PyTorch & Tensorflow

Yuntian He
• Paszke et al.,
  Automatic differentiation in PyTorch

• Martin Abadi et al.,
  Large-Scale Machine Learning on Heterogeneous Distributed Systems (TensorFlow)

• Agrawal et al.,
  TensorFlow Eager: A Multi-stage, Python-embedded DSL for Machine Learning

Links can be found: http://web.cse.ohio-state.edu/~panda.2/5194/reading.html
TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems

Martín Abadi et al.
What is TensorFlow?

- TensorFlow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms.
- Supported on a variety of hardware platforms
  - Mobile phones, PCs, distributed machines with multi-GPUs
- Python and C++ front ends
- Open source: www.tensorflow.org
History of TensorFlow

DistBelief
- 2011
- First generation scalable distributed training and inference system
- Machine Learning system built for deep neural networks

Tensorflow
- 2015
- 2nd generation system for implementation and deployment of largescale machine learning models
- More flexible programming model
- Better performance
Applications

- Google search
- Google Translate
- YouTube
- Google advertising
Programming Model

- Dataflow-like model
- Directed Graph with a set of nodes
- Each node has zero or more inputs and outputs
- Each node denotes an operation
Edges

- Normal **edges**: where tensors flow
  - Tensor is an N-dimensional array or list
- Control Dependencies
  - To enforce happens-before relationships and orderings
Operations and Kernels

- **Operations**: node that perform computation on tensors
  - Examples: add, matmul, softmax..
- Polymorphic: support different tensor element types
- **Kernels**: particular implementation of an operation that can be run on a particular type of device
Sessions

- Encapsulate the environment in which Operation objects are executed, and Tensor objects are evaluated
- Provide an interface for the client program to communicate with Master and Worker processes
- `Extend()`: augment the current graph
- `Run()`: run one time of the computation graph
  - Takes a set of output names (for output)
  - Feed some tensors into the graph (for input)

```python
tf.Session.run([Output wanted], feed_dict={})
```
Variables

- A special kind of operations
- Return a persistent, mutable tensor
- ML parameters are typically in Variables
Implementation

- Tensors
  - a variety of element types
- Devices
  - Identified by its name, type, index within the worker
- Client
  - Use the session interface to communicate with master/worker processes
- Master
  - Provide instructions to worker processes
- Worker
  - Arbitrate access to computational devices
  - Execute graph nodes
Implementation

Local

- Client, master, and worker run on a single machine
- Single operating system process

Distributed

- Client, master, and workers run in different processes on different machines
Single-device execution

- The simplest case!
- Maintain a count for each node representing its dependences that have not executed
- A node is eligible if its count drops to zero
Multi-device execution

- Once you have multiple devices...
  - Map the computation nodes onto the set of devices
  - How to manage communication across devices
Node Placement

- Cost model
  - estimates of the sizes of input/output tensors for each graph node
  - estimates of required time for computation

- Device selection
  - Greedy heuristic based on effect of node placement on completion time
  - User can control by specifying device constraints
Cross-device Communication

- Insertion of Send/Receive nodes
- A needed tensor only flow once
- Decentralize the nodes into its worker
  - Master doesn’t need to be involved
Distributed execution

- Send/Receive nodes communicate via TCP/RDMA
- Fault Tolerance
  - Failure detection
    - Error in Send/Receive pair
    - Periodic health-checks from master to every worker
  - Once detected, abort and restart from scratch
    - Variable is connected to Save/Restore Nodes
Gradient computation

- Forward graph and backward graph

- Usage

\[
[db, dW, dx] = tf.gradients(C, [b, W, x])
\]
• Run a subgraph of the entire graph

• Recall `tf.Session.run([Output wanted], feed_dict={})`

• **Feed**: tensors are patched directly into any operation in the graph

• **Fetch**: output can be fetched by passing tensors to retrieve
Control flow

- Operations and classes that control the execution of operations and add conditional dependencies to graphs
- Handle cyclic dataflow graphs
  - Switch, Merge, Enter, Leave, NextIteration
Other extensions

- Device constraints
- Input operations
  - Read data directly from the underlying storage system into memory
- Queues
  - Allow different portions of graph to execute asynchronously
- Containers
  - Managing longer-lived mutable state
• Common subexpression elimination
• Controlling data combination and memory usage
  • Particularly for GPU
• Asynchronous Kernels
• Optimized libraries for kernel implementation
  • Extend existing libraries for arbitrary dimensionality tensor operations
• Loss Compression
  • Reduce precision for more efficient data transmission
Common programming idioms

- How to speed up the training of neural networks?
- Data Parallel Training
Common programming idioms

- Model Parallel Training
• **TensorBoard** for visualization
  • Computation graph
  • Summary statistics
• EEG for performance tracing
Conclusion

- TensorFlow is a flexible model for implementing ML algorithms
- Support for distributed implementation
- Provides graph visualization using TensorBoard
- Open source
- Currently being deployed in many research and commercial projects
Automatic differentiation in PyTorch

Adam Paszke et al.
What is PyTorch?

- A machine learning library based on Torch
- Developed by Facebook’s AI Research Lab

- Two high-level features:
  - Tensor computing with GPU acceleration
  - DNNs built on an automatic differentiation system
Automatic Differentiation

- Existing DL libraries followed by PyTorch:
  - Chainer
  - HIPS autograd
  - twitter-autograd
## Dynamic define-by-run execution

<table>
<thead>
<tr>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTorch</td>
<td>Tensorflow 1.x, Caffe</td>
</tr>
<tr>
<td>Define-by-run</td>
<td>Define a model before runtime</td>
</tr>
<tr>
<td>No special Session interfaces</td>
<td>Communication via Session</td>
</tr>
<tr>
<td>Not needed</td>
<td>Placeholder needed for input</td>
</tr>
<tr>
<td><strong>backward approach</strong></td>
<td>differentiated symbolically</td>
</tr>
</tbody>
</table>
Example

**Dynamic:** differentiated *when needed*, more flexible

**Static:** build a backward computation graph that can be *reused* many times

```python
a = torch.tensor([3.0, 1.0], requires_grad=True)
b = a * a
loss = b.mean()
loss.backward()  # Success
loss.backward()  # RuntimeError

a = torch.tensor([3.0, 1.0], requires_grad=True)
b = a * a
loss = b.mean()
loss.backward()  # Success
```
Immediate eager execution

- Immediate tensor computation when encountered
- Record only what is necessary for differentiation
- Unable to optimize the whole network
Other properties of PyTorch

- In-place operations
  - Address issues such as invalidation and aliasing
- No tape
  - Records only the relevant computation subgraph
- Core logic in C++
  - Lower overhead
Interface

backward()

• Chainer-style
• Compute the gradient of all involved Variables
• Update the .grad attributes

```python
input = torch.ones([2, 2], requires_grad=False)
w1 = torch.tensor(2.0, requires_grad=True)
w2 = torch.tensor(3.0, requires_grad=True)

l1 = input * w1
l2 = l1 + w2

loss = l2.mean()
print(w1.grad, w2.grad)  # None None
loss.backward()
print(w1.grad, w2.grad)  # tensor(1.) tensor(1.)
```
• HIPS autograd-style
• Return a tuple of wanted gradients
• No update on .grad attributes
Interface

Variable flags

- **requires_grad**
  - If **any** input requires grad, the output requires grad

- **volatile**
  - If **any** input is volatile, the output is volatile

- Disable unneeded derivative computation
- Allow user to inspect gradients
  - Print the gradient whenever it is computed

```python
loss = l2.mean()
loss.register_hook(lambda grad: print('loss grad: ', grad))
l1.register_hook(lambda grad: print('l1 grad: ', grad))

loss.backward()
... 
loss grad:  tensor(1.)
l1 grad:  tensor([[0.2500, 0.2500],
               [0.2500, 0.2500]]
```
• Custom differentiable operations
• Can be used to make other Python libraries differentiable
Implementation

- Memory management
  - Free unneeded intermediate values
  - Reference counted regime
  - Avoid reference cycles
- C++ operators
In-place operations

• Invalidation

```python
y = x.tanh()
y.add_(3)
y.backward()
```
version counter: raise error when values don’t match

• Aliasing

```python
y = x[:2]
x.add_(3)
y.backward()
```

x, y share storage
Conclusion

- Dynamic, define-by-run execution
- Intermediate, eager execution
TensorFlow Eager: a Multi-stage, Python-embedded DSL for Machine Learning

Akshay Agrawal et al.
Introduction

- Differentiable programs
  - Data dependent control flow, complicated data structure
- Domain-specific Languages (DSL)
  - Embedded in a host language (mostly Python)
• Domain-specific Languages (DSL)
  • Imperative DSL
    • Full extent of host language
    • Familiar programming model
    • Rapid development
    • Cons: Performance bottlenecked on interpreter
  • Declarative DSL
    • Separate the specification of models from executions
    • Define-before-run: stage the models as dataflow graphs
    • Allow compiler optimizations and simplify deployment
    • Cons: steep learning curves, limited programming constructs
Bridging the gap

- An Ideal DSL would offer
  - the flexibility of imperative execution
  - the benefits of declarative programming

- **TensorFlow Eager**: mix imperative and staged execution in Python via [multi-stage programming](#)
TensorFlow Eager

- Execute imperatively
  - `tf.ones()`: an actual numerical data, not just a tensor

```python
import tensorflow as tf
tf.enable_eager_execution()

def add(a):
    return a + a

add(tf.ones([2, 2])) # [[2., 2.], [2., 2.]]
```
TensorFlow
Eager

- **Graph function**
  - Decorator `@tf.function`: returns an executable dataflow graph
  - A Python-like function compiled to optimized codes
  - Functions are polymorphic

- No Session anymore in sight!

```python
import tensorflow as tf
tf.enable_eager_execution()

# Calling `add` runs an op that computes an equivalent dataflow graph
@tf.function
def add(a):
    return a + a

add(tf.ones([2, 2]))  # [[2., 2.], [2., 2.]]
add(tf.ones([2, 2], dtype=tf.int32))  # functions are polymorphic
```
Design principles

Recognizable to Python programmers
- privilege imperative execution
- seamlessly embed into Python

Smooth way of deployment on heterogenous devices
- stage imperative code as dataflow graphs
Multi-stage workflow
• JIT tracer, but not a Python compiler
  • Extract a TensorFlow graph
• Polymorphic
  • Traces dtypes and shapes of tensor inputs, and runtime values of non-tensor arguments
  • May have multiple graph functions for one single Python function
• Lexcial closure
• Composable
Automatic differentiation

- Reverse-mode automatic differentiation via tracing
- Can be nested for computing higher-order derivatives
- Allow user to control which parts of computation are traced
- Now support for computing Jacobians
• Copy data across various types of devices

```
  a = tf.constant(1.0)  # stored on CPU
  b = a.gpu()           # stored on GPU
```

• Copy the input to the correct device

```
# stored on CPU
  a = tf.constant(1.0)
  b = tf.constant(2.0)

with tf.device("/gpu:0"):
  c = tf.add(a, b)

  assert c.numpy() == 3.0
```
Distributed execution

- Single central server + several worker servers
  - Workers add its devices to the pool of main program
Implementation

- Not large implementation
  - 4K lines for imperative runtime, 2K for staging, etc.
- Similar to Tensorflow
  - Cross-platform
  - Run in parallel, across multiple CPUs/GPUs
  - Primitive operations
- Difference
  - Staged execution: more Pythonic, representing the exact computation of interest
• Training a ResNet-50 on a GPU
Evaluation

- Training L2HMC on a CPU
Conclusion

- TensorFlow Eager is a multi-stage, Python-embedded DSL for ML
- Imperative behavior makes TFE more suitable for beginners
- Allow a trade-off between the benefits of imperative programming and excellent performance of dataflow graphs
- Used in TensorFlow 2.0 by default
Thanks

Q & A