Ph.D. DISSERTATION

Unifying Imperative and Symbolic Deep Learning Execution

명령형과 심볼릭 그래프 기반 딥러닝 수행 방식의 통합

FEBRUARY 2021

DEPT. OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Eunji Jeong

Ph.D. DISSERTATION

Unifying Imperative and Symbolic Deep Learning Execution

명령형과 심볼릭 그래프 기반 딥러닝 수행 방식의 통합

FEBRUARY 2021

DEPT. OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Eunji Jeong

Unifying Imperative and Symbolic
Deep Learning Execution

명령형과 심볼릭 그래프 기반 딥러닝 수행 방식의 통합

지도교수 전 병 곤

이 논문을 공학박사 학위논문으로 제출함

2020 년 10 월

서울대학교 대학원

컴퓨터공학부

정 은 지

정은지의 공학박사 학위논문을 인준함

2020 년 12 월

위 원 장	이재욱	(인)
부위원장	전병곤	(인)
위 원	허충길	(인)
위 원	김건희	(인)
위 원	황승원	(인)

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 thesis presents systems to unify two existing paradigms in current deep learning frameworks, symbolic and imperative, to achieve the performance and programmability at the same time. First we present Janus, a system that combines the advantages from both sides by transparently converting an imperative DL program written in Python, the 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.

Next, we propose Terra, an imperative-symbolic co-execution framework for imperative DL programs. As the usability of deep learning (DL) framework is getting more important, the imperative programming model has become an essential part of recent DL frameworks. However, optimizing individual operations in imperative programs has limited opportunities compared to optimizing them as a group in a symbolic graph format. Still, existing approaches that convert imperative DL programs into optimized symbolic graphs cannot provide a general solution due to their limited program coverage. Terra decouples the

actual computation of DL operations from imperative programs and converts the DL operations into an optimized graph. Then the optimized graph and the skeleton imperative program are executed at the same time in a complementary manner to each other, so that we can achieve high performance of optimized graph execution while supporting the whole semantics of the original imperative program.

Among various DL models, we additionally delve into recursive neural networks (TreeNNs), which are important yet highly challenging to be represented as DL graphs. We introduce new DL abstractions, SubGraph and InvokeOp, which naturally capture any tree- or graph-like structure of the input data as DL graph elements. Then, we present our underlying system that supports the automatic differentiation of the abstractions and efficiently executes TreeNNs by running InvokeOps in parallel.

We implemented a system using the proposed Janus architecture, which additionally exploits recursive DL abstractions. Our evaluation 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.

Keywords: imperative, symbolic, deep learning framework, Python

Student Number: 2017-37662

Contents

Abstract **i**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Challenges 2

1.3 Contribuiton 2

1.4 Organization 4

Chapter 2 Background **5**

2.1 Symbolic Graph Definition and Execution 5

2.2 Imperative Graph Definition and Execution 6

2.3 Execution models of Imperative Programs 6

2.3.1 Python-oriented approaches 7

2.3.2 Graph-oriented approaches 8

Chapter 3 Speculative Graph Generation and Execution **10**

3.1 Motivation 10

3.1.1 Challenges in Graph Generation 10

3.1.2 Related Works 14

3.2 Proposed Solution: Speculative Graph Generation and Execution 16
3.3 Janus System Design 17
3.3.1 Fast Path for Common Cases 19
3.3.2 Accurate Path for Rare Cases 20
3.4 Symbolic Graph Generation 21
3.4.1 Graph Generation Basics 22
3.4.2 Dynamic Features 24
3.4.3 Python Syntax Coverage 32
3.4.4 Imperative-Only Features 32
3.5 Implementation 34
3.6 Evaluation 35
3.6.1 Experimental Setup 35
3.6.2 Model Convergence 38
3.6.3 Training Throughput 39
3.7 Summary 43

Chapter 4 Imperative-Symbolic Co-Execution 44

4.1 Motivation 44
4.1.1 Limitations of Existing Approaches 44
4.1.2 Motivating Example 46
4.1.3 Proposing Solution 46
4.2 Terra Overview 47
4.3 System Design 49
4.3.1 Graph Merging 49
4.3.2 Inter-runner Communication 49
4.3.3 Graph Validation 50

4.4 Implementation 50
4.5 Evaluation 50
4.5.1 Experiment Setup 50
4.5.2 Performance 52
4.6 Summary 54

Chapter 5 Graph Generation for Recursive Networks 55

5.1 Introduction 55
5.2 Motivation 57
5.2.1 Embedded Control Flow Frameworks and Their Limitations 57
5.2.2 Example: TreeLSTM 60
5.2.3 Recursion in Embedded Control Flow Frameworks 61
5.3 Programming Model 63
5.3.1 Unit of Recursion: `SubGraph` 63
5.3.2 Recursion in Dataflow Graphs: `InvokeOp` 65
5.3.3 TreeLSTM with `SubGraph`s & `InvokeOp`s 66
5.4 System Design 68
5.4.1 Graph Execution 68
5.4.2 Graph Backpropagation 72
5.5 Implementation 75
5.6 Evaluation 78
5.6.1 Experimental Setup 79
5.6.2 Throughput and Convergence Time 80
5.6.3 Analysis of Recursive Graphs: Parallelization 83
5.6.4 Comparison with Folding 86
5.7 Related Work 89
5.8 Summary 92

Chapter 6 Conculsion and Future Work **94**

6.1 Conclusion 94

6.2 Future Work 95

6.2.1 Lightweight Imperative Runtime 96

6.2.2 Alternative Languages for Deep Learning 96

List of Figures

Figure 3.1 A Python program that implements training process of a recurrent neural network (RNN) in an imperative manner. For each `item` in the sequence, `rnn_cell` function is called to produce the next state required for the next `rnn_cell` invocation. After finishing up processing the whole sequence, the model holds the final `state` by replacing `self.state` attribute for processing the next sequence. 12

Figure 3.2 An illustration of the execution model of Janus, showing how a DL program is processed by several components. *Profiler* observes imperative program execution and collects information to make the realistic assumptions. *Speculative Graph Generator* generates dataflow graphs from the program and hands the optimized graphs over to Speculative Graph Executor. The *Speculative Graph Executor* actually runs the generated graph and handles assumption failures. 18

Figure 3.3 The Python source code, AST, and symbolic graph of a simple linear model that receives several external inputs. The static features of the program are represented as nodes in the AST, which in turn are converted to vertices of the symbolic graph. 23
Figure 3.4 Type, shape, and value specialization hierarchy for an example tensor. 28
Figure 3.5 Symbolic dataflow graph generated graph from Figure 4.1 and the global states. 29
Figure 3.6 (a) The test error of ResNet50, (b) validation perplexity of LM, (c) test accuracy of TreeLSTM, (d) episode reward of PPO, and (e) discriminator loss of AN measured on Janus, TensorFlow (Symbolic), TensorFlow Eager (Imperative), and TensorFlow `defun` (Tracing) according to the elapsed time in seconds. Each marker in (b) represents each training epoch, describing that per-epoch convergence is slower on TensorFlow `defun` compared to other frameworks. 38
Figure 3.7 The contribution of optimizations to improve training throughput. Optimizations are cumulative. **IMP**: Imperative, **BASE**: Janus without following optimizations, **+UNRL**: control flow unrolling, **+SPCN**: type specialization, **+PARL**: graph executor with 72 threads in threadpool (Janus) 41

Figure 3.8 Training throughput for the ResNet50, Inception-v3, LM, and PPO models on Janus, TensorFlow (Symbolic), TensorFlow Eager (Imperative), using varying numbers of GPUs. 42

Figure 4.1 A part of a PyTorch implementation of the BERT [24] model 45

Figure 4.2 System overview 48

Figure 4.3 The training speedup of Terra and AutoGraph+ relative to TensorFlow imperative execution. The baseline is the training throughput of the imperative execution, which is presented as a dotted horizontal line. 52

Figure 4.4 Due to the long Python execution path of TensorFlow, the Graph Runner stalls and waits for a label `y` to be fed from Python Runner. When the Graph Runner gets the label, it can resume the execution and calculate a loss. . 53

Figure 5.1 Iterative implementation of the TreeLSTM model in pseudocode. 62

Figure 5.2 Recursive implementation of the TreeLSTM model with `SubGraph` definitions. After declaring the start of a `SubGraph` in Line 2, we indicate the inputs of the `SubGraph` in Line 3. The body of the `SubGraph` is defined in Lines 5-16, while recursive calls are made on Lines 10-11. Note that `SubGraph` outputs must be given as in Lines 14-16. The completed `SubGraph` definition can now be used in Line 18. 65

Figure 5.3 An illustration of how an unrolled computation graph of the TreeLSTM model (a, b) can be transformed into a recursive graph with `InvokeOp`s (c). The base case, depicted in the boxes of (a), and the recursive case, indicated by the boxes in (b), can be combined to succinctly describe the model as a recursive `SubGraph` as shown in (c). `InvokeOp`s have been added at the appropriate places to mark the points where a recursive function call to the `SubGraph` must occur. 67

Figure 5.4 The execution model of embedded control flow frameworks with `InvokeOp`s. (1) After the client initiates the job with a dataflow graph, (2) the master decomposes the graph into operations and places them into either the ready queue or the waiting line of the worker, depending on the number of unresolved inputs. (3) Operations are dequeued from the queue by idle execution threads, while new operations are enqueued when input dependencies are resolved. (4) When an `InvokeOp` gets executed, its associated `SubGraph` is passed to and processed by the master, similar to step (1). Only one worker is shown for the sake of clarity. 69

Figure 5.5 Backpropagation of dataflow graphs with and without `InvokeOp`s. Notice how (a) and (b) are structurally very similar, except for the enclosing `InvokeOp`s. 73

Figure 5.6 Concurrent hash table being used between multiple forward and backward executions of the same operation (`InvokeOp`). 77

Figure 5.7 Training throughput for the TreeRNN, RNTN, and TreeLSTM models with the Large Movie Review dataset. Numbers are shown for our recursive implementation, TensorFlow's iterative implementation, and PyTorch's static unrolling implementation. Our recursive implementation outperforms the other frameworks for all models and all batch sizes except when training TreeLSTM with a batch size of 25, at which point the amount of system resources is insufficient to completely parallelize the computation. We did not observe any significant performance gain for the static unrolling approach when the batch size was increased. 80

Figure 5.8 Inference throughput for the TreeRNN, RNTN, and TreeLSTM models with the Large Movie Review dataset. Measurements are presented for our recursive implementation, TensorFlow's iterative implementation, and PyTorch's static unrolling implementation. Our recursive implementation outperforms the other frameworks for all models and all batch sizes. 81

Figure 5.9 Validation accuracy for the binary sentiment classification task with (a) TreeRNN, (b) RNTN, and (c) TreeLSTM models. Results are shown for training each model with the recursive and iterative implementations, using the Large Movie Review dataset. The time to reach 93% accuracy for each setup is also plotted, showing that the recursive implementation converges faster for all models. 82

Figure 5.10 Training throughput for the TreeLSTM model on our recursive implementation, using varying numbers of machines for data parallelism. The performance increases almost linearly as more machines are used for training. . 83

Figure 5.11 Time taken for processing each data instance, in the TreeLSTM model using the Large Movie Review dataset. The bold lines represent the average time for each specific sentence length in the whole dataset, and the enclosing colored areas represent the range of time taken to process the specific length of sentences. No batching is used for this experiment. As the number of words inside a data instance increases, our recursive implementation outperforms the iterative implementation thanks to the parallelized execution of tree cells. For inference, the computation load is low enough for the framework to utilize system resources without hitting the resource limit, and the processing time of the recursive implementation is $O(logN)$, where N is the number of words. 84

List of Tables

Table 3.1 Comparison of DL frameworks with respect to correctly supported features for converting imperative programs into symbolic graphs ("Correctness") and the ability to optimize the generated graphs with the information given only at program runtime ("Optimization w/ runtime info"). Optimizations can be incorrect in some frameworks ("○(unsafe)"), not preserving the original semantics of Python. The host language is also specified. 11

Table 3.2 The mapping of the full list of CPython opcode and the corresponding sections. 31

Table 3.3 Categories, models, datasets, batch sizes ("BS"), and the dynamic features of the applications used for evaluation. 37

Table 3.4 Training throughput of all models evaluated on a single machine with a single GPU in Janus, TensorFlow (Sym.), and TensorFlow Eager (Imp.). The numbers represent processed images/s for CNN and GAN models, processed words/s for RNN models, processed sentences/s for TreeNN models, and processed frames/s for DRL models. 40

Table 4.1 The ratio of the Graph runner stall time and the Python Runner stall time against the total execution time. 52

Table 5.1 Throughput for the TreeRNN model implemented with recursive dataflow graphs, using datasets of varying tree balancedness. The balanced dataset exhibits highest throughput thanks to the high degree of parallelization, but at the same time does not improve as well as the linear dataset when the batch size increases from 1 to 25, because there is only a small room of performance improvement left, w.r.t parallelization. 85

Table 5.2 Throughput for processing the TreeLSTM model on our recursive framework, Fold's folding technique, and TensorFlow's iterative approach, with the Large Movie Review dataset. The recursive approach performs the best on inference with efficient parallel execution of tree nodes, while the folding technique shows better performance on training thanks to its GPU exploitation. 87

Table 5.3 Throughput for evaluating the TD-TreeLSTM model on our recursive framework and TensorFlow's iterative implementation, on batch sizes of 1 and 64.[1] Being able to execute tree nodes in parallel lets our framework perform better than the iterative approach. Fold's folding technique is inapplicable to the TD-TreeLSTM model. . . 89

Chapter 1

Introduction

1.1 Motivation

In recent years, deep neural networks have been widely used in various application domains such as computer vision, speech, and natural language processing for their powerful capabilities of extracting abstract features from data. Scientists have created deep learning (DL) frameworks – TensorFlow [8], PyTorch [70], Caffe2 [27], MXNet [17], and many more [98, 94, 68, 84, 66, 30, 86, 12] – to improve the performance of deep neural networks in various jobs and promote the use of deep neural networks in both production and research.

Such DL frameworks can be classified into two distinct families depending on their execution models. One family comprises frameworks that base their execution on symbolic graphs constructed from DL programs. The other family consists of frameworks that directly execute DL programs in an imperative manner.

1.2 Challenges

The different characteristics of DL frameworks suggest that we cannot achieve high performance and good usability at the same time. To reach high performance, we must sacrifice framework usability to a certain extent, and vice versa. Otherwise, users are forced to resort to an awkward approach of learning how to use several frameworks and switching between them according to the current task in hand.

In this thesis, we propose architectures to transparently convert imperative Python DL programs into symbolic dataflow graphs. By not altering the user-facing interface for building neural networks, we maintain the flexible programmability of frameworks with imperative execution models. At the same time, behind the scenes, we execute the symbolic graph versions of the imperative programs to enjoy the performance optimizations done by symbolic graph execution models.

However, this approach introduces a technical challenge of capturing the dynamic semantics of an imperative Python program in a symbolic dataflow graph. The dynamic aspects of Python, including dynamic control flow, dynamic typing, and impure functions, are difficult to embedded in a symbolic graph correctly while providing the performance of symbolic graph execution frameworks.

1.3 Contribuiton

To this end, we first present Janus, a DL framework that achieves the best of both worlds by receiving an imperative DL program as input and creating symbolic graphs of the program accordingly with speculative program context assumptions. Janus makes environment assumptions on the program context

(e.g., constant variables and branches) based on past iterations to simplify the dynamic nature of the program and transform the program into a symbolic graph. These assumptions are speculative, because the context may change during execution; an incorrect assumption results in an invalidation of a symbolic graph, in which case Janus falls back to imperative execution to guarantee correctness. For design (Section 3.4.4) and implementation (Section 3.4.4) reasons, Janus converts only the subset of Python programs into the efficient symbolic graphs, but the rest of them still can be executed imperatively, ensuring the full Python coverage.

We also propose a new imperative-symbolic co-execution architecture to overcome the limitation of Janus. Existing approaches that convert imperative DL programs into optimized symbolic graphs, including Janus, cannot provide a general solution due to their limited program coverage. Terra decouples the actual computation of DL operations from imperative programs and converts the DL operations into an optimized graph. Then the optimized graph and the skeleton imperative program are executed at the same time in a complementary manner to each other, so that we can achieve high performance of optimized graph execution while supporting the whole semantics of the original imperative program.

This thesis also presents new DL abstractions for the special type of input imperative programs, where the programs include recursive function calls. We propose SubGraph and InvokeOp in DL graph context, which corresponds to a function and a function call expression, respectively, in general-purpose programming languages. Our abstractions enables converting recursive functions in imperative programs into DL symbolic graph elements, and also enables efficient execution compared to the original imperative program by running multiple InvokeOps in parallel.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 describes the programming and execution models of existing deep learning frameworks. In Chapter 3, we present Janus, which introduced speculative graph generation and optimization technique to unify imperative and symbolic graph execution. Chapter 4 proposes Terra, an imperative-symbolic co-execution framework that enables to widen the Python coverage of previous approaches including Janus. Chapter 5 covers the special case where the input imperative programs given to the systems include the recursive function. We conclude in Chapter 6 and discuss the future work.

Chapter 2

Background

2.1 Symbolic Graph Definition and Execution

Frameworks such as TensorFlow [8], Caffe2 [27], and MXNet [17] formulate neural networks as symbolic dataflow graphs. Graph vertices denote the states and operations of a neural network, while graph edges indicate the flow of data between vertices. Operations in the graph are executed as their dependencies are solved, similar to how most dataflow systems process dataflow graphs [22, 40]. The graph representation allows the framework to identify which operations can be run in parallel, and apply various compiler optimization techniques such as common subexpression elimination or constant folding to generate optimized versions of graphs. Moreover, it is easy to process dataflow graphs on accelerator devices or deploy graphs across multiple machines by assigning an operation to the appropriate device or machine [60].

However, the separation of building a symbolic graph and executing it complicates user experience, because users are not actually running any numerical

computations when defining neural networks through the framework interface. Rather, they are constructing graphs that will be executed later through separate functions.

2.2 Imperative Graph Definition and Execution

In contrast, frameworks including PyTorch [70], TensorFlow Eager [84], and MXNet Imperative [64] have adopted the execution model of running operations imperatively, without going through a separate graph construction phase. Stemming from popular Python libraries for scientific, numerical computation such as NumPy [97] and Scikit-learn [15], this imperative approach is useful for rapidly experimenting and working with new neural network models, particularly those with complex structures. The native control flow statements of Python can be exploited to build models of interest. Unfortunately, skipping the formation of a dataflow graph means that such frameworks lose the chance to apply the many optimizations that were possible in the symbolic graph execution model, leading to significant performance differences for certain models.

2.3 Execution models of Imperative Programs

Although the symbolic graph programming model achieves higher performance than the imperative programming model, we notice that the shift from the former to the latter is a major trend to fulfill requirements of deep learning researchers. The imperative programming model is easy to use and debug and suits well for rapid development and experimentations of various neural networks. They provide the imperative programming model but differ from each other in terms of their execution model and flexibility in programming.

2.3.1 Python-oriented approaches

Python-oriented approaches treat a DL program entirely as a Python program. TensorFlow Eager [84] and PyTorch (synchronous) [70] follow synchronous Python-oriented approaches. In these approaches, DL operations are executed one by one under the control of the Python interpreter.When the Python interpreter encounters a Python statement that declares a DL operation, the operation is immediately executed as a blocking call. This approach allows users to exploit the imperative programming model of Python safely. DL models are Python programs, and the features developed for general-purpose programming in Python can be reused without limitations.

PyTorch (asynchronous) [70] and DyNet [68] are similar in the sense that they make the Python interpreter control the execution of DL operations, but they have additional optimizations to achieve higher performance. In these approaches, the Python interpreter interprets each Python statement as synchronous approaches do, but the actual execution of the operation is a non-blocking call. The operation either runs asynchronously with the interpreter [70] or gets delayed to be executed together with other operations [68]. The asynchronous option is useful for hiding the Python interpreter overhead to some degree, while delayed execution provides opportunities for optimizing the partial lineage of the operations.

Although the Python-oriented approaches have been quite successful at gaining performance, it is restricted to apply additional optimizations to achieve higher performance. Most graph-level optimization techniques (e.g., operation fusion [69, 44, 89, 18, 104], device placement [58, 59, 104], layout optimization [55, 44], memory optimization [42, 37, 9]) assume that the whole lineage (or at least a substantial portion) of computation is visible at the time of opti-

mization, which is infeasible in the Python-oriented approaches.Moreover, since the Python-oriented approaches do not construct symbolic graphs that can be optimized once and used repeatedly, any applicable optimizations must take place every training step, incurring overheads.

2.3.2 Graph-oriented approaches

Recent works present approaches to convert whole imperative programs into symbolic graphs transparently if they can [5, 30, 62]. The symbolic graphs are further optimized and are executed on a graph executor, a dedicated runtime specialized for symbolic graphs.

We group these approaches according to how they convert imperative programs into symbolic graphs. TF `function` [88], PyTorch JIT `trace` [4], and MXNet Gluon [66] take the tracing (record and replay) approach. At initialization time, they construct a symbolic graph with respect to the given inputs by saving the trace of DL operations. The extracted graph is cached and reused for further execution. However, since the tracing approach uses a fixed trace obtained by running the program only once, they cannot deal with dynamic features of the program. When the program takes a different control flow branch in the next training step, this approach silently performs the same computation as the previous step, which is incorrect.

TorchScript [5] guarantees correctness by defining its own subset of the Python language and statically compiles a program into a TorchScript graph in PyTorch. If the program contains any syntax that cannot be translated, TorchScript fails to generate a symbolic graph. Autograph [62] operates as an extension of TF `function` [88] to alleviate the tracing approach's correctness problem. Autograph first compiles Python features (mostly control flow statements) into TensorFlow operations, then TF `function` saves the trace of DL

operations of the transformed program. However, since Autograph also adopts static compilation, it supports only a subset of Python features.

Since Autograph and TorchScript do not support all Python features, they provide the official language references [2, 6] that users should be familiar with beforehand. However, such reference documents are complicated for users to understand. For example, the Autograph document spans roughly ten pages and divides dynamic features that Autograph does not support into four categories and 11 subcategories. TorchScript provides a language specification [7] similar to Autograph. It enumerates all 139 Python features from the official Python language reference [3] and shows whether TorchScript supports them or not. Among the 139 features, TorchScript does not support 43 features and it partially supports 22 features.

Chapter 3

Speculative Graph Generation and Execution

3.1 Motivation

3.1.1 Challenges in Graph Generation

Converting an imperative program written in Python into a DL dataflow graph brings on many challenges, because dataflow graphs consist of a restrictive set of operations, lacking the dynamic semantics of the programming language. More specifically, 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 thesis, we will refer to these characteristics as the *dynamic* features of Python. In contrast, DL dataflow 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, DL dataflow graphs are usually considered to be *static* [56, 68, 70].

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 3.1: Comparison of DL frameworks with respect to correctly supported features for converting imperative programs into symbolic graphs ("Correctness") and the ability to optimize the generated graphs with the information given only at program runtime ("Optimization w/ runtime info"). Optimizations can be incorrect in some frameworks ("○(unsafe)"), not preserving the original semantics of Python. The host language is also specified.

```
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 3.1: A Python program that implements training process of a recurrent neural network (RNN) in an imperative manner. For each `item` in the sequence, `rnn_cell` function is called to produce the next state required for the next `rnn_cell` invocation. After finishing up processing the whole sequence, the model holds the final `state` by replacing `self.state` attribute for processing the next sequence.

The difference in characteristics makes it difficult to embed dynamic Python features in static dataflow graphs.

Figure 4.1 depicts a DL program written in Python, of which semantics are difficult to be captured in a dataflow graph **correctly** due to the following representative dynamic features of Python.

• **Dynamic control flow (DCF)** Conditional branches and iterative loop constructs have different execution paths depending on intermediate values.

Lines 5-7 of Figure 4.1 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 (DT)** 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 4.1 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 (IF)** Another useful feature for using Python is the ease of accessing and mutating global states within functions. In Figure 4.1, 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 issue when converting an imperative program; achieving the high **performance** of state-of-the-art symbolic graph execution DL frameworks is also a challenge on its own. State-of-the-art frameworks require additional information on dynamic types and control flow in order to optimize graph execution. However, a naïve, one-shot converter would be unable to extract this information from an imperative program before execution,

[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.

and thus is incapable of supplying frameworks with such hints. For instance, if the input `sequence` at Line 2 in Figure 4.1 is expected to always have a fixed length, then that information can be exploited to unroll the following loop at Line 5 when generating the corresponding dataflow graph. It is unclear how a naïve converter would do this without actually executing the program to check the loop length.

3.1.2 Related Works

Previous works that try to translate a Python DL program into a dataflow graph either fail to capture the important dynamic semantics of Python, or run in slower performance due to the lack of sufficient information at graph build time. Table 3.1 summarizes state-of-the-art DL frameworks alongside their execution models and their status regarding the coverage and efficiency of graph conversion support.

Tracing-based graph generation approaches such as PyTorch's JIT compiler (`torch.jit.trace`) [70], MXNet Gluon [66], and the `defun` [88] functionality of TensorFlow Eager [84] execute the imperative program once, and convert the single execution trace directly into a dataflow graph. Though 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, leading to incorrect computation results for dynamically changing execution paths, dynamic types of non-tensor or non-input expressions, or impure functions of Python at runtime. Moreover, these approaches currently do not give any feedback about incorrectly-converted control flows to users, making the problem even worse.

On the other hand, there exist other approaches that select a less-dynamic host language and therefore succeed in capturing the wider semantics of source

programs. JAX [30] limits the Python syntax and supports converting only pure-and-statically-composed functions. S4TF [86] supports Swift, losing the merit of supporting Python, the de-facto standard programming language for DL programming, and introduces new programming models that most DL researchers are unfamiliar with. Moreover, since the graph conversion occurs before actually executing the program, these approaches can miss the opportunity to further optimize the graph with the information only obtainable during the program execution. For example, always converting a Python loop into control flow operations can be sub-optimal if the loop iteration count is known to be fixed.

Concurrent works including AutoGraph-enabled TensorFlow `defun` functionality [62] and the "scripting" mode of PyTorch JIT (`torch.jit.script`) [70] also have limitations. AutoGraph makes users to explicitly provide the necessary information, or generates incorrect or sub-optimal graph in some cases, all of which could be avoided if sufficient information existed. For example, users must explicitly specify the types of Python lists, prohibiting the dynamic typed or heterogeneous elements. For another example, for dynamic control flow statements, the statements with non-tensor predicates are always unrolled, which is error-prone, and the statements with tensor-typed predicates are always converted to control flow operations, which can be sub-optimal. In the "scripting" mode of PyTorch JIT, users must use TorchScript, a subset of Python which does not allow variables to have dynamic types. Further graph optimizations based on the runtime information are also not possible.

3.2 Proposed Solution: Speculative Graph Generation and Execution

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 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 approach is a good fit since the interpreter is likely to execute specific code blocks of the program repeatedly.

Unlike the JIT compilers of managed language runtimes, however, the goal of Janus is not to optimize the host language execution itself. In fact, when running imperative DL programs, the execution time of the language runtime is usually much shorter compared to the execution time of the mathematical operations for DL, such as convolution or matrix multiplication. However, since these mathematical operations are usually implemented in separate low-level language like C++, existing JIT compilers of managed language runtimes would execute them just as separated function invocations. Under such an execution model, it is impossible to see the multiple mathematical operations at once and apply compiler optimizations or execute them in parallel. On the other hand, Janus understands the function invocations for such mathematical operations, and converts them into appropriate target graph operations, which can be optimized and be executed efficiently by symbolic graph executors.

3.3 Janus System Design

In this section, we introduce Janus, a DL framework that receives an imperative DL program and either executes it as is directly, or generates a symbolic graph version of the program and executes the graph instead.

The input program for Janus is assumed to be written using the API and the programming model of existing imperative DL frameworks like TensorFlow Eager [84]. Given an input program, Janus extracts the main neural network computation part, over which the automatic differentiation is performed, and starts the speculative graph generation and execution process. From the user's point of view, the whole graph conversion and execution process is done transparently; in other words, the given DL program is automatically transformed into a corresponding graph representation without any interactions.

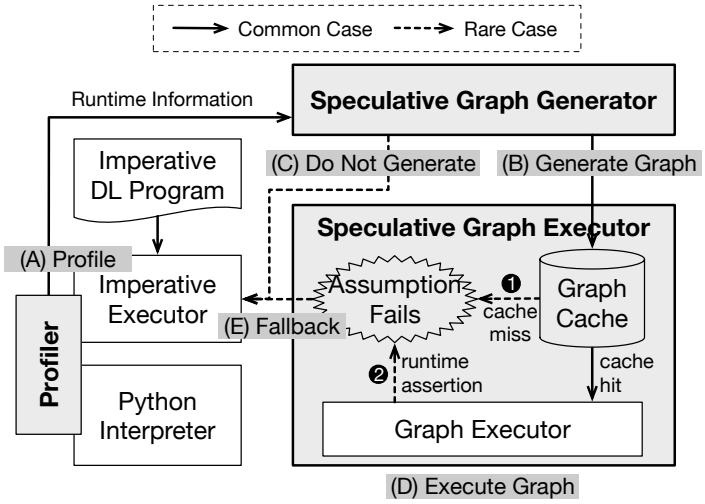


Figure 3.2: An illustration of the execution model of Janus, showing how a DL program is processed by several components. *Profiler* observes imperative program execution and collects information to make the realistic assumptions. *Speculative Graph Generator* generates dataflow graphs from the program and hands the optimized graphs over to Speculative Graph Executor. The *Speculative Graph Executor* actually runs the generated graph and handles assumption failures.

Figure 3.2 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.

3.3.1 Fast Path for Common Cases

Runtime profiling. Once Janus receives a DL program, the program is first executed imperatively, while the *Profiler* gathers runtime information required for making reasonable assumptions (Figure 3.2 (A)). Various information is collected, including control flow decisions on conditional branches, loop iteration counts for iterative loop constructs, variable type information, non-local variables, object attributes, and so on.

Symbolic graph generation. After a sufficient amount of information has been collected, the *Speculative Graph Generator* tries to convert the program into a symbolic dataflow graph with the assumptions based on the runtime information (Figure 3.2 (B)). To avoid making any hasty generalizations, Janus does not begin graph generation until the executor has profiled the program for a certain amount of iterations.[3] First, Janus traverses the abstract syntax tree (AST) of the DL program and generates the corresponding graph elements for each AST node, along with assertion operations that can validate the context assumption at runtime. Since Janus targets DL programs, operations for automatic differentiation and model parameter updates are also automatically inserted if necessary. Next, the generated graph is further optimized by the post-processor, of which optimizations were not applicable to the original imperative DL program. Finally, the optimized graph and the assumption that

[3]We found that 3 iterations were enough to come up with a decent program context assumption, for our experimental workloads.

were used to generate the graph are saved into the graph cache.

Graph execution. If a graph representation with correct assumptions regarding the program context is available, the *Speculative Graph Executor* executes the symbolic graph (Figure 3.2 (D)). Note that the same graph can be reused multiple times, given that the runtime context assumption holds for future invocations.

3.3.2 Accurate Path for Rare Cases

Assumption failure. Handling the assumptions is important to guarantee the correctness of the converted graph. If an assumption is proven to be wrong, the associated graph cannot be executed for the current runtime as it may produce incorrect results. Instead, Janus falls back to the imperative executor (Figure 3.2 (E)) and resumes runtime profiling to make more relaxed assumptions for subsequent executions.

Assumptions that can be validated before actually executing the associated graph, such as type assumptions on input arguments, are checked when retrieving the graph from the graph cache (Figure 3.2 ①). In the unfortunate case where such an assumption is wrong, Janus regards this as a cache miss and falls back to imperative execution.

On the other hand, for assumptions that can only be validated during graph execution (Figure 3.2 ②), it can be erroneous to simply abort the current execution to fall back to the imperative executor, because the global state may have been changed during the current execution. To solve this issue, Janus defers state update operations until every assumption is validated (Section 3.4.2). This way, even if an assumption turns out to be wrong during computation, no state update operation has been triggered yet and thus no state has been

mutated. Knowing this, the system can safely stop the current execution. In other words, states are updated in an all-or-nothing manner.

In order to validate an assumption, a runtime assertion is encoded into the symbolic graph as an operation called `AssertOp`. The `AssertOp` aborts the graph execution if the given condition fails. It also reports which assumption has been broken, and this information is used to give up further optimizations that rely on the assumptions that repeatedly break. If the same assumption breaks too frequently, Janus discards all symbolic dataflow graphs that were generated based on the assumption from the graph cache, and does not optimize this part of the program again.

Imperatively executed programs. With Turing-complete graph representations, any Python program can be represented as a symbolic graph, in theory. However, the *Speculative Graph Generator* does not convert every single Python feature into a symbolic graph operation (Figure 3.2 (C)). For example, to ensure the all-or-nothing characteristic of state updates, programs that include invisible state mutations are not converted into symbolic graphs. Some complicated Python features such as *coroutine*s and *generator*s are also not converted, since they do not have any clear graph representations. Section 3.4.4 describes the design choices and current limitations of the *Speculative Graph Generator* in terms of Python coverage. In spite of such limitations of the *Speculative Graph Generator*, however, it is worth noting that Janus users can still freely use the all features of Python on the imperative executor.

3.4 Symbolic Graph Generation

In this section, we describe in detail how Janus converts an imperative DL program into a symbolic dataflow graph. We start the section by showing the con-

version process of a basic DL program free of dynamic features (Section 3.4.1). Next, we explain how Janus converts dynamic features of Python, including dynamic control flow, dynamic types, and impure functions, into symbolic graph operations (Section 3.4.2). Janus uses the runtime information to simplify the dynamic program and treat it as a program of only static aspects, which is then easily transformed into a static graph. Finally, we discuss the Python coverage limitations of the *Symbolic Graph Generator* (Section 3.4.4). More thorough discussion about the Python coverage of Janus is in Appendix **??**.

For simplicity, we describe our design using various operations of TensorFlow [8], a widely-used DL framework. However, our design is not necessarily coupled with TensorFlow and can be applied to other DL frameworks.

3.4.1 Graph Generation Basics

Figure 3.3(a) is a simple, imperative Python program that calculates a linear model, written as a pure function without any dynamic control flow or arbitrary Python objects. We use this program as an example to show the basic graph conversion process.

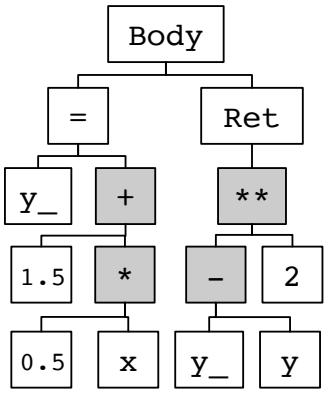
Input parameters (`x` and `y`) are converted into graph input objects that require external inputs in order to execute the graph. In the case of TensorFlow, this corresponds to `PlaceholderOp`[4]s. At runtime, they are filled with the actual argument values. The return value of the `return` statement 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

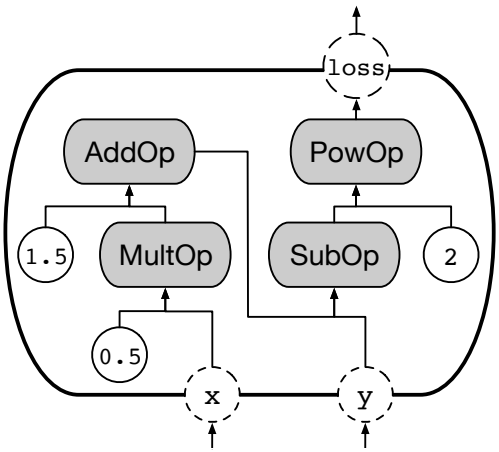
[4] `PlaceholderOp`s are unique operations that generate errors unless they are provided with external inputs before graph execution. TensorFlow expects users to feed a dictionary `{ph1: v1, ph2: v2, ...}` to a `PlaceHolderOp`.

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

(a) Source code of a DL program calculating a linear model



(b) AST of `loss_fn`



(c) Generated graph from `loss_fn`

Figure 3.3: The Python source code, AST, and symbolic graph of a simple linear model that receives several external inputs. The static features of the program are represented as nodes in the AST, which in turn are converted to vertices of the symbolic graph.

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 calculation 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 [67]. Figures 3.3(b) and 3.3(c) illustrate how such a connection is made for the variable `y_` in Figure 3.3(a), along with the rest of the program.

3.4.2 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 dataflow graphs while maintaining the same 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.

Dynamic Control Flow

Basic translation rules. Among various dynamic control flow statements, Janus focuses on conditional branches, loop constructs, and function calls, similar to Pydron [67]. As shown in Pydron, these three constructs are enough to express most complex dynamic control flows in Python. Furthermore, they can all be expressed using special control flow graph operations proposed in recent works [100, 43] as follows.

Python's conditional statement, the `if` statement, can be obtained by com-

bining *switch* and *merge* primitives. The switch and merge primitives, originating from classic dataflow architectures [23, 21, 11], act as demultiplexers and multiplexers, respectively, selecting a single path to pass their inputs or outputs. In TensorFlow, the `SwitchOp` and `MergeOp` [100] serve as symbolic dataflow graph counterparts for these primitives, allowing Janus to plant conditional branches in graphs.

The iterative statements of Python, `while` and `for`, are handled by using the switch and merge primitives together with loop context primitives that hold iteration *frame*s. TensorFlow conveniently provides `EnterOp`, `ExitOp`, and `NextIterationOp` [100] for creating iteration frames and passing values over them.

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 a TensorFlow implementation of this operation called `InvokeOp` [43], which can represent an invocation of a recursive function with automatic differentiation support.

Speculative graph generation: unrolling and inlining. If Janus detects that only a single particular path is taken for a certain control flow statement during profiling, 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. 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. If the assertion operation fails, Janus falls back to imperative execution.

To be more specific, for conditional branches, if the program takes only one side of the branch during profiling, Janus generates that particular side of the branch in the final graph without any switch or merge primitives and adds an assertion operation that can detect a jump to the other side of the branch. For iterative statements, if the number of iterations of a loop is discovered to be fixed, Janus unrolls the loop with this fixed iteration count, and adds an assertion operation to check that the number of iterations is indeed correct.

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 which behave the same as the original callee functions, based on the prior knowledge about their behaviors. Section 3.4.4 includes more details and limitations about such function calls.

Dynamic Type

Basic translation rules. The types of all expressions within a Python program must be known before Janus can convert 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 [97] arrays into corresponding tensors, and converts non-numerical values, including arbitrary class instances, into integer-typed scalar tensors which hold pointers to the corresponding Python values. Next, Janus infers the types of other expressions that are derived from expressions covered by the basic rule.

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 dataflow graph. With statically determined shapes or values, the graph can be further optimized, or even be compiled to the efficient machine code [90].

Figure 3.4 shows an example hierarchy of shapes and values that a certain tensor may have. After profiling the first few runs, Janus finds out that even though the values of the tensor are different every time, they all have the same shape, for example (4, 8), as in the figure. Janus exploits this information to

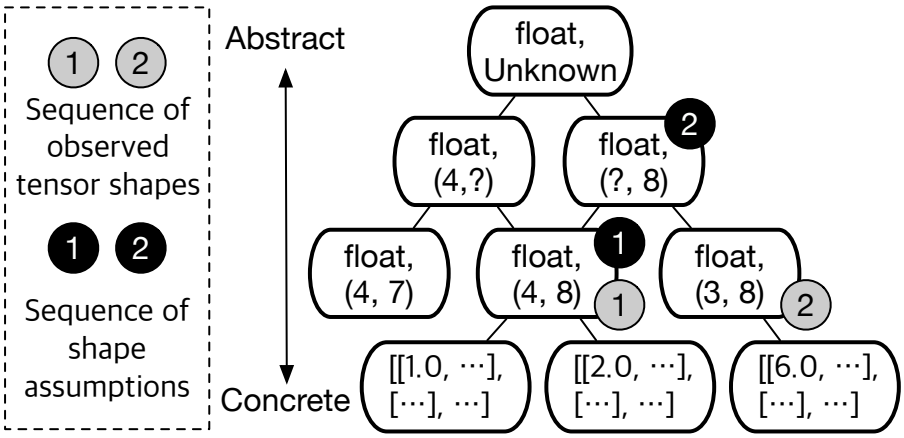


Figure 3.4: Type, shape, and value specialization hierarchy for an example tensor.

generate a dataflow graph with an assumption that the shape of this tensor is (4, 8). When the assumption fails, Janus tries to relax the assumption. For instance, in case the tensor has a shape (3, 8) for the next iteration to process a different size of mini-batch, Janus modifies the assumption to suit both shapes (4, 8) and (3, 8), resulting in another dataflow graph with a shape assumption of (?, 8). The system does not have to repeat the graph generation process for a possible future case in which the example tensor has yet another unpredicted shape of (2, 8) or (6, 8).

Impure Functions

Naïve 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.

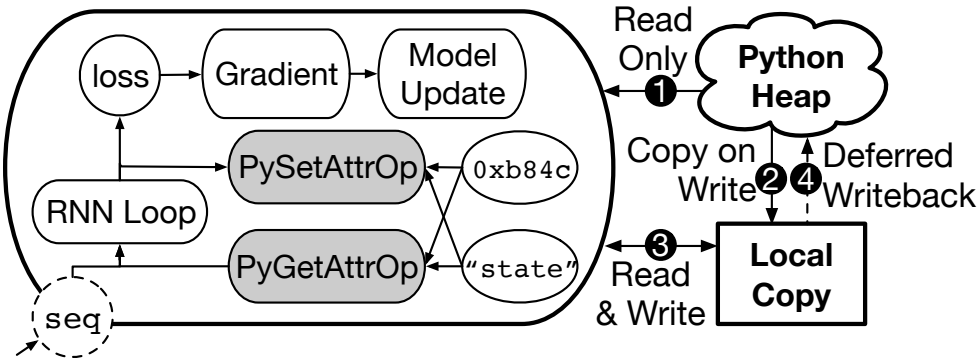


Figure 3.5: Symbolic dataflow graph generated graph from Figure 4.1 and the global states.

A trivial solution is to use TensorFlow's `PyFuncOp`s, 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 reduced 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.

Optimized 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, and mutates only the local copies during symbolic graph execution.

Figure 3.5 shows the symbolic dataflow graph version of the program in Figure 4.1, 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 value is inserted to the local copy instead of directly updating the Python heap. ③ Further read and write operations are redirected to the local copies. Note that Janus inserts appropriate dependencies between `PyGetAttrOp`s and `PySetAttrOp`s if necessary to prevent any data hazards. ④ After the graph executor finishes this run, the local copies 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`.

By not mutating the Python heap directly, Janus can always bypass the Python GIL to execute more read and write operations in parallel. In addition, the fallback mechanism of Janus can be simplified thanks to the all-or-nothing based state update mechanism. It is notable that as the input and output of `PyAttrOp`s are not mathematically related, the partial derivative of `PyAttrOp` does not have to be calculated.

Opcode	Num	Description	Section Ref.
POP_TOP, ROT_TWO, ROT_THREE, DUP_TOP, DUP_TOP_TWO, NOP, EXTENDED_ARG	7	stack manipulation	No conversion is necessary
LOAD_CONST	1	constant	Section 3.4.1
UNARY_INVERT, UNARY_NEGATIVE, UNARY_NOT, UNARY_POSITIVE, BINARY_ADD, BINARY_AND, BINARY_FLOOR_DIVIDE, BINARY_LSHIFT, BINARY_MATRIX_MULTIPLY, BINARY_MODULO, BINARY_MULTIPLY, BINARY_OR, BINARY_POWER, BINARY_RSHIFT, BINARY_SUBTRACT, BINARY_TRUE_DIVIDE, BINARY_XOR, INPLACE_ADD, INPLACE_AND, INPLACE_FLOOR_DIVIDE, INPLACE_LSHIFT, INPLACE_MATRIX_MULTIPLY, INPLACE_MODULO, INPLACE_MULTIPLY, INPLACE_OR, INPLACE_POWER, INPLACE_RSHIFT, INPLACE_SUBTRACT, INPLACE_TRUE_DIVIDE, INPLACE_XOR, COMPARE_OP	31	mathematical operators	Section 3.4.1
LOAD_FAST, STORE_FAST, DELETE_FAST, UNPACK_SEQUENCE, UNPACK_EX	5	local variables	Section 3.4.1
JUMP_ABSOLUTE, JUMP_FORWARD, JUMP_IF_FALSE_OR_POP, JUMP_IF_TRUE_OR_POP, POP_JUMP_IF_FALSE, POP_JUMP_IF_TRUE, POP_BLOCK, GET_ITER, FOR_ITER, BREAK_LOOP, CONTINUE_LOOP, SETUP_LOOP	12	dynamic control flow	Section 3.4.2
CALL_FUNCTION, CALL_FUNCTION_KW, CALL_FUNCTION_VAR, CALL_FUNCTION_VAR_KW, RETURN_VALUE, MAKE_FUNCTION	6	function call	Section 3.4.2, Section 3.4.4
LOAD_ATTR, STORE_ATTR, DELETE_ATTR	3	arbitrary object	Section 3.4.2, Section 3.4.2
BUILD_LIST, BUILD_LIST_UNPACK, LIST_APPEND, BUILD_MAP, BUILD_MAP_UNPACK, BUILD_MAP_UNPACK_WITH_CALL, MAP_ADD, BUILD_SET, BUILD_SET_UNPACK, SET_ADD, BUILD_SLICE, BUILD_TUPLE, BUILD_TUPLE_UNPACK, BINARY_SUBSCR, STORE_SUBSCR, DELETE_SUBSCR	16	list, set, map	Section 3.4.2, Section 3.4.2
LOAD_GLOBAL, LOAD_DEREF, LOAD_NAME, STORE_GLOBAL, STORE_DEREF, STORE_NAME, DELETE_GLOBAL, DELETE_DEREF, DELETE_NAME, LOAD_CLOSURE, MAKE_CLOSURE	11	non-local variables	Section 3.4.2
POP_EXCEPT, SETUP_EXCEPT, SETUP_FINALLY, RAISE_VARARGS, END_FINALLY	5	exception handling	Section 3.4.3
SETUP_WITH, WITH_CLEANUP_FINISH, WITH_CLEANUP_START	3	`with`	Section 3.4.3
YIELD_FROM, YIELD_VALUE, GET_YIELD_FROM_ITER	3	`yield`	Section 3.4.4
IMPORT_FROM, IMPORT_NAME, IMPORT_STAR	3	in-line `import`	Section 3.4.4
LOAD_BUILD_CLASS, LOAD_CLASSDEREF	2	in-line class definition	Section 3.4.4
GET_AITER, GET_ANEXT, GET_AWAITABLE, BEFORE_ASYNC_WITH, SETUP_ASYNC_WITH	5	coroutine	Section 3.4.4
Total	113		

Table 3.2: The mapping of the full list of CPython opcode and the corresponding sections.

3.4.3 Python Syntax Coverage

Table 3.2 describes the entire set of opcode in the CPython [71] 3.5.2 interpreter, and maps them to the sections which describe the corresponding graph generation rules. Python programs whose opcodes are mapped to Section 3.4.4 can only be executed on the imperative executor, and the others can be executed on the graph executor. Python features that are not covered in previous sections are briefly discussed in the rest of this section.

Exceptions. A Python `raise` statement can be represented as an `AssertOp` in the dataflow graph. When the `AssertOp` for an exception aborts the graph execution, the fallback occurs, and the actual, Python-style exception can be safely raised on the imperative executor. Under the same principle, for `try-except-finally` statements, only the `try-finally` part is converted into the graph elements, and the `except` part is simply not converted, since the exception will never be caught by the symbolic graph. By avoiding exception handling inside the symbolic graph, we can protect users from having to debug through symbolic graph execution traces, which are relatively more complicated than imperative execution traces.

Context manager. Since exception handling always occurs on the imperative executor as described in the previous paragraph, the `with` statement can be converted into the simple function calls to `__enter__` and `__exit__` of the corresponding context manager object.

3.4.4 Imperative-Only Features

Albeit being able to support a wide range of imperative DL programs, the current Janus graph generator does not convert some particular features of

Python into dataflow graph elements. Programs with such features are executed only on the imperative executor.

Coverage Limitations from Design

Alignment with the design principles. To be aligned with the design of Janus in previous sections, the Janus graph generator does not convert some features of Python. For example, to keep the implementation of local copies of global state simple (Section 3.4.2), Python objects with custom accessor functions (e.g., `__setattr__`) are not supported by the Janus graph generator. Also, a function should always return the same type of value, to infer the type of call expressions (Section 3.4.2).

External function calls. Janus must understand the behavior of the external functions, i.e., the framework-provided functions or foreign functions[5], to convert them into corresponding graph operations. The Janus graph generator converts the external functions into the graph operations based on a separate whitelist. Most of the framework-provided functions such as `matmul` or `conv2d`, and many commonly-used Python built-in functions such as `print` or `len` are included in this whitelist. We plan to cover more functions in the Python standard library.

Janus handles such external functions with extra caution to ensure correctness. First, since the underlying assumption here is that the implementation of external functions never changes, Janus prohibits the modification of the functions included in the whitelist. Also, if an external function includes state mutation (e.g., `assign()` in TensorFlow), the execution of the corresponding graph operation is deferred until all the other assumptions are validated, under

[5]functions written in the languages other than Python

the same principle about the deferred state update in Section 3.4.2.

Coverage Limitations from Implementation

Currently, Janus does not cover a few features from Python that do not have clear graph representations. Such Python features include *coroutine*s, *generator*s, in-line class definitions and in-line import statements. We plan to support these features as future work.

3.5 Implementation

We implemented Janus on top of TensorFlow [8] 1.8.0 and CPython [71] 3.5.2. Janus exploits the existing TensorFlow graph executor and TensorFlow Eager imperative executor as its components. In this section, we explain the modifications to existing systems, and then describe how Janus supports data-parallel training.

Modifications to existing systems. TensorFlow has been modified for several reasons. First, to transparently separate out the neural network computation from the rest of the Python program without extra user intervention, the automatic differentiation functionality of TensorFlow Eager is modified to trigger Janus graph conversion. Second, to share the model parameters between eager mode and graph mode, Janus slightly modifies the parameter storing mechanism of TensorFlow Eager. Third, several custom operations had been added, including the `InvokeOp` and `PyAttrOp` as described in earlier sections.

CPython has also been modified to have bytecode-level instrumentation functionality for non-intrusive profiling. Without modifying the interpreter, instrumentation for the profiling should exist at the Python source-code level, which would significantly affect the performance and the debuggability of the

imperative execution.

Data-parallelization on Janus. Using multiple machines equipped with multiple GPUs is a common approach for accelerating DL jobs. We integrate Janus with Horovod [75], a distributed training module for TensorFlow that encapsulates the MPI collective communication [33] (e.g. AllReduce and AllGather) as an operation inside the symbolic graph. After converting an imperative program into a dataflow graph, Janus inserts appropriate communication operations to the graph in order to get the average of gradients generated by multiple workers. Since the generated dataflow graph contains both communication and computation operations, we can parallelize their execution and therefore achieve higher throughput.

3.6 Evaluation

We present experimental results that show how imperative DL programs can be executed both correctly and efficiently when converted into symbolic graphs on Janus.

3.6.1 Experimental Setup

Frameworks. As baseline frameworks representing symbolic graph execution frameworks and imperative execution frameworks respectively, we use TensorFlow [8] and TensorFlow Eager [84]. We could run the same DL program on Janus as on TensorFlow Eager, thanks to the transparent graph conversion feature of Janus. In addition, to demonstrate the correctness of graph conversion of Janus, we also compare Janus with TensorFlow `defun` [88], which implements a trace-based graph conversion mechanism. TensorFlow-based frameworks have been chosen to avoid implementation-dependent performance differences.

Applications. We have evaluated Janus with 11 models in five major neural network types, covering three convolutional neural networks (*CNN*; LeNet [50], ResNet50 [34], Inception-v3 [82]), two recurrent neural networks (*RNN*; LSTM [101], LM [46]), two recursive neural networks (*TreeNN*; TreeRNN [80], Tree-LSTM [83]), two deep reinforcement learning models (*DRL*; A3C [61], PPO [74]), and two generative adversarial networks (*GAN*; AN [31], pix2pix [41]) as shown in Table 3.3. The datasets and the mini-batch sizes used for evaluation are also specified in the table.

These models are implemented in an imperative programming style, using a number of dynamic features in Python as shown in Table 3.3. First, large CNN models such as ResNet50 and Inception-v3 have conditional statements for handling batch normalization [39], which make them behave differently under particular conditions when training and evaluating the model. Next, RNNs include Python `for` loops, and they also include global state mutation statements to retain hidden states inside the models. Next, TreeNNs[6] require all three kinds of dynamic features. They include recursive function calls, and conditional statements to separate recursion base cases and inductive cases. They also include values with undecided type; the return type of a recursive function is unknown until the function returns certain values. In addition, they include the Python object access to fetch the information of the current subtree. For DRL models[7], Python `for` loops are used for handling an arbitrary length of the states of an episode, and global state mutation statements are used for storing the intermediate computation results to monitor the progress of the training. GAN models also use global state mutation statements for the same reason. All

[6]The implementation of TreeNN models on TensorFlow follows the recursion-based implementation with `InvokeOp` [43], and Janus converts an imperative Python program into similar recursion-based graphs.

[7]The DL framework only handles model training and policy evaluation, and the environment simulation is handled by an external library [14].

Category	Model	DataSet	BS	DCF	DT	IF
CNN	LeNet	MNIST [51]	50	×	○	×
	ResNet50	ImageNet [73]	64	○	○	×
	Inception-v3	ImageNet [73]	64	○	○	×
RNN	LSTM	PTB [101]	20	○	○	○
	LM	1B [16]	256	○	○	○
TreeNN	TreeRNN	SST [81]	25	○	○	○
	TreeLSTM	SST [81]	25	○	○	○
DRL	A3C	CartPole [14]	20	○	○	○
	PPO	Pong [14]	256	×	○	○
GAN	AN	MNIST [51]	128	×	○	○
	pix2pix	Facades [96]	1	×	○	○

Table 3.3: Categories, models, datasets, batch sizes ("BS"), and the dynamic features of the applications used for evaluation.

models use Python function calls, including Python class methods of high-level DL programming APIs such as Keras [19]. Training data instances fed into each neural network have different shapes over different training iterations, when the length of the dataset cannot be divided by the batch size.

Environments. A homogeneous GPU cluster of 6 machines, connected via Mellanox ConnectX-4 cards with 100Gbps InfiniBand is used for evaluation. Each machine is equipped with two 18-core Intel Xeon E5-2695 @ 2.10 GHz, and 6 NVIDIA TITAN Xp GPU cards. Ubuntu 16.04, Horovod 0.12.1, CUDA 9.0, cuDNN 7, OpenMPI v3.0.0, and NCCL v2.1 are installed for each machine.

LeNet, LSTM, AN, and pix2pix models are evaluated on a single GPU, since these models and the datasets are regarded to be too small to amortize the communication cost of parallel execution. Similarly, TreeRNN, Tree- LSTM, and A3C models are evaluated on CPUs on a single machine, since these models and datasets are regarded to be too small to amortize the communication between CPU and GPU. The other models are evaluated using multiple GPUs.

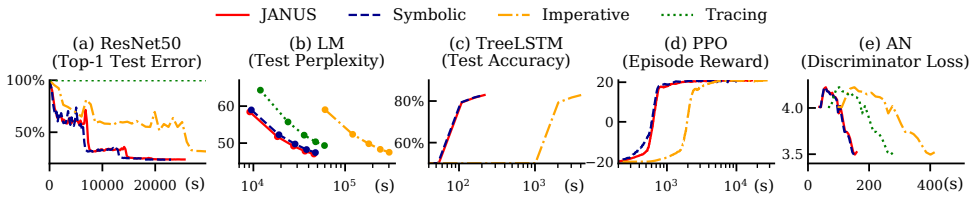


Figure 3.6: (a) The test error of ResNet50, (b) validation perplexity of LM, (c) test accuracy of TreeLSTM, (d) episode reward of PPO, and (e) discriminator loss of AN measured on Janus, TensorFlow (Symbolic), TensorFlow Eager (Imperative), and TensorFlow `defun` (Tracing) according to the elapsed time in seconds. Each marker in (b) represents each training epoch, describing that per-epoch convergence is slower on TensorFlow `defun` compared to other frameworks.

ResNet50 and Inception-v3 models are evaluated using up to 36 GPUs, and LM is evaluated on up to 12 GPUs. The network bandwidth made the throughput of LM saturated on more than 2 machines with MPI collective communication, due to the huge parameter size of LM (0.83 billion parameters). Therefore, model convergence of LM is experimented with 6 GPUs. We evaluated the model convergence of PPO using 4 GPUs on a single machine, since the number of parallel actors used in the original paper was only 8.

3.6.2 Model Convergence

Figure 3.6 shows how the neural networks converge on various underlying frameworks, with ResNet50 with the ImageNet dataset, LM with the 1B dataset, TreeLSTM with the SST dataset, PPO with the Pong-v4 environment, and AN with the Facades dataset on four frameworks. For all evaluated models, Janus, TensorFlow, and TensorFlow Eager succeeded to make the neural networks converge correctly as reported in literatures: 23.7% top-1 error for ResNet50 after 90 epochs, perplexity 47.5 for LM after 5 epochs, 82.0% binary accuracy for Tree- LSTM after 4 epochs, 20.7 mean final score for PPO after 40M game

frames, and 3.52 discriminator loss for AN after 30 epochs.[8] Also, Janus could make the model to converge up to 18.7 times faster than TensorFlow Eager, while executing the identical imperative program. The performance difference between Janus and TensorFlow was within 4.0%.

On the other hand, trace-based TensorFlow `defun` failed to make the models to converge correctly. The ResNet50 model includes the conditional statement to distinguish the behavior of the batch-normalization [39] layer on model training and evaluation. If a user evaluates the initial accuracy before training the model by manipulating the model object attribute, TensorFlow `defun` converts the first execution trace into graph operations, which silently leads to an inaccurate result. Similarly, the LM model does not converge properly with TensorFlow `defun`, since it failed to capture state passing across sequences, due to its trace-based conversion mechanism. The TreeLSTM model could not be converted into the symbolic graph at all with TensorFlow `defun`, since it does not support recursive function call. We could not get the convergence metrics for PPO model with TensorFlow `defun`, as it does not support global state update statements. TensorFlow Eager converges slowly, since its training throughput is much lower than TensorFlow and Janus. We next analyze the training throughput of the frameworks, excluding TensorFlow `defun`, which fails to make models converge correctly.

3.6.3 Training Throughput

Single-machine Throughput

Table 3.4 presents the training throughput of all models executed with Janus, TensorFlow Eager, and TensorFlow on a single machine with a single GPU. As shown in the table, Janus outperforms TensorFlow Eager (imperative execution)

[8] We measured the training loss with the official implementation in Tensorflow Eager [85].

Model	(A) Imp.	(B) Janus	(C) Sym.	$\frac{(B)}{(A)}$	$\frac{(B)}{(C)}-1$
LeNet	7.94k	25.84k	26.82k	3.25x	-3.6%
ResNet50	188.46	200.37	207.39	1.06x	-3.4%
Inception-v3	108.36	119.32	124.33	1.10x	-4.0%
LSTM	2.75k	22.06k	22.58k	8.03x	-2.3%
LM	19.02k	40.18k	40.45k	2.11x	-0.7%
TreeRNN	20.76	988.72	928.66	47.6x	+6.5%
TreeLSTM	7.51	138.12	141.71	18.4x	-2.5%
A3C	220.66	1132.9	1178.6	5.13x	-3.9%
PPO	596.80	1301.0	1306.4	2.18x	-0.4%
AN	4.34k	11.33k	11.56k	2.61x	-2.1%
pix2pix	4.04	8.69	8.88	2.15x	-2.1%

Table 3.4: Training throughput of all models evaluated on a single machine with a single GPU in Janus, TensorFlow (Sym.), and TensorFlow Eager (Imp.). The numbers represent processed images/s for CNN and GAN models, processed words/s for RNN models, processed sentences/s for TreeNN models, and processed frames/s for DRL models.

by up to 47.6 times, and shows throughput similar to TensorFlow (symbolic graph execution) by up to 4.0% performance degradation. Janus even performs slightly better (+6.5%) for TreeRNN, since there is no need to pre-process the input sentences, which are the tree-structured Python objects.

Janus achieves bigger performance gains on RNNs, TreeNNs, DRLs, and GANs than on CNNs, since those networks have many concurrently executable operations. In addition, the performance gain of Janus on a single machine is larger on models with fine-grained graph operations such as LeNet, LSTM, TreeRNN, A3C, and AN, compared to the models with coarse-grained operations such as ResNet50, Inception-v3, LM, PPO, and pix2pix, since the gain from bypassing the Python interpreter and applying compiler optimizations is bigger when the computation time of each operation is short.

For large CNN models such as ResNet50 and Inception-v3, optimized GPU kernel computation accounts for most of the computation time, which makes

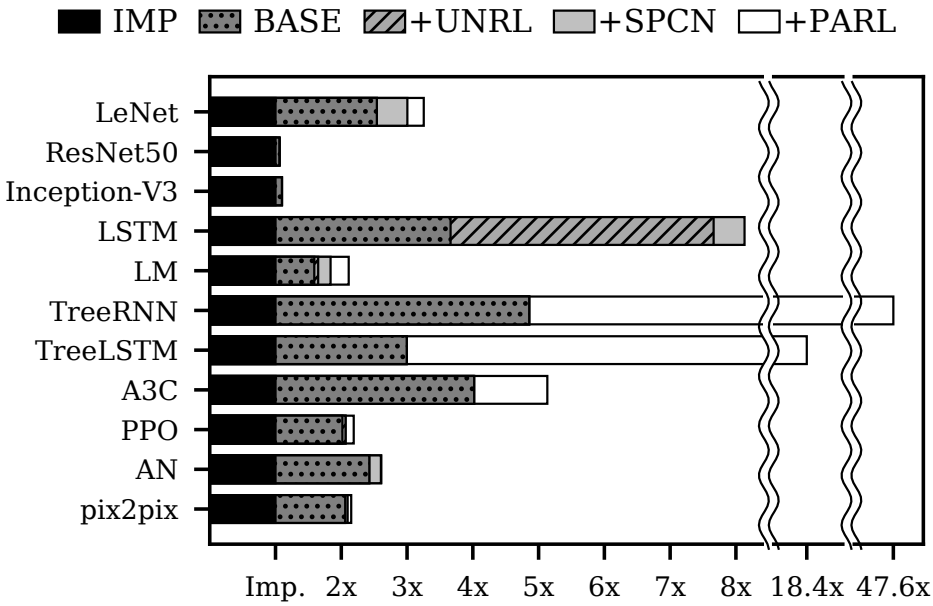


Figure 3.7: The contribution of optimizations to improve training throughput. Optimizations are cumulative. **IMP**: Imperative, **BASE**: Janus without following optimizations, **+UNRL**: control flow unrolling, **+SPCN**: type specialization, **+PARL**: graph executor with 72 threads in threadpool (Janus)

the performance difference among Janus, TensorFlow, and TensorFlow Eager relatively small.

Optimization effect. Figure 3.7 analyzes the cause of the performance improvement of Janus in detail. Converting the imperative program into the symbolic graph without any following optimizations (**BASE**) enabled up to 4.9x performance improvement compared to the imperative execution (**IMP**). It removes the Python interpreter and framework code overhead, which has the bigger effect when each graph operation is relatively smaller. Control flow unrolling (**+UNRL**) and type specialization (**+SPCN**) enable more aggres-

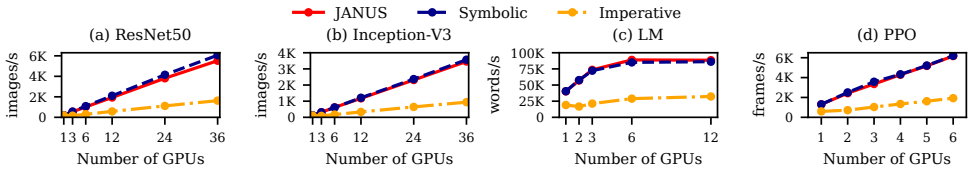


Figure 3.8: Training throughput for the ResNet50, Inception-v3, LM, and PPO models on Janus, TensorFlow (Symbolic), TensorFlow Eager (Imperative), using varying numbers of GPUs.

sive compiler optimizations. On RNNs, **+UNRL** improved the performance of LSTM and LM by 2.09x and 1.04x, respectively. The control flow statements in CNNs, TreeNNs and DRLs could not be unrolled due to their dynamicity. **+SPCN** enabled some compiler optimizations and improved the throughput up to 18.3% in small neural networks. Finally, executing multiple operations in parallel (**+PARL**) improved the throughput up to 9.81x. Especially higher gain could be achieved for TreeNNs, since there exist many operations that could be executed in parallel in multiple independent tree nodes.

We have also measured the effect of assumption validation, but the effect was negligible (in the error range), since the `AssertOp`s can be executed with the main neural network in parallel.

Scalability

Figure 3.8 shows the scalability of ResNet50, Inception-v3, LM, and PPO models on Janus, TensorFlow, and TensorFlow Eager on the cluster with 36 GPUs (12 GPUs for LM, 6 GPUs for PPO). We measured the scale factor, which is defined as *Multi-GPU Throughput / (Single-GPU Throughput* $\times$ *Number of GPUs)*. Janus achieves similar scalability (scale factor 0.77, 0.81, 0.18 each) as TensorFlow (0.81, 0.80, 0.18 each), but TensorFlow Eager does not scale well (0.24, 0.24, 0.14 each), due its inability to overlap computation and communi-

cation.

The performance difference between Janus and TensorFlow becomes smaller when the synthetic dataset is used, since the input processing of TensorFlow is highly optimized. The slight difference in the scalability of ResNet50 comes from the under-optimized input pipeline of TensorFlow Eager, which Janus also uses. Optimizing the input processing pipeline for Janus will further reduce the performance difference between Janus and TensorFlow. We leave this optimization as future work.

3.7 Summary

In this chapter, we introduced 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 assuming that DL programs inherently have the static nature. To preserve the dynamic semantics of Python, Janus generates and executes the graph speculatively, verifying the correctness of such assumptions at runtime. Our experiments showed that Janus can execute various deep neural networks efficiently while retaining programmability of imperative programming.

Chapter 4

Imperative-Symbolic Co-Execution

4.1 Motivation

4.1.1 Limitations of Existing Approaches

As the usability of deep learning (DL) framework is getting more important, the imperative programming model has become an essential part of recent DL frameworks. PyTorch [70], which provides an imperative and Pythonic programming style as default, has been chosen by an increasing number of machine learning researchers in recent years [36]. And TensorFlow [8] has recently changed its default programming model from symbolic to imperative. However, optimizing individual operations in imperative programs has limited opportunities compared to optimizing them as a group in a symbolic graph format. For this reason, TensorFlow provides AutoGraph and PyTorch provides TorchScript for efficient execution. Still, existing approaches that convert imperative DL programs into optimized symbolic graphs cannot provide a general solution

```
def forward(self, input_ids, attention_mask=None, token_type_ids=None,
            position_ids=None, head_mask=None):
  if attention_mask is None:
    attention_mask = torch.ones_like(input_ids)
  if token_type_ids is None:
    token_type_ids = torch.zeros_like(input_ids)

  extended_attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
  extended_attention_mask = extended_attention_mask.to(dtype=next(self.parameters()).dtype)
  extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

  if head_mask is not None:
    if head_mask.dim() == 1:
      head_mask = head_mask.unsqueeze(0).unsqueeze(0).unsqueeze(-1).unsqueeze(-1)
      head_mask = head_mask.expand(self.config.num_hidden_layers, -1, -1, -1, -1)
    elif head_mask.dim() == 2:
      head_mask = head_mask.unsqueeze(1).unsqueeze(-1).unsqueeze(-1)
    head_mask = head_mask.to(dtype=next(self.parameters()).dtype)
  else:
    head_mask = [None] * self.config.num_hidden_layers

  embedding_output = self.embeddings(input_ids, position_ids=position_ids,
                                     token_type_ids=token_type_ids)
  encoder_outputs = self.encoder(embedding_output, extended_attention_mask,
                                 head_mask=head_mask)
  sequence_output = encoder_outputs[0]
  return (sequence_output, self.pooler(sequence_output),) + encoder_outputs[1:]
```

Figure 4.1: A part of a PyTorch implementation of the BERT [24] model

due to their limited program coverage.

4.1.2 Motivating Example

Listing 1 is a part of a PyTorch implementation [5] of the BERT [6] model, of which behavior cannot be fully represented with DL operations. For example, Python has "generator" functions which can be paused and resumed while being executed. A built-in function "next()" at Lines 8, 17 fetches a value from an iterator, by resuming a generator function until it is paused again to return an intermediate value. Such behavior is almost impossible to be represented with symbolic graphs. For another example, "None" value at Lines 1, 2, 4, and 19 cannot be correctly handled with type systems for existing symbolic DL frameworks. In fact, this style of programming is not uncommon. We analyzed 14 popular DL models implemented in PyTorch[1]; 12 out of 14 models use various features that do not have trivial graph representation, which make it difficult to use existing graph conversion frameworks like AutoGraph [62], TorchScript [1] and JANUS. AutoGraph and TorchScript may fail with exceptions, and JANUS runs in its imperative mode.

4.1.3 Proposing Solution

To address the above challenge, we propose Terra, an *imperative-symbolic co-execution* framework for imperative DL programs. Unlike previous approaches, we do not attempt to substitute a whole imperative program with a symbolic graph. Instead, we propose a novel approach that performs imperative and symbolic execution simultaneously to leverage the optimized symbolic graph and also preserve the behavior of the original imperative program. Given an

[1]Four CNN models (GoogleNet, ShuffleNet, DenseNet, ResNet) from the official implementations, six RNN models (OpenNMT, BERT, GPT2, XLNET, XLM, RoBERTa) from a popular repository, four GAN models (ACGAN, COGAN, Pix2Pix, CycleGAN).

imperative program, Terra executes the program on the Python interpreter but leaves out DL operations that compose the core computation logic for neural networks, including loss functions and gradient computation algorithms. The DL operations are converted into a symbolic graph, whose execution is delegated to a separate symbolic graph executor. The generated graph runs concurrently with the Python interpreter.

Although Terra does not require a symbolic graph representation that corresponds to a whole imperative program, it still needs a symbolic graph for the DL operations. To this end, Terra first executes the imperative program as-is for a few iterations and collects traces of executed DL operations. Terra then merges the collected traces into a single symbolic graph, not relying on static analysis of the source code as previous approaches did. In order to enable seamless execution between the skeleton imperative program and the delegated symbolic graph, Terra opens a communication channel between the Python interpreter and symbolic graph executor. Whenever the Python interpreter requires a computation result from a certain DL operation of the symbolic graph, or vice versa, it waits until the requested data are prepared and continues to execute as soon as the data transfer finishes.

4.2 Terra Overview

This section briefly describes how Terra adopts imperative-symbolic co-execution to overcome the limitations of existing approaches to execute imperative DL programs. Figure 4.2 shows the overall workflow of Terra with its components. Given an imperative DL program, Graph Extractor executes the program imperatively and extracts the trace of executed DL operations. Since the trace only covers a single program path out of various possible paths, Graph Extractor runs the program multiple times to collect several traces and merges the traces into

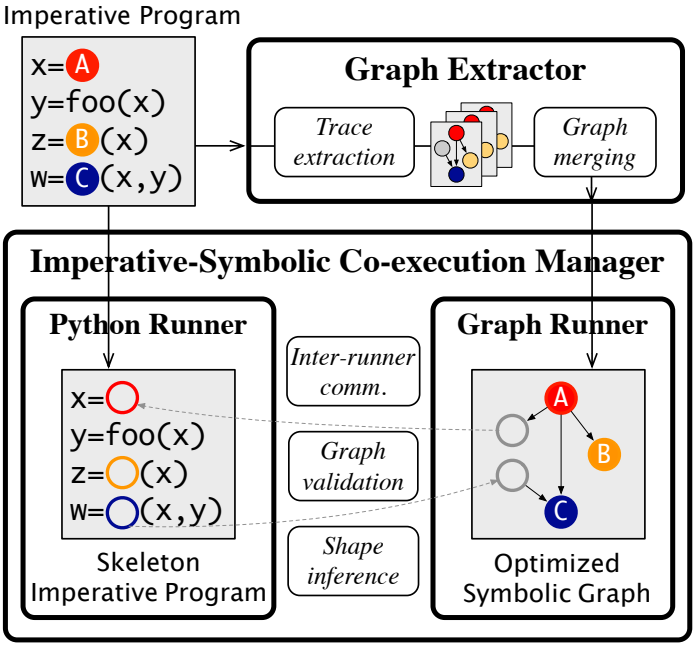


Figure 4.2: System overview

a single symbolic graph.

After the symbolic graph generation, Co-Execution Manager takes over the role of controlling the program execution. Co-Execution Manager is composed of Python Runner and Graph Runner. Python Runner executes the skeleton imperative program in which the original program's semantics is preserved except for the DL operations. It omits the DL operations and leaves them to the Graph Runner, which executes the symbolic graph generated by Graph Extractor concurrently with the skeleton imperative program. In case the Graph Runner requires data from the Python Runner or vice versa, Co-Execution Manager opens a communication channel between the two. Data from the skeleton program is passed to the symbolic graph through the graph source node and data from the graph to the skeleton program is passed through the graph sink node. Finally, since the generated symbolic graph may not cover all possible

program paths, Co-Execution Manager validates the graph by checking whether the skeleton program follows one of the paths that the program has taken during the trace extraction. If the program goes into a path that has never been taken, Co-Execution Manager invalidates the symbolic graph and informs Graph Extractor that it should collect more traces and re-generate the symbolic graph.

4.3 System Design

This section describes the design of Terra to efficiently handle DL programs.

4.3.1 Graph Merging

Instead of generating and optimizing a new graph for every gradient descent step, we build a general graph once that can be reused for the following steps. To build such a general graph that is applicable for multiple gradient descent steps that may take different control flow paths, we merge multiple execution traces of the same program to form a single graph. We use information from source code to correctly find out which part of the graph should be in a branch, and which part should be in a loop, etc. The merged graph may include control flow operations to cover dynamic control flow paths, but many control flows will be able to be unrolled in the graph using speculative graph generation and optimization techniques of JANUS.

4.3.2 Inter-runner Communication

It is sufficient to infer the shape and type of each DL operation's output in order to continue the execution of the skeleton program for most of the cases. However, in case where the actual value of a DL operation's output is required, e.g. to print the intermediate values for debug purpose, or to handle the data-dependent control flow, the Python runner may request the computation results

from the graph runner. Likewise, the graph runner may request input values from the Python runner. To handle such inter-runner communication, we append additional graph source and sink nodes to the original graph, and use proper synchronization between runners.

4.3.3 Graph Validation

Even if we try to generate a general graph to be reused for multiple steps, the graph may become invalid due to the dynamic nature of Python. For example, someone can even silently overwrite the behavior of the convolution operation. To address this problem, we check if the sequence of DL operations of each gradient descent step matches with the generated graph.

4.4 Implementation

Terra is implemented atop TensorFlow [8] 2.3.1 with 4K lines of code in C++ and 3K lines of code in Python. Although our implementation is based on TensorFlow, our approach is applicable to other DL frameworks that support both imperative and symbolic execution of DL models (e.g., PyTorch [70] and MXNet [17]). The Graph Runner of Terra uses `FuncGraph` as a symbolic graph representation and utilizes the execution model of `ConcreteFunction`, which TF `function` [88] also adopts.

4.5 Evaluation

4.5.1 Experiment Setup

Frameworks. We use TensorFlow [8] v2.3.1 as a baseline framework, and Terra is also built on TensorFlow v2.3.1. All evaluated DL programs are implemented with the imperative API of TensorFlow, which has become the standard interface since TensorFlow v2.0. Python-oriented approach of our experiment

executes the imperative programs as they are. For the Graph-oriented approach, we adopt TensorFlow TF `function` [88] with AutoGraph [**?**], which is the state-of-the-art Graph-oriented approach. We compile a single training step function by AutoGraph, then trace the function by TF `function`. We refer to this system setting as AutoGraph+ onward.

Environments. We conducted all the experiments on a single machine that is equipped with two 18-core Intel Xeon Gold 6254 @ 3.10 GHz and an NVIDIA TITAN Xp GPU. We used Ubuntu 18.04, CUDA 10.1, cuDNN 7.6 and Python 3.8.

Programs. For the experiments, we use imperative DL programs collected from open-source GitHub repositories. We use following programs in the evaluation: ResNet-50 [87], EfficientNet-B0 [87], DCGAN [87], CycleGAN [87], BERT-Base [26], BERT-CLS [49], GCN [32], MelGAN [93], YOLO-v3 [103], Music-Transformer [47], GPT [79], DropBlock [25], SlimmableNet [99], SDPoint [48], To collect various programs that use the TensorFlow imperative programming API, we search on GitHub with two keywords: "tf 2" and "tensorflow 2". The search results are sorted by "Most Stars", and the ones with less than 100 stars are filtered out for the credibility of the implementation. Among them, we excluded non-English-written, tutorial-purpose, and DL-irrelevant repositories. To make each program much more "imperative", we replace some symbolic APIs such as `tf.cond` and `tf.while_loop` by Python `if-else` and Python `while` statement respectively. In addition to the programs collected by the mechanical process described above, we collect more programs from GitHub to cover diverse types of DL programs. ResNet-50, EfficientNet-B0, DCGAN and CycleGAN are programs that train well-recognized image-classification and gen-

erative models[2], while DropBlock, SlimmableNet, and SDPoint correspond to programs that use dynamic training techniques[3].

4.5.2 Performance

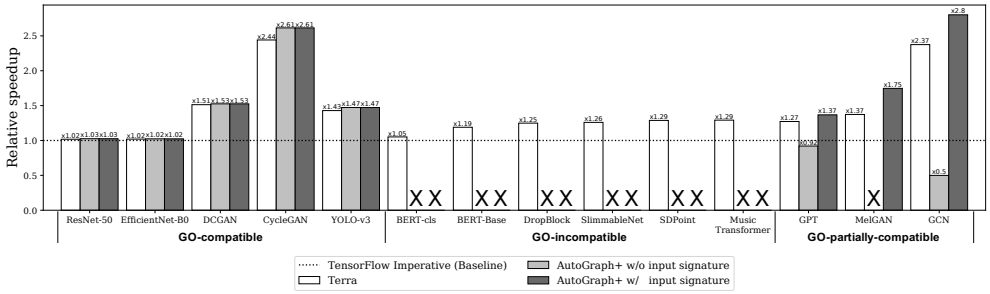


Figure 4.3: The training speedup of Terra and AutoGraph+ relative to TensorFlow imperative execution. The baseline is the training throughput of the imperative execution, which is presented as a dotted horizontal line.

Program	Stall time (%)	
	Graph Runner	Python Runner
ResNet-50	1.94	23.54
DCGAN	0.95	2.25
EfficientNet-B0	2.67	9.65
CycleGAN	7.96	0.58
MelGAN	6.22	16.99

Table 4.1: The ratio of the Graph runner stall time and the Python Runner stall time against the total execution time.

Figure 4.3 presents the training speedup of Terra and AutoGraph+ over the imperative execution. Terra outperforms the imperative execution in every

[2]These programs are from the official model garden [87] maintained by TensorFlow developers.

[3]Note that SlimmableNet and SDPoint are originally written in PyTorch. We conducted unbiased conversion based on the one-to-one mapping between the framework APIs (e.g. from PyTorch `nn.Conv2d` to TensorFlow `keras.layers.Conv2d`) without harming any of the original Python semantics and the code structures.

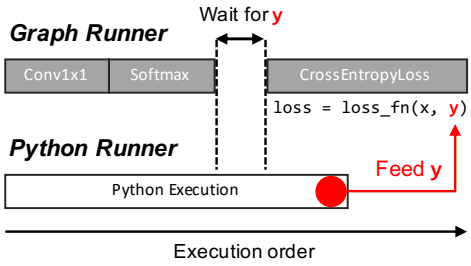


Figure 4.4: Due to the long Python execution path of TensorFlow, the Graph Runner stalls and waits for a label y to be fed from Python Runner. When the Graph Runner gets the label, it can resume the execution and calculate a loss.

evaluated program by up to 2.44 times. While AutoGraph+ cannot generate the symbolic graph for the GO-incompatible programs ("X" marked bars in Figure 4.3), Terra successfully runs all the programs with the performance gain (white bars). Moreover, Terra outperforms AutoGraph+ by up to 4.75 times when the input signature is not properly provided.

Stall time and performance. Since Terra executes the optimized symbolic graph, it can ideally obtain the same performance gain as AutoGraph+. In most programs, Terra achieves comparable performance to AutoGraph+, yet some of the programs like MelGAN and GCN show a relatively big performance gap. The reason of the performance gap is the system stall time caused by communication between the Python Runner and the Graph Runner.At the point when no operation can be executed until a value is fed from the Python Runner, the Graph Runner is stalled. For example, Figure 4.4 depicts the case that the Graph Runner waits for variable y to be fed from the Python Runner to calculate the loss.

To analyze the correlation between the system stall time and the performance, we profile the stall time of the each Runner as Table 4.1 shows. When

a program has a short Graph Runner stall time, Terra attains almost the same throughput as AutoGraph+ (ResNet-50, EfficientNet-B0, and DCGAN) no matter how long the Python Runner stall time is. In contrast, for the programs that show the longer Graph Runner stall time, the performance gap between Terra and AutoGraph+ goes bigger (CycleGAN, and MelGAN). Especially for the MelGAN, we found that the concurrency was broken in certain parts of the training step, because both Runners are alternately stalled.

4.6 Summary

We propose Terra, a novel approach to execute imperative Python DL programs. Terra performs imperative-symbolic co-execution, which addresses the problem of covering Python program features in prior graph generation approaches. Terra carves out DL operations from an imperative DL program and converts the DL operations into an optimized symbolic graph. It then executes the graph and the skeleton imperative program concurrently in a complementary manner to improve performance while maintaining the programmability of the imperative program. Our evaluation shows that Terra can optimize imperative DL programs that are difficult to optimize with the state-of-the-art graph generation approach.

Chapter 5

Graph Generation for Recursive Networks

5.1 Introduction

Recursive neural networks have widely been used by researchers to handle applications with recursively or hierarchically structured data, such as natural language processing [83, 80, 13] and scene parsing [77, 80, 76, 53].

In order to implement such models, *embedded control flow deep learning frameworks* (in short, *embedded control flow frameworks* [1]), such as TensorFlow [8], Theano [94], Caffe2 [27], and MXNet [17], embed control flows within dataflow graphs, i.e., the control flow is represented as a type of operation of the dataflow graph, which can trigger conditional execution or iterative computation. However, the programming model proposed by such frameworks fails to efficiently represent and execute neural networks with recursive structures. The designs of these frameworks do not consider recursive models and instead

[1] This is the same as "imperative deep learning frameworks" in Chapter 3. In this chapter, however, we introduce new name for the same category of frameworks for readability.

urge users to either write their models with iterative constructs [92] or completely unroll models without exploiting control flow at all [91, 65]. Meanwhile, *non-embedded control flow deep learning frameworks* (in short, *non-embedded control flow frameworks* [2]) such as PyTorch [70] or DyNet [68] allow users to define control flows from the client-side, creating new computation graphs for all possible control flow paths of a model. This approach trades performance for programmability, losing optimization opportunities because each graph is usually executed only once.

An important example of recursive neural networks is the TreeLSTM [83] model, a tree-shaped network with recursively definable nodes, demanding complicated execution mechanisms. In existing frameworks, the TreeLSTM network is handled by either statically unrolling the full network graph beforehand [70, 68], or using a single LSTM cell to iteratively compute all intermediate nodes [8, 94]. For the former case, it is difficult to process multiple data instances together because the tree structure differs for each instance. For the latter case, the iterative execution is inherently sequential and thus is incapable of computing multiple nodes in parallel.

In this chapter, we introduce *recursive* definitions into the programming model of existing embedded control flow frameworks [8, 27, 17, 94], adding first-class support for recursion. By allowing users to directly express recursive definitions in application code with enhanced programmability, models with recursive data structures such as trees or graphs can be written without requiring users to use a separate complex API to express the control flow [56]. Also, optimization opportunities can be exploited to boost performance, such as concurrently executing child nodes in tree structures that have no dependencies between each other.

[2] This is the same as "symbolic deep learning frameworks" in Chapter 3.

We make recursive definitions possible by introducing a special graph operation, `InvokeOp`, that abstracts the execution of a `SubGraph`. Users can incorporate recursion in models by invoking a `SubGraph` within the `InvokeOp` that abstracts the same `SubGraph`. The framework handles the execution of an `InvokeOp` as the initiation of a new `SubGraph` containing a bundle of inner operations, which are treated the same as the original running operations.

We implemented support for recursively defined dataflow graphs on TensorFlow [8], a widely used deep learning (DL) framework. To show the expressive power and the performance of recursive graphs, we implemented three applications using our framework: sentiment analysis with the TreeRNN [80], RNTN [81], and TreeLSTM [83] models. For every model, we succeeded in capturing the recursive semantics of the computation graph, and achieved competitive performance compared to other state-of-the-art deep learning frameworks such as TensorFlow [8] and PyTorch [70].

The rest of the chapter is organized as follows. Section 5.2 explains the limitations of existing embedded control flow frameworks regarding recursive models, and Section 5.3 provides a high-level API for efficiently representing such recursive models. Section 5.4 describes the design aspects of our framework, and Section 5.5 presents the implementation details. Section 5.6 presents evaluation results on various applications. Section 5.7 covers related work and Section 5.8 concludes.

5.2 Motivation

5.2.1 Embedded Control Flow Frameworks and Their Limitations

Modern deep learning frameworks use directed acyclic graphs (DAGs) to represent mathematical computations of deep learning applications and the execution

order of such computations. The vertices of graphs represent the mathematical operations, while the edges represent the dependencies between two operations. An edge from operation a to operation b implies that the output of a is fed into b as the input value. As the execution order between any two operations in the computation graph is statically determined, it is a non-trivial task to represent dynamic control flow within computations, such as conditionally executing only a part of the graph, or jumping to a nonadjacent operation.

Based on how to handle dynamic control flow, we can divide deep learning frameworks into two categories: *embedded control flow* frameworks and *non-embedded control flow* frameworks. Embedded control flow frameworks such as TensorFlow [8] and Theano [94] include control flow concepts inside the computation graph. They define special kinds of control flow operations to embed the control flow within the graph. This way, a single computation graph is able to express multiple control flow paths. Since these frameworks can build a single graph and execute it repeatedly, aggressive performance optimization can be done while hiding the optimization overhead.

On the other hand, non-embedded control flow frameworks including PyTorch [70], DyNet [68], and Chainer [95] do not represent the control flow inside the computation graph. Instead, they create a new static computation graph for every activated control flow. This approach enables fast prototyping and easy development of various deep neural networks. However, this approach leaves little room to optimize the performance of computation graph execution, because each graph gets executed only once.

Embedded control flow frameworks. In embedded control flow frameworks, graph vertices represent not only arithmetic operations (e.g., `Add` or `MatMul`) and data transformations (e.g., `Concat`), but also data-dependent control flow mechanisms. Conditional expressions are often made available by many

embedded control flow frameworks. A predicate is expected as the first input argument, and two other operation groups as the `true` and `false` inputs. Based on the predicate value, only one of the two operation groups are executed and passed to the output operation. Another useful control flow construct in existing deep learning frameworks is the iterative loop construct, namely the `while_loop` operation in TensorFlow and the `Scan` operator in Theano. This kind of API enables adding a group of operations, referred to as a loop body, to be executed multiple times iteratively. Conditional expressions are usually used with loop constructs to denote the termination condition of the loop body.

By planting dynamic control flow inside the computation graph and thus decoupling the client-side code execution from computation graph execution, frameworks can exploit parallelism while executing jobs by handling mutually independent operations in a concurrent manner, and can also exploit graph optimization techniques for faster execution that would otherwise be impossible for non-embedded control flow frameworks. This chapter will focus on embedded control flow frameworks, building up on the provided optimizations to produce maximum performance.

Limitations of embedded control flow frameworks. The computation graphs of embedded control flow frameworks do not fully cover every possible control flow construct, however. Designing recursive neural networks efficiently using embedded control flow of iterative loop constructs is difficult. Not only is it unclear how to parallelize independent operations with iterative loops, recursion and iteration are fundamentally different and thus converting one into another involves a nontrivial conversion process [54, 29, 28]. The following subsection shows an example demonstrating the difficulties of designing recursive neural networks with just loop constructs.

5.2.2 Example: TreeLSTM

The long short-term memory [35] (LSTM) cell is a block of functions that is well-known for its ability to "remember" past computations of a neural network, and is often used for networks that process data of sequential characteristics such as text data with sentence structures.

TreeLSTM [83] is a widely used recursive neural network based on LSTMs that is known to perform well for applications with tree-shaped data instances such as parse trees for natural language processing and hierarchical scene parsing [80]. In an ordinary linear recursive neural network, LSTM cells are placed sequentially regardless of the input data structure. On the other hand, in the TreeLSTM model, LSTM cells are combined to form a tree, mimicking the shape of the input data tree. Sentiment analysis is often used as an application of the TreeLSTM. For example, with movie review sentences and the corresponding ratings as training input data and labels, the TreeLSTM network can be trained to predict the sentiment of each movie review sentence.

There are two approaches to implement this TreeLSTM network with current deep learning frameworks, both having its own limitations.

The first approach is *unrolling* the whole tree structure to the computation graph, so that LSTM cells are duplicated for each tree node. To train multiple trees with this approach, however, a new graph must be created for all input training instances. Not only does this result in an excessive amount of graph objects and significant construction overhead, the effect of compile-time graph optimization is near zero as all graphs are used only once.

The second approach is using *iterative* control flow operations provided by frameworks. Figure 5.1 shows pseudocode of an iterative implementation of the TreeLSTM model. In this implementation, a single LSTM cell can be

used multiple times for multiple input data instances. After the leaf nodes are processed sequentially in Line 14, the internal nodes with their dependencies resolved get processed in Line 15. In order for this approach to work, the input tree must be preprocessed so that its nodes are assigned with topologically sorted indices, i.e., executing the tree nodes in an iterative manner does not violate the computational dependencies. Since the recursive nature of the tree structure cannot be directly represented by iteration, it is difficult to write and understand this code.

The process of topologically sorting the tree nodes loses the parent-child node relationships of the tree, and thus the iterative implementation can only view the tree nodes as a linearly ordered list. A recursive formulation, on the other hand, would be able to utilize the information on parent-child relationships to concurrently execute nodes, and is inherently more suitable for representing recursive neural networks, preserving their recursive nature.

5.2.3 Recursion in Embedded Control Flow Frameworks

The drawbacks of the unrolling method and the iterative method suggest the need for a more effective and intuitive solution to implement TreeLSTMs, and recursive neural networks in general. We propose that recursively defining and executing recursive neural networks is a simple yet powerful approach.

Recursive execution of computation graphs has many similarities with recursive invocation of functions in general programming languages. Recursive function invocation in programming languages is supported by allowing a function to call itself inside the function body. This is usually more complicated than executing non-recursive functions, since when parsing the source code of a recursive function, the recursive function call must be processed before the parsing of the function gets finished.

```
1  states = array()
2
3  def compute_leaf(idx):
4    curr_state = lstm(embed(tree.leaves[idx]))
5    states.insert(idx, curr_state)
6
7  def compute_internal(idx):
8    left_idx, right_idx = tree.children[idx]
9    left_state = states.get(left_idx)
10   right_state = states.get(right_idx)
11   curr_state = lstm(left_state, right_state)
12   states.insert(idx, curr_state)
13
14 for_loop(range(num_leaves), compute_leaf)
15 for_loop(range(num_internals), compute_internal)
16
17 root_state = states[root_idx]
```

Figure 5.1: Iterative implementation of the TreeLSTM model in pseudocode.

Inspired by the concept of functions and function invocations, we propose to design similar ideas in embedded control flow frameworks to support recursive execution. First, a programming interface for defining a subset of the computation graph that will be executed recursively is required. Then, an invocation operation inside the graph subset is also needed, to trigger the recursive execution of the graph subset. No modern embedded control flow framework supports these functionalities and, at the same time, is able to train a recursive neural network, to the best of our knowledge.

Our observations above suggest that an implementation of recursion, for embedded control flow frameworks, must satisfy two conditions. First, recursion must be expressible as part of a valid computation graph. Despite the fact that recursion implies the usage of a call stack of arbitrary length, the graph representation of recursion must be finite and executable by the framework.

The graph representation of recursion corresponds to the recursive *function* definition; the definition simply denotes what computation is involved and how the recursion occurs, while not actually running the function. Moreover, this representation must be usable together with other non-recursive parts of the computation graph (Section 5.3.1).

Second, an operation included in a recursive computation graph must be able to trigger the surrounding computation graph recursively. The operation triggering the recursive graph execution corresponds to the function *invocation*, which can further unfold the computation until the recursion termination condition is satisfied (Section 5.3.2).

5.3 Programming Model

In this section, we describe our modifications to the programming model of existing embedded control flow frameworks, as well as how they are translated into dataflow graph components.

5.3.1 Unit of Recursion: `SubGraph`

It is infeasible to implement the dynamism and recurrences of recursive computations using the static components of dataflow graphs provided by existing embedded control flow frameworks. In response to this shortcoming, we first propose an abstraction, `SubGraph`, that represents basic recursive blocks and, at the same time, can be used in conjunction with existing operations to create a dataflow graph with recursive computations.

`SubGraph`s are created by grouping operations of a given computation graph that will be executed recursively. `SubGraph`s represent fractions of a dataflow graph. Executing a `SubGraph` refers to executing the operations that belong to that `SubGraph`. The inputs and outputs of operations that are connected to

outer operations (operations that reside outside of the current `SubGraph`) are assigned as inputs and outputs of the `SubGraph` itself. During execution, the inputs of a `SubGraph` are passed to the corresponding inner operations, while operation outputs that must be shipped out to outer operations are passed as `SubGraph` outputs. A `SubGraph` can be regarded as a function in general programming languages.

Additionally, we allow `SubGraph`s to invoke other `SubGraph`s. A `SubGraph` invocation within an outer `SubGraph` is connected to the other inner operations to form an inner dataflow graph, just as the outer `SubGraph` is connected to outer operations. A `SubGraph` invocation in a `SubGraph` simply implies that there is yet another group of operations to be executed at that particular graph position. Coming back to the function analogy, placing a `SubGraph` invocation within a `SubGraph` is identical to calling a function within another function.

More importantly, a `SubGraph` may recursively invoke itself. This aspect makes possible the definition of a recursive computation; we define a recursive block as a `SubGraph` and insert a invocation to itself in the same `SubGraph`.

Figure 5.2 shows the recursive implementation of the TreeLSTM model, with details omitted for brevity. After defining a `SubGraph` for the TreeLSTM model in Line 2, we reuse the definition in Lines 10-11 to complete the recursive tree structure of the model. Notice how a conditional expression is used (`if` in Line 14) to separate the base case from the recursive case. Comparing with Figure 5.1, this recursive version follows the definition of the TreeLSTM model more clearly; the recursive nature of the tree structure is explicitly represented in this implementation.

```
1  # TreeLSTM: index(int32) -> hidden_state(Tensor)
2  with SubGraph() as TreeLSTM:
3    idx = TreeLSTM.input(int32)
4
5    def compute_leaf_node():
6      return LSTM(embed(tree.leaves[idx]))
7
8    def compute_internal_node():
9      left_idx, right_idx = tree.children[idx]
10     left_state = TreeLSTM(left_idx)
11     right_state = TreeLSTM(right_idx)
12     return LSTM(left_state, right_state)
13
14   TreeLSTM.output(if(is_leaf_node(idx),
15                      compute_leaf_node,
16                      compute_internal_node))
17
18 root_state = TreeLSTM(root_idx)
```

Figure 5.2: Recursive implementation of the TreeLSTM model with `SubGraph` definitions. After declaring the start of a `SubGraph` in Line 2, we indicate the inputs of the `SubGraph` in Line 3. The body of the `SubGraph` is defined in Lines 5-16, while recursive calls are made on Lines 10-11. Note that `SubGraph` outputs must be given as in Lines 14-16. The completed `SubGraph` definition can now be used in Line 18.

5.3.2 Recursion in Dataflow Graphs: `InvokeOp`

While `SubGraph`s provide a convenient way to define recursive computations, the framework is still left with the task of actually executing the operations gathered as `SubGraph`s. However, as `SubGraph` operations are expected to be executed in a recursive fashion, an additional mechanism for "re-executing" the operations of `SubGraph`s repeatedly (until some termination condition is met) is required. To this end, we introduce a new operation named `InvokeOp`.

An `InvokeOp` is an operation that takes a set of tensors as input, runs an

associated SubGraph (i.e., executes the inner operations of the SubGraph) with the provided inputs, and returns the resulting tensors as output. InvokeOps are execution objects instantiated from SubGraph invocations; as SubGraphs are semantically close to function definitions, InvokeOps can be considered as function calls to the functions specified by SubGraphs. As such, it is possible for a single SubGraph to be associated with more than one InvokeOp.

Despite the special property of having an associated SubGraph, an InvokeOp is fundamentally the same as other operations such as Add or MatMul, and is generally treated as an ordinary operation. The difference with other operations comes from the operation kernel implementation; instead of performing a mathematical calculation, an InvokeOp abstracts the execution of an entire SubGraph. This difference also affects a process called automatic differentiation, a core functionality provided by modern deep learning frameworks for training models. Instead of calculating mathematical derivates of some numerical function like other operations, the framework must take into account the associated SubGraph of an InvokeOp. We will discuss this further in Section 5.4.2.

5.3.3 TreeLSTM with SubGraphs & InvokeOps

Figure 5.3 portrays an example on how InvokeOps are used to represent the TreeLSTM (Section 5.2.2) model with recursion. A completely unrolled depiction of the model for a full binary tree is shown in Figures 5.3(a) and 5.3(b). It is not hard to observe that the model can be expressed using recursion: the embed operation and the LSTM cell at the leaves form the base case (Figure 5.3(a)), while the two-input LSTM cell at the intermediate notes corresponds to the recursive case (Figure 5.3(b)).

Merging the base case and the recursive case into a SubGraph with a conditional branch (if), we now have a concise representation of the TreeLSTM

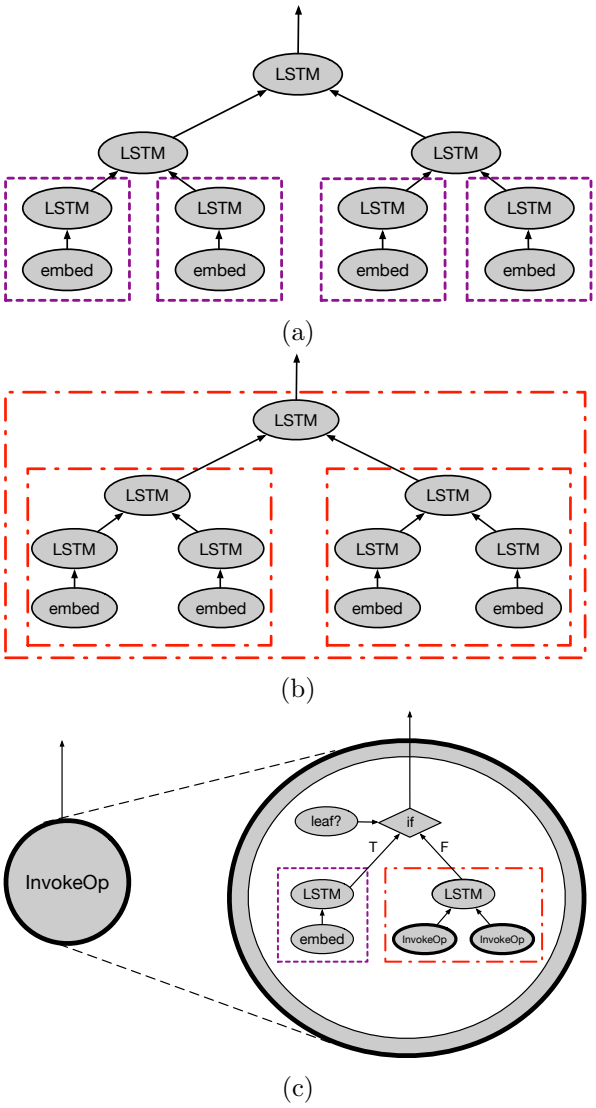


Figure 5.3: An illustration of how an unrolled computation graph of the TreeLSTM model (a, b) can be transformed into a recursive graph with `InvokeOp`s (c). The base case, depicted in the boxes of (a), and the recursive case, indicated by the boxes in (b), can be combined to succinctly describe the model as a recursive `SubGraph` as shown in (c). `InvokeOp`s have been added at the appropriate places to mark the points where a recursive function call to the `SubGraph` must occur.

model, as shown in Figure 5.3(c). Note that the condensed `SubGraph` is able to represent TreeLSTMs of arbitrary height or shape, and not just a single particular structure. `InvokeOp`s are inserted at all inner recursive call points to complete the computation graph.

5.4 System Design

In this section, we discuss various system design aspects for supporting the recursive programming model of the previous section.

Our design complements existing embedded control flow frameworks with additional APIs for declaring recursive graphs and core changes for executing such recursive graphs. Models declared using the `SubGraph` API from Section 5.3 are transformed into a dataflow graph containing `InvokeOp`s. In turn, the framework core engine runs the resulting graph with the same mechanism used to run non-recursive graphs, accessing additional graph and value cache structures when dealing with `InvokeOp`s. The design does not involve any implementation details of a particular framework, and can be applied to any DL framework that incorporates control flows in computation graphs.

5.4.1 Graph Execution

Background: Execution Model of Existing Frameworks

The execution model of embedded control flow frameworks can be characterized by three components: the *client* who builds and submits dataflow graphs to the framework, the *master* which parses the given graphs and schedules the execution of operations, and one or more *workers* that carry out the actual computation of the operations. The master coordinates the execution of operations on the workers, running operations in an order that respects the inter-operation dependencies.

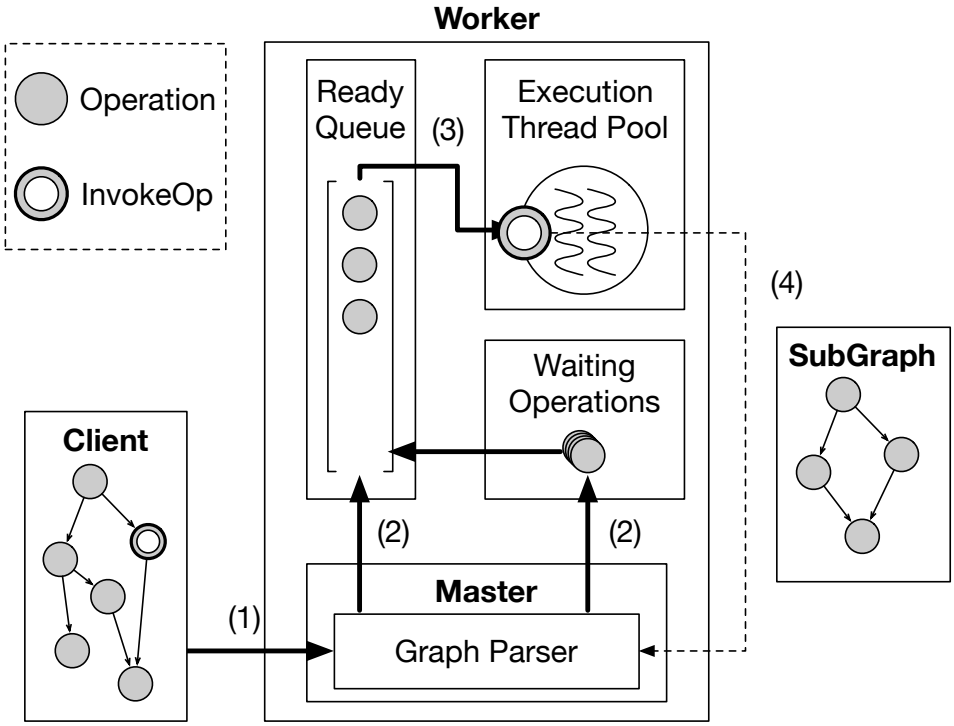


Figure 5.4: The execution model of embedded control flow frameworks with `InvokeOp`s. (1) After the client initiates the job with a dataflow graph, (2) the master decomposes the graph into operations and places them into either the ready queue or the waiting line of the worker, depending on the number of unresolved inputs. (3) Operations are dequeued from the queue by idle execution threads, while new operations are enqueued when input dependencies are resolved. (4) When an `InvokeOp` gets executed, its associated `SubGraph` is passed to and processed by the master, similar to step (1). Only one worker is shown for the sake of clarity.

Steps (1)-(3) of Figure 5.4 displays an illustration of the execution model, with only one worker shown for simplicity. When the master first analyzes the input dataflow graph, operations that require no inputs are enqueued directly into the *ready queue* of the worker, whereas operations that need at least one input are put on standby. Next, execution threads of the worker's *execution thread pool* grab operations from the operation queue and perform the computation for those operations in parallel. When an execution thread finishes running an operation, the master checks the waiting operations that have a dependency on the completed operation, and enqueues operations whose dependencies have all been resolved to the ready queue. This process is repeated until all operations have been processed.

Recursive Execution

The execution mechanism for executing a recursively defined dataflow graph is no different from the mechanism for executing non-recursive graphs. This is possible because the execution of an `InvokeOp` mimics the initiation of a new dataflow graph, with the exception of reusing the same master scheduler as well as the same worker ready queues, as illustrated in step (4) of Figure 5.4. When an `InvokeOp` becomes executable and is dequeued by an execution thread, the graph associated with the `InvokeOp` is processed by the master, similar to how a graph submitted by the client is parsed by the master. Operations that are immediately ready to be run are enqueued into the ready queue, behind the existing operations. Likewise, operations that have at least one unresolved input dependency are added to the waiting list, together with other previous standby operations.

This design allows recursive dataflow graphs to be processed on existing embedded control flow frameworks without drastic changes. Recursive graphs

can enjoy graph optimizations supplied by such frameworks and achieve good performance while providing intuitive, recursive APIs at the same time. In fact, from the framework's point of view, a recursive graph is the more general representation, while non-recursive graphs are simply special cases which have no recursive `SubGraph`s and `InvokeOp`s.

It is also worth noting that performing priority scheduling of operations instead of simple FIFO scheduling may possibly yield significant effects on the execution time of recursive computation graphs, depending on the inter-operation dependencies of the given recursive model. For example, if the model contains a `SubGraph` whose inner operation must be processed in order for many outer operations to be enqueued into the ready queue, then a scheduling decision of processing inner operations before others would lead to a shorter execution time overall. Although this is an interesting problem, it is usually not an issue for servers with many parallel computation threads to spare and thus we leave this as future work.

Graph execution stack. When a function is invoked in programming languages, the language runtime maintains a call stack to record the call history and relevant information. This enables the program to correctly return from the callee function to the corresponding caller function, and also provides helpful information to programmers such as backtrace information when an exception occurs while executing the function. A similar process of keeping track of the `SubGraph` invocation history is required for the recursive graph execution engine. However, the caller-callee relationship of `InvokeOp`s cannot be managed with a linear stack, because an `InvokeOp` can branch out into multiple child `InvokeOp`s in parallel. Rather, the relationship is maintained as a tree, where each `InvokeOp` holds a pointer to its parent `InvokeOp` (i.e., return location).

5.4.2 Graph Backpropagation

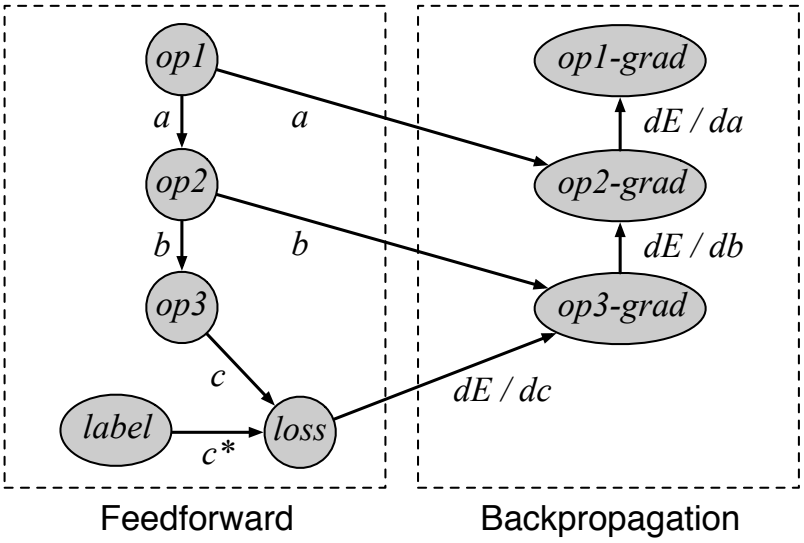
Background: Automatic Differentiation

Neural networks are normally trained via the backpropagation algorithm [72]. Backpropagation calculates the errors of all operation output values, by first comparing the final outputs of a neural network with expected ground-truth values[3] (labels) to compute output errors, and then propagating the output errors all the way back up to the input according to the chain rule. The calculated errors – often referred to as *gradients* – are used to update model parameters so that operation outputs are pushed towards the expected values.

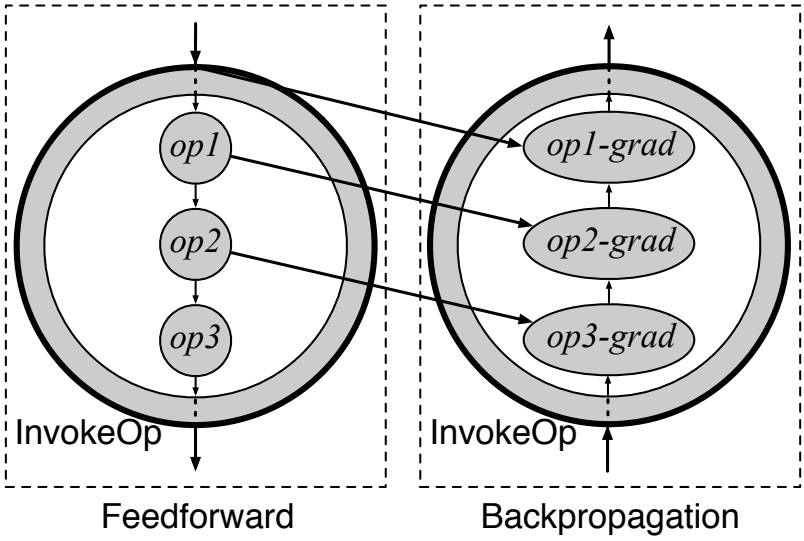
Backpropagation of a simple linear network is shown in Figure 5.5(a). Starting from operation *op1*, all forward operations *op1*, *op2*, and *op3* are computed in succession to produce values a, b, and c, respectively. The final output c is checked with the expected value $c*$ to produce the loss value E, as well as the gradient of E with respect to c, denoted as dE/dc. Next, the other gradients are generated one by one, this time through the backpropagation line of operations, *op3-grad*, *op2-grad*, and *op1-grad*.

Note that in order to calculate a certain gradient, both the previous gradient and the corresponding forward value are required. For instance, the gradient dE/db is computed with the previous gradient dE/dc and the forward value b (*op3-grad*). Likewise, dE/db and a are used to compute dE/da (*op2-grad*). This results in a final dataflow graph where a forward operation shares its inputs with its backpropagation equivalent (e.g. *op2* and *op2-grad* both take a as input). As a precaution to prevent values from being invalidated (released from memory) before being consumed by all dependent operations, DL frameworks always keep

[3]Even for unsupervised learning and reinforcement learning tasks in which there is no explicit ground-truth value, the backpropagation algorithm is still applicable if a well-defined loss function exists.



(a) A simple linear feedforward network is shown on the left, while the backpropagation side of the same network is shown on the right. All gradient operations receive previous gradient values from its gradient predecessor as well as the original feedforward value from the feedforward network.



(b) An `InvokeOp` and its gradient operation for backpropagation are shown on the left and right, respectively.

Figure 5.5: Backpropagation of dataflow graphs with and without `InvokeOp`s. Notice how (a) and (b) are structurally very similar, except for the enclosing `InvokeOp`s.

all operation outputs as valid data until that particular iteration terminates.[4]

Automatic differentiation. Deep learning frameworks relieve users from the burden of manually inserting backpropagation operations, with the help of a process called automatic differentiation. In the case of embedded control flow frameworks, after a user submits a feedforward neural network to the framework, the framework automatically adds all operations required for computing gradients to the given dataflow graph. Maintaining a catalogue of predefined gradient operations, the framework backtracks along the feedforward path and adds the corresponding gradient for each and every feedforward operation. The resulting computation graph can then be processed by the framework for execution. As setting up the backpropagation path is usually much more tedious than defining the forward path, the automatic differentiation process is very helpful for users and currently supported by all deep learning frameworks.

Recursive Backpropagation

Backpropagation of a recursive dataflow graph is similar to backpropagation of a non-recursive dataflow graph. The only nontrivial issue is how to define and calculate gradients for `InvokeOp`s. As the feedforward output of an `InvokeOp` is the execution output of its associated `SubGraph`, naturally the gradient of an `InvokeOp` is also generated from the gradients of the associated `SubGraph`.

During automatic differentiation, we inspect the `SubGraph`s associated with `InvokeOp`s and perform automatic differentiation on them as well. For each `SubGraph`, we collect the gradient operations that were inserted by automatic differentiation. At this point, it is possible to simply add the inserted gradient operations to the backpropagation path of the computation graph. However,

[4]Technically, we could recompute the forward operation values during backpropagation instead of retaining them to save memory. However, this incurs a significant increase in training time and is generally not preferred.

in case the `SubGraph` was used for recursion, the gradients for the inner recursive computations would not be generated and thus backpropagation would be returning incorrect results.

Instead, we wrap each set of gradient operations from `SubGraph`s with yet another `SubGraph`. If a feedforward `SubGraph` contains a recursive invocation to itself, then its corresponding backpropagation `SubGraph` will also hold a recursive invocation, at the same position. Later, `InvokeOp`s are inserted at `SubGraph` invocation points for both the feedforward `SubGraph` and the backpropagation `SubGraph` to complete the computation graph.

Figure 5.5(b) illustrates how a gradient operation of an `InvokeOp` is formed. The associated `SubGraph` is shown in the inner side of the feedforward `InvokeOp`, while the corresponding gradient operations of the `SubGraph` are shown inside the backpropagation `InvokeOp`. Carrying over operation outputs from the feedforward phase to the backpropagation phase is done by connecting the outputs and inputs of the relevant operations, same as in Section 5.4.2.

5.5 Implementation

We implemented our framework atop TensorFlow [8] 1.3.0. Framework changes, including the kernel implementation of `InvokeOp` as well as internal data structures, were done in the C++ core, while client-side APIs for recursive definitions are exposed in Python. Here, we describe several implementation issues of our framework.

Forward declaration. In embedded control flow frameworks, all operations must have well-defined inputs and outputs before they are used (comparable to function signatures in programming languages). `InvokeOp`s are not exceptions; the framework does not allow the creation of a recursive `InvokeOp` unless the operation definition for the recursive call is specified beforehand.

This rule can be bypassed by using forward declarations for `InvokeOp`s that are recursively defined; when a `SubGraph` is defined, we first predeclare an empty `InvokeOp` that has the same signature as the `SubGraph`, and later "register" the `SubGraph` definition to the empty `InvokeOp`. Note that this procedure is automatically done by the framework, and is not exposed to users. Gradients for backpropagation are defined in a similar manner, with the operation declaration coming before the actual definition.

Backpropagation cache implementation. As described in Section 5.4.2, operation output values from the feedforward phase must be retained until backpropagation and be fed into the corresponding gradient operations. For non-recursive computation graphs, embedded control flow frameworks would accomplish this simply by holding a feedforward value entry for each required operation and later passing the values to the appropriate gradient operations. Unfortunately, for recursive graphs, an operation within a `SubGraph` may be called more than one time across multiple recursions. Multiple output values generated during multiple executions must all be passed to the corresponding gradient operation, without losing their topological position information.

We resolve this issue by maintaining a concurrent hash table for storing and fetching operation output values of `SubGraph`s. Figure 5.6 describes the whole procedure of passing multiple feedforward output values from `InvokeOp`s. A hash table is externally generated and managed per `SubGraph`, and a unique hash entry key is used to distinguish table entries. During the feedforward phase, we store all output values of `InvokeOp` instances that come from the `SubGraph` in the table. An `InvokeOp`'s key is defined by combining the `InvokeOp`'s topological position within the `SubGraph` with the key of the parent `InvokeOp`, guaranteeing uniqueness. By using a concurrent hash table, multiple instances of the same operation in the graph can concurrently access and update the hash

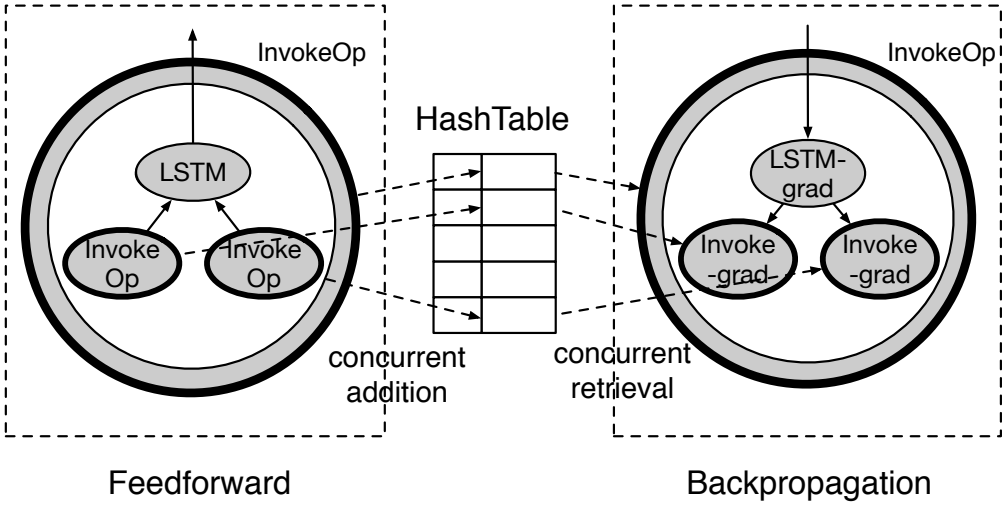


Figure 5.6: Concurrent hash table being used between multiple forward and backward executions of the same operation (`InvokeOp`).

table.

Next, during backpropagation, we perform a hash table lookup for each gradient operation of the `InvokeOp` instances and feed the stored value as input. This enables feedforward output values to be retained and correctly retrieved for backpropagation. While the concurrent insert operations may incur minor overhead, the lookup operations are thread-safe and are negligible. Is it notable that using a simple queue or stack to store activation values is inadequate, as the order of enqueue and dequeue operations or push and pop operations is not deterministic and thus output values may be directed to the wrong gradient operation.

Outer reference. It is common for a recursive `SubGraph` to not refer to the external input values explicitly, but rather implicitly. For instance, a static value that is required for all levels of recursion of a `SubGraph` should not be included as an input of the `SubGraph`, as the value does not change anyway. Nonetheless, the TensorFlow framework regards a `SubGraph` and the outer graph as two

separate graphs and is unable to understand the identities of such implicit external values unless they are specified as actual operation inputs. Therefore, when a `SubGraph` is created, we analyze the operations to check whether there are any external inputs that have not already been specified as the `SubGraph`'s inputs and add them to the input list.

Implementation on other frameworks. Recursively defined `SubGraph`s and `InvokeOp`s can be implemented on not only TensorFlow but any other embedded control flow DL frameworks as well, with the computation graph and the operations as its elements. A `SubGraph` can be provided as an abstraction that is similar to the framework's original graph structure but contains only a subset of all operations to mark a recursion block. An `InvokeOp` can be implemented as a new kind of user-defined operation type, which recursively executes a `SubGraph`.

For instance, Caffe2 [27] uses a `NetDef` protocol buffer as its computation graph, and allows feeding `NetDef`s as inputs to operators. By extending `NetDef`s to recursively represent subgroups of operators, we can create a Caffe2 version of `InvokeOp` that receives such subgroups as inputs and executes them. Theano [94] provides the `Scan` operator which abstracts the execution of a loop in a single operator. Although the `Scan` operator is usually used to express iterative control flow, the concept of maintaining a separate graph isolated from the main computation graph fits well with `SubGraph`s and is a good starting point for implementing recursive computations.

5.6 Evaluation

We evaluate our framework while focusing on the performance benefits of the recursive nature of the framework, mostly made possible by exploitation of parallelism in recursive neural networks.

5.6.1 Experimental Setup

Applications. We implemented a variety of neural network models from the recursive neural network family, namely the aforementioned TreeLSTM [83] model as well as the TreeRNN [80] and the RNTN [81] model. All models were trained and tested to perform sentiment analysis on the Large Movie Review [57] dataset, where data instances are sentences in the form of binary parse trees. For this dataset, we used a pre-trained network (for each model) to label all nodes. For all models, we used the same hyperparameters as the ones suggested in the original papers. We also considered smaller batch sizes to investigate the performance trends of our framework without mixing in additional performance gains obtainable from batching instances.

Frameworks. Along with our implementation of *recursive* dataflow graphs (built on TensorFlow 1.3.0), we also implemented neural networks on other frameworks, including TensorFlow [8] (TF 1.3.0) without recursive graphs, which allows an *iterative* way of programming, and PyTorch [70] (PTH 0.3.1), which only supports the static *unrolling* technique. Since native TensorFlow does not support recursive definitions, we used TensorFlow's control flow operators to train the neural networks in an iterative fashion, as shown in Section 5.2. For PyTorch, we dynamically create a new graph structure for each sentence. Although implementing the static unrolling technique on TensorFlow is possible, the graph generation overhead tends to be very large; instead, we use PyTorch for the unrolling technique, which incurs negligible graph construction overhead.

Hardware specification. All numbers reported in this section were retrieved from experiment results on a single machine of two 18-core Intel Xeon E5-2695 @ 2.10 GHz processors with 256GB RAM, unless otherwise specified.

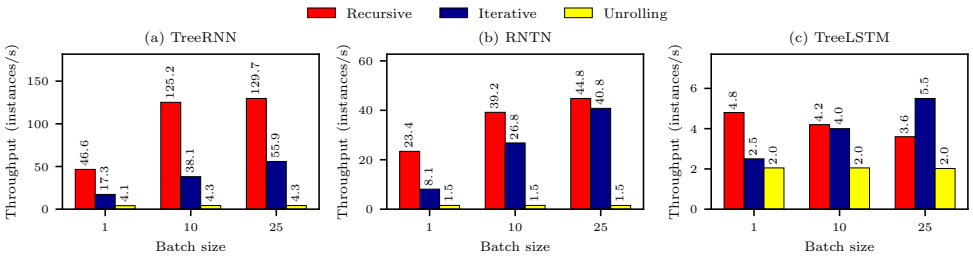


Figure 5.7: Training throughput for the TreeRNN, RNTN, and TreeLSTM models with the Large Movie Review dataset. Numbers are shown for our recursive implementation, TensorFlow's iterative implementation, and PyTorch's static unrolling implementation. Our recursive implementation outperforms the other frameworks for all models and all batch sizes except when training TreeLSTM with a batch size of 25, at which point the amount of system resources is insufficient to completely parallelize the computation. We did not observe any significant performance gain for the static unrolling approach when the batch size was increased.

We also used an NVIDIA Titan X GPU for certain models. Unlike other common neural networks such as convolutional models, the unstructured input data of recursive neural networks makes it difficult to exploit the full computational power of GPUs. Thus, we use GPUs only if they introduce performance gain compared to CPU-only execution. Our implementation and TensorFlow showed greater performance on CPUs, while PyTorch performed better on a GPU.

5.6.2 Throughput and Convergence Time

We start our analysis by measuring the training and inference throughputs with the recursive, iterative, and static unrolling implementations.

Training throughput. Figure 5.7 shows the throughput of training the TreeRNN, RNTN, and TreeLSTM models using the Large Movie Review dataset with recursive, iterative, and static unrolling implementations. The models were trained with batch sizes of 1, 10, and 25.[5]

[5]The original TreeRNN, RNTN, and TreeLSTM papers state that using a batch size of 25

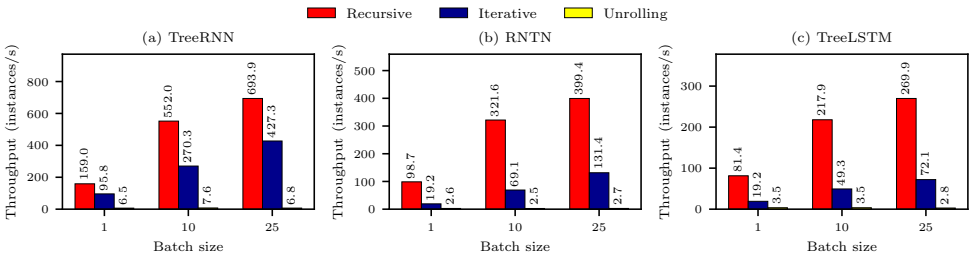


Figure 5.8: Inference throughput for the TreeRNN, RNTN, and TreeLSTM models with the Large Movie Review dataset. Measurements are presented for our recursive implementation, TensorFlow's iterative implementation, and PyTorch's static unrolling implementation. Our recursive implementation outperforms the other frameworks for all models and all batch sizes.

Thanks to the parallelism exploited by recursive dataflow graphs, our implementation outperforms other implementations for the TreeRNN and RNTN models at all batch sizes, by up to 3.3x improved throughput over the iterative implementation, and 30.2x improved throughput over the static unrolling approach. Note that the performance gap between the recursive and iterative approach for the TreeRNN model is bigger than that of the RNTN model. This is due to the fact that the TreeRNN model involves much less computation in its recursive function body compared to the RNTN model, therefore having bigger room for performance optimization via computation parallelization. We will further discuss the effectiveness of parallelization in Section 5.6.3.

For the TreeLSTM model, our implementation performs better than other frameworks at batch sizes 1 and 10. On the other hand, at a batch size of 25, our implementation is slower than the iterative implementation. Generally, recursion has additional overheads compared to iteration, including passing around arguments and return values, caller and callee context setup, etc. We also have additional overheads related to backpropagation, as discussed in pre-

yielded the best model.

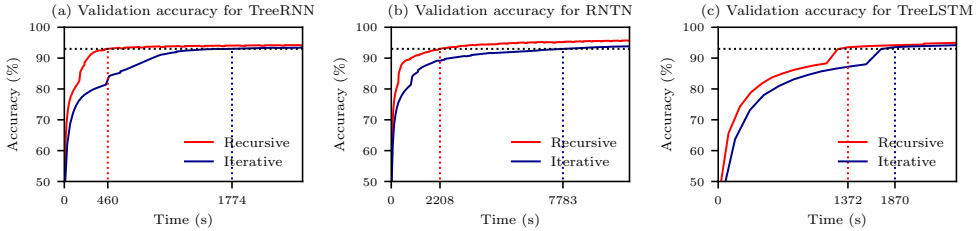


Figure 5.9: Validation accuracy for the binary sentiment classification task with (a) TreeRNN, (b) RNTN, and (c) TreeLSTM models. Results are shown for training each model with the recursive and iterative implementations, using the Large Movie Review dataset. The time to reach 93% accuracy for each setup is also plotted, showing that the recursive implementation converges faster for all models.

vious sections. Consequently, our recursive implementation exhibits excessively high resource utilization for computing large batches, making the throughput lower than the iterative computation.

Inference throughput. *Inference* refers to the process of computing the operation output values of the feedforward phase, stopping before backpropagation. Aside from training throughput, inference throughput is also a useful metric for computing the performance of a neural network, indicating how quickly a deployed model can process unseen data, e.g., in serving systems.

Figure 5.8 shows the inference throughput, with identical environments with the previous experiments on training throughput. Our framework demonstrates throughput up to 5.4x higher than the iterative implementation, and 147.9x higher than the static unrolling approach. Unlike training throughput, our recursive implementation greatly dominates other implementations, since our framework can fully utilize the given resources and the additional overheads introduced by backpropagation are not present.

Convergence. We also measured how the accuracy of the model increases as the training progresses, in Figure 5.9. Since our implementation calculates

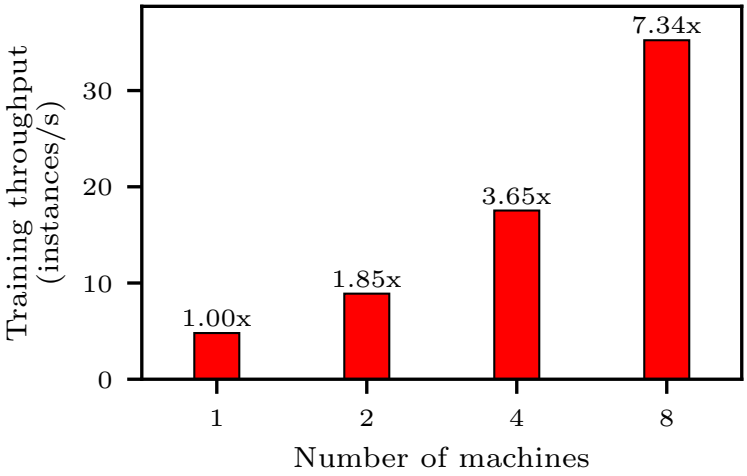


Figure 5.10: Training throughput for the TreeLSTM model on our recursive implementation, using varying numbers of machines for data parallelism. The performance increases almost linearly as more machines are used for training.

numerically identical results as the iterative implementation, the accuracy improvement per epoch is the same. However, thanks to our higher throughput, training with our framework results in faster convergence than the iterative implementation.

Training throughput with multiple machines. One way to overcome the resource limitations is scaling out to multiple machines. Figure 5.10 shows how the training throughput for the TreeLSTM model on our recursive implementation changes, as the number of machines used in training gradually grows from 1 to 8. Utilizing the well-known data parallelism technique [52], the training throughput improves almost linearly up to 8 machines.

5.6.3 Analysis of Recursive Graphs: Parallelization

The performance difference between the iterative and recursive implementation of the same recursive neural network mainly comes from the parallelization of

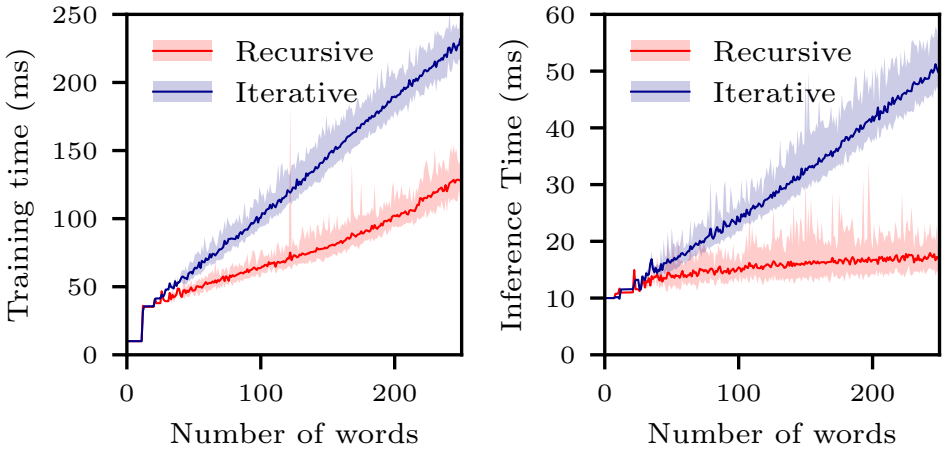


Figure 5.11: Time taken for processing each data instance, in the TreeLSTM model using the Large Movie Review dataset. The bold lines represent the average time for each specific sentence length in the whole dataset, and the enclosing colored areas represent the range of time taken to process the specific length of sentences. No batching is used for this experiment. As the number of words inside a data instance increases, our recursive implementation outperforms the iterative implementation thanks to the parallelized execution of tree cells. For inference, the computation load is low enough for the framework to utilize system resources without hitting the resource limit, and the processing time of the recursive implementation is $O(logN)$, where N is the number of words.

recursive operations. In this subsection, we analyze how the performance varies depending on various aspects related to parallelization.

Sentence length. A close inspection of the training time per data instance sorted by sentence length gives us interesting results. As shown in Figure 5.11, the time required for processing a single sentence generally increases as sentences become longer, regardless of whether the implementation is based on native TensorFlow or our recursive implementation. This is an expected phenomenon, because longer sentences form larger tree structures consisting of more cells which require more computation.

Batch size	Throughput (instances/s)		
	Balanced	Moderate	Linear
1	46.7	27.3	7.6
10	125.2	78.2	22.7
25	129.7	83.1	45.4

Table 5.1: Throughput for the TreeRNN model implemented with recursive dataflow graphs, using datasets of varying tree balancedness. The balanced dataset exhibits highest throughput thanks to the high degree of parallelization, but at the same time does not improve as well as the linear dataset when the batch size increases from 1 to 25, because there is only a small room of performance improvement left, w.r.t parallelization.

However, there is a clear difference in the increasing slope; the training time grows at a steeper slope for TensorFlow than that of our implementation. This is because the recursive implementation allows tree cells to be processed concurrently, whereas the iterative TensorFlow implementation is only capable of processing one tree cell at a time. Theoretically, our implementation is able to process a tree structure consisting of N cells in $O(logN)$ time (native TensorFlow requires $O(N)$ time), though the parallelization effect is diminished by the framework overhead and therefore the performance is more close to a linear trend rather than a logarithmic trend. On inference workloads with much less resource needs, the trend is clearly closer to a logarithmic scale.

Balancedness of trees. To analyze the influence of tree balancedness on training throughput on our recursive implementation, we prepared several modified versions of the Large Movie Review dataset, that contain the same sentences as the original dataset but have different parse tree shapes. Specifically, we prepared 1) a *balanced* dataset consisting of only complete binary trees, 2) a *moderate* dataset that contains moderately balanced binary trees, and 3) a *linear* dataset comprising only extremely unbalanced binary trees.

Table 5.1 shows the throughput of training the TreeRNN model using these three datasets. For all batch sizes, the training throughput on the balanced dataset is the highest, while the throughput on the linear dataset is the lowest. This trend occurs because the maximum possible execution concurrency of a tree is affected by the balancedness of the tree. A full binary tree of N cells can be processed concurrently with at most $\frac{N+1}{2}$ threads, because all $\frac{N+1}{2}$ leaf nodes are mutually independent. On the other hand, an extremely unbalanced binary tree can be processed with only one or two threads at a time due to the linearity of the tree. As a result, our implementation can train input data of balanced trees with greater throughput than input data of unbalanced trees.

Resource Utilization. Another interesting fact in Table 5.1 is that the training throughput on the linear dataset scales better than the throughput on the balanced dataset, as the batch size increases. For the balanced dataset, the recursive implementation efficiently utilizes many threads to process the data even at a small batch size of 1, and thus increasing the batch size leads to a relatively small speed boost. On the contrary, for the linear dataset, the recursive implementation fails to efficiently make use of CPU resources and thus the performance gain provided by increasing the batch size is relatively high.

5.6.4 Comparison with Folding

The performance improvement of our recursive framework discussed in previous subsections comes from executing multiple tree nodes in parallel. On the other hand, another approach for efficiently executing recursive neural networks exists: identifying concurrently executable nodes and batching them into a single node to be run on GPUs. We refer to this technique as *folding*, following the name of a framework, TensorFlow Fold [56], that implements this technique.

The folding technique hardly suffers from resource limitations, as GPUs are

Batch size	Throughput (instances/s)					
	Inference			Training		
	Iter	Recur	Fold	Iter	Recur	Fold
1	19.2	81.4	16.5	2.5	4.8	9.0
10	49.3	217.9	52.2	4.0	4.2	37.5
25	72.1	269.9	61.6	5.5	3.6	54.7

Table 5.2: Throughput for processing the TreeLSTM model on our recursive framework, Fold's folding technique, and TensorFlow's iterative approach, with the Large Movie Review dataset. The recursive approach performs the best on inference with efficient parallel execution of tree nodes, while the folding technique shows better performance on training thanks to its GPU exploitation.

very efficient in batching computations. However, batching multiple nodes leads to overheads that are not present in other approaches. Due to the various tree structures in the input data, the batching decision must be done in a depth-wise manner, thus the ungrouping and regrouping of tree nodes across multiple depths lead to numerous memory reallocations and copies. Moreover, folding is applicable only if the tree structure of the input data is known before executing the computation; for dynamically structured models the folding technique cannot be implemented. Here, we discuss and compare our recursive framework with the folding technique. Experiment results for folding were obtained using the TensorFlow Fold framework.

Statically Structured Models

Table 5.2 compares the throughput of performing inference and training on the TreeLSTM model using our implementation, the iterative approach, and the folding technique. The amount of resources is sufficient for executing forward computations, and therefore our framework outperforms the folding technique for the inference task with up to 4.93x faster throughput. Unlike folding, the

recursive approach does not have any overheads regarding batch regrouping, since the calculated values can be directly passed between caller and callee `SubGraph`s.

However, when the resource usage is high, not every scheduled tree node in the worker ready queue can be executed concurrently, even if the dependencies have been fully resolved. While the scalability of the recursive approach is limited by this drawback for the training task, the folding technique can exploit the GPU and scales better. As a result, the folding technique performs better than the recursive approach for the training task. We can improve the performance of the recursive approach by conditionally deciding whether to batch the operations or not similar to the folding technique, and we leave this as future work.

Dynamically Structured Models

While the models presented in the previous sections demand support for dynamic control flow, there is yet another collection of models that boast an even greater degree of dynamism, in which the model structure is gradually formed depending on intermediate values calculated from prior steps. Top-down TreeLSTM [102] (TD-TreeLSTM) is a dynamic model proposed for sentence completion and dependency parsing. When a trained model receives root node information as an input, the model can generate child nodes based on the information and completes the rest of the tree sentence. The decision of generating a child node or stopping tree expansion is conditionally made based on the computed value of the current node at runtime, so the structure of the complete tree is not known before actually executing the graph. DRNN [10] is a neural network model that can generate tree-structured data from sequences, and therefore the tree structure in unknown before graph execution, similar to TD-TreeLSTM.

Batch size	Throughput (instances/s)		
	Iterative	Recursive	Folding
1	0.30	5.59	Not supported
64	0.34	9.30	

Table 5.3: Throughput for evaluating the TD-TreeLSTM model on our recursive framework and TensorFlow's iterative implementation, on batch sizes of 1 and 64.[6] Being able to execute tree nodes in parallel lets our framework perform better than the iterative approach. Fold's folding technique is inapplicable to the TD-TreeLSTM model.

The Hierarchical Mixtures of Experts [45, 78] model has a similar structure, where the whole tree structure is decided at runtime. The network structure of HMRNN [20] is also dynamically determined by the intermediate computation values.

Our framework performs well for such dynamic models. Table 5.3 shows the throughput of the sentence completing task with the TD-TreeLSTM model. Our implementation performs better than the iterative approach by up to 18.6x, since multiple tree nodes are executed in parallel. For this kind of model, techniques that rely on heavy preprocessing of input data to batch operations (folding) are ineffective because the model structures are unknown until the main computation. We note that it is impossible to express such models using the API provided by the Fold framework.

5.7 Related Work

Embedded control flow frameworks. DL frameworks with a computation graph comprised of control flow operators along with the mathematical operators to represent a DL job are called *embedded control flow* frameworks [8, 94, 27, 17]. This class of frameworks does not use the programming lan-

[6] We follow the suggestions of the original TD-TreeLSTM paper to use a batch size of 64.

guage's control flow support (e.g., Python's `if` clause) for representing dynamic control flow. Instead, they provide certain primitives for embedding dynamic control flow in the dataflow graph; the framework cores evaluate a boolean expression and decide what to apply for the next operation at graph execution time.

Although our implementation is based on the embedded control flow framework TensorFlow [8], the key difference is the ability to express recursive functions. In our implementation, a user can define an arbitrary function and use it as an operation to compose a graph. The arbitrary function can call another function including itself without restriction, allowing recursive definitions of functions. TensorFlow and Theano [94] also let users write user-defined functions, but do not support recursion; the user must not create a cycle of dependencies between functions.

Non-embedded control flow frameworks. Unlike embedded control flow frameworks, PyTorch [70], DyNet [68], and Chainer [95] do not embed control flow operators inside their computation graphs. Rather, the computation occurs along with the dynamic control flow of the host language, removing the need to embed the control flow operators inside the computation graph. In other words, these *non-embedded control flow* frameworks behave just like numerical computation libraries such as NumPy [97] and MKL [38], so one can directly exploit the underlying language's abilities for handling conditional branches, loops, and recursive functions. Thanks to this behavior, a user can easily build a prototype of a new neural network architecture or optimization algorithm.

However, since neural networks are usually trained for numerous steps until they converge, non-embedded control flow frameworks suffer from repetitive construction of graphs composed of hundreds or thousands of nodes, resulting in

substantial object creation and removal overhead. More importantly, embedded control flow frameworks employ graph compilation techniques like operation fusion or in-place operation conversion to optimize execution, but non-embedded control flow frameworks cannot since they do not reuse the graphs.

Recursive dataflow graphs are designed to provide a similar level of programmability to non-embedded control flow frameworks, without losing optimization opportunities by using an embedded control flow framework (TensorFlow) to declare computations with recursion.

Other frameworks with recursion support. TensorFlow Fold [56], a library for handling models with dynamic computation, allows recursion for writing computation graphs. Fold provides a number of new APIs for creating and managing *blocks* (sets of low-level operations). A block behaves as a scheduling unit to enable dynamic batching of different computation graphs. Using these blocks, Fold constructs an execution loop that resembles recursion and starts running the loop from base cases, wiring intermediate results to the appropriate positions for subsequent recursive cases. From the perspective of programmability, Fold provides a whole new set of functional programming style APIs to preprocess input data and build the computation graph. It is required to mix the control flow API of Fold and the computational API of TensorFlow to represent a complete DL job, which is not a trivial task. Also, since the structure and execution order of the computation graph becomes completely different after graph preprocessing, it becomes impossible to pinpoint the location of errors on failures, resulting in poor debuggability.

On the other hand, our framework adds a simple abstraction, `SubGraph`, to the programming model to support recursion. `SubGraph`s can be used with existing operations analogously and does not import any additional execution details other than those already provided by the underlying embedded control

flow framework. Moreover, the final computation graph of `InvokeOp`s retains the original position information of `SubGraph`s, allowing the same debugging experience as the underlying framework.

CIEL [63] is a dynamic task (operator) creation framework that allows users to declare data processing jobs recursively. The operators of CIEL are relatively more coarse-grained compared to DL frameworks, which means the number of recursion calls is not large. The different granularity comes from the characteristics of the target domain; CIEL targets batch processing applications, whereas recursively defined graphs were designed for deep learning. More fundamentally, CIEL cannot be integrated with modern DL frameworks because CIEL does not consider DL-specific mechanisms such as backpropagation or typed operator definitions, which are highly important for DL applications.

5.8 Summary

In this chapter, we have introduced recursive declarations and recursive execution mechanisms for running recursive neural networks on top of existing embedded control flow frameworks. With recursively defined computation graphs, recursive neural networks can be implemented in a fashion that better portrays the recursion aspect, and be executed efficiently by letting the framework exploit parallel execution of computations, both of which were very difficult to achieve on existing frameworks due to the lack of support for recursion. To achieve this goal, we designed and implemented a programming model and a runtime execution model, including automatic differentiation support for deep learning jobs. We have demonstrated the expressive power and performance of recursive graphs by implementing various recursive neural network models using our programming model and comparing them with iterative and unrolling implementations, showing that recursive graphs outperform other approaches

significantly.

Chapter 6

Conculsion and Future Work

6.1 Conclusion

In this thesis, we explored the architectures that unify the imperative and symbolic deep learning execution to improve the performance and programmability of deep learning.

We first introduced 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 assuming that DL programs inherently have the static nature. To preserve the dynamic semantics of Python, Janus generates and executes the graph speculatively, verifying the correctness of such assumptions at runtime. The experiments showed that Janus can execute various deep neural networks efficiently while retaining programmability of imperative programming.

Second, we additionally proposed Terra, a new system architecture for imperative-

symbolic co-execution that overcomes the limitation of Janus. Terra executes the symbolic graph and the skeleton imperative program concurrently in a complementary manner to improve performance while maintaining the programmability of the imperative program. The evaluation showed that Terra can optimize imperative DL programs that are difficult to optimize with the state-of-the-art graph generation approach.

We have additionally introduced recursive declarations and recursive execution mechanisms for running recursive neural networks on top of existing embedded control flow frameworks. With recursively defined computation graphs, recursive neural networks can be implemented in a fashion that better portrays the recursion aspect, and be executed efficiently by letting the framework exploit parallel execution of computations, both of which were very difficult to achieve on existing frameworks due to the lack of support for recursion. To achieve this goal, we designed and implemented a programming model and a runtime execution model, including automatic differentiation support for deep learning jobs. We have demonstrated the expressive power and performance of recursive graphs by implementing various recursive neural network models using our programming model and comparing them with iterative and unrolling implementations, showing that recursive graphs outperform other approaches significantly.

6.2 Future Work

In this thesis, we explored the architectures that unify the imperative and symbolic deep learning execution to improve the performance and programmability of deep learning.

6.2.1 Lightweight Imperative Runtime

In order for Terra to be more effective when optimizing the deep learning execution, we need to minimize the stall time. And to minimize the stall time, we need to minimize the Python execution time. Although we selected TensorFlow as the experiment framework to exploit its powerful graph optimizer, TensorFlow's imperative runtime is much heavier compared to PyTorch's lightweight imperative runtime. Combining the lightweight imperative runtime of PyTorch and the powerful graph optimizer of TensorFlow will improve the performance of Terra in the future.

6.2.2 Alternative Languages for Deep Learning

Python has been the de-facto standard language for describing deep learning models and training them for a while. However, there could be alternative languages other than Python that may enable new opportunities for improving the programmability and performance of deep learning. Such languages should be able to export the model to other runtimes for efficient model inference. And the native type and shape inference of tensors should be supported for model optimization.

Bibliography

[1] PyTorch JIT. https://github.com/pytorch/pytorch/tree/master/torch/csrc/jit.

[2] Autograph Limitation Document. https://github.com/tensorflow/tensorflow/blob/r2.3/tensorflow/python/autograph/g3doc/reference/limitations.md.

[3] Python Language Reference. https://docs.python.org/3.8/reference/.

[4] torch.jit.trace. https://pytorch.org/docs/stable/generated/torch.jit.trace.html.

[5] TorchScript. https://pytorch.org/docs/stable/jit.html.

[6] TorchScript Language Reference. https://pytorch.org/docs/stable/jit_language_reference.html#language-reference.

[7] TorchScript Python Language Reference Coverage. https://pytorch.org/docs/stable/jit_python_reference.html#python-language-reference.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[9] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices.

[10] David Alvarez-Melis and Tommi S. Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. In *ICLR*, 2017.

[11] Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on computers*, 39(3):300–318, March 1990.

[12] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *Siam Review*, 59(1):65–98, February 2017.

[13] Samuel R. Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D. Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. In *ACL*, 2016.

[14] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[15] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *LML Workshop at ECML PKDD*, 2013.

[16] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. Technical report, Google, 2013.

[17] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Workshop on Machine Learning Systems in NIPS*, 2015.

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning.

[19] François Chollet et al. Keras. `https://keras.io`.

[20] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. In *ICLR*, 2017.

[21] David E Culler. Dataflow architectures. *Annual review of computer science*, 1(1):225–253, June 1986.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[23] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Computer Architecture News*, 3(4):126–132, December 1974.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[25] DHZS. tf-dropblock. `https://github.com/DHZS/tf-dropblock`.

[26] Hugging Face. Transformers. `https://github.com/huggingface/transformers`.

[27] Facebook. Caffe2. `https://caffe2.ai`.

[28] Andrzej Filinski. Recursion from Iteration, 1994. *Lisp and Symbolic Computation*, 7, 1, 11-37.

[29] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages*. MIT Press, 2008.

[30] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *SysML*, 2018.

[31] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, 2014.

[32] Daniele Grattarola. Spektral. `https://github.com/danielegrattarola/spektral`.

[33] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[35] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory, 1997. *Neural Computation*, 9, 8, 1735–1780.

[36] Horace He. PyTorch vs TensorFlow. `http://horace.io/pytorch-vs-tensorflow/`.

[37] Chienchin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing Deep Learning Beyong the GPU Memory Limit via Smart Swapping.

[38] Intel Corporation. *Intel Math Kernel Library Reference Manual*. 2009.

[39] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[41] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. In *CVPR*, 2017.

[42] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization.

[43] Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byeong-Gon Chun. Improving the expressiveness of deep learning frameworks with recursion. In *EuroSys*, 2018.

[44] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions.

[45] Michael I. Jordan and Robert A. Jacobs. Hierarchical Mixtures of Experts and the EM Algorithm, 1994. *Neural Computation*, 6, 2, 181–214.

[46] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[47] Kevin-Yang. MusicTransformer-tensorflow2.0. `https://github.com/jason9693/MusicTransformer-tensorflow2.0`.

[48] Jason Kuen. Stochastic Downsampling for Cost-Adjustable Inference and Improved Regularization in Convolutional Networks (SDPoint). `https://github.com/xternalz/SDPoint`.

[49] kyzhouhzau. NLPGNN. `https://github.com/kyzhouhzau/NLPGNN`.

[50] Yann LeCun, Leon Buttou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[51] Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[52] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[53] Liang Lin, Guangrun Wang, Rui Zhang, Ruimao Zhang, Xiaodan Liang, and Wangmeng Zuo. Deep structured scene parsing by learning with image descriptions. In *CVPR*, 2016.

[54] Yanhong A. Liu and Scott D. Stoller. From recursion to iteration: What are the optimizations? In *PEPM*, 2000.

[55] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN Model Inference on CPUs.

[56] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. In *ICLR*, 2017.

[57] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *ACL*, 2011.

[58] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A Hierarchical Model for Device Placement.

[59] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning.

[60] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *CoRR*, abs/1706.04972, 2017.

[61] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[62] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. Autograph: Imperative-style coding with graph-based performance. *CoRR*, abs/1810.08061, 2018.

[63] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[64] MXNet. Deep Learning Programming Style. `https://mxnet.incubator.apache.org/architecture/program_model.html`.

[65] MXNet. RNN Cell API, 2018. `https://mxnet.incubator.apache.org/api/python/rnn.html`.

[66] MXNet Developers. Gluon. `http://gluon.mxnet.io/`.

[67] Stefan C. Müller, Gustavo Alonso, and Adam Amara André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *OSDI*, 2014.

[68] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *CoRR*, abs/1701.03980, 2017.

[69] NVIDIA. NVIDIA TensorRT: Programmable Inference Accelerator. `https://developer.nvidia.com/tensorrt`.

[70] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and

Adam Lerer. Automatic differentiation in pytorch. In *Autodiff Workshop in NIPS*, 2017.

[71] Python Software Foundation. Python programming language. `https://www.python.org/`.

[72] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors, 1986. *Nature*, 323, 6088, 533–536.

[73] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, December 2015.

[74] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[75] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[76] Abhishek Sharma, Oncel Tuzel, and David W Jacobs. Deep hierarchical parsing for semantic segmentation. In *CVPR*, 2015.

[77] Abhishek Sharma, Oncel Tuzel, and Ming-Yu Liu. Recursive context propagation network for semantic scene labeling. In *NIPS*, 2014.

[78] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR*, 2017.

[79] Abhay Singh. gpt-2-tensorflow2.0. `https://github.com/akanyaani/gpt-2-tensorflow2.0`.

[80] Richard Socher, Cliff Chiung-Yu Lin, Andrew Y. Ng, and Christopher D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.

[81] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

[82] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[83] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, 2015.

[84] TensorFlow. Eager Execution. `https://www.tensorflow.org/programmers_guide/eager`.

[85] TensorFlow. GAN with TensorFlow eager execution. `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/eager/python/examples/gan`.

[86] TensorFlow. Swift for TensorFlow. `https://github.com/tensorflow/swift`.

[87] TensorFlow. TensorFlow Model Garden. `https://github.com/tensorflow/models`.

[88] TensorFlow. tf.contrib.eager.defun. `https://www.tensorflow.org/versions/master/api_docs/python/tf/contrib/eager/defun`.

[89] TensorFlow. XLA: Optimizing Compiler for TensorFlow. `https://www.tensorflow.org/xla`.

[90] TensorFlow. XLA Overview. `https://www.tensorflow.org/performance/xla/`.

[91] TensorFlow. Recurrent Neural Networks, 2018. `https://www.tensorflow.org/versions/master/tutorials/recurrent`.

[92] TensorFlow Whitepaper. *Implementation of Control Flow in TensorFlow*. 2017.

[93] TensorSpeech. TensorFlow TTS. `https://github.com/TensorSpeech/TensorFlowTTS`.

[94] Theano Development Team. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[95] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Workshop on Machine Learning Systems in NIPS*, 2015.

[96] Radim Tyleček and Radim Šára. Spatial pattern templates for recognition of objects with regular structure. In *GCPR*, 2013.

[97] Stéfan Van Der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[98] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*, 2014.

[99] Jiahui Yu. Slimmable Networks. `https://github.com/JiahuiYu/slimmable_networks`.

[100] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *EuroSys*, 2018.

[101] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

[102] Xingxing Zhang, Liang Lu, and Mirella Lapata. Top-down tree long short-term memory networks. In *NAACL*, 2016.

[103] Zihao Zhang. yolov3-tf2. `https://github.com/zzh8829/yolov3-tf2`.

[104] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phitchaya Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. Transferable Graph Optimizers for ML Compilers.