Equivalent Graph via Equivalent Operators

SHIR-KANG SCOTT JIN, University of California Riverside, USA
HUGO TZU YU WAN, University of California Riverside, USA

Machine learning libraries, notably PyTorch[7], are fundamental for building and deploying modern AI systems. Recent research has rigorously examined PyTorch's diverse capabilities. Traditionally, testing methods often involved comparing CPU versus GPU execution outcomes. However, with the introduction of TorchInductor and TorchDynamo[1] in late 2022, PyTorch integrated sophisticated code optimization tools. These tools enable varied optimizations across different parts of PyTorch code, raising a crucial question: Can PyTorch consistently handle operations and complex code segments with semantically equivalent operators, ensuring correct data and control flow execution? To investigate this, we propose utilizing LLMs to generate intricate flow models and basic ASTs for translating equivalent operators.

ACM Reference Format:
Shir-Kang Scott Jin and Hugo Tzu Yu Wan. 2025. Equivalent Graph via Equivalent Operators. 1, 1 (), 4 pages.

1 Introduction

PyTorch is widely used in academia and industry, featuring in 63% of machine learning training[4]. Its popularity drives continuous enhancements and new feature development. The release of PyTorch 2.0 with Inductor and Dynamo[1] marks a milestone by optimizing modules using various techniques, although it has also introduced translation issues that necessitate thorough testing.

Many approaches exist for fuzzing PyTorch, often involving model creation, execution, and differential testing[2; 5; 6; 11]. Cradle [6] focused on direct comparisons between executions, calculating differences between examples and backends with input equivalence. While newer methods like WhiteFox[11] have a broader scope, they only conduct differential testing on different backends. Given PyTorch Inductor's features and observed problems, we believe evaluating models with complex control/data flow or equivalent function operators is crucial to assess post-optimization effects.

Our methodology leverages LLMs to generate initial PyTorch modules, streamlining the creation of baseline models. With these functional modules, we then employ abstract syntax tree (AST) analysis to transform Python mathematical operators into their Torch equivalents. This transformation allows us to investigate potential discrepancies in optimization and behavior between functionally equivalent Python and Torch operators.

2 Background

2.1 PyTorch Advances

The 2022 release of PyTorch 2.0 brought substantial performance improvements by optimizing the custom PyTorch code for specific hardware backends. This is enabled by just-in-time (JIT) compilation using tools like NCCL and Triton, Python-based compilers that translate PyTorch code before execution. This compilation step is highly efficient for machine learning tasks, as the upfront cost is offset by the repeated execution of compiled components, leading to overall performance gains. The compilation process includes applying numerous optimizations to the underlying computation graph. PyTorch optimizes the code by doing the following:

(1) Operator Fusion: Combines multiple operations into a single kernel to reduce memory access overhead and improve computational efficiency. For example, fusing element-wise operations like addition and multiplication into one kernel.
(2) Loop Optimizations: In Loop Fusion, it merges loops that iterate over the same data, minimizing memory access redundancy. And in Loop Tiling, it breaks down loops into smaller blocks to enhance cache utilization and reduce memory latency.
(3) Memory Layout Optimization: Adjusts data layouts, such as adopting a channel-last format, to align with hardware preferences, thereby improving access patterns and performance.
(4) Kernel Specialization: Generates device-specific kernels tailored to particular workloads, leveraging Just-In-Time (JIT) compilation to adapt to varying input sizes and hardware configurations.
(5) Vectorization: Utilizes SIMD (Single Instruction, Multiple Data) instructions, such as AVX2 and AVX512 on CPUs, to perform parallel computations, enhancing data processing efficiency.
(6) Weight Prepacking and Post-Operation Fusion: Employs libraries such as oneDNN to prepack weights and fuse post-operations, optimizing convolution and matrix multiplication operations.
(7) Parallelization: Implements parallel processing techniques using OpenMP for CPUs and CUDA streams for GPUs to distribute computations across multiple cores, reducing execution time.

Our work in this paper centers on this particular facet of PyTorch, recognizing its heightened susceptibility to errors. The significance of this issue is evidenced by the extensive PyTorch documentation dedicated to debugging compiled models, highlighting the challenges developers face in ensuring the correctness of optimized code.

2.2 Previous Approaches

The growing popularity of machine learning is driving the rapid expansion of machine learning libraries, creating significant testing hurdles. These hurdles stem from several factors: the rapid update cycle, which can introduce regressions; the pressure to release features quickly, sometimes at the expense of good design and correct implementation; and inherent programming errors leading to inaccurate results. Moreover, the stochastic nature of many machine

.

learning algorithms and the complexity of linear algebra computations make thorough verification exceptionally difficult.

2.2.1 Differential testing. Due to the high difficulty in recreating the computation ground up, the authors of Eagle created the idea to use differential testing on machine learning libraries by cross-checking the results using the same model description but on different backends. Although this approach cannot inherently identify every potential error in the code, it presents a convenient method to ensure consistency among different. However model level differential testing, does not guarantee absolute correctness as it can not discover errors that both backends contains. Many papers continue this approach by devising multiple ways to create models to test out the libraries, this includes papers such as NNsmith[5], DeepRel[3], and DocTer[10] that aim to devise ways to cover more types of operators. These methodologies rely on scrutinizing documentation and employing exact operator definitions to construct or extrapolate rules for generating models. However, the manual creation of these rules is a significant bottleneck, requiring considerable effort and meticulous precision. Moreover, the scalability of these generators is limited; creating larger models becomes increasingly difficult as maintaining adherence to generation rules becomes more complex with the increasing number of layers.

2.2.2 Equivalent graphs. Beyond using the same model and assuming they yield similar results, another approach is to create graphs that leads to the same result. Eagle[8] for example, splits inputs to sparse or changes the data format. Another example is PolyJuice[12], which tries to make the graph calculate to the same result, such as transposing twice. These tests are able to check the stability of operators and get a deeper insight of the actual calculation of the operation under test.

3 Exploration

These methodologies rely on scrutinizing documentation and employing exact operator definitions to construct or extrapolate rules for generating models. However, the manual creation of these rules is a significant bottleneck, requiring considerable effort and meticulous precision. Moreover, the scalability of these generators is limited; creating larger models becomes increasingly difficult as maintaining adherence to generation rules becomes more complex with the increasing number of layers.

To overcome these challenges, contemporary techniques like TitanFuzz[2] and Fuzz4ALL[9] advocate for the application of LLMs in model generation for fuzzing. With the ever expanding capacity to handle expanded input token sequences, LLMs can process substantial volumes of documentation, API descriptions, and source code. This capability enables the generation of code expressly designed to probe and analyze specific code behaviors.

3.1 PyTorch module

In this project, we will primarily focus on PyTorch modules that resemble the following structure.

```
class PtModule(nn.Module):
    def __init__(self):
        super(PtModule, self).__init__()
```

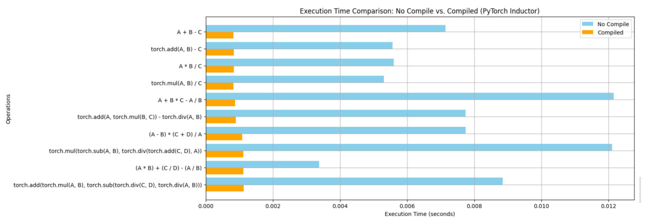


Fig. 1. Execution speed on different configurations

```
    def forward(self, x):
        # Some operators
```

We will then compile this with. Our main programming effort is to transform the operators in the forward section.

```
model = torch.compile(model, backend="inductor")
```

This module structure serves as the foundation for our experimental design. Consequently, given that mathematical operators are targeted for fusion by TorchInductor, the transformation of these operators constitutes a central focus of our investigation.

```
ast.Add->torch.add
ast.Sub->torch.sub
ast.Mult->torch.mul
ast.Div->torch.div
ast.MatMult->torch.matmul
ast.Pow and node.right.value == 0.5->torch.sqrt
ast.Pow-> torch.pow


ast.USub->torch.neg
ast.Not->torch.logical_not
ast.Invert->torch.bitwise_not
```

3.1.3 Comparators. While PyTorch provides a dedicated set of comparator operators, they are less versatile than standard Python operators. PyTorch comparators are specifically designed for tensor operations and cannot be directly applied to other Python data types. Furthermore, their utility is restricted when models are transformed into intermediate representations within PyTorch's compilation pipeline. Thus, we do not transform on comparators.

3.1.4 Compile and speed. To gain initial insights into execution performance, we performed a preliminary experiment evaluating speed across several configurations, detailed in Figure 1. Our observations indicated that uncompiled PyTorch code exhibited significant performance variability even when processing computationally equivalent operations. In contrast, compiled code demonstrated a marked reduction in this variability, with equivalent code segments achieving comparable execution times. Furthermore, the experiment revealed that increases in operator count led to only a fractional increase in total computation time, providing empirical evidence for the benefits of computation and operator fusion implemented by the compiler.

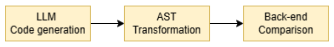
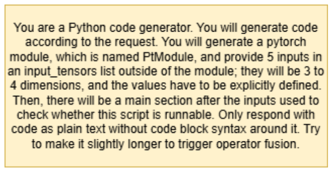


Fig. 2. General flow of our project



You are a Python code generator. You will generate code according to the request. You will generate a pytorch module, which is named PtModule, and provide 5 inputs in an input_tensors list outside of the module; they will be 3 to 4 dimensions, and the values have to be explicitly defined. Then, there will be a main section after the inputs used to check whether this script is runnable. Only respond with code as plain text without code block syntax around it. Try to make it slightly longer to trigger operator fusion.

Fig. 3. System instruction

3.2 Exploration

3.2.1 Property-based generator. While we posit that LLMs represent a valuable methodology, we also examined property-based generators as a potential alternative, motivated by their capacity to yield more controlled and reproducible test outcomes. We utilized the Hypothesis Library for this evaluation. Hypothesis exhibits proficiency in automated option selection and context-aware data generation, facilitating the creation of intricate tensor inputs, for instance, those suitable for matrix multiplication. Nonetheless, generating control flow using Hypothesis encountered significant obstacles. The inherent recursive nature of control flow specification necessitates deep recursion, leading to practical challenges associated with recursion depth limits.

3.2.2 ANTLR. While ANTLR is conventionally utilized for parser construction, it can also be adapted for generative purposes. This approach offers the benefit of defining generation logic through expressive, formula-like rules, similar to those used in pedagogical settings. Nevertheless, our experience revealed significant challenges in utilizing ANTLR4 as a generator. In particular, maintaining correct indentation within the generated code through ANTLR4's rule system proved to be a substantial obstacle, impeding practical application.

4 Approach

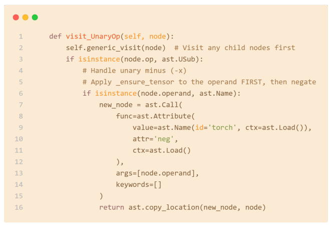
After determining that it is difficult to implement effective generation using previous, we devised the new architecture for our project as Figure 2. .

In our approach, Large Language Models (LLMs) were utilized to generate seed programs as a basis for testing. Subsequently, these programs were subjected to code transformation via Abstract Syntax Tree (AST) manipulation. The final stage involved comparing the performance of the transformed code across various devices to assess consistency and correctness.

4.1 Code generation

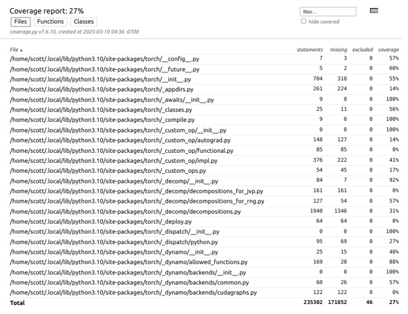
For code generation, we employed openai-gpt-4o-mini to automatically create PyTorch modules for testing. We instructed the model to generate modules with inputs that adhere to specific requirements, ensuring ease of loading and execution, as detailed in Figure 3.

The system instructions delineate the structural requirements for the generated test cases. These instructions specify the expected format of the inputs and the set of operators to be utilized within the modules and include a concise main execution block. This block is designed to verify the basic validity and runnability of the generated script. Subsequently, prompts are employed to define the specific module configurations desired, including features such as loops and conditional branches.



```python
def visit_UnaryOp(self, node):
    self.generic_visit(node)  # Visit any child nodes first
    if isinstance(node.op, ast.USub):
        # Handle unary minus (-x)
        # Apply _ensure_tensor to the operand FIRST, then negate
        if isinstance(node.operand, ast.Name):
            new_node = ast.Call(
                func=ast.Attribute(
                    value=ast.Name(id='torch', ctx=ast.Load()),
                    attr='neg',
                    ctx=ast.Load()
                ),
                args=[node.operand],
                keywords=[]
            )
            return ast.copy_location(new_node, node)
```

Fig. 4. Example transform code



Coverage report: 27%

coverage.py v7.6.10, created at 2025-03-10 04:36 -0700

File ▲	statements	missing	excluded	coverage
/home/scott/.local/lib/python3.10/site-packages/torch/__config__.py	7	3	0	57%
/home/scott/.local/lib/python3.10/site-packages/torch/__future__.py	5	2	0	60%
/home/scott/.local/lib/python3.10/site-packages/torch/__init__.py	704	318	0	55%
/home/scott/.local/lib/python3.10/site-packages/torch/_appdirs.py	261	224	0	14%
/home/scott/.local/lib/python3.10/site-packages/torch/_awaits/__init__.py	9	0	0	100%
/home/scott/.local/lib/python3.10/site-packages/torch/_classes.py	25	11	0	56%
/home/scott/.local/lib/python3.10/site-packages/torch/_compile.py	9	0	0	100%
/home/scott/.local/lib/python3.10/site-packages/torch/_custom_op/__init__.py	0	0	0	100%
/home/scott/.local/lib/python3.10/site-packages/torch/_custom_op/autograd.py	148	127	0	14%
/home/scott/.local/lib/python3.10/site-packages/torch/_custom_op/functional.py	85	85	0	0%
/home/scott/.local/lib/python3.10/site-packages/torch/_custom_op/impl.py	376	222	0	41%
/home/scott/.local/lib/python3.10/site-packages/torch/_custom_ops.py	54	45	0	17%
/home/scott/.local/lib/python3.10/site-packages/torch/_decomp/__init__.py	84	7	0	92%
/home/scott/.local/lib/python3.10/site-packages/torch/_decomp/decompositions_for_jvp.py	161	161	0	0%
/home/scott/.local/lib/python3.10/site-packages/torch/_decomp/decompositions_for_rng.py	127	54	0	57%
/home/scott/.local/lib/python3.10/site-packages/torch/_decomp/decompositions.py	1940	1346	0	31%
/home/scott/.local/lib/python3.10/site-packages/torch/_deploy.py	64	64	0	0%
/home/scott/.local/lib/python3.10/site-packages/torch/_dispatch/__init__.py	0	0	0	100%
/home/scott/.local/lib/python3.10/site-packages/torch/_dispatch/python.py	95	69	0	27%
/home/scott/.local/lib/python3.10/site-packages/torch/_dynamo/__init__.py	25	15	0	40%
/home/scott/.local/lib/python3.10/site-packages/torch/_dynamo/allowed_functions.py	169	20	0	88%
/home/scott/.local/lib/python3.10/site-packages/torch/_dynamo/backends/__init__.py	0	0	0	100%
/home/scott/.local/lib/python3.10/site-packages/torch/_dynamo/backends/common.py	60	26	0	57%
/home/scott/.local/lib/python3.10/site-packages/torch/_dynamo/backends/cudagraphs.py	122	122	0	0%
Total	**235302**	**171852**	**46**	**27%**

Fig. 5. Code coverage

4.2 Transforming operators

Transforming operations is done by using Python Abstract Syntax Trees; it is capable of understanding ordinary Python operators, thus we only have to overload the visit functions, such as in 4 to go through the Python script. As our task is to transform operators in the PyTorch module, we specifically find the "forward" function of the module and only transform operators in the function.

4.3 Comparison

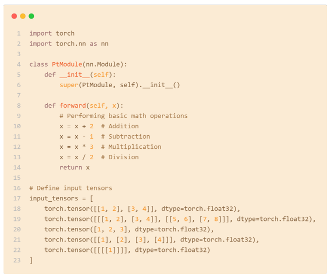
In the final part we get the results of the original script, compiled original script, transformed script, and compiled transformer script. Then, we compare it; the comparison is very complex, it includes tensors, lists, boolean, and different data types.

5 Evaluation

5.1 Results

We successfully executed our test suite; however, our chosen testing methodology resulted in a limited code coverage of 27%, as shown in Figure 5. This coverage constraint is a direct consequence of our approach, which focuses on converting operators.

Our tests, which employed basic mathematical operators, loops, and branches, detected only a few errors. This suggests either a high level of stability in PyTorch for these operations or that our test cases were not sufficiently sophisticated to reveal potential issues.



```
import torch
import torch.nn as nn

class PtModule(nn.Module):
    def __init__(self):
        super(PtModule, self).__init__()

    def forward(self, x):
        # Performing basic math operations
        x = x + 2  # Addition
        x = x - 1  # Subtraction
        x = x * 3  # Multiplication
        x = x / 2  # Division
        return x

# Define input tensors
input_tensors = [
    torch.tensor([[1, 2], [3, 4]], dtype=torch.float32),
    torch.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]], dtype=torch.float32),
    torch.tensor([1, 2, 3], dtype=torch.float32),
    torch.tensor([[1], [2], [3], [4]]], dtype=torch.float32),
    torch.tensor([[[[1]]]], dtype=torch.float32)
]
```

Fig. 6. Pytorch module with error

Error detection was primarily limited to cases of incorrect code generation rather than runtime errors in valid code. Although we allocated a $5 budget for OpenAI LLM queries, our actual spending was remarkably low, with 954 queries and 139,734 tokens costing only $0.23. This indicates that we significantly underutilized our allocated resources, suggesting that we could have explored the LLM's capabilities more extensively within our budget. In retrospect, a more ambitious approach to LLM query generation might have led to a more thorough investigation.

5.2 Shortfalls

5.2.1 Code generation. Our reliance on LLM-generated code inevitably led to the inclusion of some non-executable code in our tests. This highlights a known challenge with current LLMs: their ability to generate perfectly runnable code remains imperfect. A specific instance of this issue is the LLM's tendency to generate code with improperly balanced delimiters. For example, we observed cases where the LLM erroneously used a closing square bracket "]" in situations requiring a closing parenthesis ")" to maintain correct syntactic structure.

We recognize that our current methods for guiding the LLM in generating branching and looping patterns need further optimization. As a result, the generated code demonstrates a wide range of disparate and often unpredictable control flow implementations.

5.3 Inductor coverage

A notable limitation of our current approach is its exclusive focus on Python mathematical operators. We have not yet explored the use of NumPy operators in our testing framework. While incorporating NumPy operators could potentially broaden our test coverage, it would also introduce significant challenges. NumPy's extensive operator set and its capacity to handle diverse data types would substantially increase the complexity for both LLM-based code generation and our abstract syntax tree-based code transformation process.

5.4 Code coverage evaluation

We utilized Pytest Coverage for code coverage measurement, recognizing its potential imprecision for a library like PyTorch, particularly its C++ code. However, more accurate coverage tools proved too difficult to implement effectively. We encountered persistent build errors in PyTorch unit tests, and trace files for tools like gcov were transient and required repeated, time-consuming recompilation. These practical challenges made Pytest Coverage the most feasible option for our experiments.

6 Conclusion

We investigated PyTorch optimizations by transforming semantically equivalent mathematical operations using AST manipulation. Our approach leveraged LLM-generated PyTorch modules to evaluate discrepancies in performance and correctness. While TorchInductor optimizes many operations effectively, we observed inconsistencies in code generation and limited coverage. Our 27% code coverage highlights gaps in current testing methods, suggesting the need for broader evaluation of optimization stability.

References

[1] 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. 2024. 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.* ACM, La Jolla CA USA, 929–947. doi:10.1145/3620665.3640366

[2] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. ISSTA2023 Artifact for "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models". (May 2023). doi:10.5281/ZENODO.7980923 Publisher: Zenodo.

[3] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Automated Relational API Inference. doi:10.48550/ARXIV.2207.05531

[4] Adrienn Lawson, Stephen Hendrick, Nancy Rausch, Jeffrey Sica, and Marco Gerosa. 2024. *Shaping the future of generative AI: The impact of open source innovation.* Technical Report.

[5] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.* ACM, Vancouver BC Canada, 530–543. doi:10.1145/3575693.3575707

[6] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, Montreal, QC, Canada, 1027–1038. doi:10.1109/ICSE.2019.00107

[7] PyTorch. 2025. *PyTorch.* https://pytorch.org/

[8] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering.* ACM, Pittsburgh Pennsylvania, 798–810. doi:10.1145/3510003.3510165

[9] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* ACM, Lisbon Portugal, 1–13. doi:10.1145/3597503.3639121

[10] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, Virtual South Korea, 176–188. doi:10.1145/3533767.3534220

[11] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 709–735. doi:10.1145/3689736

[12] Chijin Zhou, Bingzhou Qian, Gwihwan Go, Quan Zhang, Shanshan Li, and Yu Jiang. 2024. PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 1309–1335. doi:10.1145/3689757