Demonstration of JANUS[†]: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs

Eunji Jeong [1] **Sungwoo Cho** [1] **Gyeong-In Yu** [1] **Joo Seong Jeong** [1] **Dong-Jin Shin** [1] **Byung-Gon Chun** [1]

Abstract

We demonstrate Janus, a system that achieves the performance of symbolic DL frameworks while maintaining the programmability of imperative DL frameworks. To achieve the performance of symbolic DL frameworks, Janus converts imperative DL programs into static dataflow graphs by exploiting the inherently static nature of DL programs. To preserve the dynamic semantics of Python, Janus generates and executes the graph speculatively, verifying the correctness of the graph at runtime.

1 Introduction

As the complexity of deep neural networks are 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 (Abadi et al., 2016) and MXNet (Chen et al., 2015) 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 (Paszke et al., 2017) or TensorFlow Eager (Agrawal et al., 2019) 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. Recent works such as AutoGraph (Moldovan et al., 2019), PyTorch JIT (Paszke et al., 2017), and MXNet Gluon (glu) attempt to combine the two approaches.

However, converting an imperative program into symbolic graph(s) is not trivial due to the discrepancy between Python programs and symbolic graphs. Various characteristics of Python, including variable types, control flow decisions and the values to read from or write to the heap, cannot be determined at static time without runtime information. On the other hand, such characteristics are mandatory for 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Moreover, as such characteristics can change after generating graphs, it can be erroneous to reuse the graphs based on outdated context information.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 preserving the merits of the two approaches.

To overcome such challenges, Janus (Jeong et al., 2019) adopts speculative symbolic graph generation and execution. Janus first introduces a profiling phase to gather information required for generating optimized symbolic graphs. Second, to ensure the correctness of graph execution, Janus takes a speculative graph generation and execution approach, automatically analyzing the program context and invalidating the graphs with outdated context information. With this technique, Janus successfully converts various neural networks written in TensorFlow Eager, including convolutional, recurrent and recursive neural networks and deep reinforcement learning models, achieving up to 47.6 times higher training throughput than TensorFlow Eager. In the demonstration, we will show that Janus can correctly convert imperative DL programs into symbolic graphs with improved performance, requiring no additional input from users.

2 Janus

We briefly explain how the dynamic features of Python are converted into symbolic graph elements via profiling and speculative approach in Janus, then describe the system design of Janus that employs the technique.

Handling Dynamic Features of Python. Janus can convert various dynamic features of Python correctly with little performance penalty, by specializing dynamic characteristics based on the runtime information. For example, for dynamic control flow statements, if a certain control flow statement takes only a particular path during profiling, Janus converts only the path to graph elements without complex symbolic

[†]Appeared in NSDI '19. [1]Department of Computer Science & Engineering, Seoul National University, Seoul, Korea. Correspondence to: Eunji Jeong <ejjeong@snu.ac.kr>, Byung-Gon Chun <bgchun@snu.ac.kr>.

Proceedings of the 2^{nd} *SysML Conference*, Palo Alto, CA, USA, 2019. Copyright 2019 by the author(s).

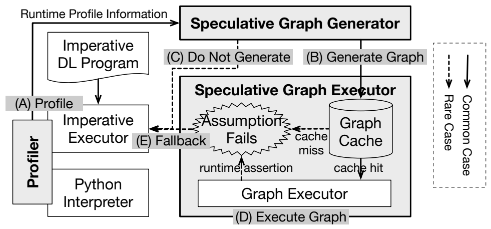


Figure 1: An illustration of the execution model of JANUS

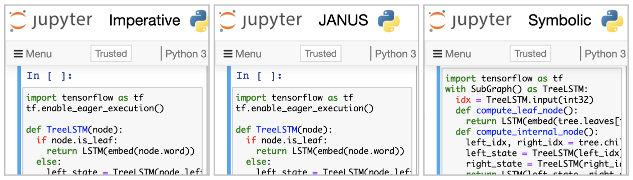
control flow operations, assuming that the control flow decision is actually fixed. In addition, to ensure the correctness, JANUS inserts an assertion operation that validates the assumption. If the assumption does not hold, JANUS makes more relaxed assumptions to embrace additional runtime context information and constructs corresponding graphs. Dynamic types and impure functions of Python can be handled similarly (Jeong et al., 2019).

System Design. Figure 1 depicts the system components and the overall execution model of JANUS. Once JANUS receives a DL program, the program is first executed imperatively, while Profiler gathers runtime information required for making reasonable program context assumptions (A). After a sufficient amount of information has been collected, Speculative Graph Generator tries to convert the program into symbolic dataflow graph(s) with the assumptions based on the runtime information (B). Imperative Executor executes the part of the program that is not converted into symbolic graphs (C). If the system already has a graph with correct assumptions regarding the program context, Speculative Graph Executor executes the symbolic graph instead of imperative execution of the program (D). If the runtime context does not comply with the assumptions or the executor detects any broken assumptions during the graph execution, the system cancels the graph execution and falls back to imperative execution (E). In this case, JANUS starts the procedure from profiling again to generate a graph with updated context information.

3 DEMONSTRATION

We aim to demonstrate that JANUS achieves symbolic graph framework performance while maintaining the simple programmability of imperative execution by correctly handling complex features of Python. We will showcase our system with various DL models including convolution and recursive neural networks.

Programmability and Performance. To show that JANUS can transparently convert various imperative DL programs, we use Jupyter (Kluyver et al., 2016) Notebook to execute Python programs with various execution models, as shown in Figure 2. The exact same program written in the intuitive imperative programming model will run correctly, yet much faster on JANUS, compared to the imperative execution. In addition, we also show that the users must write complex programs to achieve the high throughput in symbolic DL



```
In [ ]:
import tensorflow as tf
tf.enable_eager_execution()

def TreeLSTM(node):
  if node.is_leaf:
    return LSTM(embed(node.word))
  else:
    left_state = TreeLSTM(node.lef
```

```
In [ ]:
import tensorflow as tf
tf.enable_eager_execution()

def TreeLSTM(node):
  if node.is_leaf:
    return LSTM(embed(node.word))
  else:
    left_state = TreeLSTM(node.lef
```

```
import tensorflow as tf
with SubGraph() as TreeLSTM:
  idx = TreeLSTM.input(int32)
  def compute_leaf_node():
    return LSTM(embed(tree.leaves[
  def compute_internal_node():
    left_idx, right_idx = tree.chi
    left_state = TreeLSTM(left_idx
    right_state = TreeLSTM(right_i
```

Figure 2: Jupyter Notebooks with Imperative (left), JANUS (middle), and Symbolic (right) frameworks installed.

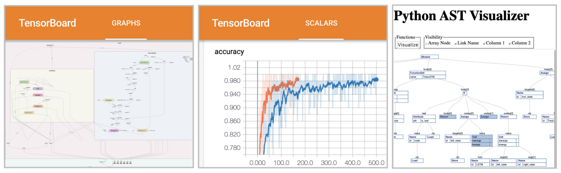


Figure 3: Visualization of a generated graph (left), training progress (middle), an annotated AST (right).

programs. We use TensorBoard (ten) to visualize the live training progress, as shown in Figure 3. Using Jupyter Notebook the audience can directly modify the code and compare the results.

Correctness. To demonstrate the correctness of JANUS, first we will show that it is easy to produce erroneous results when converting imperative DL programs into symbolic graphs in existing graph conversion frameworks without our speculative execution approach. We then show that JANUS handles dynamic Python features correctly. We will also provide separate notebooks for executing other related work. To provide more detailed information, we will use vpyast (vpy) and TensorBoard to visualize annotated ASTs and generated graphs as shown in Figure 3.

Required Equipment. Our demonstration does not require any specific equipment, except for a laptop connected to the Internet, which we will prepare for. However, a large display can be useful for the audience to read source code, generated graphs, and training dashboards.

REFERENCES

Gluon. http://gluon.mxnet.io/.

TensorBoard. https://github.com/tensorflow/tensorboard.

vpyast. https://github.com/ivan111/vpyast.

Abadi, M. et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

Agrawal, A. et al. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. In *SysML*, 2019.

Chen, T. et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Workshop on Machine Learning Systems in NIPS*, 2015.

Jeong, E. et al. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *NSDI*, 2019.

Kluyver, T. et al. Jupyter notebooks - a publishing format for reproducible computational workflows. In *ELPUB*, 2016.

Moldovan, D. et al. Autograph: Imperative-style coding with graph-based performance. In *SysML*, 2019.

Paszke, A. et al. Automatic differentiation in pytorch. In *Autodiff Workshop in NIPS*, 2017.