decisions and challenges of writing an imperative deep learning framework

Adam Paszke, Sam Gross, Soumith Chintala & the team
Facebook AI Research
Overview of the talk

Computation Graph Toolkits → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

- Added features on top of computation graphs
- Design Challenges
- Overheads
- Lazy Execution
- JIT Fusion
- Subgraph caching
- Differences from AOT

Implementation
Advantages & Disadvantages

- Declarative
- Imperative
Deep Learning Frameworks
Deep Learning Frameworks

In addition to Computation Graph Toolkits

- Provide gradient computation
  - Gradient of one variable w.r.t. any variable in graph
Deep Learning Frameworks

In addition to Computation Graph Toolkits

- Provide gradient computation
  - Gradient of one variable w.r.t. any variable in graph

\[
d(i2h)/d(W_h)
\]
Deep Learning Frameworks

In addition to Computation

Graph Toolkits

- Provide gradient computation
  - Gradient of one variable w.r.t. any variable in graph
- Provide integration with high performance DL libraries like CuDNN
Computation Graph Toolkits
Computation Graph Toolkits

Declarative Toolkits

Caffe

- Computation Graphs
- Deep Learning Frameworks
- Imperative Frameworks
- JIT Compilation
Computation Graph Toolkits

Declarative Toolkits

• Declare a computation
  • with placeholder variables
• Compile it
• Run it in a Session
Computation Graphs

Declarative Toolkits

• Declare a computation
  • with placeholder variables
• Compile it
• Run it in a Session

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```
Computation Graph Toolkits

• Declare a computation
  • with placeholder variables
  • Compile it
  • Run it in a Session

A separate turing-complete Virtual Machine

```
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```
Computation Graph Toolkits

Declarative Toolkits
- Declare a computation
  - with placeholder variables
- Compile it
- Run it in a Session

Can handle loops, conditionals (if, scan, while, etc.)

```python
from __future__ import division, print_function
import tensorflow as tf

def fn(previous_output, current_input):
    return previous_output + current_input

elems = tf.Variable([1.0, 2.0, 2.0, 2.0])
elems = tf.identity(elems)
initializer = tf.constant(0.0)
out = tf.scan(fn, elems, initializer=initializer)

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    print(sess.run(out))
```
Computation Graph Toolkits

• Declare a computation
  • with placeholder variables
• Compile it
• Run it in a Session

Has it’s own execution engine

# Computation Graphs

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```
Computation Graphs

Declarative Toolkits
- Declare a computation
- with placeholder variables
- Compile it
- Run it in a Session

Has it’s own compiler
- fuse operations
- reuse memory
- do optimizations

```python
import tensorflow as tf
import numpy as np
trx = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33
x = tf.placeholder("float")
y = tf.placeholder("float")
def model(x, w):
    return tf.multiply(x, w)
w = tf.Variable(0.0, name="weights")
y_model = model(x, w)
cost = tf.square(y - y_model)
train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    for i in range(100):
        for (x, y) in zip(trx, trY):
            sess.run(train_op, feed_dict={x: x, y: y})
    print(sess.run(w))
```
**Computation Graph**

Declarative Toolkits
- Declare a computation
- with placeholder variables
- Compile it
- Run it in a Session

Graph can be serialized easily

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```
Computation Graph

Declarative Toolkits

• Declare a computation
  • with placeholder variables
• Compile it
• Run it in a Session

Own Virtual Machine
Computation Graphs

Declarative Toolkits
- Declare a computation with placeholder variables
- Compile it
- Run it in a Session

Deep Learning Frameworks

Imperative Frameworks

JIT Compilation

Own Virtual Machine
- separate debugging tools
Computation Graphs

Declarative Toolkits

• Declare a computation
  • with placeholder variables
• Compile it
• Run it in a Session

Own Virtual Machine
- separate debugging tools
- non-linear thinking for user

```python
import tensorflow as tf
import numpy as np

trX = np.linspace(-1, 1, 101)
trY = 2 * trX + np.random.randn(*trX.shape) * 0.33

X = tf.placeholder("float")
Y = tf.placeholder("float")

def model(X, w):
    return tf.multiply(X, w)

w = tf.Variable(0.0, name="weights")
y_model = model(X, w)

cost = tf.square(Y - y_model)

train_op = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for i in range(100):
        for (x, y) in zip(trX, trY):
            sess.run(train_op, feed_dict={X: x, Y: y})

    print(sess.run(w))
```
Imperative Toolkits
Computation Graph Toolkits

Imperative Toolkits

• Run a computation
• computation is run

Deep Learning Frameworks

JIT Compilation

HIPS Autograd

Dynet

Chainer

PyTorch

Computation Graphs ➔ Deep Learning Frameworks ➔ Imperative Frameworks ➔ JIT Compilation
Computation Graph

Imperative Toolkits

- Run a computation
- computation is run!

```
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33
w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph

Imperative Toolkits

- Run a computation
- Computation is run!

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits

- Run a computation
- Computation is run!
- No separate execution engine

```python
import torch
from torch.autograd import Variable

trx = torch.linspace(-1, 1, 101)
try = 2 * trx + torch.randn(*trx.size()) * 0.33

w = Variable(trx.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trx, try):
        X = Variable(x)
        Y = Variable(y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits

• Run a computation
• computation is run!
• no separate execution engine
• debugging is easy

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits

• Run a computation
• computation is run!
• no separate execution engine
• debugging is easy

Oldest debugging method of all time

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

    w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits

- Run a computation
- Computation is run!
- No separate execution engine
- Debugging is easy

Oldest debugging method of all time

```
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits
- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy

Oldest debugging method of all time

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```

print("hello")
y = foo(x)
print("hello2")
Computation Graph Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy

Oldest debugging method of all time

print("hello")
y = foo(x)
print("hello2")

```
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33
w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)
        print(Y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits

- Run a computation
- Computation is run!
- No separate execution engine
- Debugging is easy
- Linear program flow
- Linear thinking for user

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)
        print(Y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graphs

Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- Cannot compile program
- Linear program flow
- Linear thinking for user

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)
        print(Y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits

Imperative Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- debugging is easy
- Linear program flow
  - Linear thinking for user

Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation
Computation Graph Toolkits

- Run a computation
- computation is run!
- no separate execution engine
- Linear program flow

Imperative Toolkits

Cannot compile program
Cannot optimize
Cannot do static analysis

```python
import torch
from torch.autograd import Variable

trX = torch.linspace(-1, 1, 101)
trY = 2 * trX + torch.randn(*trX.size()) * 0.33

w = Variable(trX.new([0.0]), requires_grad=True)

for i in range(100):
    for (x, y) in zip(trX, trY):
        X = Variable(x)
        Y = Variable(y)
        print(Y)

        y_model = X * w.expand_as(X)
        cost = (Y - y_model) ** 2
        cost.backward(torch.ones(*cost.size()))

        w.data = w.data + 0.01 * w.grad.data

print(w)
```
Computation Graph Toolkits

• Run a computation
• computation is run!
• no separate execution engine

Imperative Toolkits

• Cannot compile program
• Cannot optimize
• Cannot do static analysis
(more on this later)
Imperative Frameworks
Imperative Frameworks: PyTorch

Graph is built on the fly

```python
from torch.autograd import Variable
```
Impressive Frameworks: PyTorch

Graph is built on the fly

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```
**Imperative Frameworks: PyTorch**

Graph is built on the fly

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
```
Imperative Frameworks: PyTorch

from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
**Imperative Frameworks: PyTorch**

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```
Impreative Frameworks: PyTorch

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
```
Imperative Frameworks: PyTorch

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

Diagram: Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation
Imperative Frameworks: PyTorch

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

Hence, graph construction has to be FAST
Imperative Frameworks: PyTorch

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

Initially written in Python
Imperative Frameworks: PyTorch

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

Overhead too high
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
Imperative Frameworks: PyTorch

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

Moved to CPython

used Flame graphs to find and optimize hotspots
Imperative Frameworks: PyTorch

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
next_h.backward(torch.ones(1, 20))
```

Moved to CPython

used Flame graphs to find and optimize hotspots
Imperative Frameworks: PyTorch

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```

Overall speed as fast as declarative frameworks
## Performance stats from Nov 2016

<table>
<thead>
<tr>
<th>Task</th>
<th>Torch</th>
<th>PyTorch</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-101</td>
<td>544ms</td>
<td>516ms</td>
</tr>
<tr>
<td></td>
<td>10GB (5,4GB)</td>
<td>10GB (5,6GB)</td>
</tr>
<tr>
<td>ResNet-101 2GPU</td>
<td>580ms</td>
<td>640ms</td>
</tr>
<tr>
<td></td>
<td>10GB (5,6GB)</td>
<td>10GB (5,6GB)</td>
</tr>
<tr>
<td>Penn Treebank 2-layer LSTM</td>
<td>57ms</td>
<td>62ms</td>
</tr>
<tr>
<td></td>
<td>865MB</td>
<td>370MB</td>
</tr>
</tbody>
</table>
Performance stats from Nov 2016

<table>
<thead>
<tr>
<th>Task</th>
<th>Torch</th>
<th>PyTorch</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-101</td>
<td>544ms</td>
<td>516ms</td>
</tr>
<tr>
<td></td>
<td>10GB (5,4GB)</td>
<td>4,9GB</td>
</tr>
<tr>
<td>ResNet-101 2GPU</td>
<td>580ms</td>
<td>640ms</td>
</tr>
<tr>
<td></td>
<td>10GB (5,6GB)</td>
<td>4,9GB</td>
</tr>
<tr>
<td>Penn Treebank 2-layer LSTM</td>
<td>57ms</td>
<td>62ms</td>
</tr>
<tr>
<td></td>
<td>865MB</td>
<td>370MB</td>
</tr>
</tbody>
</table>

Pure Python internals = **140ms**
Graph Construction Speed

- PyTorch: nanoseconds to microseconds
- TensorFlow: milliseconds to several seconds
- Theano: seconds to minutes (hours?)
- MXNet: I don't know -> ask your instructor :)

Computation Graphs → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation
Additional valuable features in PyTorch

• Low memory usage even without a static optimizer
## Performance stats from Nov 2016

<table>
<thead>
<tr>
<th>Task</th>
<th>Torch</th>
<th>PyTorch</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-101</td>
<td>544ms</td>
<td>10GB (5.4GB)</td>
</tr>
<tr>
<td>ResNet-101 2GPU</td>
<td>580ms</td>
<td>10GB (5.6GB)</td>
</tr>
<tr>
<td>Penn Treebank 2-layer LSTM</td>
<td>57ms</td>
<td>865MB</td>
</tr>
</tbody>
</table>
Additional valuable features in PyTorch

- Low memory usage even without a static optimizer
- Intermediate buffers are always freed
- Developers given constructs to allocate temporary buffers
  - save_for_backward
  - requires_grad
Additional valuable features in PyTorch

- Low memory usage even without a static optimizer
- Intermediate buffers are always freed
- Developers given constructs to allocate temporary buffers
  - `save_for_backward`
  - `requires_grad`
- In-place operations
JIT Compilation
JIT Compilation

• Possible in Imperative Frameworks
• The key idea is deferred or lazy evaluation

- \( y = x + 2 \)
- \( z = y \times y \)
- # nothing is executed yet, but the graph is being constructed
- print(z) # now the entire graph is executed: \( z = (x+2) \times (x+2) \)

• We can do just in time compilation on the graph before execution
Lazy Evaluation

\[ x = \text{Variable}(\text{torch.randn}(1, 10)) \]
\[ \text{prev}\_h = \text{Variable}(\text{torch.randn}(1, 20)) \]
\[ W\_h = \text{Variable}(\text{torch.randn}(20, 20)) \]
\[ W\_x = \text{Variable}(\text{torch.randn}(20, 10)) \]

\[ i2h = \text{torch.mm}(W\_x, x\_t()) \]
\[ h2h = \text{torch.mm}(W\_h, \text{prev}\_h\_t()) \]
\[ \text{next}\_h = i2h + h2h \]
\[ \text{next}\_h = \text{next}\_h\_tanh() \]

\[ \text{next}\_h\_backward(\text{torch.ones}(1, 20)) \]
Lazy Evaluation

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
```
Lazy Evaluation

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

print(next_h)
```

Data accessed. Execute graph.
Lazy Evaluation

- A little bit of time between building and executing graph
  - Use it to compile the graph just-in-time
JIT Compilation

• Fuse and optimize operations

Fuse operations. Ex:

```python
1  x = [0, 1, 2, 3, 4]
2  for i in range(len(x)):
3      x[i] = x[i] + 1
4  for i in range(len(x)):
5      x[i] = x[i] * 2
6
7  # Fused
8  for i in range(len(x)):
9      x[i] = (x[i] + 1) * 2
```
JIT Compilation

- Cache subgraphs

I’ve seen this part of the graph before, let me pull up the compiled version from cache.
Compilation benefits

Kernel fusion

Out-of-order execution

Automatic work placement

ReLU
BatchNorm
Conv2d

1 2 3
3 1 2

Node 0
GPU 1
CPU

Node 0
GPU 0

Node 1
GPU 1

Node 1
GPU 0

Node 0
GPU 0
JIT Compilation

• Possible in Dynamic Frameworks
• The key idea is deferred or lazy evaluation
  - y = x + 2
  - z = y * y
  - # nothing is executed yet, but the graph is being constructed
  - print(z) # now the entire graph is executed: z = (x+2) * (x+2)

• We can do just in time compilation on the graph before execution
• We can cache repeating patterns in subsets of the graph
  - to avoid recompilation
• Compiler is very different from Ahead-of-time compiler
  - fast compilation
  - compile traces rather than full graph
Review

Computation Graph Toolkits → Deep Learning Frameworks → Imperative Frameworks → JIT Compilation

Declarative & Imperative

Added features on top of computation graphs

Design Challenges & Overheads

Lazy Execution, JIT Fusion, Subgraph caching differences from AOT

Implementation

Advantages & Disadvantages
http://pytorch.org
Released Jan 18th
40,000+ downloads
250+ community repos
4200+ user posts
330k+ forum views

With ❤ from

You?