Learning Bug Context for PyTorch-to-JAX Translation with LLMs

Anonymous Authors[1]

Abstract

Large language models (LLMs) have shown strong performance on code translation between widely used programming languages. However, translation becomes much less reliable for domain-specific code, where correctness depends on framework-specific APIs and execution semantics. One example is translating deep-learning code from PyTorch to JAX, where LLM outputs often contain subtle bugs or non-idiomatic usage that prevents execution or changes behavior. Prior work suggests that curated bug–fix data from LLM-generated code can help improve code generation quality, but such resources are still limited for PyTorch-to-JAX translation.

In this work, we introduce **T2J**, a benchmark of LLM translation bugs paired with developer-written fixes for PyTorch-to-JAX code. We start from 20 kernels in the TorchLeet dataset, translate them to JAX using the weak LLM gpt-4o-mini, and hire software developers to debug and repair the generated JAX implementations. In total, **T2J** contains over 160 bug–solution pairs collected from real LLM outputs. We then use **T2J** to improve PyTorch-to-JAX translation for the weak LLM gpt-4o-mini via in-context learning. Our evaluation shows that using **T2J** yields up to 20% improvement of our proposed metric **T2J**$_CodeTrans_Score$. We release our replication package at here[1]

1. Introduction

Code translation involves converting a program from one programming language to another while preserving the original functionality. This process is useful for cross-language and cross-domain migration, allowing organizations to transition their code base to more modern languages or to various purposes. It also supports the modernization of legacy systems by re-implementing them in languages that promote greater maintainability and scalability as part of system refactoring efforts . Furthermore, in enterprises that employ multiple programming languages, code translation enhances the productivity of the programmer.

However, recent research work (Pan et al., 2024), (Dou et al., 2024) indicates that LLMs-generated programs in the target language continue to encounter various quality problems, including compilation errors or functional inconsistencies. In response to these issues, Yang et al (Yang et al., 2024) developed UniTrans, which augments LLM-based code translation through an iterative correction mechanism. In particular, UniTrans utilizes LLMs to refine the translated program by incorporating test inputs and outputs or compilation error feedback. While demonstrating potential, UniTrans nonetheless fails to rectify the translated program in a considerable number of instances, particularly when relying on smaller LLMs (those with fewer than 10 billion parameters). For example, when integrated with LLaMA-7B, UniTrans only elevates translation accuracy from 31.25% to 31.82% (representing a mere 0.57% improvement) for translations from Java to Python; and for translations from Python to C++ it boasts a translation accuracy improvement from 92.72% to 94.43% (barely 1.70% leap).

These limitations become more profound in specific situations, such as translating PyTorch code/programs to JAX. This is because of core differences in dynamic graph execution, just-in-time (JIT) compilation, automatic differentiation approaches, and vectorization flows. During PyTorch-to-JAX migrations, LLMs often struggle to handle tensor operations, gradient calculations, or device-free optimizations. This results in non-functional or even incorrect code that strays from the original intent of code.

Our contribution. To address these challenges, we introduce T2J, a prompt augmentation framework designed to enhance LLM-based PyTorch-to-JAX code translation by leveraging curated datasets and structured prompting strategies. The framework proceeds in several key stages: first, we construct parallel corpora from established PyTorch datasets, in particular TorchLeet and CodeParrot. Then we employ

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

[1]https://tinyurl.com/4ehky8hb

high-quality GPT models such as GPT-4o to produce initial JAX translations. Subsequently, professional human developers iteratively refine the translated JAX program to achieve functional equivalence with the original PyTorch input, producing a curated fixed-bug dataset that systematically documents prevalent translation errors—such as API mismatches, gradient computation discrepancies, and optimization failures—and their corresponding resolutions. The dataset also includes error-by-error fix instruction, and also. Building on this, we design augmented prompts that integrate targeted, structured guidance derived from the fixed-bug dataset. Finally, we evaluate T2J performance across both datasets using the CodeBLEU metric, alongside three novel metrics—Fixing Cost (quantifying the effort required for post-translation corrections), CodeTran_Score (leveraging LLM-as-a-judges for evaluating code usefulness and functional correctness), and Comparison-Score (leveraging LLM-as-a-judges for binary judgement)—to provide a comprehensive assessment of translation quality. We contribute (1) the creation of the first fixed-bug dataset specifically for PyTorch-to-JAX translations, encompassing detailed annotations of error patterns and fixes to facilitate improvements in reliability on LLM code translation; (2) the T2J framework, which innovates prompt augmentation techniques to bridge domain-specific gaps, carving path for a scalable approach for other cross-ecosystem migrations; (3) rigorous empirical validation showing substantial performance enhancements of our framework.

The remainder of this paper is structured as follows. Section 2, Motivation Example, presents a case study demonstrating the use of LLMs for PyTorch-to-JAX translation. Section 3, Background, introduces key concepts relevant to our work. Section 4 describes the core modules of the T2J framework. Section 5 discusses the central research questions and presents evaluation results. Finally, we summarise related work, limitations and provide future directions in the last three sections.

2. Motivation Example

An illustration of how low-cost LLM as 4o-mini generate JAX buggy code from Pytorch code can be shown in Figure 1. The input Pytorch program, which is a simple neural network, defines a single-layer feedforward network that flattens a 28×28 input image and applies a linear transformation to produce a 10-dimensional output tensor. The JAX code generated by 4o-mini has an extra function called $init_params$, while in Pytorch the corresponding function was performed in the construction of the Neural Network object. We can see that the generated code consumed an error in Line 8 (see Figure 1b)), where it incorrectly passed the shape of a linear object defined by the parameter W. The correct version of the generated code, shown in Figure 1, requires one modification step to correct the argument passed onto the $random.normal$ JAX API call. This example shows specifc challenge of Pytorch-to-JAX translation that even with this simple code snippet, low-cost LLM still failed to extract the correct code.

3. Background

Direction of Prompt Augmentation. Compared to open-source LLMs, closed-source LLMs developed by large organizations generally demonstrate higher accuracy in question/answering tasks across multiple domains (Hanke et al., 2024; Du et al., 2024). In the context of code translation, existing studies have shown that closed LLMs outperform open LLMs across a variety of programming languages (Yan et al., 2023). To the best of our knowledge, the most advanced and widely adopted closed LLMs are the OpenAI's GPT models (OpenAI, 2025). A major challenge in optimizing GPT models for code translation lies in the high computational cost of traditional machine learning techniques such as pre-training and fine-tuning. As a result, prompt augmentation has emerged as a practical and cost-effective alternative to improve closed LLM performance. One recent work in this direction is UniTrans (Yang et al., 2024), which introduces a framework for prompt augmentation using LLM-generated test cases, specifically targeting translations among widely used languages such as Python, C++, and Java. In our work, we focus on translating Python code between machine learning frameworks, specifically from PyTorch to JAX. We hypothesize that existing LLM-generated test cases may be less reliable in this setting due to the niche nature of such translations. To address this, we propose a mechanism for augmenting prompts with common error patterns and their corresponding workaround solutions, tailored to the PyTorch to JAX translation task. Supporting our motivation, Pan et al. (Pan et al., 2024) presents evidence that translated code across major programming languages often contains diverse types of errors. In this paper, we call the direction that we augment the prompt for Pytorch-to-JAX code translation as Fixed-bugs Prompt Augmentation.

Optimizing Metrics for Code Translation Evaluation. Traditional metrics for evaluating translated code often rely on the availability of a reference or "oracle" code. In our study, we use CodeBLEU (Ren et al., 2020) as a representative traditional metric to assess the quality of PyTorch-to-JAX translations. With the rise of Large Language Models (LLMs), however, these models are increasingly utilized not only for generating code translations but also for evaluating them. To the best of our knowledge, LLM-based evaluation offers two key advantages. First, such methods can function with or without reference code. In these cases, LLMs can generate synthetic labels to serve as pseudo ground

(a) Input PyTorch Code

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

# Define model
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc = nn.Linear(28 * 28, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Flatten
        return self.fc(x)

# Example
model = SimpleNN()
input_tensor = torch.randn(1, 1, 28, 28)
output = model(input_tensor)
print(output)
```

(b) Incorrect JAX code

```python
import jax
import jax.numpy as jnp
from jax import random

# Initialize parameters
def init_params(key):
    w_key, b_key = random.split(key)
    # Wrong shape
    W = random.normal(w_key, @(10, 28*28)@) * jnp.sqrt(1 / (28*28))
    b = jnp.zeros((10,))
    return {'W': W, 'b': b}

# Forward function
def forward(params, x):
    x = x.reshape(x.shape[0], -1)
    return jnp.dot(x, params['W']) + params['b']

# Example
key = random.PRNGKey(0)
params = init_params(key)
input_tensor = random.normal(key, (1, 1, 28, 28))
output = forward(params, input_tensor)
print(output)
```

(c) Correct JAX Code

```python
import jax
import jax.numpy as jnp
from jax import random

# Initialize parameters
def init_params(key):
    w_key, b_key = random.split(key)
    # Correct shape
    W = random.normal(w_key, ^(28*28, 10)^) * jnp.sqrt(1 / (28*28))
    b = jnp.zeros((10,))
    return {'W': W, 'b': b}

# Forward function
def forward(params, x):
    x = x.reshape(x.shape[0], -1)
    return jnp.dot(x, params['W']) + params['b']

# Example
key = random.PRNGKey(0)
params = init_params(key)
input_tensor = random.normal(key, (1, 1, 28, 28))
output = forward(params, input_tensor)
print(output)
```

Figure 1. Example of PyTorch-to-JAX translation. (a) Input code; (b) Incorrect translation by 4o-mini; (c) Correct code.

truths for comparing translated outputs. Second, LLMs are capable of assessing multiple dimensions of code quality, including functional correctness and readability (Weyssow et al., 2024). One of the most recent LLM-based evaluation techniques is ICE-score (Zhuo, 2024), proposed by Zhuo et al., which targets natural language-to-code generation. In this framework, the authors define evaluation criteria and a scoring rubric to evaluate Python code based on two dimensions: functional correctness and usefulness. They employ GPT models, guided by specially designed prompts and rubrics, to assign scores ranging from 1 to 5. Building on ICE-score, we propose two optimization strategies tailored to code-to-code translation. First, we design customized prompts and scoring rubrics specifically for PyTorch-to-JAX translation, which we term **T2J_CodeTrans_Score**. Second, we develop a separate set of prompts—agnostic to specific coding preferences—to enable direct comparison between two translated code snippets. The LLMs are asked to judge which translation is superior. We refer to these as **Code Comparison Prompts**.

4. Approach

We depict modules of **T2J** in Figure 2. In the first module, we describe how we hire software developers to check and modify JAX-generated code to ensure it is correct. We also define the definition of correct generated code in our work. The second module introduces our proposed metrics for comparing between predicted and expected generated code, in terms of using LLMs and leveraging the human bug fixing process's output. In the third module, we go in to details about the fixed bug dataset, which we will use for in-context learning with data about fixed bugs as extra context. In the fourth and fifth sections, we describe how **T2J** performs translation and how we generate the ground-truth dataset for evaluation. We also describe what baseline setting we use for comparison with our proposed pipeline. We call these modules intrinsic evaluation and extrinsic evaluation, and

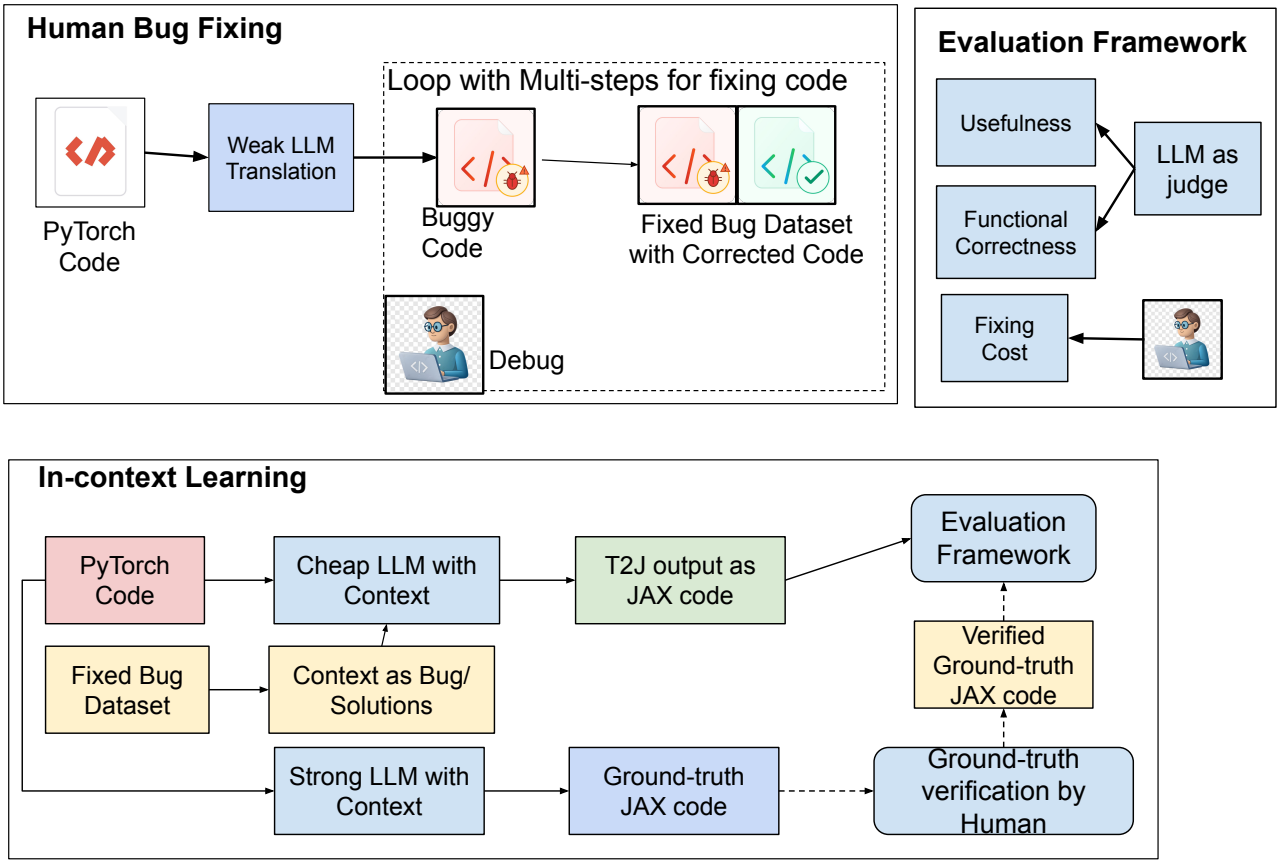


Figure 2. Overview Architecture of **T2J** . In Module One, a multi-step bug fixing process for the problem-solving dataset's code snippet was performed. In Module Two, Evaluation Framework, we proposed metrics for this domain code translation evaluation. In module Three, we measure the accuracy of translation with different settings using an intrinsic dataset of 20 problem-solving code snippets and an extrinsic dataset of 100 randomly selected PyTorch code snippets selected from Github corpus.

these modules are performed on two different datasets. Next, we discuss about important concepts and design selection we use for **T2J** .

4.1. Design Selection

4.1.1. Selection of PyTorch Dataset

We curate datasets of PyTorch code for two tasks. First, we create a dataset of fixed bugs as JAX-generated code from PyTorch input code for our proposed in-context learning process. This process requires the involvement of software developers and LLM for PyTorch-to-JAX code translation. The second task is the evaluation process, where the PyTorch code will be used as input for baseline models or our proposed translation framework to get the output as JAX code snippets for further processing to correct the code and evaluation. Depending on the tasks, we collect the PyTorch dataset from two domains: problem-solving code and general-purpose code.

Problem-Solving Code Dataset. We construct a PyTorch dataset based on popular coding interview problems implemented in PyTorch. Owing to the popularity of these problems, we assume that developers can readily verify solutions and fix bugs by consulting existing online resources. For this purpose, we leverage the TorchLeet dataset (Aroori & Chien, 2025) as our problem-solving code corpus. We chose this dataset because each problem is scoped at the file level and because TorchLeet has been highly ranked by GitHub users. From this dataset, we collect all **20** code snippets, covering three difficulty levels: easy, medium, and hard. All snippets are compilable and runnable using the default test cases included within the dataset.

GitHub Code Dataset. To further evaluate our proposed PyTorch-to-JAX translation approach on a broader range of PyTorch code, we consider code snippets drawn from high-quality repositories on GitHub. Specifically, we use the PyTorch subset of the large-scale CodeParrot dataset (Wolf et al., 2022) as our second evaluation corpus. From this dataset, we extract **100** PyTorch code snippets. Unlike the Problem-Solving Code dataset, the GitHub PyTorch

snippets originate from diverse repositories and are not guaranteed to be directly compilable.

4.1.2. Selection of LLMs

Cheap (Weak) LLMs. We define low-cost LLMs as the target models to improve in the PyTorch-to-JAX translation task. These models, referred to as *Cheap LLMs*, can be used without commercial API keys or additional costs. Moreover, we leverage their JAX-generated code to construct the fixed-bug dataset, under the assumption that cheap LLMs produce a higher proportion of buggy code, which can enrich this dataset. For our experiments, we select GPT-4o-mini, the least expensive model offered by OpenAI, as the representative cheap LLM.

Costly (Strong) LLMs. We define high-cost LLMs, referred to as *Costly LLMs*, as the source of ground-truth JAX code for comparison with the translations produced by cheap LLMs. We employ the GPT-4o model as the costly LLM. GPT-4o is one of the most widely used models provided by OpenAI.

4.1.3. Prompts

We define these following types of prompt in our work.

Standard Prompt. We define the standard prompt as the basic prompt for translating code snippet from Pytorch to JAX (see Appendix A.1). In this prompt, we define the translation request by annotating the role of LLM as an expert in code translation and provide a basic request that translate from source language as PyTorch to target language as JAX code.

Augmented Prompt. We design augmented prompt with following information. First, similar to the standard prompt, the augmented prompt will have information about the role of LLM and the requirement about input and output. Differ from standard prompt, the augmented prompt will specify a hint for LLMs as the list of errors and errors' solution provided by the fixed bug dataset, uploaded in the JSON format as context of prompt. Definitions of each fields in the fixed bug dataset are also included in the prompt content (see Appendix A.2).

Evaluation Prompt. We propose two metrics that leverage LLMs for code-to-code translation. From our knowledge, there are yet existing LLM prompt for doing this task. We define a set of prompts, called evaluation prompts (see Appendix A.3), to ask costly LLMs to evaluate source code in different aspects. We will describe in details of these metrics in the next section.

4.1.4. Types of Evaluation

Intrinsic and Extrinsic Evaluation. Our evaluation is divided into two parts. The first, referred to as intrinsic evaluation, is conducted on 20 problem-solving PyTorch code snippets using a cross-validation setting. The second, extrinsic evaluation, is performed on 100 samples of PyTorch code collected from GitHub. There are two key differences between these configurations. First, for intrinsic evaluation, the ground-truth JAX code is obtained through human verification and bug-fixing, whereas for extrinsic evaluation, we rely on JAX code generated by a costly LLM for the GitHub dataset.

4.2. Human Bug Fixing Process

The core module of **T2J** is a systematic bug-fixing process designed to improve the reliability of LLM-generated translations from PyTorch to JAX. Starting with Python code written in PyTorch and the corresponding JAX-generated code from LLMs, this process produces a parallel dataset that pairs the original PyTorch snippets with their corresponding corrected JAX translations. To ensure correctness, we employed two professional software developers with over five years of Python programming experience to analyze and fix the outputs produced by LLMs. The JAX translations are then subjected to careful manual verification: the developers perform multiple rounds of debugging and correction until the translated JAX code passes all test cases and produces results equivalent to the original PyTorch implementation. During this stage, the verifiers execute the translated and fixed code using Python compilers to confirm correctness. At the end of the process, two complementary datasets are created: one containing pairs of PyTorch snippets and their validated JAX counterparts, and another capturing the bugs identified in LLM outputs along with their corresponding fixes. Importantly, this methodology is flexible, as we apply the same procedure to different PyTorch datasets and experiment with different LLMs depending on the objectives of other modules within our framework.

Verified Code Correctness by Human. We consider a fixed version as JAX code snippet verified by a human as correct if and only if the JAX-generated code can be compiled, runnable, and returns the same output as its corresponding Pytorch code snippet, given the same test case. In our work, the human bug fixing process was performed in the problem-solving dataset only since its code snippets have test case and can be runnable which we can rely on their execution output for comparison with the JAX generated code.

Fixed Bug Dataset. Given the input of 20 PyTorch samples from the TorchLeet dataset, two professional developers are hired to modify the code snippets generated from the LLM code translation process. The output of this process for each generated code is a set of multiple fix steps. We store

Table 1. Evaluation preferences and their descriptions for **T2J**_CodeTrans_Score.

Preference	Description
Usefulness	How useful the JAX code is for replicating or adapting the functionality of a typical PyTorch source code implementation.
Functional Correctness	How well the JAX code preserves the behavior of the original PyTorch code. You are to assess whether the JAX code would produce equivalent outputs to the original PyTorch code across possible inputs, even though the PyTorch code is not shown. Consider unit-test-style logic and general expectations of equivalence.

this data in JSON array format to be usable as the context for **T2J** prompting technique. The fixed bug dataset was constructed by our selected cheap LLM 4o-mini. In total, this dataset contains **163** pairs of bugs/ solutions to fix bug.

4.3. Evaluation Framework

We use CodeBLEU (Ren et al., 2020), a well-known code evaluation metric, as the baseline metric for PyTorch-to-JAX translation. Since CodeBLEU relies on the AST similarity between two code snippets, we assume that it cannot be a good metric for PyTorch-to-JAX translation output comparison, since both the JAX-generated code might be very similar to the ground truth code, such as in Motivation Example 1. Recently, an LLM-based code evaluation metric has been proposed (Zhuo, 2024). In this work, the authors proposed the ICE-score, a metric for natural language to code translation based on usefulness and functional correctness. In our work, we inherit the idea of the ICE-score and leverage our fixing process to propose three metrics for evaluating JAX-generated code. Given the n pairs of Pytorch and JAX codes in the corresponding Pytorch code set P, the predicted code set JAX J^p, the ground truth (reference) code set JAX J^r as (p_i,j_i), our proposing metrics will be provided as following.

T2J_CodeTrans_Score. Similar to the ICE-Score metric (Zhuo, 2024) for natural language to code translation, we use the GPT-4o model to evaluate the quality of translated code by usefulness and functional correctness. We design prompt, called CodeTrans prompt, with corresponding evaluation criteria and scoring rubric from 0 (lowest) to 4 (highest). CodeTrans also be useable without having the reference code. Thus, we have a set of following metrics: **T2J**_CodeTrans_Use_Ref (i.e. the metric for usefulness with reference), **T2J**_CodeTrans_Func_Ref, **T2J**_CodeTrans_Use_NoRef, **T2J**_CodeTrans_Func_NoRef. Explanation of two aspects/ preferences is shown in Table 1.

T2J_FixCost_Score. We measure the number of fix steps required to have the JAX correct code from input JAX initial translated code from LLM with this equation:

$$\textbf{T2J}_FixCost_Score(J^{before_fix}, J^{correct}) = \frac{1}{n}\sum_{i=1}^{n} count_fix_step(j_i^{before_fix}, j_i^{correct}) \quad (1)$$

In equation 1, J^{before_fix} is the set of JAX generated code as input for human verification process, while $J^{correct}$ is the final version of JAX fixed code that is correct, i.e. it can run and returns consistent output with its corresponding PyTorch code given the same input test case.

T2J_Comparison_Score. In this metric, we propose a direct comparison between two translation sets J^1 and J^2 to see which one is closer to the input PyTorch code set P, implemented by this equation:

$$\textbf{T2J}_Comparison_Score(J^1, P, J^2) = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} 1, & \text{if } \texttt{is_better}(j_i^1, j_i^2, p_i) \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

In equation 2, the $\texttt{is_better}(j_i^1, j_i^2, p_i)$ function returns 1 if j_i^1 is considered as better code than j_i^2, given the input prompt for comparison called Comparison prompt (see Appendix A.3). Note, we also include the content of PyTorch code p_i as a required input to help LLM comparing the input with each code candidate. The scale of this score is from 0 to 1.

Code generation as Baseline. We perform the code translation process using the cheap LLM 4o-mini. This process leverages the standard prompt (see Appendix A.1) to translate input PyTorch code to JAX-generated code. We consider this configuration as baseline for **T2J** for comparison.

T2J 's In-context learning for code generation. We perform this process with following modules. First, the augmented prompt will be created with input information as the given Pytorch code snippet and the JSON format of other bug-solutions from other code samples of the problem-solving code dataset. Next, through the cheap LLM, the JAX generated code by **T2J** 's approach is created. This process was done without the need of any fine-tuning steps, which are usually costly.

Ground-truth JAX code generation. We take advantage of costly LLM gpt-4o to generate the JAX code, called JAX initial code from given problem-solving code snippet. Next, the human bug fixing process was performed on this JAX initial code to make the JAX corrected code as ground truth. The main different between this process and the fixed bug dataset creation we mentioned earlier is that this module works with costly LLM. Finally, the output of these three

modules will be use as the input of evaluation framework along with the PyTorch source code.

4.4. Translation with different settings

Intrinsic Setting. In the intrinsic evaluation setting, we want to check if we can use knowledge from fixing 19 problem-solving code pairs of Pytorch and JAX correct code to improve the quality of cheap LLM to translate Pytorch code of the remaining sample. We perform this evaluation as a cross validation process over 20 samples of the fixed bug dataset. Next, we compare the output of three following modules.

Extrinsic Setting. There are two main differences between intrinsic evaluation and extrinsic evaluation. First, for this set up we use the code snippets collected from another dataset, called Github dataset. We evaluate on 100 sample code snippets that are at repository level, meaning that the input code snippets, usually collected from single files, are not guarantee to be runnable and having test cases. Thus, we will use automated metrics we propose for the evaluation. Second, the extrinsic evaluation considered the output of costly LLMs, i.e. JAX translated code by this process) as the ground-truth data point.

5. Experiment

5.1. Setup

Hardware Configuration. For the human bug fixing process, software developers work in the Google Colab environment to debug and fix the code. They use Python 3 as a compiler and use one T4 GPU for running all sample code.

Question-answering for cheap and costly LLMs. For both cheap LLM (4o-mini) and costly LLM(gpt-4o), we perform the code generation process through the official interface of ChatGPT-pro. Each question will be created solely in a new topic, and from the answer given by ChatGPT's interface, we manually extract the code snippet as JAX-generated code. Other textual explanation in the output will be omitted. To add context to the existing prompt, we use the Upload function provided by ChatGPT's interface to assign a fixed bug dataset as a JSON file to our designed prompt.

Executing Evaluation Prompts. We leverage the access on OpenAI gpt-4o models by API key to get the scoring results for **T2J**_CodeTrans_Score. For **T2J**_Comparison_Score, we use ChatGPT-pro's interface to ask questions and receive answers. The reason for using ChatGPT-pro's interface instead of using API key is that some tasks of our work require JSON file upload.

Table 2. Results with Cheap LLM (gpt-4o-mini) and Costly LLM (gpt-4o).

Metrics	Intrinsic Evaluation		Extrinsic Evaluation	
	Baseline	T2J	Baseline	T2J
CodeBLEU	0.19	0.29	0.41	0.38
T2J_CodeTrans_Use._Ref	1.75	2.55	2.74	2.94
T2J_CodeTrans_Func._Ref	0.35	1.3	2.43	3.02
T2J_CodeTrans_Use._NoRef	1.60	2.45	2.81	2.98
T2J_CodeTrans_Func._NoRef	0.70	2.15	2.74	3.37
T2J_FixCost_Score	163	87	N/A	N/A
T2J_Comparison_Score	0	1	0.18	0.82

Table 3. Correlation of other metrics with **T2J**_FixCost_Score on Fixed Bug Dataset.

Metric	Correlation with T2J _FixCost_Score	
	Pearson	Spearman
CodeBLEU	0.04	0.2
T2J_CodeTrans_Use._Ref	0.2	0.25
T2J_CodeTrans_Func._Ref	0.07	0.07
T2J_CodeTrans_Use._NoRef	0.09	0.19
T2J_CodeTrans_Func._NoRef	0.11	0.29

5.2. Result

5.2.1. Translation Accuracy

From Figure 2, our pipeline improves the CodeBLEU score to 0.29, representing a 10% relative gain. In terms of **T2J**_CodeTrans_Use_Ref, the augmented prompt yields an improvement of 0.8 points over the baseline. For functional correctness, **T2J** achieves an improvement of 0.95 point. Under the no-reference configuration—where the LLM evaluates only by comparing the input and translated code—**T2J** still delivers gains of 0.85 and 1.15 points for usefulness and functional correctness, respectively, as measured by the **T2J**_CodeTrans metrics. Regarding the **T2J**_Comparison_Score, 100% of the translations generated by **T2J** are judged superior to the baseline outputs. Finally, in terms of fixing cost, **T2J** enables GPT-4o-mini to produce code requiring only 87 fixing steps—roughly half the effort compared to fixing the baseline JAX outputs.

The extrinsic evaluation on the GitHub PyTorch code also highlights the superiority of **T2J** over the baseline in generating precise code. Interestingly, in this setting the baseline approach outperformed **T2J** by 3% according to CodeBLEU. In terms of the **T2J**_CodeTrans metrics, our approach achieves improvements of up to 1.2 point in usefulness and 0.6 point in functional correctness.

Table 4. Comparison of running time (seconds) on Intrinsic Evaluation.

PyTorch	Ground Truth	Baseline	T2J
1003	851	1232.9	449

Table 5. Comparison of human fixing costs between baseline (cheap LLM with standard prompt) and JAX code initially generated by a costly LLM.

Num. of Fixes	Correcting Cheap LLM's Code	Correcting Costly LLM's Code
Minimum	1	0
Maximum	32	12
Mean	8.15	2.77
Median	5	2
Total	163	61

5.2.2. Correlation of Code Translation Metrics vs Human Fixing Cost

From Table 3, we observe that the **T2J** _CodeTrans_Score metrics show the strongest correlation with **T2J** _FixCost_Score under both Pearson and Spearman measures. . Overall, all metrics exhibit weak correlation (below 0.3) with fixing cost. One possible reason is that other metrics are continuous, whereas fixing cost is measured as discrete steps.

5.2.3. How close JAX-generated code by costly LLM is to being correct

We further analyze the quality of JAX generated code by costly LLM by meassuring the fixing cost to get the JAX correct code by costly LLM as ground-truth code. The result, shows in Table 5, shows that while it requires much less effort for correcting costly LLM's output than baseline's output, it still requires in total 61 fixing steps to get the correct code set.

5.2.4. Analysis on Running Time

We analyze the running time of the corrected code from 3 settings for intrinsic evaluation in Table 4. Results show that **T2J**can provide significant improvement as 2.5 times faster than running the baseline's output. Details of running time can be seen in Appendix B.

6. Related Work

Challenges of Code Translation. One of the important challenges of code translation is to provide a metric for comparison between predicted and expected code. Research works show that just simply comparing source code by traditional textual similarity scores is not efficient (Tran et al., 2019). Instead, code metrics that included information of syntactic/ semantic similarity between code snippets have been proposed (Zhou et al., 2023). Another challenge is that collecting parallel corpus for code translation is very expensive and require human effort for verification (Husain et al., 2020). To evaluate unsupervised code translation's output, automated test cases generation approaches have been proposed (Roziere et al., 2022; Peng et al., 2024). Finally, studies show that there are many types of bugs extracted from LLMs' generated code (Dinh et al., 2023; Zhang et al., 2024).

Machine Learning-based Approaches for Code Translation. Supervised methods for code translation are typically trained on well-established datasets such as CodeXGLUE (Lu et al., 2021). Among these methods, BERT-based models have proven particularly effective not only for code-to-code translation but also for a wide range of code generation tasks (Guo et al., 2021; 2022; Wang et al., 2023; Ahmad et al., 2021). Unsupervised code translation typically relies on transforming the source code into an intermediate representation (IR), followed by learning to generate target language code from that intermediate form. (Szafraniec et al., 2023) proposed *Transcoder-IR*, a system that uses IR as a pivot language to translate between widely-used languages such as Java, Python, and C++. (Huang et al., 2023) proposed *Codist*, which adopts a filtered IR to improve the precision of code translation through a process called code distillation. (TehraniJamsaz et al., 2024) presented *CodeRosetta*, a framework for unsupervised translation from C to CUDA. Their method exploits the syntactic similarity between the two languages, leveraging abstract syntax trees (ASTs) as the pivot representation to learn structural correspondences. (Roziere et al., 2021) emphasized the significance of a pre-training objective based on recovering broken or obfuscated code.

7. Limitations

First, **T2J** has not yet been applied to improving open LLMs, due to budget constraints that limit our ability to hire software professionals for the human bug-fixing process on these models. Second, our measure of fixing cost is currently based only on counting fixing steps, whereas in practice, each fix may vary in difficulty. There is a need of an algorithm that estimates the relative effort of each bug-fixing step. Third, we conducted the human bug-fixing process only on the problem-solving code dataset, which we want to extend this process for other domains.

8. Conclusion and Future Works

In this work, we show that **T2J** can achieve significant improvement compared to baselines as original 4o-mini model for PyTorch-to-JAX code translation. In future work, we attempt to apply our approach for newer open-source LLMs and leverage more advance techniques as supervised fine-tuning and direct preference optimization.

References

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation, 2021. URL https://arxiv.org/abs/2103.06333.

Aroori, C. and Chien, C. Torchleet: Leetcode for pytorch. https://github.com/Exorust/TorchLeet, 2025. Accessed: YYYY-MM-DD.

Dinh, T., Zhao, J., Tan, S., Negrinho, R., Lausen, L., Zha, S., and Karypis, G. Large language models of code fail at completing code with potential bugs, 2023. URL https://arxiv.org/abs/2306.03438.

Dou, S., Jia, H., Wu, S., Zheng, H., Zhou, W., Wu, M., Chai, M., Fan, J., Huang, C., Tao, Y., Liu, Y., Zhou, E., Zhang, M., Zhou, Y., Wu, Y., Zheng, R., Wen, M., Weng, R., Wang, J., Cai, X., Gui, T., Qiu, X., Zhang, Q., and Huang, X. What's wrong with your code generated by large language models? an extensive study, 2024. URL https://arxiv.org/abs/2407.06153.

Du, J., Wang, Y., Zhao, W., Deng, Z., Liu, S., Lou, R., Zou, H. P., Narayanan Venkit, P., Zhang, N., Srinath, M., Zhang, H. R., Gupta, V., Li, Y., Li, T., Wang, F., Liu, Q., Liu, T., Gao, P., Xia, C., Xing, C., Jiayang, C., Wang, Z., Su, Y., Shah, R. S., Guo, R., Gu, J., Li, H., Wei, K., Wang, Z., Cheng, L., Ranathunga, S., Fang, M., Fu, J., Liu, F., Huang, R., Blanco, E., Cao, Y., Zhang, R., Yu, P. S., and Yin, W. LLMs assist NLP researchers: Critique paper (meta-)reviewing. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 5081–5099, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.292. URL https://aclanthology.org/2024.emnlp-main.292/.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., and Zhou, M. Graphcodebert: Pre-training code representations with data flow, 2021. URL https://arxiv.org/abs/2009.08366.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. Unixcoder: Unified cross-modal pre-training for code representation, 2022. URL https://arxiv.org/abs/2203.03850.

Hanke, V., Blanchard, T., Boenisch, F., Olatunji, I. E., Backes, M., and Dziedzic, A. Open LLMs are necessary for current private adaptations and outperform their closed alternatives. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=Jf40H5pRW0.

Huang, Y., Qi, M., Yao, Y., Wang, M., Gu, B., Clement, C., and Sundaresan, N. Program translation via code distillation, 2023. URL https://arxiv.org/abs/2310.11476.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search, 2020. URL https://arxiv.org/abs/1909.09436.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021. URL https://arxiv.org/abs/2102.04664.

OpenAI. OpenAI. Web page, 2025. Accessed July 17, 2025, from https://openai.com.

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., and Jabbarvand, R. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639226. URL https://doi.org/10.1145/3597503.3639226.

Peng, Q., Chai, Y., and Li, X. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization, 2024. URL https://arxiv.org/abs/2402.16694.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. Codebleu: a method for automatic evaluation of code synthesis, 2020. URL https://arxiv.org/abs/2009.10297.

Roziere, B., Lachaux, M.-A., Szafraniec, M., and Lample, G. Dobf: A deobfuscation pre-training objective for programming languages, 2021. URL https://arxiv.org/abs/2102.07492.

Roziere, B., Zhang, J. M., Charton, F., Harman, M., Synnaeve, G., and Lample, G. Leveraging automated unit tests for unsupervised code translation, 2022. URL https://arxiv.org/abs/2110.06773.

Szafraniec, M., Roziere, B., Leather, H., Charton, F., Labatut, P., and Synnaeve, G. Code translation with compiler representations, 2023. URL https://arxiv.org/abs/2207.03578.

TehraniJamsaz, A., Bhattacharjee, A., Chen, L., Ahmed, N. K., Yazdanbakhsh, A., and Jannesari, A. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=V6hrg4O9gg.

Tran, N., Tran, H., Nguyen, S., Nguyen, H., and Nguyen, T. N. Does bleu score work for code migration? In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pp. 165–176. IEEE Press, 2019. doi: 10.1109/ICPC.2019.00034. URL https://doi.org/10.1109/ICPC.2019.00034.

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., and Hoi, S. C. H. Codet5+: Open code large language models for code understanding and generation, 2023. URL https://arxiv.org/abs/2305.07922.

Weyssow, M., Kamanda, A., Zhou, X., and Sahraoui, H. Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences, 2024. URL https://arxiv.org/abs/2403.09032.

Wolf, T., Werra, L., Allal, L., and Zdar. Codeparrot dataset. https://huggingface.co/codeparrot, 2022. Accessed: YYYY-MM-DD.

Yan, W., Tian, Y., Li, Y., Chen, Q., and Wang, W. Codetransocean: A comprehensive multilingual benchmark for code translation, 2023. URL https://arxiv.org/abs/2310.04951.

Yang, Z., Liu, F., Yu, Z., Keung, J. W., Li, J., Liu, S., Hong, Y., Ma, X., Jin, Z., and Li, G. Exploring and unleashing the power of large language models in automated code translation. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3660778. URL https://doi.org/10.1145/3660778.

Zhang, Q., Fang, C., Xie, Y., Ma, Y., Sun, W., Yang, Y., and Chen, Z. A systematic literature review on large language models for automated program repair, 2024. URL https://arxiv.org/abs/2405.01466.

Zhou, S., Alon, U., Agarwal, S., and Neubig, G. Codebertscore: Evaluating code generation with pretrained models of code, 2023. URL https://arxiv.org/abs/2302.05527.

Zhuo, T. Y. ICE-score: Instructing large language models to evaluate code. In Graham, Y. and Purver, M. (eds.), *Findings of the Association for Computational Linguistics: EACL 2024*, pp. 2232–2242, St. Julian's, Malta, March 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.findings-eacl.148/.

Appendix

A. Prompt Template

A.1. Standard Prompt for PyTorch-to-JAX translation

> You are an expert in programming language translation from PyTorch to JAX. In this task, I will give you input as PyTorch code. Please translate this input PyTorch code to JAX code:
> Input Source Code Snippet: {CODE}

Figure 3. Standard Prompt for PyTorch-to-JAX code translation. The prompt in blue shows the immediate query after the prompt. {CODE}is the starting string of the code.

A.2. Augmented Prompt for PyTorch-to-JAX translation

> You are an expert in programming language translation from PyTorch to JAX. In this task, I will give you two inputs:
> 1. Pytorch source code.
> 2. A JSON file that contains a dataset of common errors in PyTorch-to-JAX translation by Weak LLM 4o-mini. Each data point contains the following fields:
> - `Example_id`: ID of the source code.
> - `Input_Code`: Source code in Pytorch.
> - `LLM_weak_output`: JAX translated code of `Input_Code` using a weak LLM (4*o*-mini). `LLM_fix_output`: Fixed JAX code from `LLM_weak_output` by the process of manually check and fix errors conducted by software developers.
> - `Errors`: This is a list of errors that appeared in the process of manually checking and fixing bugs from `LLM_weak` . Each error item has the following labels:
> • `"Error_Code"`: The part of `LLM_weak_output` that caused the error.
> • `"Error"`: the error message returned by compilation.
> • `"Fix_info"`: the textual description of how to fix the error code
> • `"Fixed_Code"`: The fixed code corresponding to the `"Error_Code"` part.
> 3. The data.csv file thich stored possible input when running some examples in the JSON file. Your task is to reason and get the output JAX code from these above inputs. Please note that you can learn the process of error fixing in Torch-to-JAX translation in 2) JSON file. Now I will give you a set of input in the next query.
>
> Input Source Code Snippet: {CODE}

Figure 4. Prompt for Augmenting to the weak LLM. The prompt in blue shows the immediate query after the prompt. {CODE}is the starting string of the code.

A.3. Evaluation Prompt for T2J_CodeTrans_Score

A.3.1. Functional Correctness

See the prompt without reference at Figure 5 and the prompt with reference at Figure 6.

A.3.2. Usefulness

See the prompt without reference at Figure 7 and the prompt with reference at Figure 8.

A.4. Evaluation Prompt for T2J_Comparison_Score

The prompt for querying the **T2J**_Comparison_Score can be seen in Figure 9.

You will be given a JAX code snippet that was translated from PyTorch source code. Your task is to rate the snippet on **one metric only**: its **functional correctness**.
Please ensure you read and understand these instructions carefully before reviewing. Refer to this guide as needed during the evaluation process.
Evaluation Criteria:
Functional Correctness (0–4) — How well the JAX code preserves the behavior of the original PyTorch code.
You are to assess whether the JAX code would produce equivalent outputs to the original PyTorch code across possible inputs, even though the PyTorch code is not shown. Consider unit-test-style logic and general expectations of equivalence.
- A score of 0: The translation is completely incorrect and meaningless.
- A score of 4: The translation is fully correct and handles all core functionalities as expected.
Evaluation Steps:
1. Assume the code was translated from PyTorch and should preserve its logic.
2. Evaluate whether the JAX code appears complete, meaningful, and implementationally correct based on general expectations for such translations.
3. Assign a score for functional correctness on a scale from 0 to 4.
Input Source Code in PyTorch:
{SOURCE_CODE}
Translated JAX Code Snippet:
{TRANSLATED_CODE}
Evaluation Form:
Functional Correctness (scores ONLY):

Figure 5. Prompt for Scoring Functional Correctness by **T2J** _CodeTrans_Func_NoRef

B. Additional Results

B.1. Analysis on Categories of Bugs

We perform a study on the categorization of bugs on the fixed bug dataset as following. First, two software professionals will go through all the bugs and discuss about the categorizations. Second, from this categorization, they go to the dataset's entities for the second time and do the annotation for categories. We summarize the categorization in Table 6. We further classify types of bugs for some categories to sub-categories, shown in Table 7 and Table 8. We upload each case of this categorization process in the replication package.

B.2. Running time Analysis

We perform the running process on a T4 GPU for all the code. We set the timeout of program to run as 180 seconds. Results for each sample in the intrinsic evaluation are shown in Table 9.

C. Configurations

For the task that required user interface action with LLMs, we use the default ChatGPT-pro setting for gpt-4o and 4o-mini. Most of the data were created before July 31st, 2025 when 4o-mini was still available on ChatGPT's interface. For task like LLM-based metric calculation, we leverage OpenRouter's API[2] to perform the implementation of these tasks. We also report the hyper parameters for querying costly LLMs for code evaluation in the replication packages.

[2]https://openrouter.ai/

You will be given a JAX code snippet that was translated from PyTorch source code. Your task is to rate the snippet on **one metric only**: its **functional correctness**.
Please ensure you read and understand these instructions carefully before reviewing. Refer to this guide as needed during the evaluation process.
Evaluation Criteria: Functional Correctness (0–4) — How well the JAX code preserves the behavior of the original PyTorch code.
You are to assess whether the JAX code would produce equivalent outputs to the original PyTorch code across possible inputs, even though the PyTorch code is not shown. Consider unit-test-style logic and general expectations of equivalence.
- A score of 0: The translation is completely incorrect and meaningless.
- A score of 4: The translation is fully correct and handles all core functionalities as expected.
Evaluation Steps:
1. Assume the code was translated from PyTorch and should preserve its logic.
2. Evaluate whether the JAX code appears complete, meaningful, and implementationally correct based on general expectations for such translations.
3. Assign a score for functional correctness on a scale from 0 to 4.
Input Source Code in PyTorch:
{SOURCE_CODE}
Translated JAX Code Snippet:
{TRANSLATED_CODE}
Reference JAX Code Snippet:
{REFERENCE}
Evaluation Form:
Functional Correctness (scores ONLY):

Figure 6. Prompt for Scoring Functional Correctness by **T2J** _CodeTrans_Func_Ref

Your task is to rate the snippet on **one metric only**: its **usefulness** for understanding and reusing the logic of a typical PyTorch implementation.
Please ensure you read and understand these instructions carefully before reviewing. Refer to this guide as needed during the evaluation process.
Evaluation Criteria: Usefulness (0–4) — How useful the JAX code is for replicating or adapting the functionality of a typical PyTorch source code implementation.
- A score of 0: The JAX translated snippet is irrelevant or confusing and does not help at all.
- A score of 1: The JAX translated snippet includes some related elements but is mostly unhelpful.
- A score of 2: The JAX translated snippet is somewhat useful but needs substantial modification.
- A score of 3: The JAX translated snippet is helpful with minor revisions needed.
- A score of 4: The JAX translated snippet is very helpful and covers the intended functionality clearly.
Evaluation Steps:
1. Assume the PyTorch source code performs a well-defined functionality.
2. Determine whether the JAX translated code snippet enables meaningful reuse or guidance toward equivalent implementation.
3. Assign a score for usefulness from 0 to 4.
Input Source Code in PyTorch:
{SOURCE_CODE}
Translated JAX Code Snippet:
{TRANSLATED_CODE}
Evaluation Form:
Usefulness (scores ONLY):

Figure 7. Prompt for Scoring Usefulness by **T2J** _CodeTrans_Use_NoRef

Your task is to rate the snippet on **one metric only**: its **usefulness** for understanding and reusing the logic of a typical PyTorch implementation.
Please ensure you read and understand these instructions carefully before reviewing. Refer to this guide as needed during the evaluation process.
Evaluation Criteria: Usefulness (0–4) — How useful the JAX code is for replicating or adapting the functionality of a typical PyTorch source code implementation.
- A score of 0: The JAX translated snippet is irrelevant or confusing and does not help at all.
- A score of 1: The JAX translated snippet includes some related elements but is mostly unhelpful.
- A score of 2: The JAX translated snippet is somewhat useful but needs substantial modification.
- A score of 3: The JAX translated snippet is helpful with minor revisions needed.
- A score of 4: The JAX translated snippet is very helpful and covers the intended functionality clearly.
Evaluation Steps:
1. Assume the PyTorch source code performs a well-defined functionality.
2. Determine whether the JAX translated code snippet enables meaningful reuse or guidance toward equivalent implementation.
3. Assign a score for usefulness from 0 to 4.
Input Source Code in PyTorch:
{SOURCE_CODE}
Translated JAX Code Snippet:
{TRANSLATED_CODE}
Reference JAX Code Snippet:
{REFERENCE}
Evaluation Form:
Usefulness (scores ONLY):

Figure 8. Prompt for Scoring Usefulness by **T2J** _CodeTrans_Use_Ref

You are an expert in PyTorch to JAX translation. I provide 3 inputs: 1 . PyTorch input code; 2. Translated Code Candidate A; 3. Translated Code Candidate B. Which candidate is a better translation result for this Pytorch code.
Input Pytorch code:
{CODE}
2. Translated Code A:
{TRANSLATE_CODE_A}
3. Translated Code B:
{TRANSLATE_CODE_B}
Please also provide the reason why you consider a candidate better than the other translated code candidate.

Figure 9. Prompt for **T2J** _Comparison_Score.

Table 6. Error categories and their counts in Human Bug Fixing dataset.

Error Main Category	Count
Training loops, training steps, model fitting	32
Other miscellaneous	43
Model definitions, LinearModel classes, encoders/decoders	51
Loss functions, gradient computation, criterion	12
JAX-specific constructs: jit, grad, PRNG, etc.	18
Iteration patterns: for, while, data loops	1
Parameter updates	3
Final layers, return statements, outputs	3
TOTAL	163

Table 7. Error subcategories under **training loops, training steps, and model fitting**.

Error Subcategory	Count
Misc training issues	8
Improper passing/using `rng_key`/`prng_key`	6
Epoch in `range(...)` loop issues	3
Incorrect usage of Flax `TrainState` and `state.apply_gradients`	3
Incorrect usage of wrappers (e.g., `train_model(...)` / `fit(...)`)	3
Train steps return only new state/params without loss at epoch level	3
JIT/static argument handling for training functions	3
Errors with batches or dataloaders in training	1
Loop constructs that break vectorization	1
Optimizer update/apply patterns in the training loop	1

Table 8. Error subcategories under **other miscellaneous**.

Error Subcategory	Count
Data arrays, tensors, and dataset values (e.g., creating arrays, specifying shapes)	8
Dot products with parameters (e.g., `params["w"]`)	5
Initialization, often in class constructors (`__init__`)	1
Dot products, sums, or nonlinear transforms	3
Neural network layers and activations (e.g., `nn.relu`, `nn.Dense`, LSTMs, decoders/encoders)	5
Tensor dimension errors	2
Generating synthetic data for CSV	1
Errors with constant declaration (e.g., epoch)	1
Incomplete functions/placeholders	17

Table 9. Example results comparing PyTorch, ground truth, baseline, and **T2J** (T2J) outputs.

Example ID	PyTorch	Ground Truth	Baseline	T2J
e1	6.61	12.7	9.78	8.79
e2	10.00	60.0	24.4	3.59
e3	8.57	17.3	4.14	5.18
e4	8.82	22.1	21.8	5.36
e5	8.48	35.0	35.2	6.53
e6	13.0	24.0	3.0	13.68
e7	8.23	5.0	90.0	14.53
m1	17.0	26.0	180.0	15.62
m3	180.0	180.0	4.0	1.05
m4	180.0	18.0	180.0	29.0
m5	102.0	57.0	180.0	19.5
m6	18.0	6.0	61.0	4.0
m7	78.0	65.0	47.9	62.0
m8	240.0	19.0	11.8	10.0
h1	0.48	4.0	2.5	10.0
h3	97.0	7.0	44.1	19.0
h4	31.1	180.0	180.0	15.0
h5	19.6	180.0	180.0	42.0
h6	0.31	9.0	4.0	180.0
h10	0.89	14.0	7.6	2.0
Total	1028.09	941.1	1271.22	466.83