GemmaDev: Code debugging using Large Language Model

Nitesh Arya[1], Aryan Deshpande[2], and S. Sathyalakshmi[3]

[1] Hindustan Institute of Technology and Science, Chennai, India.
Department of Computer Science and Engineering
niteshsharya387@gmail.com
[2] Hindustan Institute of Technology and Science, Chennai, India.
Department of Computer Science and Engineering
deshpandearyan1@gmail.com
[3] Hindustan Institute of Technology and Science, Chennai, India.
Department of Computer Science and Engineering
slakshmi@hindustanuniv.ac.in

Abstract. Large Language Models (LLMs) are the advent of current Natural Language processing (NLP) technology with it's strong contextual and complex pattern understanding built using the attention mechanism, embedded into the transformer architecture. Developers, researchers and organizations require a robust, portable and functional code based language model that are efficient enough in tasks involving code debugging and understanding. Existing implementations either have a very high number of model parameters hence increasing the size of the model and decreasing portability, Or less number of parameters, trading off with the accuracy of the model for code generation and debugging. The aim of this paper is to leverage transfer learning from an existing pre-trained model with the best accuracy and tune it to be task specific as well as implement self-debugging abilities for an agent. In this work, we utilize Gemma to fine-tune for code task specific problems and produce a state of the art GemmaDev model. Google's Gemma outperforms all the previous LLMs in various domains including mathematics, and coding. Techniques such as LoRA (Low-Rank Adaptation) and RLHF (Reinforcement Learning with Human Feedback) have shown promising results with high accuracy and portable size of the model balancing the trade off with minimal loss. Utilizing these novel fine-tuning techniques on a strong base model, along with a carefully curated datasets specific to the python programming language, with instruction, input and output to transform Gemma. Instead of training the model all over again, or using a substantial quota of high-performance compute, the project is able to save on a lot of time and resources by applying these methodologies.

Keywords: LLM · Gemma · GemmaDev · LoRA · RLHF.

1 Introduction

Large Language Models (LLMs) are the category of NLP models which have shown superior performance over earlier models built using RNN, LSTM ar-

chitectures[10]. Impeccable performance of LLMs with alignment of output with reinforcement learning (RL) has resulted language models to produce human-like responses, translate from one natural language to another, question answering, text summarization etc. Transformers[12] play a pivotal role in giving rise to lan-

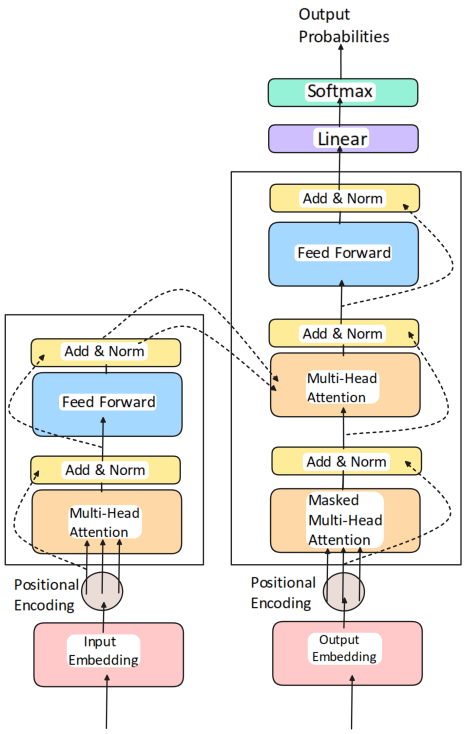

Fig. 1. Transformer

guage models like GPT [16], BERT [3], CodeGen [9], Gemma [11] etc. From the positional encoding layer, through the Multi-Head Attention modules, and till the terminal feed forward layer as shown in Fig 1, each component plays a crucial role in capturing contextual understanding in long and complex statements. Transformer's [12] capacity to create a probabilistic model is instrumental in capturing the dependency of word in the upcoming word prediction. Attention mechanism in fig 2, highlights the component that captures this dependency and probabilities. In the downstream tasks like code generation, this ability combined with syntactic understanding of particular programming language results in error free code generation. Furthermore, this advantage can be leveraged for code understanding to make the programs bug free, hence refining the generated code as per the required semantics and producing executable code.

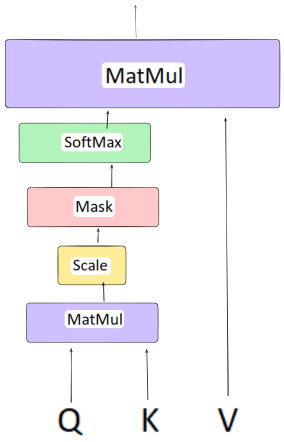

Fig. 2. Attention

One of the major challenges involved with LLMs in code specific task involves LLM hallucination. It is a form of impertinent information that is generated which holds no value to task completion or user input. One of the main reasons why this phenomenon is observed, is due to the output generated solely from the LLMs and not from a database or a result of a search engine. The Output produced is inferenced from the model training which involves generation based on prompt input provided and it's correlated probabilities. This can be useful to have a certain degree of entropy and randomness, so that outputs can be a little creative rather than being a replica version of the training dataset. In this work, LLM hallucination is tackled by fine-tuning over a diversified dataset to improve correct syntactic and semantic code generation through input, instruction and output feature sets.

In recent times, there has been a rise in computational power, leading to an increased number of model parameters in LLMs, as a result enlarging the size of model. This increased number of parameters helps to capture pattern and dependency of a word represented as a weight matrix which affects the subsequent words. conventional fine-tuning approaches further increases the size of model with it's update weight matrix. In our work we have improvised over Gemma by using LoRA adapters which results in minimal change to the model size.

Through our work efficacy of fine-tuning methods like LoRA and QLoRA adopted in creation of GemmaDev from the pre-trained version of Gemma. Gemma [11] is a family of light-weight, state-of-the-art open-source model performing better than rest of the open-source LLMs, specializing in different tasks involving reasoning, mathematics and natural language generation. Performance achieved by Gemma in task related to mathematics and logic can be leveraged in down stream task of code generation and code understanding upon fine-tuning. In our work, we fine-tune Gemma-2B model on various text-code datasets.

2 Related Works

2.1 Language Modeling

Language models have seen an increase in popularity due to their ability to possess large token lengths and context sizes. The introduction of large language models has led to enhancements in natural language understanding, generating better probability distributions over long sequence of words than their predecessors LSTMs[10]. While demonstrating remarkable ability to follow natural language instructions, they exhibit a constrained understanding of code execution [1]. This has led to a problem of hallucination and error-prone code generation. Large language models trained over code yield excellent results in code generation [8, 9]. In spite of good accuracy, these models lack a concrete method for interpretation of the code, thus leading to a limited ability to decompose the program into sub-tasks and generate intermediate results. Intermediate result generation in [5] has been proven to be successful in enhancing accuracy, a critical factor in increasing the likelihood of successful code execution.

2.2 Prompt Engineering

Previous techniques in prompt engineering, e.g., few-shot prompting [7], which involves presenting a fixed set of labeled examples directly within the prompt. This approach enhances the model's comprehension of contextual nuances essential for creating accurate responses. In addition, another prompting technique, the Chain of Thought [14] facilitates intricate reasoning through intermediate reasoning steps. This method encourages the LLM to carefully consider the task at hand by establishing a continuous dialogue of natural thought processes. While Chain of Thought demonstrates performance gains exclusively when applied to models with 100B parameters or more. This method [5] has emerged as a superior prompt engineering technique, as evident by the comparison of results against established benchmarks. Chain of Code breaks down problems into sub-tasks and tracks intermediate output, which aids in the generation of output that is executed. LLMs generate code and simulate the interpreter by producing the anticipated output of specific lines of code. The steps include prompting LLMs to structure semantic sub-tasks at runtime to emulate within the LLM [5]. It can be effectively utilized with GemmaDev as a prompting technique and can demonstrate promising results.

2.3 Quantization and Optimization of language models

Parameter Efficient Fine-tuning (PEFT) [13, 15] offers an efficient method for tuning large language models for downstream tasks. It presents a better alternative to conventional fine-tuning methods[17]. Accompanied by the rise in parameter count of large language models reaching billions, fine-tuning the entire model has become inefficient computationally and very expensive, often impractical for normal users.

PEFT addresses this issue by fine-tuning only a subset of the model's parameters. Therefore, this significantly reduces the need for advanced computational requirements. [13] explores the critical need for parameter-efficient tuning in the context of large language models (LLMs). [13] Through empirical analysis, it demonstrates the effectiveness of PEFT. Providing compelling reasons for our project to adopt this technique and create a code-task-driven model by leveraging Gemma's [11] large token size, contextual length, and mathematical capabilities, fine-tuned on code-task data.

3 Methodology

Pre-training of generic LLMs involves training on a large corpus of data, ranging from user session data, web-pages, code-bases, e-books, Wikipedia data, and many more. Whereas Code-LLMs are pre-trained language models which are trained for the purpose of code synthesis. Pre-trained Code-LLMs are based on code bases such as git issues, git commit, stack overflow data and various other sources involving code examples to teach model pertaining to syntactic understanding. Code LLMs are mainly utilized in software development related tasks such as code synthesis and code summarization. Code debugging is a recurring process in the software development cycle with the goals to improvise and make the program bug free.

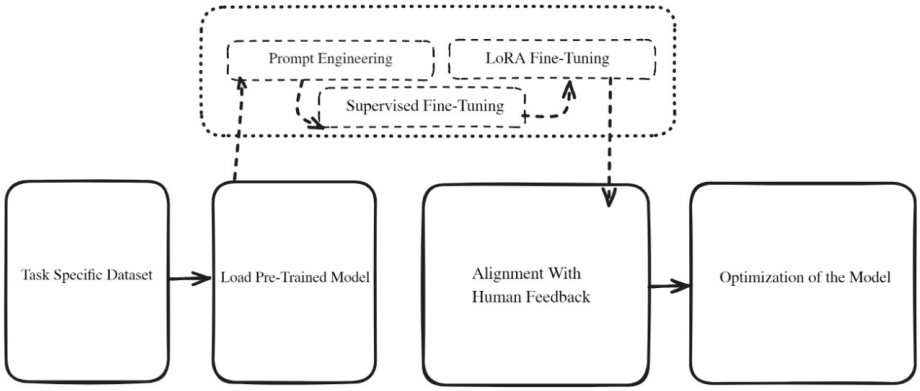

Fig. 3. End-to-End pipeline

The paper proposes to leverage the contextual understanding ability of LLMs that are fine-tuned to specific tasks, such that it aids in the error debugging process. Since the 'context window' of the base model shifts from being trained on general data encompassing everything, to code-task specific. The 'context window' converges towards distinct tasks, mitigating the impact of 'hallucination', reducing its occurrences throughout the responses generated by the model.

Initially the paper employs an instruction-tuning technique or Supervised fine-tuning (SFT). SFT is a method to align language models on labeled data. Consequently, it then employs two fine-tuning methods, Low-Rank Adaptation [4] (LoRA) and Quantized Low-Rank Adaptation [2] (QLoRA) which performs parameter efficient tuning on specific trainable parameters and adds them as an adapter to the model. Lastly, AI alignment technique namely Reinforcement Learning with Human Feedback [6] (RLHF), ensures the structure of output generated is aligned along with the human expected output for better understanding. End-to-end pipeline starting from selecting dataset and generating an optimized model is depicted in fig 3.

3.1 Low Rank Adaptation

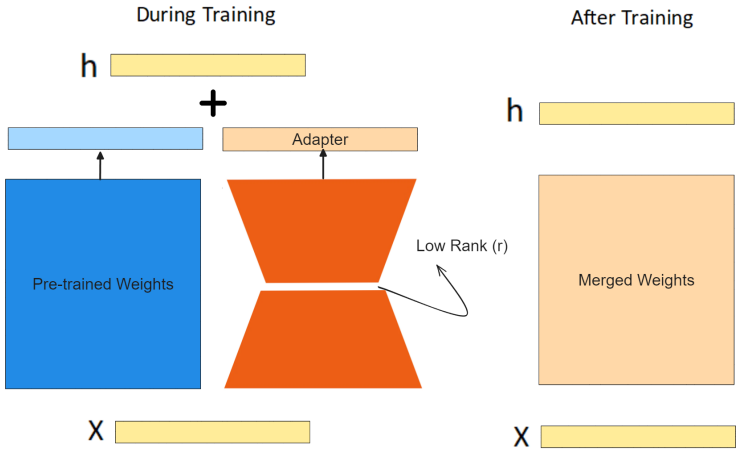

Fig. 4. LoRA

Low Rank Adaption (LoRA) is an adapter method of fine-tuning where certain parameters from the whole weight matrix is trained and then added back to the weight matrix. LoRA approximates the parameters from the weight matrix using rank structure, where rank of the weight matrix is evaluated which is used for fine-tuning [4]. The rank of the weight matrix is used keeping in mind that rank is the minimal required columns of weight matrix which is factor of whole matrix.

$$W_{\text{updated}} = W + \Delta W \tag{1}$$

Proposed approach is to transform Gemma into GemmaDev, which stands out from other fine-tuning methods as it consumes less memory and trains specific parameter compared to conventional fine-tuning approaches [17]. Here low rank

parameters are trained and updated rather than training all the parameters and updating every weight during the back-propagation phase, which is not time and memory efficient. In the above equation (1), ΔW are the trained low rank weights added as an adapter to the weight matrix.

Parameter-Efficient Fine-Tuning [13, 15] is a method for effectively adapting large pre-trained models to various applications without tuning all the model parameters. This subsequently decreases the storage and computational costs. At best it yields performance that is comparable to the fully fine-tuned model. Moreover, it makes the models easy to train and store on consumer hardware. Once the low rank weights (trainable parameters) is obtained PEFT is used to efficiently fine-tune these trainable parameters as per the task required at hand. Fig 4, highlights the merged adapter to weight matrix. Thus LoRA with PEFT produces the training which is cost and time effecient.

3.2 Quantized Low Rank Adaptation

Quantization is an approach in making models memory efficient where the precision of model is changed to a lower precision while balancing the tradeoff with the accuracy. Here the model precision is changed to a lower precision such as changing from float-point precision to integers or fixed-point precision. QLoRA combines this two techniques i.e Quantization and LoRA to not only make training efficient but also reducing model precision, so that the size of model is reduced thus making it a better option to inference in a memory constrained environment. It proposes to Quantize the pretrained model in a 4-bit normal precision, then use the adapters to fine-tune the model. QLoRA reduces the average memory requirements of fine-tuning a 65B parameter model from >780GB of GPU memory to <48GB without degrading the runtime or predictive performance compared to a 16-bit fully fine-tuned baseline model performance. BitsAndBytes library is used to load the model in a 4-bit (4b) version using normal float-4 (NF4) quantization and double quantization for quick training. In this project, we utilize Gemma [11] with 2 billion parameters, while the model is loaded in its 4 bit precision mode.

3.3 Reinforcement Learning for Human feedback

A Large Language model that is fine-tuned upon a large amount of dataset is not enough as it tends to hallucinate. The term hallucination comes from the LLM generating a response that is false. AI alignment with RLHF [6] allows the language model to understand what/how answers are supposed to be generated. Alignment is the method where models learn to generate output which aligns with the user's input and context, while producing accurate output. Training with RL makes model responsive to user input and context.

3.4 Debugging Agent - python interpreter integration

This study presents an innovative approach to automating code repair, incorporating a python interpreter. A self-debugging agent is developed which not

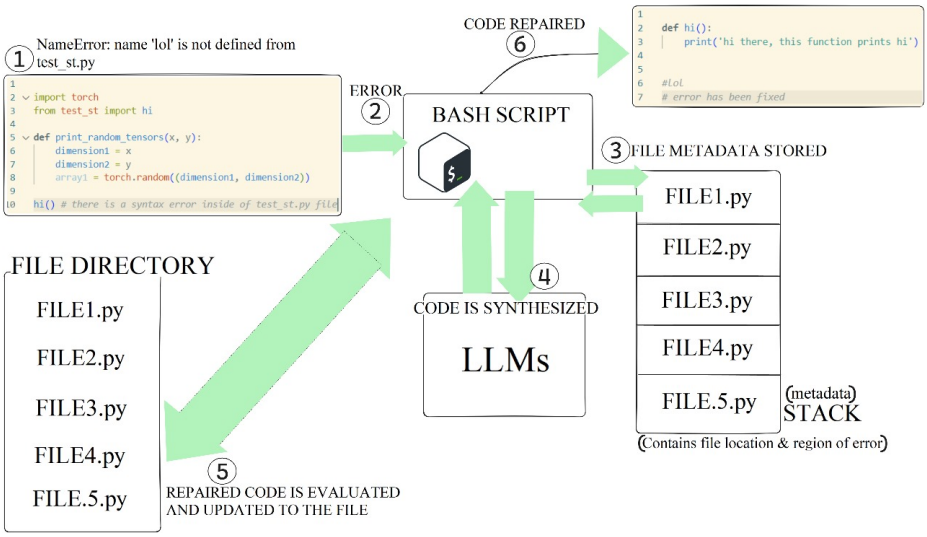

Fig. 5. Agent Integration

only identifies errors in multiple files within a folder context, but also repairs them, leveraging the capabilities of large language models (LLMs) fine-tuned on high quality datasets for the python programming language. By analyzing error traces and utilizing the python interpreter to detect further errors, the agent can rectify code issues, enabling efficient and accurate code synthesis. OLLAMA integrates fine-tuned large language models (LLMs), including LoRA/QLoRA models or SOTA models like LLaMA3, enabling interaction with self-debugging automation and user to LLM chat functionality.

In fig 5, the self-debugging process starts with step 1: running the current python file (File1). In step 2, the agent captures the error trace, which is stored in a stack data structure. Each element in the stack represents a file in the directory, containing metadata such as file path and line of error. The agent then leverages a fine-tuned Large Language Model GemmaDev or native models from Meta/Google to facilitate code synthesis in step 4. The synthesized code is then written to its designated file in the current directory. In step 5, the finalized code is verified and deemed whether it's capable of execution, using the python interpreter. If there are errors, the LLM synthesizes the code again, and the agent repeats this process until the stack is empty, indicating that all errors have been addressed.

The agent successfully demonstrated its ability to repair various types of code errors in python, including syntax errors, runtime errors (exceptions), logic errors, semantic errors, indentation errors, import errors, attribute errors, key-error, and type errors. To evaluate its performance, 10 debugging cases with diverse error scenarios were created, featuring random problem sets, distinct functionalities within the code file, and different layers of file imports. Notably,

the agent achieved a remarkable success rate of 9 out of 10 cases, showcasing its proficiency in it's self debugging capability.

4 Datasets

Quality of datasets are directly proportional to the model's performance, thus datasets used in this paper is an Instruction dataset which contains input, output, and instruction which gives direction to generate specific results tailored for the given task.

4.1 18K Alpaca Dataset

This dataset is specific to python programming language and the dataset features consist of input, instruction, output and prompt. Each row of dataset is transformed into a string template which will be forwarded to the model while fine-tuning. The sample dataset row is given in fig 6

INSTRUCTION	INPUT	OUTPUT	PROMPT
Create a function to calculate the sum of a sequence of integers.	[1, 2, 3, 4, 5]	# Python code def sum_sequence(sequence): sum = 0 for num in sequence: sum += num return sum	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: Create a function to calculate the sum of a sequence of integers. ### Input: [1, 2, 3, 4, 5] ### Output: # Python code def sum_sequence(sequence): sum = 0 for num in sequence: sum += num return sum

Fig. 6. Alpaca Dataset

4.2 Codeparrot - xlcost-text-to-code

Codeparrot is a cross-domain dataset consisting of different programming languages such as Java, Go, Ruby, python, PHP etc. It consist of two columns text, code corresponding to the particular text. Text is passed as a question and output is used as label for supervised fine-tuning. The Sample dataset row is given in fig 7

TEXT	CODE
Create a function to calculatMaximum Prefix Sum possible by merging two given arrays | C ++ Program to implement the above approach ; Stores the maximum prefix sum of the array A [ ] ; Traverse the array A [ ] ; Stores the maximum prefix sum of the array B [ ] ; Traverse the array B [ ] ; Driver codee the sum of a sequence of integers.	#include <bits/stdc++.h> NEW_LINE using namespace std ; int maxPresum (vector < int > a , vector < int > b) { int X = max (a [0] , 0) ; for (int i = 1 ; i < a . size ( ) ; i ++) { a [i] += a [i - 1] ; X = max (X , a [i]) ; } int Y = max (b [0] , 0) ; for (int i = 1 ; i < b . size ( ) ; i ++) { b [i] += b [i - 1] ; Y = max (Y , b [i]) ; } return X + Y ; } int main ( ) { vector < int > A = { 2 , -1 , 4 , -5 } ; vector < int > B = { 4 , -3 , 12 , 4 , -3 } ; cout << maxPresum (A , B) << endl ; }

Fig. 7. Sample Codeparrot

4.3 Evol-Instruct-Code-80k-v1

This python dataset is similar to the earlier datasets with the increased number of rows and training samples, consisting of 80k training rows. Sample dataset row is given in 8

INSTRUCTION	OUTPUT
Create a nested loop to print every combination of numbers between 0-9, excluding any combination that contains the number 5. Additionally, exclude any combination that contains a repeating digit. Implement the solution without using any built-in functions or libraries to check for repeating digits.	Here is an example of a nested loop in Python to print every combination of numbers between 0-9, excluding any combination that contains the number 5 or repeating digits: ```python for i in range(10): # First digit for j in range(10): # Second digit for k in range(10): # Third digit # Checking for the conditions if i != 5 and j != 5 and k != 5 and i != j and i != k and j != k: print(i, j, k) ``` This code will generate and print every combination of three digits between 0-9 that do not contain the number 5 and do not have any repeating digits.

Fig. 8. Sample Evol-Instruct-Code-80k-v1

5 Results and analysis

The cornerstone of our fine-tuning of the Gemma 2B model on 3 code example datasets unequivocally demonstrated the model's swift inference capabilities, while being accurate for the size of the model. The test was conducted on NVIDIA's T4 GPU (16GB of V-RAM). Fine-tuned model exhibited commendable performance, further attesting its reliability and adaptability in handling different types of code examples. The choice of hardware is crucial in understanding the practical implications of our results. It becomes evident that a model with a large number of parameters was able to run seamlessly using LoRA and QLoRA fine-tuning provided computational constraints. GemmaDev was obtained by fine-tuning Gemma 2B on 16GB of V-RAM, the decrease in

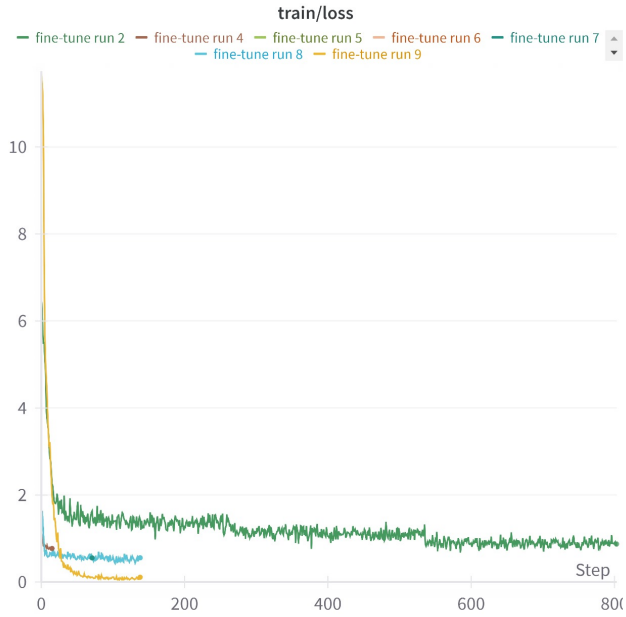

Fig. 9. Loss vs Steps

loss value compared to training steps is evident in fig 9. Convergence of learning rate alongside decrease in loss value is highlighted in fig 10.

This study highlights the significant potential of integrating Large Language Models (LLMs) with python interpreters in automating code execution and self-debugging. The developed agent demonstrates impressive capabilities in repairing diverse code errors, achieving a high success rate. Showcasing far-reaching implications, poised to transform the code debugging and repair processes across multiple programming languages.

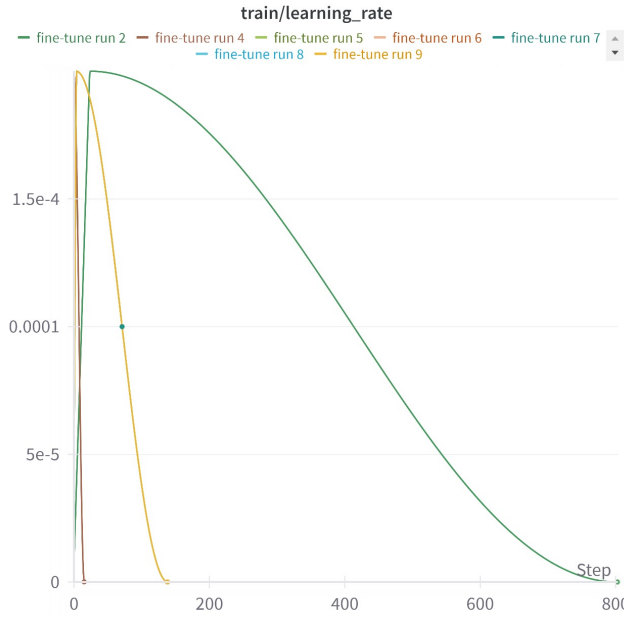

Fig. 10. Learning rate vs Steps

5.1 Benchmark

GemmaDev model underwent testing on the Human-Eval, a state of the art benchmarking tool used by OpenAI, Google, etc. Human-Eval contains a total of hand-crafted dataset comprising 164 programming challenges. The responses generated by the model were composed into a eval.jsonl. This file contains generated samples that will be evaluated, utilizing the pass@k metric to gauge performance. The evaluation focuses on models that generate single solution, designated by pass@1. Our model demonstrates performance with a pass@1 score of 0.06707317073170732.

6 Conclusion and Future work

As we conclude our investigation, we recognize the potential for further enhancements and extensions to our agent's capabilities. Future advent for our project would enable the agent to traverse into multiple levels of folder context, so that it can be used for complex application development. In addition, implementing multi-instance capability of the agent for parallel self-debugging process and execution. This will significantly benefit users by streamlining development and debugging workflows.

Another avenue for future work involves the development of a Visual Studio Code extension tailored to enhance developer's convenience. Such an extension

would not only streamline the integration of our model into existing development workflows but also significantly reduce startup costs. By providing a seamless interface within a widely-used Integrated Development Environment (IDE) like Visual Studio Code, developers can effortlessly leverage the power of our model without the need for extensive setup or configuration through the cloud or locally. This not only saves development time but also eliminates the arduous process of searching for errors manually.

References

1. Chen, A., Scheurer, J., Korbak, T., Campos, J.A., Chan, J.S., Bowman, S.R., Cho, K., Perez, E.: Improving code generation by training with natural language feedback. arXiv preprint arXiv:2303.16749 (2023)
2. Dettmers, T., Pagnoni, A., Holtzman, A., Zettlemoyer, L.: Qlora: Efficient finetuning of quantized llms. Advances in Neural Information Processing Systems **36** (2024)
3. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. CoRR **abs/1810.04805** (2018), http://arxiv.org/abs/1810.04805
4. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021)
5. Li, C., Liang, J., Zeng, A., Chen, X., Hausman, K., Sadigh, D., Levine, S., Fei-Fei, L., Xia, F., Ichter, B.: Chain of code: Reasoning with a language model-augmented code emulator. arXiv preprint arXiv:2312.04474 (2023)
6. Li, Z., Yang, Z., Wang, M.: Reinforcement learning with human feedback: Learning dynamic choices via pessimism. arXiv preprint arXiv:2305.18438 (2023)
7. Liu, H., Tam, D., Muqeeth, M., Mohta, J., Huang, T., Bansal, M., Raffel, C.A.: Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. Advances in Neural Information Processing Systems **35**, 1950–1965 (2022)
8. Mastropaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., Bavota, G.: On the robustness of code generation techniques: An empirical study on github copilot. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 2149–2160. IEEE (2023)
9. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022)
10. Sherstinsky, A.: Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. Physica D: Nonlinear Phenomena **404**, 132306 (Mar 2020). https://doi.org/10.1016/j.physd.2019.132306, http://dx.doi.org/10.1016/j.physd.2019.132306
11. Team, G., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M.S., Love, J., et al.: Gemma: Open models based on gemini research and technology. arXiv preprint arXiv:2403.08295 (2024)
12. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. CoRR **abs/1706.03762** (2017), http://arxiv.org/abs/1706.03762

13. Wang, C., Yan, J., Zhang, W., Huang, J.: Towards better parameter-efficient fine-tuning for large language models: A position paper. arXiv preprint arXiv:2311.13126 (2023)
14. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al.: Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems **35**, 24824–24837 (2022)
15. Xu, L., Xie, H., Qin, S.Z.J., Tao, X., Wang, F.L.: Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment. arXiv preprint arXiv:2312.12148 (2023)
16. Yenduri, G., Ramalingam, M., Chemmalar Selvi, G., Supriya, Y., Srivastava, G., Maddikunta, P.K.R., Deepti Raj, G., Jhaveri, R.H., Prabadevi, B., Wang, W., et al.: Gpt (generative pre-trained transformer)–a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. arXiv preprint arXiv:2305.10435 (2022)
17. Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., He, Q.: A comprehensive survey on transfer learning. CoRR **abs/1911.02685** (2019), http://arxiv.org/abs/1911.02685