JANUS: Fast and Flexible Deep Learning via Symbolic Graph Execution of Imperative Programs

Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Byung-Gon Chun
Deep Learning (DL) Frameworks

Define

Execute

Images From:
- http://www.mdpi.com/
- Short-Term Load Forecasting Using EMD-LSTM Neural Networks with a Xgboost Algorithm for Feature Importance Evaluation, Energies 2017
Two Paradigms

Symbolic DL Frameworks

✓ Build a Symbolic Graph
✓ Execute the Graph

```python
def build_graph(g):
    x = g.input(float)
    linear = g.add(g.mul(W, x), b)

build_graph(graph)
run_graph(graph, x_data)
```

Imperative DL Frameworks

✓ Directly Execute the Computations

```python
def linear(x):
    return W * x + b
linear(x_data)
```
Two Paradigms

Symbolic DL Frameworks
✓ Build a Symbolic Graph
✓ Execute the Graph

```python
def build_graph(g):
    x = g.input(float)
    linear = g.add(g.mul(W, x), b)

build_graph(graph)
run_graph(graph, x_data)
```

Imperative DL Frameworks
✓ Directly Execute the Computations

```python
def linear(x):
    return W * x + b
linear(x_data)
```
Symbolic DL Frameworks

✓ Build a Symbolic Graph
✓ Execute the Graph

```
def build_graph(g):
    x = g.input(float)
    linear = g.add(g.mul(W, x), b)
    build_graph(graph)
    run_graph(graph, x_data)
```

Imperative DL Frameworks

✓ Directly Execute the Computations

```
def linear(x):
    return W * x + b
linear(x_data)
```
Symbolic DL Frameworks

✓ Build a Symbolic Graph
✓ Execute the Graph

```python
def build_graph(g):
    x = g.input(float)
    linear = g.add(g.mul(W, x), b)
    build_graph(graph)
    run_graph(graph, x_data)
```

Imperative DL Frameworks

✓ Directly Execute the Computations

```python
def linear(x):
    return W * x + b
linear(x_data)
```
Two Paradigms

Symbolic DL Frameworks

✓ Build a Symbolic Graph
✓ Execute the Graph

```python
def build_graph(g):
    x = g.input(float)
    linear = g.add(g.mul(W, x), b)
build_graph(graph)
run_graph(graph, x_data)
```

Imperative DL Frameworks

✓ Directly Execute the Computations

```python
def linear(x):
    return W * x + b
linear(x_data)
```
Two Paradigms

Symbolic DL Frameworks
✓ Build a Symbolic Graph
✓ Execute the Graph

```python
def build_graph(g):
    x = g.input(float)
    linear = g.add(g.mul(W, x), b)
    build_graph(graph)
    run_graph(graph, x_data)
```

Imperative DL Frameworks
✓ Directly Execute the Computations

```python
def linear(x):
    return W * x + b
linear(x_data)
```
## Pros & Cons

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symbolic DL Frameworks</strong></td>
<td><strong>Imperative DL Frameworks</strong></td>
</tr>
<tr>
<td>+ Easy to Optimize</td>
<td>+ Direct Execution:</td>
</tr>
<tr>
<td>+ Compiler Optimization</td>
<td>Easy to Program &amp; Debug</td>
</tr>
<tr>
<td>+ Parallel Execution of Operations</td>
<td>- Hard to Optimize</td>
</tr>
<tr>
<td>+ Deploy on GPU, Cluster, Mobile, ...</td>
<td></td>
</tr>
<tr>
<td>- Decoupled View:</td>
<td></td>
</tr>
<tr>
<td>Hard to Program &amp; Debug</td>
<td></td>
</tr>
</tbody>
</table>

### Performance
- **Pros**
  - Easy to Optimize
  - Compiler Optimization
  - Parallel Execution of Operations
  - Deploy on GPU, Cluster, Mobile, ...
- **Cons**
  - Decoupled View: Hard to Program & Debug

### Programmability
- **Pros**
  - Direct Execution: Easy to Program & Debug
- **Cons**
  - Hard to Optimize
<table>
<thead>
<tr>
<th>Pros &amp; Cons</th>
<th>Performance</th>
<th>Programmability</th>
</tr>
</thead>
</table>
| **Symbolic DL Frameworks** | + Easy to Optimize  
  - Compiler Optimization  
  - Parallel Execution of Operations  
  - Deploy on GPU, Cluster, Mobile, ... |  |
| **Imperative DL Frameworks** | + Direct Execution:  
  Easy to Program & Debug |  |
| **Pros** |  |
|  | + Easy to Optimize |
| - Decoupled View:  
  Hard to Program & Debug |  |
| **Cons** | - Hard to Optimize |  |
Symbolic DL Frameworks

**Pros**
- Easy to Optimize
  - Compiler Optimization
  - Parallel Execution of Operations
  - Deploy on GPU, Cluster, Mobile,...

**Cons**
- Decoupled View:
  - Hard to Program & Debug

Imperative DL Frameworks

**Performance**
- Direct Execution:
  - Easy to Program & Debug
- Hard to Optimize
<table>
<thead>
<tr>
<th>Pros &amp; Cons</th>
<th>Performance</th>
<th>Programmability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Symbolic DL Frameworks</strong></td>
<td><strong>Easy to Optimize</strong></td>
<td><strong>Direct Execution:</strong> Easy to Program &amp; Debug</td>
</tr>
<tr>
<td>Pros</td>
<td>+ Compiler Optimization</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+ Parallel Execution of Operations</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+ Deploy on GPU, Cluster, Mobile,...</td>
<td></td>
</tr>
<tr>
<td>Cons</td>
<td>- Decoupled View: Hard to Program &amp; Debug</td>
<td>-</td>
</tr>
<tr>
<td><strong>Imperative DL Frameworks</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Pros & Cons

<table>
<thead>
<tr>
<th>Pros</th>
<th>Symbolic DL Frameworks</th>
<th>Imperative DL Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Easy to Optimize</td>
<td>+ Compiler Optimization, Parallel Execution of Operations, Deploy on GPU, Cluster, Mobile, ...</td>
<td>+ Direct Execution: Easy to Program &amp; Debug</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons</th>
<th>Symbolic DL Frameworks</th>
<th>Imperative DL Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Decoupled View: Hard to Program &amp; Debug</td>
<td>- Hard to Optimize</td>
<td></td>
</tr>
</tbody>
</table>
### Pros & Cons

<table>
<thead>
<tr>
<th>Symbolic DL Frameworks</th>
<th>Imperative DL Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros</strong></td>
<td><strong>Pros</strong></td>
</tr>
<tr>
<td>+ Easy to Optimize</td>
<td>+ Direct Execution:</td>
</tr>
<tr>
<td>+ Compiler Optimization</td>
<td>Easy to Program &amp; Debug</td>
</tr>
<tr>
<td>+ Parallel Execution of Operations</td>
<td></td>
</tr>
<tr>
<td>+ Deploy on GPU, Cluster, Mobile, ...</td>
<td></td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td><strong>Cons</strong></td>
</tr>
<tr>
<td>- Decoupled View:</td>
<td>- Hard to Optimize</td>
</tr>
<tr>
<td>Hard to Program &amp; Debug</td>
<td></td>
</tr>
<tr>
<td>Pros</td>
<td>Cons</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><strong>Symbolic DL Frameworks</strong></td>
<td><strong>Imperative DL Frameworks</strong></td>
</tr>
<tr>
<td>+ Easy to Optimize</td>
<td>- Decoupled View: Hard to Program &amp; Debug</td>
</tr>
<tr>
<td>+ Compiler Optimization</td>
<td></td>
</tr>
<tr>
<td>+ Parallel Execution of Operations</td>
<td></td>
</tr>
<tr>
<td>+ Deploy on GPU, Cluster, Mobile,...</td>
<td></td>
</tr>
<tr>
<td>+ 1 Easy to Optimize</td>
<td>- Hard to Optimize</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imperative DL Frameworks</strong></td>
<td></td>
</tr>
<tr>
<td>+ Direct Execution: Easy to Program &amp; Debug</td>
<td></td>
</tr>
<tr>
<td>+ 1 Hard to Optimize</td>
<td></td>
</tr>
</tbody>
</table>

What People Want Is...

**Pros**
- Programmability
- Performance

**Cons**
- Hard to Program & Debug
JANUS: Combining the Best of Both Worlds

Imperative DL Program

```python
def foo(x):
    prod = mul(3, x)
    return add(prod, 2)
```

Symbolic DL Graph

“Easy Programmability”

“High Performance”
JANUS: Combining the Best of Both Worlds

- 11 models in 5 major neural network categories:
  - Convolutional Neural Networks (CNN) LeNet, ResNet-50, Inception-v3
  - Recurrent Neural Networks (RNN) LSTM, LM
  - Recursive Neural Networks (TreeNN) TreeRNN, TreeLSTM
  - Generative Adversarial Networks (GAN) GAN, PIX2PIX
  - Deep Reinforcement Learning (DRL) A3C, PPO

- Up to 47.6x speedup compared to imperative DL framework, comparable performance (within 4%) to symbolic DL framework with unmodified imperative DL programs
Outline

- Approach
- Challenges
- JANUS
- Evaluation
Challenges in Graph Conversion

Imperative DL Program

def foo(x):
    tmp = mul(3, x)
    return add(tmp, 2)

Transparent Conversion

Symbolic DL Graph

\[
\begin{align*}
x & \rightarrow \\
3 & \rightarrow \\
\text{Mul} & \\
2 & \rightarrow \\
\text{Add} & \\
\end{align*}
\]
Challenges in Graph Conversion

De-facto Standard Language for DL Programming

Imperative Python DL Program

```python
def foo(x):
    tmp = mul(3, x)
    return add(tmp, 2)
```

Transparent Conversion

Symbolic DL Graph
Challenges in Graph Conversion

**Imperative Python DL Program**

```python
def foo(x):
    tmp = mul(3, x)
    return add(tmp, 2)
```

**Transparent Conversion**

**Symbolic DL Graph**
Discrepancy between Python Programs and DL Graphs

“Dynamic”

Imperative Python DL Program

def foo(x):
    tmp = mul(3, x)
    return add(tmp, 2)

“Static”

Symbolic DL Graph

Transparent Conversion
Discrepancy between Python Programs and DL Graphs

“Dynamic”

Imperative Python DL Program

def foo(x):
    tmp = mul(3, x)
    return add(tmp, 2)

Characteristics
- determined at runtime
- change at runtime

“Static”

Symbolic DL Graph

x
Mul
Add
3
2

Transparent Conversion
Discrepancy between Python Programs and DL Graphs

“Dynamic”

Imperative Python DL Program

def foo(x):
    tmp = mul(3, x)
    return add(tmp, 2)

Characteristics
- determined at runtime
- change at runtime

“Static”

Symbolic DL Graph

x

INT, 10x1

INT, 10x1
Mul

INT, 10x1
Add

INT 3

INT 2

Characteristics
- must be given when building a graph
Discrepancy between Python Programs and DL Graphs

**Imperative Python DL Program**

```python
def foo(x):
    tmp = mul(3, x)
    return add(tmp, 2)
```

**Characteristics**
- Determined at runtime
- Change at runtime

**“Dynamic”**

**Characteristics**
- Must be given when building a graph

**“Static”**

**Transparency Confusion**

**SRC:** NO INFO

**DST:** NEED INFO
Example: Recurrent Neural Network (RNN)

class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))

seq[0]: They saw dogs

seq[1]: Was she sick?
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
```python
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
```

**RNN Example**

**Dynamic Control Flow**

**Dynamic Types**

**Impure Function**
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []

        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]

        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
```python
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)
```

```
for sequence in sequences:
    optimize(lambda: model(sequence))
```
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
class RNNModel(object):
    def __call__(self, sequence):
        state = self.state
        outputs = []
        for item in sequence:
            state = rnn_cell(state, item)
            outputs += [state]
        self.state = state
        return compute_loss(outputs)

for sequence in sequences:
    optimize(lambda: model(sequence))
Challenge: achieving **Correctness & Performance** at the same time

- Imperative Python DL Program with Dynamic Features
- Correct & Fast Symbolic DL Graph

Challenge Summary
Challenge: achieving **Correctness & Performance** at the same time

**Challenge Summary**

**Imperative Python DL Program with Dynamic Features**

- **Correct Graph**
  - Slow 😞
- **Fast Graph**
  - Incorrect 😞
Challenge: achieving **Correctness & Performance** at the same time

**Challenge Summary**

**Correct Graph**

**Fast Graph**

**Incorrect**

**Imperative Python DL Program with Dynamic Features**

**Slow 😞**
Outline

- Approach
- Challenges
- JANUS
- Evaluation
Solution: Speculative Graph Generation and Execution

- Goal: Correctness & Performance

- [Performance] Speculatively Specialize the Graph
  - Make reasonable assumptions based on the execution history (Profiling)
  - Run specialized graph (Common Case)

- [Correctness] Validate Assumptions
  - Fallback if an assumption is broken (Rare Case)
for item in sequence:
    state = rnn(state, item)
    outputs += [state]
Imperative DL Program

for item in sequence:
    state = rnn(state, item)
    outputs += [state]

Imperative Executor

Python Interpreter        .
Profiler

len: 3

Fast Path
(Common Case)

Correct Path
(Rare Case)

Modified Python Interpreter
for Transparent Profiling

Overall Workflow on JANUS
Overall Workflow on JANUS

Imperative DL Program

```python
for item in sequence:
    state = rnn(state, item)
outputs += [state]
```

Symbolic DL Graph

Pre-defined DL Operations

Python Interpreter

Profiler

Graph Generator

Standard Compiler Pass
- Reaching Definition Analysis
- Type inference
- Constant Propagation
- ...

Optimize Graph with Profile Information

Fast Path (Common Case)

Correct Path (Rare Case)

len:3

Fast Path

Correct Path

50
Overall Workflow on JANUS

Imperative DL Program:

```
for item in sequence:
    state = rnn(state, item)
    outputs += [state]
```

Symbolic DL Graph:

- Fast Path (Common Case):
  - Graph Generator
  - Validate Assumption for Correctness

- Correct Path (Rare Case):
  - Assert

Python Interpreter

Pre-defined DL Operations

Profiler
Overall Workflow on JANUS

Imperative DL Program

```
for item in sequence:
    state = rnn(state, item)
    outputs += [state]
```

Symbolic DL Graph

Fast Path
(Common Case)

Correct Path
(Rare Case)

Graph Generator

Profiler

Graph Cache

Python Interpreter

Pre-defined DL Operations
Overall Workflow on JANUS

**Imperative DL Program**

\[
\text{for item in sequence:} \\
\text{    state = rnn(state, item)} \\
\text{    outputs += [state]}
\]

**Symbolic DL Graph**

- **Graph Generator**
- **Graph Cache**
- **Symbolic Graph Executor**
  - **Python Interpreter**
  - **Profiler**
  - **Pre-defined DL Operations**
  - **Fast Path** (Common Case)
    - **Cell**
    - **state**
    - **len == 3 ?**
      - **Assert**
  - **Correct Path** (Rare Case)
Overall Workflow on JANUS

**Imperative DL Program**

```python
for item in sequence:
    state = rnn(state, item)
outputs += [state]
```

**Symbolic DL Graph**

- Graph Generator
- Graph Cache
- Symbolic Graph Executor

**Assumption Failure**

```
def len(state):
    return state
```

**Fast Path (Common Case)**
- Pre-defined DL Operations
- Python Interpreter

**Correct Path (Rare Case)**
- Assumption
- Failure
Overall Workflow on JANUS

**Imperative DL Program**
```
for item in sequence:
    state = rnn(state, item)
outputs += [state]
```

**Symbolic DL Graph**

**Graph Generator**

**Profiler**

**Graph Cache**

**Symbolic Graph Executor**

**Pre-defined DL Operations**

**Fast Path**
(Common Case)

**Correct Path**
(Rare Case)
Imperative Executor

for item in sequence:
    state = rnn(state, item)
    outputs += [state]

Graph Generator

Profiler

Graph Cache

Imperative DL Program

len:3

Fast Path
(Common Case)

Correct Path
(Rare Case)

Overall Workflow on JANUS

Python Interpreter

Pre-defined DL Operations
Imperative Executor

Python Interpreter

Pre-defined DL Operations

Graph Cache

Graph Generator

for item in sequence:
    state = rnn(state, item)
    outputs += [state]
Overall Workflow on JANUS

Imperative Executor

for item in sequence:
    state = rnn(state, item)
    outputs += [state]

Symbolic DL Graph

Graph Generator

Profiler

Graph Cache

Fast Path
(Common Case)

Correct Path
(Rare Case)

Python Interpreter

Pre-defined DL Operations

Imperative DL Program

len:?

len:?
Overall Workflow on JANUS

Imperative DL Program

Symbolic DL Graph

for item in sequence:
    state = rnn(state, item)
    outputs += [state]

Profiler

Graph Generator

Graph Cache

Imperative Executor

Symbolic Graph Executor

Python Interpreter

Pre-defined DL Operations

len:3
Additional System Aspects

Imperative DL Program

```
len:3
for item in sequence:
    state = rnn(state, item)
    outputs += [state]
```

Graph Generator

Imperative Execution for **Full Python Coverage**

Profiler

Imperative Executor

Python Interpreter

Pre-defined DL Operations

Python Coverage

Global State Consistency

See our paper for more details!
"Impure" 
Imperative DL Program

```python
def foo(obj):
    obj.data = value
    do_sth if pred else pass
```

"Impure" 
Symbolic DL Graph

- `0xb84c`
- "data"
- `value`
- `SetAttr`
- `pred?`
- `do_sth`
- `Assert`

Python Heap

Symbolic Graph Executor

Pre-defined DL Operations
Additional System Aspects

Impure
Imperative DL Program

```python
def foo(obj):
    obj.data = value
    do_s.th if pred else pass
```

Impure
Symbolic DL Graph

- `0xb84c` "data"
- `pred?`
- `do_s.th`
- `Assert`
- `value`
- `SetAttr`

Unsafe to fallback after heap update

Assumption Failure

Modified Python Heap

Symbolic Graph Executor

Pre-defined DL Operations

Python Interpreter

Python Coverage

Global State Consistency
Impure
Imperative DL Program

```
def foo(obj):
    obj.data = value
do_sth if pred else pass
```

Impure
Symbolic DL Graph

- `0xb84c`` “data”
- `pred?`
- `do_sth`
- `Assert`
- `SetAttr`
- `value`

Symbolic Graph Executor

Write-back after validating assumptions

Python Heap

Local Copy

Global State Consistency

See our paper for more details!
Outline

- Approach
- Challenges
- JANUS

- Evaluation
  - ✓ Correctness and Performance
  - ✓ Breakdown of Performance Improvement
Evaluation Setup: Frameworks & Environments

- **Frameworks**
  - **JANUS**: Implemented on top of TensorFlow and CPython
  - **Symbolic**: TensorFlow
  - **Imperative**: TensorFlow Eager

- **Hardware & Software Setup**
  - 6 machines connected via Mellanox ConnectX-4 cards w/ 100Gbps InfiniBand
  - Each machine w/ 2x(Intel Xeon E5-2695)+6x(NVIDIA GeForce Titan Xp)
  - Ubuntu 16.04, TensorFlow 1.8.0, CUDA 9.0
  - Horovod 0.12.1, NCCL v2.1, OpenMPI v3.0.0

**Executors**: TensorFlow (TF) + TF Eager (modified)

**LoC**: 4700 LoC / TF diff 771 LoC / CPython diff 1096 LoC
Evaluation Setup: Applications

11 models in 5 categories using various dynamic characteristics of Python

- Convolutional Neural Networks (CNN) LeNet, ResNet-50, Inception-v3
- Recurrent Neural Networks (RNN) LSTM, LM
- Recursive Neural Networks (TreeNN) TreeRNN, TreeLSTM
- Deep Reinforcement Learning (DRL) A3C, PPO
- Generative Adversarial Networks (GAN) AN, PIX2PIX
ImageNet Test Error with ResNet50

Test Error (%)

Symbolic

Imperative

Time (s)

Faster

36 GPUs
ImageNet Test Error with ResNet50

![ImageNet Test Error with ResNet50](ImageNet Test Error with ResNet50.png)

- **Symbolic**
- **JANUS**
- **Imperative**

**Test Error (%)** vs **Time (s)**

- 36 GPUs
- Faster
ImageNet Test Error with ResNet50

Test Error (%)

Symbolic

JANUS

3.4x Faster Convergence

Imperative

Time (s)

Faster

36 GPUs
ImageNet Test Error with ResNet50

Overlapping computation and communication

3.4x Faster Convergence

Faster
Model Convergence

- **RNN** (1B Test Perplexity with LM): 6 GPUs
- **TreeNN** (SST Test Accuracy with TreeLSTM): CPU
- **DRL** (Pong Episode Reward with PPO): 4 GPUs
- **GAN** (MNIST Discriminator Loss with AN): 1 GPU

**Faster**
Model Convergence

**RNN**
(1B Test Perplexity with LM)

6 GPUs

3.1x

**TreeNN**
(SST Test Accuracy with TreeLSTM)

CPU

18.4x

**DRL**
(Pong Episode Reward with PPO)

4 GPUs

3.2x

**GAN**
(MNIST Discriminator Loss with AN)

1 GPU

2.6x
## Normalized Training Throughput

<table>
<thead>
<tr>
<th>Model Type</th>
<th>Architecture</th>
<th>Symbolic Improvement</th>
<th>Imperative Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>LeNet</td>
<td>5x</td>
<td>3x</td>
</tr>
<tr>
<td></td>
<td>ResNet-50</td>
<td>8x</td>
<td>4x</td>
</tr>
<tr>
<td></td>
<td>Inception-v3</td>
<td>12x</td>
<td>6x</td>
</tr>
<tr>
<td>RNN</td>
<td>LSTM</td>
<td>40x</td>
<td>20x</td>
</tr>
<tr>
<td></td>
<td>LM</td>
<td>20x</td>
<td>10x</td>
</tr>
<tr>
<td>TreeNN</td>
<td>TreeRNN</td>
<td>30x</td>
<td>15x</td>
</tr>
<tr>
<td></td>
<td>TreeLSTM</td>
<td>40x</td>
<td>20x</td>
</tr>
<tr>
<td>DRL</td>
<td>A3C</td>
<td>15x</td>
<td>7.5x</td>
</tr>
<tr>
<td></td>
<td>PPO</td>
<td>15x</td>
<td>7.5x</td>
</tr>
<tr>
<td>GAN</td>
<td>AN</td>
<td>10x</td>
<td>5x</td>
</tr>
<tr>
<td></td>
<td>PIX2PIX</td>
<td>5x</td>
<td>2.5x</td>
</tr>
</tbody>
</table>

- **JANUS** shows significant improvement over Imperative and Symbolic approaches.
- **96.0% of Symbolic** improvement when compared to Imperative.
- **47.6x over Imperative** improvement when compared to Symbolic.

*Single machine w/ Single GPU*
### Normalized Training Throughput

<table>
<thead>
<tr>
<th>Model</th>
<th>CNN</th>
<th>RNN</th>
<th>TreeNN</th>
<th>DRL</th>
<th>GAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LeNet</td>
<td>LSTM</td>
<td>TreeRNN</td>
<td>A3C</td>
<td>AN</td>
</tr>
<tr>
<td></td>
<td>ResNet-50</td>
<td>LM</td>
<td>TreeLSTM</td>
<td>PPO</td>
<td>PIX2PIX</td>
</tr>
<tr>
<td></td>
<td>Inception-v3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Performance:
- **Imperative**
  - LeNet: 2x
  - ResNet-50: 3x
  - Inception-v3: 4x
- **JANUS**
  - LSTM: 5x
  - LM: 6x
- **Symbolic**
  - TreeRNN: 8x
  - TreeLSTM: 18.4x
  - A3C: 47.6x

*Note: JANUS outperforms both Imperative and Symbolic models in terms of normalized training throughput.*

**Hand-optimized GPU ops dominated execution time;**

**Working on applying further graph optimizations!**

---

**Single machine w/ Single GPU**
JANUS Speedup over Imperative Execution

<table>
<thead>
<tr>
<th>Category</th>
<th>Model</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>LeNet</td>
<td>3x</td>
</tr>
<tr>
<td></td>
<td>ResNet-50</td>
<td>2x</td>
</tr>
<tr>
<td></td>
<td>Inception-v3</td>
<td>2x</td>
</tr>
<tr>
<td>RNN</td>
<td>LSTM</td>
<td>8x</td>
</tr>
<tr>
<td></td>
<td>LM</td>
<td>4x</td>
</tr>
<tr>
<td>TreeNN</td>
<td>TreeRNN</td>
<td>18.4x</td>
</tr>
<tr>
<td></td>
<td>TreeLSTM</td>
<td>47.6x</td>
</tr>
<tr>
<td>DRL</td>
<td>A3C</td>
<td>5x</td>
</tr>
<tr>
<td></td>
<td>PPO</td>
<td>4x</td>
</tr>
<tr>
<td>GAN</td>
<td>AN</td>
<td>3x</td>
</tr>
<tr>
<td></td>
<td>PIX2PIX</td>
<td>2x</td>
</tr>
</tbody>
</table>

Single machine w/ Single GPU
JANUS Speedup over Imperative Execution: Breakdown

Single machine w/ Single GPU

Speedup “without” specialization by runtime profiling
## JANUS Speedup over Imperative Execution: Breakdown

### Single machine w/ Single GPU

<table>
<thead>
<tr>
<th>Category</th>
<th>Model</th>
<th>Imp. 2x</th>
<th>Imp. 3x</th>
<th>Imp. 4x</th>
<th>Imp. 5x</th>
<th>Imp. 6x</th>
<th>Imp. 7x</th>
<th>Imp. 8x</th>
<th>Imp. 18.4x</th>
<th>Imp. 47.6x</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>ResNet-50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inception-v3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RNN</td>
<td>LSTM</td>
<td></td>
<td></td>
<td>4x</td>
<td>5x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LM</td>
<td></td>
<td></td>
<td>2x</td>
<td>3x</td>
<td>4x</td>
<td>5x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TreeNN</td>
<td>TreeRNN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3x</td>
<td>4x</td>
<td>5x</td>
<td>6x</td>
<td>7x</td>
</tr>
<tr>
<td></td>
<td>TreeLSTM</td>
<td></td>
<td>3x</td>
<td>4x</td>
<td>5x</td>
<td>6x</td>
<td>7x</td>
<td>8x</td>
<td>18.4x</td>
<td>47.6x</td>
</tr>
<tr>
<td>DRL</td>
<td>A3C</td>
<td></td>
<td></td>
<td>2x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PPO</td>
<td></td>
<td>2x</td>
<td>3x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GAN</td>
<td>AN</td>
<td></td>
<td></td>
<td></td>
<td>2x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PIX2PIX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speedup “with” specialization by runtime profiling.
JANUS Speedup over Imperative Execution: Breakdown

- **CNNs**: LeNet, ResNet-50, Inception-v3
- **RNNs**: LSTM, LM
- **TreeNNs**: TreeRNN, TreeLSTM
- **DRLs**: A3C, PPO
- **GANs**: AN, PIX2PIX

**Single machine w/ Single GPU**

- **Type Specialization**
- **Control Flow Unrolling**
- **Bypassing Python Heap**

Composed of CNNs
Related Works

- Imperative to symbolic: one-shot converters
  - TensorFlow: defun, AutoGraph, Swift for TensorFlow, JAX, ...
  - PyTorch JIT trace, script
  - MXNet Gluon

- Cannot handle the dynamic semantics of Python **correctly & efficiently**
Conclusion

- Programmability and debuggability of imperative DL frameworks with the performance of symbolic DL frameworks
- Speculative graph generation and execution with runtime profiling
- Up to 47.6x speedup over imperative DL framework, within up to 4% difference compared to symbolic DL framework, while transparently and correctly executing imperative DL programs