presents a comprehensive survey of various methodologies that leverage LLMs, including MiniGPT4, NanoGPT, and a streamlined version of NanoGPT – a simplified bigram model with a decoder-only architecture. Emphasis is placed on the lite version of NanoGPT, which was meticulously developed and employed to generate trajectories for a mobile robot with two degrees of freedom. This robot, pre-trained on a set of trajectories, showcases the practical application of LLMs in creating efficient, adaptable, and intelligent robotic movement patterns.

Through rigorous analysis and experimentation, this report demonstrates the potential of LLMs in enhancing robotic capabilities, offering insights into future advancements in the field. The following sections will delve into the methodologies, implementation, results, and implications of this pioneering approach.

Overview of MiniGPT4

MiniGPT4 is an innovative AI model that merges vision and language processing capabilities. It's designed to understand and generate language based on visual inputs.

1. **Functionality**: MiniGPT4 specializes in interpreting visual data and converting it into language. This includes tasks like image description and translating visual concepts into text-based outputs.

2. **Architecture**:

    - **Vision Encoder**: The model includes a vision encoder, equipped with a Vision Transformer (ViT) and Q-Former. This part of MiniGPT4 is responsible for processing and understanding visual information.

- **Language Model**: Linked to the vision encoder is the Vicuna large language model. Vicuna's role is to handle the language aspect, translating the visual data processed by the vision encoder into coherent text.

- **Linear Projection Layer**: A key component of MiniGPT4 is a linear projection layer that bridges the vision encoder and the language model. This layer is essential for aligning visual and textual data, ensuring seamless integration of the two. This layer is the only component that requires training to align the visual features effectively with the Vicuna language model, highlighting the model's efficiency

3. **Training Approach**: MiniGPT4 employs a two-stage training process. Initially, it is trained with a large set of image-text pairs, which helps the model learn to associate visual content with appropriate language descriptions. The second stage involves fine-tuning the model to improve its ability to perform vision-language tasks.

4. **Capabilities**: MiniGPT4 is notable for its ability to perform a range of vision-language tasks without the need for additional external vision models. Its architecture allows for direct alignment of visual information with language processing, enabling it to handle various tasks that combine both elements.

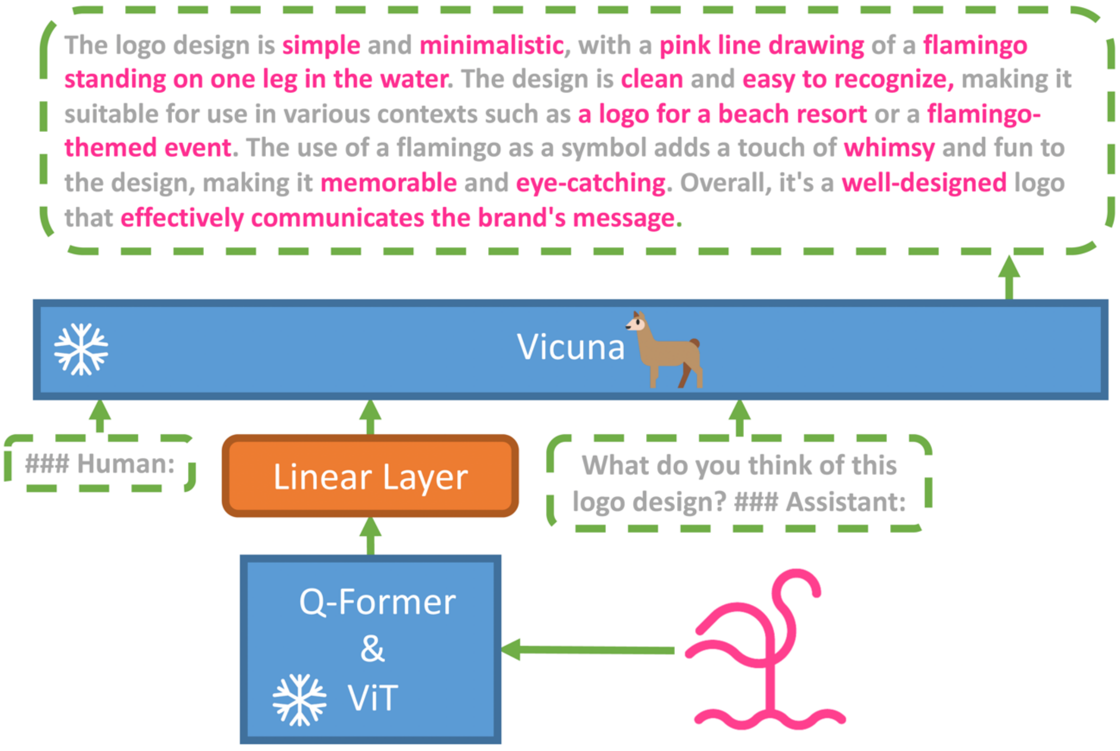

The logo design is simple and minimalistic, with a pink line drawing of a flamingo standing on one leg in the water. The design is clean and easy to recognize, making it suitable for use in various contexts such as a logo for a beach resort or a flamingo-themed event. The use of a flamingo as a symbol adds a touch of whimsy and fun to the design, making it memorable and eye-catching. Overall, it's a well-designed logo that effectively communicates the brand's message.
Vicuna
### Human:
Linear Layer
What do you think of this logo design? ### Assistant:
Q-Former & ViT

Figure 1: MiniGPT-4 Architecture, source: MiniGPT-4: Enhancing Vision-language Understanding with Advanced Large Language Models

Using MiniGPT-4 Architecture for Trajectory Generation

Understanding the integration of MiniGPT4 for trajectory generation in mobile robotics, we can focus on the two critical components: the vision encoder and the Vicuna LLM, which work in tandem to map the environment and generate a trajectory.

1. **Environment Mapping through Vision Encoder**: The first step involves the vision encoder, which includes a Vision Transformer (ViT) and Q-Former. This component of MiniGPT4 is tasked with analysing and understanding the robot's surroundings. It processes the visual data to create a comprehensive vector, encapsulating all necessary information about the environmental conditions and scenarios. This vector serves as a detailed digital map, reflecting elements like obstacles, terrain, and spatial layout, crucial for informed navigation.

2. **Trajectory Generation using Vicuna LLM**: Once the environment is mapped, the output vector from the vision encoder is then relayed to the Vicuna large language model. However, before reaching Vicuna, this vector passes through an input layer designed to translate the vision encoder's output into a format suitable for the language model. The Vicuna LLM, upon receiving this translated input, undertakes the task of trajectory generation. It processes the environmental information and converts it into a trajectory plan. This plan is articulated as a string that specifies the required velocities and angular velocities at each instance for the mobile robot. The trajectory string is not just a path but a detailed set of instructions that guide the robot to successfully accomplish its task, considering the nuances of the given environment.

In this setup, the vision encoder and Vicuna LLM are intricately connected, where the encoder's detailed environmental mapping is seamlessly transformed into actionable trajectory plans by the LLM. This integration allows the mobile robot to navigate complex environments intelligently and efficiently, adapting to changing conditions and achieving its objectives with precision.

While the initial goal of this project was to implement a complex architecture akin to MiniGPT4 by pretraining it on a large dataset for operation on local computers, the attempt was met with substantial challenges. The lack of sufficient computational resources and power made it unfeasible to pursue this implementation within a local system environment.

However, the promise of utilizing architectures similar to MiniGPT4 for trajectory generation in mobile robotics remains and the implementation effort provided great insights in the domain. But as mentioned above, training such a complex architecture to assess surroundings and produce a meaningful trajectory is a resource-intensive endeavour. Therefore, this report will pivot to focus on a more feasible aspect: generating a meaningful trajectory continuation using a large language model (LLM) given an initial set of trajectory points, which is one of the two key aspects used in the process mentioned above.

For this task, we turn our attention to models that excel in **self-attention** and possess a **decoder**-like architecture, key features for efficiently processing sequential data like trajectory points. One such architecture is NanoGPT.

Overview of NanoGPT

NanoGPT is a scaled-down version of the larger GPT models, designed to offer similar functionalities but with a structure that is more manageable and less resource-intensive.

Architecture of NanoGPT

1. **Transformer-Based Model**: At its core, NanoGPT follows the transformer architecture, which is fundamental to GPT models. This architecture is renowned for its ability to process sequential data, making it ideal for tasks involving natural language understanding and generation.

2. **Decoder-Only Structure**: Similar to GPT3/4, NanoGPT employs a decoder-only structure. This means it's designed primarily for generating text rather than encoding or interpreting input data.

3. **Self-Attention Mechanism**: A key feature of NanoGPT, borrowed from its larger counterparts, is the self-attention mechanism. This allows the model to weigh the importance of different parts of the input sequence, leading to more contextually relevant outputs.

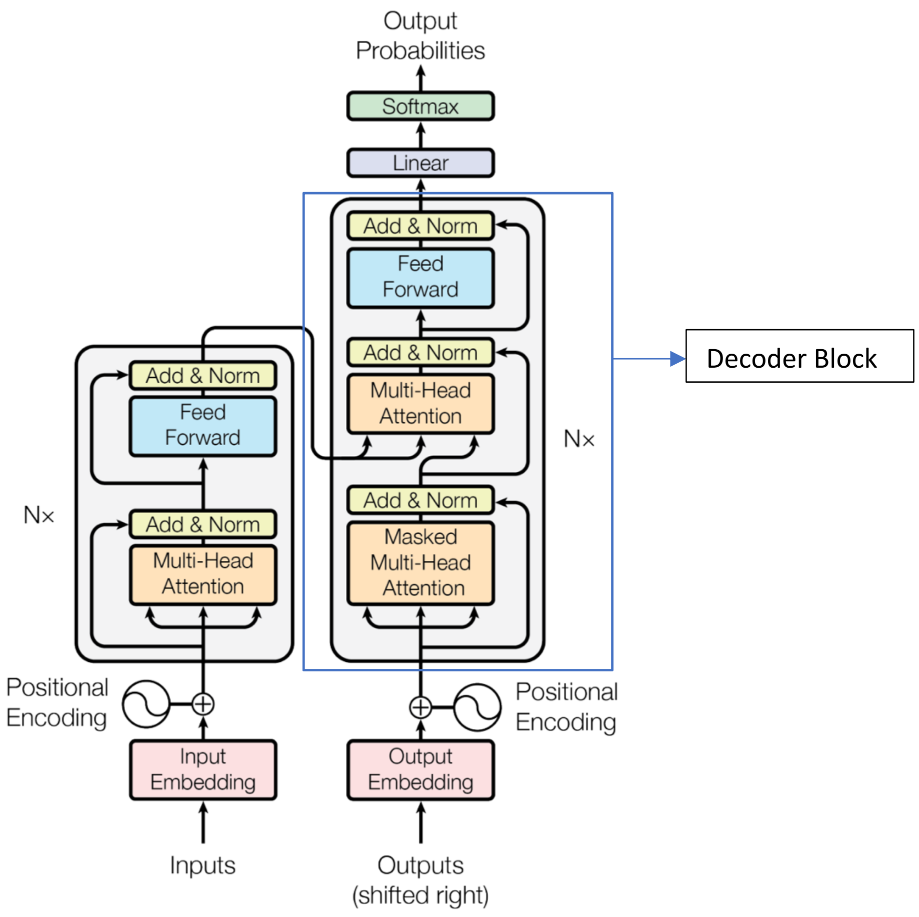


Figure 2: The Transformer – model Architecture, source: Attention Is All You Need

Similarities to GPT3/3.5/4

1. **Architecture and Mechanism**: Like GPT3/4, NanoGPT relies on the transformer architecture and the self-attention mechanism. This similarity ensures that NanoGPT retains the core functionalities that make GPT models effective at sequence generation and prediction.

2. **Language Processing Abilities**: Despite its smaller size, NanoGPT shares the language processing capabilities of its larger siblings. It can understand and generate text, making it suitable for applications that require language-based predictions or outputs.

Differences from GPT3/4

1. **Size and Scale**: The most notable difference is in the size and scale of the model. NanoGPT is significantly smaller than GPT3/4. This scaling down means it has fewer parameters, which makes it less resource-intensive and faster to train and deploy.

2. **Resource Efficiency**: Because of its reduced size, NanoGPT can be run on less powerful hardware and requires less data for effective training. This makes it more accessible and practical for applications where computational resources are a constraint.

3. **Application Focus**: While GPT3/4 are designed for a wide range of general-purpose applications, NanoGPT's smaller scale makes it more suitable for specific, targeted applications. This can include specialized tasks in mobile robotics, where the ability to process sequences efficiently is more critical than the broad language understanding capabilities of larger models.

Using Lite Version of NanoGPT Architecture for Trajectory Generation

The NanoGPT architecture has been adapted into a more accessible, lite version suitable for trajectory generation, particularly for a 2DOF ground mobile robot. This section details the implementation of this simplified model, the data preparation process, the training methodology and the results.

Simplification of NanoGPT

- **Architecture Overview**: The lite version of NanoGPT employs a simplified bigram model, adhering to a decoder-only architecture. This modification ensures that the model is lightweight enough to be run on personal computers without compromising its trajectory prediction capabilities.

Implementation and Data Preparation

1. **Trajectory Dataset Generation**:

    - A dataset representing the circular trajectory of a 2DOF ground mobile robot was created. Each point in this trajectory includes two critical pieces of information: the velocity relative to the robot's body frame and the angle of rotation relative to the ground frame.

2. **Encoding Trajectory into Text Format**:

    - To make the data compatible with the LLM, which is inherently text-based, a unique encoding scheme was devised.

    - **Velocity Encoding**: Velocity values range from 0 to 99 and are rounded to two decimal points. These values are encoded into alphabets where '0' is represented by 'a', '1' by 'b', and so forth, up to '9' represented by 'j'.

- **Angle Encoding**: The angle (theta), ranging from -π to π, is similarly rounded to two decimal places. The positive angles are denoted by '@' and negative angles by '$'.
- **Formatted String**: Each trajectory point is encoded into a string starting with '#', followed by four alphabets for velocity, a symbol (@ or $) for angle sign, and three alphabets for the angle. For example, a vector [22.12, -0.24] is encoded as '#ccbc$ace'.

3. **String Concatenation**:

    - After encoding each point, these strings are concatenated to form a single, long string representing the entire trajectory.

Training the Model

1. **Training Data Setup**:

    - The concatenated string forms the training data for the model.
    - During training, the model processes blocks of 'n' consecutive characters. For instance, if 'n' is set to 3, and the string is 'abcdef', the input to the model is 'abc', and it's trained to predict 'bcd' as the output, and so on.

2. **Training Methodology**:

    - The model is trained to predict the next set of 'n' characters based on the current input block. This sequential prediction is critical for generating meaningful trajectory continuations.
    - Loss is calculated based on how accurately the model predicts these subsequent characters.

3. **Model Parameters and Hyperparameters:**

    - The final list of parameters and hyperparameters after hyperparameter tunning is as follows:

Table1

Hyperparameters	**Value**	**Description**
batch_size	32	Number of independent sequences processed in parallel during training.
block_size	8	Maximum context length for predictions.
max_iters	10000	Total number of iterations for training the model.
eval_interval	300	Frequency of model evaluation on training and validation data.
learning_rate	1e-2	Learning rate for the optimizer.

device	'cuda'/'cpu'	Computing device for training ('cuda' for GPU, 'cpu' for CPU).
eval_iters	200	Number of iterations for evaluating the model's performance.

Table2

Parameters	Details	Description
text	Imported data	Raw text data loaded from 'generated_traj.txt', used to create training and validation datasets.
chars	List	List of unique characters in the text.
vocab_size	Integer	Number of unique characters, defining the size of the embedding table.
stoi	Dictionary	Mapping of characters to integers (String to Integer).
itos	Dictionary	Mapping of integers back to characters (Integer to String).
train_data	Tensor	Training dataset split from the original text data.
val_data	Tensor	Validation dataset split from the original text data.
model	Neural Network	Instance of BigramLanguageModel, a simplified language model with a single embedding layer.
optimizer	Optimizer	Instance of **torch.optim.AdamW**, used for optimizing the model parameters.

Trajectory Generation

- **Initial Input and Generation**: After training, the model takes an initial string as input and generates a longer string based on its learned patterns. The length of this generated string can be preset, depending on the requirements of the trajectory.

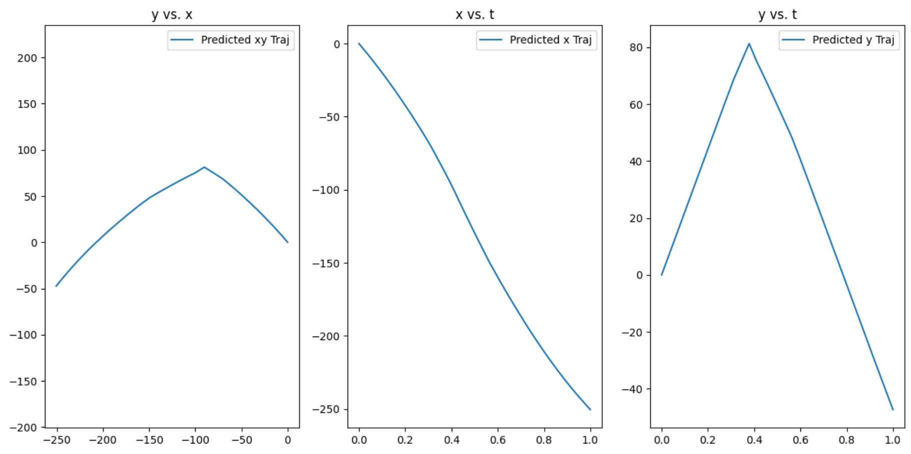
- **Decoding Trajectory**: The trajectory output from the NanoGPT model, initially in a text format, undergoes a decoding process to transform it back into a series of velocity and angle values. Each point in the trajectory is separated by the delimiter '#', allowing for the segmentation of individual vectors. Upon decoding, where alphabets are translated back to numerical values (e.g., 'a' to 0, 'b' to 1), we obtain a sequence of velocity-angle pairs. These pairs are then used to plot the robot's trajectory on an X vs Y plot, visually representing the path that the robot is expected to navigate. This decoding step is essential as it converts the model's text-based output into actionable data for real-world robotic applications.

Results

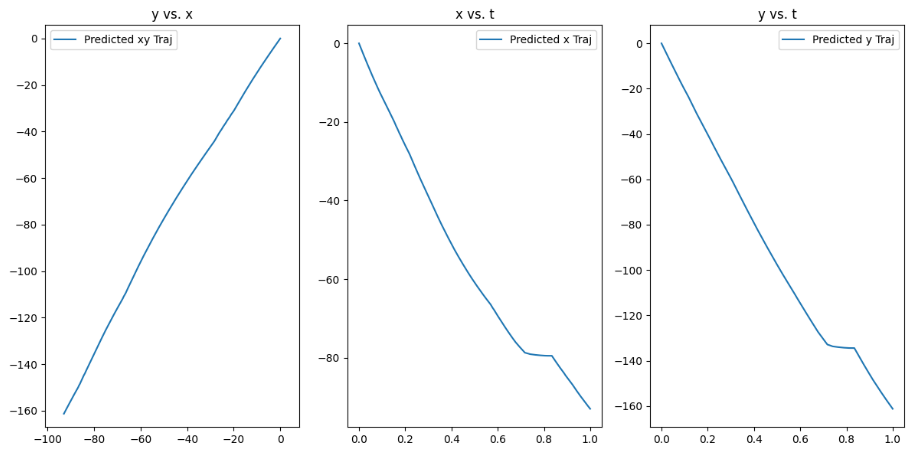
The implementation of the Lite version of NanoGPT in trajectory generation has yielded promising results. The model's application in this domain signifies a successful step towards advanced trajectory planning.

Some Generated Trajectories:

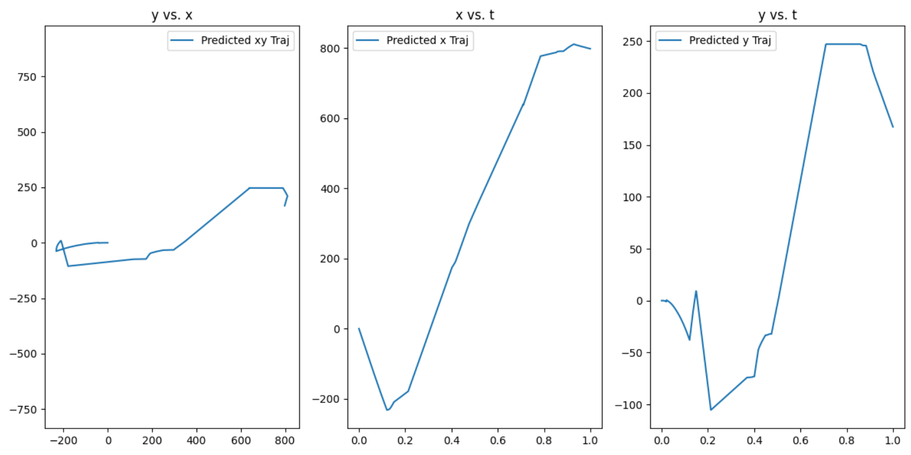
Plot 1



Plot 2



Plot 3



Inferences from the Results

1. **Format Preservation**:

    - The LLM has impressively succeeded in preserving the specific string format required for decoding the trajectory, which was a critical first test. Each set of nine characters consistently adhered to the designated format (e.g., '#aabc@cde'), indicating the model's strong ability to understand and maintain positional relationships within the sequence. This high fidelity in format retention underscores the effectiveness of the positional encoding employed by the model's architecture.

2. **Data Efficiency**:

    - Despite the limited dataset comprising a single circular trajectory with 1000 points, the model has demonstrated remarkable results. The plots show a high degree of consistency and minimal noise, which is commendable given the small amount of training data. This suggests that the model can capture and reproduce the underlying patterns in the data quite effectively.

3. **Trajectory Quality**:

    - The trajectories depicted in the plots, although not forming a perfect circle, show a structured and consistent path. This consistent shape formation, without significant deviations or randomness, implies that the model has learned the essence of the circular trajectory from the limited dataset provided.

4. **Potential with More Data**:

    - The success with a minimal dataset is promising and suggests that, with a larger dataset and increased computational power, the model could potentially generate more complex and accurate trajectories. Given sufficient training time and resources, there's a strong possibility that the model could learn to recreate not only circular trajectories but various other forms as well.

Future Work and Additional Research On Liquid Neural Networks (LNNs)

As we look toward the horizon of robotic innovation, the prospect of employing Liquid Neural Networks (LNNs) emerges as an intriguing addition to the field of trajectory generation for mobile robotics. The distinctive feature of LNNs is their dynamic architecture, which allows for real-time adaptation of network parameters, a feature not typically found in conventional neural networks. This fluid adaptability is akin to the ever-changing connections in the human brain, enabling the LNN to recalibrate its output in response to varying inputs continuously.

The potential for LNNs in trajectory generation is rooted in their efficiency and adaptability. Remarkably, what requires upwards of 100,000 neurons in a traditional neural network to model complex behaviours and patterns could potentially be accomplished with a mere 19 neurons in an LNN, thanks to their dynamic nature. This remarkable reduction in necessary computational units could lead to more efficient processing, lower energy consumption, and the ability to run complex models on less powerful hardware—benefits that are especially valuable in mobile robotics.

Implementing LNNs could dramatically improve a robot's ability to navigate. Instead of relying on static models that may struggle with unpredictable variables, LNNs could enable robots to understand and react to their environment in real-time. Such a network would continually update and refine its model of the world, allowing for the generation of trajectories that adapt on-the-fly to new obstacles, changes in terrain, or shifts in the robot's objectives.

The adaptability of LNNs also suggests a promising avenue for learning from minimal data. Where traditional neural networks might require vast datasets to accurately model an environment or task, the adaptive nature of LNNs could allow them to make more out of less, learning and generalizing from smaller datasets without compromising on performance.

In the broader scope of this report, while models like MiniGPT4 offer substantial benefits in terms of language processing capabilities and trajectory generation, the integration of LNNs could complement and enhance these models. By combining the predictive power of language models with the dynamic adaptability of LNNs, there is potential to create a hybrid system that not only predicts and generates viable paths but also adapts those paths to the real-time complexities of the physical world.

Future work in this area could involve experimental integration of LNNs with existing trajectory generation models, comparative studies to benchmark their performance against traditional networks, and the development of new algorithms to fully harness the dynamic capabilities of LNNs in mobile robotics. This approach could mark a significant step forward in the quest for truly autonomous robotic navigation systems.

GitHub Repository Link

The link to code files to obtain the results discussed above is: https://github.com/anavartpandya/Integrating-Large-Language-Models-for-Enhanced-Trajectory-Generation-in-Mobile-Robotics

The link to the MiniGPT4 architecture code files to pretrain and finetune the model is: https://github.com/Vision-CAIR/MiniGPT-4

References

- **https://minigpt-4.github.io/**
- **https://github.com/karpathy/nanoGPT/blob/master/README.md**
- **https://arxiv.org/pdf/1706.03762.pdf**
- **https://towardsdatascience.com/liquid-neural-networks-in-computer-vision-4a0f718b464e**
- **https://news.mit.edu/2021/machine-learning-adapts-0128**
- **https://github.com/aladdinpersson/Machine-Learning-Collection/blob/master/ML/Pytorch/more_advanced/transformer_from_scratch/transformer_from_scratch.py**
- **https://peterbloem.nl/blog/transformers**
- **https://youtu.be/kCc8FmEb1nY?si=6iDYC_JddLU6EAOu**