Speculative Symbolic Graph Execution of Imperative Deep Learning Programs*

Eunji Jeong
Seoul National University
ejjeong@snu.ac.kr

Sungwoo Cho
Seoul National University
sungwoocho@snu.ac.kr

Gyeong-In Yu
Seoul National University
gyeongin@snu.ac.kr

Joo Seong Jeong
Seoul National University
joosjeong@snu.ac.kr

Dong-Jin Shin
Seoul National University
dongjin.shin@spl.snu.ac.kr

Taebum Kim
Seoul National University
k.taebum@snu.ac.kr

Byung-Gon Chun
Seoul National University
bgchun@snu.ac.kr

ABSTRACT

The rapid evolution of deep neural networks is demanding deep learning (DL) frameworks not only to satisfy the requirement of quickly executing large computations, but also to support straightforward programming models for quickly implementing and experimenting with complex network structures. However, existing frameworks fail to excel in both departments simultaneously, leading to diverged efforts for optimizing performance and improving usability.

This paper presents JANUS, a system that combines the advantages from both sides by transparently converting an imperative DL program written in Python, a de-facto scripting language for DL, into an efficiently executable symbolic dataflow graph. JANUS can convert various dynamic features of Python, including dynamic control flow, dynamic types, and impure functions, into elements of a symbolic dataflow graph. Our experiments show that JANUS can achieve fast DL training by exploiting the techniques imposed by symbolic graph-based DL frameworks, while maintaining the simple and flexible programmability of imperative DL frameworks at the same time.

1 INTRODUCTION

As the complexity of deep neural networks is growing more than ever, scientists are creating various deep learning (DL) frameworks to satisfy diverse requirements. Current DL frameworks can be classified into two categories based on their programming and execution models. Symbolic DL frameworks including TensorFlow [**?**] and MXNet [**?**] require users to build symbolic graphs to represent the computation before execution, ensuring much efficient execution. On the other hand, imperative DL frameworks such as PyTorch [**?**] and TensorFlow Eager [**?**] directly execute DL programs, providing a much more intuitive programming style without a separate optimization phase.

Since both camps have clear advantages and also limitations, combining their advantages can improve the programmability and performance of DL frameworks at the same time. It can be achieved by converting imperative DL programs into symbolic DL graphs. However, such conversion is not trivial due to the discrepancy between imperative DL programs and symbolic DL graphs. Imperative DL programs, written in Python, a de-facto standard language for DL, have various *dynamic* features which can only be determined after actually running the program, such as variable types, control flow decisions, and the values to read from or write to the heap. On the other hand, such features must be *statically* fixed when building symbolic graphs. Moreover, as such characteristics can change after generating graphs, it can be erroneous to reuse the graphs based on outdated context information. For this reason, recent works [**?** **?**] that attempt to combine the two approaches require users to explicitly provide the necessary information, or generate incorrect or sub-optimal results when converting an imperative program to symbolic graph(s), failing to fully preserve the merits of the two approaches.

To overcome such challenges, we propose JANUS, a system that adopts a speculative symbolic graph generation and execution technique. Even though it is impossible to statically determine the program characteristics, we can make reasonable assumptions about program behavior based on the execution history. JANUS first profiles the program execution a few times, and speculatively generate a specialized graph based on the profile information. When training deep learning models, it is common to run a particular code block multiple times to apply the iterative gradient descent optimization algorithm. Running the specialized graph is likely to be valid for such repeated executions. However, due to the dynamic nature of Python, sometimes it is unavoidable that some of the assumptions become invalid, so JANUS checks the assumptions at runtime. If an assumption is broken, JANUS falls back to the imperative executor, and then generates another symbolic graph based on updated context information.

We have implemented JANUS on TensorFlow 1.8.0 [**?**]. To demonstrate the performance of JANUS, we evaluated JANUS with 11 imperative DL programs in five categories: convolutional, recurrent, and recursive neural networks, generative adversarial networks, and deep reinforcement learning models that extensively use the dynamic features of Python. JANUS converted the programs into symbolic dataflow graphs successfully, trained the models to reach target accuracy with up to

*This paper is an abridged version of the paper published in the *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, 2019.

```
1  class RNNModel(object):
2     def __call__(self, sequence):
3        state = self.state
4        outputs = []
5        for item in sequence:
6           state = rnn_cell(state, item)
7           outputs += [state]
8        self.state = state
9        return compute_loss(outputs)
10
11 for sequence in sequences:
12    optimize(lambda: model(sequence))
```

Figure 1: An imperative Python program that implements a training process of a recurrent neural network (RNN). For each `item` in the `sequence`, the `rnn_cell` function is called to produce the next state required for the next `rnn_cell` invocation.

47.6 times higher throughput compared to TensorFlow Eager, while executing the identical imperative programs.

The rest of this paper is organized as follows. Section **??** explains the challenges to address when imperative and symbolic approaches are combined. We propose our speculative approach in Section **??**. Section **??** describes detailed graph generation rules, and Section **??** presents implementation details. Section **??** presents evaluation results, which demonstrate that JANUS achieves the programmability of imperative DL frameworks and the performance of symbolic DL frameworks at the same time.

2 MOTIVATION

2.1 Challenges: Dynamic Characteristics of Imperative DL Programs

Various characteristics of a Python program, such as the execution count and execution order of statements, the types of expressions, or the global program execution state, can only be determined after the program is actually executed. For the rest of this paper, we will refer to these characteristics as the *dynamic* features of Python. In contrast, symbolic DL graphs are expected to be defined before the computation starts, to apply aggressive graph optimizations and efficiently schedule the graph operations by viewing the entire graph. In this sense, symbolic DL graphs are usually considered to be *static* [**? ? ?**]. The difference in characteristics makes it difficult to embed dynamic Python features in static DL graphs. Figure **??** depicts an imperative DL program written in Python, of which semantics are difficult to be captured in a symbolic DL graph **correctly** due to the following representative dynamic features of Python.

• *Dynamic control flow*: Conditional branches and iterative loop constructs have different execution paths depending on intermediate values. Lines 5-7 of Figure **??** show an example of an iterative loop construct used in a DL program. Such control flow statements are intensively used in Python and must be correctly represented in the dataflow graph.

• *Dynamic types*: Python is a dynamically-typed language, i.e., the type of a Python expression can only be determined at program execution time. The example program in Figure **??** does not have any type annotations (e.g., `int` or `float`), which makes it difficult to statically decide the type of target dataflow graph operations. Furthermore, various non-numerical types of Python, such as lists, dictionaries, and arbitrary class instances, are even harder to be converted into elements of a dataflow graph, of which vertices usually output numerical arrays.

• *Impure*[1] *functions*: Another useful feature for using Python is the ease of accessing and mutating global states within functions. In Figure **??**, the function `__call__` reads from and writes to an object attribute[2] at Lines 3 and 8, to pass the final state of a sequence to the next sequence. Since the modified global states can make the following function call behave differently, such reads and writes of global states must be handled correctly while generating dataflow graphs.

Moreover, correctness is not the only goal when converting an imperative program. Even if it were possible to correctly capture dynamic semantics in a symbolic DL graph, executing the graph that exactly follows the dynamic behavior of Python would be inefficient, compared to the **optimized** graph execution that could have been achieved with symbolic DL frameworks. For instance, if the input `sequence` at Line 2 in Figure **??** is expected to always have a fixed length, unrolling the following loop at Line 5 can improve the performance of the generated graph execution. However, to correctly preserve the semantics of the loop statement, we might end up with generating a complicated graph that contains dynamic control flow operations, prohibiting further graph optimizations. To achieve the performance of symbolic DL frameworks, we should provide additional information on dynamic types or control flow when building symbolic graphs, but the original imperative programs do not contain such information.

Category	Imp. pgm	Sym. exec	Conversion: Corr.	Conversion: Opt.	Framework(s)
Symbolic	×	○	–	–	TensorFlow (TF), Caffe2, MXNet
Imperative	○	×	–	–	PyTorch (PTH), TF Eager, DyNet
One-shot converters					
Record&Replay	○	○	×	○	TF `defun`, PTH JIT `trace`, Gluon
Static compiler	○	○	△	△	TF AutoGraph, PTH JIT `script`
Non-Python	○	○	○	△	Swift for TensorFlow
Speculative	○	○	○	○	**JANUS**

Table 1: Comparison of DL frameworks with respect to correctly supported features for converting imperative programs into symbolic graphs ("Corr.") and the ability to optimize the generated graphs with the information given at program runtime ("Opt.").

2.2 Related Works

Due to the aforementioned challenges, there is no framework that succeeds to fully combine the programmability of imperative DL frameworks and the performance of symbolic DL

[1] A *pure function* is a function whose return value is determined only by its parameters, and has no side effects.

[2] "class members" in C++ terminology, except that the attributes are stored in dictionaries, without fixed data layout.

frameworks. Table **??** summarizes state-of-the-art DL frameworks alongside their execution models and their status regarding the correctness and efficiency of graph conversion support. Symbolic and imperative frameworks support only one of imperative programming or symbolic graph execution model. There exist approaches that try to convert imperative programs to symbolic graphs. However, as all of them are one-shot converters, they either fail to capture important dynamic semantics of Python, or run slowly due to the lack of sufficient information at graph build time.

Frameworks with the record-and-replay approach [**? ? ?**] execute the imperative program once, and convert the single execution trace directly into a symbolic graph. Although this approach enables generating optimized symbolic graphs with sufficient information gathered from a specific execution trace, it fails to capture dynamic semantics of the Python interpreter correctly. Next, a few concurrent works support wider semantics of Python by using static compilation techniques [**? ?**]. However, as they are still one-shot converters, they sacrifice either correctness or performance when handling dynamic features. Finally, selecting a less-dynamic host language compared to Python [**?**] may enable capturing all semantics of a source program. However, it loses the merit of supporting Python, the de-facto standard language for DL programming, and introduces new programming models that DL researchers are unfamiliar with. Moreover, there may still exist dynamic behavior that does not originate from the host language, such as the dynamic shape of input data, which can result in sub-optimal performance.

3 JANUS

In this section, we propose a *speculative* execution approach, which makes it possible to preserve dynamic semantics of Python, and also achieve the performance of symbolic DL frameworks at the same time.

3.1 Speculative Execution Approach

Existing optimizers and compilers for dynamic languages suggest a useful technique for performing such conversions from imperative programs to symbolic dataflow graphs: *speculative optimization*. Managed language runtimes have succeeded in exploiting the inherent static nature of dynamic programs which rarely changes during the execution to convert them into static, low-level representations while maintaining correctness. For example, JavaScript just-in-time (JIT) compilers convert dynamic JavaScript programs into efficient machine code, and this conversion is done speculatively assuming that the program inherently maintains some statically fixed structures over repeated executions. In case this assumption breaks, the program falls back to the interpreter and attempts to compile the program again with different assumptions.

We propose to adopt this concept of speculative optimization when converting imperative DL programs into symbolic dataflow graphs. Converting various dynamic features like dynamic control flow and impure functions correctly may impose some inevitable overheads if we generate dataflow graphs in a conservative manner. To overcome this challenge, JANUS

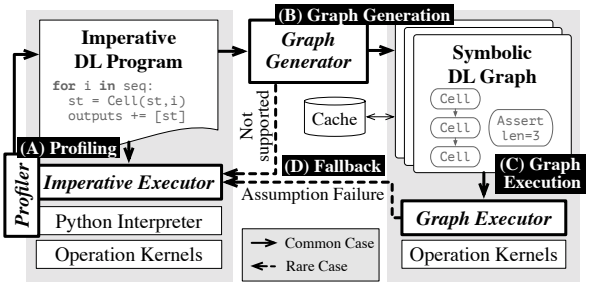


Figure 2: An illustration of the system design of JANUS, showing how an imperative DL program is processed by several components.

makes assumptions about the program's behavior based on the runtime profiling information, and generates a symbolic graph tailored for the assumptions. This speculatively constructed dataflow graph can show much better performance compared to the conservative counterpart due to specializations. If the assumptions do not hold, JANUS builds a new dataflow graph based on different assumptions. Since a DL program comprises a number of iterations of an optimization procedure, the speculative execution approach is a good fit since the interpreter is likely to execute specific code blocks of the program repeatedly.

3.2 Execution Model

Figure **??** depicts the system components and the overall execution model of JANUS. The common case in which an efficient dataflow graph is utilized is depicted as solid lines in the figure, while the rare case where the graph representation is not available is depicted as dotted lines.

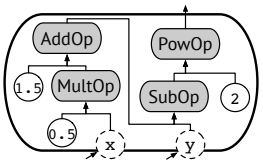
Fast Path for Common Cases. The input program is first executed imperatively with the Profiler (Figure **??** (A)). The Profiler collects required runtime information such as control flow decisions on conditional branches, loop iteration counts for iterative loop constructs, variable type information, non-local variables, object attributes, and so on. After collecting a sufficient amount of information about the input program, the Graph Generator tries to convert the program into symbolic graph(s) (Figure **??** (B)). With runtime profile information, the Graph Generator can make appropriate assumptions about the inherent static behavior of the original program. We store generated graphs in the graph cache. Then, on following invocations of the same program with the same program context, JANUS executes the cached symbolic graph (Figure **??** (C)), instead of running the original program on the Imperative Executor.

When generating a graph with assumptions, we append additional operations that validates the assumptions. For example, when we unroll a loop statement assuming that the loop count would always be a fixed value, we also append an assertion operation that checks the loop count while running the graph. In common cases where our assumption is correct, this operation rarely affects the performance of graph execution.

Accurate Path for Rare Cases. In the unfortunate case where an assumption is proven to be wrong, the associated graph cannot be executed anymore as it may produce incorrect results.

```
def loss_fn(x, y):
    y_ = 0.5 * x + 1.5
    return (y_ - y) ** 2
```

(a) A source code snippet of an imperative DL program



(b) A symbolic DL graph generated from `loss_fn`

Figure 3: Python source code, AST, and symbolic graph of a simple linear model that receives several external inputs.

With assertion operations, JANUS can safely abort the graph execution (Figure **??** (D)), and run the original input program on the Imperative Executor with the Profiler (Figure **??** (A)) to generate a graph with updated context information (Figure **??** (B)).

We make sure that no global state has been modified at the time we abort the graph execution in the middle of the program (Section **??**). We also have a fallback path from the Graph Generator to the Imperative Executor. which ensures the full python coverage even though the Graph Generator does not convert every feature of Python, for design (Section **??**) or implementation (Section **??**) issues.

4 GRAPH GENERATION

In this section, we describe the detailed graph generation rules for various features in Python.

4.1 Basic Graph Generation

Figure **??** is a simple, imperative Python program that calculates a linear model. We use this program as an example to show the basic graph conversion process. After converting the source code into an abstract syntax tree (AST), the JANUS graph generator traverses the nodes of the AST to generate corresponding graph operations (Figure **??**).

Input parameters (`x` and `y`) are converted into graph input objects that require external inputs in order to execute the graph. In the case of TensorFlow, this corresponds to `PlaceholderOp`[3]s. At runtime, they are filled with the actual argument values. The return value of the `return` statement is marked as the computation target of the graph, so that we can retrieve the value after executing the graph. Python literals such as `0.5`, `1.5` and `2` are simply converted into operations that output constant values – `ConstantOp` for TensorFlow. The conversion of mathematical operators is done by finding the corresponding mathematical graph operations and replacing them one-to-one. For standard Python operators such as `+` and `**`, JANUS places the appropriate primitive mathematical operations in the graph, like `AddOp` and `PowOp` for TensorFlow. An assignment to a Python local variable and a value retrieval from the same variable is converted into a connection between two operations, just as in Pydron [**?**]. Figure **??** illustrates how such a connection is made for the variable `y_` in Figure **??**, along with the rest of the program.

[3] `PlaceholderOp`s are unique operations that generate errors unless they are provided with external inputs before graph execution. TensorFlow expects users to feed a dictionary `{ph1: v1, ph2: v2, ...}` to a `PlaceHolderOp`.

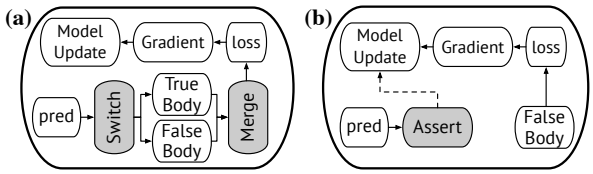


Figure 4: Symbolic DL graphs with dynamic control flow generated by (a) basic translation rules and (b) speculative graph generation rules.

4.2 Dynamic Features

In addition to the basic features, JANUS converts the dynamic features of Python into the elements of the symbolic DL graph as well to provide the performance of symbolic DL frameworks while maintaining the programmability of imperative DL frameworks. Moreover, JANUS exploits the fact that the dynamism in Python DL programs can often be simplified to static dataflow, treating a dynamic program as a program of only static aspects with appropriate program context assumptions. Context assumptions are generated based on the profile information JANUS gathers at runtime.

4.2.1 Dynamic Control Flow

Basic translation rules. Among various dynamic control flow statements, JANUS focuses on conditional branches, loop constructs, and function calls, which are enough to express most complex dynamic control flows in Python, similar to Pydron [**?**]. Python's conditional statement, the `if` statement, can be obtained by combining *switch* and *merge* primitives as in Figure **??**(a). Those primitives act as demultiplexers and multiplexers, selecting a single path to pass their inputs or outputs. Similarly, the iterative statements of Python, `while` and `for`, can be converted using additional primitives for creating iteration frames and passing values repeatedly over them. TensorFlow conveniently provides operations for handling such dynamic control flow statements [**?**]. Finally, for function calls, a separate graph is generated for the callee function, and a function invocation operation that points to the generated graph is inserted in the position of the function calls. Recent work proposes an implementation of this operation called `InvokeOp` [**?**], which can represent an invocation of a recursive function with automatic differentiation support.

Speculative graph generation: unrolling and inlining. If the JANUS profiler detects that only a single particular path is taken for a certain control flow statement, JANUS presumes that the control flow decision is actually fixed. The system replaces the control flow operation with an assertion operation that double-checks the assumption for this control flow decision, and proceeds with graph generation as if the control flow statement were unrolled. Figure **??**(b) illustrates a specialized version of the graph in Figure **??**(a), generated under the assumption that the predicate of the conditional statement will always be false. This allows JANUS to remove control flow operation overheads and apply graph optimizations such as common subexpression elimination or constant folding in broader portions of the graph. When the assumption is broken, the assertion operation raises

an error so that we can fall back to the imperative executor. Loops can also be unrolled in a similar manner.

For function calls, if the callee is expected to be fixed for a function call at a certain position, JANUS inlines the callee function body inside the caller unless that function call is identified as a recursive one. In addition, for callee functions whose implementation is already known for JANUS, e.g., the functions provided by the framework such as `matmul()` or `conv2d()`, or Python built-in functions like `print()` or `len()`, JANUS adds the corresponding graph operations that behave the same as the original callee functions, based on the prior knowledge about their behaviors. Section **??** includes more details and limitations about such function calls.

4.2.2 Dynamic Types

Basic translation rules. The types of all expressions within a Python program must be known before JANUS converts the program into a symbolic graph, because graph operations require operands to have fixed types. This is a challenging task for Python programs because we cannot determine the type of an arbitrary Python expression before actually executing the expression. Fortunately, it is possible to infer the types of some expressions, given the types of other expressions; for example, it is clear that the variable `c` in `c = a + b` is an integer if `a` and `b` are integers.

As a basic rule, JANUS converts numerical Python values such as scalars, list of numbers, and NumPy [**?**] arrays into corresponding tensors, and converts non-numerical values, including arbitrary class instances, into integer-typed scalar tensors that hold pointers to the corresponding Python values. Next, JANUS infers the types of other expressions that are derived from expressions covered by the basic rules.

Speculative graph generation: specialization. Expressions whose types cannot be inferred from other expressions require a different measure. For instance, it is impossible to identify the types of input parameters for functions, or Python object attribute accesses (`obj.attr`) without any external clues. Similarly, inferring the return types of recursive function calls is also challenging due to the circular dependencies. To make proper assumptions about the types of such expressions, *Profiler* observes the types of the expressions during imperative executions. Given these context assumptions, JANUS can finish inferring the types of remaining expressions, and construct a specialized dataflow graph accordingly.

In addition, JANUS makes further assumptions about the expressions to apply more aggressive optimizations. For numerical expressions, we can try to specialize the shape of tensors before constructing the graph. Furthermore, if a Python expression always evaluates to the same value while profiling, JANUS converts it into a constant node in the symbolic graph. With statically determined shapes or values, the graph can be further optimized, or even be compiled to the efficient machine code [**? ?**].

4.2.3 Impure Functions

Basic translation rules. It is common for a Python function to access global variables to calculate return values and have side-effects, mutating its enclosing Python context during execution. Likewise, it is common for a Python DL program to read from and write to global states such as global or nonlocal variables and heap objects. JANUS respects this characteristic and handles global state accesses alongside symbolic graph execution.

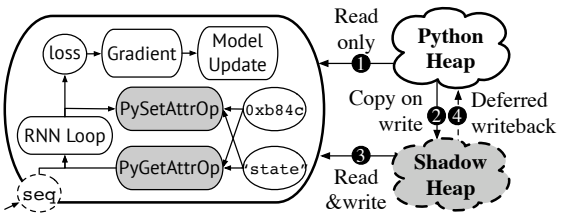


Figure 5: Symbolic dataflow graph generated graph from Figure ?? and the global states.

A trivial solution is to use TensorFlow's `PyFuncOps`, which can execute arbitrary Python functions as graph operations. A function for reading and updating a certain global state can be created and inserted in the appropriate position within the graph. However, this trivial approach has clear limitations. First, since only one Python function can be executed at a time due to the global interpreter lock (GIL), the overall performance can be degraded when multiple operations should be executed in parallel. It also complicates the fallback mechanism of JANUS. If a global state has already been mutated before the fallback occurs, instead of starting the imperative executor from the function entrance at fallback, execution must start from the middle of the function to be correct, by mapping the state update operation with corresponding Python bytecode.

Speculative graph generation: deferred state update. To make things simpler and also faster, JANUS does not mutate global states in place on the fly. JANUS instead creates local copies of global states on the shadow heap, and only mutates the shadow heap during symbolic graph execution.

Figure **??** shows the symbolic dataflow graph version of the program in Figure **??**, which includes the object attribute expressions (`self.state`) that access and mutate the global states. We add new graph operations `PyGetAttrOp` and `PySetAttrOp` to represent Python attribute read and write. Each of them receives an object pointer (`0xb84c`) and a name of the attribute (`"state"`) as inputs, and behaves as follows: ① The `PyGetAttrOp` can access the Python heap to read the state unless a corresponding local copy exists. ② When the `PySetAttrOp` wants to update the attribute, a new entry is written to the shadow heap, instead of directly updating the Python heap. ③ Further read and write operations are redirected to the shadow heap. Note that JANUS inserts appropriate dependencies between `PyGetAttrOps` and `PySetAttrOps` if necessary to prevent any data hazards. ④ After the graph executor finishes this run, the values in the shadow heap are written back to the Python heap. Global or nonlocal variables can also be regarded as the object attributes, where the global variables are the attributes of the global object, and the nonlocal variables are the attributes of the function's closure objects. Subscript expressions (`obj[subscr]`) are similarly implemented with equivalent custom operations, `PyGetSubscrOp` and `PySetSubscrOp`.