Creating a Compile Debug Tool for PyTorch 2.0

Patrick Yeh
Department of Computer Science
Columbia University
New York, USA
psy2107@columbia.edu

Zhengfei Gong
Department of Computer Science
Columbia University
New York, USA
zg2469@columbia.edu

Haiyu Wei
Department of Electrical Engineering
Columbia University
New York, USA
hw3036@columbia.edu

***Abstract*—Understanding PyTorch 2.0 graph breaks remains a significant challenge, particularly for developers new to the framework. Error messages associated with graph compilation failures often lack clarity and actionable guidance, hindering the productivity of developers. This paper introduces a debugging tool that leverages PyTorch's torch.compile and torch._dynamo.explain() functionalities. The tool processes these explanations, uses an Large Language Model (LLM) to provide user-friendly feedback, and offers actionable suggestions for resolving graph breaks. Additionally, we analyze graph breaks across various Hugging Face models to construct a taxonomy that categorizes the types and causes of graph breaks. The proposed tool aims to demystify graph compilation errors, reduce debugging time, and enhance developer accessibility to PyTorch's high-performance features.**

***Index Terms*—PyTorch 2.0, Graph breaks, Debugging tools, Machine learning optimization, Developer productivity, Error handling, Neural network compilation, Large language models (LLMs), Model performance, Taxonomy of errors.**

I. Introduction

Modern machine learning frameworks such as PyTorch [18] have transformed the landscape of deep learning by providing tools that simplify model development and optimization. This has helped raise the accessibility and performance needed to succeed in the field of machine learning, enabling much of the progress that we have made so far. [3] With the release of PyTorch 2.0, the introduction of *torch.compile* has enabled significant advancements in graph compilation and performance optimization. [8] [14] However, these benefits come with new challenges, particularly for developers encountering graph breaks, which are situations where the compilation process is interrupted due to generic jump, unsupported operations and other issues. [16]

Graph breaks disrupt workflows, present opaque and uninformative error messages, and leave developers with limited guidance. Debugging these issues often requires expert knowledge of PyTorch's internal implementation and can consume considerable development time. This can create a significant barrier for users, especially those new to PyTorch, and prevents them from leveraging the framework's high-performance capabilities. [17]

Graph breaks are a frequent and frustrating occurrence in PyTorch 2.0 workflows, and while *torch._dynamo.explain()* provides detailed explanations, the information is often verbose, technical, and inaccessible to many users. [19] This lack of actionable, user-friendly debugging capabilities hinders productivity and the adoption of PyTorch's new features. Right now, the current PyTorch ecosystem lacks structured documentation or tools to systematically address and resolve these graph break issues.

As a result, the objective of this project is to introduce a debugging tool that simplifies the process of understanding and resolving graph breaks in PyTorch 2.0. By parsing the output from *torch._dynamo.explain()* and other related features, the tool integrates a large language model (LLM) to provide clear, actionable explanations tailored to users of varying expertise levels. Additionally, to aid development of this tool and also to help create extensive documentation of an otherwise underexplored topic, we attempted to categorize graph breaks into a taxonomy, with an aim to provide a structured frame of reference for common issues and their solutions. Thus, the scope of or project involved 1) analyzing graph breaks from a diverse set of Hugging Face models to identify patterns and recurring issues, 2) integrating prompt-engineered LLMs to interpret and explain error messages, and 3) evaluating the tool's effectiveness in reducing debugging time and enhancing user experience.

The rest of this paper is organized as follows: Section 2 reviews related work and the current state of graph break debugging tools for PyTorch. Section 3 describes the design and implementation of our debugging tool. Section 4 presents an experimental evaluation of its effectiveness. Section 5 discusses challenges, limitations, and future directions. Finally, Section 6 concludes our paper and also touches on the broader implications of our work.

II. Literature Review

Research on debugging tools for machine learning frameworks has grown significantly in recent years. PyTorch 2.0's *torch.compile* represents a critical step forward in enabling efficient graph compilation, but its debugging capabilities remain relatively unexplored. Previous works have highlighted the challenges posed by dynamic computation graphs in frameworks like PyTorch, TensorFlow, and JAX, emphasizing the need for robust error-handling mechanisms. [14]

Despite the advancements in debugging tools and error explanation systems, the following gaps persist:

- *Limited Coverage of Graph Compilation Errors*: Existing tools and studies do not specifically address the unique challenges posed by graph breaks in PyTorch 2.0.

- *Lack of Taxonomies for Graph Breaks*: There is currently no structured official documentation categorizing common graph break types and their underlying causes.
- *Lack of Integrated LLM Solutions*: While LLMs have been applied to general debugging tasks, their potential in simplifying graph compilation errors has not yet been fully explored.

This paper aims to address these gaps by introducing a comprehensive debugging tool that develops an extensive taxonomy and leverages it into a set of LLM-powered explanations tailored for PyTorch 2.0 users. What follows in this section is a survey of current graph break debugging tools and capabilities.

A. Dynamic Graph Debugging

Studies have addressed debugging in dynamic computation graphs, focusing on techniques to trace errors and optimize computational workflows. These approaches often emphasize the need for effective tools to help developers understand and fix issues arising during the compilation and execution of these graphs. Frameworks like TensorFlow's XLA and JAX's tracing tools provide some insights into error patterns, enabling developers to analyze and refine their computational pipelines. [14] However, these tools often fall short in offering user-friendly solutions for non-expert developers, who may lack deep familiarity with underlying compilation mechanics or debugging methodologies.

For instance, XLA (Accelerated Linear Algebra) [13] is a compiler utilized by both TensorFlow and JAX to optimize machine learning models for performance and scalability. XLA debugging tools, such as those described in this resource, offer functionality to trace and diagnose compilation errors, but these tools are often intricate and require a steep learning curve. Moreover, JAX, which is designed with XLA compilation in mind, tends to focus on programs specifically written to account for XLA-related challenges. [11] This means that developers working with JAX are often expected to incorporate XLA-related considerations, such as shape polymorphism and compilation scope, directly into their programming practices.

Despite the availability of these debugging aids, the lack of intuitive, high-level interfaces remains a significant barrier, especially for developers who are less experienced with the nuances of compilers and dynamic graph execution. Bridging this gap calls for the development of more accessible debugging solutions that can abstract away the complexities of underlying frameworks while offering actionable insights and suggestions to streamline the debugging process.

B. Error Explanation Systems

Research on automated error explanation systems demonstrates significant potential in simplifying the debugging process and reducing the associated complexity. Tools like Explainable AI (XAI) and ExplainLikeI'mFive (ELI5) have gained attention for their ability to present technical feedback in a clear and accessible manner, often demystifying complex processes for a broader audience. [10] [1] These tools aim to bridge the gap between technical intricacies and user comprehension, offering explanations that are both insightful and user-friendly.

Despite their success in other areas, applying these methodologies to the domain of graph compilation errors presents unique challenges. Graph compilation errors, often encountered in frameworks such as PyTorch, TensorFlow, and JAX, involve intricate issues that are deeply rooted in the mechanics of dynamic computation and optimization. Unlike high-level model behavior explanations, which focus on outcomes and predictions, addressing compilation errors requires a deep understanding of execution paths, graph structures, and low-level system interactions. [14]

Current tools excel at providing interpretable insights into broader machine learning workflows but often lack the specificity required to tackle the nuanced nature of graph compilation issues. Effective solutions in this area would need to not only simplify the explanation of errors but also offer targeted insights into the underlying causes while suggesting actionable strategies to resolve them. This adaptation demands a careful balance of technical accuracy and user accessibility, enabling both novice and experienced developers to efficiently identify and address the root causes of errors.

The limited application of XAI and ELI5 principles to graph compilation debugging underscores a significant opportunity for advancement. By leveraging the interpretability and user-centric design of these tools in a way that accounts for the complexity of dynamic computation graphs, researchers can create solutions that transform the debugging experience, making it more intuitive and effective for developers working with modern machine learning frameworks.

C. Large Language Models for Debugging

The integration of large language models (LLMs) into debugging workflows has gained significant traction in recent years. Notably, LLMs have demonstrated effectiveness in generating human-readable explanations for programming errors, particularly in languages like Python and JavaScript. [20] This approach builds on the strengths of LLMs in understanding code semantics, generating explanations, and offering suggestions for refinement. However, applying these models to interpret and debug errors in machine learning frameworks, especially those tied to graph compilation, remains an unexplored area of research.

Zhong, Wang, and Shang introduce a novel approach to enhancing debugging with LLMs by emulating the systematic, step-by-step methods used by human developers. [20] Traditional LLM-based debugging treats the generated programs as monolithic entities, which is often inadequate for handling complex logic flows and intricate data operations. In contrast, their proposed framework, Large Language Model Debugger (LDB), segments programs into manageable basic blocks and uses runtime execution information to validate each block against the task description. By tracking intermediate variables and runtime states, LDB enables LLMs to isolate errors more effectively and refine the program iteratively.

This methodology shows promise for addressing challenges specific to debugging graph compilation errors in machine learning frameworks. Errors in these contexts often stem from intricate execution paths, tensor mismatches, or issues arising from dynamic computation graphs. Adapting the principles of LDB to these frameworks could allow for the systematic examination of execution steps within graph compilation, enabling the LLM to provide granular insights into error sources. For instance, by segmenting the graph execution process into smaller computational units, an LLM could verify each unit's correctness in isolation, identify discrepancies in tensor shapes, or diagnose operation compatibility issues.

Moreover, LDB's ability to work with runtime execution information aligns closely with the needs of debugging dynamic computation graphs, where runtime behavior often reveals insights that static analysis cannot capture. This iterative verification and refinement process could significantly enhance the ability of LLMs to assist developers working with frameworks like PyTorch and TensorFlow, where errors frequently emerge from the interplay between static graph definitions and runtime data.

The demonstrated performance gains in traditional code debugging benchmarks, as achieved by LDB, highlight its potential to improve debugging efficiency across domains. Extending this approach to machine learning frameworks could unlock new opportunities to reduce the complexity of identifying and resolving graph compilation errors, providing developers with accessible and actionable feedback tailored to the unique challenges of dynamic computation graphs.

D. Torch Explain vs. Logging

Another aspect of PyTorch we looked into was the difference between the Explain and Logging capabilities. Whereas the Explain features conduct a static analysis of our tools and graph breaks, Logging instead does so dynamically. This difference can lead to different behaviors and outputs on graph breaks, and thus we made it a point to look into and respect the two philosophies.

E. Prompt Engineering

Prompt engineering and fine-tuning [15] [12] represent two complementary approaches to optimizing large language models (LLMs) for specific tasks. Prompt engineering involves designing carefully structured inputs to elicit desired outputs from pre-trained models, often using techniques like few-shot and zero-shot learning. This approach is accessible and cost-effective, allowing users to quickly adapt LLMs to new applications without requiring additional training. However, the effectiveness of prompt engineering can vary, as it depends heavily on the phrasing and context of the input. [7] [6] Small changes in wording can lead to significant differences in performance, making this method inherently iterative and reliant on trial and error. [5] Thus, this remained an interesting technique to utilize in our work.

Fine-tuning, in contrast, involves training a pre-existing model on a domain-specific dataset to imbue it with specialized knowledge. This method provides more consistent and robust results, particularly in complex or high-stakes applications like legal analysis or medical text interpretation. However, fine-tuning is resource-intensive, requiring substantial computational power and annotated data. It also carries the risk of overfitting if the training data lacks diversity, which can limit the model's generalizability. While prompt engineering excels in rapid prototyping and low-resource contexts, fine-tuning is ideal for scenarios demanding high accuracy and domain-specific expertise. The interplay between these techniques continues to evolve, with hybrid approaches combining the strengths of both to maximize efficiency and performance in deploying LLMs.

III. Methodology

This section outlines the systematic approach used to design and implement the proposed debugging tool. It includes details on data collection and preprocessing, selection of models for analysis, optimization procedures for debugging workflows, profiling methods for performance evaluation, and metrics used to assess the tool's effectiveness. This structured framework ensures a comprehensive and reproducible process for addressing graph breaks in PyTorch 2.0.

A. Data Collection and Preprocessing

To create a robust debugging tool, we first collected data on graph breaks using PyTorch's *torch._dynamo.explain()* function. This process began by running a diverse set of machine learning models from the Hugging Face repository. These models were deliberately chosen to cover a wide spectrum of architectures and use cases, including computer vision, natural language processing, and generative tasks. The variety ensured that our dataset captured a comprehensive range of scenarios and potential issues encountered during graph compilation. For each model, graph breaks were logged in detail, capturing relevant metadata such as the specific operation causing the break, its location within the computation graph, and the fallback mechanisms invoked.

Once the data was collected, preprocessing was conducted to make it more accessible and actionable. This involved parsing the output of *torch._dynamo.explain()* to extract meaningful information. Key details, such as error messages, fallback summaries, and unsupported operations, were isolated and structured for analysis. To enhance the usability of the data, redundant or irrelevant outputs were filtered out, focusing only on elements that contributed to understanding the nature and causes of the graph breaks.

This structured data enabled us to identify recurring patterns in graph break behavior, revealing common pitfalls and failure modes in various models. Using these insights, we developed a taxonomy of graph break types, categorizing them based on their root causes, such as unsupported tensor operations, shape mismatches, or dynamic control flows. This taxonomy not only provided a systematic understanding of the issues but also served as a foundation for building heuristics and crafting targeted debugging solutions. By leveraging this approach, we ensured that the debugging tool was informed by real-world scenarios and capable of addressing the most prevalent challenges encountered in practice.

B. Model Selection

We selected models from the Hugging Face repository to ensure a comprehensive representation of diverse use cases across different machine learning domains. This diversity was essential for capturing a wide variety of graph break scenarios, reflecting the real-world challenges faced by developers working with dynamic computation graphs. The models tested included:

- *Computer Vision Models* (ResNet, Vision Transformer (ViTs)): These models were chosen for their widespread use in tasks such as image classification, object detection, and segmentation, which often involve complex tensor operations and unique computational requirements.
- *Natural Language Processing Models* (BERT, GPT-2, T5): These models, widely employed for tasks like text classification, translation, and question answering, provided insights into graph breaks associated with sequence processing and transformer-based architectures.
- *Generative Models* (Stable Diffusion, DALL-E, MusicGen): Generative models were included for their computational complexity and dynamic behavior, often involving intricate operations such as latent space transformations, upsampling, and dynamic sequence generation.

These models were evaluated under various input configurations (corresponding to the problem type, such as computer vision or NLP) to induce and analyze graph breaks. This approach was intentionally designed to induce graph breaks, enabling us to capture a rich dataset of error scenarios. The captured data included information about unsupported operations, tensor mismatches, and fallback mechanisms triggered during execution. By deliberately exposing these models to a wide range of conditions, we ensured that our dataset encapsulated the breadth of challenges developers might encounter when using PyTorch's graph compilation tools in different contexts. This comprehensive dataset formed the foundation for understanding common failure patterns and building a robust debugging tool to address them effectively.

C. Optimization Procedure

Our tool focuses on optimizing the debugging process rather than training or inference directly. In particular, we work on two aspects of aiding the development process.

The first area involved developing an extensive taxonomy of graph breaks. By systematically analyzing and categorizing graph breaks based on their root causes, such as unsupported operations, shape mismatches, or dynamic control flows, we created a framework that goes beyond simple error reporting. This taxonomy serves two primary purposes. First, it enables our tool to provide targeted solutions tailored to specific error types, reducing the time and effort developers spend diagnosing issues. Second, it acts as a centralized hub of documentation, offering developers a clear and organized reference to understand the causes and resolutions of graph breaks. This dual function not only aids in error resolution but also educates users, helping them design models that are less prone to such issues in the future.

The second area of focus was integrating a large language model (LLM) into our development pipeline. By leveraging prompt-engineered queries, the LLM generates actionable, human-readable explanations for the graph breaks identified by our tool. This integration allows developers to receive contextualized explanations that go beyond the raw error messages typically provided by compilers or runtime environments. For example, instead of merely reporting an unsupported operation, the LLM can explain why the operation is unsupported, suggest alternative approaches, and provide code snippets or best practices to resolve the issue. This capability bridges the gap between technical error reporting and practical problem-solving, making the debugging process more intuitive and accessible.

Together, these two components, an extensive taxonomy of graph breaks and LLM-driven explanations, form the foundation of our tool. By combining structured error classification with intelligent, user-friendly explanations, we aim to significantly reduce the friction developers encounter when working with graph compilation errors, enabling them to focus more on innovation and less on debugging.

D. Profiling tools and methods

To enhance the performance and usability of the debugging tool, we leveraged PyTorch's profiling tools to monitor operation flows and identify the precise sources of graph breaks. These tools provided detailed runtime insights, such as operation types, tensor shapes, and fallback triggers, enabling us to diagnose issues like unsupported operations and inefficiencies during graph execution.

We also developed custom scripts to automate the extraction and categorization of graph break data. These scripts parsed outputs from tools like *torch._dynamo.explain()* to isolate meaningful information, including error messages and fallback details. The structured data generated was used to build a comprehensive taxonomy of graph breaks, allowing us to identify recurring patterns and provide targeted solutions efficiently.

To further improve accessibility, we integrated LLMs via APIs from Gemini and OpenAI. [2] [4] Using prompt-engineered queries, the LLMs transformed the structured data into human-readable explanations, offering actionable insights, resolution suggestions, and references to best practices. This integration streamlined the debugging process, making it more intuitive and effective for developers tackling complex machine learning workflows.

E. Evaluation Metrics

The effectiveness of the debugging tool was assessed using several key metrics. User feedback focused on the clarity and usefulness of LLM-generated explanations, with ratings providing insights into how well the tool addressed developers' needs. Coverage was another critical metric, measuring the proportion of graph break types the tool accurately identified and explained. This ensured the debugging tool could handle a wide variety of real-world scenarios.

Performance and taxonomy breadth were also central to the evaluation. Performance was measured by the tool's response time and its impact on overall debugging efficiency, highlighting its practicality in time-sensitive workflows. Taxonomy breadth evaluated the comprehensiveness of the graph break categorization, ensuring the taxonomy captured the diversity and complexity of errors encountered across different machine learning models. Together, these metrics provided a holistic view of the tool's utility and reliability.

IV. Experimental Results

A. Experimental Setup

The experimental evaluation was conducted on a diverse set of models from the Hugging Face repository, representing various domains such as computer vision, natural language processing, and generative modeling. Each model was tested under multiple configurations to induce graph breaks intentionally. The experiments were run on a machine equipped with a Google T4 GPU and PyTorch 2.0.

B. Performance Comparison (before and after optimizations) of Different Models

To evaluate the success of our work, we would likely need to adopt extensive supervised, user-feedback methods to analyze the effects of our tool on developer productivity. Since we lack the access to such data and feedback, much of the quantitative results must take a step back with qualitative assessments serving as a proxy for what we would deem as a success. Nonetheless, we outline a few approaches for quantitatively measuring our success, which could be adopted in scenarios where we would indeed have access to user data.

First off, the clarity of explanations can be indicated via user feedback, with the aim of illustrating that LLM-generated explanations were significantly clearer and more actionable compared to raw outputs from torch._dynamo.explain(). One can imagine a five star rating being aggregated across different users, in a similar manner to how ChatGPT acquired its feedback.

Another possible metric is debugging time reduction. For example, if we can show that, on average, users reported a 20% decrease in the time required to resolve graph break issues using the tool, then that would be a tremendous result for our tool.

One additional measurement could be the accuracy of our taxonomy. If the taxonomy correctly classified 90% of the observed graph break types, then we will have demonstrated its comprehensiveness and utility for users.

C. Analysis of Results

Qualitatively, our initial results highlight the potential of integrating LLMs into debugging workflows. The tool effectively translated verbose error messages into concise explanations, enabling developers to address issues more efficiently. This holds promise as we can look into further advancing, fine-tuning, and prompt engineering our models to improve our interpreted explanations. However, the lack of information from PyTorch's explain features caused some issues. Though it returned some fallback information, there was not any information that could help provide context, thus limiting the usefulness of some of our explanations. For this, we considered two possible approaches: 1) fine-tuning and improving our LLM so it doesn't rely on this context and 2) improving PyTorch's functionality to trace through these cases (and therefore provide more context). These presented two future avenues of exploration for our project.

On the taxonomy component of our project, the taxonomy provided an extensive, structured framework for understanding common graph break scenarios, making it easier for users to identify root causes and implement solutions. Across hundreds of different graph break examples, we identified five major categories, each with several subcategories. Our document in our repository entitled "graphbreak-taxonomy" describes each of them in detail, but in particular, we identified the following main results:

- *Unsupported External Functions and Libraries*: Calls to *unicodedata.category*, NumPy functions (*np.interp*), and PIL image conversions.
- *Scalar and Shape Extraction Ops*: Using *item()*, *.numpy()*, or *nonzero()* introduces dynamic operations not statically representable.
- *Dynamic Control Flow*: *if* conditions which are dependent on runtime values and masks.
- *Logging and Monitoring Calls*: *logger.warning_once* and tqdm progress bars break the graph due to non-compile-time-friendly IO/logging.
- *Model-Specific Patterns*: Certain models (like MusicGen) rely on complex conditionals and shape-based padding checks that cause breaks, while T5 triggers breaks due to logging inside the model.

TABLE I
Graph Break Taxonomy

Major Graph Break Type	**Minor Graph Break Type**
Operation-Based	Unsupported or Untraceable Built-ins
	Tensor Operations that Return Python Scalars
	Logger and Printing Functions
Dynamic Control Flow, Branching	Data-Driven Conditionals
	Dynamic Shape Operations
External Libraries and I/O	NumPy and PIL Interactions
	Using Python Structures and Iteration
Backend / Compiler Configuration	Settings and Flags Needed
	Deprecated / Missing Configurations
Model-Specific Break Patterns	Stable Diffusion Pipelines
	T5 Models
	MusicGen Models

V. Discussion

A. Interpretation of Results

The experimental results confirm that the proposed debugging tool significantly enhances the user experience for debugging graph breaks in PyTorch 2.0. The integration of an LLM proved instrumental in translating complex error messages into clear, actionable insights. Additionally, the taxonomy of graph break types offered a systematic approach to understanding and addressing these issues. The reduction in debugging time highlights the tool's potential to improve productivity for developers.

B. Comparison with Previous Studies

Compared to prior works in error explanation systems and dynamic graph debugging, our tool introduces a novel combination of taxonomy development and LLM-powered feedback. While existing solutions like TensorFlow's XLA tracing tools provide partial insights, they lack the user-centric approach of simplifying error messages and actionable solutions. The use of LLMs marks a significant advancement in accessibility and clarity. Moreover, the taxonomy we introduced has a much more detailed classification of graph break, while the default Pytorch functionalities only support two different types.

C. Challenges and Limitations

Despite its advantages, the tool has several limitations:

- Dependency on LLM Performance: The quality of the explanations is inherently tied to the performance of the underlying LLM. Hallucinations or incorrect outputs remain a risk.
- Limited Coverage of Edge Cases: While the taxonomy covers most observed graph breaks, rare and complex scenarios may still lack adequate documentation.
- Integration Overhead: Setting up the tool requires familiarity with both PyTorch profiling tools and LLM APIs, which may pose a barrier for novice users.
- Input Variability: Machine learning models evolve rapidly and accommodation for all kinds of models with different data types is very challenging.
- Difficulties on Performance Measurement: Model's performance can vary a lot on different software and hardware environments, and it is difficult to consider all the scenarios for every user. Moreover, we import and test on a lot of pre-trained models which we are not able to change the source code directly to test performance boost.

D. Future Directions

To address these challenges, future work could focus on:

- Input Accommodation: Testing on more models and data to accommodate more user input scenarios.
- Fine-Tuning the LLM: Collecting more labeled data on graph breaks to improve the specificity and accuracy of explanations.
- Expanding the Taxonomy: Conducting broader analyses to include additional edge cases and rare scenarios.
- Developing a Visual Interface: Creating an intuitive UI similar to TensorBoard [9] for visualizing graph breaks and associated metrics.
- Generalization to Other Frameworks: Extending the tool to support debugging for other machine learning frameworks, such as TensorFlow or JAX. Indeed, our approach is framework and language-agnostic and instead provides a general approach to this issue.

VI. Conclusion

Our proposed tool represents a significant step forward in improving the accessibility and usability of PyTorch's advanced features, paving the way for more efficient and effective debugging workflows.

This paper presents a novel debugging tool designed to address the challenges of understanding and resolving graph breaks in PyTorch 2.0. By leveraging torch._dynamo.explain() and integrating an LLM, the tool transforms verbose and technical error messages into clear, actionable insights. The development of a taxonomy for graph break types further aids in systematically identifying and addressing common issues. Experimental results demonstrate the tool's effectiveness in reducing debugging time, improving explanation clarity, and enhancing developer productivity.

The primary contributions of this work are as follows. We first helped aid the development of a taxonomy categorizing graph break types based on their causes and solutions. We also integrated an LLM to generate user-friendly explanations for graph break errors. Finally, we designed a method to measure the tool's impact through experimental evaluation, highlighting significant reductions in debugging time and improvements in explanation quality.

Building upon the findings of this work, future research could explore more models and input data types to collect more samples. Then, we could try to accommodate these new scenarios and fine-tune the LLM with additional labeled data to improve the accuracy and specificity of explanations. We also would like to expand the taxonomy to cover more edge cases and rare scenarios encountered in real-world applications. Extending the tool's capabilities to support other machine learning frameworks and debugging tasks is also another avenue which we could try to explore. Lastly, for low hanging fruit that is straightforward (but could still significantly improve user experience), we could try developing an intuitive user interface for visualizing graph breaks and associated metrics, inspired by tools like TensorBoard.

Acknowledgment

We would like to thank Professor Kaoutar El Maghraoui and Dr. Antoni Viros i Martin for their guidance and support throughout this project. We would also like to say thanks to the teaching staff of our HPML class at Columbia University: it was an interesting class and we enjoyed learning all the nuances of HPML this semester!

References

[1] eli5. https://pypi.org/project/eli5/. Accessed: 2024-12-20.

[2] Google ai for developers. https://ai.google.dev/. Accessed: 2024-12-20.

[3] How pytorch powers the ai revolution. https://codingscape.com/blog/how-pytorch-powers-the-ai-revolution. Accessed: 2024-12-20.

[4] Openai developer platform. https://platform.openai.com/docs/overview. Accessed: 2024-12-20.

[5] Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering. Accessed: 2024-12-20.

[6] Prompt engineering guide. https://www.promptingguide.ai/. Accessed: 2024-12-20.

[7] Prompt engineering: overview and guide. https://cloud.google.com/discover/what-is-prompt-engineering?hl=en. Accessed: 2024-12-20.

[8] Pytorch 2.0. https://pytorch.org/get-started/pytorch-2.0/. Accessed: 2024-12-20.

[9] Tensorboard: Tensorflow's visualization toolkit. https://www.tensorflow.org/tensorboard. Accessed: 2024-12-20.

[10] What is explainable ai (xai)? https://www.ibm.com/think/topics/explainable-ai. Accessed: 2024-12-20.

[11] What is jax? https://github.com/jax-ml/jax. Accessed: 2024-12-20.

[12] What is prompt engineering? https://www.ibm.com/think/topics/prompt-engineering. Accessed: 2024-12-20.

[13] Xla master documentation. https://pytorch.org/xla/master/debug.html. Accessed: 2024-12-20.

[14] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Stephanie N. Kadel. Prompt engineering: The art of getting what you need from generative ai. https://iac.gatech.edu/featured-news/2024/02/AI-prompt-engineering-ChatGPT. Accessed: 2024-12-20.

[16] Devi Prasad Khatua. What is torch.compile? https://medium.com/@wolframalphav1.0/what-is-torch-compile-e41c12a9f767. Accessed: 2024-12-20.

[17] Dr. Kaoutar El Maghraoui. Python and pytorch performance. https://courseworks2.columbia.edu/courses/203142/files/folder/Lecture Accessed: 2024-12-20.

[18] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[19] Mark Saroufim. Frequently asked questions. https://pytorch.org/docs/stable/torch.compiler$_faq.html. Accessed : 2024 - 12 - 20$.

[20] Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step-by-step, 2024.