Bridging Operator Semantic Inconsistencies: A Source-Level Cross-Framework Model Conversion Approach

XINGPEI LI, National University of Defense Technology, China
YAN LEI, Chongqing University, China
ZHOUYANG JIA, National University of Defense Technology, China
YUANLIANG ZHANG, National University of Defense Technology, China
HAORAN LIU, National University of Defense Technology, China
LIQIAN CHEN, National University of Defense Technology, China
WEI DONG, National University of Defense Technology, China
SHANSHAN LI*, National University of Defense Technology, China

As deep learning (DL) frameworks become widely used, converting models between frameworks is crucial for ecosystem flexibility. However, interestingly, existing model converters commonly focus on syntactic operator API mapping—transpiling operator names and parameters—which results in API compatibility issues (i.e., incompatible parameters, missing operators). These issues arise from semantic inconsistencies due to differences in operator implementation, causing conversion failure or performance degradation.

In this paper, we present the first comprehensive study on operator semantic inconsistencies through API mapping analysis and framework source code inspection, revealing that 47% of sampled operators exhibit inconsistencies across DL frameworks, with source code limited to individual layers and no inter-layer interactions. This suggests that layer-wise source code alignment is feasible. Based on this, we propose ModelX, a source-level cross-framework conversion approach that extends operator API mapping by addressing semantic inconsistencies beyond the API level. Experiments on PyTorch-to-Paddle conversion show that ModelX successfully converts 624 out of 686 sampled operators and outperforms two state-of-the-art converters and three popular large language models. Moreover, ModelX achieves minimal metric gaps (avg. all under 3.4%) across 52 models from vision, text, and audio tasks, indicating strong robustness.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Model conversion, Deep learning library, Semantic compatibility

ACM Reference Format:
Xingpei Li, Yan Lei, Zhouyang Jia, Yuanliang Zhang, Haoran Liu, Liqian Chen, Wei Dong, and Shanshan Li. 2025. Bridging Operator Semantic Inconsistencies: A Source-Level Cross-Framework Model Conversion Approach. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE091 (July 2025), 23 pages. https://doi.org/10.1145/3729361

*Corresponding author

Authors' Contact Information: Xingpei Li, College of Computer Science and Technology, National University of Defense Technology, China, lixingpei123@nudt.edu.cn; Yan Lei, School of Big Data & Software Engineering, Chongqing University, China, yanlei@cqu.edu.cn; Zhouyang Jia, College of Computer Science and Technology, National University of Defense Technology, China, jiazhouyang@nudt.edu.cn; Yuanliang Zhang, College of Computer Science and Technology, National University of Defense Technology, China, zhangyuanliang13@nudt.edu.cn; Haoran Liu, College of Computer Science and Technology, National University of Defense Technology, China, liuhaoran14@nudt.edu.cn; Liqian Chen, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, lqchen@nudt.edu.cn; Wei Dong, College of Computer Science and Technology, National University of Defense Technology, China, wdong@nudt.edu.cn; Shanshan Li, College of Computer Science and Technology, National University of Defense Technology, China, shanshanli@nudt.edu.cn.

ACM 2994-970X/2025/7-ARTFSE091
https://doi.org/10.1145/3729361

1 Introduction

As deep learning (DL) research progresses rapidly, DL frameworks [1, 7, 25, 28, 38, 47] have become critical tools in both academic and industrial fields, offering robust support for training, evaluating, and deploying complex models [6, 36, 37, 39, 58, 64]. With the widespread adoption of these frameworks, the demand for cross-framework model conversion has increased, as it enables efficient model reuse across different platforms [12]. This makes high-quality model converters essential for ensuring the compatibility and performance of models in the DL ecosystem [27, 44].

Previous studies [26, 27, 44] have investigated model conversion errors, revealing that 74% of these errors stem from failures in operator conversion. A primary contributor to these failures is API compatibility issues [26]. The main root cause of these issues is operator semantic inconsistencies arising from operator implementation across frameworks, which significantly impact operator behavior, potentially causing converted model execution failure or performance degradation. Figure 1 illustrates an example of operator semantic inconsistencies between PyTorch [47] (left sub-figure) and Paddle [38] (right sub-figure) in *MaxPool2d*. Both frameworks use *kernel_size* to define the pooling window and *padding* for input padding, but PyTorch additionally supports *dilation*[1], which expands the effective pooling window by increasing the spacing between kernel elements (Lines 2-7). This effectively enlarges the receptive field and unexpectedly reduces the output size. For instance, given an 8×8 input with kernel_size=3, stride=2, and dilation=2, PyTorch computes an effective kernel size of 5×5, producing a 4×4 output, whereas Paddle, which lacks *dilation* support (Lines 2-5), applies a 3×3 kernel and outputs 3×3. This operator semantic inconsistency directly causes converted model execution failures through the change in output size. Even if conversion succeeds (i.e., the converted model runs without syntax errors or crashes), the altered receptive field due to missing dilation support degrades feature extraction capability [23, 35, 51, 53]. Thus, bridging operator semantic inconsistencies is essential for reliable cross-framework conversion.

However, current model converters [4, 9, 20, 24, 30, 34, 41] primarily focus on the source-to-source API mapping process, which involves transpiling the operator name and parameters via computation graphs [24, 34, 41] or unified operator APIs [4, 9, 20, 30] as intermediate representations (IRs) across different frameworks. While these approaches effectively align the syntax of API calls (i.e., renaming functions and parameters and reordering or combining parameters to match across frameworks), they have limitations in bridging semantic inconsistencies. This is because IRs, compared to source code, do not capture the fine-grained execution details of operators.

In this paper, we propose ModelX, a source-level model conversion approach that builds on current model converters [4, 9, 20, 24, 30, 34, 41] and extends them with a finer-grained strategy to bridge operator semantic inconsistencies directly at the source code level, going beyond traditional API-level mapping. The insight is that by modifying framework source code, we can bridge operator semantic inconsistencies during the operator mapping process. To achieve this goal, we have to solve the following challenges:

(1) *How to extract framework source code from the source framework and locate modification points in the target framework?* Extracting relevant framework source code from the source framework is essential to determine the part causing inconsistencies, while locating modifications in the target framework determines what needs to be modified. However, the dynamic scheduling mechanism introduces numerous function calls, making it difficult to trace kernel functions and extract specific code snippets, causing inconsistencies. Moreover, the framework source code in the source framework may not match the code structure in the target framework, making direct modification points unclear.

[1]Dilation: this introduces gaps in coverage of the filter in pooling layers, allowing it to sample over a wider area of the input without enlarging the filter size.

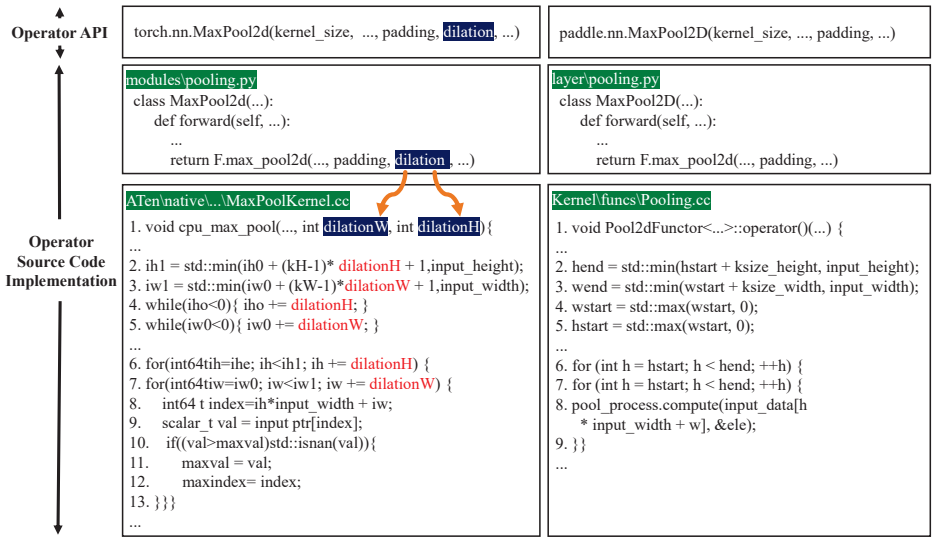

Fig. 1. An example of how operator semantic inconsistency impacts operator behavior (i.e., MaxPool2d) from the ground up.

(2) *How to align framework source code across different layers within the target framework?* Aligning framework source code effectively bridges operator semantic inconsistencies, but code snippets are distributed across multiple layers within the framework, making it unclear whether alignment should be done independently at each layer or globally, requiring inter-layer dependency analysis.

To address the above challenges, we conduct the first empirical study on operator semantic inconsistencies between PyTorch [47] and Paddle [38]. Specifically, we manually analyze the source-to-source operator API mapping to identify operator semantic inconsistencies (see Section 3.1.1) and extract relevant framework source code by tracing kernel functions in the source framework and comparing similar code regions in the target framework (see Section 3.1.2). These two steps are undertaken to construct a comprehensive operator mapping table that stores the mapping relationships, including compatible and incompatible operators (see Section 3.1.3). Finally, we provide findings into operator semantic inconsistencies and framework source code characteristics. Our study qualitatively analyzes 1,349 operators from two popular frameworks (i.e., PyTorch [47] and Paddle [38]), categorizing operator API mappings into three categories: **consistent API**, **inconsistent API**, and **no API pair**. We find that 47% of operator API mappings have semantic inconsistencies (i.e., corresponding to the categories of **inconsistent API** and **no API pair**). Moreover, the relevant framework source code is independently distributed across multiple layers within DL frameworks, and there is no inter-layer dependency across different layers. These findings reveal that operator semantic inconsistencies are widespread during cross-framework model conversion and can be bridged effectively by aligning relevant code snippets layer by layer.

Based on these findings, ModelX first parses the input source framework model to extract source framework operators. API mapping is performed for each operator using the operator mapping table from Section 3.1.3. To bridge operator semantic inconsistencies, ModelX employs an on-demand layered alignment approach that modifies framework code snippets only when incompatible parameters occur and an approximate expression synthesis approach that builds missing operators based on the function call stack.

We evaluate our experiments on PyTorch [47] and Paddle [38] due to their widespread adoption in academic research and industrial applications [31, 61]. Specifically, we test the conversion of 686

sampled operators from PyTorch to Paddle, achieving a 91% conversion success rate, demonstrating strong equivalence with low maximum absolute error (MAE) and root mean squared error (RMSE). Moreover, compared with two state-of-the-art (SOTA) model converters [4, 41] and three popular large language models (LLMs) (i.e., ChatGPT-3.5, ChatGPT-4o, DeepSeek-Coder), ModelX not only improves model performance by an average of 2% across 18 vision inference models but also maintains efficient evaluation latency and throughput with minimal overhead, comparable to IR-level converters. For robustness, we conduct tests involving 52 models from three common application fields, whose results indicate very minimal average metric gaps between frameworks, all under 3.4%. The results suggest that ModelX is highly robust, effectively bridging a wide range of operator semantic inconsistencies with consistent reliability and applying broadly across various application fields.

In conclusion, the primary contributions of this paper are systematically enumerated as follows:

- We conduct an empirical study on analyzing operator semantic inconsistencies between two DL frameworks, and we summarize three categories of semantic inconsistencies and describe their relevant framework source code characteristics.
- We devise and implement ModelX, a source-level cross-framework model conversion approach that builds upon previous converters by extending API mapping to bridge operator semantic inconsistencies at the source code level, going beyond simple API-level mapping, thus improving reliability and compatibility of cross-framework model conversion.
- We evaluate ModelX on PyTorch and Paddle, achieving a 91% success rate of 686 sampled operators and outperforming two SOTA model conversion approaches and three LLMs by supporting more models and improving model performance. Robustness tests across 52 models showed minimal performance gaps, further confirming the robustness of ModelX.

2 Background

In this section, we give some background on DL model converters with their associated problems (i.e., model conversion errors) and the multi-layer structure of DL libraries.

2.1 DL Model Converters

DL model converters are crucial for ensuring interoperability across different DL frameworks. Jajal et al [26] found that 52% of developers rely on conversion tools for framework compatibility [15, 26, 34]. These tools act as source-to-source transpilers, enabling model conversion between frameworks such as PyTorch [47], Paddle [38], and TensorFlow [1]. Beyond interoperability, cross-framework conversion enhances efficiency and supports the development of new DL frameworks. The conversion approaches typically employ either high-level intermediate representations (IR) for standardization and optimization or direct operator API mappings for model conversion:

Computation Graphs-Based Model Conversion. This conversion category [24, 34, 41] maps DL model layers and operations into a computation graph, which is an IR structured as a directed acyclic graph (DAG) [33, 56]. While capturing dependencies and data flows, IR-level converters may modify operators due to framework-specific exporters and importers (e.g., ONNX uses *torch.onnx* and *X2Paddle* for PyTorch-to-Paddle conversion), which often introduce approximations or simplifications. These conversions frequently face compatibility issues due to inconsistencies in operator semantics and definitions across frameworks [18, 26].

Operator APIs-Based Model Conversion. This conversion category [4, 9, 20, 30] reconstructs models by directly mapping each operator in models to the corresponding operator APIs in the target framework. Unlike computation graph-based model conversion, this approach works at the operator level without relying on an IR. While simplifying the conversion process, it shares

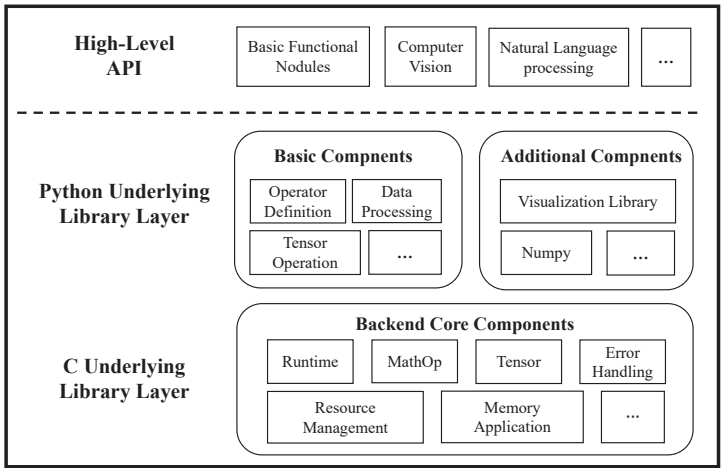

Fig. 2. Multi-layer structure of the DL library

compatibility challenges with computation graphs-based model conversion approaches, such as mismatches in operator definitions and parameterizations across frameworks.

2.2 Model Conversion Errors

Researchers have characterized model conversion errors [15, 18, 26, 44, 52]. Typically, these errors are classified into two main categories: *Crash* and *Wrong Model* (i.e., inconsistent behavior) [15, 26]. Shen et al [52] and Jajal et al [26] further identified that the majority of these errors occur during the operator conversion process, with their analysis indicating that operator compatibility issues are the primary cause. This highlights the inherent challenges of conversion, as it involves mapping between graphs expressed with different operators and semantics [18, 40, 41]. These API compatibility issues between frameworks emphasize the need for effective solutions and underscore the importance of improving operator compatibility in cross-framework model conversions.

2.3 Multi-Layer Structure of DL Libraries

As illustrated in Figure 2, DL libraries—key components of DL frameworks [59, 62]—are structured into three layers: a high-level API for model construction, a Python layer for operator and tensor management, and a C layer for low-level execution [59]. This hierarchical architecture poses significant challenges for addressing operator semantic inconsistencies, as resolving such issues demands cross-layer code analysis and modification. Consequently, it exacerbates the difficulty of achieving reliable cross-framework model conversion.

3 Empirical Study of Operator Semantic Inconsistency

Our empirical study focuses on operator semantic inconsistencies across different DL frameworks and is dedicated to answering the following two research questions (RQs).

Table 1. Distribution of operators sampling in PyTorch and Paddle

Framework	Company	Stars	Version	Tensor Ops[1]	Layer Ops[2]	Other Funs[3]	Total
PyTorch	Meta	81.8k	2.0.1	278	190	218	686
Paddle	Baidu	22.0k	2.5.2.post117	269	171	223	663

[1]**Tensor operations** (e.g., add, full); [2]**Network layers** (e.g., Conv, Loss);
[3]**Auxiliary tasks** (e.g., data preprocessing, device management, data loading).

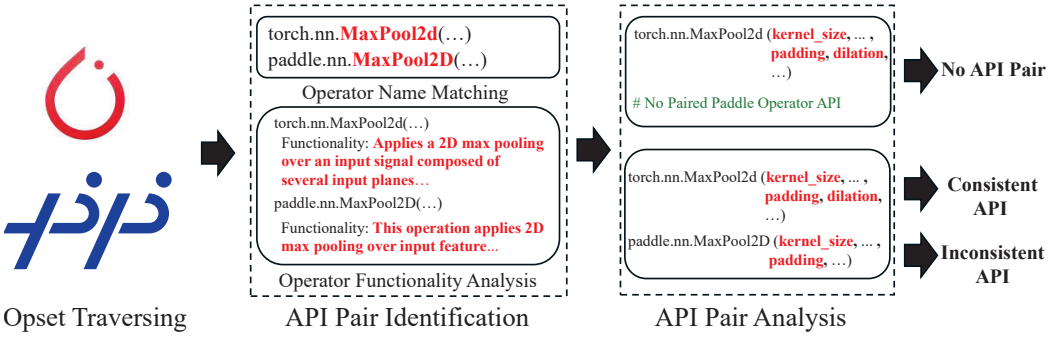

Fig. 3. Operator API mapping analysis

- **RQ1:** What kinds of semantic inconsistencies might exist among operators, and to what extent do they affect operator semantics?
- **RQ2:** What are the characteristics of framework source code related to operator semantic inconsistencies during the operator conversion process?

To answer the above RQs, we collect and analyze semantic inconsistencies in 1,349 sampled operators from two popular DL frameworks (i.e., PyTorch [47] and Paddle [38]). Our selection is based on operators frequently used in real-world models. We analyze 174 model types (i.e., categories like ResNet or VGG, not specific instances like resnet50 or vgg16) across six key fields in the official PyTorch repository [10] (i.e., audio, time series, text, vision, recommendation, and reinforcement learning), identifying 271 high-frequency operators (i.e., those appearing more than twice). Based on both frequency and functional similarity, we sample 686 PyTorch operators. Using the same criteria, we sample 663 Paddle operators from the official Paddle repository [46], ensuring representative and diverse operator selection across frameworks. Table 1 shows the distribution.

In this empirical study, we first identify operator semantic inconsistencies during the API mapping process (see Section 3.1), then provide findings to support the above RQs by analyzing identified semantic inconsistencies (see Section 3.2).

3.1 Operator Semantic Inconsistencies Identification

Our approach follows three key steps: First, we manually analyze the source-to-source operator API mapping to collect the operator API mapping relationship and classify mapping categories (see Section 3.1.1). Next, we extract framework source code of inconsistent mapping categories (see Section 3.1.2). Finally, we construct an operator mapping table that stores the operator API mapping relationship and framework source code related to operator semantic inconsistencies (see Section 3.1.3).

3.1.1 Operator API Mapping from Different Frameworks. We analyze operator API mappings between frameworks and assess the semantic consistency of paired operators. As shown in Figure 3, we first search for equivalent operators to form API pairs (i.e., equivalent operators from different frameworks that allow one API to replace another, referring to them as source_API and target_API, respectively) in PyTorch[47] and Paddle [38] by traversing their opsets. Using the Python *ast* module, we parse these APIs into abstract syntax trees (ASTs) to extract operator names, positional parameters, and keyword parameters. We then determine valid API pairs and evaluate their semantic consistency by comparing name and parameter mappings (i.e., name, type, and order). These mappings are established once and can be incrementally updated for new operators.

- **API Pair Identification.** We identify potential pairs for each operator API in both frameworks based on name similarity and functionality. We first use string-matching algorithms [2, 22] to

identify potential API pairs based on name similarity. Afterward, we manually verify its functionality by reviewing the official API documentation [45, 49]. An API pair is considered valid if it meets the following criteria: 1) the operators perform similar functions in both frameworks, 2) the input/output types match, and 3) the computational behavior and parameters align functionally. If any of these criteria are not met, the pair is considered invalid, and further analysis or alternative pairings are explored. If no valid pair is found, we conclude there is no valid API pair.

- **API Pair Analysis.** We analyze API compatibility to determine whether the API semantics are consistent. Once an API pair is identified, we extract operator names, positional parameters, and keyword parameters using the Python *ast* module, then establish the mapping relationships by establishing name mapping and parameter mappings. Name mapping is directly obtained by pairing operators with the same or equivalent names. For parameter mappings, we match parameters by name, analyze function descriptions, and manually check if any parameters in the APIs cannot be directly matched or mapped through combinations and reordering. If parameters remain incompatible (e.g., *dilation* in *torch.nn.MaxPool2d* is missing in Paddle), it indicates an API semantic inconsistency. Otherwise, the API semantics are considered consistent.

Based on this analysis, we categorize the mappings into three categories: (1) Valid API pair with consistent API semantics (i.e., **consistent API**); (2) Valid API pair with inconsistent API semantics (i.e., **inconsistent API**); (3) No valid API pair (i.e., **no API pair**). The second and third categories fall under *inconsistent mapping*. In the following subsection, we will further track framework source code related to operator semantic inconsistencies for the second and third categories.

3.1.2 Framework Source Code Extraction Associated with Inconsistent Mapping. We analyze framework source code and extract relevant code snippets. Algorithm 1 takes two inputs: *Category* (inconsistent mapping categories: **inconsistent API** and **no API pair**) and *APIPair*, and outputs *FrameworkCode* containing the relevant framework code snippets (Line 1).

For **no API pair** (Lines 2-8), the algorithm traces the function call stack in either the Python or C implementation of source_API (Line 3), typically in C. It filters out irrelevant allocation functions based on keywords (e.g., *dispatch*, *route*) (Line 6) and extracts function names and code snippets (Line 7). Non-dispatch functions are added to *FrameworkCode* (Line 8). For example, *torch.addcdiv* is traced to its kernel function *addcdiv_cpu_kernel* for reconstruction in the target framework.

For **inconsistent API** (Lines 9-19), the algorithm identifies incompatible parameters in source_API (Line 10) and traces relevant Python functions and framework source code in both source_API and target_API using Python Debugger (PDB) [48] (Lines 12-13). For source_API, it tracks code lines where parameters appear. For target_API, it extracts relevant functions (typically in *init* or *forward*) and their full framework source code, using a tree isomorphism algorithm [21] to locate similar code lines as modification points. If the parameter flows into the C backend (Lines 15-16), the algorithm traces the kernel function and collects the full C framework source code using GDB [19] (Lines 17-18). Finally, all framework source code related to operator semantic inconsistencies is recorded (Lines 14, 19). For example, Table 2 shows how the algorithm maps the incompatible parameter *dilation* in *torch.nn.MaxPool2d* to the modification point in *paddle.nn.MaxPool2D*.

3.1.3 Operator Mapping Table Construction. To support the above RQs and guide the source-level model conversion approach design, we organize a mapping table that includes operator API relationships from Section 3.1.1 and relevant framework source code related to semantic inconsistencies in API mappings from Section 3.1.2. Table 2 provides a detailed description. For instance, the mapping table shows *torch.nn.MaxPool2d*, which from PyTorch is mapped to *paddle.nn.MaxPool2D* in PaddlePaddle, illustrating the category of **inconsistent API**. The parameter mapping includes matched parameters such as *return_indices* to *return_mask*. The variant code records the details of

Algorithm 1: An analysis and tracking algorithm for framework source code related to operator semantic inconsistencies

Input: **Category**: Operator mapping category; **APIPair**: API pair involved in the mapping;
Output: **FrameworkCode**: Relevant framework source code snippets;

```
frameworkCode ← [];
if Category = no API pair then
    functionCallStack ← trace_source_API_function_call_stack(APIPair);
    for function in functionCallStack do
        functionName ← extract_function_name(function);
        if filter_dispatch_functions(functionName) = False then
            functionCode ← extract_function_code(function);
            frameworkCode ← frameworkCode ∪ {null, functionCode};
if Category = inconsistent API then
    incompatibleParams ← identify_incompatible_parameters(APIPair);
    for param ∈ incompatibleParams do
        PythonImpactFunctions ← trace_python_parameter_usage(param);
        PythonCodes ← capture_python_impact(param, PythonImpactFunctions);
        frameworkCode ← frameworkCode ∪ {param, PythonCodes};
        backendFlow ← trace_parameter_flow_to_backend(param, PythonImpactFunctions);
        if backendFlow exists then
            CImpactFunctions ← trace_source_API_parameter_usage(param, backendFlow);
            CCodes ← capture_C_parameter_impact(param, CImpactFunctions);
            frameworkCode ← frameworkCode ∪ {param, CCodes};
```

the framework source code necessary for bridging operator semantic inconsistencies. Specifically, this entry introduces *dilation* as an incompatible API parameter. It specifies that the corresponding source code is located within the C layer of DL frameworks and includes the necessary code context. Furthermore, it provides detailed code lines and modification points needed for alignment, such

Table 2. Operator mapping table definition

Entries	Description	Example
Operator Name	Associated source framework operator API name	torch.nn.MaxPool2d
Mapping Category	Category of operator API mapping	inconsistent API
Name Mapping	Operator API name mapping relationship	torch.nn.MaxPool2d→paddle.nn.MaxPool2D
Parameter Mapping	Operator API parameter mapping relationship	return_indices→return_mask, ...
Variant Code	Framework source code related to API incompatibility	
– Param	Incompatible API parameter	dilation
– Level	Specific Framework Layer for code snippet	C
– Context	Environment & dependencies for code snippet[1]	"headers": "#include xxx" ... , ...
– Content	Code lines and modification points for code snippet	"point": "hend = std::min(hstart + ksize_height, ..." "line": "ih1 = std::min(ih0 + (kH-1) * dilationH + 1, ..."

[1] **VariantCode Context:** Details of associated header files, namespaces, dependent sub-functions, and the source code of the corresponding functions.

as *"ih1 = std::min(ih0 + (kH-1) * dilationH + 1, ..."* and *"hend = std::min(hstart + ksize_height, ...;"*. Notably, when missing target framework operators, the modification point is set to *null*.

3.2 Findings of Empirical Study

To answer RQ1, we categorize operator semantic inconsistencies based on categories of operators API mapping (i.e., each sampled operator corresponds to one operator API mapping). We find that (1) 637 out of 1,349 operator mappings have operator semantic inconsistencies between DL frameworks; (2) Only 149 operator mappings lack a valid API pair in the target framework. In addition, we summarize three main categories of operator semantic inconsistencies, each illustrated in Figure 4 and Figure 5 and described below:

- **Operator Syntax Inconsistency** (712, 53%). These inconsistencies solely affect operator syntax (i.e., operator API), neither involving framework source code nor altering operator semantics. They manifest as differences in function paths, parameter definitions, or usage patterns across frameworks. They can be categorized into two main types: function signature inconsistencies, which involve differences in paths (i.e., module or namespace like *torch.nn* and *paddle.nn*), parameters, names, and return types across frameworks, accounting for 51%; and usage inconsistencies, where operators are combined differently within a model, requiring graph-level modifications, such as changing the order of operators. As shown in Figure 4, function signature inconsistencies are highlighted by inconsistencies in both function names and parameter labels (e.g., *torch.cat* versus *paddle.concat*, *dim* versus *axis*). Moreover, usage inconsistencies arise as PyTorch applies the scheduler after the optimizer, whereas Paddle sets it up before.
- **Operator Semantic Inconsistency** (488, 36%). These inconsistencies arise from different implementations of operators that should behave the same, resulting in conflicting results even

```
# Syntax Inconsistency in Signature
torch.cat(tensor1, dim=0)
torch.from_dlpack(...)

# Syntax Inconsistency in Usage
import torch.optim as optim
import torch,optim.lr_scheduler as lr
Optimizer=optim.Adam(...)
Scheduler=lr.StepLR(Optimizer, ...)
```

```
# Syntax Inconsistencies in Signature
paddle.concat(tensor1, axis=0)
paddle.utils.dlpack.from_dlpack(...)

# Syntax Inconsistency in Usage
import paddle.optimizer as optimizer
import paddle.optimizer.lr as lr
Scheduler=lr.StepDecay(...)
Optimizer=optimizer.Adam(Scheduler, ...)
```

Fig. 4. Examples of operator syntax inconsistency: PyTorch (left-listing) and Paddle (right-listing). Note the **red highlighted code** of instances.

```
// Semantic Inconsistency
torch.divide(a, b, rounding_mode=Trunc)

void div_trunc_kernel(auto& iter) {
...
cpu_kernel_vec(iter,
[](scalar_t a, scalar_t b){
   return std::trunc(a / b);},
[](...){
   return (a / b).trunc();});
});}
```

```
// Semantic Inconsistency
paddle.divide(a, b)

struct DivideFunctor{
...
inline HOSTDEVICE T operator()(const T a
    , const T b) const{
   return a / b;
  }
...
};
```

Fig. 5. Examples of operator semantic inconsistency: PyTorch (left-listing) and Paddle (right-listing). Note the **red highlighted code** of the instance.

when their APIs are similar. They are further categorized by underlying library layers of the DL framework: 4% occur in the Python layer, mainly involving parameter initialization and error handling, while 32% are in the C layer, with 7% related to tensor representation and 25% to computational logic. Notably, the Python and C layers (handling defaults/error management and tensor computation, respectively) are architecturally decoupled with no inter-layer dependencies. This resolves semantic inconsistencies through layer-specific code alignment without cross-layer coordination. As shown in Figure 5, *torch.divide* supports a *rounding_mode* parameter to control rounding, such as *trunc*, *floor*, while *paddle.divide* lacks this option, highlighting inconsistencies in operator semantics between the frameworks.

- **Operator Missing** (149, 11%). These inconsistencies arise when the corresponding operator is absent in the target framework, such as *torch.addcdiv* in PyTorch. We can partially achieve functional equivalence to bridge these gaps by combining existing target framework operators with arithmetic operations.

Finding 1: Bridging operator semantic inconsistencies is crucial, as 47% of inconsistencies alter operator semantics between frameworks. Moreover, alignment strategies depend on whether the target framework operator exists. Existing operators can be aligned directly, whereas missing ones require finding an equivalent.

To answer RQ2, we further analyze the "variant code" entry in Table 2, identifying relevant framework source code characteristics and their impact on operator semantics. The analysis targets two primary layers of the DL libraries: the Python layer, which handles defaults and error management, and the C layer, responsible for tensor representation and computation. Additionally, we examine inter-layer dependencies to assess if inconsistencies in one layer affect another, guiding strategies to bridge semantic inconsistencies across layers without causing inter-layer impacts.

Finding 2: Operator semantic inconsistencies are not confined to a single layer of DL libraries. Framework source code related to semantic inconsistencies is distributed across different layers without inter-dependencies between layers, making it feasible to resolve them layer by layer during cross-framework model conversion.

Our study highlights two important findings. Firstly, 47% of inconsistencies stem from operator semantics within DL libraries, underscoring the need to address these for successful cross-framework model conversion. Secondly, code snippets related to these semantic inconsistencies are spread across various layers of DL libraries. Crucially, there are no inter-dependencies among these snippets, enabling a layer-by-layer resolution of inconsistencies during the conversion process.

4 Design

In this section, we describe the design of ModelX, a source-level cross-framework model conversion approach that effectively bridges operator semantic inconsistencies in Section 3. ModelX is source-level (analyzing the specific implementation of each operator), layer-sensitive (considering framework source code that is distributed across the multi-layer structure of DL libraries), and designed for cross-framework model conversion. By using this approach, researchers and developers can effectively map operator APIs and address operator semantic inconsistencies during the mapping process across DL frameworks.

Figure 6 shows the overview of ModelX, which takes the source model code and the operator mapping table (which is described in Section 3.1.3) as inputs and outputs the target model code. Building on PaConvert [4] and the ONNX exporter in PyTorch [11], ModelX parses the source model code into an abstract syntax tree (AST), extracts source framework operators, and analyzes

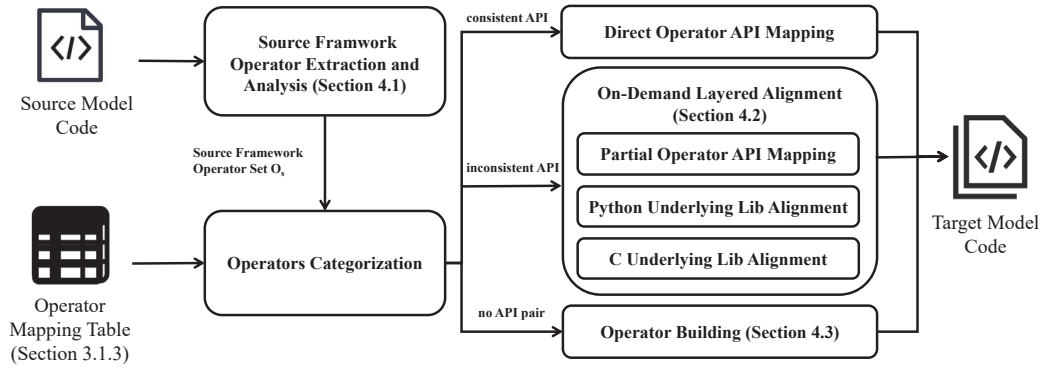

Fig. 6. Overview of ModelX

framework-specific nodes to extract operator context in the AST (*CC*) (see Section 4.1). We abstract the cross-framework mapping of each operator as a function $f : O_s \rightarrow O_t$, mapping operators from the source framework operator set O_s to the target framework operator set O_t. Then, for each o_s, we find the corresponding entries from the operator mapping table associated with o_s (Δ) to determine the mapping category, dividing them into three cases:

$$f(o_s, o_t, CC, \Delta) = \begin{cases} \sum_{i=1}^{1} h_i(o_s, o_t, CC, \Delta) & \text{if } \textbf{consistent API} \\ \sum_{i=1}^{L} h_i(o_s, o_t, CC, \Delta) & \text{if } \textbf{inconsistent API} \\ \text{operator\_building}(o_s, CC, \Delta) & \text{if } \textbf{no API pair} \end{cases}$$

If the mapping category is **consistent API**, ModelX directly maps the operator API to the target framework using the API mapping relationship from Δ, ensuring consistency with existing works [4, 30, 41]. If the mapping category is **inconsistent API**, f applies an on-demand layered alignment, where h_i handles parameter mappings across different layers. Compatible parameter mappings can be directly completed using Δ, while the remaining incompatible parameters require modifying relevant framework code snippets across different L layers of the target DL framework (see Section 4.2). If the mapping category is **no API pair**, f performs operator building, constructing the missing operator by analyzing the function call stack and generating an approximate expression (see Section 4.3). Finally, ModelX completes cross-framework conversion by updating *CC*.

4.1 Source Framework Operator Extraction and Analysis

Operator extraction parses the source model into an AST and identifies framework-specific nodes representing O_s. This involves traversing the AST from the root, focusing on nodes like *ast.Call* and *ast.Attribute*, to extract each operator and its context for cross-framework mapping.

- **Operator Extraction**. Operator extraction begins by traversing each node in the AST and checking the node's name against predefined keywords, typically the root module names of DL frameworks (e.g., *torch*, *torchvision*, *paddle*, *tensorflow*), which are derived from the official API documentation [16, 45, 49]. If the node's name matches any of these keywords, the node is identified as a framework-specific node. Finally, all identified nodes together from O_s.
- **Operator Context Analysis**. Operator context analysis extracts key attributes of each framework-specific node to identify each source framework operator and its context. Specifically, it retrieves the operator context *CC* from the source model AST. *CC* includes input/output shape and type, tensor type, parameters, values, parent-child relationships, and node position. Additionally, it locates Δ related to the operator in the operator mapping table, as shown in Table 2.

4.2 On-Demand Layered Alignment

We employ an on-demand layered alignment algorithm to map the operator name and parameters, and bridge incompatible API parameters, as shown in Algorithm 2. The algorithm traverses the alignment-oriented operators collected by node analysis and repeats the following process for each operator:

1) Get API mapping relationship and variant codes from the operator mapping table (Line 1).
2) Parse operator API in the source framework model to separate name, positional parameters, and keyword parameters (Line 2).
3) Map API name and parameters by API name mapping and API param mappings (Lines 3-4).
4) If args and kwargs from step 2 include incompatible API parameters (i.e., variant_code.Param), align variant code to bridge these inconsistencies. (Lines 5-13).
   – Locate and align framework source code in O_t (Line 9).
   – Pack and import aligned Python code or dynamically compile and link aligned C code (Lines 10-13).
5) Generate the aligned operator and update the source model AST. (Lines 14-15).

As shown in the function g, Algorithm 2 is divided into three components ($L = 3$) based on the multi-layer structure of DL libraries. These components are associated with specific functions: h_{API} for operator API mapping, $h_{PythonLayer}$ for Python framework source code alignment, and h_{CLayer} for C framework source code alignment. These functions are described as follows:

$$\sum_{i=1}^{L} h_i(o_s, o_t, CC, \Delta) = \begin{cases} h_{\text{API}}(o_s, CC, \Delta) = f(o_s, CC, \Delta) \\ h_{\text{PythonLayer}}(o_t, CC, \Delta) = \sigma(CC, \tau(O_t, \Delta)) \\ h_{\text{CLayer}}(o_t, \Delta) = \Omega(\tau(o_t, \Delta)) \end{cases}$$

In the formulas, f maps the operator API, σ packages and imports Python code, and Ω links the compiled dynamic libraries in the C layer. τ applies mappings based on Δ. Each component of Algorithm 2 is detailed as follows:

- **Partial Operator API Mapping (Lines 2-4).** This step maps only the operator API name and compatible parameters, leaving incompatible parameters unmapped. For instance, when converting *torch.nn.MaxPool2d* from PyTorch to Paddle, the mapped entries in Δ provide the updated name (i.e., *paddle.nn.MaxPool2D*) and parameters (e.g., *kernel_size*, *stride*), which are then applied to update *CC* and replace the operator context in the source model AST.
- **Python Underlying Lib Alignment (Lines 9-11).** This step aligns framework Python source code related to incompatible API parameters, then packs and imports the aligned code to bridge operator semantic inconsistencies. For instance, to bridge the incompatible parameter *alpha* from *torch.add*, We first locate similar code lines within the *add* function in O_t (i.e., if O_t is a class, modifications are made in the *init* or *forward* function; if O_t is a function, it is modified directly) based on modification points from Δ. If a match is found, we associate *alpha* with the corresponding code lines in Δ; If no match is found, we insert code lines from Δ directly into the function. This aligned function (i.e., *modified_add*) along with its context from Δ is packed in a new Python file. We then update *CC* with the import of this file and replace the original function with the aligned function (i.e., *add* replaced with *modified_add*). The updated *CC* replaces the operator context in the source model AST.
- **C Underlying Lib Alignment (Lines 9, 12-13).** This step aligns framework C source code related to incompatible API parameters, then compiles the aligned code into a dynamically linked shared library file and links the file. For instance, to bridge the incompatible parameter *dilation* from *torch.nn.MaxPool2d*, we adopt the same alignment approach as used for framework Python

Algorithm 2: A dynamic layered alignment algorithm for mapping operator API and bridging operator semantic inconsistencies

Input: **SO**: Operator in source framework model; **T**: Operator mapping table;
Output: **TO**: Aligned operator in target framework model;

```
1  name_mapping, param_mappings, variant_codes ← get_from_table(T, SO);
2  name, args, kwargs ← parse_api(SO);
3  new_name ← map_API_name(name, name_mapping);
4  new_kwargs ← map_API_params(args, kwargs, param_mappings);
5  if variant_codes exist then
6      for variant_code in variant_codes do
7          if variant_code.Param ∈ {args, kwargs} then
8              for line, point in variant_code.Content do
9                  ailgned_code ← locate_and_align_code(variant_code.Context, line,
                     point);
10                 if variant_code.Level = Python then
                       // Aligning Framework Python Source Code
11                     pack_and_import_code(ailgned_code);
12                 else
                       // Aligning Framework C Source Code
13                     compile_and_link_code(ailgned_code);
   // Aligned Nodes Generation
14 TO ← generate_aligned_node(SO, new_name, new_kwargs);
15 Return TO
```

source code to obtain the aligned function. This aligned function (i.e., *modified_maxpool2d*) along with its context from Δ is packed in a new C file. We then use a preconfigured CMakeList to compile the C file into a dynamically linked shared library file (i.e., *modified_maxpool2d.so*) and link it by setting *LD_PRELOAD* on Linux or *LoadLibrary* on Windows. The preconfigured CMakeList is configured as follows:

- **Initial CMake Configuration.** It sets the minimum required version of CMake, defines the project name, and specifies language standards (e.g., *C++11*, *C++14*, or *CUDA*), providing the necessary environment parameters.
- **Third-Party Library Configuration.** It configures all third-party libraries required for the project. It sets environment variables and extends CMake search paths to efficiently locate and integrate the necessary library configuration files. This streamlined process ensures the seamless integration of all required dependencies.
- **Compilation and Linking Configuration.** CMake commands are used to define and compile dynamic libraries in the project. This setup helps ensure they are correctly linked to all dependencies. It is necessary to obtain the current Conda environment (i.e., *CONDA_PREFIX*), DL library files (e.g, *PADDLE_LIBRARIES*), and the libraries for optimal linking (i.e., *add_library*).

The final step (Lines 14-15) of Algorithm 2 updates the source model AST to align operator semantics within the target framework by replacing framework-specific nodes with newly aligned operator nodes.

4.3 Operator Building

We attempt to generate an approximate expression in the target framework by analyzing the framework source code at the C layer of the source framework, as there are no inter-layer dependencies. We retrieve function calls from Δ and read their execution order, using target framework operators, parameters, and arithmetic operators to construct functionally equivalent code expressions.

To achieve this, we first extract function calls from Δ, corresponding to the "Variant Code" entry in the mapping table, which outlines the source framework's computation. Their order reflects the execution sequence, serving as a reference for the target framework. We then use regular expressions to track and parse:

- **Sub Function Call.** Identifying sub-functions invoked within the same code file that contribute to the computation. The extracted sub-functions are recursively tracked and parsed to reveal deeper computation logic.
- **Argument List.** Extracting key parameters such as *ksize, strides*, which influence computation.
- **Target Framework Operator.** Matching extracted operations and parameters against available operators in the target DL framework (e.g., *paddle.add*, *paddle.nn.functional.interpolate*) to build semantically equivalent expressions.
- **Arithmetic Operator.** Parsing binary and ternary arithmetic operations, including *+, -, *, /, &*, and conditional expressions like *? :*.

These elements are stored sequentially, matched with target framework operators, and reordered by precedence to reconstruct functionally equivalent expressions. For instance, *torch.addcdiv* follows an execution path from the high-level API to the kernel implementation (*addcdiv_cpu_kernel*), where the operation is computed as $value + value * tensor1/tensor2$, with *value*, *tensor1*, and *tensor2* as parameters of o_s.

5 Evaluation Setup

To evaluate ModelX, we consider the following three research questions (RQs):

- **RQ3:** How reliable and equivalent is ModelX in converting operators across frameworks? This question evaluates the conversion success rate, MAE, and RMSE of ModelX in 686 sample PyTorch operators.
- **RQ4:** Can ModelX outperform the existing state-of-the-art (SOTA) approaches for cross-framework model conversion? This question compares ModelX with ONNX [41], PaConvert [4], and three popular LLMs (i.e., ChatGPT-3.5, ChatGPT-4o, DeepSeek-Coder).
- **RQ5:** How robust is ModelX when applied to cross-framework model conversion tasks? This question evaluates the robustness by applying ModelX to cross-framework model conversion in three application fields: vision, text, and audio.

All of our experiments are conducted on a machine running 64-bit CentOS7.9 with a 52-core CPU (Intel(R) Xeon(R) Platinum8358), four A100 GPUs, and 394G RAM. To ensure the fairness of the comparison experiments, we conduct each experiment with identical configurations (one CPU core and the same GPU). We evaluate ModelX in cross-framework conversion from PyTorch 2.0.1 to Paddle 2.5.2, as detailed in Table 1, to answer these RQs. PyTorch and Paddle are popular DL frameworks, with 71.7K/20.9K stars on GitHub.

5.1 Implementation

We evaluate ModelX's reliability and equivalence using 686 operators, each with 10 LLM-generated test cases [42, 43], covering diverse conditions to compute success rate, MAE, and RMSE. We compare ModelX with two SOTA IR-level converters, ONNX [41] and PaConvert [4], using 18 vision models on ImageNet-1K [14] in dynamic graph mode. Latency is measured as the time to evaluate

the entire ImageNet-1K dataset, averaged over 10 independent executions of each model. To mitigate cold-start bias (e.g., initial GPU warmup), we discard the first execution and report the mean of the subsequent 10 runs. ONNX uses pytorch2onnx [10] and X2paddle [3], while ModelX standardizes weights via NumPy [57] for consistency. We also test three LLMs (`gpt-3.5-turbo-0125`, `gpt-4o-2024-05-13`, `deepseek-coder-V2-0724`) via OpenAI and DeepSeek APIs [13, 42, 43], using both original and chain-of-thought prompts [32, 60, 63] to identify operator semantic inconsistencies. For robustness, we select 13 high-frequency model types (e.g., ResNet, VGG) and choose representative instances (e.g., resnet50, vgg16), totaling 52 models. These are evaluated on CIFAR-10, CIFAR-100, FashionMNIST, IMDB, and Urbansound8K with task-specific training setups. Vision models are trained for 100 iterations (learning rate 0.001), text models on IMDB for sentiment analysis, and audio models on Urbansound8K with a reduced 0.0001 learning rate for better feature extraction.

5.2 Evaluation Metric

We evaluate the above RQs with the following metrics.

To evaluate the equivalence of ModelX, we utilize two key metrics: Maximum Absolute Error (MAE) and Root Mean Squared Error (RMSE), as suggested by existing studies [5]. These metrics evaluate the similarity of outputs before and after conversion. Following the approach in [15], we set a threshold ϵ of 1×10^{-4} to determine the equivalence of operators before and after conversion. Lower MAE and RMSE scores indicate consistent and equivalent conversions, confirming that our evaluation is reliable and based on established research.

- **MAE:** This metric measures the largest absolute difference between the predicted outputs in Paddle and the actual outputs observed in PyTorch during the experiment.

$$MAE = \max_{i=1}^{n} |y_i^{\text{actual}} - y_i^{\text{predicted}}|$$

  where y_i^{actual} and $y_i^{\text{predicted}}$ are the metrics from the Paddle and PyTorch operators, respectively.
- **RMSE:** This metric measures the square root of the average of the squared differences between the predicted outputs in Paddle and the actual outputs observed in PyTorch during the experiment.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i^{\text{actual}} - y_i^{\text{predicted}})^2}$$

  where y_i^{actual} and $y_i^{\text{predicted}}$ are as defined above.

In these comparison experiments, we evaluate model performance using four key metrics: Accuracy$\left(\frac{TP+TN}{TP+TN+FP+FN}\right)$, Precision$\left(\frac{TP}{TP+FP}\right)$, Recall$\left(\frac{TP}{TP+FN}\right)$, and F1 Score$\left(\frac{2\times\text{Recall}\times\text{Precision}}{\text{Recall}+\text{Precision}}\right)$. These metrics measure overall correctness, positive prediction accuracy, instance identification, and balance between precision and recall.

Additionally, to evaluate the robustness of ModelX in cross-framework model conversion, we utilize the **Metric Gap** to quantify performance variances before and after conversion, defined formally in the equation below.

$$Metric\ Gap = |Metric_{before} - Metric_{after}|$$

6 Result Analysis

6.1 RQ3: Reliability and Equivalence of ModelX

In this RQ, we first conduct a comprehensive experiment with 686 sampled operators from PyTorch [47], as detailed in Table 1. We evaluate the reliability of ModelX by calculating the proportion

Table 3. Overall conversion success of ModelX in sampled operators conversion

Operator Type	Number of Sampled Operators	Conversion Success Rate	Avg. Conversion Time (ms)	Metrics	
				Avg. MAE	Avg. RMSE
Tensor Ops	278	97.12% (270/278)	948.67	4.57×10^{-7}	2.13×10^{-6}
Layer Ops	190	98.95% (188/190)	924.14	3.82×10^{-6}	1.43×10^{-6}
Other Ops	218	76.14% (166/218)	1102.21	1.67×10^{-5}	4.36×10^{-5}

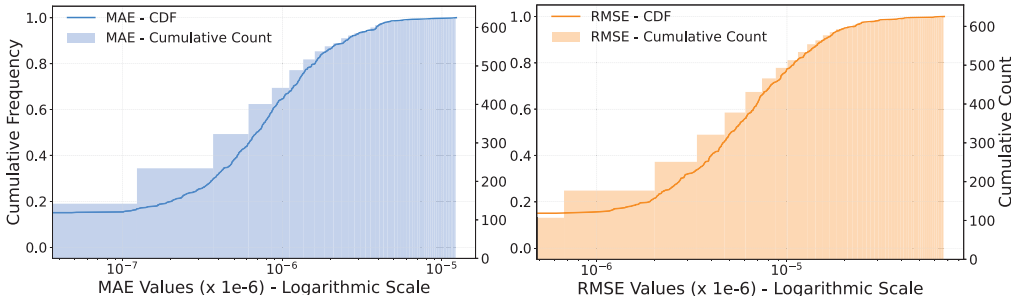

Fig. 7. Cumulative distribution of MAE and RMSE in assessing the equivalence of ModelX

of sampled operators that successfully pass all generated test cases, emphasizing the coverage of these test cases to encompass various operator usage scenarios, including extreme values and boundary conditions. Moreover, we assess the equivalence of ModelX by calculating MRE and RMSE, comparing these values against the threshold ϵ of 1×10^{-4} [15]. The experimental results are shown in Table 3 and Figure 7. Table 3 reveals that (1) ModelX converts 624 of 686 operators (91%), with two key categories (i.e., Tensor Ops and Layer Ops) achieving success rates over 95%, highlighting its reliability. Additionally, it successfully supports all 271 high-frequency operators in the real world (see Section 3 before Section 3.1); (2) Average MAE and RMSE for all operator types are well below ϵ, indicating high equivalence, whereas higher averages for Other Ops indicate precision challenges; (3) Average conversion times for all types of operators are low, highlighting the efficiency of ModelX. Figure 7 shows the cumulative distribution of MRE and RMSE with a logarithmic horizontal axis for variance clarity. Results indicate MRE and RMSE are mostly below the threshold ϵ. Specifically, 60% of MAE values fall under 1×10^{-6}, and over 90% under 5×10^{-6}. For RMSE, 30% are below 1×10^{-6}, 50% under 1×10^{-5}, and over 90% below 5×10^{-5}. Additionally, about 15% of MAE and RMSE values are zero, likely due to boolean outputs or simple computations.

Experimental Result to RQ3. The result indicates that ModelX successfully converts 91% of sampled operators, including key operator types like tensors and layers, confirming its reliability in cross-framework model conversion. The experiment validates its equivalence with consistently low MAE and RMSE across a wide range of values under the threshold ϵ. Further details on unsupported operators are provided in Section 7.

6.2 RQ4: Comparison against Existing IR-Level Works and LLMs

6.2.1 Comparison against IR-Level Model Conversion Approaches. We first compare ModelX with state-of-the-art ONNX [41] and PaConvert [4] for cross-framework model conversion using 18 vision inference models detailed in Table 4. We evaluate the converted Paddle models on ImageNet-1K [14], measuring accuracy (i.e., Precision, Recall, F1 Score) and efficiency (i.e., Evaluation Latency, the time to evaluate the entire dataset). Table 4 summarizes these well-known vision inference models, each with over 100 lines of code and a moderate number of operators. These models are commonly used in vision-related tasks. Table 5 presents the significance in two aspects: (1) With operator semantic inconsistencies (i.e., DenseNet, ShuffleNet, ResNet, Inception3), where ModelX significantly outperforms the baselines by supporting more models and improving performance by up to approximately 2%, indicating better compatibility and efficiency; (2) Without semantic

Table 4. Statistical comparison among vision inference models

Model Type \ Statistical	Tested Model Instances	Number of Operators in Model Type	Lines of Code
AlexNet	alexnet	23	45
Inception3	inception3	53	400
ResNet	resnet18, 34, 50, 101	45	258
DenseNet	densenet121, 161, 169, 201	42	190
ShuffleNet	shufflenetv2_0_5, v2_1_0, v2_1_5, v2_2_0,	43	151
VGG	vgg11, 13, 16, 19	31	72

Table 5. Result of comparison with SOTA approaches in cross-framework model conversion

Model type	Tool	Evaluation	Metrics			Model type	Tool	Evaluation	Metrics		
		Latency (s)	Precision	Recall	F1 Score			Latency (s)	Precision	Recall	F1 Score
AlexNet	ONNX	**116.01**	0.526	0.5256	0.5187	VGG	ONNX	116.54	0.6934	0.6893	0.6852
	PaConvert	119.52	0.526	0.5256	0.5187		PaConvert	117.12	0.6934	0.6893	0.6852
	ModelX	116.24	0.526	0.5256	0.5187		ModelX	**115.56**	0.6934	0.6893	0.6852
DenseNet	ONNX	Not supported				ShuffleNet	ONNX	Not supported			
	PaConvert	Not supported					PaConvert	Not supported			
	ModelX	**125.63**	**0.7507**	**0.7451**	**0.7423**		ModelX	**119.35**	**0.6793**	**0.6748**	**0.6708**
ResNet	ONNX	118.51	0.7418	0.737	0.7338	Inception3	ONNX	119.44	0.6801	0.6676	0.6635
	PaConvert	120.01	0.7417	0.7369	0.7337		PaConvert	Not supported			
	ModelX	**118.44**	**0.7642**	**0.7575**	**0.7614**		ModelX	**117.59**	**0.6921**	**0.6851**	**0.6812**

Note: (1) DenseNet, ResNet, ShuffleNet, and Inception3 **have operator semantic inconsistencies**, while AlexNet and VGG do not; (2) For each model type, latency and performance are averaged over **10 warm runs per instance**, then across instances (see Table 4).

inconsistencies (i.e., AlexNet, VGG), where ModelX significantly matches the baselines, ensuring no performance loss. Moreover, ModelX supports all tested models, while ONNX and PaConvert, as IR-level converters, fail to bridge incompatible parameters, causing conversion failures (e.g., *dilation* in *torch.nn.MaxPool2d* for DenseNet and ShuffleNet, *a* and *b* in *torch.nn.init.trunc_normal_* for Inception3). In addition, ModelX reduces average evaluation latency by approximately 0.46% over ONNX and 1.50% over PaConvert, while maintaining efficient dynamic graph execution.

6.2.2 Comparison against Three Popular LLMs. Besides comparing IR-level model conversion approaches, we evaluate three popular LLMs: ChatGPT-3.5, ChatGPT-4o, and DeepSeek-Coder by using both original prompting and chain-of-thought (COT) prompting (note that prompt details can be accessed via the GitHub link[2]). We first utilize LLMs to randomly generate five test cases for each model, followed by a cross-framework model conversion for each model. We then evaluate whether the converted model can pass all the test cases and check for any syntax and semantic errors in the converted model. The experiment results are shown in Table 6. We find that (1) ChatGPT-4o outperforms the other models, successfully passing all test cases (9/18); (2) Prompt type shows limited effects on LLMs during cross-framework model conversion tasks. In Table 6, COT is identified as a SOTA prompting technique, which benefits only ChatGPT-4o, particularly in addressing syntax errors in ShuffleNet and ResNet; (3) The main syntactic errors highlighted in the results include model structure errors and operator declaration errors; (4) LLMs fail to bridge operator semantic inconsistencies, as evidenced in ResNet conversions using ChatGPT-4o. Specifically, *mode* parameter in *torch.nn.init.kaiming_normal_* affects model weight updating during processing.

Experimental Result to RQ4. This result indicates that ModelX matches the capabilities of IR-level converters while improving performance in cross-framework model conversion. It effectively bridges operator semantic inconsistencies and maintains efficient evaluation latency and throughput with minimal overhead. Moreover, ModelX significantly outperforms LLMs, which struggle with syntactic correctness, especially when handling more complex model code.

[2]Prompt details are available at https://github.com/zuishenke123/ModelX/tree/master/EVALUATION/LLMsExperiment

Table 6. Overall effectiveness of LLMs in cross-frameworks model conversion

Model type	LLM	Prompt Type	All Cases Passed	Error Type	Model type	LLM	Prompt Type	All Cases Passed	Error Type
AlexNet	ChatGPT-3.5	Original, COT	Success	-	ShuffleNet	ChatGPT-3.5	Original, COT	Failed	Syntax
	DeepSeek-Coder	Original, COT	Success	-		DeepSeek-Coder	Original, COT	Failed	Syntax
	ChatGPT-4o	Original, COT	Success	-		ChatGPT-4o	Original	Failed	Syntax
							COT	Success	-
DenseNet	ChatGPT-3.5	Original, COT	Failed	Syntax	VGG	ChatGPT-3.5	Original, COT	Failed	Syntax
	DeepSeek-Coder	Original, COT	Failed	Syntax		DeepSeek-Coder	Original, COT	Failed	Syntax
	ChatGPT-4o	Original, COT	Failed	Syntax		ChatGPT-4o	Original, COT	Success	-
ResNet	ChatGPT-3.5	Original, COT	Failed	Syntax	Inception3	ChatGPT-3.5	Original, COT	Failed	Syntax
	DeepSeek-Coder	Original, COT	Failed	Syntax		DeepSeek-Coder	Original, COT	Failed	Syntax
	ChatGPT-4o	Original, COT	Failed	Semantic		ChatGPT-4o	Original, COT	Failed	Syntax

6.3 RQ5: Robustness of ModelX

To evaluate ModelX's robustness, we conduct performance and sensitivity analyses across various application fields: vision (41 models), text (3 models), and audio (8 models). The performance analysis involves calculating and displaying the average metric gaps for these models before and after conversion, using datasets like CIFAR10, CIFAR100, and FashionMNIST for vision models. Sensitivity analysis assesses model adaptability across different datasets within the same field. The results are presented in Table 7, Figure 8, and Figure 9. Table 7 presents average metric gaps across three application fields for ModelX, as well as two baselines (i.e., PaConvert and ONNX). The smallest average metric gap is 0.0188 for FashionMNIST, indicating strong consistency, while CIFAR100 shows the largest gap at 0.0335, highlighting areas where conversion robustness could be improved. Further analysis focuses on Precision and Recall, as F1 Score is simply a balance between the two and is not critical for this analysis. Moreover, compared to both baselines, ModelX achieves the best performance with the smallest metric gaps. Figure 8 reveals that (1) Most models have small gaps in Precision and Recall, centered around 0.0255 and 0.0222, indicating that ModelX effectively maintains the stability of these metrics during a model conversion; (2) These distributions closely follow the normal distribution, emphasizing the robustness of our model conversion tool. Moreover, we conducted a further sensitivity analysis on vision field models, analyzing the distribution of metric gaps across different datasets. The heatmap in Figure 9 shows that the FashionMNIST dataset has smaller metric gaps (0.0177 to 0.0206), indicating consistent performance across metrics. While CIFAR10 and CIFAR100 show slightly larger precision gaps (i.e., 0.0596 and 0.0362, respectively), overall consistency still demonstrates that ModelX is robust across datasets.

Experimental Result to RQ5. This result highlights the robustness of ModelX in cross-framework conversions, with minimal metric gaps (all under 3.4%) and stable behavior across

Table 7. Results of performance metrics for ModelX in cross-framework model conversion.

Fields	Datasets	Metrics	Frameworks		Metric Gap	ONNX Baseline	PaConvert Baseline
			PyTorch	Paddle			
Vision	CIFAR10	Precision	0.8222	0.7626	**0.0334**	0.0661	0.0680
		Recall	0.7698	0.7486			
		F1 Socre	0.7685	0.7490			
	CIFAR100	Precision	0.4897	0.4535	**0.0335**	0.0685	0.0713
		Recall	0.4603	0.4289			
		F1 Socre	0.4534	0.4204			
	FashionMNIST	Precision	0.9133	0.8956	**0.0188**	0.0490	0.0374
		Recall	0.9098	0.8918			
		F1 Socre	0.9093	0.8887			
Text	IMDB	Accruacy	0.7701	0.7824	**0.0261**	0.0572	0.0450
		Precision	0.7607	0.8065			
		Recall	0.7886	0.7431			
		F1 Socre	0.7742	0.7734			
Audio	Urbansound8K	Accuracy	0.3160	0.3042	**0.0097**	0.0380	0.0321
		Precision	0.2838	0.2770			
		Recall	0.3138	0.3243			

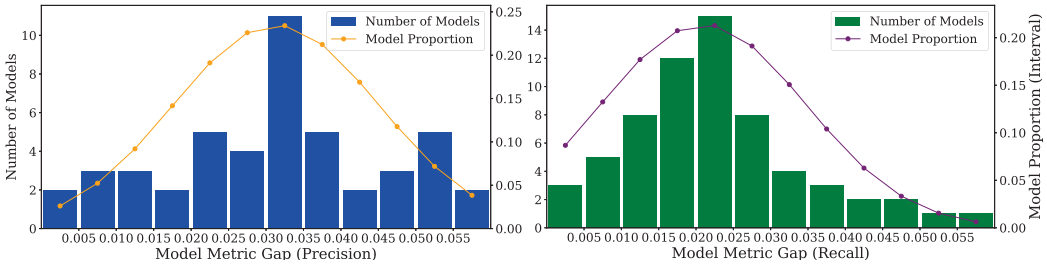

Fig. 8. Distribution of metric gaps (i.e., Precision, Recall) for ModelX in cross-framework model conversion

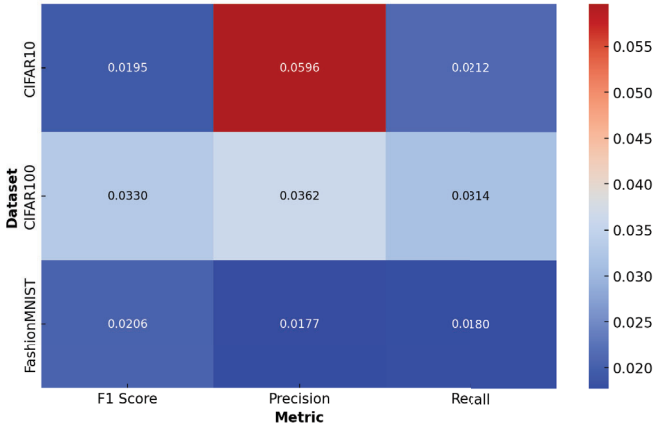

Fig. 9. Heatmap of sensitivity analysis for ModelX in vision fields

datasets, demonstrating strong robustness and practical applicability in real-world cross-framework deployment scenarios.

7 Limitation and Discussion

Unsupported Operators Analysis. Despite its broad capabilities, ModelX does not support 62 specific operators from PyTorch [47] to Paddle [38], mainly those infrequently used. Key reasons for these limitations include: (1) Unsupported framework-specific mechanisms: *TorchScript* [17] in PyTorch supports dynamic graph generation with flexible tracing and scripting. In contrast, JIT, using the *@paddle.jit.to_static* decorator, is less adaptable to dynamic changes, complicating the direct conversion of PyTorch operations reliant on complex control flows; (2) Operator compatibility issues: We bridge missing operator inconsistencies by attempting to map PyTorch operators to existing Paddle equivalents. However, if directly building PyTorch operators with available Paddle operators is not feasible, the conversion is not supported. Creating new operators within the DL library, a process known as operator porting, is resource-intensive and complex. Instead, operator conversion focuses on aligning existing inconsistencies more practically.

Threats to Viability of Conversion. DL frameworks like TensorFlow [1], PyTorch [47], and PaddlePaddle [38] support multiple backends (e.g., CPU, GPU), impacting operator semantic consistency. ModelX guarantees that each converted model is executable on at least one supported backend. While our tests focus on PyTorch and Paddle, future work will expand to more frameworks and models for validation.

Generalizability and Specificity. For generalizability, ModelX extends to TensorFlow [1] and MXNet [7], converting the same 18 vision inference models from Table 4 (see Section 6.2.1). As shown in Table 8, it achieves success rates of 94.4% and 78%, where success means all operators are

Table 8. Generalizability of ModelX in cross-framework conversion

Conversion Process	Tested Number	Successful Conversions	Failed Conversions	Success Rate
PyTorch -> TensorFlow	18	17	1	94.4%
PyTorch -> MXNet	18	14	4	78%

correctly mapped (no failures, MRE/RMSE below ϵ) and the model runs on ImageNet-1K [14]. Most failures are due to MRE/RMSE exceeding ϵ. Although supporting new frameworks requires building mapping tables, this is a one-time and reusable effort. We plan to extend to more frameworks to improve generalizability further. For specificity, ModelX effectively bridges operator semantic inconsistencies, outperforming IR-level converters. Conversion starts from IR for efficient API mapping, and when needed, proceeds to source-level adjustments to handle semantic and compatibility issues (e.g., incompatible parameters, missing operators), leveraging IR and source code complementarily for better accuracy. ModelX targets open-source DL frameworks [1, 7, 38, 47], enabling the reconstruction of missing operators in target frameworks.

Impact on Deep Learning Transpilers. ModelX enhances existing DL transpilers [4, 9, 20, 24, 30, 34, 41] by bridging operator semantic inconsistencies at the source code level. In contrast to traditional approaches that focus on API syntax mapping and graph-based conversion, ModelX modifies framework source code to improve cross-framework compatibility. Furthermore, it reduces complexity by eliminating the typical two-step conversion pipeline (A→IR, IR→B), and introduces layered code alignment techniques that preserve semantic consistency in converted models.

8 Related Work

Computation Graph-Based Model Converters. Related works in graph-based model conversion include ONNX [41], MMdnn [34], Jittor [24], TVM [8], and MLIR [29]. ONNX enables framework interoperability, while MMdnn provides a toolkit for model migration. Jittor supports dynamic optimization of computation graphs, TVM compiles models into optimized low-level code, and MLIR facilitates conversion with domain-specific languages.

Operator APIs-Based Model Converters. Unified operator APIs facilitate model migration across frameworks through high-level interfaces. For example, Keras [9] provides a versatile interface compatible with TensorFlow [1], Theano [55], and CNTK [50], enhancing model portability. PaConvert [4], created by the Paddle team, efficiently converts models from PyTorch to Paddle, ensuring consistent performance. TensorLayerX [54] offers unified APIs for constructing and converting models across various frameworks.

9 Conclusion

This paper presents the first empirical study of operator semantic inconsistencies in DL frameworks, demonstrating their critical impact on model reliability during cross-framework conversion. We propose ModelX, a source-level approach that directly modifies framework code (beyond API-level mapping) to resolve inconsistencies. Experiments show that ModelX outperforms existing converters and LLMs while maintaining robustness across various application fields.

10 Data Availability

ModelX and its data are publicly available at: https://github.com/zuishenke123/ModelX.git

Acknowledgments

The authors express thanks to the anonymous reviewers for their insightful comments. This research was funded by NSFC (No. 62272473, No.U2441238 and No.62202474) and the Science and Technology Innovation Program of Hunan Province (No.2023RC1001 and No.2023RC3012).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] K Azarudeen, G Vinoth Chakkaravarthy, Premkumar Murugiah, and S Kharthikeyan. 2021. A novel approach for pattern string matching in intrusion detection system. In *Journal of Physics: Conference Series*, Vol. 1916. IOP Publishing, 012007. doi:10.1088/1742-6596/1916/1/012007

[3] Baidu. 2019. X2Paddle. https://github.com/PaddlePaddle/X2Paddle. Accessed: 2024-07-05.

[4] Baidu. 2022. PaConvert. https://github.com/PaddlePaddle/PaConvert. Accessed: 2024-07-05.

[5] Alexei Botchkarev. 2018. Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology. *arXiv preprint arXiv:1809.03006* (2018). doi:10.28945/4184

[6] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 357–369. doi:10.5281/zenodo.7060209

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015). doi:10.48550/arXiv.1512.01274

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[9] François Chollet and the Keras Team. 2015. Keras: The Python Deep Learning library. https://keras.io.

[10] PyTorch Contributors. 2024. PyTorch. GitHub repository. https://github.com/pytorch/pytorch

[11] PyTorch Contributors. 2025. Torch ONNX Exporter. https://pytorch.org/docs/stable/onnx.html Accessed: 2025-01-30.

[12] James C Davis, Purvish Jajal, Wenxin Jiang, Taylor R Schorlemmer, Nicholas Synovic, and George K Thiruvathukal. 2023. Reusing deep learning models: Challenges and directions in software engineering. In *2023 IEEE John Vincent Atanasoff International Symposium on Modern Computing (JVA)*. IEEE, 17–30. doi:10.1109/JVA60410.2023.00015

[13] DeepSeek. 2024. Models - DeepSeek-coder. https://platform.deepseek.com/api-docs Accessed: 2024-08-18.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 248–255. doi:10.1109/CVPR.2009.5206848

[15] Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. 2023. Differential Testing of Cross Deep Learning Framework APIs: Revealing Inconsistencies and Vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7393–7410. https://www.usenix.org/conference/usenixsecurity23/presentation/deng-zizhuang

[16] TensorFlow Developers. 2025. TensorFlow Official API Documentation. https://www.tensorflow.org/api_docs Accessed: 2025-01-30.

[17] Zachary DeVito. 2022. Torchscript: Optimized execution of pytorch programs. *Retrieved January* (2022).

[18] Adrien Gauffriau, Iryna De Albuquerque Silva, and Claire Pagetti. 2023. Formal description of ML models for unambiguous implementation. *arXiv preprint arXiv:2307.12713* (2023). doi:10.48550/arXiv.2307.12713

[19] GNU Project. 2023. *GDB: The GNU Project Debugger*. https://www.gnu.org/software/gdb/ Accessed: 2024-06-02.

[20] Linyuan Gong, Jiayi Wang, and Alvin Cheung. 2024. ADELT: transpilation between deep learning frameworks. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*. 6279–6287. doi:10.24963/ijcai.2024/694

[21] Yuanjun Gong, Jianglei Nie, Wei You, Wenchang Shi, Jianjun Huang, Bin Liang, and Jian Zhang. 2024. SICode: Embedding-Based Subgraph Isomorphism Identification for Bug Detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 304–315. doi:10.1145/3643916.3646556

[22] Saqib Iqbal Hakak, Amirrudin Kamsin, Palaiahnakote Shivakumara, Gulshan Amin Gilkar, Wazir Zada Khan, and Muhammad Imran. 2019. Exact string matching algorithms: survey, issues, and future research directions. *IEEE access* 7 (2019), 69614–69637. doi:10.1109/ACCESS.2019.2914071

[23] Kaiyang Han, Fanzhi Cao, Tianxin Shi, and Pu Wang. 2023. A Dual Attention Network for Multimodal Remote Sensing Image Matching. In *2023 4th International Conference on Computer Vision, Image and Deep Learning (CVIDL)*. IEEE, 128–134. doi:10.1109/CVIDL58838.2023.10166096

[24] Shi-Min Hu, Dun Liang, Guo-Ye Yang, Guo-Wei Yang, and Wen-Yang Zhou. 2020. Jittor: a novel deep learning framework with meta-operators and unified graph execution. *Science China Information Sciences* 63 (2020), 1–21. doi:10.1007/s11432-020-3097-4

[25] Ltd. Huawei Technologies Co. 2022. Huawei mindspore ai development framework. In *Artificial Intelligence Technology*. Springer, 137–162. doi:10.1007/978-981-19-2879-6_5

[26] Purvish Jajal, Wenxin Jiang, Arav Tewari, Erik Kocinare, Joseph Woo, Anusha Sarraf, Yung-Hsiang Lu, George K Thiruvathukal, and James C Davis. 2024. Interoperability in deep learning: a user survey and failure analysis of ONNX model converters. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1466–1478. doi:10.1145/3650212.3680374

[27] Wenxin Jiang, Vishnu Banna, Naveen Vivek, Abhinav Goel, Nicholas Synovic, George K Thiruvathukal, and James C Davis. 2024. Challenges and practices of deep learning model reengineering: A case study on computer vision. *Empirical Software Engineering* 29, 6 (2024), 142. doi:10.1007/s10664-024-10521-0

[28] Haifeng Jin, Franois Chollet, Qingquan Song, and Xia Hu. 2023. AutoKeras: An AutoML Library for Deep Learning. *Journal of Machine Learning Research* 24, 6 (2023), 1–6. http://jmlr.org/papers/v24/20-1355.html

[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020). doi:10.48550/arXiv.2002.11054

[30] Daniel Lenton, Fabio Pardo, Fabian Falck, Stephen James, and Ronald Clark. 2021. Ivy: Templated deep learning for inter-framework portability. *arXiv preprint arXiv:2102.02886* (2021). doi:10.48550/arXiv.2102.02886

[31] Jean-Sébastien LERAT, Ahmed Sidi Mahmoudi, and Saïd Mahmoudi. 2021. Deep Learning Frameworks: Performances Analysis. (2021).

[32] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* (2023). doi:10.1145/3690635

[33] Jialin Li, Xueyi Li, and David He. 2019. A directed acyclic graph network combined with CNN and LSTM for remaining useful life prediction. *IEEE Access* 7 (2019), 75464–75475. doi:10.1109/ACCESS.2019.2919566

[34] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1320–1330. doi:10.1145/3368089.3417051

[35] SR Lyernisha, C Seldev Christopher, and SR Fernisha. 2023. Object recognition from enhanced underwater image using optimized deep-CNN. *International Journal of Wavelets, Multiresolution and Information Processing* 21, 04 (2023), 2350007. doi:10.1142/S0219691323500078

[36] Yingwei Ma, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. 2023. At which training stage does code data help llms reasoning? *arXiv preprint arXiv:2309.16298* (2023). https://arxiv.org/abs/2309.16298

[37] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to understand whole software repository? *arXiv preprint arXiv:2406.01422* (2024). doi:abs/2406.01422

[38] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. 2019. PaddlePaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing* 1, 1 (2019), 105–115. doi:10.11871/jfdc.issn.2096.742X.2019.01.011

[39] Gaurav Menghani. 2023. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *Comput. Surveys* 55, 12 (2023), 1–37. doi:10.1145/3578938

[40] MLEAP Consortium. 2023. *EASA Research – Machine Learning Application Approval (MLEAP) Interim Technical Report*. Technical Report. European Union Aviation Safety Agency. Horizon Europe research and innovation programme report.

[41] ONNX. 2017. Open Neural Network Exchange. https://onnx.ai/. Accessed: 2024-05-08.

[42] OpenAI. 2024. Models - GPT-3.5-turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo Accessed: 2024-08-18.

[43] OpenAI. 2024. Models - GPT-4o. https://platform.openai.com/docs/models/gpt-4o Accessed: 2024-08-18.

[44] Moses Openja, Amin Nikanjam, Ahmed Haj Yahmed, Foutse Khomh, and Zhen Ming Jack Jiang. 2022. An empirical study of challenges in converting deep learning models. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 13–23. doi:10.1109/ICSME55016.2022.00010

[45] PaddlePaddle. 2025. *PaddlePaddle Documentation*. https://www.paddlepaddle.org.cn/documentation/docs/en/2.5/api/index_en.html Accessed: 2025-01-30.

[46] PaddlePaddle. 2025. PaddlePaddle GitHub Repository. https://github.com/PaddlePaddle Accessed: 2025-02-10.

[47] A Paszke. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019). doi:10.48550/arXiv.1912.01703

[48] Python Software Foundation. 2023. *pdb - The Python Debugger*. https://docs.python.org/3/library/pdb.html Accessed on 2023-10-25.

[49] PyTorch. 2025. *PyTorch Documentation*. https://pytorch.org/docs/2.0/ Accessed: 2025-01-30.

[50] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2135–2135. doi:10.1145/2939672.2945397

[51] Thomas Serre, Lior Wolf, and Tomaso Poggio. 2005. Object recognition with features inspired by visual cortex. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 2. IEEE, 994–1000. doi:10.1109/CVPR.2005.254

[52] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 968–980. doi:10.1145/3468264.3468591

[53] Luna Sun, Zhenxue Chen, QM Jonathan Wu, Hongjian Zhao, Weikai He, and Xinghe Yan. 2021. AMPNet: Average-and max-pool networks for salient object detection. *IEEE Transactions on Circuits and Systems for Video Technology* 31, 11 (2021), 4321–4333. doi:10.1109/TCSVT.2021.3054471

[54] TensorLayer Team. 2023. TensorLayerX: An Open-Source Deep Learning Library. https://github.com/tensorlayer/tensorlayerx. Accessed: 2024-09-12.

[55] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* (2016). doi:10.48550/arXiv.1605.02688

[56] Veronika Thost and Jie Chen. 2021. Directed acyclic graph neural networks. *arXiv preprint arXiv:2101.07965* (2021). doi:10.48550/arXiv.2101.07965

[57] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering* 13, 2 (2011), 22–30. doi:10.1109/MCSE.2011.37

[58] Mohammad Wardat, Breno Dantas Cruz, Wei Le, and Hridesh Rajan. 2022. DeepDiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs. In *Proceedings of the 44th international conference on software engineering*. 561–572. doi:10.1145/3510003.3510071

[59] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007. doi:10.1145/3510003.3510041

[60] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[61] Xiaoyan Xie, Wanqi He, Yun Zhu, and Hao Xu. 2022. Performance evaluation and analysis of deep learning frameworks. In *Proceedings of the 2022 5th International Conference on Artificial Intelligence and Pattern Recognition*. 38–44. doi:10.1145/3573942.3573948

[62] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing automatic differentiation in deep-learning libraries. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1174–1186. doi:10.1109/ICSE48619.2023.00105

[63] Zihan Yu, Liang He, Zhen Wu, Xinyu Dai, and Jiajun Chen. 2023. Towards better chain-of-thought prompting strategies: A survey. *arXiv preprint arXiv:2310.04959* (2023). doi:10.48550/arXiv.2310.04959

[64] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140. doi:10.1145/3213846.3213866

Received 2024-09-11; accepted 2025-04-01