PyTorch

Researcher Edition

Adam Paszke, Sam Gross, Soumith Chintala, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Alban Desmaison, Andreas Kopf, Edward Yang, Zach Devito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy & Team
What is PyTorch?
What is PyTorch?

- Automatic differentiation engine
- Ndarray library with GPU support
- Gradient based optimization package
- Utilities (data loading, etc.)

Deep Learning

Reinforcement Learning

Numpy-alternative
**ndarray library**

- `np.ndarray <-> torch.Tensor`
- 200+ operations, similar to numpy
- Very fast acceleration on NVIDIA GPUs
import numpy as np

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# Randomly initialize weights
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

import torch

dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random input and output data
x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

# Randomly initialize weights
w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6
for t in range(500):
    # Forward pass: compute predicted y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Update weights using gradient descent
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
Tensors are similar to numpy’s ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```python
from __future__ import print_function
import torch

x = torch.Tensor(5, 3)
print(x)
```

Construct a 5x3 matrix, uninitialized:

```
1.00000e-25
0.4136  0.0000  0.0000
0.0000  1.6519  0.0000
1.6518  0.0000  1.6519
0.0000  1.6518  0.0000
1.6520  0.0000  1.6519
```

[torch.FloatTensor of size 5x3]
ndarray / Tensor library

Construct a randomly initialized matrix

```python
x = torch.rand(5, 3)
print(x)
```

Out:
```
0.2598  0.7231  0.8534
0.3928  0.1244  0.5110
0.5476  0.2700  0.5856
0.7288  0.9455  0.8749
0.6663  0.8230  0.2713
[torch.FloatTensor of size 5x3]
```

Get its size

```python
print(x.size())
```

Out:
```
torch.Size([5, 3])
```
You can use standard numpy-like indexing with all bells and whistles!

```python
print(x[:, 1])
```

Out:

```
0.7231
0.1244
0.2700
0.9455
0.8230
[torch.FloatTensor of size 5]
```
ndarray / Tensor library

```python
y = torch.rand(5, 3)
print(x + y)
```

Out:

```
0.7931  1.1872  1.6143
1.1946  0.4669  0.9639
0.7576  0.8136  1.1897
0.7431  1.8579  1.3400
0.8188  1.1041  0.8914
[torch.FloatTensor of size 5x3]
```
NumPy bridge

Converting torch Tensor to numpy Array

```python
a = torch.ones(5)
print(a)
```

Out:
```
1
1
1
1
1
[torch.FloatTensor of size 5]
```

```python
b = a.numpy()
print(b)
```

Out:
```
[ 1.  1.  1.  1.  1.]
```
NumPy bridge

Converting torch Tensor to numpy Array

```python
a = torch.ones(5)
print(a)
```

Out:
```
[ 1.  1.  1.  1.  1.]
```

```
[torch.FloatTensor of size 5]
```

```python
b = a.numpy()
print(b)
```

Out:
```
[ 1.  1.  1.  1.  1.]
```
NumPy bridge

See how the numpy array changed in value.

```python
a.add_(1)
print(a)
print(b)
```

Out:
```
2
2
2
2
2
[torch.FloatTensor of size 5]
[ 2. 2. 2. 2. 2.]
```
NumPy bridge

Converting numpy Array to torch Tensor

See how changing the np array changed the torch Tensor automatically

```python
import numpy as np
da = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

Out:

```
[ 2.  2.  2.  2.  2.]
2
2
2
2
2
[torch.DoubleTensor of size 5]
```

All the Tensors on the CPU except a CharTensor support converting to NumPy and back.
Seamless GPU Tensors

CUDA Tensors

Tensors can be moved onto GPU using the `.cuda` function.

```python
# let us run this cell only if CUDA is available
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    x + y
```
automatic differentiation engine for deep learning and reinforcement learning
PyTorch Autograd

```
from torch.autograd import Variable
```
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))
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())
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
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
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()
```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))
```

PyTorch Autograd

Add

MM

Tanh

MM

i2h

Add

h2h

next_h

W_h

h

W_x

x
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()
Optimization package

SGD, Adagrad, RMSProp, LBFGS, etc.

```python
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```
Research workflows

Pain Points

Core Philosophy (of PyTorch)

Upcoming features

what do they look like in the deep learning space

how does PyTorch deal with them?
Research Workflows
Research Workflow

1. idea / theory → design experiments → pick datasets / environments
2. Add results to paper ← Training & Validation ← Implement models
Research Workflow

1. Literature Review
2. Idea / Theory
3. Design Experiments
4. Training & Validation
5. Pick datasets / environments
6. Add results to paper
7. Implement models
Research Workflow

1. Literature Review

2. Idea / Theory → Design Experiments → Pick Datasets / Environments

3. Add Results to Paper → Training & Validation → Implement Models
Work items in practice

- Writing Dataset loaders
- Building models
- Implementing Training loop
- Checkpointing models
- Interfacing with environments
- Building optimizers
- Dealing with GPUs
- Building Baselines
Work items in practice

- Writing Dataset loaders
- Building models
- Implementing Training loop
- Checkpointing models
- Interfacing with environments
- Building optimizers
- Dealing with GPUs
- Building Baselines

Python + PyTorch - an environment to do all of this
Writing Data Loaders

- every dataset is slightly differently formatted
Writing Data Loaders

- every dataset is slightly differently formatted
- have to be preprocessed and normalized differently
Writing Data Loaders

• every dataset is slightly differently formatted
• have to be preprocessed and normalized differently
• need a multithreaded Data loader to feed GPUs fast enough
Writing Data Loaders

PyTorch solution:

- share data loaders across the community!

Datasets

The following dataset loaders are available:

- MNIST
- COCO (Captioning and Detection)
- LSUN Classification
- ImageFolder
- Imagenet-12
- CIFAR10 and CIFAR100
- STL10
- SVHN
- PhotoTour

Datasets have the API: `__getitem__` - `__len__`. They all subclass from `torch.utils.data.Dataset`. Hence, they can all be multi-threaded (python multiprocessing) using standard `torch.utils.data.DataLoader`.
Writing Data Loaders

PyTorch solution:
• share data loaders across the community!

This repository consists of:
• `torchtext.data`: Generic data loaders, abstractions, and iterators for text
• `torchtext.datasets`: Pre-built loaders for common NLP datasets
• (maybe) `torchtext.models`: Model definitions and pre-trained models for pytorch. The situation is not the same as vision, where people can download a pre-trained model and make it useful for other tasks -- it might make more sense to leave NLP models as they are.
Writing Data Loaders

PyTorch solution:

• use regular Python to write Datasets:
  leverage existing Python code
Writing Data Loaders

PyTorch solution:

• use regular Python to write Datasets:
  leverage existing Python code

Example: ParlAI

ParlAI (pronounced "par-lay") is a framework for dialog AI research, implemented in Python.

Its goal is to provide researchers:

• a unified framework for training and testing dialog models
• multi-task training over many datasets at once
• seamless integration of Amazon Mechanical Turk for data collection and human evaluation

Over 20 tasks are supported in the first release, including popular datasets such as SQuAD, bAbI tasks, MCTest, WikiQA, WebQuestions, SimpleQuestions, WikiMovies, QACNN & QADailyMail, CBT, BookTest, bAbI Dialog tasks, Ubuntu Dialog, OpenSubtitles, Cornell Movie and VQA-COCO2014.
Writing Data Loaders

PyTorch solution:
• Code in practice
if opt.dataset in ['imagenet', 'folder', 'lfw']:
    # folder dataset
    dataset = dset.ImageFolder(root=opt.dataroot,
                               transform=transforms.Compose([
                                               transforms.Scale(opt.imageSize),
                                               transforms.CenterCrop(opt.imageSize),
                                               transforms.ToTensor(),
                                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                               ]))
elif opt.dataset == 'lsun':
    dataset = dset.LSUN(db_path=opt.dataroot, classes=['bedroom_train'],
                        transform=transforms.Compose([]
                                               transforms.Scale(opt.imageSize),
                                               transforms.CenterCrop(opt.imageSize),
                                               transforms.ToTensor(),
                                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                        ]))
elif opt.dataset == 'cifar10':
    dataset = dset.CIFAR10(root=opt.dataroot, download=True,
                           transform=transforms.Compose([]
                                               transforms.Scale(opt.imageSize),
                                               transforms.ToTensor(),
                                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))

assert dataset
data_loader = torch.utils.data.DataLoader(dataset, batch_size=opt.batchSize,
                                          shuffle=True, num_workers=int(opt.workers))
def __init__(self, root, annFile, transform=None, target_transform=None):
    from pycocotools.coco import COCO
    self.root = os.path.expanduser(root)
    self.coco = COCO(annFile)
    self.ids = list(self.coco.ims.keys())
    self.transform = transform
    self.target_transform = target_transform

def __getitem__(self, index):
    """
    Args:
    index (int): Index

    Returns:
    tuple: Tuple (image, target). target is a list of captions for the image.
    """
    coco = self.coco
    img_id = self.ids[index]
    ann_ids = coco.getAnnIds(imgIds=img_id)
    anns = coco.loadAnns(ann_ids)
    target = [ann['caption'] for ann in anns]

    path = coco.loadImgs(img_id)[0]['file_name']

    img = Image.open(os.path.join(self.root, path)).convert('RGB')
    if self.transform is not None:
        img = self.transform(img)

    if self.target_transform is not None:
        target = self.target_transform(target)

    return img, target

def __len__(self):
    return len(self.ids)
Interfacing with environments

Cars

Video games

Internet

Measurement and training for artificial intelligence.
Interfacing with environments

Cars

Video games

Pretty much every environment provides a Python API
Interfacing with environments

Cars  Video games

Natively interact with the environment directly
Debugging

- PyTorch is a Python extension
Debugging

• PyTorch is a Python extension
• Use your favorite Python debugger
Debugging

- PyTorch is a Python extension
- Use your favorite Python debugger
Debugging

• PyTorch is a Python extension
• Use your favorite Python debugger
• Use the most popular debugger:
Debugging

• PyTorch is a Python extension
• Use your favorite Python debugger
• Use the most popular debugger:
  \[\text{print}(\text{foo})\]
Identifying bottlenecks

- PyTorch is a Python extension
- Use your favorite Python profiler
Identifying bottlenecks

- PyTorch is a Python extension
- Use your favorite Python profiler: Line_Profiler
Compilation Time

• PyTorch is written for the impatient
Compilation Time

• PyTorch is written for the impatient
• Absolutely no compilation time when writing your scripts whatsoever
Compilation Time

• PyTorch is written for the impatient
• Absolutely no compilation time when writing your scripts whatsoever
• All core kernels are pre-compiled
Ecosystem

• Use the entire Python ecosystem at your will
Ecosystem

• Use the entire Python ecosystem at your will
• Including SciPy, Scikit-Learn, etc.
Ecosystem

• Use the entire Python ecosystem at your will
  • Including SciPy, Scikit-Learn, etc.
Ecosystem

- Use the entire Python ecosystem at your will
- Including SciPy, Scikit-Learn, etc.
Ecosystem

• A shared model-zoo:

We provide pre-trained models for the ResNet variants and AlexNet, using the PyTorch torch.utils.model_zoo. These can constructed by passing pretrained=True:

```python
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
```
Ecosystem

• A shared model-zoo:

We evaluate the performance of popular dataset and models with linear quantized method. The bit-width of running mean and running variance in BN are 10 bits for all results. (except for 32-float)

<table>
<thead>
<tr>
<th>Model</th>
<th>32-float</th>
<th>12-bit</th>
<th>10-bit</th>
<th>8-bit</th>
<th>6-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>98.42</td>
<td>98.43</td>
<td>98.44</td>
<td>98.44</td>
<td>98.32</td>
</tr>
<tr>
<td>SVHN</td>
<td>96.03</td>
<td>96.03</td>
<td>96.04</td>
<td>96.02</td>
<td>95.46</td>
</tr>
<tr>
<td>CIFAR10</td>
<td>93.78</td>
<td>93.79</td>
<td>93.80</td>
<td>93.58</td>
<td>90.86</td>
</tr>
<tr>
<td>CIFAR100</td>
<td>74.27</td>
<td>74.21</td>
<td>74.19</td>
<td>73.70</td>
<td>66.32</td>
</tr>
<tr>
<td>STL10</td>
<td>77.59</td>
<td>77.65</td>
<td>77.70</td>
<td>77.59</td>
<td>73.40</td>
</tr>
<tr>
<td>AlexNet</td>
<td>55.70/78.42</td>
<td>55.66/78.41</td>
<td>55.54/78.39</td>
<td>54.17/77.29</td>
<td>18.19/36.25</td>
</tr>
<tr>
<td>VGG16</td>
<td>70.44/89.43</td>
<td>70.45/89.43</td>
<td>70.44/89.33</td>
<td>69.99/89.17</td>
<td>53.33/76.32</td>
</tr>
<tr>
<td>VGG19</td>
<td>71.36/89.94</td>
<td>71.35/89.93</td>
<td>71.34/89.88</td>
<td>70.88/89.62</td>
<td>56.00/78.62</td>
</tr>
<tr>
<td>ResNet18</td>
<td>68.63/88.31</td>
<td>68.62/88.33</td>
<td>68.49/88.25</td>
<td>66.80/87.20</td>
<td>19.14/36.49</td>
</tr>
<tr>
<td>ResNet34</td>
<td>72.50/90.86</td>
<td>72.46/90.82</td>
<td>72.45/90.85</td>
<td>71.47/90.00</td>
<td>32.25/55.71</td>
</tr>
<tr>
<td>ResNet50</td>
<td>74.98/92.17</td>
<td>74.94/92.12</td>
<td>74.91/92.09</td>
<td>72.54/90.44</td>
<td>2.43/5.36</td>
</tr>
<tr>
<td>ResNet101</td>
<td>76.69/93.30</td>
<td>76.66/93.25</td>
<td>76.22/92.90</td>
<td>65.69/79.54</td>
<td>1.41/1.18</td>
</tr>
<tr>
<td>ResNet152</td>
<td>77.55/93.59</td>
<td>77.51/93.62</td>
<td>77.40/93.54</td>
<td>74.95/92.46</td>
<td>9.29/16.75</td>
</tr>
<tr>
<td>SqueezeNetV0</td>
<td>56.73/79.39</td>
<td>56.75/79.40</td>
<td>56.70/79.27</td>
<td>53.93/77.04</td>
<td>14.21/29.74</td>
</tr>
<tr>
<td>SqueezeNetV1</td>
<td>56.52/79.13</td>
<td>56.52/79.15</td>
<td>56.24/79.03</td>
<td>54.56/77.33</td>
<td>17.10/32.46</td>
</tr>
<tr>
<td>InceptionV3</td>
<td>76.41/92.76</td>
<td>76.43/92.71</td>
<td>76.44/92.73</td>
<td>73.67/91.34</td>
<td>1.50/4.82</td>
</tr>
</tbody>
</table>
Linear style of programming

- PyTorch is an imperative / eager computational toolkit
Linear style of programming

- PyTorch is an imperative / eager computational toolkit.

```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)
```
Linear style of programming

- PyTorch is an imperative / eager computational toolkit
  - Not unique to PyTorch
Linear style of programming

- PyTorch is an imperative / eager computational toolkit
  - Not unique to PyTorch
    - Chainer, Dynet, MXNet-Imperative, TensorFlow-imperative, TensorFlow-eager, etc.
Linear style of programming

• PyTorch is an imperative / eager computational toolkit
  - Not unique to PyTorch
    • Chainer, Dynet, MXNet-Imperative, TensorFlow-imperative, TensorFlow-eager, etc.
  - Least overhead, designed with this in mind
    • 20 to 30 microseconds overhead per node creation
    • vs several milliseconds / seconds in other options
Go Through an example
The Philosophy of PyTorch
The Philosophy of PyTorch

• Stay out of the way
• Cater to the impatient
• Promote linear code-flow
• Full interop with the Python ecosystem
• Be as fast as anything else
Upcoming features
Distributed PyTorch

- MPI style distributed communication
- Broadcast Tensors to other nodes
- Reduce Tensors among nodes
  - for example: sum gradients among all nodes
Higher order derivatives

• $\text{grad}(\text{grad}(\text{grad}(\text{grad}(\text{grad}(\text{grad}(\text{torch.norm}(x)))))))$
Higher order derivatives

• \( \text{grad(grad(grad(grad(grad(grad(torch.norm(x))))))))} \)
• Useful to implement crazy ideas
Broadcasting and Advanced Indexing

• Numpy-style broadcasting
• Numpy-style indexing that covers advanced cases
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

```
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

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

```
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
x = [0, 1, 2, 3, 4]
for i in range(len(x)):
x[i] = x[i] + 1
for i in range(len(x)):
x[i] = x[i] * 2
# Fused
for i in range(len(x)):
x[i] = (x[i] + 1) * 2
```
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

BatchNorm → Conv2d → ReLU

1 → 2 → 3 → 3 → 1 → 2

Node 0
- GPU 0
- Node 0 GPU 0
- Node 0 GPU 1

Node 1
- CPU
- Node 1 CPU
- Node 1 GPU 0
- Node 1 GPU 1
http://pytorch.org
Released Jan 18th
100,000+ downloads
950+ community repos
8200+ user posts
245 contributors